NoC-Based Multiprocessor Architecture for Mixed-Time-Criticality Applications

Kees Goossens, Martijn Koedam, Andrew Nelson, Shubhendu Sinha, Sven Goossens, Yonghui Li, Gabriela Breaban, Reinier van Kampenhout, Rasool Tavakoli, Juan Valencia, Hadi Ahmadi Balef, Benny Akesson, Sander Stuijk, Marc Geilen, Dip Goswami, and Majid Nabi

Abstract

In this chapter we define what a mixed-time-criticality system is and what its requirements are. After defining the concepts that such systems should follow, we described CompSOC, which is one example of a mixed-time-criticality platform. We describe, in detail, how multiple resources, such as processors, memories, and interconnect, are combined into a larger hardware platform, and especially how they are shared between applications using different arbitration schemes. Following this, the software architecture that transforms the single hardware platform into multiple virtual execution platforms, one per application, is described.

Contents

K. Goossens (✉) • M. Koedam • A. Nelson • S. Sinha • S. Goossens • Y. Li • G. Breaban • R. van Kampenhout • R. Tavakoli • J. Valencia • H.A. Balef • B. Akesson • S. Stuijk • M. Geilen • D. Goswami • M. Nabi
Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: k.g.w.goossens@tue.nl; m.l.p.j.koedam@tue.nl; a.t.nelson@tue.nl; s.sinha@tue.nl; s.l.m.goossens@tue.nl; yonghui.li@tue.nl; g.breaban@tue.nl; j.r.v.kampenhout@tue.nl; r.tavakoli@tue.nl; j.valencia@tue.nl; h.ahmadi.balef@tue.nl; k.b.akesson@tue.nl; s.stuijk@tue.nl; m.c.w.geilen@tue.nl; d.goswami@tue.nl; m.nabi@tue.nl

© Springer Science+Business Media Dordrecht 2016
S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7358-4_17-1

Acronyms

AHB	Advanced High-performance Bus
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BD	Budget Descriptor
CCSP	Credit-Controlled Static Priority
CDC	Clock Domain Crossing
CM	Communication Memory
DLMB	Data Local Memory Bus
DMA	Direct Memory Access
DMAMEM	DMA Memory
DMEM	Data Memory
DRAM	Dynamic Random-Access Memory
ELF	Executable and Linkable Format
ET	Execution Time
ETSCH	Extended TSCH
FBSP	Frame-Based Static Priority
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
GALS	Globally Asynchronous Locally Synchronous
ILMB	Instruction Local Memory Bus
IMEM	Instruction Memory
I/O	Input/Output
IP	Intellectual Property
IPB	Intellectual Property Block
KPN	Kahn Process Network
MAC	Media Access Control
MMIO	Memory-Mapped I/O
MPSoC	Multi-Processor System-on-Chip
NI	Network Interface

NoC	Network-on-Chip
PLB	Processor Local Bus
RR	Round Robin
RT	Response Time
RTOS	Real-Time Operating System
SI	Scheduling Interval
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
TDM	Time-Division Multiplexing
TFT	Thin-Film Transistor
TIFU	Timer, Interrupt, and Frequency Unit
TSCH	Time-Synchronised Channel Hopping
TTA	Transport-Triggered Architecture
UART	Universal Asynchronous Receiver/Transmitter
VEP	Virtual Execution Platform
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

1 Introduction and Requirements

Electronics is pervasive: it enables applications and functions that we have come to expect from appliances as diverse as cars, planes, mobile phones, fridges, and light switches. At the heart of these appliances are Systems-on-Chips (SoCs) that execute the applications. Traditionally, each SoC executed one application, but to reduce cost, multiple applications are increasingly executed on the same SoC. Different applications have different requirements, such as high performance, varying degrees of real time, and safety. In this chapter, we focus on *mixed-time-criticality systems*, i.e., those featuring a combination of applications with and without real-time requirements, respectively. We do not consider other, equally important, criticality aspects, such as safety or resilience. Applications with real-time requirements can be as diverse as motor management, braking, or vehicle stability in a car or wired and wireless communication stacks in mobile phones or computers. This type of applications should always finish computations before given deadlines to ensure correctness and/or safety. In contrast, applications such as a graphical user interface or file management should be responsive to the user but do not have real-time requirements. Audio and video analysis, e.g., for night vision in a car, and media playback are an intermediate category where deadlines should generally be met but may occasionally be missed.

In this chapter, we will define a *mixed-criticality platform*, i.e., a general template, for systems that execute multiple applications with different time criticalities. Given the examples of (non)-real-time applications, we can state the requirements for such platforms.

1. *Guaranteed worst-case performance for real-time applications.* We call this *predictability*, by which we mean that the Worst-Case Response Time (WCRT) of an application can be computed at design time. The Worst-Case Response Time is what has to be guaranteed, but it is usually advantageous to additionally minimize the actual Response Time (RT).
2. *As good as possible, actual-case performance for non-real-time applications.* Unlike real-time applications, the worst-case response time is not relevant and may not even exist. The average or actual response time should therefore be minimized instead.
3. *The absence of interference between applications.* The guaranteed or best-possible performance has to be guaranteed for each real-time and non-real-time application, even though they share the same platform (resources). To be able to do this independently per application, we additionally require that the *actual* execution time of an application is independent of other applications. It then follows that worst-case execution and response times are independent too. We call this *composability*. It helps to isolate (software) faults of applications, increasing robustness. Perhaps more importantly, it allows each application to be developed, tested, and deployed independently, which is required for certification. It also eases upgrading part of a system, without having to retest or recertify the system as a whole.

In this chapter, we describe the CompSOC platform, which is an example of a mixed-criticality platform that meets all the requirements. First, we define the concepts that underpin the CompSOC platform but which are common to many of mixed-criticality platforms (discussed in Sect. 6). We describe the hardware architecture of CompSOC in Sect. 3 and the software architecture in Sect. 4. An example usage of the CompSOC platform is given in Sect. 5. We conclude in Sect. 7.

2 Concepts for a Mixed-Time-Criticality Platform

First we need to introduce some terminology. An *application* is an independent (possibly cyclic) graph of communicating tasks. Tasks use platform *resources*, such as processors, memories, Direct Memory Accesss (DMAs), and interconnect. We model this with the notion of a *requestor*, e.g., a software task requesting computation from a processor, communication from the Network-on-Chip (NoC), or storage from a memory. Each requestor uses only one resource. A requestor generates *requests*, such as computing a function or a memory transaction. As illustrated in Fig. 1a, resources serve requestors in discrete uninterrupted units called *service units*, and a request consists of one or more service units (possibly infinitely many). The execution of a service unit by a resource takes an actual Execution Time (ET), which is less than or equal to its Worst-Case Execution Time (WCET) if it exists (i.e., it is finite). The WCET of a requestor is equal to the largest WCET of its requests. A resource is *predictable* if all service units have a WCET. A resource is *composable* if the response and the execution time of every request of an application

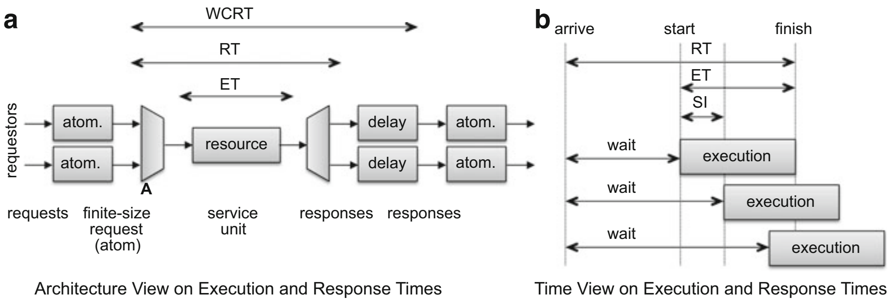


Fig. 1 Terminology. (**a**) Request, response, service unit, execution and response time, arbitration (*A* in the figure), and resource. The atom and atomizer will be discussed in Sect. 3.2. (**b**) Arrival, start, finish, and scheduling interval

do not change when also executing any number of requests of different applications on the resource.

A resource is *shared* if it executes requests of multiple requestors, which may belong to different applications. The execution time of a service unit does not take into account waiting time for other service units from the same or different requestors. Instead, this is captured by its actual response time and its Worst-Case Response Time (WCRT). (As before, we say the WCRT exists if it is finite. The WCRT of a requestor is equal to the largest WCRT of its requests.) The waiting time depends on the arbitration employed to determine the order of execution of the service units of the different requestors, shown by A in Fig. 1a. Possible arbitration policies include Round Robin (RR) , Time-Division Multiplexing (TDM) , Credit-Controlled Static Priority (CCSP) [5], Frame-Based Static Priority (FBSP) [2], etc., each with their own characteristics and (dis)advantages which we will discuss later. An arbiter is *predictable* if all requests have a WCRT (i.e., it is finite), assuming that their ETs are finite. (Thus, even with a predictable arbiter, a request may have no (i.e., infinite) WCRT if the resource is unpredictable.) Similarly, an arbiter is *composable* if the response and the response time of every request of an application do not change when also executing any number of requests of different applications on the resource.

The resource requirements of a requestor are specified using a *budget*. A requestor can use a resource only after its budget has been reserved. A resource may be *idle*, i.e., no service unit is executed. This may occur if not all of its capacity has been reserved or if a requestor does not use all of its reservation. Arbitration is *work conserving* if the resource is not idle whenever a service unit is waiting to be executed.

Figure 1b illustrates the Scheduling Interval (SI) of a resource as the time between accepting successive service units, and its reciprocal (service units per second) defines the throughput. Resources are often pipelined, and then the reciprocal of the scheduling interval defines a higher throughput than the reciprocal of the execution time. From the response times of individual requests (executing on processors, interconnect, or accessing memories), it must be possible to compute the performance (worst-case response time, throughput, etc.) of a real-time application as a whole. As an example, [37] illustrates how to do this using the dataflow model of computation.

Given the terminology, these are the seven concepts on which we base our mixed-time-criticality platform:

1. **Budgets.** Reserving part of a resource for a requestor belonging to an application according to its budget results in a *virtual resource*. Only then can a requestor use the resource.
2. **Predictability.** *Arbitration between requestors of a real-time application must be predictable*. Predictable arbitration ensures that a virtual resource offered by a single resource has a minimum guaranteed performance as specified in the budget. Arbitration is preferably work conserving such that requestors of an application can use each other's unused capacity.
3. **Composability.** *Arbitration between requestors of different applications must be composable*: the behavior of one application must not be affected by the behavior of other applications. Composable arbitration ensures that (the performance of) a virtual resource is independent of other virtual resources on the same resource. This implies that the arbitration cannot be work conserving because then (variable) execution times of one application could affect the response time of other applications.
4. **Scalability.** *Decouple resources* as much as possible. Logically, this means that *each resource arbitrates locally*, with a suitable service unit and arbiter to enforce the reserved budgets. This disallows synchronization hardware, such as mutexes, locks, and semaphores, as we will discuss in Sect. 3.8. Physically, decoupling requires that all hardware Intellectual Property Blocks (IPBs) use independent clocks because in modern Multi-Processor Systems-on-Chips (MPSoCs), it is no longer possible to use a single clock. This is called the *Globally Asynchronous Locally Synchronous (GALS) design style*. Logical and physical decoupling improve scalability by avoiding centralized and tightly synchronized architectures.
5. **Finite scheduling interval.** As a consequence of Bullet 3, resources that are shared by multiple applications must ensure that no (service unit of a) requestor can indefinitely block others from using the resource. This is achieved with a finite scheduling interval, implemented either by chopping (possibly infinitely large) requests into service units with a finite WCET or else by preempting the resource within a finite time.
6. **Efficient arbitration.** If possible, *avoid arbitration* on a resource; e.g., a DMA is cheap enough to just replicate. Otherwise, at least *one level of composable arbitration* is required to arbitrate between requests belonging to different applications as well as between requests of the same application. (If a resource is used by only one application, then only one level of arbitration is needed, which must be predictable at most.) Preferably, *two levels of arbitration* are used: the first level to separate different applications and the second level to separate requests of the same application. The former must be composable, and the latter

predictable. For each resource, we will discuss in depth what kind of arbitration it can efficiently support.

7. **Efficient resource sharing.** As an optimization, *the scheduling interval should be as small as possible.* This allows interleaving the service of requestors as finely as possible and reduces response times [57]. How small the scheduling interval (and service units) can be depends strongly on the resource.

The above ingredients can be combined in the concept of a *Virtual Execution Platform (VEP) per application*. This means that an application's budget is reserved on all of the platform resources it requires, creating a smaller virtual platform on which it executes. A VEP is composable, i.e., independent of other VEPs and applications running therein. Within a VEP, an application may use its own programming model, arbitration, and so on, as long as it complies with the requirements outlined above. The CompSOC platform is an operational prototype implementing the concepts just described. It is our running example, as we go through the details in the remainder of this chapter. In the next section, we describe the hardware components of the CompSOC platform, followed by the software stack in Sect. 4. We discuss related work in Sect. 6 before concluding in Sect. 7.

