IPSJ Transactions on System LSI Design Methodology Vol. 4 91–116 (Aug. 2011)

Invited Paper

Mastering MPSoCs for Mixed-critical Applications

Philip Axer,[†1] Jonas Diemer,[†1] Mircea Negrean,[†1]
Maurice Sebastian,[†1] Simon Schliecker[†2]
and Rolf Ernst[†1]

Multi-Processor Systems-on-Chips (MPSoCs) emerge as the predominant platform in embedded real-time applications. A large variety of ubiquitous services should be implemented by embedded systems in a cost- and power-efficient way, yet providing a maximum degree of performance, usability and dependability. By using a scalable Network-on-Chip (NoC) architecture which replaces the traditional point-to-point and bus connections in conjunction with performant IP cores it is possible to use the available performance to consolidate functionality on a single MPSoC platform. But especially when uncritical best-effort applications (e.g., entertainment) and critical applications (e.g., pedestrian detection, electronic stability control) are combined on the same architecture (mixed-criticality), validation faces new challenges. Due to complex resource sharing in MPSoCs the timing behavior becomes more complex and requires new analysis methods. Additionally, applications that may exhibit multiple behaviors corresponding to different operating modes (e.g., initialization mode, fault-recovery mode) need to be also considered in the design of mixed-critical MPSoCs. In this paper, challenges in the design of mixed-critical systems are discussed and formal analysis solutions which consider shared resources, NoC communication, multi-mode applications and their reliabilities are proposed.

1. Introduction

Driven by power constraints and efficiency considerations, embedded system designers are adopting the trend towards multi-processor systems-on-chips (MPSoC). MPSoCs integrate an increasing number of heterogeneous processing cores ranging from general purpose processors of varying complexity to special purpose accelerator IP. Individual IP cores, I/O and main memory are interconnected with each other via a Network-on-Chip (NoC), which offers a scalable communication infrastructure. **Figure 1** shows a generic MPSoC architecture.

MPSoCs are being or will be used in all embedded application domains like telecommunication (e.g., smart phones), transportation (e.g., car electronics) and industrial automation (e.g., process control). In all domains, it is appealing to use the vast amount of available resources on MPSoC platforms for efficient consolidation of functionalities. By using a powerful MPSoC platform, it becomes possible to integrate computationally intensive algorithms (e.g., camera supported pedestrian detection) with control oriented applications (e.g., active steering). These applications have **mixed-critical** requirements. Some applications may have hard real-time constraints or are safety-critical whereas others are non-critical at all (e.g., best-effort entertainment). This requires special considerations in the design of MPSoC platforms and deployment of applications. Furthermore, increasing design costs will force MPSoC manufacturers to offer more flexible solutions that can target a wider range of applications. This means that the hardware support required by mixed-critical real-time applications must be flexible to be adapted for different applications.

Many embedded systems combine functions of different relevance to the overall system mission. For instance, integration of pedestrian detection and entertainment applications on the same platform is already a mixed-critical system, as-

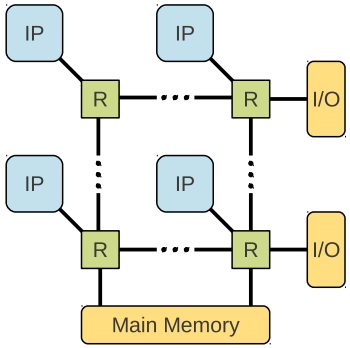

Fig. 1 Generic MPSoC architecture with different IP cores (e.g., processors, special-purpose accelerators), I/O and main memory interconnected by a set of routers (R).

†1 Institut für Datentechnik und Kommunikationsnetze, Technische Universität Braunschweig
†2 Symtavision GmbH

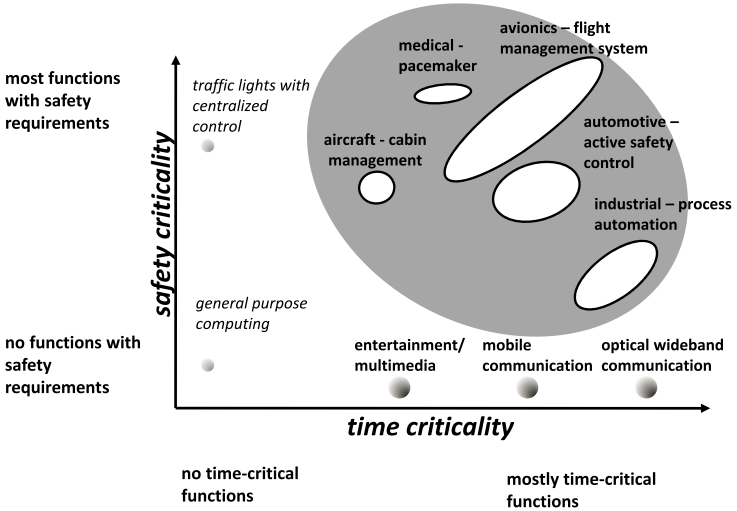

Fig. 2 The two dimensions of criticality: safety and time.

suming that pedestrian detection is far more important than e.g., entertainment. This property is called mixed criticality. *Criticality* can be broken down into two orthogonal dimensions or aspects as depicted in **Fig. 2**.

- *time criticality*
- *safety criticality*

For time critical applications without safety requirements, the sole focus lies on timeliness of computation or communication. Typical examples for this domain are embedded mobile communication applications e.g., UMTS or LTE. Here, guaranteed data integrity is not safety-relevant because transient or permanent error conditions do not have catastrophic consequences. The methods and concepts generally used to provide sufficient timing guarantees for timing critical systems are especially focus of the hard-real time domain.

For purely safety defined systems it is crucial that integrity of computation is preserved, even in presence of errors, e.g., traffic lights which are controlled by a centralized control facility. However, it is not critical if traffic light phases are of accurate timing, as long as the light signals are consistent. In case of failures, pedestrians and car passengers can be injured or even killed, thus safety demands are high. The taxonomy of safety is defined by safety standards and focus of dependability research. A concise overview of dependability taxonomy and terminology can be found in Ref. 5).

Highly safety and time critical functions with significant computation requirements are no longer limited to niche markets, such as military and avionics, but have started to appear in high volume embedded system markets such as the automotive domain. One reason among others for this development are new statutory provisions for liability as for instance passed in the European Union [19),43)].

Deployment in safety-critical domains makes a strongly safety oriented product life-cycle necessary which qualifies a product for deployment in safety-critical missions. This is not only crucial in order to minimize the risk of casualties in case of system failure but also ensures product liability. To unify safety requirements, safety standards such as the industrial-oriented IEC-61508 [26)] specify a safety certification process. Domains with application-specific requirements either have developed dedicated standards such as DO-178B [49)] and DO-254 [48)] for aerospace, or have derived new standards e.g., from IEC 61508, such as the automotive domain specific standard ISO-26262 [27)] which inherited common concepts from the IEC-61508.

Most functional safety standards follow a top-down approach which considers the entire system for certification: For IEC 61508, safety-functions need to be defined first. These are technical functions which are intended to achieve or maintain a safe state of the system. Secondly, a risk assessment identifies the safety integrity level (SIL) of each safety function. This SIL is a general starting point for system design and defines guidelines for HW-architecture (e.g., fault tolerance), testing and validation effort. If the system is certified according to SIL guidelines then risks are reduced to a tolerable level.

Of special interest are domains with both safety and timing requirements. This is also highlighted in Fig. 2. It is especially challenging because techniques from real-time and dependability domains need to be combined.

1.1 Challenges in Mixed-critical MPSoC Design

Providing performance (e.g., timing) guarantees is one of the key aspects of system architecture design and has been recognized as a major challenge when designing time-critical architectures. Supporting real-time applications in clas-

sical single-core and distributed systems is already challenging. Dependencies of individual application components (tasks) using the same resource (e.g., processor, bus, NoC, memory) must be considered to assure that individual timing constraints (e.g., task response times, end-to-end deadlines) are met under the available resource constraints (e.g., processing time, memory, buffer size).

The same applies also to MPSoCs, however with additional challenges caused by accesses to shared resources such as Network-on-Chip , IP-cores or main memory. The use of physically shared hardware (e.g., shared memory), or synchronization via logical resources (i.e., semaphores) introduces a new level of inter-core dependencies that are not observed in distributed systems. Tasks which are mapped on different cores and share common memories or synchronize resource accesses via locks administered according to lock-based arbitration policies [51], will delay their core-local execution when waiting for the arrival of the requested data or to access the required resource [39),56),61]. The local execution of a task on a core is now influenced by the local execution of other tasks on other cores, thus challenging the real-time behavior of the entire MPSoC and with this the expected benefits of MPSoC setups [39),56)]. In order to employ such components in critical real-time systems, care must be taken to consider the implications on the system timing.

To guarantee the required timing, designers have two classes of solutions: *isolation* and *analysis*. The former employs hardware and software mechanisms to isolate individual applications (or tasks) from each other so that they can not potentially interfere by design. This can be achieved by the orthogonalization of system resources. As a hardware solution, conflicting tasks can be placed on separate processors. Alternatively, an isolating TDMA schedule (software) could be employed. Busses and other shared resources may be assigned to different processors in alternation according to a time-driven schedule [2),8),31)]. Such mechanisms guarantee an execution that is independent of the actual run-time behavior of other applications. Each application component can then be verified in isolation. While isolation simplifies the verification procedure, it also implies a conservative design that is inflexible to application changes and in general with increased resource requirements. Furthermore, application which exhibit dynamic behavior, such as varying bandwidth requirements, can only very inefficiently be mapped to static schedules.

Formal analysis approaches, on the other hand, allow for dynamic system behavior and more flexible sharing of resources. By using mathematical analysis, it is necessary to proof that the interference between tasks does not lead to timing violations, even in the worst-case [56),68)]. For analysis approaches addressing the use of shared resources, the amount of resource accesses per task activation needs to be known, as well as the number of task activations. The latter however can not trivially be bounded in event-driven applications where, for example, a task activation is the result of a message produced on another processor. For logical protection of shared resources, i.e., the execution of exclusive critical sections, a number of high-level protocols have been proposed (e.g., Refs. 10), 51)). These methods provide an arbitration scheme, possibly tailored towards a particular task scheduler, and an analysis to calculate the upper bounds on the time before a lock is granted. As opposed to single processor protocols (such as the priority ceiling protocol [51)]), the time a task is delayed due to stalled shared resource accesses (blocking time) in MPSoC setups usually depends on the pattern of task activations. This dependency is a major hurdle in a general multiprocessor shared resource analysis. Given event-driven task activations and dynamic scheduling, the amount of interfering task activations is unknown without a system-level analysis, which can only be done when every task's resource usage is known, this again leads to a cyclic dependency [56)]. Thus, the problem can not be generally solved with classical methods.

Analysis aproaches require precise knowledge of the temporal behavior of each application, such as activation pattern and task execution times, to derive accurate timing bounds. If this information is not available, e.g., for non-critical best-effort applications, a safe overestimation must be done, which can also lead to inefficient solutions. In case no safe overestimation is feasible, the adherence to a timing model must be enforced during runtime (e.g., through traffic shaping).

Hence, for systems executing both real-time and best-effort applications concurrently, isolation and analysis approaches must be combined. Isolation techniques are used to avoid interference between different application classes (such as real-time and best-effort) at places where the interference from unknown best-effort tasks can not be reasonably bound. Analysis can then be employed among the

real-time applications.

The safe design becomes even more complicated when systems have to accommodate multi-mode real-time applications. Some applications, for example in safety-critical avionic and automotive control systems, exhibit multiple behaviors corresponding to different operating modes (e.g., initialization mode, fault-recovery mode). At runtime, in order to adapt their behavior to system internal and external changing conditions, multi-mode systems will switch between different operational modes. This means, the systems will experience transitions from an old mode characterized by a set of functionalities (i.e., a set of tasks) to a new mode characterized by a different set of functionalities.

To control the transition between operational modes, designers can opt for *synchronous* or *asynchronous* mode change protocols [52]. The first type of protocols ensures isolation between the execution of mode specific functionalities, i.e., tasks corresponding to a new operational mode will not be started before all the tasks of the old mode have completed their execution. Synchronous protocols do not require specific schedulability analysis for the transition phase, however delaying the start of the new mode applications is not always suitable as this could counter the timing of new mode actions that must be performed as soon as possible (e.g., when switching to an emergency mode). Asynchronous protocols overcome this limitation and allow functionalities of the new mode to be started simultaneously to the old mode functions. However, the execution of functionalities of both modes generates an increased workload during the transition and can potentially lead to timing violations. Therefore, asynchronous protocols require specific schedulability analysis.

In order to facilitate a structured design process that leads to a reliable system it is important to provide solutions to bound the system transition latency and to guarantee that timing constraints are not violated at any moment of the system's execution, neither in the individual operational modes nor during the transition phases.

As already discussed, safety regulations dictate a minimal level of dependability for safety critical functions. In order to obtain the required reliability, it is necessary to introduce a certain degree of redundancy to detect and either correct errors or switch to a safe state which in turn can be considered as a switch between operational modes.

