# MULTI SLOTH: An Efficient Multi-Core RTOS using Hardware-Based Scheduling*

Rainer Müller, Daniel Danner, Wolfgang Schröder-Preikschat, Daniel Lohmann
Friedrich–Alexander–Universität (FAU) Erlangen–Nürnberg, Germany
{raimue,danner,wosch,lohmann}@cs.fau.de

***Abstract*—Multi-core operating systems inherently face the problem of concurrent access to internal kernel state held in shared memory. Previous work on the SLOTH real-time kernel proposed to offload the scheduling decisions to the interrupt hardware, thus removing the need for a software scheduler; no state has to be managed in software. While our existing design covers single-core platforms only, we now present MULTI SLOTH, a multi-core AUTOSAR OS implementation. In this paper, we show that our hardware-centric approach enables us to easily make the transition to multi-core platforms without the need for explicitly synchronizing kernel data. Even in the case of cross-core interactions, MULTI SLOTH keeps the unique SLOTH properties of strict priority obedience and complete prevention of rate-monotonic priority inversions.**

**AUTOSAR OS mandates only unordered spinlocks, which do not guarantee predictable timing. We show the advantages of the MULTI SLOTH design by additionally providing a wait-free and efficient implementation of the priority-aware Multiprocessor Priority Ceiling Protocol (MPCP). On our reference platform, we achieve overheads as low as 1.1 µs for acquiring a globally shared resource using the MPCP and round-trip times of 1.4 µs for cross-core task activations.**

## I. INTRODUCTION

The ongoing trend to multi-core systems is spreading in the embedded systems at the moment, which also leads to fundamental changes in system software for the management of tasks and the provisioning of synchronized access to shared resources. In the automotive industry, for example, the widely used AUTOSAR OS interface [1] only recently added support for multi-core systems. As AUTOSAR builds their specification on the older OSEK OS specification [18], the abstractions for control flow and resource management are still mainly targeting a single-core system.

In previous work on the SLOTH real-time operating system [12], [13], we have shown that the interfaces of the single-core OSEK OS specification can be implemented by using interrupt controllers in off-the-shelf hardware for scheduling and dispatching. By mapping all control flows in the system to interrupts, efficient management leads to a high performance with only little support in software necessary. This approach leads to a concise kernel implementation with a low memory footprint. Furthermore, it allows organizing both tasks and interrupt service routines (ISRs) in the same unified priority space, taming the problem of *rate-monotonic priority inversion* [8] by construction. In traditional software-scheduled systems, in contrast, interrupt handlers with semantically low priorities can defer the execution of any task despite its priority.

In this paper, we present MULTI SLOTH as an adoption of the SLOTH concepts to multi-core platforms following the AUTOSAR OS multi-core specification. The AUTOSAR system model for multi-core real-time systems uses partitioned fixed-priority scheduling, where each task is statically allocated to a core and each core is scheduled using a single-core policy. Our focus is providing the system interface of the operating system with abstractions for task and resource management. By identifying and combining the building blocks of the original SLOTH design, we show how little effort is required for the transition of our implementation to multi-core platforms—in particular, to the Infineon AURIX [14], which is one of the major target platforms for upcoming applications in the automotive industry. Additionally, to explore the possibilities offered by our hardware-centric design, we present an efficient implementation of a priority-aware synchronization protocol.

For the management of resources, OSEK mandates a stack-based priority ceiling protocol (PCP) that builds on temporarily raising a task's execution priority while inside the critical section. On a single-core processor this approach effectively prevents unbounded priority inversions. With the addition of multi-core in the AUTOSAR OS specification, spinlocks have been added to perform inter-core synchronization. However, no semantics for these spinlocks is specified in AUTOSAR OS. The use of simple unordered spinlocks allows priority inversions, for example when a low-priority task is preempted in a critical section by an independent mid-priority task, effectively delaying the execution of a high-priority task on another core that is waiting for access to the shared resource locked by the low-priority task. AUTOSAR recommends to make critical sections completely non-preemptable by suspending interrupts before acquiring a lock, although the spinning itself could still be implemented preemptable. An analysis of different spinlock types in the AUTOSAR context can be found in [23].

Synchronization protocols have been developed to provide predictable synchronization in the multi-core scenario for real-time systems. In [19], for example, Rajkumar extended the priority ceiling protocol for use with multiple processors defining the Multiprocessor Priority Ceiling Protocol (MPCP).

Due to the concise kernel design of MULTI SLOTH, we are able to use its building blocks to implement the MPCP efficiently and our implementation of the acquire and release operations is wait-free. Moreover, we are not only able to reduce the overall system overhead due to asynchronous scheduling

*This work was partly supported by the German Research Foundation (DFG) under grants no. SCHR 603/8-1 and by the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89), Project C1.

**Appeared in:** Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS '14), pages 189–198, IEEE Computer Society, 2014, ISBN 978-1-4799-5798-9

decisions in the interrupt controller hardware, but are also able to remove priority inversions that would occur when using a software scheduler.

### A. Contributions

In this paper, we present MULTI SLOTH as an extension of our hardware-centric SLOTH RTOS to multi-core platforms with the following contributions:

- We show that MULTI SLOTH keeps the beneficial properties of SLOTH even with cross-core interactions, while providing fast and bounded system service execution without rate-monotonic priority inversions.
- Based on these non-functional system properties, we show on the example of MPCP that it is possible to implement advanced synchronization protocols for multi-core systems in an efficient and deterministic manner.
- In our reference implementation on the Infineon AURIX we show the round-trip time of a cross-core task activation to be as low as 171 cycles and a global resource acquisition with MPCP takes only 112 cycles.

### B. Organization of this Paper

First, we present the building blocks of the SLOTH real-time operating system in Section II as background information. In Section III, we analyze the challenges of the transition to multi-core following the AUTOSAR system model. With regard to these challenges, in Section IV and V we present MULTI SLOTH adapting the existing building blocks of SLOTH for multi-core on the Infineon AURIX as reference platform. In Section VI, we discuss the implications of the AUTOSAR OS multi-core synchronization interface and explain the MPCP as an alternative. Subsequently, Section VII details the design and implementation of the MPCP in MULTI SLOTH. The whole system is evaluated in Section VIII, the results are discussed in Section IX and Section X discusses related work. Finally, Section XI concludes the paper with a summary.

## II. THE BUILDING BLOCKS OF SLOTH

First we want to provide an overview of the concepts and mechanisms used in the original single-core design of the SLOTH real-time operating system.

