Mixed-Criticality Runtime Mechanisms and Evaluation on Multicores

Lukas Sigrist, Georgia Giannopoulou, Pengcheng Huang, Andres Gomez, Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
Email: firstname.lastname@ee.ethz.ch

***Abstract*—Multicore systems are being increasingly used for embedded system deployments, even in safety-critical domains. Co-hosting applications of different criticality levels in the same platform requires sufficient isolation among them, which has given rise to the mixed-criticality scheduling problem and several recently proposed policies. Such policies typically employ runtime mechanisms to monitor task execution, detect exceptional events like task overruns, and react by switching scheduling mode. Implementing such mechanisms efficiently is crucial for any scheduler to detect runtime events and react in a timely manner, without compromising the system's safety. This paper investigates implementation alternatives for these mechanisms and empirically evaluates the effect of their runtime overhead on the schedulability of mixed-criticality applications. Specifically, we implement in user-space two state-of-the-art scheduling policies: the flexible time-triggered FTTS [1] and the partitioned EDF-VD [2], and measure their runtime overheads on a 60-core Intel®Xeon Phi and a 4-core Intel®Core i5 for the first time. Based on extensive executions of synthetic task sets and an industrial avionic application, we show that these overheads cannot be neglected, esp. on massively multicore architectures, where they can incur a schedulability loss up to 97%. Evaluating runtime mechanisms early in the design phase and integrating their overheads into schedulability analysis seem therefore inevitable steps in the design of mixed-criticality systems. The need for verifiably bounded overheads motivates the development of novel timing-predictable architectures and runtime environments specifically targeted for mixed-criticality applications.**

I. Introduction

As a result of the constantly increasing share of multicore and manycore platforms in the electronics market, the field of embedded systems follows an unprecedented trend towards integrating multiple applications into a single platform. This trend applies even to real-time systems for safety-critical domains, such as avionics and automotive. Applications in these domains are characterized by several criticality levels, which express the required protection against failure. The problem of scheduling *mixed-criticality* applications has thus become a major concern in industry and in the real-time research community. Appropriate scheduling policies must ensure minimal interference from lower criticality to higher criticality applications and at the same time efficient resource utilization, i.e., restricting over-provisioning for higher criticality applications.

Traditionally, the industrial practice in safety-critical domains favors *partitioning* mechanisms at the operating-system and hardware level, e.g., based on the ARINC-653 standard [3]. Within a partition, tasks of the same criticality level can exclusively access the platform resources, while failures in one partition cannot affect the task execution in other partitions. This way, partitioning enforces timing and performance isolation among applications of different criticality levels, thus facilitating the certification of mixed-criticality systems. Following this concept, the flexible time-triggered and synchronization-based (FTTS) scheduling policy was recently proposed [1] and adopted by the CERTAINTY project [4] as a software-based, flexible partitioning solution for enhanced resource efficiency. On the other hand, a plethora of scheduling policies have been proposed in the literature [5], which co-schedule applications of different criticality levels without explicit partitioning. To protect the timing correctness of higher criticality applications from the possibly unpredictable behavior of lower criticality applications, these policies are based on *runtime monitoring*. In cases where the execution time of some tasks surpasses given thresholds, immediate actions, such as degrading the service of lower criticality tasks, take place. Such scheduling policies aim for more efficient resource utilization than static partitioning methods, while also preserving the timing correctness of the higher criticality tasks. A representative multicore scheduling policy of this category is the partitioned EDF with virtual deadlines (pEDF-VD) [2].

A major difference between traditional (single-criticality) and mixed-criticality real-time scheduling lies in the implementation of the partitioning, monitoring and service degradation mechanisms that enforce the isolation among criticality levels. The runtime overhead of such mechanisms is crucial to the performance of the schedulers, which need to detect runtime events and react on time to initiate the required scheduling mode switches. Especially on multicores, where frequent inter-core synchronization and/or parallel periodic monitoring of several tasks need to be implemented, the cost paid for isolation can have a severe effect on the schedulability of mixed-criticality applications. In this paper, we focus exactly on this cost. We investigate different ways to implement mixed-criticality mechanisms and quantify their runtime overhead within a Linux user-space framework, on two platforms, the commercial 60-core Intel®Xeon Phi [6] and the 4-core Intel®Core i5. Based on empirical results, we evaluate the effect of the runtime overheads on schedulability for two state-of-the-art scheduling policies. Our contributions can be summarized as follows:

- We motivate the need to consider runtime mixed-criticality mechanisms, such as task monitoring and mode switching, by analytically characterizing the impact of their overhead on the schedulability condition of the state-of-the-art single-core scheduling policy EDF-VD [7].
- We propose different ways to implement common mixed-criticality mechanisms, i.e., task monitoring, task termination, global transition among partitions, mode switching, in a light-weight manner. To concretely define these mechanisms, we focus on two recently proposed scheduling policies: the FTTS [1] policy, which is inspired by the industrial approach of partitioning, and the pEDF-VD [2], which represents the monitoring-based multicore scheduling policies.
- We implement the two schedulers within a user-space scheduling framework and identify the mixed-criticality related overheads.
- We evaluate the above overheads on the manycore Xeon Phi through extensive executions of synthetic task sets for variable number of cores and variable system utilization. We show that the user-space implementation of mixed-

criticality mechanisms plays a crucial role, as it can lead to a schedulability loss as high as 97% for a global scheduler on a 32-core architecture. For partitioned scheduling, the respective loss is minimal.
- To demonstrate the need to adopt efficient platform-specific mechanisms in early design phases of industrial applications, we employ a real-world avionics application and compare the overall overhead of scheduling it on two different platforms, i.e., a Xeon Phi and a Core i5.

Aware of the fact that the two considered platforms and the Linux OS (user-space) are not designed for hard real-time systems, we believe that the evaluation indicates nonetheless the scalability challenges of mixed-criticality policies on multicores. Future research for developing timing-predictable architectures and operating system services for runtime monitoring, synchronization and scheduling mode switching seems inevitable for the deployment of mixed-criticality systems on parallel architectures.

Sec. II provides an overview of the mixed-criticality literature. Sec. III presents the system model and the two considered multicore mixed-criticality scheduling policies. Sec. IV provides a motivational example showing the impact of runtime overheads on mixed-criticality schedulability under EDF-VD. Sec. V discusses design trade-offs for implementing runtime monitoring, task termination, inter-core synchronization, and mode switching. Sec. VI presents the scheduling framework that was developed as the testbed for the runtime overhead measurements in Sec. VII, and Sec. VIII concludes the paper.

II. Related Work

Mixed-criticality scheduling is a research field attracting increasing attention nowadays. Following the original work of Vestal [8], several scheduling policies have been proposed for single-core and multi-core systems, see e.g., [2], [7], [9]–[11]. For an up-to-date compilation of these policies, we refer the interested readers to [5]. In the following, we categorize multicore mixed-criticality scheduling policies into (I) those targeting at strict *global timing isolation* among criticality levels by means of partitioning and (II) those targeting at *efficient resource utilization* by allowing interference of mixed-criticality tasks and monitoring task runtimes to react to unexpected events.

The first category, which is well accepted by industry for certification purposes, features mainly reservation-based scheduling policies, which adopt time-triggered schemes or resource servers to ensure that lower criticality tasks incur no or bounded interference to higher criticality tasks. In this line, Anderson et al. proposed adopting different scheduling strategies (cyclic executive, partitioned / global EDF, best-effort) for different criticality levels and a bandwidth reservation server for timing isolation [12], [13]. Tamas-Selicean et al. presented an optimization framework for task partitioning and time-triggered scheduling on multicores [14], complying with the ARINC-653 standard [3]. Giannopoulou et al. introduced a flexible time-triggered scheduling policy, which allows only tasks of the same criticality to be executed in parallel within global time frames [1]. The time frames are dynamically dimensioned at runtime, which is a major difference to the static resource allocation of the previous policies. In this paper, we use the last policy as representative of the first scheduling category.

The second category features usually mode-switched policies. Initially, all tasks are scheduled according to "optimistic" worst-case execution time parameters. While these might be unsafe bounds on the execution times, they model the tasks' commonly observed timing behavior. Exceptionally, if a task overruns its "optimistic" worst-case execution time at runtime, a mode switch occurs. Subsequently, lower criticality tasks receive degraded service and more conservative worst-case execution times are assumed for the higher criticality tasks. It is assumed that a mode switch may occur only rarely. Examples of such scheduling policies include the global policies in [11], [15] and the partitioned policies in [2], [16]. Among those, we focus on the partitioned EDF policy with virtual deadlines by Baruah et al. [2]. This policy is expected to incur lower runtime overhead and provide increased schedulability compared to its global counter-part, which is presented in [15].

