UNIVERSITÄT SIEGEN

Naturwissenschaftlich Technische Fakultät

Predictable Transactional Memory Architecture for Hierarchical Mixed-Criticality Systems

DISSERTATION
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation von:
Zaher Owda

vom Fachbereich Elektrotechnik und Informatik der Universität Siegen
Siegen - Oktober 2016

———

die Promotionskommission:
Prof. Dr. Roman Obermaisser, Universität Siegen
Prof. Dr. Achim Rettberg, Universität Oldenburg
Prof. Dr. Roland Wismüller, Universität Siegen
Prof. Dr. Marcin Grzegorzek, Universität Siegen

Tag der mündlichen Prüfung:13 März 2017

UNIVERSITÄT SIEGEN

Naturwissenschaftlich Technische Fakultät

Predictable Transactional Memory Architecture for Hierarchical Mixed-Criticality Systems

This dissertation is submitted for the degree of

Doctor of Engineering

by

Zaher Owda

Submitted to the Faculty of Science and Technology of
the University of Siegen
Siegen - October 2016

Examination commission:
Prof. Dr. Roman Obermaisser, Universität Siegen
Prof. Dr. Achim Rettberg, Universität Oldenburg
Prof. Dr. Roland Wismüller, Universität Siegen
Prof. Dr. Marcin Grzegorzek, Universität Siegen

Defense date:13 March 2017

To my mother and the living memory of my *father*

⋆⋆⋆⋆⋆

To my beloved wife and our source of happiness *Eyas*

Acknowledgements

There is no words to utter my gratitude to all incredible people that have encouraged and motivated me during this phase of my life.

I would like to express my sincere acknowledgment to my late father, my mother and sister for their infinite love and patience. My supportive wife and lovely son *Eyas* I cannot thank you enough for the inspiration you are giving me.

I greatly appreciate the professional support and advice received from the members of my examination committee. Particularly, many thanks for Prof. Dr.-Ing. Roman Obermaisser for his guidance, assistance and constructive criticism throughout all the years of our cooperation. Moreover, I am very grateful to Prof. Dr. Achim Rettberg from the The Carl von Ossietzky University of Oldenburg for the collaboration and for his insightful feedback.

Finally, my appreciation is also extended to all the colleagues and friends with whom I had many passionate and fruitful discussions.

Abstract

A transactional memory simplifies the concurrency management in multicore systems by permitting sets of load and store instructions to be executed in an atomic way. The correct results for concurrent transactions and the execution time strongly depend on the coherency potentials, rollback capabilities and strategies of the transactional memory.

A transactional memory can be implemented as a Hardware Transactional Memory (HTM), as a Software Transactional Memory (STM), or as a hybrid combination of both called Hybrid Transactional Memory (HyTM). STM is the most common implementation of the transactional memory models, which is slower but simpler and more flexible than hardware transactional memories. HyTM is an approach that combines both STM and HTM by using architectural support to accelerate particular algorithms of the STM or by allowing hardware and software transactions to operate in the same address space.

Mixed-Criticality Systems (MCSs) combine applications and subsystems at different levels of criticality on multicore systems. The development of such a safety-critical architecture requires a transactional memory architecture that guarantees the predictability, fault isolation and heterogeneity of concurrent safety-critical subsystems. Available transactional memory architectures do not support mixed-criticality at the chip level. Additionally, existing memory solutions spanning from multi-core chips to the cluster level are missing. A hierarchical transactional memory protocol is required to provide hierarchical support at all levels of the system architecture.

In this dissertation, two transactional memory architectures are proposed, namely a transactional memory for chip level architectures and a hierarchical transactional memory architecture for both multi-core chips and the cluster level.

In case of the chip-level transactional memory architecture, the predictability of the memory operations is guaranteed based on a global time base and the interarrival times of transactions. Different roll-back strategies with selective committing/aborting of requests are introduced based on the criticality of the components. This requires additional functionalities of the transactional memory such as temporal and spatial partitioning.

The hierarchical solution extends the previously mentioned properties and services to a hierarchical transactional memory protocol that guarantees the requirements for distributed

MCSs. This architecture includes novel transactional memory extensions at cores, network interconnections, memory and network gateways.

The proposed transactional memory architectures introduce and exploit novel transactional memory algorithms and protocols developed for MCSs. The applied scientific and technical methods include the definition of the system and memory architecture with novel conceptual models and algorithms. A trace-based simulation framework was implemented in systemC to simulate the chip-level architecture. Additionally, this framework was extended to a co-simulation framework combining systemC with AUTOSAR for the experimental evaluation of the models and algorithms of the proposed hierarchical transactional memory architecture. Use cases from the automotive area served for the evaluation.

Better fault isolation at all levels of the chip and cluster components is obtained due to the proposed architectures. The presented solutions handle efficiently the temporal predictability at transaction level, interconnection level, memory gateway level and cluster level. For the first time, a hierarchical transactional memory–based architecture for MCS supporting chip and cluster level is presented. The proposed protocol concurrently manages the reliable execution of MCS transactions. Finally, the proposed protocol is technology independent and hides the heterogeneity of the components.

Kurzfassung

Ein transaktionaler Speicher vereinfacht das Nebenläufigkeitsmanagement in Mehrkernsystemen, indem Sätze von Lade- und Speicherbefehlen auf atomare Weise ausgeführt werden. Die korrekten Ergebnisse für gleichzeitige Transaktionen und die Ausführungszeiten hängen stark von den Kohärenzpotentialen, Rollback-Fähigkeiten und Strategien des transaktionalen Speichers ab. Ein transaktionaler Speicher kann als Hardware-Transaktionsspeicher implementiert werden (HTM), als Software-Transaktionsspeicher (STM) oder als Hybrid-Kombination von Hardware und Software (HyTM). STM ist die häufigste und flexibelste Implementierung, jedoch langsamer als ein Hardware-Transaktionsspeicher. HyTM ist ein Ansatz, der sowohl STM als auch HTM kombiniert, indem Architekturunterstützung verwendet wird, um bestimmte Algorithmen des STM zu beschleunigen oder Hardware- und Software-Transaktionen im gleichen Adressraum zu ermöglichen.

Mixed-Criticality-Systeme (MCS) kombinieren Anwendungen und Subsysteme auf verschiedenen Ebenen der Kritikalität eines Multicore-Systems. Die Entwicklung einer solchen sicherheitskritischen Architektur erfordert eine transaktionale Speicherarchitektur, die die Vorhersagbarkeit, Fehlerisolierung und Heterogenität von gleichzeitigen sicherheitskritischen Subsystemen gewährleistet. Verfügbare Transaktionsspeicherarchitekturen unterstützen keine gemischte Kritikalität auf der Chipebene. Darüber hinaus fehlen vorhandene Speicherlösungen, die sich von Multi-Core-Chips bis auf die Cluster-Ebene erstrecken. Ein hierarchisches Transaktionsspeicherprotokoll ist erforderlich, um hierarchische Unterstützung auf allen Ebenen der Systemarchitektur bereitzustellen.

In dieser Dissertation werden zwei transaktionale Speicherarchitekturen vorgeschlagen, nämlich ein Transaktionsspeicher für Chip-Level-Architekturen und eine hierarchische transaktionale Speicherarchitektur für Multi-Core-Chips und die Cluster-Ebene.

Im Falle der transaktionalen Speicherarchitektur auf der Chip-Ebene wird die Vorhersagbarkeit der Speicheroperationen basierend auf einer globalen Zeitbasis und den Zwischenankunftszeiten von Transaktionen garantiert. Auf der Grundlage der Kritikalität der Komponenten werden unterschiedliche Roll-Back-Strategien mit selektivem Commit/Abort von Requests eingeführt. Dies erfordert zusätzliche Funktionalitäten des Transaktionsspeichers wie zeitliche und räumliche Partitionierung.

Die hierarchische Lösung erweitert die zuvor erwähnten Eigenschaften und Dienste zur Realisierung eines hierarchischen Transaktionsprotokolls, das die Anforderungen für verteilte MCS gewährleistet. Diese Architektur enthält neue Transaktionsspeichererweiterungen bei Prozessorkernen, beim Netzwerk, sowie bei Speicher- und Netzwerk-Gateways.

Die vorgeschlagenen Architekturen führen neuartige Transaktionsalgorithmen und Protokolle ein, die für MCSs entwickelt wurden. Die angewandten wissenschaftlichen und technischen Methoden umfassen die Definition der System- und Speicherarchitektur mit neuen konzeptuellen Modellen und Algorithmen. Eine trace-basierte Simulationsumgebung wurde in SystemC implementiert, um die Chip-Level-Architektur zu evaluieren. Darüber hinaus wurde dieses Framework auf eine Co-simulations Umgebung erweitert, die SystemC mit AUTOSAR kombiniertum die hierarchische transaktionale Speicherarchitektur zu evaluieren.

Ergebnisse umfassen einebessere Fehlerisolation auf allen Ebenen des Chips und der Cluster-Komponenten aufgrund der vorgeschlagenen Architekturen. Die vorgestellten Lösungen behandeln die zeitliche Vorhersagbarkeit auf der Transaktionsebene, der Netzwerkebene und der Speicher-Gateway-Ebene. Erstmals wird eine hierarchische transaktionsspeicherbasierte Architektur für MCS vorgestellt.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Mixed-criticality systems combine subsystems with different criticality levels on a shared computing platform. Mixed-criticality systems provide potential for higher reliability, energy efficiency and adaptability while also reducing the size, cost and weight of embedded systems [OOA+14a]. The increasing demand for these properties can be observed in many application domains (e.g. automotive, avionics and healthcare systems). Mixed-criticality systems are a key enabler to reduce the cabling and the number of devices in many application domains including automotive, avionic, healthcare and industrial systems [OOA+14a]. The resulting benefits include lower cost, weight and maintenance efforts due to the higher integration and the more efficient use of computational resources.

Transactional memories were initially introduced for databases and later they were proposed as a solution for concurrency control to ease parallel programming and parallel processing in multicore systems. Three types of transactional memories can be distinguished based on their implementation: hardware (e.g, 4^{th} generation Intel processor [Int14]), software (e.g., D2STM [CRCR09] and DiSTM [KAJ+08]) and hybrid combinations of both [RMN+], [CGS+14], [BNZ08]. A transactional memory introduces atomicity of memory instructions of a complete transaction in multicore systems. In addition, it manages concurrent executions and ensures consistency of the transactions.

The use of transactional memories has been proposed in dependable embedded systems with challenges of handling faults, real-time constraints and consistency of transactional memories [FF11]. Correct execution of concurrent transactions and the WCET strongly depend on the type (i.e. eager or lazy) of the version management, conflict detection, conflict resolution services, and the available computational resources of the transactional memory [MWU13].

Transactional memories are also useful for easing the development of mixed-criticality systems, where subsystems with different safety assurance levels coexist on a shared com-

puting platform. Mixed-criticality systems comprise subsystems with varying degrees of assurance and timing guarantees [Ves07b]. A prerequisite for the deployment of transactional memories in mixed-criticality systems is the prevention of unintended interference of the transactions with different criticality.

Transactional memories offer the potential to combine an optimistic operation of non safety-critical subsystems with strict real-time guarantees of safety-critical subsystems. However, a fundamental requirement in mixed-criticality systems is the segregation of subsystems with different criticality. Non safety-critical tasks should either have no effect at all on the execution times of safety-critical tasks or the effect must be bounded and known.

In case of transactional memories we can identify the following three sources of temporal interference between computational cores:

1. **Conflict Resolution of Transactions:** Transactional memories have the potential to cause interference between tasks due to rollbacks, which must be controlled and analyzed as a prerequisite for the deployment in mixed-criticality systems.

2. **Interconnect:** Segregation between applications of different criticality must be addressed at the on-chip interconnect where memory requests and data are exchanged between the computational cores and the memory gateway.

3. **Memory Gateway:** The memory gateway has to schedule memory requests from different computational cores. Typically, memory controllers optimize throughput, while temporal interference between cores is not addressed (e.g., bank switching).

Therefore, mixed-criticality systems require a transactional memory to provide concurrency control and timing guarantees for the subsystems by prohibiting temporal interference of the transactions in the transaction management, the interconnect and the memory controller.

Major research gaps are architectures, system models and algorithms for transactional memories in hierarchical mixed-criticality systems comprising networked multi-core chips. These systems combine two integration levels. Firstly, a multi-core processor consists of a set of computational cores that interact via an on-chip interconnect. Secondly, the cluster-level uses off-chip networks for the interconnection of several multi-core processors. A single multi-core chip is often insufficient to meet the resource requirements of large embedded applications. In addition, the failure rates of a single chip are too high to meet the reliability requirements of fail-operational systems with ultra-high dependability [SWH95] (e.g., Class A according to DO-178C [RTC11]). Hence, fault-tolerance at system level is required by exploiting redundancy with multiple independent chips.

1.1 Contributions

This thesis introduces a mixed-criticality aware architecture, starting at the chip-level and then extending a hierarchical distributed systems. The architectures offer temporal predictability and fault isolation of the transactional memory in mixed-criticality systems. The algorithms and protocols for transaction management and conflict resolution ensure that the execution time of safety-critical applications does not depend on applications of lower-criticality. This property is significant for modular certification, where separate safety arguments are established for application subsystems with different criticality levels. In contrast, in case the conflict resolution would not prevent low-critical applications from affecting safety-critical ones, the criticality of all application subsystems would be elevated to the highest criticality level in the system.

Additionally, memory operations of the hierarchical architecture are managed in a distributed manner at the cores of different multi-core chips. This is achieved by executing a transactional-memory protocol spanning both on-chip and off-chip networks while hiding the heterogeneity of the implementation technologies. Memory pages are relocated between the local caches at cores and the external memories of the multi-core chips. In addition, commits and rollbacks are performed to ensure atomicity, consistency and isolation in the presence of memory conflicts.

The contributions of this dissertation for resolving the problem are as follow:

- **A Transactional Memory System-on-a-Chip (TMSoC) architecture** that is based on a deterministic Network-on-Chip (NoC). The TMSoC utilizes a transactional memory to provide segregation and fault containment at the different levels of the chip-level architecture. The architecture consists of a configurable number of cores and a memory gateway that exploits the Mixed-Criticality Transaction Controller (MTC) extension for mixed-criticality systems support.

- **The Mixed-Criticality Transaction Controller (MTC) extension** introduces criticality-aware algorithms that are responsible for executing so-called selective rollbacks of transactions based on their criticality.

- **A hierarchical distributed architecture** extends the TMSoC chip-level architecture in order to provide segregation and predictability at cluster-level. The architecture uses a transactional-memory based protocol that is spanning both on-chip and off-chip networks handling transactions based on their criticality.

- **The Distributed Mixed-criticality Transactional Controller (DMTC)** is deployed as a transactional-memory protocol at on-chip and off-chip network levels of the

distributed architecture. This protocol hides the diversity of the components in the system ,and guarantees the criticality-aware execution of the application subsystems by executing selective rollbacks when required.

- **Simulation frameworks**: A fully configurable and extendable simulation framework supporting message-based and shared memory interactions has been implemented. The framework provides a trace-based simulation environment for chip-level mixed-criticality systems. Additionally, it has been extended to support distributed mixed-criticality simulations.

- **Evaluation and experiments**. Automotive use cases have been used to evaluate the architecture models and algorithms. The simulation framework was used to perform the experiments, which have shown that the proposed architectures and their algorithms meet the requirements including temporal predictability and fault isolation.

1.2 Thesis Organization

The remainder of the thesis is structured as follows.

- Chapter 2 introduces the basic concepts and provides the background knowledge used in this work. It starts with a classification of real time embedded. Afterwords, dependability attributes, and the notion of faults and fault tolerance are discussed. Next, a comparison between shared memory and message-based communication paradigms is given. The control signals for triggering interactions in these paradigms are also analyzed. The chapter ends with a presentation of the available memory technologies including memory controllers, and the notion and types of transactional memories.

- Chapter 3 provide an analysis of the state-of-the-art. It starts with the requirements for mixed-criticality systems and transactional memories. The next two sections investigates existing architectural solutions at both chip and cluster levels. The presented solutions are analyzed toward the fulfillment of the requirements. Based on this analysis, the research gaps in the state-of-the-art are highlighted to be addressed in this dissertation.

- Chapter 4 presents an architectural solution for MPSoCs that is based on a deterministic NoC and a transactional memory, known as the Transactional Memory System-on-a-Chip (TMSoC). This architecture uses the newly introduced Mixed-Criticality Transaction Controller (MTC) algorithms to guarantee the predictability and fault isolation

in mixed-criticality systems. In the second part of this chapter, a hierarchical transactional memory architecture for distributed mixed-criticality systems is defined. The introduced hierarchical architecture uses the so-called Distributed Mixed-criticality Transactional Controller (DMTC) protocol that hides the heterogeneity of the distributed components and provides a reliable memory protocol for mixed-criticality systems.

- Chapter 5 offers a trace-based simulation framework for shared memory and message-based communication in multi-core chips. The framework is dedicated to mixed-criticality system simulations based on the proposed TMSoC. It provides high abstraction levels using SystemC/TLM and it exploits DRAMSim2 for simulating the external memory of the TMSoC.

- Chapter 6 extends the earlier described framework to support the simulation of the hierarchical transactional memory architecture. This is achieved by implementing the DMTC protocol and integrating a co-simulation mechanism between the SystemC/TLM multi-core simulation and the cluster level communication. The introduced co-simulation of the hierarchical framework guarantees the time integrity of the different instances with the off-chip communication simulated in VEOS/AUTOSAR.

- Chapter 7 describes three automotive use cases for the evaluation of the simulation framework and the proposed TMSoC and hierarchical transactional memory architectures. The simulation results of the use cases are presented and discussed at the end of each evaluation scenario.

- Chapter 8 concludes the thesis through a discussion of the overall results of the presented solutions.

Chapter 2

Background and Basic Concepts

The basic concepts and the required background understandings are presented in this Chapter. Starting with the notion of real-time system, and its different classifications. Then, the behavioral model of the distributed real-time system is depict. Introducing later the basic concepts of timing properties and measurements in real-time system.

Afterword, the dependability attributes and threats are illustrated to define what is a dependable system. In the next subsection, the difference between message-based and shared memory communication is presented. These paradigms of communication are also analyzed in regard to the triggering control signals in the system. Finally, we exhibit the different types of communication messages.

In the last subsections, memory technologies and hierarchy is interpreted covering the basics of this large theme. The concept of transactional memory, its architecture and types are explained at the end of this Chapter to wrap-up all required knowledge for this work.

2.1 Real-time Embedded Systems

A set of dependent components with distinct timing and spatial limits that are forming an entity is defined as a *system*. This includes the description of the overall system's functionalities, behavior and the interacting means with its environment represented in the inputs/outputs of the system. The system user can be a human operator or another computer system. The service delivered by a system is its behavior perceived by the user.

Several definitions for embedded systems are given in literature. An *embedded system* as defined by Steve Heath [Hea02], is a microprocessor-based system which is built to accomplish a function or a sort of functions and is not designed to be programmed by the end-user in the same manner that a personal computer is. Fundamentally, an *embedded system* comprises one or more processors, a set of memory components and a number of

peripherals to the environment. A processor consists of an instruction control unit and an arithmetic unit, which carry out instructions of a computer program by performing the basic arithmetical, logical and input/output operations of a system [Sta96].

Memory is an essential part of an embedded system. Today's systems involve different types of memories in addition to memory management and memory protection units that provide translation interfaces, partitioning and access permissions. In order to deal with the increasing complexity and performance demands of embedded systems, caches are used to provide direct and fast access to data in order to increase the performance and reduce memory access delays. Moreover, non-volatile memories (i.e. ROM memory) are used to store program code and configuration information, since even when power is removed this memory retains its content. Another type of memory that is used in embedded systems is volatile memories (i.e. RAM memory) to provide interim storage to the applications data variables that are used or computed during the execution [LS11].

Interactions and data exchange between embedded systems and their external environment is achieved through the peripherals. Dedicated operations at the *Input/Output (I/O)* ports are executed to collect data (e.g. temperature and pressure sensors), and transmit actions and signals to the system's actuators and other output devices.

Bell Telephone Laboratories and the Massachusetts Institute of Technology (MIT) initially raised the term of *real-time systems* during the Second World War, which was obviously required due to the need to improve the computational and physical time correctness of the systems. The first high speed digital computer that was able to operate in real-time was introduced by the Whirlwind project [RS80]. IBM developed a flight simulator for the American army in 1947 which was the largest computer project until the early 1950s.

The first use of the term real-time in non-military domains was in the context of real-time operating systems for airline reservation systems. SABRE was one of the first reservation systems, it was introduced for American Airlines in the year 1964. This led to the purposeful involvement of real-time software in process control for spacecraft-control and space-telemetry by the American national space programs [LO11]. During the 60s and 70s the progress at the integration levels and processing speed resulted in the enhancement of the real-time process control systems.

Moreover, a definition based on the German industry standard DIN 44300 for the real-time operations presented in Kavi [KS92] states that a *real-time operation* is an operation that meets the following conditions: all data inputs arriving to the operation shall be available any time to be processed and the operation provides its result within a given period of time. The arrival time of the data can be randomly distributed or can be already a priori determined depending on the type of the applications. For instance, in the control loop of an Anti-lock

braking system (ABS), the vehicle speed is continuously monitored and the breaking systems has to react according to the collected data. In case of skidding, the system is required to react within bounded time to avoid an accident.

Nowadays, demands for high performance in systems such as flight control, power plant control, autonomous driving and automatic train control have resulted in increasing reliability requirements not only in the value domain but also in the time domain. Such real-time systems involve more and more embedded systems with dedicated functions, control services and strict real-time constrains. Based on that, a *real-time embedded system* is defined as a system in which the correctness of its behavior depends on the computational results as well as the physical time that these results are produced with respect to the global time base [Kop13].

Real-time systems are mostly part of larger systems. Self-contained real-time subsystems that change as a function of physical time, inputs and states are called real-time clusters or Cyber-Physical Systems (CPSs).

Understanding CPS can lead us to a better realization of real-time systems. As illustrated in Figure 2.1, such systems consist of two spaces that are interacting through a network or a cloud, which depends on the scale of the system. The physical space can be any real-life space such as a vehicle, a power plant, a hospital, a transportation network, etc. Furthermore, a cyber space is an artificial space that processes the collected physical inputs using the required computational resources. The cyber space includes processing units, I/Os, computer networks, etc. The resulting outputs of the cyber space are activation decisions and signals for the physical space. The resulting outputs shall lead to the correct behavior of the CPS with respect to its time restrictions on a global time base.

In order to take the desired decisions at any CPS, the decision making process is divided into two phases (cf. Figure 2.2). A cyber phase involves the artificial factors, and a physical phase depends on direct involvement of the human factors. The cyber space can employ artificial intelligence that is responsible for executing designated algorithms for maximizing the chances getting the desired decisions. These algorithms use the data collected from both physical and cyber (digitized) worlds for learning and self-improving their performance. Humans are able to make heuristic judgments and take decisions, thus human intelligence is an essential part of any CPS. These collaboration between artificial and human intelligence is essential for the correctness of the CPS. Finally, simulating the physical world can reduce the costs, efforts and time of developing CPSs. Assistant training from virtual reality, for instance, is the base training of every future pilot. A recent study on CPSs, includes military, aerospace, communication and automotive sectors has shown the increasing usage of Field-

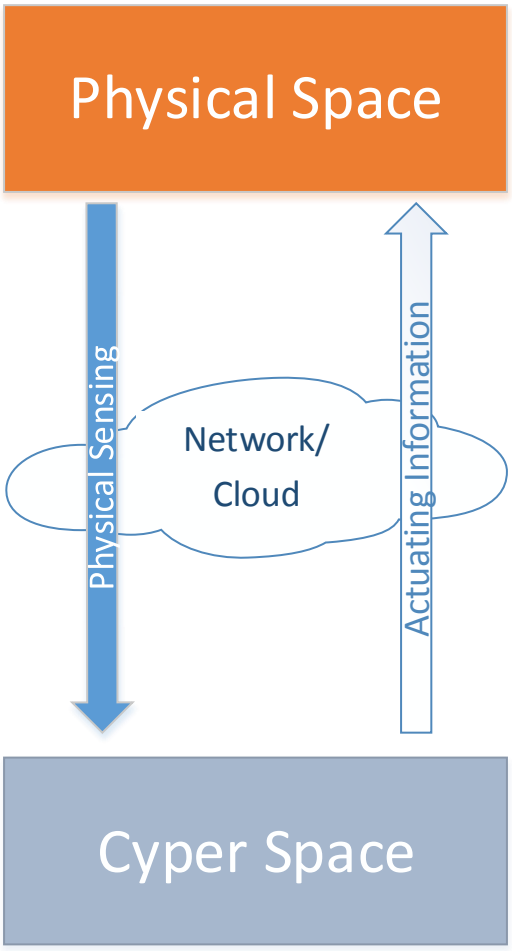

Fig. 2.1 Cyber-Physical System

Programmable Gate Array (FPGA) in simulating embedded systems. Moreover, the FPGA market value in these sectors is growing over 9% from 2014 to 2020 [Inc15].

2.1.1 Classification of Real-time Systems

Real-time systems can be classified from different perspectives. Classifications that are relevant for this thesis are presented in this subsection.

A real-time system must execute its services based on the collected inputs and calculate its outputs within bounded time intervals. The time instant at which an output must be produced by the system is called *deadline*. Real-time systems with at least one hard deadline that has to be met are called *hard real-time systems* or *safety-critical systems* [Kop97]. The

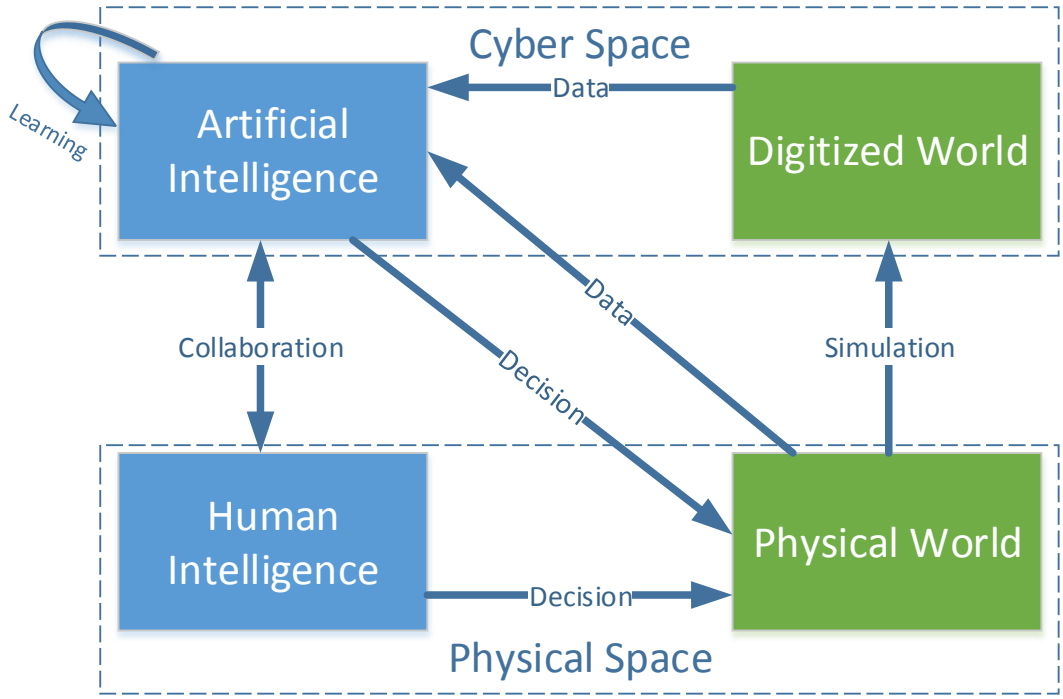

Fig. 2.2 Decision making in cyber-physical systems

design of hard real-time systems requires guaranteed temporal behavior under specified load and fault restrictions. On the other hand, in case no hard deadlines are declared, the system is called *soft real-time system* and deadlines are allowed to be missed occasionally. *Firm real-time systems* are systems where deadline misses are tolerable but this may reduce the system's utility.

In case of hard real-time systems, the ability of reacting to a failure determines whether the system can be called fail-safe or fail-operational. If the system can reach a safe state quickly in case of such a failure then it is called *fail-safe* system. For instance, a traffic light at a busy cross road can be set to red in case of a failure in the traffic control system. On the other hand, systems that have to stay operational regardless of possible failures are *fail-operational* systems, e.g., the flight management system of an airplane.

Another classification of real-time system is the classification with respect to the type of the temporal control signals of the system. A *trigger* in a real-time system is a stimulus that causes the start of an action. Triggers can be initiated in one of the following two ways, using event-triggered or time-triggered control signals.

Event-triggered control signals are not necessarily bound to timing events. Significant events can be any relevant state changes e.g., receiving a specified message. An event-

triggered system handles these events based on interrupts to serve the event accordingly. On the other hand, all communication and processing activities in *time-triggered* systems are initiated periodically based on a priori known clock ticks. Clocks of the distributed time-triggered system are synchronized based on a global time base, as will be explained in section 2.3.

2.1.2 Distributed Real-time System Model

The notion of real-time can be extended to the levels of operating systems, communication networks and scheduling. In this dissertation we are mainly focusing on the real-time communication and its design decisions.

Distributed real-time systems are composed of a set of computational nodes that are connected though a real-time communication network to control the functions of a controlled object. This model structure requires the definition of two interfaces, the so-called man-machine interface (e.g., keyboard, touch screen) between the operator and the real-time computer system. Secondly, the instrumentation interface is located between the controlled object and the real-time computer system. This interface converts physical signals of the sensors and actuators to digital forms in both directions.

The sequence of the results over time is called the distributed real-time system's *behavior*. Based on that, a *behavioral model* defines the behavior of a real-time system. A behavioral model can be defined using components, states and messages.

A *component* is a unit built on hardware and executes a software application that performs well-defined function within a larger system. The behavior of the component is defined in terms of interfaces from and to the component. The behavioral model contains a composition of computational components that implement specified functions, exchange messages and alternate between the system's states to produce results. During a component execution, all past information that can be accumulated at a given instance is called *state*, and this information is directly relevant to upcoming operations of the component. Furthermore, the change of the state is called an *event* [OK05].

A *message* in the behavioral model is a data structure that contains a number of data fields with dedicated names. These fields comprise semantics of the message such as source, destination, priority and payload. Moreover, data exchange via shared memory is also a means of communication.

Interfaces are needed in large system in order to provide functional and temporal abstractions between the different components of the distributed system. There are many types of interfaces depending on the purpose of the interface, i.e., Technology Independent Control

Interfaces (TIIs), Technology Dependent Debug Interfaces (TDIs), linking interfaces and local interfaces [Kop13].

Linking Interfaces (LIF) entail operational and meta-level specifications of the inter-components communication. This includes the structure of the exchanged messages, and the temporal details for the messages. This represents the operational specification of the syntactic and temporal properties accordingly. The semantics of the message are described based on the meta-level specification of the interface. A component can provide LIF services and it can request linking services provided by other components. An example of such interface is the component interface to the communication network.

Local interfaces are responsible for providing the link to the external environment. However, the semantic content relevant for this interface is specified in the LIF.

Components that link the internal world of a cluster with the external environment are called *gateway* components. These components have two interfaces, a LIF that provides the link to the cluster's communication networks and a local interface that provides the link to the external environment. Network and memory gateways are good examples of gateway components.

2.1.3 Concept of Timing in Real-time Systems

The *granularity* of a time base is the duration between two consecutive micro-ticks in a digital physical clock.

Synchronization of the system activities requires the preservation of the time information related to the occurrence of an event or system activity. This can be established by attaching a dedicated time stamp to each of the activities. Time stamps are called *absolute* in case they are synchronized to a reference clock. A *reference clock* is an external clock that is detecting and observing all activities of the system as a timekeeper.

Assuming that we have a distributed system that consists of multiple nodes. Each node has its now local physical clock and all local clocks have the same granularity, and are synchronized with the same precision. The time *precision* is the maximum offset between the micro-ticks of two different clocks. As a result, a selected set of micro-ticks for each local clock can be considered as global ticks of the system's global time.

Clock synchronization is done at two levels, internally and externally. *Internal* synchronization ensures that the occurrence of all global ticks is always within the precision defined for the system. This is done by synchronization messages between the different nodes. Synchronization messages might be subject to a so-called malicious or Byzantine error, where such errors can easily result in inconsistency of the subsystem clocks. To resolve this

problem, either interactive-consistency algorithms or inconsistent algorithms with bounded effects of malicious clocks (e.g., fault-tolerant-average algorithm [LMS85]) are used.

External clock synchronization is responsible for aligning the global time ticks of the system to an external time standard, e.g. global positioning system. Time synchronization in this case is done through time gateways that are also responsible for the timely correct initialization of the system, as well as time-format mapping from one time standard to another one. Finally, the local time gateway is responsible for communicating and synchronizing with the other time gateways and executing time re-initialization when needed.