One main design property of SLOTH is that it abolishes the traditional division of control-flow abstractions into threads and interrupt handlers. Instead, all threads (or *tasks* in OSEK terminology) are implemented as interrupts, moving their management off the CPU onto the interrupt subsystem. By mapping each task to a separate interrupt and assigning an IRQ priority according to the application configuration, all scheduling decisions are made autonomously as part of the interrupt arbitration. When a task is activated by triggering its assigned IRQ source (see, for example, ① in Figure 1), the interrupt controller hardware (see ②) determines the highest-priority pending interrupt source and compares it to the current execution priority on the CPU (see ③). If it is lower, the CPU is interrupted, effectively preempting the currently running task. In case of a low-priority task activation, however, the CPU

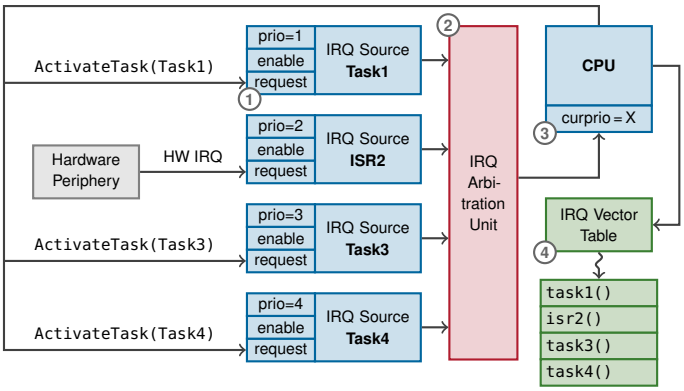


Fig. 1: Design of a SLOTH operating system for a single-core platform, implementing all control flow types as interrupts, leaving scheduling decisions to the interrupt hardware and dispatching both tasks and ISRs as IRQ handlers.

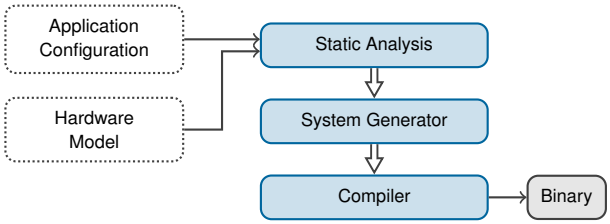


Fig. 2: Outline of the generative framework used for building applications in SLOTH.

remains uninterrupted as the pending interrupt priority is lower than the execution priority. For the dispatching of the interrupt handler, the CPU will look it up in the interrupt vector table (see ④), from where the corresponding task function will be called in a SLOTH system. This approach not only allows for a concise implementation that reduces kernel overhead and latency, but also avoids the problem of rate-monotonic priority inversion that comes with a priority space that is split into tasks and ISRs. These non-functional properties allow the implementation of real-time systems using the SLOTH concept on many platforms, for example on our single-core reference platform, the Infineon TriCore.

The interface that SLOTH offers to the user application follows the OSEK OS standard as well as its compatible successor AUTOSAR OS, which is a statically configured system using fixed-priority tasks.

From an architectural point of view, SLOTH is built on a generative framework that is composed of three main components, as shown in Figure 2. Starting with a given application and its configuration, a static analysis step first extracts the requirements of the application to obtain a mapping of operating system abstractions used by the application to specific hardware resources, such as interrupt sources, timer units and interrupt priority levels. By combining this mapping and the hardware model information of the selected target platform, the system generator then creates the actual kernel code, which is specifically tailored to both the application and

the target platform. In the final step, compiling the generated system code and the user-provided application code as a single program allows the compiler to extensively optimize towards the concrete behavior of the application by, for instance, inlining system service calls, propagating constant parameters, and eliminating dead code.

The system design of SLOTH consists of four fundamental building blocks, which need to be implemented according to the properties of the underlying platform. These blocks are (a) the task activation via interrupts, (b) dispatching tasks as interrupt handlers, (c) blocking tasks, and (d) raising and lowering task priorities for synchronization.

*a) Task Activation:* With tasks implemented as interrupts, synchronous task activations essentially consist in triggering the associated interrupt from software. Usually, this is as simple as writing to a certain memory-mapped register or executing a special machine instruction. Naturally, the execution time of the task activation itself is bounded as no dynamic decisions are required.

*b) Task Dispatching:* OSEK differentiates between two types of tasks, of which only *extended* tasks have the ability to block in order to wait for events signaled by other tasks, whereas *basic* tasks are constrained to run-to-completion semantics. As a consequence, the IRQ handlers registered for *basic* tasks merely have to save register contents not implicitly saved by the hardware, followed by jumping to the actual task function provided by the user application. Here, the nested, stack-sharing nature of interrupts handling matches exactly the desired task model.

*c) Task Blocking:* Extended tasks may block to wait for an event, which means they leave their control flow context, allowing other lower-priority tasks to run. Therefore, they require a dedicated stack and a more complex IRQ handler, which distinguishes between dispatching a freshly activated task or resuming an existing context. The particular mechanism for blocking a task is mostly up to hardware specifics. Generally, the current context is stored, the task's interrupt source is disabled and then the task yields the CPU by leaving the interrupt context. The reverse procedure to unblock a task essentially only needs to re-enable and trigger the interrupt source, such that it is considered in the IRQ arbitration once again. For example on the Infineon TriCore, blocking a task is implemented by lowering the current execution priority to zero, which leads to immediate preemption by other tasks as their priority is higher. The IRQ handler for the preempting task is then in charge of preserving the previous (now blocked) task's state by performing the switch to its own dedicated stack. When unblocking the task later on, a switch to the preserved stack facilitates the full restoration of its state at the point blocking. For a more detailed insight into task blocking in SLOTH, see [13].

*d) Resource Management:* OSEK also prescribes resource management in the form of a stack-based priority ceiling protocol. The mechanism central to this is the raising and lowering of a task's priority while it is executing. In SLOTH implementations, this is done by manipulation of the current execution priority of the CPU directly, which inhibits the preemption by interrupts of equal or lower priority.

## III. EMBEDDED MULTI-CORE OS CHALLENGES

In our adoption to a multi-core architecture, we closely follow the system model prescribed by AUTOSAR OS, which specifies interfaces for event-triggered embedded kernels that use partitioned fixed-priority scheduling. With regards to multi-core platforms, AUTOSAR OS considers tasks to be statically assigned to a certain core and to not migrate at run time. The execution model within each core matches that of single-core AUTOSAR systems. These sets of tasks per core are, however, not entirely isolated from each other but may invoke task activations across core boundaries and can also globally share events to wait for.