Implementation aspects of mixed-criticality scheduling have started being addressed very recently [17], [18]. First, Herman et al. consider the implementation and runtime overhead of multicore mixed-criticality scheduling in [17], where the scheduling method of [12], [13] is implemented in the real-time operating system LITMUS [19]. The implemented framework operates at kernel level and is customized for the specific scheduling scheme. Since a kernel-level implementation can pose challenges regarding ease of implementation, extensibility and portability, we opt for a user-space solution in our work. This enables us to rapidly prototype various scheduling policies/mechanisms and compare their performance at different target platforms. Closer to our work lies the framework of Huang et al. [18], where several mixed-criticality policies are implemented on top of a standard Linux kernel and their runtime overheads are evaluated on an Intel®Core i7 platform. Unlike our work, the evaluated policies are designed for single-core systems and they are mainly priority-driven. Additionally, the authors do not explore alternative implementation choices for the mixed-criticality mechanisms or alternative platforms. To the best of our knowledge, this paper presents the first study of implementation alternatives towards light-weight monitoring, synchronization and scheduling mode switching.

III. Mixed-Criticality Scheduling

This section presents the considered mixed-criticality periodic task model and provides an overview of the two implemented scheduling policies in our framework. For an in-depth presentation of the scheduling policies, we refer the readers to [1] and [2], respectively.

A. Task Model

We consider mixed-criticality periodic task sets $\tau = \{\tau_1, \ldots, \tau_n\}$ and restrict our interest to two criticality levels, which are denoted as high (HI) and low (LO). Each task in τ is characterized by a 4-tuple $\tau_i = \{T_i, \chi_i, C_i(\text{LO}), C_i(\text{HI})\}$, where:

- $T_i \in \mathbb{R}^+$ is the task period,
- $\chi_i \in \{\text{LO}, \text{HI}\}$ is the task criticality level,
- $C_i(\text{LO})$ represents the worst-case execution time (WCET) of τ_i at criticality level LO, and
- $C_i(\text{HI})$ represents the WCET of τ_i at criticality level HI.

We assume that each task τ_i can emit an infinite sequence of jobs with period T_i and that the first job of all tasks is released at time 0. The relative deadline D_i of τ_i is equal to its period, i. e., $D_i = T_i$. Additionally, we assume for the WCET of a task τ_i at the two criticality levels:

- $C_i(\text{HI}) \geq C_i(\text{LO})$ if $\chi_i = \text{HI}$, i.e., the HI-level WCET estimate is more conservative than the LO-level WCET. This is because at a higher assurance level more stringent safety guarantees need to be provided.

- $C_i(\text{HI}) = 0$ if $\chi_i = \text{LO}$, i.e., at a high assurance level the execution of LO-criticality tasks can be ignored.

The actual execution time of each job of a task τ_i is not known. If the execution time of a job τ_i does not exceed $C_i(\text{LO})$ at runtime, we claim that the job is executed according to τ_i's LO-level profile, otherwise according to its HI-level profile.

Based on the system model, a mixed-criticality scheduling policy must guarantee that:

- If all jobs of the tasks in τ execute according to their LO-level profile, then *all* jobs receive enough resources between their release time and deadline to complete;
- If at least one job of a task in τ executes according to its HI-level profile, from that point onwards all jobs of HI-criticality tasks receive enough resources between their release time and deadline to complete. However, the service to jobs of LO-criticality tasks may be degraded.

In the remainder of the paper, we use the term *utilization* of a mixed-criticality task set τ, which is defined as in [2]:

$$U_\tau = \max\left(U_{\text{LO}}^{\text{LO}}(\tau) + U_{\text{HI}}^{\text{LO}}(\tau), U_{\text{HI}}^{\text{HI}}(\tau)\right), \quad (1)$$

where $U_{\text{x}}^{\text{y}}(\tau)$ represents the total utilization of the tasks with criticality level x for their $y-$level WCET estimates, i.e., $U_{\text{x}}^{\text{y}}(\tau) = \sum_{\chi_i=\text{x}} C_i(y)/T_i$.

B. Flexible Time-Triggered Scheduling

The Flexible Time-Triggered and Synchronization-based scheduling policy [1], henceforth denoted FTTS, is designed for global timing isolation among tasks of different criticalities, in a similar sense as partitions enforce isolation in the ARINC-653 standard [3]. Isolation is achieved by allowing only a statically known subset of tasks in τ with the *same* criticality level to be executed across the cores of a multicore platform at any time. Despite the non-negligible cost of inter-core synchronization for achieving global isolation, separating the execution of tasks temporally depending on their criticality level seems necessary for certification purposes. Otherwise, tasks of different criticalities could interfere (delay each other) on shared platform resources, e.g., memories and caches, thus violating the isolation property, which typically needs to be demonstrated/proven for certification.

FTTS is non-preemptive and combines time and event-triggered task activation. A global FTTS schedule repeats over a *scheduling cycle* equal to the hyper-period of the tasks in τ. The cycle consists of fixed-size *frames* (set $\mathcal{F}$). Each frame is divided further into two flexible-length *sub-frames*, the first containing HI-criticality tasks (HI sub-frame) and the second containing LO-criticality tasks (LO sub-frame). The beginning of frames and sub-frames is synchronized among all cores. Frames start at predefined time points. Within a frame, the HI sub-frame begins immediately. The LO sub-frame begins once all tasks of the HI sub-frame complete execution across all cores. This way LO-criticality tasks can never delay HI-criticality tasks, e.g., by blocking their access to shared resources such as a memory bus (isolation). Synchronization for switching from the HI to the LO sub-frame is achieved dynamically via a barrier mechanism, for efficient resource utilization. Within the sub-frames, tasks are scheduled sequentially on each core following a predefined order. The mapping of tasks to cores is fixed.

At runtime, the length of each sub-frame varies based on the different execution times of the tasks. We use $barrier(f, 1, \text{LO})$ (resp. $barrier(f, 1, \text{HI})$) to denote the worst-case length for the 1st (HI) sub-frame of frame $f \in \mathcal{F}$, when the tasks in it exhibit their LO (resp. HI)-level execution profile. Similarly, $barrier(f, 2, \text{LO})$ denotes the worst-case length for the 2nd (LO) sub-frame of frame f, when the tasks exhibit their LO-level execution profile. At runtime, the FTTS scheduler monitors for each frame the actual length of its sub-frames. If the length of the 1st sub-frame does not exceed $barrier(f, 1, \text{LO})$, it triggers normally the execution of tasks in the 2nd sub-frame (*LO mode*). However, if the length of the 1st sub-frame exceeds $barrier(f, 1, \text{LO})$, the 2nd sub-frame is not triggered since it is not guaranteed that the tasks in it can meet their deadlines (*HI mode*). A LO to HI mode switch depends, therefore, on the monitored length of the 1st sub-frame and it can affect the execution of LO-criticality tasks only for the current frame. The same procedure is repeated independently for the following frames.

The worst-case sub-frame lengths, namely function $barrier$, can be computed offline for an FTTS schedule according to the time analysis presented in [1]. The mixed-criticality schedulability condition (see Sec. III-A) is fulfilled if in every frame $f \in \mathcal{F}$ (with fixed length L_f), the 2nd sub-frame finishes by the end of the frame under all possible exhibited profiles of the tasks in f (under LO or HI mode) [1], i.e., $\forall f \in \mathcal{F}$:

$$\max\left(\begin{array}{l} barrier(f, 1, \text{LO}) + barrier(f, 2, \text{LO}), \\ barrier(f, 1, \text{HI}) \end{array}\right) \leq L_f, \quad (2)$$

The mapping of tasks to processing cores and the scheduling, i.e., the mapping of tasks to sub-frames and the execution order in each sub-frame, such that condition (2) holds, are optimized according to [1].

C. Partitioned EDF with Virtual Deadlines

The partitioned EDF with Virtual Deadlines [2], henceforth denoted pEDF-VD, is based on the EDF-VD scheduling policy, which was introduced for uniprocessors in [7]. In fact, EDF-VD is an adaptation of the classic preemptive EDF, where HI-criticality jobs of a task set τ are assigned reduced (virtual) deadlines. These are derived by multiplying the original deadlines with a common factor $x \in (0, 1]$, which is computed offline [7] as:

$$x = \frac{U_{HI}^{LO}(\tau)}{1 - U_{LO}^{LO}(\tau)}. \quad (3)$$

The uniprocessor EDF-VD scheduler works in two modes. It starts from *LO mode*, where all tasks are expected to run according to their LO-level profiles. Jobs are ordered based on earliest deadline first (EDF), using the original deadlines of LO-criticality jobs and the virtual deadlines of HI-criticality jobs. If at any time a HI-criticality job exceeds its LO-level WCET estimate, the scheduler switches to *HI mode*. In this mode, execution of LO-criticality jobs is skipped and HI-criticality jobs (incl. the active jobs at mode switch) are scheduled based on EDF, using their original deadlines. Note that the EDF-VD scheduler [7] does not switch from HI mode back to LO mode. In our work, however, we extend the scheduler such that when the CPU is idle for the first time after entering the HI mode, a switch to LO mode occurs.