As discussed earlier, distributed real-time systems with time-triggered control have to be synchronized to establish a global time base for all time-triggered subsystems. Without clock synchronization, internal clocks of a set of subsystems may differ even if they have initially started accurately due to clock drifts [TS01]. Therefore, clock synchronization in distributed embedded systems is very essential. There are a number of existing solutions for clock synchronization, e.g., precision time protocol, global positioning system, Network Time Protocol, IEEE1588. Network Time Protocol (NTP) is a frequently used clock synchronization solution between computer systems. In previous work of the author, a distributed system based on time-triggered control was presented where NTP was used to provide clock synchronization [OAOD14].

Timing Properties and Time Analysis

It is essential that all time constrains and synchronization requirements of real-time embedded systems are met. Hence, timing models and analysis techniques are required to calculate and ensure the achievement of these time requirements. In the following, a number of timing properties that are used for embedded systems are discussed.

The *execution time* is the time required for a task or action within a component to finish after it has started [Zur09]. This measure cannot easily be derived as it inherently depends on the hardware and the execution flow of the system components and services. Execution time estimation provides more flexibility for system analysis. The WCET for instance, is a way to bound the execution time of a task or action. WCET approximation methods are important to deal with the unpredictability of present-day computing platforms (e.g., speculative execution, caches). [WEE+08].

The *response time* is the time calculated from the difference between the task completion time and its release time. The release time is the time when the task becomes ready for execution [But11]. The task response time must be bounded especially in hard real-time tasks. For instance, the required response time in control loops is often in the order of milliseconds or even less. System components involved in the time analysis might change

based on the dependencies and interference within on the examined subsystem. For instance, the calculation of the response time in a system might include the network between the components. This depends on whether the network itself is considered autonomous or as part of the components.

The *end-to-end delay* at communication levels of an embedded system, is the time property that relates the occurrence of an event in time to the occurrence of another event. As an illustration, this delay in message-based systems is calculated from the time that a message requires to be transmitted from a source component to the delivery at its destination. This transmission could possibly include multiple networks such as on-chip and off-chip network interconnects [OOA+14b].

2.2 Dependability

There is a precise and rigorous terminology used in the literature to describe the basic concepts of dependable computing codified by Laprie [cLR01]. Based on that, the *dependability* of a computing system is its ability to deliver services that can justifiably be trusted. Additionally, dependability is defined by the *IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance* as "the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers". Alternatively, it was defined in *IEC IEV 191-02-03* as follows: "dependability is the collective term used to describe the availability performance and its influencing factors: reliability performance, maintainability performance and maintenance support performance".

2.2.1 Dependability Attributes

In the following subsection a number of key dependability attributes that have to be taken into consideration to deliver a dependable system are presented.

Reliability

Reliability in embedded systems is defined as the continuity of the services. In other words, reliability is the ability of the systems to provide failure-free operations for a specified period of time in a specified environment. Based on this definition, a reliable service shall not fail for a given period of time, or it shall be able to successfully recover to a safe-state in case of a failure. This safe-state is a state that can be quickly identified in *fail-safe* real-time systems in case of a failure. Highly reliable embedded system services in safety-critical systems are demanded to exhibit failure rates are in the order of 10^{-9} failures/hour [LHSC10].

Availability

Availability is the probability of the system to be ready when needed, i.e., it is the system readiness to provide sufficient quality of service. Reliability and availability are directly connected. Therefore, the following reliability parameters are used in order to calculate availability.

$$Availability = MTBF/(MTBF + MTTR) \tag{2.1}$$

Availability is the rate of the estimated average time between failure of system components (known as Mean Time Between Failures (MTBF)) divided by the sum of the MTBF added to the average time needed to repair the failed system module (known as Mean Time To Repair (MTTR) [LS11]).

Safety

Safety is the avoidance of catastrophic consequences on the environment by avoiding critical failure occurrence. Such failures impose significantly higher cost than the utility of the real-time system. As a result, safety is a characteristic of the whole system and not only of a single component [Sto96]. For instance, one traffic light at a road junction can be set to "red" in case of a failure of the system, but this is not enough to avoid possible collisions. All traffic lights at the junction shall stay at "red". In order to design a safe system, safety shall be considered in the context of the system and not only at the component or the software levels. It is important to mention that the previous example can be called a safe system, but it is not necessarily a reliable one.

Different industry segments (e.g., automotive, avionic, medical) are demanding systems with specific safety features. Despite the size of investments in this area, designing truly safe systems is sill highly challenging. Modern cars contain a big number of embedded computing systems that are connected hierarchically. For example, these embedded systems exchange highly important inter-process messages relevant for advanced driver assistance systems (ADAS) in parallel with higher level infotainment and navigation messages that are of lower importance. Such a system, also known as a MCS, is required to react safely in case of faults.

MCSs consist of a number of subsystems that are sharing their communication and computational resources, where these subsystems have two or more different criticality levels. These levels are defined based on certification standards. The certification process is based on evidences that are established by activities such as documentation, reviews, audits or

testing. This includes the identification of the possible hazards, determine the risks and then defining the safety measures.

Safety standards have been evolving over the years, DINVVDE 0801, DINV 19250, EUROCAE-ED-12B, EN 954 and ANSI/ISA S84.01 are examples of standardization attempts for safety standards relating to computers in control [Col08]. Nowadays, IEC 61508 is a cross-domain functional safety reference that is widely used in the industry. Potentially, this standard can be extended to any other industrial domains, e.g., medical electrical equipment, trains, wind power. For instance, it has been extended to meet the functional safety requirement of the automotive domain (i.e., ISO 26262). The IEC 61508 standard requires risk assessment and hazard avoidance to be carried out in the whole system. This requires the definition of the so-called Safety Integrity Levels (SILs) of the system, which are the criticality levels of the system based on that standard. The demanded reliability in these four SILs in case of IEC 61508 can vary from 10^{-2} to 10^{-1} at low demand, up to 10^{-9} to 10^{-8} at high demand mode [Bel06].

Security

The last attribute of dependability that we discuss is *security*. It is the ability of the system to prevent unapproved access to data or services including the reduction of vulnerabilities and protection against threats, attacks, interference and espionage [Gas88] [WLSC07]. Security attacks aim to obtain access to hardware, software, information and possibly to modify specific service functionality. *Threats* exist vertically at all levels of the system from hardware/physical level up to the operating system and application level. *Attacks* that results in these threats are usually categorized into the following main categories: backdoor, denial-of-service, direct-access, eavesdropping, spoofing, tampering, privilege escalation, phishing and click-jacking attacks. Attackers might attempt their attack on the computer systems or even on the users as in social engineering attacks.

The security issue of computer systems has been keeping engineers and researchers busy since the beginning of the digital system evolution. Therefore, a number of defense strategies have been developed, e.g., access control systems and application security using antivirus software. Despite the long list of defense options to obtain better secured systems, security remains a prevalent challenge in present day embedded systems. An example for the vulnerability of automotive systems was demonstrated by the automotive security researchers Charlie Miller and Chris Valasek. They hacked a Chrysler Jeep using a “wireless attack” at the Black Hat 2015 conference in Las Vegas. This hack has caused the recall of 1.4 million vehicles by the car manufacturer [Yos16].

2.2.2 Faults and Fault Tolerance

Dependability and security threats are linked as described by Avizienis and Laprie [ALRL04]. They are divided into faults, errors and failures.

The *function* of a system is what the system is intended to do, as described by its functional specification. Faults occur often in systems such as a frozen memory bit, an uninitialized variable in software, or a cosmic ray ionizing its way through an embedded system. The ensuing failures indicate the disagreement between the defined service and the actual behavior in the system. An *error* is a system state that may lead to a failure. An error is detected if an error message or signal is produced within the system or latent if not detected. A *fault* is the cause of an error, and it is active when it results in an error, otherwise it is dormant. A system *failure* occurs when the delivered service does not comply with the specification.

The cause of faults is vary diverse, hence they are classified into three major classes. Firstly, *design faults* can occur at both hardware and software levels. It can be due to software flaws or due to hardware errata. Moreover, the erroneous logic can appear upon the integration of both hardware and software of the system. The second class of faults are the *physical faults*. This type of faults is non-human made, which means that they exist due to product defects or physical damage of the system. Finally, malicious attacks or faulty inputs in addition to possible physical interference can result in *interaction faults*.

Fault-tolerance is the ability of a system to preserve its ability to deliver a correct service in the presence of faults [ALRL04]. The ability to detect and process error states and assess the consequences is essential for having a fault-tolerant design. This is achieved by removing system error states, and then treating the source of the fault. Our goal as engineers is to prevent the occurrence of system failures in our embedded systems, which requires a well defined process. Therefore, quality control techniques at both software and hardware levels shall be applied. Each class of faults has its own techniques to prevent failures and acquire its means to achieve the dependability of the system. Model-based and object-oriented design for instance are introduced to prevent software design faults and consequently failures. In distributed systems failures caused by such faults can propagate from a system to another one through a message failure in the time or value domain. Therefore, proper design decisions have to avoid this propagation, using proper network interfaces as in [OKSH07]. Moreover, radiation hardening and foolproof packages are used to mitigate physical faults and interaction faults [cLR01].

Error containment is the ability of the system to handle errors. This can be performed in two phases, error handling and fault handling. Identifying and saving specific safe states (so-called checkpoints) of the system can help to rollback to points in time where the system

was free of that error. Such an *error handling* technique for eliminating errors is used in distributed shared memory systems [JF95]. What is important in *fault handling* is to identify the cause of the error and isolate the faulty component of the system, which can be a software component or a hardware component.

Software and hardware fault tolerance is achieved by using redundancy [KK07]. Hardware redundancy techniques often make use of multiple identical components or subsystems, in addition to means for arbitrating the resulting output, e.g., majority voting. ECC memory, for example, uses a few extra bits to detect and correct errors resulting from faults in the individual storage bits. Running the same input data through a faulty software module multiple times yields the same erroneous result each time. Redundancy can be done by 1-to-1 redundancy or up to N redundant instances. However, Triple Modular Redundancy (TMR) is commonly used in both software and hardware fault tolerance.

Software fault tolerance is built by applying algorithmic diversity, computing results through independent paths, and by judging the results. This adds complexity to the system in general. Adding software fault tolerance will improve system reliability only if the gains made by the added redundancy are not offset by commensurate new faults introduced by the redundant parallel code. In case of N-modular redundancy in hardware fault tolerance, the cost of the system health monitoring for all instances is very high. Therefore, a hierarchical structure of health monitoring units can be applied.

Based on Kopetz [Kop06], the following instructions have to be followed in order to establish a fault-tolerant system. Initially, assumptions must be specified for the types and numbers of faults that will be tolerated in the designed system. This is called the phase of the fault hypothesis. The definition of specific Fault Containment Regions (FCRs) is part of this phase. An FCR is the set of components or subsystems that shares one or more common resources that can be affected by a single fault and is assumed to fail independently from other FCRs of the system. Moreover, it is mandatory to identify specified failure modes for each FCR at this phase. Each single FCR in the designed embedded system can fail in a specific failure mode, e.g., fail-stop, crash, omission, timing, byzantine or babbling idiot [Kop05]. The design phase is the second phase of the process, taking into account the previous assumptions. Finally, the designed architecture can be implemented and validated using the selected fault-tolerance mechanisms and the fault hypothesis that was assumed.

2.3 Paradigms of Communication

2.3.1 Shared Memory and Messages-based Communication

The increasing number of cores integrated on a single chip has introduced the need for efficient inter-core communication means. These means are used to exchange data and provide access to the required memory resources. Cores might share specific data blocks with other components, where this data shall be exchanged concurrently.

In multi-core embedded systems, memory data is usually shared and distributed between local caches at the cores and external memories. Each core will have its corresponding memory segments (locally or remotely). Moreover, data can be exchanged between the cores though a shared memory by employing overlapping memory segments. Shared memory-based communication requires the support of a hierarchical memory architecture and concurrency algorithms (i.e., cache coherency protocols, cf. Section 2.4.1). This type of communication is more expensive from the overall execution time point of view, and has a high hardware complexity.

In case of message-based communication, a message structure is defined to wrap all needed information for data exchange between the cores. Message-based communication ease of use for application developers, it involves better utilization from a bandwidth point of view, and it typically provides better spacial isolation between the components. It can also be easily managed to provide better timing isolation in combination with a TDMA communication scheme, as will be explained later.

These two forms of communication can be combined to benefit from both paradigms as described in this dissertation.

2.3.2 Event Triggered vs. Time Triggered Communication

An overview of the different types of control paradigms is discussed in this subsection. Control signals in embedded systems are initiated when an action occurs, such as, a transmission of a message, the start of a service, or a memory request. As discussed in previous sections, embedded systems can be classified based on their control signals into event triggered and time triggered systems. This differentiation depends on the source of the triggers.

In case control signals are generated based on events then we are talking about *event triggered communication*, where every change of the system state is considered as an event [Kop13]. This event might be generated internally within the real-time system, or it might be originated from one of the controlled objects of the real-time system. A good example of event triggered communication is the CAN bus that is widely used in the chassis

control systems and power train communication [Amo04]. An event message is a message that contains event observations of the difference between the last observed state and the new state. The notion of time in such messages denotes the point of time at which the state has changed. Communication guarantees in event triggered real-time systems are provided by using control flow protocols with some sort of acknowledgment. Positive Acknowledgment with Re-transmission (PAR) for instance, is one of the control flow protocols using positive acknowledgment for successful reception of the message. The usage of such protocols provides resilience against transient failures in event triggered systems. The basic idea of PAR is very simple, each sender has a predefined time-out to receive the reception acknowledgment from the receiver. In case, the time-out is over without the reception of the acknowledgment, the sender attempt to re-transmit the message considering that some failure occurred in the reception of the previous attempt. Moreover, explicit control flow can be used to solve timing failures, where the receiver can send a control signal to inform the sender about its desire to send another message [OK05]. Generally, PAR based communication protocols (e.g., TCP/IP) are not considered as suitable for safety-critical systems.

In case of time triggered communication, the communication activities are periodically triggered and synchronized based on a global time base. If multiple components in a real-time system are observing each other, then the observed events shall have the same temporal order in all components. Therefore, all control signals are generated in restricted and particular points in time in relation to the global time based on the a priori knowledge of the real-time system. The task period, the phase, the sender component and receivers are examples of the a priori knowledge. Time triggered communication provides deterministic behavior and guarantees high temporal predictability and composability of the real-time system.

Event-triggered protocols such as Ethernet and CAN do not provide sufficient guarantees with respect to timeliness and reliability in safety-critical systems. However, a communication control strategy is needed to provide temporal control and avoid conflicts between messages. As explained by Lee in [LLS07], safety-critical systems typically use static TDMA-based protocols, e.g., TTP/C or FlexRay. TDMA defines the time-based communication scheme that is used by the communication controller of each component to disseminate the messages. The communication scheme is defined by dividing the capacity of the communication channel is statically divided into a number of slots. Each slot is dedicated for a certain component of the real-time system. Each component has it own sequence of sending slots in the so-called TDMA round. The completion of the TDMA rounds of all components denotes the cluster/system cycle that is periodically repeated. The resulting static schedule defines the communication between all components of the system with respect to the global time. Additionally, a priori knowledge of the system (e.g., the number of components, sending time,

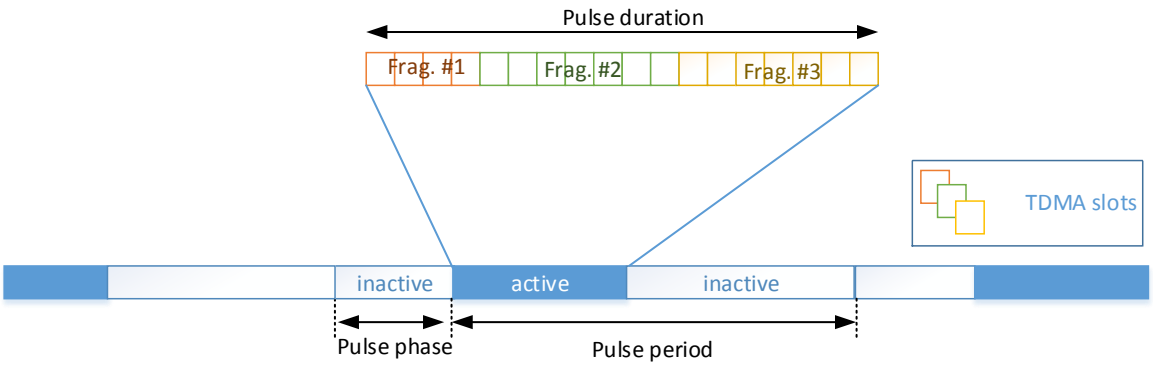

Fig. 2.3 TDMA-Pulsed Data Stream [OESHK08]

receiving time, etc) is required. Finally, message handling is done at design time without any need for explicit control flow as in event triggered communication systems.

In Time-Triggered Ethernet (TTE) [Obe11], frames are based on Ethernet as standardized in IEEE 802.3. To start the communication, Pulsed Data Streams (PDSs) specify the carrying signals and their clock pulses. A PDS is a time triggered, periodic, uni-directional data stream, which is identified by periodic pulses with defined pulse period and defined pulse phase. As illustrated in Figure 2.3, at least one fragment of variable size construct the PDS and the fragments of a PDS do not require to be transmitted without interruption. The time between the start of the transmission of the first fragment and the end of the transmission of the last fragment is known as the duration of the pulse. The phase of the pulse is the offset until the transmission is started. Each fragment is composed of a set of fixed-size flits.

2.3.3 Types of Communication Messages

Communication activities in real-time systems are initiated at specific points of time (cf. Section 2.1.3). Moreover, three message types can be distinguished in real-time systems: periodic, aperiodic and sporadic messages. Determinism and real-time support in time triggered communication requires *periodic message* exchanges, where messages contain the absolute real-time value. These periodic messages are transmitted at predefined points in time, with a specified period and phase, according to a communication schedule. The communication of periodic messages can be planned at development time in order to minimize latency and jitter. Finally, message queuing is not required at the communication interfaces in time triggered communication, since real-time applications are interested only in the most recent value of the observed object.

Messages with unspecified bit rate and delivery timing constrains are called *aperiodic messages*. This type of messages is typically served based on the current traffic load of the network. Namely, these messages use the remaining bandwidth of the network to be

transmitted. Hence, there are no guarantees for the transmission or receiving of the message. Additionally, possible delays or even message loss might occur.

Sporadic messages (a.k.a., rate-constrained messages) are messages with bounded minimum interarrival times. The total allocated bandwidth is calculated from the bandwidth allocation gap (BAG), and shall not exceed the maximum available bandwidth. Although rate-constrained messages are suitable for real-time applications (e.g., ARINC-664P7/AFDX [TPS15]), sporadic messages are determined for less stringent communication constrains than the periodic messages.

2.4 Memory Technologies and Hierarchy

Memories are one of the main components comprised in all embedded systems. Nowadays, computational components are becoming faster and more numerous. Moreover, the performance and size of memory is growing due to the low cost. The real limitation factor in embedded systems is memory efficiency and memory access. Existing memory handling approaches, both in software and hardware, promise to provide higher efficiency for memory architectures in embedded systems. Figure 2.4 shows a typical memory hierarchy in a single processor core. It is important to mention that the cost for data fetching is increasing dramatically as much as location of the required data away from the requesting processor. This expansion can be noticed in larger scale memory architectures in distributed systems. Cross-cutting system-level issues e.g., performance and reliability [JNW07] become much more complex to manage and handle at that level.

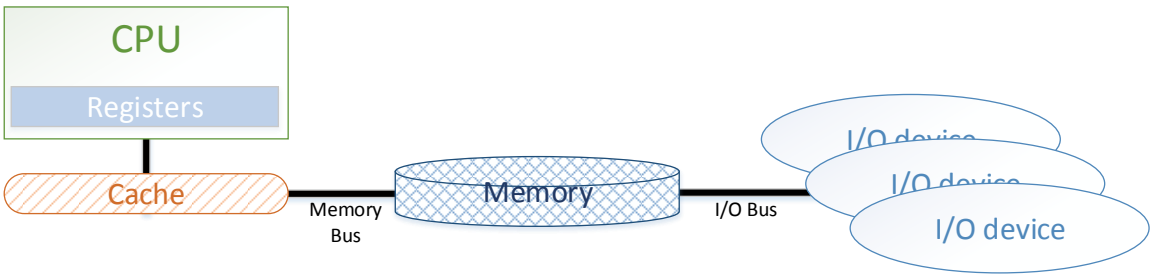

Fig. 2.4 Typical memoy hierarchy in embedded systems

The following characteristics are used to classify memories. The *capacity*, i.e., the global volume of information that the memory is able to store can be in the order of gigabytes or even terabytes (e.g., the world's largest hard drive is 16 TB [Mos16]). Secondly, the *access time* corresponds to the time interval between the read/write request and the availability of the data. Third, the *memory cycle* time represents the minimum time interval between two

successive accesses. Fourth, the *throughput* defines the volume of information exchanged per unit of time, expressed in bits per second. Finally, memories are characterized based on their ability to retrieve the stored digital data after being switched off and on again. Magnetic tapes, mechanical drives, solid-state drives, flash memories and different forms of Read-only Memory (ROM) belong to the *non-volatile memories*. This type of memories is able to store information over long term, and stored data is not lost after power down. The second type of memories is called *volatile memory*, where stored data will be irremediably erased in case of power down. Volatile memories such as general purpose RAM including dynamic and static RAMs are directly accessible to the computational cores (CPUs) using the memory bus. This bus comprises an address bus and a data bus. In case that a memory instruction is triggered, the desired memory address is sent through the address bus, then the data for the read (or write) request is sent through the data bus.

The previously mentioned memory types are known as physical memory [LS11]. Another form of memory technologies that is used in embedded system is the so-called *virtual memory*. It is a memory management technique that can combine the usage of both volatile and non-volatile memories of a system. Virtual memory provides an adjustable abstraction layer by assigning the physical memory address space of the system to virtual addresses. An additional translation layer between the computational core (CPU) and the system's memory is needed to reallocate the virtual memory addresses to physical addresses again. This is done by the Memory Management Unit (MMU). The MMU improves its capability in address translation speed with the help of a Translation Lookaside Buffer (TLB).

2.4.1 RAM Technologies

In the dissertation, we will focus on the analysis of RAM-based storage in this sub-section, namely dynamic RAM and cache memories. Most embedded systems contain an amount of RAM-based memory storage to provide relatively quick access to the main memory of the system. Such types of memories can be either Dynamic Random-access Memory (DRAM) or static Static Random-access Memory (SRAM). Both volatile memories are useful in embedded systems, SRAM is able to maintain its data better than DRAM. This problem appears due to the way DRAM operates, where each memory location must be periodically refreshed. Otherwise, data might get lost even while the power is maintained. This refresh cycle might cause delays for the memory access. Another key factor in accessing a required memory address is the last accessed memory address. To overcome this problem, DRAM are used in combination with a *memory controller* that is responsible for handling memory requests in case the memory is busy refreshing memory locations. Despite that SRAM consumes more power and is expensive in comparison to DRAM, there is no embedded

system that does not include one. However, memory requirements, especially in complex embedded system applications, cannot be fulfilled only with SRAM. Therefore, DRAM is used to in conjunction with SRAM. It is important to mention that embedded system developers have to be aware of the system's memory address map. This map defines how addresses are mapped to the hardware (e.g., an address can be mapped to SRAM or DRAM or even both).

DRAM technology has been evolving fast in the last decades. Several improvements have been made in regard to capacity, performance, integrity (e.g. Error-Correcting Code (ECC) memory [HP03]) and security. But most importantly, memory operations in typical asynchronous DRAM were managed by an external *clock signal*. This restriction limited the data transfer rates and the concurrency performance of the memory. Pipelining memory operations was the solution by introducing synchronous DRAM (known as Synchronous Dynamic Random-access Memory (SDRAM)) thereby increasing efficiency and throughput. The memory is divided into independent banks allowing the memory to process multiple memory operations at a time without waiting [Cra96].

Currently, many embedded platforms use Double Data Rate (DDR) SDRAM that transfers data elements on both rising and falling edges of the clock effectively doubling the bandwidth over its predecessors. Moreover, special propose SDRAM was firstly introduced by ATI technologies. Graphics DDR for instance, is a special SDRAM used for graphics processing units (GPUs) with increased throughput and more precise timings. Next generation graphics cards will be using the newly standardized GDDR5X specification for their Synchronous Graphics Random Access Memory (SGRAM). This standard was certified at the beginning of 2016 by JEDEC, a major semiconductor engineering trade organization for memory standardization [Des16].

The structure of an SDRAM consists of banks, rows, and columns. Figure 2.5 illustrates a simplified abstraction of the building blocks constructing a four bank SDRAM. Banks are essentially independent memory blocks, but with shared data, command, and address buses to reduce the number of off-chip pins. This can be achieved using different control mechanisms and decoders of the SDRAM structure.

Generally, the SDRAM protocol consists of six commands: activate, read, write, precharge, refresh and no-operation. The activate command opens a row in the memory array and stores it in a row buffer. Once the requested row is opened, read and write commands can be issued to access the columns in the row buffer. These bursts have a length of either 4 or 8 words. The precharge command is the converse of the activate command, as it copies the contents of the row buffer back into its place in the memory array.

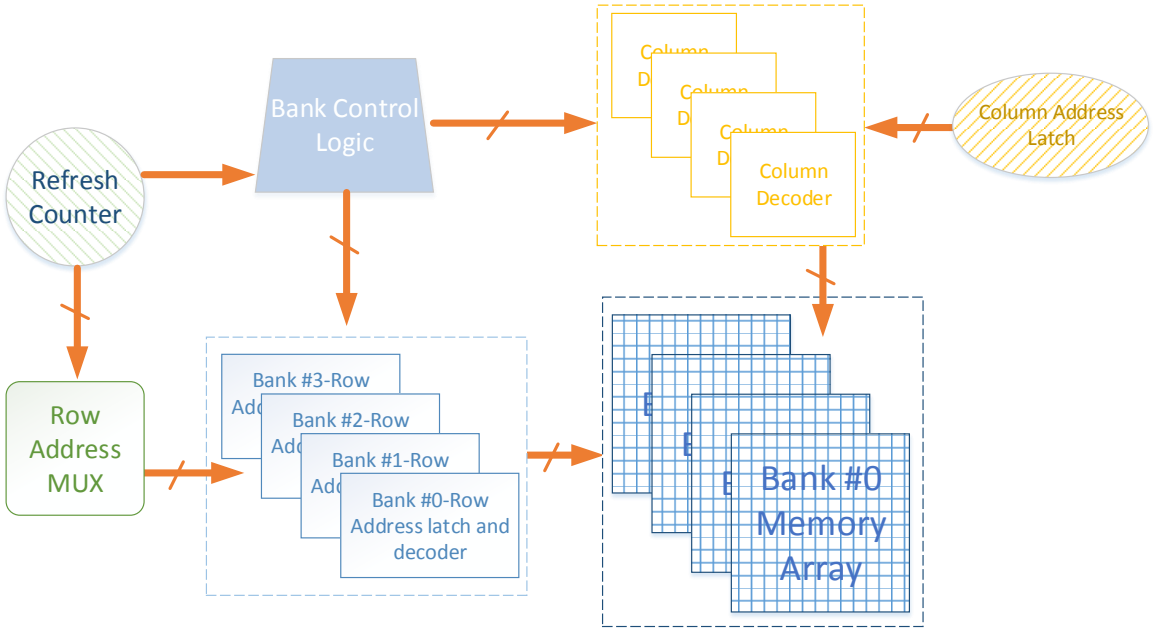

Fig. 2.5 DDR SDRAM memory block diagrams

It is possible to initiate read and write commands with an auto-precharge flag, resulting in an automatic precharge at the earliest possible moment after the transfer is completed. A memory cell stores a bit as a charge in a capacitor. Refresh commands are regularly executed in order to protect data loss, but no data can be transferred when the memory is being refreshed. Finally, if no more commands are buffered during the current clock cycle, then a no-operation command is issued.

The execution of memory commands in a SDRAM is technology dependent, and it is bounded by its timing constrains [JNW07]. These constraints aim to minimize the execution delays between consecutive commands, and they also define which command can be executed during a particular clock cycle. The peak of memory bandwidth can be determined by the width of the memory interface, its clock frequency, and its data rate. The overhead caused by the timing constraints can result in difficulties reaching this peak in SDRAM.

Cache memory is small and fast RAM-based memory that can be directly integrated in computational units or cores. Cache memory usage increases the overall system performance and it reduces the average time and power costs by providing direct and quick access to the frequently used data. Information from previous readings of memory commands can be stored locally at the core.

Data concurrency can relatively easily be managed in case of a single core level-1 (L1) cache. On the other hand, caches have to be part of a defined hierarchy in more complex systems. In multi-core systems for instance, all cores are executing in parallel, and most

likely, sharing same (or all) memory resources at overlapping time periods. This parallelism in data usage by the cores could result in wrong application outputs. In case of critical task executions this might be fatal (e.g., advanced driver assistant systems) or might result in high financial risk (e.g., high-frequency trading algorithms). Therefore, cache memory of level-2 (L2) and level-3 (L3) can be used to efficiently define the data accessibility. In this way higher levels of caches shares some data and information about lower caches.

Cache sizes are always powers of two, and they are usually in the order of a few kilobytes. A cache memory consists of a predefined number of cache lines that contains the frequently accessed data blocks. When a memory read or write request is initiated at a core, the required data might be located at its cache. Otherwise, it has to be located and copied from the system's main memory. This two cases are called *cache hit* and *cache miss*.

The structure of a cache is organized in sets of cache lines with data blocks. Each cache line contains three identifiers, a tag, data block and the flag bits. The tag represents the most important bits that are used to locate a specific cache line based on the actual address. The data block contains the cached data, and the flag bits are bits that are used to declare the validity of the loaded data. Data can be either *valid* or *dirty*. Valid data is data that has not been modified yet, otherwise it is called dirty. In order to locate a data block in the cache, the exact *cache set* has to be found. Later the *tag* identifier is used to locate the desired cache line. Finally, the *block offset* helps to locate the exact data block. These three parameters can be extracted directly from the address of the desired memory block.

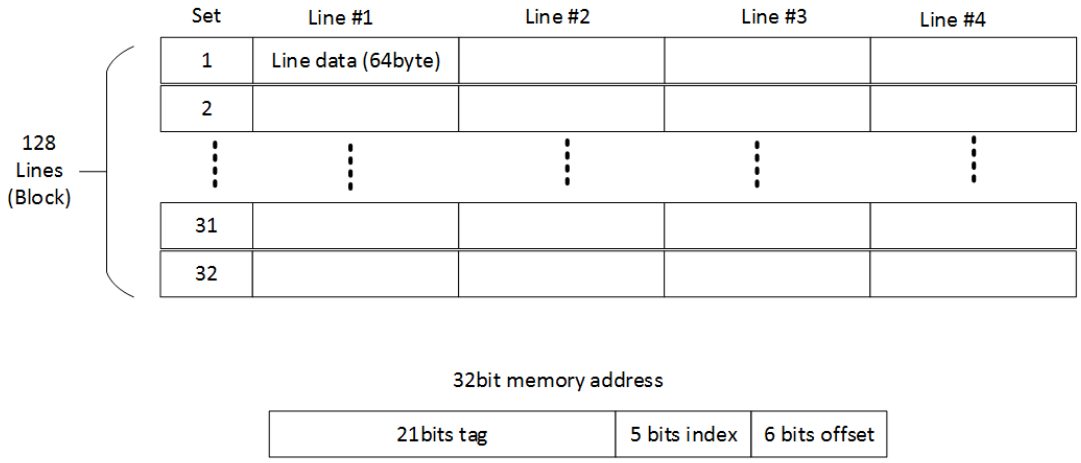

Fig. 2.6 4-ways set associative cache structure (8KB cache size of 64 byte cache block size)

As explained earlier, the size of a extracted is very limited, thus unused cache blocks shall be replaced. The way that the cache is configured defines its replacement policy of the cache blocks. There are three main ways of this configuration. First, if the fetched cache

block can be placed at exactly one location then it is called a *direct mapping* [Jou90]. Second, if the cache has the full flexibility to place the cache block at any location then it is called *fully associative*. Third, the cache is grouped in sets of N cache lines in which the new cache line and can be placed at any of the N places. *N-ways set associative* caches are commonly used caches in the industry (e.g., AMD ZEN uses L2 8-ways cache of 512K[1]).

N-ways associativity is usually implemented in even based scale, i.e., two, four, eight, etc [CJ06]. Increasing the ways that the cache is associated would increase the possible latency for the searching all N locations. Furthermore, it reduces the rate of memory misses. Handling memory misses requires the classification of the misses. A compulsory miss occurs in case the required cache block is not located at the cache an it has to be fetched from the main memory. A capacity miss is miss that occurs in case that the required cache blocks cannot be adapted in the cache due to its size. Later, further attempts shall be executed to fulfill this request. Conflict misses occur due to performing cache replacements and the limitation of the associativity structure. Each memory miss has its *miss penalty*, which is the time required to move and replace a cache line from a distant cache or the main memory to the local cache. Cache misses cannot be totally avoided but their frequency can be reduced. This can be done in many ways, such as having higher associativity or using pseudo-associativity or by using a victim cache [SMM+11].

Having N options for the replacement would raise the following question. Which cache line shall be removed? Existing algorithms (e.g., Least Recently Used (LRU), Most Recently Used (MRU), Least-Frequently Used (LFU)) are used for making the replacement decision. There are many other algorithms to resolve this issue, where the decision depends totally on usage case-specific conditions and the compromises designers are willing to do.