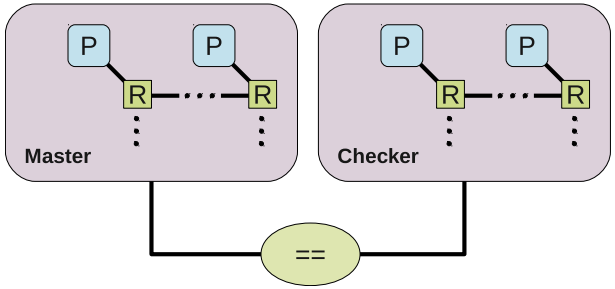

Fig. 3 Example for a typical DMR implementation, leading to 100% overhead.

Traditionally, methods such as replication [41], for instance dual modular redundancy (DMR) and triple modular redundancy (TMR) are used for this. This is generally referred to as *spatial redundancy*. For the automotive domain, spatial redundancy trade-offs have been identified in Ref. 6).

A simple example for a DMR version of an MPSoC is shown in **Fig. 3**. In this example, the architecture is entirely replicated, hence it consumes at least 100% more resources compared to the non fault-tolerant case and is capable of detecting errors in all functional units with very good coverage.

Contrary to this concept, redundancy can also be implemented in time, by reevaluating computation multiple times which is called *temporal redundancy*. Obviously spatial and temporal redundancy can be combined in various ways to gain the desired trade-off. For traditional single critical applications, in which all of the functionality requires a high safety standard, the mentioned over provisioning is inevitable, simply because there is no alternative. But for mixed-critical applications the inherent problem of coarse granular concepts is that redundancy is potentially introduced in places where this is not even needed or the required overhead (space, power) is so high, that coarse grained methods are not feasible at all.

One solution to this problem is to replicate only critical tasks. In this approach the *inherently redundant architecture* of today's MPSoCs is leveraged. The timing behavior can be influenced by errors, especially if methods such as checkpointing and rollback are used. It is especially challenging to analyse these effects

and include this in a reliability analysis which incorporates these timing effects. Thus, the consideration of mixed-critical applications introduces new challenges in platform design. The dramatic safety consequences have been highlighted in Ref. 7). Here the important aspect of certification requirements on mixed critical applications has been recognized. Most safety standards explicitly require guarantees of sufficient isolation between applications of different criticality. Without new analysis methodologies these guarantees cannot be provided.

This paper addresses the challenges outlined above from different angles. As a foundation, in Section 2 we first introduce a general system model and we give an overview of existing analysis techniques that can be applied for the design of heterogeneous MPSoCs. This system model is then specialized for the individual aspects in the following sections. In Section 3 we highlight the impact of the shared resource usage on the MPSoC's timing and present a dedicated formal analysis approach. The interference in the on-chip communication infrastructure is resolved via hardware quality-of-service mechanisms and a corresponding analysis in Section 4. Section 5 discusses the effects of mode changes and how they can be considered in the system design. The reliability and redundancy aspects of MPSoCs are covered in Section 6. Based on these aspects, we outline in Section 7 how a system-wide analysis of timing and reliability can be performed. Finally, we conclude in Section 8 where we also provide an outlook of open challenges.

2. System Model and Analysis Concepts

2.1 MPSoC System Model

The analyses which will be introduced in the following sections are based upon a common MPSoC system model. This is composed of a set of heterogeneous applications consisting of computation and communication tasks $\tau = \{\tau_1, \ldots \tau_n\}$. Tasks are assumed to be statically mapped on a set of processing (CPUs) and communication (Buses) resources and executed according to scheduling policies (e.g., round-robin, static-priority preemptive, etc.), see **Fig. 4**.

Each instance of a task, called a job, is activated by an event, which can be either external (such as interrupts), or the result of another task or bus communication being finished.

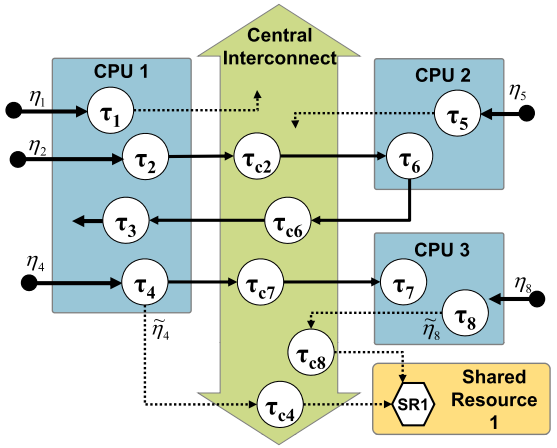

Fig. 4 MPSoC system model – illustrative example.

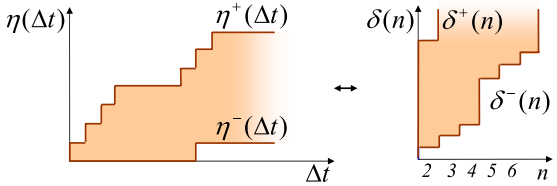

Fig. 5 Event stream representations.

We express task activation patterns with event streams (see **Fig. 5**) using the upper event arrival function $\eta_i^+(\Delta t)$, and the lower event arrival function $\eta_i^-(\Delta t)$ [53]. These specify the maximum and the minimum number of events that occur in the event stream during any time interval of length Δt. Inversely, event streams can be specified using the functions $\delta_i^+(n)$ and $\delta_i^-(n)$ that represent the largest and smallest time window in which n events can be observed in the stream. The functions η and δ are pseudo-inverse to each other.

Each job of a task τ_i has an execution time with a known lower and upper bound $[C_i^-, C_i^+]$, referred to as best-case execution time and the worst-case execution time. For hard real-time jobs, the response time R_i (i.e., the time between job

release and job completion) must be smaller than a given (relative) deadline D_i. This deadline may be smaller, equal or larger than the distance to the successive activation.

If the worst-case response time, which is the largest response time under conservative conditions, of a task is larger than $\delta_i^-(2)$, it is possible that another instance of this task may be activated before the previous one has completed. In this case, we assume that new jobs may not actually start to execute before the previous job is complete, and this queueing delay will be considered as part of the job's response time.

A chain of communicating tasks is called a path. For paths, a worst-case end-to-end latency can be defined, which denotes the maximum time span from the occurrence of an event at the first task of the chain until the event is produced by the last task of the chain. Intuitively, the summation the tasks' worst-case response times along the path give an upper bound for the end-to-end latency, but more efficient analysis methods exist [58].

In addition, as depicted in Fig. 4, tasks mapped on different CPUs may arbitrarily access shared resources (such as logical semaphores, physical coprocessors or shared memories) during execution. Shared resources in the system are objects that require serialized access. Accesses to the shared resources can be arbitrated according to lock-based arbitration policies which can be performed either with suspending, as done by the multiprocessor priority ceiling protocol MPCP [51] (when a task suspends, the processor will become available for other work) or with spinning [13] (tasks perform a busy-wait until the lock of the required resource is released; in this time the processor and potentially the interconnect cannot be used by other tasks). A mixture of suspension-based and spinning-based resource arbitration can also be used as proposed e.g., in the automotive standard AUTOSAR [4]. The load imposed by tasks on the shared resources is denoted with the functions $\tilde{\eta}$. This is expressed using the event model concept, which is used to model task activations, as proposed in Refs. 38), 56).

2.2 Formal Performance Analysis

Two classes of approaches for performance analysis of heterogeneous distributed systems and MPSoCs can be found in literature: the holistic [42),45),67)] and the compositional approach [16),24),53),54)].

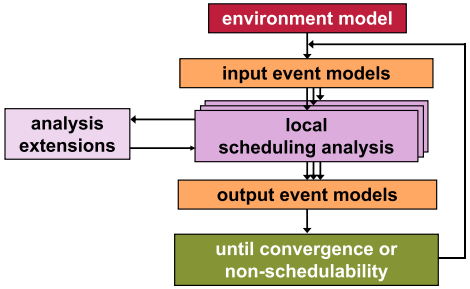

Fig. 6 Compositional system level performance analysis loop.

The holistic approaches systematically extend the classical single-processor scheduling theory to distributed systems. The global view on the system allows to take global dependencies into account, thus providing tightly calculated analysis bounds. However, because of the very large number of dependencies, the complexity of the analysis grows with system size and heterogeneity. Therefore, holistic approaches are difficult to be used for arbitrary systems and are, in practice, limited to deterministic system configurations such as TDMA networks.

The basic idea of compositional approaches as illustrated in **Fig. 6**, is to break down the analysis complexity of complete systems into separate local component analyses and to integrate local performance analysis techniques, e.g., uniprocessor scheduling analysis known from real-time research, into system-level analyses. First, the external activation patterns are derived from the environment (e.g., sensor sampling rates, maximum engine rpm, minimum human response time). The behaviors of the individual tasks are investigated in detail to gather all relevant data such as the best-case and worst-case execution times. This can be derived with formal methods such as in Ref. 70), but extensive simulation is also common in practice. This information is then used to derive the behavior within individual components (such as a processor or a bus), accounting for local scheduling interference. The composition is achieved by connecting the component's inputs and outputs by stream representations of their communication behavior using event models [16),23),53)]. Based on the results of the individual components, tasks' output event models are determined. These are then propagated to the inputs of the connected components, where they are used as activating event models for

the subsequent iteration. System performance analysis can then be performed by iteratively alternating local (i.e., component) analysis, and event stream propagation between components. In each iteration, with knowledge of the tasks' activation pattern and of the components' scheduling strategy, local component analyses systematically derive worst-case (best-case) scenarios to calculate worst-case (best-case) task behavior.

Response time analyses are available from real-time research for a large variety of different scheduling policies, which can be directly applied. For example, when computing the worst-case response time of a task τ_i on a single-resource (i.e., processor or bus) under static priority preemptive scheduling, one can rely on the *busy window* technique [35),68)]. The *busy window of a task* τ_i is defined as the maximal time interval for which a resource executes only tasks of priority greater than or equal to the priority of task τ_i and during which the resource is never idle [68)]. The maximum response time of task τ_i can then be derived from the busy window [68)].

This iterative analysis — alternating local analyses based on current event models and the derivation of updated output event models as in Fig. 6 — represents a fixed-point problem. For systems containing cyclic dependencies between two or more components, initial event models are required to begin the local analysis. A solution for the so-called *starting point generation* is proposed in Ref. 24). The starting-point is generated by propagating all output event models along all paths until an initial activating event model is available for each task. After each local analysis iteration, all output event models can only become more generic [29),54),65)], meaning that each iteration contains the previous models. Thus, the complete procedure is monotonic. The global analysis is repeated until the event models in two consecutive iterations remain unchanged (i.e., a fixed-point is found) or if an abort condition (e.g., violation of a timing constraint) is reached. If a fixed-point is found, it is conservative, meaning that all observable response times in the real implementation are less than or equal to the computed one [29),54),65)].

Traditional scheduling analysis only considers independent resources (e.g., busses or processors with different tasks). However, multi-core systems include also tasks which require access to other shared resources (e.g., memory controller in a multiprocessor or a mutex variable) during their execution. Recently, extensions for multi-core system analysis have been proposed [38)]. For every shared resource, an arbitration protocol must be specified that resolves concurrent accesses to the shared resource similar to the scheduling protocol of the processing resources. In the global analysis procedure, the local analysis can be extended (see Fig. 6) to account for additional resources, e.g., by extra blocking from tasks which potentially contend for a shared resource [56)].

The advantage of the compositional performance analyses is their great flexibility and scalability, thus they allow considering arbitrarily complex architectures and overcome the restrictions of the holistic approaches.

3. Considering Shared Resources in MPSoCs

3.1 Timing Implications of Shared Resources in MPSoCs

To illustrate the timing implications of shared resources in MPSoCs, consider the system model in Fig. 4. Assume that tasks τ_1, τ_2 on CPU1 and τ_5 on CPU2, which are scheduled on their CPUs according to the static priority preemptive scheduling, share a common memory that can only serve one request at a time. Furthermore, assume that a core is stalled whenever a task performs a memory access until that request has been served ("busy waiting").

A possible schedule is depicted in **Fig. 7**. In the case where only tasks τ_1, τ_2 execute from the same memory (Scenario 7 a) the low priority task τ_2 is kept from

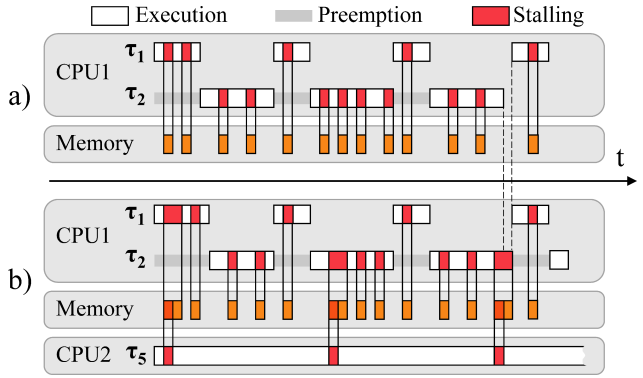

Fig. 7 a) Tasks on a single processor accessing a remote memory. b) Conflicting accesses from tasks mapped on different processors.

executing by three invocations of the high priority task τ_1. The completion time of the lower priority task is delayed due to itself fetching data from the memory, and due to the prolonged preemptions by the higher priority task (as its requests will also stall the processor). Scenario 7b shows the case where the memory is also used by task τ_5 on CPU2, in this case periodically. Whenever the memory is also used by a task on CPU2, CPU1 is stalled for longer time, which increases the task response times. In addition, the response time of the low priority task τ_2 has grown so much that it suffers a fourth preemption by the high priority task. These effects challenge the safety of the task's deadlines. Additionally, the busy wait adds to the execution time of a task and, hence, increases the processor load. As a result, not only response time grows but the overall system may become unschedulable. This example shows that the tasks' response times are influenced by the delay caused by the use of shared resources.

