The ACROSS MPSoC – A New Generation of Multi-Core Processors designed for Safety-Critical Embedded Systems

Christian El Salloum, Martin Elshuber, Oliver Höftberger, Haris Isakovic, Armin Wasicek
{christian.el-salloum, martin.elshuber, oliver.hoeftberger, haris.isakovic, armin.wasicek}@tuwien.ac.at
Vienna University of Technology – Institute of Computer Engineering
Treitlstrasse 3/3rd floor, 1040 Vienna

***Abstract*— The European ARTEMIS ACROSS project aims to overcome the limitations of existing Multi-Processor Systems-on-a-Chip (MPSoC) architectures with respect to safety-critical applications. MPSoCs have a tremendous potential in the domain of embedded systems considering their enormous computational capacity and energy efficiency. However, the currently existing MPSoC architectures have significant limitations with respect to safety-critical applications. These limitations include difficulties in the certification process due to the high complexity of MPSoCs, the lacking temporal determinism and problems related to error propagation between subsystems. These limitations become even more severe, when subsystems of different criticality levels have to be integrated on the same computational platform. Examples of such mixed-criticality integration are found in the avionics and automotive industry with their desire to integrate safety-critical, mission critical and non-critical subsystems on the same platform in order to minimize size, weight, power and cost. The main objective of ACROSS is to develop a new generation of multi-core processors designed specially for safety-critical embedded systems; the ACROSS MPSoC. In this paper we will show how the ACROSS MPSoC overcomes the limitations of existing MPSoC architectures in order to make the multi-core technology available to the safety-critical domain.**

I. Introduction

A Multi-Processor System-on-a-Chip (MPSoC) incorporates multiple, potentially heterogeneous processing cores and other functional units in a single case on a single die. Compared to general purpose single core processors, MPSoCs can provide enormous computational capacity in an energy efficient and cost efficient way. The roadmaps of the semiconductor industry [1] show a very clear trend towards multi-core technology, and we can safely assume that the majority of future high-end processors will be MPSoCs. Today, MPSoCs are typically applied in personal computers or consumer electronic devices like smart phones or tablets.

The scope of this paper is to elaborate MPSoCs in the domain of safety-critical embedded systems. Safety-critical systems are systems whose failure could result in loss of life, significant property damage, or damage to the environment. Examples are flight control systems for aircrafts, automotive control systems, medical devices, industrial control systems or nuclear power plants. As we will point out, MPSoCs could bring many benefits to safety-critical applications, but unfortunately, the currently existing MPSoC architectures were not designed with a strong focus on safety and certification and thus have serious drawbacks and limitations with respect to this domain. To overcome these limitations, a European consortium of 16 partners of industry and academia joined together in the ARTEMIS ACROSS project. A major result of this project is a new generation of multi-core processors designed specially for safety-critical embedded systems; the ACROSS MPSoC. In this paper we will show how the ACROSS MPSoC overcomes the limitations of existing MPSoC architectures in order to make the multi-core technology available to the safety-critical domain.

The remainder of the paper is structured in the following way: Section II motivates the use of MPSoCs in embedded systems, while Section III points out problems and limitations of existing MPSoC architectures with respect to safety-critical systems. Section IV introduces the ACROSS MPSoC architecture, of which we claim to overcome these limitations, and Section V describes a prototype implementation of the architecture. Section VI outlines the related work and Section VII concludes the paper.

II. Benefits of MPSoCs in Embedded Systems

In order to motivate our work we start with showing the most important expected benefits of MPSoCs in the context of safety-critical embedded systems:

a) Energy and area efficiency: For decades, an increasing number of transistors was used to push the performance of a *single* processing core by developing larger micro-architectures with a higher complexity. Examples are micro-architectures based on super-pipelined designs featuring speculative, super-scalar, and out-of-order execution. The problem is that performance increases that are exclusively based on advances in micro-architecture are governed by *Pollack's Rule* [2]. Pollack's Rule states that, in the same process technology, a leading micro-processor consumes twice the area and power over the previous generation microprocessor, compared with a performance increase by a factor of 1.4. In other words, doubling the number of transistors in a single processor core results only in 40% additional performance, which means that ever increasing single core sizes yield diminishing performance in a power and area envelope. In contrast to the single core approach, a multi-core architecture has the potential to provide near linear performance improvement.

b) Computational performance: We have to be aware, that the potential speedup gained by a multi-core is limited by *Amdahl's Law*, which states that the theoretical maximum speedup achievable by parallelization is limited by the relative size of the non-parallelizable part (i.e. the serial part) of a program. Equation 1 shows how the maximum speedup (S) relates to the relative size of the parallelizable part (p), the relative size of the serial part ($s = 1 - p$), and the number of parallel cores (n).

$$S = \frac{s + p}{s + p/n} = \frac{1}{s + (1 - s)/n} \tag{1}$$

The formula shows that even a small percentage of non-parallelizable code in a program leads to a saturation of the achievable speedup at a small number of cores.

