parMERASA

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

D4.9 – Evaluation of Different Parallel Patterns on top of Tiny Automotive RTE and Tiny Avionic RTE

Nature:	**R - Report**
Dissemination Level:	**PU - Public**
Due date of deliverable:	**31/03/2014**
Actual submission:	**31/03/2014**
Responsible beneficiary:	**UAU**
Responsible person:	**Theo Ungerer**

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:	June 20, 2012
Project start:	October 1, 2011
Duration:	36 month

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
Christian Bradatsch, Florian Kluge	author	2014-03-28
Theo Ungerer	WP leader	2014-03-28
Theo Ungerer	coordinator	2014-03-28

Deliverable Summary

Deliverable D4.9 “Evaluation of Different Parallel Patterns on top of Tiny Automotive RTE and Tiny Avionic RTE” concerns the integration and evaluation of parallel patterns outlined in D2.4 on top of tiny automotive RTE and tiny avionic RTE.

This deliverable comprises the support of a parallel pattern for the automotive domain by the tiny automotive RTE and likewise the adaption of the tiny avionic RTE. Furthermore, it describes different implementation approaches of cross core OS system services relevant to AUTOSAR basic software as well as application software and their evaluation.

Deliverable D4.9 summarises the work done in tasks T4.6 and T4.7 to fulfil milestone 11.

Task description of T4.6 (m25::6m):
Refinement of System Architecture and System-Level Software (see DoW, sect. 1.3.3, p. 49)

- Multi-tasking on a node integrated into the RTOS kernel.
- Synchronisation and node internal scheduling.
- Adjust system functions for cache and MMU management.
- Target after month 30 is an optimised RTOS kernel providing full synchronisation primitives.

Task description of T4.7 (m25::6m):
Parallelisation techniques for tiny automotive RTE and tiny avionic RTE (see DoW, sect. 1.3.3, p. 49)

- React on current (state of month 24 after start of project) AUTOSAR and IMA standardisation releases.
- Investigation of multi-core specific parallel interfaces for AUTOSAR and IMA.
- Parallel execution of AUTOSAR and IMA system services.
- Mapping of parallel execution patterns (from WP2) to tiny automotive RTE and tiny avionic RTE for multi-cores.
- Report on findings with parallel execution patterns on top of tiny automotive RTE to the AUTOSAR Standardisation committee (see also D6.8 in WP6).
- Target after month 30 are evaluation of different parallel execution patterns on top of tiny automotive RTE and tiny avionic RTE interface and recommendations for parallel support in future AUTOSAR standards to the Standardisation Committee.

Conclusion of task T4.6 and T4.7:

All targets of task T4.6, T4.7 and deliverable D4.9 have been reached.

Milestone M11 (m25 :: 6m):
Optimisation and Refinement (see DoW, sect 1.3.3, p. 52)

- optimised RTOS kernel providing full synchronisation primitives.
- evaluation of different parallel execution patterns on top of tiny automotive RTE and tiny avionic RTE interfaces and recommendations for parallel support in future AUTOSAR standards to the Standardisation Committee.
- parMERASA system software is sufficiently optimised to allow making it publicly available under an Open Source license at end of project.

Conclusion of milestone M11:

All objectives of milestone 11 have been achieved.

Table of Contents

1 Introduction

Deliverable 4.9 comprises the adaptions and extensions to the tiny automotive RTE and the tiny avionic RTE to support the parallel execution patterns of the automotive and avionic applications. The higher level communication mechanisms of both tiny RTEs are also adjusted to the needs of the parallel patterns. The next section describes the extensions that were introduced for all RTEs. Sections 3 and 4 explain the work performed for the tiny automotive RTE respectively the tiny avionic RTE.

2 System Software

The system software is extended by an inter-core notification mechanism and a highly reduced message passing interface to support basic techniques used for the realization of some parallel patterns. The inter-core notification mechanism is based on the configurable interrupt system and enables to trigger interrupts between arbitrary cores. The hardware support is explained in deliverable 5.x and the corresponding low level services are integrated into the Kernel Library. The service enables to trigger the interrupt of a specific core. It is up to the particular runtime environment to provide a proper interrupt handler, which performs the desired action.