In the transition of an existing embedded operating system implementation to shared-memory multi-core platforms it is first necessary to identify all situations that require concurrent access to kernel data structures. Then, proper internal synchronization mechanism need to be introduced internally, in order to keep each core's view on the system state consistent. In the case of the AUTOSAR OS model, such need for synchronization arises in all cross-core system services, such as activating remote tasks.

Considering a traditional kernel implementation, which schedules and dispatches in software, this is usually realized by placing the relevant information in a shared memory location. A high-priority inter-processor interrupt is then sent to the target core to signal the incoming event. Regardless of the priority of the currently executing control flow, the remote core will get arbitrarily interrupted for performing duties on behalf of a task with a lower priority, constituting a case of rate-monotonic priority inversion.

One of the goals of SLOTH, however, has always been to prevent such behaviour by design. In order to retain the beneficial properties of SLOTH on the multi-core domain, we need to devise how to perform kernel signaling across cores without impeding the execution of higher-priority tasks.

## IV. DESIGN OF MULTI SLOTH

Due to the simplicity of the SLOTH design, extending the system to multi-core can be achieved by combining the existing building blocks from the single-core system in new ways.

### A. Extending the Building Blocks to Multi-Core

Only a few of the building blocks of SLOTH introduced in Section II need changes to be used in a multi-core system, while others can be used unmodified. Most noticeable, MULTI SLOTH needs to support task activations across multiple cores. For this, the task activation has to be extended to send interrupt requests to another core, where it will participate in the arbitration process. This will either require write access to the interrupt sources of other cores with a memory-mapped interrupt controller, or a special instruction to send an inter-processor interrupt. In both cases, this does require changes in the building block for task activation, which still boils down to

a single instruction setting the pending bit of the corresponding interrupt source. No knowledge about the target core handling the activation is required on the core executing the system service, seeing that the arbitration in the interrupt controller will determine the highest pending interrupt request individually for each core.

In contrast, the handling of waiting for events and their signaling in extended tasks needs to be adapted for use in a multi-core system. When a task in AUTOSAR waits for a specific event, it blocks and yields the CPU in order to let lower-priority tasks run. It is then later unblocked by a signal when the event is set by another task. The mechanisms for blocking and unblocking of a task will be the same in MULTI SLOTH as in the single-core SLOTH, as they merely involve a change of the current task priority and another trigger of the interrupt source for unblocking. As the events can be signaled from any core, variables indicating events set and waited for by each task need to reside in shared memory. However, due to possible concurrent access to the data structures, additional synchronization is required. When multiple control flows set an event for a task in parallel, it must only lead to a single unblock of this task. Otherwise, starting the unblock mechanism although the blocked task already continued due to another event signal would be mistaken as a spurious task activation as both are performed with the same interrupt request. Additionally, the decision whether a task needs to block or the event was signaled before needs to be synchronized as well to counter the lost wakeup problem.

For the handling of resources *locally* bound to one core only, no changes on the existing building blocks implementing the OSEK priority ceiling protocol are required. However, for *global* resources shared across multiple cores, new functionality is mandatory in a multi-core system. The AUTOSAR OS standard only prescribes spinlocks for this, while leaving the exact semantics unspecified. For the sake of completeness, we implemented spinlocks as mandated by AUTOSAR, but we will get back to this topic and its implications in Section VI below.

### B. Hardware Requirements

For this approach with a interrupt-based multi-core system, we have requirements on the hardware architecture that the MULTI SLOTH is implemented for. As in the existing SLOTH kernel [12], [13], the hardware needs to allow interrupt triggering from software either by a memory-mapped register or a special machine instruction. In MULTI SLOTH, however, it is also required to send interrupts to remote cores as inter-processor interrupts. In a similar way, the number of available interrupt sources and priorities per core needs to be at least as high as the number of tasks and interrupts to be handled on this core, as each task and interrupt is assigned to a dedicated interrupt source and priority. Furthermore, for MULTI SLOTH it is required to have at least a small shared memory region where all cores are able to access the event variables. Of course, additional local memory per core is also supported when the data stored there must not be accessed by other cores. These requirements are fulfilled by most embedded microcontroller platforms, like the ARM Cortex-A series, the Freescale MPC55xx, or—our reference platform used for this paper—the Infineon AURIX.

## V. IMPLEMENTATION OF MULTI SLOTH

The reference implementation of MULTI SLOTH targets the Infineon AURIX platform, a next-generation embedded multi-core platform based on the TriCore architecture, which is widely used in the automotive industry. In this section, we present the relevant features and properties of this platform, followed by a presentation of the implementation details of MULTI SLOTH following the AUTOSAR OS specification.

### A. The Infineon AURIX Multi-Core Platform

The Infineon AURIX [14] offers up to three cores in a shared-memory architecture, with each core sharing much similarity and compatibility with the predecessor single-core TriCore platforms.

The interrupt subsystem provides up to 1023 interrupt sources called *service request nodes* (SRNs), most of which are connected to hardware periphery. A subset of SRNs serve as general purpose nodes that can solely be triggered from software. The SRNs can be individually assigned an 8-bit priority vector and target processor core to deliver the interrupt request to. Each core provides its own *interrupt control unit* (ICU) handling the arbitration of incoming interrupt requests. If the determined highest priority of all interrupts pending at a given core surpasses the current execution priority, the core processes the request by executing the corresponding handler function. Since both types of SRNs—periphery and general purpose—offer the same features regarding software-triggered interrupts, any SRNs connected to unused hardware units can be repurposed for MULTI SLOTH. The upper limit of tasks configured in an MULTI SLOTH system on this platform is therefore only limited by the maximum number of 255 interrupt levels. Caching is optional on the AURIX, but it does not provide cache coherence. Therefore, we consider caches to be disabled in our design as well as for the evaluation.

In MULTI SLOTH, the configuration interface exposed to the application designer allows them assigning each task to a specific core, which is then translated to a suitable SRN configuration. Figure 3 illustrates the AURIX platform architecture and includes an example application setup consisting of three tasks and one ISR triggered by some peripheral hardware. Each SRN represents one of these control flows and—through its `core` field—is assigned to the ICU of a given core. In turn, each core has an individually assembled IRQ vector table, mapping incoming IRQs onto the proper task or ISR user code.

