Journal of
Low Power Electronics and Applications

MDPI

Article

Global Adaptation Controlled by an Interactive Consistency Protocol

Alina Lenz *,†,‡ and Roman Obermaisser †

Institute of Embedded Systems, University of Siegen, Adolf-Reichwein-Straße 2, 57068 Siegen, Germany; roman.obermaisser@uni-siegen.de

* Correspondence: alina.lenz@uni-siegen.de; Tel.: +49-271-740-3210

† These authors contributed equally to this work.

‡ Current address: Hoelderlinstr. 3, 57076 Siegen, Germany.

Academic Editor: Davide Patti

Received: 11 February 2017; Accepted: 25 May 2017; Published: 28 May 2017

Abstract: Static schedules for systems can lead to an inefficient usage of the resources, because the system's behavior cannot be adapted at runtime. To improve the runtime system performance in current time-triggered Multi-Processor System on Chip (MPSoC), a dynamic reaction to events is performed locally on the cores. The effects of this optimization can be increased by coordinating the changes globally. To perform such global changes, a consistent view on the system state is needed, on which to base the adaptation decisions. This paper proposes such an interactive consistency protocol with low impact on the system w.r.t. latency and overhead. We show that an energy optimizing adaptation controlled by the protocol can enable a system to save up to 43% compared to a system without adaptation.

Keywords: real-time; low power; mixed-criticality

1. Introduction

With the increased usage of mixed criticality systems (MCS), a single system on chip (SoC) can harbor an ample amount of applications at the same time. These various applications share the SoC resources, thus using the hardware at its full capacity. The MCS are popular because of their reduced size, weight and the resulting reduced manufacturing cost. The wrong implementation of the MCS can lead to failures, when the applications are not segregated properly, and low criticality applications can affect high criticality ones. To ensure the system stability, safety-critical systems use precompiled schedules to ensure spatial and timely segregation. These schedules, while vital to the system health, are unfortunately pessimistic in terms of the makespan. To compute static schedules, the scheduler has to assume the worst case for the execution time (WCET) of a process, to guarantee that it can finish within its time slot.

At runtime, the difference between the worst case and the real system state can be exploited for the benefit of the system. Using a meta-scheduler [1], a set of schedules can be computed at design time, between which the system can switch at runtime. A so-called meta-scheduler is an application that computes a directed acyclic graph (DAG) of schedules that can be reached at runtime. The DAG dictates when a schedule change can be performed and which requirements need to be observed for the change to happen. The adaptation can be performed based on all kinds of runtime phenomena, which can be quantified and observed. This finite set of values is called the $context_t$ and is defined as the system state w.r.t. the observed attributes at time t. At runtime, a mode-change state machine realizes the behavior computed by the meta-scheduler. The mode-change state machine observes the system at all instances t where, based on the context at this time $context_t$, it either changes into a new schedule or continues with the current schedule, as shown in Figure 1.

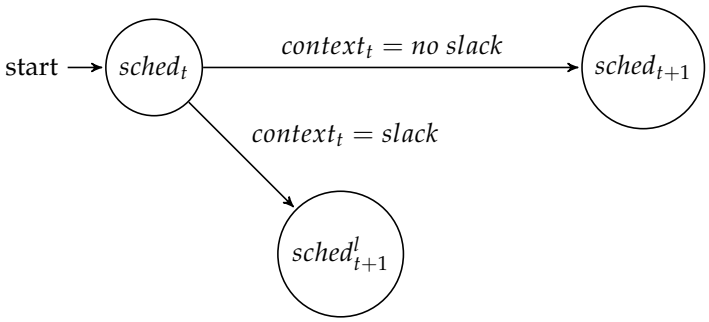

Figure 1. Each node represents a schedule. Based on the context of the system, the schedules can be changed into a low power schedule or remain the same, if an adaptation is not feasible. In this example of the mode-change state machine, the state q_t can observe two context types: either its observes slack or not. In case the context indicates slack, the schedule will be changed, and an alternative state q^{alt}_{t+1} is reached. If no slack is observed, the schedule will stay the same, but the meta-scheduler changes into state q_{t+1}, where it can observe a new instance of slack.

One example for an exploitable runtime behavior is dynamic slack, which is defined as the time difference between the WCET of a process and its real execution time at runtime. As the schedules have to be computed with the goal to provide each process enough time to finish during its slot, the WCET is usually much longer than the average execution time. As can be seen in Figure 2, the WCET assumption is especially pessimistic for highly critical applications as these applications must finish their execution at all costs. At runtime, this unaccounted time can be used to rearrange the later scheduled processes to optimize the system performance. The slack is only one example for exploitable system behavior.

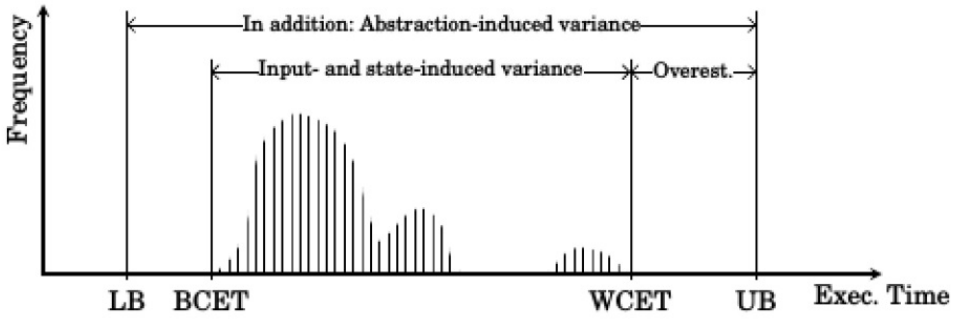

Figure 2. WCET distribution; only a few processes actually use the whole WCET time; one third already stops within the first third of the time slot. The execution time (Exec. Time) of a process depends on so many factors that definite knowledge about the best case execution time (BCET) as well as the worst case execution time (WCET) cannot be made. WCET estimations can only place a lower (LB) and upper bound (UB) on the expected execution time, which will include an overestimation [2].

Locally, these changes can be performed directly, since only the local core's context must be known to perform a change. The core is aware of its own schedule, state and future scheduled processes; therefore, the decision to switch schedules can be taken without impacting the overall system performance. However, if, for example, one wants to use the advantage of dynamic slack to start a process that is performed on another core, then one needs to assure that the other core is not busy. These changes can only be performed if the current state of all cores that are affected by the change is known. This paper proposes a broadcast protocol with low overhead and latency for this purpose.

This paper is structured into six sections. It begins with the Introduction followed by Section 2, which presents related work and points out where our contribution provides improvements. Then, Section 3 will describe the system model and the requirements. In Section 4, we will describe the interactive consistency protocol and its implementation. In Section 5, we will describe the simulation we used to verify our solution, before we draw a conclusion in Section 6 and describe the plans for future work within this field.