Hierarchical shared cache setups are required to reduce the complexity and execution costs in multi-core systems. In an N core multi-core chip, each core can have its L1 cache and each pair of cores can share an L2 cache and one L3 cache is needed for all N cores. Possibly multiple copies of the same memory block and deviations of timing in such cache structures raise the issue of memory consistency in the system. Consequently, cache coherency is used to refine the correctness of the memory references based on one of the consistency models. Many consistency models have been defined in the literature that define a specific level of predictability in memory operations. The most known consistency models for multi-core chips are the release- and sequential-consistency models.

Directory-based and *snooping* are the most known mechanisms to achieve coherency control. Both mechanisms keep track of all shared data in the system and maintain the consistency by validating the shared data with the collected tracking information. This

[1] AMD ZEN press release, August 2016

information is related to the cache line states, which must be *valid* for the main memory copies. The *Dirty* state is used for locally modified copies, and the *invalid* state marks shared copies of the cache line [Law98].

A number of snooping-based coherency protocols are used for cache and shared memory coherency. The idea of these protocols is to identify the different states of a given cache line during execution. These states can be **M**odified, **S**hared, **I**nvalid, **O**wned, **E**xclusive and **R**ead only. Coherency protocols such as MSI-based protocols and MOSI-based protocols, use a subset of the previously mentioned states to derive their decisions that guarantee the coherency of the shared memory.

2.4.2 Memory Controller

The main objective of a memory controller is to interface between the different components of the embedded system and their main memory (i.e., DRAM). A memory controller is responsible for managing and translating memory operations into instructions that can be executed by the memory. According to Akesson [AG11b], a memory controller has two ends. The *front-end* is agnostic to the memory and it buffers memory requests and responses in both directions. The *back-end* of a memory controller is responsible for handing incoming memory requests, managing responses and memory commands. These two ends have four components, an arbiter at the front-end, a command generator and a memory mapper at the back-end. In addition, the memory controller contains a data path that is horizontally passing data through both ends from and to the memory.

The *arbiter* is responsible of handling and scheduling memory requests. It defines the bandwidth and latency based on the requirements of the components accessing the memory. Scheduling at the arbiter can be either static or dynamic. Statically scheduled arbiters are predictable, whereas dynamic scheduling is often preferred since it supports run-time decisions that will result in better bandwidth utilization and lower latency. Round-robin and static priority scheduling are dynamic arbiters. Despite that Time-Division Multiplexing (TDM) is also a dynamic arbiter, its schedule is defined at design time [GGA+15].

The *command generator* is also a scheduler that is responsible for scheduling and generating memory commands according to the memory requests at the back-end. The command generator is customized depending on the generation and type of memory connected to the memory controller. It maintains the timing constraints of the system according to its schedule. Therefore, in case of a static schedule it considers the timing constrains at design time for well-known traffic. In case of a dynamic command generator these time constraints are checked at run-time. The run-time decisions aim to improve the flexibility in prioritizing

memory commands depending on the run-time conditions, with a trade-off towards the command generator's predictability.

The translation of the memory addresses into a form that can be used and understood directly by the memory is done at the *memory map*. These addresses are related to the memory requests received at the back-end of the memory controller. The outcome of the translation would result in the information about the desired bank, row and column of the memory. Memory maps define also the way for accessing the memory using memory patterns. The most commonly used memory maps are the bank sequential access or the interleaving access [GKAG12].

2.5 Transactional Memory

As Moore's law, in its classical form, is coming to an end, single core architectures stalled to give its place to multi-core architectures where two or more cores are integrated on a single chip sharing resources such as memory. The number of cores per chip increases steadily and today's multi-core chips have up to 72 cores as in the Xeon Phi super computing chip. This trend demands for a new way of handing shared memory resources. Transactions provide an abstraction that supports handling concurrency of a shared memory in parallel programming and multi-core systems. In contrast, today many programmers are asked to solve concurrency problems with very limited abstractions of the programming languages (e.g., locks, mutexes and semaphores). Even higher level programming languages are not composable enough and can easily result in failures in the shared memory concurrency. Locking-based mechanisms are used in handling a shared memory, but there are a lot of drawbacks such as priority inversion, deadlocks, missing composability, inability to scale easily with increasing numbers of cores and threads [FF11]. Transactional memory was introduced initially for databases, and later it has been proposed as a concurrency control mechanism to ease parallel programming and parallel processing in multi-core systems. It offers atomicity, consistency, durability and isolation guarantees in the access to a shared memory. By using a transactional memory, locks are not needed due to the fact that the sequence of the memory operations are executed in one transaction. This is similar to a database operation, that executes the operations and commits the changes at a certain time. Therefore, transactional memory is a competitive way for handling and coordinating concurrency of memory rather than using lock mechanisms.

All transactional memories have three main components, a version management: a conflict detector and a conflict resolution component.

Version management is a concurrency control protocol that holds different information of the transactions (e.g., ID, time stamps, versions and values) to keep track of all versions of the accessed memory locations. In order to understand the idea behind the version management, one can simply recall the different version control tools based (e.g., subversion) that are used by groups of developers to track current and historical versions of their projects.

Since a transactional memory is a lock-free mechanism, conflicts can occur. Therefore, a dedicated component detects conflicting transactions. The *conflicts detector* runs algorithms that detect transactions trying to access shared memory addresses.

Given a conflict is detected, the *conflict resolution* component is called to decide how to resolve those conflicts. A variety of different algorithms (e.g., priority based decision) has been introduced to resolve conflicts depending on the system-specific design and the requirements.

Both version control and conflict detection can either be eager or lazy. Eager version management means that updates and changes are directly reflected in the transaction records, which is making committing faster. A lazy policy is usually followed for fine-grained transactions. Eager conflict detection can identify conflicts early enough to avoid accumulation of conflicts that would result in having complex conflicts to be resolved later by the conflict resolution. This policy is commonly used in coherency protocols.

The use of transactional memories in embedded systems has also been proposed for dependable embedded systems [FF11], where both concurrency control and failure control must be addressed. The benefits compared to lock-based synchronization include a higher level of abstraction, better scalability and lower programming efforts.

In particular, transactional memories are useful for mixed-criticality systems, where subsystems with different safety assurance levels coexist on a shared computing platform. Mixed-criticality systems comprise subsystems with varying degrees of assurance and timing guarantees [Ves07a]. Mixed-criticality systems are a key enabler to reduce the cabling and the number of devices in many application domains including automotive, avionic, healthcare and industrial systems [OOA+14b]. The resulting benefits include lower cost, weight and maintenance efforts due to the higher integration and the more efficient use of computational resources.

2.5.1 Types of Transactional Memory

Three types of transactional memories can be distinguished based on their implementation: hardware, software and hybrid.

A *transaction* is an ordered sequence of memory operations that starts with a *start-transaction* instruction and contains a *commit instruction* at the end of the sequence.

HTM was introduced in 1993 by Herlihy and Moss [HM93]. This type of transactional memory directly targets the multi-core architectures, where additional transaction coordination components are integrated. Additionally, special extensions in cache forms are used to implement different cache coherency protocols for the support of the HTM architecture. Furthermore, low level modifications at core level provide capabilities for handling transaction in an atomic way, updating or even splitting transactions (when needed) in the multi-core system. For instance, Intel's 4th generation processors use HTM [Int14].

The classic implementation of HTM is done by having two types of caches. One dedicated cache for non-transactional memory operations handles all read operations as in a conventional direct-map cache. The second cache coordinates the transactional memory operations (i.e. write operations). This cache applies an additional logic that holds tentatively write operations until the transaction commits or abort. Generally, if a transaction is ready to be transmitted to the main memory in case of no conflicts then it is called ready to be *committed*. Otherwise, if conflicts cannot be resolved the transaction has to be dropped and is called *aborted*.

As the name states, *STM* is purely implemented in software. The idea of this solution is to store read and write logs while executing the version management component to maintain the system logs. The updates can be directly written and an additional log is used to track the value changes. This additional log is used later in case of conflict detection to undo or redo changes. High level programming languages provide an abstraction level of their languages supporting transactional memory. Programmers in this case have to handle and defined the atomic blocks considering all details themselves.

The initial software-based implementation of the transactional memory was introduced by Shavit and Touitou back in 1995. The idea was to bound the delays introduced in the memory hierarchy by decreasing the number and size of the critical sections in the multiprocessor applications using a non-blocking method [ST95]. This implementation proved the ability of the STM to manage the concurrency problem cheaper and more flexibly than the hardware solution. Moreover, software-based transactional distributed shared memory solutions were also proposed for distributed embedded real-time systems. These solutions are mainly focusing on improving the programmability of the system as in [HLRP] and D2STM [CRCR09].

Generally, HyTM offers significant performance gains compared to STM due to the implications of using high level programming languages (Java and C++ based solutions) [DFL+06]. Therefore, *HTM* was proposed to utilize the STM when the resources are sufficient to perform the transactional memory through software, and to use HTM in order to maintain the atomicity of the transactions executed in software. In such a way, transaction con-

flicts can be easily detected and resolved. The combination of the two approaches does not necessary mean that they directly depend on each others. In some cases HyTM simply switches between the two implementations depending on the performance of the transactional memory [CGS+14], [RMN+].

Chapter 3

Analysis of the State-of-the-Art

This chapter analyzes existing architectures for chip and cluster levels and their suitability for MCSs. Additionally, the architectural support for a hierarchical transactional memory is discussed. The analyzed architecture will be compared towards four major requirements. These requirements apply for both single chip systems and distributed MCSs. Later, the fulfillment of these requirements in the existing state-of-the-art solutions is analyzed. Thus, the outcome of this process illustrates the research gap in the state-of-the-art.

3.1 Requirements for Mixed-Criticality Systems and Transactional Memory Architectures

3.1.1 Real-time

The increase of complex and critical computational demands in the industry (e.g., automotive, avionic, health monitoring) has created significant pressure for designing embedded systems with real-time capabilities. As explained in Section 2.1, the correctness of the resulting outputs in an embedded systems is not attained if they are not produced at the right time. In this context, real-time is directly related to the ability of the system to be predictable and temporally synchronized. This depends on the real-time support in the system architecture at both software and hardware levels, including the cores, the interconnects between cores, the operating system, I/O, etc. In other words, *predictability* is what characterizes real-time systems [SR90]. Furthermore, regardless of the tolerance level for missing deadlines (i.e., hard, firm or soft), the real-time system shall guarantee a predictable behavior.

We can consider that all MCSs are real-time systems, but not necessarily the other way around. As already explained, the subsystems of a MCS have different criticality levels. The

predictability support in such systems shall take into account the criticality levels of the subsystems.

Design decisions of the MCS system architecture play an important role to ensure predictability. The architecture of the system (i.e., gateways, chip and cluster levels) shall support the timely execution the application service for safety-critical applications. Moreover, the architecture of a real-time system shall be able provide all required temporal properties at all levels of the system. These properties enable the monitoring of the system and the calculation of the delays and WCETs. Still, predictability of a service cannot be guaranteed if the system is not deterministic at all levels (e.g., subsystem level, communication level, memory hierarchy and memory access level).

MCSs shall guarantee the temporal order of the critical applications and services through a deterministic execution environment. Time predictability includes bounded latency and bounded jitter for the transferred messages and memory interactions in the real-time system. Bounded latencies are required to guarantee upper bounds on the response times of distributed application services and shared memory requests. Bounded jitter is required when the system has to react at a specific point in time for a safety-critical application or its memory requests.

3.1.2 Fault Containment

In MCSs, we need to guarantee that components with various criticality are not affected from faults in the systems. The isolation of faulty components shall be assured for the following two cases: a faulty (low-critical) component shall not affect higher criticality components. Additionally, faulty components of higher or equal criticality shall not affect lower criticality components. In general, components exchange their results continuously to achieve the required function that the system is dedicated to serve. The provision of faulty inputs can be masked by fault-tolerance such as n-modular redundancy. However, a necessary prerequisite is the independence of the replicated components by means of fault containment in the value and time domains.

Spatial isolation in a conventional shared memory hierarchy is controlled by the MMU [Joh99]. In case of systems with a transactional memory, the required isolation is established by performing atomic executions of the transactions. In other words, transaction-based memory segments can be isolated in case of failures. Atomicity of the transactional memory is considered as an isolation mechanism that provides deadlock avoidance and efficient concurrency control.

The isolation must be performed also at the communication level by defining a variety of communication channels connecting the components. In addition to the components, criticality, these channels shall consider different types of communication messages (e.g.,

periodic, aperiodic and sporadic). By enforcing the a priori knowledge about the permitted temporal behavior of components, communication channels can eliminate temporal fault propagation in the system.

Moreover, LIFs shall be used to provide an interface between the components in the system. A time sensitive LIF in combination with a MCS-aware TDMA scheme provides spatial and temporal isolation in the system. In case of a faulty component, its LIF shall isolate that component and potentially reset the component before reconnecting it to the remaining components of the system. Components are globally synchronized and the TDMA schedule enforces the occurrence of the events only within the active intervals of the execution time-line [Kop92]. Hence, faults can be encapsulated in case of their occurrence.

Based on these time sensitive LIFs, FCRs as introduced in Section 2.2.2 are introduced. Hence, the failure of one FCR will not negatively affect the remaining system. Additionally, FCR shall be defined based on the assigned criticality of the components. Moreover, a faulty behavior of a FCR (e.g., babbling idiot component) can be accurately detected and efficiently isolated.

3.1.3 Heterogeneity

MCSs are complex systems that are usually comprised of multiple nodes combined and connected through a chip level or cluster level network. These nodes can contain a variable number and different types of cores that differ in their criticality and models of computation. For instance, an embedded system could be composed of cores with different Instruction Set Architectures (ISA), or it could combine graphical processors with general-purpose computational cores. Furthermore, it is known that networks at chip level often run at different frequencies and timing restrictions (e.g. NoC) in comparison to cluster level networks (e.g. Ethernet). This *heterogeneity* of the system shall be considered in our design decisions at all system levels (i.e., nodes, cores, interfaces, gateways, memory hierarchy, chip and cluster level networks). The consideration of the system heterogeneity determines hardware design decisions and software support (e.g., protocol translation, communication management and serialization, etc).

Hardware components are provided by different IP suppliers such as processor designers. These components can use different communication protocols and memory hierarchies. This has resulted in the need to define common specifications to deal with the heterogeneity at the whole system architecture. At chip level, the foundation for Heterogeneous System Architecture[1] has defined a set of specifications with the help of the main market vendors (i.e.,

[1] HSA Foundation - http://www.hsafoundation.com/

ARM, Qualcomm, AMD, and others) to hide heterogeneity. These specifications are in the form of acknowledged architecture requirements, programming guidance and formal memory models. The foundation has specified general propose API specifications that promise to avoid operating system calls and provides an abstraction level that can support the hardware. However, these specifications at both hardware and software levels do not consider the high integrity demands of MCSs at chip level.

At cluster level, the distribution of the system and the heterogeneity of technologies used at its nodes shall be also hidden. This can be done by providing message translation services at the network gateways. In addition, a consistent memory hierarchy and shared memory protocol provides an abstraction layer to absorb and hide the heterogeneity of the distributed MCSs. This level of abstraction shall be able to handle the exchange of messages and memory requests at both chip and cluster levels correctly and concurrently while preserving atomicity.

3.1.4 Support for Hierarchical System Structures

The increasing number of components used in a single embedded system is not the only reason that embedded systems shall have a hierarchical architecture. The tremendous number of dependent services and functional features provided in such systems can neither be designed nor managed in a flat-structured system. Adding to that, many MCSs exhibit dissimilar requirements in terms of timing. Thus, the hierarchical architecture is essential in distributed MCSs, it supports the refinement of timing segments defined at the platform independent level, depending on the allocation choices made at the platform specific level. A hierarchical systems structure can support the optimization of message passing, shared memory data exchange and fault isolation in the system.

For example, an airplane system can only be designed hierarchically. The electrically controlled hydraulic system of the Flight Control Surfaces (FCS) in an airplane has hundreds of control functions that are executed concurrently to pilot a plane. This system is just one of the many systems comprising an airplane (cf. Figure 3.1) [2].

The architecture shall be defined in a hierarchical structure as described in Subsection 2.1.2. This means that each MCS shall consists of a number of interconnected nodes. A network gateway at each node shall perform protocol translation and message buffering between the nodes. Moreover, each node shall have a memory gateway and a configurable number of cores that are interconnected with a deterministic communication network. The

[2] Source: Federal Aviation Administration (FAA)

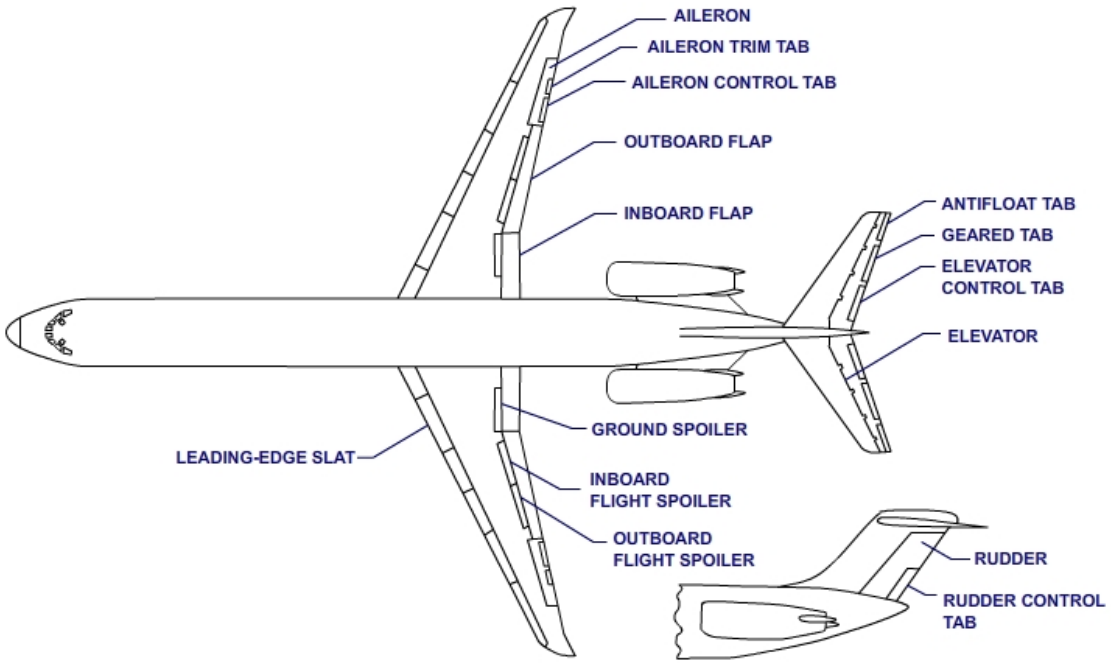

Fig. 3.1 DC-9-82 Flight Control Surfaces

communication infrastructure at both chip and cluster levels shall support hierarchical end-to-end communication channels that provide a predictable message exchange [OOA+14b].

The hierarchical architecture illustrated in Figure 3.2 shall be complemented with a hierarchical memory architecture with guaranteed atomic memory operations support. The architecture shall include predictable memory controllers and shared memory systems. Memory isolation must be improved at the system level by providing the ability to hierarchically manage the system's memory controllers. This memory hierarchy can utilize a local address space at each node and a common address space tracker at system level. This tracker is responsible for the concurrency at the system level. Cores shall have their own local cache, and these caches shall be maintained during data accesses by using a hierarchical real-time memory protocol for distributed systems. This memory hierarchy and its coherency protocol shall be aware of the criticalities of the system.

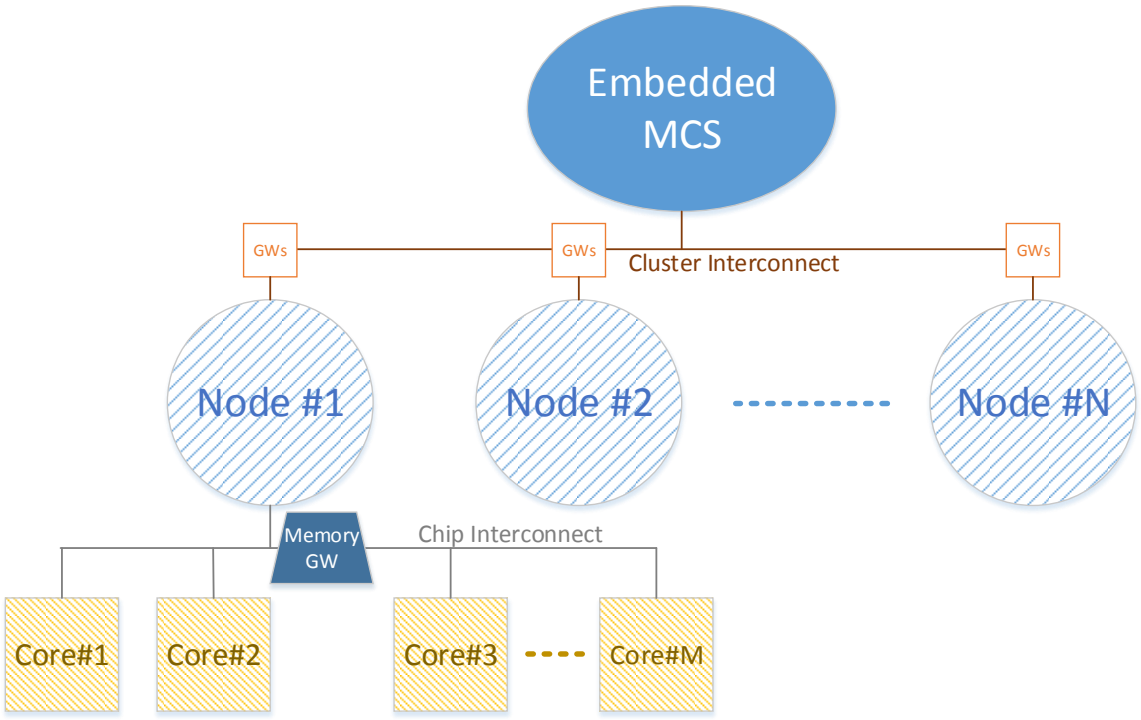

Fig. 3.2 Hierarchical MCS Architecture

3.2 Architectures and Solutions at Chip Level

3.2.1 Existing MPSoC Architectures

There are a numerous MPSoC architectures in the literature. In this section, two typical architectures were chosen to discuss the satisfaction of the requirements discussed earlier.

CompSOC Architecture

CompSOC is an MPSoC developed by the Eindhoven University of Technology and NXP semiconductors. CompSOC can easily be uploaded on a FPGA board where performance and energy consumption analysis can be performed. Therefore, it is used by many educational and research facilities.

The CompSOC architecture connects a configurable number of processor-tiles (or simply cores) implemented using Silicon Hive VLIW and tile-based memory units to the Æthereal NoC [HGB]. At NoC level, Æthereal consists of a number of communication routers linked together. The Network Interfaces (NIs) of the NoC are associated to the MPSoC tiles. Æthereal provides two types of communication services: guaranteed services and best effort services. The guaranteed services use TDM arbitration to ensure bounded latency for a given

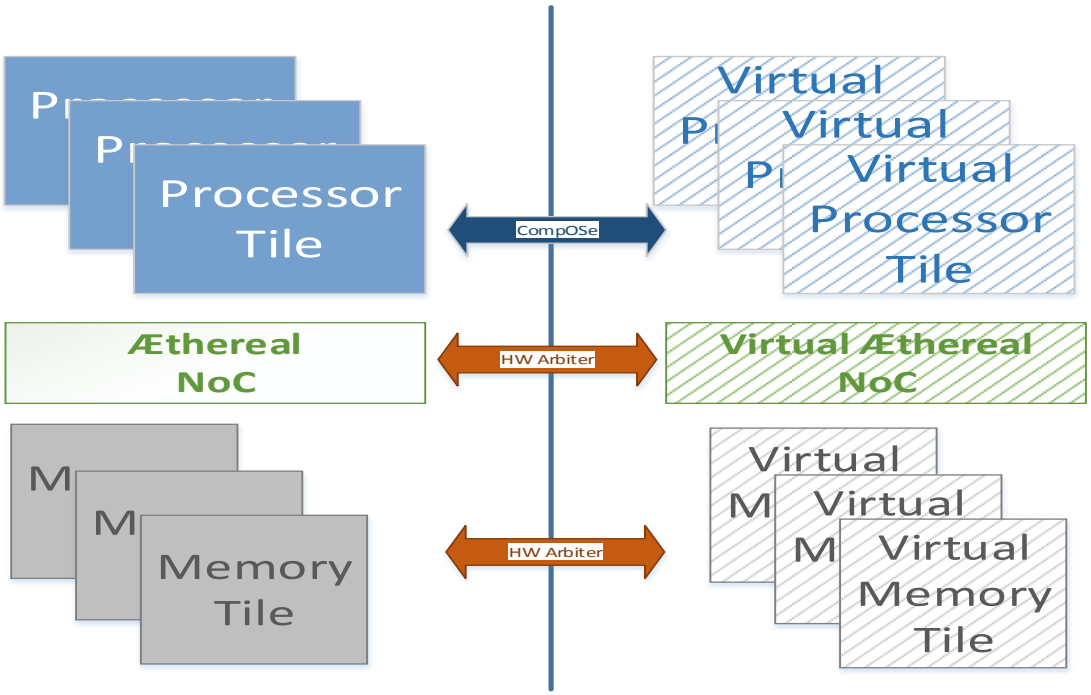

Fig. 3.3 CompSoC Architecture

communication link between two NIs. Best effort services utilize the remaining bandwidth with no guarantees [GDR05].

The NIs and the preemptive arbitration of resource scheduling are used to provide predictability in the system. Composability is addressed in CompSOC by eliminating the interference between the applications using the TDM arbitration.

Lately an improved version of CompSOC was introduced [GAC+13]. The new solution is a virtual execution platform with support for mixed-time criticality applications. Composability and predictability are addressed in the platform by enabling an isolated execution of the virtually partitioned applications, where each application has its virtual partitions by means of process and memory components. Virtual partitions are isolated using preemptive TDM in such a way that applications cannot be affected by other application partitions. The design flow of CompSoC includes all time constrains of the system to generate the virtual application software (i.e., virtual processor, virtual NoC and the virtual memory partitions). As illustrated in Figure 3.3, CompOSe is a real-time operating system that is responsible for the management of the virtual processors with the real processor tiles. The hardware arbiter is responsible for managing the TDM-based access between the (real and virtual) NoC and the memory components.

CompSoC is locally synchronous but globally asynchronous, where each tile operates at its own frequency. Therefore, is has multiple asynchronous resource schedulers. Resources

can be one of the following: a local instruction or data memory, a communication memory, Direct Memory Access (DMA), NoC or memory elements (i.e., SRAM, DRAM).

This newly introduced CompSoC platform uses one single global address space and it supports both non-shared and shared platforms. More specifically, in case of an application task accessing a non-shared resources the task's data and instructions shall be loaded to the respective memory resources. Local data is accessible in the data memory, whereas remote data located on another tile can be accessed using DMA and the communication memory.

In case of shared resources, DMA shall not be shared and the remaining resources are statically assigned to their owners and cannot be reconfigured. The composability and the predictability of the CompSoC is achieved by utilizing a strict and predictable scheduler that allocates resource budgets based on the time criticality (i.e., F/S/NRT).

Time-Triggered System-on-a-Chip (TTSoC) Architecture

This MPSoC is a component-based System-on-a-Chip (SoC) that incorporates determinism with encapsulation and fault containment. Time-Triggered System-on-a-Chip (TTSoC) was designed at Vienna University of Technology based on the MPSoC presented in [KOSH07]. The architecture and concepts of the TTSoC have been extended to cover the requirements for many application domains, especially for mixed-criticality applications [WESK10].

The generic architecture of the TTSoC is illustrated in Figure 3.4. It consists of a Trusted Subsystem (TSS) shown in blue color, and application subsystems that encompass a configurable number of micro components (μC). TTSoC realizes an inherent global time base for all components of the SoC [KB03].

The trusted subsystem comprises a number of linking interfaces known as Trusted Interface Subsystems (TISS). These interfaces in combination with the TTNoC [PK08] are responsible for guaranteeing predictable communication services and preventing fault propagation. The trusted subsystem is responsible for providing the globally synchronized local clocks, and supports periodic and sporadic message-based communication. Finally, the TSS provides the execution control services to the components. Namely, the TTNoC contains a number of interconnected communication routers, which define dedicated communication channels between different TISSs based on the a priori knowledge of the system communication requirements. The predictable time-triggered communication scheme used in the TTSoC defines a TDMA-based communication schedule (cf. Section 2.3). Each micro component has a preassigned communication slot in the time-triggered communication schedule. This temporal alignment minimizes the end-to-end latency of the encapsulated communication channels.

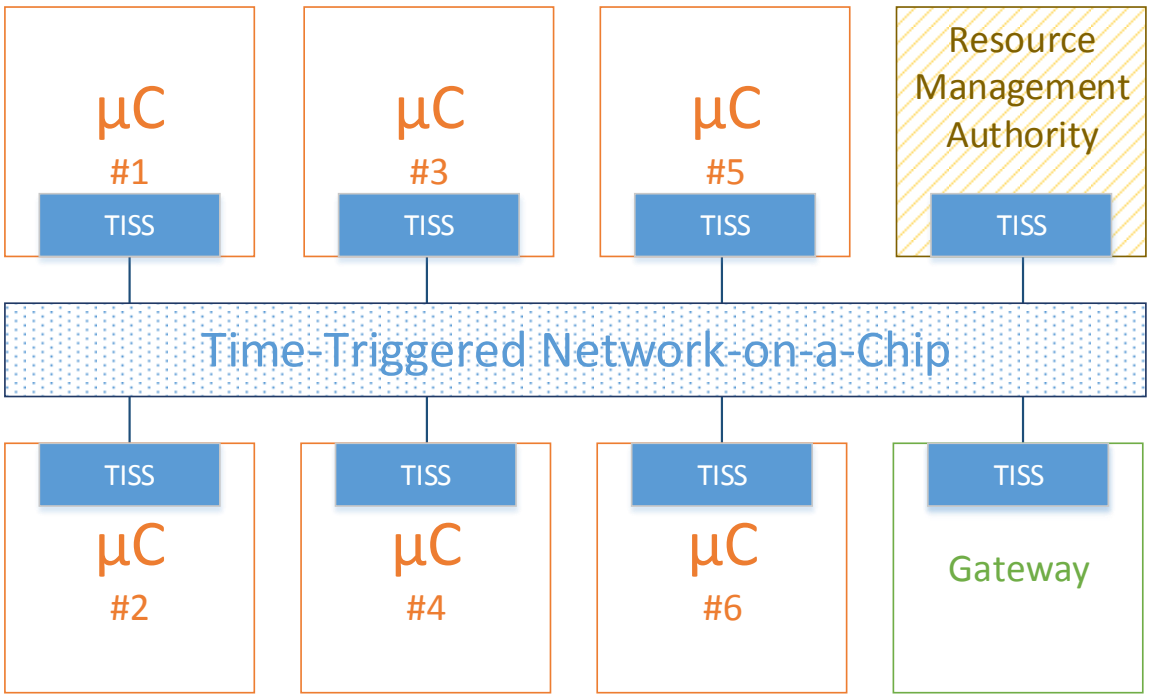

Fig. 3.4 Structure of Time-Triggered SoC Architecture

A micro-component can be one of the followings: an application host, resource management authority (RMA), trusted network authority (TNA) or a gateway. Application hosts implement heterogeneous application services of different domains where criticalities can be assigned. RMA and TNA are Intellectual Property (IP) cores provides resource management and reconfiguration services to the SoC. Additionally it is possible to use an diagnostic unit IP core to provide diagnostic services about the SoC behavior, e.g., for fault containment.

To our knowledge, few research has been done at the gateway level. Gateways are conceptually supported in TTSoC, and they are mainly presented as network gateways to provide access to cluster level communication [AOOM15]. This provides the ability to connect multiple TTSoCs to create a distributed time-triggered system based on SoCs. Moreover, it enhances the interoperability of the TTSoC with other cluster level communication network protocols (e.g., CAN, TTE, EtherCat).

3.2.2 Existing Memory Solutions

Shared-memory Solutions

Real-time requirements from a shared memory point of view are covered in the literature from different perspectives. Several authors provide solutions for predictable memory

controllers (e.g., [AGR07a], [GGA+15], [AG11a]), or even solutions for memory controllers specified for MCSs [GCAG16], [GAG], [ETSE14], [JQA+14]. Other solutions tried to emphasize the advantages of having a predictable TDM-based NoC that promises to guarantee predictable memory arbitration for the multi-core SoC [SCPS14]. Furthermore, shared memory abstractions can be deployed to provide guaranteed and best effort services at NI level [RDP+05].

An approach for task scheduling and memory partitioning for a MPSoC using a scratch pad memory is presented by [PAV14]. In [MPSV06] the focus of the work was on a Distributed Shared Memory (DSM) architecture suitable for low-power multiprocessors. A hybrid organization including private and shared memory space for DSM is introduced for multi-core processors in [CLJC10a]. This organization and run-time partitioning techniques are used in order to improve the system performance by reducing the Virtual-to-Physical (V2P) address translation overhead. Moreover, Chen in [CLJC10b] proposed a microcoded controller as a hardware module in each node to interconnect the cores, the local memory and the network. None of the previously mentioned solutions proposed the use of transactional memories.