The downsized message passing interface provides adapted blocking send (`MessageSend`) and receive (`MessageRecv`) operations. Both operations have the following parameters: pointer to the send/receive buffer containing the message, buffer respectively message size, destination/- source core, and a message tag.

```
void MessageSend (void *buf, int size, int dest, int tag);
```

```
void MessageRecv (void *buf, int size, int source, int tag);
```

The `MessageSend` parameter `dest` indicates the target core and the `MessageRecv` parameter `source` is used to accept only those messages from the specified source core. The message tag contains a service ID to distinguish between the various RTE services. Only if the tag on sender and receiver side equals, the message is received. `MessageRecv` also accepts an ANY_SOURCE and ANY_TAG parameter for `source` and `tag`. So, it can accept any message from an arbitrary source. The message itself has variable size and content, depending on the particular service of the RTE. It is stored in a local buffer with the `size` of the message and referenced by `buf`.

The send operation writes the message stored in the local buffer referenced by `buf` into a global buffer. `MessageSend` blocks until the message is completely buffered. For each receiving core a separate global message buffer exists. Moreover, each global receiving buffer is further divided into *channels*, one for each sender core. In a quad-core processor, where all cores are exchanging messages with each other, four global buffers exist, each being divided in four channels.

3 Tiny Automotive RTE

The tiny automotive RTE together with the Kernel Library [1] forms the domain specific runtime environment for the automotive application. It provides an OS based on AUTOSAR and the system services required by the application. Furthermore, communication and synchronization mechanisms are provided to exploit application parallelism.

3.1 Support for Parallel Pattern

In deliverables 2.4 and 6.8 the parallel execution pattern of the automotive application is described. It is based on the time and event triggered model of PharOS [2]. The tiny automotive RTE is extended to support this execution pattern [3].

PharOS is based on the time-triggered execution model of OASIS [4]. OASIS defines an approach for the design of safety-critical systems, e.g. in nuclear power plants. In OASIS, all tasks are executed in a time-triggered manner. Each task ω has an associated real-time clock H_ω. This clock is defined by the time instances at which input and output of the task can occur. Tasks can use an instruction *ADV(n)* to advance their clock by *n* instants. A task is then activated again *n* instants after its last activation. During clock advancement, execution of the calling task is blocked.

For communication between tasks, OASIS defines temporal variables and an asynchronous message passing mechanism. Both mechanisms are coupled to real-time clocks. Sent data is visible at the receiver at predefined time instants. The implementation of these mechanisms ensures that neither senders nor receivers experience blocking times. This model allows implementing concurrent tasks without explicit synchronisation, leading to a less pessimistic WCET analysis as no blocking times through interferences between tasks can occur. Indeed, the only blocking times that can occur are those requested by tasks themselves through the use of the *ADV* instruction.

PharOS extends the OASIS concepts for automotive systems using multi-core processors. PharOS partitions a system into time and event triggered domains. Each domain is assigned to dedicated cores. Time-triggered tasks are referred to as *agents* and executed on *computing cores*. In addition, PharOS places event-triggered tasks, called *handlers*, on *control cores*. The control core also processes I/O interrupts. This concept is demonstrated by an example where a handler task monitors a PWM signal for its duty cycle and sends measured data to an agent for further processing.

In the tiny automotive RTE, we go one step further. Our aim is to parallelise an interrupt handler and benefit from the advantages of the OASIS/PharOS execution model. The need for such an approach arises, if an interrupt handler not only has to read some input data, but also must process the data and set some output within a short deadline. If, furthermore, such an interrupt can occur with widely varying inter-arrival times, e.g. crank angles interrupt, it is hard to dislocate processing and output into the time-triggered domain, if possible at all.

In the tiny automotive RTE we realized a parallelised and synchronous processing of an IRQ handler. We assume that *agents* are executed exclusively on *computing cores* and *handlers* on *control cores* accordingly. An I/O interrupt is only triggered at one dedicated control core. On this core, an *initial handler* is executed which does not use the time-triggered execution model. The initial handler sends *messages* to activate processing of the IRQ by the *processing handlers* running on other control cores. Thereby, two requirements have to be fulfilled: (1) the latency between the occurrence of the

IRQ and the start of the actual processing must be statically boundable and sufficiently low to allow schedulability, and (2) the processing handlers must start execution at the same time such that their implementation can also use the time triggered execution model of OASIS/PharOS. From the second requirement follows, that an event triggered processing handler must also be able to use the *ADV* instruction like tasks executed in the time-triggered domain. For the *ADV* instruction a clock reference point is required. In the time-triggered domain this reference point is the start of the program.