3 Hardware Architecture

MPSoCs contain multiple processors with local and shared memories. The processor's local memories are always on-chip Static Random-Access Memory (SRAM), close to the processor. Nonlocal memories shared between processors may be on-chip SRAM but often include off-chip Dynamic Random-Access Memory (DRAM). The latter has a much larger capacity (number of bits) than the on-chip memory, but at the cost of a longer execution time. Processors reach shared memories using a communication infrastructure, which is increasingly a NoC. A NoC is a miniature version of the Internet in the sense that communication is concurrent, is distributed, and is either packet based or circuit switched. In this section, we introduce the CompSOC hardware platform. It not only fits the generic MPSoC description, it also addresses all the requirements listed in Sect. 1. As a result, it can run multiple applications of different criticalities at the same time.

The CompSOC platform consists of multiple tiles interconnect by a NoC. Tile types are master tiles, slave tiles, or a mix and include processor tiles, memory tiles, peripheral tiles, etc. In the following sections, we discuss each component in turn. Regarding terminology, an IPB, such as a memory, processor, or DMA, may have zero or more master and slave ports. Master ports initiate requests, i.e., read and write transactions, using a standard communication protocol, such as Processor Local Bus (PLB), Advanced High-performance Bus (AHB), or Advanced eXtensible Interface (AXI). Slave ports accept requests, and the slave executes them, possibly returning a response. In the following, we will shorten "master (slave) port on an IPB" to just "master (slave) IPB."

In the following sections, we will introduce master and slave IPBs , followed by processor tile that is a hybrid master-slave IPB. Described next is how all IPBs use distributed shared memory to communicate and how this is implemented by the NoC and the memory map. Finally, we will discuss atomic and synchronized communication.

3.1 Generic Master IP Block

A master IPB initiates read or write transactions that are to be transported by the NoC to a slave IPB for execution. The generic architecture of a master IPB is shown in the middle of Fig. 2. As will be explained in more detail in Sect. 3.6, ports on the NoC are connected pair-wise: only one slave IPB can be reached from a master single port. For this reason, a master IPB requires a NoC port for each slave it communicates with. The multiplexer labeled M3 determines to which slave a transaction is bound, depending on the transaction address. Any responses are interleaved in the order of the requests by the (de)multiplexer M3, before being returned to the master IPB [21]. The multiplexer and the master IPB may be programmed using the rightmost slave port on the tile.

3.2 Generic Slave IP Block and Memory Tile

Memory tiles only contain one slave resource that accepts requests from multiple requestors. Although we describe the architecture of the memory tile, it is essentially the same as that of any generic shared slave IPB. Each requestor has a dedicated NoC port on the tile, as illustrated by the general slave tile on the left in Fig. 2. Requests take the form of transactions, using, e.g., the PLB or AXI protocol. It is possible to read or write one or more words of 4 bytes, with a byte mask applied to each word in the transaction. Note, however, that in theory transactions can be infinitely long and that they can take a long (or even infinite) time to arrive. It may thus not be possible to buffer an entire transaction (request). For this reason, an atomizer chops incoming requests into complete fixed-size aligned requests (also called *atoms*). Note that even though these requests have a finite size, their execution time may be infinite; consider, for example, a `while(1);` request.

3.2.1 Arbitration

When a service unit is complete, it is ready to be scheduled by the arbiter, according to some policy. If the slave is only used by a single non-real-time application, then any arbitration policy may be used; if it is a real-time application, then the policy must be predictable for a WCRT, such as RR, TDM, or CCSP. However, it is likely that slaves such as SRAM are used by multiple applications. In this case, the actual execution times of service units of different applications must be independent. TDM is a simple arbiter that achieves this [18], but it is also possible to use any predictable arbiter (RR, CCSP, etc.) in combination with delay blocks [3]. A *delay block*, shown

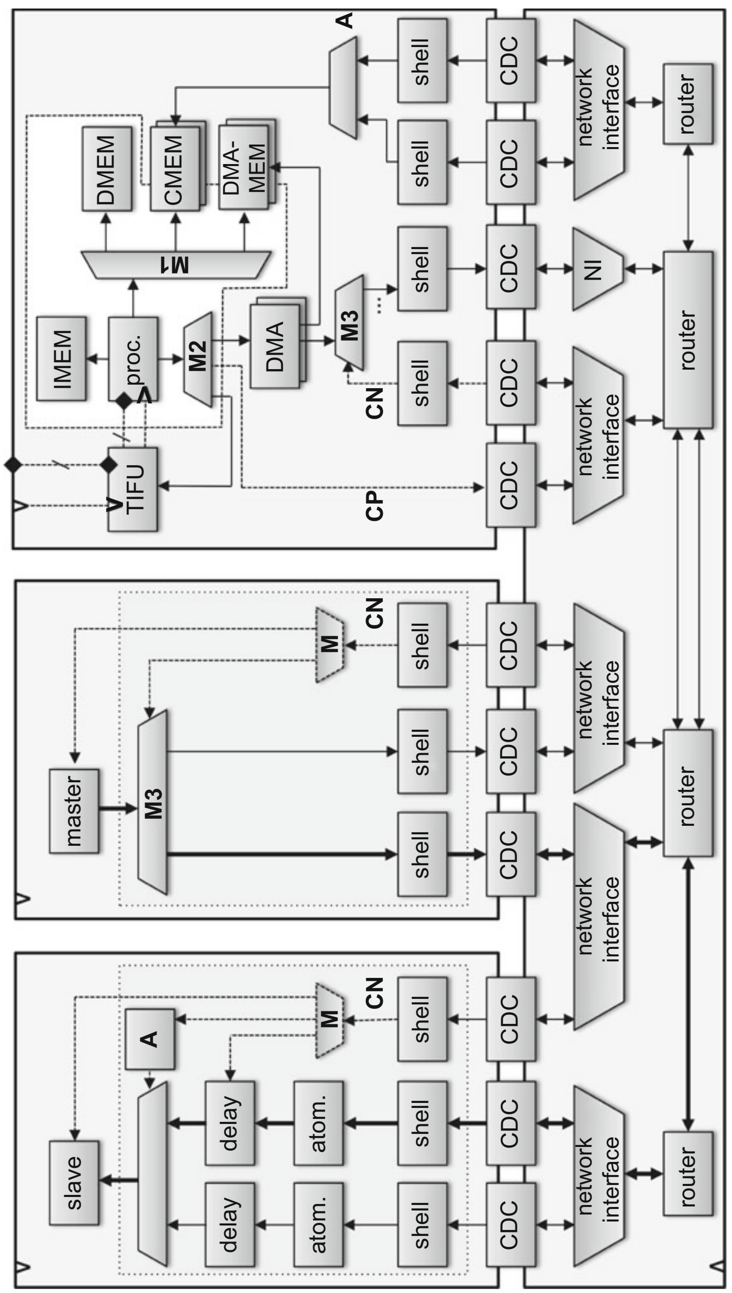


Fig. 2 Example CompSOC hardware platform instance. *Arrows* point from a master (initiator) port to a slave (target) port. *Dashed arrows* indicate control connections to program arbiters, memory maps, and IPBs. A is an arbiter, C is a control connection, and M are memory maps

in the slave tile of Fig. 2, only releases a response from the slave at the WCRT, cf. Fig. 1. Since the WCRT takes into account the worst-case interference from other applications, the delay block enforces a response time that is independent from other applications.

When arbitration has to be composable, predictable arbiters (RR, CCSP, etc.) with a delay block may offer more flexibility than a composable arbiter (TDM). In particular, since TDM inversely couples throughput and response time, a small WCRT is only possible with large budget, which the requestor may not need. Especially for memories that are loaded heavily, such as DRAM (see below), over-reservation may not be acceptable. On the other hand, TDM has the advantage that it is easier to reprogram the budget of a single application without affecting other running applications [18]. Resources that have small service units, such as memories, require arbitration to be implemented in hardware. It is then not practical to use a two-level arbiter that is composable between applications and predictable within an application, because it requires fixing the number of requestors per application in the hardware, which is expensive and inflexible.

The arbiter in the tile, the delay block, and the IPB may all be programmable, which is done by write transactions on the tile's slave port, which is then demultiplexed with a fixed memory map (M in Fig. 2) to the appropriate block.

3.2.2 SRAM

SRAM is the prototypical slave IPB. The service unit is usually as small as one word, which necessitates a fast (and thus simple) hardware arbiter. SRAMs are often shared within a single application, e.g., Communication Memory (CM), and round robin arbitration is then used. To share between applications, atomizers are required, and usually service units of a single word and a RR arbiter are used.

Chapter ▶ "Memory Architectures" gives more information about general SoC memory hierarchies.

3.2.3 DRAM

A DRAM tile has the same structure as an SRAM tile, but the DRAM itself has quite different characteristics. In particular, its service units are more complex [1, 4, 9, 17, 30]. Reading and writing in a DRAM require sending a number of commands (activate, read/write, precharge, refresh). For a reasonable efficiency, it is required to use bursts of data, typically with a length of eight words for most contemporary DDR memories, and then pipeline the DRAM commands to a single DRAM bank and/or across multiple DRAM banks. A traditional (non-real-time) DRAM controller schedules commands dynamically as service units arrive, using an open-page policy [4]. It is hard to analyze the execution time of each service unit because the time between successive DRAM commands varies a lot, depending whether the data that is accessed is in a bank that is open (activated) or not.

For this reason, CompSOC's real-time Raptor memory controller uses a close-page policy that ensures that the ET of a service unit is predictable [13] or even constant [18]. The memory commands for the service units have to be programmed into the memory controller, using the rightmost dotted line in the slave tile of Fig. 2.

With a predicable ET of a service unit, the execution time of a service unit can still depend on the preceding service unit, e.g., depending on whether it was a read or a write. This can be prevented by scheduling memory commands differently, resulting in composable service units that have a constant execution time. It has been shown in [18] that composable service units can be used with several generations of DRAM memories with a negligible impact on performance.

The DRAM is usually a heavily loaded resource and used by multiple applications. It is possible to use predictable service units with a predictable arbiter (e.g., RR, CCSP) and delay blocks (recall that the constant WCRT eliminates all interference) [3]. Alternatively, composable service units may be used, with a composable arbiter (TDM) [18], with the advantages and disadvantages discussed in the previous section.

3.3 Processor Tile

Having introduced basic master and slave tiles, we now discuss the processor tile, which is a mix of both. As shown on the right of Fig. 2, it contains a processor (in our prototype, a Microblaze or ARM Cortex M0) with an Instruction Memory (IMEM) and a Data Memory (DMEM). The memories are usually tightly coupled (i.e., have a single-cycle access time) using Instruction Local Memory Bus (ILMB) and Data Local Memory Bus (DLMB) busses.

There are no caches in the tile, because it must be possible to compute the WCET of a task on the processor if predictability is required. We do not use caches because the WCET would then not only depend on the processor but also on the time for cache misses. It is possible to take into account the WCET of the interconnect and remote (off-tile) memories to compute remote memory accesses [23]. However, we decided to keep the architecture and performance analysis simple and not use caches. A second, more important reason to omit caches is to adhere to Concept 1 of budgets: the WCRT of a task on a processor then only depends on the budget of the processor on which it runs. This allows the WCRT of all tasks to be computed independently of other tasks and of how their communication and storage are mapped on the platform. Performance analysis is thus compositional, i.e., consisting of independent smaller analyses. This simplifies the design and verification flow [14].

A processor cannot access remote memories directly. Otherwise, similarly to caches, the WCET of a task would depend on read and write transactions to a remote memory, i.e., the arbitration in the NoC and remote memory. Additionally, since sharing a processor between applications must be composable, its service units are enforced by preemption through interrupts, as explained in Sect. 4.1. However, since simpler processors, such as Microblaze and ARM Cortex M0/3/4, do not allow read and write transactions to be interrupted, the interrupt service latency could be very long. By using a DMA to access remote memories, the data transfer between local and remote memories is executed independently and concurrently with the processor. The task that programmed the DMA can be interrupted in a few cycles and swapped out, for an efficient implementation of composable sharing of the processor.

The execution time of a task on the processor may depend on tasks that executed before it, due to the processor's internal state. For example, when the processor implements branch prediction, the predictor state is not automatically reset between task switches. Cache pollution is another example, although we already eliminated it by excluding the use of caches. Since composability requires that the execution time of an application is independent of other applications (running earlier or concurrently), we ensure that the processor is reset to a *neutral state* [14] between application switches by resetting the branch predictor state. For caches, it would be enough to flush them between task switches. In general, an instruction to reset the processor's entire internal state would make it easier to make it predictable and composable.

As already mentioned, and shown in Fig. 2, a processor uses DMAs to read or write data in a remote memory. Each DMA connects to a port of a DMA Memory (DMAMEM). The DMAMEM connects with another port to the processor DLMB (other options are discussed in [34]). Using two ports on the memory avoids arbitration between the processor and the DMA, which would necessitate changing the single-cycle turn around DLMB bus to a slower PLB bus. (When using an ARM processor, these would be AHB or AXI busses.) The DMA has a Memory-Mapped I/O (MMIO) port on the PLB, through which it is programmed to either copy data from the DMAMEM to a remote memory accessed over the NoC or vice versa. The DMA has a NoC connection to each remote memory. Since it is cheap, a DMA is not shared between applications to avoid interference. Although it may be used (sequentially) by different tasks of the same application, avoiding this simplifies the computation of the WCET of a task by eliminating interference from other tasks.