2. State of the Art

This section points out related work in the targeted research field. We differentiate between adaptation on the network on chip (NoC) and the establishment of global states on an SoC, as to our best knowledge, no combined approach exists.

2.1. Dynamic Adaptation on NoC

The super scheduler proposed by [3,4] is used to react to unforeseen catastrophic events. In these cases, the system needs to reach a safe state as soon as possible. The super scheduler injects new high criticality messages into the system to deal with the sudden events while maintaining the real-time deadlines. The previous high critical events of the normal application execution get downgraded and interrupted by the new events; this induces errors of up to 0.7% failure rates.

In our approach, we adapt the schedules according to a predefined plan therefore avoiding errors that result from unpredicted schedule changes. Moreover, we expand the adaptation to enable the system to adapt to a user-definable context.

2.2. Establishment of Global State

Paukovits [5] introduced the pulsed data stream for a time-triggered NoC (TTNoC). It is a stream that goes around the mesh of an NoC. Each core reads the message and relays it to the next neighbor. The transmission is effective and has a time boundary, but we wish to have a protocol that would strain the network less, as depending on the context, the state consistency could be performed frequently, e.g., to react to slack, the system state must be exchanged often.

Lodde et al. [6] have shown that the NoC is not suitable for broadcast protocols as such. They proposed to add a control network within the NoC switches: each switch is equipped with an additional set of wires, which can be used for the broadcast. They have shown that the additional network increases the performance of their coherence protocols. Our approach is similar in a way, in that we also aim to create a dedicated interconnect for the consistency protocol, as we also reached the conclusion that the NoC is ill suited for it.

Manevich et al. [7] proposed centralized adaptive routing, which reacts to the current network status. Their scheme continuously monitors the network status and adapts the routing accordingly [7]. They discuss that the NoC's small size allows establishing a global state of the NoC and, based on that, make optimized decisions. They also exchange the network state using a separate network, realized in hardware next to the normal NoC. As mentioned before, we adapt the idea of a separate network for the exchange of our context information.

3. Interactive Consistency Protocol

The interactive consistency protocol is a way to establish a globally-known state of the system. The goal of the protocol is to propagate the locally-observed core's context $context_t$ quickly and with low overhead, therefore enabling the system to establish a global knowledge of the $context_t$ without disturbing the application's communication.

3.1. System Model

We assume a system where an arbitrary amount of cores is connected via an NoC. The whole system is an MCS; therefore, each core can contain applications of varying criticality. To ensure that processes of lower criticality cannot impact processes of high criticality, the processes are temporally and spatially separated by a predefined execution schedule [8]. The schedule defines the execution start time and allocates the resources for the duration of the processes' WCET.

We assume that the applications need to exchange messages between cores via the NoC due to dependencies between the processes. This means that the local execution schedules can only be defined with knowledge regarding the message's transmission delays, which is why we schedule the communication. The communication schedules define the message injection times, meaning that a message will not be injected at the time the application sends it, but at the scheduled time. The message will be buffered within the NI until it can be sent [9]. An example of the schedule dependencies can be seen in Figure 3. In this case, the process p_1 can only start after it receives message m_1 sent by process p_0.

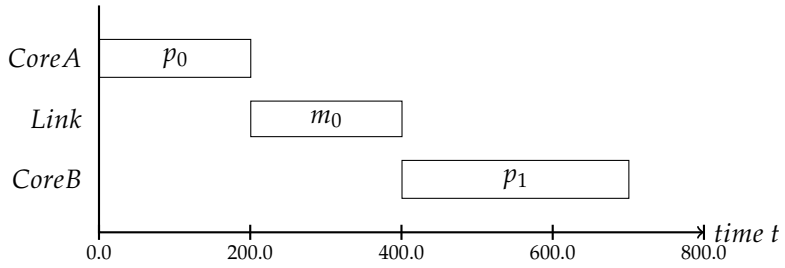

Figure 3. An example scenario where two cores are interdependent because the process p_1 is waiting for a message from process p_0 to start its computation.

As one can see, it is essential for such a system that the communication and the execution schedules are aligned with each other. Changes on one schedule impact the other one as the process could miss the message injection slot, and the message could be delayed for up to a period in the worst case. Such missing temporal alignment of the cores can lead to failures, when inputs of processes are not available in time. Sensor values are a good example of this, as they loose their validity as time passes. When a flight controller needs the sensor data to compute the next motor impulses, a delay of sensor data can lead to wrong flight maneuvers.

We can distinguish two types of adaptation in a time-triggered system:

- Local adaptation
- Global adaptation

Local adaptation optimizes the schedule of one core without affecting the rest of the system. The interface between the core's local schedule and the NoC's communication schedule is kept as it is; therefore, the message injection does not change. If for example the adaptation would react to dynamic slack, Core 1 could be clocked down to provide the message exactly at the scheduled time for the message injection.

Global optimization can use the slack to execute dependent processes in other tiles at a lower frequency and voltage, thus saving more energy. To use the slack globally, the communication schedule of the NI needs to be changed to inject the message directly into the NoC, and the local schedules of the depending cores have to be changed to start their computations earlier accordingly.

The global adaptation requires additional functionality within the NI where a state machine is implemented, which dictates for which context the schedule will be changed. A global consistent system state is essential for the synchronous and aligned switching between schedules at different cores and NIs, thereby preserving the properties of time-triggered systems such as implicit synchronization,

avoidance of contention and determinism. A state machine realized in hardware enables a fast reaction to the volatile context information. If a predicted context is observed, the NI needs to change the local communication and execution schedule. Establishing the system-wide knowledge of the global system state for adaptation, often also called consensus [10], is the goal of our work.

3.2. Requirements

To be applicable in embedded real-time systems, the consistency protocol must ensure low overhead and satisfy real-time constraints: The low overhead is essential for the real-time requirements of the system it is supposed to optimize. The usage of the protocol may in no way lead to deadline misses in the applications. The protocol execution time must therefore be kept to a minimum. If the overhead in terms of time and resources exceeds the gain from the adaptation, the protocol defeats its purpose. The fast protocol runtime is also essential in terms of context relevance. The faster the consistency of a context that can be provided, the more relevant the adaptation and its optimization will be for the system. Further, the real-time requirements must be ensured at all times; therefore, the consistency protocol has to be implemented in a way that no deadlines are missed due to the additional protocol.

3.3. Protocol Description

The protocol we propose is a broadcast protocol that distributes the local context to all cores. This way, all cores know the context of all other cores and share a unified view on the global context. The protocol is performed periodically at a rate that can be optimized for each system, because the need for adaptation varies according to the observed context. It will be realized in hardware within each NI where the context can be used by the mode-change state machine to perform eventual adaptations.

