System-Wide, Fault-Tolerant State Agreement Protocol for Time-Triggered MPSoC

Dissertation
zur Erlangung des Grades eines Doktors der Ingenieurwissenschaften

vorgelegt von
Alina Lenz, M.Sc.

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen

Tag der mündlichen Prfung
20.05.2020

October 28, 2020

Betreuer und erster Gutachter
Prof. Dr. Roman Obermaisser
Universität Siegen

Zweiter Gutachter
Prof. Dr. Roland WismÏler
Universität Siegen

Abstract

Adaptive time-triggered systems use a set of schedules that the system can switch into at runtime. All schedules and their changing conditions are connected in a graph which is traversed based on the system state. If a condition to trigger a schedule change is met at the associated schedule will be changed into. Each schedule is computed w.r.t. the optimal energy usage and fault mitigation under the condition of the observed system state.

Critical domains like avionics and railway to profit from the advantages of online system reconfiguration because the schedule changes, while optional as they are based on the system state, are still planned within the potential system states. As no unpredicted changes can occur, the system's runtime behavior is predetermined by the schedule and thus providing the safety guarantees needed for safety critical systems.

While locally adapting single cores is already common practice, this thesis shows that performing the adaptation on a global level provides more advantages w.r.t. energy savings and fault mitigation. We show that the global approach leads to higher energy savings as the exploitable system state can be used not only by the tile where it occurred but other tiles can also profit.

The adaptation is performed decentralized within the distributed network of tiles of the MPSoC, which is why a common view on the global state is paramount. Before a schedule change decision can be taken a global view must be agreed on.

After introducing the adaptive architecture needed to perform a global adaptation this thesis proposes an agreement protocol for a network on chip. We show how such a protocol can be implemented while keeping the energy and transmission overhead minimal. The challenge within a network on chip is the fact that broadcast protocols are not available and each message has to be sent individually. Given that all tiles need to inform all other tiles about their current state during an agreement, this can to a heavy load on the network. The thesis shows that an additional and dedicated agreement network enables the system to run the agreement protocol without causing transmission delays on the normal network. We further show that by using such a protocol the adaptation can save up to 40% of it's usual energy consumption.

In the second part of the thesis we will introduce a fault-tolerant version of the protocol. We introduce two fault tolerant architectures that enable the system to use the adaptation under the occurrence of a fault. We will show that the fault-tolerant version of the protocol is able to cope with arbitrary hardware faults enabling the system to reconfigure itself. By providing the adaptation even if nodes fail, we can enable the system to enter specialized safe states, that can adapt to the specific fault that was observed. The functional system lifetime can be prolonged, as the fault can be handled by the rest of the system without having to enter a minimal functionality safe state.

Zusammenfassung

Adaptive zeitgesteuerte Systeme nutzen eine Auswahl an Schedules aus denen Sie zur Laufzeit den aktuell besten anwenden können. Alle Schedules werden in einem Graphen gespeichert in welchen sie anhand ihrer Übergänge verbunden sind. Jeder Übergang gibt an, welche Bedingungen erfüllt sein müssen, um den Schedule nutzen zu können. Falls solch eine Bedingung zur Laufzeit beobachtet wird, kann der neue Schedule angewendet werden. Die Berechnung der einzelnen Schedules erfolgte unter der Zielsetzung einer optimalen Energienutzung und Fehlerreduzierung unter Berücksichtigung des Systemzustands. Kritische Bereiche, wie die Luftfahrt oder dem Zugverkehr, profitieren von der Re-Konfiguration. Schedule Änderungen bleiben, obwohl nur optional, vorhersagbar, da sie nur unter den richtigen Laufzeitbedingungen durchgeführt werden. Es werden keine unvorhergesehenen Zustände angenommen, so dass der Systemzustand zur Laufzeit immer durch den Schedule vorherbestimmt ist. Diese Vorhersagbarkeit ist somit eine Grundvorrausetzung für sicherheitskritische Systeme bei der Zertifizierung und wird mit der Re-Konfiguration erfüllt. Während es bereits gängige Praxis ist einzelne Kerne bei Laufzeit zu optimieren, zeigt diese Arbeit, dass die Anwendung der adaptiven Verfahren auf globaler Ebene gröere Vorteile in Bezug auf Energiemanagement und Fehlertoleranz bringt als der lokale. Wir zeigen, dass der globale Ansatz mehr Energie spart, da der aktuellen Zustand eines Knotens nicht nur lokal, sondern zusätzlich auch an allen anderen Knoten des Netzwerkes genutzt werden kann. Ein Beispiel dafür ist dynamischer Slack, der bei Laufzeit entsteht. Bei einem globalen Energiemanagement können auch andere Knoten von dem lokalen Slack eines Knotens profitieren. Dadurch wird zu jedem Zeitpunkt die grötmögliche Energieeinsparung erreicht. Aufgrund der dezentralen Adaption für alle Knoten des Netzwerks muss ein zuverlässiger und einheitlicher Blick auf den Systemzustand gewährleistet sein. Vor jeder potenziellen nderung des Schedules, muss daher ein global einheitlicher Zustand unter allen Teilnehmer des Netzwerks ausgetauscht werden. Diese Arbeit führt zunächst diesbezüglich eine adaptive Architektur ein, die notwendig ist, um eine global Adaption zu verwirklichen, um dann den Hauptfokus der Arbeit zu beschreiben: ein Protokoll zur Abstimmung eines globalen Zustandes für ein Network on Chip. Wir zeigen, wie dieses Protokoll implementiert werden kann, während der dadurch verursachte Energie- und Übertragungs-Overhead minimal bleibt. Die gröte Herausforderung hierbei ist die Tatsache, dass das Network on Chip keine Broadcast-Protokolle unterstützt und daher alle Nachrichten einzeln gesendet werden müssen Bei dem Austausch dieser Nachrichten müssen alle Knoten die übrigen Teilnehmer des Netzwerks über ihren aktuellen Zustand informieren. Dadurch wird das Netzwerk stark belastet Das zusätzliche und dedizierte Protokoll-Netzwerk erlaubt es hingegen, dass sich alle Knoten in einem Netzwerk über ihren aktuellen Zustand informieren können. Eine globale Abstimmung über den Netzwerkzustand wir daher möglich, ohne den normalen Netzwerk Verkehr zu blockieren. Auerdem zeigen wir, dass durch das vorgestellte Protokoll eine Energieeinsparung von bis zu 40% im Gesamtsystem erreicht werden kann. Des Weiteren stellen wir stellen wir eine fehlertolerante

Version des Protokolls vor. Wir erläutern verschiedene fehlertolerante Architekturen, die dem System erlauben, die Adaption auch unter dem Einfluss von Fehlern zu nutzen. Durch die fehlertolerante Version des Protokolls ist das System in der Lage beliebige Hardware Fehler zu verarbeiten und sich gemä des Fehlerstatus neu zu konfigurieren. Im Fehlerfall kann sich das System dadurch sich in spezialisierte sichere Zustände begeben, welche für den eingetretenen Fehlerfall ideal sind. Die funktionale Lebenszeit des Systems kann damit verlängert werden, da sich das System nun eigenständig, um den Fehler neu konfigurieren kann, ohne die restlichen Knoten im Netzwerk zu beeinträchtigen.

Contents

Chapter 1

Introduction

Mixed criticality systems bear many integration advantages for industrial use as application of different criticality are able to share the same system on chip (SoC). Formerly these application were run on dedicated hardware [Burns and David2016] to guarantee their correct functionality by eliminating any interferences. The integration of applications lead to smaller devices, which take up less space and have an overall reduced energy consumption because the core's idle time within an application can be used for another application.

In safety-critical domains these systems have to additionally comply under strict certification regulations (e.g., ISO26262, IEC61508, and DO178/254), as their failure can cause massive damage for human lives, property or the environment. These systems are especially regulated and must guarantee a predictable runtime behavior.

In order to ensure their correct functionality, even under the presence of other tasks and the certification regulations, safety-critical applications employ a precomputed temporal behavior defined in a schedule. This schedule defines when a task is allowed to be executed and when messages may be sent. In safety critical domains these schedules not only assign the when a task is executed but also system wide communication between the different building blocks via the network on chip (NoC). This way the system can communicate collision free and guarantee a maximal message traversal time.

These schedules must ensure that the applications are separated in both timely and spatial domains and a perfect alignment of the execution and the communication schedule must be ensured at all times. It is paramount that message injections are scheduled when a message is able to be sent by the task. Since both the execution and communication schedule are computed at design time, a runtime behavior of the tasks can only be approximated with a worst case execution time (WCET) which defines the interval in which a task is able to provide its functionality at the latest. A task will under no circumstance need more time to finish, as we already assume the worst possible scenario for the WCET estimation. With this very pessimistic approximation comes inevitable idle times as the absolute worst case rarely occurs and usually average runtimes only take a third of the assumed WCET. Hence the resulting systems are limited in its optimal usage of the SoC's hardware by its requirements for predictability.

Other Systems that are scheduled online at runtime can exploit these idle times to dynamically reschedule the upcoming tasks to start earlier or perform the task under a lower frequency to save energy. And with the rapid growth of industrially used multiprocessor systems-on-chip (MPSoCs) the energy consumption grows in importance. Each energy consumer produces heat which in turn negatively impacts the hardware. The authors of [Hong1999] have shown that each 10°C the system's failure rate increases by 50%. For this reason fans and other dynamic elements need to be integrated to cool down the system. These dynamic elements are prone to failure and weigh a lot. Reducing them is a major concern in today's industry as energy and reliability optimization are considered to be among of the most critical issues of MPSoCs [Anrup2014].

As a result, extensive research has been done on how to effectively reduce the energy consumption by applying dynamic energy saving mechanisms like dynamic voltage and frequency scaling (DVFS).

This thesis introduces how such a dynamic adaptation can be applied in safety critical systems. This approach mimics the dynamic behavior of online scheduled systems. We can pre-plan all possible scenarios the system can enter at runtime and precompute schedules that optimally utilize the hardware for the different scenarios. This mechanism, as it is precomputed, can compute schedules for a global scale and thus maximizing the energy savings for each schedule. At runtime the system would have to observe its current state to decide if one of the predicted scenarios is reached. One example for such an global adaptation of the system is slack: The schedules are computed under the assumption that the tasks can finish after half of its WCET. The scheduler computes specific schedules for these scenarios. When the system goes online, and at runtime a task reports that it finished after half its WCET, the system must be made aware that one of the presumed scenarios is reached. It can then change into the associated schedule for the scenario and operate at an more optimized rate during the current period. At the beginning of the next period the system starts in its base schedule and reiterates the adaptation for the new period again.

This global adaptation is not only limited to energy saving based on slack, but can be applied to any kind of adaptation that can be executed based on a runtime system state. It can be used to increase the safety argument of the system as faults can be handled more precisely. If a specific fault is observed the system can reconfigure itself based on the new schedule to, for example, migrate important tasks to a new tile or change message routes to avoid faulty routers.

A key element to the global adaptation is the establishment of a consistent global state. This global state is indicated by the so-called context. The context defines a system state at a specific time and thus can be used as input for possible schedule changes. It contains indicators of runtime events, for example slack is indicated when a task finished.

The context cannot be observed globally but is limited to the tiles that are connected to its context monitors. After a context is observed this local event must be announced to the rest of the network. Since all schedule changes are computed globally, the global state is taken into account for the optimal schedule. Hence, all nodes must be aware of the other tiles' state to make a schedule change decision. As all the tiles are running decentralized and manage themselves at runtime, thus a agreed upon system state must be known before a schedule decision is taken locally. If two tiles would assume different system states, they would take different scheduling decisions. Such an inconsistency can cause the system to fail and would invalidate the safety requirements as well as the runtime behavior guarantees.

The correctness of a time-triggered system depends on a consistent view of the system state in all tiles. As each node is an independent system with its own clock, which may slightly differ from each other, getting a consistent view of the whole system at a time t is challenging. By constantly synchronizing the tiles and making them agree on the global view of the system it can be ensured, that each tiles' view on the global system sate is consistent with the other tiles. This way ensuring that no tile assumes a different state and unsafe behavior is prevented: Collisions on the network are avoided, precedence constraints are satisfied and the tiles are also implicitly synchronized. Observance of context events triggers schedule changes and these changes must be aligned throughout the system to preserve the properties of a correct time-triggered system.The agreement must further be completed within a short time, to exploit volatile context such as dynamic slack. This type of optimization possibility must be exploited before the next scheduled task starts.

Agreement protocols are well known in distributed systems, but given the different system attributes regarding available energy sources and timeliness of the tasks that are executed, these protocols are not applicable to the time triggered multi-core architecture.

Our Contribution In this thesis we propose the following contributions:

- We propose a system model of an adaptable time-triggered multi-core chip.
- We introduce an agreement protocol for this multi-core chip, that provides consistency and aligned schedule changes.
- Lastly, we defined an extension of this agreement algorithm to provide correctness in the presence of faults.

This thesis is structured into nine chapters. It begins with the introduction in this first chapter and a state of the art analysis in chapter 2 before outlining how our contribution provides improvements. Followed by chapter 3, where recurring important concepts of the thesis are explained. Then, chapter 4 will describe the system model and the requirements before describing the global agreement in more detail in chapter 5. In chapter 6, we will describe the agreement protocol and its implementation in chapter 7. In chapter 8 we introduce a fault tolerant version of the protocol and in chapter 9 we draw a conclusion and describe the plans for future work within this field.

Chapter 2

Related Work and Research Gap

In this section we will take a look at the state of the art of related functionalities and discuss why our agreement protocol fills a void in this particular field of research.

Our research aims to improve the runtime performance of time-triggered mixed criticality MPSoC. On these systems applications of varying criticalities share the same hardware. As such it is essential that the different application are separated temporally as well as spatially. It is paramount that the low criticality applications cannot impact those of higher criticality. The error-free execution of the critical applications must be guaranteed at all time.

This can only be provided by strictly scheduling when and how long an application may access a hardware resource, be it memory access, a processor or the network connection while sending a message. All actions on the hardware must be pre-planned and precisely timed to ensure the separation. This means that not only each core's execution must be scheduled but also the on-chip communication.

This prescheduled approach is especially difficult for MPSoC that use networks on chip (NoC)s as a message based interconnect, in which the nodes are connected to routers which in turn are all connected in a mesh structure. This mesh enables the communication to be executed in parallel as long as the messages do not share the same link at the same time. The on-chip communication schedule dictates when messages are injected into the network and which path they have to take, this way the system guarantees that no collision will occur. As no collisions can occur, dynamic routing mechanisms are not needed and the path can be predicted which in turn enables us to predict the precise message transmission times. The complete schedule will therefore include the execution times for the application on the cores and the message injection times for all messages that need to pass over the NoC.

Due to the co-scheduling of execution and communication the communication is decoupled from the execution in a way that the message is not directly injected as soon as the application finished but is buffered in the network interface (NI) until the scheduled injection time is reached. As mentioned in the introduction most runtime execution times are only a fraction of the assumed worst case execution times (WCET) that are used for the schedules. This means that the message is usually buffered for a long time, using up resources and blocking tasks that are waiting for the message to arrive.

2.1 Atomic Broadcasts

To design an agreement protocol that can reliably inform all other networks participant about the current state of a node we took a look into broadcast protocols in general and how these protocols need to be designed to provide the level of reliability that safety critical applications need.

In broadcast protocols a set of requirements was defined that ensure that the broadcast messages are provided reliably [Hadzilacos and Toueg1994]. A broadcast that meets these requirements is called an atomic broadcast. An atomic broadcast is a communication primitive which ensures a fault-tolerant agreement while ensuring system consistency. While it is applied in large-scale coordination services such as replicated state machines [Hunt et al.2010] or travel reservation systems [Unterbrunner et al.2014], its implementations often rely on leader-based approaches, such as Paxos [Lamport et al.2001] [Lamport1998]. This way the implementations suffer from bottleneck of the coordinator. Paxos is a widely known implementation of an atomic broadcast as can be seen in its several practical applications [Boichat et al.2003] [Corbett et al.2013], [Krishnamoorthy and Krishnamurthy1987], [Marandi et al.2012]. And it has been optimized several times, where the load is distributed among the network participant or the order of processing is changed for not-interfering requests [Lamport2005] [Barcelona2008] [Moraru et al.2013]. Nonetheless, until now this commonly used replication scheme is not supporting large networks containing hundreds of instances.

2.2 Consensus

When designing the agreement protocol we heavily profited by the extensive research that was done in the area of distributed networks. In this section we will highlight the research results which impacted our protocol design decisions.

The concept of agreeing on a certain value in a set of not directly connected participants has been widely researched for distributed systems in a very general form called consensus [Guerraoui and Schiper1997] [Li et al.2013].

Although the target systems needs can vary from distributed services of datacenters [Corbett et al.2013] [DeCandia et al.2007] [Unterbrunner et al.2014] to distributed operating systems such as barrelfish [Schüpbach et al.2008] and Mesospheres DC/OS [mes] they all need a shared view on the system state.

In the consensus approach a set of processes running on distributed nodes need to decide on a common value under the assumption of faults in the system. Each process possesses a value at the beginning that needs to be considered for the common value. Usually it is assumed that the processes are asynchronous as this assumption put the weakest constraints on the model and is applicable to most scenarios.

By agreeing on the value certain global decisions can be taken unanimously thus avoiding inconsistencies in the nodes. Examples for such decisions are leader elections and agreeing on which process has failed. While the premise of reaching a globally shared state is similar to our problem within the MPSoC, the conditions for the implemented protocols vary.

Distributed systems cover wide areas of varying ranges which need to be kept in sync with each other w.r.t their current state or structure within the network. Often nodes act as relays to other areas of the network. Erroneous nodes introduce wrong values and thus impact whole subareas of the network, preventing the system from reaching an agreement.

For the consensus the initial node values can for example be the next step a process wants to take. If we now assume a weakest form of consensus where only one process decides on the common value [Fischer et al.1982], this central node needs to collect all values from the participating processes. As such it acts as an observer of the current states of the processes and can conclude observations and influence the process by giving acknowledgments to the processes. This type of consensus is not fault tolerant, since the observer can crash before sending acknowledgments and block the system. Even worse is a scenario where the observer is faulty and sends arbitrary messages to the process.

Making the consensus fault tolerant means to inform all nodes about the other nodes information. This way the single point of failure can be avoided. If we now also assume that a subset of the nodes may act malicious, the mechanism needs to be more complicated which is used in [Lamport et al.1982]. A key to fault-tolerant consensus is the transmission phase. In all types of consensus cases the information transmission phase is essential, since all correct nodes must be informed of the information of all other correct nodes in order to allow them to reach a common decision [Bracha and Toueg1985] [Barborak et al.1993] [Chandra et al.1996]. This means that because all nodes may crash, each node must become an independent observer. Regarding the solvability of the consensus problem [Fischer et al.1982] concluded, that there is no solution to the problem in asynchronous nodes if one allows at least one node to crash. To solve the issue some weak forms of synchronization was introduced [Dolev et al.1987] [Fischer et al.1990] [Dwork et al.1988] [Chandra et al.1996]. With this new assumption there even exist solutions for the problem under the assumption that a limited amount of nodes behave maliciously according to the Byzantine fault model [Lamport et al.1982].

2.2.1 Fault-Tolerant Consensus

The Byzantine errors are especially hard to deal with, as they assume a malicious adversary model which is not limited [Barborak et al.1993]. For protocol designing purposes the adversary is usually restricted in its maliciousness, its computational power or the processes it may impact. Chor and Coan [Chor and Coan1985] gave the adversary four principal restrictions:

- The adversary may corrupt only up to a third of all processing nodes.
- The communication system must be reliable and unreliable links must be simulated as a corruption on the sending or receiving core.
- The adversary may not predict random events.
- The synchrony of the system may not be interrupted.

These restrictions were hard to abide. For this reason the community adopted another classification of faults soften but adhere to the rules. This new taxonomy divides processor faults into various groups where each stronger class is a subset of a weaker class. In order from strongest to weakest:

- **Fail-Stop Fault**: The processor stops working and informs other processors of his failure [Schlichting and Schneider1983].
- **Crash Fault**: A processor loses its internal state or halts.
- **Omission Fault**: A processor fails to meet a deadline or to begin a task on time [Cristian et al.1986].
- **Timing Fault**: A processor finished its task before or after its expected time frame [Cristian et al.1986].
- **Incorrect Computation Fault**: A processor produces an incorrect result while receiving correct inputs [Laranjeira et al.1991].
- **Byzantine Fault**: Every possible fault. This is the universal fault set.

We based or fault-tolerance on these fault assumptions and used a derived fault model to abide to the accepted and applied community standards.

2.2.2 Applied Consensus Protocols

In recent years leaderless approaches have been pushed [Poke et al.2017] where the total order is achieved by using destination agreement, meaning for example the agreement on a message set. The total order is a main classifier for atomic broadcasts as defined by [Défago et al.2004]. In the basic concepts section 3 we explain the atomic broadcast in more detail.

A major focus of the field goes into proving that these leaderless agreement problems are actually solvable and a agreement on a common value can be found. [Mostefaoui and Raynal2000] shows that in asynchronous, fully distributed systems an agreement on a value can be reached if within a system of n nodes, f of which may fail, there is a set of $(kn+f)/(k+1)$ nodes that propose at most k different values. In this case the nodes can agree on the common value while more failing nodes will leave the agreement problem unsolved.

How our contribution expands the described state of the art: In our fault tolerant protocol version the agreement is ensured by a redundant infrastructure. As we assume a limited network within a single MPSoC, the fault assumption is also limited to one fault at runtime. As the protocol allows us to detect single faults, the system can be exchanged in time before further areas of the system fail.

The protocols to find an agreement have been extensively researched and advanced w.r.t. their time until an agreement is reached and how many messages are needed [Aikebaier et al.2010]. The most used algorithm is the gossip algorithm [Dimakis et al.2006] where a pair of nodes is chosen randomly to exchange information and update their statuses. While it is easy to implement, it generates message overhead due to the random information exchanges thus leading to longer times till an agreement can be reached. Additionally this protocol is strongly affected by the nodes structure, as the connectivity between the nodes affects the performance. If the nodes are connected sparsely less exchanges can be initiated.

How our contribution expands the described state of the art: Our solution uses a predicted network structure where only a restricted set of network participants can communicate to each other, thus reducing the execution time of the protocol. While the protocol is thus more sensitive to disturbances, because the message propagation is limited, we avoid the negative impacts of transmission faults by creating redundant paths that are independent to each other.

A second often used protocol is the binary majority consensus [Perron et al.2009], [Al-Nakhala et al.2013], which terminates after a defined time even if no consensus is reached. As the execution time of the protocol is limited, the efficiency is lower and it is very sensitive to faults and disturbances in the transmissions.

How our contribution expands the described state of the art: Our solution only waits for a certain time as well to reach for an agreement. The protocols main goal is to keep the runtime predictable and as short as possible. If no agreement could be found by the protocol, we can at least tell which value was corrupted. To avoid any problems for the safety critical services of our MPSoC we define fall back values that can be used instead. The values lead the system into a stable and save state, while maintaining the runtime functionality as well as possible under the observed conditions.

2.3 Energy Saving Mechanisms

The agreement protocol should not only ensure that all nodes are aware of the same system state but also improve the MPSoCs energy management while at the same time imposing as little energy as possible. There are ample approaches to limit the energy consumption in the MPSoC. We can differentiate between dynamic approaches that apply online-scheduler to save the energy at runtime and ones that pre-plan the optimized behavior.

One example for an dynamic adaptation approaches of the system to increase the safety argument if the system is proposed by [Ogrenci Memik2001] [Persya2013]. In this a super scheduler manages a safety-critical system, which is running under normal assumptions of faults. This super scheduler is designed to be applied for unpredictable events like an environmental catastrophe or a system wide device failure caused by failing nuclear power station. As these events are so rare, the super schedulers goal is to quickly enter a special safe state and shut the system down. For this it injects new high criticality tasks into the system while maintaining the normal application's deadlines. The normal critical tasks get downgraded and interrupted. This approach introduces errors into the runtime behavior, as the online scheduled additional tasks cannot guarantee that the original deadlines are kept.