The deadline down-scaling for the HI-criticality jobs in the LO mode is performed to promote their execution and ensure schedulability across the mode switch. The EDF-VD scheduler is guaranteed to provide enough resources for all jobs of task set τ in LO mode and for all HI-criticality jobs of τ in HI mode to meet their original deadlines, provided that the task

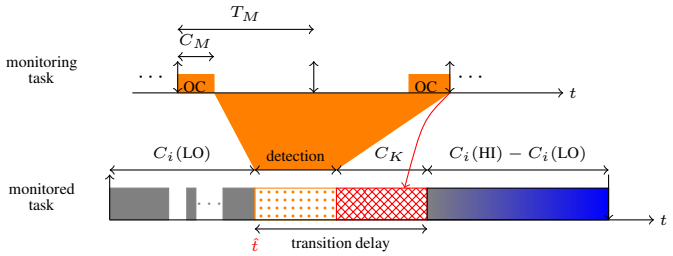


Fig. 1: Overrun detection and mode switching

set utilization U_τ (1) is not greater than 3/4 [7]. Namely, the *schedulability condition* for the uniprocessor EDF-VD is:

$$\max\left(U_{LO}^{LO}(\tau) + U_{HI}^{LO}(\tau), U_{HI}^{HI}(\tau)\right) \leq 3/4. \tag{4}$$

Note that (4) is a derived utilization-based test from the original EDF-VD test [7]. We adopt this in our later experiments (Sec. VII), since it is the standard test in case of partitioned EDF-VD [2]. Specifically, for partitioned EDF-VD scheduling on multicores, the tasks of τ are mapped to processing cores following a first-fit bin packing approach, such that the utilization of the task subset on every core does not exceed 3/4. If such a partitioning of tasks to cores can be found, then from condition (4) it follows that the mixed-criticality schedulability is ensured. The task subset on every core is scheduled based on the uniprocessor EDF-VD policy, while mode switches are performed, if needed, locally.

Note that, in contrast to FTTS, pEDF-VD can execute tasks of different criticality concurrently, which might interfere when accessing shared resources, e.g., memory. Therefore, for certifiability, pEDF-VD may need to be combined with additional partitioning mechanisms on the resource controllers, which however are outside the scope of this paper.

IV. Motivational Example

For mixed-criticality scheduling, accurate runtime monitoring of task execution and fast reaction to task overruns are of ultimate importance, as late detection or initiation of a mode switch can cause critical tasks to miss their deadlines. In this section, we motivate the need for light-weight runtime mechanisms by evaluating their effect on system schedulability.

A. Impact of Task Monitoring and Mode Switch Overhead

A popular way to implement runtime monitoring is by heartbeat monitoring, where a programmable timer periodically triggers the system monitoring task τ_M. This task checks the elapsed execution times of running tasks and activates system protection mechanisms in case of overrun. Note that the parameters of such a system monitoring task, e.g., period (deadline) T_M and WCET C_M, can have a strong impact on the system schedulability. Furthermore, such a monitoring task is a critical system service, on which other tasks depend, and hence it must always be assured itself to the highest criticality level in the system.

Fig. 1 depicts the monitoring task τ_M together with a task being monitored. We assume here that in case of overrun, all active LO-criticality tasks are terminated, which takes in the worst-case C_K to finish (the actual semantics of task termination depend on the implementation, which will be detailed in Section V). Let us assume that at time $\hat{t}$, the monitored task exceeds its LO-level WCET. The worst-case scenario happens when the monitoring task has just finished. Thus the overrun detection is delayed until the next activation of the monitoring task, which finishes at the end of the next monitoring period, at the latest. Therefore, the overrun is detected in the worst-case $2T_M$ after $\hat{t}$, assuming that the monitoring task is executed in zero time in the best-case. Following the overrun detection, it will take further C_K to terminate all LO-criticality tasks for the system to transit to HI mode.

During the transition phase (overrun detection & reaction), any task will continue to execute, thus possibly exceeding its LO-level WCET. To take this into account and preserve the validity of the theoretical model, one could integrate such a transition delay into established mixed-criticality schedulability analysis techniques [20], [21]. For ease of presentation here, we incorporate the transition delay by adding it to tasks' LO-level WCETs and focus on the single-core mixed-criticality scheduling policy EDF-VD. By considering the runtime monitoring and mode switch as discussed, we establish the following schedulability test result under EDF-VD.

Lemma IV.1. *Given a dual-criticality sporadic task set τ and a monitoring task τ_M, with the following utilization notations:*

$$U_i^\chi = \frac{C_i(\chi) + 2T_M + C_K}{T_i}, \quad \chi \in \{HI, LO\} \wedge \tau_i \in \tau,$$
$$U_{\chi_1}^{\chi_2} = \sum_{\tau_i \in \tau_{\chi_1}} U_i^{\chi_2} \text{ and } U_M = \frac{C_M}{T_M}, \tag{5}$$

the system is schedulable under EDF-VD if:

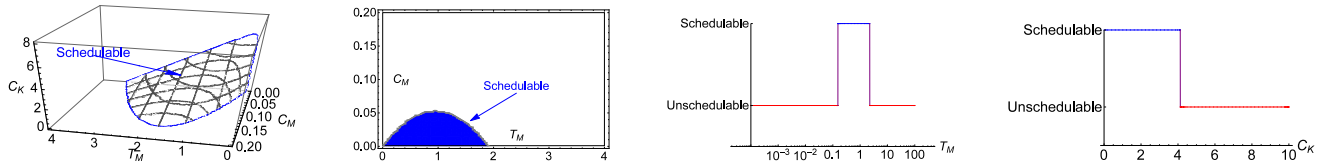
$$\max\left(\begin{array}{l} U_{HI}^{LO} + U_{LO}^{LO} + U_M, \\ U_{HI}^{HI} + U_M + \frac{U_{HI}^{LO} + U_M}{1 - U_{LO}^{LO}} \cdot U_{LO}^{LO} \end{array}\right) \leq 1. \tag{6}$$

Proof: The monitoring task needs to be always guaranteed and treated as a HI-criticality task. In addition, its LO- and HI-level utilization is always $\frac{C_M}{T_M}$. Thus, we can first derive that as long as $U_{\text{HI}}^{\text{LO}} + U_{\text{LO}}^{\text{LO}} + U_M \leq 1$, the system is schedulable in LO mode. Now, we can "tighten" the LO scheduling mode by shortening the deadlines of HI criticality tasks to ensure the HI mode schedulability. According to EDF-VD (Theorem 1 in [7]), we choose the minimal deadline shortening factor $x = \frac{U_{\text{HI}}^{\text{LO}} + U_M}{1 - U_{\text{LO}}^{\text{LO}}}$ to guarantee the maximum schedulability in HI mode. For the selected factor x, schedulability condition (6) follows from Theorem 2 in [7]. Notice that (6) is derived according to the original EDF-VD test [7] rather than the utilization-based test in (4). ■

Using Lemma IV.1, we can evaluate the impact of the monitoring and mode switch mechanisms on system schedulability. We illustrate this with the following example.

Example IV.1. *We consider randomly generated periodic tasks by a task set generator similar to the one proposed in [15]. One generated task set is adopted and the tasks therein have periods in the range from 2ms to 2s, with the system utilization bound U_τ (1) being 0.8. To evaluate the impact of different implementation choices, we regard T_M, C_M and C_K as variables depending on the actual implementation. We summarize our results in Fig. 2, where we can observe the following:*

- *The overhead of task monitoring and termination are critical to the system schedulability. As shown in Fig. 2a and 2b, all parameters need to be chosen carefully to make the system schedulable.*
- *For task monitoring, a higher frequency is not necessarily better, as it implies a higher utilization for the monitoring task. On the other hand, a low monitoring frequency increases the delay for overrun detection. This trade-off is depicted in Fig. 2c.*



(a) 3D contour plot of feasible region (b) 2D contour plot of T_M, C_M (c) Impact of monitoring period T_M (d) Impact of termination overhead C_K

Fig. 2: Evaluation of the impact of runtime overheads on system schedulability: All task parameters are in units of ms. In Fig. 2b, $C_K = 3$ms. In Fig. 2c, $C_M = 20\mu$s and $C_K = 2$ms. In Fig. 2d, $T_M = 1$ms and $C_M = 20\mu$s. The selected parameters are based on measurements on the Intel®Xeon Phi coprocessor, for which C_M lies in the range [8,20] μs and C_K for a single task in the range [12,20] μs.

- *The task termination overhead is crucial for the system schedulability. In Fig. 2d the system is unschedulable when the total termination overhead is greater than 4.1 ms. Intuitively, a slow task termination process leads to delayed reaction to task overrun, hence to missed deadlines for HI-criticality tasks. Note that the termination process may indeed incur a significant overhead for task sets with a large number of LO-criticality tasks, since termination requires to clean all the footprints of these tasks.*

B. Discussion

As shown by our preliminary experiments in Example IV.1, the implementation choices of runtime mechanisms and their incurred overheads play a critical role for the schedulability of mixed-criticality systems. Therefore, such mechanisms must be carefully designed to be light-weight. This holds for both single-core and multi-core implementations. Note that in this section we focused on scheduling policies based on runtime monitoring, such as pEDF-VD. However, also in partitioning-based policies such as FTTS, the overhead from the partition switching, overrun detection, etc. can impact the system schedulability and must be considered in their respective analysis. We review qualitatively different implementations choices for the above mechanisms in the following section.

