

Chapter 2

Composability and Predictability for Independent Application Development, Verification, and Execution

Benny Akesson, Anca Molnos, Andreas Hansson,
Jude Ambrose Angelo, and Kees Goossens

Abstract System-on-chip (soc) design gets increasingly complex, as a growing number of applications are integrated in modern systems. Some of these applications have real-time requirements, such as a minimum throughput or a maximum latency. To reduce cost, system resources are shared between applications, making their timing behavior inter-dependent. Real-time requirements must hence be verified for *all possible combinations* of concurrently executing applications, which is not feasible with commonly used simulation-based techniques. This chapter addresses this problem using two complexity-reducing concepts: *composability* and *predictability*. Applications in a composable system are completely isolated and cannot affect each other's behaviors, enabling them to be independently verified. Predictable systems, on the other hand, provide lower bounds on performance, allowing applications to be verified using formal performance analysis. Five techniques to achieve composability and/or predictability in soc resources are presented and we explain their implementation for processors, interconnect, and memories in our platform.

Keywords Composability · Predictability · Real-Time · Arbitration · Resource Management · Multi-Processor System

2.1 Introduction

The complexity of contemporary Systems-on-Chip (soc) is increasing, as a growing number of independent applications are integrated and executed on a single chip. These applications consist of communicating tasks mapped on heterogeneous multi-processor platforms with distributed memory hierarchies that strike a good

B. Akesson (✉)

Eindhoven University of Technology, Postbus 513, 5600 MB Eindhoven, The Netherlands
e-mail: k.b.akesson@tue.nl

balance between performance, cost, power consumption and flexibility [14, 22, 38]. The platforms exploit an increasing amount of application-level parallelism by enabling concurrent execution of more and more applications. This results in a large number of *use-cases*, which are different combinations of concurrently running applications [15]. Some applications have *real-time requirements*, such as a minimum throughput of video frames per second, or a maximum latency for processing those video frames. Applications with real-time requirements are referred to as *real-time* applications, while the rest are *non-real-time* applications. A use-case can contain an arbitrary mix of real-time and non-real-time applications.

To reduce cost, platform resources, such as processors, hardware accelerators, interconnect, and memories, are shared between applications. However, resource sharing causes *interference* between applications, making their temporal behaviors inter-dependent. Verification of real-time requirements is often performed by system-level simulation. This results in three problems with respect to verification, since inter-dependent timing behavior requires that all applications in a use-case are verified together. The first problem is that the number of use-cases *increases rapidly* with the number of applications. It hence becomes infeasible to verify the exploding number of use-cases by simulation. This forces industry to reduce coverage and verify only a subset of use-cases that have the toughest requirements [14, 37]. The second problem is that verification of a use-case cannot begin until all applications it comprises are available. Timely completion of the verification process hence depends on the availability of all applications, which may be developed by different teams inside the company, or by independent software vendors. The last problem is that use-case verification becomes a *circular process* that must be repeated if an application is added, removed, or modified [23]. Together these three problems contribute to making the integration and verification process a dominant part of soc development, both in terms of time and money [22, 23, 34].

In this chapter, we address the real-time verification problem using two complexity-reducing concepts: *composability* and *predictability*. Applications in a composable system are completely isolated and cannot affect each other's functional or temporal behaviors. Composable systems address the verification problem in the following four ways [17]: 1) Applications can be verified in isolation, resulting in a linear and non-circular verification process. 2) Simulating only a single application and its required resources reduces simulation time compared to complete system simulations. 3) The verification process can be incremental and start as soon as the first application is available. 4) Intellectual property (IP) protection is improved, since the verification process no longer requires the IP of independent software vendors to be shared. These benefits reduce the complexity of simulation-based verification, making it a feasible option with a larger number of applications. An additional benefit is that *composability does not inherently make any assumptions on the applications*, making it applicable to existing applications without any modifications.

Predictable systems, on the other hand, bound the interference from the platform and between applications. This enables bounds on performance, such as upper bounds on latency or lower bounds on throughput, to be provided. Applications

in predictable systems can hence be verified using formal performance analysis frameworks, such as network calculus [9] or data-flow analysis [36]. The benefit of formal performance verification is that conservative performance guarantees can be provided for all possible combinations of initial states of resources and arbiters, all input stimuli, and all concurrently executing applications. The drawback is that formal approaches require performance models of the software, the hardware, and the mapping [8, 25], which are not always available. Composability and predictability both solve important parts of the verification problem and provide a complete solution when combined.

The two main contributions of this chapter are: 1) An overview of five techniques to achieve composability and/or predictability in multi-processor systems with shared resources. 2) We show how to design a composable and predictable system by applying the proposed techniques to three typical resource types: processor tiles, interconnect (a network on chip), and memory tiles (with either on-chip SRAM or off-chip SDRAM).

The rest of this chapter is organized as follows. Section 2.2 describes a number of techniques to achieve composability and/or predictability for shared resources. We then proceed in Sections 2.3, 2.4, and 2.5 by explaining which of these techniques are suitable for our processor tiles, network-on-chip, and memory tiles, respectively. Section 2.6 then demonstrates the composability of our soc platform by showing that the behavior of an application is unaffected at the cycle-level, as other applications are added or removed. Lastly, we end the chapter with conclusions in Section 2.7.

2.2 Composability and Predictability

The introduction motivates how composability and predictability address the increasingly difficult problem of verifying real-time requirements in socs. The next step is to provide more details on how to implement these concepts. Firstly, we establish some essential terminology related to resource sharing, which allows us to define composability and predictability formally. We then discuss five techniques to achieve these properties and highlight their respective strengths and weaknesses. This illustrates the design space for composable and predictable systems, and allows us to explain how different techniques are suitable for different resources depending on their properties, such as whether execution times are constant or variable, and whether the resource is abundant or scarce.

2.2.1 Terminology

Our context is a tiled platform architecture following the template shown in Fig. 2.1. At the high level, this platform comprises a number of processor tiles

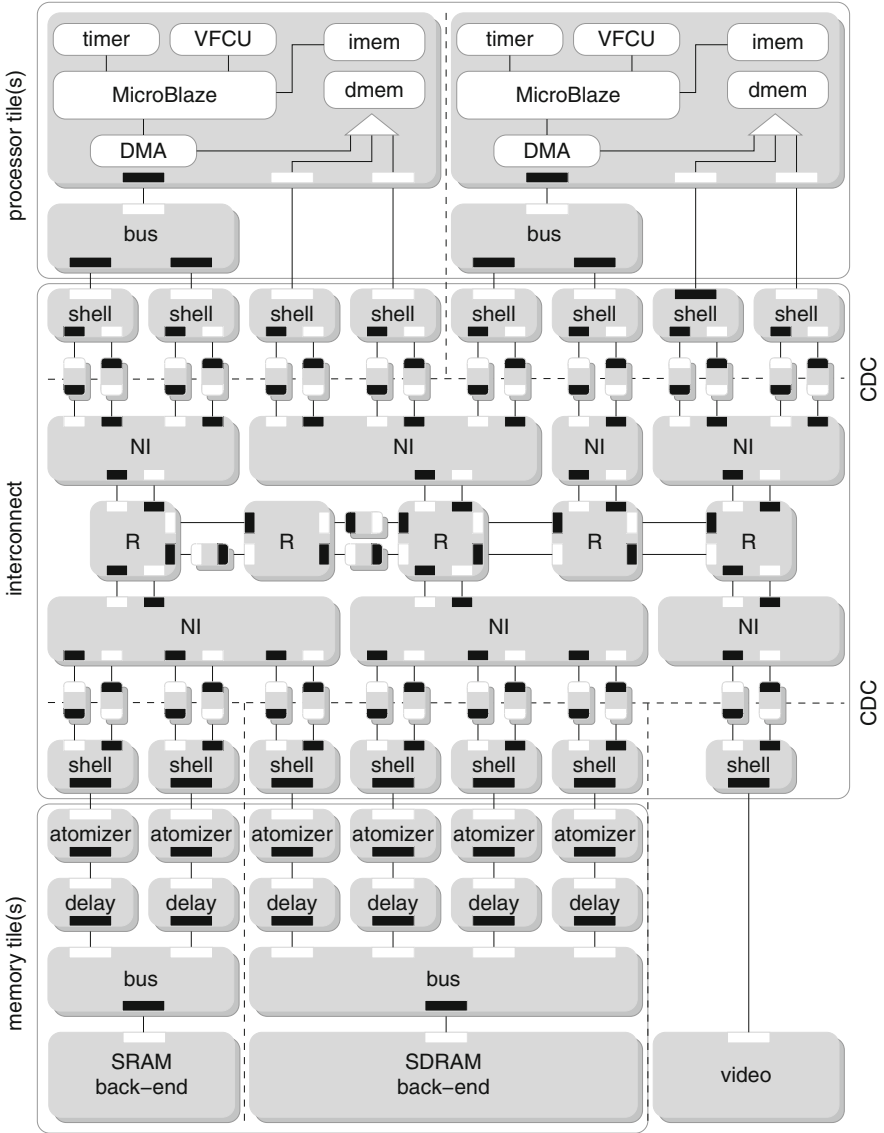


Fig. 2.1 The architecture of the considered MPSoC platform

and memory tiles interconnected by a network-on-chip. We return to discuss the details of this architecture in Sections 2.3, 2.4, and 2.5, respectively. An *application* consists of a set of *tasks* that may be split across several processor tiles to enable parallel processing. We assume a static task-to-processor mapping, which implies that task migration is not supported. Non-real-time tasks can communicate in any way they like using distributed shared memory, obeying only the restrictions

on processors, discussed later in Section 2.3.1. However, tasks of real-time applications operate in a more restrictive fashion to ensure that their temporal behavior can be bounded. Each real-time task continuously *iterates*, which means that it reads its inputs, executes its function, and writes its outputs. Inter-task communication is implemented using FIFOs, according to the C-HEAP protocol [31], with blocking read and write operations. Inside a FIFO token, data can be accessed in any order. We choose this programming model because it perfectly fits the domain of streaming applications and enables overlapping computation with communication. It furthermore allows modeling an application as a data-flow graph, which enables efficient timing analysis. Communication between processor tiles and memory tiles takes place via the interconnect.

