e-Informatica Software Engineering Journal, Volume 12, Issue 1, 2018, pages: 305–320, DOI 10.5277/e-Inf180112

Experience Report: Towards Extending an OSEK-Compliant RTOS with Mixed Criticality Support

Tarun Gupta*, Erik J. Luit**, Martijn M.H.P. van den Heuvel**, Reinder J. Bril**
*Sioux Embedded Systems B.V.
**Mathematics and Computer Science, Technische Universiteit Eindhoven (TU/e)
tarun.gupta@sioux.eu, e.j.luit@tue.nl, m.m.h.p.v.d.heuvel@tue.nl, r.j.bril@tue.nl

Abstract

Background: With an increase of the number of features in a vehicle, the computational requirements also increase, and vehicles may contain up to 100 Electronic Control Units (ECUs) to accommodate these requirements. For cost-effectiveness reasons, amongst others, it is considered desirable to limit the growth of, or preferably reduce, the number of ECUs. To that end, mixed criticality is a promising approach that received a lot of attention in the literature, primarily from a theoretical perspective.

Aim: In this paper, we address mixed criticality from a practical perspective. Our prime goal is to extend an OSEK-compliant real-time operating system (RTOS) with mixed criticality support, enabling such support in the automotive domain. In addition, we aim at a system (*i*) supporting more than two criticality levels; (*ii*) with minimal overhead upon an increase of the so-called criticality level indicator of the system; (*iii*) requiring no changes to an underlying operating system; and (*iv*) featuring further extensions, such as hierarchical scheduling and multi-core.

Method: We used the so-called adaptive mixed criticality (AMC) scheme as a starting point for mixed criticality. We extended that scheme from two to more than two criticality levels (satisfying (*i*)) and complemented it with specified behavior for criticality level changes. We baptized our extended scheme AMC*. Rather than selecting a specific OSEK-compliant RTOS, we selected ExSched, an operating system independent external CPU scheduler framework for real-time systems, which requires no modifications to the original operating system source code (satisfying (*iii*)) and features further extensions (satisfying (*iv*)).

Results: Although we managed to build a functional prototype of our system, our experience with ExSched made us decide to rebuild the system with a specific OSEK-compliant RTOS, being μC/OS-II. We also briefly report upon our experience with AMC* and suggest directions for improvements.

Conclusions: Compared to extending ExSched with AMC*, extending μC/OS-II turned out to be straightforward. Although we now have a basic system operational and available for experimentation, enhancements of the AMC*-scheme are considered desirable before exploitation in a vehicle.

Keywords: OSEK, RTOS, mixed criticality

1. Introduction

A growing trend in the automotive domain is a feature intensive vehicle. These features may be safety related, driver assistance related, connected services, or multimedia and entertainment related. With an increase of the number of features, the computational requirements also increase. Nowadays, a vehicle may be controlled by over 100 million lines of code, that are executed on up to 100 Electronic Control Units (ECUs) [1]. For reasons of cost, space, weight, and power con-

sumption, amongst others, adding more ECUs is undesirable. Instead, even a reduction of ECUs is preferred, with appropriate means for (temporal and spatial) isolation between applications, efficient and effective resource management, assurance against failure, and graceful degradation upon overloads. Given the distinct criticality levels of these features, e.g. safety critical, mission critical, and low-critical, application of mixed criticality theory and practice [2] may be beneficial. Within the context of the i-GAME [3] and EMC^2 projects[1], we therefore explored the option to apply mixed criticality. Whereas there exists an overwhelming number of papers on mixed criticality, the majority addresses theoretical aspects, with a focus on schedulability analysis. Although some address implementation aspects, such as [4], only a few present actual implementations extending an operating system, such as [5–7]. None of these aim at the automotive domain, however, which is the main focus of this paper.

In this paper, we report upon our initial efforts to extend an OSEK-compliant [8] real-time operating system (RTOS) for a single-core with support for mixed criticality. In addition, we aim at a system (*i*) supporting more that two criticality levels; (*ii*) with minimal overhead upon an increase of the so-called criticality level indicator of the system; (*iii*) requiring no changes to an underlying operating system; and (*iv*) featuring further extensions, such as hierarchical scheduling and multi-core.

For our mixed criticality scheme, we selected an existing scheme, Adaptive Mixed Criticality (AMC) [9], as a basis. We extended the scheme from two criticality levels to multiple criticality levels (satisfying (*i*)), and complemented it with specified behavior upon criticality level changes. Rather than selecting a specific OSEK-compliant operating system, we selected ExSched [10]. ExSched is an operating system independent external CPU scheduler framework for real-time systems, which requires no patches (i.e. modifications) to the original operating system source code (satisfying (*iii*)), unlike, for example LITMUS$^{\mathrm{RT}}$ [11] and AQUOSA [12], making it easier to update to newer kernel versions. Moreover, ExSched supports multiple operating systems, in particular Linux and VxWorks, and comes with hierarchical and multi-core schedulers (satisfying (*iv*)), amongst others. In our initial experiments, we used ExSched in combination with Linux version 2.6.36, which we downloaded from [13], on an Intel Core I5 processor. We intended to subsequently develop support of ExSched to support an OSEK-compliant RTOS. Although we managed to build a functional prototype of our system, we decided to abandon ExSched, however, based on our experiences with and insights gained during the extension of ExSched with mixed criticality support. Our subsequent experiments therefore concerned the move from ExSched with Linux towards an OSEK-compliant RTOS, in particular μC/OS-II [14]. We used μC/OS-II in combination with RELTEQ (Relative Timed Event Queues) [1], which supports hierarchical scheduling (*iv*), amongst others. Our final contribution concerns a reflection on AMC*.

This journal paper is an extended version of a workshop paper [15]. Compared to [15], this extended version has the following two major contributions. Firstly, it presents the extension of μC/OS-II with mixed criticality (Section 6.2). Secondly, it presents the experience with and the evaluation of AMC*, including improvements of the scheme (Section 7).

The remainder of this paper is organized as follows. We start by a brief discussion of related work in Section 2. Next, in Section 3, we present our real-time scheduling model and a brief recapitulation of ExSched. Our extended AMC scheme, baptized AMC*, is the topic of Section 4. Extending ExSched with mixed criticality support is addressed in Section 5. The move from ExSched with Linux to μC/OS-II is addressed in Section 6. In Section 7, we reflect on AMC*. The paper is concluded in Section 8.

[1]The work presented in this paper was funded in part by the EU 7^{th} framework programme through the i-GAME (Interoperable GCDC AutoMation Experience) project (grant agreement 612035) and the ARTEMIS Joint Undertaking EMC^2 project (grant agreement 621429).

2. Related work

There exists a plethora of papers on mixed criticality systems; see [2] for a review. Here, we focus on two specific mixed criticality aspects, being mixed criticality schemes and actual implementations extending an operating system, and briefly discuss existing support of OSEK-compliant RTOSs.

