

parMERASA

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

D5.3

Predictable parMERASA Multicore Processor

Nature:

Dissemination Level:

Due date of deliverable:

Actual submission:

Responsible beneficiary:

Responsible person:

Report

Public

September 30, 2013

September 30, 2013

BSC

Eduardo Quiñones

Grant Agreement number:

Project acronym:

Project title:

Project website address:

Funding Scheme:

Date of latest version of Annex I

against which the assessment will be made:

Project start:

Duration:

Period covered:

FP7-287519

parMERASA

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

<http://www.parmerasa.eu>

STREP

SEVENTH FRAMEWORK PROGRAMME

THEME ICT – 2011.3.4 Computing Systems

June 20, 2012

October 1, 2011

36 month

2012-07-01 to 2013-09-30

Project coordinator name, title and organisation:

Tel: + 49-821-598-2350 Fax: + 49-821-598-2359

Prof. Dr. Theo Ungerer, University of Augsburg

Email: ungerer@informatik.uni-augsburg.de

Release Approval

Name	Role	Date
Eduardo Quiñones	WP leader	30 – September – 2013
Theo Ungerer	Coordinator	30 – September – 2013

DELIVERABLE SUMMARY

Deliverable 5.3 “Predictable parMERASA Multicore Processor” covers the work done during the second phase of the project within WP5. The deliverable spans fifteen month of work and handles the work done in tasks 5.2, 5.3 and 5.4 to reach milestone MS13:

Task description of T5.2 (m10 :: 15m): Predictable interconnection network

The research in this task has focused on the following topics:

- Network topology, including centralised switched (indirect) and distributed switched (direct) networks.
- Predictable routing strategies, including time predictable, deterministic and adaptive approaches and deadlock avoidance mechanisms
- Predictable arbitration policies, including starvation-avoidance techniques.
- Switching mechanism, including store-and-forward, cut-through, virtual cut-through and wormhole.

Task description of T5.3 (m10 :: 15m): Predictable memory hierarchy structure

The research in this task has focused on the following topics:

- Distributed shared memory (DSM) approaches, combining core local memory modules with global external memory.
- Core local memory management units (MMU) that translate core local and core external addresses (responsible for memory thread isolation).
- Cache placement and cache organisation (hardware versus software cache coherence protocol).

Task description of T5.4 (m10 :: 15m): Predictable connection of I/O devices

Key questions addressed in this task are:

- Where to connect the I/O devices.
- How to deliver interrupt requests to the corresponding cores.
- How to perform direct memory accesses (DMA) requested from I/O devices, if necessary.

Conclusion of tasks 5.1, 5.2 and 5.3:

All intended subtasks have been carried out successfully and MS13 has been reached.

Milestone 13:

- Collection of time predictable interconnection networks suitable for hard real-time parallel applications defined.
- Time predictable memory organisation suitable for hard real-time parallel applications defined.
- Time predictable connections of I/O devices suitable for hard real-time parallel applications defined.

- parMERASA multi-core defined and implemented in the simulator.

All objectives of MS13 have been reached. The first three parts of MS13 are documented in this deliverable; the fourth part is included in D5.4.

TABLE OF CONTENTS

1	Overview of the parMERASA Processor Architecture	7
1.1	Motivation	7
1.2	Parallel Software Partitions (pSWP)	8
1.3	parMERASA Processor Design: Guaranteed Resource Partition (GRP)	9
2	Network on Chip	11
2.1	Definition of Physical GRPs	11
2.2	Definition of Virtual GRPs	13
3	Memory Hierarchy	15
3.1	Cache Hierarchy	15
3.1.1	Cache Coherence Protocols	16
3.1.2	On-demand Coherent Cache (ODC ²)	17
3.1.3	Technique of the ODC ²	17
3.1.4	Synchronisation techniques for ODC ²	18
3.1.5	Performance impact of ODC ²	21
3.1.6	Analysability of ODC ²	23
3.2	Memory Controller	23
3.3	Memory Map	24
4	I/O Subsystem	26
4.1	CAN Module	27
4.2	Generic I/O Module	27
4.3	EEPROM Module	27
4.4	Interrupt System	28
4.4.1	Smart Interrupt Controller (SIC)	29
4.4.2	Tiny Interrupt Controller (TIC)	29
4.5	Communication with the I/O Subsystem	29
5	Timing Analysis of the parMERASA Architecture	30
5.1	Computing the WCET estimation of applications in Isolation	30
5.2	Computing Δ_{inter} of the parMERASA Architecture	30
5.3	Evaluation of Δ_{inter} at System Integration	32
6	Consolidated Requirements	35
7	References	37

1 OVERVIEW OF THE PARMERASA PROCESSOR ARCHITECTURE

1.1 Motivation

parMERASA target applications rely on *incremental qualification*, that allows each system component to be subject to formal certification (including timing analysis) in isolation and independently of other components, with obvious benefits for cost, time and effort.

In current safety critical real-time systems, incremental qualification is enabled by using standardised system software architectures, such as the Integrated Modular Avionics (IMA) in the avionics domain or the AUTomotive Open System ARchitecture (AUTOSAR) in the automotive domain.

Both software frameworks guarantee incremental qualification by providing robust space and time partitioning to applications, i.e. the functional and timing behaviour of each application is not affected by other applications, so certification or validation processes can be carried out. To do so, applications are encapsulated in software partitions (SWPs) (as defined in avionics and automotive standards ARINC 653 and ISO 26262 respectively), which guarantees the robust partitioning properties required. In other words, SWPs are the software incarnation of the robust partitioning property.

Avionics and automotive applications are encapsulated into SWP as shown in Figure 1. Within a SWP, applications are composed of several tasks (named *processes* in the avionic domain and *runnables* in the automotive domain) that share the same memory address space).

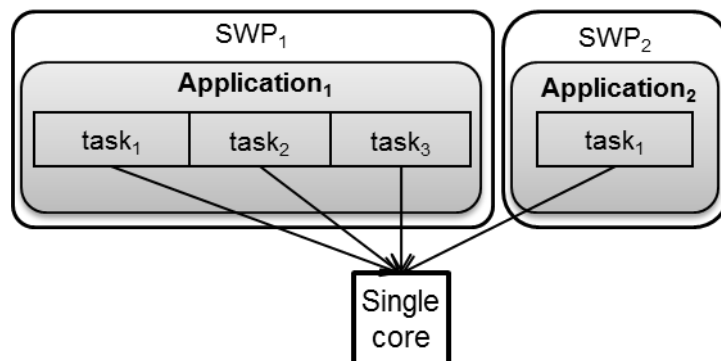


Figure 1. Time partitioning considered in safety real-time embedded systems

As stated in deliverable D5.1 from milestone MS12 and R4.2 requirement defined in WP4, SWPs define two communication methods that facilitate time isolation: *intra-partition communication* and *inter-partition communication*. The former uses global variables to exchange data among processes belonging to the same SWP; the latter uses messages to exchange data among applications.

Therefore, SWP impose the parMERASA 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. Such a requirement (captured in the consolidated requirement CR5.19 presented in deliverable D5.1 of MS12) has guided the development of the parMERASA architecture.

1.2 Parallel Software Partitions (pSWP)

In single-core execution, for each SWP, a time capacity is assigned, which defines the amount of CPU given to satisfy its processing requirements. A deadline is defined for each process within a SWP for scheduling purposes.

However, when moving towards parallel execution on many-core processors, tasks that form an application can be executed in parallel. Moreover, multiple parallel applications can run simultaneously (see Figure 2). In such an environment in order to guarantee incremental qualification, the time isolation property must remain the same as in single-core execution, i.e. the execution of a parallel application cannot affect the timing behaviour of other parallel applications. By doing so, we can guarantee time composability so the timing analysis of parallel applications can be done in isolation and will not be affected by other applications. This section presents the software mechanisms upon which the parMERASA architecture relies upon to accomplish the functional and time isolation.

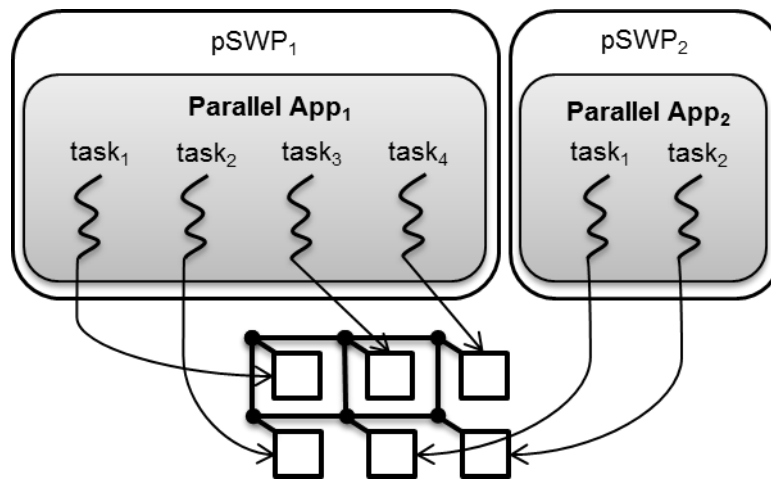


Figure 2. Two parallel applications composed of 4 and 2 parallel tasks respectively

parMERASA extends the functionality of SWPs, to allow parallel execution on multi-core processors. To do so, we introduce the *parallel software partition* (pSWP) concept. pSWP guarantees that parallel tasks belonging to one application cannot affect the timing (and functional) behaviour of parallel tasks belonging to other applications. Moreover, parallel tasks belonging to the same application, i.e. belong to the same pSWP, do not require fulfilling the robust partitioning, and so can interfere among them. Such interactions must be considered at WCET analysis time as defined in WP3 deliverables.

In order to fulfil the requirements imposed by safety critical applications, pSWPs must support the communication methods defined in the IMA and AUTOSAR software frameworks, i.e. intra-partition and inter-partition. To do so, communication among parallel tasks belonging to the same application is performed through shared memory using intra-partition communication methods. Such communication must occur within partition boundaries in order to guarantee time isolation properties. Similarly, communication among applications is performed through message passing methods using inter-partition communication methods. In order to guarantee functional isolation, pSWPs define two logical memory regions: A private memory region accessible only by parallel tasks belonging to the same pSWP through intra-partition communication methods, and a shared memory

region accessible only by the source and destination pSWP through inter-partition communication methods.

Unfortunately, inter-partition communication methods can break the time isolation property, because the memory device in which the shared memory region is stored can be in use by another application at the time the inter-partition message (which will be accessed by the destination pSWP in a future) is stored. That is, the inter-partition communication imposes an order in which partitions are executed, because applications are built with the assumption that the data coming from other partitions is available when they start executing and so guaranteeing no time isolation among producer and consumer. As a result, at the time the message is sent to the destination memory, the message can generate conflicts with shared resources (e.g. NoC, memory device) currently being used by other applications.