Requests are defined as uses of a *resource*, such as a processor, interconnect, or a memory. The originators of requests, and hence the users of the resources, are referred to as *requestors*. Requests for a processor resource correspond to application tasks that are ready for execution. In case of a memory or an interconnect, requests are transactions originating from ports on IP components. These transactions are communicated using standardized protocols, such as AXI [6], DTL [33], or OCP [32]. Common examples of transactions are reads and writes of either single data words or bursts of data to a memory location.

The *execution time* (ET) of a request determines the amount of time a request uses a resource before finishing. However, a requestor may not have exclusive access to the resource, due to interference from other requestors. Interference may prevent a request from accessing the resource straight-away and its execution may be pre-empted several times before finishing. This is considered in the *response time* (RT) of a request, which accounts for both the execution time and the interference. The response time is hence the total time it takes from when the request is eligible for scheduling at the resource until it has been served. The point in time at which a request is scheduled to use the resource for the first time is referred to as its *starting time*. It is important to note that the execution time, response time, and starting time of a request from a requestor often depend on other requestors. The execution time may depend on others if a request from one requestor alters the state of a resource in a way that affects the execution time of a following request. A common example of this is when a memory request from a requestor evicts a cache line from another requestor, turning a future cache hit into a cache miss. The response time and starting time both typically vary with the presence or absence of requests from other requestors in systems with run-time arbitration, such as round robin or static-priority scheduling. This results in a varying interference that causes both the starting time and response time to change. We now proceed by defining composability and predictability in terms of the established terminology.

The functional behavior of a request is defined as composable when its output is independent of the behavior of requestors belonging to other applications. The temporal behavior of a request from a requestor using a resource is defined as composable if its starting time and response time are independent of requestors from other applications, since this implies that the request starts and finishes using the resource independently of others. We refer to a resource as a *composable*

resource if both functional and temporal composability holds for any set of requestors and their associated requests. A composable system contains only composable resources. Such a system enables independent verification of applications, as their constituent requestors and requests are completely isolated from each other in the time and value (functional) domains. The verification complexity hence becomes linear with respect to the number of applications. It also makes the resulting system more robust at run time, because there is no interference from unknown, failing, or misbehaving applications. In this chapter, we focus on verification of real-time requirements. We hence limit the discussion to temporal composability and do not further discuss how to achieve functional composability. For simplicity, we let composability refer to temporal composability in the rest of this chapter.

For predictability, every request on a resource must have both a *useful* worst-case execution time (WCET) and worst-case response time (WCRT). Unlike composability, which inherently considers multiple requestors and applications on a shared resource, predictability can be considered for a non-shared resource with only a single requestor. For shared resources, the WCRT can be determined if there is a bound on the interference from other requestors. A resource is a *predictable resource* if all requests from all the requestors mapped on it are predictable. Similarly, a predictable system is a system only comprising predictable resources. Predictable systems enable formal verification of real-time requirements, since applications are sets of requestors for different resources that all provide bounded WCRT. For a complete end-to-end analysis, these WCRTs have to be used in a performance analysis framework. We use data-flow [36] analysis to compute bounds on throughput and latency for real-time applications, although time-triggered [23] or network calculus [9] methods can also be used.

It is important to realize that predictability and composability are two different properties and that one does not imply the other. Predictability means that a useful bound is known on temporal behavior and is hence a property of a *single application* mapped on a set of resources. Composability, on the other hand, implies complete functional and temporal isolation between applications and is a property of *multiple applications* sharing resources, where each application may be predictable or not. We illustrate the difference by discussing four example systems, shown in Fig. 2.2, that cover all combinations of composability and predictability. The first system, depicted in Fig. 2.2a, consists of two processors (P), each executing a single application (A1 and A2, respectively). We assume that both applications are predictable and hence that worst-case execution times are known for all tasks when running on predictable hardware. Data is stored in a shared remote zero-bus-turnaround SRAM that is reached via a bus. This type of SRAM has an execution time of one clock cycle per read or written word that is independent of other requestors. The SRAM is shared using time-division multiplexing (TDM) arbitration, which is a composable and predictable arbitration scheme, since the WCRT of a requestor is both bounded and independent of other requestors. This makes this system as a whole both composable and predictable. For our second system in Fig. 2.2b, we replace the TDM arbiter with a round robin arbiter (RR). This system is not composable, since response times of requests vary depending on the presence or

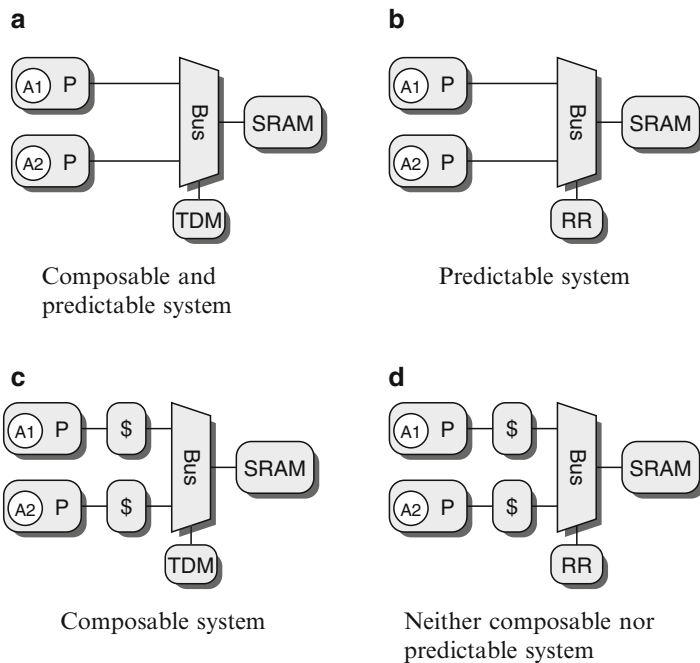


Fig. 2.2 Four systems demonstrating all combinations of the composability and predictability properties.

absence of requests from other requestors. However, it is still predictable, since this interference is easily bounded. We create our last two systems by adding private L1 caches (\$) with random replacement policies to the processors in both previous systems. A private cache is composable, since it is not shared between applications. However, the random replacement policy makes the systems unpredictable, since a useful bound cannot be derived on the time to serve a sequence of requests. The third system, in Fig 2.2c, is hence composable, but not predictable. The last system, shown in Fig 2.2d, is neither composable, nor predictable.

2.2.2 Composable Resources

This section discusses designing composable resources that may or may not be predictable. As previously explained in Section 2.2.1, composability implies that the starting time and response time of a request from a requestor must be completely independent of requests from requestors belonging to other applications. Composability is trivially achieved by mapping applications to different resources, an approach used by federated architectures in the automotive and aerospace industries [24]. However, this method is prohibitively expensive for

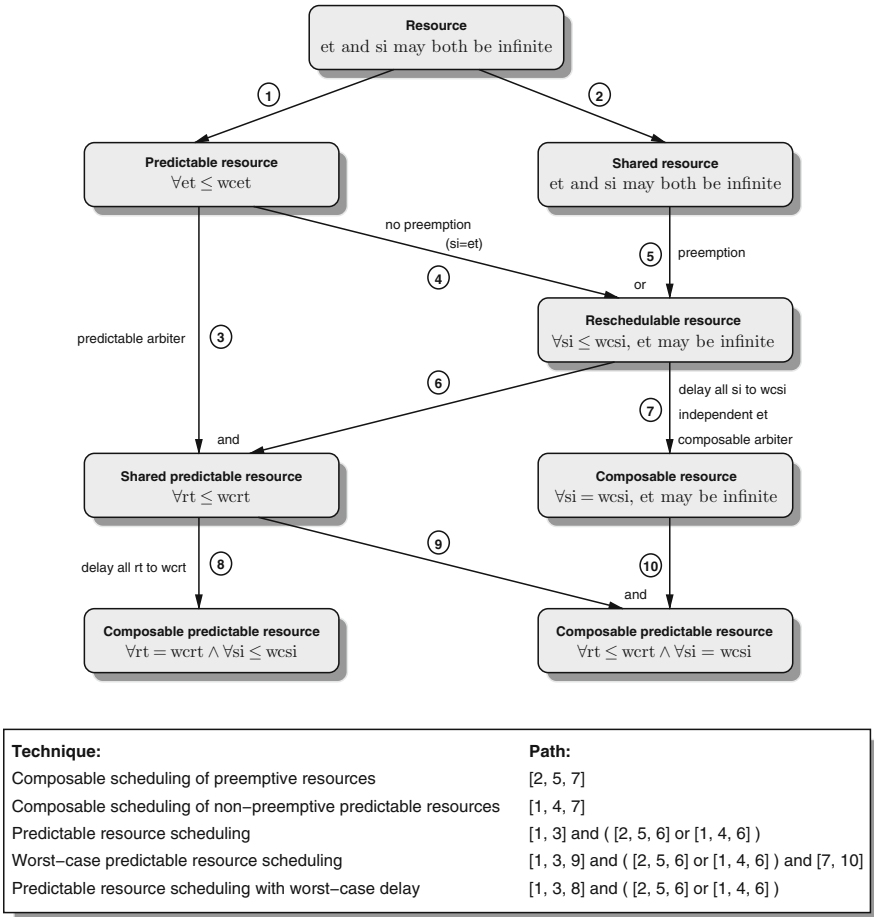


Fig. 2.3 Overview of techniques to achieve composability and predictability

systems that are not safety-critical. We proceed by looking at two alternatives to composable sharing of resources. These correspond to the two paths ② → ⑤ → ⑦ and ① → ④ → ⑦ in Fig. 2.3, which provides an overview of the five techniques presented in this chapter.