### B. Implementing MULTI SLOTH Building Blocks

For each building block introduced in Section II, we outline in the following, which changes were necessary for transition of our implementation to the Infineon AURIX.

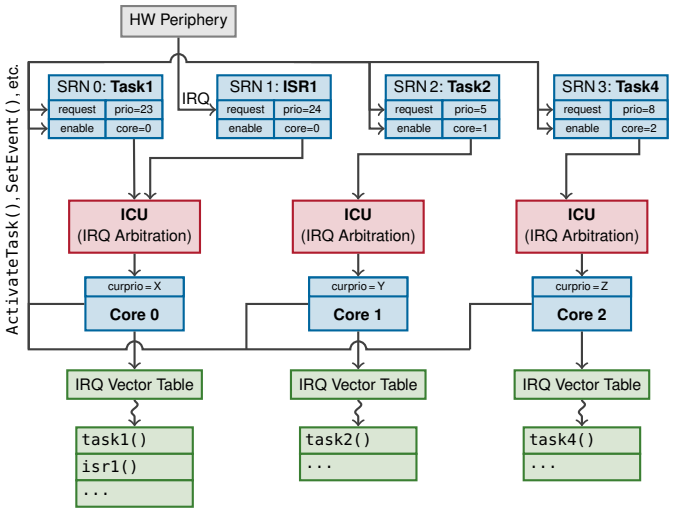


Fig. 3: Design of a MULTI SLOTH system on the Infineon AURIX multi-core platform. The three cores each perform individual IRQ arbitration for source assigned to them. Both local and remote task activations are performed by triggering interrupts from software via shared, memory-mapped registers of the interrupt hardware.

*a) Task Activation:* Given that all SRNs are accessible globally by any core, the implementation of synchronous task activations on the AURIX does not need to pay attention to whether it constitutes a local or cross-core activation. In either case, tasks are activated by triggering their assigned SRN, which means setting the `SETR` bit of the memory-mapped control register of this SRN. Since the AURIX provides dedicated instructions that allow to set single bits in memory atomically, no additional synchronization measures are necessary to be able to deal with concurrent task activations on multiple cores.

*b) Task Dispatching:* Since each core on the AURIX performs its IRQ arbitration individually and resolves incoming interrupts based on a core-specific IRQ vector table, the task dispatch mechanism is not affected by the fact that there are multiple cores.

*c) Task Blocking:* Similarly, the task blocking procedure is confined to activity on the local core and the local interrupt controller. Unblocking a task, however, can be effected locally as well as remotely from any other core. Since the only modifying operation done for unblocking a task is enabling the corresponding IRQ source, concurrent unblocking does not need to be synchronized per se. Nevertheless, when using these building blocks, attention needs to be paid to whether unblocking a remote task which is in the process of blocking itself at the same time might entail a lost wakeup for this task. As we will show in our design for the MPCP in MULTI SLOTH, this issue can be tackled by properly ordering each step leading to blocking or unblocking a task as to prevent both happening concurrently.

*d) Resource Management:* The stack-based priority ceiling protocol prescribed for single-core AUTOSAR systems is present in the multi-core case as well, but strictly confined and partitioned onto each core. For MULTI SLOTH, this means that besides adding a configuration directive for specifying which core each local resource is assigned to, no extras measures are needed for this building block to work on a multi-core platform.

*e) AUTOSAR Spinlocks:* Additionally to adopting all existing functionality for multi-core platforms, MULTI SLOTH offers a spinlock interface as prescribed by AUTOSAR OS. On the AURIX, this is implemented by using the `swap` instruction for atomically replacing a memory word with another.

## VI. SYNCHRONIZATION IN REAL-TIME MULTI-CORE SYSTEMS

MULTI SLOTH as presented so far supports task management on an embedded multi-core system following the AUTOSAR OS specification.

However, another challenge in a multi-core system is the management of logical and physical resources. For multiple tasks accessing the same indivisible resource in parallel, the operating system should provide a system interface for the application programmer to efficiently ensure mutual exclusion. With this abstraction, to maximize utilization of the computing resources, tasks may block and yield the CPU while waiting for a resource. In particular for a real-time system, the duration of this blocking has to be bounded in order to be able to give guarantees on the timing. For single-core systems, these problems have been solved in OSEK with a *priority ceiling protocol* [2], [21]. Upon acquiring a resource, this protocol mandates to raise the execution priority to the highest priority of all tasks accessing this resource, preventing preemption by any of these tasks. On release of the resource, the execution priority is lowered to the previous value. Using this approach, the acquisition of a resource always succeeds as otherwise the task holding the resource would prevent the dispatch of other tasks accessing the same resource. Also, no blocking is required for the resource management in OSEK and therefore, it can be applied to both *basic* and *extended* tasks.

In a multi-core system with fixed task partitioning, resources can be divided into *local* resources accessed by tasks on a single core, and *global* resources accessed across multiple cores. As local resources do not have interactions with other cores, they can be managed with the same single-core priority ceiling protocol. For the synchronization of global resources additional problems arise due to the parallel execution of multiple control flows. When a task wants to access a global resource that is held by another task on another core, the latter must not be preempted in the critical section by other independent tasks with a higher priority, as that would lead to remote blocking.

In the AUTOSAR multi-core standard only spinlocks are prescribed for the synchronization of resource access between application tasks. Moreover, no specific semantic is mandated for the spinlocks in AUTOSAR; the execution of multiple tasks trying to acquire the same lock is unordered. When using these simple spinlocks, an independent mid-priority task preempting a low-priority task, which holds a spinlock, can inhibit progress

of a remote high-priority task waiting on the same spinlock. This constitutes a priority inversion, which must not occur in a real-time system. Furthermore, deadlocks are possible when a task tries to acquire a spinlock held by a preempted task on the same core. The AUTOSAR specification suggest to avoid these issues by making critical sections non-preemptable, which is also assumed by prior work analyzing spin locks. However, the order of execution when multiple tasks compete for a global resource is not defined and does not take task priorities into account.

Thus, for predictable resource sharing, synchronization protocols beyond the AUTOSAR specification have to be explored. Sophisticated priority-aware multi-core synchronization protocols have been developed for many years, but none were considered for the AUTOSAR specification. In particular, the Multi-Core Priority Ceiling Protocol (MPCP) [20], [19] has been proposed for use in AUTOSAR before [15].