It must be considered that this limitation is only valid if one tries to parallelize a single application with a single continuous serial part across all the cores in the chip. In the domain of embedded systems, a device typically integrates multiple application subsystems that are inherently parallel. As an example consider the electronic control system in a modern vehicle executing tasks for the power train, the comfort electronics, or for the vehicle dynamics management system. In such a system the real challenge is not to parallelize algorithms solving *one* big problem as it is done for high-performance computing, but to efficiently integrate multiple tasks that are inherently parallel. So what we want to point out here, is that Amdahl's Law [3] which is considered as the most limiting factor concerning parallelization for high-performance computing is not necessarily a significant constraint in many embedded applications.

c) Heterogeneity: A typical embedded application consists of multiple subsystems and tasks with different requirements to the underlying hardware. Examples are encoding and decoding of signals like video or audio streams, encryption and decryption tasks in order to establish required security properties, or data transformations for control loops. Of course, all these tasks could be theoretically executed on general purpose CPUs, but in practice this is very inefficient and often infeasible due to the strict time and energy constraints of embedded applications. Good examples are high frequency control loops where specialized hardware like Digital Signal Processors (DSPs) is required to meet the specified deadlines, or security hardware extensions that significantly increase the energy efficiency of the implementation of a cipher.

Constructing a single core that strives to satisfy all possible kinds requirements, always result in a more or less inefficient trade-off solution. MPSoCs enable the combination of multiple heterogeneous cores on a single die that can be highly optimized for the individual tasks in an embedded application. This approach is already followed in many consumer devices, where dedicated cores with different functionalities (e.g., graphic processors, wireless communication components, general purpose CPUs) are integrated to form an efficient System-on-a-Chip.

d) Reduction of physical units: The high computational capacity and the possibility to integrate multiple heterogeneous cores on a single die, make an MPSoC optimally suitable for integrating multiple functionalities of an embedded system into a single chip, and thus into a single device. In many embedded applications, the consolidation of multiple functionalities in a single device can lead to massive savings with respect to cost, energy, volume and weight.

Modern cars already contain up to 70 Electronic Control Units (ECU), which are interconnected via multiple different communication networks. By integrating multiple functionalities in the ECUs, the total number of ECUs can be reduced, which directly leads to lower manufacturing costs since fewer cases, electronic parts, cables, connectors and assembly steps on the production line are required.

Beside these savings, a reduction of the numbers of ECUs can lead to improved reliability of the entire system. The rationale is that if there are fewer components in the system, there are also fewer components that can fail. In particular this is true regarding the involved reduction of connectors. By analyzing field data from the automotive industry it was shown that more than 30% of electrical failures are caused by connector problems [4].

III. Limitations of existing MPSoC architectures with respect to safety-critical applications

Since MPSoCs can bring many benefits to the domain of embedded systems, developers and manufacturers are eager to employ this technology for their products. However, existing MPSoC architectures have significant limitations with respect to safety-critical applications. In this section we point out these limitations and explain why current architectures are designed that way.

A. Complexity and certification

Typically safety-critical systems have to be certified by a designated certification authority before they are allowed to be put in operation. The higher the criticality level, the more costly and time-consuming also the certification process. For systems belonging to the highest criticality class, the certification process can be the most costly part in the entire development process. The effort to be done to certify a given system depends strongly on its architectural properties. The effort to certify a system grows with the system's complexity, and the complexity of a system depends highly on the possible interaction patterns of its constituting elements.

According to the European Aviation Safety Agency (EASA) [5], most existing MPSoCs are classified as "highly complex micro controllers". The EASA defines a micro controller as highly complex if at least one of the statements below is true:

- "more than one Central Processing Unit (CPU) is embedded and they use the same bus (which is not strictly separated or which uses the same single port memory)"
- "several complex interfaces are dependent on each other and exchange data"

- "several internal busses are integrated and are used in a dynamic way (for example, a dynamic bus switch matrix)"

Most of the existing MPSoCs were not designed with a special focus on certification, and fall into this category. The classification as a highly complex micro controller entails very high certification efforts for systems of the highest criticality class. In many cases, this classification renders certification infeasible with respect to cost and time.

B. Lacking of temporal determinism

Many safety-critical embedded systems are classified as *hard real-time systems* [6]. A real-time system is a system where its correct behavior does not only depend on the value domain but also on the temporal domain (e.g., results have to be delivered before a specified deadline). If there is at least one deadline, where a deadline miss could lead to catastrophic consequences (i.e., damage to the equipment, environment, or even the loss of human life), the system is called a hard real-time system. An example is the airbag control unit of a car, which continuously monitors a number of related sensors within the vehicle in order to judge on the actual driving situation. In the case of an accident it has to trigger the ignition of a gas generator propellant to inflate the airbag, within a specified deadline. If this deadline is missed, the airbag looses is functionality to save life (e.g., the head of the driver could crash into the steering wheel before the airbag is inflated).

In a hard real-time system one has to guarantee that deadlines are met even in worst-case scenarios (e.g., worst-case load on the communication bus). Therefore, the entire design has to be based and optimized with respect to the worst-case execution times of all time-critical activities (e.g., computation or communication activities, memory access, IO access, etc.).

Since most of the existing multi-core architectures were not designed with a special focus on hard-real time applications, they are not optimized towards worst-case execution time, but towards the maximal average performance. Optimizing the maximal average performance means optimizing the throughput which makes perfect sense for non real-time systems. In order to optimize the throughput, these architectures employ sophisticated mechanisms like complex implicitly loaded cache architectures or automatic coherence protocols that let a distributed memory architecture appear as a single shared memory with a continuous address space. In order to make life simpler for the (non real-time) programmer, these mechanisms are hidden below nice and clean abstraction layers that hide the implementation details of these mechanisms.

The problem with respect to real-time behavior is that it is very difficult to reason about the worst-case temporal behavior of these performance optimization mechanisms. This problem gets even more complex due to the before mentioned abstraction layers, that hide away implementation-specific details (e.g., exact replacement strategy of a distributed cache architecture in a multi-core). In order to determine the worst-case execution time, exact models of the hardware are required, and even if such models are available, an exact analysis might be infeasible due to the large state space implied by the complexity of the architecture.