Building upon the seminal work of Vestal [16], which was the first paper addressing schedulability analysis for mixed criticality systems given a basic mixed criticality scheme, a lot of theoretical work has been done on mixed criticality systems. The scheme presented by Vestal consists of an ordered set of four criticality levels. At any moment of time, the system is running at a particular criticality level. The scheme describes the cause (i.e triggering event) and behavior of a criticality level up, i.e. when the system makes the transition from a lower to a higher criticality level, but lacks a description of a criticality level down. This initial scheme was later refined and the schedulability of mixed criticality systems improved by Baruah et al. in [4, 9, 17], amongst others. Although the restriction on the number of criticality levels is lifted in these later works, the description of the latest scheme [9] called adaptive mixed criticality (AMC) and its analysis has been restricted to two levels for simplicity. The cause and behavior of a criticality level down was first described in [4]. The AMC scheme was later relaxed in [18] at the cost of increased implementation complexity. In this paper, we therefore selected AMC as a basis. For a detailed comparison of the schemes mentioned above, the interested reader is referred to [19, 20].

Whereas a lot of papers address theoretical aspects, only a few papers describe actual implementations extending an operating system with mixed criticality support, such as [5–7]. Kim et al. [6] studied the actual implementation of a criticality level change in the RTOS eCOS [21], with the aim to minimize the scheduler overheads. They assume a mixed criticality scheme with two criticality levels. Herman et al. [5] describe RTOS support for multi-core mixed criticality systems, using the academic RTOS LITMUS$^{\mathrm{RT}}$ [11], an extension to the Linux kernel. The number of criticality levels assumed in that work is four. Kritikakou et al. [7] describe support for multi-core mixed criticality systems using their own developed bare-metal library [22]. In their model, only a single task of a high criticality level is assumed. To the best of our knowledge, there does not exist an OSEK-compliant [8] RTOS with an extension for mixed criticality, which is the focus of this paper.

There exist many OSEK-compliant RTOSs, such as ETAS RTA-OSEK[2], μC/OS-II [14], and Erika Enterprise RTOS[3]. The specification of the OSEK operating system [8] explicitly states that the "*OSEK operating system is a single processor operating system meant for distributed embedded control units*". An OSEK-compliant RTOS may therefore provide support for hierarchical scheduling and multi-core, but need not provide such support. As examples, both ETAS RTA-OSEK and μC/OS-II provide neither hierarchical scheduling nor multi-core support, whereas Erika Enterprise RTOS only provides multi-core support. An extension of μC/OS-II with hierarchical scheduling has been described in [23]. To the best of our knowledge, there does not exist an OSEK-compliant RTOS providing support for both hierarchical scheduling and multi-core. In this paper, we focus on μC/OS-II, because we gained significant experience with that RTOS over the past years [24–26].

3. Preliminaries

In this section, we present our real-time scheduling model in Subsection 3.1 and a brief recap of ExSched in Subsection 3.2.

3.1. Real-time scheduling model

After presenting a basic real-time scheduling model for fixed-priority pre-emptive scheduling (FPPS), we extend the model with mixed criticality conform AMC [9].

[2]Details about ETAS RTA-OSEK can be found at http://www.etas.com.
[3]Details about Erika Enterprise OS can be found at http://www.tuxfamily.org.

3.1.1. Basic model for FPPS

We assume a single processor and a set $\mathcal{T}$ of n independent sporadic tasks τ_1, τ_2, ..., τ_n, with unique priorities π_1, π_2, ..., π_n. At any moment in time, the processor is used to execute the highest priority task that has work pending. For notational convenience, we assume that (i) tasks are given in order of decreasing priorities, i.e. τ_1 has the highest and τ_n the lowest priority, and (ii) a higher priority is represented by a higher value, i.e. $\pi_1 > \pi_2 > \cdots > \pi_n$.

Each task τ_i is characterized by a *minimum inter-activation time* $T_i \in \mathbb{R}^+$, a *worst-case computation time* $C_i \in \mathbb{R}^+$, and a (*relative*) *deadline* $D_i \in \mathbb{R}^+$. We assume that the constant pre-emption costs, such as context switches, are subsumed into the worst-case computation times. We assume constrained deadlines, i.e. the deadline D_i may be smaller than or equal to period T_i. The *utilization* U_i of task τ_i is given by C_i/T_i, and the *utilization* U of the set of tasks $\mathcal{T}$ by $\sum_{1\leq i\leq n} U_i$.

We also adopt standard basic assumptions [27], i.e. tasks do not suspend themselves and a job does not start before its previous job is completed.

3.1.2. Extended model for mixed criticality

We assume a set $\mathcal{L}$ of m criticality levels[4] Λ_1, Λ_2, ..., Λ_m. For notational convenience, we assume that (i) criticality levels are given in order of decreasing criticality, i.e. Λ_1 represents highest and Λ_m represents lowest criticality, and (ii) a higher criticality level is represented by a higher value, i.e. $\Lambda_1 > \Lambda_2 > \ldots > \Lambda_m$.

Each task τ_i has a particular criticality level $\lambda_i \in \mathcal{L}$, termed its *representative* criticality level. We now define subsets $\mathcal{T}^\Lambda$ of $\mathcal{T}$, i.e.

$$\mathcal{T}^\Lambda \stackrel{\text{def}}{=} \{\tau_i | \lambda_i \geq \Lambda\} \tag{1}$$

When the system is executing at criticality level Λ, i.e. the criticality level indicator Γ is equal to Λ, the processor is used to execute only tasks in the subset $\mathcal{T}^\Lambda$.

Moreover, the worst-case computation time of a task τ_i becomes a vector $\vec{C}_i$ indexed by criticality level. These computation times are monotonically non-decreasing for increasing criticality levels, i.e.

$$\Lambda_k \leq \Lambda_\ell \leq \lambda_i \Rightarrow \vec{C}_i(\Lambda_k) \leq \vec{C}_i(\Lambda_\ell) \tag{2}$$

The actual execution time of the current job of task τ_i at time t is denoted by $\beta_i(t)$.

The following condition defines when a criticality level up occurs[5].

Condition 1. *When a job of task τ_i is executing at time t while the system is running at level Λ with $\Lambda \leq \lambda_i < \Lambda_1$ and the actual execution time $\beta_i(t)$ equals the worst-case computation time $\vec{C}_i(\Lambda)$ of τ_i, a criticality level up occurs.*

A criticality level down occurs upon a so-called criticality level Λ idle time.

Definition 1. *A criticality level Λ idle time is an instant at which there is no pending load of the tasks in $\mathcal{T}^\Lambda$.*

Intuitively, a task has *pending load* [28] larger than zero at time t when it has been activated strictly before time t and did not complete yet at time t.

Condition 2. *Upon a criticality level Λ idle time with $\Lambda > \Lambda_m$ a level down change occurs.*

3.2. Recapitulation of ExSched

The ExSched framework [10] is a loadable Linux kernel module and an extension of the REal-time SCHeduler (RESCH) framework [29]. Figure 1 shows the structural components of ExSched. An application uses the ExSched APIs provided by the ExSched Library in user space to communicate with the main ExSched Module in kernel space. This communication takes place via the ioctl () system call, i.e. ExSched is built as a character-device module. The ExSched framework supports development of plug-ins with the help of callback functions. Plug-ins for hierarchi-

[4] In [9], a so-called *dual-criticality system* is assumed, i.e. $m = 2$. In this paper, we assume more than 2 criticality levels.

[5] The AMC scheme assumes that a criticality level up is handled instantaneously.

