

parMERASA

Multi-Core Execution of Parallelised Hard Real-time Applications Supporting Analysability

D5.1 - Hardware Requirements

Nature:	Report
Dissemination Level:	Public
Due date of deliverable:	30 – June – 2012
Actual submission:	30 – June – 2012
Responsible beneficiary:	TUDO
Responsible person:	Sascha Uhrig

Grant Agreement number:	FP7-287519
Project acronym:	parMERASA
Project title:	Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability
Project website address:	http://www.parmerasa.eu
Funding Scheme:	STREP SEVENTH FRAMEWORK PROGRAMME THEME ICT – 2011.3.4 Computing Systems
Date of latest version of Annex I against which the assessment will be made:	July 14, 2011
Project start:	October 1, 2011
Duration:	36 month
Periodic report:	First Periodic Report
Period covered:	2011-10-01 to 2012-06-30

Project coordinator name, title and organisation:	Prof. Dr. Theo Ungerer, University of Augsburg
Tel: + 49-821-598-2350 Fax: + 49-821-598-2359	Email: ungerer@informatik.uni-augsburg.de

Release Approval

Name	Role	Date
Eduardo Quiñones	WP leader	30 – June – 2012
Theo Ungerer	Coordinator	30 – June – 2012

DELIVERABLE SUMMARY

Deliverable 5.1 “Hardware Requirements” covers an overview of currently available Commercial-off-the-Shelf (COTS) multi-core processors, a collection of requirements defined by the target applications that affect work package 5, and an initial parMERASA many-core architecture proposal upon which the parMERASA research will be conducted. Since the parMERASA many-core processor design must be suitable to execute sample applications from three different domains (automotive, avionics, and construction machinery), its architecture must satisfy a diversity of needs. Therefore, a common basic architecture with only a few degrees of freedom has been defined by analysing these needs together with the features of existing solutions (COTS processors).

The requirements derived from the three target application domains are condensed into *Consolidated Requirements* for work package 5 (CR5) which will form the base for the parMERASA hardware architecture design. Moreover, these CR5s will serve as success criteria at the end of the project in order to demonstrate the suitability of the parMERASA hardware architecture to be used for the targeted sample applications.

The deliverable spans nine month of work and handles the work done in task 5.1 to reach milestone MS12:

Task description of T5.1 (m1 :: 9m):

Study of hardware requirements and architectural trade-offs (see DoW, sect. 1.3.3, p. 53/54)

- Evaluation of current multi-core commercial off-the-shelf (COTS) processors used in high performance embedded domains (e.g. 8-core Freescale P4080, 64-core Tilera, etc.). Evaluations will be based on existing documentation, papers, white papers and data sheets.
- Selection of ISA (TriCore, SPARCv8 or PowerPC) considering the COTS processor analysis, requirements of applications in WP2, and partner expertise.
- Requirement definition and success criteria of interconnection networks, memory hierarchy and I/O devices.

Conclusion of task 5.1:

The intended subtasks have been carried out successfully and MS12 has been reached.

Milestone 12 (see DoW, sect. 1.3.3, p. 56):

- Requirement specification and concept results of multi-core architecture design space exploration finalised.
- Decision taken on the supported Instruction Set Architecture (ISA).
- Preliminary multi-core simulator available.

All objectives of MS12 have been reached. The first two parts of MS12 are documented in this deliverable; the third part is included in D5.2.

TABLE OF CONTENTS

1	Overview of Current Multi- and Many-Core COTS Processors.....	7
1.1	Freescall P4080.....	7
1.1.1	Virtualization Mechanisms	8
1.1.2	Memory Architecture	8
1.2	Tilera Gx and Pro Series.....	8
1.2.1	Memory Architecture	9
1.2.2	Tilera iMesh Network-on-Chip	10
1.3	Intel Single-Chip Cloud Computer (SCC).....	10
1.3.1	Message-passing Communication Mechanism	11
1.3.2	System Address Lookup Table	12
1.3.3	Network on Chip.....	12
1.4	Suitability of COTS Processors for the parMERASA Target Applications.....	13
2	Requirements from other work packages.....	13
2.1	General Architecture and Programming Model.....	14
2.1.1	Requirements from System Software (WP4)	14
2.1.2	Requirements from the Target Applications (WP2)	14
2.2	Core and Instruction Set.....	15
2.3	Network on Chip (NoC).....	17
2.4	Interrupts and Global Synchronized Time Basis.....	17
2.5	Peripheral I/O	19
2.6	WCET Analysis	20
3	Proposed parMERASA Architecture	21
3.1	Memory Hierarchy	21
3.1.1	Local Code Memory.....	22
3.1.2	Local Data Memory	23
3.1.3	Global Memory.....	25
3.1.4	Arrangement of Memories.....	25
3.1.5	Support for Synchronization.....	26
3.2	Demands on the Network on Chip (NoC)	27
3.3	IO/IRQ Mechanism	32
3.3.1	Interrupt Request Mechanism	32
3.3.2	Required IO Devices	33

4	Requirements – Summary	34
4.1	Consolidated Requirements (Internal)	34
4.2	Requirements to Other Work Packages	35

1 OVERVIEW OF CURRENT MULTI- AND MANY-CORE COTS PROCESSORS

This section presents an evaluation of three multi-core processor architectures, i.e. Freescale P4080, Tilera processor families (Gx and Pro) and Intel SCC. The evaluation has been done based on existing documentation, papers, white papers and data sheets as stated in task 5.1 of the DoW.

The multi-core processors presented in this section have been selected in order to cover different system domains. Thus, the Freescale P4080, which has attracted the attention of avionic industries, is intended to be used in critical real-time systems. Tilera processors are mainly used in high performance embedded systems such as network domains and multimedia. Finally, the Intel SCC is a processor prototype intended to promote many-core processor and parallel programming research.

Overall, the material presented in this section represents an excellent starting point for the parMERASA many-core processor research.

1.1 Freescale P4080

The Freescale P4080 multi-core processor, which belongs to the QorIQ communication processor family based on PowerPC Architecture, integrates eight e500mc cores connected through *CoreNet™*, a coherency fabric Network-on-Chip (NoC) proprietary of Freescale¹. The CoreNet NoC, which is one of the key design components of the P4080, integrates a queue manager fabric at packet-level and supports cache coherent and non-coherent transactions amongst CoreNet end-points.

Figure 1 shows the top-level block diagram of the Freescale P4080 processor architecture [1].

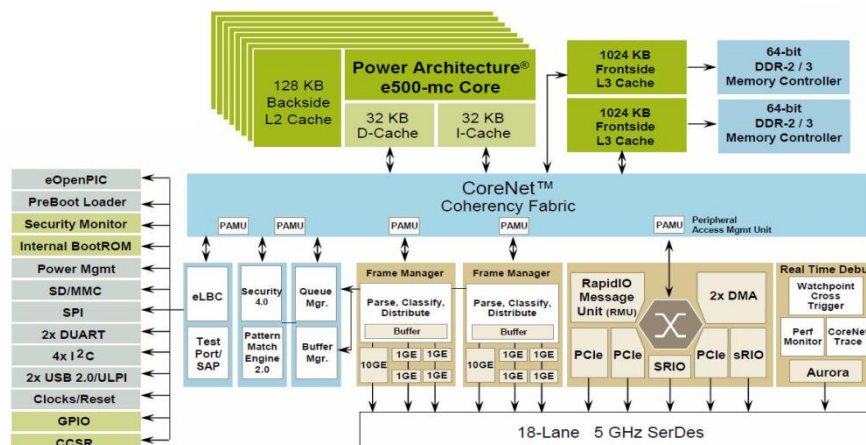


Figure 1. Freescale P4080 processor architecture block diagram

The technical documentation of the P4080 is limited and it lacks of important technical details, especially those concerning the CoreNet design. Therefore, for the scope of the parMERASA project we have focused our attention only on the memory architecture and the virtualization mechanisms of the P4080.

¹In this document we use the terms on-chip interconnection network, and Network-on-Chip (NoC) interchangeably. NoC refers to any on-chip communication infrastructure, e.g. bus or mesh.

1.1.1 Virtualization Mechanisms

The P4080 processor architecture provides virtualization technology that allows cores to work as eight symmetric multi-processing (SMP) cores, eight asymmetric multi-processing (AMP) cores, or as a combination of SMP and AMP core groupings. This allows the P4080 to run different operating systems, allowing the user to partition applications with different timing and functional requirements.

Moreover, the virtualization technology allows to partition the access to hardware resources through an *embedded hypervisor*, guaranteeing that the software running on any core can have access only to a set of hardware resources (e.g. memory, peripherals) that it is explicitly authorized. The embedded hypervisor enables safe and autonomous operation of multiple individual operating systems, allowing them to share system resources, including processor cores, memory and other on-chip functions.

1.1.2 Memory Architecture

The P4080 processor architecture defines a globally shared 36-bit physical address space for both instructions and data, which can be mapped to local and external address spaces.

Each core implements private L1 caches for instructions and data of 32 KB each, and a private L2 cache of 128 KB. All caches run at the same frequency as the core. The L2 cache provides support for way partitioning allowing to assign a certain number of ways to allocate instruction misses and a certain number of ways to allocate data misses, effectively allowing to configure L2 cache as instruction, data, or unified. CoreNet implements hardware mechanisms that provide a cache-coherent view of the shared memory.

P4080 also features two shared L3 caches of 1 MB each that form the shared L3 *CoreNet Platform Cache* (CPC). CPC can function as a general purpose write-back/write-through cache with I/O capabilities, a memory mapped SRAM device or a combination of both of these functions. I/O capabilities allocate writes from I/O devices to main memory into cache, in order to reduce latency and improve bandwidth for future read operations to the same address. Two DDR memory controllers supporting DDR2 and DDR3 SDRAM modules are connected to each of the L3 caches. This allows each L3 cache to cache only the address range that is covered by the memory controller behind it (see figure 1).