In order to solve this issue, we propose that in the context of pSWP execution, the impact that inter-partition communication requests can have on the WCET estimates of applications computed in isolation is not taken into account at application WCET analysis time, but at system integration time. This can be computed because the applications that will be affected by inter-partition requests are known at system integration time, when the different pSWPs are mapped into the many-core processor. In other words, pSWPs require mechanisms to increase the computed WCET estimate of an application in isolation by a delta factor which bounds the increment on the WCET estimate due to an inter-partition communication when the application is integrated into the system. This WCET estimate increment is shown in the Equation (1).

$$WCET_{\text{integration}} = WCET_{\text{isolation}} + \Delta_{\text{inter}} \quad (1)$$

Where $WCET_{\text{integration}}$ is the final WCET estimate of the application after system integration, $WCET_{\text{isolation}}$ is the WCET estimate of the application computed in isolation, i.e. assuming that no other application is running in the system, and Δ_{inter} the maximum increment that $WCET_{\text{integration}}$ can suffer due to inter-partition communications. It is important to remark that Δ_{inter} does not break time composability as the WCET estimate computed in isolation remains valid after system integration. Finally, pSWPs must also guarantee that the increment in the WCET estimate does not have side effects, i.e. do not affect any other pSWP.

Finally, the impact that inter-partition communication requests may have on the source pSWP is considered as part of its execution. Similarly, the destination partition assumes the data coming from an inter-partition communication as private data and the access to it will be performed through intra-partition communication methods. Next section proposes the hardware support required by any processor architecture to fulfil the requirements imposed by pSWPs to guarantee time isolation properties.

1.3 parMERASA Processor Design: Guaranteed Resource Partition (GRP)

Inter-partition communication methods defined by pSWPs generate requests to hardware shared resources, such as the NoC and memory devices, which may potentially generate interferences and break the time isolation guarantees imposed by pSWPs. Therefore, it is fundamental that the hardware provides mechanisms to ensure that the impact of such communication requests is avoided or bounded on the WCET estimates of applications.

We propose that many-core processor architectures tailored for use in safety critical real-time systems introduce a new hardware feature called *Guaranteed Resource Partition* (GRP). GRP defines an execution environment composed of a cluster of processor resources, including cores, NoC resources, memory, etc., in which pSWPs run, providing the desirable time isolation properties as defined above. A fundamental property that GRPs must accomplish is time predictability, i.e. the response time of the different processor resources must be known. Figure 3 shows a block diagram of the envisioned architecture composed of two GRPs, each formed by four cores, a NoC and a memory device.

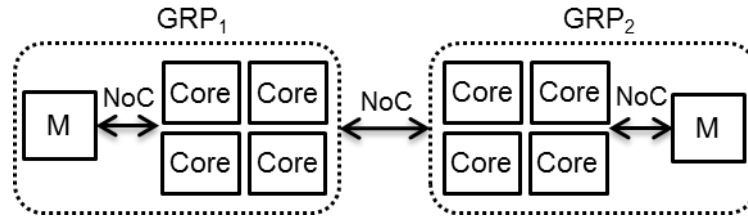


Figure 3. A processor architecture composed of two guaranteed resource partitions or GRPs.

In order to provide the time isolation properties to pSWPs, a request generated among parallel tasks belonging to the same application, i.e. intra-partition communication request (as well as instruction fetch memory request or process private data access) cannot exceed GRPs boundaries. To that end, each GRP has a private memory region in which intra-partition requests will access without interference from other GRPs. Moreover, the NoC design must guarantee that it does not exist a path from cores to memory that exits GRP boundaries. In case of inter-partition communication requests, sent data will be stored in the same memory associated to the GRP in which the destination pSWP will run.

However, at the time the message is sent, another pSWP can be under execution in the destination GRP, potentially generating interferences to the application that runs in it. In order to consider the impact that inter-partition communication may have on its WCET estimate (as expressed in equation 1), the memory device and the NoC must provide mechanisms to *freeze* intra-partition communication requests (as well as instruction requests and process' private data accesses), and let inter-partition communication requests to proceed. Since inter-partition communication is known statically at system integration and GRPs components are time predictable, the impact of inter-partition communication requests when traversing the NoC and the memory device can be easily determined [9], [12], [7], [2].

Therefore, the hardware shared resources implemented in the parMERASA architecture must implement *freeze mechanisms*. An important property of freeze mechanisms must accomplish is to guarantee that the resource state is not affected by the execution of inter-partition communication requests, so the contribution of intra-partition communication requests to the WCET estimate remains the same when running the application in isolation and in conjunction with other applications. This is not the case, for instance, of shared caches, in which the access of inter-partition communication requests may change the cache state, making intra-partition communication requests vary its timing behaviour with respect to running the application in isolation. In this case, the resource would require implementing cache partitioning techniques [9].

Note that the GRP is, in fact, the hardware counterpart of the pSWP: while the pSWP encapsulates parallel avionics applications to provide the desirable time isolation properties imposed by the standards, the GRP encapsulates the pSWP to provide the required time isolation guarantees at the hardware level. Next section presents the many-core processor designs that support the definition of GRPs.

2 NETWORK ON CHIP

In order to guarantee GRP properties, the network topology and the routing policy plays a fundamental role. In deliverable D5.1 from milestone MS12, we defined two network topologies of interest: *clustered* and *regular* topologies. This section presents the outcome of this research and proposes different parMERASA NoC designs that allow the definition of GRPs.

2.1 Definition of Physical GRPs

The processor architectures that fit the GRP requirements well are *clustered architectures* [3], [1]. In clustered architectures, usually a two-level hierarchical NoC is used: a first-level of NoC connects cores that compose a cluster and a second-level of NoC connects clusters among them. Figure 4 shows a clustered architecture deploying a two-level hierarchical NoC: a first-level composed of a tree, and a second-level composed of a bus.

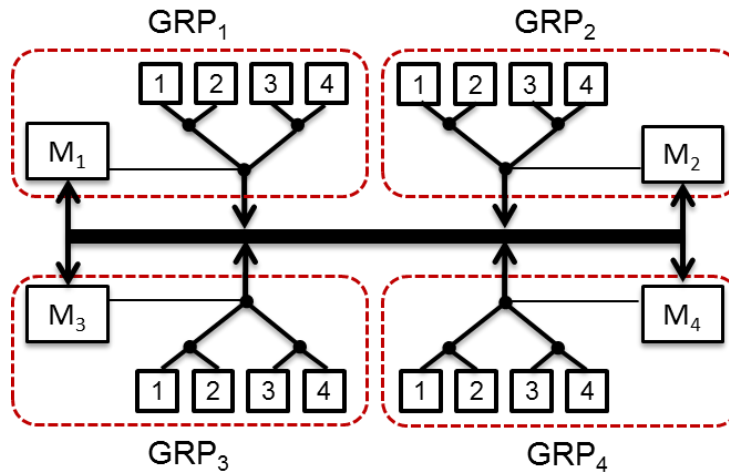


Figure 4. Clustered architecture composed by four clusters, each with four cores.

A proper design of hierarchical NoCs provides isolated islands of communications, in which different communication requests, i.e. intra- and inter-partition, use different levels of NoC that do not interfere among them. In Figure 4, the memory address space of the pSWP executed in GRP₁ will reside in memory M₁ and so intra-partition requests will only use its corresponding first-level NoC, i.e. a tree, without interference with other clusters. When a pSWP wants to communicate with other pSWP, the requests will traverse the second-level NoC, i.e. a bus, and the message will be directly stored in the memory resource corresponding to cluster in which the destination pSWP will run in the future without affecting other clusters. Note, however, that both communication requests will conflict in the memory device. Section 3 will address this issue.

The different NoC levels must also provide time predictability, i.e. the WCET estimate requires to account for the worst-case traversal time (WCTT), which expresses the maximum time a message can take due to NoC interference and that both communication flows, intra- and inter-partition, may suffer. The WCTT is composed of the sum of factors: (1) The *zero load latency* (zll) and (2) the *NoC request interference delay* (NoC_{RID}). The former provides the traversal time of a NoC request assuming zero interferences¹. The latter provides the maximum time a message may be delayed due to contending flows in the network when accessing the main memory.

In this deliverable, we use a notation similar to the one used in [10]: We define L as the number of flits of the request packet, $L_{I(x)}$ the number of flits of a packet of the contending flow $I(x)$ and z_c the number of contending flows. Moreover, we consider that requests and responses use different networks so the same analysis can be applied to requests and responses. Section 5 provides further details of how to consider the WCTT in the computation of the WCET estimate.

For the first-level parMERASA NoC design, we consider a wormhole-based tree topology implementing $2N - 1$ simple pipelined 2-to-1 router, with a traversal time of D_{router} cycles, to connect N cores [11], so each core requires $\log_2(N)$ hops to reach the memory or the second-level NoC. In such a NoC design z_c equals to one, so the maximum time a message is blocked at each hop is set the number of flits of the contending message ($L_{I(x)}$). Equation 2 computes the factors required to derive the WCTT of a tree.

$$\text{NoC}_{\text{RID}}^{\text{tree}} = \sum_{x=1}^{\log_2(N)} L_{I(x)}$$

$$\text{zll}_{\text{tree}} = (\log_2(N) \times D_{\text{router}}) + (L - 1) \quad (2)$$

The second-level NoC considers a non-pipelined bus with a latency D_{bus} implementing a round robin arbitration policy as designed in the FP7 MERASA project and presented in [9]. In a bus, the maximum time a message is blocked is set by the latency of the bus (D_{bus}), the number of flits of each contending message ($L_{I(x)}$) and z_c that equals to the number of GRPs minus one. Equation 3 computes the factors required to derive the WCTT of a bus.

$$\text{NoC}_{\text{RID}}^{\text{bus}} = \sum_{x=1}^{z_c} D_{\text{bus}} \times L_{I(x)}$$

$$\text{zll}_{\text{bus}} = D_{\text{bus}} \times L \quad (3)$$

Table 1 shows the zll and NoC_{RID} values for the clustered architecture shown in Figure 4, assuming $D_{\text{router}} = 1$, $D_{\text{bus}} = 2$ cycles and $L_{I(x)} = 4$.

¹ Note that given formulation holds only if the packet injection rate is limited, which is the case in parMERASA architecture.

Table 1. WCTT factors ($zll + NoC_{RID}$) for the clusterized NoC design (tree and bus) assuming pipelined routers with $D_{router} = 1$ for the tree, and $D_{bus} = 2$ and $L_{l(x)} = 4$.

Tree		Bus	
RID	zll	RID	zll
8	5	24	8

Clustered designs do not require implementing freezing mechanisms at the level of the NoC, because the two communication types do not interfere among them as they travel through different NoCs. This is not the case of the memory which will be addressed in section 3.

Despite the benefits of using clustered architectures to provide support to GRPs, the number of cores per GRP is fixed, and this determines the maximum parallelisation level that the pSWP can exploit. Increasing the number of cores would require pSWPs to use multiple GRPs, which would force intra-partition communication requests to traverse the second level of NoC, and so conflicting with inter-partition communication requests. In case the pSWP requires fewer cores, the remaining cores in the GRP cannot be assigned to other pSWPs as conflicts from different pSWPs would appear. Next section presents an alternative parMERASA NoC design in which the number of cores assigned to each GRPs can vary.