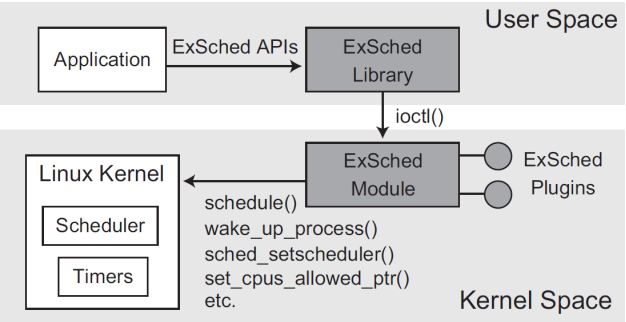

Figure 1. ExSched: structural components [10]

cal scheduling and multi-core scheduling are part of ExSched's release.

The ExSched Library provides methods to

- register/de-register tasks: rt_init() and rt_exit();
- set parameters of tasks: rt_set_wcet(), rt_set_period(), rt_set_deadline(), and rt_set_- priority();
- start a task: rt_run(); and
- activate a next job, i.e. wait (sleep) until the next period: rt_wait_for_period().

The ExSched Module uses the POSIX-compliant SCHED_FIFO scheduling policy provided by the Linux kernel. The module maintains its own task structure, which extends the encapsulated Linux task structure with additional timing parameters provided through the ExSched Library. The ExSched Module provides a dedicated interface to install and un-install a plug-in; see Table 1. Only one plug-in can be installed in ExSched at the time.

4. AMC* scheme

In Subsection 3.1.2, a general model for mixed criticality has been given, leaving specific details unspecified, such as (*i*) what happens when a task τ_i exceeds its worst-case computation time at its representative criticality level, (*ii*) what will be the new criticality level at which the system will execute upon a criticality level change, and (*iii*) what happens with the (jobs of the) tasks that are no longer executed when a criticality level up occurs and accordingly how to deal with tasks that are again allowed to execute when a criticality level down occurs. In this section, we consider these three topics for our AMC*-scheme.

4.1. Overrun of $\vec{C}_i(\lambda_i)$

For AMC*, we consider an overrun of the worst-case computation time of a task τ_i at its representative criticality level λ_i *erroneous behavior*, similar to [18]. Upon such an overrun, a criticality level up occurs if $\lambda_i < \Lambda_1$. The behavior is unspecified for $\lambda_i = \Lambda_1$; see also Condition 1.

4.2. New criticality level upon a criticality level change

Because an overrun at criticality level Λ_1 is considered erroneous behavior and worst-case computation times are monotonically non-decreasing for increasing criticality levels (2), we consider three cases when a criticality level up occurs, assuming the system is executing at criticality level Λ:

1. $\Lambda \leq \lambda_i < \Lambda_1 \wedge \vec{C}_i(\Lambda) = \vec{C}_i(\lambda_i)$: When a task τ_i overruns its worst-case computation time at its representative criticality level λ_i and λ_i is smaller than the highest criticality level Λ_1, the new criticality level Λ^{new} is the smallest criticality level larger than λ_i, i.e.
$$\Lambda^{\text{new}} = \min\{\lambda \in \mathcal{L} | \lambda > \lambda_i\} \qquad (3)$$
2. $\vec{C}_i(\Lambda) < \vec{C}_i(\lambda_i)$: When a task τ_i overruns its worst-case computation time at the criticality level Λ ($\vec{C}_i(\Lambda)$) and that computation time is less than its worst-case computation time at its representative criticality level λ_i ($\vec{C}_i(\lambda_i)$), the new criticality level Λ^{new} is the smallest

Table 1. Existing API provided by the ExSched Module to install and un-install a plug-in

Method	Description
extern void install_scheduler(void (*task_run_plugin)(resch_task_t*), void (*task_exit_plugin)(resch_task_t*), void (*job_release_plugin)(resch_task_t*), void (*job_complete_plugin)(resch_task_t*));	Install Plug-in
extern void uninstall_scheduler(void);	Un-install Plug-in

criticality level not giving rise to an overrun for τ_i, i.e.

$$\Lambda^{\text{new}} = \min\left\{\lambda \in \mathcal{L} | \vec{C}_i(\Lambda) < \vec{C}_i(\lambda)\right\} \quad (4)$$

3. **Unspecified behavior** An overrun of $\vec{C}_i(\lambda_i)$ of task τ_i is unspecified for $\lambda_i = \Lambda_1$; see Subsection 4.1.

When a criticality level down occurs, the system returns to the lowest criticality level Λ_m.

4.3. Policies for criticality level changes

We considered three policies for mixed criticality, i.e. *suspend*, *resume*, and *abort*.

Definition 2. *The* suspend *policy for a task (*i*) temporarily does not give any execution time to a currently active job of that task and (*ii*) suppresses new releases of jobs of that task.*

Definition 3. *The* resume *policy allows suspended tasks to release new jobs.*

The release of new jobs shall satisfy the constraints of the system, i.e. no earlier than allowed according to the minimal inter-activation time.

Definition 4. *The* abort *policy for a task decides whether or not the current job of a suspended task is discarded or allowed to continue at a later time.*

The *abort* policy is conditional, i.e. depending on the context, suspended jobs of a task may, but need not, be aborted. As an example, in a reservation-based resource management context, where suspension is used to prevent jobs of tasks to execute upon depletion of a budget, abortion will not be applied. In our initial experiments extending ExSched with mixed criticality, we suspend jobs of tasks that are no longer executed when a criticality level up occurs and abort those jobs when a criticality level down subsequently occurs. By delaying the actual abort, we minimize overhead upon a criticality level up.

5. Extending ExSched with mixed criticality support

In this section, we describe our extension of ExSched with mixed criticality support. We start with a description of the required extensions in Section 5.1. The design of the system is the topic of Section 5.2. We demonstrate our implemented system by means of an example in Section 5.3.

5.1. Basic mechanisms

To support the AMC* scheme, the following basic mechanisms, are required:

- *run-time monitoring*, to keep track of the amount of time a job of a task has spent on execution, to detect depletion of a "*budget*", and to realize (i.e. trigger the handler for) the criticality level up functionality;
- *task-management services*, i.e. the *suspend*, *resume*, and *abort* policies, which have been described in Subsection 4.3;
- *idle-time detection*, to realize (i.e. trigger the handler for) the criticality level down functionality.

We briefly consider these mechanisms in the following subsections.

5.1.1. Run-time monitoring

Run-time monitoring is a basic mechanism that is not only required for mixed criticality, but also for reservation-based resource management.

Rather than incorporating run-time monitoring in a to-be-developed AMC∗ plug-in, we therefore decided to extend the ExSched Module.

The timers used in ExSched do not satisfy our needs, however. In particular, the values of the execution times are stored in so-called "*jiffies*", which is the time between two successive clock ticks of the real-time clock. Instead of using the (low-resolution) real-time clock, we decided to base monitoring on high-resolution timers (provided through hrtimer.h), with a resolution in the order of nanoseconds on an Intel Core I5 processor. The methods for run-time monitoring are described in Table 2.

5.1.2. Task management services

The task management functionality to realize criticality level changes is described in Table 3. We believe that this functionality is of a generic nature, i.e. that it can also be used by other plug-ins. Moreover, in order to be able to "*hide*" the specific details of the actual operating system, which is one of the design goals of ExSched [10], plug-ins shall not be aware of specific operating system functionality. As a result, the ExSched Module is the only place where this functionality can be implemented.