C. Error propagation and non-intended interference

This point is closely related to the integration of multiple subsystems within a single MPSoC. The subsystems to be integrated do not necessarily share the same criticality levels (e.g., a non-critical comfort function and a high-critical control function). If the MPSoC architecture does not ensure sufficient independence between the functions of different criticality levels, it could always happen that an error in a non-critical function propagates to a critical function. Therefore, a designer has to choose between two options:

1) To certify all functions, even the functions with a low criticality level, according to the requirements of the most critical function in the system (i.e., treating all functions according to the highest criticality level in the system). This option is a very bad choice, considering that certification is a very costly factor, and it gets more costly the higher the criticality level of a function is. Moreover, many low-critical functions are so complex (e.g., a modern multi media system) that a certification according to the highest criticality level would be infeasible.
2) The second option is to build reliable partitioning mechanisms into the architecture, that assure sufficient independence between the different functionalities. In this case, each function can be certified according to its own criticality level. This fact is also reflected in one of the most relevant certification standards, the IEC61508 standard titled *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE, or E/E/PES)[7]*: "An E/E/PE safety-related system will usually implement more than one safety function. If the safety integrity requirements for these safety functions differ, unless there is sufficient independence of implementation between them, the requirements applicable to the highest relevant safety integrity level shall apply to the entire E/E/PE safety-related system."

Since most of the existing multi-core architectures are not designed with a strong focus on certification, they are lacking adequate partitioning mechanisms and exhibit non-intended interference between subsystems (e.g., the execution of a given subsystem can lead to a longer execution time of another subsystem). Sources of this non-intended interference are for example shared resources without rigorous access control like shared caches or shared I/O (e.g., the execution of one task can lead to a cache miss of another task).

IV. The ACROSS MPSoC

In the previous section we have outlined the problems and limitations of existing MPSoC architectures with respect to safety-critical embedded systems. In this section we introduce the ACROSS MPSoC architecture which was specifically

targeted for applications including safety functions that belong to the highest criticality class.

The major design objectives of the ACROSS MPSoC architecture are:

- **Certifiability:** The architecture shall enable time-efficient and cost-efficient certification of systems belonging to the highest criticality class. In such systems the rate of failures with potential catastrophic consequences has to be lower than 10^{-9} failures per hour, which means approximately less than one failure in 100 000 years.
- **Complexity management:** The architecture shall enable to manage the complexity of highly integrated designs. Therefore, the architecture shall support the independent development and verification of subsystems (i.e., different application functionalities). The integration of the subsystems into the system shall not invalidate the properties that have been already validated on the subsystem level (e.g., the temporal behavior of an already integrated subsystem should not be changed by integration of a new subsystem).
- **Temporal determinism:** The architecture shall enable the efficient construction, validation and certification of systems with a temporally deterministic behavior (e.g., guaranteed deadlines).
- **Mixed-criticality integration:** The architecture shall enable the efficient integration of subsystem with different criticality levels into a single MPSoC. This requires temporal and spatial partitioning mechanisms that prevent any non-intended interference and error propagation between subsystems.

A. Overview of the architecture

The ACROSS MPSoC architecture approaches the above stated objectives with the following means:

a) Increased level of abstraction: The first distinct property of the ACROSS MPSoC is the increased level of design abstraction. In contrast to traditional MPSoC architectures, we consider the MPSoC as a *networked multicomputer* on a chip. To reflect this point of view, we do not speak of interconnected cores, but have introduced the notion of a *Micro Component (μComponent)*. A μComponent is a highly autonomous component, like a node in a distributed system. It has sufficient local memory for executing its program code, and a precisely defined message-based interface as described below.

b) Massage-based Linking Interfaces (LIFs): A μComponent has an explicitly defined LIF via which it interacts with the other μComponents on the ACROSS MPSoC. The LIF is a *message-based* interface, which is not only defined in the *value domain* (e.g., message length) but also precisely in the *temporal domain* (e.g., periodic points in time when the message is sent). The message-based interface constitutes a significant increase of the level of abstraction compared to traditional MPSoC architectures, where the interface of a core is typically defined by low-level load and store instructions to a shared address space. In the ACROSS MPSoC, the messages crossing the LIF represent self-contained chunks of information that can be directly interpreted at the application level (e.g., the value of sensor or a more complex data structure like the state of a flight controller).

c) Restrict interactions to occur exclusively over the LIFs: The architecture restricts the interaction between the μComponents to occur exclusively via the explicitly defined LIFs described above (i.e., there are no other physical connections between the μComponents). This restriction prevents any non-intended interference or hidden channels among the μComponents. Thus, understanding the LIFs of the μComponents is sufficient to understand the interaction patterns of the entire system (i.e., the MPSoC). Restricting the interaction of μComponents to their LIFs is a fundamental part of the complexity-management strategy in the ACROSS architecture. It eases system integration, verification and certification.

d) Deterministic Interconnect: In order to enable temporally deterministic behavior of the system, the network infrastructure interconnecting the LIFs of the different μComponents has to be temporally deterministic. The ACROSS architecture implements a time-triggered NoC that satisfies this requirement (see Section IV-B).

e) Fault and Error Containment: Fault and error containment is one of the most important objectives to enable efficient certification. The *fault hypothesis* in ACROSS, distinguishes *systematic design faults* and *physical random faults*. Systematic design faults are faults which are introduced during the design or implementation phase of the development process (e.g., a coding error in a given part of an application). In contrast, physical random faults are faults in any hardware component due to physical events like Single Event Upsets (SEUs) or faults created during the manufacturing or deployment phase.