Transactional Memory Solutions

As explained earlier in Section 2.5, transactional memories can be implemented in hardware, software or a hybrid combination of both [HLR10]. A study on transactional memories for dependable embedded systems is presented in [FF11], which shows the challenges in handling software and hardware faults. In addition, real-time constraints and consistency of transactional memories must be addressed. Therefore, further research in languages, tools, algorithms, run-time systems and hardware architectures is required to support transactional memories in embedded systems.

Several approaches for software transactional memories in dependable embedded systems were presented [BP11] [BC11] [ESR12], which focus on the schedulability of a given task set by pre-estimating the WCET. [PMM+13] presents a dynamically partitioned cache interface that handles the memory transactions on top of the memory controller to ensures a bounded maximum delay for hard real-time tasks. In [MWU13] conflicts and aborts with hard real-time and best-effort transactions were analyzed by inspecting the set of possibly overlapping transactions that may conflict. Moreover, a STM contention manager for priority-based transactions for real-time systems has been introduced in [GC08]. While these solutions address the conflict resolution of transactions, they do not avoid temporal interference in the memory gateway and the interconnect. Mixed-criticality aspects of a transactional memory

were addressed in [FRJ], which introduces a methodology to compute the upper bounds for the response time when concurrency control is managed using STM.

The concept of prioritizing the transactions in such a way that transactions with higher priority take precedence over lower priority transactions is well-known from the database world [GR93], [MSAHB04] and also used in embedded systems. For instance, the Transactional memory Coherence and Consistency (TCC) model [HWC+04] is widely used for automatic rollbacks to resolve conflicts in embedded systems without criticality considerations. TCC inspired the development of RTTM, a Java-optimized processor [SBV10], [SH10] that supports time-predictable and bounded WCET. RTTM proposed a time-predictable synchronization solution built on the assumption that memory access is based on a TDMA scheme. It also provides a bounded number of maximum transaction retries for real-time systems. While real-time aspects are addressed in these software-based solutions, they are restricted to the Java programming language. Additionally, the memory gateway offers only a simple connector to the memory arbiter with no consideration in regard to the criticality of the transactions.

LogTM [M+06] and its Signature Edition (LogTM-SE) extension are well-known hardware transactional memory solutions. LogTM-SE provides eager conflict detection and eager version management by summarizing transaction sets in logs and signatures. ATLAS [NCW+] is a prototype of a multiprocessor with hardware transactional memory support, where cores can access coherent shared memory in a transactional manner. TMNoC is a proposed design approach that aims at mitigating false-forwarding of unsuccessful transactional memory requests [ZCCD13]. A performance analysis of a hardware transactional memory solution in an NoC-based MPSoC environment is presented in [KGaW]. Moreover, this solution was compared to a traditional shared memory model that uses locks to provide consistency. Furthermore, the performance and power evaluation of lock-based synchronization over a set of different system and application settings shows that a transactional memory is a promising solution even for resource-constrained embedded multiprocessors [F+07]. However, these existing architectures do not focus on temporal predictability and real-time aspects of mixed-criticality embedded systems.

Existing hardware-based transactional memory solutions for non-distributed systems propose a small, fully-associative transactional cache at the same level as the L1 cache to reduce effects of capacity and conflict avoidance [FWM+10]. To our knowledge, transactional memory solutions for mixed-criticality embedded systems address one multi-core chip only. In [CCP+15], spatial and temporal segregation using a dynamic TDMA-based memory arbiter based on the MultiPARTES platform is provided. Moreover, memory access control in a multiprocessor for real-time systems with mixed criticality is proposed by [YYP+].

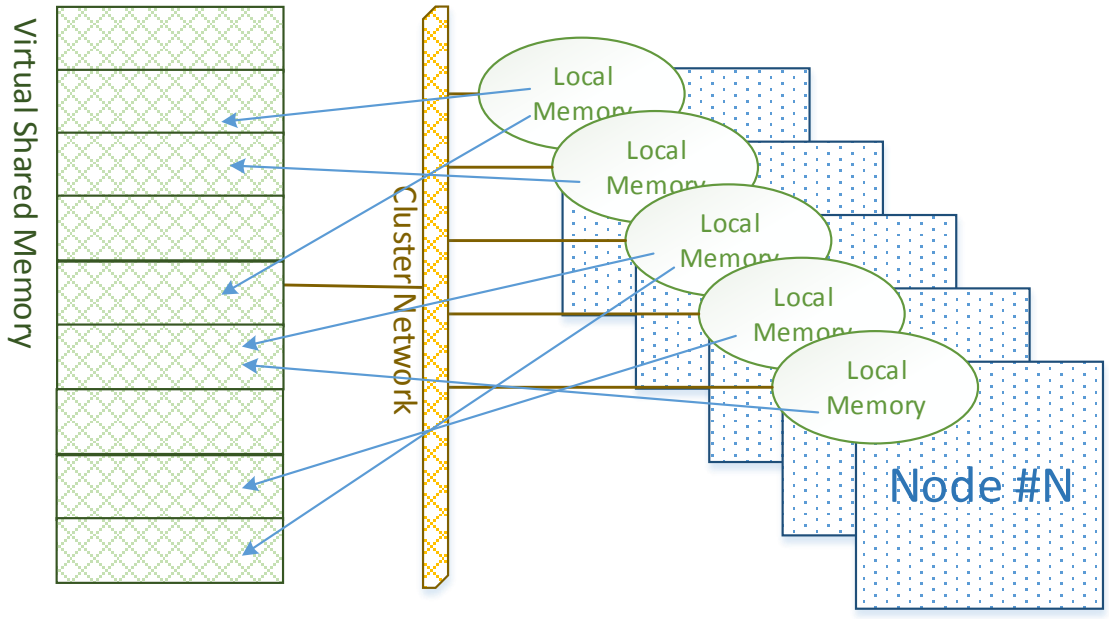

Fig. 3.5 DSM Architecture

This work provides a software-based memory throttling mechanism to explicitly control the memory interference in the Linux kernel.

3.3 Cluster level Distributed Memory Solutions

The description and discussion of existing memory solutions for distributed systems at cluster level is presented in this subsection. Initially, generic DSM solutions are discussed and later solutions based on transactional memories are presented.

3.3.1 DSM Solutions

A DSM architecture consists of a number of nodes that share a virtual memory. The system memory is split into several segments that are distributed between the nodes and the system's main memory. Data exchange in between the nodes is achieved using the nodes' memory managers and the system's virtual memory mappers (cf. Figure 3.5). The major advantage of DSM is the high portability and easy to use communication model in comparison to message passing. Memory instructions can be executed during non-overlapping time periods, yet exchanging data between system nodes can easily cause errors and performance penalties. This is a major drawback in real-time and fault containment.

DSM can be on both hardware or software levels. At hardware level, cache coherency circuits and network interfaces can provide support for DSM solutions. At software level, DSM solutions can be implemented with the help of high-level programming languages that offer lock-based solutions (e.g., mutexes). Shared memory in this case can be organized in tuples, fixed-size pages, or as abstract objects of variable size.

It can be noticed from the literature that few research has been performed on DSM solutions dedicated to multi-core processors with a NoC in comparison to off-chip solutions. DSMs have been investigated in the context of cluster level architecture as well as in operating systems [CYY13]. Furthermore, partitioning of virtual memory resources and task scheduling is proposed as a solution for managing memory accesses at operating system level. It is common to use a monitoring unit called *central manager* which is responsible of tracking and logging all shared pages between the nodes of the system. This manager is responsible of handling the ownership and accesses of the pages in a directory-based DSM.

3.3.2 Transactional Memory Solutions

A Distributed Transactional Memory (DTM) is the realization of a transactional memory using off-chip communication networks. The so-called Ballistic protocol [HS05] for instance, uses a cache-coherence protocol based on a transactional memory for a network of nodes for tracking and moving up-to-date copies of cached objects. Additionally, the Spiral directory-based protocol presented in [SBS] is a distributed implementation of a software transactional memory based on sparse covers, where clusters at each level are ordered to avoid race conditions while serving concurrent requests. Transactional Distributed Shared Memories (TDMS) were proposed for distributed embedded real-time systems (e.g. [HLRP]). The focus is improved programmability in conjuction with temporal predictability and composability.

The java-based transaction execution engine of DSTM2 [HLM06] is used as a baseline for some off-chip distributed system solutions. The prototype of a distributed software transactional memory framework is described in [KAJ+07] based on a modified version of DSTM2, where a master node is responsible for conflict detection and a contention manager resolves the conflicts. All client nodes have to update and synchronize their local copies of the global data. Moreover, in [DSRSZ10] an extension of the transactional engine DSTM2 establishing a transactional memory for distributed memory architectures is introduced for providing transactional consistency.

Most existing DTM frameworks are prototyped on top of VM-based programming languages, e.g., Scala and Java. HyFlow [SR11] is a Java framework for a distributed software transactional memory with pluggable support for directory look-up protocols, transactional

synchronization and recovery mechanisms, contention management policies, cache coherence protocols, and network communication protocols. On the other hand, locks are realized using Jave 5 annotations and transactions are defined as atomic sections, in which reads and writes to shared, local and remote objects appear to take effect instantaneously. [MTPR13] presents a distributed transactional memory framework for distributed concurrency control in C++ based on [SR11] called HyflowCPP. The Real-Time Transaction Forwarding Algorithm [HLRP] extends the distributed concurrency control scheme of the Transactional Forwarding (TFA) [SR12]. It bounds transactional retries by resolving transactional contention using time constraints. The authors used JChronOS, a user-space library which provides the hooks to interface with the ChronOS kernel from a Java application to define the time constraints.

3.4 Research Gaps in the State-of-the-Art

3.4.1 Analysis

As shown in the previous sections, a significant number of memory-based architectures were presented. To clarify the research gaps, existing solutions will be analyzed from different perspectives at different architectural levels (i.e., component level, network level, gateway level, and cluster level).

Existing chip level architectures provide hierarchical support at chip level for both message based and shared memory interactions. Due to the use of the deterministic TTNoC and Æthereal networks, the TTSoC and CompSoC architectures guarantee time predictable communication behavior.

The latest research results for supporting MCSs show the ability of TTSoC to provide the required fault isolation and predictability for MCSs. Additionally, MCSs-aware memory controllers compatible with CompSoC were introduced. These memory controllers can assure the predictability and timeliness of the memory instructions at memory gateway level.

The properties and characteristics of both architectures provide the baseline for a chip architecture that can be used for message-based and shared memory interactions in a MCS. The resulting architecture shall also take the advantages of utilizing the transactional memory to provide the missing segregation at the different levels of the entire chip architecture. This segregation shall avoid conflicts due to memory rollbacks, and between applications of different criticalities at the level of the network interconnect, and provide segregation between memory requests of different cores. Moreover, maintaining a predictable behavior of the message-based and shared memory instructions at the whole chip architecture.

Chip level shared memory solutions consider partly the requirements presented in subsection 3.1. These solutions focus mainly on real-time requirements at memory controller level by introducing predictable memory arbiter schedulers. Moreover, some memory controller solutions prioritize memory requests based on their criticalities.

Special extensions for memory controllers supporting a hardware transactional memory were presented to establish bounded maximum delays. It is common in chip level transactional memory solutions to choose logging-based STM with support of a hierarchical memory structure at component level. This type of solution provides the required predictability for the memory instructions, yet it neglects fault isolation and message-based interactions of the on-chip components.

Solutions at cluster level are mainly for distributed networked nodes with no consideration for the nodes' multi-core interconnect. Present DSM research is mainly focusing on algorithms for virtual memory management and memory segments location services (e.g., cloud). To our knowledge, current solutions do not consider MCS requirements and they have very limited considerations for the system hierarchy support and fault isolation. For instance, hiding heterogeneity at memory and network gateways has not been investigated for MCSs.

Distributed shared memory solutions at the level of operating systems do not satisfy the requirements of hierarchical system structures as well as the diversity of execution time models. Such solutions focuses mainly on schedulability of the memory accesses at the node level.

Transactional memory is often used in cluster level DSM solutions for embedded systems. STM-based protocols are mostly implemented with high-level programming languages. The predictability of the system is relevant to manage the complexity of the virtual-machines. Based on the state-of the-art analysis, fault isolation is not considered in existing solutions, and there is no hierarchy support in these transactional memory protocols.

3.4.2 Conclusion

Current chip level architectures do not hide the heterogeneity of message-based and shared memory interactions while satisfying the requirements of MCSs. In particular, existing transactional memory solutions do not ensure predictability and fault containment at the levels of the interconnect, the transactions and the memory gateway.

Existing transactional memory solutions that deal with predictability in MCSs limit their interest to the transactional memory itself. These solutions do not deliver an overall architectural solution for both transactional memory and NoC for the predictable resolving of real-time concurrency in mixed-criticality embedded systems.

The focus in conventional solutions is either the chip or cluster level memory, while focusing mainly on the predictability and schedulability aspects of one component (e.g., memory controller). A major research gap are architectures, system models and algorithms for transactional memories in hierarchical systems comprising networked multi-core chips. These systems combine two integration levels. Firstly, multi-core processors consist of a set of computational cores that interact via an on-chip interconnect. Secondly, the cluster-level uses off-chip networks for the interconnection of several multi-core processors. A single multi-core chip is often insufficient to meet the resource requirements of large embedded applications. In addition, the failure rates of a single chip are too high to meet the reliability requirements of fail-operational systems with ultra-high dependability [SWH95] (e.g., Class A according to DO-178C [RTC11]). Hence, fault-tolerance at system level is required by exploiting redundancy with multiple independent chips.

The hierarchical support for an architecture providing a transactional memory at both levels is missing. Moreover, predictability, fault isolation and reliability requirements are not addressed in existing solutions for distributed MCSs. The requirement for hierarchical support at both levels is limited to one of levels. Namely, a MCS-aware hierarchical architecture that is utilizing transactional memory, and supports communication protocols and algorithms at both chip and cluster levels does not exist.

The state-of-the-art does not offer memory and network gateway services between chip and cluster networks with a transactional memory protocol for selective redirection of information, fault isolation, and name space mapping. Additionally, the state-of-the-art provides no services for access to remote virtual memory resources and seamless network protocols for off-chip resources in satisfying the presented requirements.

Chapter 4

Transactional Memory Architectures for Mixed-Criticality Systems

Transactional memories offer the potential to combine an optimistic operation of non safety-critical applications with strict real-time guarantees of safety-critical applications. However, a fundamental requirement in mixed-criticality systems is the segregation of subsystems with different criticality. Non safety-critical task should either have no effect at all on the execution times of safety-critical tasks or the effect must be bounded and known. The requirements discussed in Section 3.1 are addressed in the proposed architectures. We support predictability, partitioning in time and spatial domains as well as heterogeneity of the architectures and their underlying algorithms and protocols. In addition , the architectures offer fault containment and support for hierarchical system structures.

Two architectures for MCSs with a transactional memory are presented in this dissertation. The first architecture introduces an architectural model of a multi-core chip (cf. Section 4.1). This architectural model comprise cores and a gateway, which are interconnected through a deterministic and reliable NoC. Moreover a memory controller with support for predictable transactional memory services is used. The second MCS architecture (cf. Section 4.2) presents a distributed hierarchical model for multi-core chips at the cluster level. Furthermore, it integrates both on-chip and cluster networks with different memory technologies and communication protocols into a coherent reliable protocol for networked multi-core chips.

4.1 TMSoC System Architecture

The proposed system architecture, denoted from now on as Transactional Memory System-on-a-Chip (TMSoC), aims to assure the prescribed requirements by introducing a component-based architecture. The physical separation of the components and their independent design

are the basis for the Fault Containment Regions (FCRs). In case of a design fault or a physical fault within a given component that might lead to a failure, this specific FCR will be isolated. The idea behind this system architecture is to enforce segregation of applications based on their criticality, which is also applied at the network interconnect between the cores and gateways. Particularly, the message-based and shared memory interactions are executed concurrently between the cores and the memory gateway in a way that guarantees predictability and coherency in the memory hierarchy of the architecture.

Conventional memory controllers are responsible for scheduling memory request from different cores to optimize the throughput. In contrast, the memory controller introduced in this system architecture is able to handle requests based on their criticalities and it bounds temporal interference between requester cores at the gateway level. The TMSoC algorithms introduce *selective conflict resolution* in order to avoid timing effects of non safety-critical applications on safety-critical ones. A commit operation is only performed in case there is no resulting rollback of an application with a higher criticality. Therefore, the Worst-case Execution Time (WCET) of an application only depends on transactions by applications of the same or higher criticality. Hence, dependencies are avoided in applications with lower assurance levels as a prerequisite for modular validation and certification. The above mentioned issues are discussed in details while introducing the system architecture in this subsection.

As illustrated in Figure 4.1, the TMSoC consists of the Multi-Processor System-on-a-Chip (MPSoC) with a memory gateway providing bi-directional access from/to the external memory. Moreover, the memory gateway is responsible for providing transactional memory services through the Mixed-Criticality Transaction Controller (MTC). The MPSoC contains a configurable number of cores. The cores and the memory gateway are connected to the TTNoC using Time-Triggered Network Interfaces (TTNIs) (cf. Section 3.2). The TTNoC is a TDMA-based NoC that supports different topologies and it provides fault isolation and temporally predictable communication between the cores and the memory gateway. Namely, one core cannot affect the other cores in the value or time domain. The configuration of the TMSoC is defined based on the a priori knowledge of the communication topology and the timing.

4.1.1 Core Architecture

The *cores* implement the application services using the *host processor* and the *memory service module*, both of which are connected to the TTNoC through a TTNI. *Subsystems* with different criticalities are distinguished in the proposed architecture, where each subsystem comprises one or several cores. For instance, subsystems can be assigned a Safety Integrity

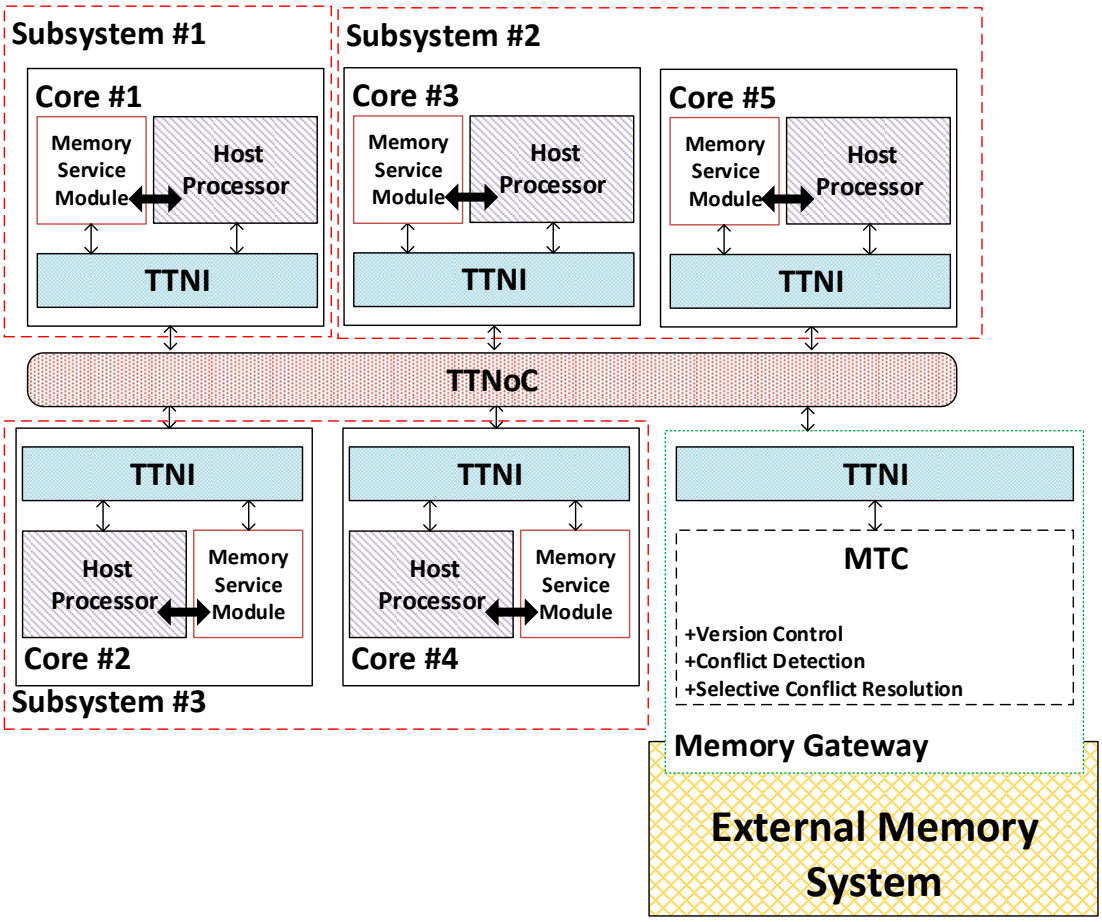

Fig. 4.1 The TMSoC System Architecture Overview

Level (SIL) according to IEC 61508 [Bel06]. The definition of the subsystems and their criticality levels is done at design time as part of the TMSoC configuration (cf. end of Section 4.1).

Host Processor

The *Host Processor* implements custom applications of the TMSoC. An application can be implemented either on a single core or it can be distributed between more that one core of a subsystem. Hence, the access to the data for the application execution requires the TMSoC external memory. This access is managed and achieved through the services of the memory service module.

Memory Service Module

The *memory service module* provides basic cache services, where memory instructions relevant to this host processor are processed. This module is responsible for handling memory requests and replies. In addition, it also maps these instructions to time-triggered messages that can be exchanged via the TTNI. Additionally, it sends commit requests and receives acknowledgments from the memory gateway in case of successful commits or rollbacks. The criticality-aware transactional memory algorithms introduced in TMSoC are responsible for assuring the correctness of the memory instructions. These algorithms will be explained in details later in this section.

Time-Triggered Network Interface (TTNI)

The TTNI has a well defined temporal behavior and offers a standard interface to the host processors with incoming and outgoing message ports. The TTNI is responsible for transmitting the messages according to the time-triggered communication schedule. It is considered that applications and their memory access are also TDMA-aware in such a way that their temporal behavior is mapped to the TTNIs by utilizing the schedule. The schedule in the TTNI controls the message path using source-based routing as well as the transmission behavior of the core where messages are injected to the TTNoC at predefined points in time. Based on the predefined temporal and spatial resource allocation of the NoC, the TTNI isolates faulty applications with respect to the interconnect and prevents interference between subsystems. Host processors can provide data, but do not influence the scheduled transmission times of the messages.

4.1.2 Time-Triggered Network-on-a-Chip (TTNoC)

The TTNoC is configured with a communication schedule, which defines the required time-triggered channels and their interconnection to transmit messages using source-based routing. These time-triggered channels define the temporal and spatial allocation of resources for messages traversing from the source core along routers to the destination cores. Spatial and temporal partitioning are guaranteed based on the a priori knowledge of the permitted communication behavior using the concept of time-triggered channels. It is the responsibility of the TTNI to transmit messages using the correct time-triggered channel according to the communication schedule with application data provided by the ports towards the host processors.

The applications must have pre-assigned time slots based on a global time base. Additionally, sender and receivers of each time-triggered message are defined. *Time-triggered*

messages have the following configuration information <period, phase, size, channel ID, sender core ID, receiver core IDs>. These configuration parameters represent a priori knowledge that can be used for fault containment, where the TTNI can isolate faulty applications and prevents interference between subsystems. Hence, design faults or hardware faults restricted to individual cores (e.g., local SEUs and SETs) cannot propagate to other cores in the TMSoC.

Cores interact at TTNoC level by passing messages or by using shared memory interactions via the corresponding channels. Thus, shared memory interactions are only realized by the memory gateway and the memory service modules. They wrap the related information into messages, i.e., request messages and reply messages that can be processed by the TMSoC.

4.1.3 Memory Gateway

The external memory is accessible for the cores of the MPSoC using the *memory gateway* via the TTNoC. The memory gateway is responsible for processing memory transaction operations received from the TTNoC and relaying these operations from and to the external memory. The TTNI of the memory gateway uses a time-triggered schedule with inbound and outbound time-triggered communication channels to the different cores.

Memory transactions sent from a core's memory service module to the external memory system are handled in the memory gateway. First, memory transactions are processes with the Mixed-Criticality Transaction Controller (MTC) algorithms based on their criticalities. Then they are queued in the memory controller to be executed. Later, memory replies are mapped to the appropriate time-triggered channel in order to send them to the correct requester cores.

TTNI at the Memory Gateway

The TTNI of the memory gateway follows the time-triggered schedule defined for the TMSoC with inbound and outbound time-triggered communication channels to the different cores. Incoming memory requests and transaction replies are mapped in the memory gateway to the appropriate channel in order to send them to the correct cores. Thereby faults are contained within the individual cores. Messages of the host processors and memory transactions of the memory service module do not affect the scheduled messages of the TTNI.

Mixed-Criticality Transaction Controller (MTC)

The MTC is responsible of processing the received transactions, executing commits and performing selective conflict resolution based on the criticality of the transactions. It provides

version control, conflict detection and selective conflict resolution services based on the criticality of the subsystems. The MTC is connected to the TTNI via ports, where each core has its own corresponding in-port and out-port. Moreover, sending and receiving queues are mandatory as the interface between the transactional memory controller and the memory controller.

The MTC is responsible for processing memory transactions while providing *eager* version control and *eager* conflict detection. The proposed eager version management and eager conflict detection increase the performance (cf. Section 2.5). Furthermore, conflicts are resolved using the algorithms of the MTC (cf. Section 4.1.5).

The MTC contains the following two containers that are used by the algorithms: the *transaction records* and the *transaction-version container* (cf. Figure 4.2).

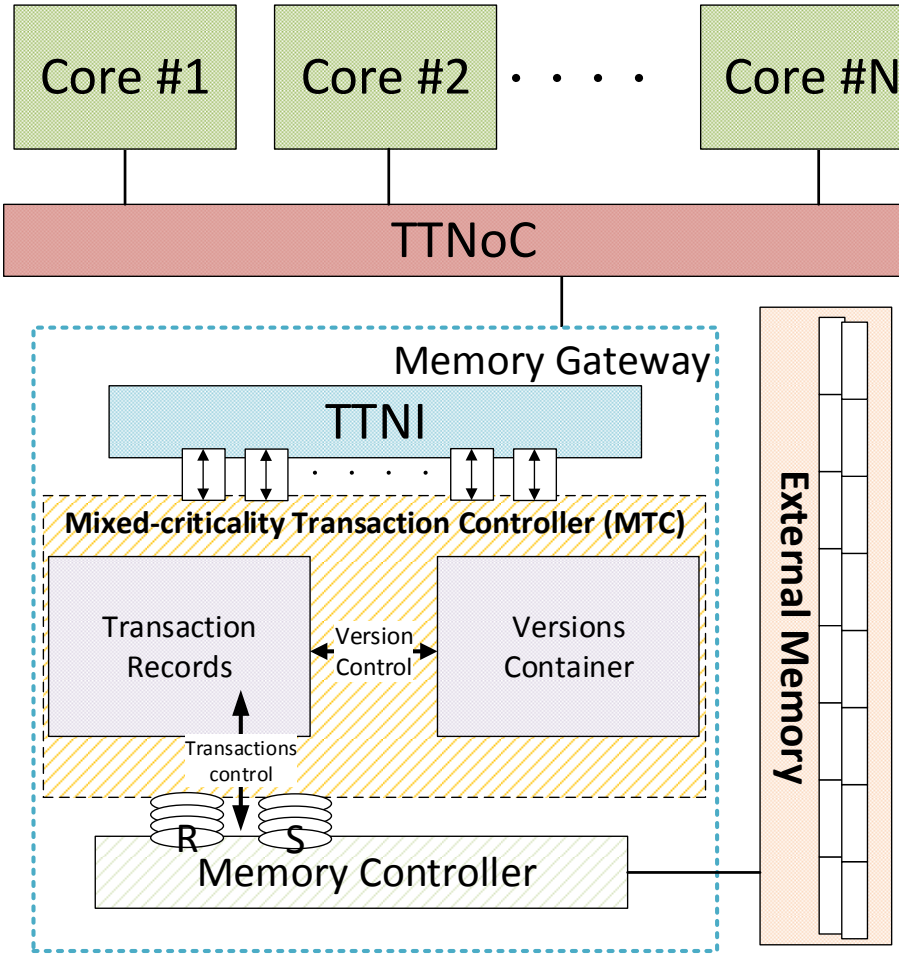

Fig. 4.2 MTC Architecture

Transaction records are required to ensure the correct committing of the transactions. This building block holds tracking records of the memory transactions. In the transaction records, each core has its own records' queue. Moreover, the last committed transaction is always known in each queue.

The *transaction-version container* handles the overall versioning of transactions. This building block implements a hash table that is always aware of the transaction status and the validity of its records. Transactions are labeled with one of the following tags: *exclusive* for transactions with so far no access to shared memory locations and *shared* for transactions that have at least one memory location, which is being accessed by another ongoing transaction.

The algorithms executed in the MTC are responsible for processing the received transactions received from the TTNI. The first step is the updating of the transaction-record container. Afterwords, if a write command is processed, it will be inspected for conflicts and the algorithms will act based on the collected knowledge in both containers. If a read command is processed, the reply of this command will be stored in the sending-buffers of the MTC. Each sending-buffer is associated with one of the cores through its ports. The separate buffers are responsible for guaranteeing the isolation and for the correct delivery of the transaction replies to the TTNI.

The cores have to be informed about the result of their committing attempts and they need to be informed in case of rollbacks. Therefore, the results of the algorithms' execution are transmitted to the memory service module of the transaction's requester core.

Memory Controller

Typically, memory controllers optimize throughput, while temporal interference between cores is not addressed (e.g., bank switching). Temporal predictability in the proposed transactional memory architecture is assured also at the memory controller level. Moreover, the memory controller schedules memory requests from computational cores with different criticalities utilizing the assurance layer provided by the MTC.

As explained in Section 2.4.2, there are two types of arbitration in memory controllers. Firstly, statically scheduled memory controllers use preconfigured schedules. This type is unable to adapt to changes in timing requirements of critical applications at run-time. Secondly, dynamically scheduled memory controllers promise bounded latencies and provide dynamic reallocation of the memory bandwidth [RDK+00]. Therefore, the proposed memory gateway assumes a compositional real-time memory controller such as [AGR07b] that supports analytical design-time verification of hard real-time requirements. This memory controller follows a two-step approach that starts by defining the memory access groups with known efficiency and latency. Then, a predictable credit-controlled static-priority

arbiter [AGR07a] is responsible for scheduling these groups dynamically. The dynamic scheduling guarantees the allocated bandwidth and the maximum latency bounds of the memory interactions.

4.1.4 External Memory

The *external memory* implements an interface that allows it to be connected to other cores of the proposed memory architectures, where incoming memory instructions from the memory gateway are redirected to the corresponding memory controller of the external memory and vice versa. The external memory involves a configurable memory unit with a memory controller. The memory system provides a single or multi-channel memory system, where each channel is composed of a configurable number of ranks and banks. Each memory channel has a pending transaction queue that is attached to a corresponding memory controller. The memory controller contains several queues for storing the received transactions, operations of the transactions, pending read transactions and returned transactions.

The following transaction queues are linked to the memory channels: The *pending queues* store the received transactions and the pending read transactions, the *transaction-command queue* contains operations for starting and committing transactions, the *result queue* contains the returned transactions.

4.1.5 Mixed-Criticality Transaction Controller (MTC) Algorithms

The algorithms, sets and entities of the Mixed-Criticality Transaction Controller (MTC) introduced in the proposed transactional memory architecture are discussed in this section. The MTC is responsible for guaranteeing the isolation and predictability as well as avoiding temporal interference at the transactions level. The following ordered list of elements, so called tuples, are used to express the data structures used in the algorithms of the transactional memory architecture:

- **Memory Transaction**: The algorithms handle memory transactions. Therefore, a memory transaction is defined as $T = \{(M, c, cs)\}$ where M is the set of memory operations, c declares the transaction's criticality (e.g. $c \in [SIL1, \ldots, SIL4]$ [Bel06]), and $cs \in [True, False]$ is the committing status of the transaction. This committing status is initially set to $False$ for all transactions.

- **Memory Operation**: memory operations contain the read and write instructions of a transaction. They are denoted as follows: $M = \{(op, a, s, v, j)\}$ where $j \in [1, J]$ is the memory operation's index, $a \in [0, address\_range]$ is the memory address, $op \in$

$[Read, Write]$ represents the memory operation type, $s \in [Exclusive, Shared]$ represents the transaction status globally in the TMSoC, and v is the memory operation value.

- **Transaction Record Container**: is a container for the eager version management. It is denoted as TR=$\{T_{n,i} \mid n \in [1,N]$ and $i \in [1,I]\}$, where N is the total number of cores in the MPSoC topology, n is the requester core ID, and i is the transaction ID.

