Overcoming the Interoperability Barrier in Mixed-Criticality Systems

Jörn Schneider[1]

Abstract Concurrent engineering of system parts with diverging requirements can be extremely challenging. One example are mixed-criticality systems that integrate hard real-time software for safety-critical functionality and general purpose software providing a sophisticated user interface. The automotive industry, as well as other industrial branches, has a growing need to integrate consumer electronics applications (e.g. Linux based) and safety-relevant applications requiring an underlying hard real-time operating system. Some established concepts for mixed-criticality systems can be found in the avionics domain. This paper demonstrates that the principles behind these concepts are a dead end regarding innovations requiring a close interoperation. The second contribution of the paper is to present a different solution approach as a potential remedy that allows the different developer groups (hard real-time and standard IT) to retain their attitude to software development. The core of the novel approach is a worst-case execution time (WCET) directed OS service, which could serve as solution pattern for further problems in mixed-criticality systems.

Keywords

Mixed-Criticality; Cyper-Physical Systems; Real-Time; Interoperability; Worst-Case Execution Time; Integration; Virtualization; AUTOSAR; Linux

1 Introduction

Industry has a growing demand for integrating applications with diverging real-time and criticality requirements on the same microcontroller. Among these future mixed-criticality systems is a class of applications for which uniprocessor solutions are sufficient, but which are implemented on separate processors in current

[1] J. Schneider (✉)
Trier University of Applied Sciences, Dept. of Computer Science, Schneidershof, 54293 Trier, Germany
e-mail: j.schneider@fh-trier.de

J. Stjepandić et al. (eds.), *Concurrent Engineering Approaches for Sustainable Product Development in a Multi-Disciplinary Environment*,
DOI: 10.1007/978-1-4471-4426-7_92,

products. A driving force for mixed-criticality systems is the need to achieve a much closer coupling between applications of different criticality to allow for further innovations. Up to now the increased demand for interoperability is usually satisfied by transferring messages in a distributed system. However, the costs for providing one hardware platform per functionality are no longer acceptable[2]. This work focusses on the question how resources, e.g. peripheral devices or memory areas can be safely and efficiently shared in future systems.

Examples of mixed-criticality systems are especially known from avionics industry where the approach referred to as Integrated Modular Avionics (IMA) and the underlying ARINC-653 standard [1] particularly targets this type of applications.

Another area where mixed-criticality systems will evolve further is the automotive industry, and the proposed approach especially targets this area. In-car electronic control units like instrument clusters and head units need to provide non, or soft real-time behavior for displaying information or audio and video streaming. In addition they have to work in the firm real-time communication of the car and must direct to the driver whatever mission or safety critical messages they receive. Traditionally, these systems have a dedicated microcontroller to handle the in-car communication. Eliminating this extra hardware promises reduced costs and new functionality by a tighter coupling of the classic automotive domain with in-vehicle infotainment applications, e.g. to provide new classes of driver assistance systems. For these cases it is of special interest to run Linux based software together with AUTOSAR [2] applications on one CPU. Ambitious activities like the GENIVI Alliance [3] demonstrate the importance of an increased interoperability for industry.

The basic conflict each development team faces when building mixed-criticality systems is independent of the industrial application domain. Integrating applications on top of the same platform requires the art of limiting the mutual influence while interacting and sharing resources. The situation is exacerbated by having different levels of criticality and development teams with different backgrounds. Developers of consumer electronics applications usually do not know how to develop hard real-time systems and specialist for safety-critical or real-time systems typically have to use quite different approaches than standard IT developers.

In mixed-criticality systems it is necessary to separate applications with different levels of criticality such that they cannot affect each other in a more than acceptable degree. Therefore, applications are usually assigned to partitions that limit their influence sphere. Different partitions are usually protected against each other in mixed-criticality system by separating them regarding space, i.e. memory, and regarding timing. As this paper concentrates on the real-time aspects, the important issues to control are:

[2] Consider the automotive industry for example, where a luxury car might have up to 100 electronic control units.

- At which points in time is a given partition allowed to obtain a shared resource?
- For which amount of time is a partition able to block out other partitions from accessing a particular resource?

Usually there are no dedicated mechanisms to control these issues in soft or non real-time systems. For hard real-time systems well-known approaches are available such as the priority ceiling protocol (PCP) by Sha, Rajkumar, and Lehoczky [4], and the Stack Resource Policy (SRP) by Baker [5]. However, these mechanisms are not suited to be used across partitions of mixed-criticality systems, as this would require one common OS for all involved application types.

