# Hierarchical Transactional Memory Protocol for Distributed Mixed-Criticality Embedded Systems

Zaher Owda, Moisés Urbina, Roman Obermaisser and Mohammed Abuteir
*Institute of Embedded Systems - University of Siegen*

***Abstract*—Transactional memories with support for real-time and reliability requirements are a research challenge for the deployment in safety-critical embedded systems. In addition, at present there are no transactional-memory architectures considering hierarchical systems of networked multi-core chips with both on-chip and off-chip networks.**

**The presented work offers a predictable transactional memory solution for hierarchical distributed systems by executing a transactional memory protocol spanning both on-chip and off-chip networks to manage the memory operations of the cores at different multi-core chips. This includes relocating memory pages between the local caches at cores and the external memories while preserving the mixed-criticality requirements of the distributed system. The proposed protocol provides temporal predictability, fault isolation and bounded execution time assurances for safety-critical applications. An automotive use case with a simulation framework serves for the evaluation of the proposed solution.**[1]

## I. Introduction

Transactional memories simplify parallel programming by ensuring atomicity, consistency and isolation of computations with access to a shared memory. The benefits compared to a lock-based synchronization include a higher level of abstraction for the safe and scalable composition of software modules [1]. These advantages are also recognized for dependable embedded systems [2], where additional challenges with respect to real-time constraints and fault-tolerance must be addressed.

Predictable transactional memories for embedded systems have been introduced at the chip-level with corresponding timing analyses (e.g., [3]). Likewise, Transactional Distributed Shared Memories (TDMS) were proposed for distributed embedded real-time systems (e.g. [4]). The focus is improved programmability in conjuction with temporal predictability and composability.

However, 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 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 [5] (e.g., Class A according to DO-178C [6]). Hence, fault-tolerance at system level is required by exploiting redundancy with multiple independent chips.

In order to offer transactional memories in these hierarchical systems, we need to manage the memory operations of the cores at different multi-core chips by executing a transactional-memory protocol spanning both on-chip and off-chip networks. Memory pages need to be relocated between the local caches at cores and the external memories of the multi-core chips. In addition, commits and rollbacks need to be performed to ensure atomicity, consistency and isolation in the presence of memory conflicts.

The presented transactional memory protocol solves these requirements, while also offering temporal predictability and fault isolation in mixed-criticality applications. The algorithms for conflict resolution ensure that the execution time of safety-critical application 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 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.

The remainder of the paper is structured as follows. Section II provides an overview of related work in the area of transactional memories. The system architecture for the hierarchical transactional memory is the focus of Section III. Section IV explains the hierarchical transactional memory protocol. In Section V an implementation of the protocol is presented using a co-simulation of multi-core chips and off-chip networks. The results for a realistic use case are summarized in Section VI. The paper concludes with a discussion in Section VII.

## II. Related Work

Distributed shared memories have been investigated in the context of chip level and cluster level as well as in operating systems [7], [8]. The description and discussion of existing memory solutions for distributed systems is presented in this section.

### A. Solutions at chip level

It can be noticed from the literature that fewer research has been performed on Distributed Shared Memory (DSM) solutions dedicated to multi-core processors with a Network-on-Chip (NoC) in comparison to off-chip level solutions. In [9] the focus of the work was on a DSM architecture suitable for low-power multiprocessors. A hybrid organization including

[1]This work has been supported in part by the European FP7 project DREAMS under grant agreement 610640.

private and shared memory space for DSM is introduced for multi-core processors in [10], 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 [11] proposed a microcoded controller as a hardware module in each node to interconnect the core, the local memory and the network. Furthermore, partitioning of virtual memory resources and task scheduling is proposed as a solution for managing memory accesses in non-distributed systems at operating system level.An approach for task scheduling and memory partitioning for a Multi-Processor System-on-a-Chip (MPSoC) using a scratch pad memory is presented by [12].

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 [13]. To our knowledge, transactional memory solutions for mixed-criticality embedded systems address one multi-core chip only. In [14], 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 [15]. This work provides a software-based memory throttling mechanism to explicitly control the memory interference in the Linux kernel. A predictable transactional memory architecture with a focus on single node mixed-criticality systems is presented in [16].

### *B. Solutions at cluster level*

A Distributed Transactional Memory (DTM) is the realization of a transactional memory using off-chip communication networks. The so-called Ballistic protocol [17] 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 [18] 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.