1.2 Tiler Gx and Pro Series

The Tiler processor architecture family, including Gx and Pro series, consist of an on-chip 2D-mesh connecting identical cores (or tiles). The number of cores integrated into a single chip ranges from 16 to 64 depending on the specific processor configuration.

Each tile is composed of a core implementing a 32-bit three-wide VLIW integer processor engine that includes a memory management unit with *Translation Lookaside Buffers* (TLBs) and a two-level cache hierarchy. A hardware mechanism called *Dynamic Distributed Cache* (DDC) maintains cache coherence among all cores and I/O memory accesses.

One of the main characteristic of Tiler processors is the Tiler iMesh™ network on chip (NoC). The iMesh is in fact formed by five different 2D meshes, each specialized for a different use: the *User Dynamic Network* (UDN), the *I/O Dynamic Network* (IDN), the *Static Network* (STN), the *Memory*

Dynamic Network (MDN) and the *Tile Dynamic Network* (TDN). Each network is in charge to connect the tiles among them and among the different I/O devices and memory controllers located in the perimeter of the NoC.

Figure 2 shows the top-level block diagram of the Tiler Pro64 processor architecture [2].

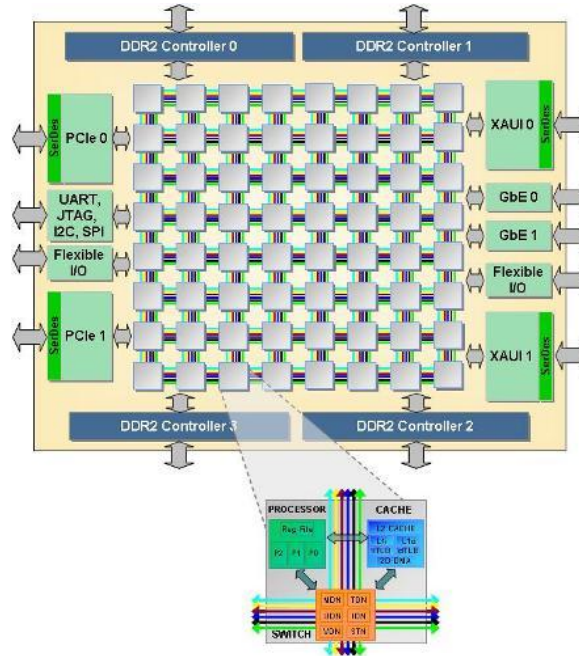


Figure 2. Tiler Pro64 processor architecture block diagram

For the scope of the parMERASA project, we are especially interested on the memory architecture and the iMesh NoC.

1.2.1 Memory Architecture

The Tiler processor architecture defines a flat, globally shared 64-bit physical address space for both instructions and data. It implements a non-blocking two-level cache hierarchy that supports multiple concurrent outstanding memory operations. Hence, the execution engine does not stall on load or store cache misses. Rather, execution of subsequent instructions continues until the data requested by the cache miss is actually needed by another instruction. Moreover, the cache subsystem supports *hit under miss* and *miss under miss*, allowing loads and stores to different addresses to be re-ordered.

The Tiler implements a hardware cache coherence mechanism named *Dynamic Distributed Cache* (DDC). The DDC allows a shared memory page to be homed on a specific tile (or distributed across many tiles), being cached remotely by other tiles. This mechanism allows a tile to view the collection of on-chip caches of all tiles as a large shared, distributed coherent cache. To do so, each tile contains a directory that specifies which cache lines are shared among the different cores.

Moreover, the cache subsystem supports using portions of the L2 cache as a scratchpad memory, and it allows I/O to read and write the on-chip caches directly. Finally, it provides atomic instructions (test-and-set) and memory fences (MF) to implement synchronization primitives. .

1.2.2 Tiler iMesh Network-on-Chip

The Tiler family of processors implements an on-die 2D network on chip called *iMesh* that distributes memory system traffic, cache system traffic, I/O traffic and software based messaging among tiles, memory and I/O devices. The iMesh is composed of five 2D meshes classified into two groups:

- *Memory Networks* handle all memory traffic coming from cache misses, DDR2 requests and DDC requests. They consist of the MDN, the TDN, and the CDN.
- *Messaging Networks* handle explicit software communication, allowing to send messages between tiles and I/O devices. They consist of the UDN and the IDN.

Tiler iMesh transmits data via packets, which are composed of multiple flits. Concretely, each packet contains a header flit designating the destination and the size of the packet, and a payload of data flits. Packets are routed using a X-Y routing algorithm and a *wormhole switching*: the header flit locks down the granted output port until the final flit of the packet has successfully traversed the router. For large packets, this type of routing may result in the reservation of multiple output ports of different routers simultaneously for the same packet. The router implements a round-robin output port arbitration, providing equivalent fairness for all input ports.

Flow control between neighbouring routers is implemented via a credit-based scheme. Each output port contains a credit count corresponding to how many available entries the neighbouring input port has available. When a flit is routed through an output port, the credit count is decremented. If the credit count is zero, the flit is blocked and cannot proceed. When an input port consumes a flit, a credit is returned to the corresponding output port. The flow control allows each router to hold up to three flits.

The Tiler networks operate at the same frequency as the processor cores. The latency for a flit to be read from an input buffer, traverse the crossbar, and reach the storage at the input of a neighbouring router is one cycle, thus allowing each flit to “hop” from one router to a neighbouring router in one cycle.

1.3 Intel Single-Chip Cloud Computer (SCC)

The *Single-Chip Cloud Computer* (SCC) [3] is a research microprocessor created by Intel in 2009. The SCC microprocessor contains 48 P54C Pentium cores grouped into 24 tiles. Each tile contains 2 cores and a message passing buffer (MPB) shared by the two cores which provides hardware support for message-passing communication. Tiles are connected with a 4×6 2D-mesh, giving access to four on-chip DDR3 memory controllers connected in the perimeter of the mesh. The SCC includes innovations for scalability in terms of energy-efficiency including techniques that enable software to dynamically configure voltage and frequency to attain power consumptions from 125W to 25W. Figure 3 shows the top-level block diagram of the Intel SCC processor architecture.

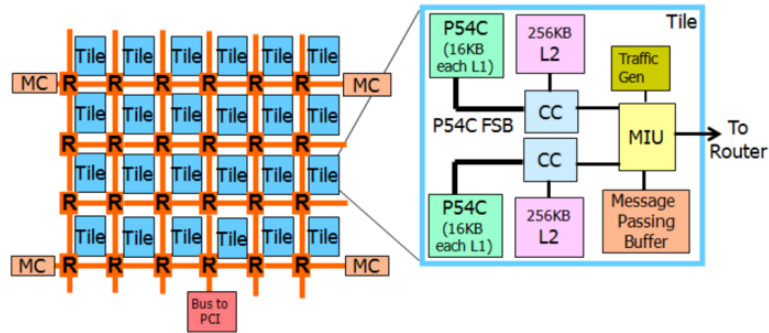


Figure 3. Intel SCC processor architecture block diagram

The Intel SCC has been designed to promote many-core processor and parallel programming research to improve average performance of a variety of high performance applications, including web servers, physics modelling and financial analytics. Therefore, it includes several non-predictable techniques that make it hard to guarantee time predictability. In fact, its name, Single-Chip Cloud Computer, reflects an architecture resembling a scalable cluster of computers (tiles) communicating through message passing such as you would find in a cloud, integrated into a single chip. However, the Intel SCC contains interesting architectural features for safety critical real-time systems. In a recent paper published in the 4th MARC symposium in 2011 [4], authors proposed an execution model consisting of a concurrent periodic task set that reduces the non-predictable timing behaviour of the Intel SCC.

For the scope of the parMERASA project, we are especially interested on three processor resources: The hardware support to message-passing communication, the network on chip, and the system address translation, named *lookup table*.

1.3.1 Message-passing Communication Mechanism

The Intel SCC provides two levels of cache per core, one instruction and data caches of 16 KB each and a unified L2 cache of 256 KB implementing write-back and write non-allocate policies. Each core can have only one outstanding memory request. In case of a read, the core is stalled until the data is returned. In case of a write, the core can continue its operation until another memory request either read or write occurs, in which case the core will be stalled.

Caches do not provide hardware cache coherence support in order to simplify the design and reduce power consumption. Instead, message-passing communication is used in order to provide a software-based coherent shared memory on the SCC. Thus, in addition to traditional caches, each tile has access to a message-passing buffer (MPB) of 16 KB which provides the equivalent of 512 cache lines of memory, in which shared data is hold. MPB cache lines can be stored in L1 cache but not in L2 cache, since the MPB is defined as non-cacheable regarding the L2 cache.

In order to support message passing, the instruction set architecture has been extended with a new instruction (CL1NVMB) that invalidates all lines in the L1 cache that contains MPB data. In order to identify them, a new status bits has been introduced in each L1 cache entry (MPBT), which is set when a new cache line from the MPB is fetched. When CL1NVMB is executed, it flushes all L1 cache lines with the MPBT set in one cycle; in case the line was modified the data is not written back into memory and so the data is lost.

1.3.2 System Address Lookup Table

In the Intel SCC, the main memory address space can be mapped to be shared among all, some, or none of the cores by the system. To do so, the software has the ability, at any time, to change the memory map for a core by configuring the *Lookup Table (LUT)*.

The LUT unit performs the address translation from core address to system address. The operating mode of this unit is governed by the tile level configuration registers. Two LUTs, one per each core, are used to translate all outgoing core addresses into system addresses. Each of the four memory controllers provides access to from 4GB to 16GB of main memory, depending on the density of the DIMMs used, for a total of up to 64GB. Each core has 32 address bits capable of addressing only 4GB.

The 4GB core address space is divided into 256 16MB pages ($256 * 16\text{MB} = 4\text{GB}$), for system address translation. Each page has an entry in the LUT that provides routing and address translation information.

Accesses to the shared address space can be marked as non-cacheable by the core or marked cacheable and the coherency between multiple caches from different cores must be handled in software.

1.3.3 Network on Chip

The Intel SCC integrates an on-die 2D mesh network composed of 24 packet-switched routers connected in a 6x4 configuration to communicate the 24 tiles. The mesh implements its own power supply and clock source.

