AUTOBEST: A United AUTOSAR-OS and ARINC 653 Kernel*

Alexander Zuepke*, Marc Bommert*, Daniel Lohmann†
*RheinMain University of Applied Sciences, Wiesbaden, Germany
†Friedrich–Alexander-Universität, Erlangen–Nürnberg, Germany
Email: {alexander.zuepke|marc.bommert}@hs-rm.de, lohmann@cs.fau.de

***Abstract*—This paper presents AUTOBEST, a united AUTOSAR-OS and ARINC 653 RTOS kernel that addresses the requirements of both automotive and avionics domains. We show that their domain-specific requirements have a common basis and can be implemented with a small partitioning microkernel-based design on embedded microcontrollers with memory protection (MPU) support. While both, AUTOSAR and ARINC 653, use a unified task model in the kernel, we address their differences in dedicated user space libraries. Based on the kernel abstractions of *futexes* and *lazy priority switching*, these libraries provide domain specific synchronization mechanisms. Our results show that thereby it is possible to get the best of both worlds: AUTOBEST combines avionics safety with the resource-efficiency known from automotive systems.**

I. INTRODUCTION

The recent ISO 26262 standard *Road Vehicles – Functional Safety* [1] mandates "freedom of interference" of independent software components in automotive. Similar to ISO 26262 and IEC 61508, the safety standard RTCA DO-178B / EUROCAE ED-12B *Software Considerations in Airborne Systems and Equipment Certification* [2] puts high safety requirements on airborne software. Yet, automotive and avionics development follows different paradigms, which are also reflected by the level of detail of the ARINC 653 and AUTOSAR-OS standards: While ARINC tries to leave no room for interpretation to cope with the strict requirements regarding functional safety in avionic systems, AUTOSAR is targeted at an extremely cost-sensitive domain and deliberately leaves aspects as *implementation-specific* in order to allow for cost-efficient implementations of its protocols and feature sets.

Major techniques used in the avionics domain to systematically guarantee freedom of interference are *spatial* and *temporal partitioning*. Partitioning allows safe consolidation of independent software modules of different provenance and safety-criticality levels into isolated resource containers on the same computing platform. While the ARINC 653 standard explicitly defines a partitioning concept, embedded AUTOSAR software is not currently executed in a partitioned environment. Instead, AUTOSAR defines a weaker concept of optional *application separation* using memory protection only, not built into the foundations of the standard. However, due to the increasing aspiration towards *hyper-integration* of independent software modules of different criticality, magnitude, and quality onto single *electronic control units* (ECUs), a demand for safe, but still cheap, consolidation has arisen. The ARINC partitioning concept is in general well-suited to satisfy this demand.

This gives rise to the questions (a) how a partitioning RTOS approach in AUTOSAR could look like and what impact it has, (b) whether both AUTOSAR- and ARINC 653-specific requirements could be consolidated into a common base, and (c) what could be done to address overheads introduced by partitioning, which is especially important in such cost-sensitive markets as automotive. To give an answer to these questions, we present AUTOBEST, a united AUTOSAR-OS and ARINC 653 RTOS kernel that covers the requirements of both domains. We provide a comparative analysis of both RTOS standards, derive a common kernel design, and describe how we have dealt with commonalities and differences of both worlds.

We mitigate performance overheads caused by partitioning and restricting hardware access by two techniques: (i) *lazy priority switching* improves performance of critical sections, both for tasks temporarily disabling preemption in a partitioned environment or following the *OSEK priority ceiling protocol*,[1] and (ii) *futexes* (fast user-space mutexes) help to abstract higher level synchronization means such as semaphores, mutexes, or ARINC 653 queuing ports. Both techniques aim to improve average-case performance by using relatively expensive system calls into the operating system kernel only in the slow path. Instead, atomic *compare and swap* operations are used in the fast path (futexes), or the scheduling priority of a task is adapted lazily on scheduling decisions only. Both techniques were formerly not used in statically configured embedded systems.

Our results show that thereby it is possible to get the best of both worlds: AUTOBEST covers both standards and combines avionics' safety with the resource-efficiency known from automotive systems.

A. Contributions

In this paper, we present the AUTOBEST kernel and its design considerations with the following contributions:

- By comparison of AUTOSAR-OS and ARINC 653 concepts, we derive a common basis and subsequently show that it is possible to implement a compliant microkernel on small MPU-based platforms with low RAM requirements.
- We emphasize specific design patterns of which we expect lowered efforts in certification processes of safety critical systems build with AUTOBEST.
- With *lazy priority synchronization*, we present a performance optimization approach to mitigate the performance impact involved in partitioning, preserving system safety.
- We focus on *futexes* to implement higher-level synchronization objects in user space in order to keep a small kernel

* This work was supported by the German Federal Ministry of Economic Affairs and Energy on the basis of a decision by the German Bundestag.

[1] OSEK is the predecessor and a true subset of AUTOSAR-OS, so most AUTOSAR-OS concepts are in fact inherited from OSEK.

footprint and provide a proper abstraction to be commonly used by the hosted ARINC and AUTOSAR environments.
- We provide an estimation of the overhead involved in our implementation of resource partitioning. To this end, benchmark results on two processor architectures, ARM Cortex-R4 and PowerPC e200, are provided and assessed.

AUTOBEST targets compliance with the basic operating-system layer as defined by the AUTOSAR-OS 4.1 revision 3 specification [3], as well as the *Required Services* defined in supplement 3 to part 1 of the ARINC 653 standard [4].

B. Organization of this Paper

We discuss the AUTOSAR-OS and ARINC 653 RTOS specifications and their key design concepts in Section II. Then Section III maps these OS concepts to the AUTOBEST kernel. Lazy priority switching and futexes are discussed in Sections IV and V, followed by a performance evaluation in Section VI. We discuss our results in Section VII. Finally, Section VIII lists related work, and Section IX concludes our findings.

II. RTOS Concepts

In this section, we discuss basic AUTOSAR and ARINC 653 concepts as well as certain extensions to the standards. We highlight their similarities and differences, and define the terminology used in the rest of this paper.

AUTOSAR and ARINC use different (and sometimes conflicting) naming for otherwise similar OS concepts: AUTOSAR refers to scheduling entities as *tasks*, while ARINC 653 names them *processes* to avoid confusion with the task language construct in Ada. In this paper, we will use the term *task*. When ambiguous, we refer to *AUTOSAR tasks* or *ARINC tasks*. Also, we shorten ARINC 653 to just *ARINC*. Further, service functions in AUTOSAR are specified in CamelCase notation following a verb-object scheme, such as `GetTaskID()`, while ARINC uses underscores and capital letters, like in `GET_PROCESS_ID()`.

A. Priority-based Scheduling

For scheduling on a single processor (AUTOSAR) or in the scope of a single partition (ARINC), both standards use *strict priority-based task scheduling* with FIFO ordering on a priority tie. Higher priority values also reflect higher scheduling priority. Task scheduling is *preemptive*, that is, an arriving higher priority task will interrupt a currently executing lower priority task. Also, tasks can raise their effective scheduling priority at run time to prevent preemption.

For scheduling on multiple processors, AUTOSAR defines independent scheduling per core with sets of tasks and *interrupt service routines* (ISRs) bound to a configured processor. However, in compliance to configured application rights, tasks on different cores can interact by activating tasks or setting events across cores. For inter-core synchronization, AUTOSAR specifies spinlocks, but is not fully set on their semantics and the underlying implementation [5].

On the ARINC side, an execution environment using more than one processor in a single partition is not yet standardized. Nevertheless, it is possible to schedule ARINC 653 partitions independently on different cores.