2.2 Definition of Virtual GRPs

In regular NoC designs, e.g. mesh networks, cores are organised in a regular 2-dimensional grid. These networks are very common in current processor designs due to their regular physical arrangement and short wire connections allowing high-speed operations among neighbor nodes.

Meshes also allow to define GRPs that fulfil the time isolation requirements. To do so, virtual clusters are defined by grouping adjacent cores in rectangular shapes (i.e. organizing cores in groups of 2, 4, 6, 8, 9) with a memory device connecting to one of the cores. In this case, if XY or YX routing policy is used [4], an isolated communication island is created with properties similar to those of clustered architectures. Figure 5 shows a processor implementing a mesh in which four GRPs are defined: GRP₁ composed of 6 cores, GRP₂ composed of 2 cores and GRP₃ and GRP₄ composed of 4 cores each.

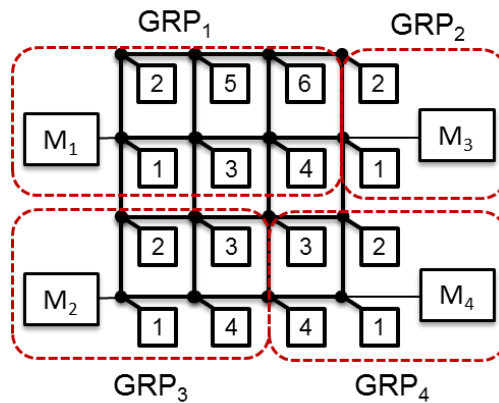


Figure 5. Regular architecture composed by four GRPs with different number of cores (6, 2, 4 and 4 cores respectively).

XY (or YX) routing policy guarantees that requests to nodes within the GRP never exceed its boundaries if the memory device resides within the virtual cluster. However, this is not the case for the requests of inter-partition communication. They can affect other partitions when accessing to memory devices belonging to other GRPs.

Hence, in order to consider such interference in the WCET estimate of the application a freezing mechanism is required. To do so, we propose the use of *virtual channels* [6], one per each communication type, providing higher priority to the inter-partition communication request queue (see Figure 6). Such design will force intra-partition communication requests to be stalled until the inter-partition communication finishes as imposed by the GRP definition. Note that the use of virtual channels accomplishes the freezing mechanism property that the state of the NoC does not change after inter-partition communication requests are executed.

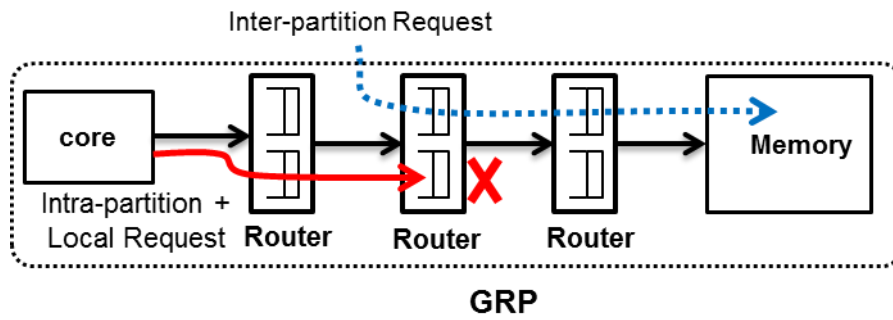


Figure 6. The freezing mechanism is implemented using virtual channels.

Meshes also allow computation of the WCTT and the sum of zll and NoC_{RID} in the same way we did for the hierarchical NoCs. We consider a mesh network design where requests and responses use different virtual networks as proposed in [6] and routers are pipelined. Overall, the proposed mesh implements 4 virtual channels: two for requests and responses of intra-partition and two for request and responses of inter-partition. Equation 4 computes the WCTT factors of a mesh.

$$NoC_{RID}^{mesh} = \sum_{j=1}^{hops} \sum_{x=1}^{z_c} U_{I(x)}^j$$

$$zll_{mesh} = (hops \times D_{router}) + (L - 1) \quad (4)$$

Where $U_{I(x)}^j$ is the time required for a message of contending flow $I(x)$ to go from the output buffer of the router in hop j to reach the input buffer of router in the next hop ($j+1$), and hops is the total number of hops to reach. Similarly to the tree, $U_{I(x)}^j$ is a function of the number of flits of the contending message ($L_{I(x)}$). We refer the reader to [10] for a detailed explanation of the expression above.

It is important to remark that, because the number of hops required to reach the memory depends on the core in which the request is issued, different cores have different WCTT. Alternatively, the worst WCTT could be considered, being a pessimistic solution though.

Table 2 shows the zll and NoC_{RID} of each core that form the four GRPs shown in Figure 5 (labelled as *2-Mesh GRP*, *4-Mesh GRP* and *6-Mesh GRP* respectively), assuming $D_{router} = 2$ and $L_{l(x)} = 4$. The *Core Id* refers to the location of cores shown in Figure 5.

Overall, meshes allow to define virtual clusters in which the number of cores is not fixed by the hardware, providing higher flexibility than hierarchical NoCs. Unfortunately, the WCTT is increased with respect to the physical definition of GRP (comparing Tables 1 and 2) with results in an increment of the WCET estimation. Moreover, the use of virtual GRPs complicates the computation of the WCET estimation as the WCTT depends on the core in which processes are located. Section 5.3 evaluates the timing impact of the two NoC designs.

Table 2. WCTT factors ($zll + NoC_{RID}$) for the regular NoC design (mesh) assuming pipelined routers with $D_{router} = 2$ and $L_{l(x)} = 4$.

Core Id	6-Mesh GRP		4-Mesh GRP		2-Mesh GRP	
	RID	zll	RID	zll	RID	zll
1	8	5	8	5	4	5
2	20	7	8	7	4	7
3	20	7	20	9	-	-
4	20	9	20	7	-	-
5	36	9	-	-	-	-
6	36	11	-	-	-	-

3 MEMORY HIERARCHY

All the communication methods considered in parMERASA are implemented through memory. Hence, inter-partition communication and intra-partition and local partition memory accesses may potentially conflict in the NoC (either clusterized or regular designs) and the memory, which implies long latencies for accessing application's shared and private data. As a result, a suitable and time predictable memory hierarchy design is required. This section presents the memory hierarchy proposed for the parMERASA architecture.

3.1 Cache Hierarchy

The parMERASA cache hierarchy comprises two private caches per core, i.e. an instruction and a data cache. A second level of caches is not foreseen because of the difficulties related to static WCET analysis. *Figure 7* shows the location of the caches in the parMERASA architecture.

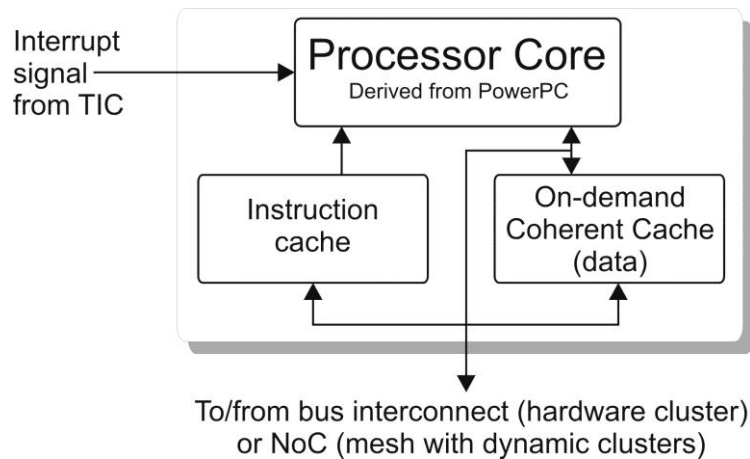


Figure 7. Caches in the parMERASA architecture

The instruction cache is a regular cache in which cache coherence is not required. This is not the case, however, for the data cache that needs to provide coherent accesses to shared data.

3.1.1 Cache Coherence Protocols

The major disadvantage of common coherence protocols is the unpredictable timing behaviour and cache state. For a tight WCET estimation, a sufficient knowledge of the cache content is required as well as defined latencies for cache hits and misses. The possibility of modifications triggered by other cores corrupts the chance to predict if data is stored in a cache line at a specific time. Moreover, the access latencies of caches are much harder to predict.

The following three scenarios demonstrate possible interferences between multiple local caches resulting in corruption of content and mutated timing behaviour:

- 1) *Validity of cached data.* In invalidation-based and update-based coherence protocols, the lifetime of a specific cache line is hard to predict. A cache line containing data which is shared with other cores can become invalid if another core intends to write on that data. This makes the prediction of the cache's contents practically impossible.
- 2) *Latency of cache misses while accessing shared data.* A cache miss of shared data typically generates a read burst from the main memory. Since the data in the main memory could be outdated and the modified data could be present in another core's cache, a previous write back procedure of the possessing core may be necessary. Considering the fact that the possessing core is also waiting for another core to proceed, the latency of cache misses at accesses to shared data is mutable since it is not known when the other core is able to handle the request.
- 3) *Latency of cache misses while shared data is hold by the cache.* The execution time of a specific task can be influenced by another core. If a core is holding a modified cache line with shared data, it can be forced to write back that cache line to re-establish the validity of data in the main memory. This intervention can delay a local cache miss, resulting in an unpredictable miss latency.

The parMERASA architecture introduces the *On-demand Coherent Cache* (ODC²) that permits a predictable, timing analysable cache coherence mechanism, suitable for hard real-time multi/many-core systems. The ODC² is designed to allow estimating reasonable worst-case latencies of cache

accesses to private as well as to shared data, omitting unpredictable properties of common coherence protocols.

3.1.2 On-demand Coherent Cache (ODC²)

The aim of ODC² is to facilitate a fast access to private as well as to shared data when required while allowing a tight static WCET analysis. Moreover, the ODC² works without any *inter-cache communication* and the state of a cache solely depends on the behaviour of the related core.

The basic idea is to hold shared data only as long as necessary and to force (re-)loading of possibly modified shared data at the time of use. Hereby, use does not mean individual accesses but rather instruction sequences which access shared data. In order to enable ODC² to operate properly, two preconditions must be fulfilled:

- At any time, only a single core accesses a particular portion of shared data. This means that accesses to shared data need to be protected by synchronisation techniques like critical sections or barriers, and
- A cache line must not contain private and shared data in parallel, i.e. a cache line can only contain private or shared data.

Both preconditions do not restrict the suitability of ODC² in practice. In order to ensure consistent accesses to (portions of) shared data, these accesses need to be protected via a synchronisation technique anyway. If only a single shared variable is accessed atomically, mutual exclusion is not necessary. In this case, a work-around is possible to allow the use of ODC². Regarding the contents of the cache lines, shared data must not be placed together with private data. Data is handled on the granularity of cache lines by ODC². Thus, it is required to force an alignment of private/shared with respect to the size of cache lines.