The question that arises here is: How can resource sharing in mixed-criticality systems work without intolerable temporal influence? In Section 2 we will see that the way existing approaches achieve this is by introducing one or more levels of indirection, such that one dedicated partition actually accesses the shared resource alone and serves the usage requests of the other partitions.

A disadvantage of this approach is that the usual way devices are accessed is no longer working for partitions that are fenced off from the actual resource. As a consequence the combination of off-the-shelf applications (including device drivers and general purpose operating systems such as Linux) with hard real-time applications to mixed-criticality systems requires tedious redesigns, leads to redundant system structures, a larger code base, and therefore adds costs, complexity, and faults. The approach presented in this paper allows for immediate sharing of resources between a hard real-time and a soft/non real-time partition, while still limiting the temporal influence reliably.

2 Current Approaches

This section describes the technological concept of current mixed-criticality systems regarding time partitioning, where the description focusses on ARINC-653 based approaches [1, 6, 7]. The three issues to consider here are time partitioning regarding processor time, communication, and shared devices.

Each set of applications exhibiting the same criticality level, is confined into a *partition*. The goal of this confinement regarding the temporal behavior is to prevent that any partition gets more time with a resource than its allowed share.

Fig. 1 Scheduling of partitions in current mixed-criticality systems

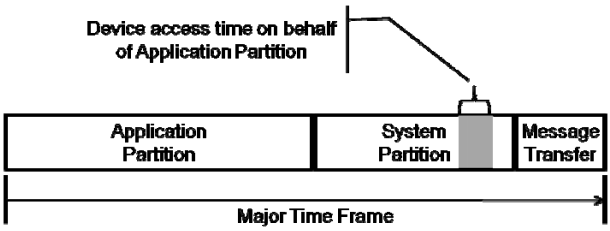

Fig. 2 Temporal effects regarding device accesses in current systems

Processor[3] Current mixed-criticality systems usually come with a static schedule of processor time slots for partitions as shown in Figure 1. A periodically repeated major time frame is divided between partitions such that each partition receives at least one time slot per major time frame.

To guarantee that no partition receives more processor time than allowed it is sufficient to ensure that no partition can overdraw its time slots. Apart from this it is necessary to find a static schedule that fulfills all real-time requirements of any involved partition. Each partition usually executes its own operating system. The underlying time slot scheduling, is under control of a separate operating system, which is referred to as partition manager throughout the paper. When referring to the partition private operating system, the term guest operating system is used.

Communication Inter-partition communication in current mixed-criticality systems is usually limited to message transfers. Whenever a partition needs to communicate with another partition, the relevant parameters (e.g. message length, and frequency of communication) have to be statically configured. The partition manager copies the data to be communicated when switching between partitions. For instance, messages produced by a partition might be copied only at the end of the major time frame and are afterwards available for any receiving partition.

Regarding communication the confinement goal is achieved by preallocating sufficient time to transmit messages outside of the application partitions time slots. As the maximum amount of data and thereby the time needed to transfer messages between the separated memory spaces of partitions is limited, no partition can receive more communication time than planned. The communication schedule and therefore also the processor schedule have to be constructed to satisfy the real-time communication requirements of all involved partitions. A drawback of message based communication is that it imposes a cumbersome programming style and less resource efficient code.

Devices Devices are typically not immediately shared in current mixed-criticality systems. Instead any device that is needed by more than one partition is assigned to a so-called system partition which serves any usage request of application partitions [6, 7]. The usage requests are transferred via message transmission across the

[3] Note that single processor mixed-criticality systems are considered in this paper.

usual communication channel. When using this approach to integrate previously separated systems on the same processor, it enforces message based communication within formerly coherent communication chains from application code to physical devices. As shown below, this is the root of many problems.

Handling device accesses on behalf of low criticality application partitions prolongs the time slots of system partitions as shown in Figure 2. However, as the maximum length of messages and the communication frequency are statically known, the time of using a device on behalf of a partition is bound. That means that the confinement goal to limit the device usage time for each partition to its allowed share, can be achieved. Nevertheless, further problems arise with this concept.

Consider a low criticality non real-time application with a device access behavior as shown in Listing 1.

Listing 1 Possible device accesses of a low criticality application

```
while true // Do this forever
     read from device X
     compute values
     write to device Y
```

If this application has to be incorporated in a mixed-criticality system with shared devices, all device accesses have to be rerouted to a system partition via message transmission. In case of a synchronous message passing mechanism the considered application process would be blocked at each device access. Moreover, together with the scheduling configuration of Figure 2 it would suffer from a delay of at least two major time frames before being unblocked and the transmission latency between process and device would be at least one major time frame. Note that asynchronous message passing is no sensible option in this case, as the code does not allow to utilize the time before the read access to the device is completed.