A remote memory can be the shared DRAM, a shared SRAM on the NoC, or an SRAM in another (remote) tile. The latter is a CM: the DMA can copy from the local DMAMEM into a CM of another processor tile, or vice versa. The CM has two ports, for the same reason the DMAMEM does. If multiple remote DMAs can access the same CM, a multiplexer with an arbiter is required (labeled A in Fig. 2, cf. Sect. 3.2).

Execution on the processor can be preempted with interrupts. This is required as soon as multiple applications run on the processor because applications should run in an interleaved fashion, not consecutively. The service unit of a processor is a TDM slot (or time slice), generally lasting 20,000 cycles or more. To achieve composability, tasks must be preempted at the end of each TDM slot, after exactly the same number of clock cycles. For this, a Timer, Interrupt, and Frequency Unit (TIFU) is attached to the processor. It can generate interrupts at precise moments in time and halt (clock gate) the processor until precise moments in time. The interrupt and clock lines from the TIFU to the processor are shown in Fig. 2 to and from the TIFU. The TIFU can also change the frequency at which the processor can run for power management [38], shown by the clock domain in white. Section 4.1 describes how the TIFU is used to preempt and arbitrate the applications and tasks within an application on a processor.

The processor tile is the most complex because the composable service units require preemption and decoupling the processor from other IPBs (in particular, remote memories). This requires the TIFU, multiple local memories, some with their own arbitration, and DMAs.

3.4 Network-On-Chip

The NoC allows all IPBs to communicate. Chapter ▸ "Network-on-Chip Design" contains a thorough introduction to NoC technology. From a real-time perspective, the NoC is problematic because it is a resource that is made of smaller components, namely, routers and Network Interface (NI), that are distributed spatially. A single centralized arbiter is thus not feasible, and each router and NI has an independent local arbiter. As a result, a service unit that enters the network may be delayed at each router that takes local decisions based on its local state only. This makes it very hard to compute the end-to-end response time and throughput due to contention and congestion. The CompSOC platform uses the synchronous daelite NoC [49], which is derived from the mesochronous Aethereal NoC [22]. The asynchronous Argo NoC [27] is based on the latter. The common underlying concept is to use a *single global TDM arbiter for the entire NoC but to implement it in a distributed way*. The routers are synchronized such that service units (flits) never arrive at the same link at the same time, thus eliminating all contention. Without contention, routers require no arbiters and no buffers (except for pipelining), making them very fast and cheap [15]. Service units only wait at the edge of the NoC, in the NIs [45]. To offer real-time timing guarantees, the global TDM schedule is programmed at run time, to offer connections, i.e., resource budgets, from a master IPB initiating transactions to a slave IPB executing transactions.

As illustrated in Fig. 2, the slave, master, and processor tiles, and the NoC operate in independent clock domains, which are straddled by the Clock Domain Crossing (CDC) blocks. The CDCs are functionally transparent, and we will not discuss them further. The NoC thus implements the GALS paradigm, which is essential for scalability (Concept 4).

The four components [21] of the NoC are illustrated in Fig. 2, and we described them following the bold arrows from a master IPB to a slave IPB and back. First, transactions from a master IPB may go to multiple slaves according to its memory map, discussed in more detail in the next section. To implement the memory map, a programmable demultiplexer sends each request to the connection to the right memory and interleaves responses in the right order. Second, the protocol shell serializes each transaction to a stream of data words, without any particular structure. This allows multiple IPB protocols, such as AHB and AXI, to be sent over the same NoC. Third, NIs preempt the data stream into service units (flits) that are injected in the NoC according to the TDM schedule. Finally, routers just move flits around, without contention and hence without arbitration. When a flit arrives at its destination NI, the constituent words are given to the protocol shell that reconstitutes the transaction in the right protocol.

Intuitively, the real-time performance of the NoC, in particular the WCRT and throughput of a connection are easily computed by looking at the number of slots reserved for the connection and the total number of slots in the TDM table. However, an additional complication is that NI buffers are finite and slave IPBs do not necessarily accept incoming transactions immediately. To avoid dropping data, NIs use end-to-end flow control, which must be modeled in the WCRT calculation [24].

The service unit of the NoC is a flit, and the protocol shells together with the NIs preempt larger transactions into flits. The NoC uses single-level TDM arbitration between flits of the same as well as different applications to eliminate costly per-application or per-connection buffers in the router. Although two-level arbitration is possible, it makes routers larger and slower [15].

3.5 Peripherals

Many peripheral tiles are simple non-shared slave tiles. For example, a Universal Asynchronous Receiver/Transmitter (UART), Serial Peripheral Interface (SPI), or audio peripheral is usually not shared between multiple requestors. As a result, they can be hooked up directly to the NoC, with only a slave port, and delay blocks and arbitration are not required. A processor usually accesses the peripheral directly (using its DMA), usually polling for data or writing data into the peripheral's local buffer.

As an example of a complex peripheral, the inter-NoC bridge [33] links the NoC on the Field-Programmable Gate Array (FPGA)/Application-Specific Integrated Circuit (ASIC) to a NoC on another FPGA/ASIC or even to another computer. It acts as both slave and master. As a slave, it receives data from the local NoC over multiple connections to be sent to the remote NoC. As a master, it produces data on multiple connections that came from the remote NoC. Given TDM arbitration in the NoCs, the bridge uses TDM arbitration too. However, its service unit depends on the medium over which the NoCs are connected. We used an Ethernet Media Access Control (MAC) with Xilinx RocketIO with large Ethernet packets for high data efficiency, but at the cost of a large scheduling interval and WCRT. The inter-NoC bridge is programmable and thus also has MMIO slave ports.

3.6 Memory Map

All the components that we have described until now communicate with each other using transactions. A transaction can read or write in a memory at a given address or address range. Additionally, IPBs and their tiles can often be programmed by writing into memory-mapped register. Similarly, data input or output is often performed through memory-mapped buffers.

The CompSOC platform implements distributed shared memory. It is distributed in the sense that there are multiple memories in the same 32-bit memory map, and shared because multiple IPBs can access a given memory. The memory map defines in which memory each of the 32-bit addresses is located. It is implemented in several places. First, hardwired demultiplexers. The demultiplexer (labeled M1 in Fig. 2) is the local DLMB bus in the processor tile. It is hardwired with the address ranges of the IMEM, DMEM, CM, and DMAMEM memories in the local tile. The demultiplexer labeled M2 is similarly hardwired and decodes the local PLB peripheral bus. Finally, the demultiplexers labeled M decode the MMIO transactions to program a slave and its arbiter and delay blocks.

Second, the programmable demultiplexers M3 and NoC connections define the remainder of the memory map. Given an address, the former first selects a NoC connection, and the latter then delivers the transaction to the memory (more generally, IPB), as illustrated by the thicker lines from the master IPB to the slave IPB. Note that the routers and NIs implement part of the memory map, but they have no knowledge of transactions or memory addresses: they just implement point-to-point connections to transport data (requests, as it happens) from master IPBs to slave IPBs and vice versa (responses). This simplifies the NoC, making it faster. It also enables multiple memory maps to coexist in the same platform. For example, multiple processors may boot from (their) address 0, which is mapped in different memories for different processors. This is useful when processors run different Real-Time Operating Systems (RTOSs) or are heterogeneous.

There are no NoC connections after a reset, and IPBs cannot communicate with each other at all. The memory map (multiplexers M3 and NoC connections) is programmed using the NoC itself at run time, using a bootstrap procedure [20, 49]. A privileged processor sets up NoC connections one by one and programs the memory map(s), offering a secure boot procedure. It directly programs the NoC using the dotted control line CP in Fig. 2. Some memory-map multiplexers may be hardwired, e.g., M. All others (M3), including those of the DMAs of all processors, are programmed over the NoC using control lines CN [21]. Programming a memory map is predictable and composable.

3.7 Atomicity

The NoC allows master IPBs to communicate using distributed shared memory. When (a task on) an IPB sends data to a shared memory, it is important that it has been written completely before it is read by another IPB. However, high-performance communication protocols, such as AXI, limit the atomicity of transactions to a single byte. It is hence not possible to send more than a single byte atomically without further precautions.

The CompSOC platform addresses atomicity at several levels. First, the hardware reads and writes 32-bit words atomically to all data memories and memory-mapped IPBs. Second, recall that each resource has a minimum service unit that is executed atomically, i.e., without interruption or preemption by other service units (they may be pipelined though). Transactions that read or write more than a service unit, or are not aligned, are chopped into multiple service units that may be arbitrarily interleaved with service units of other requestors. The minimum service unit of

SRAM is a single 32-bit word, and that of a DDR3 DRAM typically eight words. Thus, an IPB can atomically access data in a memory, as long as it is no larger than one service unit.

Regarding atomicity of computation, the service unit of a processor that runs multiple tasks is a time slice (Sect. 4.1). The task that runs is interrupted and preempted (swapped out) for another task at the end of each slice. Since interrupts can occur after each instruction, only a single instruction is atomically executed from a software perspective. Traditionally, atomicity is extended to multiple instructions (called a critical region) by disabling the interrupt before the critical region and reenabling it at the end. However, critical regions increase the . In the worst case, a non-real-time application enters but never exits a critical region, so claiming a shared resource forever. To avoid this, we do not allow critical regions in application code. We will return to this topic in Sect. 4.1.

Although atomic transactions are essential, they are not enough to safely communicate data. Data may be (atomically) overwritten before it has been read, for example. This will be addressed in the synchronization Sect. 4.4.

3.8 No Synchronization Hardware

According to the scalability Concept 4, most MPSoCs are based on the GALS paradigm. Each IPB or tile operates synchronously on its own clock but is asynchronous with other IPB or tiles, as discussed in the NoC Sect. 3.4. The TIFU in each processor tile provides local timers, but they are not synchronized with each other and may drift arbitrarily. Since there is no global clock or notion of time, pure time-triggered synchronization (whereby the parties wait until a specific time) is not possible. All synchronization strategies used in CompSOC, later described in Sect. 4.4, are therefore based on data synchronization.

We implement data synchronization based only on atomicity of read and write transactions. We do not use processor instructions, such as a test-and-set, or load-linked and store-conditional, or dedicated hardware for locks or mutexes. Test-and-set (and similar) instructions work by locking the path from the processor to the memory from the read (test) to the write (set) instruction. This is not scalable and infeasible when the processor and memory are connected by a distributed and shared interconnect, such as a NoC. Load-linked and store-conditional instructions work without locking the interconnect but require a processor with support for them, as well as a memory that keeps track of changes to its memory locations, which is not standard.

Locks and mutexes also require dedicated hardware, which we do not (wish to) have. They have an additional problem, namely, that when a resource is shared between multiple applications, it should never be locked by a single application, because that violates composability. At best a lock per application can be used, making sure that the resource is aware which requestor belongs to which application. However, in our platform, the NoC, SRAM, and DRAM have simple and fast single-level arbiters that do not distinguish between different applications.

The NoC and memory maps allow only atomic communication of memory service units. In Sect. 4.4, we describe how CompSOC offers safe communication and synchronization in software.

3.9 Conclusions

CompSOC is an example of a platform that supports running multiple applications with different time criticalities. In this section, we described the basic components comprising the CompSOC platform and justified the design choices for those components in accordance with the high-level concepts of the previous section.

In particular, each shared resource (i.e., processor, NoC, SRAM, DRAM, some of the peripherals) offers a service unit that is atomically executed with a known WCET. Requests are chopped into, or rounded up to, one or more service units.

Next, the arbitration policy is essential. To share a resource between (requestors of) non-real-time applications only, any arbitration policy may be used. However, to share a resource with requestors of multiple real-time applications, the resource arbitration must be predictable. This makes it possible to compute the WCRT from the individual WCETs and the predictable arbitration policy. If a resource is shared between real-time and non-real-time applications, which may not have a finite WCET, then arbitration must at least be composable between applications, and be predictable for requestors of the same real-time application. This can be implemented with a single-level composable arbiter (e.g., NoC, DRAM) or with a two-level arbiter (composable between applications, predictable or not within each application) as done on the processor (explained in Sect. 4.1). Composable arbitration can be implemented with TDM (e.g., NoC, processor, DRAM) or with any predictable arbiter plus delay blocks (e.g., SRAM, DRAM). Many predictable arbiters are available, such as RR, TDM, CCSP, or FBSP. Which one is used depends on the characteristics of the resource, such as cost of preemption and buffering, and the real-time requirements of the requestors.

4 Software Architecture

The CompSOC hardware platform contains computation, communication, and storage resources. Almost all can be shared between multiple requestors, and almost all can be (re)programmed at run time. The CompSOC software extends the single hardware platform to offer multiple Virtual Execution Platforms (VEPs). A VEP is an execution platform that is a subset of the CompSOC hardware platform, in terms of time (e.g., time multiplexing a processor) or space (e.g., non-shared DMA or a region in a memory). Each application runs in its own VEP, which is created, loaded, started, and possibly stopped and deleted, at run time. A CompSOC platform can run multiple VEPs concurrently, without any interference between them, i.e., composably.