To obtain such a reference point for processing handlers, we extend the messages sent by the initial handler by a future-timestamp. This future-timestamp represents the clock reference point. It is computed from the actual timestamp when entering the initial handler and the WCTT of the messages. Each receiving control core uses an *ADV* instruction to advance to this future-timestamp. Thereafter all cores start their processing handler synchronously at the same time.

To realise this approach, two requirements must be fulfilled by the underlying hardware: (1) A message mechanism must be available which signals an incoming message at receiver side immediately. (2) All cores need a common, fine-granular time base. The parMERASA many-core simulator provides a common time base for all cores by hardware and a flexible interrupt system (see D5.6).

The engine control application uses, amongst other things, an interrupt triggered by certain crank angle positions. The occurrence of this crank angle interrupt varies within a certain range depending on the rotation speed of the crank shaft. The interrupt signal is processed by the initial handler on a dedicated core. The aim is to execute a parallelised version of the crank angle interrupt handler routine. Currently, only one processing handler is executed exclusively on each control core. Due to that circumstance, a control core is actively waiting for an incoming message and thus needs not to be interrupted by a message notification. This simplifies the implementation.

After an interrupt is asserted at the dedicated interrupt core, the initial IRQ handler is started. The initial handler requests the actual time base and adds a WCTT offset that was calculated offline. The result is the clock reference point t_r which is stored at a specific location m_t in shared memory. Afterwards, the initial handler returns, and the core can also execute a processing handler. Each control core that shall execute a processing handler is spinning on the memory location m_t until its value changes. It then loads the value t_r into a register and executes an $ADV_{abs}(t_r)$ instruction, that advances to an absolute point in time. Thus, a synchronous start of all processing handlers is enabled.

In our case the WCTT of the message is calculated as the sum of the WCET of the store instruction executed by the initial handler and the WCET of the load instruction executed by the processing handlers. Measurements have shown that the deviation of the start time of each handler is up to 50 clock cycles. Additionally, the actual execution of the handlers starts 150-200 cycles after t_r elapsed. This circumstance is due to the fact that memory accesses to shared as well as private memory are routed through the interconnect of the simulated processor. Thereby, interferences on the interconnect occur and the latency for accesses is quite long. The same effect was also observed when an *ADV(n)* instruction with the same clock value *n* was executed on all cores in the time-triggered domain. So the deviations are likely to result because of the underlying hardware.

We have implemented an extension of the ADV instruction that was introduced in OASIS. This extension, ADV_{abs}, allows advancing execution to an absolute point in time. Thus, we achieve to synchronously start the threads of a parallelised interrupt handler, and their implementation is able to profit from the time-triggered execution model.

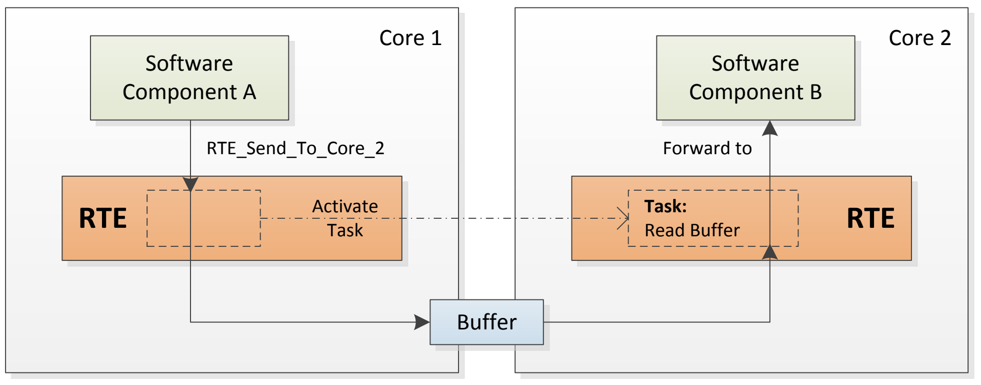

Figure 1: Sender-Receiver Communication across Cores between Application Software

3.2 Approaches of OS System Services for AUTOSAR BSW and Application Parallelisation

AUTOSAR OS [5] is an event-triggered real-time operating system descending form the OSEK/VDX OS [6]. It schedules tasks according to a fixed priority pre-emptive policy. Since the last revision of AUTOSAR Release 4.0, multi-core support is an inherent part of the OS specification. The AUTOSAR Specification of Operating System [5] describes the general multi-core concept, the API and functionality of OS services and their extensions for multi-core support. It also states the extensions required for OS services adopted from the OSEK specification. The AUTOSAR OS specification lists 17 former OS services, which have been extended to support cross core usage and 8 new services especially for usage in multi-cores. A few other services had to be adapted to work correctly.