What happens to high criticality applications with hard real-time constraints in systems that follow the message passing paradigm? Of course the real-time requirements regarding the communication with devices and externally connected components such as sensors and actuators impose even more constraints on the static schedule to be found. For instance the latencies for sensor readings or actuator positioning are determined by the scheduling of partitions.

The above described problems cause that the otherwise appealing concept of system partitions and message based access to shared devices enforces major changes to the device drivers and the application code, at least for low criticality partitions. This is a painful disadvantage for industrial needs. The communication path between devices and application code has to be redesigned. New device driver layers are necessary that add no functionality but are just needed to communicate with the system partitions that actually access the particular devices. Moreover the usual interrupt based device driver concept is no longer applicable. And as the example of Listing 1 indicates, it is very likely that the application itself has to

be adjusted. Essentially the same issues arise if memory areas shall be shared, e.g. to use or validate computation results from a different partition.

These drawbacks of the prevalent concept of device sharing in mixed-criticality systems and the lack of memory sharing possibilities lead to the question: *How could a less indirect but still safe resource sharing scheme in mixed-criticality systems be realized?* An answer to this question, for the restricted case of two partitions, is given in the remaining sections of this paper.

3 System Model

The systems under consideration comprise a soft or non real-time part, a hard real-time part and a common, underlying partition manager. The hard real-time part is considered to have a higher level of criticality than the soft or non real-time application. The complete software is assumed to run on a single processor.

For the sake of space, the abbreviations *NRT* and *HRT* are used throughout the paper to name the soft or non real-time part, and the hard real-time part, respectively. The NRT and HRT parts have their own OS. The partition manager switches between the NRT and HRT part according to a static schedule to guarantee a certain amount of computation time for both. Thus at any given point in time the system is either in the NRT phase, the HRT phase, or from the partition manager point of view in a transition between two phases.

The NRT OS can use any scheduling scheme, as long as it is possible to ensure that a process cannot be preempted for a certain section of its code. For instance the NRT OS could be an embedded Linux with the ability to run a process solitarily in a high priority class.

The HRT OS can use any real-time scheduling scheme, e.g. rate monotonic, or earliest deadline first. The scheduling objects of the HRT OS are called *tasks* throughout the paper, to distinguish them from NRT OS scheduling objects, for which the term *processes* is used.

4 Guidelines and Requirements

This section first presents a set of guidelines for the development of an improved solution arising from industrial needs. Thereafter three concrete requirements on the solution are derived.

4.1 Guidelines for improved solution

Immediate sharing It shall be possible to access shared resources directly from the HRT and the NRT partition. This gives the freedom to avoid message based communication and its drawbacks.

Consistent use A resource that is currently in use by the HRT or NRT partition may be granted to the other partition only when it is safe to do so. This implies that access to a shared resource cannot be allowed, if its state is inconsistent.

Preserve timing budgets The duration of time slots may never be overdrawn due to ongoing resource accesses. This guarantees that each partition receives its pre-configured amount of time. Note that this still allows that a resource is held for more than one time slot.

Predictable duration of resource accesses An upper bound of the holding time of a resource shall be computable from configuration information and the code sections performing the actual access. Thus the worst case impact of resource sharing can be considered at design time and essential temporal properties of the system can be verified. For instance it is possible to check whether an NRT application never suffers from starvation due to resource sharing.

Freedom from deadlocks The new approach should guarantee that no possibilities for the occurrence of deadlocks are introduced. For the HRT application, deadlocks would lead to deadline misses and for the NRT part there should at least be no additional sources of potential deadlocks.

Zero impact on HRT application There shall be no influence on the timing behavior of the HRT application due to NRT resource accesses. This allows to verify the real-time behavior of the HRT part without considering the NRT part. Especially, no changes of the NRT application can invalidate timing verification results for the higher criticality application. Thereby, it becomes possible to get the HRT system part certified, e.g. according to [8], without rendering the certification invalid due to later changes of the NRT part.

Allow for oblivious development The developers of the NRT application should not need to understand the details of the temporal behavior of the HRT application in order to use shared resources. The same should hold the other way around for the HRT programmers.

No guest OS changes The new concept shall work without significant changes to the HRT or the NRT operating system. Especially the scheduling mechanisms should not be changed. Thus the concept can be implemented without changing the operating system code base and even if the operating system source code is not available.

