Overrun Handling for Mixed-Criticality Support in RTEMS

Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen
Department of Informatics, TU Dortmund University, Germany
Email: {kuan-hsun.chen, georg.von-der-brueggen, jian-jia.chen}@tu-dortmund.de

***Abstract*—Real-time operating systems are not only used in embedded real-time systems but also useful for the simulation and validation of those systems. During the evaluation of our paper about *Systems with Dynamic Real-Time Guarantees* that appears in RTSS 2016 we discovered certain unexpected system behavior in the open-source real-time operating system RTEMS. In the current implementation of RTEMS (version 4.11), overruns of an implicit-deadline task, i.e., deadline misses, result in unexpected system behavior as they may lead to a shift of the release pattern of the task. This also has the consequence that some task instances are not released as they should be. In this paper we explain the reason why such problems occur in RTEMS and our solutions.**

I. Introduction

Over the last years the number of embedded systems has drastically increased as embedded systems are now part of many daily-used products and are used in the control of many technical devices and machines. For most of those embedded systems real-time guarantees have to be given. This means, the correct behavior of a system depends not only on the computation value but also on the satisfaction of given timing constraints, i.e., the computation has to be completed successfully before a certain deadline. This is due to the fact, that embedded systems are used in safety-critical applications where missing these timing constraints would risk of resulting in serious consequences, e.g., in automotive or aeronautical applications. Therefore, the satisfactions of those timing constraints have to be ensured, either at runtime or during the design time of the system. Real-time operation systems (RTOS) and platforms like FreeRTOS [9], Litmus-RT [3], and RTEMS [1] are not only used for running embedded real-time systems but also are useful tools for validating timing constraints at design time, either via simulations or by actually running the implemented tasks.

For many real-time applications the sporadic task model has been adopted, where each task τ_i is characterized by its minimum inter-arrival time or period T_i, its relative deadline D_i, and its worst-case execution time (WCET) C_i. Such a sporadic task τ_i releases an infinite number of task instances, called jobs, where two consecutive releases of jobs of the same task have to be separated by the minimum inter-arrival time T_i. For example, this is a good way to model an application that has to react to a value recurrently read from a sensor.

In many cases either the implicit-deadline task model, i.e., $D_i = T_i \ \forall \tau_i$, or the constrained-deadline task model, i.e., $D_i \leq T_i \ \forall \tau_i$, is adopted. Suppose that the question at hand is to validate if all tasks meet their deadlines or that we only care about the correct execution of a system, where the timing behavior was ensured in a previous analysis. For such applications, the support of current implementations of real-time operation systems is sufficient. However, some applications also have tasks with arbitrary deadlines, i.e., for some tasks $D_i > T_i$. If the tasks are strictly periodic, this leads to a situation where two or more instances of the same task are ready to be executed at the same time. It is usually assumed that multiple task instances of a task are executed in a first-come-first-serve (FCFS) manner. Thus, it is sufficient to release the second task instance at the moment the first task instance finishes, assuming that the first task instance finishes after the time point at which the second task instance would have been released according to a strictly periodic pattern.

In addition, the situation that a task instance is not finished at the end of its period (we call this behavior an overrun) may not only happen if a task has an arbitrary deadline but also when a task misses its deadline. While a deadline miss is not tolerable in most cases, in some situations rare deadline misses might be tolerable for some tasks. In Mixed-Criticality Systems [11] tasks have two different WCETs where the WCET in the high-criticality mode is longer than the WCET in the low-criticality mode. To ensure that the deadlines of the high-criticality tasks are met, the state of the art abandons the low-criticality tasks in the high-criticality mode. Instead of abandoning these tasks, running them with best effort or reduced timing guarantees, e.g., bounded tardiness [12], seems a reasonable option. However, in these cases the low-criticality tasks may miss some of their deadlines which results in an overrun. When software based recovery strategies for transient faults are considered, e.g., re-execution [10] or checkpointing [6], tasks may have a longer execution time in some rare cases due to the recovery operations. If the fault rate is slightly higher than expected, this could lead to deadline misses as well.

A real-time operating system should make sure that the system still behaves as expected if such overrun situations occur. Namely, the task instances should still be released according to the given pattern and all task instances are still released even if another task instance is still in the system at the moment the next task instance would be released according to the pattern. This makes sure that the system behavior is predictable even with very rare overrun situations. In addition, it allows to analyze the effect those overruns have on the system behavior.