The protocol has two phases:

- collection of local context data
- propagation of local context

The protocol starts with the collection of context. The core sends its observed local context to the NI. It will collect all information until the consistency phase starts. As soon as the propagation phase starts, the assumed context state freezes for the duration of the establishment process. New context events will be collected, but not included in this execution of the protocol. They will be used in the next execution. This is shown in Figure 4 where one can see that the two protocol phases overlap. The propagation phase of a protocol execution that started at time t is performed at the same time as the context collection phase of the protocol execution that will start at $t + 1$.

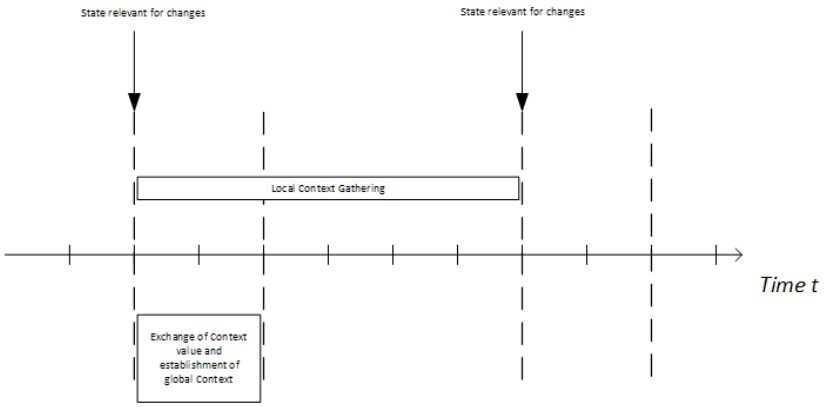

Figure 4. Protocol time-line: the protocol phases overlap, as the collection phase of the next execution starts as soon as the propagation of the current execution begins.

The collected state information is sent to all other cores. Each core concatenates the information of the other cores onto its own after receiving it, as described in Algorithm 1. Once the broadcast of local information is finished, each NI will have a string containing the global context. This global context will be sent to the mode-change state machine.

Algorithm 1: Basic consistency protocol description.

```
Input: Bitstring of local context information collected by a core
Output: Bitstring of a globally-known context
gContext = Input from core ;
if t=Period then
    send gContext to other core;
    while IncommingMsg < n;
    do
        gContext = gContext ⊕ other core's context ;
else
    Wait
```

Freedom of Interference with the Real-Time Constraints

We ensure the timeliness of the applications by scheduling the communication and computations w.r.t. their time constraints. State of the art schedulers [11,12] generate such a schedule using two models: the application and the physical model. The application model describes in detail the inter-process relationships, precedence constraints and deadlines. This model will be mapped by the scheduler on the physical model, which describes the hardware, topology and interconnection. Only schedules where processes and messages defined in the application model can be executed under their defined constraints are valid outputs of the scheduler. The interactive consistency protocol provides additional messages that need to be considered in the application model. The resulting schedule then ensures the timeliness of the application while at the same time offering communication slots for the interactive consistency protocol. If a feasible schedule is found by the scheduler at design time, no impact of the interactive consistency protocol on the delays of application message can occur at runtime. However, the additional messages can make the search for a feasible schedule harder. This drawback is avoided by a dedicated interconnect for the agreement in this paper.

3.4. Timing Model for State Establishment

The timing is crucial for the consistency protocol. The context should be known to all cores as soon as possible to maximize the potential efficiency improvements. If for example the system should adapt to slack times, the information must be used as soon as possible, because the occurrence of slack is highly volatile. The consistency protocol must therefore be finished within a fraction of the slack time. The protocol is basically a discrete sampling of the system state where the frequency defines the accuracy of the state image.

In the following, three different protocol frequencies within a period are shown: the first one establishes the system state once per period, the second one three times per period and the last one seven times, as shown in Figures 5–7. These varying frequencies can be useful for different kinds of applications. The first type in Figure 5 is applicable to systems where the application's behavior is mostly static and the context is mainly defined by slowly changing attributes like battery level or environmental temperature. The second type in Figure 6 and the third type in Figure 7 can be chosen based on the context type. The more often it is expected that the observed state is changing, the more frequently the consistency protocol is needed.

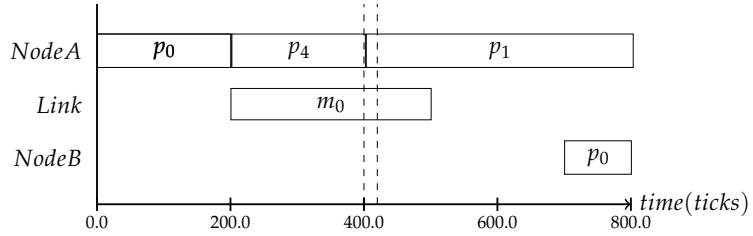

Figure 5. The protocol can be executed up to the systems needs in a frequent or less frequent way.

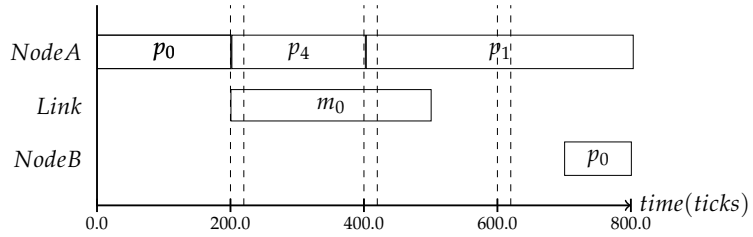

Figure 6. More frequent execution of the protocol.

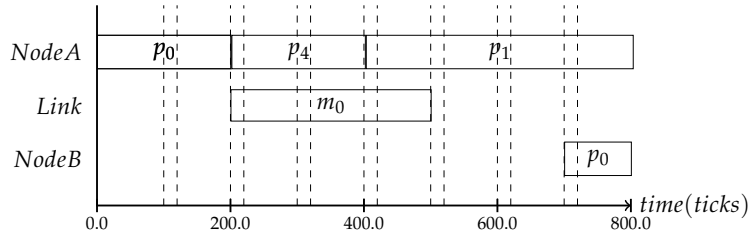

Figure 7. The more often the protocol is executed, the more information about the system state can be inferred.

The frequency further also dictates at which instances the schedule can be changed: if only one state consistency establishment is performed during a period, only one schedule change can be done. The scheduler therefore has to be aligned with the interactive consistency protocol, meaning that the meta-scheduler has to define changes for the same time steps as they will be monitored at runtime. The respective mode-change state machine must also be based on the same time steps as are used in the consistency protocol. If done correctly, possible timing inconsistencies are avoided. For example, a process could finish at a time t within a period, and the scheduler would predict that the slack can be used for adaptation and would mark this as a trigger for change. If the protocol evaluates the trigger at a slight offset, valuable time can be lost, and the decision on a change cannot be taken precisely anymore.