In this paper, we therefore integrate the MPCP as an example for a priority-aware synchronization protocol into MULTI SLOTH. While originally designed for shared-memory multiprocessor systems, the MPCP is applicable to embedded multi-core systems as they are similar to these systems with the same properties.

In MULTI SLOTH, we want to overcome synchronization challenges for multi-core systems and not only follow the AUTOSAR multi-core OS specification. An efficient, priority-aware synchronization mechanism for multi-core is required in order to implement real-time applications that can use hardware platforms to their full extent.

## VII. MPCP IN MULTI SLOTH

While MULTI SLOTH provides a full implementation of the AUTOSAR OS multi-core standard including unordered spinlocks, we added MPCP as an example of a priority-aware synchronization protocol.

### A. Multi-Core Synchronization using MPCP

MPCP classifies critical sections as *local* if the resource access only needs to be coordinated between tasks on the same core and the usual single-core PCP should be used. If a resource is used across tasks on multiple cores, it defines a *global* critical section. When a task tries to enter a global critical section previously locked on another core it is suspended, allowing lower-priority tasks to run. All global critical sections are executed at ceiling priorities that are above the normal priorities of all tasks.

For the sake of self-containment, we explain the application of the MPCP to our system model in the AUTOSAR context [15]:

- All tasks use their assigned priority unless in a critical section.
- Access to local resources will be handled according to the single-core stack-based priority ceiling protocol.
- Each global resource has a ceiling priority above all priorities assigned to tasks on any core, such that this global ceiling priority is defined as $\pi_{max} + \pi_{task}$ where $\pi_{max}$ is the highest priority assigned to any task on any core and $\pi_{task}$ is the highest priority of all tasks accessing this resource.
- Any task holding a global resource executes the global critical section at the ceiling priority of this global resource.
- On resuming execution after unblocking, a task entering a global critical section at its ceiling priority can preempt other global critical sections with a lower ceiling priority.
- When a task requests a global resource, it will be granted atomically and cannot be held by any other task at the same time.
- When access to a global resource cannot be granted, the requesting task blocks and waits for a signal indicating the acquisition of the global resource.
- When a task leaves a global critical section, the highest-priority task waiting for this global resource will be signaled and is allowed to continue at the ceiling priority of the global resource. The signaled task is then holding the corresponding global resource. When no task is waiting for the global resource, it is just released.

### B. Design of MPCP in MULTI SLOTH

The key components of the MPCP protocol are changing the current execution priority, blocking the current task, and signaling another task on a remote core. Traditionally, these are implemented with a software scheduler that manages tasks by maintaining priority-ordered queues on each core. In MULTI SLOTH, however, we can use our building blocks to implement the MPCP protocol.

In our system model following AUTOSAR, tasks using MPCP resources need to be able to block, which means they need to be *extended* tasks. Taking a global resource needs to be properly synchronized in order to ensure only one task can enter the guarded section at a time.

In our design process it became clear that the MPCP needs to be introduced as a new interface in the OS level and cannot be built on top of an existing AUTOSAR system. Global critical sections need to be entered at the ceiling priority even after a task blocked previously. This change in execution priority cannot be reproduced with the existing AUTOSAR OS system services.

The steps to acquire and release an MPCP resource in MULTI SLOTH shown in Figure 4 are as follows:

*a)* `GetMPCPResource()`*:* At first, for synchronizing changes made to the interrupt hardware configuration on the local core, the system services must be non-preemptible, which is achieved by locally disabling interrupts on the executing core in Ⓐ. Before deciding whether the task needs to be blocked or not, two steps are performed early on, which semantically are part of the *Block Task* step later on. First, the IRQ source of the calling task is disabled (see Ⓑ) in order to prevent a lost-wakeup situation that would arise if a concurrent call of `ReleaseMPCPRelease()` was unblocking this task at the same time it gets blocked. Disabling the source early on ensures that the wakeup—which works by enabling

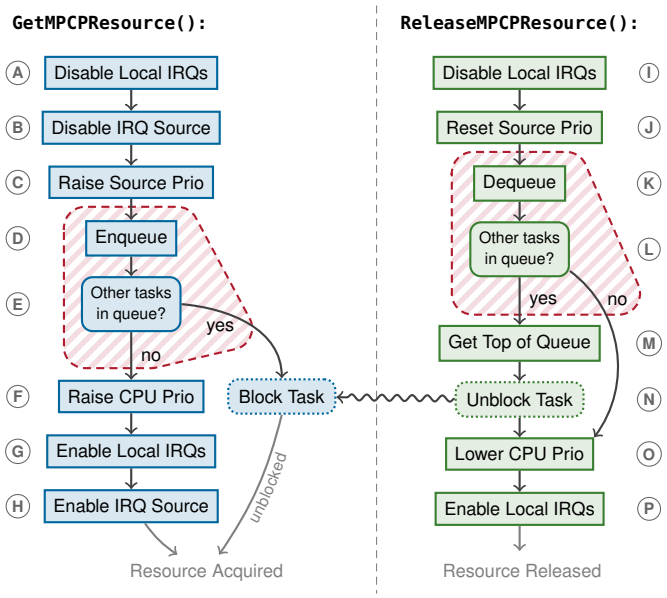


Fig. 4: Flow charts illustrating the procedures for acquiring and releasing MPCP resources in MULTI SLOTH. The striped regions indicate critical sections that must execute atomically.

the IRQ source of the affected task—will not get lost in this situation. Similarly, by reconfiguring the source to raise its priority level to the ceiling priority of the requested MPCP resource in Ⓒ, it is guaranteed that the task will resume at this ceiling priority, even if it gets unblocked immediately in the case of concurrency. In the next step Ⓓ, the task is inserted into an ordered queue that is specific to each MPCP resource and used to keep track of the tasks that compete for acquisition of this MPCP resource. After enqueueing the task, the current state of the queue determines in Ⓔ whether the resource is available or the task needs to block. For proper synchronization of multiple tasks trying to acquire the resource or another task releasing the same resource concurrently, the enqueue and check operation have to be implemented to execute atomically (indicated by the hatched areas in Figure 4). If other tasks were in the queue already, meaning that the resource is currently being held by another task, the calling task blocks to wait until the resource lock is released. In case the queue was empty when enqueueing the task, the resource is immediately acquired by raising the current execution priority to the ceiling priority in Ⓕ. The procedure then concludes by re-enabling interrupts on the current core in Ⓖ and the IRQ source of the current task in Ⓗ.