OS system services are relevant for AUTOSAR BSW as well as application software. Especially the OS service *ActivateTask* is significant for communication between cores. Figure 1 shows a send/receive communication scenario between two software components (SW-Cs) belonging to different cores. This kind of mechanism is not only used for sender/receiver communication with notification, but also for client/server communication. A lot of inter-core communication is performed in this way.

3.2.1 OS Service

An OS service is a system or function call with the following properties: It can possess *input* and *output* parameters and has a *return status*. Almost all services have an input parameter passing an object ID, e.g. TaskID. It determines the OS object affected by the service call. Figure 2 shows the general structure of an OS service. The processing sequence starts with the service call. Next, a check of certain conditions takes place, for example, whether the passed input parameters are valid. If the check fails, the service returns immediately, otherwise the service performs its specific action and returns. A corresponding error value is given back at return. The program execution is delayed until the service returns, i.e. the service call is synchronous. A few OS services only request some system status information, whereas others make major changes to the system state and additionally require a rescheduling at service return.

The OS object ID parameter of a service helps identifying the affected core. Some OS services do not require an object ID, because they affect the same core where they were called. If the action of an OS

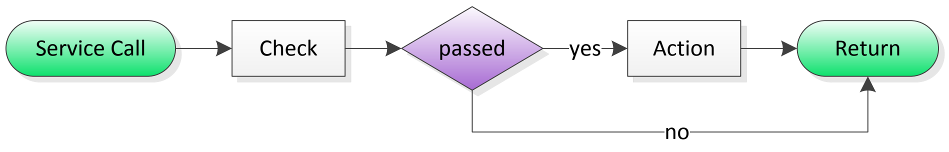

Figure 2: Service Call Processing Sequence

service called on one core (source core) is related to an OS object on a different core (target core), the action has to be performed immediately anyway. Only the core where a certain service was called gets the appropriate return status, regardless whether the action was executed on another core.

Nearly all automotive embedded systems postulate hard real-time requirements [7]. To comply with these requirements and to make analysis easier regarding real-time constraints, the whole AUTOSAR OS is statically configured. This means that the required amount of memory for all OS objects is directly derived from the configuration and known at compilation time. Furthermore, on a multi-/many-core processor each OS object, like tasks, is assigned to one specific core and each core has its own scheduler.

3.2.2 Concept

For many-core systems, we see two basic implementation approaches for cross core service calls. The two approaches are illustrated in Figure 3 and Figure 4. Both figures show the general processing sequence of an OS service call as presented in Figure 2. There are two time lines, the upper one representing the source core (SC) and the lower one the target core (TC). The first approach uses messages to notify the TC about a service call and to return the status to the SC. The service call is processed on the TC. The second approach uses a lock to work on the OS runtime data of the TC, for example, the task control block. Therefore, a global address space is necessary to access the data at the TC. The processing itself takes place on the SC. The main difference exists in that the service check and action is processed either on the SC or the TC.

A. Approach 1: Message Based

Figure 3 represents the first approach using messages. Task T1 on the SC calls a service with an ID of an OS object located on the TC as parameter. The SC sends a message to the TC, containing the service tag, e.g. *ActivateTask*, and the corresponding parameters. At this time, the TC executes task T2. The TC is interrupted immediately and starts checking the service constraints. After a positive check, the service call cannot fail anymore. Regardless of the outcome of the check, at this point a message is sent back to the SC with the return status. On the SC the service call returns and the program execution of T1 continues. If the check has passed, the TC processes the specific action of the OS service. Depending on the system state of the TC and the called service, a rescheduling may take place after it. So, either program execution of T2 continues or another task gets swapped in. Since a service call is synchronous, the calling core is blocked for the same period as the TC. If a second service targeting an OS object on the TC is called during the check or action phase, the service will wait until the action is finished.