The ODC² maintains coherent memory accesses only if accesses to shared data are intended. Otherwise, when solely private data is accessed, the behaviour of ODC² is equal to a non-coherent cache. In the following, the technique of the ODC² is described in detail.

3.1.3 Technique of the ODC²

The ODC² strategy is based on the usage of two different cache working modes, the *private mode* and the *shared mode*. During the execution it is possible to switch between these modes, triggered by the application. In private mode, ODC² acts as a standard cache controller without any coherence functionality. In this mode only private data is cached and no accesses to shared data are allowed. When the shared mode is activated, ODC² shows additional functionality and uses additional information inside a cache line.

Figure 8 shows the components of an ODC² cache line, including the additional shared bit. After a cache miss in shared mode, the loaded cache line is marked with the shared bit. This bit marks the cache line as a potential carrier of shared data on which coherence actions are required. During shared mode, all memory accesses are treated in the described way.

This procedure continues until shared mode is deactivated. With the deactivation of shared mode, all subsequent memory accesses are stalled and the cache controller performs a *restore procedure*. The task of the restore procedure is to invalidate all potential shared data inside the cache and to eliminate the data inconsistency between the cache and the memory.

Both, write-back and write-through strategies are suitable with ODC². In case of a write-back strategy, all cache lines marked as shared and modified are flushed back to the main memory. In the case of a write-through technique, no additional flushing is required. Independent from the write strategy, all cache lines marked as shared are invalidated. After this operation all data classified as shared is flushed to the main memory and no shared data remains in the cache. A succeeding access to the same data, triggered by the same or any other core needs a new load from the memory. The activation and deactivation of the shared mode can be triggered by accesses to cache control registers using normal load/store instructions.

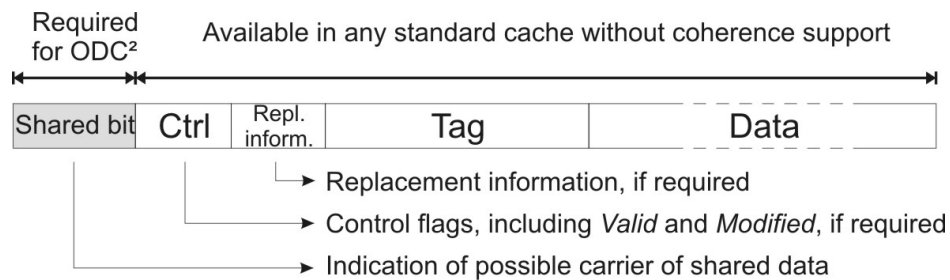


Figure 8. ODC² cache line extended by the Shared Bit

Besides the functionality of the cache controller to mark cache lines and the mentioned restore operation, the used cache memory has to be extended. For every cache line one additional status bit (shared bit) has to be included. The modified bit is already available in write-back caches. The marking of the cache lines as shared is done at the time it is loaded from the memory. Since the restore operation includes writing back the modified cache lines to the main memory, this procedure generates NoC and memory traffic depending on the amount of modified cache lines.

It is important to remark that the ODC² does not distinguish between the different communication methods defined in parMERASA, i.e. local, intra- or inter-partition communication, but instead private and shared memory accesses. Most applications define local memory accesses as private accesses and intra-partition as well as inter-partition communication as shared. However, the ODC² also allows each communication action to have individual data-sharing. It is therefore in the responsibility of the programmer to correctly define data sharing attributes. This is implicitly done by appropriate synchronization techniques as described in the next section.

3.1.4 Synchronisation techniques for ODC²

As mentioned in Section 3.1.2, the ODC² mechanism relies on the usage of synchronisation techniques to ensure consistent accesses to shared data. The ODC² mechanism can be applied with the common software synchronisation primitives that are suitable for correct parallel systems. An important precondition is the use of uncached synchronisation variables accessed with Read-Modify-Write Instructions, such as the *fetch&increment* instruction defined in the consolidated requirement CR5 introduced in D5.1 in milestone MS12.

The following synchronisation techniques are supported:

3.1.4.1 Spin Locks

Simple locking implementations using Read-Modify-Write operations and busy waiting can be used for ODC², although they are not a good choice for timing critical systems as defined in deliverable D4.1 from milestone MS12.

Example manual implementation	Example embedded implementation
<pre>-- Accesses to private data -- get_lock(lock_addr); ENTER_SHARED_MODE -- Critical section -- -- Accesses to private and shared data -- LEAVE_SHARED_MODE release_lock(lock_addr); -- Accesses to private data --</pre>	<pre>get_lock(lock_addr){ while !(RMW(lock_addr, 1)); ENTER_SHARED_MODE } release_lock(lock_addr){ LEAVE_SHARED_MODE lock_addr = 0)); }</pre>

3.1.4.2 Ticket Locks

Ticket locks are similar to spin locks, supplemented with a ticket based arbitration for fairness between concurrent threads. Use of ticket locks is recommended for hard real-time systems in D6.6 (Report on Experiences with Tiny Automotive RTE for Multi-cores to the AUTOSAR Standardisation Committee and to ISO 26262).

Example manual implementation	Example embedded implementation
<pre>-- Accesses to private data -- get_ticket(lock_addr); ENTER_SHARED_MODE -- Critical section -- -- Accesses to private and shared data -- LEAVE_SHARED_MODE release_ticket(lock_addr); -- Accesses to private data --</pre>	<pre>get_ticket_lock(lock_addr){ ticket = F&I(lock_addr->ticket_id) while (ticket != lock_addr->serve); ENTER_SHARED_MODE } set_ticket_lock(lock_addr){ LEAVE_SHARED_MODE F&I (lock_addr->serve); }</pre>

3.1.4.3 Mutex Locks

Instead of busy waiting for a lock, in mutex locks the waiting threads are managed and can be switched to suspend or to another task.

Example manual implementation	Example embedded implementation
<pre>-- Accesses to private data -- mutex_lock(mutex); ENTER_SHARED_MODE -- Critical section -- -- Accesses to private and shared data -- LEAVE_SHARED_MODE mutex_unlock(mutex);</pre>	<pre>mutex_lock(mutex){ get_lock(mutex->guard); if (mutex->lock != 0) then Enter waiting list release_lock(mutex->guard) Suspend thread else mutex->lock = 1 release_lock(mutex->guard) end if</pre>

<pre>-- Accesses to private data --</pre>	<pre> Set as mutex owner ENTER_SHARED_MODE } mutex_unlock(mutex){ LEAVE_SHARED_MODE get_lock(mutex->guard); if thread_in_waiting_list then wake thread unset as mutex owner else mutex->lock = 0 end if release_lock(mutex->guard) } </pre>
---	---

3.1.4.4 Semaphores

Similar to mutex locks, semaphores manage the activity of waiting threads. The arbitration is organised by a FIFO queue. In principle, semaphores allow multiple threads to access shared resources. Using ODC², only binary semaphores are allowed, guaranteeing only one thread can enter the lock.

Example manual implementation	Example embedded implementation
<pre> -- Accesses to private data -- wait(semaphore); ENTER_SHARED_MODE -- Critical section -- -- Accesses to private and shared data -- LEAVE_SHARED_MODE post(semaphore); -- Accesses to private data -- </pre>	<pre> wait (sem){ get_lock(sem->waitlist_lock); if (sem->counter <= 0) then add_to_wait_list release_lock(sem->waitlist_lock); suspend else release_lock(sem->waitlist_lock) end if ENTER_SHARED_MODE } post (sem){ LEAVE_SHARED_MODE get_lock(sem->waitlist_lock); if waiting_thread then unsuspend_waiting_thread release_lock(sem->waitlist_lock); else sem->counter++; release_lock(sem->waitlist_lock); end if } </pre>

3.1.4.5 Barriers

Barriers can be used with ODC² since the code section containing accesses to shared data is encapsulated by such barriers. It has to be ensured by design, that multiple cores executing that code simultaneously, access disjunctive sections of shared data.

Example manual implementation
<pre>barrier(); ENTER_SHARED_MODE -- Critical section -- -- Accesses to private and shared data -- LEAVE_SHARED_MODE barrier();</pre>

3.1.4.6 Barriers with conditionals

Barriers with conditionals solve the problem of possible deadlocks in barriers, but yielding in high WCET overestimation.

The above mentioned synchronisation techniques are just examples for common techniques supported by ODC². Many other are suitable, too. In general, ODC² supports any synchronisation mechanism which fulfils the following condition: *For all shared data which is accessed in code sections encapsulated by entering and leaving of the shared mode, solely one core is allowed to access a specific portion of data at a time.*

3.1.5 Performance impact of ODC²

The performance of ODC² must be discussed regarding the two modes, private and shared. During private mode only private data can be accessed without any coherence mechanism. This means that the performance of the ODC² is similar to any other cache following the same cache strategies (replacement and write strategy) and organisation: the number of cache hits and misses is identical as well as the latencies.

During shared mode, private as well as shared data can be accessed. Since in the beginning no shared data is hold by the cache, all first accesses to shared data result in cache misses. This could lead to an increased miss rate compared to other coherence protocols which could keep the shared data inside the cache. Moreover, private accesses that generate a miss are also treated as shared accesses. This private data, wrongly supposed to be shared, will be invalidated when leaving the critical section. Hence, if this private data is re-used outside the critical section it needs to be reloaded into the cache again.

The forced invalidation of shared data at the end of a critical section has also an advantage if a set- or fully-associative cache is used: If a replacement strategy like Least-Recently-Used (LRU) is applied together with a standard cache, the next eviction following the end of the critical section can concern private data loaded before the critical section. But the shared data, which has been used more recently, will stay unnecessarily inside the cache. A further access to the same private data will lead

to a cache miss. In case of ODC² the shared data will be invalidated immediately and hence, no eviction of private data occurs at the next miss.

We performed evaluations of the ODC² with the 3D Path Planning application from Honeywell, running on the parMERASA many-core simulator. We evaluated platforms with 1 up to 8 cores (on a 3x3 cluster topology) involved in the parallel execution. The execution times of the ODC² are compared with a (also statically analysable) platform using uncached accesses to shared data. For ODC², private and shared data is cached while synchronisation variables remain uncached. Only private data is cached during the execution on the uncached platform.

In Figure 9 the execution time of the 3D Path Planning application executed with the first configuration (shared memory in lower left cluster) is shown. The graph presents the results of parallel computation normalised to the uncached platform with 1 participating core. An immense reduction of the execution time can be seen with ODC² compared to uncached. Using ODC², the application benefits from the higher parallelisation degree when increasing the number of cores. On the unshared platform, parallelisation benefits can be achieved only up to 4 cores. With more than 4 cores, the results start to overtake, caused by the higher amount of accesses to the shared memory. The exceptionally low execution time with one core is partly caused by the omitted synchronisation overhead, but mainly by the fast access to the shared memory that lies in the same cluster. To provide a fair chance for all cores to access the shared data without hindrance, in the second configuration the global memory is placed in the middle cluster. As seen in Figure 10, the parallel execution with up to 4 cores is considerably improved by the centralised memory. Still the execution with ODC² outperforms the uncached execution and is less sensible to layout variations.