For our work on *Systems with Dynamic Real-Time Guarantees* [12] we ran simulations with RTEMS (*Real-Time Executive for Multiprocessor Systems*) [1] to analyze the impact of transient faults during the execution of a task instance. We assumed those faults to happen randomly under a given fault rate and wanted to analyze the impact of the fault rate on the system behavior. During these analyses we discovered

that RTEMS (version 4.11) did not behave as expected when these faults result in a task missing its deadline. On one hand, missing the deadline of a task instance leads to a shift of the release pattern of the task. On the other hand, if the deadline was missed by more than one period, the task instance that should have been released during this period was never released. We extended the current release of the RTEMS source code to tackle these two problems.

Our Contributions: This paper focuses on explaining the problems that arise from the current implementation of real-time operating systems when overruns take place and provides a solution that tackles these problems.

- An introduction to the general overrun handling mechanism implemented in well-known real time operating systems is given and compared with the model usually considered in academic publications. We explain how these overruns can occur in practical cases, e.g., arbitrary deadlines, Mixed-Criticality systems, software based solutions for handling transient faults like re-execution and checkpointing.
- We discovered two major problems in the current implementation of *Real-Time Executive for Multiprocessor Systems (RTEMS)* [1], namely that the release pattern of tasks is shifted, and that jobs are not released at all if the overrun of a task is longer than one period.
- A comprehensive extension to the latest RTEMS [1] source code (version 4.11) is provided that enhances the fixed-priority scheduler to provide a proper behavior for overrun handling, especially tackling the two determined major problems.

In addition we shortly explain the case study we performed for the paper *Systems with dynamic real-time guarantees in uncertain and faulty execution environment* [12], which was the reason why we discovered those problems and enhanced the implementation of RTEMS [1].

II. Overrun Handling in RTEMS

In this section, we will first describe the original design of overrun handling in *RTEMS (Real-Time Executive for Multiprocessor Systems)* [1] and present how the enhancement integrates perfectly in the latest RTEMS release. A motivational example is provided to demonstrate the differences between the original design and the enhanced version of overrun handling. In addition to the proper overrun handling routine, we also provide a useful helper function, which is detailed at the end of this section.

A. Task Model

In this paper we consider n independent periodic real-time tasks $\mathbf{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ in a uniprocessor system. Each task τ_i releases an infinite number of jobs (also called task instances) under the given period (temporal) constraint T_i. As we assume the tasks to be strictly periodic this means if at time θ_a a job of task τ_i arrives, the next instance of the task must arrive at $\theta_a + T_i$. The relative deadline of task τ_i is D_i, i.e., a task instance released at θ_a must be finished before $\theta_a + D_i$. We mainly consider implicit-deadline task sets in this paper, i.e., $D_i = T_i \ \forall \tau_i$.

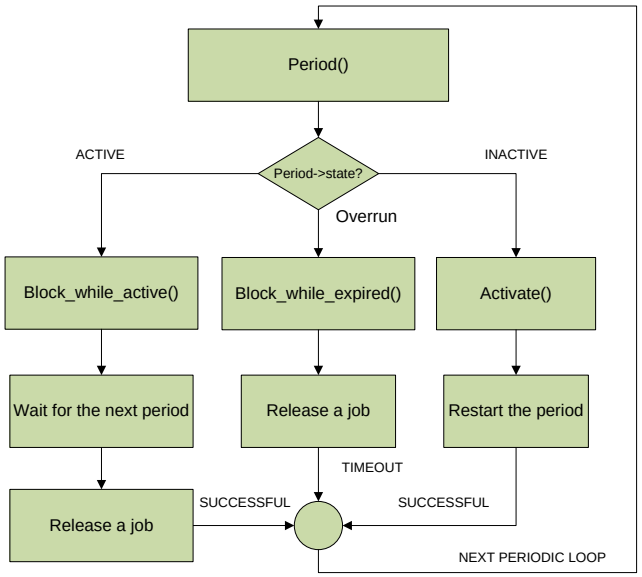


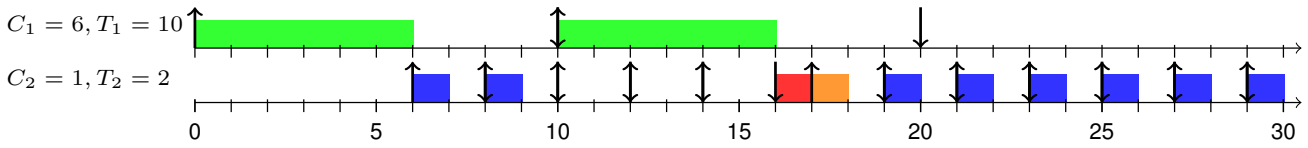
Fig. 1: Flowchart of the RMS manager in RTEMS