- **Version Container Records** are represented as: $A = \{(m, T_{n,i}, vn, v, rv)\}$, where $m \in M$, $vn \in [1,VN]$ represents the version of the memory operation record, v is the memory operation value in the record, and rv represents the reference value that is used in case of a transaction rollback.

- **Transaction Version Container**: $V = \{A_{a,i} \mid a \in [0, address\_range]$ where $a = T_{n,i}.M.a_j$, and $i \in [1,I]\}$. The first term a represents the address of the shared memory operation based on its transaction and its requester core and i represents the transaction ID of the memory operation record.

A conceptual snapshot of the containers used by the MTC algorithms for three cores with three levels of criticality in a TMSoC is presented in Figure 4.3. It illustrates the direct connections and links between the tuples and entities. For simplicity, it is assumed that the algorithms have not performed any committing nor detected any conflict yet in this instance. The transaction record container TR denoted as *Tx Records* in the figure, creates linked lists that contain the received transactions of each core including their criticality level. The transaction records in this instance (cf. Figure 4.3) are represented in such a way to express the arrived transactions over time and the expected conflicts.

The transaction version container (V) shown in the Figure as *Tx version container*, creates a version container record (A) for each memory operation based on the processed address. The first record of each address contains the reference value rv of the processed address. This value has to be continuously updated to be used in case of a rollback as will be described in the next subsection.

From the instance in Figure 4.3 it is noticed that some transactions i.e. #21, #31 and #11, #22 also #34, #24 are transactions that are processing the same address (address #2, #1 and #6 respectively). Hence, conflicts have to be tracked and processed eagerly for these cases based on their criticalities. The relation between the latter two containers (i.e. V and TR) provides the knowledge required to detect conflicts and execute selective rollbacks based on the algorithms introduced later.

Memory transactions are processed by the MTC using the Algorithm 1, which defines the transaction processing procedure in the proposed transactional memory architecture. Multiple

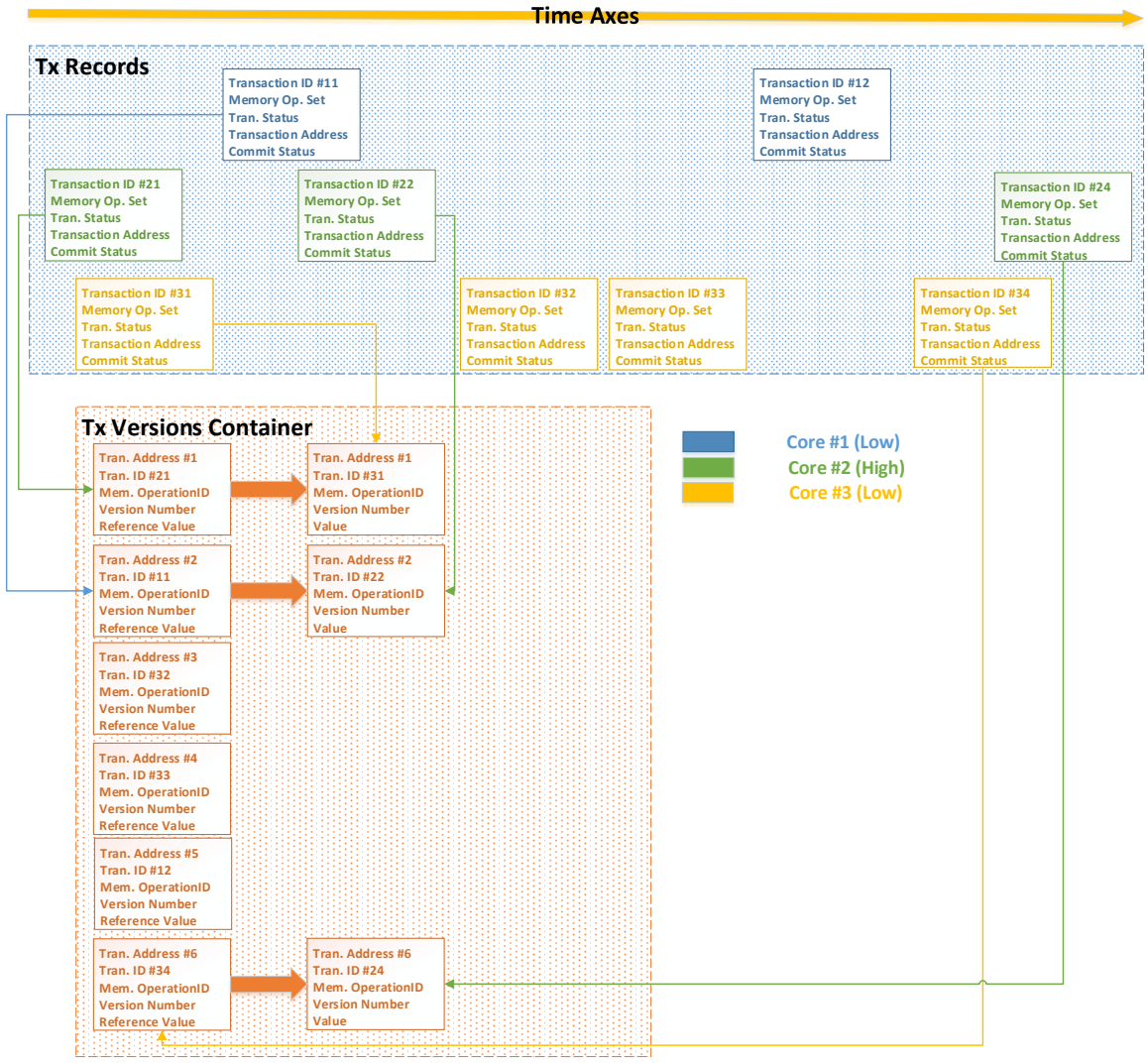

Fig. 4.3 Instance Representation of the MTC Containers

instances of this algorithm will run concurrently. Moreover, the handling of committing attempts, conflict detection and selective conflict resolution procedures are presented in Algorithm 2.

Transaction Processing Algorithm

As explained earlier, the MTC is responsible for processing the incoming memory transactions. The procedure "TransactionProcessing()" (cf. Algorithm 1) is called for each memory transaction ($T_{n,i}$), where n represents the requester core ID, and i is the transaction ID index.

The algorithm will repeatedly wait for incoming memory operations of the transaction before processing them. Meanwhile, it starts updating the TR container by adding a new

transaction record ($Tx_{n,i}$) to the corresponding core ID queue. The newly added transaction record contains the memory operations derived from the incoming transaction. The TR container stores the criticality level of the incoming transaction and it sets the committing status of the transaction record to $False$.

Afterwards, all incoming memory operations (M) of the pending transactions ($T_{n,i}$) have to be inspected for conflicts and processed accordingly. If the address a of the memory operation M is located in V then the corresponding records at this address will be stored in $\overline{A}$ based on the version container. $\overline{A}$ is a set that contains conflicting memory operation records.

Initially, for a newly destined address in the transaction version container V the algorithm will return an empty set ($\overline{A} = \emptyset$), and will enter the alternative statements (cf. Line 28). This case means that the specific address is accessed for the first time by a memory operation, and there are no conflicting operations up to now. Therefore, the status of this memory operation will be set to *Exclusive*, and a new V record is created (ar_i) at the address a_i with the index equal to the transaction ID i. The memory operation has to be processes based on its type. If it is a *READ* operation, the requested value of this operation is retrieved from the DDR memory to set the value and the initial reference value rv for memory operations of this transaction ID. Then the result is stored in the sending-buffers to send it to the corresponding requester core. On the other hand, if it is a *WRITE* memory operation, the incoming value of the write operation is used to assign $ar_i.v$. Additionally, the initial reference value $ar_i.rv$ is retrieved from the last committed value at this address in the DDR memory. Then, the new value ($ar_i.v$) is written into the DDR memory.

In case of a non-empty set $\overline{A}$ a number of memory operations are conflicting at this address according to the V records. At that point, a second check whether there is an existing V record of this specific transaction ID (i) is performed. If so, this indicates that we are already tracing the versions of the transaction i at this address. Moreover, the located V record contains the required data value (v) and reference value (rv) required for executing the memory operations (cf. Line 6).

In case a *READ* memory operation is processed, the read value is retrieved from the located memory operation (ar_i) and sent to the corresponding core through its sending-buffer. Otherwise, if a *WRITE* operation is processed then the write value is stored in $av_i.v$. Thereafter, the stored value is used to write in the DDR memory.

In the alternative case of the if-condition at line 13, the address a_i of the memory operation has been located in V but there is no record of the transaction i at this address. Therefore, a new record ar_i has to be created and added at the corresponding address of this transaction. The initial reference value at this address of the previous transaction k is assigned to the reference value ($ar_i.rv$) of the newly created record.

Algorithm 1 MTC Algorithms - Transaction Processing

```
1:  procedure TRANSACTIONPROCESSING(T_{n,i})
2:      AddToTransactionsRecord(TR, T_{n,i}, n, i)
3:      for all T_{n,i}.M.a | T_{n,i}.M.j = j ∧ j ∈ [1,J] do
4:          Ā ← locate records in V for a
5:          if Ā ≠ ∅ then
6:              if ar_i ∈ Ā | i = transaction ID i then
7:                  if T_{n,i}.M.op == READ| T_{n,i}.M.j = j then
8:                      send ar_i.v to Core
9:                  else
10:                     ar_i.v ← T_{n,i}.M.v | T_{n,i}.M.j = j
11:                     write ar_i.v to DDR
12:                 end if
13:             else
14:                 create new record ar_i
15:                 ar_i.rv = ar_k.rv | ar_k ∈ Ā
16:                 if T_{n,i}.M.op == READ| T_{n,i}.M.j = j then
17:                     ar_i.v = ar_k.rv
18:                     send ar_i.v to Core
19:                 else
20:                     ar_i.v ← T_{n,i}.M.v | T_{n,i}.M.j = j
21:                     write ar_i.v to DDR
22:                 end if
23:             end if
24:             for all ar_i do
25:                 ar_i.T_{n,i}.M.s ← SHARED | T_{n,i}.M.j = j
26:             end for
27:         else
28:             T_{n,i}.M ← EXCLUSIVE | T_{n,i}.M.j = j
29:             create new record ar_i | ar_i.a = a ∧ i = transaction ID i
30:             if T_{n,i}.M.op == READ | T_{n,i}.M.j = j then
31:                 ar_i.v ← v from DDR
32:                 ar_i.rv ← ar_i.v
33:                 send ar_i.v to Core
34:             else
35:                 ar_i.v ← T_{n,i}.M.v | T_{n,i}.M.j = j
36:                 ar_i.rv ← from DDR
37:                 write ar_i.v to DDR
38:             end if
39:         end if
40:         ar_i.m.op ← T_{n,i}.M.op | T_{n,i}.M.j = j
41:         ar_i.T_{n,i} ← T_{n,i}
42:     end for
43:     TryCommit(T_{n,i})
44: end procedure
```

Subsequently, if a *READ* memory operation is processed then the read value ($ar_i.v$) will be set based on the reference value of the transaction k to be sent later to the requester core. Alternatively, the *WRITE* value is saved as the value of the new record ar_i and then written into the DDR memory.

Eventually, the status of all memory operation records at the address $T_{n,i}.M.a_i$ has to be set to *Shared*. This step is of key importance for the committing decision in the next procedure, i.e. "TryCommit()".

Conflict Detection and Selective Conflict Resolution Algorithms

The procedure "TryCommit()" is called in order to commit a processed memory transaction $T_{n,i}$. A transaction can only commit if the status of all of its memory operations is set to *Exclusive*. This indicates that there are no conflicts at the addresses processed by this transaction, or conflicting memory operations were resolved and their status parameters were set to *Exclusive*. Therefore, the transaction $T_{n,i}$ can commit (cf. Line 14) and the committing status parameter cs is set to $True$. This parameter is used for cleaning the records.

If one of the memory operations of $T_{n,i}$ has a shared status, this means that it is conflicting with other memory operations at this address. Hence, the committing status parameter cs is set to $False$ and the procedure "ConfDetectionTracker()" is called for this memory operation in order to detect the conflict and resolve it (cf. Line 12).

The "ConfDetectionTracker()" procedure is responsible to locate conflicts and execute a so-called selective conflict resolution based on the criticality levels of the conflicting transactions. Conflicting memory operations with $T_{n,i}.M.a$ are located in V and listed in the set C (cf. Line 20).

As described earlier, the proposed architecture supports multiple criticalities. In the resulting set C, there are transactions of different criticalities. In order to resolve the conflicts with the listed transactions in C, these transactions of C have to be categorized into a subset of higher criticality transactions C_h and a subset of lower/equal criticality transactions C_{le}.

In case C_h is an empty set (cf. Line 23), this shows that the transaction $T_{n,i}$ is of a high criticality. As a result, all conflicting transactions will be listed in the C_{le} subset. Consequently, all transactions of C_{le} except for $T_{n,i}$ have to rollback. $T_{n,i}$ will be committed and its committing status cs will be set to $True$.

Alternatively, if C_h is not empty then transactions of C_h are of a higher criticality and they can try to commit later while transactions of the subset C_{le} including $T_{n,i}$ have to roll back (cf. Line 28).

In case the "Rollback()" procedure (cf. Line 31) is called for a transactions $T_{n,i}$, all memory operations of this transaction have to be located in V. The reference value of each

Algorithm 2 MTC Algorithms - Conflict Detection and Selective Conflict Resolution

```
    procedure TRYCOMMIT(T_{n,i})
2:      conf ← False
        p = 0
4:      for all T_{n,i}.M.s | T_{n,i}.M.j = j ∧ j ∈ [1,J] do
            if T_{n,i}.M.s == Shared then
6:              conf ← True
                p = j
8:              break loop;
            end if
10:     end for
        if conf == True then
12:         T_{n,i}.cs ← False
            ConfDetectionTracker(T_{n,i}.M.a, p)
14:     else
            commit T_{n,i}
16:         T_{n,i}.cs ← True
        end if
18: end procedure
    procedure CONFDETECTIONTRACKER(T_{n,i}.M.a, j)
20:     C ← locate conflict for T_{n,i}.M.a in V
        C_h = {t ∈ C | t.c > T_{n,i}.c}
22:     C_le = {t ∈ C | t.c ≤ T_{n,i}.c}
        if C_h = ∅ then
24:         Rollback(C_le − T_{n,i})
            commit T_{n,i}
26:         T_{n,i}.cs ← True
        else
28:         Rollback(C_le)
        end if
30: end procedure
    procedure ROLLBACK(T_{n,i})
32:     for all T_{n,i}.M.a | T_{n,i}.M.j = j ∧ j ∈ [1,J] do
            located ← Locate T_{n,i}.M.a in V
34:         store A_{located,i}.rv in DDR
            remove all A_{located,i}
36:     end for
        remove T_{n,i} from TR
38: end procedure
```

located memory operation has to be stored to the DDR memory and then the located record has to be removed from V. Eventually, transaction $T_{n,i}$, has to be removed from TR.

4.1.6 TMSoC Configuration

As discussed earlier, time-triggered messages impose resource reservations through a TDMA scheme. The timely block mechanism is realized as a time-triggered guarding time slot which blocks the transmission of any other conflicting messages. The timely blocking is ensured in the proposed architecture by the TTNIs and their time-triggered channels determined by the configuration files established at design time.

The configuration parameters of the TMSoC include the subsystem criticalities, the communication topology, the timing information and the messages paths. The proposed architecture supports different criticality levels, e.g. SIL according to IEC61508 [Bel06].

The proposed architecture requires two major configuration inputs. The first one is related to the external memory system specifications, which includes the memory initiation files to define the memory technology, block size and the transaction queue depth. Secondly, the TTNoC configuration includes the following information:

- Definition of the number of the cores, subsystems and criticalities.
- Static time-triggered configuration that defines two sets of communication types: a memory transaction related configuration and a message-based communication configuration.

Time-triggered messages are scheduled to be transmitted periodically. The configuration table defines the time-triggered message period, phase, message size, sending Buffer ID, sender core ID, number of receiver cores, receiver core IDs and the criticality. The message phase parameter defines the start transmission time in relation to the defined period. In addition to that, the static configuration defines the message size in bytes, the sending buffer IDs, the applications core IDs and the receiving cores.

In case of memory transactions, all cores have a scheduled timeslot to send transaction requests to the memory gateway. Moreover, the memory gateway has a dedicated time slot to send memory replies to each of the cores. Messages between cores can be defined with multiple recipients by modifying the *Number Of Receiver Core IDs* parameter and by listing the required core IDs in the *Receiver Core ID* field.

The scheduling and timing restrictions of the proposed memory architecture are reflected in the static configuration of the MPSoC framework. Scheduling anomalies are avoided when defining the system timing parameters based on the TDMA scheme. The scheduling

problem is complementary work and it is not in the focus of this thesis. On the other hand, the proposed architecture is compatible with exiting scheduling algorithms [PEP99].

4.1.7 WCET Analysis

The WCET analysis of the proposed chip-level transactional memory architecture is based on the WCET analysis presented in [MWU13] and [SBV10]. We determine the WCET recursively in an accumulated way. The WCET of a task depends on the execution time t_a of the atomic transactional memory sections and the execution time t_c outside of them. Depending on conflicts with other tasks, rollbacks can lead to the repeated execution of the transactional memory sections. We start by considering the critical instant where all tasks perform the memory transactions at the same time. If hp denotes the set of tasks with equal or higher criticality, then a given task T can be required to roll back for $|hp|$ times:

$$t_{wcet}^{(1)}(T) = t_c + t_a|hp| \tag{4.1}$$

Depending on the minimum interarrival time ($mint(\tau)$) of the tasks, they can arrive again during $t_{wcet}^{(1)}(T)$. Therefore, we need to recursively compute an accumulated time $t_{wcet}^{(x+1)}(T)$ of the task T:

$$t_{wcet}^{(x+1)}(T) = t_c + t_a \sum_{\tau \in ht} (\lfloor \frac{t_{wcet}^{(x)}(T)}{mint(\tau)} \rfloor + 1), \tau \in hp \mid criticality(\tau) \geq criticality(T)$$

The additional arrival of tasks with equal or higher priority depends on the *mint* of the tasks in hp. By dividing $t_{wcet}^{(x)}(T)$ by $mint(\tau)$ we obtain these additional task arrivals. The recursion of computing the WCET $t_{wcet}^{(x+1)}(T)$ is performed until there are no more changes in the iterations:

$$t_{wcet}^{(x+1)}(T) = t_{wcet}^{(x)}(T) = t_{wcet}(T) \tag{4.2}$$

$$\forall x \geq c, c \in \mathbb{N}$$

4.2 Hierarchical Transactional Memory Architecture for Distributed MCSs

As shown in the state-of-the-art analysis presented in Section 3.4, a system architecture based on a hierarchical transactional memory for both chip and cluster level is missing. This

section introduces the required system architecture (denoted from now on as Distributed Transactional Memory Architecture (DTMA)) and its hierarchical protocol (a.k.a Distributed Mixed-criticality Transactional Controller (DMTC)) which hides the heterogeneity of the system. This system architecture guarantees the real-time requirements and provides fault containment as will be illustrated in details later in this section.

The architecture of the DTMA shown in Figure 4.4, it consists of a number of networked MPSoCs that are interconnected through a reliable off-chip network. Moreover, the DTMA has a hierarchical memory architecture at all levels of the system, i.e., cores, gateways, external memory and the DMTC protocol. Each MPSoC of the distributed system architecture includes a number of cores, which are interconnected through a NoC. From now on, this MPSoC will be called a *node*. Moreover, all nodes are connected to a reliable off-chip network through their network gateways to establish the off-chip communication. The nodes' memory gateways are responsible for providing the transactional memory control, the memory controller services and the connection to the external memory.

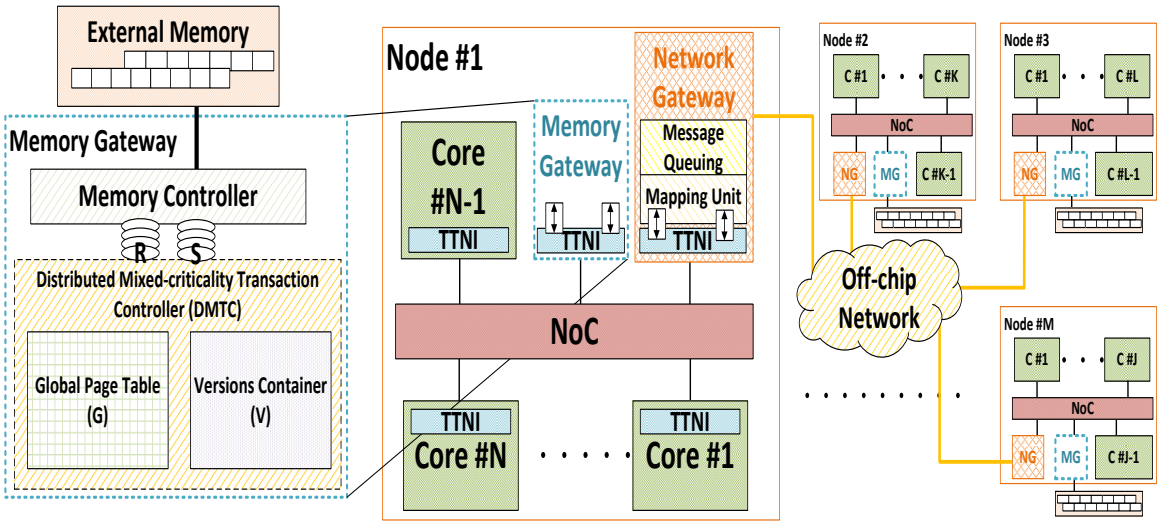

Fig. 4.4 The Distributed Transactional Memory Architecture (DTMA)

The system architecture supports message-based as well as shared memory interactions between a configurable number of nodes and cores. In order to understand the system architecture, first the nodes of the systems are analyzed, then the off-chip communication architecture is discussed.

4.2.1 Node Architecture

Each *core* of a node provides an application service, a memory service module and a network interface. *Application services* are realized either in hardware or in software. Subsets of

cores can define subsystems within the node that implement a specified task. Each core has a preassigned criticality, e.g. according to IEC61508 or ISO 26262. Each core has its own local cache, while the *memory service module* is responsible for generating the corresponding memory requests of the application and handling memory replies, especially in case of memory rollbacks. The application service and the memory service module support bidirectional communication via the *TTNI* of the on-chip network. We assume a time-triggered NoC in the architecture such as AEthereal or TTNoC within inherent fault isolation and temporal predictability.

Message-based and memory interactions are mapped by the TTNI to messages that are sent to the NoC based on pre-assigned time-triggered communication slots, where the communication is based on a global time base.

The proposed on-chip network can support different topologies (e.g., mesh, ring) and offers temporally predictable communication between the cores and the gateways using *time-triggered channels*. Those channels define virtual links between the senders and the receivers based on the configuration parameters of the NoC, which requires a priori knowledge of the network topology and the timing of the messages. These configuration parameters are the period and the phase of the messages, the size of the messages, the corresponding virtual link IDs, the sender cores and the receiver cores, and finally the criticality levels of the cores.

As noticed, the node architecture is very similar to the basic concepts of the TMSoC. Therefore, the reader can refer to Section 4.1 for more details. The major differences between the TMSoC architecture and the node utilized in the DTMA are the new memory and network gateway architectures that support the hierarchical DMTC protocol.

4.2.2 Off-chip Communication Architecture

A node's gateways are connected to TTNIs through different ports that are corresponding to the cores (cf. Figure 4.4). The *mapping unit* of the *network gateway* (NG) provides message redirection between the corresponding ports and message queues (cf. Figure 4.5. Additionally it performs protocol conversion for the incoming and outgoing messages, where messages are translated accordingly between the NoC message format and the off-chip message format. In addition, the *message queuing* service is responsible for serializing and handling messages in both directions based on their criticality, where messages with higher criticality are prioritized.

The *Memory Gateway* (MG) is connected to an *external memory* unit that provides a single or multi-channel memory system, where each channel has its own transaction queues. A memory gateway consists of the *memory controller* which is connected to the external memory, and the DMTC. The memory controller implements real-time memory gateway

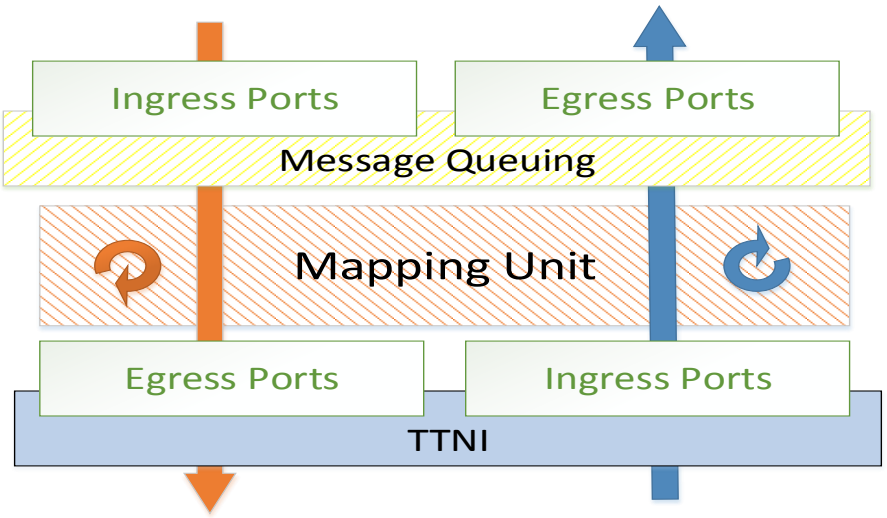

Fig. 4.5 Network Gateway Architecture of the DTMA

functionality that is responsible for providing temporal segregation and high throughput by defining memory access groups with known latency.

The scope of the *DMTC* is to support the presented system architecture with a hierarchical transactional memory while providing a criticality-aware conflict resolution. It is connected to the TTNI via ports, where each core has its own corresponding in-port and out-port in order to support fault isolation. Moreover, multiple sending and receiving queues are mandatory as the interface between the DMTC and the memory controller. Read and write memory operations are processed based on the received time-triggered messages, while memory replies and rollback instructions are redirected to the sending buffers to be sent to their requester source core, where a source core can also be the network gateway.

The DMTC is responsible for preserving atomicity across different levels by handling and managing address versioning and memory page exchanges between the different nodes and cores. It contains a version container, a global page table and a set of locally stored memory pages. The *version container* tracks and handles the versions of all addresses locally stored in this node for uncommitted transactions. The *global page table* is used to locate required pages within a node or at remote nodes. It is globally shared and synchronized based on the requests for memory pages in the hierarchical system. These modules of the DMTC are used by the algorithm in Section 4.1.5 to execute criticality-aware conflict resolution.

A *memory transaction* is an ordered set that has a *start* instruction, a set of memory operations, and finally a *commit* instruction. By committing a transaction, all changes of the related memory operations are written to the external memory. A memory operation can be *local* in case that the related memory page of the memory operation is located in the same

node. Otherwise, the memory operation is called a *remote* memory operation which means that the accomplishment of this operation requires an access to a remote memory page that is located in a remote node.

The DMTC uses a page-based memory synchronization, which means that the distributed system is handling and exchanging all memory data using fixed-size memory pages. The size of the page has to be defined at design time (e.g., 1Kb). Memory pages of the proposed distributed system have a unique ID ($page_{id}$) that is computed from the addresses. The *global page table* (G) is a 1-to-1 table that contains management information about the pages located in all nodes of the system including the external memories. This table has to be up-to-date and synchronized in case of page movement between the cores or nodes. The form of a record in this table is as follows $\{< page_{id}, node_{id}, core_{id} >\}$, where $page_{id}$ is the ID of the memory page, and $node_{id}$ represents the ID of the node that this page is located at currently. The core ID that is using this page currently is denoted as $core_{id}$. In case the page is located in one of the external memories the record has the following format $\{< page_{id}, node_{id}, ex\_mem >\}$, where *ex_mem* is the identifier of the external memory.

The *version container* (V) includes the local versions of all the addresses and their transaction IDs that are not yet committed within a node. This information is based on the list of pages that exist locally in the node, and the updates of the V container as will be described later. A record of V is as follows: $\{x, \{< version_1, Tx_a >, < version_2, Tx_b >, ..\}\}$, where x is the address of a memory operation processed at this node, then a list of the different versions of this address and their transaction IDs.

The hierarchical characterization of the proposed protocol is driven from its ability to handle the following five different cases of executing memory operations and their required memory pages:

- **Locally at the core**. If the required memory page is at the same requester core the page can directly be accessed.

- **Locally at the node**. In case the required memory page is at another core of the same node, the DMTC locates the page and the ownership of this page is obtained by the requester core in order to preserve atomicity and consistency.

- **Locally at the external memory**. If either of the above cases occurs, then the G table acts as a Translation Lookaside Buffer (TLB) allowing to search for a quick reference to the location of the page in the external memory of the same node. In case the page is located in G, then it will be fetched by the requester core and corresponding updates are performed to G and V.

- **Remotely at another core**. For memory pages located at cores of a remote node, the requested page is located in G and moved to the requester core. The requester core takes the ownership of this page, and the page's relevant records of that remote V container are moved to the local V container of the requester node.

- **Remotely at an external memory**. Pages located at the remote external memory of another node are detected based on the address space of each external memory. Thereafter, the requested memory page is moved to the requester core and the G table has to be updated.

The detailed operation of these cases in the distributed system with the use of the DMTC is described in the next section.

The proposed hierarchical transactional memory protocol builds on the system architecture, where this composition of the architecture and the transactional memory protocol provides: temporal predictability, fault containment at all levels of the system, heterogeneity of subsystems and support for diversity.

The configuration of the DTMA at node level is very similar to the configuration of the TMSoC (cf. Subsection 4.1.6). Additionally the off-chip configuration details are mainly related to the selected cluster level network and setup. As explained earlier in this section, the only restriction at off-chip network level is to adopt a reliable and predictable network.

4.2.3 Hierarchical Transactional Memory Protocol

A memory transaction (Tx) has a "start_transaction" instruction, then a set of memory operations, and finally a "commit transaction" instruction. Memory operations have to be handled differently based on their type (i.e. READ, WRITE). They are performed by a core involving access to memory at the same core or node, at the node's memory or at the memory in other nodes.

The DMTC's controlling mechanisms have to hierarchically guarantee the atomicity, consistency and isolation of the memory transactions at on-chip and off-chip levels. The DMTC executes conflict-detection algorithms and performs selective criticality-aware conflict resolution as described in this section.

The state machine illustrated in Figure 4.6 describes the stages of the transaction processing within the DMTC at each of the nodes of the proposed distributed architecture. The state machine waits until it receives a new memory operation (m) with address x. First it performs a look-up search in the global page table G in order to determine whether the address (x) of the memory operation m exists within in the requester core. If the page does not exist at the requested core, then it either exists at another core of the same node or remotely at

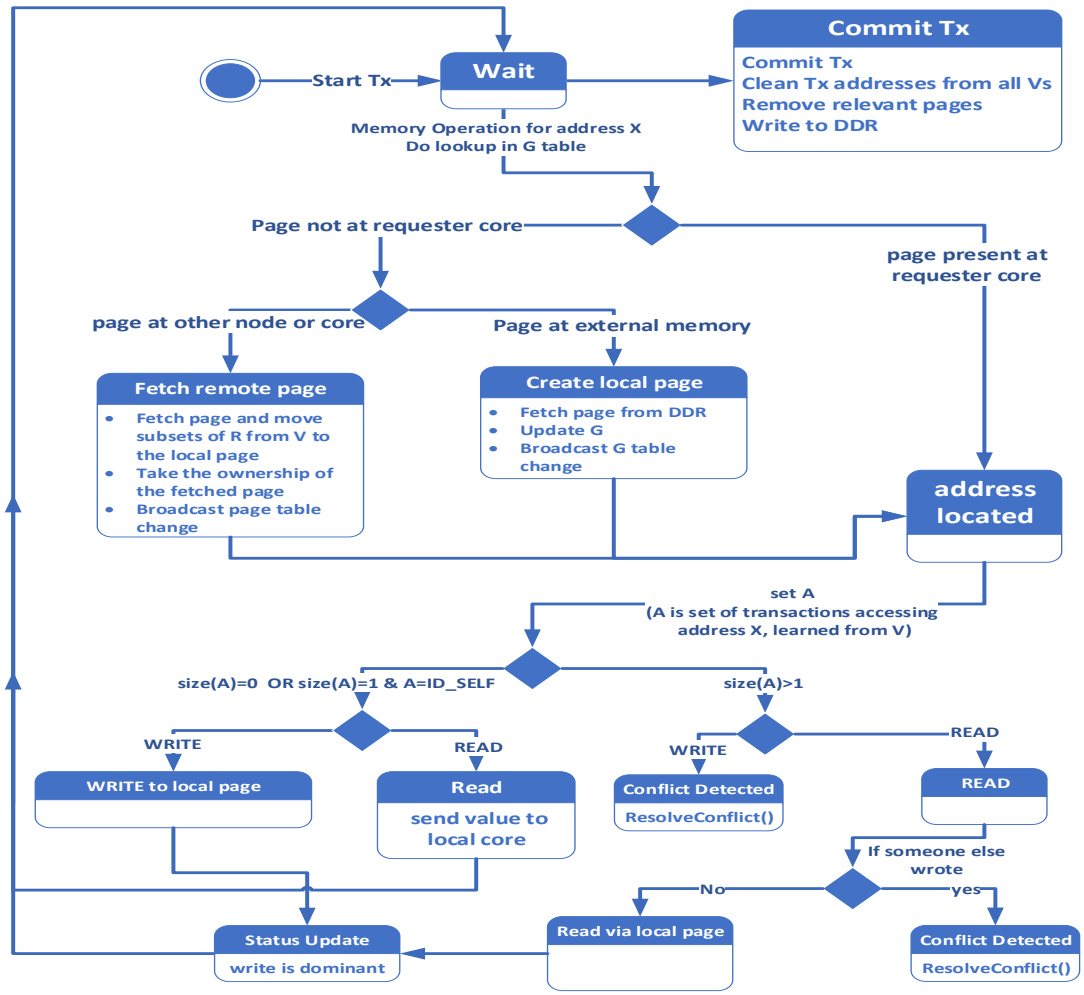

Fig. 4.6 Transaction Processing State Machine

another node. Otherwise, it is not created yet by any of the nodes. In this case the page has to be fetched from one of the external memories (DDR). It can be the local external memory or a remote one, where the targeted external memory is determined from the records in the *G* table. Thereafter, the *G* table has to be updated locally using the position of the newly fetched memory page. Later this update has to be broadcast to all other nodes in order to synchronize the *G* tables.