The java-based transaction execution engine of DSTM2 [19] is used as a baseline for some off-chip distributed system solutions. The prototype of a distributed software transactional memory framework is described in [20] 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 [21] 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 [22] 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. [23] presents a distributed transactional memory framework for distributed concurrency control in C++ based on [22] called HyflowCPP. The Real-Time Transaction Forwarding Algorithm [4] extends the distributed concurrency control scheme of the Transactional Forwarding (TFA) [24]. 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 constrains.

### *C. Research Gap*

As shown from the earlier analysis, existing solutions are focusing either on on-chip or off-chip memory solutions. Thus, the hierarchical support for an architecture providing a transactional memory at both levels is missing. Moreover, predictability and reliability requirements are not addressed in existing solutions for distributed systems.

## III. Distributed System Architecture for Hierarchical Transactional Memory

In this section, a distributed system architecture for a transactional memory with networked MPSoCs is defined (cf. Figure 1). 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.

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.

Each *core* of a node provides an application service, an external memory service 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. Safety Integrity Level (SIL) according to IEC61508 [25]. Each core has its own local cache, while

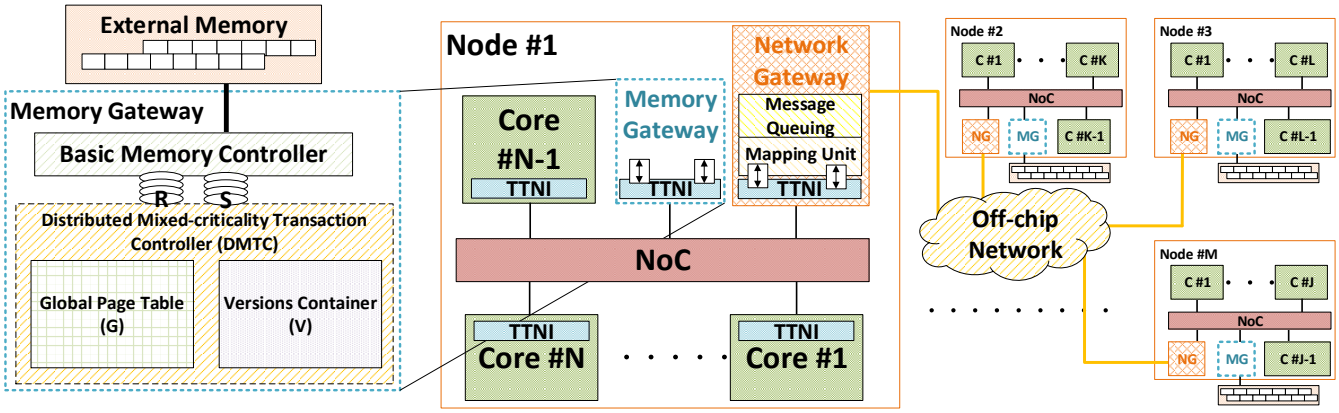


Figure 1: DTM architecture for networked multicore chips

the *external memory service* is responsible for generating the corresponding memory requests of the application core and handling memory replies, especially in case of memory rollbacks. The application service and the external memory service support bidirectional communication via the *Time-Triggered Network Interface (TTNI)* of the on-chip network. We assume a time-triggered NoC in the architecture such as AEthereal [26] or TTNoC [27] 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.

A node's gateways are connected to TTNIs through different ports that are corresponding to the cores (cf. Figure 1). The *mapping unit* of the *network gateway* (NG) provides message redirection between the corresponding ports and message queues. 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 Distributed Mixed-criticality Transactional Controller (DMTC). The memory controller implements real-time memory gateway 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 IV 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 a predictable distributed mixed-criticality system.

## IV. Distributed Mixed-criticality 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 2 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 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