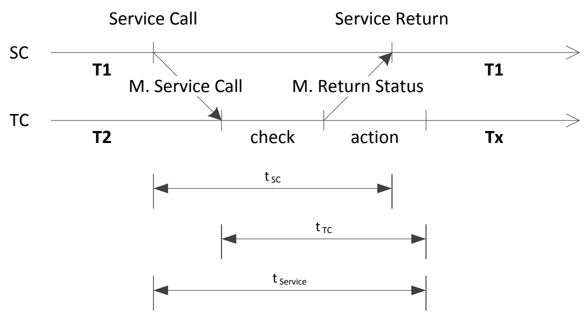

Figure 3: Message Based Service Call

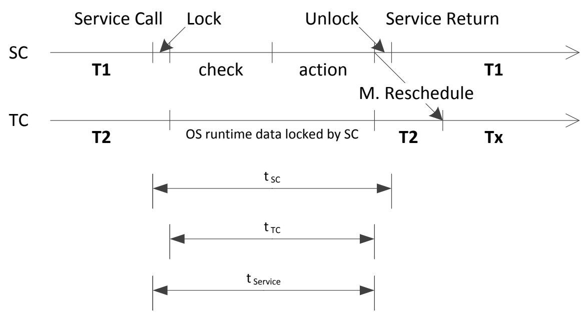

Figure 4: Lock Based Service Call

B. Approach 2: Lock Based

The second approach using locks is presented in Figure 4. Task T1 calls a service targeting an OS object located on the TC. The service tries to get the lock for the OS runtime data of the TC. After acquiring it, the preconditions are checked. Thereby, the SC accesses the data of the OS object, assigned to the TC by configuration. If all conditions are met, the action is performed on these object data. For example, in the case of an OS service requesting the actual information of a task bound to another core, the service action reads the status of the task control block on the TC. If the service does not require a rescheduling (cf. [6] p. 20 ff.), the lock is released and the service returns. Task T1 is resumed on the SC and T2 continues. Otherwise, the SC checks if a task other than the currently running task T2 has to be swapped in. If so, a message with the service tag *Reschedule* is sent to the TC. Afterwards, the SC releases the lock and resumes execution of T1. Task T2 gets interrupted by the message and checks again, if another task has to be started or resumed. Depending on the result, T2 is resumed or the highest priority task is loaded. The double check, whether a task swapping has to be done or not, is because between unlock and receive of the reschedule message the system state on the TC might have changed by other OS service calls. During the lock/unlock period, all service calls even called on the TC and regarding an OS object on the TC are blocked until the release of the lock.

C. Summary

In approach 1, the TC is interrupted for each OS service call affecting an OS object on it. The second approach only interrupts the TC, if a rescheduling is required. Both approaches have in common that during an OS service call an access to OS object data of the TC is not possible from a competing service call. On the one hand the access is prevented through the execution of the service on the TC (cf. Figure 3) and on the other hand by locking the OS object data (cf. Figure 4).

3.2.3 Discussion of Timing Behaviour

All AUTOSAR services can be used by application software that has to operate under hard real-time conditions. Therefore, the timing behaviour of a service call is of paramount significance for the system. Hard real-time constraints require that the execution time of any service call is statically analysable and safe timing bounds can be computed. Furthermore, possible interferences between different tasks must be boundable, if they cannot be omitted completely.

Concerning the execution of a service call, we argue as follows: the concepts underlying the service calls are already used in today's single-core implementations, where they provide timing predictability. This means that if the phases *check* and *action* are executed on a single-core processor, a WCET bound can be derived. Since the core implementation also applies to the multi-core variants of the service calls, the WCET predictability is conserved. The migration to multicore introduces some additional mechanisms into the service call that require further examination. Especially, we have to examine interferences that cannot happen in the single-core case, but most probably will do in the multi-core implementation.

In approach 1, where the service call is executed on the target core, we have to examine what can be happening while the service call message is being transmitted. Obviously, the worst case would be if similar messages were issued from other cores, and all would be processed before the one under examination. In such a scenario, the processing of the service call will be delayed. However, the delay can be bounded: Service calls are executed synchronously, meaning that a task that has a pending service call is blocked until the call returns. Thus, no task can send more than one service call at any time. As the system itself is configured statically, the total number of tasks n_t is already known at integration time, thus providing an upper bound to the number of service calls that might be active at a target core. The service call messages themselves are processed in order of their arrival. Therefore, a naive bound for the worst-case waiting time (WCWT, the time between issuing a service call and the start of the actual processing) would be $n_t * w_S$ with w_S being the worst-case execution time of the most complex service call that can occur. Further analysis of the system most probably will yield the additional information that not all of the n_t tasks will issue a service call to the specific target core, which will lead to a diminished WCWT.

In approach 2, the source core might also experience a waiting time until access to the lock is granted. Since we require a fair locking mechanism that provides FIFO access, the above considerations can also be applied here. Therefore, timing predictability of the service call is preserved for both implementations.