A major part of a fault hypothesis is the definition of the *fault-containment regions (FCRs)* which constitute the units of failure in a system. An FCR is a set of subsystems that is assumed to fail independently from other FCRs [6]. For the ACROSS architecture we distinguish FCRs according to the two fundamental types of faults defined above. With respect to systematic design faults, each μComponent is considered as an FCR, which means, that any systematic design fault within a μComponent can neither disrupt the correct behavior of any other μComponent, nor the communication among other μComponents. Fault containment for design faults is achieved by the temporal and spatial partitioning mechanisms of the TTNoC described in Section IV-B.

With respect to physical random faults the entire MPSoC has to be considered as a single FCR, since common mode failures (e.g., a fault in the power supply of the MPSoC) cannot be totally avoided. For systems belonging to the highest criticality class, it has to be assumed that a single physical random fault can lead to a failure of the entire MPSoC. To tolerate random faults, the ACROSS project has developed a replica-deterministic Tripple Modular Redundancy (TMR)

scheme where redundant MPSoCs are interconnect via a dependable and redundant off-chip network. Off-chip TMR is not in the scope of this paper.

B. The Time-Triggered Network-on-Chip

The time-triggered network-on-chip (TTNoC) constitutes the core of the ACROSS MPSoC architecture. This section describes the basic functionality of the TTNoC and its properties.

a) Encapsulated Communication Channels: The TTNoC provides a configurable set of independent logical communication channels on top of a shared physical communication infrastructure. These communication channels provide spatial and temporal partitioning, and are therefore called *encapsulated communication channels*. Spatial partitioning means that messages sent on a given communication channel are only visible on that channel and cannot be overwritten by messages from other channels. Temporal partitioning means that the temporal behavior (e.g., bandwidth or latency) of any communication channel is independent of the communication activities of any other channel.

An encapsulated communication channel is unidirectional, has a defined sender, a defined set of receivers and is associated to a specific periodic or sporadic message. Communication is performed according to the time-triggered paradigm. Each µComponent has access to a consistent chip-wide notion of time which is called the *Macro-Tick*. All communication activities are driven by the Macro-Tick and follow an a-priori defined *time-triggered message schedule*. The time-triggered message schedule defines for each encapsulated communication channel: the sender, the set of receivers, the message length, and the periodic or sporadic send instants of the associated message with respect to the Macro-Tick.

The time-triggered schedule is defined at design time. It is free of any temporal conflicts, which means that there is no point in time where any two messages would collide on any shared physical link. The a-priory defined schedule has the advantage that no on-line arbitration is required. Whenever the message is scheduled, it is guaranteed that all the required links in the NoC from the sender to all receivers will be free, and the message can pass on without being blocked or discarded. Thus, the temporal behavior is ”designed”, into the system and not wearisomely analyzed after the implementation by complex methods (e.g., methods from queuing theory). This property eases certification to a significant extend.

b) TISS: The Trusted Interface Subsystem (TISS) implements a message based interface that acts as a temporal firewall [8] between each µComponent and the shared communication resources. It is the responsibility of the TISS to achieve temporal coordination of communication activities and task executions between different µComponents.

Therefore, all TISSes within the MPSoC share the same global notion of time (i.e., the Macro-Tick). Based on this global time communication activities are only allowed at predefined instants. The time-triggered schedule defines at which points in time a message has to be sent to/received from the TTNoC, or when tasks have to be triggered. In order to prevent a faulty host from disrupting the communication between correct µComponents, the memories in which the schedules are stored cannot be modified by the µComponent itself.

The implementation of the TISSes in the ACROSS MPSoC provides the following three interfaces (see Figure 1):

- **Network Interface (NI):** connects the µComponent to the TTNoC. It consists of an incoming and one outgoing lane; each 34-bit wide (i.e., 32-bit data signal, 2-bit control signal).
- **Port Interface (PI):** is the interface for the actual transfer of data between the TISS and the computing host of the µComponent. Messages that have to be sent are written by the host into a dual-ported memory, denoted as Port Memory. Similarly, the host reads messages that have been received from the Port Memory. Access to the Port Memory on the host side is typically not conducted at dedicated points in time. In contrast, the TISS reads/writes messages with fixed size according to the predefined schedule from the dual-ported Port Memory. In order to prevent data corruption in the Port Memory due to simultaneous access to the same memory space, synchronization between the host and the TISS is required. This synchronization is done via the control interface.
- **Control Interface (CI):** is used by the host to configure communication channels and core services (cf. Section IV-C), as well as to synchronize the message exchange between host and TISS.