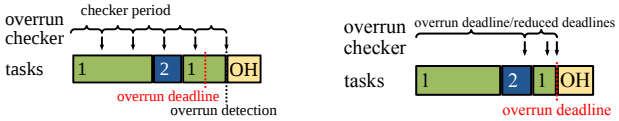
V. Implementation of Mixed-Criticality Mechanisms

So far, we have introduced mechanisms for multicore mixed-criticality scheduling, incl. runtime monitoring, task termination, inter-core barrier synchronization and scheduling mode switching. In this section we propose possible implementation approaches for these mechanisms in the user-space and highlight their advantages and disadvantages, using the FTTS [1] and pEDF-VD [2] schedulers as motivating examples.

A. Runtime Monitoring

Runtime monitoring is necessary to enable a mixed-criticality scheduler to react to HI-criticality tasks overrunning their LO-level WCETs. Because monitoring each individual task can significantly increase the system utilization, it is important to design this mechanism for fast overrun detection, yet at low overhead. A comparison of the two discussed monitoring methods in this section is shown in Fig. 3.

Heartbeat Monitoring A straightforward, commonly adopted runtime monitoring method is heartbeat monitoring. As presented in Sec. IV, a separate overrun checker thread checks periodically (with period T_M) if a monitored thread executes beyond its overrun deadline, D_o. This deadline is defined as the LO-level WCET $C_i(\text{LO})$ of the monitored task for EDF-VD or the LO-level worst-case length of a HI subframe ($barrier(f, 1, \text{LO})$ for frame f) for FTTS. In case of overrun, the monitoring thread initiates the scheduler-specific actions to handle it.



(a) Heartbeat monitoring (b) Deadline-based monitoring

Fig. 3: Comparison of the overrun checker activations for two monitoring techniques. OH denotes the overrun handling.

The advantage of heartbeat monitoring is the predictability w.r.t. its overhead, because checking (polling) is performed periodically. Assuming that the overrun checker can check the monitored task immediately after activation (in contrast to the assumptions made in Fig. 1, our overrun checker thread runs with higher priority than the tasks, see Sec. VI), overrun detection delay is, at most, T_M. The selection of the monitoring period is a trade-off between faster overrun detection and higher runtime overhead.

Deadline-Based Monitoring To reduce the monitoring overhead, we introduce an improved non-periodic monitoring method, based on the known overrun deadline D_o. At the start of a new task, the overrun checker initializes a timer with the relative deadline D_o. At wakeup, it checks the task's current execution time, C_{cur}. If the task is executing beyond the overrun deadline, i.e., $C_{\text{cur}} > D_o$, the scheduler-specific actions to handle the overrun are initiated. Otherwise, the monitoring thread enters a new sleep phase with the reduced sleep duration:

$$C_{\text{sleep}} = D_o - C_{\text{cur}}, \tag{7}$$

which represents the minimum time before the task overruns the deadline D_o. If the monitored task finishes before D_o, it deactivates its overrun checker upon task completion.

In contrast to heartbeat monitoring, this approach checks for an overrun only when it is possible to occur. Also, since the checks are performed at the overrun deadline, overruns are detected without delay. However, this method runs into problems when the task is preempted close to the deadline, i.e., when C_{sleep} becomes very low. Then it is possible that the overrun checker checks for an overrun so frequently that it always preempts the monitored task. This can increase significantly the system utilization, thus compromising schedulability.

To circumvent this problem, we introduce a lower bound $\hat{T}_M$ for the sleep delay update, such that:

$$C_{\text{sleep}} = \max(D_o - C_{\text{cur}}, \hat{T}_M). \tag{8}$$

With this addition, the worst-case monitoring utilization becomes equal to that of heartbeat monitoring with period $\hat{T}_M$. Consequently, the overrun detection delay will vary from zero to $\hat{T}_M$.

Note that yet another alternative, which would incur only one wakeup of the checker at deadline overrun, would be

(a) Immediate task termination at overrun before switching to HI mode.

(b) Deferred task termination after executing HI-criticality tasks.

Fig. 4: Two different task termination approaches. "T" denotes the task termination of LO-criticality tasks.

to calculate C_{sleep} according to (7) whenever the monitored thread is preempted and restore the value when the thread execution is resumed. In addition, there are other alternatives to monitor threads, such as custom call-back functions, kernel modules or patches. While these methods could bypass some of the limitations of our generic user-space implementation, in the present paper they are not used due to overhead, implementation complexity and loss of portability.

B. Scheduling Mode Switch

Several mixed-criticality schedulers use different modes for scheduling either all tasks (LO mode) or only HI-criticality tasks (HI mode), after a task overrun is detected. Transition between these modes depends on the particular scheduler and should be optimized to enable a quick reaction to overruns.

FTTS Scheduler For FTTS, a mode switch may occur at sub-frame boundaries between HI and LO sub-frames. The switch to HI mode is needed, when the LO-level worst-case sub-frame length ($barrier(f, 1, \text{LO})$ for frame f) is overrun. In this case, the following LO sub-frame is skipped, by dropping all its tasks before starting their execution. The scheduling mode is reset to LO at the next frame switch.

pEDF-VD Scheduler All single-core EDF-VD schedulers of the pEDF-VD scheme maintain their own scheduling mode. Depending on the mode, a different subset of their tasks is scheduled according to EDF. We propose two approaches to manage the active task queue on each core. In the first alternative, a single queue is used for either LO or HI mode. Queue updates have a moderate overhead in LO mode, but at mode switches the overhead increases because a complete queue update (removal of LO-criticality tasks) is required. In the second alternative, two separate task queues are maintained in parallel on each core, i.e., one for each scheduling mode. This allows fast mode switching by just switching the considered task queue, but it adds overhead to the queue update in LO mode since both queues need to be updated. Common for both approaches is the termination of LO-criticality task when switching to HI mode, which is discussed below. In our framework, we adopt the double-queue approach to enable mode switch at a minimum delay after overrun detection. Further discussion and empirical evaluation of the two implementation alternatives can be found in [22].

C. Task Termination (EDF-VD)

For fast mode switching, termination of active LO-criticality tasks is required by the EDF-VD scheduler upon switching from LO to HI mode. Immediate, abrupt killing of a task is usually not appropriate, as it is desirable to allow a reasonable shut-down period for the task termination (clean-up operations, etc.). In our implementation, we use the immediate termination, though, for simplicity. We present two different ways to handle termination, which are illustrated in Fig. 4.

Immediate Termination One approach is to terminate a task and wait until this process (memory deallocation, removal

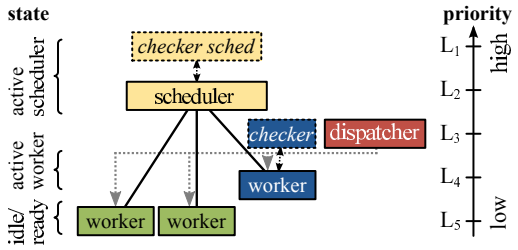


Fig. 5: Priority management of the framework to schedule tasks from user-space [27, Extended Fig. 5].

from task queue) has finished, before continuing with the next scheduling action. This method guarantees that the task will never execute before its next job arrives, however it may incur considerable overhead for a large number of active LO-criticality tasks at mode switch. Also, bounding the overhead of immediate termination in the user-space is notoriously challenging.

Deferred Termination If temporary deactivation of active tasks and blocking of new job arrivals is acceptable, task termination can be deferred to a later point to reduce the mode switching delay. In this case, the active LO-criticality tasks are deactivated by lowering their priority (infinite deadline) and new job arrivals are no longer inserted into the task queue. The actual task termination is performed when the HI-criticality tasks have been executed and the CPU becomes idle for the first time. Note that after completing task termination, the EDF-VD scheduler can switch back to LO mode.

D. Inter-Core Synchronization (FTTS)

Another mechanism required for global scheduler techniques, such as FTTS, is the synchronization of tasks across all cores to make global decisions. The framework presented in the next section runs in the Linux user-space and uses the pthread library [23] to manage different threads. For portability purposes, we select the barrier synchronization method that this library provides (`pthread_barrier_wait()`). The GNU libc implements this barrier using futexes (fast user-space mutexes) [24], an efficient sleep-based locking mechanism. Several platform-specific alternatives exist, which may lead to reduced overhead. As examples we mention the specialized barrier implementations for x86 [25] and the Kalray MPPA-256 [26]. More alternatives can be considered depending on the target platform for a mixed-criticality system.

VI. Experimental Set-up

This section presents the scheduling framework that was developed for the empirical evaluation and comparison of mixed-criticality mechanisms for multicore scheduling. The target platforms for this evaluation are Intel® Xeon Phi and Intel® Core i5, whose relevant features are described in Sec. VI-B. In Sec. VI-C, we detail all individual runtime overheads and their measurement method, for each implemented scheduler in the framework.