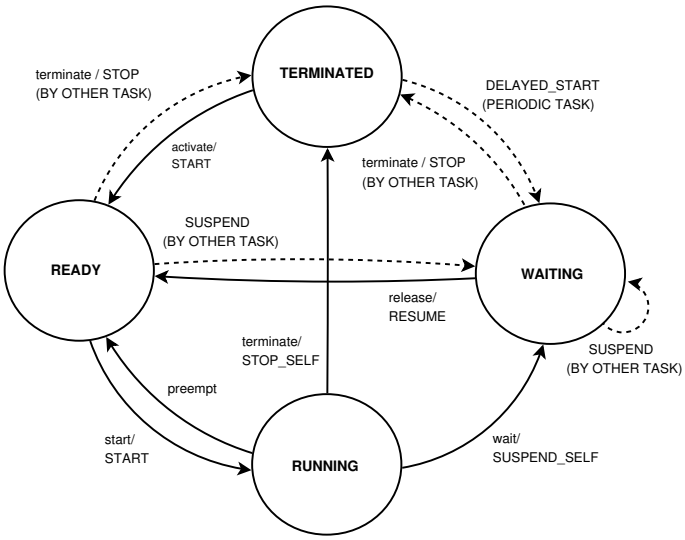

Figure 1: Task states and state transitions of AUTOSAR and ARINC 653: Transition labels in capital letters refer to the ARINC task API, whereas dashed lines express transitions restricted to the extended task model of ARINC 653.

B. Task Models

Both standards also define similar task models with four execution states per task, as depicted in Figure 1:

- **RUNNING** for the currently executing task on a processor.
- **READY** for tasks eligible for execution, but currently not selected by the scheduler.
- **WAITING** for tasks not running while waiting for an external event (in the abstract sense).[2] When the event is signalled, tasks become READY. We will use the verb *to wait* to denote a transition into WAITING state, while *to release* expresses a transition from WAITING to READY state.
- **TERMINATED** for tasks that have completed execution and wait to be re-activated. Semantically, both standards define this state, but the naming diverges: AUTOSAR calls it *SUSPENDED*, in ARINC's terminology it's *DORMANT*. The TERMINATED state is also the initial state of a task. The verb *to terminate* expresses a transition to TERMINATED state, and *to activate* is used for the transition from TERMINATED state to READY state.

However, the allowed state transitions differ between AUTOSAR and ARINC. Except for the purposes of activating and releasing other tasks, AUTOSAR restricts the scope of the task API to transitions of the currently RUNNING task only. On the other hand, ARINC extends AUTOSAR capabilities and offers much more control over other tasks, such as external task termination or enforced waiting states:

- `START()` and `STOP_SELF()` for task activation, similar to `ActivateTask()` and `TerminateTask()` in AUTOSAR.
- `STOP()` to terminate other tasks than the caller.
- `DELAYED_START()` to activate a task after a given wait time. This leads to the transition sequence TERMINATED $\rightarrow$ WAITING $\rightarrow$ READY for the target task.

[2] Both AUTOSAR and ARINC also define an event mechanism with different semantics.

- `SUSPEND()` and `SUSPEND_SELF()` enforces a task to be exempted from scheduling and enter WAITING state, while `RESUME()` reverses suspension.
- An optional *relative timeout* with nanosecond resolution can be supplied to all potentially waiting operations, allowing a transition back to READY state when the timeout expires.
- Additionally, ARINC supports changing the scheduling priority of other tasks at run time.

Neither of these ARINC-required extensions conflict with AUTOSAR.

C. Task Types, ISRs, and Hooks

Both systems handle different types of tasks: ARINC supports *periodic processes* with built-in mechanisms to track tasks' periodic activations and deadlines, and *aperiodic processes* for even-driven workloads. On the other hand, AUTOSAR distinguishes between *basic tasks* and *extended tasks*. While extended tasks can wait on an AUTOSAR event, basic tasks cannot enter the WAITING state. Instead, basic tasks support a concept named *multiple activation* where pending activations are queued up to a configured upper limit to immediately restart the tasks upon completion. Combined with non-preemptive scheduling, this allows stack sharing between basic tasks on systems where memory resources are tight. Different AUTOSAR *conformance classes* further allow implementations to remove support for extended tasks or to simplify ready queue management operations to further save memory for small-scale automotive systems.

Additionally, automotive systems handle interrupt-triggered workloads, therefore the specification also discusses interrupt handling and the interaction of *interrupt service routines* (ISRs) with other tasks. AUTOSAR classifies interrupt handlers into two categories: ISRs of category 1 run outside the scope of the operating system and interrupt normal execution without impacting scheduling. Contrary, ISRs of category 2 are handled like high priority pseudo-tasks and may interact with other (normal) tasks in the system. The AUTOSAR interrupt concept further allows higher priority interrupts to preempt lower priority ones, therefore ISRs fit into the AUTOSAR task state model as a special kind of basic tasks.

In contrast, interrupt handling is kept outside the scope of ARINC, as ARINC neither specifies API services for interrupt handling nor an interrupt model at all. Instead, interrupt handling can be provided by vendor specific extensions to called *system partitions*. ARINC only demands that interrupts must not interfere with partitioning.

The remaining entities in both systems are *hooks* and *callbacks* used for error handling, system tracing, startup, shutdown, and further notification means. Conceptually, these hooks are implemented as pseudo-tasks and run uninterruptedly at a priority level above all tasks and ISRs. In both systems, the hooks related to error handling, namely AUTOSAR's *error hook* and *protection hook* and ARINC's *error handler process*, stand out and have special privileges to change other task's state or influence scheduling.

D. Task Synchronization, Notification, and Activation

For critical sections, both systems allow to disable preemption of the currently running task. ARINC's `LOCK_PREEMPTION()` and `UNLOCK_PREEMPTION()` support nested critical sections by using a global preemption counter, which,

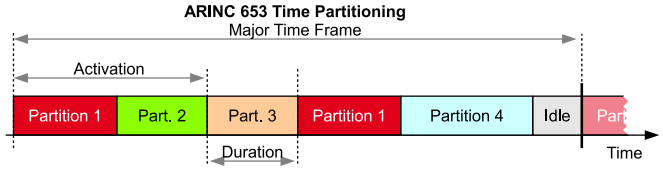

Figure 2: In ARINC 653. the major time frame describes a recurring cyclic time partition schedule. Each partition has one or more time partition windows (time slots) in the schedule. Partition 1 has two windows to support task periods of half the major time frame.

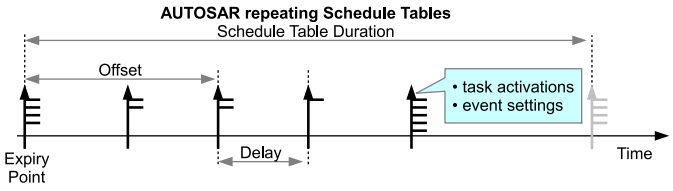

Figure 3: AUTOSAR schedule tables comprise multiple expiry points which in turn activate tasks or set events. Schedule tables either repeat or have one-shot characteristics.

when nonzero, prevents preemption. This relates conceptually to AUTOSAR's `RES_SCHEDULER`. Using *AUTOSAR resources*, the two operations `GetResource()` and `ReleaseResource()` temporarily raise a task's scheduling priority to the resource's statically configured priority level in order to access a resource exclusively or to even prevent preemption in case of the highest priority resource `RES_SCHEDULER`. To form groups of tasks that do not preempt each other, so called *internal resources* boost a basic task's priority to a configured ceiling priority upon being scheduled the first time. Resource access can be nested, defining an immediate priority ceiling protocol named the *OSEK Priority Ceiling Protocol*. Additionally, AUTOSAR specifies services to enable and disable interrupts. Similar services are not defined by ARINC for normal applications.

Next to the priority ceiling mechanism for synchronization, AUTOSAR provides an *event* mechanism for notification. Each task can suspend execution and wait for any event in a set of events to be signaled by other tasks.