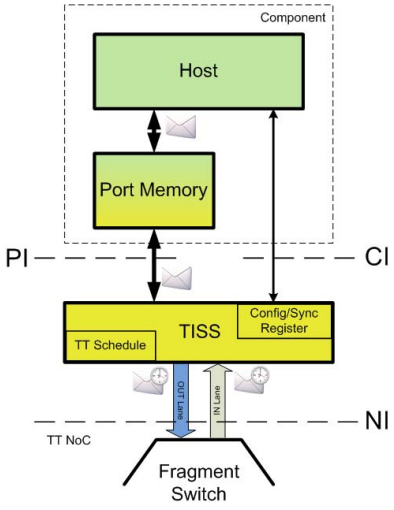

Fig. 1. Interfaces of the TISS

The Trusted Resource Manager (TRM) is the only µComponent within the MPSoC that is allowed to reconfigure the time-triggered schedules of all other µComponents. This enables dynamic changes of communication bandwidth and component interactions. The TRM uses encapsulated communication channels on the TTNoC to transmit reconfiguration

information to those TISSes for which the schedules have to be changed. These reconfiguration messages are directly interpreted by the TISS hardware, so that no interaction of the host is required. Because each TISS receives its reconfiguration data at a different point in time, the new schedule is not valid until all TISSes received their respective reconfiguration messages. Otherwise, inconsistencies in the schedules and the disruption of communication activities between µComponents could be the consequence. Due to the time-triggered schedule, the instant at which all TISSes received the new schedule is known in advance. Hence, a common reconfiguration instant, at which new schedules are activated, is scheduled at each TISS that is reconfigured.

c) Routing: Communication within the TTNoC is based on the exchange of *messages*. For schedulability reasons (e.g., one long message with low period has to be intercepted by some short message with high period) it might be necessary to split one message into several parts, which are called *fragments*. Each fragment is transmitted in a *packet* that contains a header and a data section. Furthermore, packets are decomposed into single *flits*, which is the smallest unit of information within the TTNoC (i.e., 32 bit in the implementation of the ACROSS MPSoC).

The TTNoC is composed of a mesh of interconnected *Fragment Switches*, which transport packets from an incoming port to up to three outgoing ports. Each fragment switch has 4 bi-directional interconnects, where an interconnect can be further decomposed into an incoming and one outgoing lane. A lane is a unidirectional bus which features a 32-bit *data* signal, a 1-bit *valid* signal, and a 1-bit *header* signal. The valid signal is used to signal when actual data is put on the data lines. To configure the input-output relation of each fragment switch, a message on the TTNoC carries header information. The header is identified by the header signal of each lane.

The interconnects of the fragment switches are either connected to other fragment switches to form the mesh network, or they are coupled to the network interface of a TISS. Hence, the route from the sender of a message to the receiver(s) is defined by the fragment switches that have to be passed. The TTNoC implements the following wormhole routing scheme for the transmission of messages:

Header information is sent at the beginning of each fragment. This is used by the fragment switches to switch the packet from the incoming lane (i.e., the lane at which the packet is received) to the outgoing lane(s) (i.e., the lane to which the packet has to be sent). The switching relation between incoming and outgoing lane(s) is valid until the end of the packet. As the header does not contain the destination address, but information for each fragment switch to which direction a packet has to be output, the usage of header information is consuming. This means, that it has to be removed from the header after it has been used.

Fragment switches are able to simultaneously transmit different packets into different directions. The only condition to allow more than one packet at a time is, that no output lane is shared between two simultaneous packets. Figure 2 presents examples of possible and impossible routes in the TTNoC.

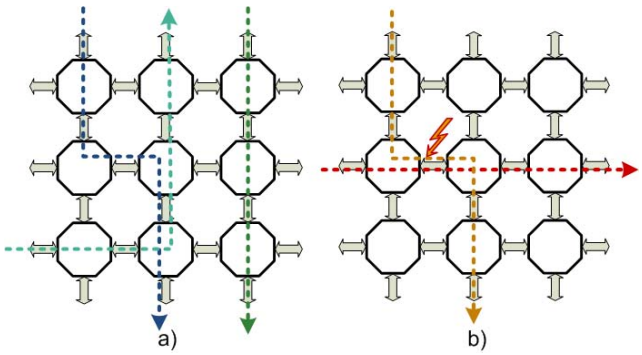

Fig. 2. Possible a) and impossible b) routes in the TTNoC

C. Hardware provided Core Services

The ACROSS project defines several core services that allow µComponents to communicate and to temporally coordinate distributed actions in the system. Most of the services are implemented as modular HDL building blocks and encompass services for:

a) Common Time Service: The most basic and important service is the Common Time Service. This service provides a common time base with a granularity of one Macro-Tick ($1\mu s$). The Macro-Tick is generated by a dedicated Macro-Tick generator and distributed to all TISSes.

The Common Time Service uses the Macro-Tick to drive the time-triggered schedule and a 64 Bit *Time-Stamp Counter*. The Time-Stamp Counter can be read by the host and other core services. Although the Common Time Service is instantiated multiple times (once at each TISS) the counters at each Common Time Service instance stay synchronous as they are all driven by the same Macro-Tick. During startup all counters are initialized with zero and a global reset line drives all Common Time Service instances. After releasing this reset line all Common Time Service instances start counting synchronously.

Exploiting this synchronicity each Common Time Service instance traverses the time-triggered schedule. When reaching a time instant defined in the time-triggered schedule, either the Basic Communication Service or the Task Trigger Service is activated.

b) Basic Communication Service: The Basic Communication Service is used to allow µComponents to exchange arbitrary data with each other. For communication the ACROSS MPSoC allows to configure up to 128 Ports per µComponent. Each Port provides a unidirectional one to many encapsulated communication channel. Data is transmitted or received based on the off-line computed periodic time-triggered schedule.

Each message is associated with a specific Port which type characterizes the transmission and reception behavior.

- **Periodic Ports** best fit to messages with state semantics, which are transmitted within each period even if the host does not actively transmit something. These ports do not

use any queues. Hence received data replaces old data. Integrity is ensured by executing the non-blocking writer protocol[9] for ingress *Ports*, and by using shadow buffers for egress *Ports*.