To implement VEPs, the CompSOC software is constructed in a number of layers, as shown in Fig. 3. From the lower to higher layers, we have:

1. A *microkernel or RTOS* to share a resource with a software arbiter. Software arbitration is only used for the processor, since the service unit of all other resources is too small to be managed in software.
2. A *driver* library is used to program, load, start, use, and stop a virtual resource.
3. The resource requirements of requestor are specified using a *budget*. Reserving part of a resource for a requestor according to its budget results in a *virtual resource*. Most resources may be programmed to be shared in time and/or space. A VEP is a hierarchical virtual resource, made up of, for example, a region of DRAM, some NoC connections, and several virtual processor tiles. A virtual processor tile is again a hierarchical virtual resource, consisting of several memory regions, DMAs, and a virtual processor. The *budget management* (BM) library is used to reserve and release virtual resources, and thus keeps track of who has been reserved what part of the resource (TDM slots, memory regions, etc.).
4. The resource driver and budget management libraries comprise the *resource management* library.
5. When an application runs within its VEP, its tasks will need to synchronize and communicate, according to some *model of computation*. We offer several programming models or models of computation (threads, dataflow, Kahn Process Network (KPN), time-triggered) that are implemented using more primitive communication (barriers, FIFOs, sampling).
6. A *bundle* contains the code and data of an application together with the specification of the VEP it requires to run.
7. The *system application* manages a CompSOC platform. It uses the resource management library to create and remove VEPs and uses the *bootloader* library to start and stop applications.
8. Finally, an application programmer creates *applications* using one of the supported programming models or model of computation. He or she develops and validates the application in a VEP, which is either given up front or codefined with the application. The application is deployed as a bundle, i.e., the application's code (ELFs) together with the definition of the VEP. Composability guarantees that executing an application (bundle) before and after integration with other applications (bundles) gives exactly the same results.

Since the system software extends the hardware, it has the same requirements, and it must adhere to the same requirements and concepts. We now discuss each of the software components.

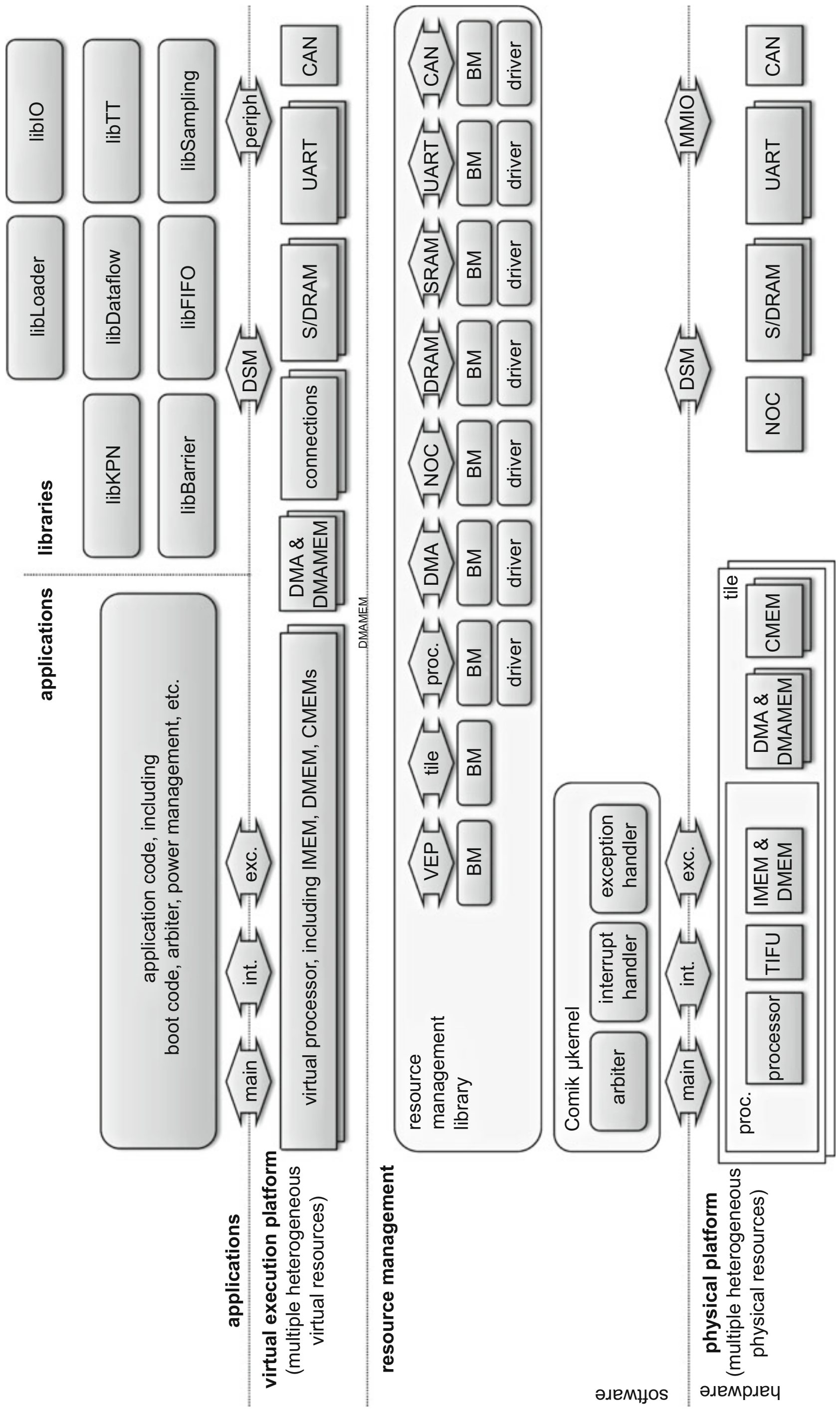


Fig. 3 Software architecture. DSM is distributed shared memory

4.1 Microkernel and RTOS

Processors are the only resources that are shared with software arbitration; all other resources use a (programmable) hardware arbiter. This is possible because the service unit of a processor, called a time slice, is in the order of tens of thousands of clock cycles and the cost of saving the state of the running application and arbitration is a few thousand clock cycles.

Recall that the concepts for a mixed-time-criticality platform include composable arbitration between applications (inter-application), predictable arbitration within an application (intra-application), or no arbitration for non-shared resources. Inter- and intra-application arbitration may be either separate (two-level) or combined (single-level). In the latter case, it must be both predictable and composable.

Task arbitration can be classified along several axes. First, it may be absent when there is only one task on a resource. Otherwise it is required. Second, it may be preemptive or not. Third, arbitration may be static and follow a static-order schedule or be dynamic where the order of tasks is determined at run time. Figure 4 and Table 1 illustrate how a processor can be (a.A) not shared or shared between tasks of one application only, either (a.B) with a static-order schedule, or (a.C) cooperatively scheduled (i.e., non-preemptive dynamic). Alternatively, multiple applications can share the processor using a microkernel such as CoMik, which arbitrates only between applications (b and c). Each application can use the techniques from (a) or even a virtualized RTOS, such as μC-OS III (c), to independently arbitrate between application tasks. The two levels of arbitration can also be combined in a single software arbitration layer (d), as done by the CompOSe RTOS [6, 19]. Finally (e), a traditional RTOS can be used with a single-level arbitration, e.g., TDM. However, if, as is often the case, priority-based arbitration without delay blocks is used, only a single application can use the processor.

As Fig. 4 shows, a physical processor can run software ("main") but also optionally has an interrupt handler ("int.") and exception handler ("exc."). When running the application directly on the processor (a), at least the main software must be defined by the programmer. A microkernel virtualizes the main, interrupt and exception handling and passes them on to the application (b) or RTOS (c). An RTOS that runs directly on the processor typically hides the interrupt and exception handling from the applications. In almost all cases, the programming model or model of computation used in an application hides interrupt and exception handling (see Sect. 4.4).

Because the execution of an application should not depend on another application, inter-application arbitration must be preemptive with service units (time slices) that are of constant length (down to the level of a clock cycle). As mentioned in Sect. 3.3, the TIFU allows the microkernel or RTOS to interrupt the running application at a precise point in the future. After saving the application context, the next application is scheduled, and its context restored. The time from the interrupt to the restored context may vary, e.g., due to critical regions, multi-cycle instructions, or variable execution time. Critical for composability CoMik and

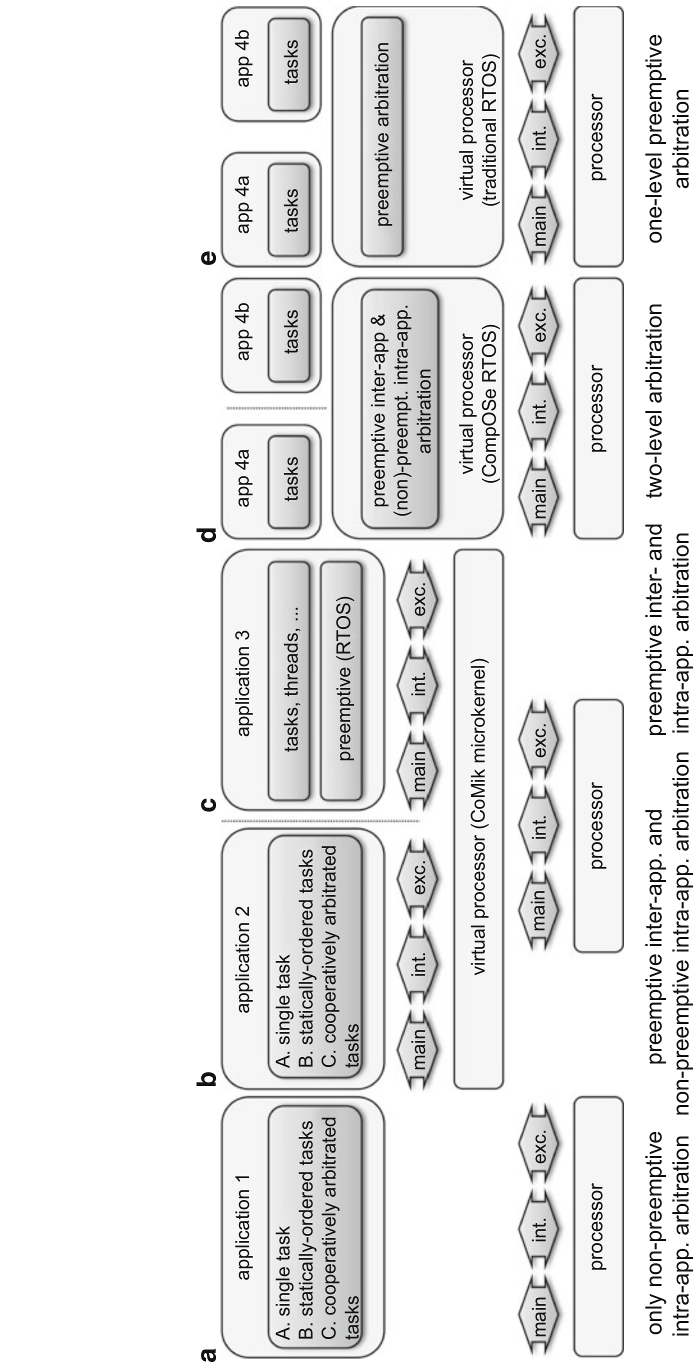

a
application 1
A. single task
B. statically-ordered tasks
C. cooperatively arbitrated tasks
main
int.
exc.
processor
only non-preemptive intra-app. arbitration
b
application 2
A. single task
B. statically-ordered tasks
C. cooperatively arbitrated tasks
c
application 3
tasks, threads, ...
preemptive (RTOS)
main
int.
exc.
virtual processor (CoMik microkernel)
processor
preemptive inter-app. and non-preemptive intra-app. arbitration
preemptive inter- and intra-app. arbitration
d
app 4a
tasks
app 4b
tasks
preemptive inter-app & (non)-preempt. intra-app. arbitration
virtual processor (CompOSe RTOS)
processor
two-level arbitration
e
app 4a
tasks
app 4b
tasks
preemptive arbitration
virtual processor (traditional RTOS)
processor
one-level preemptive arbitration

Fig. 4 Processor virtualization

Table 1 Inter- and intra-application arbitration on processors and other resources

Figure 4	Inter-app.	Intra-application		Scenario
	Preemptive	Preemptive	Non-preempt	
(a.A)	–	–	–	Single task
(a.B)	–	–	Static order	Dataflow actors of 1 app.
(a.C)	–	–	Cooperative	Tasks & FIFO, KPN processes of 1 app.
(b.A)	TDM	–	–	CoMik microkernel, multiple apps.
(b.B)	TDM	–	Static order	CoMik microkernel, multiple apps.
(b.C)	TDM	–	Cooperative	CoMik microkernel, multiple apps.
(c)	TDM	RTOS	–	CoMik microkernel, multi-app. incl. RTOS
(d)	TDM	–	Any/pred.	CompOSe RTOS with 2-level arb.
(d)	TDM	Any/pred.	–	CompOSe RTOS with 2-level arb.
(e)	–	Priority	–	Traditional RTOS with 1-level arb.
(e)	TDM		–	CoMik or traditional RTOS with 1-level arb.
–	–	–	Any/pred.	CM, 1-level arb.
–	TDM & composable		–	NoC, DRAM, SRAM, 1-level arb.
–	Atomiser & composable		–	DRAM, SRAM, 1-level arb.

CompOSe uniquely achieve application slots of constant length (at the level of an individual cycle) by halting (clock gating) the processor until the WCET of the CoMik slot [19, 35].