Contrary, ARINC provides more sophisticated means: on the lower levels, it supports counting *semaphores* for synchronization and *events* for notification. With ARINC, waiting on events is restricted to a single event at a time. Further two unidirectional message-passing interfaces are defined between tasks: *Buffers* queue messages up to a certain configured size/depth with optional waiting on full/empty conditions of the queue. *Blackboards* provide asynchronous multicast behavior: Multiple tasks can read the latest message until it is cleared. Similarly, ARINC provides *queuing ports* and *sampling ports* with similar queuing/non-queuing semantics for communication between tasks of different partitions. Finally, all synchronization objects in ARINC can be instantiated with either a FIFO or priority-ordered *queuing discipline* of its waiting tasks.

AUTOSAR also defines higher-level message-passing interfaces, but these are built on top of the event mechanism and are therefore not part of AUTOSAR's operating-system specification. For example, AUTOSAR's *Run-Time Environment* (RTE) generates communication code to transfer data between different applications, regardless of whether the applications reside on the same or different ECUs.

Both standards support time-triggered and event-driven task activations. ARINC addresses both paradigms in the task model with the concept of periodic and aperiodic processes. The concept of time-triggering further extends to *time partitioning*, as Figure 2 shows. Time partitioning allows the execution of sets of temporally isolated tasks in predefined *time partition windows* (time slots) in a recurring cyclic *time partition schedule* named the *major time frame*. Because of that, periodic processes in ARINC have to synchronize their execution with the global time partition schedule, for example a task's period must be a multiple of its time partition's period.

In contrast, AUTOSAR was mainly designed with event-driven workloads in mind. Here, concepts for time-triggered workloads are mapped onto event-based mechanisms. For time-triggered tasks, AUTOSAR defines cyclic *schedule tables* to release tasks at certain defined *expiry points*, as Figure 3 shows. The schedule tables are driven by *counters* of fixed or variable events in a vehicular environment, such as cyclic messages between ECUs or the current rotational speed of an engine. Additionally to schedule tables, AUTOSAR supports *alarms*, which are conceptually simplified versions of the schedule tables with just one expiry point.

Finally, both systems define similar techniques to detect if a task misses its deadline by monitoring certain state transitions of a task and accounting its execution time. ARINC supports built-in deadline monitoring for all kinds of tasks. The *timing protection* facilities in AUTOSAR additionally detect violations of a task's configured inter-arrival time or if a task exceeds its assigned execution time budget.

E. Isolation and Protection

The concept of robust partitioning is deeply embedded into the ARINC 653 standard, with the goal of separation of applications of different criticality levels for fault isolation reasons in avionic systems. A *partition* in the abstract sense defines a set of tasks related to certain functionality, separated and isolated from tasks in other partitions. Partitioning is provided on two levels: *spatial* or *resource partitioning* restricts the scope of resources a task can access to its own partition with the exemption for explicitly configured communication channels to other partitions, while *time partitioning* defines strict temporal isolation, as described in the previous section. Additionally, ARINC defines a concept for *health monitoring* (HM): On exceptions or deadline misses, the OS kernel checks statically configured *HM tables*, and then performs a configured action, such as a shutdown/restart of the faulty partition or the entire system, or forwarding the error to the application-level error handler process.

For spatial partitioning on the automotive side, AUTOSAR provides an optional *application* concept for grouping tasks into applications, based on the OSEK extension *protected applications* [6]. This concept basically resembles ARINC resource partitioning, although it is less strict and not fundamentally applied in AUTOSAR. AUTOSAR applications are distinguished into *trusted* and *non-trusted* ones, where trusted applications may run in supervisor mode. With the AUTOSAR timing protection, an optional mechanism is specified to monitor execution time budgets, resource access durations and task inter-arrival times in order to detect timing errors and consequently restart tasks, applications, or the ECU.

F. System Configuration and Software Development Process

In general, both systems follow similar concepts to maintain the software life cycle in vehicular and airborne systems, using modular structures and clearly defined interfaces between (software) components: AUTOSAR defines a static configuration approach where the whole system configuration with all resources is known at compile-time. System configuration can be considered as a process with a fine-granular tuning possibilities on all levels of the system. Application integrate into this picture as configurable components and code generators are used to tailor a specific system and omit unused features.

In contrast, ARINC defines different roles for configuration and development for an even higher amount of portability and code reuse: the *system integrator* defines the partition environment by statically assigning resources to partitions and defining explicit communication channels between the partitions, and the *application developer* implements the application on the abstract interfaces provided by the integrator. Application startup differs in such a way that avionics software allocates and configures all required resources dynamically at partition startup in run time, not at compile time. But since the maximum amount of resources provided to each partition is bounded and each partition's configuration is defined at compile time, ARINC effectively supports a static configuration as well, and dynamic resource allocation is reduced to a configuration check only.

Additionally, AUTOSAR provides a built-in *application mode* concept to activate different subsets of tasks at startup using an *autostart* mechanism to switch between different setups for different use case scenarios, like parking/driving.

G. Discussion of the RTOS Concepts

The comparison shows that both standards define compatible task models. While ARINC extends and completes the task transitions defined by AUTOSAR, AUTOSAR allows to further restrict the task model in favor of small scale systems. We omit these memory saving optimization and focus on AUTOSAR's largest conformance class *ECC2* only.

From an implementer's point of view, AUTOSAR exposes a much simpler structure than ARINC, as AUTOSAR's concepts of ISRs and hooks map directly to a processor's hardware capabilities for interrupt and exception handling. AUTOSAR implementations often maintain state data only for tasks and offload handling of ISRs to the processor's interrupt controller. In fact, the AUTOSAR OS abstraction was designed to be light-weight and tailorable to exploit existing hardware as much as possible. Additionally, static task state data like task type, entry point, or stack pointer, is known at compile time already.

For each kind of AUTOSAR task, this run-time state data comprises:

- the task's current scheduling priority and scheduling state,
- the position in the ready queue (e.g., a linked-list node), and
- a set of saved registers (the task's execution state) while the task is waiting or got preempted.

For an extended task, an implementation has to provide two additional bitmasks for the task's event state, that is, one for the task's pending events, and one bitmask for the events the task is waiting on.

For a basic task, even less additional state has to be

maintained. The only challenge for an implementation is to track the task's pending activations correctly, for example by keeping multiple nodes in the ready queue.

In contrast, ARINC requires much more state data per task:

- As tasks are created dynamically at partition startup, entry point and stack pointer become dynamic state as well.
- Termination or priority changes of tasks other than the current one require removal of tasks from the ready queue from arbitrary positions.
- Tasks can wait with timeouts. This requires management of pending timeouts and saving a task's requested expiry time.
- Additionally, tasks can wait on an ordered wait queue of an ARINC synchronisation object, such as, a semaphore.
- For periodic execution, a task needs to track the time of its last activation to calculate the next activation point.
- Finally, ARINC deadline management requires tracking of a task's deadline expiry time.

House-keeping the additional ARINC-related data requires significantly more memory per task, and, while neither of these features requires a complicated implementation, the overall implementation of an ARINC-compatible system will be obviously more complex than an AUTOSAR one. However, if we consider adding the timing protection facilities to AUTOSAR, both implementations come closer again.

On the other hand, from the perspective of an ARINC implementation, compatibility to AUTOSAR's task model needs just minor additional adjustments to support multiple activations for basic tasks. We assume such an ARINC implementation to also support some kind of interrupt handling for device drivers in system partitions, providing similar interaction capabilities as AUTOSAR requires.

To summarize this discussion, AUTOSAR and ARINC have a lot in common, and where their concepts differ, they do not conflict. The main difference is on the cultural side: while in ARINC partitioning is rooted in the foundations of the system, it is just an optional feature for AUTOSAR. In general, the requirements in ARINC are more strict than in AUTOSAR, but AUTOSAR concepts can be mapped to ARINC ones. From the operating system perspective, we can consider ARINC a superset of AUTOSAR.