In this way the local execution of a task on a core is now infuenced by the local execution of other tasks on other cores, a fact that challenges the real-time behavior of the entire MPSoC.

This behavior challenges also the analysis methodology. In the general case, the duration of the delay that tasks experience when accessing shared resources depends on the amount of traffic imposed on the shared resources by e.g., other processors in the MPSoC setup. The respective local analysis of the other processors however also requires the shared resource delay which closes a cycle: The timing interference in the system translates into a mutual dependency between the local analyses of the different cores. To provide a reliable upper bound on the tasks' timing which is required in real-time systems, additional measures must be taken. Formal performance analysis methods which have been successfully applied for single-core distributed systems need to be extended in order to allow the timing analysis of the more complex MPSoC architectures.

3.2 MPSoC Analysis in Presence of Shared Resources

The previous section has highlighted the complex dependencies that arise in MPSoC setups through the common use of shared resources. Contrary to traditional system verification, a subcomponent, such as a task or a processor, can not be verified in isolation, because elementary properties are broken when the components are integrated.

A methodology that allows to capture the inter-core timing dependencies and calculate bounds on the tasks' response times even in the presence of dynamic scheduling and shared resources has been proposed in Refs. 38), 56). The main idea is to separate the timing analysis procedure into three disjoint steps:

(1) First, the load imposed by tasks on shared resources $\tilde{\eta}$ has to be determined. By considering the pattern of task activations η and the distance between requests issued by each task, the overall load imposed on the shared resource can be derived for each task and all tasks on a processor [1),60)]. Depending on the accuracy of the request model, the analysis of the shared resource delay can vary significantly, because if larger request distances can be formally guaranteed, many conflicts can potentially be ruled out. An accurate model of the shared resource load is therefore fundamental for obtaining tight performance analysis results. A new and precise method for the derivation of shared resource load was proposed in Ref. 57).

(2) Second, the information about the load imposed on the shared resources has to be used to derive the maximum delay that a task may experience when accessing shared resources. On the shared resource, coinciding requests will be arbitrated according to a specific policy, which leads to specific delays for each request. The aggregate delay that is experienced by the requests of a specific task can be conservatively bounded by considering the competing requests from all tasks in the system. An efficient solution has been proposed in Ref. 56), by focusing on the aggregate busy time of all requests. In Ref. 38) the blocking time under MPCP [51)] was derived.

(3) Third, the obtained delays need to be integrated in the worst-case response time. As shown in Refs. 56), 59), for priority-based schedulers it is an accurate and conservative method to accumulate both the local execution and the remote delays.

In event-driven multiprocessor systems, this analysis procedure has various mutual dependencies (between the task activating event models η, the shared resource delays $\tilde{\eta}$, and the task response times). To deal with these cyclic dependencies, the proposed solution is to integrate the shared resource analysis into the iterative analysis flow of the compositional analysis procedure introduced in Section 2.2. For this, one can rely on the event model propagation concept of the

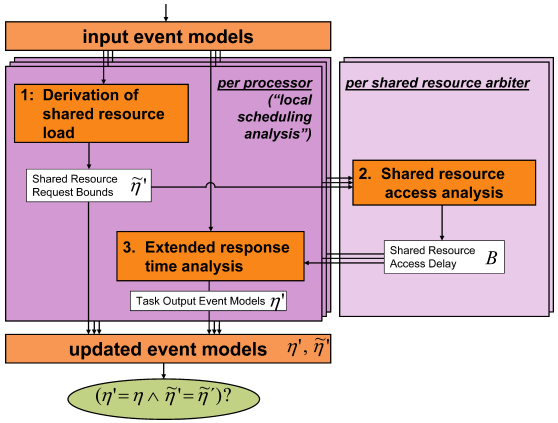

Fig. 8 Extended compositional analysis procedure in the presence of shared resources.

compositional analysis to express not only the task activations (e.g., using the functions η) but also the load imposed on the secondary resources $\tilde{\eta}$.

Figure 8 illustrates the general analysis procedure for a system with three processors and one shared resource. Given a set of activating event models (η) for each task in the system, a set of shared resource access event models ($\tilde{\eta}$) can be derived as presented for example in Ref. 56). Based on the shared resource access event models per processor, the shared resource access delays B can be computed for each task. The respective blocking times then become part of the response time analysis of each task on each processor, from which the updated output event models η' can be derived. The process is repeated as long as any event model estimate has been refined. The convergence condition is fulfilled when all task activating event models η and all shared resource request bounds $\tilde{\eta}$ have not changed after an iteration.

4. Considering NoC Communication

The NoC forms the central system interconnect of an MPSoC, as it participates in almost any transaction in the system, be it core-to-core communication, main memory access, access to shared special-purpose IP, or I/O transfers. This leads to interference of traffic (i.e., communication tasks), because the NoC is shared between all system components.

The NoC has to support the different requirements originating from the applications, which is referred to quality of service (QoS). *Real-time* (*RT*) applications usually require a maximum bound on transfer latency and a minimum throughput guarantee. *Best-effort* (*BE*) traffic does not require a guaranteed service other than being eventually delivered. However, best-effort traffic is often very sensitive to latency, as it results from cache misses which stall the processor. Note that this is different from "timing-critical" real-time applications, as increased latencies only reduce the performance of BE applications, but do not make the computation result useless, as it is the case for real-time applications. Hence, the NoC should provide a low latency to best-effort traffic to increase the throughput and responsiveness of these applications.

4.1 Resolving Interference

Interference occurs at the routers when packets of different traffic streams try to use the same route at the same time. **Figure 9** a) shows a generic router architecture with input buffering and a crossbar switch that multiplexes packets from the input modules to the output. Within the router, traffic streams contend for two different resources: FIFO queues (or virtual channels) for buffering at the input and output ports for transmission. Due to the high cost of on-chip buffers, NoCs usually employ wormhole switching which divides packets into smaller units (flits) that are transferred and buffered individually. This leads to the distribution of individual packets across multiple routers and hence multiplies the effects of contention compared to a distributed off-chip network.

Contention for virtual channels (VC) must be resolved before a packet can advance to the next router. Once a VC has been assigned to a specific packet, it can not be used by other traffic until that packet has been completely forwarded to the next router. This in turn depends on the arbitration at the output and on the interference at downstream routers, so a tight bound can not be easily derived. For best-effort traffic, it is sufficient to assign the VC on a first-come first-serve basis. Real-time traffic, however, must reserve VCs in advance to avoid potentially long blocking times and overestimation in the analysis.

The interference at the output port is resolved by an output arbiter, which decides which traffic stream may progress at each cycle. Isolation between indi-

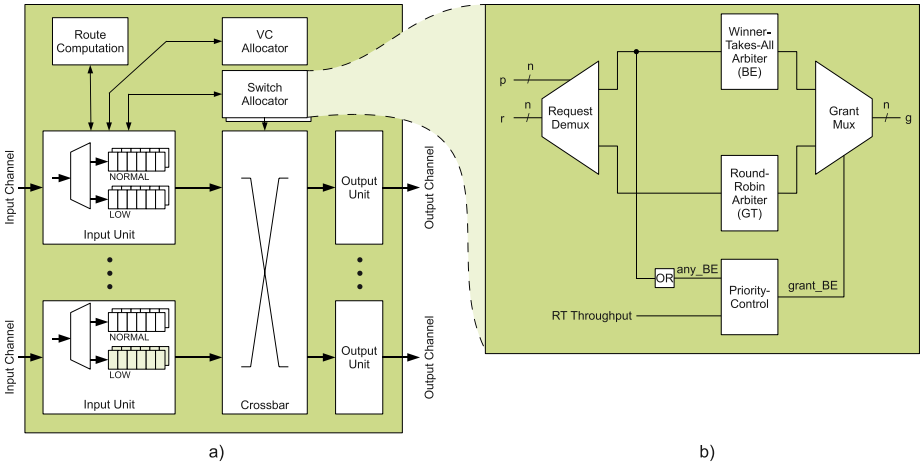

Fig. 9 Output Arbiter with Quality-of-Service and improved support for best-effort traffic.

vidual traffic streams can be implemented by considering different traffic classes (e.g., real-time and best-effort) during the arbitration. Because providing a distinct traffic class for every stream in the system is not feasible, multiple streams of equal criticality usually share the same class.

There are several existing schemes concerned with providing service guarantees (or quality-of-service, QoS) in Networks-on-Chip, which use either static scheduling (such as time-division multiple-access) [21),37)] or dynamic scheduling (e.g., based on priorities) [9),11),17),20),33),36)]. The former group is less flexible and can not react to changes in traffic behavior very well. For the latter group, an analysis must be performed to assure real-time guarantees, because guarantees of individual streams depends on the behavior of others (e.g., those with higher priority).

4.2 Improving the Latency of Best-effort Traffic

Most of the QoS schemes proposed so far treat best-effort traffic as "second-class citizen," allowing it only to utilize idle time slots or assigning it to the lowest priority. While this certainly makes sense from an isolation perspective, it leads to an unnecessary increase of transfer latency for best-effort traffic. At the same time, the improved latency for real-time traffic is often not required. Due to its regularity, real-time traffic can often tolerate high latencies as long as a certain latency bound is met. In other words, there is no need to finish a real-time transfer way ahead of its deadline. Hence the reduced latency implied by prioritizing real-time traffic is of no immediate value.

This problem must be solved, because the performance of best-effort applications is important in mixed-critical systems. An approach is to allow best-effort traffic to pass real-time packets as long as the real-time deadlines are not jeopardized. This can be assured by a selective prioritization of best-effort traffic which is reverted if real-time packets are blocked too often.

Figure 9 b) shows an arbiter that implements such an arbitration scheme. Requests "r" are handled by separate sub-arbiters based on their priority "p". A priority control logic determines which traffic class is currently prioritized based on the observed real-time throughput.

The priority control can be driven by Distributed Traffic-Shaping (DTS) [18)], which uses a traffic shaper at every output port to limit the prioritization of best-effort traffic. The shaper contains a bucket of tokens, which are good for one prioritized transfer of best-effort traffic. Each time a best-effort flit is sent on a high priority, a token is removed from the bucket. The bucket is periodically replenished at a rate which leaves enough unprioritized time for real-time traffic.

An alternative strategy to control the priority logic is Back Suction (BS) [17)]. This mechanism monitors the buffer occupancy of each real-time virtual channel at each router. As long as all real-time buffers are sufficiently filled, best-effort traffic is prioritized. Once the buffer occupancy drops below a specified threshold, a router requests the prioritized transfer of real-time data from its upstream neighbor using a "Back Suction" signal. Back Suction prevents the depletion of real-time buffers and can be seen complementary to back pressure (which prevents buffer overflow). The suction signal is generated at the sink of an real-time traffic stream at a limited rate (the guaranteed rate of that stream) and propagates towards the source as long as the buffers are not sufficiently filled. The resulting behavior is that buffer space is used to allow idle progress of real-time traffic, which can later be exploited to allow best-effort traffic to be prioritized. This way, best-effort traffic receives a minimum latency as long as it leaves enough bandwidth for real-time traffic.

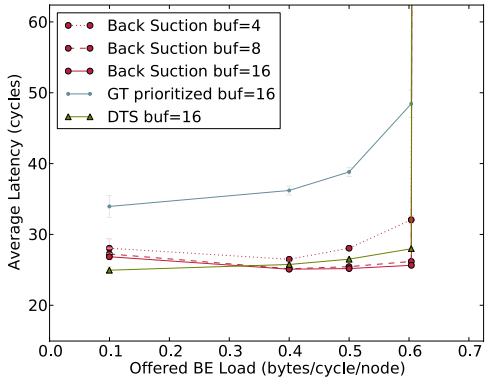

Fig. 10 Latency of a best-effort (BE) traffic overlapping with a real-time traffic stream (GT) for increasing best-effort load with different arbitration techniques (Back Suction, Distributed Traffic Shaping (DTS) and simple priorization).

Figure 10 shows the average latencies of best-effort traffic when overlapping with a real-time traffic stream with increasing best-effort load. The different curves represent different quality-of-service schemes, "GT prioritized," Distributed Traffic Shaping (DTS) and Back Suction. The former corresponds to a prioritization scheme where real-time traffic is always favored over best-effort, which is done by most regular QoS schemes. All schemes are suitable to meet the real-time requirements (not shown). For this, DTS and BS require that the rate limitation is configured according to the throughput requirement of the real-time traffic streams. Regarding the best-effort traffic, it can be seen in Fig. 10 that the latency under low and medium load is vastly improved for the DTS and BS schemes compared to the standard prioritization scheme. Back Suction achieves this using less buffer space than DTS.