How our contribution expands the described state of the art: In contrast to this work, our solutions will not disturb the current high critical applications as the changes will be planned at design time. No scheduling decision is taken online. Every schedule change is decided before runtime. Hence, our approach can guarantee that at runtime no deadline is missed and all system requirements are met. Our schedule approximates the dynamic behavior with precompiled schedules, guaranteeing that no real-time constraints will be violated.

Apart from the crisis injection of unplanned events the dynamic approaches mainly focus on a dynamic change of the current schedule based on the current state. As such there are multiple ways to manipulate the system to save energy:

DVFS Dynamic voltage and frequency scaling (DVFS) has been widely researched in embedded systems. DVFS reduces the frequency of the system which leads to lower energy consumption. As the system operates under a lower frequency the execution time rises. Dynamic systems exploit this fact if a process is expected to finish earlier than needed, the frequency can be lowered so that less energy is consumed. As the lower frequency can lead to an unexpected long runtime this approach may lead to deadline misses and must be monitored closely by an online scheduler. For reason DVFS is currently avoided in critical systems. This technique exploits the fact that the consumed energy is calculated as: $E = c * V^2 * f$ with c being the system capacitance, V the voltage and f the frequency. Both factors can be reduced to lower the energy consumption. Most approaches aim for a localized tile-based optimization where dynamic slack is distributed to other locally executed tasks or the core frequency is dynamically lowered during a task execution. In these approaches the aim is to optimize the time-slot given by the schedule.

Inter-Tile Slack Propagation [20] proposed an inter-tile slack propagation to optimize the effect that slack has on the system. Each time when slack is observed in the system the available free time is used to reschedule the upcoming tasks online and thereby giving the slack to the next tasks. This slack can be used by DVFS to perform task execution at a lower frequency.

Their approach to use the available slack globally is very similar to our approach. However, we extend it into a temporally predictable and certifiable technique for mixed-criticality systems. By pre-computing the possible schedules, we avoid the possible deadline misses that a online scheduled system has. Additionally the predictability of an offline scheduled system is a key requirement for certification.

Static Approaches The static approaches provide better suited schedules for the systems. Energy efficiency based on scheduling is a highly researched topic ranging from application of energy management techniques to very specific energy management schemes on MPSoC.

Approaches aim to save energy focus on improving the schedule a system is running upon [Chen and Kuo2007] [Li and Ampadu2015]. They define the timeslots of the tasks based on heuristics to implement less pessimistic timeframes and hence reduce the idle time.

The approaches use these heuristics for their online scheduling and aim to dynamically adjust the system state and thus approximate an optimal offline scheduler [Paterna and Benini2013] which would have known the real execution times beforehand.

2.4 Conclusion

While our agreement protocol can benefit from the extensive research of agreement protocols in distributed systems, it faces fundamentally different challenges on the MPSoC. Within MPSoCs in safety critical domains the energy and time efficiency is important.

While agreement protocols are widely used the specific scope of the NoC imposes harsh requirements that the currently available protocols are not able to solve. In distributed systems the time aspect is less pressing and protocols can cost more in terms of transmitted messages as the energy and the message transmission time is not a main concern.

In our protocol the implementation has to impose a minimal overhead in terms of time and messages. The agreement is used to enable the system to adapt to its current state w.r.t. energy and faults, thus improving the runtime behavior. The faster our protocol can provide a commonly shared view on the state, the faster the system can adapt and improve. Especially in embedded systems the states change quickly as the applications runtimes are short and values are updated quickly. If the agreement takes too much time, the system will never be able to have an up to date view on the current state and thus adapt to a non optimal schedule.

As one can see the optimization of MPSoC under the aspect of energy optimization has been researched extensively in non-safety critical domains. There is ample research and approaches on how to save energy by applying online scheduling. Our goal is to enable safety-critical areas to apply these techniques as well. As such our approach mimics the well researched dynamic versions by representing them in a set of static schedules. This way we can provide the safety guarantees with the advantages of dynamic adaptation.

Chapter 3

Basic Concepts

This chapter is used to introduce the reader to the concepts on which the contribution of the thesis is based. In here we shortly present all concepts that are needed for the thesis. We also use this chapter to define the nomenclature that is used throughout the thesis.

3.1 Name Convention regarding Tile and Core

Throughout the thesis we use of the term tile and core. We will refer to a *tile* as the cluster of cores and other Intellectual Properties (IP) that share a common bus and are connected by the same Network Interface (NI) to the common Network on Chip (NoC). *Cores* are the actual executing hardware blocks used for computations. Each tile can be assigned an arbitrary amounts of cores.

3.2 Real-Time Systems

Real-time systems must operate under the restrictions of timing constraints. Services must be performed under the assumptions of deadlines after which the function output is not useful anymore or loses its validity.

Real-time systems are commonly applications used to control objects, usually implemented on an embedded computed systems. The embedded systems are used to manipulate the controlled object into a wanted behavior based on the overall system state that can be observed via sensors.

3.3 Dependability

Dependability is the ability of a system to perform its intended task as predicted and produce an outcome that can be trusted [Obermaisser2012]. It is especially important in safety critical applications.

Dependability Threats The dependability of a system is threatened by unpredictable behavior of a system. Any unpredictable behavior is caused by *Faults* occurring at runtime, causing for corrupted functional outputs, which are considered *errors*.

Faults can be based in all areas of the system. The hardware itself may be subject to faults as it digresses of its lifetime or external influences may negatively influence the system. The possible causes are ample. These errors that results from occurring faults, the wrong function outputs and behaviors in turn cause for failures within the whole application.

These failures are defined as a state in which the ”delivered service deviated from fulfilling the functional specification” [Obermaisser2012].

Meaning that the real-time system caused the controlled to behave out of the expected way. This can cause mild annoyances if a multimedia system is not functioning correctly. But in safety critical applications these behaviors can cause harm to human lives or massive financial damage. In these domains it is paramount that the systems are dependable under the assumption of faults. No error may under any circumstance impact the overall behavior of the system in a negative way.

Fault Containment Unit To ensure that a fault does not propagate within the whole system the fault containment region (FCR) is defined as the biggest area within which a fault may immediately impact the system. We therefore defined what part of a system may fail due to a fault and any impact outside of this FCR must be treated. Approaches to the fault treatment are split into resilience and recovery. The resilience approach means that the fault is detected and a safe state is reached and normal operation is terminated. The recovery approach corrects the fault and isolates the area which was corrupted, while the rest of the system continues to operate normally.

Failure Modes Failure modes are definitions on how the fault impact the service the FCR provides, therefore it defines in which way the service deviates from the expected behavior. As in multicore architectures, where the thesis is applied to we usually define FCRs as a tile.

- Fail-Stop Failure: The tile does not provide any output anymore. This failure is assumed to be detected by the rest of the tiles.
- Crash Failure: The service stops to produce outputs, but the other tiles may not be aware of the failing of the tile.
- Omission failures: A sender fails to send or a receiver fails to receive a message. Therefore the receiver is not capable of responding to the message. The detection of this error is not guaranteed.
- Timing Failure: The node does not act within its timing restrictions, meaning that messages are transmitted too early or too late.

- Byzantine or Arbitrary Failures: Arbitrary failures are recognize by a ”two-faced” and inconsistent behavior and output of a node.

3.4 Global Time

As mentioned before real-time systems operate under the assumption that their functions must perform within a defined time frame. For this a global notion of time is introduced to the system. The main challenge of a global clock is the synchronization of the many locally used clocks within distributed systems. In safety critical domains it is especially important that a system is properly synchronized and the nodes can validate event timings properly and thus enabling each node to determine the temporal oder of two events. It is important to note that a global time is an abstract notion. As real clocks are not perfect a system can only try to mimic a global time by constantly synchronizing all nodes.

3.4.1 Sparse Time Base

To establish a system state in a distributed system, the network needs to exchange a *snapshot* of all node’s states at the time t. This is especially difficult as the nodes differ slightly in their local time. Creating a snapshot therefore is only possible if the system is able to distinguish which events on the nodes can distinguish at which time in relation to the global time base the events on the nodes happened.

To enable the system to reconstruct which events happened in which causal and temporal order, the sparse time base can be used. If one controls the event execution we can put events into temporal limits the possible times when they can occur. The possible occurrence time is reduced to so-called *active intervals* of duration ε after which an interval of silence will follow for the duration of Δ.

If two nodes, nodes j and k of cluster A, generate two events at the same clusterwide tick t_i, i.e., at tick t_i^j and at tick t_i^k, then these two events can be a distance Π apart from each other, where g $> \Pi$, the granularity of the cluster-wide time. Because there is no intended temporal order among the events that are generated at the same cluster-wide tick of cluster A, the observing cluster B should *never* establish a temporal order among the events that have been sent at about the same time. On the other hand, the observing cluster B should *always* reestablish the temporal order of the events that have been sent at different cluster-wide ticks. [Obermaisser2012].

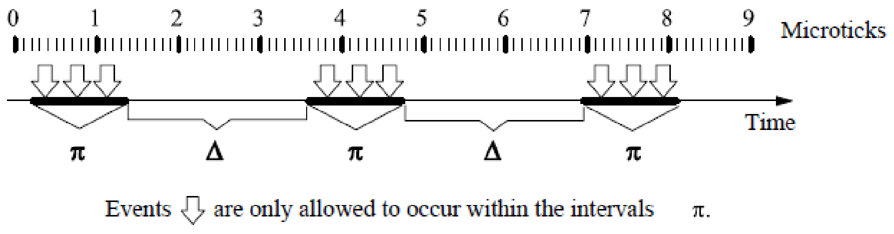

Figure 3.1: Timeline of a sparse time base. Events are only allowed to occur within the intervals π [Obermaisser2012]

3.5 Event-Triggered and Time-Triggered Systems

In general there are two types of embedded systems:

- *Event-Triggered Systems*: In these systems the execution of the services is based on events. Applications can trigger other applications to start operating by sending messages. In best and average case scenarios this leads to a fast reaction to the runtime behavior of the different tasks. Everything is executed as fast as possible. At the same time faulty events can trigger tasks out of order and thus cause the error to propagate through the system.

- *Time-Triggered Systems*: In time-triggered systems all tasks are executed at a predefined time. This separates the tasks and nodes in time and prevents error propagation. At the same time it forces the system into a strict time schedule that must be met by all nodes. For example if one node wants to read the message with the input it needs to operate at time t the predecessor must have provided the message up to this point. As the runtime execution time is varying due to the many tasks interacting on one system, the behavior must be predicted in the most pessimistic way. This ensures that even under the worst case the tasks can finish on time before the next task is started. This schedule in turn leads to dynamic slack, as the worst case execution time is needed. In safety critical areas the time-triggered paradigm has established itself due to its predictability and timing guarantees.

3.6 Mode-Switching in Time-Triggered Systems

The rigidness of the time-triggered systems has been subject to research over the years, as the predefined schedules prevent the system from fully exploiting the advantages of mixed criticality applications on a safety critical system. While non critical systems can operate event-triggered and execute lower criticality tasks in idle times of the core, the time-triggered system is entrapped in its own schedule. No additional tasks may be executed as it is not sure whether their execution can be finished in time. Therefore recent years research [Ahmadian; H.; Obermaisser2015] has integrated mode changes into the systems that, driven by a server node, changes between system modes. These modes define if a system is to strictly be executed by the current schedule, or if predictable idle times can be used to execute lower criticality tasks. These mode changes operate under the premise that the lower criticality tasks can be pre-empted at all times and thus not risk a negative impact on the high criticality tasks.

3.7 Agreement

Agreement, or also called *Consensus*, allows tiles to reach a common decision that depends on their initial inputs despite failures. All correct tiles propose a value and must agree on a value related to the proposed values [Hadzilacos and Toueg1994].

Atomic Broadcast An atomic broadcast is a broadcast that satisfies the following properties regarding the message distribution [Hadzilacos and Toueg1994]:

- *Validity:* If a correct process broadcasts a message *m*, then it eventually delivers *m*.
- *Agreement:* If a correct process delivers a message *m*, then all correct processes eventually deliver *m*.
- *Integrity:* For any message *m*, every correct process delivers *m* at most once, and only if *m* was previously broadcast by *sender(m)*.
- *Total Order:* If correct processes *p* and *q* both deliver messages *m* and *m'*, then *p* delivers *m* before *m'* if and only if (iff) *q* delivers *m* before *m'*

3.8 State, Event and Context

State: The system state is a reflection of events that occurred within the system at runtime. It is the value of real-time entities at a particular point in time.

Events: An event is a change of a state.

Context: Context is a special kind of state. It describes the state which is relevant for adaptation.

Context String: The context is encoded in a context string. This context string is a bit vector, where each bit defines whether a certain context event occurred in a specific period of time.

Chapter 4

System Model

This chapter introduces the system structure of our target system with its building blocks, interfaces and abstractions. It explains in a generic way how the adaptation in a time-triggered MPSoC works to introduce the system and its constraints for which our contribution is designed.

4.1 Adaptive Time-Triggered Multi-Core Architecture

Our system model contains a tile based architecture with an arbitrary system size. Each tile supports cores which can be operated bare-metal or managed by a hypervisor. This way the architecture is flexible to be adapted in a wide range of safety critical domains. The tiles are interconnected by a network on chip (NoC) which is operated in a time-triggered manner. This means that all message injections are performed based on a predefined schedule, which decouples the applications from each other, as the messages no longer dictate when depending applications start. This model depends on the message injection to be aligned with the execution schedules, as the application might need input to run correctly. The alignment of these schedules must be kept under all circumstances even with adaptation schemes performed within the MPSoC.

Such an adaptive architecture is shown in figure 4.1 where the blocks need to ensure the correct adaptation functionality. The conventional time-triggered architecture [Steiner and Kopetz2006] containing tiles of multiple cores and NIs is extended with context monitors and the *Context Agreement Unit (AU)*. The monitors allow the AU to have a view on the local context observed by the tile. The integration into the NI allows the AU to use the NoC to exchange the local information and establish a global context where all tiles are aware of the local context of the other tiles.

In detail the architecture consists of the following building blocks:

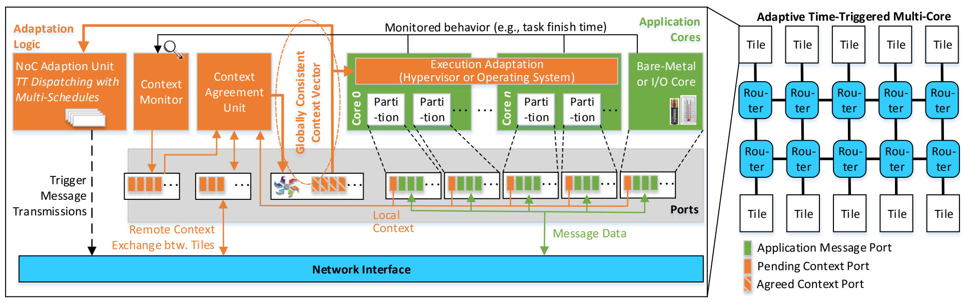
Adaptation Logic
NoC Adaption Unit
TT Dispatching with Multi-Schedules
Context Monitor
Context Agreement Unit
Globally Consistent Context Vector
Monitored behavior (e.g., task finish time)
Application Cores
Execution Adaptation (Hypervisor or Operating System)
Core 0
Parti -tion
Core n
Bare-Metal or I/O Core
Trigger Message Transmissions
Remote Context Exchange btw. Tiles
Local Context
Message Data
Ports
Network Interface
Adaptive Time-Triggered Multi-Core
Tile
Rou-ter
Application Message Port
Pending Context Port
Agreed Context Port

Figure 4.1: Architecture of a globally adaptable time-triggered System

4.1.1 Tile

A *tile* is defined as the biggest cluster of cores or IPs that share a Network Interface (NI) to the NoC which connects all tiles. Internally the tile's components are connected by a shared bus. Each tile is connected to the rest of the system by a Network on Chip (NoC). The tile can consist of one or more cores and IPs and communicates all messages that it wants to transmit to other tiles by using unidirectional ports within the NI. Each core has a dedicated input and output port in the NIs core interface. The application itself is unaware of the communication schedule, thus it just writes the message into the port based on its configuration and does not know when it will be sent.

We differentiate between application cores, that are programmed onto the FPGA part of the MPSoC and classically dedicated to a single task due to their limited resources and hardware multicores. The hardware cores provide a much larger computational power and with the help of hypervisors these cores can be used to host a larger amount of virtual cores while at the same time being able to provide safety guarantees w.r.t. to runtime behavior thus strengthening the multicore architecture.

Monitoring partition To provide a close monitoring on the varying partition behavior the hypervisor implements a *monitoring partition* which can be used to locally accumulate the information that is to be observed by the tile. This information is gathered either by the software running on the hypervisor or by sensors that are connected to the tile. The monitoring partition can also be used to pre-process the collected software information and sensor values to create a local context overview. This partition is only executed once per period at the end, therefore it is not suitable for context information that needs to be reacted to in short time, e.g. slack.

If the hypervisor schedules its local partitions under customized schemes and not based on simple Time Division Multiple Access schemes like round robin, the monitoring partition can further be used to feedback to the partition scheduling and thus optimize the partition scheduling based on the context. For this the monitoring partition can be informed of the NIs current schedule and can perform local adaptation schemes to further optimize the system performance. This local adaptation focuses on optimizing the tile's performance under the timing constraints given by the communication schedule. All adaptation that can happen locally is bounded by the communication deadlines. As the local adaptation only affects the local tile, the messages that are meant to be sent to other cores must be available at the ports when they are planned to be injected.

This local adaptation can only be performed after a decision on a possible global adaptation is taken, since the global communication schedule defines the timing constraints.

Context monitor The context monitor is a building block designed to poll the current context values and report the adaptation relevant information to the agreement unit.

They are executed in a time-triggered fashion therefore automatically introducing a *polling delay*. The delay depends on the types of events that are observed by the CM. There are two types of events that can be observed:

- *Synchronous Events*: These events are predictable in their occurrence. We know that the event can either happen or not at time t_e within the period. These kind of events allow for predictable changes to optimize the system performance. Slack for example can, if it occurs, be used to core gate devices or apply *Dynamic Voltage Frequency Scaling* (DVFS). These events can be checked for at the expected time.

- *Asynchronous Events*: These events happen at a random point in time throughout the system's lifetime, e.g. faults or battery health. They indicate a degradation of system health which needs to be adjusted to. These events must be caught at runtime by constantly checking the indicating context monitors.

The synchronous events have a predictable offset between the potential occurrence and the beginning of the agreement protocol. Since we know when they are to be expected we can schedule the agreement to fit the event's volatility. Asynchronous events on the other hand are unpredictable in their occurrence. These events can only be sampled and agreed on regularly. This can lead to protocol overhead, as the events may not occur at all during runtime. The more often the asynchronous events are checked and agreed on the faster the system can adapt to them. In case of high impact failures the frequent observation of the system state can minimize the time until a fault can be detected and mitigated. If faults occur the network can be reconfigured accordingly as soon as the error is noticed. The best frequency for the asynchronous event sampling is a trade-off between the message overhead caused by the agreement and the safety that a time-sensitive reconfiguration can provide for the system.

The output of a context monitor varies depending on the system needs, if it is only necessary for the system to know if an event occurred, e.g. the battery level dropped under a pre-defined threshold, the context monitor will need only a bit to encode the information. If the actual value of the context is needed a more elaborate context type must be used to exchange the information.

Depending on what context is observed the monitors can vary widely. They can be implemented in two ways:

- *Dedicated IPs*: Used to collect context information. Sources for the context information can be sensors that observe the environmental system state, or services that monitor the system's performance at runtime. Examples for a context monitor are temperature sensors that observe the SoCs temperature or the environmental temperature. The temperature of the SoC can indicate if a node is running too hot and thus can lead to an increase the energy consumption or application failures [Hong1999]. By re-assigning tasks away from these hot spots, the energy usage can be balanced. The environmental temperature can be vital for SoCs that are used in open areas and can suffer from direct exposure to sunlight.

- *Self Reporting Applications*: Applications can report on runtime events as well. An example is slack, as the application can report that it finished its operation.

4.1.2 Network Interface

The NI is the component in charge of the time-triggered message injection. It contains a dispatcher which is aware of the communication schedule and triggers the message injection according to it. For each injection time the schedule dictates which message needs to be transmitted. This message is read from the port by the NI's dispatcher and relayed to the Serializations Unit where the core network transmission functionalities are implemented. The Serializations Unit provides services like the fragmentation of a message into flits and the transmission of it according to the NoC configuration.

Ports Ports are used as an interface between the cores and the NI. All application communication as well as the context reporting is handled via ports.

To separate the communication from the reporting the AU port can only be read by the AU while the communication ports are read by the NI's core functionality block, where the message injection is handled. The ports are designed as buffer where the communication driver inserts the message. Each new message overwrites the previous one. A port is uni-directional. The number of ports per NI depends on the application and the amount of CMs that are connected to the NI. Each Application and CM have at least one input port within the NI where they report their context to. If needed a output port can be added to provide a status update on the current schedule and system state by the AU to the CMs. This can be useful for the monitoring partition.

Each port is dedicated to a specific context. This way the AU knows how to interpret the information placed in the port. The information is then interpreted w.r.t. the occurrence of an event.

NoC Adaptation Unit This unit executes the schedule changes based on the global view established by the AU. The input provided by the AU is taken as a trigger to either initiate a schedule change in the NI and the core or keep the current schedule as it is.

Agreement Unit The agreement unit executes the agreement protocol to establish a global view on the system state.

Chapter 5

Safe Adaptation for Time-Triggered MPSoCs

With the restrictions put on safety critical domains like railway and avionics the systems are not allowed to behave in unpredictable ways. It must always be guaranteed that the system behavior may not differ from the a priori computed schedules and at no point in time an unexpected system state may be entered. These schedules are by default very pessimistic and the real runtime behavior leads to long idle times within the cores. To avoid the under utilization caused by these pessimistic schedules, an adaptation scheme is introduced into time triggered systems [Lenz et al.2017]. This adaptation enables the MPSoC to observe its current state and change into a more optimized schedule based on it. Classical energy optimization utilizes dynamic behavior that is scheduled online based on the current system state, which can lead to deadline misses and introduces additional overhead. Additionally it leads to unpredictable runtime behavior, which is unwanted by certification standards, which demand a prior analysis of the system behaviors. By operating the system under a precompiled schedule one also gains the advantage of implicit synchronization and one avoids race conditions at runtime.

[Lenz et al.2017] combines the energy saving potential with guarantees of an offline scheduler. The adaptation scheme precomputes possible scenarios, calculate according schedules which the MPSoC may face at runtime and computes schedules for all the scenarios. If the system observes at runtime that a certain assumed scenario occurred the tiles can switch into the according schedule and thus operate more optimized w.r.t. performance and safety.

The global adaptation scheme can further be used to adapt to faults at runtime:

- Faulty nodes that don't respond to messages can be avoided by the rest of the network.
- Message routes with broken links can be rerouted.
- Tasks that ran on a faulty node can be migrated to healthy nodes.

In this section we describe the particularities of such an adaptation for time-triggered systems. As a motivation we want to focus on the energy saving aspect of the adaptation in this section.

5.1 Local vs Global Adaptation

Most energy management schemes for distributed systems are performed dynamically with an online scheduler, which leads them to focus their scope on a tile. By only observing the tiles current state fast scheduling decisions can be taken. We expand this *local* energy management to a *global* energy management scheme. By using the local slack for a global optimization, the network can be optimized to a global rather than just the local optimum of the energy usage at runtime. This all leads to a maximized system lifetime. It is especially applicable for systems with high task interdependencies. In those systems, the output of one tile is needed by other tiles to start their computation. For example, a four tile architecture has dependencies where the first tile computes inputs for the other three tiles. In turn, the other tiles compute inputs for the first tile, as shown in Figure 5.1.