The first technique is called *composable scheduling of preemptive resources* and corresponds to following the edges ②, ⑤, and ⑦. This approach considers that the execution times of requests may be variable and unknown a priori. An example of this is the time required by a video decoding task executing on a processor to decode a frame, which is highly dependent on the image contents. This results in non-composable behavior, as the starting time of a request becomes dependent on the execution time of the previous request, which may have been issued by a requestor belonging to a different application. A solution to this problem is to *preempt* an executing request after a given time, referred to as the *scheduling interval* (si) of the resource arbiter. This is shown in Fig. 2.4a, where the request of requestor 2 is

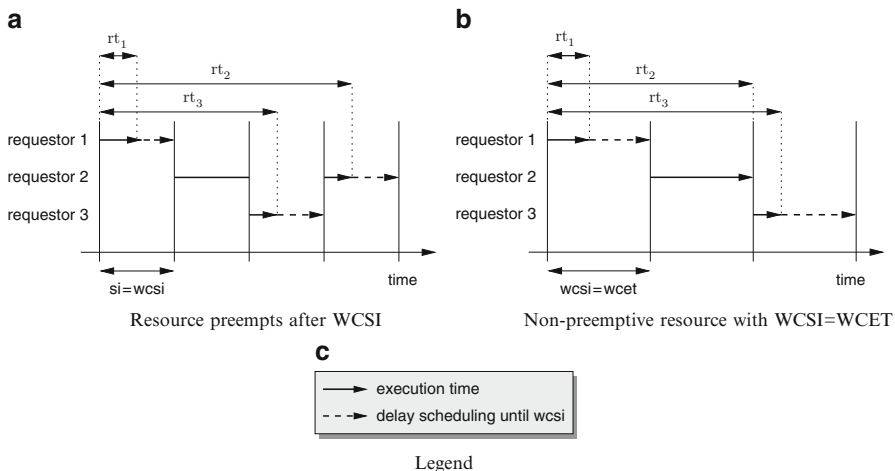


Fig. 2.4 Composable scheduling for preemptive and non-preemptive resources, respectively

preempted before finishing its execution. We refer to a resource with a worst-case scheduling interval (wcsi) as a *reschedulable resource*, as shown in Fig. 2.3, since it is guaranteed to take new scheduling decisions within a bounded time. Such a resource *ensures progress* of all requestors if it is paired with a *starvation-free* arbiter, which is a class of arbiters that guarantee that all requestors are scheduled in a finite time. Both round robin and TDM are examples of arbiters in this class. A static-priority scheduler, on the other hand, is not free of starvation, since a low-priority requestor starves if high-priority requestors are constantly requesting.

The next step with this technique is to make all scheduling intervals equal to the wcsi by delaying the arbiter in case the request finishes early, as shown in Fig. 2.4a. This step decouples the starting time of a request from the execution time of the preceding request, which is one of the two requirements to achieve composability. The second requirement is that the response time must be independent from requestors of other applications. We achieve this by using a composable arbiter, such as TDM, where the presence or absence of other requestors does not affect the interference. This results in independent response times for resources where the execution time is independent of previous requests, such as a zero-bus-turnaround SRAM. We have now fulfilled both requirements for a resource to be considered composable. Note that this type of composable resource is not necessarily predictable. It may, for example, include a cache that is private or shared between requestors belonging to the same application, which results in non-useful bounds on execution time for memory requests, although they are independent of other applications.

Next, we explore a second method of designing composable resources called *composable scheduling of non-preemptive predictable resources*, which follows the edges ①, ④, and ⑦ in Fig. 2.3. This method is motivated by the main limitation of the first approach, which is restricted to preemptive resources. Some important resources, such as SDRAM memories cannot be preempted during a burst, as they require all the data associated with a request to be transferred on consecutive clock cycles to function

correctly. Achieving composability with non-preemptive resources is still possible, assuming that the resource is predictable and hence has a known $wcET$. For these resources, we make the scheduling interval equal to the longest $wcET$ of any request executing on the resource. This is illustrated in Fig. 2.4b, where the request from requestor 2 is assumed to have the longest $wcET$. This technique makes starting times independent of requests from other applications, which is required for composability. Supporting non-preemptive resources with bounded execution times is the major benefit of this technique. However, this method arrives at a reschedulable resource by characterizing the requests and the resource rather than by enforcement, which has three drawbacks. Firstly, it cannot be applied to mixed time-criticality systems where real-time applications share resources with non-real-time applications that do not have bounded $wcET$. Secondly, the system is less robust, as it becomes non-composable if the characterization is incorrect or if a requestor misbehaves. Finally, making the scheduling interval equal to the longest $wcET$ results in low resource utilization if there is a large difference between the average and worst-case execution time. This is not acceptable for scarce resources, such as SDRAM memories.

Since composable scheduling of non-preemptive predictable resources implies that the $wcET$ of requests have to be bounded, it may result in a system that is also predictable. This depends on whether or not the composable arbiter is also predictable. Although this is typically the case, such as for TDM, it is not inherent to composability. For example, an arbiter that randomly schedules requestors every $wcSI$ is composable, as it is independent of applications, but it is unpredictable, since the $wcRT$ can be infinite. We will return to discuss techniques to share resources in ways that are both composable and predictable in Section 2.2.4.

The proposed techniques for composable resource sharing make the temporal behaviors of the requestors independent of each other, thus implementing composability at the level of requestors. This is a sufficient condition to be composable at the level of applications, which is the actual requirement from Section 2.2.1. However, composability at the level of requestors is stricter in the sense that requestors belonging to the same application are allowed to interfere with each other in a composable system. It is hence possible to let requestors benefit from unused resource capacity (slack) reserved by requestors belonging to the same application to increase performance or reduce power [27]. This can be accomplished by using a two-level arbiter, as proposed in [17], where the first level is a composable inter-application arbiter, and the second an intra-application arbiter that does not have to be composable. This type of arbitration enables requestors from the same application to use slack created in the intra-application arbiter to boost performance without violating composability at the application level.

2.2.3 Predictable resources

Having discussed two ways of building resources that are composable, but not necessarily predictable, we proceed by discussing how to build resources that are

predictable, but not necessarily composable. As previously mentioned in Section 2.2.1, this requires useful bounds on both the $wcET$ and the $wcRT$.

Our approach to predictable resource sharing is based on combining resources and arbiters, each with predictable behaviors. In Fig. 2.3, this intuitively corresponds to following the edges ① and ③ from a general resource to a predictable shared resource. More specifically, we require bounds on the $wcET$ for each request executing on the resource, since these characterize the worst-case behavior of the unshared resource. Some resources, such as zero-bus-turnaround SRAMS, are predictable and have constant execution times that are easy to determine. However, other resources, such as SDRAM, have variable execution times that depend on earlier requests and cannot be usefully bounded at design time in the general case [1]. In this case, the resource controller must be implemented in a way that makes the resource behave in a predictable manner. We discuss how to accomplish this for an SDRAM resource in Section 2.5.

If the resource is shared, we require *predictable arbitration* that bounds the time within which a request finishes receiving service. Note that by this definition, *all predictable arbiters are starvation free*. Predictable arbiters enable the $wcRT$ to be computed if the resource is reschedulable and hence makes new scheduling decisions within a bounded time, determined either by a chosen scheduling interval (preemptive resource) or by the longest $wcET$ of any request executing on the resource (non-preemptive resource). This is illustrated in Fig. 2.3, where a predictable shared resource has to be both predictable and reschedulable and there are two possible paths to achieve the latter. Computing the $wcRT$ takes the effects of sharing the resource into account.

An important property of our approach is that it is based on combining *independent analyses* of the resource and the arbitration. The arbiter analysis bounds the number of scheduling decisions that are made by the arbiter from a request is eligible for scheduling until it finishes receiving service. The $wcRT$ is then conservatively computed by multiplying the number of decisions with the $wcSI$ and adding the number of pipeline stages between the request buffer and the response buffer in the architecture. Note that this conservatively accounts for both the execution time of the request and any preemptions from other requestors during the execution. The strength of this approach is the generality, as *any combination of predictable resource and predictable arbiter* results in a predictable shared resource. This makes it easy to change the arbiter to fit with the response time requirements of the requestors in the system, which is exploited by the processor tile in Section 2.3 and the memory tile presented in Section 2.5.

2.2.4 Composable and predictable resources

Section 2.2.1 explained that composability and predictability are different properties and that one does not imply the other. We then showed in Sections 2.2.2 and 2.2.3 how to make resources that are either composable or predictable. In this

section, we discuss two ways of making resources that are both composable and predictable.

The first and most straight-forward technique to get composable and predictable resources is to simply combine the approaches in Sections 2.2.2 and 2.2.3. We call this technique *worst-case predictable resource scheduling* and it corresponds to moving from a predictable shared resource via edge ⑨ and from a composable resource via edge ⑩ to a composable and predictable resource. This implies that the resource is predictable and that each request has a useful bound on $wcET$ that is independent of other requestors. It also means that the resource is shared using an arbiter that is both composable and predictable, such as TDM. Such an arbiter provides bounded interference from other requestors that is independent of their actual behaviors, making the resource composable and bounding the $wcRT$. Since the original approaches to composable and predictable resources apply to both preemptive and non-preemptive resources, the same property holds for this combination. It furthermore inherits the possibilities for slack management, previously explained in Section 2.2.2.

A benefit of this approach to make resources composable and predictable is that it is easy to conceptually understand and implement. A drawback is that it only applies to resources where the execution time of a request is independent of requests from requestors belonging to applications other, as previously described in Section 2.2.2. If this is not naturally the case, it can be achieved by delaying all executions to be equal to the $wcET$. However, this may be costly if the variation in execution time due to other applications is large, preventing it from being efficiently applied to scarce resources, such as SDRAM. Instead, this technique is used in the processor tile presented in Section 2.3 and for composable and predictable SRAM sharing using TDM in [17].