*b) `ReleaseMPCPResource()`:* When releasing an MPCP resource, local interrupts are first disabled for synchronization as well (see Ⓘ), then the priority of the corresponding interrupt source is reset back to the normal priority of the calling task in Ⓙ. In order to release the resource and pass it on to the next task, the calling task removes itself from the queue in Ⓚ and checks if other tasks are still waiting to acquire the resource (see Ⓛ). If so, the resource is handed over to the highest priority task in the queue (determined in Ⓜ) by unblocking it in Ⓝ. Again, the dequeue step and the test for other waiting tasks must be safe from concurrent queue operations on the same MPCP resource, and therefore be implemented atomically. After possibly unblocking the next task, the current execution priority is eventually lowered again in Ⓞ and local interrupts are re-enabled in Ⓟ. In case the task that was unblocked in Ⓝ executes on the same core the resource was released on, the calling task would be preempted promptly after re-enabling local interrupts in Ⓟ.

Ensuring the atomicity of queue operations in our design depends on the capabilities of the target hardware platform. In the following section, we will present that this can be implemented using atomic hardware instructions without resorting to additional lock variables for the queues.

### C. Wait-Free Priority Queue for MPCP

As laid out above, an implementation of our MPCP design in MULTI SLOTH needs to provide a priority queue that offers synchronized operations for either enqueueing or dequeueing an item and observing the queued items. These operations must work without concurrent queue manipulations interfering between both steps. A straight-forward implementation of this would be to make these sections non-preemptive and employing an additional lock variable for mutual exclusion.

For our implementation on the AURIX platform, however, we opted for a solution using hardware instructions for atomic memory access. This allows us to realize the aforementioned critical sections as single queue operations, which atomically perform both the requested operation and return the queue state.

To this end, we use a 32-bit word to represent the queue, in which each bit designates the unique priority of a task within the configured set of tasks that access the particular MPCP resource. The slots in this bit mask are ordered according to the priority in such a way that a simple `clz` (count leading zeros) instruction provides the means for determining the highest priority in the queue.

The atomicity of queue operations is then ensured by using the `swapmsk` instruction available on the AURIX. This instruction is similar to a regular atomic swap, except that the swapping affects only certain bits in the memory location, depending on a mask given as an operand. By specifying a mask with only a single bit set, which corresponds to a certain task, our implementation is able to enqueue or dequeue a task *and* acquire the previous queue state in a single, atomic operation. Ultimately, implementing the critical section indicated in Figure 4 resolves to a single `swapmsk` instruction atomically performing both the enqueue (or dequeue) step and the retrieval of the queue state.

As a consequence, the execution of MPCP operations in MULTI SLOTH is not influenced by concurrent invocations on the same resource and is guaranteed to progress on a per-thread basis. Therefore, acquiring and releasing MPCP resources in MULTI SLOTH is wait-free.

### D. Limitations

Each MPCP resource is organized with a queue that is represented by a 32-bit word, which is the largest data type on which atomic operations can be performed on the Infineon AURIX. While the whole system can use all 255 different priority levels available on this platform, a single MPCP resource can only be shared by up to 32 distinct tasks due to the dependency on the atomic hardware instruction.

In case an application needs more than 32 tasks sharing the same global resource, a possible fallback is to arbitrate accesses from tasks on the same core with a local resource first. A single high-priority task on this core will then participate in the synchronization using the global MPCP resource.

## VIII. Evaluation

In order to assess the performance of MULTI SLOTH, we set up several scenarios with test applications and carried out microbenchmarks measuring the execution times of each system service. The measurements were obtained using a Lauterbach hardware debugging and tracing unit, by instrumenting internal benchmark counters and averaging the length of the examined sections over 10,000 iterations each. Due to the constant complexity of the measured code, no standard deviations discernible from the few cycles of measurement accuracy were observed.

We group our scenarios into three separate parts. The first two reflect the difference in run-time overhead between a system configured to support only basic run-to-completion tasks and one to allow the blocking of tasks. The third group considers an application that makes use of MPCP resources and aims to cover the various transitions that can take place in such a system.

Table I shows the results obtained for a basic system. In cases where the given transition includes the dispatching of another task, the measurement includes this task's IRQ handler up to the first instruction of the user code. Since the hardware limitations in the evaluation setup only allow us to measure one processor core at a time, we assess the behavior of cross-core task activations by performing a round-trip back to the initial core. The resulting 135 cycles therefore cover the local `ActivateTask()` invocation, then a remote task dispatch followed by another task activation and a local dispatch in turn. The transitions measured for task activation, chaining, termination as well as acquiring and releasing resources take place locally on one core.

For the extended system, the test scenario is similarly constructed but more extensive. Firstly, those transitions that execute only on a single core have additional variants reflecting whether or not they include switching between the dedicated stacks of extended tasks. Secondly, the added support for task blocking introduces `WaitEvent`, `SetEvent` and `ClearEvent` as additionally available system services to be measured. The figures given in Table II reveal that the added costs for handling blocked tasks amount to 28-103 cycles, depending on the type of system service and whether a stack switch is in order. Notably, benchmarks that do not involve dispatching another task yield about the same run-time overhead as they do in a basic system (see Table I).

| Transition | Cycles |
|---|---|
| `ActivateTask` w/o dispatch | 65 |
| `ActivateTask` w/ dispatch | 87 |
| `ChainTask` w/ dispatch | 97 |
| `GetResource` | 36 |
| `ReleaseResource` w/o dispatch | 19 |
| `ReleaseResource` w/ dispatch | 41 |
| `TerminateTask` w/ dispatch | 20 |
| `ActivateTask()` cross-core round-trip | 135 |

TABLE I: Evaluation results for a **basic** MULTI SLOTH system on the Infineon AURIX

| Transition | Dispatch | Stack Switch | Cycles |
|---|---|---|---|
| `ActivateTask()` | w/ dispatch | w/o stack switch | 107 |
| `ActivateTask()` | w/o dispatch | - | 65 |
| `ActivateTask()` | w/ dispatch | w/ stack switch | 168 |
| `ChainTask()` | w/ dispatch | w/o stack switch | 125 |
| `TerminateTask()` | w/ dispatch | w/o stack switch | 36 |
| `TerminateTask()` | w/ dispatch | w/ stack switch | 121 |
| `ClearEvent()` | - | - | 17 |
| `SetEvent()` | w/ dispatch | w/ stack switch | 215 |
| `WaitEvent()` | w/ dispatch | w/ stack switch | 234 |
| `GetResource()` | - | - | 38 |
| `ReleaseResource()` | w/o dispatch | - | 19 |
| `ReleaseResource()` | w/ dispatch | w/ stack switch | 122 |
| `ActivateTask()` cross-core round-trip | | w/o stack switches | 171 |
| `ActivateTask()` cross-core round-trip | | w/ stack switches | 299 |