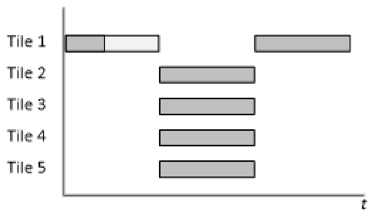

Figure 5.1: Example for task dependencies: We can see a schedule of five tiles. The tiles 2 to 5 are scheduled to wait for a task on tile 1 to finish.

If the first task finishes its execution earlier than the assumed WCET, this slack time can be used for energy optimization. For online scheduler, a currently often used technique is to gradually check the execution time and recalculate how long the remaining task may take to finish. When the monitoring software then recognizes a faster execution than expected, it dynamically reduces the voltage and frequency of the execution. This way, the WCET timeframe is used more efficiently. Unfortunately these online scheduler cannot guarantee that their predicted time will actually be upheld. It is possible that in online scheduled systems the deadlines of tasks cannot be met and tasks fail to finish on time. These type of scheduler are usually only used in non-critical applications like media systems, where failure only cause mild inconveniences but the time saving can enhance the overall user experience.

For safety critical systems failure is not an option. In our proposed system we execute safe frequency adaptations, as we assume possible runtime scenarios, calculate the corresponding schedules and then switch between the precalculated schedules if the runtime conditions are met. All schedules contain excess buffer to compensate for unexpected runtime behavior and can guarantee that the timing restrictions of the tasks are met. In our first example, local optimization would monitor the task execution and once the execution executes fasten than expected, which is indicated by the delta of the expected task status and the runtime task status, it can adapt the frequency on core 1. This would extend the core's execution time under lower frequency to reduce the energy consumption. However, this leads to a longer computation time as shown in Figure 5.2. We cannot just use the slack by sending the message directly to the other tiles because the message injection is bound to the communication schedule. Therefore, in such a local adaptation the goal is to optimize the time slot given by the schedule. This would save energy, but the other cores do not profit from this approach.

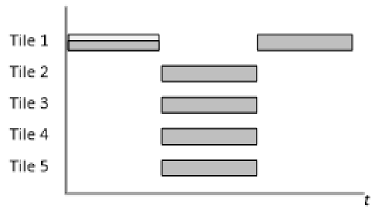

Figure 5.2: Local energy management: When the execution is faster than the schedule's WCET, the tile can locally perform DVFS and lower the execution frequency to finish at the expected WCET avoiding idle times in the scheduled time

In systems with interdependent applications, the later tasks can also profit from faster execution on a predecessor. Once slack is observed on a tile it can be passed to the other four tiles, as Figure 5.3 shows, where the execution can be started earlier, providing more time to finish the tasks. These four tasks could be started right after the first one finishes, therefore giving them more time to execute. The local tasks then can optimize their newly assigned timeslots using the local DVFS. Our approach is to use the slack where the energy savings are best which in this case allows us to multiply the saved energy by four. Not only the energy savings are great for the energy usage aware system: Considering that especially in mixed criticality systems highly critical tasks are planned with a very pessimistic WCET assumption, these slack times unnecessarily block resources for other applications. Reconfiguring these access times on-line is another advantage of our approach.

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

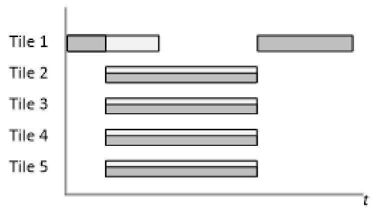

Figure 5.3: Global energy management: The idle time can be relocated to the four waiting tiles. This way these four tiles have a longer time frame and all four tiles can execute their tasks at lower frequencies thus using less energy than anticipated.

message directly into the NoC, and the local schedules of the depending cores have to be changed to start their computations earlier accordingly.

5.2 Schedules

Calculating a schedule is an important task when designing a real time embedded system. Basically, the schedule provides information about when and where a task can be executed without violating resource restrictions, dependency relations and timing constraints. The *schedule* is a function which temporally and spatially allocates a set of executable actions to a physical model and a period w.r.t. their deadlines and dependencies. The executable actions are those that need to be performed within a defined period of time by a defined set of hardware. In our case we look at two kinds of schedules: the computational schedule which defines when applications are to be started on the cores and the communication schedule for the message injection. These schedules depend on each other as the messages to be sent are computed by the tasks managed by the hypervisor. The computational schedule is computed for the tasks, w.r.t. their start and their end time and their criticality. The communication schedule contains all messages that will be send to the NoC and their injection times.

The schedule assumes the *Worst Case Execution Time (WCET)* when planning the order of task executions. This assures enough time for all applications to finish the task in the assigned time frame. In mixed criticality systems, WCET assumptions can vary based on the criticality level. Low criticality tasks can be assigned an optimistic time slot, as faults are acceptable here. High criticality tasks will always be assigned pessimistic time frames, as the tasks have to be performed at all costs. Reassigning allocations too early will cause fatal system failures. To ensure meeting the deadline the WCET assumption is always an overestimation. This is shown in Figure 5.4. In case the task actually finishes at the WCET the overestimation saves the system from deadline misses. However, the figure also shows that the usual execution time is often within the first third.

The difference between the estimated execution time and the real execution time is called slack and can only be determined at runtime. This makes an optimi-

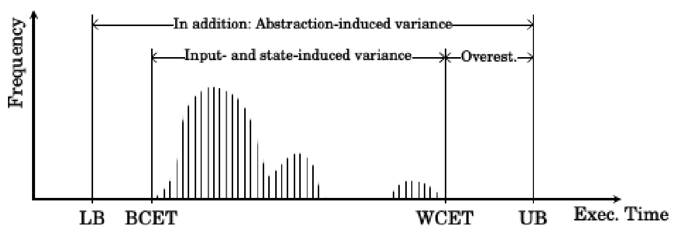

Figure 5.4: WCET distribution [Axer et al.2014]

zation impossible in static, prescheduled systems.

The communication schedule depends on the computation schedule: it awaits and plans a message at the expected end-time of a computation, which must be approximated a priori with the WCET. In turn, even if a computation is finished before the WCET, the resulting message will not be transmitted directly but only after the WCET has elapsed. This dependency can be interpreted as a time contract where the communication commits to send messages in provided time slots. Any changes on the system behavior must comply to these predefined ”contractual” times. State of the art power management schemes use the inevitable slack resulting from the difference between the WCET and the real execution time to perform DVFS within the tile. These schemes ensure that the prolonged execution due to DVFS ends by the assumed WCET to comply with the communication schedule. Such software based approaches are usually performed by the hypervisor or the partitioning software (RTOS).

5.2.1 Mapping the Application to the Platform

Our thesis contribution operates on the assumption that it receives schedules as an input. For testing purposes and to evaluate how different schedules impact on the thesis contribution we have adapted a genetic scheduler which we will explain in this subsection.

We choose a non-preemptive, static scheduling model similar to *partitioned scheduling*. As discussed in [Guan2016], a non-preemptive scheduling model has advantages on multi-core systems or distributed systems, as the overhead for migrating a task is more difficult to predict in case of preemptive scheduling. The disadvantage of non-preemptive tasks reducing the responsiveness of a system is not valid for those systems, since the natural parallelism of such a system can hide latencies [Guan2016]. In our model tasks are allocated to exactly one computing tile. They may be migrated at the beginning of the period, iff the assigned tile is unavailable for some reason. However, tiles are not chosen at random as the allocation has a significant impact on the temporal behavior. To maintain precedence relations and predictable communication times, the allocation must follow the following rules.

An allocation function is a mapping $A : T \rightarrow C$ which maps tasks to computation tiles. We require that for every channel $e = (t,t') \in E$ and $A(t) \neq A(t')$ there exists a path of the form $A(t), r_1, \ldots, r_n, A(t')$ in the network Net, where $n \geq 1, r_1, \ldots, r_n \in R, (A(t), r_1) \in L, (r_i, r_{i+1}) \in L$ for all $1 \leq i \leq n-1, (r_n, A(t')) \in L$, and $r_i \neq r_j$ for $i \neq j$ with R being the set of all routers r_i and L being the set of all links l_i. For further consideration we fix such a path and denote it with $A(e)$. Note, that if more than one path exists for a single channel we enumerate them as $A^0(e), A^1(e), \ldots, A^{n-1}(e)$.

Every path $A(e)$ has the following properties relevant for scheduling:

- it starts and ends with a computation tile,
- only routers are allowed between those computation tile,
- it is simple, i.e., a tile does not appear twice on the path, and
- its length (number of tiles on the path) is between 3 and $|R| + 2$.

A path $A(e)$ can intersect another path $A(e')$ in one or more computation tiles or routers. In order to avoid conflicts, one has to make sure, that the shared resources are used in mutual exclusion.

5.2.2 Genetic Algorithm for Scheduling

We implemented a genetic algorithm to solve this scheduling problem. Introduced by Holland [Holland1992], genetic algorithms are a long known method in the field of metaheuristic optimization. Compared to so-called optimal methods like *mixed-integer linear programming*, metaheuristic methods tend to have a better scalability and typically provide feasible solutions for the scheduling problem in a shorter time than optimal methods. These advantages come with a loss of optimality and the non-deterministic behavior. A genetic algorithm can deliver the optimal solution or the worst one (or anything in between).

Nowadays, a variety of libraries and frameworks are available that help programmers and scientist to implement genetic algorithms. However, the major parts like designing a genome prototype and defining a fitness-function must be done manually. In this section we will describe the details of our genomes and evaluation functions.

Genomes

The genome defines how many genes we can use and which values they can attain. The combination of all possible manifestations of these genes is called the *search space*, while a specific manifestation is called a *genotype*. A genotype encodes a possible solution, i.e. a schedule, which is called a *phenotype*. The design of the genome is highly problem-specific, so there is no ready-made solution for this task. However, there are some rules to consider:

- similar genotypes should have a similar fitness

- similar genotypes should represent similar phenotypes
- the search space should be closed under the used genetic operators

A genome must contain all information that is necessary to decode the genotype into a phenotype. For our scheduling problem this means that the genome must at least hold information about allocation, route selection and frequency scaling.

$$\underbrace{\boxed{t_0}\boxed{t_1}\boxed{t_2}}_{\text{Allocation}}\underbrace{\boxed{o}}_{\text{Order}}\underbrace{\boxed{e_0}\boxed{e_1}}_{\text{Path}}\underbrace{\boxed{e_0}\boxed{e_1}}_{\text{Frequency}}$$

Considering that, our genome is defined as follows: For each task we have one gene holding the *ID* of the computing tile this task is allocated to. Each gene can only attain the IDs of the computing tiles the task is allowed to run on. One additional gene is used to choose a topological order of the communication tree. This allows the algorithm to vary the processing order when calculating the phenotype. The path-selection is done on a per-channel basis. Each channel e has its own gene determining the path $A^i(e)$ that is used for communication. The genes can attain integral numbers between 0 and k, where k is the maximum number of paths that exist between *any* two computing tiles. The last genes of the genome are used for frequency scaling. For each task they contain the frequency that a computing tile should run on while processing this task. Putting all together the length of the genome, and thus of each genotype, is given by $2 \cdot |T| + 2 \cdot |E| + 1$, the sum of the tasks and their corresponding frequencies and the communication channels which are defined by a gene tupel of the beginning and the end of a channel plus the single digit topology code of the network. The length of the genome therefore scales linear with the application model.

Fitness Function

A fitness function transforms a genotype into the corresponding phenotype and calculates the phenotype's fitness according to the objective. In our case the objective is minimizing the overall dynamic energy consumption $\sum_{t \in T} E(t, \eta^t)$ restricted by the deadlines of the tasks, here calculated as the sum of the energy consumption η^t of all tasks $t \in T$. If a schedule cannot guarantee these deadlines, the schedule is called *invalid*.

As the fitness is always calculated at the phenotype level, the genotype has to be translated into a phenotype. Algorithm 1 works as follows: The tasks are initialized (line 5) with information we already have from the model and the genotype. This includes setting the actual execution time according to the task's WCET, the frequency scaling parameter given by the genotype handled by this task. Also, every task's start time is set to 0. The initialization of messages is quite similar (line 6). All start times are set to 0. he path is set based on the allocation of the source- and target tiles and the path number stated in the genotype.

Algorithm 1 Phenotype implementation

```
function TO_PHENOTYPE(genotype)
    valid ← true
    makespan ← 0
    energy ← 0
    INIT_JOBS(genotype)
    INIT_MSGS(genotype)
    for $m \leftarrow 0$ to $size(\text{MSGS})$ do
        sender ← m.sender
        recv ← m.receiver
        FINDSLOT(sender)
        FINDSLOT(m, sender.finish)
        UPDATE_RECEIVER(m.arrival)
        UPDATE_MAKESPAN(m.arrival, recv.finish)
    end for
    for $j \leftarrow 0$ to $size(\text{JOBS})$ do do
        energy ← energy + $j$.energy
        if $j$ has no slot then
            FINDSLOT(j)
        end if
        if $j$.deadline < makespan then
            valid ← false
        end if
    end for
    return $\{\text{JOBS}, \text{MSGS}, \text{makespan}, \text{energy}, \text{valid}\}$
end function
```

For calculating the actual start- and finish times, the messages are processed in the order given by the dedicated gene (line 7). This incremental way ensures that all precedence constraints are met by every phenotype. At first, we need to find an execution slot for the sending task (line 10). The function will set the task's start time to the beginning of the first time slot fitting the timing requirements. If the start time is not 0 when entering the function, the slot will not start before the given start time (line 11). Also, if the task already has a slot from processing another message, we will not allocate a new slot. Slots are allocated on a per-tile basis, so that two tasks may run at the same time but on two different computing tiles. In turn, multiple tasks can run at the tile but at different times. As we know every task's WCET, we can easily calculate its finish time. Remember that all tasks are non-preemptive. The message we are currently processing needs a slot on the network that starts after the execution of the task is completed. The function considers that messages cannot be buffered along the path, but only at the end-points. This makes the arrival of the message predictable, so that we can update the receiver's start time (line 12). This function initialized the receiver part of the genome tupel that encodes the communication channel with its receiving node and the duration of the transmission. Processing the messages in a topological order ensures, that chaining the receiver's start time will not invalidate any slots allocated to jobs or other messages, as neither the receiving task nor messages sent by it were processed before.

Tasks that do not send any messages must be processed in a separate loop. The jobs are assigned to a slot on the execution nodes, As these jobs do not rely on network wide dependencies of messages we can assign them in a second loop after the network traffic is planned. This is the place where we also add up the consumed energy (line 15). The energy is determined based on the genomes cores frequency and the duration of the task under the assumption of the thesis' power model. The phenotype is then returned to the fitness function shown in algorithm 2 where the energy field is used as the fitness.

Algorithm 2 Fitness function

```
1: function FITNESS(genotype)
2:     schedule ← TO_PHENOTYPE(genotype)
3:     if schedule is invalid then
4:         return $\infty$
5:     end if
6:     return $\text{schedule.energy}^2 \cdot \text{schedule.makespan}$
7: end function
```

5.3 Global Adaptation

Seeing the restrictive nature of schedules and the idle times created by the static and dynamic slack it becomes apparent that time-triggered systems can by definition not be run at optimal capacity as no guarantees can be given when the schedules are computed without the safety time period.

To fill this need for adaptive behavior while at the same time providing safety and predictable system behavior the global adaptation is introduced by this thesis [Lenz et al.2017]. Global adaptation is a *quasi dynamic* adaptation technique for time triggered pre-scheduled systems. It provides a set of safe schedules that can be switched between at runtime while guaranteeing that the time constraints given by the application are upheld. The system thus is enabled to observe the runtime state and change its current schedules to optimize the runtime performance w.r.t. energy or to enhance the system's safety argument.

The schedules are prepared by a meta scheduler which computes a schedule graph containing these possible runtime schedules. Each graph begins with a baseline schedule that operates the system and from there the graph fans out based on assumed runtime events that if happened will set the system into a new schedule which in turn is optimized for the new system restriction that the occurring event ensued.

5.4 Meta-Scheduling

The goal of meta-scheduling is to calculate a set of applicable schedules for a period that can be switched between at runtime. These schedules ensure that the application deadlines are met at runtime while at the same time optimize the system performance w.r.t the system goal parameter. The meta scheduler therefore is given a physical model (PM), an application model (AM) and a context model (CM) that define the hardware that is available, the tasks and their deadlines and interdependencies and the system runtime events that can be observed.

The context model defines which events are considered for possible runtime adaptation and at which time during the period they are likely to occur. After compiling a baseline schedule with the AM and PM the meta scheduler will go through the schedule and step by step assume that the context event happened, to calculate how the rest of the period need to be scheduled for an optimized energy usage. This will lead to a set of binary decision paths that can be walked through at runtime depending on if the CM events happened or not. This graph is the schedule graph that is used at runtime by the adaptation.

Schedule Graphs G= (V,E) are directed acyclic graphs (DAG)s that consist of the set of *vertices V* that represent all schedules that can be executed during runtime and the *edges E* that represent the context under which the schedule is changed into the edge's end vertex. Each vertex represents a schedule that the system can switch into at the time at which the corresponding context was observed. A path within this graph therefore represents a series of context events c_0 to c_t that need to occur to reach the vertex V_t from the beginning vertex V_0.

Branching Points Within a schedule graph we have a finite set of *branching points*, points in time within the execution period where potential schedule changes can be done. At each branching point the adaptation has at least two possible branches to follow, each associated with a distinct context, indicating that the event linked to the context occurred or not. They are calculated by the meta-scheduler. The system state must be agreed on right before the branching points to have the most updated view on all tiles' status. Especially with context values that quickly lose their validity as for example slack.

In such cases it is essential that the context is made available as close to the branching point as possible. If for example slack of $20\mu s$ was reported and agreed on at a time t but the branching point which tries to use the slack only happens at time $t + 18\mu s$, the system will not be able to use full potential of the observed slack. While the remaining $2\mu s$ still can be used for adaptation a better scheduling of the agreement w.r.t. the branching points can optimize the system's performance.

To prevent delays which could negatively impact these events' optimization potential, it is important to run the agreement protocol at a high frequency. The more frequent the tiles are updated on the current state, the 'fresher' the view on the system is at the actual branching point time.

We differentiate between *branching points* and *mode changes*. A branching point denotes a point within a periods schedule graph that can lead to a better runtime performance by adapting the rest of the period time. The mode change leads the system into a different execution mode, which saves larger amounts of energy by performing low power techniques that cannot be quickly changes during a period. This includes moving tasks from one node to another or shutting down areas of the NoC completely. The mode change leads to a degraded behavior once the new period starts, as the changes are performed once to save as much energy as possible.

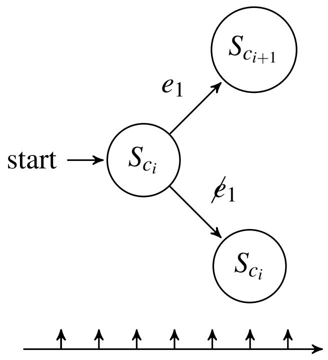

Figure 5.5: The branching point marks an instant in time where a schedule switch can be executed. Based on the context event the new schedule can be switched into or the current schedule will be kept.

5.4.1 State Conversion to Reduce State Explosion

By applying the adaptation globally the potential schedule space can grow exponentially with the number of events. If one were to mimic a dynamic energy management each fine grained context change would lead to a dedicated new schedule. Such a system would collide with the initial plan to save energy, as the management would introduce an overhead in energy consumption. Therefore we designed two mechanisms to avoid state explosion, as proposed in [Lenz et al.2017]:

- discrete context intervals
- forced convergence of the schedules

Discrete Context Intervals

The first mechanism is controlling the amount of the context that the system can react to. If certain execution times are passed and the slack is too small to perform a global adaptation, these intervals are ignored by our system and the local adaptation can use the slack. Within the viable slacks we define discrete intervals, e.g. if the monitored slack is within 50%-30% we will jump into a new optimized schedule. Figure 5.6 shows how a baseline schedule can lead into 3 possible slack optimization states.

The boundary for context can be set according to the designed system. If many other context variables are already observed, the slack options can be reduced. If the context will be defined solely by slack, more specific slack options and reactions can be modeled. We keep our system intentionally as generic as possible to be applicable to as many applications as possible.

Schedule Convergence

The second way to reduce the potential schedules is to force a schedule convergence by bounding the amount of tasks that can profit from the available slack. To realize that we must look at the different ways the meta-scheduler can evolve:

1. the context implies that the whole system mode has to change to assume a low energy mode, e.g. battery status.
2. the context implies that an advantage can be reached by changing the following tasks' execution, e.g. by applying DVFS or gating specific areas for some time.

The number of additional states generated by the first context type is just one. When a low battery status will be reached, the whole system will go into a low power mode defined with a low power schedule. There will be no further changes in the state to save as much energy as possible and prolong the system's lifetime until maintenance can recharge the battery. If a rechargeable battery is used the low power mode can be left and a normal schedule can be applied when a satisfying threshold of battery capacity is restored. In the low power state itself the adaptive services will be deactivated to save as much energy as possible.

The second context type is generated by introducing *windows* that can be used to apply low power techniques. Theoretically such additional time can be used by all following tasks, leading to exponential growth of the meta-schedule states. By bounding the states by restricting the amount of tasks that can benefit from such a context change to a value c, we can force the schedules to converge after c units of time. This may also bound the impact the changes can have. Additionally, it may use e.g. slack times as soon as possible, rather than where they could be used best. However, with the volatile system context, we cannot assume that the best task to benefit from slack times is in the near future.

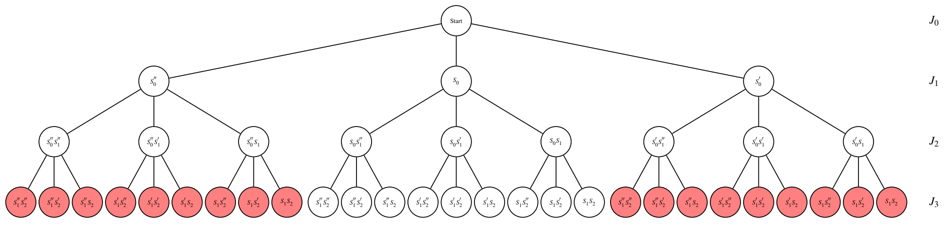

Figure 5.6: Bounded expansion of possible states

As an example we choose a schedule which can be changed based on three slack options. If each task can pass three possibilities of slack s to the next ones, the resulting tree grows exponentially according to the number of states and tasks (s_j). The problem is not scalable in this way, therefore the passing of slack is bounded to the next two tasks ($c = 2$). Then, the tree grows in the same way as before until task three has been reached. At task four, the convergence occurs. Regarding each subtree of the tiles of level J_1, the following paths are equal for each of the three tiles. If J_1 passes possibility S''_1 to J_2 and J_2 passes S''_2 to J_3, the path in the tree is

$$S_0 \rightarrow S_0S''_1 \rightarrow S''_1S''_2 \tag{5.1}$$

which is the left path in the subtree. The paths for the other subtrees are similar:

$$S_0'' \rightarrow S''_0S''_1 \rightarrow S''_1S''_2 \wedge S'_0 \rightarrow S'_0S''_1 \rightarrow S''_1S''_2 \tag{5.2}$$

all ending in the same state $S_1S''_2$. It is shown in Figure 5.6, that this is also true for the other states in layer J_3 printed in red. Each of those states has a related state in the subtree of S_0.

We can formulate the state expansion with the following equation:

$$N = \sum_{i=0}^{c-1} i + s^c(j-c) \tag{5.3}$$

Until level $c-1$ is reached with c as the convergence constant, the total number of states can be calculated as the sum over s_i. The variable s represents the number of supported context changes, in this case the number of slack possibilities. As soon as level c has been reached, the number of possible states on each level is equal to s^c. Then, the number of tasks can be reduced by the convergence constant as they are already regarded in the exponential part. The resulting number of residual tasks can be multiplied with the number of states per level.