4.3 Real-time Analysis of the NoC

By using the Distributed Traffic Shaping or Back Suction techniques, we can efficiently isolate real-time from best-effort traffic. To determine the maximum latency and minimum throughput guaranteed to each real-time stream, an analysis of the arbitration at each router is performed. This is required because all real-time traffic shares the same priority class using a round-robin scheduler. Hence, the guarantees for an individual stream depend on the behavior of other

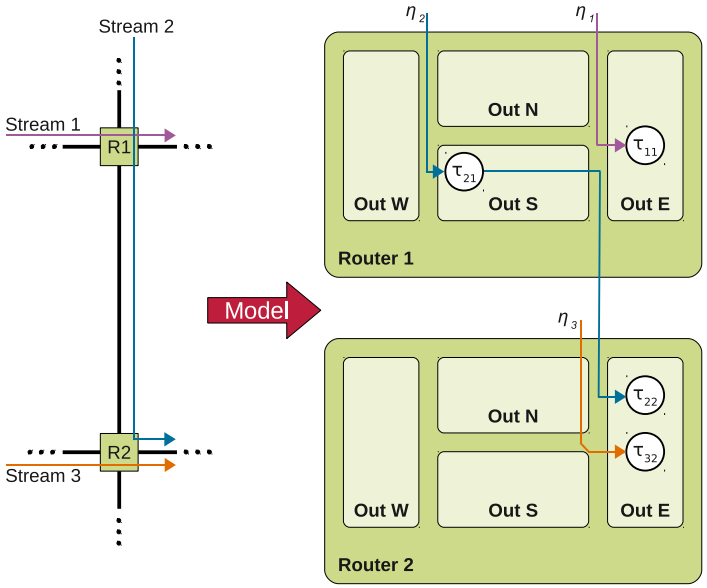

Fig. 11 Extended system model for Networks-on-Chip.

real-time streams. These are fortunately well defined for real-time traffic allowing an analysis resulting in low overestimation (tight guarantees). The analysis of real-time streams follows the same principle as outlined in Section 2. Other analysis approaches for NoC communication exist [22),34),50),64)], which rely on a similar principle for real-time traffic but assume different architectures that favor real-time traffic at the cost of reduced best-effort throughput.

We extend the system model introduced in Section 2 by replacing the central interconnect, which was represented by a communication resource, by multiple interconnected resources representing the individual routers. **Figure 11** illustrates the system model for two routers and three traffic streams which are represented in the system model by the tasks τ_{11}, $\tau_{21} \rightarrow \tau_{22}$, and τ_{32} and their respective activation models η_i. A traffic stream is modeled as a series of tasks and an input event model. Each router output is represented by a separate resource, containing one task per stream which is serviced by that output.

The scheduling strategy and parameters are similar to those used by processing

resources. For the Back Suction mechanism, the scheduler is round-robin with only the real-time streams being considered. Best-effort streams can be ignored because Back Suction eliminates their interference. A round-robin scheduling analysis is then performed at every resource locally to derive output event models which are then propagated in the system, see Fig. 6. From the analysis results, the required performance metrics are derived, such as per-hop worst-case latency (worst-case response time of a task), end-to-end latency (path latency), or maximum backlog. By checking against the constraints implied by the application (e.g., deadlines) or the architecture (e.g., FIFO depth), the feasibility of the real-time streams is proven.

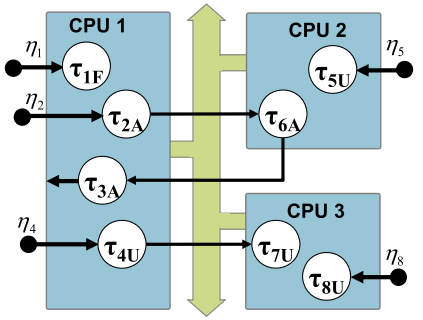

Fig. 12 Example of a multi-mode system during the transition phase — tasks from both modes execute on the system.

5. Considering Multi-Mode Applications

Predicting timing behavior is essential for the design of real-time systems that can switch between different operational modes at runtime. Several approaches that address the problem of timing analysis for multi-mode real-time systems have been proposed by the research community [25),40),44),52),66)].

However, most of the existing solutions rely on the assumption that applications consist only of tasks which execute independently [40),44),52),66)]. But, many real-time systems are composed of multiple processors and accomodate distributed applications consisting of multiple communicating tasks as illustrated in **Fig. 12**. Through various dependencies in MPSoCs, the initiation of a mode change has not only a local effect but also impacts the timing of tasks executing on other resources [25)]. In order to highlight the influence of mode changes on the system's timing, and to overview analysis solutions, we first extend the system model introduced in Section 2.1 to consider multi-mode applications with communicating tasks.

5.1 Multi-Mode System Model

Figure 12 illustrates the multi-mode system model during the transition phase from an old mode to a new mode. The general assumption is that each individual mode of the system is characterized by a different behavior and is associated with a specific set of tasks together with its timing properties, e.g., task execution times, priorities and deadlines.

We consider that a mode change request (MCR) occured at time t_{MCR} triggered by the environment or by the system internal requirements. In order to exclude interference of multiple mode changes, the common assumption is that a MCR can be served only if the system is running in a steady mode, i.e., a MCR cannot be considered if the system executes a transition between two modes as a result of a previous MCR. As discussed in Section 1.1, even if synchronous mode change protocols isolate the execution of old and new mode tasks and thus do not require explicit analysis methods, they are not suitable when a fast mode change has to be performed (e.g., when switching to an emergency mode). Therefore, we focus our attention to asynchronous protocols which however impose a more complex timing behavior in MPSoCs. Thus, triggered by the occurrence of the MCR we assume that an *asynchronous* mode change protocol imposes for example a mode change that consist in removing task τ_{1F} from CPU1 and adding the tasks τ_{2A}, τ_{3A} on CPU1 and τ_{6A} on CPU2.

Activations of the finished task τ_{1F} initiated before the t_{MCR} are allowed to finish their execution after the MCR. Added tasks are activated for the first time after the MCR and thus will execute only in the new mode. Each of the added tasks τ_{iA} is introduced in the system with an offset ϕ any time later than the mode change request, i.e., at $t_{MCR} + \phi$. The rest of the tasks τ_{4U}, τ_{5U}, τ_{7U} and τ_{8U} represent unchanged tasks and execute independent of the mode change.

A static priority preemptive scheduling is assumed on each processor and a priority assignment such that the lowest task numerical index indicates the higher

priority. Analysis solutions addressing other scheduling policies on the individual resources in a MPSoC represent an open issue. For simplifying the next explanations, in Fig. 12 tasks executing on the interconnect are not represented and will also not be further explicitly referred. However, the analysis method we overview next accounts for the mode change effect on the communication medium.

5.2 The Mode Change Recurrent Effect: Problem Statement and Analysis Concepts

In multi-mode real-time systems with communicating tasks as introduced above, unchanged tasks will experience different worst-case response times (WCRT) in the old and the new configuration as well as during the transition. For exemplification, we focus on the timing behavior of task τ_{4U} which has the lowest priority on CPU1. In **Fig. 13** a) we introduce WCRT diagrams, which show the WCRT of tasks as a function of time after the occurrence of the MCR. The upper diagram illustrates the transition effect on the timing behavior of task τ_{4U}. From the moment when the added tasks τ_{2A} and τ_{3A} are released on CPU1, i.e., at $t_{MCR} + \phi$, task τ_{4U} will experience additional interference and its WCRT will increase in comparison to the steady state before the MCR, i.e., $WCRT^{\tau_{4U}}_{M1} < WCRT^{\tau_{4U}}_{Transition}$. After task τ_{1F} finishes its execution corresponding to the activations released in the old mode it ceases to interfere with the lower priority local tasks, i.e., τ_{2A}, τ_{3A} and τ_{4U}. Thus, the WCRT of task τ_{4U} will decrease in comparison to the transition phase $WCRT^{\tau_{4U}}_{Transition} \geq WCRT^{\tau_{4U}}_{M2}$.

The WCRT values for all the tasks executing in the two mutual exclusive execution modes (i.e., $M1$ and $M2$) can be computed using existing analysis techniques (e.g., Refs. 25), 68)). The WCRT during the transition phase can also be computed by assuming a compound system that includes all tasks executing in both operational modes, i.e., all unchanged, finished and new tasks in $M1$ and $M2$, as it was proposed in Ref. 25). Although this constitutes a conservative approximation of the system's behaviour before, after and during the transition phase, it does not constitute a feasible approach for real-time systems that undergo multiple configuration transitions (e.g., $M1$, $M2$, ..., Mx) as the conservatism accumulates. For example, in case of a system that performs transitions from mode $M1$ to $M2$ and later from mode $M2$ to $M3$ existing schedulability analysis solutions should be applyied for the compound system comprising tasks of

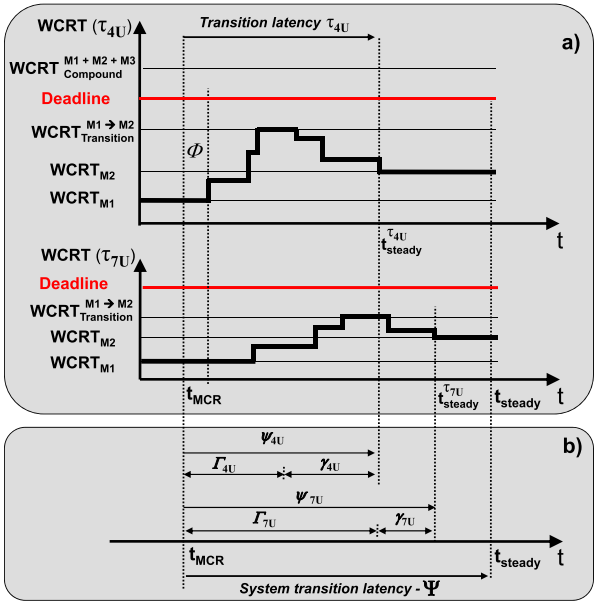

Fig. 13 a) Illustrative example of a possible settling behavior for tasks τ_{4U} and τ_{7U} in Fig. 12. b) Potential mode change time line of tasks τ_{4U} and τ_{7U} in the context of the system transition latency.

the three modes $M1$, $M2$ and $M3$. For such a case, as illustrated in Fig. 13 a), the worst-case response time $WCRT^{M1+M2+M3}_{Compound}$ of task τ_{4U} would exceed the deadline.

However, as already mentioned in Section 5.1, the execution of tasks corresponding to more than two operational modes is avoided in literature by assuming that transitions can be initiated only if the system executes in a steady state corresponding to one operational mode. In our example this means that a transition from mode $M2$ to $M3$ can be initiated only when the system executes in the steady state corresponding to mode $M2$. However, as of today, none of the existing analysis approaches is able to indicate when a real-time system that accomodates multi-mode distributed applications executes in a steady state. Thus, even if tasks' WCRTs (in each operational mode and during the transition phases between every two modes) can be determined with existing schedulability

analysis solutions, the duration of the mode change transition phases have to be computed in order to avoid the overlap of multiple mode changes (e.g., from $M1$ to $M2$ and from $M2$ to $M3$) that can cause violation of the timing constraints.

Therefore, in order to provide real-time guarantees for multi-mode real-time systems we focus next on the upper bounds derivation of the duration of the mode change transition phases.

Similar to τ_{4U}, the timing behavior of other tasks in the system changes during the transition phase and will eventually settle at a time instant t^i_{steady} at which the $WCRT_{Mode2}$, corresponding to the new mode, can be safely assumed. As illustrated in Fig. 13 a), although the MCR is assumed to be a system-wide event and thus t_{MCR} marks a global point in time, timing effects may affect different tasks in the system for a different amount of time.

The **task transition latency** ψ_i of a given task τ_i is the maximum amount of time that passes from the initiation of a mode change at t_{MCR} until t^i_{steady} when all transient effects caused by the mode change have ceased to affect the timing of this specific task.

In order to make a system-wide decision on when a new mode change may be started without the risk of overlapping with the effects of a previous mode change, we do however need to compute the **system transition latency Ψ**. This represents the maximum amount of time that passes from t_{MCR} until t_{steady} when all transient effects caused by the associated mode change have ceased to affect the timing of all the tasks in the system.

$$\Psi = \max(\psi_i \mid \forall \tau_i) \tag{1}$$

Thus, t_{steady} ($t_{steady} = t_{MCR} + \Psi$) indicates the latest point in time relative to the initiation of the mode change at t_{MCR} when the system reached the steady state corresponding to the new mode.