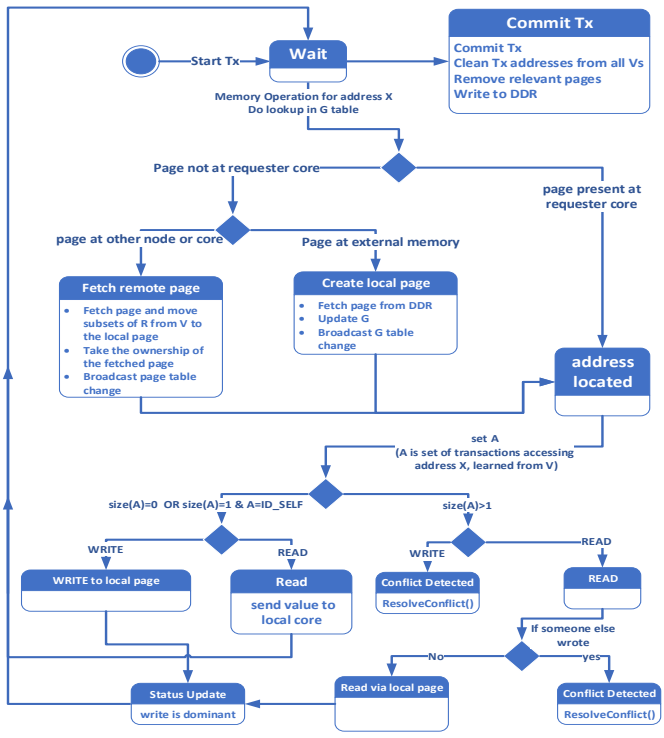


Figure 2: Transaction Processing State Machine

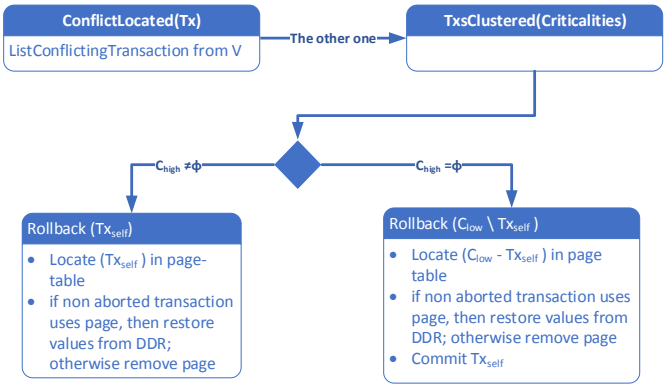


Figure 3: Criticality Based Conflict Detection State Machine

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 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 2), 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 3), 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 3, 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 3), 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 transactions subset $C_{low}$ without $Tx_{self}$ are 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 2. 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.

## V. Implementation Using an Automotive Simulation Environment

The realization of the proposed distributed system architecture with the hierarchical DMTC protocol is presented in this section. For this purpose, SystemC/TLM [28] is used for the protocol and node implementation. VEOS [29] is used for simulating the FlexRay bus and the application components within the cores of an MPSoC. Finally, DRAMSim2 [30] is used to simulate 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. This paper uses the co-simulation technique presented in [31] and [32] to accurately coordinate VEOS and multiple SystemC-based nodes.

The previously mentioned tools are used for the following reasons: SystemC is a system-level modeling language for event-driven modeling that provides precise temporal behaviors of the simulated modules. It is widely used in the literature for modeling and simulation of on-chip architectures e.g., [33], [34]. Moreover, Transaction Level Models (TLMs) are used to enhance the overall simulation speed by providing a higher level of abstraction for the simulation modules. Interface-based communication, blocking and non-blocking process structures, bidirectional and unidirectional transactions are some of the useful capabilities provided by the TLMs.

DRAMSim2 is a widely used cycle-accurate open-source DRAM simulator that models the memory controller, memory channels, ranks, banks and timing constraints [35], [36].

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. Simulink models can be integrated into the simulation system for representing the physical environment. In addition, the experimental tool ControlDesk can access the VEOS simulator for testing the AUTOSAR software.

### A. Implementation

The proposed distributed system architecture is mapped to a synthetic distributed automotive system as shown in Figure 4. The distributed system consists of two MPSoCs that are connected through a FlexRay bus. Each node has its own on-chip communication schedule, 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.