A. Multicore Mixed-Criticality Scheduling Framework

The mechanisms discussed in Sec. V were used to extend the existing SF3P framework [27] for multicore and mixed-criticality scheduling. To provide rapid prototyping and testing of different scheduler implementations, the framework operates in the user-space of any POSIX-compliant operating system. As a result, its fast reconfigurability and portability allow us to easily test different policies under different architectures.

While there is a performance loss when compared to some kernel-space implementations, our study still offers key insight to the effects runtime overheads have on schedulability. In the following, we highlight our extensions to the SF3P and the integration of the mixed-criticality mechanisms of Sec. V.

User-Space Scheduling The framework consists of different threads for dispatching, scheduling, monitoring, and task execution. These threads are scheduled from the user-space using a priority-based kernel scheduler. Different priorities allow to activate, deactivate or preempt the threads. Fig. 5 gives an overview of the different priorities used in the framework. The worker threads are used to manage execution of the real-time tasks. Note that both the scheduler and task overrun checkers run with higher priority than the threads they monitor. This is important to guarantee that the overrun checker can always handle an overrun before the monitored thread continues its execution.

Multicore Support To support multicore scheduling, the framework has been extended with core mapping support. The POSIX standard [23] does not specify an interface to set processor affinity of threads, but its Linux implementation provides an extra interface for processor affinity. The use of this solution restricts portability to systems running a Linux kernel. However, the advantage of seamless integration with the pthread library and the wide range of target systems that are capable of running a Linux kernel outweighs this disadvantage.

Timed Sleep and Execution Time Tracing For the implementation of the overrun checkers and dispatchers, it is important to have an accurate timed sleep primitive. The POSIX standard provides the `clock_nanosleep` interface, which allows to define sleep intervals with an accuracy of up to 1 ns. However, the real sleep delay highly depends on the platform used: the empirical analysis of the sleep delay in Sec. A of the Appendix shows that the smallest sleep delay for the Xeon Phi platform lies in the region of 10 ms and for the Core i5 in the region of few microseconds. This large discrepancy between the specified and the actual sleep interval can increase the overrun checking period and task execution time, which may cause scheduling mode switches at runtime.

POSIX provides also an interface to access timers with an accuracy of 1 ns; the real resolution again depends on the platform and its clock speeds. The clock accesses are handled over a system interface, which introduces some non-negligible overhead. Since fast and accurate timing is crucial for the task execution tracing and overrun deadline set-up, our framework supports a second timer access method. Particularly, the x86 instruction set provides the time stamp counter instruction [28], which allows reading a timestamp with a single processor instruction and with accuracy of few (order of tens) CPU clock cycles. We use this method on our target platforms, while unsupported platforms fall back to the POSIX interface.

Scheduler Implementation With the timed sleep, multiple thread priority levels and core mapping support, we are able to implement our schedulers and the selected mechanisms of Sec. V.

FTTS is implemented as a separate scheduling thread. The initialization of the first task and the completion of the last task per sub-frame across all cores are handled globally by this thread. That is because distributed sub-frame initialization can lead to a race condition, where a task of an already started sub-frame on one core interferes with the scheduler on a core that is still starting the sub-frame. The global task management may incur high inter-core communication overhead, but is required in our framework to completely isolate the execution of the scheduler and the tasks in the FTTS sub-frames. Communication between the worker and scheduler threads is based on shared memory and locking mechanisms like mutexes and semaphores. The GNU libc implementation of these mechanisms is, in turn, futex-based [24]. The FTTS scheduler thread along with its associated overrun checker (for sub-frame overruns, see Fig. 5) are mapped to the same processing core, which unless otherwise mentioned, does not host any worker threads.

For pEDF-VD, the global task set is initially partitioned as in Sec. III-C. The partitioned task sets are handed over to the EDF-VD schedulers on the corresponding cores. The single-core scheduler threads and the per-task overrun checkers execute then without explicit inter-core communication. Note that because of the decentralized scheduler design, the total overhead of the pEDF-VD scheduler is expected to be significantly lower than for FTTS on massively parallel architectures.

B. Target Platforms

Overhead evaluation of the implemented mixed-criticality schedulers is performed on the manycore Intel® Xeon Phi Coprocessor 5110P [6]. Xeon Phi features 60 processing cores, which allows to explore the scalability of the scheduling strategies (for the first time on such scale), and is compatible with the x86 instruction set architecture. The cores are connected to a shared bidirectional ring bus and have private L1 and L2 caches. Full L2 cache coherency is provided by globally distributed tag directories. Each processing core supports up to 4 threads by simultaneous multi-threading. In our measurements, we employ maximally one of these 4 logical cores to ensure that no inter-thread interference occurs. The used Xeon Phi runs the Intel Manycore Platform Software Stack version 3.1, which is based on Linux kernel 2.6.38.8. For compilation of the framework, we used the Intel C++ Composer 2013 SP1 Update1 (14.0.1), with optimization level `-O2` along with the GNU libc v2.14.

To compare the overhead of the underlying system, a commodity desktop platform is also used, with an Intel® Core i5-4670 processor running a Linux kernel version 3.13.0-37. On this platform, the framework was compiled using GCC 4.8.2, with optimization level `-O2` and the GNU libc v2.19. For all experiments, the operating system's runlevel was lowered to 1 so that only essential system services are running.

C. Overhead Estimation

In the following we introduce various measured scheduler overheads. Because the FTTS and pEDF-VD schedulers differ in core features, we need to distinguish between their overhead components.

FTTS Scheduler The overhead of the FTTS scheduler is divided into the following components.

- *Frame initialization:* The global scheduler overhead for switching between FTTS time frames, which checks if the previous frame has overrun its duration, i.e., if the LO sub-frame finishes after the end of the frame, and loads the HI sub-frame of the next frame. Note that if a "frame overrun" occurs, the system is no longer guaranteed to be schedulable.
- *Sub-frame initialization:* The overhead of the scheduler logic that handles start and completion of tasks in a sub-frame. In this phase, the scheduler checks if a task is ready for execution, it sets up the overrun checker and configures the barrier synchronization to synchronize all cores after task execution. Since the scheduler needs to interact with all worker threads for this, sub-frame initialization is expected to be one of the main overhead contributors.

- *Barrier synchronization:* The overhead for the inter-core synchronization, which is used to make global decisions, e.g., for mode switching. The synchronization overhead can have a considerable impact on the total overhead for architectures with (i) no hardware support for barrier synchronization and/or (ii) complex communication protocols, such as the Xeon Phi.
- *Sub-frame overrun:* The overhead of reacting to sub-frame overruns, i.e. when a sub-frame executes beyond its statically defined LO-level worst-case length. This overhead depends on the frequency of overruns and equals zero if no sub-frame overrun occurs.
- *Job arrival:* The overhead for signaling and registering the arrival of a new job in a bitmap data structure, maintained by the global scheduler. The total overhead depends on the task set parameters, i.e., the number of periodic job arrivals within a scheduling interval.
- *Job finish:* The overhead for signaling the completion of a job to the scheduler and for clearing the corresponding ready flag in the bitmap structure.

pEDF-VD Scheduler The overheads of the pEDF-VD scheduler are listed below.
- *Overrun monitoring:* The overhead for monitoring every HI-criticality task, which enables the scheduler to react to runtime overruns. The implementation choice of deadline-based monitoring (see Sec. V-A) is expected to keep this overhead low.
- *Job overrun:* The overhead for handling a runtime overrun, i.e., when a HI-criticality task executes beyond its LO-level WCET. The implementation choices of deferred termination and maintaining two active task queues (see Sec. V-C, V-B) are expected to keep this overhead low.
- *Job arrival:* The overhead for registering the arrival of a new job with the local scheduler. This includes the update of the active task queue(s), which is (are) sorted according to EDF.
- *Job finish:* The overhead for removing a task from the active task queue(s) upon job completion. Note that a special case occurs when tasks are terminated (deactivated) at mode switch. Because our pEDF-VD implementation uses deferred termination, the task termination is also handled during the finish sequence of a task and hence, accounted for in this overhead.

Scheduler Overhead The total scheduler overhead is calculated as the average or maximum (as specified) of the per-core total overheads. The global overheads in FTTS (frame and sub-frame initialization, barrier synchronization and sub-frame overrun) are accounted once for each core.

System Overhead We use this term to summarize all measured overheads that are not part of framework threads, like the overhead for thread switches.

VII. Evaluation

We present in this section our evaluation results: (i) to get insights into the runtime overheads of the two multicore mixed-criticality schedulers in our user-space framework, (ii) to evaluate empirically the effect of the runtime overheads on system schedulability, and (iii) to compare the scheduler and system overheads for a real-world application (flight management system) on two commercial multicore platforms.