Note that we combined the *resume* and *abort* policy into a single primitive resume_task(). For the AMC* implementation described in this paper, we always pass a value true for the parameter abort when calling resume_task().

The abort functionality has been implemented using Linux signals, in particular the POSIX compliant SIGUSR1 signal. Before a task is actually resumed and only when a job of the task has been suspended, a SIGUSR1 is sent to the task. This allows the task to perform clean-up activities as required when resumed.

5.1.3. Extended plug-in interface

Table 4 presents the methods that the AMC∗ plug-in provides to the ExSched Module, allowing the latter to bring criticality level up and criticality level down events to the attention of the plug-in. We expect the methods to be of a sufficient generic nature to justify incorporation in the generic install-methods, e.g. to monitor individual tasks. The description given in Table 4 is therefore from the perspective of the AMC∗ plug-in. The method to install a plug-in given in Table 1 is extended with parameters for these two methods.

5.2. System design

Figure 2 shows the static structure of ExSched extended with AMC*. A similar structure has been used for AMC* as for ExSched, i.e. a library AMC∗ Library in user space and a loadable kernel module AMC∗ Plugin Module in kernel space, using the ioctl () system call for communication. Although we generalized ExSched by extending the ExSched Module with run-time monitoring and additional task-management services, amongst others, its general architecture remained unchanged, i.e. Figure 2 is an extension of Figure 1.

The generic interfaces of the ExSched Module towards a plug-in have been described in the previous section. Below, we consider the AMC∗ Library and the AMC∗ Plugin Module in more detail. Both modules share a header file defining a constant NO_OF_CRIT_LEVELS denoting the number of criticality levels supported by the system.

5.2.1. AMC∗ Library

The AMC∗ Library provides a method to allow a task τ_i to set its representative criticality level λ_i and its worst-case computation times for each criticality level $\lambda \in \mathcal{L}$; see Table 5.

5.2.2. AMC∗ Plugin Module

The AMC∗ Plugin Module stores the representative criticality level λ of each task and the worst-case computation time $\vec{C}$ of each task for every criticality level. Moreover, it stores and maintains the criticality level indicator Γ of the system. In particular, it implements the handlers for the criticality level up and criticality level

Table 2. New local methods of the ExSched Module for run-time monitoring

Method	Description
void start_monitor_timer (resch_task_t *rt)	Start timer for a task τ_i denoted by *rt for an amount of time $\vec{C}_i(\Lambda) - \beta_i(t)$. Both $\vec{C}_i(\Lambda)$ and $\beta_i(t)$ are stored in the task's control block.
void stop_monitor_timer (resch_task_t *rt)	Stop timer for a task τ_i denoted by *rt and update $\beta_i(t)$.
enum hrtimer_restart monitor_expire_handler (struct hrtimer *timer)	Interrupt handler of the timer identified by *timer.

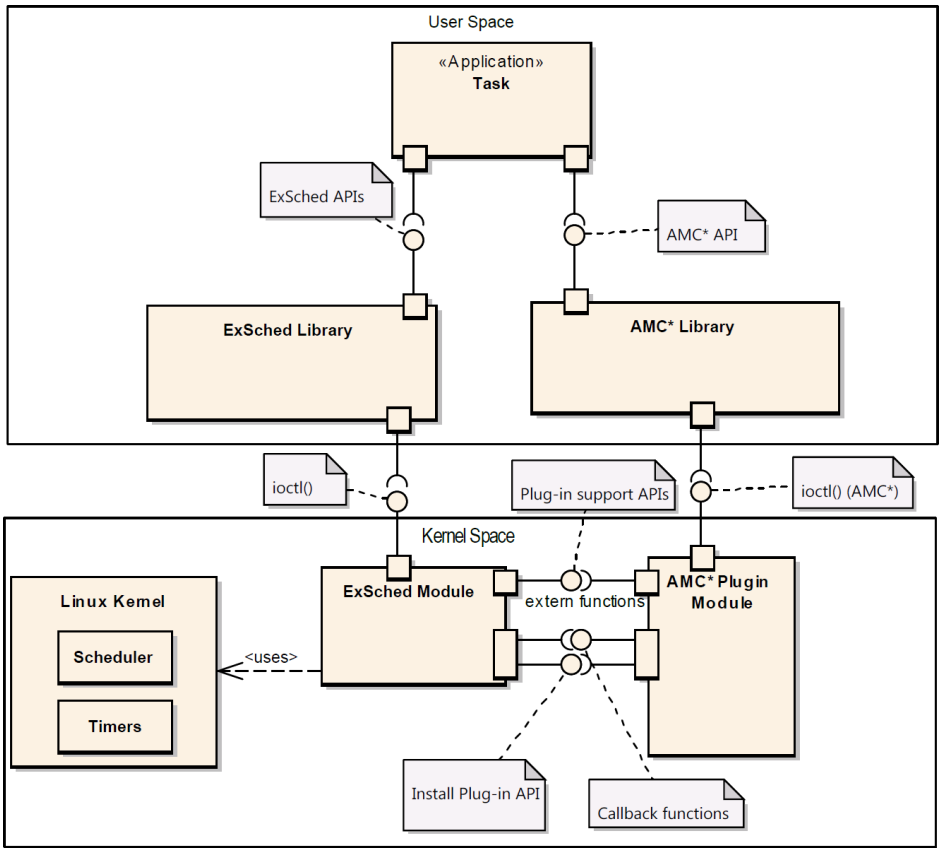

Figure 2. ExSched extended with AMC* [19, 20]

down handler, using the functionality provided by the ExSched Module.

5.3. An example

In this section, we illustrate our system by means of an example[6], with 3 criticality levels and 4 tasks. The characteristics of the synthetic task set are given in Table 6[7].

Figure 3, which has been created by means of Grasp [30][8], shows a timeline with the executions of the tasks. The figure shows both a criticality level up, at time 57 ms and 61 ms, as well as a criticality level down at time 73 ms.

[6]The interested reader is referred to [19, 20] for other examples.

[7]Don't-care values for $\vec{C}$ are specified as zero, i.e. $\lambda_i < \lambda \Rightarrow \vec{C}_i(\lambda) = 0$.

[8]A version of Grasp is available in the ExSched distribution at http://www.idt.mdh.se/~exsched/.

Table 3. New methods provided by the ExSched Module to its plug-ins for task management

Method	Description
void suspend_task (resch_task_t *rt)	Suspends a task.
void resume_task (resch_task_t *rt, bool abort)	Resumes a task. When abort is true, a pending job will be aborted.
void abort_job (resch_task_t *rt)	Aborts the pending job of a task.

Table 4. New methods expected by the ExSched Module from its plug-ins, i.e. callback functions, to handle criticality level changes

Method	Description
void (*monitor_expire_plugin) (resch_task_t *rt)	Criticality-level up handler.
void (*idle_time_plugin) (resch_task_t *rt)	Criticality-level down handler.

Table 5. AMC∗ API: Method provided by the AMC∗ Library

Method	Description
int rt_set_rep_crit_level (int rep_crit, unsigned long[NO_OF_CRIT_LEVELS] wcet_per_crit)	Method to set a task's representative criticality level and worst-case computation time per criticality level.