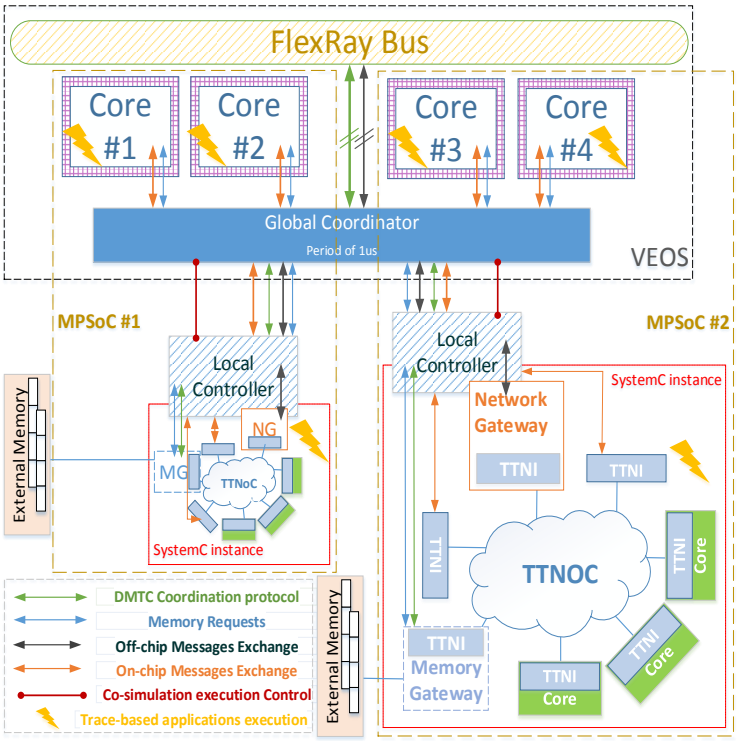


Figure 4: Implementation Using VEOS-SystemC for Distributed MPSoCs

Each node can have a configurable number of cores that are interconnected through a deterministic Time-Triggered Network-on-a-Chip (TTNoC) [27]. Cores are simulated either as SystemC-based or VEOS-based application cores. Each core runs a trace-based application that is assigned to it. The trace files are generated based on the process presented in [35]. As described in Section III, 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 [35] by integrating VEOS-based application 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 [37]. In addition to DMTC, the proposed memory gateway assumes a compositional real-time memory controller [38] 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 controller 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 4) 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 4) 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 IV. 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 application cores in an MPSoC. These application cores 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.

### B. 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 controller 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 $1\mu s$. However 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 V-A. 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 *node 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 know from the on-/off-chip communication schedules.
- *Payload:* Contains the data or the memory operations.

In case of the DMTC, coordination messages are mes-

sages 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 application cores, 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.

## VI. Usecase & Evaluation

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

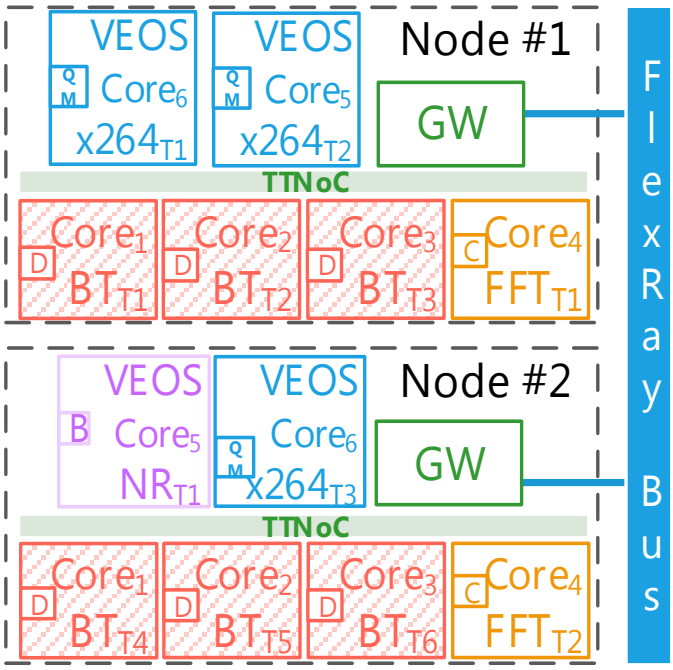


Figure 5: Distributed Automotive Use-case

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 [39]. 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 [40] in order to generate the trace-files for the distributed 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 systemC/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 5) represents the trace file identifier of each core.

As described in Section V, 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 $200\mu s$ phase for messages sent from node#1 to node#2. The phase of messages sent from node#2 to node#1 is set to $300\mu s$. Additionally, messages between VEOS-based cores and their node are exchanged based on the co-simulation steps of $1\mu s$.

| 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 |

Figure 6: 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 is 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.

The execution time of an application 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 6 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 the multiple attempts of the application 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 6 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 |

Figure 7: Execution Time Per Core for Both Nodes

Figure 7 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.

## VII. Discussion and Conclusion

Distributed memory solutions have previously been presented either for on-chip or off-chip systems. Despite that a transactional memory ensures atomicity, consistency and isolation by providing lock-free solutions, existing transactional memory solutions are mainly investigating the on-chip level. Moreover, predictability and reliability requirements of transactional memories 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 in this work. This distributed mixed-criticality transactional memory protocol provides a high level of predictability and reliability assurances at both on-chip and off-chip levels, and offers temporal predictability and fault isolation in 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 work, in which the cores of the two nodes are sharing the execution of four applications with different criticalities.