The challenge of computing the system transition latency Ψ can be broken down to computing the tasks transition latencies ψ_i. However, in MPSoCs where applications consist of communicating tasks, the task transition latencies ψ_i does not only depend on the tasks executing on the same processor but also on the other tasks in the system. For the multi-mode system example in Fig. 12, when the mode change is initiated, the execution of the finished task τ_{1F} may delay the execution of several jobs of task τ_{2A} activated after the MCR. This delay translates into a burst of events at the input of task τ_{2A}. The execution of task τ_{2A}, corresponding to the bursty input activation pattern, translates into a burst of events at the output of task τ_{2A} which then propagates to the input of task τ_{6A} on CPU2 and later to the input of task τ_{3A} on CPU1. Therefore, for each task in a multi-mode system, the mode change timing has a local (resource-level) and a global (system-level) aspect. The transition latency ψ_i of a task τ_i has two components, namely the *latency of the mode change effect* propagated by other tasks in the system to the resource on which τ_i is mapped, denoted with Γ_i and the *task local transition latency*, denoted with γ_i:

$$\psi_i = \gamma_i + \Gamma_i \tag{2}$$

From the perspective of task τ_{4U}, during the transition phase its jobs are delayed initially by the higher priority tasks τ_{1F} and τ_{2A}. This may lead to a burst of events at τ_{4U} output which propagates to the input of τ_{7U} on CPU3. After τ_{1F} completely finishes its execution and before the burst arrives at the input of τ_{3A}, task τ_{4U} is only delayed by the execution of task τ_{2A} which leads to a more relaxed output pattern of task τ_{4U} and therewith at the input of task τ_{7U}. When the burst of events arrives at the input of task τ_{3A}, task τ_{4U} will experience again increased interference from the higher priority tasks, which also means a possible new burst of activations at the input of τ_{7U}.

Thus, as effect of the MCR initiated at t_{MCR}, the transient overload caused by the mode change propagates recurrently through the system. For those resources (i.e., individual CPUs or buses) on which the mode change imposes a configuration change, the arrival of a mode change effect coincides with the arrival of the MCR at time t_{MCR}. However, in MPSoCs the effect of a mode change propagates between the interconnected tasks, such that there are different and eventually multiple arrivals of the mode change effect at the input of different tasks mapped on the same or different resources – e.g., in the example above the effect of the mode change will propagate twice to the input of τ_{7U} even if on its host resource (i.e., CPU3) there is no change imposed. Thus, the mode change effect latency Γ_{7U} for task τ_{7U} represents the amount of time that passes from t_{MCR} until the second burst of activations at τ_{4U} output propagates to the input of τ_{7U}. A possible mode change time line for the tasks' transition latencies ψ_i is depicted in Fig. 13 b).

The problem of computing the tasks transition latencies ψ_i maps to the computation of the parameters γ_i and Γ_i for all the tasks in the system. For this we propose a solution in the next section.

5.3 Analysis of Mode Change Transition Latencies

5.3.1 Derivation of Task Local Transition Latency γ_i

In order to derive the worst-case transition latency analysis for the mode change model in Section 5.1 we rely on concepts used in the real-time scheduling theory. When computing the maximum busy window $W_i^{\max}$ of a task τ_i executing on a single-resource (i.e., processor or bus) the maximum workload of tasks with priority greater than or equal to the priority of task τ_i has to be considered.

When a MCR imposes a configuration change on a resource such that tasks are added, removed or both, the equation for computing the maximum busy window [68] has to be adapted to consider the execution of finished and added tasks [25]. A key challenge is to identify, for each task τ_i, the worst-case scenario when the MCR shall occur such that it certainly leads to the worst-case execution during the transition phase.

In Ref. 25) it was proven that the worst-case mode change scenario for a task τ_i on a resource (e.g., processor) is obtained when t_{MCR} coincides with the activation instant of a finished higher priority task, i.e., a task in $hp_F(i)$. Additionally, the higher priority added tasks are considered to be released with an offset ϕ after the initiation of the MCR and all other unchanged higher priority local tasks are assumed released simultaneously with τ_i, i.e., in the classical critical instant. However, as there may be multiple higher priority finished tasks (tasks in $hp_F(i)$) and as for each of these tasks there may be several possible activations (i.e., jobs) released at different moments in time, one must identify all the time instances where the occurrence of the MCR_i should be assumed in order to find the worst-case transition scenario. A solution has been presented in Ref. 25).

Relying on the busy windows characteristics [25),35),68)], the maximum busy window $W_i^{\max}$ of a task τ_i in a multi-mode system represents the longest time interval required by jobs of this task to complete their execution affected by the arrival of a mode change effect.

Thus, for each arrival of a mode change effect, the local transition latency γ_i of a task τ_i is upper bounded by the task's maximum busy window computed for the configuration of tasks' input activation pattern corresponding to the mode change arrival.

$$\gamma_i \leq W_i^{\max} \tag{3}$$

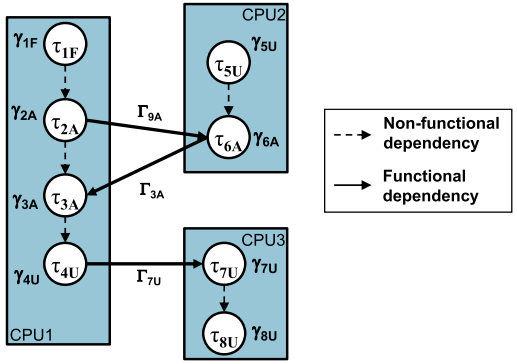

Fig. 14 Timing dependency graph for the system example in Fig. 12.

5.3.2 Derivation of the Mode Change Effect Latency Γ_i

As discussed in Section 5.2, in multi-mode MPSoCs the recurrent effect of a mode change propagates through system and affects the transition latency ψ_i of a task τ_i. In order to derive the mode change effect latency Γ_i for all tasks τ_i we integrate the local resource-level timing view into a global system-level timing view. In **Fig. 14** we introduce a *timing dependency graph* which indicates the functional and non-functional dependencies between the tasks in the example from Fig. 12.

Functional dependencies are those dependencies given through the task graph and non-functional dependencies are those which arise from the local scheduling on a processor. In this paper we consider systems for which the timing dependency graph does not contain cyclic dependencies (i.e., no backwards edges exist between the nodes of the timing dependency graph). Cyclic dependencies introduce additional challenges such as unbounded transition times and their consideration is currently an open issue.

The nodes of the graph in Fig. 14 are annotated with the values γ_i corresponding to the tasks local transition latencies. The edges which correspond to the functional dependencies between tasks are annotated with the values Γ_i. These

indicate that the effects of a mode change propagate after a time interval Γ_i to the input ports of the functionally interconnected tasks.

Thus, the activating task τ_i^p of a task τ_i (which is the immediate predecessor of task τ_i in the task graph) will propagate the mode change effect to the input of task τ_i (e.g., $\tau_{6A}^p = \tau_{2A}$). For the next explanations we denote with $T_{hep(i)}$ the set of tasks which contains task τ_i and the other local tasks with priorities higher and equal to the priority of τ_i and with $T_{hep(i)}^p$ the set of tasks which are the immediate predecessors or the activating tasks of the tasks in $T_{hep(i)}$. In the system example in Fig. 14 $T_{hep(4U)} = \{\tau_{1F}, \tau_{2A}, \tau_{3A}, \tau_{4U}\}$ and $T_{hep(4U)}^p = \{\tau_{6A}\}$.

A modification of the input activation pattern (given by the input event stream represented by the functions $\eta^+(\Delta t)$) of the tasks in $T_{hep(i)}$ will modify the local transition latency of task τ_i. Thus, the only tasks that can propagate the effect of a mode change to the input port of a task τ_i are the activating tasks of all tasks with priority higher than or equal to the priority of task τ_i, i.e., tasks in $T_{hep(i)}^p$. The mode change effect will not be further propagated to a task τ_i after the time instant when the mode change effect has ceased to affect the timing of all the tasks that can propagate this effect to task τ_i, i.e., the timing of all the tasks in $T_{hep(i)}^p$.

But, the mode change effect ceases to affect the timing of a task τ_i at the end of its transition latency ψ (cf. Section 5.2). Thus, the mode change effect latency Γ_i of a task τ_i is given by the maximum of the task transition latency ψ_j over all of the tasks that can propagate the mode change effect to the input port of a task τ_i, if any.

$$\Gamma_i = \max(\psi_j, 0), \forall \tau_j \in T_{hep(i)}^p \tag{4}$$

If for any task τ_i, there is no task $\tau_j \in T_{hep(i)}^p$, then the mode change effect does not propagate to τ_i and the mode change effect latency Γ_i is 0. However, if the set $T_{hep(i)}^p$ is not empty, the maximum of all transition latencies ψ_j indicates the latest moment in time relative to the initiation of a mode change at t_{MCR} when the timing of a task τ_j that can propagate the mode change effect ceased to be affected. After this moment in time, none of the tasks in $T_{hep(i)}^p$ will further propagate the effect to τ_i.

As can be observed from Eq. (4) and Eq. (2) there is a backwards dependency between the values Γ and γ of different tasks, which means the latency of the mode change effect Γ_i of a task τ_i depends on the local transition latency γ_j of another task τ_j which in turn may depend on the mode change latency Γ_j propagated by other tasks. However, as the timing dependency graph does not contain cycles, the tasks mode change effect latencies Γ can be computed by applying Eq. (4) and Eq. (2) for all the tasks in the system.

5.3.3 Computation of the System Transition Latency Ψ

In Section 5.3.1 we showed that the local transition latencies γ_i of any task τ_i is upper-bounded by the maximum busy window, computed under the worst-case assumptions for any input event models at the input of the local tasks with τ_i.

A major issue, when computing the tasks local transition latencies γ_i, is that the activating event models of some tasks may not initially be known - in particular when these are the effect of e.g., other tasks finishing or data arriving over a bus. This problem is addressed by embedding the local worst-case transition busy window analysis in the compositional analysis approach presented in Section 2.2, where task activating event models are provided and iteratively refined during the analysis procedure [24].

Thus, in each iteration of the global analysis procedure, during the local analysis step, scheduling analysis calculates tasks' worst-case response times (e.g., as introduced in Section 2.2). In the same step, on each resource the tasks maximum busy windows (i.e., local transition latencies γ_i) are also computed. By deriving the task transition latencies in the same step with the worst-case response times, in each iteration we perform the transition latency analysis under the worst-case assumptions. These are ensured by using the same input event models as for the worst-case response time analysis. The rest of the global analysis procedure remains the same.

The investigation of the mode change transition latencies, as described in Section 5.3.2, assumes that the compositional analysis has successfully finished, i.e., the event streams converged toward a fixed-point. After the general convergence of the event models, for all tasks on all processors the largest possible local transition latencies γ_i have been computed and can be used for computing the mode change effect latencies Γ_i. Further, by including these values in Eq. (2), for each task τ_i we obtain the maximum transition latency ψ_i. Having all the values

ψ_i, the system transition latency Ψ is obtained with Eq. (1). The dependency between the mode change effect latency Γ of a task and the local transition latencies γ of other tasks can be easily solved when the timing dependency graph does not contain cyclic dependencies. The computation of the task transition latencies ψ on each processor has to be performed top-down, starting with the highest-priority task.

6. Considering Reliability and Redundancy

As discussed in Section 1, safety regulations dictate a minimal level of dependability for safety critical functions. Therefore, most standards define a metric such as Mean Time To Failure (MTTF) in order to quantify reliability.

Since a function is generally implemented as a task graph which is mapped to computational and communication resources, those resources inherit the dependability requirements. If the intrinsic reliability is not sufficiently high, it is necessary to introduce fault-tolerance concepts which increase the reliability e.g., through parallel computation. Especially in deep submicron silicon processes there are various sources for faults, which lead to different manifestations of errors [12]. To summarize the findings in Ref. 12), we must differentiate between *transient* and *permanent* errors e.g., caused by variability, radiation and aging.

One mitigation strategy for permanent errors is to remap the task graph to resources which provide integer service. Hence, it is necessary to have backup mapping strategies for each failing hardware unit. The amount of possible mappings can be arbitrarily large, but it is sufficient to assume that not more than N components actually fail and it is feasible to validate these candidates beforehand. Practically, in case a processor fails, the processor is immediately removed from active participation and its local task set is remapped according to the backup strategy. Calculating the reliability is then trivial because the architecture can modelled as an classical *N out of M* design. This remapping is a change of the operation mode as tasks are removed and added (see Section 5).

Especially challenging for real-time systems are transient faults due to their sporadic nature. The correctness of behavior depends not only on the logical result, but especially in time-critical systems, also on the instant in time at which the result is produced [30]. So even if the system is capable of detecting and recovering from errors on the fly, it is still possible to miss deadlines caused by transient overload effects. Thus for real-time systems we must differentiate between logical correctness and timing correctness. The system is working correctly only if the platform satisfies two criteria:

(1) the result must be logically correct according to the specification
(2) the data must be delivered in time (i.e., deadlines must be met)

Vise versa, a **failure** is defined as either logical incorrectness or a timing violation.