Routers (named RXB) operate at 2GHz with a response time of 4 cycles (including link traversal). Links have a width of 16 bytes per data and 2 bytes per control. The RXB implements eight virtual channels (VC); only two of them are reserved for the two classes of messages: request and response message.

In the SCC, the communication through the mesh fabric is performed at packet granularity, formed by up to three flits. Packets are routed using a X-Y routing algorithm. The flow control is managed through credit-based across RXB routers, so a router can send a packet to another router only when it has a credit from that router. Each router has eight credits per port which are automatically routed back to the sender when the packet moves on to the next destination. Control flits are used to communicate credit information.

Each tile implements a *mesh interface* unit (MIU) that interfaces with the router (through a clock crossing FIFO due to different clock domains). It performs the following operations: packet management, command interpretation, address decode/lookup, local configuration registers, flow control, credit management and arbitration among different tile agents to access the router.

The MIU handles all memory and message passing requests coming from both, tile and router:

- *Tile to MIU interface.* It handles cache misses by decoding their addresses using the LUT to map from core address to system address. It then places the requests into the appropriate queue: main memory (DDR3) request, message passing buffer access, or local configuration register access.

- *Router to MIU.* It routes the data to the appropriate local destination. The link level flow control ensures flow of data on the mesh using a credit-based protocol. Finally, the arbiter controls tile access via a round robin scheme.

1.4 Suitability of COTS Processors for the parMERASA Target Applications

Despite the different system domains considered in the design of each of the processors presented above, i.e. critical real-time systems in case of the P4080, high performance embedded systems in case of the Tiler, and parallel high performance applications in case of the Intel SCC, the three designs contain interesting similarities that are relevant for the parMERASA research.

Hence, they implement NoC designs that guarantee the scalability of the processor. In case of Tiler and Intel SCC, both incorporate a 2D mesh implementing X-Y routing, credit-based flow control and wormhole switching. Interestingly, they follow a different approach to manage the different messages: While Tiler implements a completely separated NoC for each message type, Intel SCC implements different virtual channels per each message type. This message separation (either physically or virtually) helps to reduce the conflicts that the different messages may suffer when accessing to NoC resources, being of special interest of WCET analysis. With respect to the NoC implementation of the P4080, we have not been able to obtain trustworthy technical information about it.

Another interesting feature shared among the three processors is the implementation of a unified shared address space supported by a multiple level cache coherence protocol. However, the techniques used to provide a unified address space differs on the processor considered. Tiler and P4080 implement a flat global address space supported with hardware cache coherence mechanisms. Intel SCC provides, instead, different address spaces for cores and system (unified though the LUT) supported with hardware message-passing communication methods and software cache coherence mechanism. Interestingly, Tiler also provides a dedicated mesh (UDN) to communicate cores through message-passing.

Finally, the P4080 provides virtualization techniques, not supported in the other processor designs, which provides the time and space isolation required for IMA and AUTOSAR systems. Although this could be relevant for the parMERASA research, there exists very few publicly available information so we have not been able to obtain trustworthy technical information about it.

Overall, many of the processor features presented in this section are of interest of parMERASA research, representing an excellent starting point.

2 REQUIREMENTS FROM OTHER WORK PACKAGES

In this section, all requirements from other work packages with impact on the parMERASA hardware design are collected and translated into internal WP5 requirements, the so-called "consolidated requirements" (CR5.x). The reason for this translation is three-fold:

- Many requirements are closely related from a hardware point-of-view and can be condensed to a single requirement.

- The requirements provided by other work packages are expressed in a higher abstraction-level and so they need to be translated to concrete hardware requirements.
- Each application domain and research field uses its own nomenclature. Here, all are translated into the hardware specific nomenclature.

Additionally, the consolidated requirements serve as work plan and success criteria for the development of the predictable multi-core processor that will be delivered at the next milestone.

2.1 General Architecture and Programming Model

In this section we identify the parallel programming model used by the target applications: shared memory or message passing. Moreover, it must be clarified how multiple applications can be mapped to the parMERASA architecture and how they communicate internally and among each other.

2.1.1 Requirements from System Software (WP4)

There are four requirements from the system level software that affects the hardware architecture. It shall provide:

- *[R4.2] A physical addressable, fast, and efficient memory inside a partition, which is accessible via shared address space.*
- *[R4.3] A memory management unit (MMU) for memory protection and static mapping.*
- *[R4.4] A message passing mechanism between partitions.*
- *[R4.7] Hardware must provide the O/S with the ability to restrict for each individual **partition**: (1) memory spaces, (2) processing time², and (3) access to I/O (in order to isolate multiple partitions in a shared resource environment).*

These requirements lead directly to the following WP5 consolidated requirements:

- **CR5.1: Cores can be grouped into several virtual clusters. The number of cores per cluster can be configured individually per cluster but it cannot be changed during program execution.**
- **CR5.2: Each cluster has access to a shared memory region that is protected from accesses of other clusters.**
- **CR5.3: There is an explicit communication between clusters.**

The design space for the parMERASA many-core processor is bounded by the above mentioned consolidated requirements and lead us to several possible hardware designs. Section 3.2 discusses about the different designs alternatives.

2.1.2 Requirements from the Target Applications (WP2)

In WP2 there are also some direct requirements on the hardware:

- *[R2.H01] (to WP 4 and 5): Avionics applications shall be implemented as individual partitions.*
- *[R2.D10] (to WP 4 and 5): The system architecture shall provide shared memory that is shared between all cores that are used to execute one Atomic Software-Component.*

² The global processing time is not restricted any more. Partitions will run in parallel. Instead, they will provide time partitioning in the sense that the timing behaviour of one partition will not be affected by other partitions.

- *[R2.D11] (to WP 4 and 5): The system architecture shall allow partitioning between clusters of cores.*
- *[R2.D12] (to WP 4 and 5): The partitions shall have no direct write access to each other's memory.*
- *[R2.D13] (to WP 4 and 5): It shall be possible to map a partition to a cluster of cores.*
- *[R2.D14] (to WP 4 and 5): It shall be possible to map a Software-Component to a partition.*
- *[R2.D15] (to WP 4 and 5): It shall be possible to map a Runnable to a core or set of cores of one cluster.*
- *[R2.D16] (to WP 4 and 5): It shall be possible to execute a Runnable with data parallelism on a set of cores in parallel.*
- *[R2.D31] (to WP 4 and 5): The system software shall support routing between Software-Components in one cluster and field bus in another cluster.*

These requirements address both the system architecture (WP4) and the hardware architecture (WP5). WP4 already condensed these requirements into the requirement mentioned in the previous section and consequently the consolidated requirements CR5.1 to CR5.3 include these requirements. This section will justify that inclusion.

An AUTOSAR application consists of several "Software-Components" that are connected via the "Virtual Function Bus". Each Software-Component consists of several "Runnables" that are statically scheduled at the time of system design.

In parMERASA we map each software component to a cluster [R2.D14]. Thus each software component can use one or more cores [R2.D14] with a shared memory [R2.D10]. The clusters are partitioned in groups of cores [R2.D11], thus there are no direct write accesses across cluster boundaries allowed [R2.D12]. If runnables contain data parallelism, one runnable can be mapped to more than one core [R2.D17] within a cluster [R2.D16].

Therefore, an architecture with isolated clusters of cores that allows explicit messages between clusters and shared memory accesses within a cluster fulfils all requirements from the automotive domain concerning the hardware architecture.

2.2 Core and Instruction Set

There are only few formal requirements concerning the processor cores and their instruction sets. One of these is:

- *[R2.D09] (to WP 4): The programming language shall be C, potentially with annotations for parallelism.*

Consequently, only an instruction set that is supported by a C compiler should be chosen. Because there are C compilers for nearly every modern instruction set available, this requirement can be satisfied easily. We chose the PowerPC instruction set, which is commonly used in many high performance embedded processors (especially embedded multi cores). Additionally, its documentation is freely available and there are lots of open source compilers and processor hardware simulators for this architecture. Finally, nearly every project partner has experience with it. Therefore, a consolidated requirement demands the PowerPC instruction set architecture:

- **CR5.4 The hardware prototype supports the PowerPC instruction set.**

There are many variants of the PowerPC instruction set, in the project the instructions of the PowerPC 750 will be provided. The 750 is a 32 bit high performance embedded processor with floating point support.

Another positive aspect of the PowerPC instruction set is the fact that there are enough unused entries in the instruction opcode table to add customized instructions. This feature is needed to fulfil a requirement from the system level architecture:

- *[R4.1] The hardware shall provide an atomic read-modify-write instruction, preferably a fetch & add instruction.*

Such an instruction is used to build basic synchronization primitives like spin locks, mutexes and barriers. Unfortunately, the PowerPC instruction set provides no atomic read-modify-write instructions like *test-and-set*, *compare-and-swap* or *fetch-and-add*. Instead it implements the *load-linked-store-conditional* instruction pair (the assembly *lwarx/stwcx* PowerPC instructions), which provides an efficient way to synchronize threads on average case. Although the instruction pair could be used to emulate atomic *read-modify-write* instructions with a sequence of conventional instructions, it prevents to provide WCET analysis.

That is, the emulating code is essentially a loop that begins with a load-linked and ends with a store-conditional. Depending on the instruction that should be emulated, there are some conventional instruction between the load-linked and the store-conditional. The store-conditional instruction checks, if there was a memory access to the same address since the load-linked was executed and repeats the loop until there was none. In other words, it repeats the sequence until it was executed atomically. But the number of repetitions cannot be bounded and hence the emulation is unfeasible for WCET analysis.

There might be a solution to do the synchronization with a load-linked store-conditional pair, but this is not obvious and will be investigated within the project. But the system software must be provided very early within the project and it needs synchronization primitives. Therefore, a new *atomic read-modify-write* instruction will be added to the PowerPC instruction set. At the time of writing this document, the *fetch-and-add* instruction is the preferred one, although other alternatives such as *compare-and-swap* will be considered. We have to add such an instruction to the PowerPC instruction set:

- **CR5.5 The PowerPC instruction set is extended by an atomic read-modify-write instruction.**

It can be argued, that another instruction set that directly provides a *fetch-and-add* instruction should be chosen, but none of the common instruction set architectures for embedded systems provides this instruction.