At runtime, the current schedule is known by the network interface where the serialization units injects the time-triggered messages into the NoC according to the scheduled message injection times. Our NI receives the context information, decides on changes and changes this particular schedule into the new one.

5.5 Proof of Concept Evaluation

At the beginning we focused on evaluating if a global adaptation would be applicable in time triggered embedded systems. We simulated a time-triggered system as it was used in previous projects and applied the adaptation principles.

This section focuses on the simulation we used to test our assumptions and its results.

5.5.1 Gem5 Simulation

We used Gem5 [Binkert2011] to verify our assumption and set up a simulation based on the use case shown in Figure 5.7. Gem5 allows the user to model the NoC behavior with a high degree of freedom. It provides a high variety of NoC model which can be configured freely to the users needs. We were able to model a system which simulated exactly our system models behavior. Other simulators only provided a pre-configured configurations which were not as applicable to our source-based routing system. For the use-case we chose a 2x2 Mesh NoC with 4 tiles connected to it. We only used 2 of the four tiles for the applications, because we wanted to test the path finding of our scheduler at the same time. The tiles A and B are located In the northwest and southeast of the NoC, and x-y routing is used to determine the message paths. Figure 5.7 shows that we have two applications, the tasks t_0, t_1, t_2 and t_3 are high critical tasks while t_4 and t_5 are low critical. We also modeled inner-tile and inter-tile dependencies to proof our slack propagation assumption: t_2 depends on t_0 and t_3 depends on t_2 and t_1. In the figure, the numbers in the tiles and those associated to the messages are their assumed WCET time for computation and message transmission.

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 10ns, based on the transmission duration between two routers. Therefore, we assumed that we have 108 ticks per second. Usually the fpga consumption is highly dependent on the layout of the fpga and the runtime behavior as temperature and the transmission paths all impact to the overall energy usage. But to have a starting point for our evaluation we computed an upper bound for the per tick energy consumption of an FPGA we took the maximum power value of 30 Joule and divided it by the ticks per second, giving us the per tick energy consumption of 3×10^{-7} Joule (JF). To monitor the simulated consumption, we add this value for each tick in which a component is running.

Because Gem5 does not support DVFS, a simpler model must be applied to measure the energy consumption in the NoC. Our power model is an approximation of the consumed power in the NoC based on the known behavior of CMOS-circuits. As shown in Equation 5.4, the consumption equals the sum of the transient power consumption P_T and the capacitive-load power consumption P_L [Sarwar1997]. Both terms are calculated in a similar way and have the common factors of the squared supply voltage V_{CC}^2 and the number of bits switching N_{SW}. While the transient power consumption further depends on the dynamic power-dissipation capacitance C_{pd} and the input signal frequency f_i, the capacitive-load power consumption depends on different factors. Those are the external load capacitance C_L and output signal frequency f_O.

$$P_D = P_T + P_L = C_{pd} \cdot V_{CC}^2 \cdot f_I \cdot N_{SW} + C_L \cdot V_{CC}^2 \cdot f_O \cdot N_{SW} \tag{5.4}$$

Our model assumes that the input and the output signal frequencies in equation 5.4 are equal. Hence, the dynamic power consumption can be calculated by equation 5.5.

$$P_D = \left(C_{pd} \cdot V_{CC}^2 \cdot N_{SW} + C_L \cdot V_{CC}^2 \cdot N_{SW}\right) \cdot f \tag{5.5}$$

In the simulation, the factor in braces in equation 5.5 is not exactly determined but approximated by a constant. The value is set to 1 to simplify the model. At each tick, the constant is scaled by the current frequency in the NoC and added up. This frequency is adapted by the adaptive communication at defined instants as explained in the previous sections.

The frequency f in the NoC is adapted by the adaptive communication at defined instants. For each tick, the constant is scaled by the current frequency in the NoC to calculate the associated energy consumption.

We established a power model including idle and running states of the tiles for the normal and the low power frequency. To compute the overall energy consumption, the associated energy value is summed up each tick. The traces that were used on the tile are created by assuming a uniform probability of all execution times to simulate the unpredictable tile behavior. We compared our results to a baseline configuration. In this configuration, we let the tile run in idle mode until the WCET is finished.

To calculate the energy usage of the adaptation, we add the Δ energy value which is associated to the current schedule-change to the baseline schedule.

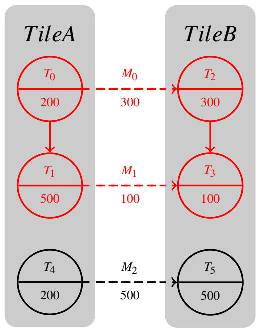

Figure 5.7: Use case

5.5.2 Simulation results

The simulations of the adaptive communication in this section show the correct behavior and the possibility of saving energy. For this, the system is configured according to the use case presented in Figure 5.7 and the previous section. There are three different simulations presented with an approximation of the resulting energy consumption. The tile's message injection is defined such that in the first case there are slack traces used that have more that 50% slack for the tasks and traces with less than 50% slack per task in the second one. Switching off the adaptive communication in the other cases enables a comparison with the baseline NI.

Each table printed in the following shows the message traffic for the cases. The first column shows the message ID. Columns two and five show the de- and en-queuing instants of the Cores. When a NoC specific guarding window, which ensures the timing guarantees by providing collision free message passing, is enabled, the message can be enqueued into the NoC directly. This instant is presented in the third column. The fourth column contains the instants when the last flit of the message leaves the NoC. In each case, first all messages are sent in the first application mode where all tasks are enabled. Finally the last column shows the total communication delay passing from dequeuing the message from the source tile and en-queuing into the destination tile. After a period passes, the application mode changes and the low-battery mode is executed. Then, only messages 0 and 100 are sent.

Table 5.1 and table 5.1 show the message traffic for the disabled adaptive communication. Table 5.1 shows the communication version where only a small slack time is observed, while Table 5.2 shows the message injection with a high slack occurrence. As the injection is performed based on the precomputed communication schedule the messages are injected at the instances 200, 400 and 900. The

Table 5.1: Message traffic for few slack case with adaptation disabled

Msg ID	Runtime Deq Tile at tic	Scheduled En NoC at tic	Scheduled Deq NoC at tic	En Tile at tic	Total Delay in tics
0	151	201	210	219	68
200	390	401	410	419	29
100	890	901	910	919	29
0	1618	1701	1710	1719	101
100	2118	2201	2210	2219	101

Table 5.2: Message traffic for high slack case with adaptation disabled

Msg ID	Runtime Deq Tile at tic	Scheduled En NoC at tic	Scheduled Deq NoC at tic	En Tile at tic	Total Delay in tics
0	8	201	210	219	211
200	390	401	410	419	29
100	890	901	910	919	29
0	1618	1701	1710	1719	101
100	2118	2201	2210	2219	101

total communication delays for message zero show, that due to dynamic slack the delays are much greater than those of messages 200 and 100. This difference can be used to apply power saving techniques. Changing the application mode at the end of the period occurs at instant 1500. Therefore, the phases of the safety critical messages 0 and 100 are 200 and 700.

Table 5.3: Message traffic for much slack with adaptation enabled

Msg ID	Runtime Deq Tile at tic	Adapted En NoC at tic	Adapted Deq NoC at tic	En Tile at tic	Total Delay in tics
0	8	12	21	30	22
200	390	404	413	422	32
100	890	901	910	919	29
0	1618	1701	1710	1719	101
100	2118	2201	2210	2219	101

In the following, the adaptive communication is enabled. Table 5.3 presents the case with much dynamic slack. Message 0 is available at the tile at tick 8 resulting in slack of 192 ticks. The matching interval end is tick 10 which enables the sending at tick 11 instead of tick 200. DVFS results in a longer communication delays such that message 200 had to be sent 3 ticks later at tick 404. At message 100, the convergence leads to a transmission at the specified instant. In the low-battery mode, all messages are sent with the lowest supported frequency and no further changes due to dynamic slack are specified.

Table 5.4 shows the behavior with few dynamic slack. Message 0 is dequeued from the tile at tick 151. Then, the matching schedule change in this case triggers the transmission at tick 166 and delays the next message by 33 ticks.

Table 5.4: Message statistics for few slack with adaptation enabled

Msg ID	Runtime Deq Tile at tic	Adapted En NoC at tic	Adapted Deq NoC at tic	En Tile at tic	Total Delay in tics
0	151	167	176	185	34
200	390	434	443	452	62
100	890	901	910	919	29
0	1618	1701	1710	1719	101
100	2118	2201	2210	2219	101

Figure 5.8 shows the resulting power consumption. The x-axis shows the consumed energy in J while on the y-axis, the different application modes are presented. In each mode, the case with the adaptive communication disabled is printed in dark gray. On the other hand, the case with few slack is colored in light gray and the one with much slack in white.

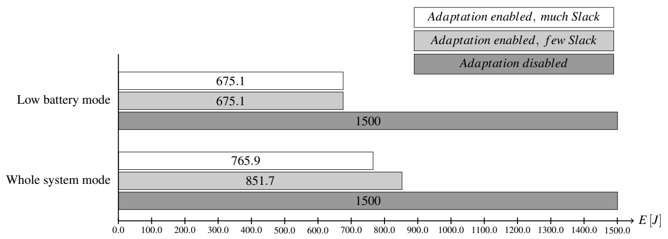

Figure 5.8: Energy consumption in simulation

Since all activities are executed with a frequency of 100% when the adaptive communication is disabled, the consumed energy amounts to $1500W$. In the whole system mode, the frequency the messages are sent with is adapted when the message transmissions start. As shown in the chart, the adaptation enables power savings of $648.3W$ for few slack and $734.1W$ for much slack, respectively. This equals 43.22% respectively 48.94% of the consumed power when the adaptation is disabled. On the other hand, in the low battery mode the frequency is set to 45% at phase zero and not changed for the complete period. This results in an energy consumption of $675.1W$ independent from the available dynamic slack which corresponds to 45% of the consumed energy without adaptation.

The results show, that adapting the communication at run-time to apply low-power techniques has the potential to save energy of more than 50% if there is sufficient slack available. However, the simulation does not consider the reduction of the supply voltage. Because there is a linear relationship between frequency and supply voltage and a cubic relation between frequency and power, adjusting the voltage can lead to higher energy savings in future optimizations.

Chapter 6

Agreement Protocol

The basic protocol idea follows the principle of periodically observing the current context of the SoC and changing into a schedule that is best suited for the observed context. The schedules are computed for the global scope, meaning that also the inter-tile communication schedules are adapted to fit the new execution times. These changes must be taken at all tiles unanimously as the communication might fail otherwise. If for example one tile does not change into the same schedule as the others, it may fail to receive the messages correctly and cause fatal deadline misses.

Since the system model considers a NoC with an arbitrary amount of connected tiles, observing the global state is quite difficult as each tile only has knowledge of its directly connected monitors and of its own computation status. This section introduces an agreement protocol that establishes a global view within all tiles of the SoC.

The protocol we propose is a broadcast type protocol that distributes the local context to all cores. This way, all cores know the context of all other tiles and share an 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 is realized in hardware within each NI where the context can be used by the mode-change state machine to perform eventual adaptations.

It is performed in three stages: *reporting*, *agreement* and finally the *convergence*.

During the reporting stage the local context monitors report their observed system state to the agreement unit via dedicated ports.

Once the agreement phase starts the ports are read and the information is exchanged via the network to the other tiles. This means that each tile broadcasts the information at the same time to have a consistent view after the transmission time of a broadcast. Each tile sends its information at the same time. Hence, to prevent message collisions the context exchange is performed in a ring, where each tile is sending messages to a successor in the ring and receiving messages from another tile, its predecessor. Each tile sends its own context values from the observing period to the next neighbor and receives its predecessors values. Once received the predecessor's values are saved locally by the agreement layer and sent to the next neighbor. After n rounds, with n being the amount of tiles connected to the NoC, of such receive and relay operations the tile will receive its original message as a relay copy from its predecessor. At this point the tile has received all tiles' context information and thus can merge the messages to a complete view on the system state. The system state at time t is thus globally know at the time $t + n*$(time for transmission).

The convergence stage can be optional depending on the values that the system agreed on. If some values were observed by two tiles a convergence function can be run on these redundant values to ensure that all tiles operate on the same data set regarding the global state. The final context string is then relayed to the adaptation unit for further assessment w.r.t. possible schedule changes.

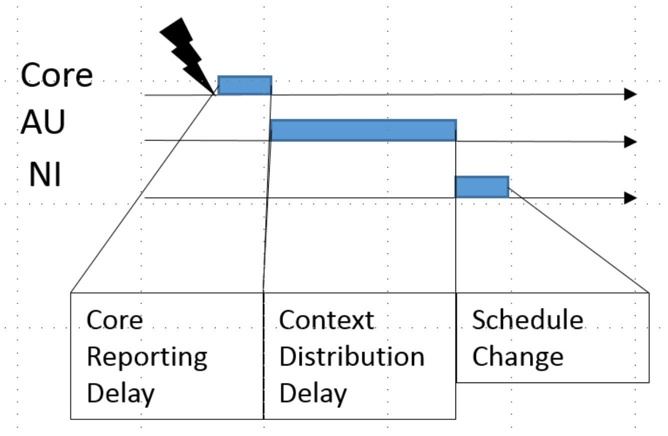

Figure 6.1: Timeline showing how the event detection on the core and the actual agreement process relate to each other. The context is reported by the core as soon as the corresponding event happened. The agreement will be triggered as scheduled within the AU, thus creating a time offset between the context occurrence and the actual agreement on the context. When all tiles are aware of the global state the AU relays the global context information to the adaptation manager, where a potential change will be performed. Once the AM made a decision the whole system will be informed of the new state.

In many NoC implementations the programming paradigm strifes to minimize the NoC size and keeping the routers to a minimum size is a key factor. The routers are only able to perform basic transmission tasks to single destinations and are not able to buffer messages or resend them to multiple receivers. Until now no need was shown for broadcast protocols. Workaround solutions at the moment include a data stream that circles through the network and is read by all tiles while it crosses their network interface [Obermaisser et al.2008].

To implement the agreement protocol where each tile will notify all other tiles of their current state generates an inevitable overhead in additional messages which block the normal communication. While the regular NoC structure can be used for the agreement we have established in experiments with synthetic use cases and NoC benchmarks, ref 7.5, that a dedicated network is better suited for such a frequently executed protocol. We design this dedicated network as a ring where all tiles are connected to a single predecessor and a single successor like a ring structure.

Algorithm 4 Protocol algorithm for the context distribution.

```
1: $state_l$ = collect locally observed context string
2: if $t = time\ to\ start\ agreement\ phase$ then
3:     send local context $state_l$ to ring neighbor
4:     for amount of $tiles - 1$ do
5:         if context from neighbor is received then
6:             save information from neighbor $state_{l-1}$ locally
7:             relay neighbors context information $state_{l-1}$
8:         end if
9:     end for
10: end if
```

6.1 Protocol Phases: Reporting

The protocol starts with the collection of context. Each core has a set of context monitors directly connected to the NI where they send their observed local context.

The agreement protocol is used to broadcast the information needed for the system state just in time to the schedule change evaluation at the next branching point. This is done to make sure that the values associated with the events are as up to date as possible when a decision is taken.

To avoid that failures within the application or the context observing hardware may affect the agreement unit, we only poll the information and do not directly react as soon as a context information is reported.

The system is set to observe all events that are defined at design time as the context. These events must be reportable in some way, meaning that there must be a value that can be observed from the outside that reflects if an event happened.

The system state at time t is therefore defined by the sequence of events that occurred at runtime until t. This unique coding of the system state is also taken as a branch id for the adaptation as explained in [Lenz et al.2017]. As we use the events to navigate the schedule DAG, we do not need to exchange the complete system state with all events that happened at runtime, we can rather reduce the exchanged information to the context information that was observed after the last branching point. All foregoing information is known to the system based on the current state within the DAG.

6.1.1 Types of Events

The events a system is able to react to depend on the design and there are no restrictions as long as the event can be observed and reported on. Having such a free range of possibilities we identified two general types of events that need to be handled differently and have different implications on the agreement protocol and its execution.

The two types of observable events are

- *Preplanned (Synchronous) Events*: These events occur on a regular basis at a rate that can be expected beforehand. While the exact occurrence cannot be predicted, an estimation on when it occurs can be given. One example for synchronous events is slack. We can tell when an occurrence of 50% slack, meaning that the task finished at half the expected WCET, can be observed. We do not know if it will be observed, but we know when we need to check for it.

- *Unplanned (Asynchronous) Events*: Events that cannot be predicted are called asynchronous events. We have no indicator when they might occur and therefore cannot predict when we need to check for them. Such events include hardware faults.

The *synchronous* events are preplanned. They include all possibilities that are considered in the design in hope they will occur as they can be exploited to optimize the system performance. These events are bound to certain times and thus can be polled easily with little delay. Each synchronous event will thus be known in the system with minimal delay and can be exploited for the system's benefit.

Asynchronous events cannot be planned. They are events that can occur at any moment, by nature these events are fault driving events, as unplanned events in time-triggered systems are always connected to unplanned states that need to be avoided at all cost. These events need to be reacted to as soon as possible. For system design that use the adaptation for fault tolerance a minimal delay can avoid a critical failure and enable a faster establishment of a safe state. Asynchronous events can only be timely observed by active polling if the AU checks for the faults continuously and reports them as soon one is found. Within the time-triggered execution and communication of our system model this event driven messaging system is difficult to implement, as a constant bandwidth within the network would need to be allocated for potential fault notifications. Further the schedule DAG would need to be expanded by a safe mode switch from any moment in time to a degraded mode that is safe. The better the systems should to adapt to faults, the more schedule change possibilities need to be provided to the system.

.

6.1.2 Context Monitor Types

The context is reported by *Context Monitors*: Dedicated IPs are used to collect sensor values and convert them into a format that the dispatcher can interpret. Examples for a context monitor are temperature sensors that observe the SoC's temperature or the environmental temperature. The temperature of the SoC can indicate if a tile is running too hot and thus risks a failure due to overheating. It was shown that such hot-spots increase the systems energy usage and failure rate [Hong1999], which is why re-assigning tasks away from these hot spots helps to reduce the potential threat. The environmental temperature can be especially vital for SoCs that are used in open areas and can suffer from direct exposure to sunlight.

Applications can provide context reporting functionalities by writing information into the dedicated reporting ports. The communication drivers must provide according functionalities for the application to use them. Using the reporting the application can inform the AU when it finished a task.

Each tile can have an arbitrary amount of CMs associated to the tile. All context monitors are dedicated to a single context type which they interpret. The source of the context can vary between hardware sensors that are directly connected to the CM, or software feedback that the context monitor polls from the hypervisor or information that is reported by the software running on the tiles and partitions. The reported values only reflect a current status on the context.

The context monitor is aware of the meaning of the incoming values and can interpret what e.g. a certain temperature means for the tile and the whole system. It will not send the whole temperature value to all other tiles but rather inform about a discrete interpretation of the value. For most context types we can encode a binary code for the values as the system only cares about negative implications which need to be avoided and can be mapped to events like a low battery or high temperature. These events can be presented based on a threshold that is either passed by the observed temperature sensor value or not.

The functionality of the CMs is depending on the functionality and application that is performed on the SoC and can be expanded to include a more detailed state description. The more details can be reported by the CM the more adaptation possibilities arise for the system. Designing the CMs therefore is a trade-off between the better adaptation possibilities and the overhead created by having more adaptation possibilities including more schedules, more observations and more communication efforts. Every system design will have a *sweet spot* where the advantages will be maximized for the associated cost.

Each context monitor has a dedicated port within the NI that the AU can access when needed. The port is designed as a buffer that gets overwritten by each new information. The port design is based on the hardware block design of the extension layer which is used to implement time-triggered behavior on source-base NoCs [Ahmadian; H.; Obermaisser2015]. The port is read as soon as the agreement phase of the protocol starts, before that the port can be rewritten by the CMs as many times as needed. This ensures that only the newest context information is used for the adaptation no matter when an agreement phase starts. On the other hand the timed port extraction means that any information that is written into the port is read at a certain time. This can lead to a delay that renders highly volatile events like slack useless if the reading delay is longer that the observed slack time.

6.2 Protocol Phase: Agreement

Nodes will collect context information until the agreement phase starts. Then the AU will read the ports dedicated to the context monitors and merge them into a local context string. As mentioned before the CM can be reporting a one bit value in case the occurrence of the event is important, or use more bits to send the whole context value. The interface between the CM and the AU must be accurately defined, as the local context string will be constructed based on the predefined bit amount for the context values. We cannot dynamically assign the values as the local information of each tile must be interpretable by all the other tiles. Dynamic messages with variable context sizes would need a second agreement round where a message mask would be exchanged that enables the other tiles to interpret the incoming information. By predefining the structure of the message each tile can save a message mask for all other tiles during the initialization phase.

Once the local context string is created the AU sends its agreement message to its ring neighbor. The agreement message contains the tile ID and the context message. This way each recipient can decode the context information. After sending its own local string the AU waits for the ring predecessor's message to arrive. It checks the source ID in the arriving message and determines if it corresponds to its own ID. If that is the case it knows that the original message that it sent was now received by all other tiles and that the agreement round is finished. If the source ID is different, the AU will save the context information and the corresponding source ID for the global context information convergence. Afterwards it will relay the message as it was to its ring successor.

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 agreement round. This is shown in Figure 6.2 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 $t+1$.

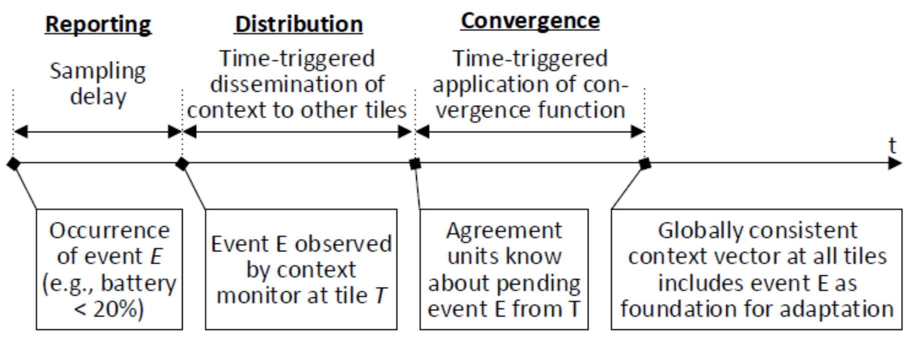
Reporting
Sampling delay
Distribution
Time-triggered dissemination of context to other tiles
Convergence
Time-triggered application of con-vergence function
t
Occurrence of event E (e.g., battery < 20%)
Event E observed by context monitor at tile T
Agreement units know about pending event E from T
Globally consistent context vector at all tiles includes event E as foundation for adaptation

Figure 6.2: 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 [Obermaisser2019].

The collected state information is sent to all other tiles. Each core saves the information received from the other tiles after receiving it. Once the broadcast of local information is finished, each NI will have a string containing all local context.

6.2.1 Big-Bang vs Incremental Exchange

When exchanging the local context to the rest of the network two paradigms with different implications on the protocol can be followed. Depending on when a context is exchanged we can either minimize the sampling delay for the context value or optimize the network load. The two paradigms are

- incremental exchange
- big-bang approach

Big-Bang Approach The *big bang approach* means that context information is sent by all tiles to the rest of the network at once. The local status is observed for a defined time and the events that occurred during this time are reported to the neighbors. All tiles are bursting the local information to the network at once, which means that the context values cannot collide with each other if they are sent in the ring as described in the agreement section. This implementation eliminates collisions on the network.

A downside to a planned big bang exchange is the delay it adds to the observed context. As all the context values are collected for the time until an exchange is scheduled, the context can loose its validity until it is exchanged. This sampling delay can hinder a fast reaction to the observed context.

To have the lowest sampling delay possible, the exchange of the context values would have to be executed in the highest possible frequency, thus putting a high stress on the network connection and creating a considerable energy overhead.