TABLE II: Evaluation results for an **extended** MULTI SLOTH system on the Infineon AURIX

The performance of the MPCP implementation on the AURIX platform was measured in a separate scenario covering the various combinations of acquiring and releasing resources, which either does or does not lead to blocking or unblocking of a task, which in turn can take place locally or remotely. The results achieved here are provided in Table III[1]. Again, in transitions that entail any remote action, such as, for instance, when an MPCP resource is released and handed over to a task on a remote core, the measurement of the transition considers only the *local* costs of performing the unblock operation and concludes when the `ReleaseMPCPResource()` call returns to the user code. Keep in mind, however, that the remote core will not be interrupted at all unless the priority of the unlocked task is in fact high enough for preemption. Overall, the run-time overhead for MPCP operations ranges between 130 and 350 cycles, equaling 1.3 µs and 3.5 µs at 100 MHz.

## IX. Discussion

MULTI SLOTH adapts the SLOTH system design of using the interrupt hardware for scheduling and dispatching to a multi-core system. Our concise kernel following the AUTOSAR

[1] A comparison to unordered spinlocks is omitted here, since their implementation on this platform boils down to merely a swap instruction followed by a branch. The resulting overheads of about 5 to 8 cycles (depending on the branch prediction) are too close to the measurement precision to be meaningful.

| Transition | Cycles |
|---|---|
| `GetMPCPResource()` w/ blocking | 217 |
| `GetMPCPResource()` w/o blocking | 112 |
| `ReleaseMPCPResource()` w/ local dispatch | 360 |
| `ReleaseMPCPResource()` w/o dispatch | 134 |
| `ReleaseMPCPResource()` w/ remote unblock | 183 |
| `ReleaseMPCPResource()` w/ local unblock and dispatch | 311 |
| `ReleaseMPCPResource()` w/o unblock w/ dispatch | 231 |

TABLE III: Run-time overhead of MPCP operations in MULTI SLOTH measured on the Infineon AURIX

system model offers low, predictable system service overheads for both basic and extended tasks even with cross-core interactions.

Porting the single-core SLOTH to a multi-core system did not require much effort as we were able to reuse the existing building blocks. Only small changes were required in MULTI SLOTH, mainly to add functionality such as task activation on a remote core. Overall, the complexity of porting our system was greatly reduced thanks to the hardware-centric approach of SLOTH. The current state of the system such as tasks in *running* or *ready* state is not kept in main memory, but rather is managed autonomously by the interrupt controller hardware. As these are modified with simple write-to-memory instructions, for example for setting the pending bit, no special synchronization in the kernel itself is required. Thus, a system built on MULTI SLOTH is still very concise as the cross-core interactions are handled asynchronously by the interrupt hardware.

Applications with real-time constraints on a multi-core platform also need predictable and priority-aware synchronization across cores. Such protocols have been developed: the Multiprocessor Priority Ceiling Protocol (MPCP) [20], [19], the Multiprocessor Stack Resource Policy (MSRP) [11], the PARALLEL-PCP (P-PCP) [9], and the FIFO Multiprocessor Locking Protocol (FMLP/FMLP$^+$) [3], [4].

Of these, the MPCP was specifically suggested for AUTOSAR [15]. Nevertheless, they were not considered by the automotive industry for the AUTOSAR standard. Instead, AUTOSAR only mandates unordered spinlocks (that is, spinlocks, but no particular ordering semantics) [1]. Wieder and Brandenburg have shown that with high workloads, ordered spinlocks achieve higher schedulability of AUTOSAR systems. They suggest that "AUTOSAR should mandate the availability of FIFO-ordered spin locks" and "AUTOSAR should also provide flexible priority-ordered spin locks" [23].

The reason that the industry has not introduced specific semantics yet is that *hardware costs* are their major concern. Only abstractions that can be implemented in memory/CPU-efficient manner on many platforms are accepted for the standard. As we learned from personal communication with involved engineers at Elektrobit Automotive, ordered spinlocks were in fact considered for the AUTOSAR multi-core standard, but eventually dropped as they are difficult to implement on architectures offering only a test-and-set instruction. However, the hardware platforms in use advance and so does their least common denominator. We are optimistic that, for instance, ordered spinlocks will be included into one of the next releases of the standard.

With our implementation in MULTI SLOTH we show that the MPCP can be implemented in an efficient manner on the Infineon AURIX, which is going to be one of the major multi-core platforms in the automotive sector. The nature of MULTI SLOTH demands the implementation to be dependent on the hardware platform, as the kernel relies on the interrupt controller to perform scheduling and dispatching. Likewise, the implementation of any synchronization protocol will always depend on the features of the hardware with regards to the available atomic instructions. Due to the generative approach in MULTI SLOTH, these hardware particularities can be hidden from the application programmer, while the resulting system still achieves efficient and predictable execution times.

The MPCP algorithm itself guarantees bounded remote blocking, in which a high-priority task is blocked by waiting for a low-priority task on another core. However, a special case in a software-scheduled implementation may exist in which the execution of a critical section can still be delayed by other low-priority tasks. When a global critical section is executed, the current priority is raised above all normal task priorities. Meanwhile, another previously suspended task on the same core may be signaled from a remote core. At this point, a scheduling decision is required. The resuming task may preempt other tasks executing global critical sections, when the ceiling priority of the signaled global resource is higher than the ceiling priority of the global resource held by the running task. In a traditional system with a software scheduler, this leads to deferred execution of the higher-priority critical section as the scheduling decision must be handled on the highest possible priority. With MULTI SLOTH, we solve this problem as the scheduling decision will be made asynchronously by the interrupt controller without interruption of high-priority tasks.

While we focus on the MPCP in this paper, the choice of the optimal synchronization protocol depends on the application and the length of the global critical sections as shown in a comparison of the MPCP and the MSRP by Gai et al. in [10]. Moreover, in the migration of real-time applications to multi-core it is vital to identify potential critical sections in the application. The utilization of a locking protocol must not always be the correct choice, as many complex data structures used in applications can be synchronized in a wait-free manner, while still adhering to timing constraints [22].