Therefore, it is also crucial to perform the consistency protocol at time instances where the context change can be detected. For example, in the context of slack, the trigger most likely occurs, as was mentioned in Section 1, within the second half of the process's time slot. If one would use a frequency as shown in Figure 8, where the dashed line indicates the protocol interval, the slack cannot be monitored, because the protocol is only performed at the end of the time slot with the duration of the assumed WCET. Even if the process ended earlier, the system would have no way of noticing that or reacting to it.

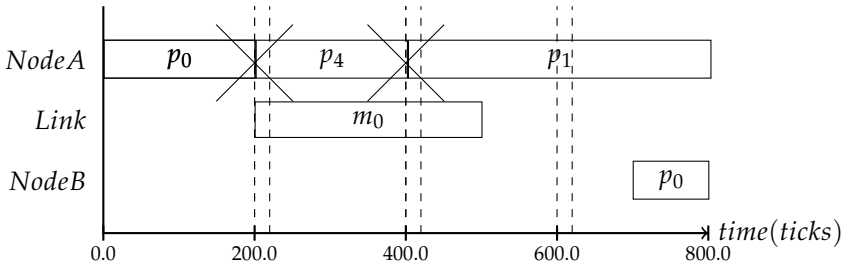

Figure 8. The timing for the state consistency depends on the context one likes to observe. In this case, the slack could not be detected, as the protocol was executed right after the WCET.

Simply performing the protocol on top of a running schedule will lead to undefined changes, as the granularity of the scheduled context change protocol can be different than the time granularity of the mode-change state machine. If a mode-change is performed based on the context gathered in the time interval $[t_0, t_1]$, the adaptation trigger can contain all context triggers that occurred until t_1. If the adaptation frequency is off by only 10 clock cycles, as can be seen in Figure 9 where the red line marks the adaptation scheduling frequency and the black line the consistency protocol interval, the adaptation will never be performed, as no input is known to the mode-change state machine. The consistency protocol must provide the adaptation with the system state whenever a schedule change can be performed. Due to the runtime of the consistency protocol, the adaptation for the system state t_0 is performed at the tick $t_1 + t_{consistency}$.

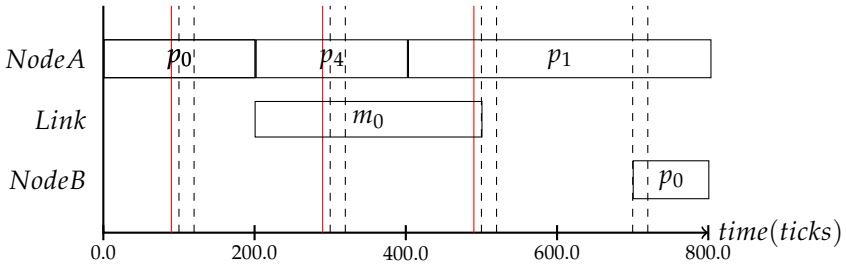

Figure 9. The mode-change state machine expects a context value that cannot be provided by the consistency protocol, because they are running out of sync. This would cause the adaptation to fail.

4. Implementation

This section describes the implementation of the protocol. We present two approaches to implement the protocol on a system with a mesh-NoC, which is the most common NoC topology due to its low manufacturing cost [13]:

1. We use the available NoC links to propagate the current system state
2. We add dedicated links to the architecture to perform the protocol

4.1. NoC-Based Approach

The consistency protocol occupies the network during its execution for a time frame of $m + n + 1$ hops to exchange the local context state via the communication network. This means that the usual NoC traffic must be scheduled to avoid any conflict with the protocol messages. Incorporating the protocol's messages into the application model of the scheduler guarantees the system's real-time requirements with the resulting schedule.

Since most NoCs do not provide broadcast algorithms, we needed to formulate a broadcast protocol for the state consistency. Our proposed protocol is round-based to reduce the time needed for

the propagation. In the first round, which starts right at the beginning of the interval, we let every core send its local information to all neighbors. Then, all cores wait till they received the information from all neighbors and concatenate the information to a new intermediate context string. Once all neighbors' information is gathered, the intermediate context will be sent to all neighbors again, as shown in Algorithm 2. Figure 10 shows how the local context bits expand through the network. In the top left corner, you can see the first stage of the protocol with each core knowing only its own local context. In the second step, shown in the top right corner, the transmission of the local context is performed in the direction of the arrows; leading to the intermediate state shown in the bottom left corner, where the cores are aware of their neighbors' context, as well. After performing another transmission of the complete locally known context, all cores know the context values of all other cores. This will be repeated for $n + m + 1$ times with n and m being the number of rows and columns in the network, because this is the amount of hops it takes for one edge to reach the opposite edge.

Algorithm 2: NoC consistency protocol description.

```
Input: Bitstring of local context information collected by a core
Output: Bitstring of a globally known context
gContext = Input from core ;
if t=Period then
    send gContext to neighbor;
    while IncommingMsg < n;
    do
        gContext = gContext ⊕ NeighborContext ;
        send gContext to all neighbors;
else
    Collect Status Data
```

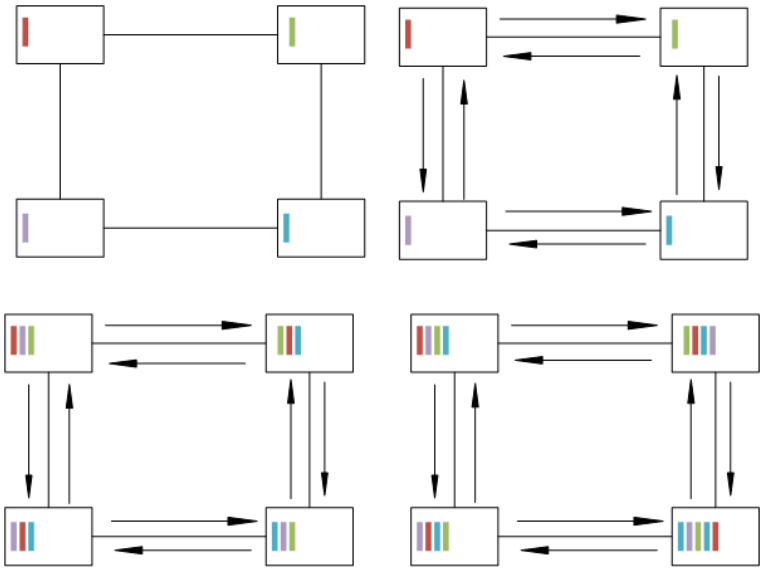

Figure 10. Propagation on NoC: the four cores are connection via the NoC, represented by the lines. The colored sticks show how the local context is traversing through the network for each propagation step.