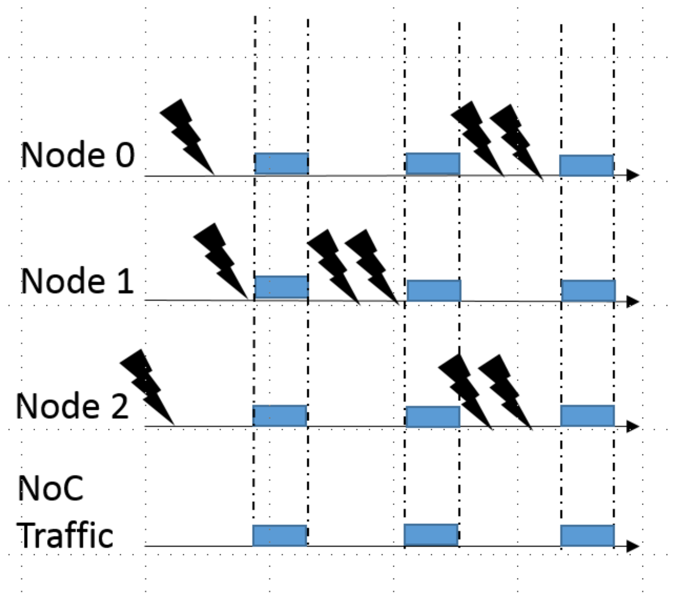

Figure 6.3: Figure of the timing, two or more tiles timelines and each send their context information at the same time

This solution is optimal for systems where multiple context values are frequently observed at all tile and need to be exchanged at a similar pace. This is also a valid solution to monitor the system for asynchronous events.

Incremental Exchange The incremental exchange tries to simulate an event triggered solution by polling for events at the time they are expected to happen and immediately exchanging the observed values. This solution will allow the system to have a near immediate reaction to events as they happen and thus optimize the system at runtime.

On the other hand the system is bound to run into problems as soon as two or more events are expected to happen at nearly the same time. The context polling will have to be scheduled by context volatility and importance to the adaptation, thus introducing a new classification that needs to be defined for the system at design time. Collisions of the same class need to have a special ranking defined to have a rule on which event should be reported on first.

This iterative solution is applicable on systems that don't have many events within the same time and space. Once the events can be distinctly told apart from each other and no collusion of two events happening at the same time in the same tile can occur, the solution can be executed to optimize the systems adaptivity.

The paradigm would be best implemented with a event-triggered system. If gains the most potential for quick changes if the system can react to a context occurrence as soon as it is observed. In a time-triggered system we always introduce a polling delay. The delay can in worst case be infinite, as the sensor might

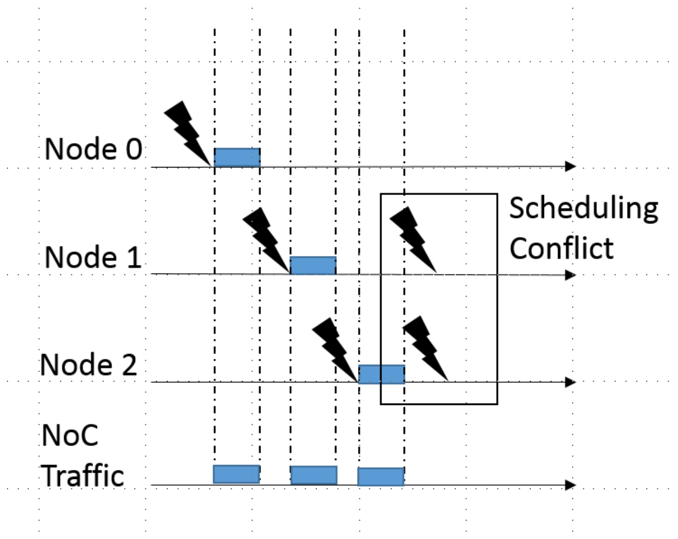

Figure 6.4: Figure of timeline of two tiles where each tile detects an context event at the same time and the agreement process must prioritize the higher level event to ensure the context exchange with no collisions

be checking for the context too early.

One can conclude that it is highly dependent on the application which of the two paradigms is the most sensible one. Each application and its context model define how plausible a context detection is and if multiple context detections can occur in close range or close time frames. Multiple context events at the same time on the same tile, render the advantage of the iterative model less appealing. Once the detection must be scheduled by other means that at the expected time of occurrence the advantage of the fast reaction is lost and a delay is introduced.

With the big bang version, all context ports are be read at all times once an agreement is triggered. Hence, in cases where the context model contains only sparse event patterns the iterative model is a better model for the iterative agreement protocol, as it only exchanges the context that is expected to happen. This reduces the traffic generated by the agreement protocol and therefore also the energy consumption. Only in such sparse context event assumptions the global state can be established definitely after each establishment round, as there is enough time for the state to be agreed before the next possible event and any action based on the current state can be performed safely.

The big bang version always exchanges the current status from all tiles even if no new context was observed. The non-occurrence of events is as important to know globally as their occurrence. To enable global adaptation all tiles need to know all tile states at all times, as each combination leads to a different path along the schedule DAG.

6.3 Protocol Phase: Convergence

In the last protocol phase the information that was agreed on is converged into a global context string which is used as input for the adaptation.

In this phase the context information which is observed by two or more context monitors at the same time will be converged to a single value. We have decided to use a mean value of the observed context information.

This convergence function will be executed in a distributed manner, locally on all tiles. We can be sure that all values that are taken into consideration are correct or will not be taken into consideration. We will take a closer look on how this can be achieved in system models where hardware faults are allowed to happen. For the basic introduction of the protocol we will assume that the validation of the globally agreed context values can be assumed to be correct.

For the tiles to know which values need to be converged the tiles need additional information on the context values they received from the other tiles. As mentioned before each tile has a predefined bit mask for its context string which all other tiles are aware of.

Based on this mask each AU can trace back extract specific bit values from the local bitstring of the other tiles. When performing the convergence the AU checks which tiles provide redundant information and extract the corresponding bits from the context string. This knowledge is provided in a map at design time, which maps for all the redundant context values C_R where its values are saved by identifying the source T_s tile and the bits within the context string B_c. Thus creating a vector of tuples $C_R = (T_{s1}, B_{c1}), T_{s2}, B_{c2}), T_{s3}, B_{c3})$ for each redundant context value.

When converging the values, the AU can now check for each context C_R if all corresponding context values saving within the receives local context string match.

In case the system has redundant context monitors to enhance the agreement's fault tolerance, these redundant context bits can vary and need to be verified w.r.t. their correctness. Any discrepancy will invalidate all the values and they will be replaced. In case of two values we can only decide if they are correct or false, then the values will be replaced with the context variable's safe value, the concept of safe values is explained in detail in section 8.1.1. If we have three or more values, we can vote on the correct value and replace all values with the correct one.

If the context value is observed by the context monitor and exchanged over the agreement layer as is, the values may minimally diverge from one another, due to different placement of the context monitors within the system, e.g. temperature sensors on two sides of the board. In this cases we first check if the divergence is in an acceptable range to avoid faulty context values. The range must be defined at design time and will just be checked at the convergence phase. If everything is within the acceptable values, they will be converged to the mean value of the observed context. When the end of the vector is reached the local context strings contain the converged context bits.

If the value is interpreted by the context monitor, by comparing the observed value to a predefined threshold value, the context monitor only sends a bit value

Algorithm 6 Protocol algorithm for the convergence

```
1:  $V$ = vector or redundant context tupels
2:  $context_n$ = local context string of tile $n$
3:  $cur$ = current redundant set of context
4:  for $j = 0$ to size of vector $V$ do
5:      $cur = V[j]$
6:      for $i = 0$ to size of $cur$ do
7:          if all context values are equal then
8:              keep value
9:          else
10:             overwrite both values with safe value or vote on correct value
11:         end if
12:     end for
13: end for
14: $output$ = merge all neighbor context messages
```

indicating if the threshold was reached or not. In this case a convergence is not performed. Rather the three values are compared to each other. Due to the varying sensor positions it is possible that the threshold interpretation may result in different interpretations, for example 2 values might indicate that a threshold of 30C for the temperature is met as the temperature that they observe is exactly 30, while the third context monitor measures 29,9 and thus does not report that the temperature meets the threshold requirements. In this cases the majority vote of a value is chosen as a global context. This also implies that these kinds of values must be observed by 3 independent context monitors or just by one without any convergence measures, if the correctness of the value isn't vital to the system health.

Afterwards each tile concatenates the received context strings into a global context string in a row. This string then identifies which events happened within the last observation phase globally and locate the current state within the DAG.

$$\underbrace{\boxed{C_0}\boxed{C_1}\boxed{C_2}}_{\text{Tile 1}}\underbrace{\boxed{C_0}\boxed{C_1}}_{\text{Tile 2}}\underbrace{\boxed{C_0}\boxed{C_1}}_{\text{Tile 3}}\underbrace{\boxed{C_0}\boxed{C_1}}_{\text{Tile 4}}$$

Example of a global context string for a network with 4 tiles. There is a dedicated area for each tile's local context C_x. We can see in the example that tile 1 has three context events that it reports on while the other tiles all have two.

Chapter 7

Implementation of the Protocol in Hardware

The protocol was implemented within an existing *Time-Triggered Extension Layer* (TTEL) [Ahmadian; H.; Obermaisser2015]. The TTEL operates on top of event triggered NoCs to add time-triggered message injection where none was implemented without affecting the NoC nor the cores or their applications.

The existing extension layer provided functionalities to test time-triggered message injection based on schedules [Ahmadian; H.; Obermaisser2015], and it was extended with an adaptation unit as shown in Figure 7.1. The 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 agreement protocol completes the setup with the propagation of the local values. We expanded the hardware model with an additional hardware block dedicated to the agreement 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.

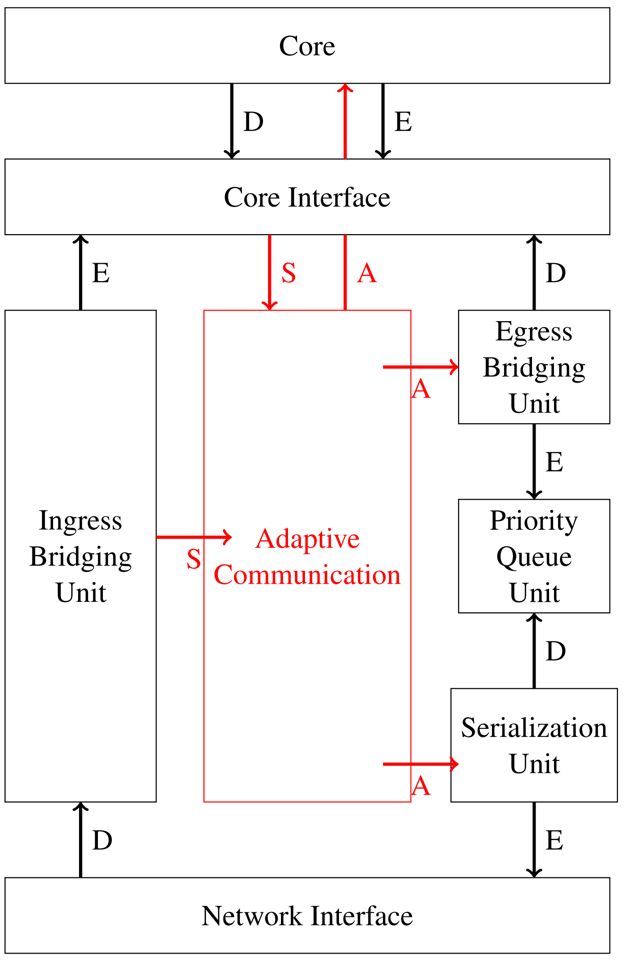

Figure 7.1: The architecture of an NI.

7.1 Agreement Implementation

In this section we focus on the implementation of the protocol. We first explain the hardware building block definition and the implementation specific details that need to be considered when planning such a protocol. Afterwards we explain how the interface to the context monitors needs to be implemented and what needs to be taken into account before finally explaining the setup of the output that the protocol generates.

7.1.1 Building Block

The agreement block consists of separate blocks, the initiation block and the main agreement block and supporting blocks: a port reader and a clock generator. The initiation block, as the name suggests, initiates the agreement based on a triggering signal which can be generated by a clock counter. The clock counter is connected to the global clock and can measure when the given time frame dictated by the frequency is passed.

The initialization block is directly connected to the next neighbor's main agreement block to exchange the local context directly. For the sending we used two signals with one signal being the local context message $m = (srcID, local\ context)$ and the second being a status bit, which is set to indicate that the signal was updated.

The main blocks are connected to each other similarly by two incoming signals from the predecessor, the local context message m and the corresponding enable bit, and two outgoing signals to the successor, the local context message m and its enabling bit.

Finally the main output of the AU block is the global context string which is connected to the NI's adaptation manager. This signal is also coupled with an enabling signal to trigger the adaptation.

Once the main agreement block receives the first local context of the current agreement round the main block blocks the initialization block for the duration of the agreement. This prevents that a poorly chosen frequency keeps restarting the agreement process before the whole agreement round is finished.

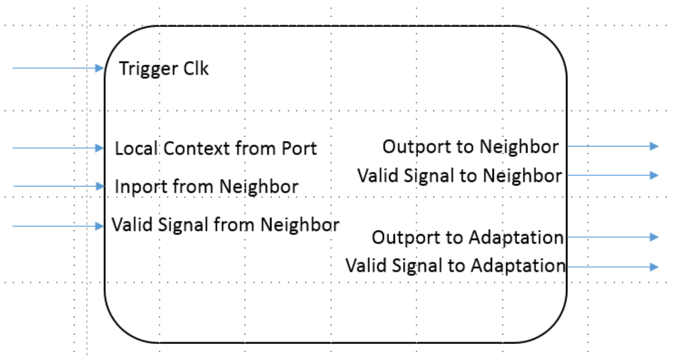

Figure 7.2: Figure of the AU building block, with its input and output ports. One can see that each signal has a supporting signal which informs the receiver that the signal is valid.

When the initiation block is triggered by the AU specific clock, the agreement begins. The port reader is triggered to read the ports and relays the port values to the initialization block. The initialization block has a BRAM attached where the local port configuration and the expected context size are saved. Based on this port information the initialization block creates the NI's local context string and sends it to the next neighbor.

7.1.2 Implementation Specific Delays

The **reporting delay** is directly correlated with the implementation of the soft and hardware reporting to the NI. The faster the software is connected to the NI the lower the reporting delay will be. Middleware can thus introduce uncertainties as the message handling within it can be obscured to the system. When using middleware the user needs to ensure that the middleware reporting is either direct or the extra time is also considered within the WCET that the scheduler assumes for the execution schedules. The **Polling delay** can be impacted by the frequency with which the agreement is executed and as such is a trade-off decision between the amount of agreement rounds and possible schedule changes versus the event detection. To have the best approximation of the system state the agreement should be started right after the last agreement ended. This way the offset between reporting of an event and awareness of the NI about the event within a systems containing n nodes is $n * message\ transmission\ time$ because the agreement takes this amount of cycles to finish. This means that per period $\frac{period}{n*transmission}$ agreement rounds are performed. In our implementation the transmission delay that was introduced by the agreement was $4*5$ as we have 4 tiles and the transmission took 5 cycles.

7.1.3 Dedicated Connection for all Context Types

Within the NI each context type must have a dedicated port assigned. This ensures, that no matter when a context monitor is writing into a port the value is not lost by another monitor overwriting the port. This means that each NI has a system specific port configuration defining the amount of input and output ports depending on the amount of CMs and applications reporting context to the AU. Each port has a specific size which is predefined based on the expected context value, ref. 7.3

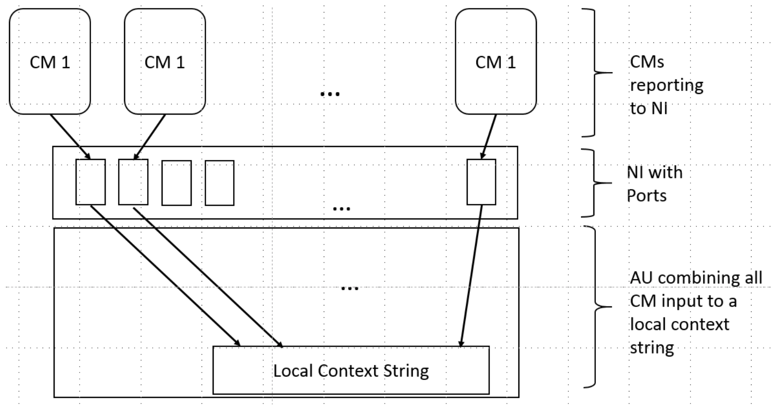

Figure 7.3: Architecture of the Context reporting. Each CM has a dedicated ports that it writes its context values into. Only th AU is allowed to read these ports. After reading the ports the AU combines the context information into a local context string which is sent to the rest of the network.

When the AU starts reading the ports it checks in its internal portmap what context size it has to expect and therefore how many and which bits within the local context string it has to write the read values into. Each NI is aware of all the other NIs' portmaps, therefore each NI is able to interpret the incoming values. This is crucial as the context bitsize can vary and the receiving NI could wrongly interpret context bits to the wrong source ports which can lead to wrong interpretation of the values in the convergence phase. It is important to note that the AU does not understand the values it relays to its neighbors, it only knows what the correct layout and bit size of the incoming values is. It is not able to perform context specific corrections and must trust the values to be correct.

Algorithm 8 Port reading instructions

```
1: $conf[i]$ = bitsize of of $port_i$
2: $context$ = local context string
3: $m$ = pointer to current string position
4: for $j = 0$ to amount of AU Ports do
5:     $context[m] = port\ reader(port_i)$
6:     $m = m + conf[i]$
7: end for
8: $output = context$
```

7.1.4 Context String Layout

At this point we define two types of context strings: an intermediate context and a global context.

Local Context A locally observed event occurrence c_l is known only locally at one tile. It must be made known to the rest of the system before it can be taken into consideration for a schedule change. It is a bitstring that encodes the locally observable events. The local context encodes all events possible on the associated tile. All bits within the string are mapped to a dedicated event. Flipping a context bit to "1" indicates that the corresponding event occurred.

Binary events are obvious to cover with this encoding, but value based context events like the current temperature can be treated in two different ways within the local context. If the value is important to all other tiles it can be added into the local context string. In this case the specific value of the context is important for the schedule DAG.

However, if only the interpretation of the value is needed for the schedule DAG, the context monitors can be used to evaluate the information they receive from the sensors. At design time a threshold needs to be defined, indicating if the sensor value is not acceptable anymore, e.g. a certain low battery charge or if the temperature becomes too high for the system to function safely. In theory the context only consists of the local contexts concatenated in the same order as the port ID by which they were observed. Since we have two types of possible context events which lead to different bitsizes per event a mask is needed for each NI to indicate where the individual context bits end. We chose not to add a specific ID to the context string, as this would bloat the context string and minimize the possible context values that we can send per message. The ID would take up bits within the message which otherwise could be used for the context events.

Global Context : Is the union $\cup c_l$ of all local context and the basis for a schedule change decision. The global context string is the concatenation of all local context strings.

In the most economic implementation of the building block the context monitors should be designed to have a single bit as the output. This minimizes the local context strings and in turn also the global context string which is sent to the adaptation manager. The less hardware is needed the smaller the building blocks energy footprint will be. Furthermore the one bit output is easier to scale for systems with a large amount of context monitors attached to them. The longer the context string is per tile the more wires are needed to transmit the local information throughout the system. In our example implementation we used a set of 32 bit wires between the AUs. Of these 32 bits we must reserve a set of bits to encode the source ID, which takes at least three bits in middle sized networks of up to 8 tiles and reduces the possible context bits to 29 bits, three ID bits and 29 value bits.

7.2 Discussion of Frequency Implications

The *frequency* by which the agreement protocol is executed dictates the delay by which adaptation changes can be executed. The higher the frequency the faster the adaptation can be performed but at the same time the energy consumption rises as the agreement messages are exchanged more often. For this reason the frequency should be weighted against the volatility of the context values. If the system is only designed to observe slowly changing values, a low frequency will be able to provide a close enough view on the system state without loosing possible change opportunities compared to if the tiles are observing values that lose their validity after a short time, e.g. slack, where the low observation frequency will lead to many missed adaptation opportunities.

In the following, we show how the different protocol frequencies can impact the adaptation potential within a period. We show a schedule of two nodes connected via a link. Both nodes execute some tasks, node A executed the tasks p_0, p_1 and p_2, node B the task p_3. The task p_0 communicates with the task p_3 with the message m_0. We indicate the protocol execution of the agreement protocol by showing its duration in the sets of dashed and dotted lines. These lines indicate the beginning (dashed) and the end (dotted) of the agreement protocol. We compare three different protocol frequencies within a period: 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 7.4–7.6. These varying frequencies can be useful for different kinds of applications. The first type in Figure 7.4 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 7.5 and the third type in Figure 7.6 can be chosen based on the context type. The more often it is expected that the observed state is changing, the more frequently the agreement protocol is needed. The protocol can be executed based on the systems' needs more or less frequently. For example in a single agreement protocol instance per period, as shown in Figure 7.4, any context that happens *before* the dashed line can be taken as context for a potential schedule change. In the time period between the two dashed lines the context is agreed upon. Once the context is agreed at the dotted line, a schedule change can be performed.

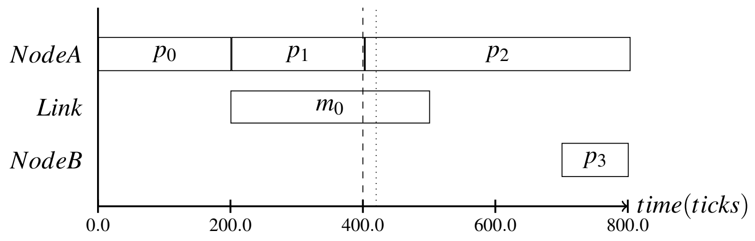

Figure 7.4: In this case only the task p_3 could be optimized by the schedule change, as all other tasks have started before the potential schedule change.

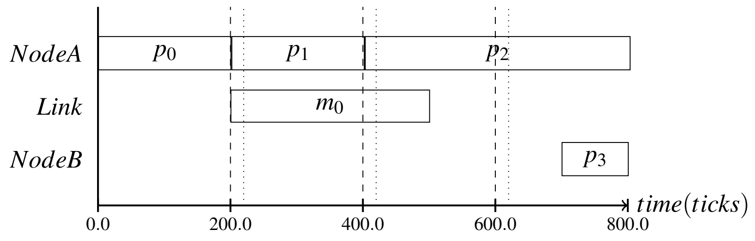

Figure 7.5: More frequent execution of the protocol.

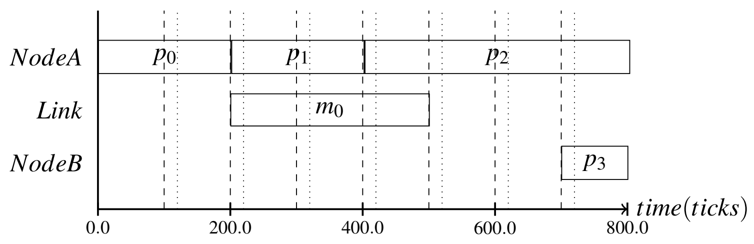

Figure 7.6: The more often the protocol is executed, the more information about the system state can be inferred.

The frequency further also dictates at which instances the schedule can be changed: if only one state agreement is performed during a period, only one schedule change can be done. The scheduler therefore has to be aligned with the agreement 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 that are used in the agreement 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 anymore.

Therefore, it is also crucial to perform the agreement 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 3, within the second half of the process's time slot. If one would use a frequency as shown in Figure 7.7, 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.