III. AUTOBEST ARCHITECTURE

This section discusses the architecture of AUTOBEST. We describe our goals and design considerations for AUTOBEST, followed by a description of the kernel, our ARINC and AUTOSAR abstraction, and give details on the implementation.

A. Motivation and Goals

Targeting automotive and industrial markets, we designed AUTOBEST to execute software of different safety and security levels on the same MPU-based platform. To ease certification efforts, we wanted to have a minimalistic implementation with strict partitioning build into the foundations of the system.

Initially, support for ARINC 653 was not a requirement, but ARINC's inter-partition communication facilities (queuing ports, sampling ports) are attractive for decoupling of data flow for security reasons. Therefore, it became important for AUTOBEST to keep the kernel API agnostic and move RTOS API differences out into user space, where possible. This also allows to implement further threading APIs in the future. Further, partitions should use all the core services to prevent dead code.

Regarding system configuration, we follow ARINC's approach of distinct *system integrator* and *application developer* roles, as AUTOSAR's system configuration model can be fitted into ARINC's, but not vice versa. Also, we wanted to use code generators as much as possible. However, *generated code* requires extensive testing, code reviews, and coverage analyses. Therefore, the tools emit *configuration data* only, comprising arrays of C language data structures and driving the internal state machines, such as for AUTOSAR schedule tables. This also applies to conditional code compilation, which we tried to reduce to a minimum in AUTOBEST. Having configuration data abstracted from the program code also helps during re-certification of a project where only small parts of the system change and a full re-certification of the other components is not necessary. Lastly, all components can be linked to different configuration sets for testing purposes or coverage analyses. While this approach is standard in avionics, it is still novel in the automotive domain.

B. Design Considerations: Hypervisor vs Microkernel

As we selected partitioning as the "boundary" for protection means, all tasks of a partition can affect each other, similar to related threads in a process of larger operating systems. This requires a design where the kernel is at least in charge of partition scheduling, programming the MPU, and management of shared resources, especially the interrupt controller. The rest could be implemented in either AUTOSAR- or ARINC-specific libraries that complete the respective APIs on a common kernel interface.

This leaves open whether other critical operating-system services, such as the task scheduler, could be implemented in user space as well. So we evaluated the following two design approaches for AUTOBEST:

- A **para-virtualizing hypervisor** that schedules virtual machines based on the time partition schedule and exposes a *virtual CPU* (vCPU) interface [7]. Partitions then implement RTOS-specific task scheduling in user space. Interrupts are received by the hypervisor and injected into the associated virtual machines through a *virtual Interrupt Controller* (vINTC) interface.
- A **microkernel** that schedules both time partitions and tasks, with the kernel offering a base abstraction of both AUTOSAR and ARINC APIs while implementing non-critical features in user space. With this approach, interrupts activate high priority pseudo-tasks in the associated partitions.

Additionally, both approaches must support drivers and interrupts in both supervisor mode and user mode, for time critical mechanisms demanding short response times and devices shared between partitions, respectively.

The hypervisor approach is appealing for three reasons: Firstly, the vCPU interface with its hardware-like abstractions simplifies porting of existing operating-system code to AUTOBEST and keeps the code running in supervisor mode as small as possible. Secondly, user code required to perform certain operations in supervisor mode can be examined using static code analysis methods at system integration time to not compromise partitioning [8]. Thirdly, a vCPU and vINTC architecture resembling hardware virtualization capabilities built into today's desktop CPUs have a high probability to appear

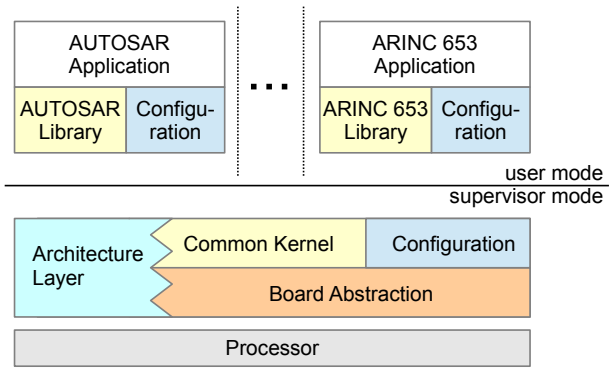

Figure 4: AUTOBEST system structure. Partitions of different type execute in user mode on top of a small microkernel, using specific libraries to abstract the desired RTOS API. Partition code and kernel are distinct binary images comprising linked object code components.

in tomorrow's embedded processors as well and, thus, provide an easy upgrade path for future compatibility.

On the other hand, typical automotive microcontrollers support prioritized interrupt handling, which interferes with time partitioning: While a partition is not running, interrupts arriving in the mean time must be queued. When the related partition is scheduled again, they have to be injected in the order of their priority. Assuming a high interrupt load, the necessary facility for interrupt ordering and individual masking eventually resembles the priority-ordered ready queues used for task scheduling. Thus, the para-virtualizing hypervisor model requires a two-level scheduler for partitions and ordering of ISRs in the kernel *and* an additional task scheduler in user space. Due to concerns about memory consumption and system complexity, we opted for the microkernel model instead and implemented full task scheduling in the AUTOBEST kernel.

Figure 4 depicts the resulting system structure of an AUTOBEST system executing an AUTOSAR and an ARINC partition. The kernel and the applications running on top are independently compiled and linked binaries. Each binary comprises multiple components, abstracting configuration data, platform and architecture dependencies, and library and application code.

C. AUTOBEST *Microkernel*

The AUTOBEST microkernel implements a superset of both AUTOSAR and ARINC task management and scheduling services, while keeping the differences of the RTOS APIs out of the kernel. To achieve this, error checks demanded by AUTOSAR or ARINC that do not affect the integrity of the kernel (and thus partitioning) were moved into user space. Similarly, return codes of kernel services are translated to the right domain specific variants by the libraries. As result, the kernel supports all task types, states, and transitions listed in Section II using an RTOS-agnostic API.

To ensure separation of partition resources, the kernel API uses indirections when addressing resources: using relative indices, partition code can only access a limited, but contiguous subset of the kernel's configuration data, such as the partition's task in the kernel's task state array. This kind of indirection is used in all APIs of the kernel, resulting in a zero-based local name space for each partition. Thus, for example when adding more tasks to a partition, task IDs in unaffected partitions do not change. This eases re-certification.

The kernel provides a *futex* mechanism to allow the implementation of all ARINC task synchronization means (semaphores and events, buffers and blackboards, queuing and sampling ports) in user space. The futex mechanism abstracts the two operations offered by all means, namely to *wait* with an optional timeout on the object, or to *notify* (and effectively release) other tasks already waiting. In ARINC, all services expose similar semantics: each synchronization object supports either a FIFO or priority ordered *queuing discipline* of its waiting tasks, with the discipline being defined at partition initialization time. The futex mechanism is discussed in Section V in detail.

Of the synchronization means required for AUTOSAR, the AUTOBEST kernel implements events, alarms, and schedule tables at kernel level. While AUTOSAR events could have been abstracted with futexes as well, we opted for an in-kernel implementation due to the tight coupling of events to alarms and schedule tables to keep the implementation simple.

Both AUTOSAR resources and the services to enable/disable interrupts are mapped to priority changes, where the RUNNING task temporarily raises its scheduling priority to a defined priority level above all other tasks or even ISRs. Similarly, ARINC's preemption control mechanism also uses priority changes to implement critical sections. As these operations are used frequently by application code, AUTOBEST employs a *lazy priority switching* scheme improving the performance of temporary priority changes, as further discussed in Section IV.