If the requested page exists at another core of the same requester node or remotely at another node, then the $node_{id}$ and $core_{id}$ are already known from the previous look-up step. The requester node will fetch the requested page from the remote core as well as the subset (*R*) of the fetched page's relevant records from that remote *V* container. In this way the requester node takes the ownership of this page, updates its ID and broadcasts this change to

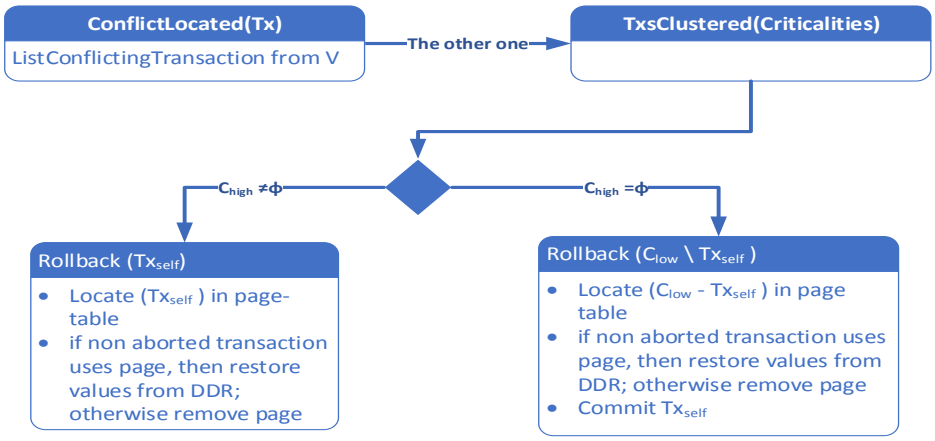

Fig. 4.7 Criticality Based Conflict Detection State Machine

all other nodes. Finally the requester node integrates the relocated version records into its local V container.

At this phase, there should be a local memory page (p) that contains the address x of the memory operation m. From the version container V the set A that contains all transactions that are accessing the address x and can result in contention are identified. In case the set A is of size zero which means that there are no other transactions accessing this address, or A is equal to one and this transaction ID is equal to the current transaction, there is no conflict. In case the type of the memory operation is *write*, m can be executed by writing the new value in the identified page p and V has to be updated. In case m is a *read* operation, then the value can be read from the memory page p and sent to the requester core.

If the set A contains more than one transaction (right part of the control statement, Figure 4.6), then the following situations occur. If m is a *write* operation then a conflict has to be handled, hence the "ResolveConflict()" function is called to trigger state machine #2 (cf. Figure 4.7), which will be explained later. In case that m is a *read* operation, we have to check whether someone else is trying to write at the same time, which can be determined using the V container. If not, then we read via the local page p and check whether V has to be updated. In case that m is a *read* operation and another transaction has written something at this address x, the second state machine has to be triggered to resolve conflicts. There might be the case that during the update of the V container, updates caused by read operations are executed at the same time by updates caused by write operations. In this case updates of write operations are always dominant.

Whenever the conflict detection mechanism is triggered by the first state machine for address x, the list of the conflicting transactions can be directly determined using the V container. As illustrated in Figure 4.7, these "conflicting" transactions are clustered based on their criticality level. C_{low} contains all the transactions with lower or equal criticality to the transaction Tx_{self} of address x. The second set, C_{high} consists of the transactions of higher criticality than the transaction Tx_{self} of address x.

According to the state machine (cf. Figure 4.7), there are two cases. In case that C_{high} is empty, this means that transactions listed in C_{low} have to rollback except for Tx_{self}. Therefore, the transaction's subset C_{low} without Tx_{self} is located in the G table in order to clarify if any of the non aborted transactions are using the page p. If not, then the values are restored from the DDR. Otherwise, page p has to be removed, and then transaction Tx_{self} commits as described in Figure 4.6. The commit process includes the cleaning of the V container, updating G and communicating this change, removing no longer used pages at the node, and finally writing Tx_{self} to the DDR.

In case C_{high} is not empty, then transactions of C_{high} have higher criticality and should not be be affected by the lower criticality. Thus, Tx_{self} has to rollback. Then the page related to Tx_{self} is located in the G table. If non aborted transactions are using this page then the values of the page p have to be restored from the DDR. Otherwise, page p is removed, V and G are updated and the rollback is communicated to the requester core.

4.3 Fault Hypothesis

The types and rates of failures that are tolerated in both TMSoC and DTMA are described in this section. Failure rates in the presented architectures for the safety-critical subsystems are assumed to be 10^{-9} for correlated failures.

Fault Containment Regions (FCRs) are distinguished in both system architectures in such a way to handle design faults and physical faults. Each component (i.e., core, memory or network gateway, node) is a Fault Containment Region (FCR) for design and physical faults. In this dissertation three types of failures are distinguished at the network, chip and cluster levels: timing failures, value failures and address failures.

Considering that TTNIs are used to interface each component in the architecture, failures in the *time domain* are contained. This is achieved for a faulty component since its TTNI is always aware of the predefined message transmission and reception times. These timing constraints are defined based on the a priori knowledge, and the synchronization on a global time base.

Value failures are in the responsibility of the application level developers who have to mask this type of faults (e.g., using active redundancy of application components).

The third type of failure is an *address fault*. This type of failures is contained due to the MMU-like functionality provided by the MTC, which contains the knowledge about the components that may access particular memory segments. Moreover, the global page table (G) declares memory segments of the components and can contain address failures by blocking memory pages of wrong addresses.

Chapter 5

Simulation Framework for Mixed-Criticality Chip Level Architectures

Numerous simulation tools are available in the literature, yet there is no support for transactional memory solutions at MPSoC level. Therefore, a novel SystemC/TLM-based framework was designed and implemented for this purpose to serve the TMSoC system architecture presented in Section 4.1. Initially, an implementation of a simulation framework for MPSoCs that support message-based and shared memory instructions is presented. Afterwards, the framework is extended to support transactional memory architectures.

In the following, available MPSoC simulation frameworks are compared with respect to their abstraction level, the ability for trace-based execution, the support for transactional memory and shared-memory simulation and support for safety-critical real-time applications.

SystemC is a widely used system-level modeling language for event-driven modeling [OG09, CMM+15]. Timing accuracy in SystemC is ranging from untimed to cycle-accurate where precise temporal specifications and SystemC modules can be simulated to validate the behavior of the platform. Additionally, Transaction Level Models (TLM) are used in the simulation frameworks to enhance the overall simulation speed [CG03, Ghe06]. TLM capabilities such as interface-based communication, blocking and non-blocking process structures, bidirectional and unidirectional transactions inspired researchers to develop simulation frameworks owing to the adaptability and accuracy that SystemC/TLM provides.

OCCN [CGLP04], Noxim [CMM+15] as well as NNSE [LTMJ05] are NoC simulators that provide a SystemC API enabling the creation of TLMs for a higher-level of abstraction between the SystemC-based modules. These simulators provide a customizable network topology (e.g., torus and mesh) as well as support for different routing and traffic distribu-

tion algorithms. Throughput, delays and power consumption measurements are supported. Another event-based NoC simulator is GARNET [AKsPJ]. GARNET is a cycle-accurate network interconnection model inside the GEMS [BBB+11] system simulation framework. It simulates flit-level buffering for five-stage pipelined routing with virtual channels as well as routing of flits with accurate memory and timing models.

Sniper [CHE11] is an x86 multi-core simulator based on interval simulation and the Graphite simulation infrastructure. It performs the timing simulation of an individual core without tracking the details of the instructions in the core's pipeline. The Sniper simulator supports flexible cache and memory configurations for multi-threaded workloads on computational cores and network-on-chip interconnections with the ability to execute benchmarks and generate traces. MacSim [KYL+] is another trace-based simulator that models micro-architectural behaviors including pipelining and memory subsystems. It supports multi-core chips containing computational cores with different Instruction Set Architectures (ISAs). Network interconnection models based on the IRIS [RCBB+12] network-on-chip simulator are available. Both Sniper and MacSim provide a power model based on McPAT [LAS+09].

Based on the latter discussion, a TLM-based simulation framework for deterministic time-triggered MPSoCs that supports message-based and shared-memory interactions with the ability to run trace-based and custom applications is not available. The presented SystemC/TLM-based simulation framework provides timing accurate simulations with high abstraction levels, taking into consideration assurances for safety-critical real-time applications based on the TMSoC system architecture. Furthermore, by analyzing the latter simulation tools, it can be noticed that benchmarks and traces are often used for simulating the application behavior of a system. However, existing transactional memory benchmarks (e.g., [MCKO08] and [GKV07]) do not support a hardware transactional memory at multi-core chip level. Therefore, a trace generation process that computes transactional memory trace files for MPSoCs is introduced in this chapter.

The realization of the MPSoC simulation framework was performed using SystemC/TLM. In addition, DRAMSim2 [RCBJ11] is used to simulate the external memory module. DRAMSim2 is a widely used cycle-accurate open-source DRAM simulator that models the memory controller, memory channels, ranks, banks and timing constraints [OO15], [KYL+]. In the following, the implementation of the simulation framework with a focus on the configuration and timing modules is discussed.

5.1 SystemC/TLM MPSoC

TLM is used in the framework to separate the details of the applications cores from their behavior at the TTNIs. Message-based communication can be expressed in TLM using channels, while transaction requests are managed using interfaces. All interfaces needed in the SystemC simulation framework are inherited from the TLM sc_interface class. In TLM, transactions can be defined as bidirectional or unidirectional. Examples of such transactions are message-based communication transactions of the proposed simulation framework and memory read/write transactions, correspondingly. Moreover, the TLM standard was used to manage blocking and non-blocking process structures (i.e., SC_METHOD and SC_THREAD).

Each core implements an *host processor* and a *TTNI*. In combination these two modules provide the communication behavior of a node as described in the following paragraphs.

Each *host processor* has an identification number and it belongs to a subsystem. The development of an host processor requires a communication interface, which abstracts from the implementation details of the application by providing procedures to send/receive messages to/from other cores. Likewise, this interface is responsible for mapping the outgoing messages to the time-triggered communication for custom applications. In the presented framework, this interface is also used to execute the trace-based simulation inputs. In this case, the definition of the traces is required for each of the host processors.

As defined earlier in the system model, the transmission of *time-triggered message* uses a priori knowledge of the period, phase, message size, sender node ID and receiver node IDs. The formed time-triggered messages in the TTNI are transmitted based on a predefined configuration, in which the time-triggered messages are scheduled to be transmitted periodically. The phase of the message defines the start time with respect to the start of the period. The message size is determined by the message payload, and the routing requires the source and the destination node IDs of the message. In addition to that, each message obtains a sequence number that can be used for delay and power calculations.

As illustrated in Figure 5.1, the user of the framework defines the number of the simulated *subsystems* based on the application-specific setup and the clustering of host processors into subsystems. Each subsystem has an identification number and can be labeled with a criticality level based on, e.g., ISO 26262 or simply *HIGH* and *LOW* criticality.

The TTNI handles the incoming time-triggered messages according to the predefined periodic transmission schedule. This message handling avoids message contention and provides predictable transmission times, bounded delays and minimal jitter.

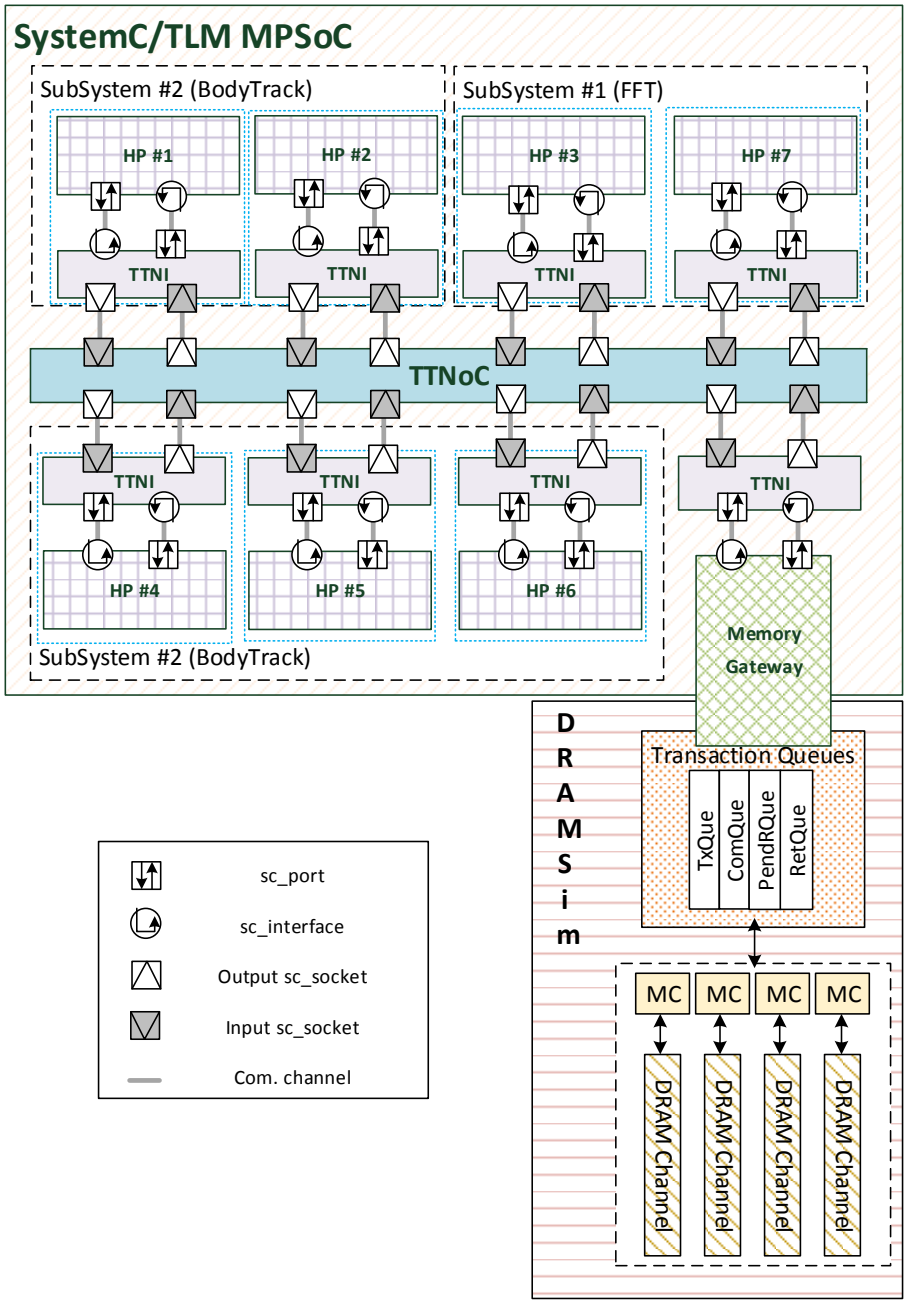

Fig. 5.1 Simulation Framework - Implementation Model

In case of *trace-based host processor* the following functions are used. The "createTTMessage()" function is executed periodically based on the schedule. This function is responsible for sending messages at the predefined time-slots. If a shared-memory interaction is planned, then the host processor reads and provides the corresponding memory transaction in the "createTTMessage()" function in order to create a time-triggered memory transaction. Thereafter, in the TTNI, a cyclic dispatcher method is called to determine if a time-triggered message has to be sent according to the predefined schedule. The upcoming transmission of a periodic message is calculated with the following equation:

$$(Period \times Index) + Phase \tag{5.1}$$

Received messages at the TTNI are queued into receiving queues based on the message type. If the received message is a memory request it will be sent to the external memory service queue in order to update the scratchpad, otherwise it will be sent to the host processor queue. The host processor is aware from its configuration information that it will receive a message at a predefined time so it will invoke its "receiveMessage()" method to retrieve it from the TTNI queue.

Each *host processor* has a TLM port and an interface to establish the communication with the TTNI (and vice versa) as depicted in Figure 5.1. If a message has to be transmitted from the host processor to the TTNI, an interface method of the host processor is called on the corresponding port at the TTNI. Consequently, this interface method invokes and triggers the related functions at both sides to start the transmission. When all messages are transmitted, the execution sequence will return to the caller module. The advantage of the port-to-interface structure is that the overall simulation speed is significantly increased.

The simulation module for the *TTNoC* contains the time-triggered table and the communication channels. The communication table is one of the most important configuration parameters. It is based on the physical layout and topology of the TTNoC. The table is generated based on the static configuration parameters of the TTNoC and loaded at the beginning of the simulation. The TTNoC uses time-triggered channels that represent the temporal and spatial allocation of physical links of the simulated TTNoC.

The constructor of the simulation class for the TTNoC is responsible for creating the overall structure of the MPSoC. The number of nodes N connected to the TTNoC is a configurable parameter that is given to the TTNoC as an input. The generated MPSoC has $N+1$ TTNIs, namely N cores and one memory gateway. The total number of the required time-triggered channels is set according to the static configuration parameters of the TTNoC.

Based on the time-triggered communication table, the instantiated nodes of the previously mentioned process will periodically call the transmit message function "trmMessage()" of the

time-triggered channel to transmit their messages. This call is done through the corresponding port and socket.

TLM socket based connections are used to simulate the connection of the TTNIs to the TTNoC. Principally, the socket-based connections and the port-to-interface connections have a similar way of communication. Both use transactions for data transmission. However, the use of sockets has the following advantages. On one hand, the communication within the node's modules (i.e., host processor, TTNI) uses custom TLM interfaces while the TTNoC uses predefined TLM interfaces to the TTNIs. On the other hand, the technical layout of the connections between TTNIs and the TTNoC is reduced by using TLM sockets, since it is possible to bind sockets directly to each other without the need for export-calls to forward incoming/outgoing time-triggered messages. In addition to that, the use of TLM sockets improves the hierarchical structure of the implemented framework for further enhancements in the future.

The implemented TLM sockets at the TTNoC level can simulate communication delays that represent the router delays of a real MPSoC. The TTNoC simulation of a concrete topology depends on the configuration and schedule of the use case. The topology and the number of routers in the TTNoC and the possible accumulated delays of the routers need to be taken into account.

To receive a message at the core, a sequence of function calls is triggered in the related objects, starting from the TTNoC sockets and ranging to the receiving queues of the TTNIs. The sc_interfaces is used to send the message to the host processor, which will call the receive message function "rcvMsg()".

5.2 DRAMSim2 External Memory

DRAMSim2 [RCBJ11] is used in this simulation framework to simulate the external memory module of the MPSoC. It is widely used for MPSoC simulations, e.g., in the MACSim and Sniper simulators. DRAMSim2 provides dynamically configurable memory models and a simple connecting interface that allows it to be connected to other simulators. It is also possible to use a trace-based execution mode (i.e., TraceBaseSim functionality) in DRAMSim2 to perform trace-based memory simulations.

DRAMSim2 has a simple interface that allows it to be connected to other frameworks. An extension of the DRAMSim2 interface was developed using SystemC/C++ in order to connect it to the memory gateway of the proposed architecture. A TLM/SystemC based interface was established, where incoming memory instructions from the memory gateway are redirected to the corresponding memory controller of the external memory and vice versa.

The *memory gateway* of the framework is inherited from the host processor class. It can send and receive the memory transactions and it is responsible for mapping the memory transactions from the memory transmission format to the DRAMSim2 instructions. The main functionality of the memory gateway is to initiate the DRAMSim2 memory based on the configuration parameters of the simulation setup and to provide the required interface from DRAMSim2 to the TTNoC. The memory gateway is responsible for interfacing the DRAMSim2 memory transaction API calls to establish the memory transaction functionalities of the MPSoC. Transaction replies are handled accordingly, where the returned data of a read transaction is sent to the corresponding core. This requires a set of reply-queues in the memory gateway, namely one for each host processor. The TTNI is responsible for fetching the corresponding reply message at the predefined timeslot according to the schedule to send it.

The configurations of the external memory module are loaded in the memory gateway to define the number of the memory channels, ranks, and banks of DRAMSim2. Each channel has its own memory controller and its transaction queues as follows:

- The transaction queue 'TxQue' receives and stores incoming transactions.
- The queue 'ComQue' stores the translated commands of each transaction.
- If a read command is dispatched to the memory, then the transaction will be stored into the queue 'PendRQue' until the data is returned.
- The queue 'RetQue' is used to store the returned transactions.

Finally, the presented simulation framework requires two configuration inputs derived from the description provided in Section 4.1.6. One input is for the DRAMSim memory specifications including the initialization files to define the memory technology, block size and the transaction queue depth. DRAMSim2 provides memory configuration templates that can be used or modified according to the simulation requirements. Secondly, the TTNoC configurations includes the definition of the number of the cores indicating their criticalities, the memory gateway and the static time-triggered schedule.

5.3 The Mixed-Criticality Transaction Controller (MTC) Implementation

This section discusses the implementation of the system architecture presented in Section 4.1 with a focus on the transactional memory support and the implementation of the Mixed-Criticality Transaction Controller (MTC). The implementation of the transactional memory

extensions is based on the simulation framework presented in the previous section. Therefore, the major implementation building blocks (i.e., *host processor*, *memory service module*, *TTNI*, *TTNoC*) are similar to the building blocks described in Section 5.1. The *memory gateway* module on the other hand, is extended to implement and support the MTC architecture and algorithms.

The memory gateway contains the following building blocks: the TTNI of the gateway, the MTC that provides the transactional memory services and the memory controller to the DRAMSim2-based external memory. The MTC delivers *eager version* control and *eager conflict* detection. This means that updates are performed in place while keeping records. Moreover, conflicts are detected at each read/write memory operation.

The MTC module is responsible for handling incoming and outgoing memory transactions. Incoming memory transaction are processed by the "transactionProcessing()" procedure based on the received time-triggered messages. Moreover, transaction replies are mapped and queued in the corresponding destination sending-buffers in order to be transmitted by the gateway's TTNI. The MTC procedures are used in combination with the *VersionContainer* and *TransactionRecord* modules to perform the desired mixed-criticality transactional memory handling.

To simplify the description, it is assumed that each node represents a subsystem with its predefined criticality level. In our $N+1$ node architecture, the transaction record container would have N rows, namely one for each node. Transactions with memory operations labeled as *exclusive* are conflict-free transactions, while transactions with memory operations labeled as *shared* are those which can introduce conflicts. Conflicting transactions are stored based on their memory addresses in the transaction version container. This container is used to process conflicting transactions based on their criticalities as described in the MTC algorithms.

Generated transaction instances are logged in the transaction record along with their criticality. Furthermore, the transaction versions of the generated transaction are checked based on the information in the transaction container and the transaction version container in order to track the transactions.

5.4 Trace Generation Process

In this section, the process for the generation of trace files is presented. Benchmarks are typically used for the evaluation of MPSoCs. However, there is an interfacing gap in the use of shared-memory benchmarks and MPSoC simulations. Therefore, a generation process was introduced to create the required input traces for benchmarks and realistic example applications. The generated trace files have to be compatible with the host processor of the

MPSoC and its DRAMSim2 external memory. The developed simulation framework adopts SPLASH-2 and PARSEC application benchmarks [BKL08, BKSL08], which are widely used for the design and evaluation of shared-memory multi-core systems.

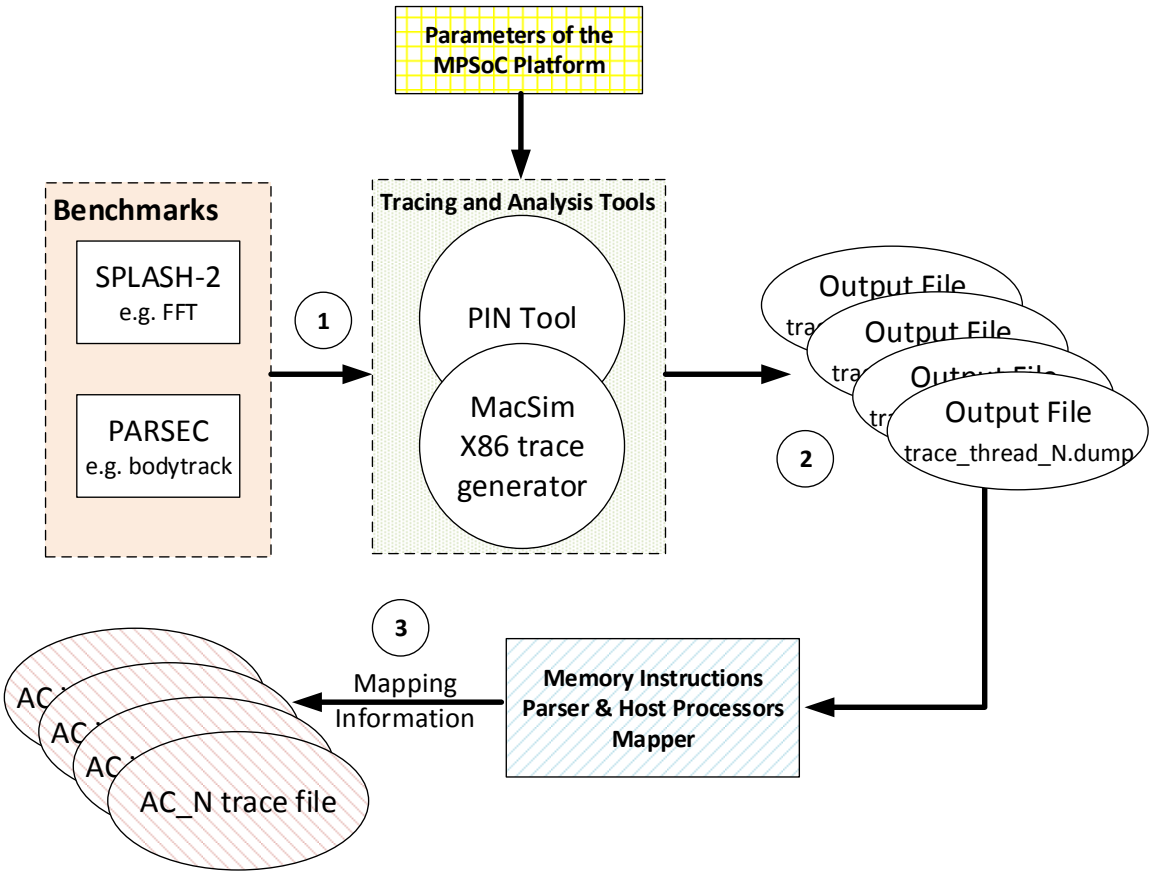

Fig. 5.2 Benchmarks Trace Generation Process

The generation of the trace files starts by defining use cases and the number N of the simulated cores of the MPSoC. N also determines the number of the threads used in the benchmark execution. As shown in Figure 5.2, after the definition of the application benchmarks (step 1), they are used as inputs for the tracing and the analysis tools. The x86 trace generator of MACSim in combination with the PINTool [kLCM+05] is used to create output dumps of the chosen application benchmarks (step 2). PINTool is a dynamic program analysis tool for run-time instrumentation. The execution of step 2 requires the definition of the parameters of the MPSoC including the number of the simulated threads, which has to be identical in steps 1 and 2. The outputs of step 2 are N trace-dump files that contain all trace instructions of the selected applications, one for each thread.

Afterwards, the trace-dump files are parsed and transformed into an MPSoC/DRAMSim compatible format by the *memory-instruction parser and host processors mapper*. The

resulting output files are the MPSoC application trace files that will be used in the host processors (step 3), one for each host processor. An instruction of a memory transaction has the following format:

$$< memory\_address, operation, delay > \tag{5.2}$$

The *memory address* indicates the requested memory address. The supported *operations* in the simulation framework are *read* and *write*. Finally, the instruction delay in cycles is specified. The delay is specified relative to the previous memory instruction including the accumulated execution times of any non memory instructions in between.

Chapter 6

Simulation Framework for the Hierarchical DTMA

The realization of the proposed distributed system architecture with the hierarchical DMTC protocol is presented in this section. Based on the system architecture introduced in Section 4.2, SystemC/TLM is used for the protocol and node implementation. VEOS [V 314] is used for simulating a FlexRay bus for the off-chip network and the cores of the MPSoC. Finally, DRAMSim2 [RCBJ11] serves for simulating the external memories of the MPSoCs. Combining multiple simulation tools requires the introduction of a coordination process that is responsible for synchronizing their time and data exchange. A co-simulation technique presented by the author and et al. in [UOO15] is used to accurately coordinate VEOS and multiple SystemC-based nodes.

The VEOS environment is the *dSpace* software for the simulation of AUTOSAR software and physical environment models on a host PC. VEOS allows the simulation of AUTOSAR Electronic Control Units (ECUs) and their application behavior using virtual validation scenarios. Using the VEOS player, simulated cores can be integrated into a simulation system and its execution on the VEOS simulator can be controlled. In addition, the experimental tool ControlDesk can access the VEOS simulator for testing the AUTOSAR software [OUOA16].

The VEOS environment can integrate other simulation tools, such as Simulink, in the simulation system for representing the physical environment.

6.1 Implementation

The proposed distributed system architecture is mapped to a synthetic distributed automotive system as shown in Figure 6.1. The distributed system consists of two MPSoCs that are connected through a FlexRay bus. Each node has its own on-chip communication schedule,

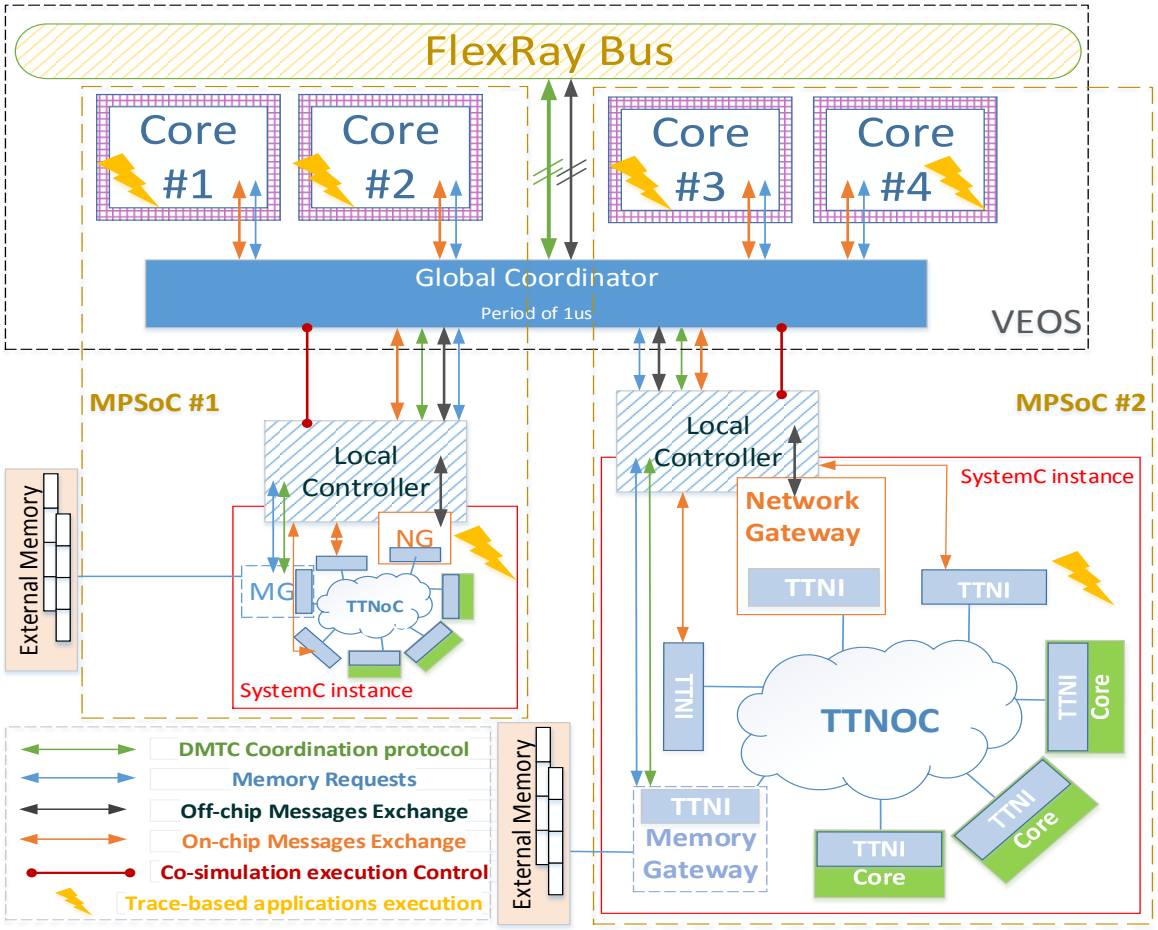

Fig. 6.1 VEOS-SystemC Implementation for a Distributed System Architecture.

while a separate schedule is defined for the off-chip communication. The coordination between the simulation tools is performed by a TCP-based interleaving execution process that is managed through *local controllers* at each node in SystemC and by a *global coordinator* at VEOS level, where a number of data and control flows are defined between the simulation tools. Each node can have a configurable number of cores that are interconnected through a deterministic TTNoC [OESHK08]. Cores are simulated either as SystemC-based or VEOS-based host processors. Each core runs a trace-based application that is assigned to it. The trace files are generated based on the process presented in [OO15]. As described in Section 4.2, the nodes' gateways provide both off-chip gateway functionality and memory gateway functionality including the DMTC protocol.