6. Moving from Linux to μC/OS-II

ExSched [10] supports both Linux and VxWorks, but lacks support for an OSEK-compliant real-time operating system, such as μC/OS-II or ERIKA Enterprise [31]. In this section, we describe our efforts in moving from ExSched with Linux to μC/OS-II. We start this section with our experience with and evaluation of ExSched. We subsequently consider the usage of μC/OS-II.

6.1. Experience with and evaluation of ExSched

Based on ExSched's features, i.e. (*i*) being an operating system independent external CPU scheduler framework, (*ii*) providing support for temporal isolation through hierarchical scheduling, and (*iii*) providing support for multi-core scheduling, selecting ExSched for our extension with support for mixed criticality seemed a good choice. As illustrated by the example in Section 5.3, we managed to build a functional prototype of our system.

Extending ExSched with mixed criticality support turned out to be laborious, however. Instead of adding just a "*mixed criticality*"-specific plug-in, we also extended and revised the ExSched Module, as described in Section 5. Although the code is documented with samples illustrating its usage, critical user documentation is missing. A conference paper [10] describes Exsched's high-level design. Other software engineering artifacts, such as requirements, specification, design and corresponding tests are unavailable. As a result, the Exsched API is hard to validate and testing our mixed criticality plugin against its API is even harder. Although we resolved the problems with Exsched that we encountered, we did not thoroughly validate and verify the existing functionality of ExSched. Based on our experience, the framework is hard to maintain.

Table 6. Task characteristics

Task	λ	π	$\vec{C}$ Λ_3	Λ_2	Λ_1	T
Task 1	Λ_3	99	6 ms	0 ms	0 ms	45 ms
Task 2	Λ_2	98	6 ms	10 ms	0 ms	50 ms
Task 3	Λ_2	97	6 ms	6 ms	0 ms	50 ms
Task 4	Λ_1	96	6 ms	9 ms	12 ms	60 ms

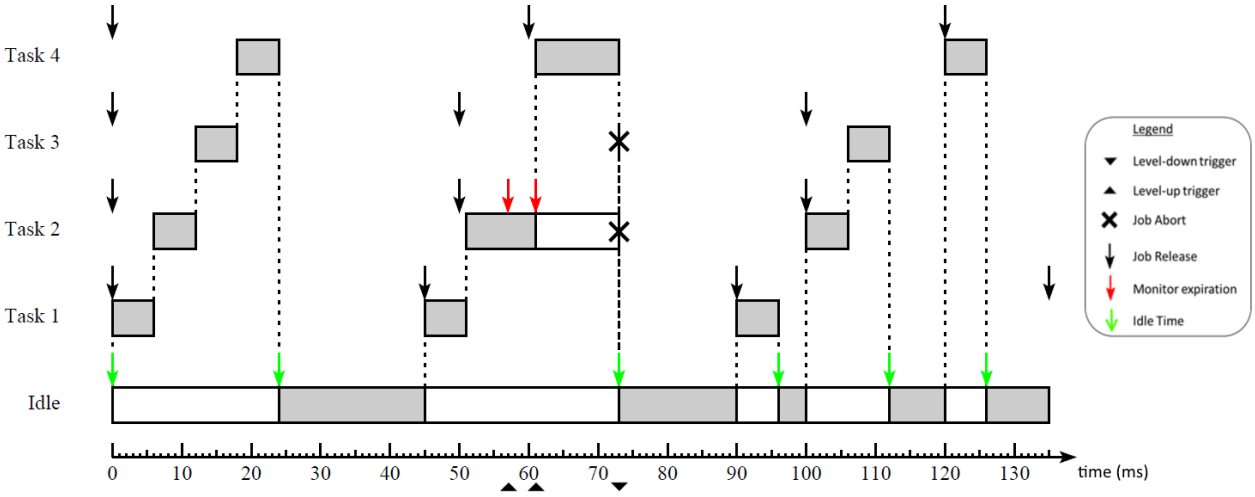

Figure 3. At time 57 ms, the job of Task 2 executed for its worst-case computation time at criticality level Λ_3 but didn't complete yet. As a result, a criticality level up change to Λ_2 occurs. At time 61 ms the job of Task 2 executed for its worst-case computation time at its representative criticality level Λ_2 but didn't complete yet, resulting in a criticality level up change to Λ_1. The active jobs of Task 2 and Task 3 at time 61 ms are suspended due to the criticality level up. A criticality level down occurs at time 73 ms to Λ_3 and the SIGUSR1 signal is sent at that time, effectively aborting the suspended job

One of the reasons to select ExSched was the availability of existing plug-ins, such as hierarchical scheduling and multi-core. As described in Section 3.2, the current implementation only allows to use a single plug-in at the time, however. Whenever multiple plug-ins are desired, a major redesign of ExSched seems to be required. Given these experiences and gained insight, we decided to entirely abandon ExSched for our future efforts.

6.2. Extending μC/OS-II with AMC*

In this section, we briefly describe the rationale for selecting μC/OS-II, the extension of μC/OS-II with AMC*, and a comparison between ExSched and μC/OS-II regarding the extension with mixed criticality.

6.2.1. Background and rationale

Based on our earlier experience with the OSEK-compliant RTOS μC/OS-II [14][9] in (*i*) research [24, 25], (*ii*) an automotive case study implementing and demonstrating active suspension in a Jaguar XF [26], and (*iii*) education, i.e. the core course *Real-time software systems engineering* (2IN70) in the master automotive technology [32] at the TU/e, we decided to select this RTOS for our next step and to use it in combination with RELTEQ (Relative Timed Event Queues) [1, 23]. RELTEQ provides a general timer management system and supports periodic tasks and a hierarchical scheduling framework (HSF) in our extended implementation of μC/OS-II; see Figure 4.

[9]Unfortunately, the supplier of μC/OS-II, Micrium, has discontinued the support for the OSEK-compatibility layer.

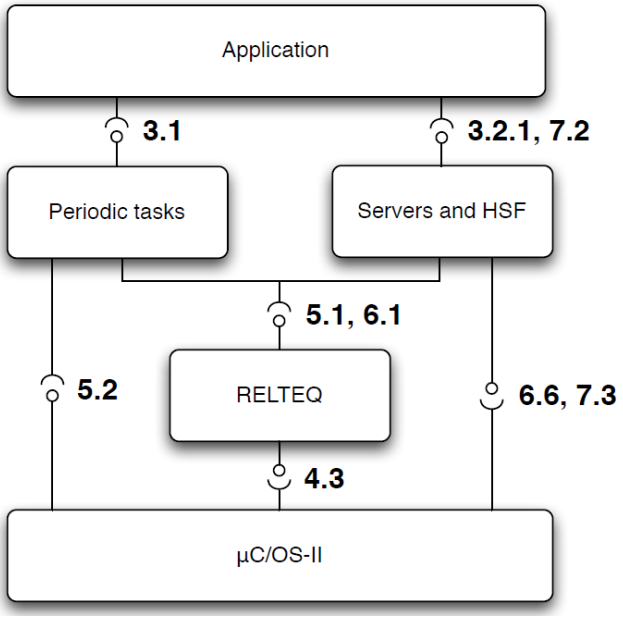

Figure 4. Interfaces between μC/OS-II and its extension [23]. The numbers indicate the sections in [23] describing the provided interfaces and their implementation

In our earlier work, we created a port for μC/OS-II to the OpenRISC platform [33] to experiment with the accompanying cycle-accurate simulator and to ease development. Our set-up also runs on a Freescale EVB9S12XF512E evaluation board with a 16-bits, MC9S12XF512 processor and 32 kB on-chip RAM.