Additionally, we have to check the interferences that task T2 running on the target core will experience. In approach 2, where the service call is executed on the SC, T2 will only be interrupted if a rescheduling is necessary, i.e. if a task with a higher priority got ready. This behaviour is not different from the one that can be observed in today's single-core implementations: tasks with low priorities might get pre-empted by tasks with higher priority. Such behaviour must be covered by regular schedulability analysis like e.g. rate monotonic [8]. Furthermore, T2 may experience an additional blocking time if it executes a (local) service call while another task is holding the lock for the core's data structures. Similar to the above discussion, this waiting time is bounded by the maximum number of tasks that might access the lock at the same time. In approach 1, T2 gets interrupted any time a task running on another core issues a service call to the target core. A thorough analysis of the system and its behaviour can yield information about the occurrence rate of these interruptions. Using well known techniques like e.g. event streams [9], it is possible to account for these interruptions in the schedulability analysis for T2 and thus still achieve a predictable timing behaviour.

Although both approaches appear feasible under real-time constraints, approach 2 seems more appealing under the following viewpoint: approach 1 interrupts the target core in any case, regardless whether actual execution on the core must actually be changed due to the service call. Thus, we expect approach 2 to achieve at least a higher average performance.

3.2.4 Implementation

For the implementation, appropriate support from the system is needed. The first approach requires a synchronous inter-core notification mechanism and the second one a fair locking mechanism. The parMERASA many-core simulator covers both requirements. Between cores, interrupts can be used for inter-core notification. The Kernel Library offers a fair locking mechanism in form of a ticket lock implementation based on a fetch-and-add instruction [10].

For the message based approach, the message passing mechanism stated in section 2 is used. In the case of the tiny automotive RTE, the RTE services are reflected by the tiny automotive RTE OS services. So, the message tag contains the OS service ID. The message content itself encapsulates the source core ID, OS object ID, and the required service call parameters. Only the OS services *StartOS*, *ShutdownOS*, and *ShutdownAllCores* do not have an object ID.

A. Approach 1: Message Based

In the first approach, each service call is forwarded and executed on the target core. At the beginning, each service routine checks, whether the passed OS object ID is valid. With its help, the target core of the OS object identified by this ID is determined. This can be done on the core where the service was called, since the object ID information is statically configured. If the OS object resides in the local core, the service is processed like in the single-core implementation. Otherwise, a message is prepared including the source (local) core ID, OS object ID, and possibly additional service call parameters. Afterwards, `MessageSend` is called. The destination core is notified by an inter-core interrupt. The source core is then waiting for a reply containing the return status to finish the service call. Therefore, `MessageRecv` is called with the target core ID as `source` parameter and the service type ID of *ReturnStatus* as message tag.

On target core side, an interrupt service routine (ISR) is responsible for receiving and analysing messages. Due to the fact that the ISR does not know from which core or of which type the next message might be, it has to accept every message. Thus, `MessageRecv` is called with ANY_SOURCE and ANY_TAG parameter. After receiving a message, the ISR evaluates the service type ID and calls the corresponding OS service with the parameters encapsulated in the message. Then, the service checks whether certain conditions are met. At this point, the return status can be evaluated, since the service action cannot fail anymore, except for an abnormal OS service termination. The status is transmitted to the source core by calling `MessageSend`. Next, the service action is performed and a rescheduling takes place eventually, depending on the OS service. Otherwise, the task executed before interruption is resumed.

B. Approach 2: Lock Based

The lock based approach starts in the same way as the message based one. First, the passed OS object ID is checked for validity and possibly the service returns with an appropriate error. If the ID is valid, the source core enters a critical section by acquiring a lock for the OS runtime data of the target core. To guarantee fairness and real-time constraints, a ticket lock implementation was chosen. The source core executes the service check and depending on that possibly the service action on the OS data of the target core. If the OS service postulates a rescheduling, the source core checks, whether a task swapping has to be performed on the target core. If so, `MessageSend` is called with the target core as destination and service type ID of the OS service *Reschedule*. Afterwards, a software interrupt is triggered at the destination core to notify it immediately about the message. The lock is released and the service returns with a certain status. At target core side an interrupt service routine processes incoming messages. After evaluation of the service ID tag of the message, the rescheduling service on the target core is called.

3.2.5 Results and Conclusion

We tested both approaches on the basis of the OS service *ActivateTask* with a synthetic test benchmark. Preliminary results show that the lock based approach outperforms the message based one. At testing time, the memory access times of the simulator for core local as well as shared memory had the same delay times. Since the message based approach does not need the OS runtime data to be globally accessible, it probably will improve if the core local memory has shorter access times. This will be evaluated when scratchpad memory is connected to each core.