4.2. FIFO Approach

An alternative to the usage of the NoC for the consistency protocol is the use of dedicated links between the NIs serving as FIFOs. These additional links will not only transmit the messages faster, because only one send operation will be performed, instead of various transmissions between the NoC's routers, but they also do not influence the scheduling problem of the NoC. To access the links, two buffers are added in the NI; one buffer is the sending buffer where the agreement layer will place the local context to send it to the next neighbor, while the other buffer is the receiving buffer for the other neighbor's local context, as seen in Figure 11.

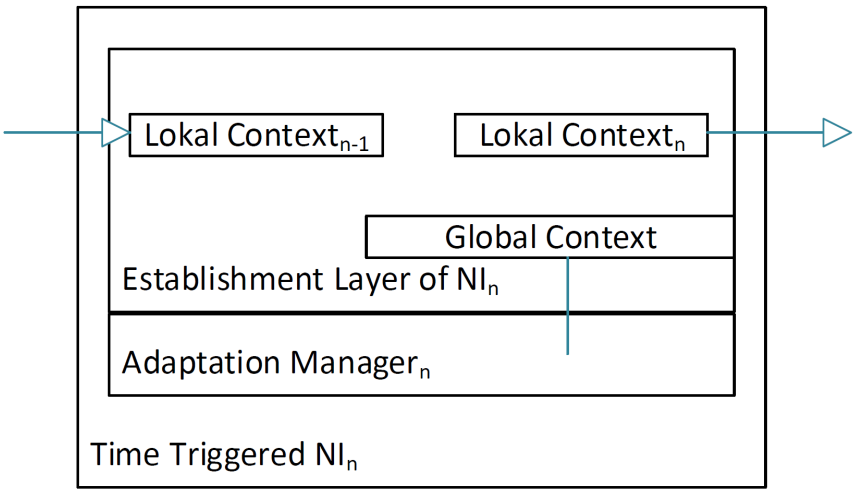

Figure 11. Hardware block of the establishment Layer. The layer is located within the NI and has dedicated connections to its direct neighbors in the dedicated FIFO ring. As the communication channels are unidirectional, the layer can only actively communicate to its right neighbor and receive information from its left neighbor. After establishing the global context, the layer has a direct link into the adaptation manager where the local NI's schedule changes will be performed by the mode change state, machine taking the global context as an input.

This means that the consistency protocol can be performed without restrictions for the regular communication. In order to minimize the additional wires, we only use a uni-directional link between the NIs that connect them in a ring shape. This way, the propagation is performed in a ring: at the beginning, each NI simultaneously sends its local context to its right neighbor and waits for the left neighbor's context to arrive, as seen in Figure 12. Once it received the neighbor's message, it concatenates the new information to its local knowledge and relays it to its right neighbor. The protocol can stop as soon as the core receives its own original information from the left neighbor, as described in Algorithm 3. The whole protocol duration increases compared to the NoC version, as each message has to pass the ring once, meaning that it has to make n hops with n being the amount of nodes in the network. However, since the transmission on the dedicated link takes only one tick, the overall duration is typically still smaller than the transmission over the NoC.

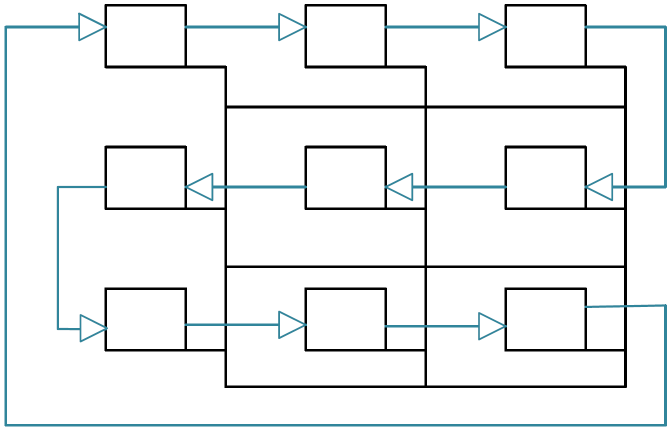

Figure 12. Propagation on FIFO; the four cores are connected via the NoC, represented by the normal line, and a directional link represented by the arrows. The colored sticks show how the local context is traversing through the network for each propagation step. In this scheme, the propagation step is equal to one tick.

Algorithm 3: FIFO consistency protocol description.

```
Input: Bitstring of local context information collected by a core
Output: Bitstring of a globally known context
gContext = Input from core ;
if t=Period then
    send gContext to right neighbor;
    while IncommingMsg < n;
    do
        gContext = gContext ⊕ NeighborContext ;
        send gContext to right neighbor;
else
    Collect Status Data
```

5. Simulation

This section focuses on the simulations that were made to evaluate the two variants of the consistency protocol. We have simulated both implementations in an already existing framework realized in gem5 [14]. The existing framework provided functionalities to test time-triggered message injection based on schedules [9], and it was extended with an adaptation unit as shown in Figure 13. The simulated NI consists of a core interface where messages are enqueued from the core. When the schedule defines a message to be sent, the egress bridging unit reads the ports and relays the message to the priority queue, where the message is enqueued in the corresponding send channels. From these channels, the serialization unit reads the message and packetizes it for the transmission. The adaptation manager gets the global context from the core interface and, based on its value, pushes the new injection schedule into the egress bridging unit, as well as into the serialization unit. In the figure, the letters D and E stand for de- and enqueue; A represent the adaptation signal, which triggers changes; and S represents the signal, which informs about slack occurrence.

Our interactive consistency protocol completes the setup with the propagation of the local values. We expanded the hardware model with an additional hardware block dedicated to the consistency protocol. The additional block injects the context messages into the normal communication. The message injection is performed based on the time-triggered schedule, which has the protocol

intervals included. In the FIFO version, the NoC remains unchanged, as the additional FIFOs that were implemented between the NIs perform their duty decoupled from the rest of the system.

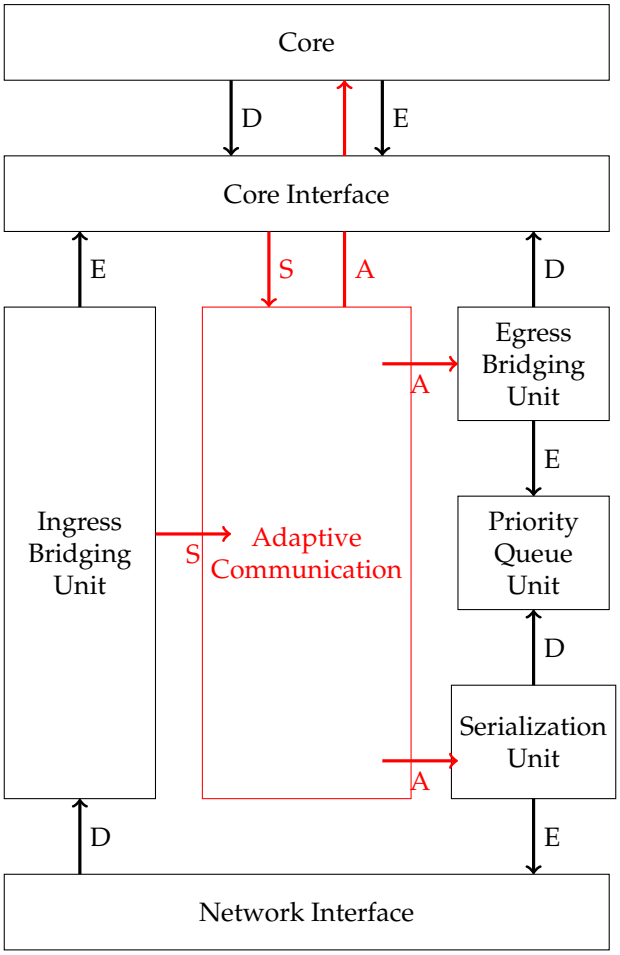

Figure 13. The architecture of an NI.