The second technique is called *predictable resource scheduling with worst-case delay* and addresses the problem of efficiently dealing with variable execution times and extends composability to support any predictable arbiter. The problem with most predictable arbiters is that they typically cause the times at which the resource accepts requests and sends responses to a requestor to change due to variable interference from other requestors, making it non-composable. The key idea behind this technique is to make the system composable by removing the variation in interference, both from other applications and the resource itself. We accomplish this by starting from a predictable shared resource and then delay all signals sent to a requestor to *emulate maximum interference from other requestors*. A requestor hence always receives the same worst-case service no matter what other requestors are doing. This technique corresponds to achieving composability for a predictable shared resource using edge ⑧ in Fig. 2.3. The implication of this approach is that the interface presented towards the requestor is temporally independent of other requestors. Variation in starting times and response times may be visible on the resource side of the interface, but not on the requestor side. This is similar to the composable component interfaces proposed in [23].

The technique implies delaying responses in a response buffer until their $wcRT$ to prevent the requestor from receiving it prematurely if there is little interference,

or if the variable execution time is short. However, making the WCRT independent of other applications is only one of the two requirements for a composable resource. The second requirement states that the starting time must also be independent. This is not the case if a request is scheduled earlier than its worst-case starting time. In this case, another request may be admitted into the resource prematurely, resulting in a different starting time. This problem is addressed by basing request accept signals on worst-case starting times of previous requests, as opposed to actual starting times. Requests are hence admitted into the resource in a composable manner, regardless of the interference experienced by others.

Figure 2.5 compares ‘predictable resource scheduling with worst-case delay’ to ‘composable scheduling of preemptive resources’, previously discussed in Section 2.2.2. Figure 2.5a illustrates that requests are scheduled immediately after a finished execution using ‘predictable resource scheduling with worst-case delay’, but that responses are delayed until the WCRT. In contrast, Fig. 2.5b (identical to Fig. 2.4a) shows that ‘composable scheduling of preemptive resources’ delays scheduling until the WCRT, but releases responses immediately after a finished execution.

‘Predictable resource scheduling with worst-case delay, has two major benefits compared to ‘composable scheduling of preemptive resources’: 1) It extends the use of composability beyond resources and arbiters that are inherently composable. It is hence not limited to resources where the execution times of requestors are independent, but can efficiently capture the behavior of any predictable resource. 2) It supports any predictable arbiter, enabling service differentiation that increases the possibility of satisfying a given set of requestor requirements [2]. For example, using an arbiter that is more sophisticated than TDM can lead to reduced over-allocation, and allow lower latencies or higher throughput on a resource. These

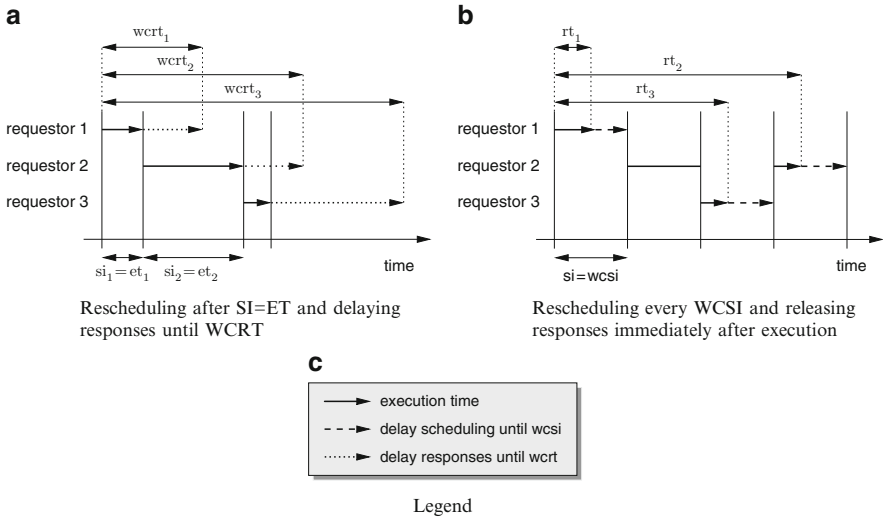


Fig. 2.5 Delaying scheduling until wcsi vs. delaying responses until WCRT

characteristics make the approach suitable for memory tiles with SDRAM, as we will further explain in Section 2.5.

The main drawback of this technique is related to slack management. This approach makes the temporal behaviors of the requestors independent of each other, thus implementing composability at the level of requestors instead of at the level of applications. It is hence not possible to benefit from unused resource capacity reserved by requestors belonging to the same application, which may negatively impact performance.

2.3 Processor tile

Having reviewed the different approaches to achieving composability and predictability, we proceed by looking at how it is actually implemented in a multi-processor system, starting with the processor tile. We consider a mixed time-criticality system, where the processor executes a mix between real-time and non-real-time applications. In this section, we first present the strategy to achieve composability of applications on a processor tile, followed by our approach to implementing predictability. The architecture of the processor tile is shown in Fig. 2.1. The components of this tile are discussed in the following sections.

2.3.1 *Composability*

Processors execute requests, corresponding to task iterations. The execution time of a request is hence the time it takes to execute a task iteration on the processor. Real-time tasks must have a WCET, which means that they complete an iteration in bounded time. This is not necessarily the case for non-real-time tasks. In mixed time-criticality systems, where these types of tasks share resources, the WCET of real-time tasks can only be bounded if resources are preemptive. Composability in the processor is hence implemented using the technique ‘composable scheduling of preemptive resources’. The key ingredients to achieve composability in this resource are thus found on the path ②, ⑤, and ⑦ in Fig. 2.3 and constitute: 1) preemption, 2) enforcing a constant scheduling interval equal to wcsi, and 3) using a composable arbitration scheme.

For a processor, the wcsi defines a task slot with bounded duration when a task can utilize the processor. After a task slot finishes, an operating system (os) decides which task to execute next during an os slot. To ensure independent starting times and response times of tasks, required for composability, not only the task slots, but also the os slot, must have a constant duration and fixed starting times.

The execution time of the os may depend on the number of applications and tasks it has to schedule. If the os slot is not forced to a constant duration at least equal to its

wcET, it is impossible to ensure that task starting times and response times are independent of the presence or absence of other applications in the system. Furthermore, common oses check if tasks are ready to execute, which depends on the availability of their input data and output space. For composability, the time at which this check is performed must be independent of other applications. ‘Composable scheduling of preemptive resources’ requires the execution times of tasks to be independent. The functional state of the processor tile at a task switch must hence be unable to affect the execution time of the scheduled task. This may imply that the processor instruction pipeline should be empty, and that potential caches should be cleared of all data to avoid cache pollution. In the following sections, we present the mechanisms to enforce constant-duration task and os slots. Following this, we describe the scheduling of applications and tasks, which relies on this property.

2.3.1.1 Constant task slots

To enforce a task slot with constant duration and fixed starting times, we use a timer that interrupts the processor after a programmable fixed duration. When receiving an interrupt, the first instruction of the interrupt service routine jumps to os code, giving control to the os. This can be implemented with a dedicated timer per tile that is accessed via a memory-mapped peripheral bus or an instruction-mapped port. By using a timer outside the processor, in an always-on clock domain, the processor can enter a low-power state during idle periods without stopping the timer [13].

To get a constant-duration task slot, the processor should be interruptible in (preferably short) bounded time. However, processors are typically not interruptible while instructions are still in the pipeline. The time to start the interrupt service routine, referred to as the interrupt latency, thus depends on the execution time of the currently executing instructions. The time it takes to finish executing an instruction depends exclusively on the processor, except for instructions that involve other resources. For example, a load from non-local memory also uses the interconnect and a remote memory. Depending on the predictability and sharing of those resources, such a load may take thousands of cycles to complete (e.g. when it has a low priority in the NOC and memory tile).

By restricting the number of outstanding remote-read transactions, the wcET of a task and its worst-case interrupt latency can be computed, but will be prohibitively high (thousands of cycles). We hence use an alternative approach by restricting the processor to only using local (instruction and data) memories and use Remote Direct Memory Access (RDMA) engines to communicate outside the processor tile. Remote accesses may stall the RDMA, while the processor only polls locally, resulting in a short interrupt latency. Note that even with only local reads, the execution time of the interrupt service time is bounded, but not constant. For example, division and multiplication instructions take more cycles than NOP or jump instructions.

The processor programs the RDMA to read or write data on remote memories residing inside another processor tile, or in a memory tile. Programming the RDMA is done using only local load and store instructions. An additional advantage of using RDMA is that they decouple computation and communication, enabling them to be overlapped in time. In this chapter, we assume that the local memories of processor tiles are large enough to store the following state for all tasks mapped on the tile: 1) instructions, 2) (private) data, and 3) all the buffers (for input and output tokens) needed for an iteration. RDMA is hence only used for inter-task communication between tasks mapped on different processors. This communication is implemented using uni-directional FIFO buffers with finite size. These FIFO buffers are located either in the local memory of the consumer (if the memory space in the processor tile is sufficiently large), or in a remote memory tile. The producer always posts the data in the buffer via a RDMA write. In Fig. 2.1, the data travels from the data memory in the producer tile, through the RDMA to the interconnect. The interconnect then delivers it to the local memory in the consumer tile. Alternatively, the producer RDMA places the data in a remote memory tile, from where it is copied by the consumer RDMA to the data memory in its tile. In all cases, the FIFO administration [31], consisting of read and write pointers, is located in the producer and consumer tiles.

To achieve composability, a RDMA has to be composable if shared between applications. Since RDMA is simple finite state machines, we do not share them between applications. Instead, each application has its own RDMA, but for maximum performance, each FIFO of each task can be given its own RDMA. For simplicity, Fig. 2.1 shows only one RDMA per tile. Note that the local memory should also be made composable using the techniques detailed in Section 2.5.