The last requirement with influence on the core architecture covers the handling of invalid operations:

- *[R4.9] The processor provides a mechanism to transfer control to the O/S if the partition attempts to perform an invalid operation.*

This requirement states the need for a defined reaction in case of a misbehaviour of the application or a hardware fault. Fault tolerance assuming systematic faults (software bugs) and hardware faults is not in the objectives of parMERASA project. In the following it is assumed that the implemented sample applications will behave correctly with respect to the used input data sets and no misbehaviour of the applications will occur during the simulations. Hence, a technique as requested by [R4.9] will not be included into the simulator.

2.3 Network on Chip (NoC)

Timing guarantees in a network on chip can only be given, if there are some restrictions on the communication schedule. But these schedules may change in different phases of the program execution. For example, during start-up the communication is different from the communication in normal operation. Therefore, the following requirement demands a switch mechanism between different communication schedules:

- *[R2.D37] (to WP 5): It shall be possible to switch between different communication patterns with fixed relative timing and a given offset.*

Leading to the following consolidated requirement:

- **CR5. 6: It is possible to change between several communication schedules.**

Furthermore there are requirements, dealing with the transmission consistency in the interconnection network:

- *[R2.D19] (to WP 4 and 5): The communication infrastructure shall provide fast data consistency for the transmission of data elements with a size up to 8 Byte plus additional administration data.*
- *[R2.D20] (to WP 5): The communication infrastructure should provide data consistency for the transmission of data elements with any limited fixed size.*

To fulfil both requirements, the hardware will provide native support for messages up to a fixed size (of at least 8 bytes) and a mechanism to allow the system software to split messages of unlimited length into smaller ones in a consistent way. This leads to a consolidated requirement that extends CR5.3:

- **CR5.7 The size of messages between partitions must be at least 8 bytes and there must be a mechanism to guarantee the consistency of longer messages.**

In addition to that, there are no restrictions on the NoC (besides timing requirements for WCET analysis, see section 2.6). In particular, it is not specified, how the inter- and intra-cluster communication should be implemented in hardware. Section 3.2 discusses about the different designs alternatives.

Additionally, there is no restriction on the type and topology of the NoC. The choice of the interconnection depends only on the timing guarantees that can be provided.

2.4 Interrupts and Global Synchronized Time Basis

The applications positively demand a synchronized time basis over all cores:

- *[R2.D35] (to WP 5): All cores shall be synchronized by clock cycle.*

Because this requirement has to be satisfied, another requirement from the system level software is satisfied automatically:

- *[R4.6] The hardware should provide support for barriers, either hardware support by adding specialised hardware components or software support by adding a global synchronised clock to each core.*

Hence there is no need for *a specialized hardware for barriers*. The system software will implement barriers by using the global synchronized time basis that is provided by the following consolidated requirement:

- **CR5.8: All cores are synchronized by a global synchronized time basis.**

Given this global time basis also a requirement for the communication can be satisfied:

- *[R2.D36] (to WP 5): It shall be possible to trigger communication at predefined points in time (real-time or crank angle time).*

By tracking the global clock, a processor core can wait for the predefined clock value and start the communication at exactly the predefined point of time.

But there is also a demand for an external clock/time source:

- *[R2.H13] (to WP5): A time-triggered interrupt should be available, triggered from an external clock source (eventually from an internal timer if the external source is not applicable)*
- *[R2.D21] (to WP 5): The parMERASA processor architecture shall have at least two synchronized global clocks on all cores.*
- *[R2.D22] (to WP 5): One global clock shall be synchronous to the crank-angle and the other synchronous to real-time.*
- *[R2.D23] (to WP 5): The crank-angle synchronous time shall be identical on all cores with a precision of one processor clock tick and with a resolution of at least 1ms.*

In the context of [R2.D21], “clock” means an external signalling or interrupt source. Consequently a mechanism for broadcasting an external signal to all cores at the same time must be provided for at least two signals.

- **CR5.9: Globally synchronized interrupts can be triggered by at least two independent external signal sources.**

The behaviour of the interrupt controller is defined by two more requirements from the system level software:

- *[R4.5] The hardware shall provide an external interrupt controller, which has the similar behaviour like the PPC timer interrupt. Specifically that means that each interrupt source can be individually enabled/disabled.*
- *[R4.8] Any interrupt required by the hardware should be serviceable by the O/S.*

This is reflected by the following consolidated requirement:

- **CR5.10: All interrupt sources can be masked individually.**

2.5 Peripheral I/O

There should be an extensive flexibility in mapping I/O peripherals to cores:

- *[R2.D17] (to WP 5): Each I/O-interface shall be assignable to one core. Hardware interrupts shall be directed to this core and I/O registers shall be accessible from this core only.*

Since there are also other target applications, the assignment of I/O to the cores should be flexible. The restricted accessibility can be guaranteed by the memory management units. This flexibility must also be mentioned in a consolidated requirement:

- **CR5.11: Each I/O-interface is flexibly assignable to cores.**

The I/O devices will be simulated in the parMERASA many-core simulator. The way the peripherals are simulated is defined by the following requirements:

- *[R2.D25] (to WP 5): The system software shall allow connections to external environment simulation.*
- *[R2.D33] (to WP 4 and 5): It shall be possible to import trace files into the simulator to fill virtual hardware registers.*
- *[R2.D34] (to WP 4 and 5): It shall be possible to export trace files of virtual hardware register from the simulator.*
- *[R2.B01] (to WP 4 and 5): It shall be possible to import trace files into the simulator to simulate the inputs*
- *[R2.B02] (to WP 4 and 5): It shall be possible to import trace files into the simulator to simulate the CAN-Bus (4 times)*
- *[R2.B03] (to WP 4 and 5): It shall be possible to export trace files of the Outputs*
- *[R2.B04] (to WP 4 and 5): It shall be possible to export trace files of the CAN-Bus (4 times)*

Hence, the input will be simulated by trace files, i.e. files that contain a list of timestamps and values that appear at the specified timestamp. To have uniform trace files, we map all peripherals (respectively their virtual registers) to memory locations and use trace files that contain triples of timestamp, memory address and value.

The output is handled in a similar way: output registers are mapped to memory locations and whenever a value is written to an address within the specified range, the address, the value and a timestamp is written to a trace file of the affected device. This leads to the following consolidated requirements for input and output:

- **CR5.12: Input and output is memory mapped.**
- **CR5.13: The input from peripheral I/O devices is simulated by input trace files.**
- **CR5.14: The output to peripheral I/O devices is stored in output trace files.**

Furthermore, the requirements [R2.B02] and [R2.B04] demand four CAN bus interfaces and in deliverable D4.1 timers are mentioned.

- **CR5.15: There are at least 4 external CAN bus interfaces and one timer per core.**

There are no requirements from the avionics domain on dedicated peripherals, only requirements to the bandwidth of external connections are stated:

- *[R2.H11] (to WP5): Data needs to be supplied to the application at the rate of 1280 kB/s. The hardware architecture needs to provide means for this data transfer; neither a particular interface nor communication protocol is required.*
- *[R2.H12] (to WP5): Bi-directional 32-bit wide external communication analogical to a local bus is required which should operate at the frequency of a core which is in charge of handling external communication; still, only a limited bandwidth is used for actual communication of roughly about 1MHz.*

Both requirements are directly translated to consolidated requirements:

- **CR5.16 The processor is able to receive 1280 kB/s of input data.**
- **CR5.17 The processor has a bidirectional external connection of at least 32 bit at 1 MHz.**

The actual interface that provides the data is not specified and of no importance, only the bandwidth requirement has to be fulfilled.

In addition, the avionic applications need programmable timers, (i) to control the time slicing of resources that are shared between partitions, (ii) to process scheduling deadlines and for (iii) inter- and (iv) intra-partition communication timeouts, as mentioned in section 3.3.4 of deliverable 4.1. All these cases can be handled by individual timers per core, as stated in a requirement of the system level software:

- *[R4.10] (to WP5): Each core shall comprise a programmable timer with support for arbitrary comparator values.*

The according consolidated requirement is

- **CR5.18 Each core has its own programmable timer.**

2.6 WCET Analysis

The requirements concerning the WCET of the hardware components are:

- *[R3.1] (to WP5): The parMERASA architecture must enforce the timing predictability of any elementary operation (instruction, access to a communication network, access to a memory).*
- *[R2.D12] (to WP 4 and 5): The partitions shall have no direct write access to each other's memory.*
- *[R2.D18] (to WP 4 and 5): The communication worst case timing overhead between partitions shall scale less than exponential with the number of partitions.*

This leaves a lot of freedom for designing the hardware and results in the following consolidated requirements:

- **CR5.19: The latency of memory and I/O accesses depends on the location of the requester and on the location of the desired address, and must be predictable.**
- **CR5.20 The timing of processor instructions must be predictable.**

3 PROPOSED PARMERASA ARCHITECTURE

The parMERASA target applications impose on the processor architecture to guarantee *time and space isolation* among *applications*, meaning that the functional and timing behaviour of an application must not be affected by others (CR5.19). When moving towards parallel execution in which applications are spawned in multiple parallel threads, the time and space isolation property must remain the same, i.e. parallel threads belonging to one application cannot affect the functional and timing behaviour of parallel threads belonging to other applications running simultaneously within the parMERASA system.

Therefore, the envisioned parMERASA architecture will guarantee time and space isolation for the set of cores that forms the cluster in which a parallel application runs as imposed by the consolidated requirements CR5.1 and CR5.2.

Figure 4 shows the envisioned parMERASA architecture block diagram composed of four clusters and four cores per cluster. Each cluster has access to a shared memory region protected from other clusters (CR5.2) in which I/O devices will be mapped (CR5.11, CR5.12). In case applications require to communicate among them, inter-cluster communication mechanisms among applications will be provided (CR5.3) without affecting the time and space isolation property.