Our goal of the simulation is to show that the additional overhead created by the consistency protocol will not diminish the effect of the adaptation. In the simulation, adaptation is used to save energy by dynamically changing the system schedule and applying power gating on the cores when they finish a computation. As a result, we designed our power model to include idle and running states of the cores. For the core energy usage, we computed the difference of idle and running states as published by [15].

The simulation is evaluated based on the assumed energy consumption at runtime. In the version we used, Gem5 does not support power models; therefore, we needed to design a customized model to measure the energy consumption in the NoC. This power model approximates the consumed power based on the behavior of CMOS-circuits, by deriving a typical energy consumption per tick. A tick in gem5 equals $10n$ s, based on the transmission duration between two routers. Therefore, we assumed that we have 10^8 ticks per second. To compute an upper bound for the per tick energy consumption of an FPGA as described in [16], we took the maximum power value of 30 Watts and divided it by the ticks per second, giving us the per tick energy consumption of 3×10^{-7} Watts. To monitor the simulated consumption, we add this value for each tick in which a component is running.

To compute the overall energy consumption, the associated core energy usage is added to the global energy counter each tick, as well. At the end of the simulation, the energy counter is shown. The traces that were used on the core are created by randomly scaling the WCET to simulate the unpredictable core behavior. For the results, we have created traces with low slack and high slack.

We compare our results to a baseline configuration. In this configuration, we let the core run in idle mode until the WCET is finished.

5.1. Use-Case

We used two use cases to evaluate the runtime behavior of our protocol: a synthetic use case and an open source benchmark.

5.1.1. Synthetic Use Case

The implemented use case assumes two nodes connected in a 4 × 4 mesh architecture with 16 cores connected to it. We used x-y routing to determine the message paths. There are seven applications consisting of a set of processes, p_0–p_7. The assumed processes are located on seven of the 16 cores some need to receive a message from another process before they can start their execution: p_2 depends on p_0; p_3 depends on p_1; p_7 depends on p_5, which in turn depends on p_2. All processes are high criticality processes. The corresponding schedule can be seen in Figure 14. The goal of our simulation is to execute the defined schedule with randomly-selected slack times. The slack times will be made globally known by the context consistency protocol, which in turn will trigger the adaptation process into a low power schedule where either the upcoming process will be executed at a lower frequency if possible or the process will be put into idle mode. The scheduling decisions were made a priori in the meta-scheduler, where the subset of reachable schedules was generated. The simulation shows that the interactive consistency protocol helps to save energy [1].

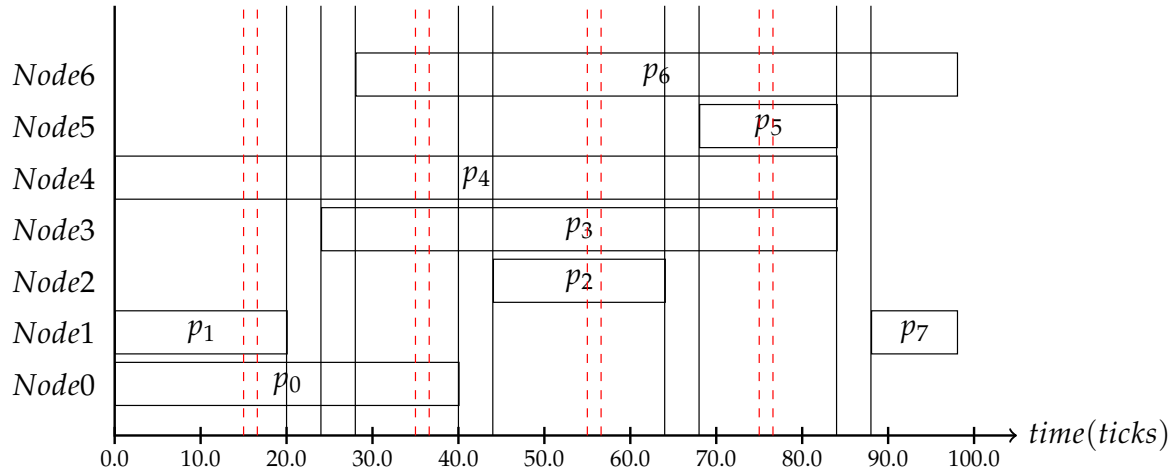

Figure 14. The use case schedule for the NoC implementation with context intervals scheduled within.

5.1.2. Results

In the synthetic use case, because all activities are executed with a frequency of 100% when the adaptive communication is disabled, the consumed energy amounts to 29.4 W. Figure 15 shows the energy saving that can be obtained with both implementations.

In the figure, one can see that using the NoC for the protocol even with the additional context consistency, the adaptation enables power savings of up to 8.26 W. Using the dedicated links, the total energy that could be preserved can be increased to 12.96 W. The increased energy savings in the FIFO version are a result of the more flexible scheduling of the consistency protocol interval as the FIFO version must not consider the on chip communication for its execution. This way, the consistency intervals can be scheduled more often, thus increasing the sampling rate of the system behavior. The difference between the implementation is marginal, but it already shows the main disadvantage of an NoC implementation. With increasing on chip communication, the scheduling complexity increases and fewer protocol instances can be placed within a period.

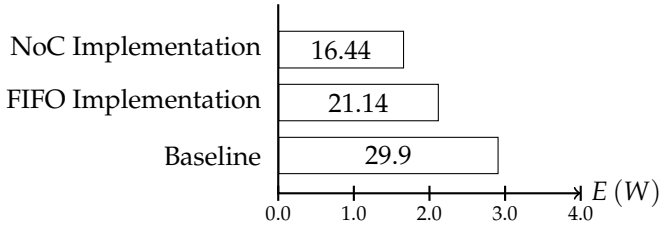

Figure 15. Energy consumption of the whole on-chip system per second in the synthetic use case simulation. The baseline energy consumption is the consumption without any adaptation. The FIFO implementation shows the consumption when the consistency protocol is executed on a dedicated network; the NoC Implementation shows the consumption when the network is shared by the protocol instances and the normal communication. The slightly increased energy savings in the FIFO implementation are the result of the more flexible scheduling of the protocol instances.