2.3.1.2 Constant OS slot

As previously explained, the OS slot should have a constant starting time and duration. Given a constant task slot duration, the only requirement to achieve a constant OS starting time is that the task-to-OS switching time should be constant. The task-to-OS switching time is equal to the interrupt latency of the timer, which depends on the instructions in-flight on the processor. We force the interrupt latency to be constant and equal to its WCET via a mechanism to delay actions (execution) until a fixed future moment in time, as described below.

Our approach to enforce a constant OS slot is to *inhibit execution* on the processor until its WCET is reached, thus making the OS execution composable. This corresponds to the technique ‘composable scheduling of non-preemptive resources’, which uses edges ①, ④, and ⑦ in Fig. 2.3. This can be implemented in several ways. Polling on a timer [10] is the simplest, but prevents clock-gating of the processor. If the processor has a halt instruction, the processor can be halted after the OS finishes its execution. The tile timer, programmed before the halt instruction, wakes up the processor at the WCET. When a halt instruction is not available, the

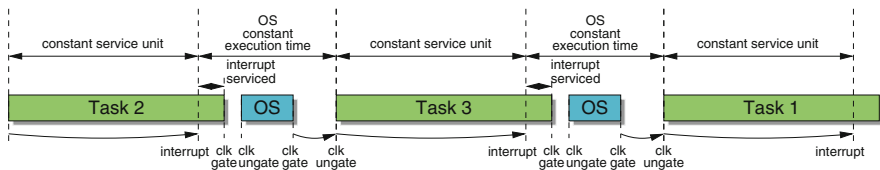


Fig. 2.6 Processor slots and task switching time line

processor clock can be disabled by a voltage-frequency control unit (VFCU in Fig. 2.1) until the $wcet$.

Figure 2.6 presents the time line with the seven main events when performing a task switch: 1) the interrupt is raised, 2) the interrupt is served, 3) the processor ungate moment in time is programmed, 4) the clock is gated up to the $wcet$ of the interrupt latency, 5) the os is executed, 6) the processor ungate moment in time is programmed, and finally 7) the clock is gated up to the $wcet$ of the os .

2.3.1.3 Two-level application and task scheduling

The constant-duration task and os slots ensure that task slots start at fixed points in time, and that there is a bounded $wcsi$. A task iteration that has a $wcet$ on a non-shared processor tile hence has bounded $wcet$ and $wcrt$ on a shared tile. As mentioned before, the functional state of the processor tile at the start of a task slot must be independent of other applications to avoid possible interference.

By using a composable scheduler, interference between all tasks is removed. However, this is unnecessarily strict, since it also prevents slack from being used by tasks belonging to the same application. Moreover, different applications benefit from using different schedulers, such as static-order, TDM, or Credit-Controlled Static-Priority arbitration [5] (ccsp, further described in Section 2.5). The processor addresses this problem by using a two-level arbitration scheme: a composable inter-application arbiter (TDM) that schedules applications, and an intra-application arbiter that schedules tasks within an application. The composable inter-application arbiter ensures the isolation between applications, while the intra-application arbiters are chosen to fit the requirements of the application tasks. The intra-application arbiters are free to distribute slack to improve performance of the tasks.

2.3.2 Predictability

As already mentioned, we target mixed time-criticality systems that concurrently execute a set of real-time and non-real-time applications. For real-time applications, we require the $wcet$ of each task iteration to be known. The execution time of a task on a processor is hence required to be predictable, which excludes the

use of out-of-order execution, speculation, and caches with random replacement policies [40].

To derive the end-to-end application performance (e.g. throughput, latency, etc.), applications are modeled as data-flow graphs [25, 36]. The nodes in the data-flow graphs are referred to as *actors* that are connected via directional *edges*. Each actor *fires* whenever its firing rule is satisfied. A firing rule specifies for each incoming and outgoing edge, the number of input tokens required and the number of output tokens produced, respectively. The data-flow model naturally describes a streaming application: a task is an actor, and a task iteration is an actor firing. FIFO communication between two tasks is represented as a pair of opposing edges, one modeling the communicated data, and the other modeling the available inter-task buffer space.

If several tasks share the same processor, predictable inter-task arbitration is required. Examples of such arbitration are TDM, CCSP, and round robin. Moreover, the sharing and arbitration effects should be taken into account when calculating the end-to-end application performance. Modeling of different arbitration policies as data-flow graphs is presented in [19, 28].

2.4 Interconnect

The processor and memory tiles in the system communicate via a global on-chip interconnect, as shown in Fig. 2.1. Typically, processors act as memory-mapped *initiators* and memory tiles as memory-mapped *targets*. This is seen in the figure, where initiator and target ports are colored black and white, respectively. When tasks execute on a processor, they give rise to read and write requests that are delivered to the appropriate memory tile based on the address, and a response is potentially delivered back to the processor. The *requestors* of the interconnect, according to Section 2.2.1, are thus the ports of the processor and memory tiles.

To deliver the aforementioned functionality, the interconnect is subdivided into a number of architectural components [16]. We first present a brief overview of the components and then continue to discuss how they provide composability and predictability. When a request is presented to the interconnect by an initiator, it is serialized by a protocol shell into a sequence of words. These words are then passed through a clock domain crossing (CDC) to transition from the clock domain of the initiator to that of the network, making the platform globally-asynchronous locally-synchronous (GALS) [30]. The data is then sent through the network, comprising Network Interfaces (NI) and routers (R), through a logical *connection*. The NI packetizes the data and determines the route through the network. The routers merely forward the data to its destination NI where it is depacketized, before transitioning to the clock frequency of the target in another clock domain crossing. The shell then deserializes the request and presents it to the actual target port. A response, if present, follows the same logical connection back through the

network until it reaches the initiator. The interconnect resource hence comprises protocol shells, clock domain crossings, NIS, routers and links.

2.4.1 Composability

The protocol shells are not shared by connections and thus require no special attention to deliver composability. They are furthermore simple state machines that can be considered predictable. Moreover, the shells serialize the memory-mapped transactions of the tiles independently of their protocol, burst size, type of transaction etc. Thus, when presented to the NIS as a stream of words, the level of flow control and preemption is a single word (using a FIFO protocol).

Once the serialized transactions are delivered to the NIS, each logical connection has dedicated input and output buffers in the NIS. At this level, the network can thus be seen as a set of composable distributed FIFOs, interconnecting pairs of protocol shells. The NIS packetize the individual words of data in units of *flits* and send them through the network links and routers. Each packet starts with a header (flit) with the path to the destination output buffer. In contrast to many on-chip networks, our interconnect does not perform any arbitration inside the network. The routers simply obey the path encoded in the packet headers, and push the responsibility of scheduling and buffering to the NIS. Thus, all arbitration takes place in the NI, and the routers merely forward the flits until they reach the destination NI, making the network appear as a single (pipelined) shared resource.

To make the network as a whole composable (and predictable), we use the technique ‘worst-case predictable resource scheduling’. We describe the implementation of this technique in three steps, corresponding to edges ⑤, ⑦, and ⑩ in Fig. 2.3. Firstly, the network resources are preemptive at the level of flits (edge ⑤). A scheduling decision is thus taken for every flit, independent of the length of the packets. Furthermore, as we have already seen, the data in the NI FIFOs has no notion of memory-mapped transactions, and there is consequently no correspondence between transactions and packets. As there is no buffering inside the router network, the NIS use *end-to-end flow control* to ensure the availability of buffer space. Consequently, flits are only injected if they are guaranteed not to stall anywhere inside the network.

Secondly, the flit size is fixed at three words, resulting in a constant scheduling interval of three cycles. If a connection’s input buffer is empty or if it runs out of flow control credits, it uses only one or two words of the three-word flit. The constant flit length corresponds to making all scheduling intervals equal to the wcsi, indicated by edge ⑦ in Fig. 2.3. It is worth noting that there is no need to determine how long it takes for other requestors flits to reach their destination, only how long it takes until a new flit can be scheduled, i.e. the execution time and response time of other requestors is irrelevant.

Thirdly, the fixed flit length is combined with a global schedule of the logical connections, where each NI regulates the injection of flits using a TDM arbiter [11], such that contention never occurs on the network links. The schedule relies on a

(logical) global synchronicity of the network components, but the concept has been demonstrated on both mesochronous and asynchronous implementations of the network [18]. The TDM schedule is programmed at run time according to the running use-case, but is typically determined at design time.

The last part of the interconnect composability is enforced insertion of packet headers for non-consecutive flits. That is, if another connection could have used the link, assume it did (even if it did not), and insert a new packet header. The header insertion ensures that the arbiter is *stateless* in terms of influence from other requestors.

2.4.2 Predictability

With the aforementioned mechanisms in place, the interconnect offers composability at the level of connections, between pairs of protocol shells. Predictability additionally requires worst-case response times for the shared resources. As discussed in detail in [19], the temporal behavior of a connection depends on the TDM scheduler settings, the path length, and the size of the input and output buffers. The scheduler determines how long words have to wait in the input buffer until injected into the network, once eligible. The path, in turn, determines the time required to traverse the network (without stalling). The input and output buffers affect the time at which words are accepted and become eligible for scheduling. All these contributions can be bounded and captured in a data-flow graph, thus offering predictability.

2.5 Memory tile

This section presents our memory tile and discusses the techniques employed to implement composability and predictability. The architecture of the memory tile, shown in Fig. 2.1, is divided into a *front-end* and a *back-end*. The front-end is independent of memory technology and contains buffering, arbitration, and components to make the memory tile composable. The back-end interfaces with the actual memory device and makes it behave like a predictable resource. The back-end is hence different for different types of memories, such as SRAM and SDRAM, as indicated by the figure. The components in the architecture are discussed further in the following sections.