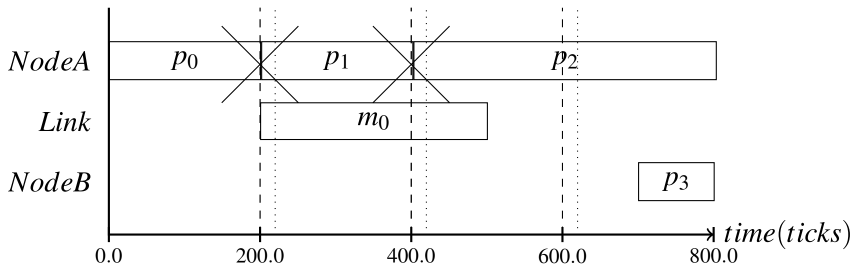

Figure 7.7: The timing for the consistency of a an agreed state depends on the context one likes to observe. In this case, the slack could not be detected, as the protocol was executed right after the WCET.

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 of the adaptation. 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 is scheduled to perform earlier then the agreement, as shown in Figure 7.8 the adaptation will never be able to execute a schedule change, as no context values are known. In the figure we show the instance when an adaptation is possible due to me meta scheduling graph as a red line while the black dashed/dotted line is the agreement protocol interval. The agreement protocol must provide the adaptation with the system state shortly before a schedule change can be performed. Due to the runtime of the agreement protocol, the adaptation for the observed state during the runtime between ticks t_0 and t_1, is performed at tick $t_1 + t_{agreement}$, since the state needs to be agreement upon globally before it can be used as adaptation input.

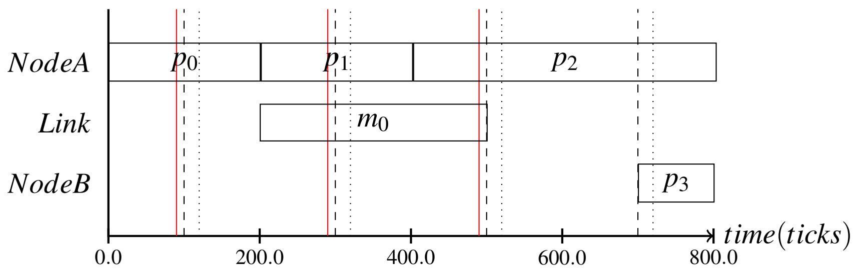

Figure 7.8: The mode-change state machine expects a context value that cannot be provided by the agreement protocol, because they are running out of sync. This would cause the adaptation to fail.

7.3 Optimized Regions of Interest Based Agreement

Until now we have executed the agreement based on a preset frequency which introduced the problem of potential delays between the protocol execution and branching point. The protocol can be optimized if its triggering is caused by upcoming branching points, which would require to execute the protocol based on a dedicated schedule. This schedule would need to be defined by a DAG specifically designed for the agreement protocol, as each change in the system's schedule also implies that upcoming branching points will change at runtime. In this section we propose a meta scheduler implementation that computes *regions of interests* (ROI) within the schedule DAG, in which an agreement execution would make sense.

7.3.1 Scheduling

Our meta scheduler uses the genetic scheduling to provide a set of schedules that cover all relevant states that the target system can reach at runtime. Each schedule by itself is a valid solution which fulfills the application model restrictions. The result of the genetic algorithm is used as an input for the meta-scheduling. The basic idea of the meta-scheduling is that a baseline schedule is found which fulfills all application restrictions and could be used throughout the period.

With the baseline schedule as input the meta-scheduler re-iterates through the schedule, looking for *branching points*, see Algorithm 9. Branching points are points in time, where a context-event, e.g. slack, may occur at runtime. Within the application DAG at these points new alternative schedule branches are located which can be exploited by an alternative schedule. For each branching point the meta-scheduler computes an alternative schedule that can be used for the possibility of slack occurring at this point. Meaning that if task t_1 would generate slack at a time t_s the meta-scheduler computes a schedule where the time $t_{wcet} - t_s$ would be used for the upcoming tasks, therefore reducing the idle time of the processors. At the end of the iteration the meta scheduler will have computed a set of schedules that fits to all possible scenarios that can be safely reached during runtime.

The reachable scenarios are defined by the application designer, who has to define a *context model* for the meta-scheduler. This context model defines which runtime events are adapted to, e.g. in our case this means core faults and slack. The designer has to provide the meta-scheduler with the expected slack time, so that an alternative schedule for its occurrence can be created. The context model therefore includes the average execution time for each task. In our model we only executed the meta-scheduler for the cases of average and worst case execution, but generally it is possible to define various slack times for a more fine-grained differentiation of execution times. One must be aware though that each context value will lead to a new schedule which needs to be saved and changed into. The designer needs to decide if the amount of observed context variations is worth the additional runtime of the meta-scheduler and the additional memory space on the system.

With the knowledge of the *branching points* defined by the context model, the meta-scheduler computes a new schedule for the tasks that have not started yet at the current branching point t_{branch}. All tasks that have started at a time $t_{prev} \leq t_{branch}$ will not be rescheduled but frozen in their execution. Only tasks with a start time $t_{follow} > t_{branch}$ can be rescheduled. This way we ensure that tasks are not preempted and deadlines are kept.

Such an approach leads to an exponential growth of the amount of possible schedules, which in turn need to be stored in memory to be reachable at runtime. This puts a strain on the design of the MPSoC. To reduce the amount of possible schedules we do not hand over the slack to all following tasks. Instead, we only consider tasks following in a configurable time window ω after the slack occurrence, so that $t_{follow} < t_{branch} + \omega$. Since calculating the slack-aware schedules is done in an incremental way (Algorithm 10), each of these schedules has a parent.

All tasks and messages are frozen to the state of the parent schedule, except those within the convergence window. This is mainly reached by pinning the corresponding genes in the parent schedule's phenotype and telling Algorithm 1 the start and injection times of the tasks and messages that should not change. The genetic algorithm will then make sure that all other requirements are still met. We will show in the next section that even though it reduces the computational time needed for schedule generation it does not negatively impact the energy savings.

Algorithm 9 Metascheduler outer loop

```
 1: function METASCHEDULER(model, ctx)
 2:     $schedules \leftarrow \emptyset$
 3:     for all $c \in$ ctx.crashes do
 4:         if $c \neq -1$ then
 5:             REMOVE_NODE_FROM_MODEL(c)
 6:         end if
 7:         base $\leftarrow$ SCHEDULE$(model)$
 8:         $schedules \cup$ WITHSLACK$(base, model, ctx)$
 9:         if $c \neq -1$ then
10:             ADD_NODE_TO_MODEL(c)
11:         end if
12:     end for
13:     return schedules
14: end function
```

Algorithm 10 Calculate Powerset

```
 1: function POWERSET(slacks, base)
 2:     $schedules \leftarrow \{base\}$
 3:     for all $event \in slacks$ do
 4:         $tmp \leftarrow \emptyset$
 5:         for all $parent \in schedules$ do
 6:             if event is viable for parent then
 7:                 $tmp \cup$ SLACKSCHEDULER$(parent, event)$
 8:             end if
 9:         end for
10:         $schedules \cup tmp$
11:     end for
12: end function
```

7.3.2 Regions-of-Interest evaluation

We further used the benchmark use case to evaluate how the ROI based meta scheduling improved the system performance and if an optimized schedule graph would lead to significant energy savings.

Table 7.1: Results for a small synthetic example

Window	Schedules	Time	Makespan	Energy
∞	176	$0m6.500s$	204	35
100	122	$0m5.400s$	217	39
50	99	$0m5.100s$	228	47

We validated our model and the implementation of the meta scheduler using parts of the communication benchmark presented in [Liu and Wang2011] as an example. Calculating the schedules is a two-step process. In the first step, a slack-free schedule is calculated for each event at the fault-tolerance level. This will give us $|F|+1$ schedules, where $|F|$ is the number fault events, e.g. crash of computing tile or router. If a computing tile is faulty, all tasks are scheduled to other computing tiles for the next period. in case of faulty routers messages must be passed using other routes in the next period. To calculate each schedule, we evolved the genetic algorithm for 50 generations with 200 individuals per generation. The individuals were mutated with a probability of 0.2 and recombined with a probability of 0.9.

The second step is calculating the slack-aware schedules. For each schedule from the first step, this will lead to at most $2^{|S|}-1$ additional schedules depending on the size of the convergation window, where $|S|$ is the number of possible slack events in the context. Again, we evolved the genetic algorithm for 50 generations per schedule, but with only 10 individuals. The probabilities for mutation and recombination are 0.1 and 0.9.

We measured the execution time of the meta scheduler on our HPC cluster for different convergation windows.

7.3.3 Small Example

The first example is the four-task DAG shown in figure 7.9 scheduled to seven computing tiles grouped around three fully connected routers shown in figure 7.10. This will give us $(7+3+1)\cdot 2^4 = 176$ different system states for full single-fault tolerance and complete slack adaptation. According to our power model the worst-case energy consumption for the unoptimized system is 80. As one can easily see from figure 7.9 only two different topological orderings exist. If you execute all tasks in a serial manner on the same tile, you would end up with a makespan of 80 ticks, as no communication is needed. But with the earliest deadline being at 250 ticks, you are able to run all tasks at 34% of the speed and still meet this deadline. In order to get at least some communication, we enforced to execute t_1 on tiles c_0 or c_1 and all other tasks on computing tiles directly connected to router R_1 or R_2.

Table 7.3.3 shows the average number of schedules found for a convergation window, along with the average makespan and the average energy consumption

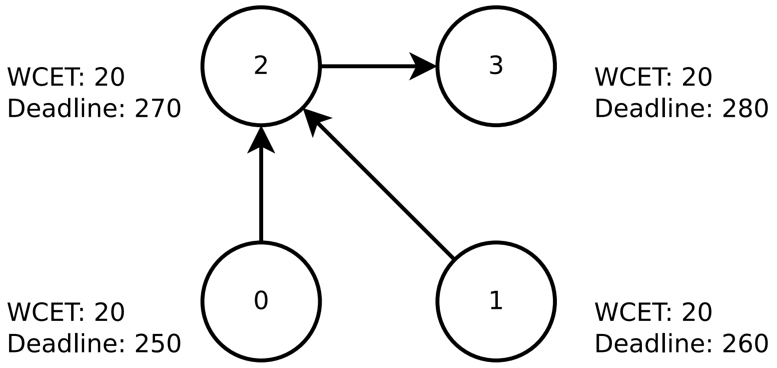

Figure 7.9: Example Application

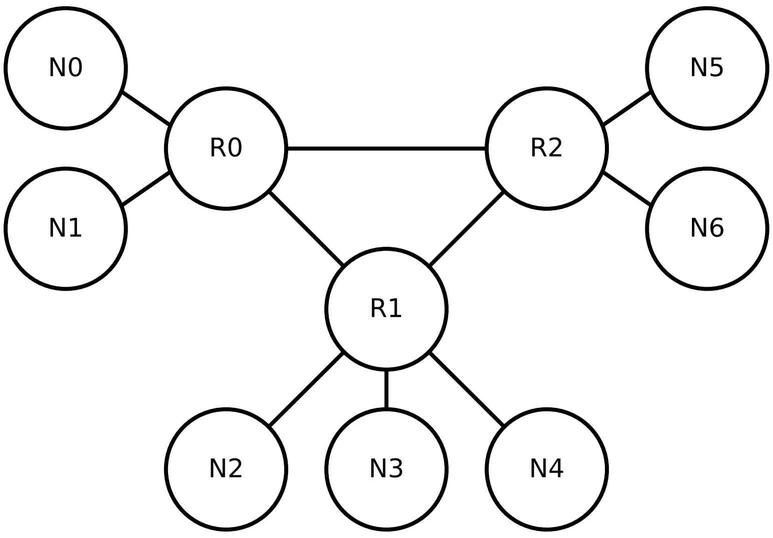

Figure 7.10: Example Network

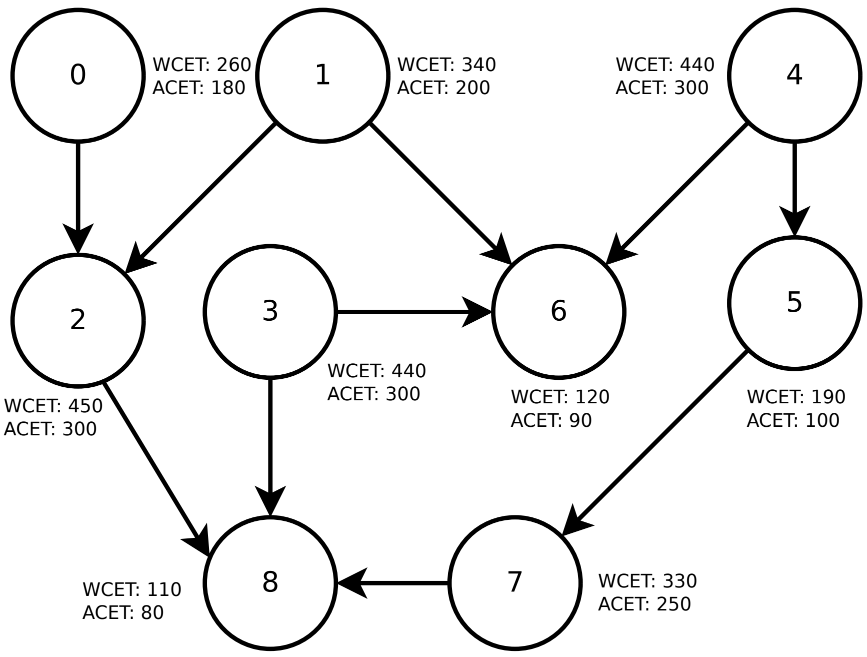

Figure 7.11: Example Application

of those schedules. As you can see from the results, utilizing the slack for the rest of period saves over 56% of the energy, while keeping the makespan below the deadline of 250 ticks. With a convergation window set to 100 ticks, which is roughly half of the period, 51% could be saved. The smallest convergation window of 50 ticks provides 41% savings.

7.3.4 Benchmark based example

Our example consists of nine task scheduled to a 4×4 NoC with 16 computing tiles and 16 routers. This system has $(16+16+1) \cdot 2^9 = 16.896$ different system states as a maximum. Figure 7.11 shows the application model. Each task is annotated with its worst-case and average-case execution time. The latter is used for computing the slack.

We computed the schedules using 33 parallel processes (one for each failure plus one for the base schedule) and observed the computational time as well as the number of calculated schedules depending on the size of the convergation window. Table 7.3.4 shows the measured results. As you can see, we reach 50% savings for a small convergation window of 500 ticks. Compared to the full adaptation with 16.896 schedules calculated in 9 minutes and 55% savings, we calculated around 2500 schedules in under three minutes.

Table 7.2: Results for Communication Benchmark

Window	Schedules	Time	Makespan	Energy
∞	16896	$9m6.000s$	5141	$1194 \cdot W$
5000	16233	$8m25.700s$	5704	$1164 \cdot W$
2500	12867	$6m29.940s$	4890	$1255 \cdot W$
1000	4442	$3m2.800s$	4670	$1293 \cdot W$
500	2448	$2m36.900s$	4670	$1350 \cdot W$

7.3.5 Conclusion and Future Work

We presented a (meta-)scheduling model for energy efficient scheduling of fault-tolerant realtime *MPSoC* systems. We described our application model as well as the platform model in order to define a power model that is used to calculate the energy consumption of the scheduled application.

The (meta-)scheduling model was implemented using a genetic algorithm. To achieve power efficiency we make use of the fact that the difference between a task's worst-case execution time and the actual execution time is usually big. This difference called slack was then used to reschedule other tasks following in a certain time window. We showed that we can save 40%–55% dynamic power depending on the window size. Our results show that even small window sizes will lead to over 40% savings while keeping the number of possible system states low. Calculating all states with an infinite window size may not be worth the computational overhead, as it does not provide significant benefit for energy efficiency. It is also not feasible for bigger examples due to memory limitations.

In future work we will investigate if the interesting points for observing slacks can be extracted from the application model and the base schedule. We hope to reduce the number of slack events that need to be considered for significant energy savings by using this technique, thus reducing the overhead in terms of storage needed for the schedules and computational complexity. These events may then be used to reschedule the rest of the period.

According to our power model the dynamic energy consumption of the application without energy-aware scheduling is $2680 \cdot W$ which is also the makespan (2680 ticks) for serial execution at full speed on a single core.

7.4 Implementation: NoC vs Dedicated Links

When implementing the agreement protocol's distribution phase one can use the existing communication platform, the NoC network to execute the protocol. In this case a ring like path has to be defined. This path predetermines the successors and how the exchange will be executed within the NoC. Further we need to plan how to integrate the agreement messages into the normal communication. As the additional traffic will hinder the normal communication and any influence on the communication deadlines must be prevented. This section discusses how these challenges need to be met and proposes a solution on how to avoid any message overhead in the first place by creating a dedicated network for the agreement protocol.

7.4.1 NoC-Based Approach

The agreement protocol occupies the $m \times n$ big mesh of the network, with m being the amount of horizontal nodes and n the amount of vertical nodes, 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 tile send its local information to all neighbors. Then, all tiles wait until 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. In a mesh the ring-like structure cannot be upheld and would cause collisions, therefore we need to send the information in a burst and iterate a second agreement round to ensure that all tiles received the same information. Figure 7.12 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 tile 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 tiles are aware of their neighbors' context, as well. After performing another transmission of the complete locally known context, all tiles know the context values of all other tiles. 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.

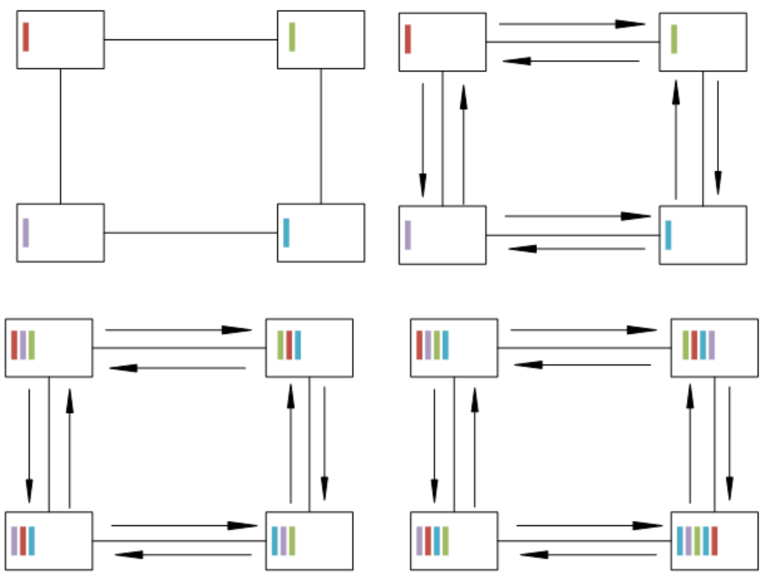

Figure 7.12: Propagation on NoC: the four tiles are connection via the NoC, represented by the lines. The colored sticks show how the local context is traversing through the network for each propagation step.

7.4.2 Dedicated FIFO Links

An alternative to the usage of the NoC for the agreement 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 7.13.

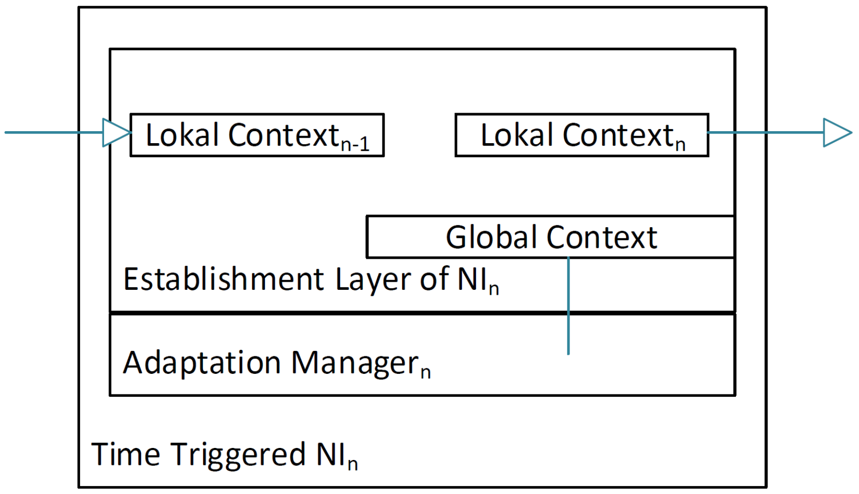

Figure 7.13: Hardware block of the establishment Layer. The layer is located within the NI and has dedicated connections to its direct neighbors in the dedicated FIFO ring. As the communication channels are unidirectional, the layer can only actively communicate to its right neighbor and receive information 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 agreement 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 7.14. 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 tiles receives its own original information from the left neighbor, as described in algorithm **??**. 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 tiles 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.

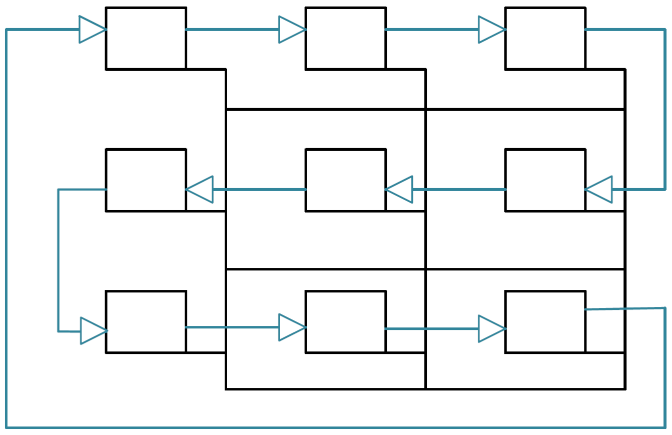

Figure 7.14: Propagation on FIFO; the four tiles are connected via the NoC, represented by the normal line, and a directional link represented by the arrows. The colored sticks show how the local context is traversing through the network for each propagation step. In this scheme, the propagation step is equal to one tick.

7.4.3 Scalability Implications

To analyze the scalability, we inspected how increasing the network size will affect the networks utilization. Adding the agreement protocol instances, we refer to the instantiation of the protocol based on a certain number of tiles, 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 tile 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 tiles. We assume that the constant wct_n depicts the worst case transmission time of a message in an NoC with ntiles and size returning the length of a message in words. We will in the formula refer to the agreement 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 tiles will increase in the following way:

$$utilization_n = \frac{\sum_{i \in Msg} wct_n \cdot size(i) + \sum_{i \in k} wct_{ICP} \cdot size(msg_{ICP})}{Period} \tag{7.1}$$

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:

$$Period \cdot utilization_n = wct_n \cdot \sum_{i \in Mgs} size(i) + k \cdot (wct_n + 1) \cdot msg_{IPC} \tag{7.2}$$

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 agreement 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 agreement 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 agreement protocol for big NoCs, a hierarchical model could be used where the domains to establish a context agreement 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 tile. 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 agreement protocol.

7.5 Evaluation of the NoC and FIFO implementation

We analyzed the performance of a NoC and a FIFO approach. We have simulated both implementations in an already existing framework realized in gem5 [Binkert2011].

Our goal of the simulation is to quantify the benefits of the adaptation with respect to energy efficiency created by the agreement protocol. In the simulation, adaptation is used to save energy by dynamically changing the system schedule and applying power gating on the tiles when they finish a computation. As a result, we designed our power model to include idle and running states of the tiles. For the tile energy usage, we computed the difference of idle and running states as published in [Goergen and Schreiner2016].

The simulation is evaluated based on the assumed energy consumption at runtime. Our power model approximates the consumed power based on the behavior of the CMOS-circuits, by deriving a typical energy consumption per tick. A tick in gem5 equals $10ns$, based on the transmission duration between two routers. Therefore, we have 10^8 ticks per second. To compute an upper bound for the per tick energy consumption of an FPGA as described in [Thomas and Luk2009], we took the maximum power value of 30 Joule and divided it by the ticks per second, giving us the per tick energy consumption of 3×10^{-7} Joule. 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 tile 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 tile are created by randomly scaling the WCET to simulate the unpredictable tile behavior. For the results, we have created traces with low slack and high slack. We compare our results to a baseline configuration. We let the tile run in idle mode until the WCET is finished.