The results have shown that the proposed hierarchical solution provides an efficient, predictable and reliable support 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


[1] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd edition*. Morgan & Claypool, 2010.

[2] C. Fetzer *et al.*, "Transactional memory for dependable embedded systems," in *Dependable Systems and Networks Workshops, 2011 IEEE/IFIP 41st Int. Conf. on*.

[3] M. Schoeberl and P. Hilber, "Design and implementation of real-time transactional memory." in *FPL*. IEEE, 2010, pp. 279–284.

[4] 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*.

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

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

[7] H. Vasava and J. Rathod, "Software based distributed shared memory (dsm) model using shared variables between multiprocessors," in *Communications and Signal Processing (ICCSP), 2015 International Conference on*, April 2015, pp. 1431–1435.

[8] T. Chiba, M. 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*, Dec 2013, pp. 248–255.

[9] 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.

[10] X. Chen, Z. Lu, A. Jantsch, and S. 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*, Dec 2010, pp. 39–46.

[11] ——, "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*, ser. DATE '10. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010, pp. 39–44.

[12] 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*, Jan 2014, pp. 295–299.

[13] C. Ferri, S. Wood, T. Moreshet, I. Bahar, and M. Herlihy, "Energy and throughput efficient transactional memory for embedded multicore systems," in *High Performance Embedded Architectures and Compilers*, ser. Lecture Notes in Computer Science, Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds. Springer Berlin Heidelberg, 2010.

[14] B. Cilku, A. Crespo, P. Puschner, J. Coronel, and S. Peiro, "A tdma-based arbitration scheme for mixed-criticality multi-core platforms," in *Event-based Control, Communication, and Signal Processing (EBCCSP), 2015 International Conference on*, June 2015, pp. 1–6.

[15] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, July.

[16] Z. Owda and R. Obermaisser, "A predictable transactional memory architecture with selective conflict resolution for mixed-criticality support in mpsocs," in *Embedded and Ubiquitous Computing (EUC), 2015 IEEE 13th International Conference on*, Oct 2015, pp. 158–162.

[17] M. Herlihy and Y. Sun, "Distributed transactional memory for metric-space networks," in *Distributed Computing*, ser. Lecture Notes in Computer Science, P. Fraigniaud, Ed. Springer Berlin Heidelberg, 2005, vol. 3724, pp. 324–338.

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

[19] M. Herlihy, V. Luchangco, and M. 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*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/1167473.1167495

[20] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. 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.

[21] M. Di Santo, N. Ranaldo, C. Sementa, and E. 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*, Feb 2010, pp. 175–179.

[22] M. M. Saad and B. Ravindran, "Hyflow: A high performance distributed software transactional memory framework," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 265–266.

[23] 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*, Aug 2013, pp. 219–226.

[24] 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*, Oct 2012, pp. 219–226.

[25] R. Bell, "Introduction to iec 61508," in *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*, ser. SCS '05. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 3–12.

[26] K. Goossens, J. Dielissen, and A. Radulescu, "Aethereal network on chip: concepts, architectures, and implementations," vol. 22, no. 5, Sept 2005, pp. 414–421.

[27] R. Obermaisser *et al.*, "The time-triggered system-on-a-chip architecture," in *IEEE Int. Symp. on Industrial Electronics, 2008*, talk: IEEE Int. Symp. on Industrial Electronics.

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

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

[30] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator." *Computer Architecture Letters*, 2011.

[31] 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*, Oct 2015, pp. 1843–1852.

[32] M. Urbina and R. Obermaisser, "A Gateway Core between On-chip and Off-chip Networks for an AUTOSAR Message-based Multi-core Platform," in *Automotive Meets Electronics 2016. VDE Conf. on*.

[33] "NOXIM : The NoC Simulator." [Online]. Available: www.noxim.org

[34] 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.

[35] 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.

[36] H. Kim, S. Yalamanchili, J. Lee, N. Lakshminarayana, A. Kerr, A. Rodrigues, and G. Hsieh, "Tutorial on ocelot and sst-macsim simulator," ser. ISCA 2012. [Online]. Available: http://comparch.gatech.edu/hparch/isca12_gt.html

[37] 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.

[38] 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.

[39] "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.

[40] C. Bienia *et al.*, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors." in *IISWC*, D. Christie, A. Lee, O. Mutlu, and B. G. Zorn, Eds.