6.2.2. Basic mechanisms and specific AMC*-functionality

As described in Section 5.1, three sets of basic mechanisms are required to support AMC*, *run-time monitoring*, *task management services*, and *idle-time detection*. All these mechanisms are essentially supported through RELTEQ and our earlier extension of μC/OS-II with an HSF.

To implement specific AMC*-functionality, we extended the task-control block with mixed-criticality specific information, such as the representative criticality level λ and the vector of worst-case computation times $\vec{C}$. In addition, we provided a method similar to rt_set_rep_crit_level (see Table 5) to set λ and $\vec{C}$. Given these mechanisms and extended data structures, implementing the specific AMC*-functionality turned out to be straightforward. In particular, we implemented a dedicated module for AMC* with the level up handler and the level down handler (see Table 4). The level up handler is called from RELTEQ, upon the detection of an overrun, and the level down handler is called from the OSTaskIdleHook within μC/OS-II. Support for run-time monitoring (Table 2) and task management (Table 3) is provided by RELTEQ and μC/OS-II, respectively. An overview of the μC/OS-II architecture including the extensions for both RELTEQ and AMC* is given in Figure 5.

6.2.3. A comparison between ExSched and μC/OS-II

Compared to ExSched with Linux, extending RELTEQ and μC/OS-II with AMC* was relatively easy. The only functionality implemented in Linux that could not be supported by our μC/OS-II extension concerned allowing a task to perform the clean-up activities as required when resumed, i.e. μC/OS-II lacks functionality similar to the POSIX compliant SIGUSR1 signals. Another disadvantage of our extension of RELTEQ and μC/OS-II with AMC* is that RELTEQ

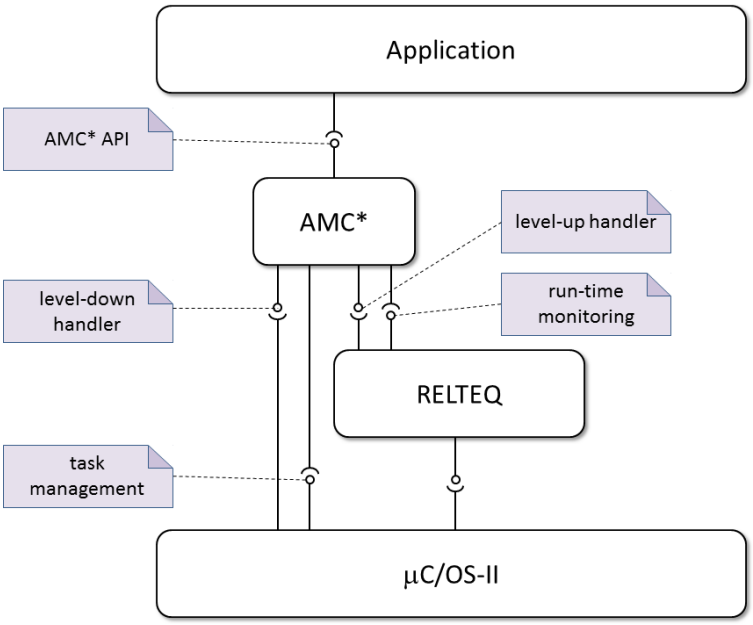

Figure 5. Interfaces between μC/OS-II, RELTEQ and AMC*

has been implemented in μC/OS-II[10], and our AMC* extension required additional changes to μC/OS-II as well. Hence, whereas ExSched requires no pathes (modifications) to the original source code of the underlying operating system, our support for AMC* using RELTEQ did require modifications to μC/OS-II. Extending μC/OS-II with AMC* without patches to the original source code would be considerably less straightforward.

As a final remark, we merely observe that whereas μC/OS-II inherently provides functionality to suspend and resume a task by means of OSTaskSuspend() and OSTaskResume(), the OSEK/VDX-standard [8] lacks such functionality. Extending an OSEK-compliant RTOS with mixed criticality without patches may therefore not be trivial.

7. A reflection on AMC*

In this section, we briefly reflect on AMC*, our mixed criticality scheme. We first report on our experience with AMC*. We subsequently describe directions for resolving the undesirable behavior encountered.

7.1. Experience with and evaluation of AMC*

Within the literature, various options for improvement of the AMC-scheme have been proposed [2, 18]. Below, we briefly report upon two aspects we encountered while experimenting with our implementation that have, to the best of our knowledge, not been reported before in the literature.

7.1.1. Erroneous and unspecified behavior

As described in Section 4.1, an overrun of the worst-case computation time of a task τ_i at its representative criticality λ_i is considered *erroneous behavior*. Moreover, the behavior is *unspecified* when $\lambda_i = \Lambda_1$. Figure 3 shows an example with erroneous behavior of a job of Task 2 with $\lambda_2 < \Lambda_1$, which gives rise to a criticality level up conform the AMC scheme (see Condition 1). A drawback of this behavior is that Task 3, which has the same representative criticality level as Task 2, is also no longer allowed to execute, and its activation at time 50 is therefore aborted as well. Moreover, this criticality level up is not

[10]Unlike the implementation of an HSF in μC/OS-II, the implementation of an HSF in VxWorks described in [34] required no changes to the operating system.

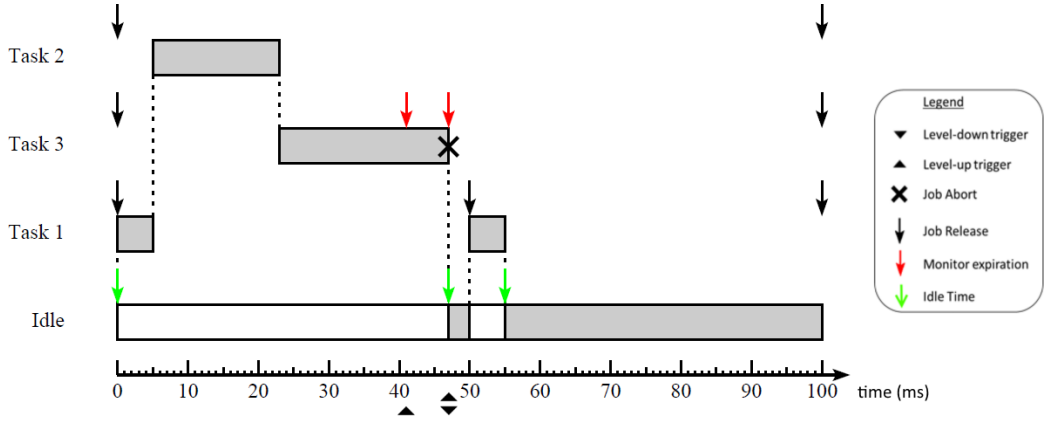

Figure 6. A criticality level up immediately followed by a criticality level down in AMC*

Table 7. Task characteristics

Task	λ	π	$\vec{C}$ Λ_3	Λ_2	Λ_1	T
Task 1	Λ_3	99	5 ms	0 ms	0 ms	50 ms
Task 2	Λ_1	98	18 ms	24 ms	24 ms	100 ms
Task 3	Λ_2	97	18 ms	24 ms	0 ms	100 ms

necessitated by a need for more anticipated resources by tasks with a higher representative criticality level than the criticality level at which the system is executing, but instead to prevent the erroneous behavior of Task 2 from jeopardizing the correct timing behavior of tasks with the same or a higher representative criticality level.