4.2 Requirements on solution mechanism

Basing on the described guidelines it is possible to derive a set of requirements on a concrete solution mechanism. One can take the consistent and immediate access to shared resources as starting point. Apparently, a mutual exclusion mechanism between HRT tasks and NRT processes (or more precise between certain code sections of them) would suit the problem. As the guest operating systems should remain unchanged, this service should be placed within the partition manager and for the sake of access time predictability there should be no interference by uninvolved tasks or processes. Because the NRT part shall not influence the temporal behavior of the HRT application, the blocking time for HRT tasks requesting this service has to be zero.

Explicitly stated these requirements are:

R1: A mechanism for explicit mutual exclusion between HRT tasks and NRT processes shall be provided by the partition manager without changes to the guest operating systems.

R2: The mechanism shall prevent that the duration of critical sections is prolonged by uninvolved tasks or processes.

R3: It shall be *impossible* for an HRT process to be blocked when it requests exclusive access to a shared resource.

5 Solution Approach

The basic idea of the proposed solution is to incorporate a mechanism into the partition manager that behaves like an ordinary mutex on the HRT side and grants access to a free mutex for an NRT process only if it is guaranteed that the lock is released before the current NRT time slot ends. As one can easily see, the latter property is sufficient to satisfy Requirement R3, as far as blocking by NRT processes is concerned.

To realize the idea, the partition manager has to provide a pair of API calls to acquire and release a mutex of the new type. For convenience let us name it a mixed-criticality lock, short *MCL* and the API calls `GetMCL()` and `ReleaseMCL()`. In truth there are two version of the two API calls, one for usage in NRT applications and one for the HRT side. As shown below, it is sensible to impose some restrictions on the usage of these API calls.

5.1 Calling conventions

There are some conventions that need to be observed when using the MCL API calls of the partition manager. This rules for using the MCL mechanism are given as restrictions below. Thereafter, the rationales behind the restrictions are explained.

5.1.1 Restrictions

Restriction 1: No task or process may hold more than one MCL at a time (i.e. nesting of MCLs is forbidden).

Restriction 2: No guest OS system calls that might cause rescheduling are allowed in code sections embraced by `GetMCL()` and `ReleaseMCL()`.

Restriction 3: No two scheduling objects of the same class (i.e. any two NRT processes or any two HRT tasks, respectively) can use the same MCL unless any accesses to the latter are protected by creating a native critical section around them[4].

5.1.2 Why are these restrictions needed?

Restriction 1 prevents that any task or process allocates more than one MCL at a time. This is a straightforward way to prevent deadlocks that might otherwise be caused by circular waiting on MCLs and serves to heed the freedom from deadlocks guideline. Of course, this could be achieved by other means also. For instance, restricting the allocation sequence of MCLs to a preconfigured order would work as well. However, such an approach would end up in a more complex algorithm and a less lean partition manager.

Restriction 2 limits the usage of system calls while an MCL is held. It disallows system calls that could cause the guest operating system to pick a different scheduling object (task or process) to be executed. This helps to realize the guideline of predictable resource access duration by contributing to the fulfillment of Requirement R2. Moreover it reduces the worst case holding time of MCLs.

Restriction 3 limits the usage of the same MCL by scheduling objects of the same guest operating system. Only if the MCL system calls are protected in a way that prevents multiple attempts to receive the same MCL, two or more scheduling objects of the same partition can utilize the same MCL. Thereby the partition manager does not need to care about managing a list of unfulfilled requests for the same MCL. Moreover, there is no point in reinventing the wheel, as the guest op-

[4] A native critical section is a critical section protected by the usual mechanisms of the guest OS.

erating systems can be expected to provide mechanisms to guarantee mutual exclusion between their native scheduling objects. A more severe reason is that including special support to share MCLs among application code of the same guest OS could lead to deadlocks in combination with native mechanisms, and on the HRT side it might violate Requirement R3.

5.2 Algorithm of NRT API calls

The basic steps for acquiring an MCL in NRT processes are as follows:

`GetMCL()`

1. Raise priority of calling process to the highest priority class (or prevent preemptions by other means). *Note that this, together with Restriction 2, guarantees that the worst case duration of the critical section is determined solely by itself. The guideline predictable resource access duration is thus realized.*
2. Test whether the lock is free and if the remaining duration of the current NRT phase suffices to complete the critical section of the requesting process[5].
    a) If the lock is free and time suffices to complete the critical section in the current NRT phase, the lock is granted to the caller.
    b) If the lock is occupied or remaining phase time is insufficient, the calling process is put to sleep and registered in a partition manager internal FIFO ordered list that holds all pending MCL requests of NRT processes. *Note that at most one process can be waiting for the same MCL at any point of time. This is guaranteed due to Restriction 3.*

Releasing an MCL in NRT processes works as follows:

`ReleaseMCL()`

1. Release the lock.
2. Resume the former priority of the calling process (or return to preemptive mode by other means).
3. If the waiting queue is not empty, test in FIFO order whether a pending request can be fulfilled, i.e. whether the requested lock is free and sufficient time in the current NRT phase is available. The first fulfillable request, if any, is then granted.