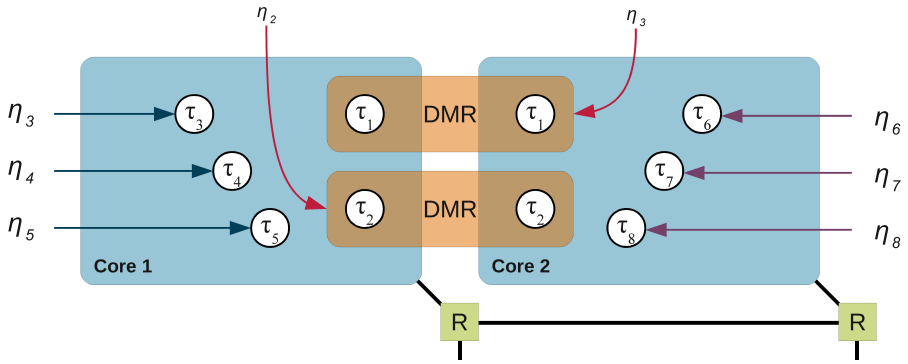

Fig. 15 Example for task level redundancy on a MPSoC.

In order to detect and recover from errors, redundancy must be introduced. But replication (e.g., DMR) of the entire MPSoC seems infeasible, due to intensive over provisioning as already highlighted in Section 1.1.

This problem can be solved by replicating only critical tasks as shown in **Fig. 15**. This particular example shows two cores which are part of a larger MPSoC. In this approach, the inherent redundant architecture of today's MPSoCs is leveraged by using spatial redundancy only on task-level and thus it is possible to cut costs and lower power consumption.

In the example shown in Fig. 15, some tasks of the task graph execute safety-critical code (τ_1, τ_2), the other represent other applications (e.g., uncritical, hard real-time or best-effort) computations. As annotated in Fig. 15, safety critical tasks are executed redundantly in a DMR fashion on both cores. Given that the operating system performs comparison operations between all external I/O issued by τ_1 and τ_2, the platform is capable of detecting logical errors for critical

tasks without massive duplication. Optionally, in case recovery is required a checkpointing / rollback technique can be used. Therefore, the state of the task is saved in a reliable fashion in regular intervals. If a recovery operation is necessary, a recent checkpoint can be used as a starting point for re-execution. The need for extended timing and reliability analysis arises as the consequence of lifting strict timing isolation compared to DMR on MPSoC level. This is caused by the recovery operations of critical and uncritical tasks in presence of errors. An analysis has to provide guaranteed timing behavior for the error-free case as well as probabilistic timing behavior in the presence of errors. First approaches are explained in the next section.

6.1 Related Work

In Refs. 14) and 15) an extension of the worst-case response time analysis was presented which incorporates re-execution. This method gives lower reliability bounds extrapolated from the critical instant. Also in Ref. 47), this work is extended to derive an optimal checkpointing strategy. However, the presented approach is based on pessimistic assumptions: The transient recovery overhead in case of error is bounded by using the largest interference possible.

If for a given distance the schedule is found feasible the probability that this minimal error distance is actually observed. In Ref. 32) the reliability of checkpointed software without considering scheduling effects is discussed. In Ref. 46) a mapping and optimization strategy for replicated and checkpointed tasks is proposed and introduces a System Failure Probability Analysis in Ref. 28). Contrary to this work, only static scheduling is supported.

A method in Ref. 7) is presented which accounts mixed criticality in real-time systems. It is assumed that task characteristics such as the WCET are annotated with a certain confidence (criticality level). The response time analysis for a particular task is then carried out in consideration of the interference based on its own criticality level. Also, no additional fault-tolerance concepts are considered in this work.

We have introduced a reliability analysis for bus based communication in Ref. 62). Here, we especially focused on the effect of retransmissions on the real-time properties of the CAN bus. This concept has recently been generalized to support multiprocessor systems in Ref. 63). Based on this work we will provide a concept of how to integrate reliability analysis in a bigger context.

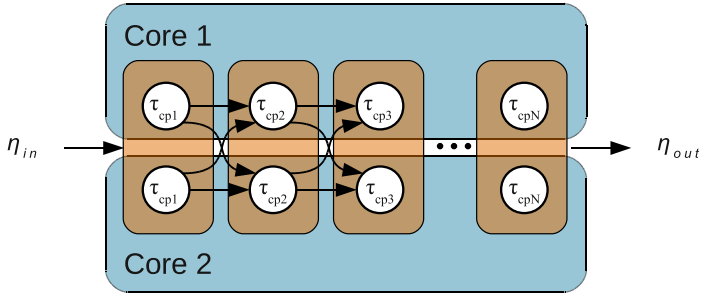

Fig. 16 Illustrative example of an equivalent task graph of a replicated task (e.g., τ_1 from Fig. 15) which is split into N checkpoints.

6.2 Modeling Fault Tolerant Tasks

In order to analyse the presented fault-tolerance concept, it is necessary to model the key aspects of task-level fault tolerance. Fault tolerant tasks (ft-tasks) are extensions of regular tasks (cf. Section 2) which are replicated among several cores to increase reliability. To model checkpointing, a fault tolerant task is split into n_i arbitrary long sub-tasks which are atomic in a sense that they contain no further interaction with other tasks or peripherals. Furthermore, each sub-task produces an intermediate result which can be fed to a voter/compare mechanism. This voting mechanism can either be implemented by the operating system or as dedicated hardware. Through this mechanism it is possible to detect errors by comparing intermediate results. A common approach to correct errors is to use temporal redundancy through checkpointing and rollback. Right before each of the n_i segments, a checkpoint of the state (e.g., register file, stack and data segment) is pushed to safe storage from which it can be reloaded in case of errors. Saving the state and restoring may impose additional execution time overhead described by parameters which we model as an increased execution time. To model this overhead we introduce the parameters t_{cov} which represents the time it takes to create a checkpoint (creation time overhead) and t_{rov}, which is the recovery overhead in case a rollback operation is performed.

For the error-free case we can model an ft-task by an equivalent system model shown in **Fig. 16**. The sub-tasks $\tau_{cp1} \ldots \tau_{cpN}$ represent the individual checkpoint

segments, in which an ft-task is split to. A sub-task pair (e.g., the τ_{cp1} pair) has to wait until the previous sub-tasks have finished their computation. We will call such a sub-task pair a *checkpoint group* referred to as $CG_{i,j,k}$. It is the set of sub-jobs in k-th stage of the equivalent task graph belonging to job $\tau_{i,j}$

By using this concept, it is also possible to model re-executions. Due to their atomicity, only sub-tasks are affected by faults and lead to re-executions. To reflect an erroneous sub-task a corresponding recovery operation (rollback) must be added to the model. We can add one dedicated rollback and re-execution task right after the affected checkpoint group. The same way, we can model two or more recovery operations for other sub-tasks. Thus, it is possible to create equivalent models for each possible error manifestation.

6.3 Error Model

For this analysis we focus on transient errors and soft errors in particular. Soft errors caused by radiation or variability will manifest as logical bit-flips in architectural registers. We assume that the fault which causes bits to flip will vanish immediately and has no temporal extent (there are no intermittent errors). Since the exact arrival of error-events is highly unpredictable, we use stochastic models to evaluate the behavior on a probabilistic base. The occurrence of errors on a core is modeled using Poisson processes with a given error-rate λ_i per core which accounts for errors affecting core components (e.g., ALU and FPU). Specifying per-core error rates models heterogeneous multi-core CPUs in which some cores may inherently be more reliable than others. This can be the case if dedicated hardware error detection and recovery mechanisms are used, such as proposed by Ref. 3) or by using more reliable but less performant silicon process parameters.

The following equations give the probability for unaffected execution of processor P_i during the time interval Δt:

$$P(\textit{no error in time } \Delta t) = e^{-\lambda_i \Delta t} \tag{5}$$

The lifetime of a task is modelled with a random variable L that represents the time the task is running until the first failure occurs. The distribution function of L is given by F_L. Throughout this paper, we will use the common notion of reliability $\mathcal{R}$ [30] as a metric which is the complement of F_L. Practically $\mathcal{R}(t)$ denotes the probability that a task is still operating without timing or logical failures in the interval $[0, t]$.

$$\mathcal{R}(t) = P(\text{no failure in interval } [0, t]) = 1 - F_L(t) \tag{6}$$

Based on $\mathcal{R}_i(t)$, we can easily calculate other common metrics such as the MTTF:

$$MTTF_i = \int_0^\infty \mathcal{R}_i(t)\, dt \tag{7}$$

6.4 Formal Analysis

For this analysis we evaluate timing constraint violations caused by recovery and re-execution and assume a perfect error detection and coverage (all errors in ft-tasks will be detected). There are two methods to derive the reliability $\mathcal{R}_i(t)$. One is to use Monte Carlo Simulation [55], in which a large amount of random experiments are carried out. By applying the law of large numbers it is possible to extrapolate from these results. In order to accumulate a sufficiently large set of representative samples, the simulation run-time can be unreasonably large. Hence, the inherent drawback of simulation based techniques is that the run-time depends on the error-rate. Alternatively, we focus on formal analysis methods which can be used to derive conservative approximations of $\mathcal{R}_i(t)$. That means that the real world $MTTF$ is equal to or larger than the predicted $MTTF'$.

To derive the reliability for each task, we use a two step algorithm. First, we identify possible error manifestations (scenario) which lead to feasible schedules. Therefore, we analyse models of error manifestations using response-time analysis methods, known from the compositional performance analysis (cf. Section 2.2). In the second step, we calculate the probability that a feasible error scenario will actually occur. We can calculate the reliability by using these probabilities.

Instead of considering a task wise approach, we consider each job individually. This way, a more precise result can be obtained by taking the interference characteristic of each job into account. For each job we enumerate scenarios which lead to a success. A job succeeds when the task meets its deadline, which will be denoted as $S_{i,j}$.

In a periodic task set, the activation pattern will appear repetitively after the hyperperiod which is the least common multiple of all periods $lcm(T_1, \ldots, T_n)$. Thus, it is sufficient to consider all activations in a hyperperiod and extrapolate from these results to future hyperperiods. The presented approach processes each

job in the hyperperiod in the order of their release.

For the analysis it is important to know which jobs can delay another job compared to the error-free case. For a fixed priority scheduler, a particular job $\tau_{i,j}$ may be directly delayed by jobs of higher priority which were released prior to $\tau_{i,j}$ on the same resource. Due to precedence constraints, a job $\tau_{i,j}$ can also be indirectly delayed by jobs running on another resource.

But, only ft-tasks may impose increased transient load in case of errors. If we recall the equivalent task graph model from Fig. 16, we see that each checkpoint group can be reexecuted arbitrarily times, depending on the actual real-world error pattern. Thus, only re-executions of checkpoint groups can delay $\tau_{i,j}$ (with respect to the error free-case), so we need to identify those checkpoint groups that can, in case of errors, interfere with the job $\tau_{i,j}$. We will call this concept *interference set*. The interference set $\xi_{i,j}$ of $\tau_{i,j}$ are exactly those other checkpoint groups, which could delay $\tau_{i,j}$. Obviously, the cardinality of an interference set is arbitrarily large, because there can be a very large amount of checkpoint groups which have been released in the past. Practically, we limit the set to a given number of checkpoint groups which have been released time d before the release of $\tau_{i,j}$, thus for large d we gain arbitrarily good estimates of $\xi_{i,j}$.

For each job, the interference set is the starting point for the scenario based feasibility analysis. An **error scenario** $s^{i,j,k} : \xi_{i,j} \rightarrow \mathbb{N}_0$ is a function which specifies a potential error situation of each checkpoint group in the interference set. Practically, $s^{i,j,k}$ maps each checkpoint group to an integer which specifies the amount of re-executions for that group. For each possible error scenario, we can construct the equivalent task graph model, evaluate the response time of $\tau_{i,j}$ and decide whether some $s^{i,j,k}$ lead to feasible systems. All scenarios which represent feasible re-execution combinations, form the working set $\mathcal{W}_{i,j}$ as depicted in **Fig. 17**. As a notation scenarios in Fig. 17, we annotated the amount of re-executions, where $(0, 1, 0)$ means zero rexecutions for the first checkpoint group, one re-execution for the second and so on. This working set can be constructively retrieved by performing a graph traversal algorithm (e.g., depth-first search) to iterate through all candidates. It must be noted that it is not required to enumerate *all* possible working scenarios, since a subset will still yield a pessimistic result. Practically, scenarios which are not enumerated are considered as non-working.

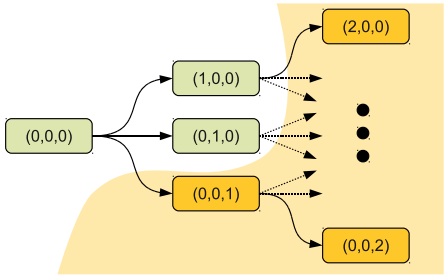

Fig. 17 Example for a working set graph. Three feasible error scenarios have been identified.

The open challenge at this point is still, how to derive the reliability for a given task $\tau_{i,j}$ from the working sets. By defining a job success probability $P[S_{i,j}]$, which is the probability that the job will meet its deadline, we can express the reliability of a task τ_i as the the probability that all jobs of τ_i have succeeded their execution that have been released in the interval $[0, t]$:

$$\mathcal{R}_i(t) = P[S_{i,1} \wedge \ldots \wedge S_{i,\eta_i^+(t)}] \tag{8}$$

Since this equation cannot be evaluated without further knowledge, we need to decompose it by using conditional success probabilities. The conditional success probability of a job $\tau_{i,j}$ is the probability that the job meets its deadline given that all jobs of the same task, released prior to $\tau_{i,j}$ succeeded.