5.1.3. Benchmark

To verify that the execution of the additional protocol does not negatively impact the timeliness and to evaluate the efficiency improvements under real and overload conditions, we tested the protocol with a benchmark. As a system architecture, we chose the same 4×4 NoC with 16 cores. As our system assumes a time-triggered message injection based on static communication schedules, we had to schedule the benchmark. For this reason, we chose the NoC traffic pattern as the source by [17] as the library provides a detailed description of the inter-process dependencies and the message sizes that we needed to successfully schedule the traffic onto our simulation platform. This library provides accurate values as it depicts recorded real traffic patterns showing the observed behaviors of the task while executed on processors within a 4×4 NoC. We could therefore simulate real execution times and therefore real slack values. The library provided a set of applications for different types of topologies and NoC sizes. We chose the random sparse matrix solver for electronic circuit simulations on a 4×4 mesh NoC.

This application contains 96 tasks with 67 communication links. The tasks were mapped to a DAG describing the applications dependencies. We expanded the application model with the consistency messages and scheduled the application. The 96 processes were mapped to the 16 cores, as shown in Table 1.

Table 1. Mapping of the processes to the cores.

Core	Process
(0,0)	0, 27, 49, 85, 92
(0,1)	1, 24, 42, 65, 58
(0,2)	2, 28, 43, 69, 53, 94, 91
(0,3)	3, 29, 53, 95, 55, 61
(1,0)	4, 25, 39, 66, 63, 79
(0,1)	5, 30, 51, 12, 41, 93, 86
(0,2)	6, 31, 44, 71, 59, 77, 88, 90
(0,3)	7, 32, 45, 72, 47
(2,0)	8, 33, 52, 73, 13, 89
(0,1)	9, 34, 56, 76, 64
(0,2)	10, 35, 57, 76, 64
(0,3)	11, 26, 46, 83, 23, 74
(3,0)	14, 36, 48, 67, 70, 75
(0,1)	15, 37, 60, 16, 54
(0,2)	17, 21, 40, 68, 18, 87, 22
(0,3)	19, 38, 62, 84, 20

We increased the scheduling problem by adding the interactive consistency protocol's messages to the regular application messages. The resulting schedule described a system execution where the time slots for the establishment protocol intervals were scheduled within the normal communication schedule, while ensuring the real-time constraints set by the application. We used the same 4×4 mesh NoC as for the synthetic use case to execute the traffic of the random sparse matrix solver for electronic circuit simulations. As a baseline, we executed the benchmark traffic without any protocol instances and without adaptation within the NI. As shown in Figure 16 we observed that both implementation reduced the energy usage of the system. The adaptation enables the system to execute the processes more quickly, and cores could be powered down in the slack times, therefore reaching savings of 14% for protocol instances using the NoC for the establishment and 21% for instances using the FIFO.

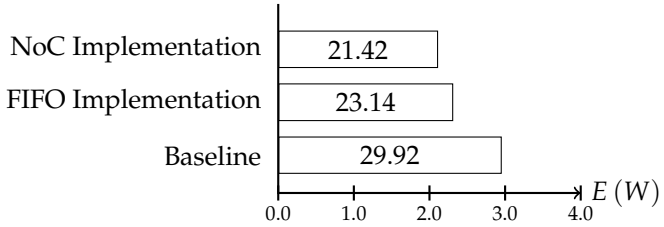

Figure 16. Energy consumption of the whole on-chip system per second in the benchmark simulation. The baseline energy consumption sums up to 29.92 W, while the NoC version uses 23.14 W, and the FIFO version consumes 21.42 W.

5.2. Scalability

To analyze the scalability, we inspected how increasing the network size will affect the networks utilization. Adding the consistency protocol instances, we refer to the instantiation of the protocol based on a certain number of cores, context values and topology as protocol instances, expanding the application model by the context messages. If a certain amount of protocol instances must be scheduled within a period, these additional time frames need to be scheduled alongside the normal communication. Ideally, the protocol instances can be scheduled in parallel to the core execution times, thus hiding the impact of the additional message transmissions in time slices where the network is not used. However, the new schedule could lead to the communication becoming a bottleneck, not allowing the full utilization of the cores. We assume that the constant wct_n depicts the worst case transmission time of a message in an NoC with nnodes and size returning the length of a message in words. We will in the formula refer to the interactive consistency protocol as ICP and assume that the protocol will be executed k times per period. With the additional messages, the NoC utilization of an NoC with n cores will increase in the following way:

$$utilization_n = \frac{\sum_{i \in Msg} wct_n \cdot size(i) + \sum_{i \in k} wct_{ICP} \cdot size(msg_{ICP})}{Period} \tag{1}$$

The upper part of the fraction can be rephrased as the following, if we assume the worst case transmission time for each message, which for x-y routing means $2\sqrt{n} - 2$, based on the Manhattan metric:

$$Period \cdot utilization_n = wct_n \cdot \sum_{i \in Mgs} size(i) + k \cdot (wct_n + 1) \cdot msg_{IPC} \tag{2}$$

We then analyze the runtime behavior:

$$\underbrace{Period}_{constant} \cdot \underbrace{utilization_n}_{O(n\sqrt{n})} = \underbrace{wct_n}_{O(\sqrt{n})} \cdot \underbrace{\sum_{i \in Mgs} size(i)}_{constant} + \underbrace{k}_{constant} \cdot \underbrace{(wct_n + 1)}_{O(\sqrt{n})} \cdot \underbrace{size(msg_{IPC})}_{O(n)}$$

As one can see, the consistency protocol increases the usage growth of the network by a factor of $O(n)$, while the normal communication utilization would grow at a pace of $O(\sqrt{n})$, as the protocol execution time is increasing more quickly than the worst-case communication time. This growth puts restrictions for the scheduling of the application messages as it takes up more time from the schedulable execution period. This problem only occurs in the shared network implementation as the additional messages generated by interactive consistency protocol increase the scheduler's application model and make it harder to find feasible schedules. With increasing message sizes, the amount of possible protocol instances can be lowered to compensate for the increased execution time, which would however lower the reaction time to context events. When using the NoC for the consistency protocol for big NoCs, a hierarchical model could be used where the domains to establish a context consistency are small and the overhead created by the messages is kept low.

Scaling the network size in the dedicated implementation type will linearly increase the protocol runtime as the path around the ring will increase with each new core. This increased protocol execution time can be avoided by changing the topology into a higher intertwined topology like the torus. Other than the protocol duration, the increased network size will not affect the utilization of the NoC, as the dedicated network leads to a separation of concern where the context messages do not imply any restrictions on the application model and need not be scheduled. Changes in the network size and the inherently longer context messages of larger systems do not need to be scheduled at the NoC. In the end, the scalability in increasingly big networks will be a trade-off between resources for the links and the latency of the interactive consistency protocol.