Although our memory tile is general and supports both SRAM and DDR2/DDR3 SDRAM, we will focus the discussion on SDRAM, since these memories have three important characteristics that make the implementation of composability and predictability challenging. 1) The execution time of a request and the bandwidth offered by the memory is variable and depends on other requestors. 2) Some memory requestors are latency critical and require low response time to reduce the number of stall cycles on the processor. 3) For cost reasons, SDRAM bandwidth is a scarce resource that must

be efficiently utilized. This section is organized as follows. Firstly, Section 2.5.1 explains how to make an SDRAM behave like a predictable shared resource. Section 2.5.2 then discusses how to make the predictable shared memory composable.

2.5.1 Predictability

Section 2.2.1 states that a predictable resource must provide a useful bound on w_{CET} to all requests. In addition, a memory tile must bound the bandwidth offered to a requestor to ensure that bandwidth requirements are satisfied. This section elaborates on how our memory tile delivers on these requirements. The memory tile follows our general approach to predictable shared resources and combines a predictable resource with predictable arbitration. First, the concepts behind an SDRAM back-end that makes the memory behave like a predictable resource, corresponding to edge ① in Fig. 2.3, are explained. We then discuss how to share the predictable memory between multiple requestors, covering edge ③.

2.5.1.1 Predictable SDRAM back-end

SDRAM memories are challenging to use in systems with real-time requirements because of their internal architecture. An SDRAM memory comprises a number of banks, each containing a memory array with a matrix-like structure, consisting of rows and columns. A simple illustration of this architecture is shown in Fig. 2.7. Each bank has a row buffer that can hold one open row at a time, and read and write operations are only allowed to the open row. Before opening a new row in a bank, the contents of the currently open row are copied back into the memory array. The elements in the memory arrays are implemented with a single capacitor and a resistor, where a charged capacitor represents a logical one and an empty capacitor represents a logical zero. The capacitor loses its charge over time due to leakage and must be refreshed regularly to retain the stored data.

The SDRAM architecture makes the execution time of requests highly variable for three reasons. 1) A request targeting an open row can be served immediately, while

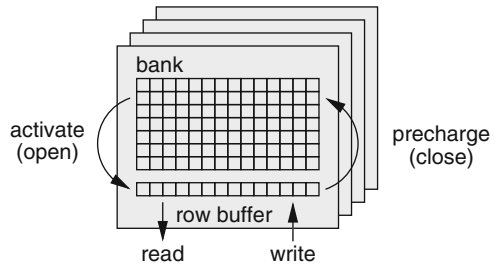


Fig. 2.7 The architecture of an SDRAM memory and behaviors of some important SDRAM commands

it otherwise needs the current row to be closed and the required row to be opened. 2) The data bus is bi-directional and requires a number of cycles to switch from read to write and vice versa. 3) The memory must occasionally be refreshed before executing the next request. The impact of these factors may cause the execution time of an SDRAM burst to vary by an order of magnitude from a few clock cycles to a few tens of cycles.

The behavior of an SDRAM memory is determined by the sequence of SDRAM commands that are communicated from the back-end of the memory tile to the memory device. These commands tell the memory to activate (open) a particular row in the memory array, to read from or write to an open row, or to precharge (close) an open row and store its contents back into the memory array. There is also a refresh command that charges the capacitors of the memory elements to ensure that the contents of the memory array are retained. The behaviors of some of these commands are illustrated in Figure 2.7. Scheduling SDRAM commands is not a trivial task, since there are a considerable number of timing constraints that must be satisfied before a command can be issued. These timing constraints are typically minimum delays between issuing particular SDRAM commands, such as two activates, or an activate and a read or a write.

Existing SDRAM controllers can be divided into two categories, depending on how they schedule SDRAM commands. Statically scheduled controllers [7] execute precomputed command schedules that are guaranteed at design time to satisfy all timing constraints of the memory. Executing precomputed schedules makes these controllers predictable and easy to analyze. However, they are also unable to adapt to the dynamic behavior of applications in contemporary socs, such as bandwidth requirements or read/write ratios that vary over time. The second category of controllers uses dynamic scheduling of commands, which requires the timing constraints to be enforced at run time. These controllers [20, 21, 26, 29, 35] have sophisticated command schedulers that attempt to maximize the average offered bandwidth and to reduce the average latency at the expense of making the resource extremely difficult to analyze. As a result, the offered bandwidth can only be estimated by simulation, making bandwidth allocation a difficult task that must be re-evaluated every time a requestor is added, removed or is modified.

We use a hybrid approach to SDRAM command scheduling that combines elements of statically and dynamically scheduled SDRAM controllers in an attempt to get the best of both worlds. Our approach is based on *predictable memory patterns* [1], which are precomputed sequences (sub-schedules) of SDRAM commands that are known to satisfy the timing constraints of the memory. These patterns are dynamically combined at run-time, depending on the incoming request streams. The memory patterns exist in five flavors: 1) read pattern, 2) write pattern, 3) read/write switching pattern, 4) write/read switching pattern, and 5) refresh pattern. The patterns are created such that multiple read or write patterns can be scheduled in sequence. However, a read pattern cannot be scheduled immediately after a write pattern. In this case, the read pattern must be preceded by a write/read switching pattern. This works analogously in the other direction. The refresh pattern can be scheduled immediately after either a read pattern or a write pattern. Both read and

write patterns can be scheduled immediately after a refresh without any preceding switching patterns.

The read and write patterns consist of a fixed number of SDRAM bursts, all targeting the same row in a bank. The bursts are issued to the different banks in sequence, since the data bus is shared between all banks to reduce the number of pins on the SDRAM interface. The fixed number of bursts is hence first sent to the first bank, then to the second, and so forth in an interleaving fashion until all banks have been accessed. This way of accessing the SDRAM results in a short period with frequent accesses, followed by a longer period without any accesses. The patterns exploit bank-level parallelism by issuing activate and precharge commands to the banks during the long intervals in which they do not transfer any data. The read and write patterns are hence very efficient in terms of bandwidth, since it is possible to hide a significant part of the latency incurred by activating and precharging rows. This limits the overhead cycles incurred by always precharging a bank immediately after it has been accessed, which is known as a closed page policy. We implement this policy, as it effectively removes the dependency on rows opened by earlier requests by returning the memory to a neutral state after every access. Removing this dependency between requests is a *key element* in our approach, since it *reduces the variation in the offered bandwidth and latency*, enabling tighter bounds on bandwidth and WCRT to be derived.

Although interleaving memory patterns allow us to bound the offered bandwidth, they come with two drawbacks. The first drawback is that continuously activating and precharging the banks increases power consumption compared to if a single bank is used at a time. The second drawback is that the memory is accessed with large granularity and hence requires large requests to be efficient. An efficient access requires at least one SDRAM burst to every bank. A typical burst size for SDRAM is eight words and the number of banks is either four or eight. The minimum efficient request size for a 32-bit memory interface is hence between 128-256 B, depending on the size and generation of the DDR SDRAM [3]. Working with large requests in a non-preemptive manner also means that urgent requests can be blocked longer, resulting in longer WCRT.

Requests are dynamically mapped to patterns in a non-preemptive manner by the command generator in the SDRAM back-end. A scheduled read request maps to a read pattern, possibly preceded by a write/read switching pattern. Similarly, a write request is mapped to a write pattern and potentially a preceding read/write switching pattern. Refresh patterns are scheduled automatically by the SDRAM back-end on a regular basis between requests. The mapping from requests to patterns and from patterns to SDRAM bursts is shown for an SDRAM with four banks in Fig. 2.8. The figure illustrates that the execution time of a request of four bursts varies depending on whether or not a switching pattern is required and if a refresh is scheduled before the request.

The benefit of memory patterns is that they raise SDRAM command scheduling to a higher level. Instead of dynamically issuing individual SDRAM commands, like a dynamically scheduled SDRAM controller, our back-end issues memory patterns that are sequences of commands. This implies a reduction of state and constraints that have

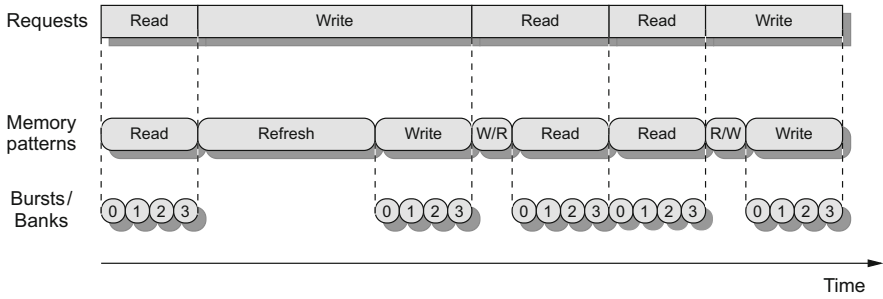


Fig. 2.8 Mapping from requests to patterns to SDRAM bursts

to be considered, making our approach easier to analyze than completely dynamic solutions. Memory patterns allow a lower bound on the offered bandwidth and WCRT to be determined, since we know the execution time of each pattern, how much data they transfer, and what the worst-case sequence of patterns is. This analysis is presented and experimentally evaluated in [3]. The use of memory patterns gives our approach the predictability of statically scheduled memory controllers. In addition, our approach has some properties of dynamically scheduled controllers, such as the ability to dynamically choose between read and write requests, and the use of run-time arbitration. The latter is discussed in the following section.