7.5.1 Synthetic Use Case

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

The implemented use case assumes two tiles connected in a 4×4 mesh architecture with 16 tiles 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 tiles 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 7.15. 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 agreement protocol helps to save energy [Lenz et al.2017].

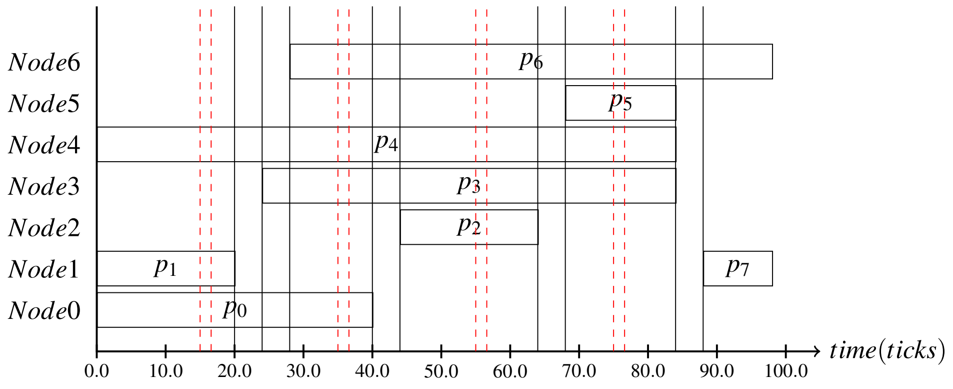

Figure 7.15: The use case schedule for the NoC implementation with context intervals scheduled within.

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 7.16 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.

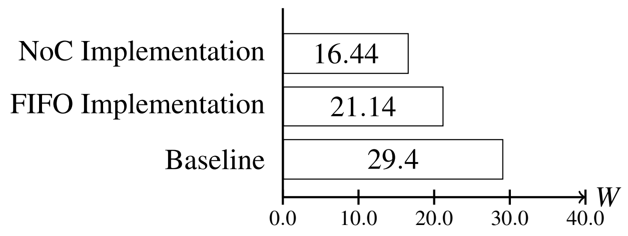

Figure 7.16: Energy consumption of the whole on-chip system per second in the synthetic use case simulation. The baseline energy consumption is the consumption without any adaptation. The FIFO implementation shows the consumption when the consistency protocol is executed on a dedicated network; the NoC 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.

7.5.2 Benchmark

The second use case implemented a NoC benchmark to evaluate how a heavy communication load would impact the agreement implementation.

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 $\times$ 4 NoC with 16 tiles. 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 [Liu and Wang2011] 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 $\times$ 4 NoC. We could therefore simulate real execution times and therefore real slack values. Since these benchmarks only showed how real time execution was observed, we had to deduct possible deadlines, as no scheduling information apriori to the execution was known for the sets. We deducted the baseline based on the range on the varying execution times. As we had multiple sets of execution times per task we modeled a worst case path and assumed these timing values as deadlines for the task execution. We deducted the communication times based on the message size and the delta of tics between arrival at the target node and the injection of the message into the network. The library provided a set of applications for different types of topologies and NoC sizes. Therefore we modeled the task execution and the timing restrictions for all tested network topologies. We chose the random sparse matrix solver for electronic circuit simulations on a 4 $\times$ 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 tiles, as shown in Table 7.3.

Table 7.3: Mapping of the processes to the tiles.

Tile	**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 agreement 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 7.17 we observed that both implementation reduced the energy usage of the system. The adaptation enables the system to execute the processes more quickly, and tiles 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.

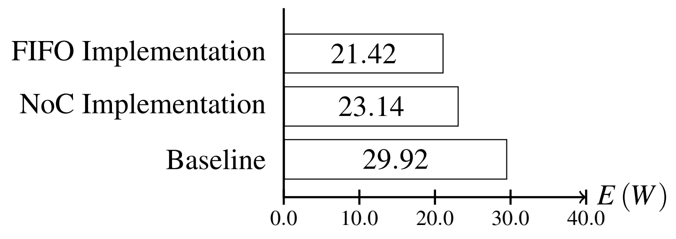

Figure 7.17: Energy consumption of the whole on-chip system per second in the benchmark simulation. The baseline energy consumption sums up to 29.92 W, while the NoC version uses 23.14 W, and the FIFO version consumes 21.42 W.

Chapter 8

Fault-Tolerant Agreement

This chapter introduces a fault-tolerant agreement protocol that ensures a continued agreement even under the occurrence of faults at runtime.

8.1 Fault Model

The adaptation as described needs reliable information regarding the current system state. It is paramount that the decentralized adaptation change decisions are all based on the same information. Therefore in this section we define how the state information is handled w.r.t. faults.

The goal for the fault tolerant protocol version is to be able to execute the protocol without any negative implications on the application even if a single failure occurs. This assumption includes a varying set of failures from simple link failures to transient faults that can impact the context information the agreement sends around.

Failure Mode We assume an arbitrary failure mode. This means that we assume hardware as well as software faults. Our agreement protocol must therefore be resilient against node and link failures as well as erroneous behavior of the core and the NI.

Fault Containment Region The *Fault Containment Region* (FCR) is the area where faults may occur at runtime and are tolerated. It is the delimiter of the immediate impact of a fault and is defined to cover the tile including the cores and the outgoing links.

No faults that originate within a FCR should negatively impact another FCR. This means that in case of hardware faults the messages need still to be made available to all the nodes that are not affected by the hardware fault. In case of faults within one FCR the remaining FCRs must be able to detect potential errors in the received messages and contain the error from being reported to the adaptation manager, therefore ensuring that the adaptation can still be executed safely without being negatively impacted at runtime.

8.1.1 Correctness of Information

Before talking about fault tolerance we need to define what correctness of information means. When we refer to correct information we mean that all tiles share the same view on the system state. This includes that all nodes have the same information regarding the other nodes' local context values. As such the correctness ensures that the agreement provides a unified view on the global state under the assumption of faults. At the end of the protocol all nodes can decide which values are known consistently by all tiles and which values were corrupted during their transmission. the corrupted values must be dropped. The agreement is only in charge of dependently exchanging the information it was given by the CMs.

The agreement protocol is not designed to verify the information it receives from its CMs as the information is very application and CM specific. Nevertheless is it possible that corrupted information can be introduced as context by the CMs and as such is exchanged on the network. This incorrect values may negatively impact the whole system by triggering adaptations out of their scheduled time frame. To avoid any of these side effect and potential exploitation we defined context types that identify the context values' importance to the system and its potential to harm the adaptation.

Context Types with Respect to Safety To further characterize the potential faults and their implications on the system safety, we define three types of context values that need varying degrees of strict fault handling at runtime.

- *Context state variables that need consistency* (e.g., dynamic slack): This context type originates from a single FCR only. It consists of context values that only can be observed locally on one tile, e.g. slack or application specific reporting. These values must be ensured to be transmitted consistently on the network. If the tiles act on the values, the protocol must ensure that all tiles react based on the same value. For example, these values by design may not be used for safety relevant adaptation changes if the tile's context message is not trusted.

- *Context state variables with safe values* (e.g., high temperature): For these values the system needs to be able to determine if the received value is corrupted to ensure that changes are performed only if the observed value is reliable. If a tile receives two values that contradict each other, the system will use a safe value. As the current state of the context is not verifiable the system should assume the worst and apply changes as needed to prevent harm. Each context of this type has such a safe value defined which will be used as fall back and directly trigger a safe state for this context. Once the values can be confirmed again, meaning that both values are received correctly, the safe state can be left. For example these context types can include temperature observations. If one tile observes too high temperature readings but a second tile does not, the system will stay in the current schedule. These context types can be considered as environmental influences that can be exploited to enhance the system's performance, but are not mandatory to ensure the system's safety.

- *Context state variables without safe values*: There are such context types that require to be known correctly at all times, as their input is paramount to the system's correct behavior. These values may include battery level and fault indications. Using a safe value for the battery status can prevent the system from properly functioning as a corrupt core may prevent the adaptation from ever changing into a low power mode and thus shortening the optimal system lifetime. These context types require three FCRs.

FCR Modes To ensure that a system is safe we need to define how these three context types can be handled safely. The three proposed context types correspond to three degrees of fault tolerance that can be associated with context types:

- *1 FCR*: In this level we only have the knowledge about the system state on the observing source FCR. The system must therefore trust the source to be reliable and not to fail or the impact of a potential error in the source generating the event must be minimal to the system's safety. Such a context message leaving its source FCR must consistently be distributed within the system. All nodes must either receive the same message and be sure that the message is correct as the source FCR has sent it or be able to tell if the received message was altered during the transmission. If the message is suspected of being corrupted it should be dropped as no corrupted information may be considered for the adaptation. Such a corrupted message would lead to inconsistencies or manipulate the system into a state that is exploitable by a malicious attacker. In this fault level the destination nodes must trust the original source to provide correct information.

- *2 FCR*: In this level we have duplicate knowledge about the information the system is exchanging. This level allows the destination nodes to detect core faults, as the information on the system state is observed by more than one source. Discrepancies can alert the destination nodes to drop the potentially harmful information.

- *FCR 3*: In the highest fault tolerance level the knowledge of the system state is available from three independent sources. This enables the destination cores to vote which of the received information is correct if one if them is inconsistent.

8.2 Fault-Tolerant All-to-All Broadcast

For our protocol we assume that the context information that the protocol is given, was handled according to its fault level and no negative impact can occur from the side of the protocol input.

For our Agreement protocol to function properly we must first define a fault-tolerant all-to-all broadcast, that ensures that the information used by the agreement is reliable. At the current state of the agreement protocol, when an agreement link would fail, the agreement could not be executed further without any additional hardware. This would while incapacitating the adaptation not cause any harm within the normal application. If an agreement round could not be finished during a period the system would be stuck with the current schedule. As each schedule is designed to run safely for the rest of a period, this would not lead to a failure of the systems. At the beginning of a new period the system automatically restarts with the baseline schedule. Just by adding the adaptation we do not introduce any potential risks into the system. Nonetheless the adaptation could not function after a link failure occurs which would mean that the fault recovery could not occur anymore and the overall system energy consumption would rise as the adaptation would not optimize the system any further.

The agreement is therefore a vital element to the adaptation chain. Each context value that was corrupted during transmission will lead to the receiving tile having an incorrect view on the system state. Without the reliable establishment of a global view on the current system state the adaptation is not able to safely function, as each tile decides on the schedule changes on their own. Inconsistent views therefore lead to inconsistent scheduling decisions, which finally result in runtime failures and message collisions.

This chapter will show how we expand the system architecture and the agreement protocol to a fault-tolerant architecture, which is able to provide additional communication links to handle hardware failures. We will show that the agreement will only provide safe context strings that under no circumstance will vary between the distributed tiles.

Our fault-tolerant protocol aims to detect changes within a message, as the presence of faults makes it possible that messages will be corrupted. The aim of the protocol is to prevent any corrupt context to be considered for adaptation decisions. This means that no schedule changes can be executed until the context string can be verified to be correct.

While many network protocols use additional acknowledgment rounds to verify if the original message was sent correctly, this kind of approach is not feasible in our system model for two reasons:

- The single connection approach without redundant hardware would render the acknowledgment useless in case of permanent faults or in case of transient faults that last longer than the retransmission time. In the FIFO ring the acknowledgment message (ACK) would need to use the same network as the originally sent message. Any corruption that happened during the transmission of the original message would also impact the ACK.

- The additional acknowledgment will increase the overhead of the protocol. In time-triggered MPSoCs time is of the essence, as every execution and communication is scheduled to fit within the application's deadline as well as the time interval with scheduled resource access. More importantly many context events require fast reactions (e.g. dynamic slack), otherwise they cannot be exploited. Therefore we cannot include additional acknowledgment rounds into the protocol to ensure that errors are detected at each round. The additional time will negatively impact the adaptation timing as the adaptation delay will increase.

Hence, to enable the system to cope with the hardware based faults described in the fault model in section 4 we need additional wires to have a redundant transmission link to the original ring network. However, the additional wires will lead to increased energy usage, which will be counter intuitive to the energy usage optimization performed by the adaptation. We must consider that in a not fault-tolerant implementation the adaptation and its energy saving advantages are rendered useless with the first occurrence of a fault. By investing into a higher baseline energy usage, we increase the potential energy savings and more importantly the system lifetime under the occurrence of a fault by up to 40% [Lenz and Obermaisser2017].

In the following sections we will introduce two architectures that provide fault-tolerant functionalities. One architecture provides fault resilience, as such that the fault is detected and any messages that were corrupted are dropped. It provides consistency while at the same time only adding a low amount of additional hardware and thus minimizing the energy overhead. The second version provides the system with enough additional functionalities to recover faults at each tile and provide consistency and availability to the agreement protocol at the cost of energy.

8.3 The Double-Ring Architecture

The first proposed fault-tolerant architecture adds a second link between each pair of tiles. This creates a second copy of the original message which can be used to evaluate the other message. The receiver can compare two messages from the same source and if they do not match the receiver knows that an error has occurred during the transmission.

The new links form the same ring structure through the system but use reverse directions.

Both messages are send separately on the two rings. This way each message uses its own network and any fault on one network does not impact the other network. As the messages are distributed differently on the ring they both need to pass the ring once before all tiles can compare them to determine correctness. As can be

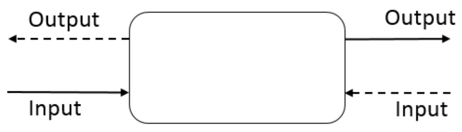

Figure 8.1: Input and output ports of a tile in a double ring architecture.

seen in figure 8.2 the tiles directly next to the source tile are the last ones to receive the second message.

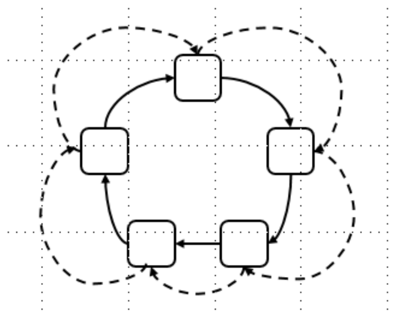

Figure 8.2: Double ring architecture.

We implement the second communication ring to ensure that each core receives the same information from two different transmission networks to ensure that the assumed single failure could not corrupt both messages as they arrive at the tile.

8.3.1 Fault Assumptions

We defined our FCR to cover the whole NI with the tile. This means that we accept that a fault can happen with this FCR but may not immediately impact the other FCRs.

We define two types of faults when talking about message correctness:

- *Corrupted Message*: A message is corrupted if it was received but their context is not identical to the originally sent message. This corruption could have been caused by tile failures during the relaying of the message, but also by transmission failures on the link.
- *Lost Message*: A message is lost if a link or a tile fails to transmit the message and the receiver never gets the message.

Both messages have different implications for the network in terms of error handling which will be discussed in section 8.3.2.

Message Verification In a double-ring architecture the receiving core is able to verify if the received message is correct by checking if the two incoming messages are identical. A bitflip could be detected directly with simple hardware methods like an XOR gate. Alternative message validation solutions can be implemented here as well, e.g. a cyclic redundancy check (CRC) or a checksum. These will add additional information about the message to the message itself. The receiving tile can then compare if the CRC or checksum information corresponds to the message bits. If the transmission would have caused bitflips the two message parts will not be consistent. We have used the XOR gate as it did not impact the context string layout. The size of the local context string limits the amount of context events. If one takes the smallest way to encode an event, by using just one bit per event, the maximal amount of events is restricted to 29 in our implementation. Each context string needs at least 3 bits for the tile ID, to ensure that a receiving tile knows where the message came from and the rest of the message can be used for the events. If one assumes a common bit size of 32 bits, the amount of events are implicitly defined. We did not want to further take bits away from the message for the CRC.

During the distribution round the messages are saved locally at each tile and relayed to the other channels. Once the distribution ends each tile can check if both messages that it received were identical. In cases where the messages match, the cores will save one copy of the context and delete the second one. If the messages are not correct they will both be dropped. In this stage each core also checks if its original context message was received back correctly. The source can tell which network failed, as it can compare both incoming messages with the original context it has saved at the beginning. This extra information round will be executed after the distribution at the end of the convergence step. A detailed flow of the protocol can be found in algorithm 12.

Finally it must be stated that if the tile receives two messages that are not identical, the tile cannot decide which message it can trust and is forced to drop these context values. The tile has no way of determining which message is correct and could be used. Hence, it will assume a safe value for the context to have a predefined safe response to the potential fault. This can lead to harmless bitflips during transmission leading the system to go into safe states. While this assumption may lead to less optimal usage of the hardware it will prevent corrupted information from influencing the system behavior negatively.

8.3.2 Fault Handling

The only way an error might impact the rest of the network is through messages going in and out of the FCR. As such we can define 4 different scenarios that our fault-tolerant protocol version needs to be able to cope with at runtime:

- Omission Fault of a Link
- Silent Tile Failures

- Arbitrary Failures
- Timing Failures

Omission Fault of a Link The transmission of the message failed due to sending, transmission or receiving errors. The receiving tile does not receive any message on the link. The receiving node knows due to its schedule when messages are scheduled to arrive. Further this error can be verified because the second link is by assumption still functioning and the second copy of the information is still received over this second ring. The second copy in this case is correct and can be used in the adaptation process. In this case the second ring works as a backup communication network for the agreement.

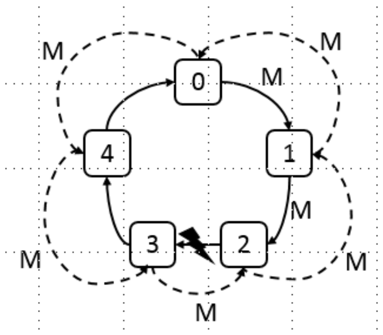

Figure 8.3: Link failure between tile 2 and 3 and the message traversal under its occurrence.

Figure 8.3 shows how the source tile 0 sends message M on the two networks. We can follow M through the two rings and see that the transmission on the clockwise ring ends after tile 2. All nodes after tile 2 do not receive a copy of M over the clockwise ring network. The second network is not impacted by the link failure and all nodes receive a correct copy of M on this ring.

Silent Tile Failures No message that is sent to the tile can be received or relayed. In such a case the double network serves to propagate the message to both sides of the ring without connecting at the broken tile. All tiles can receive at least a copy of the original message without being impacted by the missing link, as can be seen in figure 8.4

Figure 8.4 shows how the source tile 0 sends message M on the two networks. We can follow M through the two rings and see that the transmission on the clockwise ring ends after tile 3. All nodes after tile 3 do not receive a copy of M over the clockwise ring network. The second network is not impacted by the link failure and all nodes receive a correct copy of M on this ring.

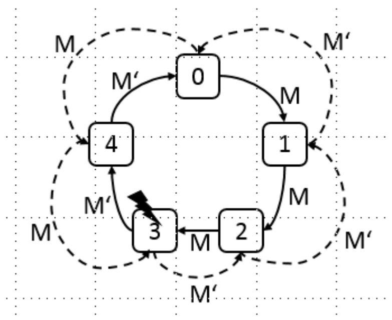

Figure 8.4: failure of tile 3 and the message traversal under its occurrence.

Arbitrary Failures In this case a tile *x* is malfunctioning and sending corrupted messages to the neighbors. All tiles that lie after tile *x* within the ring are unaware of the tile's wrong behavior. There are two types of faults that can occur in this case:

- All messages sent from the faulty tile are corrupted.
- Only some of the messages sent by the faulty tile are corrupted. This includes that the corruption only happens on one of the rings, or only occasionally on incoming messages.

Corruption of All Messages If all messages that are sent by the faulty tile are corrupted it is handled by the architecture. In this scenario a tile is corrupting all incoming messages before relaying them onto the next neighbor. This means that no matter at which input port the tile receives the message it will always send a corrupted message on the network. This means that all tiles that received a message before the faulty tile have a correct copy and the rest of the ring receives the corrupted one, as seen in figure 8.5.

The corrupted message leads to an invalidation of the correct message. With the two links, the receiver can no longer decide which message is correct and which one corrupted. Hence, both messages need to be dropped.

Figure 8.5 shows that source 0 is sending a message M. This time tile 3 is failing and corrupting the incoming messages. We can see that after passing tile 3 on both networks, the clockwise and counter clockwise one, the nodes receive a corrupted copy of the message M'. Looking at the message transfer on the clockwise network we get the following message transfers.

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \dashrightarrow 4 \dashrightarrow 0 \tag{8.1}$$

All messages sent by tiles before the faulty tile 3 are transmitted correctly. All messages after are corrupted, which is indicated by the broken arrows.

On the counter clockwise network we have the following message path.

$$0 \rightarrow 4 \rightarrow 3 \dashrightarrow 2 \dashrightarrow 1 \dashrightarrow 0 \tag{8.2}$$

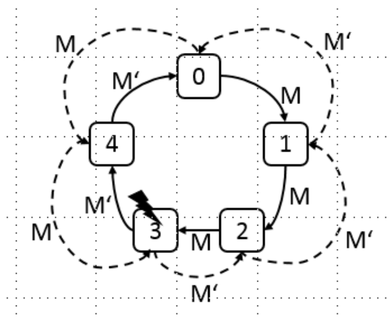

Figure 8.5: Message traversal on the ring under the occurrence of a consistent fault on tile 3

Looking at both networks together we see that all tiles received one correct and one corrupt message. Therefore they all are able to detect the message failure on their own.

Corruption of Only Some Messages If only some of the incoming messages are corrupted by a faulty tile parts of the network might not be aware of the failure happening on the network. If the node does not send corrupted messages to both outputs only one of the ring networks is corrupted. This will cause that only one side of the ring is corrupted and aware of a faulty behavior, while the other side assumes that the received context is correct, as can be seen in figure 8.6

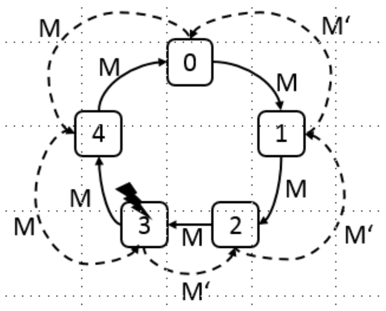

Figure 8.6: Message traversal on the ring under the occurrence of an arbitrary fault on tile 3

When we take a look at the message path we get that the clockwise path will show a correct message transmission over the ring:

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 0 \tag{8.3}$$

If we look at the counter-clockwise path, the tiles 1, 2 and tile 0 will receive corrupted messages M', again indicated by the dashed arrows. And the counter-clockwise message path shows all nodes are receiving correct messages:

$$0 \rightarrow 4 \rightarrow 3 \dashrightarrow 2 \dashrightarrow 1 \dashrightarrow 0 \qquad (8.4)$$

In this case tile 4 is not aware of a fault on the network and therefore a possible inconsistency of the network could occur. To prevent this the source node always informs the network about the correctness of the context using the two redundant rings. Even if the error occurs again, one of the rings will not be impacted by it and each tile will receive at least one error notification message. This way the network side that did not detect the faulty message, will also drop the context to ensure that a consistent global view is upheld.

Failure on the Source Tile There is a specific version of this fault, where the source tile is incorrect. The information which is sent originally is corrupted by the source tile. In this case no tile is able to detect the fault, as no correct message version is known within the network. This fault, while maybe leading to unoptimized and also unsafe system states, does not lead to inconsistencies in the network and is not part of the agreement protocol's scope. The protocol does not have the knowledge on the context information to do a sanity check of the input it receives. If desired plausibility checks or higher-level mechanisms can be realized. To prevent these kind of failure, we propose different levels of context types. The more impact a context type may have on your network the higher the context type should be. Context types of the highest level must be observed by at least three context monitors, to enable the system to verify the source value. Only if a majority of the context monitors observed a certain value the encompassing schedule change will be performed. This is to be taken into consideration during the design phase of the system.