Although it is theoretically (i.e. from an academic perspective) convenient to classify an overrun of the worst-case response time of a task at its representative criticality level as erroneous behavior, this is clearly not desirable from a practical (i.e. an industrial) perspective.

7.1.2. Criticality-level up immediately followed by a criticality level down

Using the original AMC*-scheme, a criticality level up can be immediately followed by a criticality- level down, as illustrated in Figure 6 for a task set with characteristics as given in Table 7.

At time $t = 41$, a job of Task 3 experiences an overrun of $\vec{C}_3(\Lambda_3) = 18$ ms and a criticality level up occurs from criticality level Λ_3 to Λ_2. The job of Task 3 experiences a next overrun of $\vec{C}_3(\Lambda_2) = 24$ ms at time $t = 47$ ms and a criticality level up occurs towards Λ_1. The system subsequently encounters an idle-time and the system returns to its lowest criticality level Λ_3, i.e. the system exhibits the undesirable behavior of a criticality level up immediately followed by a criticality level down.

In case we modify the characteristics of Task 3 to $\vec{C}_3(\Lambda_3) = \vec{C}_3(\Lambda_2) = 24$ ms, we even have a criticality level up from Λ_3 to Λ_1 at time $t = 47$ would immediately be followed by a criticality level down to Λ_3. This behavior is clearly undesirable.

7.2. Improving AMC*

To prevent (or at least mitigate) the undesirable behavior identified in the previous section, we propose to bound the time provided to a task τ_i at its representative criticality level λ_i to its worst-case computation time $\vec{C}(\lambda_i)$, e.g. through resource reservation with temporal protection [35], rather than raising the criticality level. Similar to Quality-of-Service like approaches [36,37], tasks therefore have to *get by* with a budget given by $\vec{C}(\lambda_i)$ at their representative criticality level. Hence, we propose to adapt both AMC and AMC*, and complement these schemes with resource reservation.

7.2.1. Adaption of AMC

First, we change the condition $\Lambda \leq \lambda_i < \Lambda_1$ in Condition 1 of the AMC scheme to $\Lambda < \lambda_i \leq \Lambda_1$, effectively suppressing a criticality level up upon an overrun at a task's representative criticality level, i.e. Condition 1 now becomes:

Condition 3. *When a job of task τ_i is executing at time t while the system is running at level Λ with $\Lambda < \lambda_i \leq \Lambda_1$ and the actual execution time $\beta_i(t)$ equals the worst-case computation time $\vec{C}_i(\Lambda)$ of τ_i, a criticality level up occurs.*

7.2.2. Adaption of AMC*

Next, we reconsider the three topics for our AMC*-scheme, that were discussed in Section 4. As mentioned above, an overrun of $\vec{C}_i(\lambda_i)$ is now prevented by a resource reservation. With this change, handling that overrun becomes the responsibility of the (developer of the) task and part of the specification of the task. In this way, we also resolved the unspecified behavior for an overrun of a task at a criticality level Λ_1. Upon a criticality level up change, the three cases distinguished in Section 4.2 simplify to only one case:

1. $\Lambda < \lambda_i \leq \Lambda_1$: When a task τ_i overruns its worst-case computation time at the criticality level $\Lambda < \lambda_i$, the new criticality level Λ^{new} remains unchanged if $\vec{C}_i(\Lambda) = \vec{C}_i(\lambda_i)$ and becomes the smallest criticality level not giving rise to an overrun for τ_i otherwise, i.e.

$$\begin{aligned} \Lambda^{\text{new}} = \; & \textbf{if } \vec{C}_i(\Lambda) = \vec{C}_i(\lambda_i) \\ & \textbf{then} \\ & \quad \Lambda \\ & \textbf{else} \\ & \quad \min\left\{\lambda \in \mathcal{L} | \vec{C}_i(\Lambda) < \vec{C}_i(\lambda)\right\} \\ & \textbf{fi} \end{aligned} \tag{5}$$

Note that by keeping the criticality level unchanged when $\vec{C}_i(\Lambda) = \vec{C}_i(\lambda_i)$, we prevent a criticality level up whenever the resource reservation already bounds the execution of task τ_i. The policies for criticality changes, being the third topic for our AMC*-scheme discussed in Section 4.3, remain unaltered.

7.2.3. Resource reservation

Finally, for every task $\tau_i \in \mathcal{T}$ we assume a resource reservation ρ_i with a priority equal to the priority of τ_i and a capacity $\vec{C}_i(\lambda_i)$ that is replenished when τ_i is activated and lost when τ_i becomes idle. Task τ_i can execute using the capacity of ρ_i as long as the system is executing at a criticality level Λ at most equal to τ_i's representative criticality level λ_i. At a higher criticality level, ρ_i will be disabled.

Additional policies and mechanisms to support a (developer and a) task upon detecting and/or handling an overrun are conceivable, such as means to measure progress [37], but fall outside the scope of this paper.

8. Conclusion

In this paper, we described our experience with extending an OSEK-compliant RTOS with mixed criticality support. Instead of selecting a specific RTOS, we started our investigations with ExSched [10], an operating system independent external CPU scheduler supporting multiple operating systems. For our initial experiments, we used ExSched in combination with Linux. We selected AMC [9] as a basic mixed criticality scheme, extended its model from two to multiple criticality levels, and complemented it with specified behavior for criticality level up and criticality level down functionality. Extending ExSched required both extensions and revisions of the ExSched Module. In particular, we incorporated generic functionality usable for multiple plug-ins, such as run-time monitoring based on high-resolution timers and task management services, e.g. *suspend* and *resume*, and extended the plug-in interface of the ExSched Module. In addition, we developed a dedicated plug-in for mixed criticality, baptized AMC* Plugin Module, and complemented that kernel module with an AMC* Library in user space. In particular, we used a similar design structure for AMC* as for ExSched itself, effectively increasing the modularity of ExSched. Our extension requires minimal overhead when a criticality level up occurs by postponing clean-up actions till

a criticality level down occurs. We built a working prototype of our system, as demonstrated through visualized traces using Grasp [30].

Despite the fact that ExSched is a great research vehicle, we decided to abandon ExSched based on our experiences with and insights gained during the extension of ExSched with mixed criticality support. During our subsequent investigations, we directed our attention to the OSEK-compliant RTOS μC/OS-II and its extension with RELTEQ [1]. Compared to extending ExSched with AMC*, extending μC/OS-II and RELTEQ with AMC* turned out to be straightforward. Unfortunately, our implementation required changes to μC/OS-II, whereas the implementation in ExSched required no patches (i.e. modifications) to the original source code of Linux. Although μC/OS-II has been stable for many years, new kernel versions will require updates of our system. Extending μC/OS-II with AMC* without making changes to the original source code seems considerably less straightforward than our implementation, however.