- **Conditional Ports** are similar to periodic Ports, except that they only send data if the host triggers it actively.
- **Sporadic Ports** are most likely used for communication with event semantics. Each side maintains a queue. Data is only transmitted if the egress queue is not empty.

This service can also drive interrupts at the host when a communication event happened. Depending on the configuration stored in the time-triggered schedule the Basic Communication Service can also insert the current time-stamp value to ingress streams.

The Basic Communication Service can not only be used by the hosts, but also by other services. For this the Basic Communication Service multiplexes the ingress/egress data stream either to the *Component Control Service*, *Health Reporting Service*, *Inter-Component Channel Configuration Service* or handles it itself by writing or reading the Port Memory as described. The information about the source or destination is taken from the off-line generated schedule.

c) Component Control Service: This service serves two purposes: First, it allows a dedicated a-priori defined µComponent (e.g., a diagnostic unit) to reset a single µComponent, if it detects a faulty behavior. Second, it can be used to trigger and to coordinate actions on distinct µComponents. Data received by this service is interpreted by a hardware module, which either triggers an interrupt at the host at a specific point in time, or it drives the reset line for a specific amount of time.

d) Error Detection Service: The Error Detection Service locally monitors the activities of the host and the TTNoC. This service detects failures like a periodic message miss, queue overflows of Sporadic Ports and invalid configuration states. In case of an error the Error Detection Service can drive an interrupt at the host.

A specific feature of the Error Detection Service is, that it can be configured read-only for the host. Doing so precludes the host to mask local faults. In this case the Health Reporting Service (see below) collects this information and sends it to a diagnostic unit.

e) Health Reporting Service: The Health Reporting Service collects error information from different µComponents and sends it to a diagnostic unit. Each host is allowed to insert application specific information in order to gather all the information that is useful for the diagnostic unit. If a host has read-only access to the Error Detection Service the Health Reporting Service is the only subsystem that is allowed to clear error conditions from the host.

f) Inter-Component Channel Configuration Service: For safety reasons the hosts are not allowed to modify the schedules within the TISS. The TRM is the only µComponent that can use this service to reconfigure the time-triggered schedule. Since there are applications with certification requirements that do not allow on-line reconfiguration, this service can be totally disabled. In this case the TRM not required in the MPSoC.

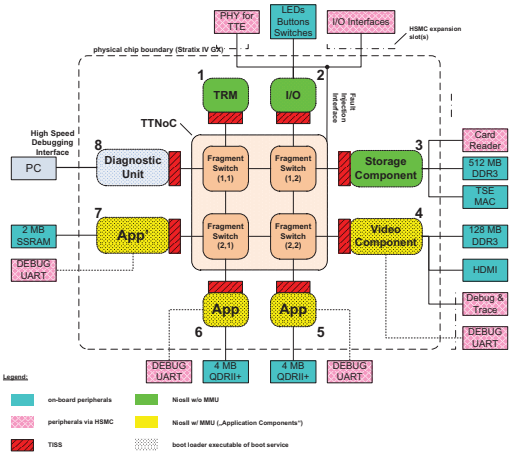

Fig. 3. MPSoC Topology

g) Task Trigger Service: Embedded control systems tasks often require periodic triggering. The Task Trigger Service manifests as a specific instant in the time-triggered schedule. When reaching this instant the TISS asserts an interrupt and provides a task number read from the schedule. The host can use this information to activate specific tasks.

h) Timer Interrupt Service: Each TISS implements a simple programmable timer that can be used by the host. This timer is driven by the Macro Tick to ease coordination with scheduled actions.

i) Basic Boot Service: Each µComponent is equipped with a small ROM where the host starts executing. The Basic Boot Service loads the application code from a dedicated storage µComponent to the hosts memory.

V. Prototype Implementation

This section describes the prototype implementation of the ACROSS MPSoC and its application on industrial demonstrators.

A. MPSoC Prototype

The ACROSS MPSoC prototype consists of a TTNoC with 4 fragment switches, 8 TISSes and µComponents (see Figure 3). The µComponents for the prototype implementation of the ACROSS MPSoC are implemented using Altera Nios 2 processors. To demonstrate the capability to integrate heterogeneous cores, one demonstrator also uses a specialized µComponent for high speed wireless communication.

Four of the integrated µComponents are system components and the other four are application components. The system components realize core system functions, i.e., off-chip communication (I/O component), file operations (Storage Component), debugging and tracing (Diagnostic Component), and TTNoC management (TRM). The application components are used to execute user defined applications. They run the real-time operating system PikeOS. PikeOS is a specially designed

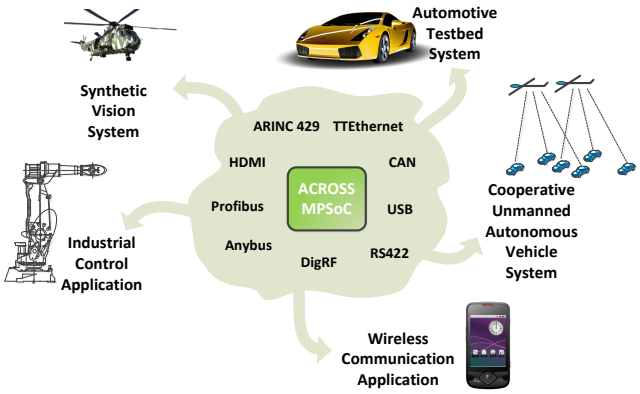

Fig. 4. Prototype Implementation

micro-kernel based real-time operating system for mixed-criticality embedded systems developed by Sysgo AG [10].