2.5.1.2 Predictable arbitration

After the previous section, we assume that we have a predictable memory, such as a zero-bus-turnaround SRAM or our SDRAM back-end based on predictable memory patterns, where useful bounds on both the offered bandwidth and the WCET of requests are known. In this section, we consider the effects of sharing the predictable memory between multiple requestors. As mentioned in Section 2.5.1, we require a predictable arbiter, where the number of interfering requests before a particular request is served is bounded. This enables the WCRT to be determined. There are a large number of predictable arbiters described in literature, such as TDM and round robin. However, most of these arbiters are unable to provide low response time to critical requestors, making them unsuitable for memory tiles. This problem is addressed by priority-based arbitration, but as previously mentioned in Section 2.2.2, conventional static-priority scheduling is not starvation-free and cannot be used to build predictable or composable systems. To address this issue, we have developed a Credit-Controlled Static-Priority (CCSP) arbiter [5]. The CCSP arbiter consists of a rate regulator and a static-priority scheduler. The rate regulator isolates requestors by enforcing an upper bound on the provided service, according to an allocated budget. It furthermore decouples allocation granularity and latency, which enables bandwidth to be allocated with an arbitrary precision without affecting latency [4]. A clean trade-off is hence provided between over allocation and area, allowing

over allocation to become negligible. This is essential for scarce SOC resources with very high loads, such as SDRAMs. The static-priority scheduler schedules the highest priority requestor that is within its budget. The use of priorities decouples latency and rate, thus enabling low latency to be provided to requestors with low bandwidth requirements without wasting bandwidth. The combination of rate regulator and static-priority scheduler makes the arbiter predictable, while still being able to satisfy the requirements of latency-critical requestors.

A rate regulator creates a separation of concerns and makes it possible to bound the WCRT of a requestor in a static-priority scheduler without relying on the cooperation of higher priority requestors. Instead, the bounds on WCRT are based on the *allocated bandwidths and burstinesses*, which are determined at design time. However, to be completely robust, we also need to be independent of the sizes of scheduled requests to prevent a malfunctioning requestor from preventing access from others by issuing very large requests. We solve this problem using preemptive service, which is enabled by the *atomizer* [17] block, shown in Fig. 2.1. The atomizer splits requests into smaller atomic service units, which are served by the memory in a known bounded time. This effectively makes the memory preemptive on the granularity of an atomic service unit. The size of the atomic requests are fixed and determined at design time. It is chosen to be the minimum request size that can be efficiently served by the resource. For an SRAM, the natural service unit is a single word, but it is much larger for an SDRAM with predictable memory patterns. For these memories, the appropriate size might be between 16 and 256 words, depending on the memory device and the desired trade-off between efficiency and latency.

2.5.2 Composability

Composability in the memory tile is achieved using the technique called ‘predictable resource scheduling with worst-case delay’. This is for two reasons related to the characteristics of SDRAM, presented earlier. Firstly, because SDRAMs have highly variable execution times that depend on other requestors. This prevents the use of ‘worst-case predictable resource scheduling’ unless the execution time is made independent of other requestors. This is possible by delaying all executions until the WCET by setting $WCSI=WCET$. For most patterns, this involves assuming a read/write switch for every memory request. Although possible to implement, this may increase the response time and decrease the offered bandwidth by up to 20% [3]. This is not a feasible option, considering that SDRAM bandwidth is a scarce and expensive resource. The second reason is that the first technique is limited to composable arbiters, such as TDM or static scheduling, which cannot distinguish requestors with low response time requirements. However, the second technique works with any predictable arbiter, such as our priority-based CCSP arbiter. The technique is implemented by the *delay block*, shown in Fig. 2.1. This component emulates worst-case interference from other requestors to provide a

composable interface towards the atomizer. This makes the interface of the entire front-end composable, since the atomizer is not shared.

It is worth noting that the delay block could have been placed in the processor tile, as opposed to in the memory tile. The advantage of this is that it offers composability to platforms with predictable, but not composable, interconnect by eliminating interference from both the interconnect and the memory tile at once. However, our interconnect is composable in itself using another technique, defeating the purpose of moving the delay block. Delaying in the processor tile furthermore comes with the drawback of making debugging of the platform more difficult, since the states of both the interconnect and memory tile change if applications are added, removed, or modified.

2.6 Experiments

The proposed composability inducing mechanisms are implemented for each resource of an SOC prototyped on FPGA having four processor tiles with one MicroBlaze core each, one memory tile and an \mathcal{A} ethereal NoC [12]. On this platform, we execute several use-cases constructed using the following applications: a simple synthetic application (A1), an H.264 video decoder [39] (A2), and a JPEG decoder (A3), each consisting of a set of communicating tasks. Figure 2.9 presents the task graphs and the task-to-processor mapping of these applications.

If the soc is composable, the behavior of an application should remain the same regardless of the presence or absence of other applications. We investigate composability in two ways: first by checking the cycle-level differences between some signals of the MicroBlaze interface in multiple simulations, and second by verifying whether the response time and starting time of an application remains constant when other applications are added in the system.

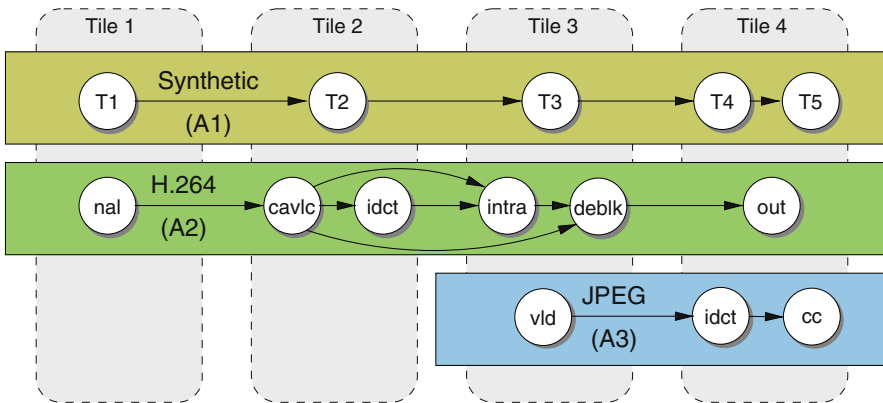


Fig. 2.9 The applications and mappings used in the experiments

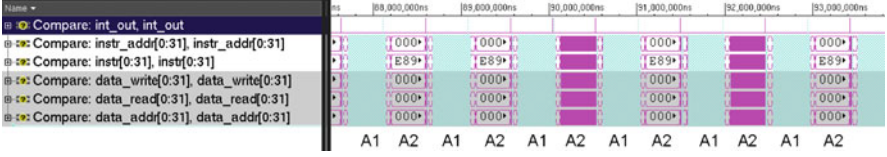


Fig. 2.10 MicroBlaze signal differences when *A1* varies its behavior

To investigate composability at the cycle level, we run two simulations and compare a number of signals in the first MicroBlaze core. For our simulations, we utilize the synthetic application, *A1*, and the H.264 application, *A2*. The *int_out* signal (the timer interrupt) indicates the border between the end of a task slot and the beginning of an os slot. This signal is kept high until the processor acknowledges that the interrupt is being served. In the first simulation, *A1* transfers data tokens of 4 KBytes and in the second it transfers data tokens of 16 Bytes. Figure. 2.10 presents the signal differences between the two simulations. The application TDM slot assignment is shown at the bottom of the diagram. We observe that signals in the task slots of *A2* are identical, whereas, the signals in *A1*'s slots change, as expected. The striped zone represents cycles that differ between the two runs. As seen in Fig. 2.10, the timer interrupt signals are not always identical in the two simulations. The reason for this is that different instructions are interrupted in different simulations, thus the *int_out* signal has different timing. The comparison between the two traces clearly shows that the only signal differences occur in the time slots of the changed applications and in the os slot, indicating that cycle-level composability is achieved.

To investigate the potential variations in the starting time and response time of applications, we run the H.264 and JPEG applications alone (*H.264-single* and *JPEG-single*, respectively), and in combination with the synthetic application (*H.264-multi* and *JPEG-multi*, respectively) on the FPGA. In these cases, we compare the response times and starting times of each iteration of each H.264 and JPEG task. If the system is composable, these times should be identical in different runs, regardless of the presence or absence of the synthetic application. Figures 2.11 and 2.12 present the response time differences for a JPEG and a H.264 task in two cases: 1) when all applications share a single RDMA engine (one RDMA per tile), and 2) when each application has its own RDMA engine (one RDMA per application).

As shown in the figures, the response times differ when using a single RDMA per tile, thus revealing interference. On the other hand, the response time difference is zero when using a single RDMA per application, showing no interference. Due to lack of space, we do not present the response times and starting times of all tasks. The observed behavior is the same, which means that the system is composable when using one RDMA per application. However, sharing a RDMA engine results in interference between applications, and variations in application timing behavior, just as expected.

In conclusion, we experimentally show that the processor behavior remains the same at both the cycle level and at the task-iteration level, indicating that our soc is temporally composable. The inspected signal traces in this section only cover the processor. However, the experiments strongly suggest that the interconnect and the

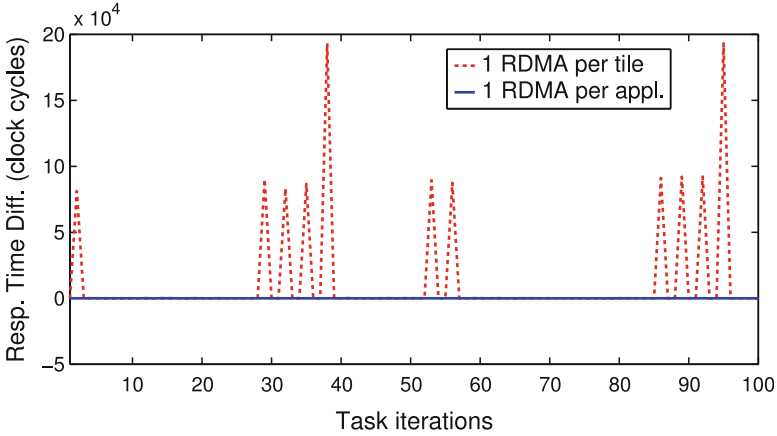


Fig. 2.11 JPEG, *vld* task response time difference between RDMA per tile or per application

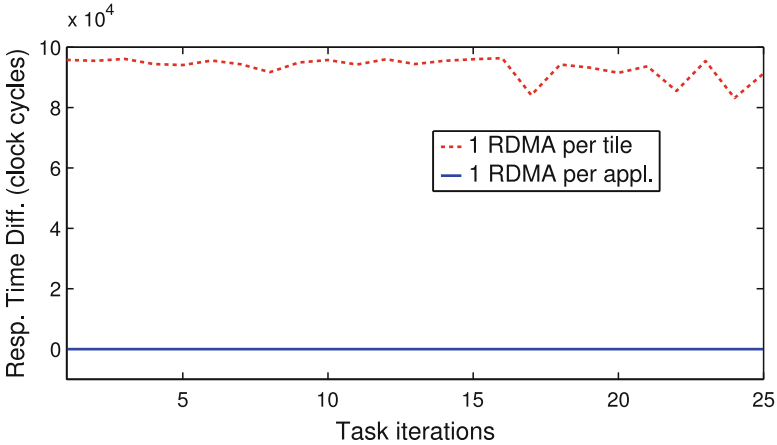


Fig. 2.12 H.264, *deblock* task response time difference between RDMA per tile or per application

memory tile are also composable. Otherwise, the timing variations in these resources would have resulted in variations in the response time of the tasks, or at the cycle-level timing of the processor signals.