D. Time Partitioning and Scheduling

Time partitioning in AUTOBEST is implemented using dedicated ready queues per partition. The current implementation of the time-partition scheduler switches *partitions windows* according to a pre-configured cyclic TDMA (time division multiple access) schedule. This schedule is implemented by using a globally active AUTOSAR schedule table comprising special expiration points to switch time partitions. Alternatively, time partitioning can be configured off, allowing the partitions to share ready queues and a single priority space for all tasks.

Ready queues are represented by an array of doubly-linked lists for 256 priority levels. AUTOSAR's simpler task transition model would have allowed AUTOBEST to use single-linked lists, but the ARINC services to terminate other tasks than the currently executing one or to change the scheduling priority of arbitrary tasks make the use of doubly-linked lists more reasonable.

E. ARINC Abstraction

The ARINC library mainly translates the ARINC 653 process API to the according kernel services and uses futexes to implement the ARINC specific *intra-partition* communication mechanisms, namely events, semaphores, buffers, and blackboards. As these objects are used for partition-internal synchronization only, ARINC does not require a static configuration for them. An implementation has to just provide enough memory to create these objects. Here, the system integrator needs to assign enough memory and futex wait queues to the partition, without further specifying their purpose, and the partition takes care of the creation of the objects itself.

Inter-partition communication channels need to be checked and validated at system startup. Therefore, the library checks if the attributes of the requested queuing or sampling ports match the configuration. Both queuing and sampling ports are eventually implemented using shared memory and futex wait queues between partitions, similar to the process local communication mechanisms. Section V explains the protocol in detail.

However, process initialization deviates from ARINC: as the kernel cannot dynamically create tasks at run time, all processes need to be properly configured upfront in the system configuration. Thus, ARINC's `CREATE_PROCESS()` reduces to a check against the partition's configuration.

In summary, ARINC services implemented in the user space library either wrap kernel services, take care of initialization and configuration checking, or provide communication and synchronization means built upon lazy priority switching and futexes.

F. AUTOSAR Abstraction

The AUTOSAR library follows the spirit of the ARINC library: it mainly performs translation of kernel services to AUTOSAR ones and maps AUTOSAR resource handling and interrupt enable / disable services to priority changes of the currently executing task.

The main difference to the ARINC library implementation however is that the AUTOSAR library checks all error codes and invokes a configured error hook when necessary. On errors, the library functions prepare an *error record* comprising the arguments and fault condition of the failed service call and activate the error hook pseudo-task, which executes at highest priority in the partition. Thus, error handling is kept in the faulting partition. However, the invocation of the error hook is more expensive compared to solutions just disabling interrupts and calling the configured error-routine directly.

On top of AUTOSAR library services, AUTOSAR RTE services provide standard communication means. Communication between different partitions is currently implemented via a virtual CAN bus driver located in the board component of the kernel or via shared memory and cross-partition event handling. However, we plan to adapt the RTE to use a futex-based approach as well.

G. System Configuration

Based on each partition's configured memory requirements, configuration tools allocate resources, assemble the system's MPU configuration, and generate linker scripts for the applications. Finally, an integration tool assembles the partition binaries and the kernel into a bootable image.

In contrast to full-blown virtual memory management, restrictions of the MPU hardware, such as the limited number of MPU windows or special alignment restrictions, often limit the scalability of this approach. For larger systems the integrator has to take this into account and can influence the memory allocation steps and simplify the memory layout manually.

Because ARINC and AUTOSAR specify different configuration formats, the tools translate and integrate the domain specific configuration data into our own internal representation of the system.

H. Device Drivers and Interrupt Handling

AUTOBEST supports implementation of ISRs at both kernel level and in user mode. On dispatching interrupts, the kernel invokes a configured callback for the current interrupt source. The callback either handles the interrupt request directly in the kernel, or masks the interrupt source and activates an associated ISR task. ISR tasks are scheduled like normal tasks in their associated partition with the only difference that the interrupt source is unmasked again on task termination. Conceptually, these *threaded interrupt handlers* in user space can be used to implement AUTOSAR category 2 ISRs only, while interrupts handled at kernel level cover category 1 and 2 ISRs. However, the scope is different: interrupts handled at kernel level have global scope and affect the whole system, while interrupts at user level only affect their related partition.

For flexibility reasons and due to hardware constraints,[3] AUTOBEST supports device drivers also in both supervisor mode and user mode.

To access kernel-mode drivers from partition code, a system-call gateway using static access tables invokes the configured callback. In the AUTOSAR library, the `CallTrustedFunction()` service is mapped to this interface. The access tables can be configured differently for each partition to enable fine-grained access permissions.

For communication to drivers implemented in different partitions in user space, the kernel's existing synchronization and notification mechanisms are sufficient. Statically configured *shared memory regions* between different partitions allow user-space code to transfer larger amount of data without involving the kernel. After transfer, one partition notifies a task in the other partition through explicitly configured channels, a shared wait queue or an inter-partition event, for instance. Section V discusses this in detail.

I. Partition-Preemptive Kernel

Most AUTOSAR services are designed to have a constant run-time behavior by affecting only a single entity. Thus, from the AUTOSAR perspective, this would allow to use a non-preemptible kernel model using a single kernel stack to achieve short and bounded execution times for all operations. However, operations related to partition startup and shutdown, such as, terminating all tasks, or a request to wake more than one task waiting on a futex take an extraordinary long time compared to other operations.

By default, the kernel is non-preemptible during most system calls, but when performing long running operations, the kernel becomes preemptible *at partition level* and does not delay execution of the next partition on a time partition switch. We implemented explicit preemption points after notification or termination of a task and use dedicated kernel stacks per partition and an extra interrupt stack for interrupt handling in kernel space.

On scheduling any task of a previously preempted partition, the kernel first completes any ongoing operation in the kernel before returning to user space and accepting new system calls. Thus, time partitioning switches and category 1 interrupts are not affected by long-running operations.

[3] On some embedded microcontrollers, write access to hardware IO registers is only allowed in supervisor mode, matching the safety model of AUTOSAR.

J. Implementation Details

For the implementation of AUTOBEST, memory usage of ARINC task attributes was our biggest concern: task periods, timeouts, deadlines, time capacities, and phases to the start of a partition window are specified in a 64-bit signed integer format with nanosecond resolution as required by ARINC.

While a task's period and time capacity are set at task-configuration time (and, thus, can be kept in ROM), the other three attributes can change at run time. Additionally, as tasks can wait on a wait queue with timeout and simultaneously have deadline monitoring enabled, three queues are needed. When using pointer-based doubly-linked lists, this consumes 48 Bytes of RAM per task.[4] Fortunately, tasks cannot wait for a timeout *and* be on the ready queue at the same time, thus the list nodes can be shared. Together with event masks and other internal data, this yields an overall requirement of 64 Bytes of RAM per task, plus an architecture specific register save area when the task is preempted.

The second most important memory hogs are the ready queues kept per time partition. With 256 priority levels and node pointers for timeout and deadline management, this adds up to a requirement of more than 2 KiB RAM per time partition.

Finally, stacks require memory as well: one small kernel stack per partition and a dedicated stack for interrupt handling.

IV. Lazy Priority Switching

When AUTOSAR resources are used, each time a critical section is entered or left the current task's scheduling priority needs to be boosted temporarily. Using system calls to change a task's scheduling priority poses a significant overhead, especially if the system call costs are high, the time spent in the critical section is short, and the executing task is not interrupted in the critical section. Therefore, it is worthwhile to optimize priority changes.

Instead of *immediately* changing a task's scheduling priority at the beginning and the end of the critical section using two system calls, we consider the following *lazy* approach: If no other task arrives while the task is inside a critical section, the system calls could be omitted. Instead, on entering the critical section, the task indicates a request to update its scheduling priority to a desired target priority level in a variable named `uprio` in user space. Conversely, the task revokes the request when leaving the critical section by restoring `uprio`'s former value which reflects the task's normal scheduling priority. Now, as long as the task is not interrupted in the critical section, the priority change is not observed by the kernel.