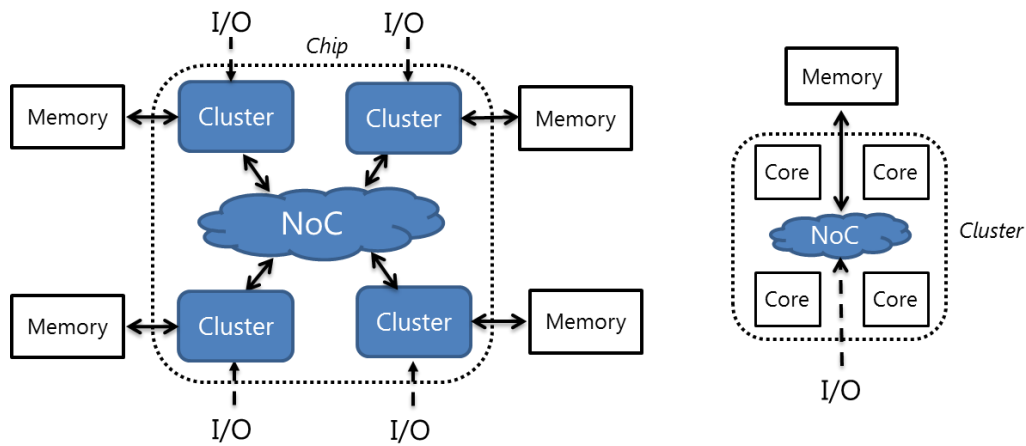


Figure 4. Envisioned parMERASA architecture block diagram

The following sections cover the envisioned design of the memory hierarchy, network on chip, and I/O system.

3.1 Memory Hierarchy

The sample target applications will perform pipelined execution, data parallel calculations, and functional parallelism. In all cases, each core needs access to program code, private data, and shared data (CR5.2). Additionally, it must be possible to logically cluster cores (CR5.1). Hereby, it must be guaranteed that one core is not able to disturb the activities of cores in other clusters (CR5.19). Disturbing means that a core in one cluster cannot harm

1. the timing behaviour of the cores in other clusters as well as the timing behaviour of the interconnect inside other clusters,
2. the data integrity of other clusters, i. e. the memory areas used by each cluster need to be protected from accesses of other clusters.

Moreover, all cores should have efficient access to all kinds of data and code.

To allow efficient access to code and data, all cores of the parMERASA many-core will be equipped with local memories for code and data, implemented partly as local scratchpad and partly as local cache memory.

3.1.1 Local Code Memory

Regarding the code memory, further evaluations based on the parallelized applications are required. This is because the code size is an important issue with respect to the type of code memory. The following options are present:

- If only a small amount of code is required to be held locally, a simple instruction scratchpad memory will be sufficient. It can be loaded with the required program code before system start by a boot loader. No further accesses to external memory are required during execution time, which keeps network load small (performance increase) and eases the analysis of the network traffic (less overestimation of WCET).
- In case of a code size which is not suitable to be held in a small local scratchpad memory, a dynamic instructions scratchpad (D-ISP) will be implemented. The D-ISP [5] does not need a boot loader: It is equipped with a special control logic that fetches instruction blocks on the basis of complete functions automatically from the main memory. Hence, network traffic is restricted to function boundaries (calls and returns) and the overall code size can be larger than the local memory itself.
- If the network traffic generated by the D-ISP turns out to be too intensive at the function boundaries, a standard instruction cache will be integrated. The reason is that memory accesses from a cache are more evenly spread over time compared to a D-ISP but they lead to more frequent cache misses.

The kind of instruction memory, which will be finally implemented in the parMERASA system, depends on the parallelized applications, the type and performance of the interconnect, and the preferences of the timing analysis tools (WP3).

A hierarchical system of multiple (shared) instruction memories (caches) will not be considered. This is because

1. the instruction fetch accesses from the local instruction memories (I-scratchpad, D-ISP, I-cache) should be restricted to a minimum and hence, a hierarchical system is not necessary,
2. the cores could execute different programs (applications, threads, or functions) and hence, because of interferences a shared cache is hard to analyse with high accuracy.

The actual sizes of the instruction memories/caches will be defined during the project. Reasonable memory sizes will be derived from several state-of-the-art commercial many-core systems like the ones mentioned in section 1.

3.1.2 Local Data Memory

With respect to local data memory, it is more clear which kinds of memory need to be implemented: local scratchpad memories as well as a local caches are required. Similarly to the code memories, a cache hierarchy will not be integrated in the first step. This is because the focus is on hard real-time applications and analysing the timing behaviour of a cache hierarchy with accesses from multiple cores is not reasonable. Interferences from different cores to the same cache line would increase the overestimation disproportionally, leading to an inefficiency of the cache hierarchy with respect to the WCET.

In a second step, a hierarchical cache system could be implemented in order to speed up non hard real-time applications. For these types of applications, a strict WCET analysis is not required. To allow parallel hard and non-hard real-time applications to be executed on the parMERASA many-core system, the use of the hierarchical cache system is optional.

The following sections describe the two kinds of data memories.

3.1.2.1 Local Scratchpad Data Memory

A local memory is required to hold the often accessed private data, especially the runtime stack of an executed thread. The special characteristics of the runtime stack are described below:

- The stack is accessed only by the local core. No other core will ever access the data on the stack of a particular core.
- The runtime stack offers a high locality since accesses concern only the frame of the currently executed function. The stack can grow and shrink but the currently accessed data is always near the top of stack.
- The maximum size of the runtime stack can be exactly predicted in case of a hard real-time application. This is because the maximum stack size directly depends on the call tree which is generated by a WCET analysis tool.

As a result, the runtime stack is predestined to be located in a fast local memory. The minimum size of that memory must be at least the maximum stack size of the executed application. Hence, the parallelized sample applications will be analysed and the size of the local data memory will be defined according to that analysis.

In addition to the stack, also other data can be stored in the local data memory. Hereby, smaller private data structures required for program execution can be stored inside the local memory. If the local memory is accessible by other cores (it is not private), it is also possible to store shared data in the local memories. As an example, assume a pipelined execution at which each pipeline stage is implemented on its own core. The data that needs to be passed from one stage to the next one should be hold in the local memory of one of the neighbouring stages. Hence, only one core (stage) has to cross the interconnect in order to access the common data. It is interesting to notice that explicit inter-cluster communication could be implemented by using that technique (CR5.3), in which the size of explicit messages is restricted by the size of the local memory. Since the local memory will be greater than 8 bytes, CR5.7 can be fulfilled.

3.1.2.2 Local Data Cache

Data shared by more than one core or larger data structures need to be stored in a memory that is not local to a core, i.e. a global memory. Accesses to that memory unavoidably need to pass the interconnect since the global memory is not connected to a core directly. Hence, these accesses introduce long latencies resulting in inefficiency.

In order to reduce the number of these inefficient long latency accesses to the global memory, local data caches need to be implemented. Moreover, since the impact of memory accesses to other hard real-time applications (executed in other logical clusters) must be reduced to a minimum, multiple of these global memories will be implemented. In this way, each global memory can be assigned to a logical cluster. A global memory can be directly accessed only from the assigned cluster which is ensured by a memory management unit (CR5.2). In case of accessing global memories associated to other clusters, an explicitly inter-cluster communication is required (CR5.3).

An essential problem of local caches in multi-core/many-core environments is the coherency of accesses to shared data. Many approaches exist that can guarantee coherent memory accesses but to the best of our knowledge, none of these approaches is suitable for hard real-time systems. The problems of the well-known coherency techniques with respect to their real-time capabilities are:

1. If there are copies of the same data in multiple local caches it is required that there is a logical link between these caches i.e., each cache is aware of the fact that there are other copies of the same data. As a result, if one cache modifies its copy, the modification (or at least the information about it) needs to be shared with the other caches holding copies of the same data. This means additional, unpredictable traffic on the interconnect.
2. Most well-known approaches just invalidate the cache line that is modified by another core. As a result, the content of a cache does not depend on the actions carried out only by the local core but also on the actions performed by other cores. This makes a tight static WCET estimate impossible. In the case local copies are updated after modifications from other cores, the content of a cache depends only on the local core but the interconnect has to transmit even more information.

A common solution to deal with coherent accesses in real-time multi-core systems is to abdicate from cache coherency completely and to use non-cached memory areas for shared data. Hence, accesses to private data benefit from fast local caches and there is no need for coherence during accesses to shared data. Unfortunately, accesses to shared data cannot profit from the principles of temporal and spatial localism because there is no cache that supports these principles. This means that multiple accesses to neighbouring data require the corresponding number of interconnect messages since each access is performed individually.

In order to allow an application to benefit from the locality principles also during accesses to shared data and to get rid of or at least reduce the unpredictable interconnect traffic and the external modification of the cache content, the parMERASA many-core system will propose a new caching strategy. Therefore, coherent data accesses are guaranteed only if shared data is accessed. Hence, accesses to shared data need to be marked (resulting in R5.T2, see section 5.2) e.g., by accessing a special memory region or by enclosing the accesses by some special environment. A plausible way to mark accesses to shared data is to enclose the accesses by a critical region, for example. As a result, there is no effort spent on coherency of accesses to private data. CR5.7 can be fulfilled by the

proposed caching technique if explicit communication is performed by accesses to shared memory. The actual architecture and implementation of such a selective data cache will be defined in the next phase of the project.

3.1.3 Global Memory

The global memory needs to be accessed by all processor cores which require larger amounts of data and/or program code. As a result, all these cores influence the timing behaviour of each other, if they access the same physical memory. Since the parMERASA system should be able to execute multiple applications (partly with different levels of criticality) the impact on any hard real-time application needs to be eliminated or at least restricted to a tightly predictable minimum.