4.2 Drivers

Drivers are software libraries that shield programmers from hardware-specific details. More specifically, we use drivers to:

1. Program the (virtual) resource before use (e.g., set up a NoC connection).
2. Use the virtual resource (e.g., program a DMA transfer, change the frequency of a processor).
3. Reset the virtual resource after use (e.g., remove reserved TDM slots).

We discuss each of these briefly below.

Hardware resources often require that they are programmed before they can be used. For example, an arbiter in front of a shared resource (e.g., SRAM or DRAM) needs to be aware of the budget of each requestor. For TDM, this means programming the slots reserved for each requestor, and for priority-based arbiters like CCSP or FBSP, a static priority is required. Delay blocks must be programmed with parameters to dynamically compute the WCRT, if they are used. If a resource is reprogrammed while it is active, e.g., to add or remove a requestor, then often extra care must be taken to not invalidate predictability or composability of requestors that are running [18]. To release a requestor's budget is usually harder than to add it, because we must ensure that no service units are still in the resource pipeline. We will return to this issue in Sect. 4.3. It is the task of a driver to ensure that a requestor's budget is programmed and reset correctly.

Furthermore, hardware resources often require that they are programmed in a certain way for use. For example, to send a block of data using a DMA, it must be programmed with the start address of where the data resides in the source memory, the block size, and the start address in the target memory. How to use the DMA also depends on whether it is used in a blocking or non-blocking manner. A driver takes care of these details and ensures that a DMA is used correctly.

The driver of the TIFU allows applications to access (virtualized) timers, schedule application interrupts, place the processor in sleep mode, and set the processor frequency. The driver ensures that these actions are done composably, i.e., without affecting other applications.

Two of the three types of driver functions (program, reset) are privileged in the sense that they manipulate the requestor budget on the resource. For this reason, their use is limited to the system application. The remaining driver function to use the resource is available to both the system and user applications.

4.2.1 Example Resource Drivers

The SRAM controller does not require programming, but its arbiter may do. It depends on the arbiter what has to be programmed, as discussed above. In fact, since the same arbiters can be used in front of the SRAM, DRAM, and similar resources, we use a general "shared resource" arbiter for all of these. Unlike the SRAM, the DRAM controller must be programmed with the DRAM commands that define the service unit [13], as well as parameters for logical to physical address translation, and this is done by the DRAM driver.

The NoC driver is complex because the source and destination NIs of a NoC connection must be programmed from the NI to which the driver is connected. The driver programs a path, TDM slots, flow-control credits, and enable/disable registers for each connection [22, 45, 49]. However, the NoC connection can be used without a driver after it has been programmed.

Before a requestor (application) can run on the processor, the microkernel must be programmed. This is straightforward since it uses a TDM arbiter that is predictable and composable. The processor is the only resource with frequency-scaling and clock-gating capabilities, for which the TIFU is used. Applications are allowed to change the frequency of the processor, while they run. However, as soon as their time slot ends, the frequency is first reset to the maximum frequency for the microkernel, and when it finishes, the frequency is reset to that of the next application [36]. This procedure is required to enable applications to manage the frequency at which they run without affecting the timing behavior of other applications, i.e., in a composable manner. Similarly, each application can use virtualized TIFU timers to measure time, to sleep for a certain time, to wake up at a certain deadline, and so on.

4.3 Virtual Resources and Their Management

Recall that each application runs inside its own composable VEP. A VEP is a hierarchical virtual resource that includes virtual DRAM, NoC connections, and virtual tiles further subdivided in virtual processors, DMAs and DMAMEMs, CMs, etc. Each virtual resource is the result of the driver programming a budget in a resource. The (hierarchical) budget is therefore the specification of the capacity that a requestor receives on a (hierarchical) resource.

Budgets are used to keep track of the reservations on the resource, such that the resource is not overloaded and such that multiple requestors do not use the same memory locations, time slots, etc. Without budgets, it is not possible to guarantee a minimum service to each requestor; in other words, using the resource would not be predictable.

The budget manager [16] is a library that offers several functions that are illustrated in Fig. 5. First, given a budget, the *reserve* function checks if the requested capacity (slots, memory range, energy, etc.) is available on the resource. If not, then the reservation fails. Therefore, loading a bundle, i.e., the application code plus its hierarchical budget, fails when not all of its requested virtual resources are

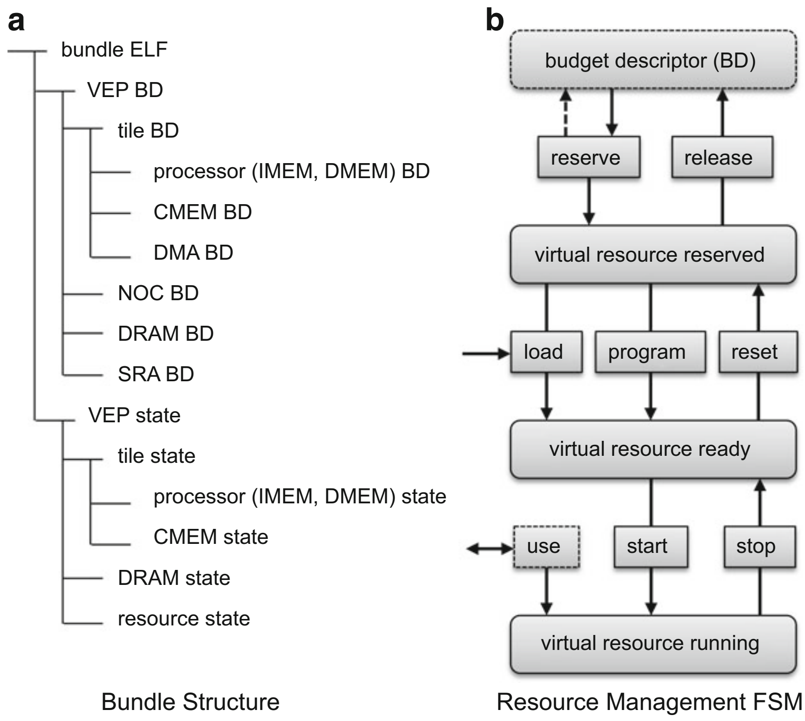


Fig. 5 (**a**) Bundle structure with Budget Descriptor (BD). (**b**) Resource management state diagram

available. This is desired, since it is not allowed to modify the VEPs of other running applications, as that would not be composable.

A successful budget reservation results in a *budget identifier* (cf. file handle), with which the budget can be *programmed* in the resource by the driver. Resources often have to be *loaded* with an initial state before they are started. For example, a virtual processor's IMEM requires a main function, interrupt and exception handlers, and its DMEM may need to be loaded with initial data. A DMA (driver) has to be loaded with a memory map to be able to perform range checking for memory protection. Programming defines the virtual resource, while loading defines the state of the virtual resource.

It is important to decouple reserving, programming, and loading. Since programming and loading a resource can be slow, and applications always require a set of virtual resources, it is better to first try to reserve all of them, before programming and loading them. Otherwise, the programmed resources have to reset. When all virtual resources specified by a hierarchical budget have been programmed, i.e., the VEP is ready to run, it can be *started*. As resources are programmed sequentially, and even from different processors, all virtual resources should be created before the application starts running. Otherwise, problems may occur, such as a DMA sending data before its NoC connection has been created. At this point in time, whatever software has been loaded in the VEP will run with the resource performance that was specified in the budget.

Applications *use* a running resource, either using a driver (e.g., for the DMA) or without (e.g., for the NoC and S/DRAM). Some resources, such as the processor and its TIFU, may be programmed (in a safe way) by the application. For example, the processor frequency may be changed by an application, and it is possible to read timers and program interrupts at deadlines.

To stop an application in its VEP and release its (hierarchical) budget, the reverse process is followed. First, all virtual resources must be *stopped*. This may be intricate because resources may be pipelined, requests may be waiting in buffers before the resource, and requests may flow through several resources. For example, a software task may write to DRAM using DMA and NoC. In general, software tasks on the processors are stopped first. For the mentioned example, the pipeline of DMA, NoC, and DRAM will be empty after some time. (For a real-time application, this time is known). It is important that *quiescence* of resources (i.e., being in the ready state) can be observed to avoid leaving a resource in an inconsistent state or to lose data. Then the DMA, NoC, and DRAM virtual resources can be stopped.

When all virtual resources are in the ready state, the virtual resource can be *reset*, and the budgets *released*.

The budget management library can only be used by the system application to ensure that applications cannot change their own VEPs, which is crucial for composability and predictability. Together, the resource driver and budget manager comprise the *resource management* library. Usually all resources of the same type are controlled from one location, the exception being the processor tiles that are managed locally. Resource management may thus be distributed in the platform (see Sect. 4.5 and [48]).

4.4 Synchronization Libraries and Programming Models

Tasks in the same application synchronize and communicate data. As described in Sect. 3.7, the CompSOC platform offers atomicity of data transfers up to a certain size, which is the minimum requirement to safely communicate. However, it is not enough, because atomically written data may be overwritten before it has been read. Fundamentally, there are two ways to synchronize, either based on data or on time. First, we describe the data synchronization styles offered by the CompSOC platform: barriers and First-In First-Out (FIFO) queues. After that, we describe how we offer timed synchronization in a synchronous platform or on top of barrier synchronization when GALS is used.

4.4.1 Barrier Synchronization

Barrier synchronization is implemented as a library. When using a barrier to synchronize, each task increments its own dedicated counter (starting at zero) and then waits until all other tasks have done so by repeatedly reading (polling) all counters. This is shown in Fig. 6a. Starting at barrier b, task 1 updates its counter to $b + 1$ and then polls task 2's counter. As soon as task 2 has updated its counter, barrier $b + 1$ is reached, which is observed by the next polls of both tasks. Each counter is a single word in a memory, and is written atomically and independently of all other counters. It does not matter if all counters together are read atomically or not because counters are only incremented, and it is therefore not possible to miss a barrier. However, the assumption that all counters start at zero is problematic. At

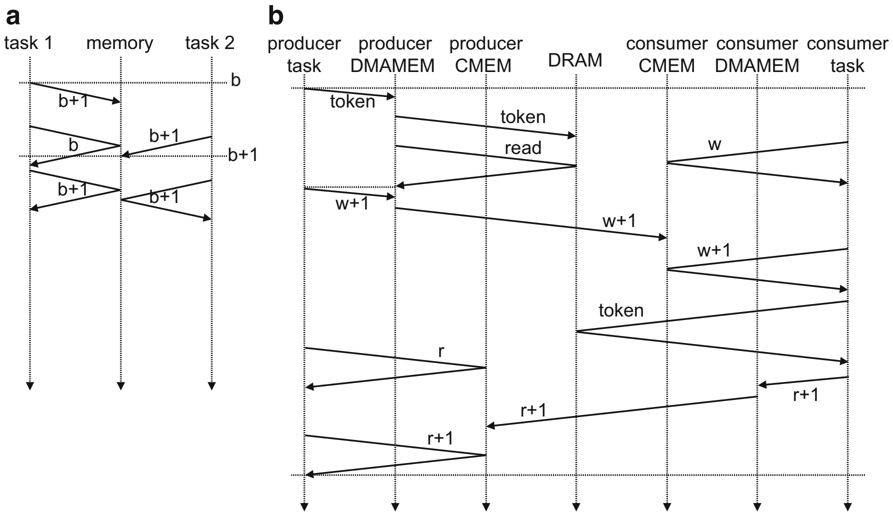


Fig. 6 (**a**) Barrier synchronization and (**b**) FIFO synchronization using DRAM. Each *arrow* points from master to slave (for the read/write request), possibly continued back to the master (for the read response). Labels indicate (b)arrier, (w)rite, and (r)ead data

reset, SRAM and DRAM contain random values, and one task has to initialize the barrier counters before the others start, which is what the barrier intended to solve. For this reason all processors except the boot processor are kept in reset until their boot programs have been loaded and the barrier initialized.

4.4.2 FIFO Queues

In a streaming or data-driven system, tasks usually communicate using First-In First-Out (FIFO) queues that ensure no data is lost. Tasks block on a full or empty FIFO and wait until there is space to write or data to read. (In a cooperatively arbitrated system, a task could yield upon blocking.) Optionally, a task can poll for space or data, and continue with processing if it is not available. Although barriers work well to synchronize the flows of control of concurrent tasks or updates to global shared state, they are cumbersome for FIFO communication. For this reason, we offer a FIFO library that also forms the basis of KPN and dataflow libraries.

The C-HEAP protocol [41] implements FIFO in distributed shared memory without hardware support, such as locks, mutexes, or processor instructions (other than read and write). A FIFO has a single producer and consumer. Data is produced and consumed as constant-size tokens, and the FIFO has a fixed token capacity. We must ensure that tokens are only written by the producer in memory locations that are guaranteed to have been freed by the consumer (i.e., it has read the token). Similarly, the consumer must only read memory locations when the producer has finished writing the token, i.e., the data is valid. The basic concept is that the start location of valid tokens is indicated by *a write pointer that is only written by the producer* and that the start of space for new tokens is indicated by *a read pointer that is only written by the consumer*. The pointers are only updated after tokens have been written to or read from the destination memory. Because each pointer is only written (atomically) by one task, consistency of the FIFO is guaranteed. No token is overwritten before it is read or accidentally read multiple times. The difference between the pointers indicates the FIFO filling, and the FIFO is full or empty when the read and write pointer are equal. We use a circular buffer, which means that wrapping of pointers must be taken into account.