However, when the task is interrupted in the critical section and the kernel checks if the task needs to be preempted, the kernel considers the task's updated scheduling priority `uprio` instead. Having now *synchronized* the kernel's view of the scheduling priority of the task, `kprio`, the kernel either preempts the task at its new priority level, or lets it continue to execute. In any case, the task in user space needs now to check if it was interrupted in the critical section. After reverting the priority change and restoring the previous value of `uprio`, the task checks if `kprio` $>$ `uprio`, and if true, the task issues a system call to enforce synchronization of `kprio` and finally switch back to its original priority.

While this protocol reduces the number of required system calls from two to *at most one* in the interrupted case of a single critical section, we still can do better. We introduce a new variable `nprio`, where the kernel reflects the priority of the *next* eligible task the scheduler would select for execution. The kernel updates `nprio` every time a new task becomes ready. If no other task is ready, the kernel uses the value 0. Now, when leaving the critical section and after lowering the priority, the task in user space checks `nprio` $>$ `uprio` instead. If true, the task issues a system call for preemption, as another task definitively has a higher scheduling priority. This optimization further reduces the number of required system calls, especially in nested critical sections, a pattern found for example when using nested AUTOSAR resources. System calls are now required only in the case of preemption and not just interruption as before.

In AUTOBEST's implementation of this protocol, each partition uses its own copies of `uprio` and `nprio` for MPU programming reasons, where both variables are kept in the partition's data segment for fast access. Also, when evaluating `uprio`, the kernel bounds the user-space provided priority value against a task's configured maximum priority limit.

Regarding functional safety, tasks must implement the protocol correctly and issue a system call when a preemption condition is detected on lowering the priority. If not, the effect is the same as if a task would not leave its critical section in the first place and delays scheduling of other high priority tasks. However, this kind of error would be considered a normal programming error, similar as AUTOSAR requires that resources need to be acquired and released in a properly nested fashion. Additionally, the priority upper bound limits the effect such an error would have and AUTOSAR timing protection or ARINC deadline detection means could detect the condition and terminate the runaway task. In no case, such an error would be propagated beyond the task's time partition.

Lastly, the kernel reveals in `nprio` that other lower-priority tasks became ready, which could be a problem from a security point of view. However, the system integrator should place security-sensitive applications into dedicated time partitions then, as each partition has its own private ready queue and instance of `nprio`.

A detailed analysis of the presented protocols can be found in our previous work [9].

V. Futexes in Statically Configured Systems

The Futex mechanism in Linux [10] [11] describes a lightweight way to implement POSIX thread synchronization objects mostly in user space. Using atomic *compare and swap* or similar operations on an integer-sized variable in user space, threads can, for example, lock and unlock an uncontended mutex without requiring system calls. The Linux kernel is involved only in the case of contention to suspend the calling thread for waiting or waking up (releasing) other threads. In this case, the Linux kernel allocates a wait queue on demand when the first thread starts to wait, and frees the wait queue when the last thread was woken up. Internally, a futex object's wait queue is referenced by hashing the futex user-space variable's address. For futexes shared between different address spaces, Linux uses the physical address of the futex variable for hashing rather than its virtual, per address-space address.

For an implementation of futexes in statically configured systems, such as AUTOBEST, neither address hashing nor

[4] Three 64-bit time values and three list nodes of two 32-bit pointers each.

dynamic allocation of a wait queue are necessary. Instead, as all synchronization objects are known at system integration time, the futex's wait queues can be statically allocated and referenced by consecutive index numbers, for example in a partition-specific access table. Further, the different kernel operations make the wait queue effectively a two-ended communication channel: one side *waits* for the other side to send a notificating *wake up*. This also allows to grant both ends to different partitions. Additionally for ARINC, the waiting side must be able to configure different queuing disciplines (priority or FIFO ordered).

This results in the following system-call API for futexes, respective wait queues, with `wq_id` denoting the wait queue:

```
err_t sys_wq_init(uint wq_id, int discipline);

err_t sys_wq_wait(uint wq_id,
    uint *futex, uint futex_compare,
    prio_t wait_priority, timeout_t timeout);

err_t sys_wq_wake(uint wq_id, int wake_all);
```

A wait operation performs two steps in the kernel: At first, the kernel compares the associated user-space futex variable `futex` against `futex_compare`. If the futex value has changed in the meantime, the calling task was preempted in user space before calling `sys_wq_wait()`. In this case, the kernel returns an error to indicate a *lost wake-up condition*. Otherwise, the kernel suspends the calling task with a specified optional timeout and enqueues the task into the wait queue according to the configured queuing discipline and given `wait_priority`. The override of the task priority is required to combine futex waiting with lazy priority switching, as the kernel is not able to know the task's original scheduling priority.

Conversely, `sys_wq_wake()` either releases one or all tasks waiting on the wait queue. Internally, the kernel uses doubly-linked lists for ordered enqueuing, to deal with waiting operations which can be cancelled due to timeout expiry or task termination.

Higher-level synchronization objects in ARINC use up to two wait queues simultaneously, while both ends can be assigned to different partitions. A queuing port between two partitions is implemented using a shared memory comprising a ring buffer and a 32-bit control word as futex value, and two cross-partition wait queues with different directions. The first wait queue on the sender's side is used to wait on a queue-full condition, while the second wait queue on the receiver's is used to wait if the queue is empty. The futex variable encodes the position of read and written buffers as well as two bits for full and empty conditions and is updated from both sides atomically using compare-and-swap instructions. On a send operation, the sending partition first checks for an available empty slot in ring buffer, copies the message into the ring buffer, updates the control word, and wakes the first receiving task waiting on the wait queue, if any. If there is no empty slot available in the ring buffer, the sender waits on its own wait queue for a free entry in the buffer. The receiver side follows the same protocol to wait for incoming messages.

The queuing port implementation further raises the calling task's scheduling priority above all other tasks during message copy operations to ensure atomicity of the message transfer. Otherwise, a low priority task could be interrupted during the message transfer part of a queuing port operation and subsequent reads could access the unfinished message. This mechanism works similar for ARINC buffers, where both wait queues relate to the same partition.

ARINC sampling ports use double buffering of the message. The sender updates one buffer and increments an atomic round counter afterwards. The reader reads the counter, checks the lowest bit whether the first or second buffer is valid, copies the message, and checks the counter again. If the counter has changed in the mean time, a new message arrived while the reader was preempted.

Using similar protocols, the ARINC library implements semaphores, events, and blackboards using a single wait queue per synchronization object.

Regarding functional safety aspects of futexes, two different cases need to be considered: futexes shared between partitions and partition-local synchronisation objects. While for the latter case tasks have to trust each other anyway (as they execute in the same address space), the former case poses a problem if one partition erroneously overwrites the shared memory containing yet unsent messages. From the security standpoint, these shared resources pose no problem as the communication channel was explicitly allowed in the configuration anyway. The only problem is that a partition can access previously sent messages in the shared memory buffer. We think that both issues are acceptable when set in contrast to the code reduction in the kernel.

VI. Evaluation

For evaluation, we present AUTOBEST's general performance characteristics (e.g., context switch time) and show the benefits of lazy priority switching and futex use for common synchronisation means compared to a system call based approach. We measure the performance on two platforms, a Freescale MPC5646C "Bolero_3M" using a 120 MHz PowerPC e200z4 core and an ARM Cortex-R4f-based Texas Instruments TMS570 processor operating at 180 MHz. Table I shows the processor architectures in detail. In general, code executes from internal flash memory and data is kept in internal SRAM.