6. Conclusions

In this paper, we proposed a protocol to establish a global state within an NoC. We have shown that the additional propagation time of the state can be bounded by a maximum duration time within a mesh-NoC, and the effect on the system does not negatively impact the possible energy saving.

The protocol is a broadcast protocol, which can be implemented in different ways, where both of them have respective advantages and disadvantages: one of which would need additional links in the MPSoC, which leads to additional fault possibilities, while the other expands the scheduling problem, making it unfeasible to find a usable adaptation schedule to begin with. Since we focus our work on safety-critical real-time systems, the real-time requirement must be ensured at all times. Therefore, the NoC implementation is not adaptable.

In the simulation, we did not take into account the impact that the additional hardware has on the energy consumption, as we used an abstract energy model. Our goal was to have an initial proof of concept. Having the initial idea verified, the next steps will be to evaluate the energy savings on a physical platform. In future work, we also plan to implement a fault tolerant version of the protocol, by introducing redundant links and a voting mechanism. In a further step, we also aim to optimize the agreement frequency. By finding the factors that determine how frequent a system needs to be observed to be able to apply improvements, we can keep the protocol overhead minimal. Finally we plan to explore the scalability of the protocol onto a distributed system.

Acknowledgments: This work was supported partly within the scope of the European project SAFEPOWER-Safe and secure mixed-criticality systems with low power requirements , which has received funding from the European Union's Horizon 2020 research and innovation program under Grant Agreement No. 687902 and partly within the European Project DREAMS-Distributed REal-time Architecture for Mixed Criticality Systems under the FP7 Grant Agreement No: 610640.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

MCS	Mixed Criticality Systems
TTNoC	Time Triggered Network on Chip
MPSoC	Multi-Processor System on Chip
NoC	Network on Chip
SoC	System on Chip
NI	Network Interface
FIFO	First In First Out
WCET	Worst Case Execution Time

References

1. Lenz, A.; Pieper, T.; Obermaisser, R. Global Adaptation for Energy Efficiency in Multicore Architectures. In Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Saint Petersburg, Russia, 6–8 March 2017.
2. Axer, P.; Ernst, R.; Falk, H.; Girault, A.; Grund, D.; Guan, N.; Jonsson, B.; Marwedel, P.; Reineke, J.; Rochange, C.; et al. Building timing predictable embedded systems. In *ACM Transactions on Embedded Computing Systems (TECS)*; ACM: New York, NY, USA, 2014; Volume 13, p. 82.
3. Ogrenci Memik, S.; Bozorgzadeh, E.; Kastner, R.; Sarrafzadeh, M. A super-scheduler for embedded reconfigurable systems. In Proceedings of the IEEE/ACM International Conference on Computer Aided Design ICCAD, San Jose, CA, USA, 4–8 November 2001; pp. 391–394.
4. Persya, A.C.; Gopalakrishnan Nair, T.R. Model based design of super schedulers managing catastrophic scenario in hard real time systems. In Proceedings of the IEEE 2013 International Conference on Information Communication and Embedded Systems (ICICES), Tamilnadu, India, 21–23 Febuary 2013.
5. Paukovits, C.; Kopetz, H. Concepts of Switching in the Time-Triggered Network-on-Chip. In Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Kaohsiung, Taiwan, 25–27 Augest 2008; pp. 120–129.
6. Lodde, M.; Flich, J.; Acacio, M.E. Heterogeneous NoC Design for Efficient Broadcast-based Coherence Protocol Support. In Proceedings of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip, Lyngby, Denmark, 9–11 May 2012; pp. 59–66.
7. Manevich, R.; Cidon, I.; Kolodny, A.; Walter, I.; Wimer, S. A Cost Effective Centralized Adaptive Routing for Networks-on-Chip. In Proceedings of the 14th Euromicro Conference on Digital System Design, Oulu, Finland, 31 Augest–2 September 2011; pp. 39–46.
8. Obermaisser, R.; El Salloum, C.; Huber, B.; Kopetz, H. The time-triggered system-on-a-chip architecture. In Proceedings of the ISIE 2008: IEEE International Symposium on Industrial Electronics, Cambridge, UK, 30 June–2 July 2008; pp. 1941–1947.
9. Ahmadian, H.; Obermaisser, R. Time-Triggered Extension Layer for On-Chip Network Interfaces in Mixed-Criticality Systems. In Proceedings of the Euromicro Conference on Digital System Design (DSD), Funchal, Portugal, 26–28 Augest 2015; pp. 693–699.
10. Hadzilacos, V.; Toueg, S. A modular approach to fault-tolerant broadcasts and related problems. In *Distributed Systems*, 2nd ed.; ACM Press/Addison-Wesley Publishing Co.: New York, NY, USA, 1993; pp. 97–145.
11. Obermaisser, R.; Murshed, A. Incremental, Distributed, and Concurrent Scheduling in Systems-of-Systems with Real-Time Requirements. In Proceedings of the 2015 IEEE International Conference on Computer and Information Technology, Ubiquitous Computing and Communications, Dependable, Autonomic and Secure Computing, Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), Liverpool, UK, 26–28 October 2015; pp. 1918–1927.
12. Schöler, C.; Krenz-Baath, R.; Murshed, A.; Obermaisser, R. Computing optimal communication schedules for time-triggered networks using an SMT solver. In Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems (SIES), Krakow, Poland, 23–25 May 2016; pp. 1–9.
13. Holsmark, R.; Palesi, M.; Kumar, S. Deadlock free routing algorithms for irregular mesh topology NoC systems with rectangular regions. *J. Syst. Arch.* **2008**, *54*, 427–440.

14. Binkert, N.; Beckmann, B.; Black, G.; Reinhardt, S.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D.; Krishna, T.; Sardashti, S.; et al. The gem5 simulator. *ACM SIGARCH Comput. Architect. News* **2011**, *39*, 1–7.
15. Goergen, R.; Graham, D.; Gruettner, K.; Lapides, L.; Schreiner, S. Integrating Power Models into Instruction Accurate Virtual Platforms for ARM-Based MPSoCs. In Proceedings of the ARM Techcon 2016, Santa Clara, CA, USA, 24–26 October 2016.
16. Thomas, D.B.; Howes, L.; Luk, W. A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation. In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, USA, 22–24 Febuary 2009; pp. 63–72.
17. Liu, W.; Xu, J.; Wu, X.; Ye, Y.; Wang, X.; Zhang, W.; Nikdast, M.; Wang, Z. A noc traffic suite based on real applications. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Chennai, India, 4–6 July 2011; pp. 66–71.