## X. RELATED WORK

The approach of using the interrupt controller for scheduling and dispatching was previously published in SLOTH [12] and SLEEPY SLOTH [13]. MULTI SLOTH presented in this paper extends this design to multi-core platforms and furthermore includes the MPCP as an example for a synchronization protocol. To the best of our knowledge, no prior implementation of the MPCP for an AUTOSAR system was presented before,

although the MPCP was proposed by Lakshmanan et al. in [15] for AUTOSAR multi-core systems.

Besides the MPCP [20], [19], other synchronization protocols for multi-core systems exist. These protocols not only differentiate in the semantics on the level of the operating system abstraction, but also in the way tasks are allocated to cores on a multi-core platform, which is not the scope of this paper. For the MPCP, the synchonization-aware task allocation is discussed by Lakshmanan et al. in [16].

With regard to our focus on partition fixed-priority scheduling on shared-memory systems, similar synchronization protocols include the Multiprocessor Stack Resource Policy (MSRP) [11], which relies on busy waiting and thus allows tasks to share the same stack. In an AUTOSAR system, the MSRP would also be applicable to basic tasks, whereas the MPCP requires extended tasks that are able to suspend due to using a separate stack for each task.

Furthermore, the FIFO Multiprocessor Locking Protocol (FMLP/FMLP$^+$) [3], [4] uses a FIFO ordering for waiting tasks instead of the priority-based approach in the MPCP. Other recent related work shows a wide variety of concepts. The OMLP [6] is for both partitioned and global scheduling with asymptotically optimal blocking behavior, the OMIP [5] focuses on FIFO-ordered locks improving latency-sensitive workloads, the MrsP [7] utilizes waiting time by delegating work of remote tasks to a waiting task while adhering to a FIFO ordering, and the MSOS [17] accomodates independent scheduling policies on multiple cores with a combination of FIFO and priority queuing. The diversity of the concepts introduced in these recent publications underlines the importance of real-time aware synchronization mechanisms for multi-core systems.

However, due to the use of the interrupt controller for scheduling decisions in MULTI SLOTH, our approach is limited to priority-based models. In off-the-shelf available microcontrollers, the arbitration in the interrupt hardware only takes the priority of the pending interrupts into account and have no record of the time they arrived. With interrupt controller hardware offering this detail, a synchronization protocol with FIFO ordering could also be implemented using the hardware-centric approach of MULTI SLOTH.

## XI. SUMMARY

In this paper, we have presented MULTI SLOTH, an efficient, AUTOSAR OS compliant real-time operating system for multi-core platforms. As the AUTOSAR OS multi-core system services do not provide resource synchronization with predictable timing guarantees, we extended the OS interface with the MPCP as an example for a priority-aware synchronization protocol. By adopting the design philosophy of the SLOTH kernel family to the multi-core domain, our implementation exhibits many properties favorable in real-time systems, including low latency and predictable overheads. On the Infineon AURIX as our reference platform, MULTI SLOTH achieves task activations in 65 cycles and round-trip times of 135 cycles for cross-core task activations. The overheads for our wait-free implementation of the MPCP range from 112 to 360 cycles.

## XII. ACKNOWLEDGEMENTS

We would like to thank our anonymous shepherd for their valuable input and guidance to improve this paper.

## REFERENCES

[1] AUTOSAR. Specification of operating system (version 5.0.0). Technical report, Automotive Open System Architecture GbR, 2011.

[2] Theodore P. Baker. A stack-based resource allocation policy for realtime processes. In *RTSS '90*, pages 191–200. IEEE, 1990.

[3] Aaron Block, Hennadiy Leontyev, Björn B. Brandenburg, and James H. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA '07*, pages 47–56. IEEE, 2007.

[4] Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[5] Björn B. Brandenburg. A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In *Eurom. Conf. on Real-Time Sys. (ECRTS '13)*, pages 292–302, 2013.

[6] Björn B. Brandenburg and James H. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS '10*, pages 49–60. IEEE, 2010.

[7] A. Burns and A.J. Wellings. A schedulability compatible multiprocessor resource sharing protocol – MrsP. In *Eurom. Conf. on Real-Time Sys. (ECRTS '13)*, pages 282–291, 2013.

[8] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *IEEE Real-Time and Embedded Technology and Applications (RTAS '06)*, pages 14–23. IEEE, 2006.

[9] Arvind Easwaran and Björn Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *IEEE Real-Time Sys. Symp. (RTSS '09)*, pages 377–386. IEEE, 2009.

[10] Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *IEEE Real-Time and Embedded Technology and Applications (RTAS '03)*, pages 189–198. IEEE, 2003.

[11] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS '01*, pages 73–83. IEEE, 2001.

[12] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *IEEE Real-Time Sys. Symp. (RTSS '09)*, pages 204–213. IEEE, 2009.

[13] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Sleepy Sloth: Threads as interrupts as threads. In *IEEE Real-Time Sys. Symp. (RTSS '11)*, pages 67–77. IEEE, 2011.

[14] Infineon Technologies AG. *AURIX™ – TC275T/TC277T – Product Brief*, 2013.

[15] Karthik Lakshmanan, Gaurav Bhatia, and Ragunathan Rajkumar. AUTOSAR extensions for predictable task synchronization in multi-core ECUs. In *Proceedings of the SAE 2011 World Congress*, 2011.

[16] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *IEEE Real-Time Sys. Symp. (RTSS '09)*, pages 469–478. IEEE, 2009.

[17] Farhang Nemati, Moris Behnam, and Thomas Nolte. Independently-developed real-time systems on multi-cores with shared resources. In *Eurom. Conf. on Real-Time Sys. (ECRTS '11)*, pages 251–261, 2011.

[18] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2011-08-17.

[19] Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Int. Conf. on Dist. Comp. Sys. (ICDCS '90)*, pages 116–123. IEEE, 1990.

[20] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Sys. Symp. (RTSS '88)*, pages 259–269. IEEE, 1988.

[21] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE TC*, 39(9):1175–1185, 1990.

[22] Philippe Stellwag and Wolfgang Schröder-Preikschat. Challenges in real-time synchronization. In *Workshop on Hot Topics in Parallelism (HotPar '11)*, pages 1–6. USENIX, 2011.

[23] Alexander Wieder and Björn B. Brandenburg. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *RTSS '13*, pages 45–56. IEEE, 2013.