Figure 7a illustrates the simplest FIFO (L), where the tokens and the read and write pointers all reside in DMEM. Only tasks on this processor can communicate using this FIFO, as shown in Fig. 7b. When a FIFO is too large to fit in the local memories of a processor tile, it can be stored in an SRAM or DRAM that is connected to the NoC, as shown by FIFO F in Fig. 7a. The protocol is unchanged, but multiple memories and DMAs are now used, as shown in Fig. 6b.

The producer of a token first checks for space for a new token (blocking *claim_space* or non-blocking *poll_space*). The token is then written into its local DMAMEM and copied (*write_token*) to the remote memory when it is ready. To ensure that the token has been written in the remote memory, the processor issues a (dummy) read of the last data word of the token, again using the DMA. Since the NoC connections do not reorder read and write transactions, the data is guaranteed to be written when the read data is returned. When the dummy read data has been received and discarded, the write counter (CW) is updated ($w+1$) in the local

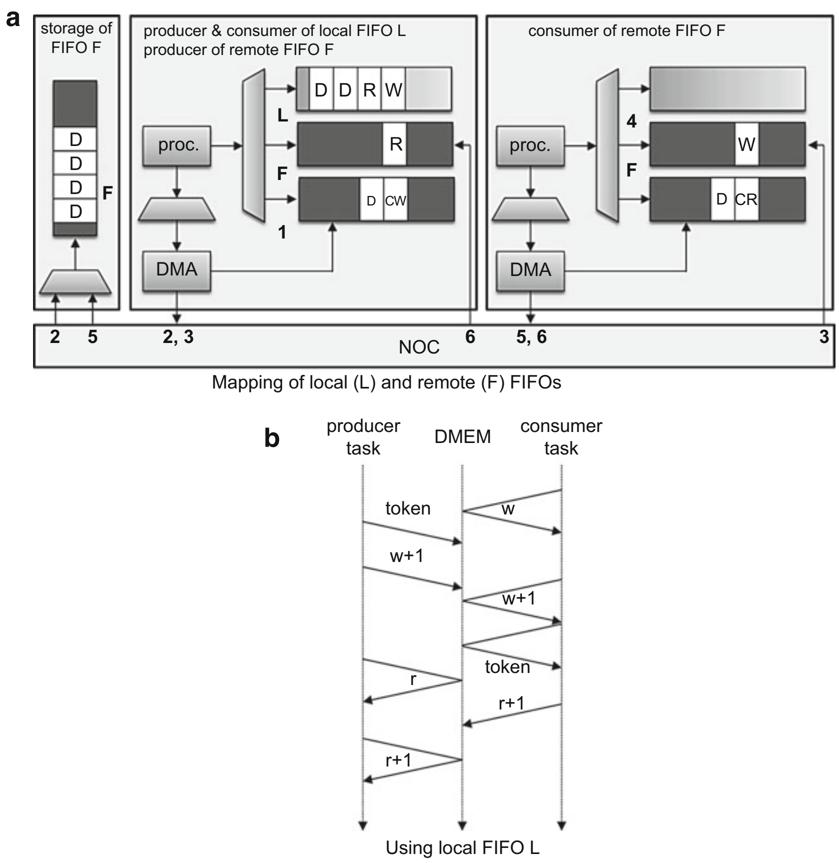


Fig. 7 (**a**) FIFO F with data in remote DRAM and FIFO L with data in local DMEM. (**b**) Using local FIFO L. *D* are tokens, *R* and *W* are read and write pointers, respectively

DMAMEM, and the DMA is programmed to copy it to the CM (W) of the consumer (*release_token*).

Assume that the FIFO was empty until the write counter update. The consumer would have known this because the read CR and write W pointers in its DMAMEM and CM would have been equal: $w = r$ in Fig. 6b. The consumer would have been waiting until the write counter update (blocking *claim_token* or non-blocking *poll_token*). When the write counter update is detected, the consumer instructs its DMA to copy the token from the remote memory to its local DMAMEM (*read_token*). When the consumer no longer requires the token, it is released (*release_space*) by increasing the local read counter CR and writing it to the producer's CM (R).

The tokens and counters may be placed in any memory but preferably as close as possible to the task that requires reading most often. The highest performance is obtained by maximizing the use of posted (non-blocking) write transactions and minimizing the number of read transactions over the NoC. This is achieved when placing the write counter W in the consumer's CM and the read counter R in the producer's CM, as well as keeping local copies of the counters (CW, CR). The data is best placed in the consumer's CM, assuming it fits. The *claim_space*, *write_token*, and *release_token* are often combined in a simpler *send_token*. *receive_token* is defined conversely. Although simpler, they require space for one more token and copy action.

4.4.3 Timed Communication

In a time-triggered system, jobs communicate through shared locations (memory regions) with space for one token. The token may be written multiple times before it is read, and read multiple times before it is written again. If under- and oversampling are not desired, then the usual way to synchronize the producer and consumer is by (statically) scheduling access to the shared location in time. For example, with token production with a period of 1 ms starting at time 0, we can schedule token consumption with the same period of 1 ms but with a starting time of 1 ms. As long as the producer ensures that the token has been written completely in the shared location before the period deadline, the consumer can safely read the token. This approach works as long as a sufficiently accurate common notion of time is available to the producer and the consumer, and the production and transport of the token is predictable and guaranteed to fit within the production and consumption periods.

In a fully synchronous CompSOC platform, i.e., where the NoC and all processor and memory tiles operate on the same clock, time-triggered communication works without surprises. The simplest scenario is that the producer computes the token and writes it to the shared location. Using the TIFU, it then sleeps until the periodic deadline and then restarts. The consumer(s) sleep until the periodic deadline and then copy the token [55]. Care must be taken to not write tokens too late (i.e., arriving after the deadline) and not too early (i.e., when consumers are still reading the previous token). Even without a global clock or notion of time, we can use barriers to safely implement time-triggered communication in a GALS MPSoC. We show a basic implementation in Fig. 8. A producer writes its token and increments and waits on the producer barrier. The producer barrier is released periodically by a clock task, after which consumers indicate in a consumer barrier when they have consumed the token. The producer waits for the consumer barrier before it updates the token and the producer barrier. Producers and consumers follow both barriers to detect any late consumption or production, which could result in transfer of corrupt data.

4.4.4 Programming Models

Different applications of different time criticalities will be developed with different requirements on timing. Real-time applications must be analyzable, which imply a more restrictive programming model (e.g., dataflow or time triggered) than a programming model that non-real-time applications can use, such as Kahn Process Network (KPN) or even arbitrary C. CompSOC therefore offers multiple

```
clock:
while true do
  wait until next periodic deadline
  increment_and_wait(producer_barrier)
end while
```

```
producer:
while true do
  write_token
  increment_and_wait(producer_barrier)
  increment_and_wait(consumer_barrier)
end while
```

```
consumer:
while true do
  increment_and_wait(producer_barrier)
  read_token
  increment_and_wait(consumer_barrier)
end while
```

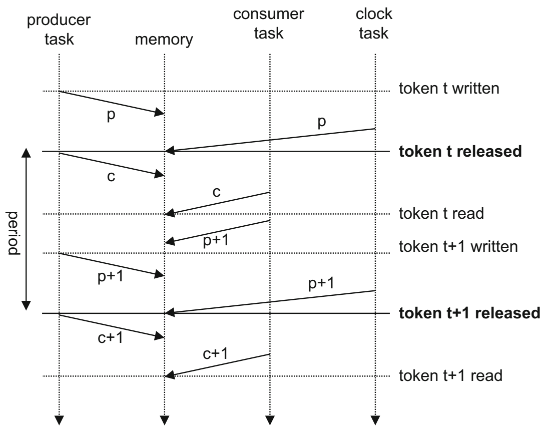
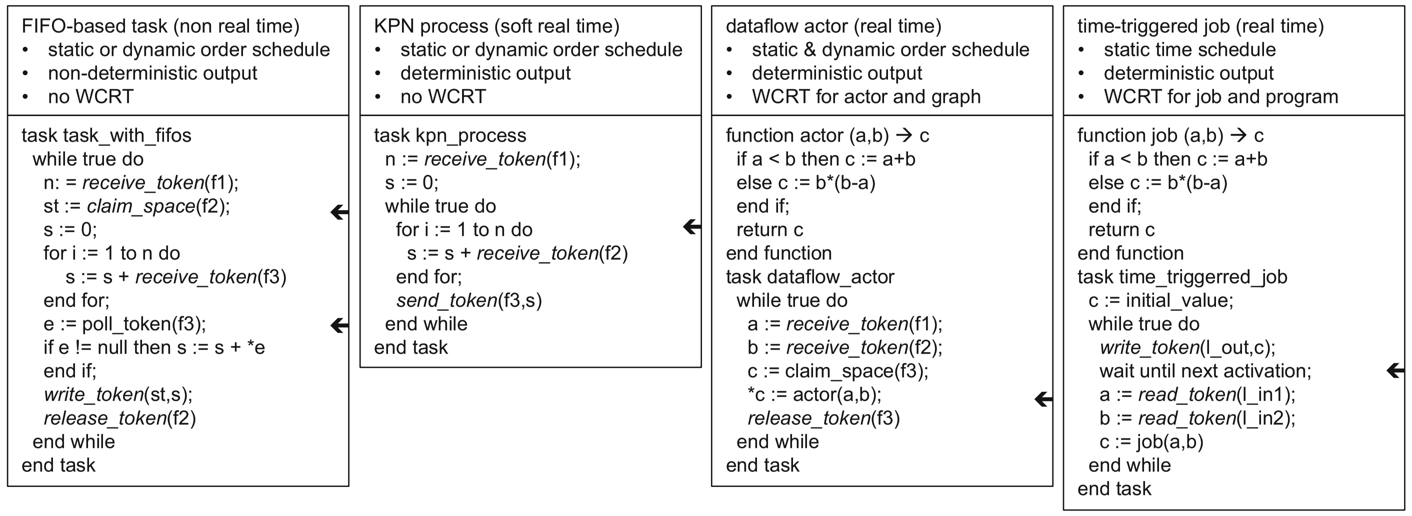


Fig. 8 Timed communication using barriers

programming models, such that each application can be designed as easily and efficiently as possible.

Although we do not encourage it, an application programmer can use multiple tasks and communicate using DMAs, without using CompSOC's synchronization libraries. All the programming models are implemented using tasks (or threads), using the templates shown in Fig. 9. We describe them from least restrictive, and thus least analyzable, to most restrictive [32]. In all cases, the task graph is static.

1. *Non-blocking and blocking FIFO communication*. The first arrow illustrates that *claim_space* and its corresponding *write_token* and *release_token* may have arbitrary code between them. The second arrow illustrates that this task uses polling, which could result in nondeterministic behavior. This is because it depends on the scheduling of other tasks, which may be non-deterministic due to GALS. This is not recommended unless the task uses some kind of sampling algorithm.
2. *Kahn Process Network (KPN)* with finite FIFOs use only blocking *send_token* and *receive_token*. Processes may use arbitrary control flow, which may result in a data-dependent number of tokens being consumed or produced (indicated by the arrow). It is, in general, not possible to compute the WCET of a process, and KPN is suitable for non-real-time or perhaps soft-real-time applications.
3. (Not shown in the figure.) Programmers can use *barriers*, as described earlier in this section, to synchronize multiple tasks using patterns such as fork-join.
4. *Dataflow* applications contain actors that consume all input tokens before computing the output tokens, which are released at the end. Note that an actor is a stateless function, unlike a KPN process. State can be implemented with a self-edge, i.e., a channel from the actor to itself. A programmer only writes the actor functions; the `dataflow_actor` wrapper is automatically generated. If the code in the actor function has a WCET, and all resources are shared



FIFO-based task (non real time)
- static or dynamic order schedule
- non-deterministic output
- no WCRT

```
task task_with_fifos
 while true do
   n: = receive_token(f1);
   st := claim_space(f2);
   s := 0;
   for i := 1 to n do
       s := s + receive_token(f3)
   end for;
   e := poll_token(f3);
   if e != null then s := s + *e
   end if;
   write_token(st,s);
   release_token(f2)
 end while
end task
```

KPN process (soft real time)
- static or dynamic order schedule
- deterministic output
- no WCRT

```
task kpn_process
  n := receive_token(f1);
  s := 0;
  while true do
    for i := 1 to n do
      s := s + receive_token(f2)
    end for;
    send_token(f3,s)
  end while
end task
```

dataflow actor (real time)
- static & dynamic order schedule
- deterministic output
- WCRT for actor and graph

```
function actor (a,b) → c
  if a < b then c := a+b
  else c := b*(b-a)
  end if;
  return c
end function
task dataflow_actor
  while true do
    a := receive_token(f1);
    b := receive_token(f2);
    c := claim_space(f3);
    *c := actor(a,b);
    release_token(f3)
  end while
end task
```

time-triggered job (real time)
- static time schedule
- deterministic output
- WCRT for job and program

```
function job (a,b) → c
  if a < b then c := a+b
  else c := b*(b-a)
  end if;
  return c
end function
task time_triggerred_job
  c := initial_value;
  while true do
    write_token(l_out,c);
    wait until next activation;
    a := read_token(l_in1);
    b := read_token(l_in2);
    c := job(a,b)
  end while
end task
```

Fig. 9 Programming models

predictably, then the throughput and WCRT of the dataflow application as a whole can be computed [37]. As a result the dataflow model of computation is suitable for real-time applications. Chapter ▶ "SysteMoC: A Data-Flow Programming Language for Codesign" contains more information about the dataflow model of computation, and chapter ▶ "ForSyDe: System Design Using a Functional Language and Models of Computation" contains more information about dataflow and other models of computation.