In order to reach that goal, the parMERASA many-core will provide multiple global memory banks which can be assigned to different clusters. In the case the target application needs external memory, multiple real-time capable memory controllers will be implemented. Each controller will have access to its own external memory bank. Inside the parMERASA chip, each memory controller can be assigned to a logical cluster of cores executing the same application (CR5.2). Hence, accesses to a cluster's memory cannot harm the timing of any other cluster since it accesses its own external memory. If there are more independent applications than external memories available, multiple selected applications have to share the same physical memory. The selection process of these applications is based on memory access behaviour of the applications and their criticality level.

3.1.4 Arrangement of Memories

The proposed memory hierarchy leads to the following arrangement of local and global memories as well as local caches and global memory controllers in the parMERASA system as shown in figure 5. In case of the automotive target application, all caches can be disabled globally (R2.D39).

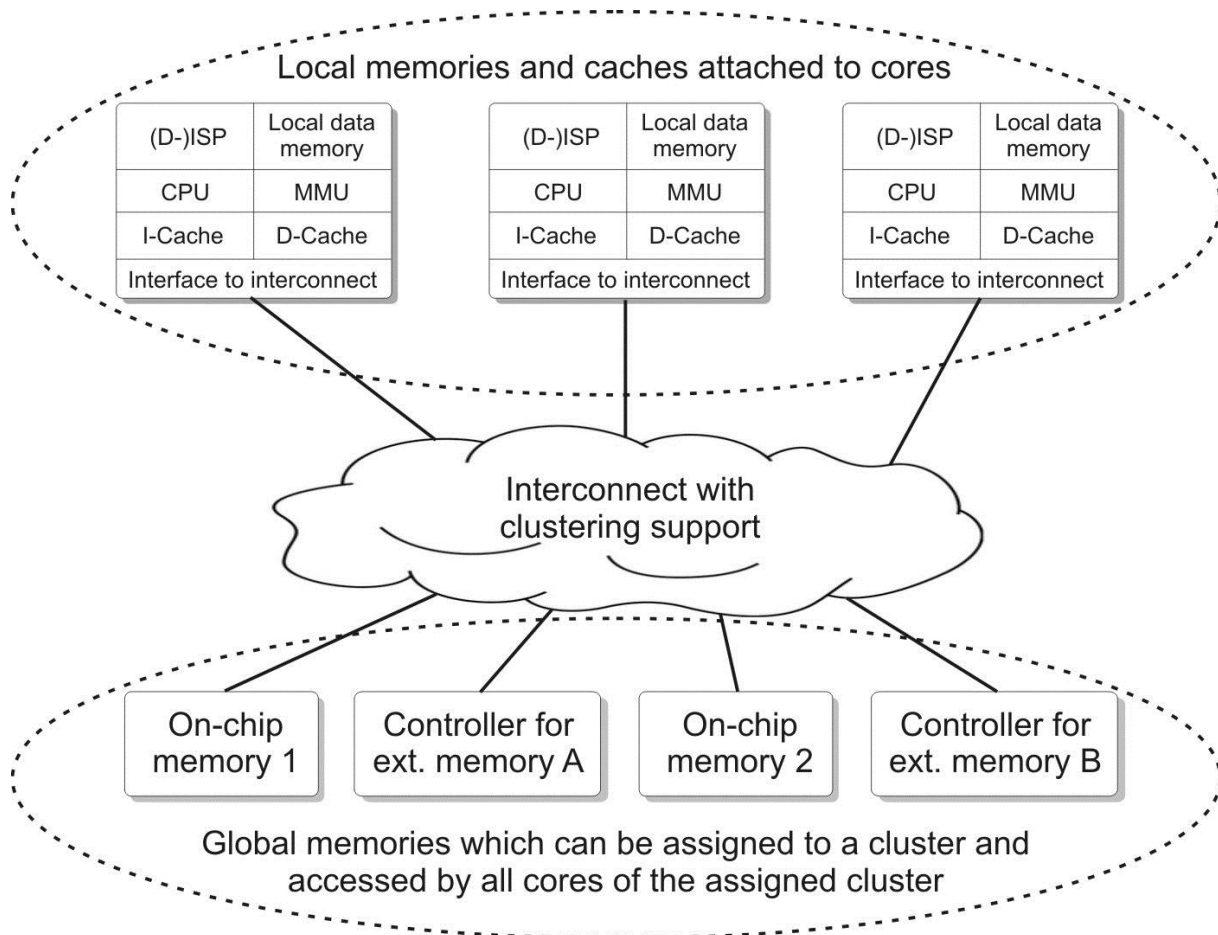


Figure 5. Arrangement of Memories

3.1.5 Support for Synchronization

All sample target applications will provide parallel program execution. In order to coordinate parallel work, it is required to synchronize the parallel tasks. This synchronization requires hardware support, at least some basic support by providing the possibility of atomic read-write accesses to memory.

A more powerful solution is to provide support for atomic fetch-and-add (CR5.5) operations or dedicated synchronization components. The parMERASA many-core system will provide one of these powerful techniques, which one will be discussed in the next months. Several advantages and disadvantages of both techniques are listed below.

3.1.5.1 Fetch-and-add

Atomic fetch-and-add allows to read a memory cell (byte, word), increment the read value immediately by a given value (most often by '1'), and write the new value back to the same memory cell. Since this operation is atomic, it cannot be harmed by any other memory operation; especially it is not possible for any other thread to read/write the cell between the read and the write operation. But, compared to a simple memory access, the fetch-and-add operation takes at least twice the time since it requires a read and a write. This circumstance needs to be taken into account during the WCET analysis, or a special technique of memory accesses is required like the one proposed by the MERASA project [6]. Moreover, the fetch-and-add instruction needs to be implemented into the used PowerPC core (CR5.4).

An advantage of the fetch-and-add operation is that any arbitrary memory cell can be used for synchronization purposes. This means, that the number of used locks is restricted only by the available memory. It is important to remark that the used memory area must be defined as non-cachable in order to guarantee consistent lock variables.

3.1.5.2 Synchronization hardware unit

In the case of specialized hardware unit used for synchronization, the number of possible locks depends on the implemented hardware i.e., less locks are supported compared to the fetch-and-add solution. But, since access to the synchronization hardware is separated from the memory, there are no interferences between the normal memory accesses and the locking operations resulting in a more tight WCET analysis.

3.2 Demands on the Network on Chip (NoC)

The NoC plays a fundamental role in the many-core processor architecture design. That is, the NoC is the resource responsible of distributing messages to the different components, i.e. to cores, memory hierarchy, I/O devices and memory controllers.

The parMERASA NoC design must guarantee that the time isolation property is accomplished among clusters as well as providing a guaranteed maximum latency and minimum bandwidth (CR5.19, CR5.20). It is important to remark that the design of components attached to the NoC strongly influences the timing behaviour of the NoC, as well. This is the case for instance of local memories, i.e. scratchpads and cache memories, which can significantly reduce the memory traffic through the NoC. However, as already stated in previous sections, the implementation of not suitable cache coherence protocols can significantly increase the traffic through NoC.

This section focuses only on the NoC mechanisms required to provide a time predictable behaviour in which the timing isolation among clusters (CR5.2) and the maximum latency and minimum bandwidth are guaranteed (CR5.19, CR5.20).

3.2.1.1 Basis of NoC Design

The NoC (also known as On-Chip Network of OCN) is a type of interconnection network used to connect micro-architectural devices (e.g. cores, caches, memory controllers, I/O devices) within a chip. The NoC is composed of a set of channels and routers that distribute the data among connected devices. The network *topology* determines the physical layout and connections between routers and channels.

NoCs implements four main mechanisms to deliver messages within the chip, including *routing*, *arbitration*, *switching* and *flow control*:

- The routing is in charge of selecting the path to connect two nodes. The topology defines all the possible paths to communicate the different nodes connected to the network.
- The arbitration is in charge of resolving the conflict when the same path is asked by two or more communication requestors. As a result, only one request is allowed to use the channels and routers of the corresponding path, stalling the other requests, which typically are buffered until the resources associated to this path are released.
- The switching is in charge of establishing how resource in routers (e.g. buffers, ports) are allocated as the message travels through the NoC.

- The flow control is in charge of guaranteeing that the message is correctly hold in the buffers of the routers and no part of the message is lost.

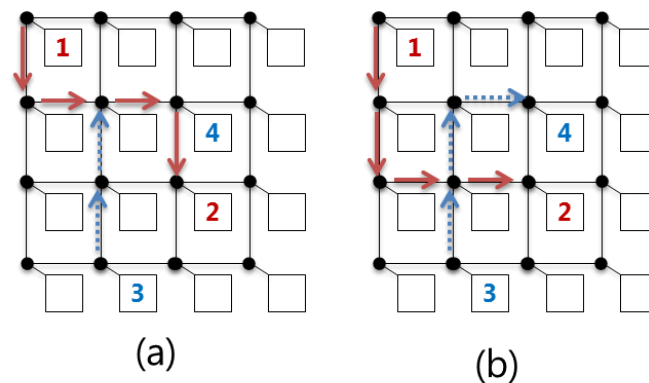


Figure 6. Two different routing mechanisms.

Figure 6 shows two different outcomes of the routing algorithm to communicate two pair of resources: 1 – 2 and 3 – 4 in a 2D mesh topology. In (a), the path selected in to communicate 1 and 2 stalls the messages going from 3 to 4, since the arbitration policy has granted 1 – 2 communication to use the channel. Instead, in (b) the path selected to communicate 1 and 2 does not stall messages from 3 to 4. In this case, the switching allows to alternate communications from different ports.

Commonly, the term “NoC” is used only to refer complex interconnections networks such as meshes, or torus. However, simple interconnection networks such as buses are also a type of NoC, in which all mechanisms existing in complex NoCs are implemented as well. However, the implementation of most of them, such as the switching and routing, are straightforward. Hence, once the request is arbitrated to accesses the bus, the granted device simply needs to connect itself to the bus, thus establishing a path to every possible destination. One well known NoC based on a bus is the AMBA from ARM.