[5] Note that the duration of each NRT phase is assumed to be statically known. This can be achieved by using a worst case execution time (WCET) analyzer [9], provided the critical sections contain no endless loops.

Naturally, both system calls have to be performed atomically within the partition manager.

5.3 Further solution parts

The further parts of the provided mechanism need to be established in the transition from the HRT phase to the NRT phase, and in the API calls used by the HRT application. Both are parts of the partition manager and both are quite simple.

Whenever the partition manager switches from HRT phase to NRT phase it has to awake the first process in its FIFO list with a fulfillable pending request for an MCL, if any such process exists. For a request to be fulfillable the same condition has to hold as always for NRT processes, i.e. the MCL has to be free and the remaining duration of the NRT phase must suffice to complete the critical section. The latter part of the condition can be tested by comparing an, at build time configured, worst case duration of the individual critical section with the remaining time to the phase switch. Note that the occurrence of phase switches is also statically predetermined, as the time slots are known.

The last part of the mechanism consists of the HRT version of the API call pair. Since, as one can easily conclude, any request by an HRT task to an MCL can immediately be granted, `GetMCL()` in its HRT version just allocates the MCL for the calling task.[6] The HRT version of `ReleaseMCL()` is also straightforward, as it just releases the MCL. In addition to this the `GetMCL()` and `ReleaseMCL()` calls have to establish a non-preemptable code section between themselves, to satisfy Requirement R2.

6 Conclusions

As shown in Section 2, the prevalent concepts for designing the system software of mixed-criticality applications induces serious limitations for the concurrent engineering of such systems. Any application developer that uses shared resources has to consider the performance effects by using a system partition as proxy. Because there is an order of magnitude between the time needed to access a device or memory region natively and via the system partition the applications themselves have to be redesigned. From a systems engineering point of view the prevalent approach has a severe problem. It lacks a separation of the two concerns computa-

[6] In a practical implementation sanity checks might be performed in addition, in order to assert correct behavior and provide fault-tolerance mechanisms where appropriate. Clearly, this holds also for the other parts of the MCL mechanism.

tion time allocation and communication latencies. The time to access a sensor or an actuator depends inherently on the underlying scheduling of partitions.

The presented novel approach eliminates these problems by allowing a direct access to shared resources while maintaining the essential properties. One important issue for mixed-criticality systems is the ability of incremental certification (i.e. changes made to one system part do not require the recertification of others). Regarding timing and from a technical point of view this requires that the temporal requirements of all system parts are still satisfied. It is immediately clear that this property is given by construction for the MCL approach as far as the requirements of the HRT part are concerned.

The developers of safety-related hard real-time applications on one side and those of soft or non real-time applications on the other side usually have different experience backgrounds and different programming styles. Many solutions for mixed-criticality systems have been proposed that imply that all system parts are more or less developed according to one and the same philosophy. These concepts might work in settings where the ”cultural chasm” between the developers of the system parts is narrower, but seems unlikely to succeed in scenarios as considered here. The novel concept does not impose particular development styles or philosophies.

7 References

1. Avionics application software standard interface, 1 1997. ARINC Report 653.
2. http://www.autosar.org/. Cited 15 May 2012
3. http://www.genivi.org/. Cited 15 May 2012
4. Sha L, Rajkumar R, Lehoczky JP (1990) Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans. Comput., 39(9):1175–1185, 1990.
5. Baker TP (1991) Stack-based scheduling of real-time processes. Real-Time Systems, 3(1):67–99, 1991
6. Windsor J, Hjortnaes K (2009) Time and space partitioning in spacecraft avionics. Space Mission Challenges for Information Technology, IEEE International Conference on, 0:13–20,
7. Craveiro J, Rufino J, Almeida C, Covelo R, Venda P (2009) Embedded linux in a partitioned architecture for aerospace applications. Computer Systems and Applications, ACS/IEEE International Conference on, 0:132–138, 2009.
8. Road vehicles – Functional safety, 12 2010. ISO/FDIS 26262.
9. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P (2008) The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst., 7(3):1–53, 2008.