5. *Time-triggered communication*. A job (non-blocking) reads tokens from its input locations (except for the first execution), computes outputs, and then (non-blocking) writes tokens in its output locations. Jobs communicate using the time-triggered synchronization library. Time-triggered applications are executed periodically according to a global static schedule, for example, using a global clock or barriers. The code of each job (computation on processor, as well as communication on NoC, and access to shared memories) must have a WCRT smaller than the period, which determines the real-time performance of the application. As a result, the time-triggered model of computation is suitable for real-time applications. Chapter ▶ "Networked Real-Time Embedded Systems" contains more information about the time-triggered model of computation.

4.5 System Application and Application Loading

The system application is like a normal application with the exception that it has access to privileged resource management library functions. The system application creates and removes VEPs, and starts and stops applications in running VEPs. The system application has a task on each processor tile, with one being the master. The tasks synchronize and communicate using the barrier and FIFO libraries.

Figure 10 illustrates the six (simplified) steps to start an application [48]:

1. The system application detects a bundle.
2. It hierarchically creates a VEP in a distributed manner.
3. It loads the bootloader in the VEP, and starts the VEP and bootloader.
4. In the VEP, the bootloader loads the application code and data.
5. For multitask applications, the bootloader creates tasks and communication channels.
6. The bootloader finishes and the application runs.

These steps are now explained in more detail. The master task detects in bundles at a predefined location in shared memory and loads them based on some strategy (e.g., as soon as they have been uploaded from outside the ASIC/FPGA or based on certain triggers [26]). The bundle contains the budget description of the VEP that the application requires to run.

The master task analyzes the hierarchical budget descriptor and sends the budget descriptors of the processor tiles to the slave tasks on those tiles. The slave tasks locally reserve and program the required resources and notify the master task. Other

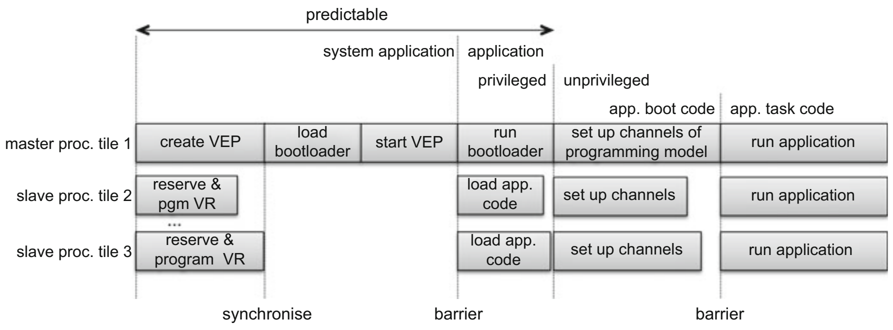


Fig. 10 Loading and starting an application

resources, such as the NoC and shared memories, are similarly reserved by the master or a slave task. If any budget reservation fails, the entire VEP creation fails, and all its resources are released. After the VEP has been created, it is loaded with a standard small bootloader application. After starting all the virtual resources, the bootloader loads and starts the application code that is specified in the bundle. In this way, loading an application, which may take some time, is performed in the VEP of the application instead of that of the system application. Multiple applications may be loading and/or running simultaneously because each VEP is independent.

The bootloader finishes and hands over to the application code. A non-preemptive application on a single processor (Fig. 4a, b) now runs. Otherwise, the application contains tasks (actors, processes, etc.) that synchronize or communicate using barriers, FIFOs, etc., and then these are set up using the relevant programming model library (see Fig. 3). When all tiles on which the application runs have finished this set-up phase, they synchronize using a barrier, and the distributed application starts.

The time from observing arrival of a bundle and starting the application is predictable and can also be made composable. In other words, the time from detecting a bundle until it is running is independent of other applications [48].

4.6 Conclusions

The software architecture of the CompSOC platform is quite complex, but this is mostly due to its versatility. The processor is the only software-arbitrated resource, and virtualization of multiple applications is inherently quite involved. The essential concept of the software architecture is the bundle which contains application code together with the requirements for its VEP structuring. Building on this, the resource management library provides a uniform way to reserve, program, and start heterogeneous resources in a hierarchical and distributed manner. After the system application creates a VEP, the bootloader library is used to load and start applications, of any kind. Various communication and programming model libraries are provided for both system application and user applications.

5 Example CompSOC Platform Instance

The CompSOC platform is a template that can be instantiated [14] and used in many different ways and applications. In this section, we describe a demonstrator showing:

1. Mixed-time-criticality: several concurrently executing applications with and without real-time requirements.
2. Predictability: guaranteed performance for real-time applications.
3. Composability: multiple applications loading and executing composably.
4. Multiple models of computation.

The demonstrator, shown in Fig. 11, contains five applications:

1. A *high-performance real-time motion controller for a two-mass spring motion system* written as a time-triggered application, shown in Fig. 12. The controller is

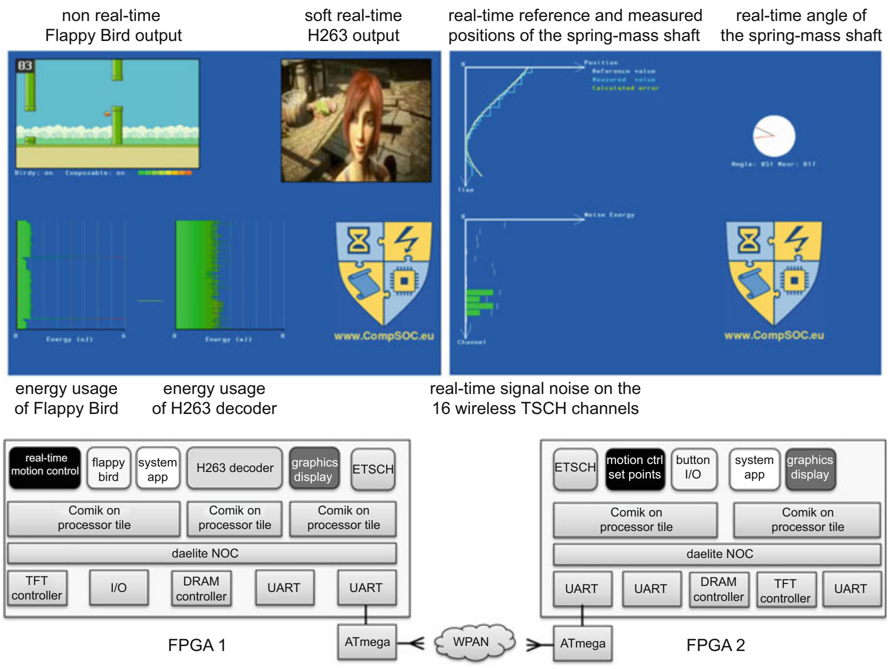


Fig. 11 Mapping of the applications on the two FPGA boards (*bottom*) and the display output of the two boards (*top*)

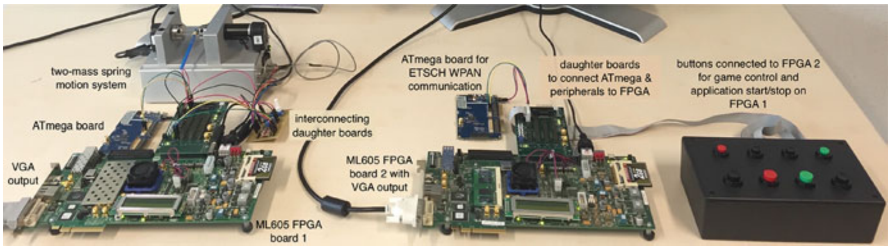

two-mass spring motion system
ATmega board
VGA output
interconnecting daughter boards
ATmega board for ETSCH WPAN communication
daughter boards to connect ATmega & peripherals to FPGA
buttons connected to FPGA 2 for game control and application start/stop on FPGA 1
ML605 FPGA board 2 with VGA output
ML605 FPGA board 1

Fig. 12 Demonstration CompSOC hardware setup, showing two Xilinx ML605 FPGA boards with ATmega 256RFR2 boards for WPAN, interconnecting daughter boards, a two-mass spring motion system, and a video game controller

a software task that regulates the current to a DC motor that drives and controls the rotational position of a flexible shaft [8]. The positions of the shaft are measured by optical encoders and read out by the software controller.

2. A CompSOC implementation of the non-real-time *popular video game Flappy Bird*. The user controls it with a simple hardware button. For demonstration purposes, the game can be played in composable mode, without interference from other applications, or in non-composable mode where the height of the bird's jumps depends on the processor load.
3. A *(soft) real-time H263 video decoder* [39], written as a dataflow application of six actors.
4. A *non-real-time graphics application* displays information on a Thin-Film Transistor (TFT) screen by updating a (triple) video buffer, which is read out by a hardware TFT controller. The graphics application has a (composable) sampling interface with the motion controller, Flappy Bird, and H263 video decoder to draw graphs and compose the image of several sub-images (video and game outputs).
5. The *system application* that manages the virtual execution platforms, including the starting and stopping of applications at run time.

The applications are mapped on two Xilinx ML605 FPGA boards, shown in Fig. 12. The first board contains a CompSOC platform with three processor tiles each running the CoMik microkernel, NoC, DRAM, and several peripherals (TFT, UART) connected to the NoC. The sensing and actuation of the spring motion system is memory mapped on the NoC using an intermediate hardware I/O block. The second FPGA board has only two processor tiles and has the hardware controller for the Flappy Bird game instead of the motion-control hardware. Each ML605 board contains an Extended TSCH (ETSCH) application that connects to an ATmega 256RFR2 board using a UART. In this way, tasks of an application mapped on both FPGA boards can communicate wirelessly using the Extended TSCH (ETSCH) [51] extension of the IEEE 802.15.4e Time-Synchronised Channel Hopping (TSCH) standard. (E)TSCH uses frequency hopping and TDM for robust real-time performance. The architecture is GALS both within and between FPGAs.

Figure 11 illustrates the mapping of applications on (multiple) boards and (multiple) processors, as well as the display output of the boards. Of particular note are the following. Flappy Bird runs and displays on FPGA 1, but the game controller hardware (buttons) is connected to FPGA 2. The latency from the button I/O application on FPGA 2 to the Flappy Bird application on FPGA 1 is mostly due to the wireless connection and is just within the limits for a playable game. Using other buttons connected to FPGA 2, the system application on FPGA 1 can be instructed to start and stop Flappy Bird in different modes, without affecting other running applications. The graphics application on FPGA 2 displays the performance of the real-time motion controller with real-time graphs of the reference, measured angles of the spring-mass shaft and the calculated error. It also displays the real-time signal noise on the 16 wireless TSCH channels that it obtains from the ATmega board.

All the concepts of Sect. 2 are proven in the demonstrator. The system application dynamically loads application bundles by creating virtual execution platforms that are predictable and composable. Although the platform contains only five processor tiles in total, they are interconnected globally asynchronously using two TDM NoCs and a robust wireless TDM connection. The finite scheduling interval and efficient arbitration are optimizations and proven on the various resources, as described in previous sections.

6 Related Work

A vast literature is available on predictability, ranging from single resources to multiple shared resources, using a variety of analytical approaches, such as real-time calculus [52], dataflow [50], and response-time analysis for priority-based scheduling [10].

In the literature, composability is especially addressed for safety-critical applications, such as those found in the automotive [43] and aeronautical [46,56] industries. Temporal and spatial partitioning [46, 56] are addressed most often, increasingly using microkernels and RTOS, such as LynxOS-178, VxWorks 653, INTEGRITY, and PikeOS.

Our focus is on complete platforms that are predictable, are composable, or offer mixed-time-criticality. Note however, that in the literature, definitions of predictability and composability vary. Apart from CompSOC, notable mature platforms include the Transport-Triggered Architecture (TTA) [29], Giotto [25] and LET [28], and PRET [11]. A number of collaborative projects have developed platforms of varying degrees of maturity, including Flextiles [26], T-CREST [47], PARMERASA [54], MULTIPARTES [53], P-SOCRATES [44], DREAMS [42], and CERTAINTY [12]. The resource-management frameworks of [7, 31, 40] are noteworthy because their approaches consider the entire system, with different resources.

These platforms focus on different aspects, and they are often updated, which means that comparisons change over time. At a high level, all platforms offer real-time performance to at least one application and sometimes to multiple applications. Dynamic loading, starting, and stopping of applications are sometimes supported. Mixed-time-criticality, where non-real-time and real-time applications coexist, is also often claimed. However, the levels of predictability, the formalisms used, and the level of automation vary considerably. Increasingly platforms claim to be composable, especially in the sense of temporal and space partitioning, but not to the extreme extent of CompSOC, i.e., no interference at the level of individual clock cycles.