By applying the concept of conditional success probabilities, Eq. (8) can be expressed as a product.

$$\mathcal{R}_i(t) = P[S_{i,1}] \cdot P[S_{i,2}|S_{i,1}] \cdot \ldots \cdot P[S_{i,\eta_i^+(t)}|S_{i,\eta_i^+(t)-1} \wedge \ldots \wedge S_{i,1}] \tag{9}$$

The remaining challenge is to derive the success probabilities and the conditional success probabilities used in Eq. (9). After discovering all working sets including $\mathcal{W}_{i,j}$ for $\tau_{i,j}$ we can use the initial Poisson error model to calculate the probability that one working scenario will actually happen. The probability for a re-execution of a checkpoint group can be expressed as the inverse probability of correct execution for all sub-tasks in the checkpoint group. By combining those probabilities, it is possible to get a conservative estimate of the conditional

success probabilities.

$$P[S_{i,\eta_i^+(t)}|S_{i,\eta_i^+(t)-1} \wedge \ldots \wedge S_{i,1}] = \sum_{s \in \mathcal{W}_{i,\eta_i^+(t)}} P\left[s|S_{i,\eta_i^+(t)-1} \wedge \ldots \wedge S_{i,1}\right] \quad (10)$$

The results from Eq. (10) can be inserted in Eq. (9) to retrieve the reliability for one hyperperiod $\mathcal{R}_i(t_{hyper})$. From this we can extrapolate the reliability for a given number of A hyperperiods:

$$\mathcal{R}(A \cdot t_{hyper}) = (\mathcal{R}(t_{hyper}))^A \quad (11)$$

This process has to be carried out for all tasks. By using Eq. (7), it is also possible to calculate the MTTF in order to decide whether the reliability is sufficiently high.

7. Putting It All Together

In the previous sections, we have covered the design challenges of mixed-critical MPSoCs from various angles. We have introduced a formal system model and corresponding worst-case analysis methodologies (Section 2), which has been extended to support implications of shared resource (Section 3). For the NoC, we presented efficient isolation techniques for mixed-critical applications and an extension to the system model and analysis to include the NoC communication (Section 4). We have discussed the challenges of multi-mode applications and how they can be addressed in the worst-case analysis (Section 5). Furthermore, we have shown how the reliability of mixed-critical systems can be analyzed considering the effects of redundancy mechanisms on timing (Section 6).

The presented problems and solutions are now taken as examples to explain some of the changes necessary in the design process, which are illustrated in **Fig. 18**. The specification needs new design data including timing and safety requirements which indicate the criticality levels. Error models are needed to derive reliability. Such error models are hard to get in practice but unavoidable if reliability shall be quantified.

The design phase must be extended by the introduction of safety concepts that cover static and transient errors that will become more important in the future. A suitable architecture must be found, which includes the necessary isolation

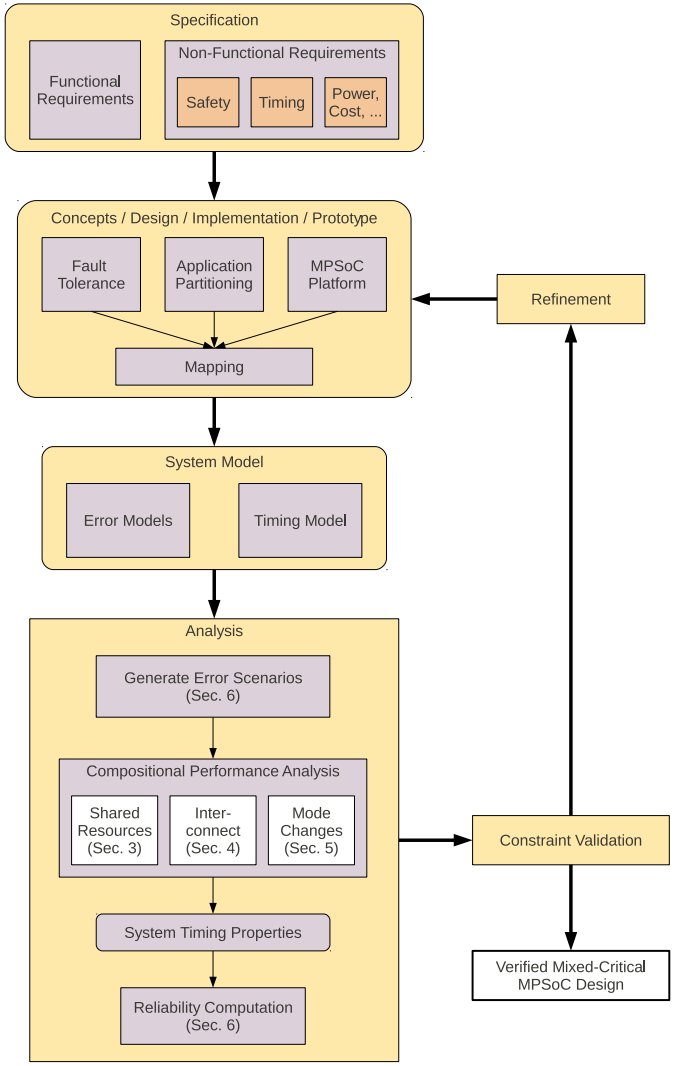

Fig. 18 Complete design flow for reliable mixed-critical systems.

support (cf. Section 2) and fault tolerance mechanisms (cf. Section 6). The NoC architecture should be treated with approaches similar to distributed automotive or aerospace designs using formal predictable scheduling to control the design process. It must however, take the different distribution of memory and shared resources into account. We gave an example for many-core architectures in Section 3.

If formal analysis is used — as suggested in this paper — an abstract timing model is needed covering execution times, shared resource utilization and task activation patterns. These data can be derived using formal WCET analysis, from execution tracing or, if no software is available yet, from estimations. In the automotive industry, such data are regularly determined, e.g., as explained in Ref. 69), while other domains still rely on simulations/execution only.

System states and mode changes can be used for system optimization and verification if available. Again, this is already applied to advanced automotive design.

8. Conclusion and Open Challenges

In this work, we have presented challenges and potential solutions of mixed-critical MPSoC designs. We categorized criticality into two aspects (timing and safety) and elaborated the challenges of mixing critical with non-critical applications. Especially, concerns of MPSoC architectural implications such the impact of shared resource contention on timing, NoC-based communication, multiple modes of operation and safety constraints have been raised. Design solutions have been presented in Sections 3, 4, 5 and 6. Finally the changes which are necessary in a typical design flow have been illustrated in Section 7. However, there are still open challenges which have to be solved in order to facilitate a structured design process for the increasing complexity of the MPSoCs.

As discussed in Section 4, the Network-on-Chip highly influences the timing of individual tasks because it is shared between applications. Hardware solutions for isolation mitigate the interference between different traffic classes. Due to the large difference in requirements and behavior between various applications and hence traffic classes (e.g., real-time vs. best-effort), these quality-of-service mechanisms must be flexible, but also low overhead, which is challenging. Furthermore, the formal analysis method used to cover the interference within a traffic class currently makes some simplifying assumptions on the router architecture (e.g., no blocking at the crossbar inputs) which imply a higher design cost (e.g., a larger crossbar). Improving the analyses to include more complex router designs is ongoing research.

As illustrated in Section 5, the effect of a mode change is recurrent and may propagate as waves through the entire system, fact that makes timing prediction for MPSoCs more difficult. A solution for deriving transition latencies in multi-mode real-time systems even in the presence of the mode change recurrent effect was presented in Section 5. However, the current analysis for multi-mode MPSoCs considers only applications without cyclic dependencies and where tasks do not share additional common resources e.g., shared memories, as considered in Section 3.

The reliability analysis as presented in Section 6 has limitations. At the moment, it uses a simple error model which does not reflect some important error types such as correlated errors like common cause errors (e.g., error in clock tree). Also, the presented approach only applies to periodic systems. Due to complex timing dependencies in case of errors, it is not easily possible to divide and conquer the problem and compose a system-wide reliability from component reliabilities. Thus, for a system-wide analysis, an error scenario has to cover the system as a whole, leading to a very large amount of error scenarios which have to be considered. Hence, it is per se a holistic approach which causes scalability problems for large task sets. The research of a compositional reliability analysis is still ongoing.

Concluding, we can say that designing mixed-critical applications is possible if all effects are accurately taken into account.

Acknowledgments This work has been funded by the “Bundesministerium für Bildung und Forschung” (BMBF), the “Deutsche Forschungsgemeinschaft” (DFG) and the Advanced Research & Technology for Embedded Intelligence and Systems (ARTEMIS) within the project ‘RECOMP’, support code 01IS10001A, agreement no. 100202.

References

1) Albers, K., Bodmann, F. and Slomka, F.: Hierarchical Event Streams and Event Dependency Graphs: A New Computational Model for Embedded Real-Time Systems, *Proc. 18th Euromicro Conference on Real-Time Systems* (*ECRTS*), Dresden, Germany, pp.97–106 (2006).
2) Andrei, A., Eles, P., Peng, Z. and Rosen, J.: Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip, *21st Intl. Conference on VLSI Design*, Hyderabad, India (2008).
3) Austin, T., Blaauw, D., Mudge, T. and Flautner, K.: Making typical silicon matter with razor, *Computer*, Vol.37, No.3, pp.57–65 (2004).
4) AUTOSAR GbR: Specification of Multi-Core OS Architecture v1.0.0 (2009), available from ⟨http://www.autosar.org/⟩.
5) Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing, *IEEE Trans. Dependable and Secure Computing*, Vol.1, No.1, pp.11–33 (2004).
6) Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A., Peri, M. and Pezzini, S.: Fault-Tolerant Platforms for Automotive Safety-Critical Applications, *Proc. Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, pp.170–177, ACM Press (2003).
7) Baruah, S., Li, H. and Stougie, L.: Towards the Design of Certifiable Mixed-criticality Systems, *Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symp* (*RTAS*), pp.13–22 (2010).
8) Bekooij, M., Moreira, O., Poplavko, P., Mesman, B., Pastrnak, M. and van Meerbergen, J.: *Proc. 8th International Workshop Software and Compilers for Embedded Systems* (*SCOPES*), *LNCS 3199*, chapter 6: Predictable Embedded Multiprocessor System Design, pp.77–91, Springer (2004).
9) Bjerregaard, T.: The MANGO Clockless Network-on-Chip: Concepts and Implementation, PhD Thesis, IMM, Danmarks Tekniske Universitet (2005).
10) Block, A., Leontyev, H., Brandenburg, B. and Anderson, J.: A Flexible Real-Time Locking Protocol for Multiprocessors, *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (*RTCSA*), Daegu, Korea, pp.47–56 (2007).
11) Bolotin, E., Cidon, I., Ginosar, R. and Kolodny, A.: QNoC: QoS Architecture and Design Process for Network on Chip, *J. Syst. Archit.*, Vol.50, No.2-3, pp.105–128 (2004).
12) Borkar, S.: Designing reliable systems from unreliable components: The challenges of transistor variability and degradation, *Micro, IEEE*, Vol.25, No.6, pp.10–16 (2005).
13) Brandenburg, B., Calandrino, J., Block, A., Leontyev, H. and Anderson, J.: Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin?, *IEEE Real-Time and Embedded Technology and Applications Symposium* (*RTAS*), St. Louis, MO, USA, pp.342–353 (2008).
14) Burns, A., Davis, R. and Punnekkat, S.: Feasibility Analysis of Fault-tolerant Real-time Task Sets, *Proc. 8th Euromicro Workshop Real-Time Systems*, pp.29–33 (1996).
15) Burns, A., Punnekkat, S., Strigini, L. and Wright, D.R.: Probabilistic Scheduling Guarantees for Fault-tolerant Real-time Systems, *Proc. Dependable Computing for Critical Applications 7*, pp.361–378 (1999).
16) Chakraborty, S., Künzli, S. and Thiele, L.: A General Framework for Analysing System Properties in Platform-based Embedded System Designs, *Design, Automation and Test in Europe Conference and Exhibition* (*DATE*), pp.190–195 (2003).
17) Diemer, J. and Ernst, R.: Back Suction: Service Guarantees for Latency-Sensitive On-Chip Networks, *The 4th ACM/IEEE International Symposium on Networks-on-Chip* (2010).
18) Diemer, J., Ernst, R. and Kauschke, M.: Efficient Throughput-Guarantees for Latency-Sensitive Networks-On-Chip, *ASP-DAC*, pp.529–534 (2010).
19) Erben, M., Guenther, W. and Sedlmeier, T.: The Impact of Automotive Standardization to Liability Risks Arising from Defective Software, Especially under European Law, *SAE World Congress* (2008).
20) Faruque, M.A.A., Weiss, G. and Henkel, J.: Bounded Arbitration Algorithm for QoS-Supported On-chip Communication, *CODES+ISSS*, New York, NY, USA, pp.76–81, ACM Press (2006).
21) Goossens, K., Dielissen, J. and Rădulescu, A.: Æthereal Network on Chip: Concepts, Architectures, and Implementations, *IEEE DESIGN & TEST*, Vol.22, No.5, pp.414–421 (2005).
22) Gopalakrishnan, S., Caccamo, M. and Sha, L.: Switch Scheduling and Network Design for Real-Time Systems, *IEEE Real-Time and Embedded Technology and Applications Symposium*, Vol.0, Los Alamitos, CA, USA, pp.289–300, IEEE Computer Society (2006).
23) Gresser, K.: An Event Model for Deadline Verification of Hard Real-Time Systems, *Proc. 5th Euromicro Workshop on Real-Time Systems*, pp.118–123 (1993).
24) Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K. and Ernst, R.: System Level Performance Analysis — The SymTA/S Approach, *IEEE Proc. Computers and Digital Techniques*, Vol.152, No.2, pp.148–166 (2005).
25) Henia, R. and Ernst, R.: Scenario Aware Analysis for Complex Event Models and Distributed Systems, *Proc. Real-Time Systems Symposium* (2007).
26) International Electrotechnical Commission (IEC): Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems (1998).
27) International Organization for Standardization (ISO): ISO/FDIS 26262: Road vehicles–Functional safety (2000).
28) Izosimov, V., Polian, I., Pop, P., Eles, P. and Peng, Z.: Analysis and Optimization