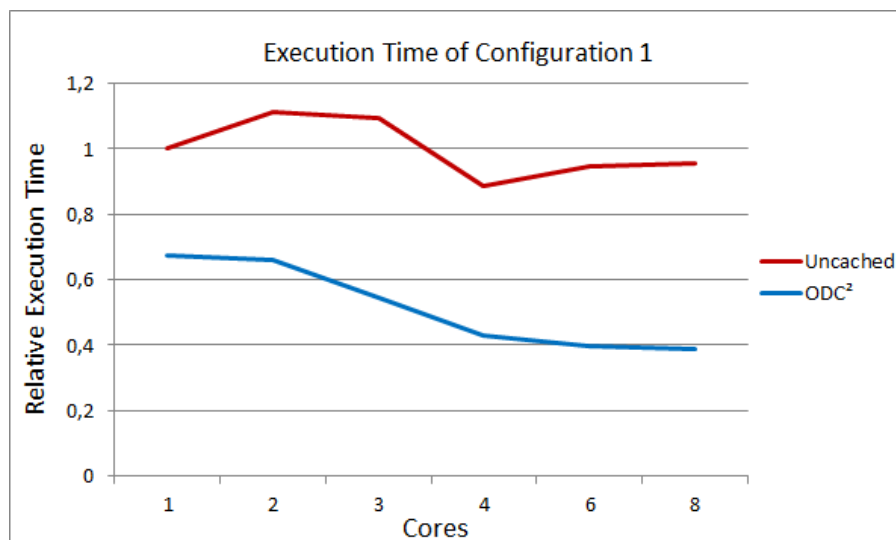


Figure 9. Normalized execution time of 3D Path Planning with memory controller in the lower left corner

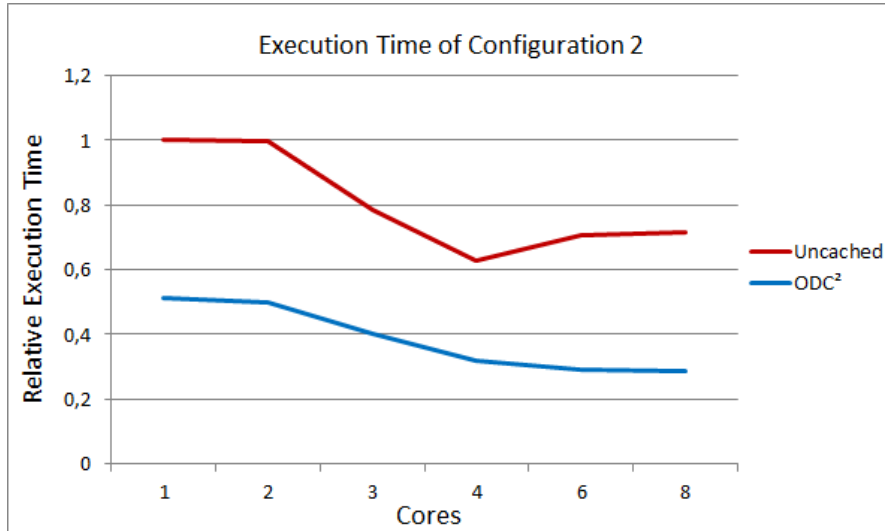


Figure 10. Normalized execution time of 3D Path Planning with memory controller in the center

3.1.6 Analysability of ODC²

Since ODC² is free of coherence transactions and access latencies do not depend on other cores, the timing analysability is similar to an uncoherent cache. In fact, during private mode, the ODC² has no impact on the analysability compared to an uncoherent cache. With ODC² a cache-line is regarded either as private or as shared. While every valid cache-line must be private during private mode, in shared mode all loaded cache-lines will be regarded as shared. The access latencies of cache-hits and cache-misses do not depend on that state so they are identical for private and shared mode. Solely the coherence operations affect the WCET estimation. In the following we regard the analysability of the particular ODC² coherent operations:

- *Entering of the shared mode.* The entering of the shared mode is done via an access to an ODC² control register and switches the internal state of the cache. This operation does not access the interconnect i.e., it shows a fixed latency, comparable to a cache hit.
- *Marking of cache lines.* During shared mode, every newly loaded cache line is marked as shared. The marking is done internally and do not create additional latency.
- *Restore procedure.* The required actions in the restore procedure are twofold. Every cache-line which has been marked before become invalid. This can be realised in parallel, resulting in a fixed latency. In case of a write-back strategy, invalidated cache-lines have to be flushed back to main memory. That is the only operation with latencies depending on the cache state. The latency is bounded by the number of modified shared cache-lines, which can be calculated by the static analysis. More details on the static WCET analysis of the ODC2 cache are given in Sections 2.3 and 3.2 of Deliverable D3.5.

3.2 Memory Controller

All memory requests that misses in the ODC² or access directly to memory (e.g. uncached memory accesses), may potentially conflict in the memory controller, which in turn can impact on the execution time and so WCET of applications, if the memory controller is not correctly designed, resulting in a violation of time isolation required by GRPs. In both NoC designs presented in section 2, this scenario can occur.

In order to address this problem, we propose a memory controller implementing a freezing mechanism in which the two types of requests are separated in two different queues, providing higher priority to the inter-partition communication request queue, so local partition memory requests are forced to be frozen until inter-partition communication finishes. Note that the use of two separated queues accomplishes the property that the state of the memory does not change after inter-partition communication requests are executed if the address space of types of accesses are separated, which is the case as imposed by pSWP.

Similarly to NoC design, the memory controller must be time predictable, i.e. the WCET estimate requires to account for the memory worst case response time (Mem_{WCRT}), which express the maximum time a message can take due to interference that both communication flows, intra- and interpartition, may suffer. To do so, we considered the memory controller design proposed in the FP7 MERASA project, in which the Mem_{WCRT} is composed of the sum of factors: (1) The request execution time (ret) [7] and (2) the Memory request interference delay (Mem_{RID}). The former provides the amount of time a request takes to be completed assuming no interferences. The latter provides the maximum time a request may be delayed due to other memory requests. In this deliverable, we use a notation similar to the one used in [10]. Equation 5 computes the Mem_{RID} factor of the memory. The ret , which depends on timing constraints of memory device operations (e.g. row buffer activation, read, write, precharge), is not placed due to its long expression. We refer the reader to [7] for a detailed explanation of the ret expression.

$$Mem_{RID} = \sum_{x=1}^{z_c} t_{LID}(x) \quad (5)$$

$t_{LID}(x)$ is the longest issue delay that a memory request may suffer considering the generic timing constraints defined in the JEDEC standard [5] and z_c the number of contenting flows.

3.3 Memory Map

Figure 11 shows a general overview of the parMERASA memory map. As pointed in Section 1.3, each GRP has its own memory; via the local interconnect, the cores of one GRP are directly connected to that memory. Physically, the cluster memory of another GRP can also be accessed by any core via the global interconnect, but this access is restricted by the Memory Protection Unit (MPU). The access to other GRPs is only used by the operating system for inter-partition communication, that is connected via a local interconnect. Therefore, every bit of memory has an individual physical address.

In addition to the physical address range, there is a logical address range that is used to map the private and shared memories of an individual core to a fixed memory address. This mechanism simplifies the address handling in the software, since code and data are always at the same logical address on every core, although the physical addresses are different.

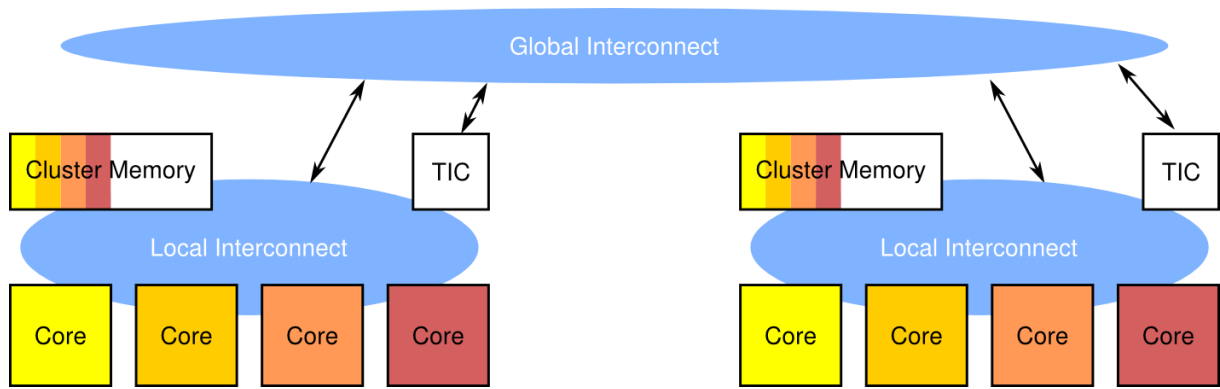


Figure 11. parMERASA memory hierarchy

This leads to the parMERASA memory map that is presented in Figure 12. The physical address space may contain up to 24 GRPs with up to 128 MB each. At the beginning of each GRP memory, each core of the GRP has a private memory space, where its code and data reside. These memory addresses are mapped to the logical address space at 0xf0000000 for code and 0xf1000000 for data (green arrow in Figure 12). The rest of the GRP memory is shared between the cores and divided into three parts: a cached memory area, where ODC² can be used, an uncached shared memory and the Message Passing Buffer (MPB). The MPB is used for inter-partition communication and is the only part of the GRP memory that can be accessed by a core from another GRP. The cached and uncached shared memory addresses are also mapped to logical addresses, but in contrast to the private mapping, this mapping is the same for every core within one GRP (red arrows in Figure 12).

Since the size of the GRP memory typically is smaller than 128 MB, the GRP address range contains addresses that are not mapped to memory. These can be used for memory-mapped communication with the Tiny Interrupt Controller (TIC) or with other external peripherals, that are linked to the GRP via the local interconnect.