A. Runtime Overhead of FTTS and pEDF-VD Schedulers

First, we evaluate how the runtime overheads of FTTS and pEDF-VD, which are detailed in Sec. VI-C, vary with increasing system utilization and available processing cores on Xeon Phi, given the implementation choices of Sec. V. For this, we use synthetic task sets.

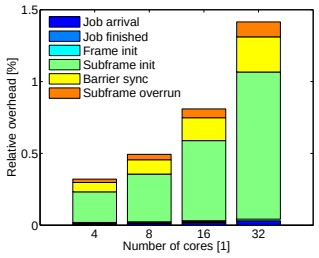


Fig. 6: FTTS overhead for task sets with overrun probability $30\%, U_\tau/m = 0.2$

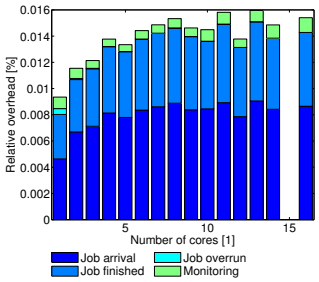


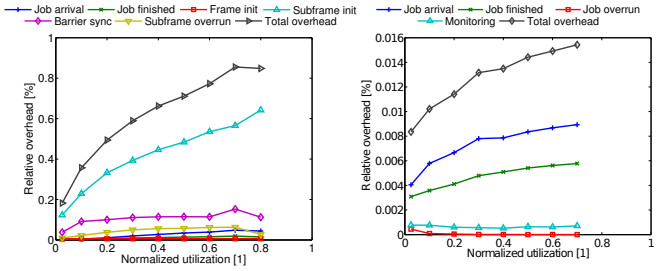
Fig. 7: pEDF-VD overhead for task sets with overrun probability 50%

The task set generation is based on the algorithm of [15]. That is, we fix the task set utilization U_τ and repeatedly add tasks to an initially empty set until U_τ is met. We consider task set utilizations U_τ in the range $[0.2, 32]$. The utilization U_i of a single task is selected uniformly from $[U_L, U_H] = [0.05, 0.5]$ and the ratio Z_i of the HI- to LO-level utilization is selected uniformly from $[Z_L, Z_H] = [1, 4]$. The probability that a task τ_i has $\chi_i = \text{HI}$ is set to $P = 0.4$. Task period T_i is randomly selected as a multiple of 100ms within $[200, 800]$ ms, such that the resulting task sets have harmonic periods (e.g., a task set with periods in $\{200, 400, 800\}$). Under the assumption of harmonic periods, FTTS and pEDF-VD are expected to have comparable schedulability based on the results of [1] and the tasks' hyper-period is constrained, which facilitates the optimization of the FTTS schedules. 100 task sets for each considered utilization are generated for our experiments, in which the tasks perform busy waiting for their LO- or HI-level WCET.

To evaluate the schedulability of a task set under pEDF-VD, we perform task partitioning following the first-fit bin packing approach of [2]. If a partitioning that satisfies condition (4) is found, the task set is deemed pEDF-VD-schedulable. For FTTS we assume a scheduling cycle with period equal to the hyper-period of the tasks in τ and fixed frame lengths equal to the greatest common divisor of the periods. We then search for a feasible task mapping and schedule as proposed in [1].

The generated schedules are executed for 10 seconds each in our user-space framework. Fig. 6 (FTTS) and Fig. 7 (pEDF-VD) show the relative scheduler overhead, i.e., the overhead divided by the execution interval (10 s), for different number of cores on Xeon Phi. The depicted overhead is the average per-core. This choice is made mainly for illustration purposes here, although for schedulability analysis one would have to consider the maximum per-core overhead (see Sec. VII-B). For FTTS, the normalized utilization was fixed to 0.2, while the number of cores was limited to powers of 2 for optimization purposes. For pEDF-VD, the number of cores employed after partitioning are shown. Note that only feasible schedules, which complete without missed deadlines, are considered in the graphs. For FTTS on 8 cores for example, we have 99 feasible schedules out of 100 executed at $U_\tau = 0.2$, 61 at $U_\tau = 3.2$, and 0 for $U_\tau \geq 7.2$ (see Sec. VII-B). Note also that each HI-criticality task is assigned a certain overrun probability (see captions of Fig. 6, 7), based on which each job can execute for its HI-level WCET, thus triggering a scheduling mode switch.

According to the FTTS results (Fig. 6), all measured overheads increase with the number of cores, but the total overhead never surpasses 1.5% of the execution interval. For our task generation, more cores imply more generated tasks, which need to be handled at e.g., sub-frame initialization and overrun. Note that the sub-frame initialization dominates the



(a) FTTS, 8 cores (b) pEDF-VD, 8 cores

Fig. 8: Scheduler overheads for increasing utilization.

total overhead, since all tasks of a sub-frame need to be activated at that point by the global scheduler. As the number of tasks increases, so does the job arrival overhead, since more arrivals need to be signaled to the scheduler. Moreover, we observe that the barrier synchronization is not the main bottleneck in terms of runtime overhead, as it represents a smaller portion of the total overhead compared to the sub-frame initialization.

The relative overheads of the pEDF-VD scheduler also tend to increase with the number of cores, but the total overhead is bounded by just 0.016% per core in the average case, as shown in Fig. 7. The overhead increase is expected, since the number of scheduled tasks increases with the number of cores and more jobs need to be handled. The dominating overheads for the pEDF-VD scheduler are those associated with job arrival and finish, because any local EDF-VD scheduler needs to update its active task queues (for HI and LO mode) at these events. Furthermore, we observe that the monitoring cost does not contribute much to the total runtime overhead for pEDF-VD, being almost steady while the number of cores increases. This can be explained in that after partitioning, HI-criticality tasks are distributed relatively evenly across the cores, such that the monitoring overhead is also distributed. Finally, note that with the implementation of deferred termination, overrun handling is very fast (hardly appearing in Fig. 7).

The analysis of the scheduler overhead for a fixed number of cores (here, 8) and varying utilization is shown in Fig. 8. Results for 16 and 32 cores can be found in Sec. B of the Appendix. Because the number of tasks increases (roughly) proportional to the utilization, most runtime overheads increase accordingly. For the FTTS scheduler, these overheads are the job arrival and finish, as well as the sub-frame initialization. Even though we expected the barrier synchronization to be constant for a fixed number of cores, it increases slightly for higher utilizations. This is because for lower utilizations, only a subset of the cores may need to be synchronized (not all cores are utilized in every sub-frame). The sub-frame overrun overhead increases as well, since more tasks need to be dropped at a mode switch, but it is kept on average below 0.1%. For pEDF-VD, the job arrival and finish overheads increase linearly. The results also show that the job overrun overhead is minimal (close to 0%) for all utilizations. The monitoring overhead for HI-criticality tasks increases for high utilizations, but it is kept on average below 0.001% in our 8-core system.

In our experiments, the total relative scheduling overhead reaches up to 1.42% for the FTTS scheduler on 32 cores and up to 0.016% for the pEDF-VD scheduler on 16 cores. The higher overhead of FTTS is mainly attributed to the global scheduler design. Namely, the sub-frame management by the global scheduler introduces high inter-core communication overhead on Xeon Phi, where all caches need to be kept coherent. pEDF-VD, in contrast, can maintain all task queues in the local caches. Recall, nonetheless, that in systems where global timing isolation among criticality levels is required for certifiability, FTTS ensures this property by allowing only tasks of the same criticality to be executed in parallel. pEDF-VD would require additional mechanisms to ensure isolation when accessing shared resources. The overhead of such mechanisms should not be neglected, although it has not been addressed in our study.

TABLE I: Barrier Sync. & Sub-frame Init. Overhead

Cores	Avg. Overhead [ms]	Max. Overhead [ms]
4	1.7	45
8	2.5	126
16	3.2	143
32	5	160

B. *Effect of Mixed-Criticality Overhead on Schedulability*

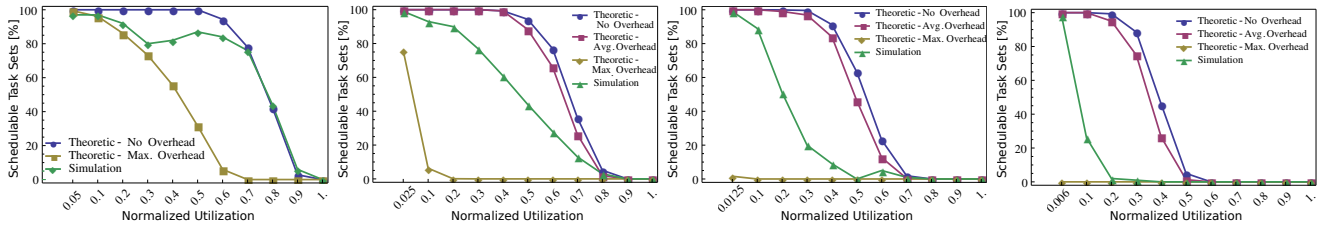
Second, we evaluate the effect of the runtime overhead of the user-space mixed-criticality implementation on the ability to find schedulable solutions under FTTS and pEDF-VD.