In this work, we extend the node implementation presented in [OO15] by integrating VEOS-based host cores to the nodes, implementing an off-chip network gateway (NG) for the off-chip distributed communication, and a memory gateway (MG) that serves the hierarchical DMTC. The NG provides off-chip gateway functionality for periodic communication based

on [AOOM15]. In addition to DMTC, the proposed memory gateway assumes a compositional real-time memory controller [AGR07a] that uses a predictable arbiter responsible for scheduling memory access groups dynamically in order to guarantee the allocated bandwidth and the maximum latency bounds.

The cores execute their assigned application trace-file that includes message and memory operations. Sending and receiving messages at core level is controlled by the TTNI of each core based on the defined communication schedule. Messages might be addressing cores within the same node (cf. arrows colored in orange, Figure 6.1) or cores at other nodes (arrows colored in black). On-chip messages between VEOS-based cores and SystemC cores are sent based on the on-chip schedule to the local controllers to be redirected within the node to their destination core through the TTNoC. Additionally, off-chip messages are collected and mapped in the node's gateway and handled according to the off-chip schedule.

The interleaving execution of the two simulation tools allows one simulation tool to execute for one microsecond per simulation step, then messages, memory and control requests are delivered to the other simulation tool to starts its turn of execution for one microsecond. The execution step of one microsecond can be changed according to the use-case granularity.

Memory operations are sent by a node (cf. colored in blue, Figure 6.1) to the memory gateway, while the DMTC coordination protocol requests (arrows colored in green) are sent between the nodes in order to execute the different memory synchronization requests and page exchanges of the hierarchical protocol as described in Section 4.1.5. It has to be mentioned that the off-chip messages and the DMTC coordination requests are sent though the gateways to the local controllers, and then to the global coordinator whenever the execution control is given to VEOS in order to deliver the messages to the FlexRay bus and then to their destination core or node and vice versa.

The AUTOSAR tool SystemDesk is used to define a set of cores that are configured based on the AUTOSAR architecture with extended communication modules for their simulation as host processors in an MPSoC. These host processors are integrated to a simulation system to be run by the VEOS platform.

During a VEOS simulation, an AUTOSAR Operating System (OS) is emulated for a PC-based simulation of the VEOS-based cores. This AUTOSAR OS invokes the OS tasks and function calls. SystemDesk is used to define AUTOSAR Software Components (SWCs) as the application layer of the cores in order to integrate and execute the trace files. Periodic tasks with different phases are configured to be performed by the OS of each core, where the period and the phases of the tasks are set according to the on-chip communication schedule of their corresponding SystemC node. Thus, before the generation of the simulation

model a trace-file application is manually integrated in each SWC, and each Run-Time Environment (RTE) of a core is modified to allocate the reading function of the trace-file application in the defined OS tasks.

In addition, the FlexRay bus simulation is performed by the mentioned global coordinator. As for the on-chip schedule in the VEOS-based cores, a set of tasks is assigned to the global coordinator according to the off-chip communication schedule. Based on this schedule, a period and a different phase is assigned to each task for sending off-chip messages between the two SystemC nodes.

6.2 Co-simulation Coordination

An integrated global coordinator and local controllers use TCP/IP for the communication between the simulation tools. The coordinator and the controllers serve for the synchronization of the AUTOSAR simulation in VEOS with the NoC simulation in SystemC. Additionally, they are responsible for correctly redirecting the message exchange between each VEOS-based core and its corresponding TTNI to the TTNoC. The local controllers support the gateway functionality of the on-chip/off-chip communication. Moreover, the global coordinator is implemented in VEOS as the server of the TCP/IP communication and the local controllers located at each of the SystemC-based nodes are TCP/IP communication clients.

A minimum interrupt detection latency is assumed for the synchronization and the exchange of data between the different simulation systems. The granularity for the execution steps is determined by the interrupt detection latency which is typically higher than $1\mu s$. Therefore using $1\mu s$ as a standard resolution for the execution control guarantees the accuracy of the simulation.

VEOS implements and simulates a FlexRay bus as described in section 6.1. Moreover, a global coordinator is responsible for synchronizing the AUTOSAR simulation with multiple instances of the SystemC-based nodes. The global coordinator has send and receive *buffers* for each of the nodes, where messages are stored in order to be sent later based on the off-chip schedule. The co-simulation uses the $1\mu s$ for the execution synchronization and the data exchange between VEOS and the systemC-based simulation nodes. Every $1\mu s$ an accumulated message is exchanged between the two simulation tools containing on-/off-chip messages, memory operations and DMTC coordination messages. This $1\mu s$ synchronization means that node buffers might be empty in some cases and this requires sending *empty* messages to maintain the execution synchronization.

A message exchanged between the simulation systems includes the following elements:

- *Header:* This field indicates whether the message represents either an on-chip message, an off-chip message or DMTC coordination message.
- *Type:* This parameter is used to distinguish between the different types of the DMTC coordination messages.
- *Status:* Indicates whether the message is empty or not. In case the message is not empty, the number of data and memory operations contained in the message is denoted.
- *Sender ID:* Contains the ID of the VEOS-based core sending the data or a memory operation. The destination core is known from the on-/off-chip communication schedules.
- *Payload:* Contains the data of the memory operations.

In case of the DMTC, coordination messages are messages exchanged between the different DMTCs in the distributed system. The protocol's synchronization and control messages are sent in the message payload, while the other fields of the memory structure are known based on the co-simulation and the communication schedule.

The global coordinator is part of the interface between each VEOS-based core and the corresponding TTNI at the SystemC node. Once a $1\mu s$ simulation step is performed by VEOS, the AUTOSAR simulation is paused. Thus, in case of any existing memory operation from the VEOS-based cores, this is forwarded as an on-chip message to the corresponding SystemC simulation instance where the VEOS-based core is mapped, otherwise the on-chip message is configured to be empty.

Additionally, an incoming off-chip message from each SystemC node is received by the global coordinator. The global coordinator provides two buffers for queuing off-chip messages from each SystemC node. In case one of the two received off-chip messages is not empty, the data is stored in the buffer available for the specific SystemC node in order to be sent through the FlexRay bus simulation based on the configured schedule. Moreover, once the off-chip messages from the SystemC nodes were received and processed by the global coordinator, this resumes the VEOS simulation and the next $1\mu s$ simulation step can be performed in the AUTOSAR simulation.

Besides the global coordinator in VEOS, each local controller in the SystemC nodes constitutes an important part of the coordination. The local controller is defined as a main task controlling the execution of the SystemC-based host processors, the memory gateway and the TTNoC based on the $1\mu s$ time steps. Once an on-/off-chip message from the global coordinator is received by the local controller the type of the message is verified. In case of an on-chip message, it is redirected to the corresponding TTNI of the VEOS-based core. In

case it is an off-chip message, it is mapped and analyzed by the node's network gateway to be processed correspondingly. Finally, in case it is addressing the memory gateway then it is redirected to that gateway in order to be processed by the DMTC as described in the earlier sections.

Chapter 7

Evaluation and Results

The proposed MCS architectures with support for transactional memory and selective rollback based on the criticalities are evaluated in this chapter. In the following, three evaluation scenarios are introduced and discussed based on the experimental results.

7.1 Evaluation of MCS Framework for Message-based and Shared Memory Interactions

7.1.1 Use-cases Description

A use case served for the verification and evaluation of the simulation framework. The MPSoC includes seven cores with a TTNoC and two subsystems (cf. Figure 5.1). The first subsystem is running a Fast Fourier Transform (FFT) application and the second subsystem executes a body tracking application. Moreover, DRAMSim2 is configured according to Table 7.1.

Subsystem#1 has nodes N4 and N6 (cf. Figure 7.1), which run the FFT application [BKL08] of the SPLASH-2 benchmark. FFT is an algorithm used in signal processing

Parameter	**Value**
Memory technology	DDR3_micron_64M_8B_x4_sg15
Main memory size	8GB
Block size	64 Bytes
Transaction queue depth	512

Table 7.1 DRAMSim2 Use Case Configuration

to compute the discrete fourier transform and its inverse. In the FFT, *M* identifies the total number of transformed data points and *P* denotes the number of processors used in the benchmark.

Subsystem#2 runs the body-track application on the remaining five cores (i.e., N0-N3 and N5) as depicted in Figure 7.1. Bodytrack [BKSL08] is used in computer-vision algorithms for video surveillance, character animation and automotive safety functions. It employs a model of the human body to detect a person being shown in multiple video streams. It tracks a marker-less human body using an annealed particle filter to track the pose using edges and the foreground silhouette as image features. The tracking is based on a 10 segment 3D kinematic tree body model. Bodytrack searches high dimensional configuration spaces without relying on any assumptions or marks. The problem size of the bodytrack benchmark is based on four frames and 4,000 particles.

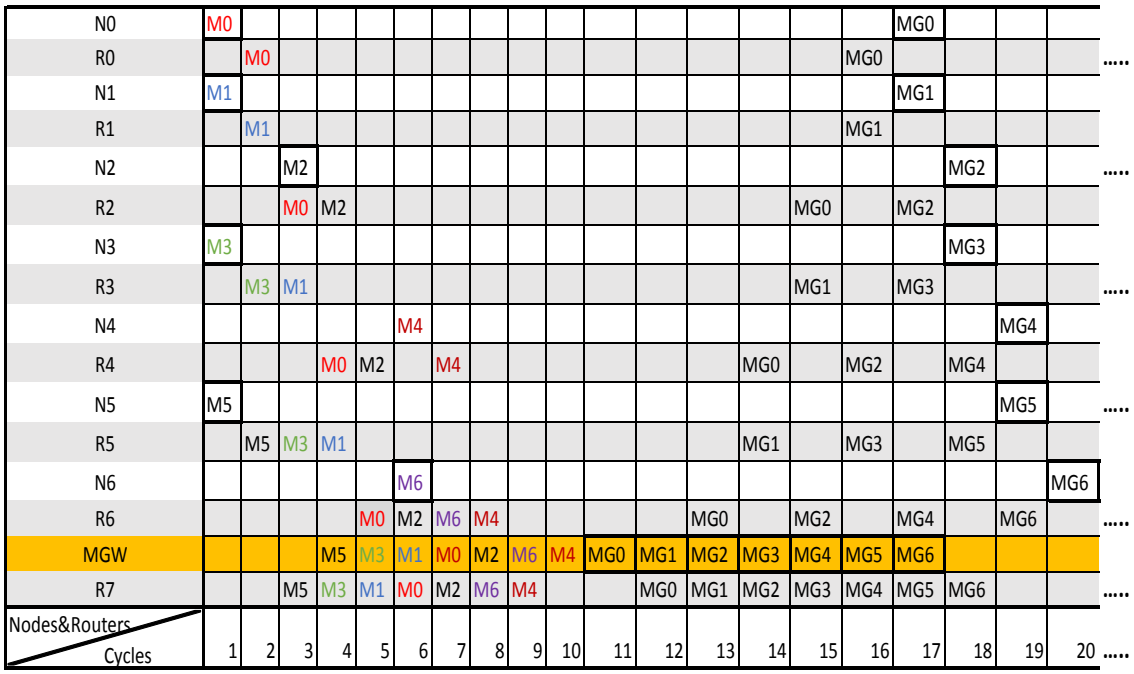

Fig. 7.1 Shared-Memory Instruction Schedule of the Use Case

It is assumed that the TTNoC topology of the simulated MPSoC in this use case is a mesh with eight routers (cf. Figure 7.1). The static time-triggered configuration schedule and the simulation parameters of the MPSoC are set based on the discussion in Section 4.1.6. The communication schedule of the use case is defined based on the previous assumptions. Figure 7.1 illustrates the scheduling procedure of the use case. The vertical axis of the table represents the nodes, routers and the memory gateway (MGW), the horizontal axis

shows the required cycles. M# indicates the shared-memory requests and MG# indicates the shared-memory responses.

The message routes are defined based on the routers that are used to reach the destination nodes. These routes have to avoid collisions in the temporal and spatial domains. For instance, a shared-memory request from node #0 to the memory gateway is defined as the route {N0,R0,R2,R4,R6,R7,MWG} and it takes seven cycles. Similarly, the routes of the remaining shared-memory instructions are defined. According to the schedule, the shared-memory requests need ten cycles to finish the transmission to the memory gateway and the memory gateway needs twenty cycles to send back its replies to all the nodes.

In addition, a custom message-based application is defined to be executed as part of the use case on the top of the TTNoC. This application creates message-based traffic between the cores $N0, N1, \ldots, N6$. The scheduling of this application follows the same procedure as described earlier.

Based on the schedule, the time-triggered configuration table is defined for the message-based application and the shared-memory application benchmarks as shown in Figure 7.2.

	Period [ns]	Phase[ns]	MsgSize	SendBufferID	VL_ID	SenderCoreID	NumberofReceiver Cores	Receiver CoreIDs
Shared-Memory Requests	20	0	1	0	0	0	1	7
	20	0	1	1	1	1	1	7
	20	2	1	2	2	2	1	7
	20	0	1	3	3	3	1	7
	20	5	1	4	4	4	1	7
	20	0	1	5	5	5	1	7
	20	5	1	6	6	6	1	7
Shared Memory Responses	20	10	1	7	7	7	1	0
	20	11	1	8	8	7	1	1
	20	12	1	9	9	7	1	2
	20	13	1	10	10	7	1	3
	20	14	1	11	11	7	1	4
	20	15	1	12	12	7	1	5
	20	16	1	13	13	7	1	6
Message-Based Node-to-Node	50	10	9	14	14	1	3	2,4,5
	50	10	9	15	15	2	4	0,3,1,5
	50	15	9	16	16	3	3	0,1,2
	50	20	9	17	17	4	3	2,4,5
	50	15	9	18	18	5	4	0,3,6,1
	100	30	9	19	19	6	5	0,1,2,3,4
	100	35	9	20	20	0	1	4
	100	45	9	21	21	2	6	0,1,5,3,6,4
	100	55	9	22	22	5	1	2
	100	85	9	23	23	1	4	2,0,4,5
	100	90	9	24	24	3	3	0,2,6

Fig. 7.2 Configuration Table for Message-based and Shared-Memory Access of the Use Case

NodeID	Trace File	Number of Memory Transactions	Total Instructions Delay (cycles)	Node Termination Time (ms)
0	Bodytrack Node#0	331547	670778	0.0234
1	Bodytrack Node#1	1843	2537	0.0341
2	Bodytrack Node#2	1869	2595	5.3830
3	Bodytrack Node#3	3065	872	4.1444
4	FFT Node#4	43122	125691	0.0231
5	Bodytrack Node#5	376810	658588	0.7187
6	FFT Node#6	292896	571055	5.5258

Table 7.2 Overview of the Instruction Delays and Overall Completion Time in the Use Case

7.1.2 Results and Discussion

The use case was executed on a 64-bit Linux PC with two i7 cores running at 2.1 GHz clock speed. The simulation was executed for 1 and 10 milliseconds (simulation time) and the corresponding real execution times were 8 minutes and 75 minutes respectively. The size of the simulation output files were 659 MB and 6.7 GB.

The output of the simulation framework includes the DRAMSim power calculations, the delays of the memory instruction and the overall completion time of the benchmark. Table 7.2 summarizes these results and also shows the total number of the benchmarks' memory instructions for each node. These numbers depend on the use case and the trace generation process. Moreover, the accumulated delays (in cycles) of the memory transactions in relation to the non-memory instructions of each node are illustrated, which provides insight on the effect of the computational instructions on the memory-related delays. Finally, the completion time of the benchmark in nanoseconds is listed in the last column of the table. It can be noticed that all benchmark applications terminated within 5.526 milliseconds.

7.2 Evaluation of TMSoC

7.2.1 Use-cases Description

A conceptual automotive use case serves for the evaluation of the proposed MTC, and allows the comparison of the WCET analysis from Section 4.1.7 with simulation results. The use case is divided into two main tasks and several applications (cf. Figure 7.6). A vehicle is equipped with a pedestrian-detection mechanism, e.g., to detect a critical condition of children running into the street. Meanwhile, the vehicle is executing a video streaming

service based on the x264 video encoding library [BKL08]. As described in Figure 7.6, the camera of the pedestrian-detection mechanism captures frames that are processed by the system, where the processing procedure includes the following algorithms: a computer vision algorithm called bodytrack (BT), a fast fourier transform (FFT) and a simple noise removal (NR) algorithm [BKL08]. The FFT and the NR filters are used to enhance the quality of the frames. Later, the processed output is provided to the BT algorithm for detecting the pedestrians. In case a pedestrian is detected, the vehicle's braking system is notified to trigger the automated braking, and an alarm indication is displayed to the driver.

The criticality of the applications is based on ISO 26262 [asi11]. The criticality of the FFT and the NR filters is ASIL B, while the BT application has the highest criticality ASIL D. On the other hand, the video streaming service x264 is not critical (ASIL QM).

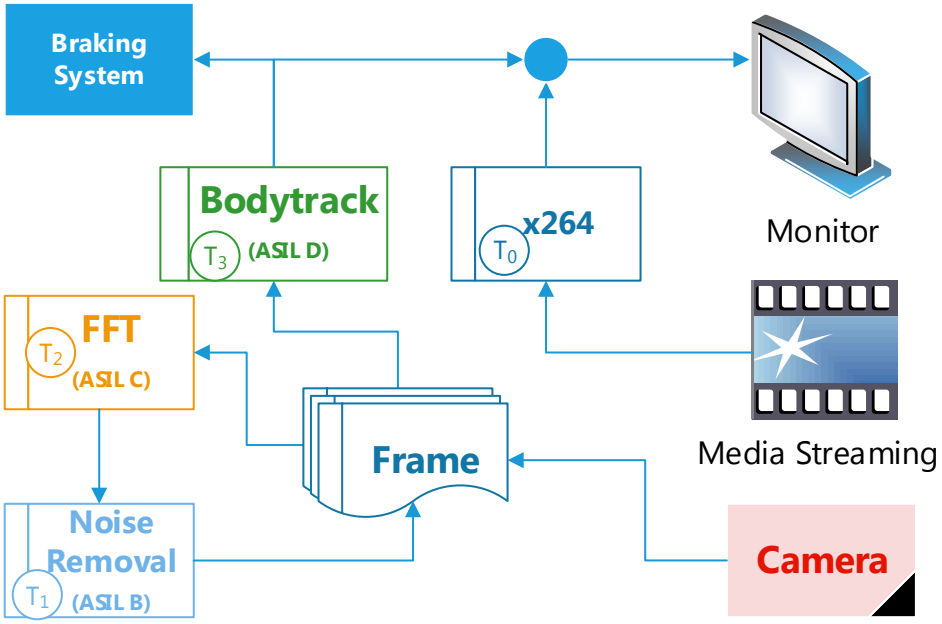

Fig. 7.3 Automotive Use case Scenario

For the evaluation purposes of this work, the transactional memory architecture and the MTC algorithms have been developed as extensions of the simulation framework previously developed in [OO15] to support a transactional memory and the MTC algorithms. This framework uses cycle-accurate systemC/TLM [Ghe06] and DRAMSim2 [RCBJ11] for the simulation of the proposed transactional memory architecture and its MPSoC setup. For the use case trace-files have been generated and used as input for the cores. The MPSoC consists of 8 cores connected to eight routers of a mesh-based NoC, namely 7 cores and one memory gateway. The applications are distributed as follows: 4 cores for the BT algorithm,

and one core each for the FFT, NR and the x264 algorithms. The DRAMSim2 based external memory configuration was set to 8GB micron DDR3 with a transaction queue depth of 512.

In order to understand the overall distribution of the memory transactions and the original memory operations in the use case, Figure 7.4 shows an external ring that represents the percentage of the memory transactions processed in the use case by each application in regard to the overall transactions, while the internal ring represents the percentage of the total memory operations for each application in regard to the overall operations. It is clear that the major load of the transactions originates from the BT application, while the remaining applications share 49% of the transaction load in the MPSoC.

7.2.2 Results and Discussion

Although the frame processing can result in memory conflicts, the BT application is of the highest criticality and therefore the execution and memory interactions of this application must be guaranteed. Thus, the MTC algorithms process the memory transactions, detect conflicts and handle them based on their criticality level. Moreover, the critical section represents 21.8% of the memory operations of the FFT, 14.3% of the NR, and 22.5% of the BT application.

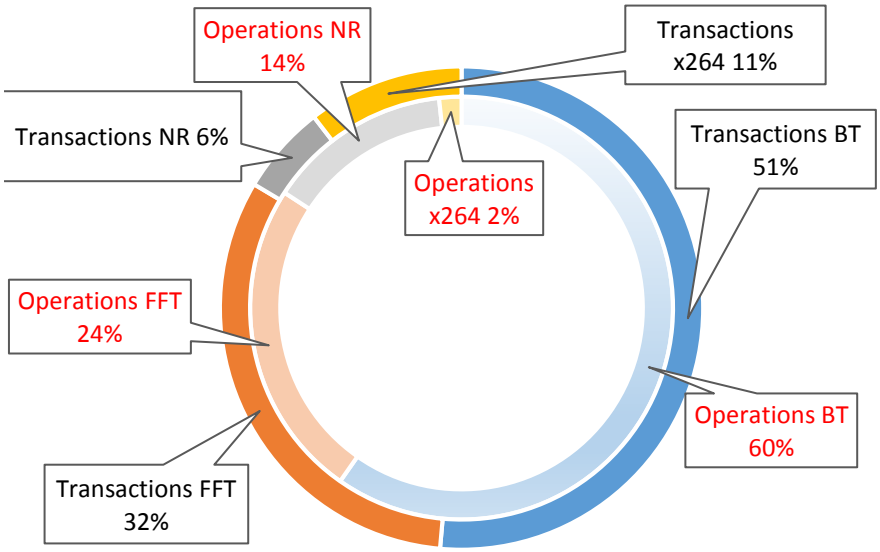

Fig. 7.4 Mem. Operations vs. Transactions Per Application

Figure 7.5 presents the execution time of the use case for each of the seven cores based on their applications. The use case was executed with and without the selective rollback algorithms of the MTC. In case no MTC is used, the transaction execution and the

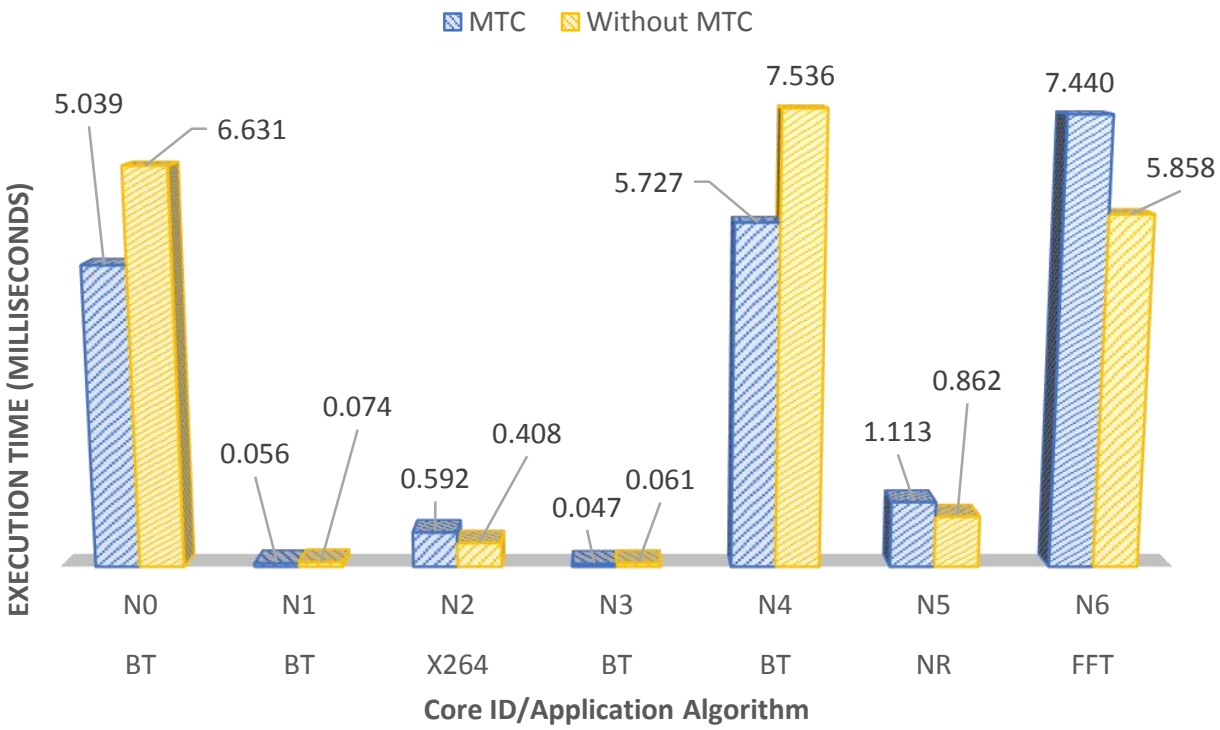

Fig. 7.5 Execution Time per Core

stalling/rollback decisions are taken based on the order of the transactions' commit operations at the memory controller.

As shown in the use case execution results, the performance of the higher criticality BT application has been improved by 24% with the use of the MTC algorithms. Additionally, the performance of the FFT and NR applications has decreased by 27% and 29% accordingly in comparison to the BT, and the execution time of the non safety-relevant x264 application has been worsened by 44%.

Furthermore, the use case serves for the evaluation of the WCET analysis presented in the Section 4.1.7. The automotive use case contains five cores each hosting one task (cf. Figure 7.6). Task T_0 has the lowest criticality ASIL QM, while tasks $T_1, \ldots, T_3$ represent different Automotive Safety Integrity Levels (ASIL) [asi11].

The overall estimated execution time t_e and the *mint* values of the tasks are illustrated in Table 7.3. Using equation 4.2, the WCET is calculated for all tasks. It is noticed that T_0 does not have a bounded WCET, and tasks T_1 to T_3 required several iterations (in comparison to the other tasks) in order to bound their WCET, which usually depends on the parameters of each task in equation 4.2.

Table 7.3 WCET of the ASIL tasks

Task ID	Criticality	t_e	t_c	t_a	$mint(\tau)$	$wcet$
T_0	ASIL QM	4500μs	4275μs	225μs	1ms	∄
T_1	ASIL B	750μs	643μs	107μs	2ms	1.18ms
T_2	ASIL C	1400μs	1095μs	305μs	0.6ms	7.39ms
T_3	ASIL D	2570μs	1992μs	579μs	0.9ms	5.97ms

Consequently, the results in Figure 7.5 are inline with our WCET analysis, and this proves that the proposed architecture and its MTC are prioritizing the memory transactions of the higher criticality tasks within bounded and predictable time. This behavior is guaranteed at the three levels of the transactional memory architecture.

7.3 Evaluation of the Hierarchical DTMA

7.3.1 Use-cases Description

A synthetic automotive use case serves for the evaluation of the proposed architecture and its hierarchical DMTC protocol. The use case represents a pedestrian-detection mechanism (PDM) running in parallel with an audio-video streaming system in a vehicle. The PDM uses frames captured by a camera, where the system is required to process these frames using computer vision algorithms to detect possible pedestrians on the way. Additionally, noise removal and transformations are applied to the frames in order to enhance the accuracy of the results. In case a pedestrian is detected, the vehicle's braking system is notified to trigger the automated braking, and an alarm indication is displayed to the driver.

The criticality of the applications is set based on the automotive ISO-26262 functional safety standard [asi11]. The use case consists of the following four applications: The bodytrack (BT) computer vision algorithm serves for detecting the pedestrians, to which the highest criticality level is assigned (ASIL D). The second application is the Fast Fourier transform (FFT) with criticality level ASIL C and the third application is a noise removal (NR) algorithm with criticality level ASIL B. These two applications are used to enhance the quality of the captured frames. Meanwhile, the vehicle is executing an audio/video streaming service based on the x264 video encoding library which is not critical, denoted as ASIL QM. The higher criticality level application should not be affected by the lower criticality ones in order to avoid accidents that might cause loss of life.

The previously mentioned applications have been executed using *Simlarge* input-sets of the PARSEC benchmarks [BKL08] in order to generate the trace-files for the distributed

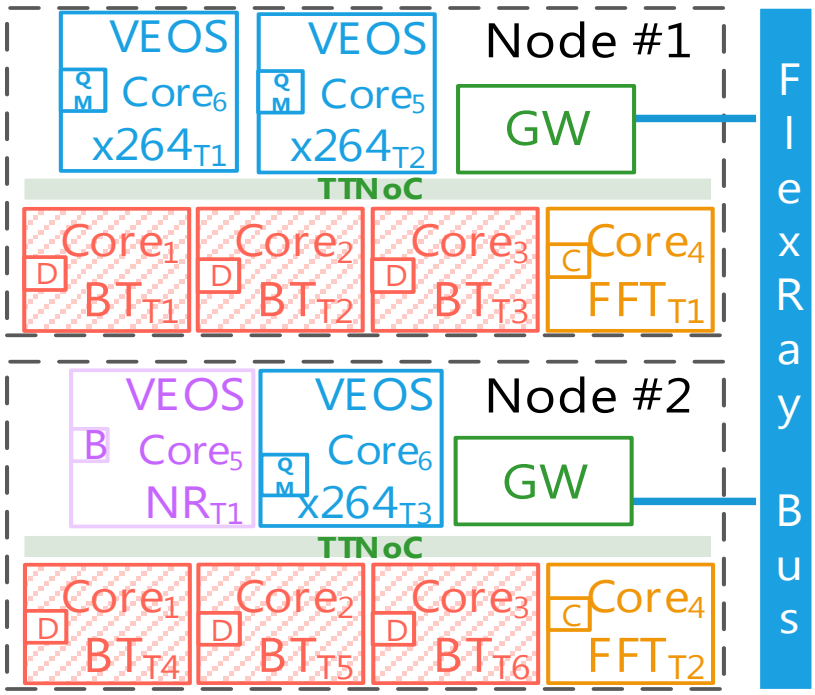

Fig. 7.6 Distributed Automotive Use-case.

system cores. The trace files of the applications are generated accordingly for 12 cores, and the use case is distributed in such a way to run on two nodes where each node consists of six cores. Each node has an external memory of 4GB micron DDR3 with a transaction queue depth of 512. The configuration parameters are defined in the initiation phase of the DRAMSim2 instances.

The distribution of the trace files and the criticality levels were assigned in the system-C/VEOS levels as follows: three cores at each node are running the BT trace files with criticality level ASIL D, and one core at each node runs the FFT trace file with criticality level ASIL C. At VEOS level, node#1 has two VEOS-based cores that run the x264 trace files with no criticality. Node#2 has one VEOS-based core that runs the NR trace file with criticality level ASIL B and another VEOS-based core runs the third non critical x264 trace file. Finally, the subscript at each application name given as T# (cf. Figure 7.6) represents the trace file identifier of each core.

Both on-chip and off-chip communication schedules are configured based on the a priori knowledge of the use case. Different periods and phases are assigned to each core that is running the trace files at both SystemC-based and VEOS-based cores. Likewise, the schedule for the off-chip communication is configured using a period of 1ms and a phase of 200μs for messages sent from node#1 to node#2. The phase of messages sent from node#2 to

node#1 is set to 300μs. Additionally, messages between VEOS-based cores and their node are exchanged based on the co-simulation steps of 1μs.

Node #1						
CoreID	Core1	Core2	Core3	Core4	Core5	Core6
Criticality	ASIL D	ASIL D	ASIL D	ASIL C	QM	QM
# Trans.	349	6	7	431	60	5
Conservative	79	1	2	94	11	5
FOW	12	2	1	7	7	3
DMTC	5	0	0	11	11	5
Node #2						
CoreID	Core1	Core2	Core3	Core4	Core5	Core6
Criticality	ASIL D	ASIL D	ASIL D	ASIL C	ASIL B	QM
# Trans.	9	608	11	434	127	7
Conservative	2	137	2	95	18	6
FOW	1	12	0	14	4	5
DMTC	0	9	1	9	6	6

Fig. 7.7 Number of Rollbacks Performed per Core.