B. Demonstrators

To assess the capabilities of the ACROSS MPSoC, demonstrators are implemented considering real-world industrial requirements. The ACROSS project implements five industrial demonstrators which are an automotive testbed system, an industrial control application, a cooperative unmanned autonomous vehicle (UAV) system, a wireless communication system, and a synthetic vision system serving as a landing aid for helicopeters (Figure 4).

C. Prototype Hardware

The MPSoC as the main part of the prototype is implemented on an Altera Stratix IV GX (DK-DEV-4SGX230N) development board. The Stratix IV GX is a general purpose board for FPGA development equipped with a 1517-pin FPGA (EP4SGX230KF40) [11]. The FPGA technology was chosen as an efficient means for prototyping and as an intermediary step towards bringing down the architecture to an ASIC. The industrial demonstrators require a unique industry oriented set of communication protocols (see Figure 4), which are not supported by the standard Stratix IV GX board due to lack of hardware interfaces. Therefore, in ACROSS we have developed a system extension board, which provides hardware interfaces for these communication protocols. The hardware interfaces used by the industrial demonstrators are: Anybus, RS422, ARINC 429, M-BUS, DigRF, USB, Ethernet, Compact Flash Card, Debug/Trace, Fault Injection, HDMI, TTEthernet. The expansion board uses a HSMC interface to connect to the Stratix IV GX board.

VI. Related Work

The ACROSS MPSoC represents the latest milestone in the research efforts to develop a reference architecture for embedded systems. A series of research projects ultimately led to the ACROSS which is conducted under the wings of

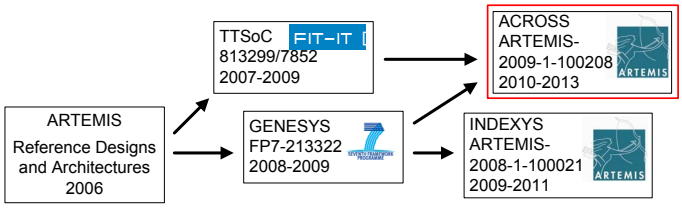

Fig. 5. Research line

the ARTEMIS joint undertaking (see Figure 5). The requirement for a SoC usable across different industrial domains was originally postulated in the ARTEMIS SRA [12]. This challenge was taken up by the GENESYS project [13] [14] which resulted in the specification of a reference architecture template also often called an 'architecture blueprint'. As a technical solution, the GENESYS project promoted an SoC which uses a time–triggered NoC to interconnect heterogeneous IP cores. The technological background was researched in the TTSoC project [15] [16]. After the completion of GENESYS, the INDEXYS project [17] took over the development by providing a proof–of–concept implementation. Finally, the ACROSS project delivers an industrial prototype, a development methodology, and to foster distribution of the key ideas within each domain. At the time of writing, the industrial prototype of the ACROSS MPSoC is used to build demonstrators from the automotive, avionics, and industrial control domains.

Related research efforts are concentrated in the RECOMP project [18] which strives to devise reference designs for certifiable, composent–based multicore systems. The CompSOC platform [19] aims to reduce system complexity by offering a virtual platform to each application. When putting an ACROSS MPSoC in the context of a larger system [20], security issues have to be considered. Some of issues are discussed in [21] and [22].

Time–triggered systems [8] are an accepted solution to build hard real–time systems [6]. The adoption of the time–triggered paradigm to build a deterministic NoC was first picked up in [23]. This approach is also pursued in the ACROSS MPSoC.

The ACROSS MPSoC architecture supports the integration of IP blocks according to the globally–asynchronous locally–synchronous (GALS) paradigm [24] [25]. In addition to a GALS glue logic, the ACROSS MPSoC introduces a Macro–tick at the system level in order to restore the determinism that is required for hard real–time applications.

It has been recongized [26] that there are many applications for NoCs that require deterministic bounds for communication delay and throughput. This is to be seen on the contrary to the bulk of current research on NoCs which target to maximize performance and throughput. Providing a guaranteed service is the subject of several research efforts like for instance AEthereal [27], Nostrum NoC [28], aSoC [29], QNoC [30], or SuperGT [31].

VII. Conclusion

In this paper we have introduced the ACROSS Multi-Processor System-on-a-Chip (MPSoC) architecture, which is targeted at embedded systems that have to be certified to the highest criticality levels. The ACROSS MPSoC enables the certification of dependable systems by lowering and managing the complexity of highly integrated designs. The major means for complexity management in ACROSS are (i) an increase in the level of design abstraction, (ii) establishment of temporally deterministic behavior and (iii) the prevention of error propagation and non-intended interference of subsystems by segregation mechanisms for temporal and spatial partitioning. We pointed out the limitation of currently existing MPSoC architectures with respect to safety-critical systems and their certification, and clearly show how the ACROSS architecture overcomes these limitations.

The entire design of the ACROSS MPSoC was driven by a large consortium of industrial and academic key-stake holders in the field of safety-critical systems. The feasibility of the architecture is demonstrated in five demonstrators in the avionics, automotive and industrial control domain.

Acknowledgment

This document is based on the ACROSS project in the framework of the ARTEMIS programme. The work has been funded in part by the ARTEMIS Joint Undertaking and National Funding Agencies of Austria, Germany, Italy and France under the funding ID ARTEMIS-2009-1-100208. The responsibility for the content rests with the authors. The authors would like to thank..