FTTS Schedulability Initially, based on the previously measured barrier synchronization and sub-frame initialization overhead for the FTTS schedules on 4, 8, 16 and 32 processing cores of the Xeon Phi, we derive the average and maximum absolute overhead as shown in Table I. Using the same synthetic task set generation as in the previous subsection, we generate 1000 task sets and apply the scheduling optimizer of [1] to determine how many task sets are FTTS schedulable. The scheduling optimization is performed under three configurations: assuming (i) zero synchronization and sub-frame initialization overhead (original optimizer), (ii) the average measured overhead, and (iii) the maximum measured overhead from Table I. In the last two configurations, the overhead is added to the sum of the worst-case sub-frame lengths within each FTTS frame f, such that the schedulability condition (2) is converted to:

$$\max \left(\begin{array}{l} barrier(f,1,\mathrm{LO}) + barrier(f,2,\mathrm{LO}), \\ barrier(f,1,\mathrm{HI}) \end{array} \right) + o \leq L_f, \quad (9)$$

where o denotes the considered sum of the barrier synchronization and sub-frame initialization overheads. Fig. 9a-9d show the fraction of task sets that are deemed FTTS schedulable under the three alternative overhead assumptions (zero/avg/max) as a function of the normalized utilization U_τ/m, with m the number of available cores. A fourth curve represents the empirical FTTS schedulability, namely the percentage of task sets which did *not* violate the schedulability condition (2) in practice when their FTTS schedule was executed for 10 seconds on Xeon Phi. Note that only theoretically schedulable (under zero overhead) task sets were selected for execution (max. 100 per utilization bound), so the empirical schedulability curve could never surpass the theoretic.

The divergence between the theoretic (zero overhead) and the empirical schedulability curve is significant, esp. for increasing number of available cores. The difference between the two curves starts from 0% and reaches 20% for 4 cores, 50.3% for 8 cores, 82.2% for 16 cores, even 97.3% for 32 cores. The theoretic curves considering the sub-frame overheads also fail to reflect closely the empirical results. On one hand, the maximum overhead assumption yields overly pessimistic estimations because in practice the worst-case measured overhead does not occur for every sub-frame switch (note the large discrepancy between avg. and max. estimation). Consequently, under this assumption and for cases with 8 or more cores, FTTS schedulability drops to zero even for very low task set utilizations, although in practice there are FTTS schedules which do not violate the mixed-criticality timing requirements.



(a) 4 cores (b) 8 cores (c) 16 cores (d) 32 cores

Fig. 9: FTTS Schedulability: Theoretic vs. Theoretic accounting for avg. / max. synchronization overhead vs. Empirical.

On the other hand, the average overhead assumption yields too optimistic estimations as expected, since several other overheads, incl. system overheads, are not accounted for when checking condition (9).

The prohibitively large worst-case measured overheads of Table I along with the inconsistency between the low relative scheduling overhead in Sec. VII-A and the significant schedulability loss here lead us to the conclusion that the user-space implementation of inter-core synchronization and communication on Xeon Phi may not have been a reliable choice to evaluate the FTTS scheduler performance. This is because synchronization involving up to 32 cores on a platform, which is not designed for hard real-time systems and which employs complex, non-predictable communication and global cache coherence protocols, without special support for barriers, cannot be trivially bounded. This reflects our current understanding of the large variance in the overheads of Table I. Investigation of the reasons for this variance on the Xeon Phi co-processor is, however, outside the scope of this paper. As future work, we intend to explore real-time implementations for barrier synchronization [29]–[31] and evaluate the FTTS scheduler on manycores with special OS and hardware support for barriers, such as the Kalray MPPA [26]. We expect that for barrier implementations with verifiably bounded and low overheads, the schedulability loss will be significantly lower, but also the theoretic schedulability results under overhead assumptions will closely follow the empirical ones.

pEDF-VD Schedulability Interestingly, for the pEDF-VD schedules of the previous subsection the absolute scheduler and system overheads had a minimal effect on system schedulability. Specifically in our experiments, the theoretic and empirical schedulability curves coincide in all cases. As possible explanations we identify (i) the lack of explicit inter-core communication due to the partitioned scheduling, which reduces drastically the runtime overhead compared to FTTS, (ii) the fact that only theoretically schedulable task sets were executed in our framework, where partitioning had been based on the rather conservative schedulability condition (4).

C. Scheduling of an Avionics Application

Last, we demonstrate the benefits of using our user-space framework in early phases of system design. We consider an industrial implementation of a flight management system and two available platforms, i.e., the Xeon Phi coprocessor and a Core i5 desktop. We assume that global timing isolation is a necessary feature of the implementation, hence FTTS is deployed for scheduling the application. Based on this, we need to decide which platform to use, by evaluating the relevant implementation overheads.

The flight management system (FMS) is an avionics application responsible for aircraft localization, flightplan computation for the auto-pilot, detection of the nearest airport, etc. We look into a subset of 11 periodic tasks, responsible for sensor reading, localization, and computation of the nearest airport. Five are characterized by safety level DAL-B (we map it to HI criticality level) and six by safety level DAL-C (we map it to LO criticality level) based on the DO-178B standard [32]. The periods of the tasks vary among 200 ms, 1 sec, and 5 sec. Based on these, we select the cycle and the frame length of the FTTS as $H = 5$ sec and $L_f = 200$ ms, respectively. The WCET of the tasks were derived based on measurements on the Xeon Phi and the Core i5, respectively, see Sec. C of Appendix. For HI-criticality tasks τ_i, we set the HI-level WCET estimates $C_i(\text{HI})$ to 10ms, which is significantly greater than the worst observed execution time. The task periods, criticality levels, LO- and HI-level WCET on both platforms are shown in Table III of the Appendix. FTTS schedules were generated for 4 processing cores, using the optimization framework of [1]. Both optimized schedules are feasible, i.e., fulfilling the FTTS schedulability condition (2).

TABLE II: Overhead analysis for the FMS application.

Overrun Trigger	Platform	Scheduler Overhead [%]	System Overhead [%]	Total Overhead
Memory allocation	Xeon Phi	0.36	$1.74 \cdot 10^{-4}$	0.36
	Core i5	0.03	4.90	4.93

To trigger runtime overruns, we add an allocation of 10 1MB-memory blocks to the code of the HI-criticality tasks. This choice is made because we empirically observed that the memory allocation increases the tasks' execution time beyond their LO-level WCET, yet never beyond 10 ms (HI-level WCET) on both platforms. This additional code is executed with probability 50% for each job at runtime. The measured overheads for executions on 4 cores and both target platforms are shown in Table II. The many but simpler execution cores on Xeon Phi incur an increased scheduler overhead. On the other hand, on Core i5 a lower scheduler overhead is achieved at the cost of a high system overhead. To some degree, this overhead stems from the standard Linux kernel (expensive context switches), over which the scheduling framework is running and which is not designed for real-time applications. In this aspect, the optimized and light-weight software stack of the Xeon Phi shows its strengths. Additionally, on Core i5 the scheduler and overrun checker threads share a core with several worker threads. This increases the number of context switches in the framework, resulting in higher system overhead.

Based on the results, we would choose the Xeon Phi platform for executing the FMS. Even with a higher scheduler overhead, the very low system overhead for 4 cores results in much lower total overhead compared to the Core i5. With this experiment we show that it is possible to schedule real-world applications from our user-space framework for rapid prototyping and testing. We can evaluate the overhead for different platforms and decide which is best for the given application. Similarly, we can use the framework to compare different scheduling algorithms for the same platform or alternative implementations of the same scheduler.

VIII. Conclusion

In this work, we have discussed alternative solutions for the implementation of common mixed-criticality mechanisms for multicores, incl. task monitoring, global and partitioned scheduling mode switch, inter-core synchronization. As motivational examples, we used two state-of-the-art scheduling policies, one representing the industrial practice of partitioning (FTTS) and one representing the most recent research efforts which are based on task monitoring and dynamic scheduling adaptations (pEDF-VD). The evaluation of a set of the implemented mechanisms on top of Linux, on the manycore Xeon Phi and a commodity 4-core Core i5 showed clearly that the runtime overheads of mixed-criticality schedulers, esp. global ones, should not be overlooked, since for a user-space implementation and a large number of cores they can have a tremendous impact on system schedulability. Our concluding thesis is that selecting a scheduling policy, a target platform and particular implementations of the scheduling mechanisms for the deployment of a mixed-criticality application cannot be performed independently. A *user-space* scheduling framework, such as the one presented, enables the rapid prototyping and comparison of various policies and implementation approaches to guide high-level design space exploration. As a next step, the development of efficient, timing-predictable mechanisms for the selected approaches at the *operating system* and *platform* levels needs to be addressed.

Acknowledgments The authors would like to thank Lars Schor and Pratyush Kumar for valuable discussions, Felix Wermelinger for the implementation of the single-core EDF-VD scheduler in the SF3P framework, and also the anonymous reviewers for the constructive feedback.