It is important to mention that the size of the trace files, the total number of the transactions and the critical sections for each application are different. This depends on the input-sets of the benchmark execution and their results at the trace-file generation phase. The size of the critical sections has a direct relation to the number of conflicts in each application, as will be shown in the results.

7.3.2 Results and Discussion

The execution time of a core is calculated from the difference between the starting time of the application until the time at which the trace file has finished its execution, which means that all messages and memory operations of the trace file have been successfully executed. The use case was evaluated toward the number of rolled back transactions and the execution time of three different conflict resolution scenarios. First, the so-called *conservative* conflict resolution rolls back all conflicting transactions. These transactions can be re-executed later with a random minimal delay. Second, the *First One Wins* (FOW) conflict resolution allows the first transaction to commit while rolling back all other conflicting transactions. Finally, the *DMTC* protocol executes selective criticality-aware conflict resolution as described in the earlier sections.

The framework requires sixteen minutes to simulate each second of the real execution time of the use case. The results illustrated in Figure 7.7 are compared to the total number of the transactions at each core (given in row *# Transactions*). The conservative conflict resolution has the highest numbers of rolled back transactions in comparison to the other two executions. This is due to multiple attempts of the cores to execute their rolled back transactions. Moreover, this has resulted in longer execution times independently from the criticality levels.

The results illustrated in Figure 7.7 have shown that the FOW execution has reduced the number of rollbacks per application in comparison to the conservative execution as follows: 87.4% for the BT application, 88.9% for the FFT application, 78% for the NR application and 31.8% for the x264 application. On the other hand, the DMTC execution has reduced them by 93.3%, 89.2%, 67% accordingly, while the x264 application had the same number of rollbacks as in the conservative execution. It can be clearly noticed how the DMTC protocol considers the application criticalities in its rollback decisions.

Node #1						
CoreID	**Core1**	**Core2**	**Core3**	**Core4**	**Core5**	**Core6**
Criticality	ASIL D	ASIL D	ASIL D	ASIL C	QM	QM
Conservative	5.024	0.078	0.078	5.507	**0.321**	0.037
FOW	2.554	0.071	0.049	4.042	**0.317**	0.021
DMTC	2.370	0.037	0.037	**5.929**	**0.424**	0.063
Node #2						
CoreID	**Core1**	**Core2**	**Core3**	**Core4**	**Core5**	**Core6**
Criticality	ASIL D	ASIL D	ASIL D	ASIL C	ASIL B	QM
Conservative	0.129	**10.452**	0.157	**12.818**	**1.010**	0.032
FOW	0.088	**7.322**	0.103	**6.297**	**0.237**	0.045
DMTC	0.052	**5.430**	0.174	4.840	**0.540**	0.054

Fig. 7.8 Execution Time per Core for both Nodes.

Figure 7.8 compares the execution time between the cores using the conservative, the FOW and the DMTC protocol executions, where the results are given in seconds. The results show that the performance of the BT application using the DMTC protocol has been improved by 48.05% in comparison to its performance using the conservative execution and 25.84% in comparison to the FOW execution. The FFT execution time was improved by 53.74% in related to the conservative execution and 6% in comparison to the FOW execution. In case of the ASIL B application it can be noticed that both FOW and DMTC execution lead to an improvement. Since the DMTC execution is prioritizing the higher criticality BT and FFT applications in comparison to the NR application it is expected that the execution

time in this case is larger that the FOW execution. As explained earlier, the FOW execution is agnostic towards criticality levels of the application. Consequently it handles the x264 application in a similar way as the higher criticality applications. This has resulted in the improvement of the execution time of this non critical application in relation to the other two executions. The performance of the non critical application was increased in case of the DMTC execution favoring the higher criticality applications.

Generally, it can be noticed that the improvement of the execution times using the DMTC protocol is in the same order of magnitude as the FOW execution. In contrast to the FOW, however, the DMTC protocol ensures that the execution time of a core with high criticality does not depend on the behavior of cores with lower criticality.

Chapter 8

Conclusion

Many embedded systems based on multi-cores combine applications and subsystems of mixed-criticality. Transactional memory support in such systems is desirable to provide atomicity, consistency and isolation guarantees. The prevention of any effect of low criticality subsystems on the temporal behavior of subsystems of higher criticality is a prerequisite for modular safety arguments in mixed-criticality systems.

Based on the analysis of the state-of-the-art presented in Chapter 3, two transactional memory-based architectures were presented and evaluated in this thesis, i.e. a chip-level and a distributed cluster-level architecture (cf. Chapter 4).

The proposed predictable Transactional Memory System-on-a-Chip (TMSoC) architecture and its Mixed-Criticality Transaction Controller (MTC) ensure that the validation and certification of high-criticality subsystems do not depend on subsystems with lower criticality. The MTC algorithms provide concurrency control and conflict detection mechanisms in addition to a selective rollback. Both safety-critical and non safety-critical transactions can be executed, but the MTC avoids timing effects of non safety-critical tasks on safety-critical ones. The rollback of a transaction is performed in case higher criticality subsystems would be affected, and the WCET of a task only depends on transactions of tasks with the same or higher criticality.

The proposed Transactional Memory System-on-a-Chip (TMSoC) architecture was evaluated using a SystemC/TLM-based simulation framework to execute a synthetic automotive use case. The used trace-based simulation framework provides early validation and design space exploration of mixed-criticality systems. This simulation framework provides a configurable and flexible MPSoC model. Also, an adjustable external memory system was integrated into the simulation framework based on DRAMSim2. Moreover, it introduces a high abstraction level using SystemC/TLM to increase the overall simulation speed. This level of abstraction in combination with the trace-based simulation can hide the computational

details and the micro-architecture of the simulated cores. In addition to custom application support, widely used application benchmarks can be used in the simulation environment. As shown in Section 7.1 and 7.2, the presented solution provides fault isolation and handles efficiently three levels of temporal predictability (i.e. transaction level, interconnection level and memory gateway level).

Distributed memory solutions have previously been presented either for on-chip or off-chip systems. Despite the benefits of the transactional memory as a lock-free solution, existing transactional memory solutions are mainly investigating the on-chip level. Moreover, the requirements presented in Section 3.1 are not investigated in distributed systems with off-chip communication networks.

A hierarchical transactional memory protocol that serves a deterministic mixed-criticality distributed architecture is presented (cf. Section 4.2). It uses a DMTC protocol that hides heterogeneity of the system, it offers a high level of predictability and reliability assurances at both on-chip and off-chip levels, and provides fault isolation for mixed-criticality applications.

The proposed protocol has been evaluated using two time-triggered networked nodes connected though a FlexRay bus. The simulation framework has been developed using SystemC and VEOS. The coordination between the two simulation tools is based on TCP. DRAMSim2 is used to simulate the external memories of the distributed system. A synthetic automotive use case for pedestrian-detection is presented for the evaluation of the architecture and its protocol, in which the cores of the two nodes are sharing the execution of four applications with different criticalities.

The results (cf. Section 7.3) have shown that the proposed hierarchical solution provides efficient, predictable and reliable support for transactional memories at different integration levels. It also ensures a bounded execution time of safety-critical applications and guarantees their independence from applications with lower criticality levels while coexisting and sharing the memory resources on the same distributed system.

References

[AG11a] B. Akesson and K. Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[AG11b] B. Akesson and K. Goossens. *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*. Embedded Systems. Springer New York, 2011.

[AGR07a] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable sdram memory controller. In *Hardware/Software Codesign and System Synthesis , 2007 5th IEEE/ACM/IFIP International Conference on*, Sept 2007.

[AGR07b] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable sdram memory controller. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2007 5th IEEE/ACM/IFIP International Conference on*, pages 251–256, Sept 2007.

[AKsPJ] Niket Agarwal, Tushar Krishna, Li shiuan Peh, and Niraj K. Jha. Garnet: a detailed onchip network model inside a full-system simulator. In *in Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2009*, pages 33–42.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[Amo04] Albert Amos. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Embedded World '04, pages 235—-252, 2004.

[AOOM15] M. Abuteir, R. Obermaisser, Z. Owda, and T. Moudouthe. Off-chip/on-chip gateway architecture for mixed-criticality systems based on networked multi-core chips. In *Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on*, Oct 2015.

[asi11] Road vehicles-functional safety-part 9: Automotive safety integrity level (asil)-oriented and safety-oriented analyses. *ISO 26262-9:2011, ICS: 43.040.10*, 2011.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[BC11] Chaitanya Belwal and Albert M. K. Cheng. Lazy versus eager conflict detection in software transactional memory: A real-time schedulability perspective. *Embedded Systems Letters*, 3(1):37–41, 2011.

[Bel06] Ron Bell. Introduction to iec 61508. In *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*, SCS '05, pages 3–12, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[BKL08] Christian Bienia, Sanjeev Kumar, and Kai Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In David Christie, Alan Lee, Onur Mutlu, and Benjamin G. Zorn, editors, *IISWC*, pages 47–56. IEEE, 2008.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[BNZ08] Lee Baugh, Naveen Neelakantam, and Craig B. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA*, pages 115–126. IEEE Computer Society, 2008.

[BP11] Antonio Barros and Luís Miguel Pinho. Software transactional memory as a building block for parallel embedded real-time systems. In *EUROMICRO-SEAA*, pages 251–255. IEEE, 2011.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition, 2011.

[CCP+15] B. Cilku, A. Crespo, P. Puschner, J. Coronel, and S. Peiro. A tdma-based arbitration scheme for mixed-criticality multicore platforms. In *Event-based Control, Communication, and Signal Processing (EBCCSP), 2015 International Conference on*, pages 1–6, June 2015.

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling: An overview. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '03, pages 19–24, New York, NY, USA, 2003. ACM.

[CGLP04] Stephane Curaba, Miltos D. Grammatikakis, Riccardo Locatelli, and Francesco Papariello. Occn: a noc modeling framework for design exploration. *Journal of Systems Architecture*, 2004.

[CGS+14] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A hybrid transactional memory for haswell's restricted transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, New York, NY, USA, 2014. ACM.

[CHE11] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, New York, NY, USA, 2011. ACM.

[CJ06] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[CLJC10a] Xiaowen Chen, Zhonghai Lu, A. Jantsch, and Shuming Chen. Run-time partitioning of hybrid distributed shared memory on multi-core network-on-chips. In *Parallel Architectures, Algorithms and Programming (PAAP), 2010 Third International Symposium on*, pages 39–46, Dec 2010.

[CLJC10b] Xiaowen Chen, Zhonghai Lu, Axel Jantsch, and Shuming Chen. Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 39–44, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[cLR01] Jean claude Laprie and Brian Randell. Fundamental concepts of computer systems dependability. In *Proc. of the Workshop on Robot Dep. , Seoul, Korea*, pages 21–22, 2001.

[CMM+15] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti. Noxim: An open, extensible and cycle-accurate network on chip simulator. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 162–163, July 2015.

[Col08] *Real-time Characteristics and Safety of Embedded Systems*, pages 3–28. Springer London, London, 2008.

[Cra96] Harvey G. Cragon. *Memory Systems and Pipelined Processors*. Jones and Bartlett Publishers, Inc., USA, 1996.

[CRCR09] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. D2stm: Dependable distributed software transactional memory. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, pages 307–313, Washington, DC, USA, 2009. IEEE Computer Society.

[CYY13] T. Chiba, Myungryun Yoo, and T. Yokoyama. A distributed real-time operating system with distributed shared memory for embedded control systems. In *Dependable, Autonomic and Secure Computing (DASC), 2013 IEEE 11th International Conference on*, pages 248–255, Dec 2013.

[Des16] Emily Desjardins. Jedec announces publication of gddr5x graphics memory standard - new standard will enable higher memory performance in the graphics and specialty markets. https://www.jedec.org/news/pressreleases/jedec-announces-publication-gddr5x-graphics-memory-standard, 2016.

[DFL+06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, October 2006.

[DSRSZ10] M. Di Santo, N. Ranaldo, C. Sementa, and Eugenio Zimeo. Software distributed shared memory with transactional coherence - a software engine to run transactional shared-memory parallel applications on clusters. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 175–179, Feb 2010.

[ESR12] Mohammed El-Shambakey and Binoy Ravindran. Stm concurrency control for embedded real-time software with tighter time bounds. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *DAC*, pages 437–446. ACM, 2012.

[ETSE14] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10, Aug 2014.

[F+07] Cesare Ferri et al. A hardware/software framework for supporting transactional memory in a mpsoc environment. *SIGARCH News*, 2007.

[FF11] C. Fetzer and P. Felber. Transactional memory for dependable embedded systems. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 223–227, June 2011.

[FRJ] Sherif F. Fahmy, Binoy Ravindran, and E. D. Jensen. On bounding response times under software transactional memory in distributed multiprocessor real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

[FWM+10] Cesare Ferri, Samantha Wood, Tali Moreshet, Iris Bahar, and Maurice Herlihy. Energy and throughput efficient transactional memory for embedded multicore systems. In YaleN. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *High Performance Embedded Architectures and Compilers*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010.

[GAC+13] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha. Virtual execution platforms for mixed-time-criticality systems: The compsoc architecture and design flow. *SIGBED Rev.*, 10(3):23–34, October 2013.

[GAG] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March.

[Gas88] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Co., New York, NY, USA, 1988.

[GC08] Justin E. Gottschlich and Daniel A. Connors. Extending contention managers for user-defined priority-based transactions. In *In Proceedings of the 2008 Workshop*, 2008.

[GCAG16] S. Goossens, K. Chandrasekar, B. Akesson, and K. Goossens. *Memory Controllers for Mixed-Time-Criticality Systems: Architectures, Methodologies and Trade-Offs*. Springer Publishing Company, Incorporated, 2016.

[GDR05] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. volume 22, pages 414–421, Sept 2005.

[GGA+15] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A generic, scalable and globally arbitrated memory tree for shared dram access in real-time systems. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 193–198, San Jose, CA, USA, 2015. EDA Consortium.

[Ghe06] Frank Ghenassia. *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[GKAG12] S. Goossens, T. Kouters, B. Akesson, and K. Goossens. Memory-map selection for firm real-time sdram controllers. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 828–831, March 2012.

[GKV07] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: A benchmark for software transactional memory. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Hea02] S. Heath. *Embedded Systems Design*. Elsevier Science, 2002.

[HGB] A. Hansson, K. Goossens, and M. Bekooij. Compsoc: A template for composable and predictable multi-processor system on chips. In *Transactions on Design Automation of Electronic Systems*, page 2009.

[HLM06] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 253–262, New York, NY, USA, 2006. ACM.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2Nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[HLRP] S. Hirve, A. Lindsay, B. Ravindran, and R. Palmieri. On transactional memory concurrency control in distributed real-time programs. In *Cluster Computing, 2013 IEEE International Conference on*.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[HP03] W. Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003. Cambridge Books Online.

[HS05] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. In Pierre Fraigniaud, editor, *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 324–338. Springer Berlin Heidelberg, 2005.

[HWC+04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, ISCA '04, Washington, DC, USA, 2004. IEEE Computer Society.

[Inc15] Radiant Insights Incorporation. Fpga market size, share, analysis, research report, 2020. *Market research reports on semiconductors*, page 74, 2015.

[Int14] Intel. Desktop 4th generation intel core processor family, desktop intel pentium processor family, and desktop intel celeron processor family: Specification update (revision 014), hsd136, software using intel tsx may result in unpredictable system behavior. 2014.

[JF95] B. Janssens and W.K. Fuchs. Ensuring correct rollback recovery in distributed shared memory systems. *Journal of Parallel and Distributed Computing*, 29(2):211 – 218, 1995.

[JNW07] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[Joh99] Rushby John. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, 1999.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 364–373, New York, NY, USA, 1990. ACM.

[JQA+14] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and F.J. Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 207–217, Dec 2014.

[KAJ+07] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Designing a distributed software transactional memory system. In *ACACES '07: 3rd International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, July 2007.

[KAJ+08] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris C. Kirkham, and Ian Watson. Distm: A software transactional memory framework for clusters. In *ICPP*, pages 51–58. IEEE Computer Society, 2008.

[KB03] H. Kopetz and G. Bauer. The time-triggered architecture. In *PROCEEDINGS OF THE IEEE*, pages 112–126, 2003.

[KGaW] Leonardo Kunz, Gustavo Girão, and Flávio R. Wagner. Evaluation of a hardware transactional memory model in an noc-based embedded mpsoc. In *Proceedings of the 23rd Symposium on Integrated Circuits and System Design*, SBCCI '10, New York, NY, USA.

[KK07] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2007.

[kLCM+05] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200. ACM Press, 2005.

[Kop92] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 460–467, Jun 1992.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[Kop05] H. Kopetz. *TTA Supported Service Availability*, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[Kop06] H. Kopetz. *On the Fault Hypothesis for a Safety-Critical Real-Time System*, pages 31–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[Kop13] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2013.

[KOSH07] H. Kopetz, R. Obermaisser, C. El Salloum, and B. Huber. Automotive software development for a multi-core system-on-a-chip. In *Software Engineering for Automotive Systems, 2007. ICSE Workshops SEAS '07. Fourth International Workshop on*, pages 2–2, May 2007.

[KS92] K.M. Kavi and IEEE Computer Society. *Real-time systems: abstractions, languages, and design methodologies,*. IEEE Computer Society Press tutorial. IEEE Computer Society Press, 1992.

[KYL+] Hyesoon Kim, Sudhakar Yalamanchili, Jaekyu Lee, Nagesh Lakshminarayana, Andrew Kerr, Arun Rodrigues, and Genie Hsieh. Tutorial on ocelot and sst-macsim simulator. ISCA 2012.

[LAS+09] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In David H. Albonesi, Margaret Martonosi, David I. August, and Jose F. Martinez, editors, *MICRO*, pages 469–480. ACM, 2009.

[Law98] Ramon Lawrence. A survey of cache coherence mechanisms in shared memory multiprocessors, 1998.

[LHSC10] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association.

[LLS07] Insup Lee, Joseph Y-T. Leung, and Sang H. Son. *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC, 1st edition, 2007.

[LMS85] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1):52–78, January 1985.

[LO11] Phillip A. Laplante and Seppo J. Ovaska. *Real-Time Systems Design and Analysis: Tools for the Practitioner*. Wiley-IEEE Press, 4th edition, 2011.

[LS11] E.A. Lee and S.A. Seshia. *Introduction to Embedded Systems: A Cyber-physical Systems Approach*. Electrical Engineering & Computer Sciences. Lulu.com, 2011.

[LTMJ05] Zhonghai Lu, Rikard Thid, Mikael Millberg, and Axel Jantsch. Nnse: Nostrum network-on-chip simulation environment. In *In Proc. of SSoCC*, 2005.

[M+06] Kevin E. Moore et al. Logtm: Log-based transactional memory. In *in HPCA*, pages 254–265, 2006.

[MCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In David Christie, Alan Lee, Onur Mutlu, and Benjamin G. Zorn, editors, *IISWC*, pages 35–46. IEEE Computer Society, 2008.

[Mos16] Angela Moscaritolo. Samsung now shipping 'world's largest' hard drive. http://uk.pcmag.com/storage-devices-reviews/75717/news/samsung-now-shipping-worlds-largest-hard-drive, 2016.

[MPSV06] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Exploration of distributed shared memory architectures for noc-based multiprocessors. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2006. IC-SAMOS 2006. International Conference on*, July 2006.

[MSAHB04] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority mechanisms for oltp and transactional web applications. In Z. Meral Özsoyoglu and Stanley B. Zdonik, editors, *ICDE*, pages 535–546. IEEE Computer Society, 2004.

[MTPR13] S. Mishra, A. Turcu, R. Palmieri, and B. Ravindran. Hyflowcpp: A distributed transactional memory framework for c++. In *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*, pages 219–226, Aug 2013.

[MWU13] Stefan Metzlaff, Sebastian Weis, and Theo Ungerer. Leveraging transactional memory for a predictable execution of applications composed of hard real-time and best-effort tasks. In Michel Auguin, Robert de Simone, Robert I. Davis, and Emmanuel Grolleau, editors, *RTNS*, pages 45–54. ACM, 2013.

[NCW+] Njuguna Njoroge, Jared Casper, Sewook Wee, Yuriy Teslyar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. Atlas: A chip-multiprocessor with transactional memory support. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, San Jose, CA, USA. EDA Consortium.

[OAOD14] Z. Owda, M. Abuteir, R. Obermaisser, and H. Dakheel. Predictable and reliable time triggered platform for ambient assisted living. In *2014 8th International Symposium on Medical Information and Communication Technology (ISMICT)*, pages 1–5, April 2014.

[Obe11] R. Obermaisser. *Time-Triggered Communication*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2011.

[OESHK08] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz. The time-triggered system-on-a-chip architecture. In *IEEE International Symposium on Industrial Electronics, 2008. ISIE 2008*, pages 1941–1947, 2008. talk: IEEE International Symposium on Industrial Electronics, 2008. ISIE 2008, Cambridge, UK; 2008-06-30 – 2008-07-02.

[OG09] R. Obermaisser and P. Gutwenger. Model-based development of mpsocs with support for early validation. In *Proceedings of Industrial Electronics, 2009. IECON '09. 35th Annual Conference of IEEE*, 2009.

[OK05] R. Obermaisser and H. Kopetz. *Event-triggered and time-triggered control paradigms*. Real-time systems series. Springer, New York, 2005.

[OKSH07] R. Obermaisser, H. Kopetz, C. El Salloum, and B. Huber. Error containment in the time-triggered system-on-a-chip architecture. In *Embedded System Design: Topics, Techniques and Trends, IFIP TC10, Working Conference: International Embedded Systems Symposium (IESS), May 30 - June 1, 2007, Irvine, CA, USA*, pages 339–352, 2007.

[OO15] Z. Owda and R. Obermaisser. Trace-based simulation framework combining message-based and shared-memory interactions in a time-triggered platform. In *Event-based Control, Communication, and Signal Processing (EBCCSP), 2015 International Conference on*, June 2015.

[OOA+14a] R. Obermaisser, Z. Owda, M. Abuteir, H. Ahmadian, and D. Weber. End-to-end real-time communication in mixed-criticality systems based on networked multicore chips. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 293–302, Aug 2014.

[OOA+14b] R. Obermaisser, Z. Owda, M. Abuteir, H. Ahmadian, and D. Weber. End-to-end real-time communication in mixed-criticality systems based on networked multicore chips. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 293–302, Aug 2014.

[OUOA16] Z. Owda, M. Urbina, R. Obermaisser, and M. Abuteir. Hierarchical transactional memory protocol for distributed mixed-criticality embedded systems. In *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 334–343, Aug 2016.

[PAV14] A. Poorani, B. Anuradha, and C. Vivekanadhan. An effectual elucidation of task scheduling and memory partitioning for mpsoc. In *Intelligent Systems and Control (ISCO), 2014 IEEE 8th International Conference on*, pages 295–299, Jan 2014.

[PEP99] Paul Pop, Petru Eles, and Zebo Peng. Scheduling with optimized communication for time-triggered embedded systems. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, CODES '99, pages 178–182, New York, NY, USA, 1999. ACM.

[PK08] C. Paukovits and H. Kopetz. Concepts of switching in the time-triggered network-on-chip. In *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 120–129, Aug 2008.

[PMM+13] Marco Paolieri, Jörg Mische, Stefan Metzlaff, Mike Gerdes, Eduardo Quiñones, Sascha Uhrig, Theo Ungerer, and Francisco J. Cazorla. A hard real-time capable multi-core smt processor. *ACM Trans. Embedded Comput. Syst.*, 2013.

[RCBB+12] Arun Rodrigues, Elliot Cooper-Balis, Keren Bergman, Kurt Ferreira, David Bunde, and K. Scott Hemmert. Improvements to the structural simulation toolkit. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, SIMUTOOLS '12, pages 190–195, ICST, Brussels, Belgium, Belgium, 2012. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[RCBJ11] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 2011.

[RDK+00] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[RDP+05] A. Radulescu, J. Dielissen, S. G. Pestana, O. P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An efficient on-chip ni offering guaranteed services, shared-memory abstraction, and flexible network configuration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):4–17, Jan 2005.

[RMN+] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing hybrid transactional memory: The importance of non-speculative operations. In *Proceedings of the 23 Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, New York, NY, USA. ACM.

[RS80] Kent C. Redmond and Thomas Malcolm Smith. *Project Whirlwind; The History of a Pioneer Computer*. Butterworth-Heinemann, Newton, MA, USA, 1980.

[RTC11] Radio Technical Commission for Aeronautics. *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*, 2011.

[SBS] G. Sharma, C. Busch, and S. Srinivasagopalan. Distributed transactional memory for general networks. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, May.

[SBV10] Martin Schoeberl, Florian Brandner, and Jan Vitek. Rttm: real-time transactional memory. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 326–333. ACM, 2010.

[SCPS14] Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparsø. A Time-Predictable Memory Network-on-Chip. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 53–62, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[SH10] Martin Schoeberl and Peter Hilber. Design and implementation of real-time transactional memory. In *FPL*, pages 279–284. IEEE, 2010.

[SMM+11] Vivek Seshadri, Onur Mutlu, Todd Mowry, Michael A Kozuch, and Intel Corporation. Improving cache performance using victim tag stores, 2011.

[SR90] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Syst.*, 2(4):247–254, October 1990.

[SR11] Mohamed M. Saad and Binoy Ravindran. Hyflow: A high performance distributed software transactional memory framework. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 265–266, New York, NY, USA, 2011. ACM.

[SR12] M.M. Saad and B. Ravindran. Transactional forwarding: Supporting highly-concurrent stm in asynchronous distributed systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 219–226, Oct 2012.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[Sta96] William Stallings. *Computer Organization and Architecture (4th Ed.): Designing for Performance*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[Sto96] Neil R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[SWH95] N. Suri, C.J. Walter, and M.M. Hugue. *Advances In Ultra-Dependable Distributed Systems*, chapter 1. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264, 1995.

[TPS15] D. TamasSelicean, P. Pop, and W. Steiner. Timing analysis of rate constrained traffic for the ttethernet communication protocol. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 119–126, April 2015.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[UOO15] M. Urbina, Z. Owda, and R. Obermaisser. Simulation environment based on systemc and veos for multi-core processors with virtual autosar ecus. In *Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 1843–1852, Oct 2015.

[V 314] dSpace. *VEOS Player Document, Release 2014-B*, 2014.

[Ves07a] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symp., 28th IEEE Int.*, 2007.

[Ves07b] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, pages 239–243, Washington, DC, USA, 2007. IEEE Computer Society.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[WESK10] A. Wasicek, C. El-Salloum, and H. Kopetz. A system-on-a-chip platform for mixed-criticality applications. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 210–216, May 2010.

[WLSC07] John Paul Walters, Zhengqiang Liang, Weisong Shi, and Vipin Chaudhary. Wireless sensor network security: A survey," in book chapter of security. In *in Distributed, Grid, and Pervasive Computing, Yang Xiao (Eds)*, pages 0–849. CRC Press, 2007.

[Yos16] Junko Yoshida. Auto security demands all-over answer. http://www.eetimes.com/document.asp?doc_id=1330037, 2016.

[YYP+] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, July.

[ZCCD13] Lihang Zhao, Woojin Choi, Lizhong Chen, and J. Draper. In-network traffic regulation for transactional memory. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 520–531, Feb 2013.

[Zur09] Richard Zurawski. *Embedded Systems Handbook, Second Edition 2-Volume Set*. CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2009.

List of Acronyms

PAR Positive Acknowledgment with Re-transmission

TMR Triple Modular Redundancy

ASIL Automotive Safety Integrity Level

SIL Safety Integrity Level

IoT Internet of Things

MCS Mixed-Criticality System

CPS Cyber-Physical System

LIF Linking Interfaces

TII Technology Independent Control Interface

TDI Technology Dependent Debug Interface

FPGA Field-Programmable Gate Array

TDMA Time Division Multiple Access

TDM Time-Division Multiplexing

PDS Pulsed Data Stream

TTE Time-Triggered Ethernet

TISS Trusted Interface Subsystems

TSS Trusted Subsystem

MPSoC Multi-Processor System-on-a-Chip

TMSoC Transactional Memory System-on-a-Chip

TTNoC Time-Triggered Network-on-a-Chip

TTSoC Time-Triggered System-on-a-Chip

TTNI Time-Triggered Network Interface

SoC System-on-a-Chip

NoC Network-on-Chip

TTNI Time-Triggered Network Interface

MU Management Unit

NI Network Interface

IP Intellectual Property

I/O Input/Output

FCR Fault Containment Region

WCET Worst-case Execution Time

COTS Commercial-Off-The-Shelf

VL Virtual Link

VC Virtual Channel

VLID Virtual Link IDentification

VCID Virtual Channel IDentification

TSU Traffic Shaping Unit

E2E End-to-End

NTP Network Time Protocol

SAE Society of Automotive Engineers

VLAN Virtual Local Area Network

VPN Virtual Private Network

AVB Audio/Video Bridging

AUTOSAR AUTomotive Open System ARchitecture

CAN Controller Area Network

CIS CAN Interface Subsystem

ABS Anti-lock Braking System

TM Transactional Memory

HTM Hardware Transactional Memory

STM Software Transactional Memory

HyTM Hybrid Transactional Memory

MTC Mixed-Criticality Transaction Controller

DTMA Distributed Transactional Memory Architecture

DSM Distributed Shared Memory

DMA Direct Memory Access

DMTC Distributed Mixed-criticality Transactional Controller

RAM Random-access Memory

DRAM Dynamic Random-access Memory

SDRAM Synchronous Dynamic Random-access Memory

SRAM Static Random-access Memory

ROM Read-only Memory

MMU Memory Management Unit

DDR Double Data Rate

TLB Translation Lookaside Buffer

TLM Transaction Level Models

Selected Publications

Conferences:

1. *Z. Owda*, M. Urbina, R. Obermaisser, M. Abuteir, “Hierarchical Transactional Memory Protocol for Distributed Mixed-Criticality Embedded Systems”, *BEST PAPER AWARD* at The 14th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC), Auckland, New Zealand - August 2016.

2. *Z. Owda*, R. Obermaisser, “Mixed-Criticality Transational Memory Controller for Embedded Systems”, The 14th International Conference on Industrial Informatics (INDIN), Poitiers-Futuroscope, France - July 2016.

3. *Z. Owda* and R. Obermaisser “A Predictable Transactional Memory Architecture with Selective Conflict Resolution for Mixed-Criticality Support in MPSoCs”, 13th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 2015). Porto - October 2015.

4. M. Urbina, *Z. Owda* and R. Obermaisser, “Simulation Environment based on SystemC and VEOS for Multi-Core Processors with Virtual AUTOSAR ECUs”, 13th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC-2015). Liverpool, England 26-28 October 2015.

5. M. Abuteir, R. Obermaisser, *Z. Owda*, T. Moudouthe, “Off-chip/On-chip Gateway Architecture for Mixed-Criticality Systems based on Networked Multi-Core Chips”, 18th IEEE International Conference on Computational Science and Engineering (CSE). Porto - October 2015.

6. *Z. Owda*, M. Abuteir and R. Obermaisser: "Co-simulation Framework for Networked Multi-core Chips with Interleaving Discrete Event Simulation Tools", ETFA 2015 - IEEE International Conference on Emerging Technology & Factory Automation, Luxembourg, September 8 -11, 2015.

7. *Z. Owda* and R. Obermaisser: ”Trace-based Simulation Framework Combining Message-Based and Shared-Memory Interactions in a Time-Triggered Platform”, IEEE International Conference on Event-based Control, Communications & Signal Processing, Special Session on Mixed-Criticality Systems, Poland- Krakow, June 16, 2015.

8. R. Obermaisser, *Z. Owda*, M. Abuteir, H. Ahmadian, D. Weber, “End-to-End Real-Time Communication in Mixed-Criticality Systems Based on Networked Multicore Chips”, in 17th Euromicro Conference on Digital Systems Design, Verona, Italy, August 27-29, 2014.

9. *Owda, Z.*; Abuteir, M., Obermaisser, R.; Dakheel, H., "Predictable and Reliable Time Triggered Platform for Ambient Assisted Living",8th International Symposium on Medical Information and Communication Technology 2014 (ISMICT 2014)",Florence, Italy", Publication Year: 2014

10. *Owda, Z.* ; Tsiatouhas, Y.; Haniotakis T., “High performance and low power dynamic circuit design”, New Circuits and Systems Conference (NEWCAS), 2011 IEEE 9th International, Digital Object Identifier: 10.1109/NEWCAS.2011.5981329, Publication Year: 2011, Page(s): 502 – 505

11. Haniotakis, T., *Owda, Z.*, Tsiatouhas, Y. “Memory-Less Pipeline Dynamic Circuit Design Technique”, VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on Digital Object Identifier: 10.1109/ISVLSI.2010.42, Publication Year: 2010 , Page(s): 201 – 205

Journals:

1. Efstathiou, C. ; *Owda, Z.* ; Tsiatouhas, Y., “New High-Speed Multioutput Carry Look-Ahead Adders”, Circuits and Systems II: Express Briefs, IEEE Transactions on Volume: 60 , Issue: 10, Digital Object Identifier: 10.1109/TCSII.2013.2278088, Publication Year: 2013 , Page(s): 667 – 671