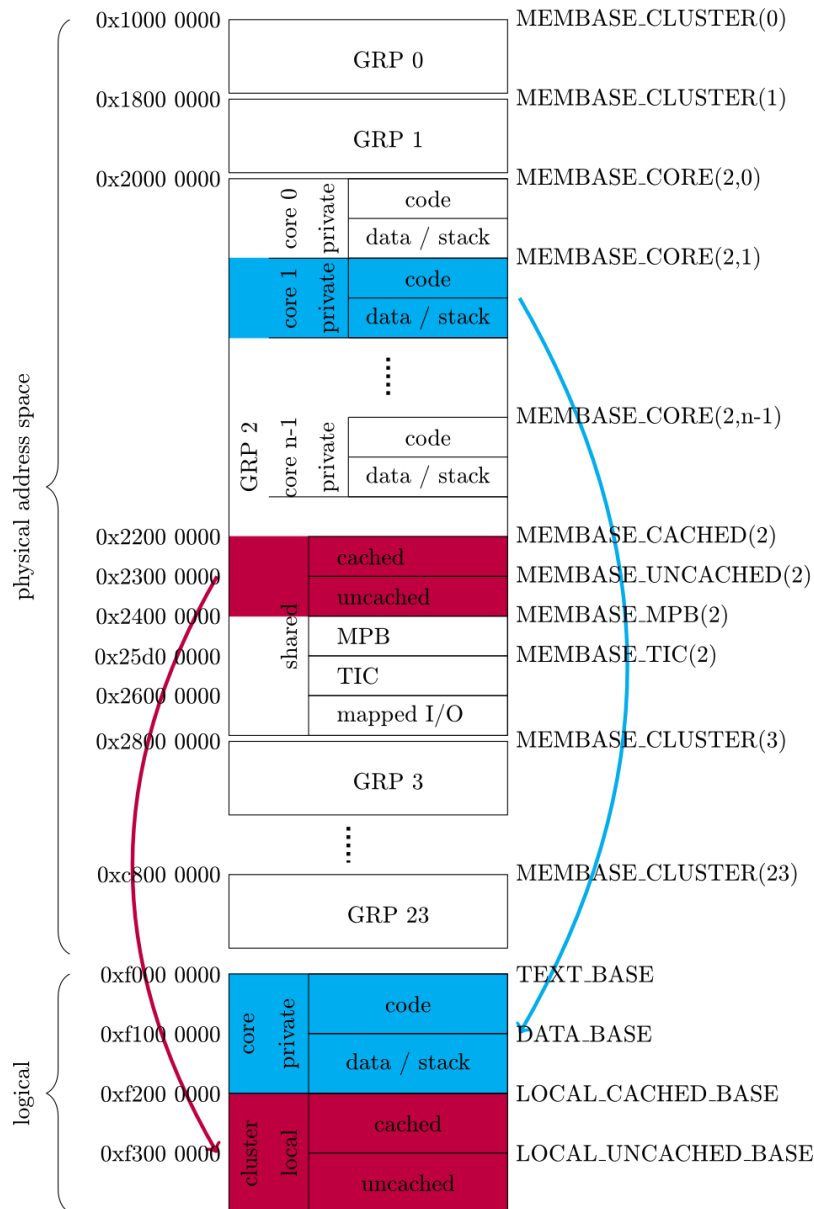


Figure 12. parMERASA memory map

4 I/O SUBSYSTEM

Because of the needs of the target applications, the parMERASA simulator has been equipped with I/O devices of three types and a flexible interrupt system. The three types of I/O device are a simulated CAN controller, a generic I/O module able to simulate several types of I/O and an EEPROM simulation. With these devices, the requirements as mentioned in Deliverable 5.1 are met, except for the DMA controller:

- 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 CAN module, the generic I/O module, and the EEPROM module are described in the following sections. Timers are integrated into the processor core itself and can be accessed by instructions writing to and reading from special control registers. The DMA functionality is not yet implemented in the simulator because there is currently no need from the applications at that time. It will be integrated in the next project phase, if required.

In addition to the I/O modules, a flexible interrupt system has been proposed and integrated into the simulation infrastructure. It is described in Section 4.4.

4.1 CAN Module

The CAN module is able to deal with incoming and outgoing CAN messages. An input trace file taken from a real hardware system (automotive and construction machinery applications) provides complete CAN messages together with timestamps. The module continuously compares the timestamp of the next message with the actual time, which is computed by the actual cycle number divided by an assumed clock frequency. If the given time is reached, the complete CAN message including identifier and eight data bytes is transferred into the registers of the CAN module. Afterwards, an interrupt is raised. The data bytes and the identifier can be accessed by any core using several functions of the system software.

Sending a CAN message works the other way round: a core needs to write the identifier and eight data bytes into the registers of the CAN module. After the eighth byte is written, the message is recorded in an output trace file, together with a timestamp. Sending a CAN message is also supported by the system software.

4.2 Generic I/O Module

The generic I/O module works similar to the CAN module with the difference, that the number of values per timestamp available in the trace file can be parameterized. Moreover, an interrupt signal is raised only if the new values read from the input trace differ in at least one value from the previous values.

The generic module can simulate parallel I/O, PWM outputs, analogue inputs as well as external interrupt sources. In case of the latter use, the input trace must contain changing values, for example alternating ones and zeros or values which are counter upwards in order to generate an interrupt signal at each time stamp.

4.3 EEPROM Module

The EEPROM module simulates the behavior of an external EEPROM used for configuration and status data. The content of the EEPROM is specified by an input file and can be modified by the application. The EEPROM's content at the end of the simulation is written to an output file which can be compared to the original input. Differences indicate possible problems or success during application execution.

4.4 Interrupt System

Peripheral devices need to gain the attention of a core in order to react on input data or to indicate the finalization of an output operation. Therefore, each peripheral device in a system is connected to an interrupt port of an interrupt controller which is connected to the core's interrupt input. The interrupt ports of the controller are prioritized so that a distribution of the devices to the ports brings a prioritization of their interrupts. To apply more devices to a processor than the number of its controller's interrupt lines, it is possible to cascade interrupt controllers. In these controller chains, the controller directly connected to the processor will work as a slave while the cascaded one works as a master.

Our approach is to take this cascading to a higher level. Instead of just connecting two interrupt controllers of the same kind in series we set the master and the slave controllers physically apart. We connect I/O devices not directly to a regular interrupt controller but to a so-called *Smart Interrupt Controller* (SIC) which then administrates the further distribution of the interrupts to the different cores of the system. To accomplish the distribution, the SIC sends a message to a so-called *Tiny Interrupt Controller* (TIC), which is located directly at the destination core. The TIC triggers the interrupt line of the destination core. The messages the SIC sends to the TIC will be transferred by the NoC. By this means it is unnecessary to wire I/O to any other device on the chip besides the SIC. The latency of the interrupt signal depends on the NoC. If the NoC provides time predictable behaviour then the interrupt handling with our approach will be timing predictable. Figure 13 shows the arrangement of SICs and TICs in the parMERASA many-core system. SIC and TIC are described in more detail in the following sections.

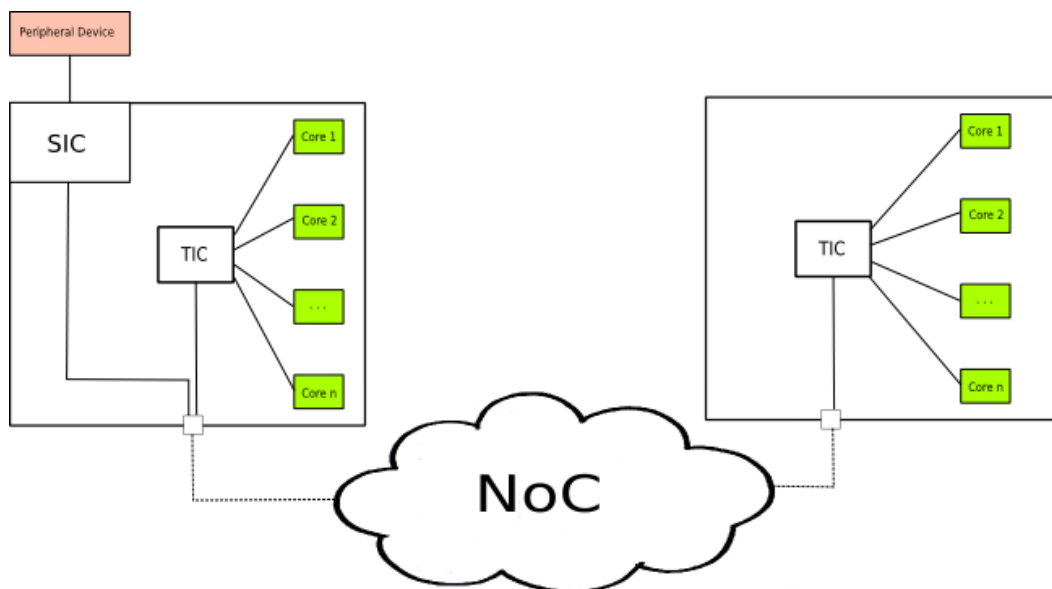


Figure 13. SIC and TIC in a clustered architecture

4.4.1 Smart Interrupt Controller (SIC)

A SIC is a hardware device which administrates the interrupt distribution in the parMERASA many-core system, independent from the actual NoC architecture. Every I/O device is connected to a SIC, each cluster contains one SIC. Each core in the system specifies during the boot-up phase which interrupts should be received from the SICs. This specification is supported by the TinyRTE. In consequence of the specification each SIC knows the TICs of the cores which need to receive interrupts. In the event of an occurring interrupt from an I/O device the SIC sends a message containing a predefined *magic number* to the corresponding TIC which is then in charge to inform the corresponding core. The *magic number* is specified by the receiving core during the boot-up phase and will be used to identify the interrupt source.

The important difference to other approaches is that there are no direct wires between interrupt sources and destinations. The messages are sent through the NoC. This means that adapting the method to a different system can be simply achieved by changes in the notification step. Nevertheless the system is deterministic because after the initial notification step the interrupt distribution remains the same.

4.4.2 Tiny Interrupt Controller (TIC)

In our approach a TIC is the hardware device which manages the transformation of the message sent by the SIC into a physical interrupt. The controller is wired to the interrupt ports of one or more cores (i.e., all cores in a cluster). If a network message reaches the TIC it sets the interrupt line for the corresponding core. The core receives the interrupt and reads the *magic number* from the TIC. Since only a single interrupt line per core is used, the source of the interrupt cannot be identified directly. Instead the *magic number* is set in a way that correlated to a single interrupt source. In the case of parMERASA, the *magic number* represents the starting address of the handling routine. This means, at boot-up phase, all SICs connected to interrupt sources are provided with interrupt destinations (target of the network message) and the address of the corresponding handling routine (magic number).

In every cluster which needs to receive interrupts a Tiny Interrupt Controller has to be placed. If multiple network messages reach the TIC (nearly) at the same time, they will be buffered and sent to the core in a FCFS manner.

4.5 Communication with the I/O Subsystem

Since parMERASA targets providing a generic architecture for multiple applications of different domains, the I/O devices can be located in clusters different from the one used by the corresponding application. This means that (slower) inter-cluster communication is required to access these I/O devices. In order to optimize access latencies, the location of the I/O devices can be taken into account at the application placement. An optimized placement can assign applications accessing I/O devices to the corresponding cluster or at least nearby. Because I/O devices are distributed to multiple clusters and an application will use several devices i.e. devices in the local cluster as well as devices in different clusters, the type of communication needs to be determined for each individual I/O device. The same issue occurs for the communication between SICs and TICs of the interrupt system. Here, the worst case latency is a major factor of the interrupt response time.