Moreover, NoCs can be classified as well in two main groups: *shared-media* and *switched-media* networks. In shared-media networks (e.g. a bus), the multiple nodes are connected to a unique channel (media) shared among them. Instead, in switched-media networks nodes are connected to a network fabric composed of channels and routers in charge of establishing point to point communications between nodes. The aggregate bandwidth in shared-media NoCs does not scale at all when increasing the connected devices. The reason is that the global arbitration scheme required to resolve the conflicts when accessing the shared link represents a bottleneck that limits the scalability. Moreover, every device attached to the shared media increases the parasitic capacitance, thus increasing the time of flight propagation delay accordingly and, possibly, clock cycle time. The main advantage of switched-media networks is its scalability as it allows multiple pairs of nodes to communicate simultaneously, increasing the aggregate network bandwidth with respect to shared-media networks. The parMERASA project will focus on switched-media networks. It is important to remark that a switched-media NoC can be composed of multiple buses as well, e.g. to connect cores within clusters.

3.2.1.2 Timing Analysis of a NoC

The performance of a network is characterized by its *latency* and *bandwidth*:

- The latency determines the amount of time a request takes to traverse the network from one node (sender) to another (receiver).
- The bandwidth is the maximum rate at which information can be transferred through the network from one node (sender) to another (receiver). The *aggregate bandwidth* refers to the total data bandwidth supplied by the network.

Therefore, in order to offer real-time performance, the NoC design must *guarantee a maximum latency* and/or a *minimum bandwidth*. The former is usually required in control flow applications in which fast access to processor resources (e.g. the main memory in case of a cache miss) is desirable. Instead, the latter is required in data flow applications in which the latency of one request is not important but to guarantee a continuous flow of data (e.g. streaming applications). In any case, providing guarantees of maximum latency and/or minimum bandwidth are fundamental to provide WCET analysis.

The latency and bandwidth of a NoC is influenced by multiple factors including *routing*, *arbitration*, *switching* and *flow control* mechanisms. Thus, the latency of a request can be computed by considering the impact of these mechanisms on the timing behaviour of a request:

$$\text{Latency} = (t_r + t_a + t_s + t_f) * d + t_t * (d + 1)$$

Being the t_r routing time, t_a the arbitration time, t_s the switching time, t_f the flow control time, t_t the link traversal time and d the number of links traversed the path selected by the routing. It is important to note that the topology influences the performance of the NoC as well, as it defines all the possible paths.

Therefore, each of these mechanisms that contribute to the timing behaviour of the NoC will be analysed during the second phase of the parMERASA project.

3.2.1.3 Message Classification

As stated in R4.2 requirement, the intra-cluster communication is performed through shared memory (e.g. through global variables). As a result, every time a data producer, i.e. a core or an I/O device, wants to communicate with a consumer, i.e. another core or I/O device, it will update the corresponding memory location making the data visible to the consumer. Instead, as stated in R4.4 requirement (WP4), the inter-cluster communication, i.e. when the memory in which the data reside belongs to other cluster, is performed through explicit messages. By doing this, we distinguish three different classes of messages (Figure 7Figure 7 shows the three classes):

- *Intra-cluster messages*. The communication occurs among cores belonging to the same cluster through the memory associated to it. In case the shared data resides in a local memory, e.g. a local cache, the coherence mechanism must guarantee that the communication considers the correct updated values.

- *Inter-cluster message.* The communication occurs among cores from different clusters. In this case a message is sent by the system software to the core responsible of performing the cluster-to-cluster communication (see deliverable D4.1 to further details).
- *I/O communication.* The communication occurs among I/O devices and clusters through a DMA device or similar.

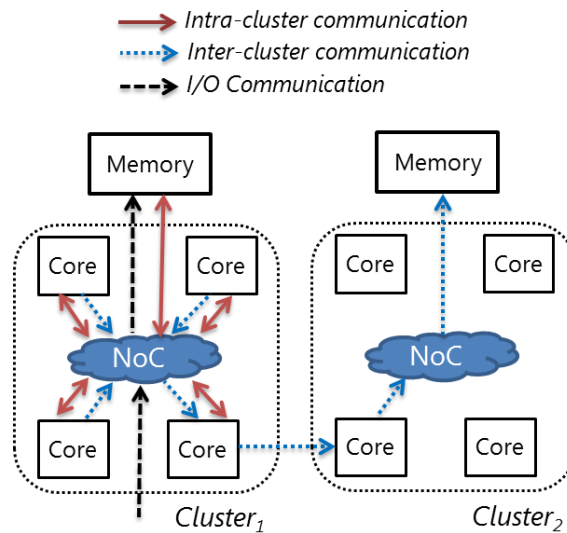


Figure 7. The NoC Messages are classified in three types: intra-cluster, inter-cluster and I/O

Thus, intra-cluster messages will communicate parallel tasks belonging to the same application running within a cluster, at which timing isolation is guaranteed. It is important to remark that this type of communication occurs when accessing to both shared and private memory regions of tasks (see section 3.1 for further details).

Instead, inter-cluster and I/O messages occur among applications being run in different clusters or among applications and I/O devices respectively. This type of communication is statically predefined at system initialization, as it involves not a single application but the system. This is the case for instance of IMA systems that defines communication methods across partitions (see deliverable D4.1 to further details), or I/O devices such as a camera in case of the stereo navigation which captures an image every 0.1 seconds (see deliverable D2.1 for further details).

Therefore, we further classify the messages in two main groups:

- *Dynamic communication.* This group is composed of intra-cluster messages, automatically generated by the hardware, every time the global memory associated to the cluster is accessed. Due to the timing isolation property at the cluster level (CR5.2), the timing analysis can consider only the maximum latency/minimum bandwidth of intra-cluster messages. The NoC design must guarantee that messages will never traverse cluster boundaries and so NoC interferences will not affect the timing behaviour of other clusters.
- *Static communication.* This group is composed of inter-cluster and I/O message. These messages travel across multiple clusters, potentially affecting the timing behaviour of applications. However, since messages are statically generated in predefined order, its effect on dynamic communication can be taken into account when estimating the WCET.

In order to guarantee dynamic communication not to exceed cluster boundaries, the network topology and the routing policy plays a fundamental role. parMERASA aims to research on two types of network topologies: clustered and flat topologies (see figure 9).

In *clustered topologies*, there exists one single path to communicate nodes within a cluster, guaranteeing by network construction that the dynamic communication will never exceed cluster boundaries. Figure 8(b) shows an example of a clustered NoC composed of hierarchical buses in which each bus concentrates the cluster communication. The intra-cluster communication of cluster 1 cannot exceed it as there is only one possible path to access to the global memory. Another example of a clustered topology is a combination of crossbars and buses to connect cores within clusters and to connect clusters among them respectively. Note that in this case, the cluster is physically defined by the NoC.

In *flat topologies* instead, there exist multiple paths in which intra-cluster messages can be routed, making the routing in charge of guaranteeing that the communication does not to exceed cluster boundaries. Figure 8(a) shows an example of a flat topology, i.e. a mesh, in which routers and links are uniformly distributed in the network fabric. In this example, the X-Y routing policy guarantees that messages among nodes belonging to cluster 1 do not exceed its limits, while Valiant's routing algorithm [7] cannot guarantee it.

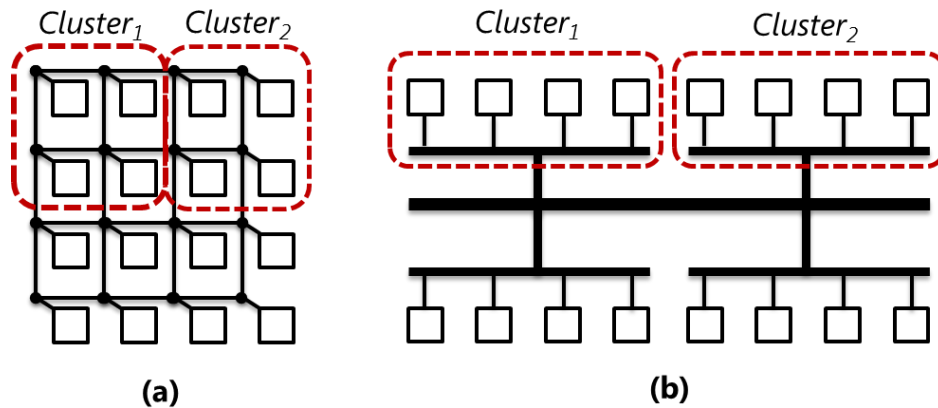


Figure 8. Cluster formation in two NoC designs: Flat design (a) and clustered design (b)

However, dynamic communication can still suffer interferences with requests coming from static communication. Therefore, in order to reduce or even eliminate the potential conflicts occurring between static and dynamic communication, parMERASA aims to research on techniques to separate the three different message types by mainly considering two techniques:

- To provide a physical separation of the different messages by implementing a separated NoC per each message type (e.g. Tilera processor, see section 1). By doing so, conflicts will be completely removed.
- To provide a virtual separation of the different messages by implement multiple virtual channels per each message type (e.g. Intel SCC, see section 1). In this case, conflicts will be reduced since messages will still share channels and router resources such as output ports.

3.3 IO/IRQ Mechanism

Connecting I/O devices to the parMERASA many-core covers two main topics: transferring data to and from the I/O devices and sending interrupt requests to the cores. Data transfer will be covered directly by the NoC as described in section 3.2. The performance of the NoC is suitable to satisfy CR5.16. Regarding the interrupt mechanism, several possibilities exist that are discussed in the following section.

3.3.1 Interrupt Request Mechanism

Two issues arise when designing an interrupt system for a many-core system: flexibility and latency. In single core systems, flexibility means to assign different priorities to the interrupting devices. Regarding a many-core system, flexibility means to be able to assign an interrupt source to individual cores. The latency is the period a request takes to reach the interrupt destination, i.e. the assigned core. Possible solutions for connecting interrupt sources to the cores in this environment could be:

1. ***n* to *n* connection** - This means each I/O device will be linked to every core by a dedicated wire. The advantage of this method is that requests reach their destination core immediately because there is always a direct link between an I/O device and a core. Unfortunately this wiring does not only result in high hardware requirements and power consumption, but also in long wire delays. In addition, connections between I/O and cores on the opposite of the mesh will be longer than to the cores of the same side.
2. **1 to 1 connection** - Every I/O device is connected only to the core it is near to. Hence, there is no flexibility but the latency is reduced to a minimum.
3. **NoC capitalizing** – The I/O devices are connected to the NoC. Interrupt requests are sent through the NoC to the destination core. No special wiring is required, full flexibility is available, but the latency is higher than in the case of dedicated wires. If the interrupt signalling uses its own virtual channel of the NoC, a worst case latency can be calculated and the higher latency can be tolerated since it is analysable.