References

[1] I. W. Group. (2011) International technology roadmap for semiconductors. [Online]. Available: http://www.itrs.net/Links/2011ITRS/Home2011.htm

[2] F. J. Pollack, "New microarchitecture challenges in the coming generations of CMOS process technologies," in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1999, p. 2.

[3] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," in *AFIPS Conference Proceedings (30)*, 1967, pp. 483–485.

[4] J. Swingler and J. McBride, "The Synergistic Relationship of Stresses in the Automotive Connector," in *Proceedings of the 19th International Conference on Electric Contact Phenomena*, 1998, pp. 141–145.

[5] EASA. (2011, feb) Notification of a proposal to issue a certification memorandum. [Online]. Available: http://www.easa.eu.int/certification/docs/certification-memorandum/

[6] H. Kopetz, *Design Principles for Distributed Embedded Systems*, 4th ed. Kluwer Academic Publishers, 1997.

[7] *S+ IEC 61508*, IEC Std., Rev. 2.0, 2010. [Online]. Available: http://webstore.iec.ch/webstore/

[8] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," in *Proceedings of the IEEE Special Issue on Modeling and Design of Embedded Software*, October 2001.

[9] H. Kopetz and J. Reisinger, "The non-blocking write protocol NBW: A solution to a real-time synchronization problem," in *Real-Time Systems Symposium, 1993., Proceedings.*, dec 1993, pp. 131 –137.

[10] *PikeOS Datasheet*, SYSGO AG, 2012. [Online]. Available: http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/embedded-virtualization/

[11] *Stratix IV GX FPGA Development Board: Reference Manual*, Altera Corporation, 2010. [Online]. Available: http://www.altera.com/literature/manual/

[12] ARTEMIS. (2011) ARTEMIS Strategic Research Agenda 2011. [Online]. Available: http://www.artemis-ia.eu/publication/download/publication/541

[13] R. Obermaisser and H. Kopetz, Eds., *GENESYS: A Candidate for an ARTEMIS Cross-Domain Reference Architecture for Embedded Systems*. Südwestdeutscher Verlag für Hochschulschriften (SVH), 2009.

[14] R. Obermaisser, H. Kopetz, and C. Paukovits, "A Cross-Domain Multi-Processor System-on-a-Chip for Embedded Real-Time Systems," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. pp. 548–567, 2010.

[15] C. El-Salloum, "Interface design in the time-triggered system-on-chip architecture," Ph.D. dissertation, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2007.

[16] C. Paukovits, "The Time-Triggered System-on-Chip Architecture," Ph.D. dissertation, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, dec 2008.

[17] A. Eckel, P. Milbret, Z. Al-Ars, S. Schneele, B. Vermeulen, G. Csertaán, C. Scheerer, N. Suri, A. K. G. Fohler, R. Obermaisser, and C. Fidi, "INDEXYS, A Logical Step Beyond GENESYS," in *Proceedings of the 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, 2010, pp. 431–451.

[18] P. Pop, "Component model specification," RECOM Project, Deliveranble D2.1, 2011.

[19] A. Hansson and K. Goossens, *On–Chip Interconnect with aelite: Composable and Predictable Systems*. Springer, 2011.

[20] A. Wasicek, C. E. Salloum, and O. Höftberger, "Embedding Complex Embedded Systems in Large Ethernet-based Networks," in *ERCIM Workshop on Dependable Cyber-physical Systems, SAFECOMP*, 2011, Research Report.

[21] A. Wasicek and C. El-Salloum, "A System–on–a–Chip Platform for Mixed–Criticality Applications," in *Proceedings of 13th IEEE International Symposium on Object/component/service-oriented Real-time distributed computing (ISORC)*, May. 2010, pp. 210–216.

[22] ——, "End-to-End Encryption in the TTSoC Architecture," in *Proceedings of the 3rd Workshop on Embedded Systems Security, ESWEEK'08, Atlanta, USA*, October 2008.

[23] C. Paukovits and H. Kopetz, "Building Encapsulated Communication Channels in the Time-Triggered System-on-Chip Architecture," Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Research Report 8/2009, 2009.

[24] P. Teehan, M. Greenstreet, and G. Lemieux, "A Survey and Taxonomy of GALS Design Styles," *Design & Test of Computers, IEEE*, vol. 24, no. 5, pp. 418–428, 2007.

[25] E. G. Friedman, "Clock Distribution Networks in Synchronous Digital Integrated Circuits," in *Proc. IEEE*, 2001, pp. 665–692.

[26] R. Marculescu, U. Y. Ogras, L.-S. Peh, N. E. Jerger, and Y. Hoskote, "Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 1, pp. 3–21, Jan. 2009.

[27] K. Goossens, J. Dielissen, J. van Meerbergen, P. Poplavko, A. Radulescu, E. Rijpkema, E. Waterlander, and P. Wielage, "Guaranteeing the quality of services in networks on chip," in *Networks on Chip*. Kluwer, 2003, pp. 61–82.

[28] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, vol. 2, 2004, pp. 890–895.

[29] J. Liang, S. Swaminathan, and R. Tessier, "ASOC: a scalable, single-chip communications architecture," in *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference on*, 2000, pp. 37 –46.

[30] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," *J. Syst. Archit.*, vol. 50, no. 2-3, pp. 105–128, Feb. 2004.

[31] T. Marescaux and H. Corporaal, "Introducing the SuperGT Network-on-Chip; SuperGT QoS: more than just GT," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, June 2007, pp. 116 –121.