We used GCC 4.9.1 for PowerPC and GCC 4.6.3 for ARM. We let the compiler optimize for speed with `-O2 -fomit-frame-pointer` and used function inlining where possible. Due to deterministic system behavior, repeated execution of a benchmark shows deviations of less than one percent compared to previous runs. However, *code alignment* has a greater effect on the performance by changing the number of fetched instruction blocks. Therefore, the presented results show either our measured best and worst-case results for ARM and PowerPC or a single value when no difference was detected. The numbers represent CPU cycles on the according processors.

Table I: Processor Characteristics

Parameter	**MPC5646C**	**TMS570**
Board	Freescale XDC564B-C board with MPC5646C	TI Hercules board with TMS570LS3137
CPU Core	PowerPC e200z4d	ARM Cortex-R4f
CPU Clock	120 MHz	180 MHz
Integrated Flash	3 MiB, 64-bit	3 MiB, 64-bit
Integrated SRAM	256 KiB, 64-bit	256 KiB, 64-bit
Pipeline	in order, dual-issue 5 stages, 1 ALU	in order, dual-issue 8 stages, 1 ALU
Data Cache	none	none
Instruction Cache	4 KiB, 2-way set associative 32-byte line size	none
Instruction Length	32-bit	32-bit

Table II lists the mean average of 1024 benchmark runs of various common higher-level task synchronisation calls in contrast to empty function calls and basic system calls. For context switching and preemption, we present round-trip performance of task interaction: A lower-priority task signals a higher-priority task and is immediately preempted, then the woken higher-priority task just waits, and finally execution resumes again in the lower-priority task. Lastly, the comparison of setting events to a higher-priority waiting task in the same or different partitions shows that the execution time is dominated by the partition-switch overhead, that is, reloading MPU registers.

Table III compares the performance of critical sections implementing an immediate priority ceiling protocol, and either use system calls (*sys*) or lazy priority switching (*lazy*) to achieve this. We show the *overall execution time* of a task spent in (i) a single, non-nested critical section, and (ii) two nested critical sections. Then, these critical sections are interrupted by a second task of different relative scheduling priority. Interruption is simulated by activation of the second task in the (inner) critical section. Four combinations are compared:

- an *uninterrupted* critical section where no task is activated,
- activation of a *low* priority task with a priority so low that it is never scheduled to determine task activation overhead,
- activation of a *medium* priority task which preempts the first task when it leaves its inner critical section,
- activation of a *high* priority task, which leads to immediate preemption of the first task inside the critical section.

In all combinations, the evaluation shows a performance gain of factor two or better when using lazy priority switching compared to system-call–based priority switching. With one interrupting task in a nested critical section, lazy priority switching requires at most one system call compared to four when using system calls to change scheduling priority.

Table II: System Call Performance Evaluation in CPU Cycles

Benchmarked Sequence	PPC	ARM	Description
Empty benchmark loop	2	7 – 14	Loop overhead
Empty function call	5	22 – 34	Function call and return overhead
Null system call	66 – 70	151 – 162	System call overhead
Yield in scheduler	271 – 272	317 – 328	Dry-run in scheduler, but no context switch
`TaskChain(self)`: terminate and activate caller	362	434 – 439	Reset task state and scheduler dry-run
Activation of a lower priority task	289	361 – 365	Task activation time, no scheduling
Round-trip: activation of a high priority task which terminates immediately	1162 – 1187	1222 – 1225	Context switch time: 2 system calls, 2 context switches
Round-trip: *set event* to waiting high priority task in *same* partition	1144 – 1189	1329 – 1333	Context switch time: 3 system calls, 2 context switches
Round-trip: *set event* to waiting high priority task in *different* partition	2140 – 2141	2647 – 2651	Partition switch time: 3 system calls, 2 context switches
ARINC semaphore: *signal* call at maximum value	56	91	Fast critical section w/ limit check only
ARINC semaphore: round-trip uncontended *wait* and *signal* pair	131	201	Round-trip time: 2 fast critical sections
ARINC semaphore: round-trip: *signal* waiting high priority task which *waits* again, using futexes	1713 – 1755	1758	Round-trip time: 2 critical section, 3 system calls, 2 context switches

Table III: Critical Section Performance in CPU Cycles

Critical Section (CS) Benchmark (Round-trip Time)	PPC		ARM	
	sys	lazy	sys	lazy
Uninterrupted **non-nested** CS	688	31	843	94
Activate low priority task: interruption, but no preemption	962	282	1237	470
Activate medium priority task: preemption when leaving CS	2041	1273	2135	1346
Activate high priority task: immediate preemption	1945	1253	2123	1334
Uninterrupted **nested** CS	1376	66	1748	167
Activating low priority task: interruption, but no preemption	1706	358	2118	529
Activating medium priority task: preemption when leaving inner CS	2743	1348	3016	1433
Activating high priority task: immediate preemption	2742	1340	3004	1414

VII. Discussion

With AUTOBEST, we have shown how to map both AUTOSAR and ARINC concepts to a single partitioning operating-system kernel. The evaluation results show that the performance impact of a strong protection model can be relieved by lazy priority switching. However, the costs for full memory protection are high and make up the main overhead on partition switches, as the *set event* round-trip performance comparison shows. The limited number of MPU windows on our evaluated hardware platforms simply does not allow to switch between complex protection configurations as tagged TLBs in MMU-based systems would.

Also, the unified task model comes at costs for additionally required memory: On automotive ECUs, where especially RAM is typically a short resource, our requirement of 64 Bytes per task might be to high for use-case scenarios with dozens of tasks compared to an approach leaving out ARINC's timing-related state and using just 24 Bytes. Similarly, additional RAM per partition for ready queues might be considered high. But as system integration continues and RAM sizes increase with each new ECU generation, we think this is an acceptable price to pay to integrate both worlds on a single platform.

Also, using a dedicated kernel stack per partition is a trade-off between preemptivity of a partition and latency of time partition/interrupt handling on the one hand, and memory consumption on the other. In further analysis, we need to figure out if it is possible to either get rid of the per-partition stack or reduce latencies with other means.

Regarding resource partitioning, AUTOSAR's application concept matches onto ARINC's partition, with trusted functions implemented as drivers in kernel space and all resources protected by partition-specific access tables.

Only the concept of time partitioning does not yet fit that well to AUTOSAR: the strict binding of category 2 interrupts to time partitions may increase their latency for some use cases too much. Becker et al present a promising technique [12] to allow interrupts in time partitioned systems to be handled at any time by monitoring interrupt inter-arrival and execution time and delaying the interrupt when configured budgets are exceeded. Their idea fits very well to AUTOSAR's optional timing protection facility [3], which, if employed, requires developers to specify exactly these properties (budget and inter-arrival time) for every interrupt in the system configuration. We intend to evaluate such reservation-

based scheduling techniques to weaken the time partitioning concept for automotive applications, while keeping it strict for avionics at the same time.

Related to this and the design of a driver model, techniques for sharing resources between multiple partitions, such as CAN controllers, need careful design. Herber et al present a network virtualization approach for CAN [13] to enable sharing of a physical CAN bus between controllers of different criticality levels with strong isolation requirements. On the system integration side, we need comparable concepts to partition access to hardware buses.

Independently of avionics, use of lazy priority switching to improve the performance of critical sections without compromising system safety seems worth for adaption in statically configured embedded operating systems.

This is also true for the use of futexes in such environments. Compared to Linux, that is, futex use in more dynamic systems, statical resource allocation reduces much of the complexity of the implementation. At least, futexes allow the implementation of higher-level ARINC synchronization objects in user space rather than in the kernel. In future work, we would also like to evaluate if higher level AUTOSAR communication means implemented by the AUTOSAR Run-Time Environment (RTE) can be efficiently mapped to futexes as well.