The first approach executes the service on the target core, interrupting the currently executing task in any case. In the second approach, the service call is executed on the invoking core and manipulates the target core's data structures residing in shared memory. The target core is only interrupted if a rescheduling needs to take place. The discussion on the timing behaviour shows the feasibility of both approaches from the point of view of a WCET analysis. Measurements performed on the parMERASA many-core simulator indicate an advantage of the lock based approach. It also has advantages regarding WCET analysability due to missing remote blocking times when no rescheduling is required.

3.3 Comparison of AUTOSAR Release 4.1 Specification with Factored Operating System

At the beginning of the parMERASA project, AUTOSAR Release 4.0 was the up-to-date specification. The tiny automotive RTE was designed following the distribution approach of the factored operating system (fos). Since the introduction of AUTOSAR Release 4.1, the multi-core support has moved towards the direction to the fos approach. Table 1 highlights the differences between these approaches.

Table 1: Comparison of AUTOSAR 4.1 with Factored Operating System

AUTOSAR 4.1 & Tiny Automotive RTE	Factored Operating System
Distribution functional clusters of BSW modules to arbitrary partitions	Distribution of services to servers
Replication of functional clusters of BSW modules to partitions using a master/satellite approach	Replication of server fleet
Service Call to satellite, forwarding to server	Service Call by message to server
Application and/or BSW core	Either application or server-core
No dynamic relocation of BSW modules	Dynamic selection of a suitable core from a fleet

The most obvious difference is the dynamic selection of cores in fos in contrast to the static placement in AUTOSAR. This originates of the real-time requirements of automotive systems, where no dynamic memory allocation is used. In the fos approach, system services, e.g. name server or file server, are replicated and distributed over a many-core processor to balance workload and to minimize communication. The main motivation for distribution of functional clusters of BSW modules in AUTOSAR is to support protection regions called partitions to guarantee freedom of interference between applications of different safety levels. In this case, BSW services or parts of them are also replicated over several cores. It can also have a positive impact on performance since the master core has not to be interrupted for each service call of other cores. Although fos is not targeting embedded systems, the main concept, the distribution of system services to servers to increase scalability, can be found similarly in AUTOSAR 4.1. In AUTOSAR, the distribution of functional clusters allows the realization of protection regions and hence to scale more applications onto one processor. So it is possible to integrate more applications into one compound electronic control unit (ECU). Otherwise, the applications would be distributed over several ECUs.

4 Tiny Avionic RTE

Deliverables D2.4 and D6.7 explain the parallel implementation of the avionics application presented with parallel execution patterns. In the referred deliverables produced in m24, we consider two implementations of the avionics applications – NoOS (based on a bare metal programming) and the tiny avionic RTE (based on the existing ARINC 653 standard). In m24 deliverables, we also list the API calls implemented in the tiny avionic RTE from the existing ARINC 653 and introduce the new API/functionalities to support multi-core execution.

Tiny avionic RTE improvements

In m30, we preserve the tiny avionic RTE API presented in m24 and we improve only part of it to support the newly introduced architectural augmentations such as the on-demand cache coherence protocol and the new memory map. In the new memory map where each core has a local memory space, we explicitly placed all A653 shared data structures in the shared memory space. Such data structures and A653 APIs are those related with intra-partition communication (reading and writing to buffers) and inter-partition communication (reading and writing to queues).

```
SHARED_VARIABLE(shu_C00) PARTITION_TYPE partitionTable[MAX_NUMBER_OF_PARTITIONS];
SHARED_VARIABLE(shu_C00) QUEUING_PORT_t queuingportTable[MAX_NUMBER_OF_QUEUING_PORTS];
SHARED_VARIABLE(shu_C00) BUFFER_t bufferTable[MAX_NUMBER_OF_BUFFERS];
SHARED_VARIABLE(shu_C00) EVENT_t eventTable [MAX_NUMBER_OF_EVENTS];
SHARED_VARIABLE(shu_C00) PROCESS_t processTable[MAX_NUMBER_OF_PROCESSES];
SHARED_VARIABLE(shu_C00) SEMAPHORE_t semaphoreTable[MAX_NUMBER_OF_SEMAPHORES];
SHARED_VARIABLE(shu_C00) ticketlock_t create_event_lock;
SHARED_VARIABLE(shu_C00) ticketlock_t set_event_lock [MAX_NUMBER_OF_EVENTS];
SHARED_VARIABLE(shu_C00) ticketlock_t wait_event_lock [MAX_NUMBER_OF_EVENTS];
SHARED_VARIABLE(shu_C00) ticketlock_t create_queue_lock = {0,0};
SHARED_VARIABLE(shu_C00) ticketlock_t send_queue_lock[MAX_NUMBER_OF_QUEUING_PORTS];
SHARED_VARIABLE(shu_C00) ticketlock_t receive_queue_lock[MAX_NUMBER_OF_QUEUING_PORTS];
SHARED_VARIABLE(shu_C00) ticketlock_t semaphore_lock = {0,0};
SHARED_VARIABLE(shu_C00) ticketlock_t process_lock = {0,0};
SHARED_VARIABLE(shu_C00) ticketlock_t create_buffer_lock = {0,0};
SHARED_VARIABLE(shu_C00) ticketlock_t send_buffer_lock[MAX_NUMBER_OF_BUFFERS];
SHARED_VARIABLE(shu_C00) ticketlock_t receive_buffer_lock[MAX_NUMBER_OF_BUFFERS];
```