5 TIMING ANALYSIS OF THE PARMERASA ARCHITECTURE

5.1 Computing the WCET estimation of applications in Isolation

pSWPs and GRPs allow application developers to compute a composable WCET estimate of parallel applications that remains valid after system integration. This section derives the *upper bound delay* (UBD) that intra- and inter-partition communication requests may suffer due to interferences with other requests when accessing the NoC and the main memory, i.e. in case memory accesses misses in cache or are defined as uncacheable. The UBD can also be defined by messages generated by the interrupt system that requires traversing NoCs within and among GRPs. The UBD of each request can be then used by the timing analysis tools to derive the WCET estimation of the application which is time composable. The UBD concept and its use in the WCET estimation was defined in the FP7 MERASA project and presented in [9].

pSWPs and GRPs guarantee that intra-partition requests are only affected by other intra- and inter-partition requests belonging to the same application. Thus, the UBD of intra-partition requests (UBD_{intra}) depends on the $WCTT$ internal to the GRP and Mem_{RID} . Equations 6 and 7 show the UBD_{intra} for clusterized and regular NoC designs respectively. Recall that $WCTT_{mesh}$ depends on the core issuing the request as shown in Table 2.

$$UBD_{intra}^{cluster} = WCTT_{tree} + Mem_{WCRT} \quad (6)$$

$$UBD_{intra}^{regular} = WCTT_{mesh} + Mem_{WCRT} \quad (7)$$

Inter-partition requests instead, are not only affected by intra- and inter-partition requests belonging to the same application, but also by inter-partition requests belonging to other applications. Therefore, the UBD of inter-partition requests (UBD_{inter}) depends on both, the $WCTT$ internal and external of the GRP and Mem_{RID} . Equations 8 and 9 show the UBD_{inter} for clusterized and regular NoC designs respectively. It is important to remark that the $WCTT_{mesh}$ must consider the path accessing to a memory not located in the same GRP, and it depends on the core issuing the request as shown in Table 2.

$$UBD_{inter}^{cluster} = WCTT_{tree} + WCTT_{bus} + Mem_{WCRT} \quad (8)$$

$$UBD_{inter}^{regular} = WCTT_{mesh} + Mem_{WCRT} \quad (9)$$

5.2 Computing Δ_{inter} of the parMERASA Architecture

At system integration, the WCET estimation of one application computed in isolation ($WCET_{isolation}$) considering the UBD presented in previous section, can be affected by inter-partition communication as expressed in Equation 1. Concretely, inter-partition requests may delay intra-partition requests because of the higher priority in NoC and memory. Note that the impact that inter-partition requests may have on other inter-partition requests coming from other pSWP is already considered in the $WCTT$ used to compute the $WCET_{isolation}$.

In safety critical real-time systems, the amount of data transferred in an inter-partition communication from the source to the destination partition is known beforehand, i.e. at system

integration time, so the application development becomes independent from the system integration. This allows computing the WCET increment (Δ_{inter}) of an application due to interferences that intra-partition requests may suffer in NoC and memory due to inter-partition communication requests. Equations 10 and 11 compute the Δ_{inter} for the clustered and regular processor architectures designs above.

$$\Delta_{inter}^{cluster} = \sum_{i \in P} N_{inter_i} \times Mem_{WCRT} \quad (10)$$

$$\Delta_{inter}^{regular} = \sum_{i \in P} N_{inter_i} \times (WCTT_{mesh} + Mem_{WCRT}) \quad (11)$$

P is the set of pSWPs that can simultaneously send an inter-partition communication requests to the GRP in which the destination pSWP runs, and N_{inter_i} is the number of inter-partition communication requests of the source pSWP i . In the worst case, all inter-partition requests will be serialised in the NoC and the memory device, not allowing intra-partition communication requests to advance until they finish. Note that in case of $\Delta_{inter}^{clusterized}$, the hierarchical NoC does not impact on intra-partition requests. The reason is that inter-partition requests do not require traversing the internal NoC of the GRP (the tree), to access the memory of the destination GRP. This is not the case of $\Delta_{inter}^{regular}$ that requires traversing the mesh to reach the memory of the destination GRP.

Δ_{inter} is a pessimistic although a trustworthy upper bound, i.e. it considers that every time an inter-partition communication request is issued, there is an intra-partition request stalled, which in turn will stall the execution of the application as well. However, such pessimistic assumption allows us not to consider the application internals and maintain the time composability property.

In order to illustrate the impact of Δ_{inter} , let's consider four pSWP (P1, P2, P3 and P4), in which P1 communicates with P4 (see Figure 14(a)). Further, consider two processor architectures with two GRPs composed of 4 cores each, in which P1 and P2 executes in GRP1 and P3 and P4 in GRP2. Note that P1 and P2, and P3 and P4 do not execute in parallel, but one after the other as only one pSWP can execute within a GRP at any point in time. Figure 14 (b) implements the hierarchical NoC described section 2.1 and Figure 14(b) implements a 4x2 mesh described in section 2.2. Under this scenario, the inter-partition communication between P1 and P4 accesses to memory M_2 at the same time P3 uses it. As a result, intra-partition communication requests of P3 can be affected due to conflicts in NoC and memory and so must be considered at system integration time as expressed in equation 1.

Finally, let's consider that P1 sends 512 bytes of data to M_2 (from the grey filled core in each processor architecture), split into 16 requests of $L = 4$ each. Assuming that both processor architectures considers two 256MBx16 DDR2 SDRAM 400B memory controllers in which $Mem_{WCRT}=72$ processor cycles [7],[8], with $ret = 32$ and $Mem_{RID} = 42$, and a $WCTT = 47$ for the mesh (corresponding to the Core Id 6 in Table 1), the increment suffered by P3 due to inter-partition communication requests in both architectures are clusterized inter = 1152 and regular inter = 1934 cycles.

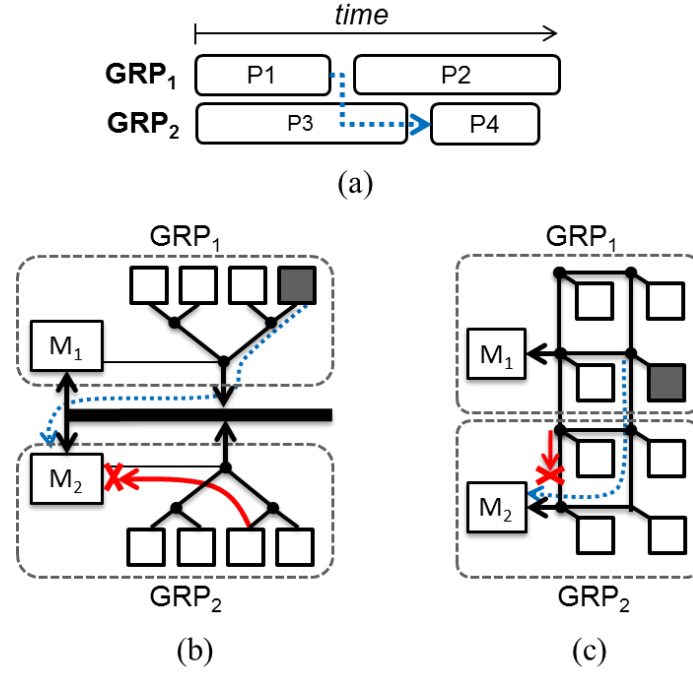


Figure 14. Four pSWPs (P1,P2,P3 and P4) executed in two many-core processor architecture with two GRPs and four cores each, (P1 and P2 run in GRP1 and P3 and P3 run in GRP2), composed of a hierarchical NoC with a tree and a bus (a), and a 4x2 mesh (b). From the grey filled core, 16 inter-partition requests are issued to M2 memory.

5.3 Evaluation of Δ_{inter} at System Integration

This section evaluates the timing impact of the two NoC designs presented in Section 2 at system integration. To do so, we compare the *observed execution time*, defined as the execution time of the application under analysis after system integration and so considering the delay due to inter-partition requests; and the *predicted execution time*, defined as the sum of the execution time of the application executed in isolation and the Δ_{inter} computed in equations 10 and 11.

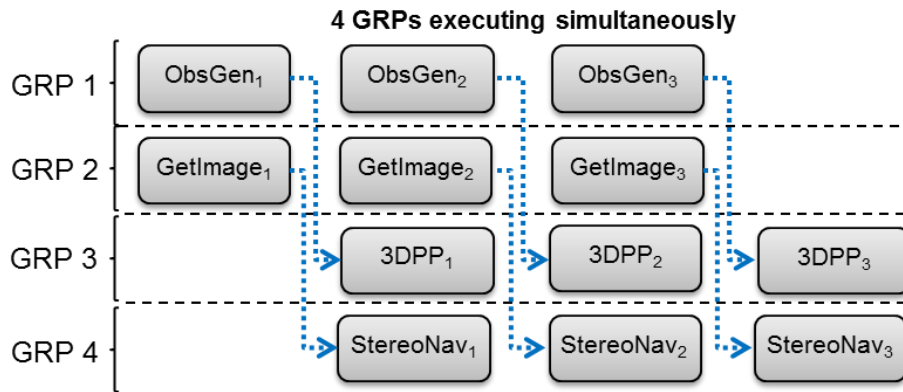


Figure 15. The four pSWPs are executed following a software pipelining approach, so the data generated by ObsGen and GetImage at stage n is consumed by 3DPP and StereoNav respectively at the next stage $n + 1$.

The system considered in this section is composed of four applications executed simultaneously in four GRPs: two parallel avionics applications provided by Honeywell, 3D path planning (3DPP) and

stereo navigation (StereoNav) (see deliverable D2.1 from Milestone MS1) and two applications that provide the data required by 3DPP and StereoNav: ObsGen and GetImage. ObsGen provides the 3D grid obstacle map required by the 3DPP; GetImage provides the two images required by the StereoNav. In both cases, the data is transmitted using inter-partition communication requests.

The four applications are executed following a software pipelining approach (see Figure 15). The data generated by ObsGen and GetImage at stage n is stored in the memory of the GRP in which the pSWP runs, and it is accessed by the 3DPP and StereoNav respectively at the next stage $n + 1$. By doing so, four pSWP can execute in parallel.

The experiments presented in this section are executed on the parMERASA simulator (see deliverable D5.4 from milestone MS13). We model the two 16-core NoC designs presented in Section 2: a clusterized and a regular architecture, both implementing 4 GRPs with 4 cores each. The two NoC designs fulfill the WCTT computed in Tables 1 and 2. Finally, the simulator also models separated instruction and data caches of 4 KB each in each core and four 256MBx16 DDR2 SDRAM 400B memory controllers, one per GRP, implementing two queues each (one per communication type). Higher priority is given to the queue used by inter-partition communication requests. We assume that CPU frequency doubles the memory frequency. This configuration provides a MemWCRT = 72 processor cycles [7],[8].