Lastly, we aim for both avionics and automotive certification for AUTOBEST. Based on previous experience in avionics certification, we tried to design the system to be easily certifiable (and testable) by using a modular architecture and keeping code and configuration separated at object level rather than using conditional compilation.

VIII. Related Work

Automotive operating systems with similar goals include microkernels from Elektrobit [14], PharOS [15], as well as the ETAS hypervisor [16]. Similarly, there are many system-software solutions for avionics, including POK [17] and XtratuM [18]. However, all these systems support either AUTOSAR or ARINC, but not both. In contrast, AUTOBEST is the first integrated platform that provides conformance to both.

For larger systems, various microkernels and hypervisors exist. On the commercial side, QNX, Green Hills' Integrity, the Wind River Hypervisor, or SYSGO's PikeOS [19] are known. Open source projects like XEN [20], NOVA [21], or the family of L4 microkernels [22] complete the picture. After all, these systems require processor support for virtual memory.

Efficient implementations to disable task preemption or for fast priority changes are known from LynxOS [23], PikeOS, and L4 [24]. Additionally, Linux's vDSO [25] and L4's user-level TCB [26] even share more scheduling related information between kernel and user space. However, these techniques just allow to disable preemption rather than giving access to the scheduling priority.

Sloth [8] schedules tasks as interrupts and delegates scheduling to the interrupt controller. Here, tasks interface the interrupt controller directly to change their scheduling priorities. Compared to AUTOBEST, this approach "turns scheduling upside down", but is also limited by the interrupt controller's scheduling capabilities, especially the number of supported priorities.

Futexes were first introduced in Linux to implement POSIX thread synchronization objects in user space [10], [11], and then later refined for real-time use cases [27], [28], and partitioning systems [29]. Again, these work target larger dynamic systems compared to AUTOBEST's smaller statically configured system.

IX. Conclusion

The avionics ARINC 653 and the automotive AUTOSAR-OS are the two most broadly adopted industry standards for real-time operating systems (RTOS). They stem from domains with originally very different requirements: While avionics engineers have to cope with strict regulations regarding functional safety, automotive engineers generally care a lot about per-unit costs in mass production. However, recent trends in automotive, like steer-by-wire and ECU consolidation, increase the demand for functional safety measures also in this domain – while the aviation industry is facing the increasing cost pressure of mass production.

In a comparison of the task models and synchronization means of AUTOSAR and ARINC, we derived a common superset for both APIs. We show that the differences between AUTOSAR and ARINC can be implemented at user level on a microkernel-based design. By combining static tailoring with futexes and lazy priority switching, we mitigate the synchronization overhead of a user-level OS abstraction compared to a system-call–based approach.

We presented the design and implementation of AUTOBEST a partitioning AUTOSAR-OS and ARINC 653 RTOS kernel that covers the requirements of both domains. AUTOBEST makes it possible to use (and combine) AUTOSAR-OS 4.1 ECC2 and ARINC 653 part 1 applications on a common platform. It enforces strict isolation of partitions in time and space with the low hardware capabilities that are provided by typical automotive ECUs: Isolation is implemented by standard (watchdog) timers and memory-protection units; our RAM requirements are moderate with less than 2.5 KiB per partition and 64 Bytes per task, considering the limited amount of 256 KiB RAM on our evaluation platforms.

Overall, we claim that with AUTOBEST it becomes possible to implement cross-domain systems with acceptable trade-offs and without compromising either domains requirements.

Acknowledgements

We especially like to thank Jochen Decker, Felix Fastnacht, André Himmighofen, Bernhard Jungk, and Tobias Jordan from Easycore GmbH for insightful discussions on AUTOSAR and the anonymous reviewers for their helpful comments on the first version of this paper.

References

[1] "ISO 26262: Road vehicles – Functional safety," ISO Norm, 2011.

[2] "DO-178B: Software Considerations in Airborne Systems and Equipment Certification," Radio Technical Commission for Aeronautics, 1992.

[3] "Specification of Operating System," AUTOSAR, March 2014.

[4] "ARINC Specification 653: Avionics Application Software Standard Interface," AEEC, November 2010.

[5] A. Wieder and B. B. Brandenburg, "On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks," in *RTSS*, 2013, pp. 45–56.

[6] "OSEK OS Extensions for Protected Applications," DaimlerChrysler AG, July 2003.

[7] A. Lackorzynski and A. Warg, "Virtual Processors as Kernel Interface," in *Twelfth Real-Time Linux Workshop*, 2010.

[8] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, "Sloth: Threads as interrupts," in *RTSS*, 2009, pp. 204–213.

[9] A. Zuepke, M. Bommert, and R. Kaiser, "Fast User Space Priority Switching," in *OSPERT Workshop*, July 2014.

[10] H. Franke, R. Russell, and M. Kirkwood, "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux," in *Proceedings of the Ottawa Linux Symposium*, 2002, pp. 479–495.

[11] U. Drepper, "Futexes Are Tricky," White Paper, Nov. 2011. [Online]. Available: http://people.redhat.com/drepper/futex.pdf

[12] M. Beckert, M. Neukirchner, R. Ernst, and S. M. Petters, "Sufficient Temporal Independence and Improved Interrupt Latencies in a Real-Time Hypervisor," in *DAC*, 2014, pp. 86:1–86:6.

[13] C. Herber, A. Richter, T. Wild, and A. Herkersdorf, "A Network Virtualization Approach for Performance Isolation in Controller Area Network (CAN)," in *RTAS*, 2014.

[14] D. Haworth, "An AUTOSAR-compatible microkernel for systems with safety-relevant components," in *Herausforderungen durch Echtzeitbetrieb - Echtzeit 2011*. Springer, 2012, pp. 11–20.

[15] C. Aussagues, D. Chabrol, V. David, D. Roux, N. Willey, A. Tournadre, and M. Graniou, "PharOS, a multicore OS ready for safety-related automotive systems: results and future prospects," in *ERTS*, 2010.

[16] D. Reinhardt and G. Morgan, "An embedded hypervisor for safety-relevant automotive E/E-systems," in *SIES*, June 2014, pp. 189–198.

[17] J. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon, "Validate, Simulate, and Implement ARINC653 Systems Using the AADL," *Ada Lett.*, vol. 29, no. 3, pp. 31–44, Nov. 2009.

[18] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "XtratuM: a Hypervisor for Safety Critical Embedded Systems," in *Eleventh Real-Time Linux Workshop*, September 2009.

[19] R. Kaiser and S. Wagner, "Evolution of the PikeOS Microkernel," in *MIKES Workshop*, January 2007.

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003.

[21] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-based Secure Virtualization Architecture," in *EuroSys*, 2010, pp. 209–222.

[22] K. Elphinstone and G. Heiser, "From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?" in *SOSP*, 2013, pp. 133–150.

[23] "LynxOS RTOS," LynuxWorks. [Online]. Available: http://www.lynuxworks.com/

[24] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards scalable multiprocessor virtual machines," in *Proc. of the 3rd Virtual Machine Research and Technology Symposium*, May 2004, pp. 43–56.

[25] "vDSO - overview of the virtual ELF dynamic shared object." [Online]. Available: http://man7.org/linux/man-pages/man7/vdso.7.html

[26] J. Liedtke and H. Wenske, "Lazy process switching," in *HOTOS*, 2001.

[27] D. Hart and D. Guniguntalay, "Requeue-PI: Making Glibc Condvars PI-Aware," in *Eleventh Real-Time Linux Workshop*, 2009, pp. 215–227.

[28] R. Spliet, M. Vanga, B. Brandenburg, and S. Dziadek, "Fast on Average, Predictable in the Worst Case: Exploring Real-Time Futexes in LITMUSRT," in *RTSS*, 2014.

[29] A. Zuepke, "Deterministic Fast User Space Synchronisation," in *OSPERT Workshop*, July 2013.