For the parMERASA many-core the third approach seems to be the most promising because it is the most flexible one (CR5.11) and benefits most from the existing NoC. Hereby, a special hardware unit will connect the interrupt sources to the NoC. It will be in charge of generating a message sent through the NoC to the destination core. Both, the destination core as well as the message content will be configurable. Requirement CR5.10 can be fulfilled by disabling this hardware unit.

Figure 9 shows an example composition of the simulator with a CAN device connected to core 00 and a digital I/O device connected to core 30. The receiver for the interrupt signals in this example is core 22, respectively core 11. If one of the I/O devices requests an interrupt, it could be forwarded on the shown path (assuming X-Y routing).

Moreover, two global interrupts will be implemented, which are able to send interrupts simultaneously to all cores. As sources, external signals and/or two dedicated global timers can be used. These two global interrupt lines are required to fulfil CR5.8 and CR5.9.

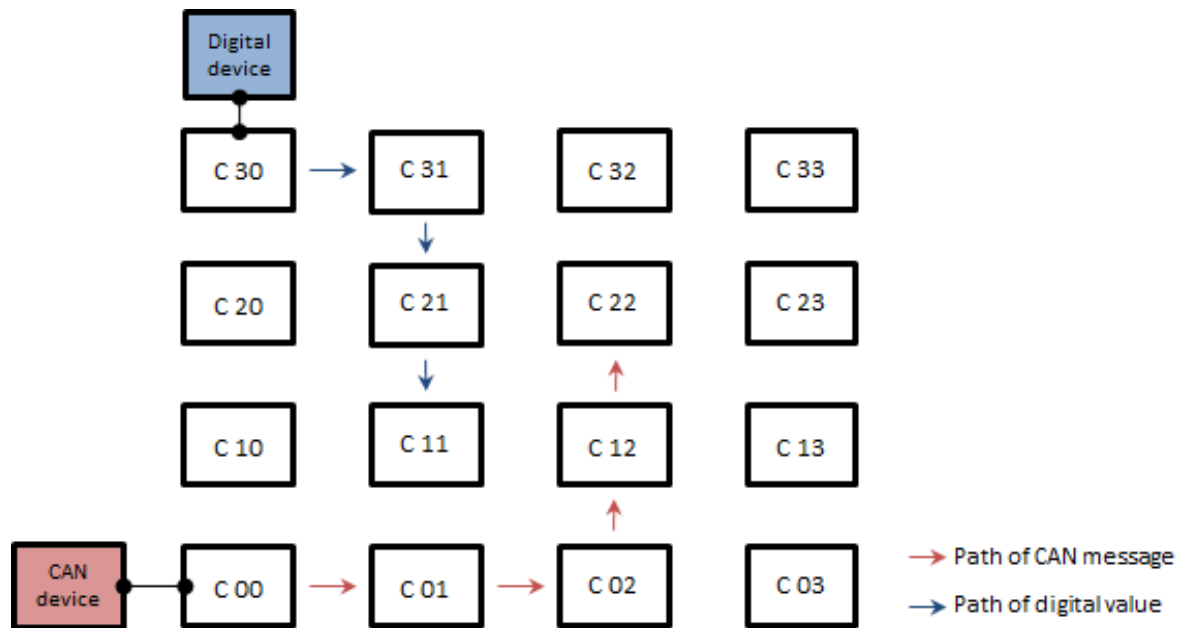


Figure 9: Routing interrupt requests through the NoC

The parMERASA simulator will consider the impact of I/O messages on the timing behaviour of the NoC as shown in section 3.2.

3.3.2 Required IO Devices

The three application domains have partly different as well as common demands on the implemented I/O devices. The following list shows the devices that will be integrated into the simulator, i.e. their behaviour will be modelled:

- Multiple CAN bus devices (send and receive)
- Parallel inputs and outputs
- PWM outputs
- Analogue inputs
- External interrupt sources
- Timers (timers are local to each core, CR5.15)
- DMA (Direct Memory Access) devices to transfer data from the CAN, the analogue inputs, and/or the parallel input into the memory

The behaviour of all I/O devices (except for the DMA) will be modelled by external trace files. These files contain triples of timestamp, memory address, and value. In case of input devices, the given value will be seen at the given address in the memory address space (CR5.12) starting from the specified time (CR5.13). All output devices will generate equivalent trace files (CR5.14).

In order to fulfil CR5.17, a DMA device can also be connected to the parallel IO. Thereby, parallel data of up to 32 bit can be transferred from and to the memory at a data rate of 4MB/s (32bit with 1MHz).

Originating from the IO input handling, a requirement from WP5 to WP2 is defined (R5.TB3): Individual trace files containing triples of timestamp, address, and value is necessary for every input device. These files will stimulate the inputs and hence, the simulated execution of the real-world target applications will be as realistic as possible.

4 REQUIREMENTS – SUMMARY

This section summarizes the Consolidated Requirements (CR5.x) which serves as basis for all design decisions for the parMERASA hardware architecture. Additionally, these CR5.x define the success criteria for the parMERASA many-core as they incorporate all requirements from other work packages to work package 5.

Section 4.2 defines requirements that need to be fulfilled by other work packages. This is because the parMERASA many-core architecture/simulator makes several demands on the programming style, the design of the WCET analysis tools, as well as the simulation environment.

4.1 Consolidated Requirements (Internal)

- **CR5.1: Cores can be grouped into several virtual clusters. The number of cores per cluster can be configured individually per cluster but it cannot be changed during program execution.**
- **CR5.2: Each cluster has access to a shared memory region that is protected from accesses of other clusters.**
- **CR5.3: There is an explicit communication between clusters.**
- **CR5.4: The hardware prototype supports the PowerPC instruction set.**
- **CR5.5: The PowerPC instruction set is extended by an atomic read-modify-write instruction.**
- **CR5.6: It is possible to change between several communication schedules.**
- **CR5.7: The size of messages between partitions must be at least 8 bytes and there must be a mechanism to guarantee the consistency of longer messages.**
- **CR5.8: All cores are synchronized by a global synchronized time basis.**
- **CR5.9: Globally synchronized interrupts can be triggered by at least two independent external signal sources.**
- **CR5.10: All interrupt sources can be masked individually.**
- **CR5.11: Each I/O-interface is flexibly assignable to cores.**
- **CR5.12: Input and output is memory mapped.**
- **CR5.13: The input from peripheral I/O devices is simulated by input trace files.**
- **CR5.14: The output to peripheral I/O devices is stored in output trace files.**
- **CR5.15: There are at least 4 external CAN bus interfaces and one timer per core.**
- **CR5.16: The processor is able to receive 1280 kB/s of input data.**
- **CR5.17 The processor has a bidirectional external connection of at least 32 bit at 1 MHz.**
- **CR5.18 Each core has its own programmable timer.**
- **CR5.19: The latency of memory and I/O accesses depends on the location of the requester and on the location of the desired address and must be predictable.**
- **CR5.20 The timing of processor instructions must be predictable.**

4.2 Requirements to Other Work Packages

[R5.T1] (to WP3): Using multiple physical memory locations for code as well as for data means that the static WCET analysis tool needs to deal with different memories. This is that different memory locations (addresses) provide different latencies, some are cached and some are un-cached.

[R5.T2] (to WP2): In order to distinguish between accesses to private data and shared data it is necessary that the type of data access is marked inside the application. One possibility could be to enclose accesses to shared data in a synchronization environment i.e., a critical section. Another way to indicate shared data is to use a special memory region that holds only shared data.

[R5.TB3] (to WP2): In order to stimulate input values, individual trace files containing triples of timestamp, address, and value are required for each input device.

LIST OF FIGURES

Figure 1. Freescale P4080 processor architecture block diagram.....	7
Figure 2. Tilera Pro64 processor architecture block diagram	9
Figure 3. Intel SCC processor architecture block diagram	11
Figure 4. Envisioned parMERASA architecture block diagram	21
Figure 5. Arrangement of Memories.....	26
Figure 6. Two different routing mechanisms.	28
Figure 7. The NoC Messages are classified in three types: intra-cluster, inter-cluster and I/O.....	30
Figure 8. Cluster formation in two NoC designs: Flat design (a) and clustered design (b)	31
Figure 9: Routing interrupt requests through the NoC	33

REFERENCES

1. Freescale Semiconductor. QorIQ™ P4080 Communications Processor Product Brief. Doc.No. P4080PB. Rev 1, Sep 2008.
2. Tilera. Tile Processor User Architecture Manual. Release 2.4. Doc. No. UG101. May 2011
3. Intel, SCC External Architecture Specification (EAS) – Revision 1.1
4. Bruno d'Ausbourg, Marc Boyer, Eric Noulard, Claire Pagetti, Deterministic Execution on Many-Core Platforms: application to the SCC. 4th MARC Symposium (Toulouse, France), 8 – 9 Dec 2011
5. A Dynamic Instruction Scratchpad Memory for Embedded Processors Managed by Hardware Stefan Metzlaß, Irakli Guliashvili, Sascha Uhrig, Theo Ungerer, Proceedings of the Architecture of Computing Systems Conference (ARCS 2011)
6. An Analyzable Memory Controller for Hard Real-Time CMPs Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Mateo Valero, In the IEEE Embedded System Letter (ESL), Vol. 1, No. 4. December 2009.
7. L.G. Valiant and G.J. Brebner. Universal Schemes for Parallel Communication. In Proceedings of the 13th Annual ACM Symposium on Theory of Computing, 1981.