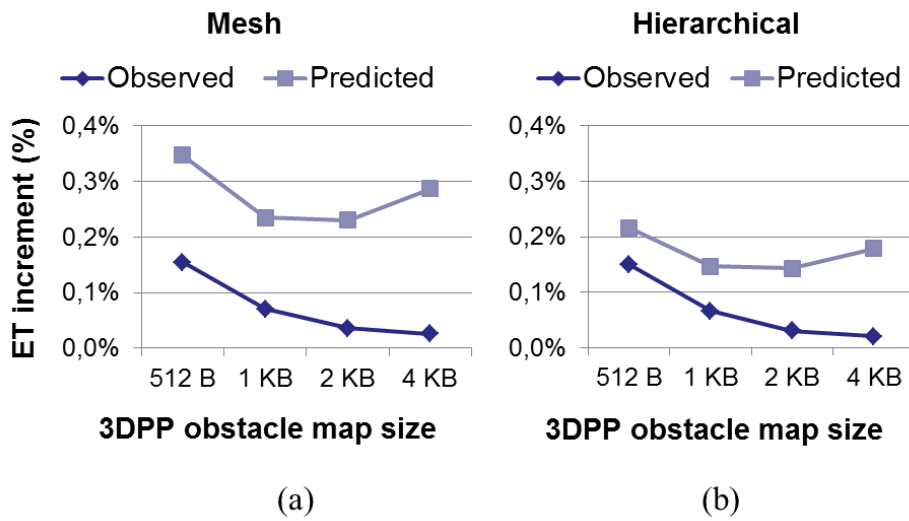


Figure 16. Observed and predicted execution times increment (in %) of 3DPP with respect to its execution time in isolation, when running simultaneously with ObsGen and considering different 3D obstacle maps sizes.

Figure 16 shows the observed and predicted execution time increments of 3DPP when ObsGen transmits obstacle maps of different sizes (512 bytes, 1 KB, 2 KB and 4 KB) for the next execution of 3DPP (see Figure 15). Figure 16(a) and (b) consider a mesh and a hierarchical NoC design respectively.

The execution time of 3DPP increases due to interferences produced by inter-partition requests. The increment is very low (less than 1% for the observed and predicted) because the time required to process the obstacle map is much higher than the time required to transmit it. As expected, the predicted execution time always upper bounds the observed execution time. In the mesh (Figure

16(a)), the difference between the predicted and the observed execution time is higher than the difference seen in the hierarchical NoC (Figure 16 (b)). The reason is that in case of the hierarchical NoC, Equation 11, only considers the impact of the memory, while in case of the mesh, Equation 10, considers the impact of the NoC and the memory. When comparing the predicted execution times of both NoC designs, hierarchical NoC is 3% lower than mesh NoC design for all obstacle map sizes.

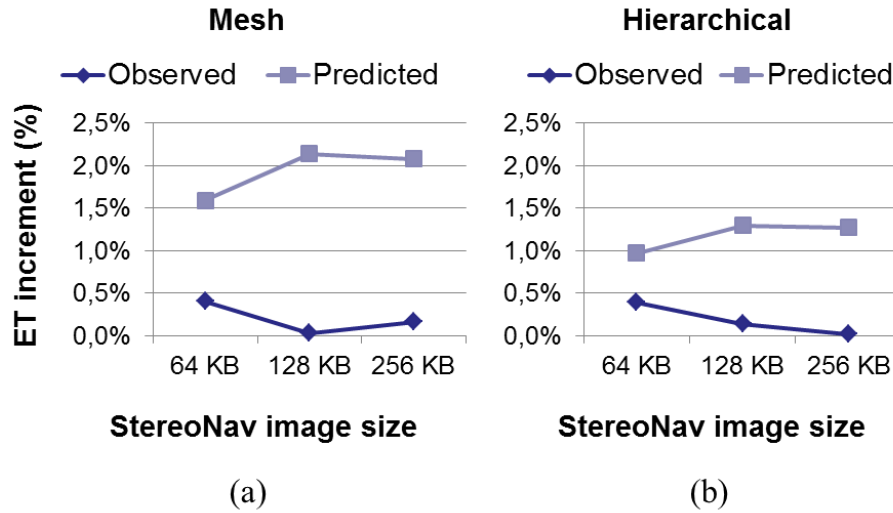


Figure 17. Observed and predicted execution times increment (in %) of StereoNav with respect to its execution time in isolation, when running simultaneously with GetImage and considering different image sizes.

Figure 17 shows the observed and predicted execution time increments of StereoNav when GetImage transmit images of different sizes (64 KB, 128 KB and 256 KB). Figure 17(a) and (b) consider a mesh and a hierarchical NoC design respectively. In all cases, the size of the image considered for the baseline, observed and predicted executions times are the same.

Similarly to 3DPP, the execution time of StereoNav increases due to interferences produced by inter-partition communications. The increment is very low, although higher than 3DPP (less than 2.5% and 1% for predicted and observed execution times). As expected, the predicted execution time always safely upper bounds the observed execution time. When comparing the difference between the predicted and the observed execution time, StereoNav follows the same trend of 3DPP: the difference in case of the mesh NoC design (Figure 17(a)) is higher than the one introduced by the hierarchical NoC design (Figure 17(b)). When comparing the predicted execution times of both NoC designs, hierarchical NoC execution time is on average 10% lower than for the mesh NoC design for all image sizes.

In the results shown in Figures 16 and 17 there is a relation between the amount of data transmitted through inter-partition communication and the amount of computation required. For example, if GetImage transmits 256 KB, StereoNav processes the same amount of data from previous stage, resulting in a small execution time increment.

6 CONSOLIDATED REQUIREMENTS

The requirements derived from the three target application domains are condensed into Consolidated Requirements for work package 5 (CR5.x) which have been considered as the baseline for the parMERASA hardware architecture design. Moreover, these CR5s have been used as success criteria to demonstrate the suitability of the parMERASA hardware architecture to be used for the targeted sample applications.

The following table shows all CR5.x defined in D5.1 together with the current state of implementation. Most of them are already completely fulfilled by the current parMERASA architecture since they are basic requirements of the target application domains. Others are to be implemented in the last phase of the project.

	Description	Compliance of current architecture	State
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.	The definition of GRPs can be done either statically, i.e. implementing a two-level hierarchical NoC, or dynamically, i.e. implementing a mesh with XY routing. In case of the static configuration, the simulator allows defining an arbitrary number of GRPs (clusters) and cores per GRPs.	√
CR5.2	Each cluster has access to a shared memory region that is protected from accesses of other clusters.	GRPs provide (local) cluster memories which can be accessed by all cores in principle. However, the memory map allows to restrict access rights.	√
CR5.3	There is an explicit communication between clusters.	The communication between GRPs is performed by memory accesses to the remote MPB. These accesses are encapsulated into explicit system functions providing inter-partition communication.	√
CR5.4	The hardware prototype supports the PowerPC instruction set.	The PowerPC 750 ISA has been chosen for the simulator.	√
CR5.5	The PowerPC instruction set is extended by an atomic read-modify-write instruction.	The instruction set has been extended by an atomic read-modify-write instruction.	√
CR5.6	It is possible to change between several communication schedules.	During the second phase of the project, parMERASA does not consider different communication patterns, but considers the UBD for each of the communication requests.	Postponed to optimization period
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.	The size of the message is considered in the timing analysis of NoC designs (see Section 2) and by the simulation infrastructure that allows defining messages of different size. In case messages require to be split in multiple packets, the different NoC designs (i.e. tree, bus and mesh) considered in parMERASA	√

		implements mechanisms to guarantee message consistency	
CR5.8	All cores are synchronized by a global synchronized time basis.	The distribution of the clock signal in many-core processor designs is a well-known problem in computer architecture. In parMERASA, we consider the solutions applied in many-core processor designs such as Tiler, Kalray or STM P2012.	√
CR5.9	Globally synchronized interrupts can be triggered by at least two independent external signal sources.	An extension of the SIC/TIC concept is planned for the last project period. This extension will allow multicast interrupts.	Postponed to optimization period
CR5.10	All interrupt sources can be masked individually.	This feature is supported by the interrupt system using SICs and TICs.	√
CR5.11	Each I/O-interface is flexibly assignable to cores.	Communication is performed using the NoC and the interrupts can be assigned flexibly by SICs and TICs.	√
CR5.12	Input and output is memory mapped.	The address space of I/O modules is located in the memory map of the parMERASA simulator	√
CR5.13	The input from peripheral I/O devices is simulated by input trace files.	Peripheral devices that need input read this input from trace files.	√
CR5.14	The output to peripheral I/O devices is stored in output trace files.	Peripheral devices generating output store that output in trace files.	√
CR5.15	There are at least 4 external CAN bus interfaces and one timer per core.	The timers are integrated into the cores for all application domains. CAN interfaces are integrated into the simulator for the construction machinery but the number has been reduced to three (changed requirement).	√
CR5.16	The processor is able to receive 1280 kB/s of input data.	The simulator assumes that all input data required by the application is available when the program starts. In any case, the parMERASA simulator can provide data any arbitrary input data rate if required.	√
CR5.17	The processor has a bidirectional external connection of at least 32 bit at 1 MHz.	See CR5.16	√
CR5.18	Each core has its own programmable timer.	Integrated into the cores (see CR5.15)	√
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.	The concept of intra- and inter-cluster communication provides predictable latencies for accesses to memory and I/O.	√
CR5.20	The timing of processor instructions must be	The chosen PowerPC 750 provides a fixed timing of all instruction except for memory	√

	predictable.	accesses. The latencies of these accesses depends on the location of the core and the memory (see CR5.19)	
--	--------------	---	--

7 REFERENCES

- [1] Kalray MPPA 256 Many-Core Processor, <http://www.kalray.eu/products/mppa-manycore>, Last access May 2013.
- [2] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable sdram memory controller. In CODES+ISSS, 2007.
- [3] L. Benini, E. Flaman, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In DATE, pages 983 –987, march 2012.
- [4] William James Dally and Brian Towles. Principles and Practices of Interconnection Networks. Elsevier, May 2004.
- [5] B. Jacob, Spencer Ng, and D. Wang. Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann Publishers Inc., 2007.
- [6] Robert Mullins, Andrew West, and Simon Moore. Low-latency virtualchannel routers for on-chip networks. In ISCA, 2004.
- [7] M. Paolieri, E. Quinones, and F. J. Cazorla. Timing effects of the memory system in real-time multicore integrated architectures: Problems and solutions. In Transactions on Embedded Computing Systems, 2012.
- [8] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time cmps. In Embedded System Letter (ESL), Vol. 1, No. 4, December 2009.
- [9] M. Paolieri, E. Quinones, F.J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In ISCA, June 2009.
- [10] D. Rahmati, S. Murali, L. Benini, F. Angiolini, G. De Micheli, and H. Sarbazi-Azad. Computing accurate performance bounds for best effort networks-on-chip. IEEE Trans. on Computers, 62(3), 2013.
- [11] A. Roca, C. Hernandez, J. Flich, F. Silla, and J. Duato. Enabling high-performance crossbars through a floorplan-aware design. In Intl. Conf.Parallel Processing (ICPP), pages 269–278, 2012.
- [12] Zheng Shi, Alan Burns, and Leandro Soares Indrusiak. Schedulability analysis for real time on-chip communication with wormhole switching. In IJERTCS, volume 1, 2010.