Timing Errors It is also possible that erroneous tiles send the agreement messages out of synchronization to their neighbors and flood the neighboring receiver. This is comparable to a babbling idiot failure [Temple1998] where the receiver is rendered useless by the flood of incoming messages. In the case of our network the tiles can only impact their direct neighbors and thus the rest of the network is safe from the faulty tile. The neighbors themselves only read their incoming ports when they are scheduled to do so, therefore no matter how much the erroneous tile floods its neighbors, no negative impact can be observed on the tiles.

Fault Tolerant Agreement Protocol on a Double Ring During the context distribution each tile will save the received context values in a matrix, each row will represent the original source tile and the columns the input port it was received on. This way the AU can quickly compare both values in the convergence step of the protocol. The protocol is described in algorithm 12.

Algorithm 12 Context Distribution on a double ring

```
1: $input_0$ = input from port 0
2: $input_1$ = input from port 1
3: $output_0$ = output to port 0
4: $output_1$ = output to port 1
5: $m$ = source ID of message
6: $check[m,n]$ = matrix filled with the received context values
7: $context_g[]$ = global context array filled with zeros
8: $error_g[]$ = global error array filled with zeros
9: Begin of the local context exchange at triggering-time
10: for $j = 0$ to amount of $tiles - 1$ do
11:     save the input into a local array:
12:     $check[j,0] \leftarrow input_0$
13:     $check[j,1] \leftarrow input_1$
14:     $output_1 \leftarrow input_0$
15:     $output_0 \leftarrow input_1$
16: end for
17: After the exchange finished: compare the matrix rows for inconsistencies
18: for $j = 0$ to amount of $tiles - 1$ do
19:     if $check[j,0]$ equals $check[j,1]$ then
20:         $context_g[j] \leftarrow check[j,1]$
21:     else
22:         $error_g[j] \leftarrow' 1'$
23:     end if
24: end for
```

8.3.3 Convergence of Context Values

In the system model in section 4 we have introduced the notion of FCR types for context values for the original context that is observed by the CMs and cannot be verified by the protocol. In the convergence phase we can apply these models to correct the global context string before it is relayed to the adaptation manager.

Based on the predefined context types each NI is aware of how to handle faulty messages. If the context is not safety relevant the value can just be dropped and the reserved context in the final global context string will be set to 0. If the values indicate a potential fault in the system, that needs some action, but the amount of data can only support an error detection, the values will be set to the predefined safe value. This safe value is comparable to the system safe state that acknowledges that something might be amiss but no further actions must be taken. The context state variables without safe values, ref section 8.1.1, requires the system to always provide the context information from 3 different sources. If one of these messages is erroneous, because of a CM fault or a faulty transmission on the network the other two values will indicate the correct context value.

8.4 Triple Ring

By adding a third ring to the architecture we have a set of three messages sent on the network which can be used to recover the original message by voting on it.

To expand the architecture with a third ring we chose an additional link between two tiles that are not direct neighbors.

The hardware is extended similarly to the double ring by a third pair of receiver and sender hardware which is directly connected to the respective source and destination tiles. All links are unidirectional to minimize the additional hardware overhead.

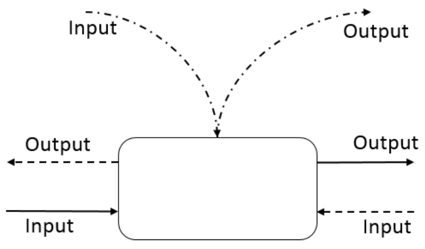

Figure 8.7: Additional input and output ports on a tile in a triple ring architecture.

The additional hardware leads to three messages being injected to the network at each agreement instance. The structure is adapted from the braided ring structure for distributed systems described in [Paulitsch and Hall2007]. The skipping link will help to push the information and not be vulnerable to tile failures in the same way as the normal links are. By skipping a tile the same source will not have a bottleneck at the same faulty tile, as at least one link will be able to skip it.

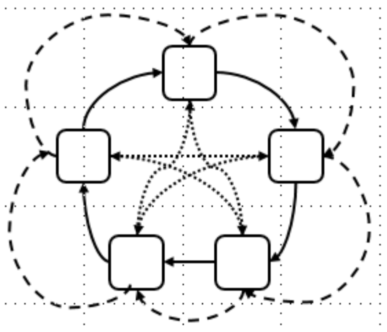

Figure 8.8: Architecture of a triple ring.

8.4.1 Propagation on the Network

The new architecture adds the advantage of additional redundant information. The three incoming messages can be used to vote on the correct message. As we assume that at one time only one error may occur, only on of the incoming messages can be corrupted, thus the node is able to vote on the correct message based on the two other incoming messages. Each message uses a dedicated set of links therefore ensuring that no link failures can impact the two of the three messages during the propagation. By having link skip a tile, the propagation gets more difficult as not all types of networks can be divided into two subnetworks that span over all nodes. If using the *skipping* links a message send only via these links can reach all nodes only if the network consists of an odd amount of nodes. But in networks with an even amount of tiles, the tiles connected by the skipped links will form a subnetwork containing half of the nodes where the message cannot break out and reach the other half of the tiles, as seen in figure 14.

The propagation must therefore vary in its message handling. For networks with an odd amount of tiles we will use the skipped link network. But when handling network sizes of even amounts of tiles, the incoming message will be propagated on the opposite exit link. Meaning that a message received on the skipped link will be send to the next neighbor via the direct link. This way we can ensure that the messages on the direct and the messages on the skipped link will be received by all nodes. By having all nodes receive three versions of the original message we can assure that each node can vote on a correct message version as they always have at least two correct messages. Each of the original three messages takes a unique path through the network, therefore assuring that a failure cannot impact more than one of the messages. Each link has a dedicated sender and receiver, meaning that no link failures can impact more than one link.

The unanimous view on the system state takes longer to establish as the messa-

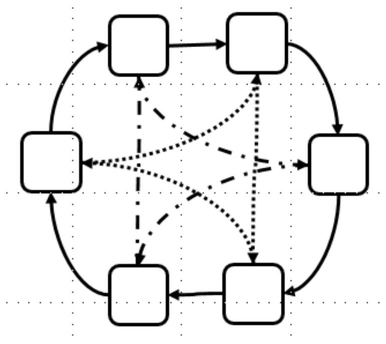
Figure 8.9: network ring with 6 tiles and its connections. The figure shows how the skipping links form subnetworks that cannot be left by the message. The two networks are indicated by the two different dashed arrows. The baseline ring network must be used as a link between those two networks to ensure that the messages reach all tiles in case of an even number of tiles.

ges need to circle the ring longer to reach all tiles. We have established that based on the network size n the time until all tiles received all three original messages is $(n+1) * transmission\ time$.

One can see that the tiles need to be able to distinguish which of the two originally sent messages they received. Therefore the agreement string is extended by a flag bit that indicates if the message was originally sent on the direct link, flag 0, or on the skipped link, flag 1.

Looking at the new structure one can see, that the messages going over the skipped path will only converge to visiting all the tiles if the number of tiles n modulo 3 does not equate 0. If the amount of tiles is a multiple of 3 the message will only move along a subset of the ring as it lands on its original source tile after $n-2$ steps.

8.4.2 Fault Resilience with the Triple Ring

Different to the double ring, the triple ring enables the tiles to vote on the correct value and thus recover the context that was originally sent. We evaluate the triple ring under the same fault assumptions as the double ring to evaluate how the triple redundancy improves the behavior under faults. The FCR is the whole tile with the NI and only one fault is assumed in our model. In this section we will look how the third ring enhances the fault recovery. Any faults that are already handled by the double ring will not be addressed further, as the double ring is a part of the triple ring and the advantages of it apply to the triple ring as well.

Arbitrary Faults If a tile x is malfunctioning and sending corrupted messages to the neighbors the double ring before just ensured the consistency of the messages. The third link enables the system to also provide availability of the adaptation, by enabling the tile to identify a correct message under the presence of a corrupted one. Thus, the tile is able to perform the adaptation despite the hardware fault.

By comparing the incoming messages to each other the tiles can detect if one message was corrupted. Further the message which was received over the third channel can be used to vote on the correct message. Before the tiles only had a corrupt and a correct message and could not verify which one to use for the adaptation. Now the tiles can perform a majority voting on the three messages. As we only assume that one failure occurs, only one of the messages can be corrupted and the two other ones must be identical. By finding the two identical messages, the tiles know which context they can use for the adaptation and safely perform system changes even though a fault occurred at runtime.

Global System State The uniform global system state is reached faster in the triple ring, reducing the execution time of the protocol. The triple redundancy enables all tiles to correct mistakes and no information must be dropped. Therefore we provide availability of the information at runtime.

8.4.3 Protocol description

The protocol used on the triple ring is expanded by a voting mechanism for the convergence phase. After the three messages have been delivered to all the tiles and returned to their original sender on the ring, all tiles vote on the majority of their three received inputs and use this value for the adaptation scheme.

8.5 Conclusion

The double ring enables the system to reliably continue to use adaptation techniques when a tile or a link fails. The fault-tolerant version not only uses low hardware overhead to provide the fault-tolerant functionality, it further also minimizes the timing overhead by adding only a second exchange phase of $n * message\ transmission\ time$ to the runtime behavior. The system is only able to be preserve consistency, meaning that a fault is not able to impact the consistency of the context information of the system.

The two message redundancy can only detect errors and not determine which of the messages is correct. The system is limited in its behavior once a fault is corrupting messages without dropping them. By corrupting messages this way the faulty core can keep the system from using the adaptation properly, as it could corrupt all context messages and force the tiles to drop them all. To identify which of the two messages is correct, we would need to add functionalities with which the tiles can verify the message's authenticity. Authenticating messages means that one can prove that the message was sent in exactly the way it was received.

The triple ring enables the system to self correct in case of failure. The added ring allows each node to vote for the correct message and thus directly pipeline

Algorithm 14 Context Distribution

```
1: $input_0$ = input from input port 0
2: $input_1$ = input from input port 1
3: $input_2$ = input from input port 2
4: $output_0$ = output to port 0
5: $output_1$ = output to port 1
6: $output_2$ = output to port 2
7: $m$ = source ID of message
8: $check[m,n]$ = matrix with $m = [0..\ amount\ of\ tiles]$ and $n = [0..2]$
9: $context_g[]$ = global context string filled with zeroes
10: $t_p$ Period of Agreement execution
11: if $t = t_p$ then
12:     for $j = 0$ to amount of $tiles - 1$ do
13:         $check[j,0] \leftarrow input_0$
14:         $check[j,1] \leftarrow input_1$
15:         $check[j,2] \leftarrow input_2$
16:         $output_0 \leftarrow input_0$
17:         $output_1 \leftarrow input_1$
18:         $output_2 \leftarrow input_2$
19:     end for
20:     for $j = 0$ to amount of $tiles - 1$ do
21:         if $check[j,0]$ equals $check[j,1]$ then
22:             $context_g[j] \leftarrow check[j,0]$
23:         else
24:             if $check[j,0]$ equals $check[j2]$ then
25:                 $context_g[j] \leftarrow check[j,0]$
26:             else
27:                 if $check[j,1]$ equals $check[j,2]$ then
28:                     $context_g[j] \leftarrow check[j,1]$
29:                 end if
30:             end if
31:         end if
32:     end for
33: end if
```

the correct context values into the adaptation's mode change machine. By adding the third ring we thus gain additional time as failures no longer lead to safe state decisions of the adaptation, but the optimized adaptation can be performed. A negative aspect of the third ring is the increased amount of wires used to set the system up.

While most NoC communities refuse to add additional wires to the hardware layout due to the energy and area usage overhead, the safety advantage of the ring is striking. While one may argue that adding more hardware to save energy is counter-intuitive we must advise that the strength of the global adaptation is the reconfigurability of the system at runtime. By applying the adaptation scheme we enable a system to react to unforeseen events like faults in a safe and controlled manner. This enhances the safety argument of the system, as safety critical operation modes can be entered to optimize the system performance under the stress of faults.

Chapter 9

Conclusion

In this thesis, we introduced the concept of global adaptation on an MPsoC and defined an architecture that is needed to provide the the needed services. After pointing out the importance of a common global state we then introduced the main focus of the thesis: a protocol to establish a global state within a NoC.

We defined a protocol outline and evaluated how an optimal implementation can be achieved under different network condition. It can be implemented in different ways, where both of them have respective advantages and disadvantages: one of which needs 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 for NoC which are frequently use for application communication.

Further we has defined a fault tolerant version of the protocol. For this we have proposed two architectures that provide different level of fault tolerance.

The double ring provides consistency even under the occurrence of faults while only adding a second ring structure and keeping the hardware overhead minimal. The second architecture adds another ring network but also ensures availability of the messages under the occurrence of faults. This triple ring provides enough redundant network links to enable each node to vote on the correct value which allows the tiles to correct any fault that impacts the incoming messages.

For future work we planned to enable the double ring to self correct. For that the double ring can be expanded with a message authentication scheme, as the main drawback is the missing message authentication to provide availability. While the CRC could be used to detect errors, it was not enable the double ring to correct the message. By signing messages the tiles of the double ring can decide which message was sent and received correctly without having to resent or to drop the message. Message authentication is largely known from cryptographic applications, where the authenticity of a message is a key requirement. In cryptographic applications the receiver must be able to proof that the original message was received correctly. In this section we will look at cryptographic message authentication solutions and discuss which is applicable for our agreement protocol.

Typical solutions are *Message Authentication Codes* (MAC) [Simmons1988]. MACs are trapdoor functions that map a distinct value to an input string. The MAC value is sent as a postfix of the message. A receiver can now perform the MAC function on the message string and compare it to the received MAC value. If they are identical the message was not tempered with. If they do not match, the message was altered. This identification can be done, because each change in the input string of the function will lead to a drastic change within the mapped MAC value. Any change would therefore be detected if the Mac value that was sent and the Mac value that was received are not identical. Each bit change to the input string produces a completely different output, hence there is no correlation between the input strings.

If the double ring would be enhanced with a MAC solution to sign messages before sending it on the network, the tiles could recover faults at runtime. There are specific lightweight cryptographic functions that are optimized for the resource restriction given by embedded systems. These functions operate with a minimal usage of area on the FPGA [Eisenbarth et al.2007]. It must be considered that if one talks about cryptographic functions, no matter how lightweight they are, they take up a considerable amount of area and time compared to the basic XOR functions that were used in the triple ring. The triple ring further reduces the protocol execution time and provides a fault resilience to hardware faults with minimal hardware overhead.

Bibliography

[mes] Mesosphere 2017. dc7os (2017). `https://docs.mesosphere.com/overview`.

[Ahmadian; H.; Obermaisser2015] Ahmadian; H.; Obermaisser, R. (2015). Time-triggered extension layer for on-chip network interfaces in mixed-criticality systems. pages 693–699.

[Aikebaier et al.2010] Aikebaier, A., Enokido, T., and Takizawa, M. (2010). Reliable message broadcast schemes in distributed agreement protocols. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 242–249. IEEE.

[Al-Nakhala et al.2013] Al-Nakhala, N., Riley, R., and Elfouly, T. M. (2013). Binary consensus in sensor motes. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1337–1342. IEEE.

[Axer et al.2014] Axer, P., Ernst, R., Falk, H., Girault, A., Grund, D., Guan, N., Jonsson, B., Marwedel, P., Reineke, J., Rochange, C., et al. (2014). Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):82.

[Barborak et al.1993] Barborak, M., Dahbura, A., and Malek, M. (1993). The consensus problem in fault-tolerant computing. *ACM Computing Surveys (CSur)*, 25(2):171–220.

[Barcelona2008] Barcelona, C.-S. (2008). Mencius: building efficient replicated state machines for wans.

[Binkert2011] Binkert, N.; Beckmann, B. B. G. R. S. S. A. B. A. H. J. H. D. K. T. S. S. e. a. (2011). The gem5 simulator. pages 1–7.

[Boichat et al.2003] Boichat, R., Dutta, P., Frølund, S., and Guerraoui, R. (2003). Deconstructing paxos. *ACM Sigact News*, 34(1):47–67.

[Bracha and Toueg1985] Bracha, G. and Toueg, S. (1985). Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840.

[Burns and David2016] Burns, A. and David, R. (2016). Mixed criticality systems a review.

[Chandra et al.1996] Chandra, T. D., Hadzilacos, V., and Toueg, S. (1996). The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722.

[Chen and Kuo2007] Chen, J. and Kuo, C. (2007). Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In *Proc. International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE.

[Chor and Coan1985] Chor, B. and Coan, B. A. (1985). A simple and efficient randomized byzantine agreement algorithm. *IEEE Transactions on Software Engineering*, (6):531–539.

[Corbett et al.2013] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. (2013). Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8.

[Cristian et al.1986] Cristian, F., Aghili, H., Strong, R., and Dolev, D. (1986). *Atomic broadcast: From simple message diffusion to Byzantine agreement*. International Business Machines Incorporated, Thomas J. Watson Research Center.

[DeCandia et al.2007] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM.

[Défago et al.2004] Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421.

[Dimakis et al.2006] Dimakis, A. G., Sarwate, A. D., and Wainwright, M. J. (2006). Geographic gossip: Efficient aggregation for sensor networks. In *Proceedings of the 5th international conference on Information processing in sensor networks*, pages 69–76. ACM.

[Dolev et al.1987] Dolev, D., Dwork, C., and Stockmeyer, L. (1987). On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97.

[Dwork et al.1988] Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323.

[Eisenbarth et al.2007] Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A., and Uhsadel, L. (2007). A survey of lightweight-cryptography implementations. *IEEE Design Test of Computers*, 24(6):522–533.

[Fischer et al.1990] Fischer, M. J., Lynch, N. A., and Merritt, M. (1990). Easy impossibility proofs for distributed consensus problems. In *Fault-tolerant distributed computing*, pages 147–170. Springer.

[Fischer et al.1982] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1982). Impossibility of distributed consensus with one faulty process. Technical report, Massachusetts Inst of Tech Cambridge lab for Computer Science.

[Goergen and Schreiner2016] Goergen, R., G. D. G. K. L. L. and Schreiner, S. (2016). Integrating power models into instruction accurate virtual platforms for arm-based mpsocs. pages 24–26.

[Guan2016] Guan, N. (2016). *Techniques for building timing-predictable embedded systems*. Springer.

[Guerraoui and Schiper1997] Guerraoui, R. and Schiper, A. (1997). Consensus: the big misunderstanding [distributed fault tolerant systems]. In *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 183–188. IEEE.

[Hadzilacos and Toueg1994] Hadzilacos, V. and Toueg, S. (1994). A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University.

[Holland1992] Holland, J. H. (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.

[Hong1999] Hong, Inki, e. a. (1999). Power optimization of variable-voltage core-based systems. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. IEEE.

[Hunt et al.2010] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA.

[Krishnamoorthy and Krishnamurthy1987] Krishnamoorthy, M. S. and Krishnamurthy, B. (1987). Fault diameter of interconnection networks. *Computers & Mathematics with Applications*, 13(5-6):577–582.

[Lamport1998] Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169.

[Lamport2005] Lamport, L. (2005). Generalized consensus and paxos.

[Lamport et al.2001] Lamport, L. et al. (2001). Paxos made simple. *ACM Sigact News*, 32(4):18–25.

[Lamport et al.1982] Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401.

[Laranjeira et al.1991] Laranjeira, L. A., Malek, M., and Jenevein, R. (1991). On tolerating faults in naturally redundant algorithms. In *[1991] Proceedings Tenth Symposium on Reliable Distributed Systems*, pages 118–127. IEEE.

[Lenz and Obermaisser2017] Lenz, A. and Obermaisser, R. (2017). Global adaptation controlled by an interactive consistency protocol. *Journal of Low Power Electronics and Applications*, 7(2):13.

[Lenz et al.2017] Lenz, A., Pieper, T., and Obermaisser, R. (2017). Global adaptation for energy efficiency in multicore architectures. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*, pages 551–558. IEEE.

[Li and Ampadu2015] Li, C. and Ampadu, P. (2015). Energy-efficient noc with variable channel width. In *IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE.

[Li et al.2013] Li, Y., Zhou, Z., and Sato, T. (2013). A cluster-based consensus algorithm in a wireless sensor network. *International Journal of Distributed Sensor Networks*, 9(3):547124.

[Liu and Wang2011] Liu, W.; Xu, J. W. X. Y. Y. W. X. Z. W. N. M. and Wang, Z. (2011). A noc traffic suite based on real applications. pages 66–71.

[Marandi et al.2012] Marandi, P. J., Primi, M., and Pedone, F. (2012). Multi-ring paxos. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE.

[Moraru et al.2013] Moraru, I., Andersen, D. G., and Kaminsky, M. (2013). There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM.

[Mostefaoui and Raynal2000] Mostefaoui, A. and Raynal, M. (2000). A condition for k-set agreement in asynchronous distributed systems. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 6–pp. IEEE.

[Obermaisser2019] Obermaisser, R.; Ahmadian, H. M. A. B. Y. L. A. S. B. (2019). Adaptive time-triggered multi-core architecture. *Designs 2019*.

[Obermaisser2012] Obermaisser, R. (2012). *Time-triggered communication*. CRC Press.

[Obermaisser et al.2008] Obermaisser, R., Salloum, C. E., Huber, B., and Kopetz, H. (2008). The time-triggered system-on-a-chip architecture. In *2008 IEEE International Symposium on Industrial Electronics*, pages 1941–1947.

[Ogrenci Memik2001] Ogrenci Memik, S.; Bozorgzadeh, E. K. R. S. M. A. (2001). super-scheduler for embedded reconfigurable systems. In *In Proceedings of the IEEE/ACM International Conference on Computer Aided Design ICCAD*, pages 391–394. IEEE/ACM.

[Paterna and Benini2013] Paterna, Francesco, A. A. and Benini, L. (2013). Aging-aware energy-efficient workload allocation for mobile multimedia platforms. In *IEEE Transactions on Parallel and Distributed Systems*, pages 1489–1499. IEEE.

[Paulitsch and Hall2007] Paulitsch, M. and Hall, B. (2007). Insights into the sensitivity of the brain (braided ring availability integrity network)–on platform robustness in extended operation. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 154–163. IEEE.

[Perron et al.2009] Perron, E., Vasudevan, D., and Vojnovic, M. (2009). Using three states for binary consensus on complete graphs. In *IEEE INFOCOM 2009*, pages 2527–2535. IEEE.

[Persya2013] Persya, A.C.; Gopalakrishnan Nair, T. (2013). Model based design of super schedulers managing catastrophic scenario in hard real time systems. In *In Proceedings of the IEEE 2013 International Conference on Information Communication and Embedded Systems (ICICES)*. IEEE.

[Poke et al.2017] Poke, M., Hoefler, T., and Glass, C. W. (2017). Allconcur: Leaderless concurrent atomic broadcast. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 205–218. ACM.

[Sarwar1997] Sarwar, A. (1997). Cmos power consumption and cpd calculation. *Proceeding: Design Considerations for Logic Products*.

[Schlichting and Schneider1983] Schlichting, R. D. and Schneider, F. B. (1983). Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3):222–238.

[Schüpbach et al.2008] Schüpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T., and Isaacs, R. (2008). Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, volume 27.

[Simmons1988] Simmons, G. J. (1988). A survey of information authentication. *Proceedings of the IEEE*, 76(5):603–620.

[Steiner and Kopetz2006] Steiner, W. and Kopetz, H. (2006). The startup problem in fault-tolerant time-triggered communication. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 35–44.

[Temple1998] Temple, C. (1998). Avoiding the babbling-idiot failure in a time-triggered communication system. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, pages 218–227.

[Thomas and Luk2009] Thomas, D.B., H. L. and Luk, W. A. (2009). A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. pages 63–72.

[Unterbrunner et al.2014] Unterbrunner, P., Alonso, G., and Kossmann, D. (2014). High availability, elasticity, and strong consistency for massively parallel scans over relational data. *The VLDB JournalThe International Journal on Very Large Data Bases*, 23(4):627–652.