Figure 5: Example of modified code to place data structures in the shared memory segment

In Figure 5, we demonstrate how to modify the definition of the tiny avionic RTE data-structures, so they are properly placed in the shared memory segment. The macro `SHARED_VARIABLE` places the corresponding variable in the memory section referenced by the macro parameter. In this example, the `shu_C00` section corresponds to an uncached shared memory section.

The tiny avionic RTE API delivered in m24 is capable to run in parallel on multiple cores and no further parallelization is envisioned.

5 Conclusion

The tasks described in T4.6 and T4.7 have been completed. For the next period the following topics will be investigated:

Tiny automotive RTE

The tiny automotive RTE will be adapted to support the execution of parallelized Runnables. So far, the execution of parallelized AUTOSAR Software Components and Tasks is enclosed by the tiny automotive RTE. Even the actual Specification of AUTOSAR Release 4.1, which has introduced some new features to exploit multi-core processors, does not cover a possibility to execute parallelized Runnables. The adaptions of the tiny automotive RTE go hand in hand with the analysis of Runnable parallelization for the automotive engine management application.

Tiny avionic RTE

The work on the tiny avionic RTE will support the avionic pilot studies based on the current tiny avionic RTE implementation, and continue to maintain the tiny avionic RTE implementation so that it is compatible with the HON applications and parMERASA platform.

List of Figures

List of References

[1] C. Bradatsch, F. Kluge und T. Ungerer, „A Cross-Domain System Architecture for Embedded Hard Real-Time Many-Core Systems," in *11th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC-13)*, Zhangjiajie, China, 2013.

[2] C. Aussaguès, D. Chabrol, V. David, D. Roux, N. Willey, A. Tournadre und M. Graniou, „PharOS, a Multicore OS Ready for Safety-Related Automotive Systems: Results and Future Prospects," in *5th International Congress on Embedded Real-Time Software and Systems (ERTS2)*, Toulouse, France, 2010.

[3] C. Bradatsch, F. Kluge und T. Ungerer, „Synchronous Execution of a Parallelised Interrupt Handler," in *20th IEEE Real-Time and Embedded Technology and Applications Sysmposium (RTAS) - to be published*, Berlin, Germany, 2014.

[4] C. Aussaguès und V. David, „A Method and a Technique to Model and Ensure Timeliness in Safety Critical Real-Time Systems," in *Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, Monterey, CA, 1998.

[5] AUTOSAR, „Specification of Operating System (Version 5.2.0)," 2013.

[6] OSEK/VDX, „Operating System (Version 2.2.3)," 2005.

[7] K. Kavi, R. Akl und A. Hurson, „Wiley Encyclopedia of Computer Science and Engineering," in *Wiley Encyclopedia of Computer Science and Engineering*, John Wiley & Sons, Inc., 2009, pp. 2369-2377.

[8] C. L. Liu und J. W. Layland, „Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM,* Bd. 20, Nr. 1, pp. 46-61, 1973.

[9] K. Gresser, „An Event Model for Deadline Verification of Hard Real-Time Systems," in *Proceedings of the fifth Euromicro Workshop on Real-Time Systems*, Oulu, Finland, 1993.

[10] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange und P. Sainrat, „Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, Dresden, Germany, 2012.