Finally, we briefly reflected on AMC and AMC*. We encountered undesirable behavior of both the original and the extended scheme, i.e. erroneous and unspecified behavior as well as a criticality level up immediately followed by a criticality level down, and described improvements for both schemes in combination with resource reservation. As future work, we aim at enhancing our implementation with the described improvement of the AMC* scheme. Additional policies and mechanisms to support (a developer and) a task upon detecting and/or handling an overrun are a topic of future work as well.

References

[1] M. Holenderski, R. Bril, and J. Lukkien, "An efficient hierarchical scheduling framework for the automotive domain," in *Real-Time Systems, Architecture, Scheduling, and Application*, D.S.M. Babamir, Ed. InTech, 2012, pp. 67–94.

[2] A. Burns and R. Davis, "Mixed criticality systems – a review," University of York, UK, Tech. Rep., 2018. [Online]. https://www-users.cs.york.ac.uk/burns/review.pdf

[3] i-GAME, *Grand Cooperative Driving Challenge 2016*, 2016. [Online]. http://www.gcdc.net/en/

[4] S. Baruah and A. Burns, "Implementing mixed criticality systems in Ada," in *Proceedings 16th Ada-Europe International Conference on Reliable Software Technologies*, 2011, pp. 174–188.

[5] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson, "RTOS support for multi-core mixed-criticality systems," in *Proceedings 18th Real-Time and Embedded technology and Applications Symposium (RTAS)*, 2012, pp. 197–208.

[6] Y.S. Kim and H.W. Jin, "Towards a practical implementation of criticality mode changes in RTOS," in *Proceedings 19th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, 2014.

[7] A. Kritikakou, C. Pagetti, C. Rochange, M. Roy, M. Faugère, S. Girbal, and G. Pérez, "Distributed run-time WCET controller for concurrent tasks in mixed-criticality systems," in *Proceedings 22nd International Conference on Real-Time Networks and Systems*, 2014, pp. 139–148.

[8] "OSEK/VDX operating system," OSEK group, Tech. Rep., 2005. [Online]. https://web.archive.org/web/20160310151905/http://portal.osek-vdx.org/files/pdf/specs/os223.pdf

[9] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *Proceedings 32nd IEEE Real-Time Systems Symposium*, 2011, pp. 34–43.

[10] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar, "ExSched: An External CPU Scheduler Framework for Real-Time Systems," in *Proceedings 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2012, pp. 240–249.

[11] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings 27th IEEE Real-Time Systems Symposium (RTSS)*, 2006, pp. 111–123.

[12] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA: Adaptive quality of service architecture," *Software Practice and Experience*, Vol. 39, No. 1, 2009, pp. 1–31.

[13] *Ubuntu Kernel Repository.* [Online]. http://kernel.ubuntu.com/~kernel-ppa/mainline/

[14] J. Labrosse, *MicroC/OS-II: The Real Time Kernel*, 2nd ed. CMP Books, 2002.

[15] T. Gupta, E. Luit, M. van den Heuvel, and R. Bril, "Extending ExSched with mixed criticality support – an experience report," in *Proceedings 3rd Workshop on Automotive System/-Software Architecture (WASA), In conjunction with IEEE International Conference on Software Architecture (ICSA)*, 2017.

[16] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proceedings 28th IEEE Real-Time Systems Symposium (RTSS)*, 2007, pp. 239–243.

[17] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *Proceedings 20th Euromicro Conference on Real-Time Systems (ECRTS)*, 2008, pp. 147–155.

[18] F. Santy, L. George, P. Thierry, and J. Goossens, "Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP," in *Proceedings 24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012, pp. 155–165.

[19] T. Gupta, "Extending a real-time operating system with a mechanism for criticality-level changes," Eindhoven University of Technology, Stan Ackermans Institute (SAI), Tech. Rep. 2015027, 2015. [Online]. https://pure.tue.nl/ws/files/32337443/Final_Report_Tarun_Gupta.pdf

[20] T. Gupta, "Interoperable robustness measures for safety-integrity levels (SILs)," European Commission 7th Framework Programme, i-GAME Deliverable D2.4, 2015. [Online]. http://www.gcdc.net/images/doc/D2.4.Interoperable.robustness.measures.for.safety-integrity.level.pdf

[21] A. Massa, *Embedded Software Development with eCOS*. Prentice Hall, 2003.

[22] G. Durrieu, M. Faugèere, S. Girbal, D. Garcia Pérez, C. Pagetti, and W. Puffitsch, "Predictable flight management system implementation on a multicore processor," in *Proceedings 9th Embedded Real-Time Software*, 2014.

[23] M. Holenderski, W. Cools, R. Bril, and J. Lukkien, "Extending an open-source real-time operating system with hierarchical scheduling," Eindhoven University of Technology (TU/e), Tech. Rep. CS-report 10-10, 2010.

[24] M. Holenderski, W. Cools, R.J. Bril, and J. Lukkien, "Multiplexing real-time timed events," in *Proceedings 14 IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2009.

[25] M. Heuvel, R. Bril, and J. Lukkien, "Transparent synchronization protocols for compositional real-time systems," *IEEE Transactions on Industrial Informatics*, Vol. 8, No. 2, 2012, pp. 322–336.

[26] M. Heuvel, E. Luit, R. Bril, J. Lukkien, P. Verhoeven, and M. Holenderski, "An experience report on the integration of ECU software using an HSF-enabled real-time kernel," in *Proceedings 11th Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2015, pp. 51–56.

[27] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a real-time environment," *JACM*, Vol. 20, No. 1, 1973, pp. 46–61.

[28] R. Bril, J. Lukkien, and W. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption," *Real-Time Systems Journal*, Vol. 42, No. 1-3, 2009, pp. 63–119.

[29] S. Kato, R. Rajkumar, and Y. Ishikawa, "A loadable real-time scheduler suite for multicore platforms," Technical Report CMU-ECE-TR09-12, Tech. Rep, Tech. Rep., 2009.

[30] M. Holenderski, M. Heuvel, R. Bril, and J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *Proceedings 1st Int. Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS)*, 2010, pp. 37–42.

[31] P. Gai, E. Bini, G. Lipari, M. Di Natale, and L. Abeni, "Architecture for a portable open source real time kernel environment," in *2nd Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, 2000.

[32] *Master Automotive Technology*, Eindhoven Univeristy of Technology (TU/e), 2017. [Online]. https://educationguide.tue.nl/programs/graduate-school/masters-programs/automotive-technology/

[33] *OpenRISC overview*, OpenCores, 2009. [Online]. http://www.opencores.org/project,or1k

[34] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R.J. Bril, "Towards hierarchical scheduling in VxWorks," in *Proceedings 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2008, pp. 63–72.

[35] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *Proceedings SPIE, Vol. 3310, Conference on Multimedia Computing and Networking (CMCN)*, 1998, pp. 150–164.

[36] R.J. Bril and E. Steffens, "User focus in consumer terminals and conditionally guaranteed budgets," in *Proceedings 9th International Workshop on Quality of Service (IWQoS)*, ser.] Lecture Notes in Computer Science (LNCS), No. 2092, 2001, pp. 107–120.

[37] C. Wüst, L. Steffens, W. Verhaegh, R.J. Bril, and C. Hentschel, "QoS control strategies for high-quality video processing," *Real-Time Systems*, Vol. 30, No. 1–2, 2005, pp. 7–29.