2.7 Conclusions

This chapter addresses the verification and integration problem in embedded multi-processor platforms that have resources shared by a mix of real-time and non-real-time applications. We discuss two complexity-reducing concepts:

composability and *predictability*. Applications in a composable system are completely isolated and cannot affect each other's functional or temporal behaviors. Applications in a use-case can hence be verified individually instead of together, resulting in smaller state spaces. This enables a faster verification process, e.g. using simulation-based techniques, that can start as soon as the first application in a use-case is available. Predictable systems, on the other hand, provide lower bounds on application performance, such as latency and throughput. This enables applications to be verified at design time using formal performance analysis frameworks. The benefit of formal performance verification is that conservative performance guarantees can be provided for all possible combinations of initial states of resources and arbiters, all input stimuli, and all concurrently executing applications. However, formal approaches require performance models of the software, the hardware, and the mapping, which are not yet widely adopted by industry. Composability and predictability hence both solve important parts of the verification problem and provide a complete solution when combined.

Composability and predictability are different properties in the sense that predictability implies the existence of useful bounds on temporal behavior and is hence a property of a single application mapped on a set of resources. Composability implies complete isolation between applications and is a property of multiple applications sharing a resource, each of which may be predictable or not. We formally consider temporal composability achieved if the starting times and response times of an application, i.e. when it is scheduled for resource access and when it finishes receiving service, are independent of other applications.

The contributions of this chapter are twofold. Firstly, we present a thorough overview of five techniques for achieving composability and/or predictability and highlight their respective strengths and weaknesses. Secondly, we show how to build a composable and predictable system by applying the proposed techniques to three common resource types: processor tiles, interconnects (networks-on-chip), and memories (both on-chip SRAM and off-chip SDRAM).

On an unshared resource, predictability means that a request with finite size has a bounded worst-case execution time (WCET). On a shared resource, we achieve predictability by combining resources and arbiters, each with predictable behaviors. This enables the worst-case response time (WCRT) of requests to be determined for any combination of predictable arbiter and resource.

Composability can be achieved in four ways, described in the following paragraphs. The first way is useful if the execution times of all requests cannot be bounded. However, this requires that they can be preempted after a chosen worst-case scheduling interval (wcsi), which is the maximum time between two arbitration decisions. To create the premises of independent starting times, all scheduling intervals must have constant length equal to the wcsi. This decouples the starting time of a request from the execution times of previous ones. To enforce independent starting and response times, requests must be scheduled by a composable arbiter, such as time division multiplexing (TDM). The main limitation of this way to implement composability is that it only applies to preemptive resources in which

the execution time of a request is independent of requests from other requestors. This is the case for zero-bus-turnaround SRAM memories, but not for SDRAM.

The second way to implement composability applies particularly to non-preemptive resources. This technique requires that the resource is predictable and has a known WCET. The idea is to set the scheduling interval equal to the largest WCET of a request on the resource to make starting times independent of previous requests. Combining this with composable arbitration ensures that the worst-case response times are also independent. The two drawbacks of this technique are: 1) that execution times of requests have to be independent of requests from other requestors, just like for the previous method, and 2) making the scheduling interval equal to the longest WCET results in low resource utilization if there is a large difference between the average and worst-case execution time, which is the case for SDRAM memories.

The third and fourth ways to implement composability are based on predictability, resulting in resources with both properties. The third method is an extension of the first with an additional requirement that the composable arbiter is also predictable, such as TDM. This enables the WCRT to be computed for predictable applications with known WCET that is independent of other requestors.

The last way to implement composability (and predictability) applies to both preemptive and non-preemptive resources and supports variable execution times that depend on other requestors. It can furthermore be used with any combination of predictable resource and predictable arbiter. The key idea behind this approach is to make the system composable by enforcing maximum interference from other requestors to remove variation caused by other applications. This is accomplished by starting from a predictable shared resource and delay responses to emulate maximum interference from other requestors.

We experimentally demonstrate some of the proposed techniques on a tiled multi-processor system with MicroBlaze cores connected to an SRAM memory tile via a network-on-chip. Netlist simulations of this platform show that the cycle-level behavior of an application is unaffected, as the behavior of other applications changes, indicating composable execution.

References

1. B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256, 2007.
2. B. Akesson, A. Hansson, and K. Goossens. Composable resource sharing based on latency-rate servers. In *12th Euromicro Conference on Digital System Design (DSD)*, 2009.
3. B. Akesson, W. Hayes, and K. Goossens. Classification and Analysis of Predictable Memory Patterns. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2010.

4. B. Akesson, L. Steffens, and K. Goossens. Efficient Service Allocation in Hardware Using Credit-Controlled Static-Priority Arbitration. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2009.
5. B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.
6. ARM Limited. *AMBA AXI Protocol Specification*, 2003.
7. S. Bayliss and G. Constantinides. Methodology for designing statically scheduled application-specific sdram controllers using constrained local search. In *Field-Programmable Technology, 2009. International Conference on*, pages 304–307, Dec. 2009.
8. M. Bekooij, A. Moonen, and J. van Meerbergen. Predictable and Composable Multiprocessor System Design: A Constructive Approach. In *Bits&Chips Symposium on Embedded Systems and Software*, 2007.
9. R. Cruz. A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.
10. M. Ekerhult. Compose: Design and implementation of a composable and slack-aware operating system targeting a multi-processor system-on-chip in the signal processing domain. Master's thesis, Lund University, July 2008.
11. K. Goossens, J. Dielissen, and A. Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5):414–421, 2005.
12. K. Goossens and A. Hansson. The aethereal network on chip after ten years: goals, evolution, lessons, and future. In *DAC '10: Proceedings of the 47th Design Automation Conference*, pages 306–311, 2010.
13. K. Goossens, D. She, A. Milutinovic, and A. Molnos. Composable dynamic voltage and frequency scaling and power management for dataflow applications. In *13th Euromicro Conference on Digital System Design (DSD)*, Sept. 2010.
14. P. Gunning. The TI OMAP Platform Approach to SoC. *Winning the SoC revolution: experiences in real design*, page 97, 2003.
15. A. Hansson, M. Coenen, and K. Goossens. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 954–959, 2007.
16. A. Hansson and K. Goossens. An on-chip interconnect and protocol stack for multiple communication paradigms and programming models. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 99–108, 2009.
17. A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–24, 2009.
18. A. Hansson, M. Subbaraman, and K. Goossens. aelite: A flit-synchronous network on chip with composable and predictable services. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Apr. 2009.
19. A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *IET Computers & Digital Techniques*, 2009.
20. S. Heithecker and R. Ernst. Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 575–578, 2005.
21. E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture. ISCA '08. 35th International Symposium on*, pages 39–50, 2008.
22. International Technology Roadmap for Semiconductors (ITRS), 2009.

23. H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.
24. H. Kopetz, C. El Salloum, B. Huber, R. Obermaisser, and C. Paukovits. Composability in the time-triggered system-on-chip architecture. In *SOC Conference, IEEE International*, pages 87–90, 2008.
25. E. A. Lee. Absolutely positively on time: what would it take? *IEEE Transactions on Computers*, 38(7):85–87, 2005.
26. K. Lee, T. Lin, and C. Jen. An efficient quality-aware memory controller for multimedia platform SoC. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(5):620–633, 2005.
27. A. Molnos and K. Goossens. Conservative dynamic energy management for real-time data-flow applications mapped on multiple processors. In *12th Euromicro Conference on Digital System Design (DSD)*, 2009.
28. O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 57–66, 2007.
29. O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers. *IEEE Micro*, 29(1):22–32, 2009.
30. J. Mutersbach, T. Villiger, and W. Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proceedings of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 52–59, 2000.
31. A. Nieuwland, J. Kang, O. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset Llopis, and P. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3):233–270, 2002.
32. OCP International Partnership. *Open Core Protocol Specification*, 2001.
33. Philips Semiconductors. *Device Transaction Level (DTL) Protocol Specification. Version 2.2*, 2002.
34. R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov. System-on-chip: Reuse and integration. *Proceedings of the IEEE*, 94(6):1050–1069, 2006.
35. J. Shao and B. Davis. A burst scheduling access reordering mechanism. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 285–294, 2007.
36. S. Sriram and S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC, 2000.
37. L. Steffens, M. Agarwal, and P. van der Wolf. Real-Time Analysis for Memory Access in Media Processing SoCs: A Practical Approach. *ECRTS '08: Proceedings of the Euromicro Conference on Real-Time Systems*, pages 255–265, 2008.
38. C. van Berkel. Multi-core for Mobile Phones. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2009.
39. S. Verdoolaege, H. Nikolov, and T. Stefanov. PN: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007, 2007.
40. R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009.



<http://www.springer.com/978-1-4419-6459-5>

Multiprocessor System-on-Chip
Hardware Design and Tool Integration
Hübner, M.; Becker, J. (Eds.)
2011, VIII, 270 p., Hardcover
ISBN: 978-1-4419-6459-5