In general, points on which platforms may be compared include the formalism for predictability, model(s) of computation offered, single or multiple applications, global or distributed arbitration, single-level or multi-level arbitration, work-conserving arbitration or not, use of budgets or virtual resources or not, time-triggered or data-driven execution, and distributed-shared-memory or message-passing architecture.

7 Conclusions

In this chapter, we first defined what a mixed-time-criticality system is and what its requirements are. After defining the concepts that such systems should follow, we described CompSOC, which is one example of a mixed-time-criticality platform. We described, in detail, how multiple resources, such as processors, memories, and interconnect, are combined into a larger hardware platform, and especially how they are shared between applications using different arbitration schemes. Following this, the software architecture that transforms the single hardware platform into multiple virtual execution platforms, one per application, was described.

Acknowledgments The development of CompSOC has been partially funded by European grants, including CATRENE CT217 RESIST; ARTEMIS 621429 EMC2, 621353 DEWI, 621439 ALMARVI, and ECSEL 692455 ENABLE-S3.

References

1. Akesson B, Goossens K (2011) Architectures and modeling of predictable memory controllers for improved system integration. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), Grenoble. IEEE, pp 1–6
2. Akesson B, Goossens K (2011) Memory controllers for real-time embedded systems. Embedded systems series, 1st edn. Springer, New York
3. Akesson B, Hansson A, Goossens K (2009) Composable resource sharing based on latency-rate servers. In: Proceedings of Euromicro symposium on digital system design (DSD), Patras, pp 547–555
4. Akesson B, Molnos A, Hansson A, Ambrose Angelo J, Goossens K (2010) Composability and predictability for independent application development, verification, and execution. In: Hübner M, Becker J (eds) Multiprocessor system-on-chip – hardware design and tool integration, circuits and systems, chap. 2. Springer, Heidelberg, pp 25–56

5. Akesson B, Steffens L, Strooisma E, Goossens K (2008) Real-time scheduling using credit-controlled static-priority arbitration. In: Proceedings of international conference on embedded and real-time computing systems and applications (RTCSA). IEEE Computer Society, Washington, DC, pp 3–14
6. Beyranvand Nejad A, Molnos A, Goossens K (2013) A software-based technique enabling composable hierarchical preemptive scheduling for time-triggered applications. In: Proceedings of international conference on embedded and real-time computing systems and applications (RTCSA), Taipei
7. Bini E, Buttazzo G, Eker J, Schorr S, Guerra R, Fohler G, Arzen KE, Romero Segovia V, Scordino C (2011) Resource management on multicore systems: the ACTORS approach. Proc Microarch (MICRO) 31(1):72–81
8. Bolder J, Oomen T (2014) Rational basis functions in iterative learning control – with experimental verification on a motion system. IEEE Trans Control Syst Technol 23(2): 722–729
9. Chandrasekar K, Akesson B, Goossens K (2012) Run-time power-down strategies for real-time SDRAM memory controllers. In: Proceedings of design automation conference (DAC). ACM, New York, pp 988–993
10. Davis RI, Burns A (2011) A survey of hard real-time scheduling for multiprocessor systems. ACM Comput Surv (CSUR) 43(4):35
11. Edwards SA, Lee EA (2007) The case for the precision timed (pret) machine. In: Proceedings of the 44th annual design automation conference, New York. ACM, pp 264–265
12. Giannopoulou G, Stoimenov N, Huang P, Thiele L, de Dinechin BD (2015) Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources. J Real-Time Syst 52(4):399–449
13. Goossens S, Akesson B, Goossens K (2013) Conservative open-page policy for mixed time-criticality memory controllers. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), Grenoble, pp 525–530
14. Goossens K, Azevedo A, Chandrasekar K, Gomony MD, Goossens S, Koedam M, Li Y, Mirzoyan D, Molnos A, Beyranvand Nejad A, Nelson A, Sinha S (2013) Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. ACM Spec Interest Group Embed Syst (SIGBED) Rev 10(3):23–34
15. Goossens K, Hansson A (2010) The Aethereal network on chip after ten years: goals, evolution, lessons, and future. In: Proceedings of design automation conference (DAC), Anaheim, pp 306–311
16. Goossens K, Koedam M, Sinha S, Nelson A, Geilen M (2015) Run-time middleware to support real-time system scenarios. In: Proceedings of European conference on circuit theory and design (ECCTD), Trondheim
17. Goossens S, Kouters T, Akesson B, Goossens K (2012) Memory-map selection for firm real-time SDRAM controllers. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE). IEEE, Dresden, pp 828–831
18. Goossens S, Kuijsten J, Akesson B, Goossens K (2013) A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In: International conference on hardware/software codesign and system synthesis (CODES+ISSS), Montreal, pp 1–10
19. Hansson A, Ekerhult M, Molnos A, Milutinovic A, Nelson A, Ambrose J, Goossens K (2011) Design and implementation of an operating system for composable processor sharing. J Micromech Microeng (MICPRO) 35(2):246–260. Elsevier. Special issue on network-on-chip architectures and design methodologies
20. Hansson A, Goossens K (2007) Trade-offs in the configuration of a network on chip for multiple use-cases. In: Proceedings of international symposium on networks on chip (NOCS). IEEE Computer Society, Washington, DC, pp 233–242
21. Hansson A, Goossens K (2009) An on-chip interconnect and protocol stack for multiple communication paradigms and programming models. In: International conference on hardware/software codesign and system synthesis (CODES+ISSS). ACM, New York, pp 99–108

22. Hansson A, Goossens K (2010) On-Chip interconnect with aelite: composable and predictable systems. Embedded systems series. Springer, New York
23. Hansson A, Goossens K, Bekooij M, Huisken J (2009) CoMPSoC: a template for composable and predictable multi-processor system on chips. ACM Trans Des Autom Electron Syst 14(1):1–24
24. Hansson A, Wiggers M, Moonen A, Goossens K, Bekooij M (2009) Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. IET Comput Digit Tech 3(5):398–412
25. Henzinger TA, Horowitz B, Kirsch CM (2003) Giotto: a time-triggered language for embedded programming. Proc IEEE 91(1):84–99
26. Jansen B, Schwiegelshohn F, Koedam M, Duhem F, Masing L, Werner S, Huriaux C, Courtay A, Wheatley E, Goossens K, Lemonnier F, Millet P, Becker J, Sentieys O, Hübner M (2015) Designing applications for heterogeneous many-core architectures with the FlexTiles platform. In: Proceedings of International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), Samos
27. Kasapaki E, Sorensen RB, Müller C, Goossens K, Schoeberl M, Sparso J (2015) Argo: a real-time network-on-chip architecture with an efficient GALS implementation. IEEE Trans Very Large Scale Integr Syst (TVLSI) 99(2):479–492
28. Kirsch C, Sokolova A (2012) The logical execution time paradigm. In: Chakraborty S, Eberspächer J (eds) Advances in real-time systems (ARTS). Springer, Berlin/Heidelberg, pp 103–120
29. Kopetz H (2011) Real-time systems: design principles for distributed embedded applications. Springer, Heidelberg
30. Li Y, Salunkhe H, Bastos J, Moreira O, Akesson B, Goossens K (2015) Mode-controlled data-flow modeling of real-time memory controllers. In: Proceedings of embedded systems for real-time multimedia (ESTIMedia), Amsterdam
31. Moreira O, Corporaal H (2014) Scheduling real-time streaming applications onto an embedded multiprocessor. Embedded systems series, vol 24. Springer, Cham
32. Nejad AB, Molnos A, Goossens K (2013) A unified execution model for multiple computation models of streaming applications on a composable MPSoC. J Syst Archit (JSA) 59(10, part C), 1032–1046. Elsevier
33. Nejad AB, Molnos A, Martinez ME, Goossens K (2013) A hardware/software platform for QoS bridging over multi-chip NoC-based systems. J Parallel Comput (PARCO)39(9):424–441. Elsevier
34. Nelson A (2014) Composable and predictable power management. Ph.D. thesis, Delft University of Technology
35. Nelson A, Beyranvand Nejad A, Molnos A, Koedam M, Goossens K (2014) CoMik: a predictable and cycle-accurately composable real-time microkernel. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), Dresden
36. Nelson A, Goossens K (2015) Distributed power management of real-time applications on a GALS multiprocessor SOC. In: Proceedings of ACM international conference on embedded software (EMSOFT), Amsterdam
37. Nelson A, Goossens K, Akesson B (2015) Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor SOC. J Syst Archit (JSA) 61(9):435–448
38. Nelson A, Molnos A, Goossens K (2011) Composable power management with energy and power budgets per application. In: Proceedings of international conference on embedded computer systems: architectures, modeling and simulation (SAMOS), Samos, pp 396–403
39. Nelson A, Molnos A, Goossens K (2012) Power versus quality trade-offs for adaptive real-time applications. In: Proceedings of embedded systems for real-time multimedia (ESTIMedia), Tampere, pp 75–84
40. Nesbit KJ, Smith JE, Moreto M, Cazorla FJ, Ramirez A, Valero M (2008) Multicore resource management. Proc Microarch (MICRO) 28(3):6–16

41. Nieuwland A, Kang J, Gangwal OP, Sethuraman R, Busá N, Goossens K, Peset Llopis R, Lippens P (2002) C-HEAP: a heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. ACM Trans Des Autom Embed Syst 7(3):233–270
42. Obermaisser R, Weber D (2014) Architectures for mixed-criticality systems based on networked multi-core chips. In: Proceedings of Conference on Emerging Technology and Factory Automation (ETFA), Barcelona, pp 1–10
43. Pelz G et al (2005) Automotive system design and autosar. In: Advances in design and specification languages for SoCs. Springer, New York, pp 293–305
44. Pinho LM, Nelis V, Yomsi PM, Quinones E, Bertogna M, Burgio P, Marongiu A, Scordino C, Gai P, Ramponi M, Mardiak M (2015) P-socrates: a parallel software framework for time-critical many-core systems. J Microprocess Microsyst 39(8):1190–1203. Elsevier
45. Rădulescu A, Dielissen J, González Pestana S, Gangwal OP, Rijpkema E, Wielage P, Goossens K (2005) An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. IEEE Trans CAD Integr Circuits Syst 24(1):4–17
46. Rushby J (1999) Partitioning in avionics architectures: requirements, mechanisms, and assurance. Technical report, NASA
47. Schoeberl M, Abbaspour S, Akesson B, Audsley N, Capasso R, Garside J, Goossens K, Goossens S, Hansen S, Heckmann R, Hepp S, Huber B, Jordan A, Kasapaki E, Knoop J, Li Y, Prokesch D, Puffitsch W, Puschner P, Rocha A, Silva C, Sparsø J, Tocchi A (2015) T-CREST: time-predictable multi-core architecture for embedded systems. J Syst Archit (JSA) 61(9):449–471. Elsevier
48. Sinha S, Koedam M, Breaban G, Nelson A, Nejad A, Geilen M, Goossens K (2015) Composable and predictable dynamic loading for time-critical partitioned systems on multiprocessor architectures. J Microprocess Microsyst (MICPRO) 39(8):1087–1107
49. Stefan R, Molnos A, Goossens K (2014) dAElite: a TDM NoC supporting QoS, multicast, and fast connection set-up. IEEE Trans Comput 63(3):583–594
50. Stuijk S, Basten T, Geilen M, Corporaal H (2007) Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In: Proceedings of design automation conference (DAC), San Diego, pp 777–782
51. Tavakoli R, Nabi M, Basten T, Goossens K (2015) Enhanced time-slotted channel hopping in wsns using non-intrusive channel-quality estimation. In: Proceedings of international conference on mobile ad hoc and sensor systems (MASS), Dallas
52. Thiele L, Chakraborty S, Naedele M (2000) Real-time calculus for scheduling hard real-time systems. In: The 2000 IEEE international symposium on circuits and systems, 2000. Proceedings. ISCAS 2000, Geneva, vol 4. IEEE, pp 101–104
53. Trujillo S, Crespo A, Alonso A, Perez J (2014) Multipartes: multi-core partitioning and virtualization for easing the certification of mixed-criticality systems. J Microprocess Microsyst 38(8, part B):921–932. Elsevier
54. Ungerer T, Bradatsch C, Gerdes M, Kluge F, Jahr R, Mische J, Fernandes J, Zaykov PG, Petrov Z, Boddeker B, Kehr S, Regler H, Hugl A, Rochange C, Ozaktas H, Casse H, Bonenfant A, Sainrat P, Broster I, Lay N, George D, Quinones E, Panic M, Abella J, Cazorla F, Uhrig S, Rohde M, Pyka A (2013) parmerasa – multi-core execution of parallelised hard real-time applications supporting analysability. In: Proceedings of Euromicro symposium on digital system design (DSD), Los Alamitos
55. Valencia J, van Horsen E, Goswami D, Heemels M, Goossens K (2016) Resource utilization and quality-of-control trade-off for a composable platform. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), Lausanne
56. Windsor J et al (2009) Time and space partitioning in spacecraft avionics. In: SMC-IT, Pasadena
57. Zhang H (1995) Service disciplines for guaranteed performance service in packet-switching networks. Proc IEEE 83(10):1374–1396