of Fault-tolerant Embedded Systems with Hardened Processors, *Proc. Conference on Design, Automation and Test in Europe*, DATE '09, 3001 Leuven, Belgium, Belgium, pp.682–687, European Design and Automation Association (2009).
29) Jonsson, B., Perathoner, S., Thiele, L. and Yi, W.: Cyclic Dependencies in Modular Performance Analysis, *Proc. 8th ACM International Conference on Embedded Software* (*EMSOFT*), New York, NY, USA, pp.179–188, ACM (2008).
30) Kopetz, H.: *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, Norwell, MA, USA (1997).
31) Kopetz, H. and Bauer, G.: The Time-Triggered Architecture, *Proc. IEEE*, Vol.91, No.1, pp.112–126 (2003).
32) Krishna, C. and Singh, A.: Reliability of checkpointed real-time systems using time redundancy, *IEEE Trans. Reliability*, Vol.42, No.3, pp.427–435 (1993).
33) Lee, J., Ng, M.C. and Asanovic, K.: Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks, *ISCA*, pp.89–100 (2008).
34) Lee, S.: Real-time wormhole channels, *Journal of Parallel and Distributed Computing*, Vol.63, No.3, pp.299–311 (2003).
35) Lehoczky, J.: Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines, *Proc. 11th Real-Time Systems Symposium*, pp.201–209 (1990).
36) Marescaux, T.M.: Mapping and Management of Communication Services on MP-SoC Platforms, PhD Thesis, Technische Universiteit Eindhoven (2007).
37) Millberg, M., Nilsson, E., Thid, R. and Jantsch, A.: Guaranteed Bandwidth using Looped Containers in Temporally Disjoint Networks within the Nostrum Network on Chip, *Design, Automation and Test in Europe*, Vol.2 (2004).
38) Negrean, M., Schliecker, S. and Ernst, R.: Response-Time Analysis of Arbitrarily Activated Tasks in Multiprocessor Systems with Shared Resources, *Proc. Design, Automation and Test in Europe Conference and Exhibition* (*DATE*), Nice, France (2009).
39) Negrean, M., Schliecker, S. and Ernst, R.: Timing Implications of Sharing Resources in Multicore Real-Time Automotive Systems, *SAE International Journal of Passenger Cars—Electronic and Electrical Systems*, Vol.3, No.1, pp.27–40 (2010).
40) Nelis, V., Goossens, J. and Andersson, B.: Two Protocols for Scheduling Multi-mode Real-Time Systems upon Identical Multiprocessor Platforms, *Proc. 21st Euromicro Conference on Real-Time Systems* (*ECRTS*), pp.151–160 (2009).
41) Nelson, V.: Fault-tolerant computing: Fundamental concepts, *Computer*, Vol.23, No.7, pp.19–25 (1990).
42) Palencia Gutierrez, J., Gutierrez Garcia, J. and Gonzalez Harbour, M.: On the schedulability analysis for distributed hard real-time systems, *Proc. 9th Euromicro Workshop on Real-Time Systems*, pp.136–143 (1997).
43) Palin, R. and Habli, I.: Assurance of Automotive Safety—A Safety Case Approach, *Proc. 29th International Conference on Computer Safety, Reliability, and Security* (*SAFECOMP'10*), pp.82–96 (2010), available from ⟨http://portal.acm.org/citation.cfm?id=1886301.1886310⟩.
44) Phan, L.T., Lee, I. and Sokolsky, O.: Compositional Analysis of Multi-Mode Systems, *22nd Euromicro Conference on Real-Time Systems* (*ECRTS*) (2010).
45) Pop, P., Eles, P. and Peng, Z.: Schedulability Analysis and Optimization for the Synthesis of Multi-cluster Distributed Embedded Systems, *Design, Automation and Test in Europe Conference and Exhibition* (*DATE*), pp.184–189 (2003).
46) Pop, P., Izosimov, V., Eles, P. and Peng, Z.: Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems With Checkpointing and Replication, *IEEE Trans. Very Large Scale Integration* (*VLSI*) *Systems*, Vol.17, No.3, pp.389–402 (2009).
47) Punnekkat, S. and Burns, A.: Analysis of checkpointing for schedulability of real-time systems, *Proc. 4th Int. Workshop Real-Time Computing Systems and Applications*, pp.198–205 (1997).
48) Radio Technical Commission for Aeronautics (RTCA): DO-254: Design Assurance Guidance for Airborne Electronic Hardware (2000).
49) Radio Technical Commission for Aeronautics (RTCA): DO-178B: Software Considerations in Airborne Systems and Equipment Certification (2001).
50) Rahmati, D., Murali, S., Benini, L., Angiolini, F., De Micheli, G. and Sarbazi-Azad, H.: A method for calculating hard QoS guarantees for Networks-on-Chip, *International Conference on Computer-Aided Design*, New York, NY, USA, pp.579–586, ACM (2009).
51) Rajkumar, R., Sha, L. and Lehoczky, J.: Real-time Synchronization Protocols for Multiprocessors, *Proc. Real-Time Systems Symposium, 1988,* pp.259–269 (1988).
52) Real, J. and Crespo, A.: Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal, *Real-Time Systems*, Vol.26, No.2, pp.161–197 (2004).
53) Richter, K., Racu, R. and Ernst, R.: Scheduling Analysis Integration for Heterogeneous Multiprocessor SoC, *24th IEEE Real-Time Systems Symposium*, pp.236–245 (2003).
54) Richter, K., Ziegenbein, D., Jersak, M. and Ernst, R.: Model Composition for Scheduling Analysis in Platform Design, *Proc. 39th Conference on Design automation* (*DAC*), ACM New York, NY, USA, pp.287–292 (2002).
55) Rubinstein, R.Y. and Kroese, D.P.: *Simulation and the Monte Carlo Method* (*Wiley Series in Probability and Statistics*), 2 edition, Wiley (2007).
56) Schliecker, S., Negrean, M. and Ernst, R.: Response Time Analysis on Multicore ECUs with Shared Resources, *IEEE Trans. Industrial Informatics*, Vol.5, No.4, pp.402–413 (2009).
57) Schliecker, S., Negrean, M. and Ernst, R.: Bounding the Shared Resource Load for the Performance Analysis of Multiprocessor Systems, *Design, Automation Test in Europe Conference Exhibition* (*DATE*), *2010*, pp.759–764 (2010).
58) Schliecker, S. and Ernst, R.: A Recursive Approach to End-To-End Path Latency Computation in Heterogeneous Multiprocessor Systems, *International Conference*

on Hardware Software Codesign and System Synthesis (*CODES-ISSS*), Grenoble, France, ACM (2009).
59) Schliecker, S., Ivers, M. and Ernst, R.: Integrated Analysis of Communicating Tasks in MpSoCs, *Proc. 3rd International Conference on Hardware Software Codesign and System Synthesis* (*CODES-ISSS*), Seoul, Korea, pp.288–293, ACM New York, NY, USA (2006).
60) Schliecker, S., Ivers, M. and Ernst, R.: Memory Access Patterns for the Analysis of MpSoCs, *IEEE North-East Workshop on Circuits and Systems*, Gatineau, Canada, pp.249–252 (2006).
61) Schliecker, S., Rox, J., Negrean, M., Richter, K., Jersak, M. and Ernst, R.: System Level Performance Analysis for Real-Time Automotive Multi-Core and Network Architectures, *IEEE Trans. Computer Aided Design*, Vol.28, No.7, pp.979–992 (2009).
62) Sebastian, M. and Ernst, R.: Reliability Analysis of Single Bus Communication with Real-Time Requirements, *Proc. 15th IEEE Pacific Rim Int. Symp. Dependable Computing PRDC '09*, pp.3–10 (2009).
63) Sebastian, M., Axer, P., Ernst, R., Feiertag, N. and Jersak, M.: Efficient Reliability and Safety Analysis for Mixed-Criticality Embedded Systems, *SAE System Level Architecture Design Tools and Methods* (2011).
64) Shi, Z. and Burns, A.: Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching, *Second ACM/IEEE International Symposium on Networks-on-Chip*, Washington, DC, USA, pp.161–170, IEEE Computer Society (2008).
65) Stein, S., Diemer, J., Ivers, M., Schliecker, S. and Ernst, R.: On the Convergence of the SymTA/S analysis, Technical report, Technische Universität Braunschweig, Germany (2008).
66) Stoimenov, N., Perathoner, S. and Thiele, L.: Reliable Mode Changes in Real-Time Systems with Fixed Priority or EDF Scheduling, *Design, Automation Test in Europe Conference Exhibition* (*DATE*), pp.99–104 (2009).
67) Tindell, K. and Clark, J.: Holistic schedulability analysis for distributed hard real-time systems, *Microprocessing and Microprogramming*, Vol.40, No.2-3, pp.117–134 (1994).
68) Tindell, K.W., Burns, A. and Wellings, A.J.: An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks, *Real-Time Systems*, Vol.6, No.2, pp.133–151 (1994).
69) Traub, M., Lauer, V., Becker, J., Jersak, M., Richter, K. and Kuehl, M.: Using Timing Analysis for Evaluating Communication Behaviour and Network Topologies in an Early Design Phase of Automotive Electric/Electronic Architectures, *SAE World Congress* (2009).
70) Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J. and Stenström, P.: The Worst-case Execution-time Problem-overview of Methods and Survey of Tools, *ACM Trans. Embed. Comput. Syst.*, Vol.7, pp.36:1–36:53 (2008).

(Received March 5, 2011)
(Released August 10, 2011)

(Invited by Editor-in-Chief: *Hidetoshi Onodera*)

Philip Axer received a Diploma degree in electrical engineering from the Technische Universität Braunschweig, Germany, in 2008. He is currently a research staff member at the Institute of Computer and Network Engineering at the Technische Universität Braunschweig, where he is working towards the Dr.-Ing. (Ph.D.) degree. His research interests include formal dependability analysis of multi- and many-core systems for mixed-critical real-time applications.

Jonas Diemer received a Diploma degree in electrical engineering from the Technische Universität Braunschweig, Germany, in 2007. He is currently a research staff member at the Institute of Computer and Network Engineering at the Technische Universität Braunschweig, where he is working towards the Dr.-Ing. (Ph.D.) degree. His research interests include multi- and many-core systems for mixed real-time applications with a focus on Networks-on-Chip.

Mircea Negrean received a Diploma degree in computer science engineering from the Politehnica University Timisoara, Timisoara, Romania, in 2004 and the Master degree in computer science from the Technische Universität Braunschweig, Braunschweig, Germany, in 2007. He is currently a research assistant with the Embedded System Design Automation Group, Institute of Computer and Network Engineering, Technische Universität Braunschweig, where he is working towards the Dr.-Ing. (Ph.D.) degree. His research activities include formal timing analysis of multiprocessor real-time systems.

Maurice Sebastian received a Diploma degree in computer science from the Technische Universität Braunschweig, Germany, in 2007. He is currently a research staff member at the Institute of Computer and Network Engineering at the Technische Universität Braunschweig, where he is working towards the Dr.-Ing. (Ph.D.) degree. His research interests include formal reliability analysis of embedded systems for mixed-critical real-time applications.

Simon Schliecker received a Diploma degree in computer and communication systems engineering from the Technische Universität Braunschweig, Germany, in 2004, where he also worked as a research assistant and Ph.D. student in various publicly and industrially funded projects until 2010. He is now working at Symtavision GmbH on system-level timing solutions for automotive.

Rolf Ernst received a Diploma degree in computer science and the Dr. Ing. degree in electrical engineering from the University of Erlangen—Nuremberg, Erlangen, Germany, in 1981 and 1987, respectively. From 1988 to 1989, he was with Bell Laboratories, Allentown, PA. Since 1990, he has been a professor of electrical engineering with the Technische Universität Braunschweig, Braunschweig, Germany. His research activities include embedded system design and design automation. Dr. Ernst is a member of the German Academy of Science and Engineering, acatech. He served as a Distinguished Lecturer for the Association for Computing Machinery's Special Interest Group on Design Automation (ACM-SIGDA).