References

[1] G. Giannopoulou *et al.*, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *EMSOFT*. IEEE, 2013, pp. 1–15.

[2] S. Baruah *et al.*, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, vol. 50, no. 1, pp. 142–177, 2014.

[3] ARINC, "ARINC 653-1 avionics application software standard interface," http://www.arinc.com/, Tech. Rep., 2003.

[4] "European fp7 certainty project," http://certainty-project.eu/.

[5] A. Burns and R. Davis, "Mixed criticality systems: A review," 2015.

[6] "Intel®Xeon Phi™coprocessor architecture," http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[7] S. Baruah *et al.*, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *ECRTS*, 2012, pp. 145–154.

[8] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *RTSS*, 2007, pp. 239–243.

[9] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *RTAS*, 2010, pp. 13–22.

[10] S. Baruah and G. Fohler, "Certification-cognizant time-triggered scheduling of mixed-criticality systems," in *RTSS*, 2011, pp. 3–12.

[11] R. Pathan, "Schedulability analysis of mixed-criticality systems on multiprocessors," in *ECRTS*, 2012, pp. 309–320.

[12] J. Anderson, S. Baruah, and B. Brandenburg, "Multicore operating-system support for mixed criticality," in *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[13] M. Mollison *et al.*, "Mixed-criticality real-time scheduling for multicore systems," in *CIT*, 2010, pp. 1864–1871.

[14] D. Tamas-Selicean and P. Pop, "Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures," in *RTSS*, 2011, pp. 24–33.

[15] H. Li and S. Baruah, "Global mixed-criticality scheduling on multiprocessors," in *ECRTS*, 2012, pp. 166–175.

[16] K. Lakshmanan *et al.*, "Resource allocation in distributed mixed-criticality cyber-physical systems," in *ICDCS*, 2010, pp. 169–178.

[17] J. Herman *et al.*, "Rtos support for multicore mixed-criticality systems," in *RTAS*, 2012, pp. 197–208.

[18] H.-M. Huang, C. Gill, and C. Lu, "Implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks," *ACM Trans. Embedded Computing Systems*, vol. 13, no. 4s, pp. 126:1–126:25.

[19] J. Calandrino *et al.*, "Litmus^{rt} : A testbed for empirically comparing real-time multiprocessor schedulers," in *RTSS*, 2006, pp. 111–126.

[20] A. Easwaran, "Demand-based scheduling of mixed-criticality sporadic tasks on one processor," in *RTSS*, 2013, pp. 78–87.

[21] P. Huang *et al.*, "Service adaptions for mixed-criticality systems," in *ASP-DAC*, 2014, pp. 125–130.

[22] F. Wermelinger, "Implementation and evaluation of mixed-criticality scheduling approaches," Master's thesis, ETH Zurich, 2013.

[23] B. Barney, "Posix threads programming," https://computing.llnl.gov/tutorials/pthreads/.

[24] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux," in *Ottawa Linux Symposium*, 2002.

[25] F. Franchetti, "Fast barrier for x86 platforms," http://www.spiral.net/software/barrier.html.

[26] B. D. de Dinechin *et al.*, "Time-critical computing on a single-chip massively parallel processor," in *DATE*, 2014, pp. 97:1–97:6.

[27] A. Gomez *et al.*, "Sf3p: A framework to explore and prototype hierarchical compositions of real-time schedulers," in *RSP*, 2014.

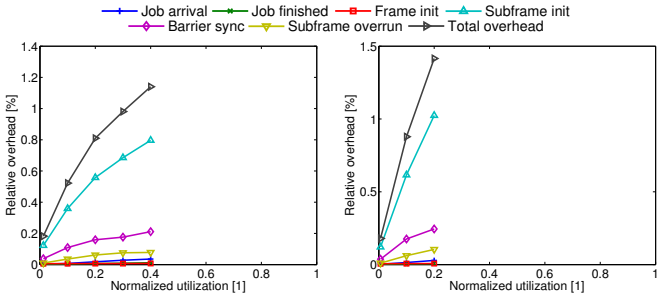
[28] G. Paoloni, "How to benchmark code execution times on Intel™IA-32 and IA-64 instruction set architectures," http://download.intel.com/embedded/software/IA/324264.pdf, 2010.

[29] M. Saito and G. A. Agha, "A modular approach to real-time synchronization," *SIGPLAN OOPS Mess.*, vol. 7, no. 1, pp. 13–20, 1996.

[30] A. Züpke, "Deterministic fast user space synchronization," *OSPERT*, 2013.

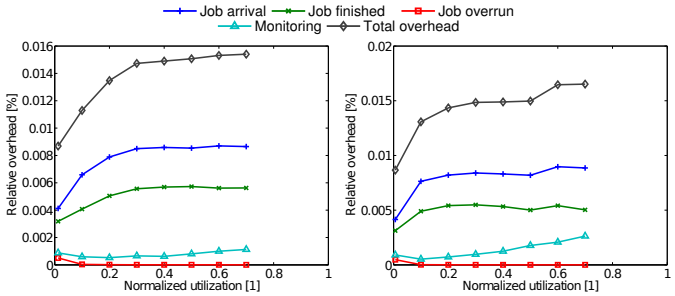
[31] R. Spliet *et al.*, "Fast on average, predictable in the worst case: Exploring real-time futexes in litmus rt," in *RTSS*, 2014.

[32] "RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification," 1992.



(a) 16 cores (b) 32 cores

Fig. 10: FTTS overheads for increasing utilization.



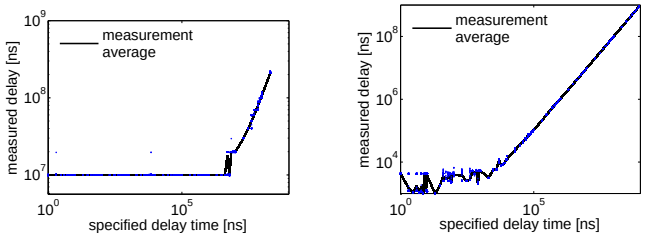
(a) 16 cores (b) 32 cores

Fig. 12: pEDF-VD overheads for increasing utilization.

APPENDIX

A. Timed Sleep Analysis

Accurate timed sleep is very important for the implementation of the overrun checkers and dispatchers, as motivated in Sec. VI-A. To this end, we have empirically analyzed the real sleep delay with respect to the time argument that is passed to the `clock_nanosleep` function. The results of our analysis for sleep delays in the range from 1ns to 1sec are depicted in Fig. 11. Each measurement in the 98% confidence interval is marked with a dot in the graph, and the line shows the average of these measurements. Note that the smallest sleep delay for the Xeon Phi platform lies around 10ms and for the Core i5 in the region of a few microseconds, respectively. Note also that the large discrepancy between the specified and the actual sleep duration can be a possible source of execution time overruns on these platforms, especially on the Xeon Phi, leading to repeated mode switches at runtime.



(a) Intel®Xeon Phi platform. (b) Intel®Core i5 platform.

Fig. 11: Measured sleep delays of `clock_nanosleep`.

B. Evaluation of Relative Scheduler Overheads on 16 and 32 Processing Cores of the Xeon Phi

For comparison to the overhead analysis in Sec. VII-A for variable utilization on 8 cores, we also show the measured overheads for 16 and 32 cores in Fig. 10 (FTTS) and 12 (pEDF-VD), respectively. We observe that similar trends as for 8 cores apply also for the 16- and 32-core systems.

C. FMS Task Execution Times

The task properties of the flight management system (FMS) that we used as real-world example in Sec. VII-C are listed in Table III. Beside the task periods and criticality level, we show their measured WCETs on our two target platforms. The WCETs were obtained by simulating a worst-case FTTS schedule, where all tasks of the same criticality level execute in parallel on separate cores. On each target platform, a 1 minute simulation was executed 10 times. For each task, the worst execution time observed is chosen as LO-criticality WCET. We set HI-criticality WCET estimates $C_i(\text{HI})$ to 10ms, which is significantly greater than the worst observed execution time. Based on these measurements, the FTTS schedules for the simulations in Sec. VII-C were optimized according to the methods presented in [1].

TABLE III: Flight Management System

Purpose	Task	CL	Period [ms]	LO-Level Exec. Time [ms]		HI-Level Exec. Time [ms]
				Xeon-Phi	Core i5	
Sensor data acquisition	τ_1	LO	100	0.230	0.014	0
	τ_2	LO	100	0.150	0.005	0
	τ_3	HI	200	1.450	0.059	10
Localization	τ_4	LO	100	0.140	0.004	0
	τ_5	LO	1000	0.150	0.006	0
	τ_6	LO	200	0.150	0.005	0
	τ_7	HI	200	1.580	0.050	10
	τ_8	HI	1000	0.380	0.021	10
	τ_9	HI	5000	0.140	0.016	10
	τ_{10}	HI	1000	0.060	0.005	10
Nearest Airport	τ_{11}	LO	1000	31.460	1.474	0