B. Original Design

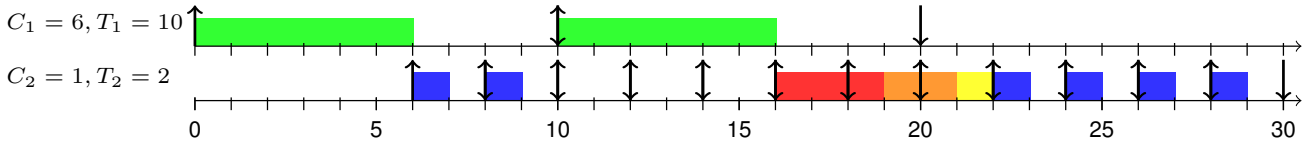
In the considered version 4.11 of RTEMS, the scheduler uses a virtual table of function pointers to hook scheduler-specific code and the thread management. In this work, we limit our attention on the fixed-priority scheduler in the RTEMS implementation. Although the related functions have a prefix *rate-monotonic* and is called RMS in RTEMS, the scheduler can be used for any fixed-priority scheduling, as the priorities can be set by the user. For the sake of clarity, we will remove the prefix when we mention the functions of the scheduler in the rest of paper. The primary source code is located in the following files in the SuperCore (cpukit/score):

- cpukit/rtems/src/ratemonperiod.c
- cpukit/rtems/src/ratemontimeout.c
- cpukit/rtems/include/rtems/rtems/ratemon.h
- cpukit/rtems/include/rtems/rtems/ratemonimpl.h

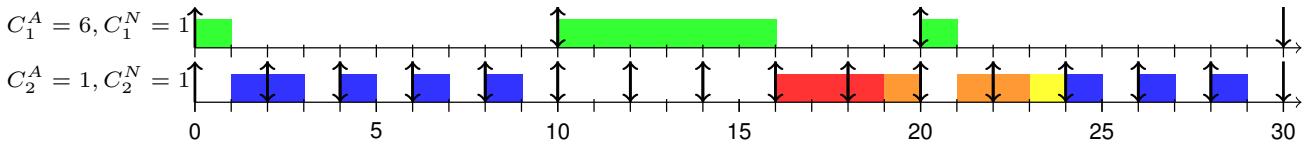
The rate-monotonic manager in RTEMS is responsible for handling the periodicity of the tasks. Based on it, this work mainly redefines the behavior of the *Period()* function and the *Timeout()* function in *ratemonperiod.c* and *ratemontimeout.c*, respectively. As mentioned in [2], for each task, its periodicity is implemented by using a timer to track its period. The application designer should create a periodic timer and implement the task body with a loop that calls *Period()* at the beginning to initialize the timer with the current system tick plus its period. Every time the task finishes its task body, i.e., at the end of the loop, *Period()* is called to setup the next iteration and immediately checks if the period is depleted yet. If the task finishes before its next period, it immediately goes to sleep (or suspends) until its period elapses, at which time its timer fires, wake the task to continue the loop body, and so on so forth. In the original design in RTEMS, if the watchdog notices that the deadline of a job expires but the job has not finished yet, i.e., a period timeout of a task takes place but the task body has not been finished, RTEMS marks the period state as EXPIRED and does nothing more. The next



(a) The original design of overrun handling in RTEMS. The red job is postponed from 10. The arrival pattern from 16 is changed due to the lateness of the red job, by which the orange job and the following jobs are all released earlier.



(b) The overrun handling with our enhancement in RTEMS. The postponed jobs due to the execution of τ_1 are marked red, the jobs postponed due to the execution of previous jobs of τ_2 that are not executed in the right period are marked orange. The yellow job is postponed due to the orange job in the same period but can still finish its execution on time.



(c) The overrun handling with our enhancement for dynamic real-time guarantees in RTEMS. Due to error recovery routine, task τ_1 runs in the second period with the abnormal mode, i.e., $C_2^A = 6$. The postponed jobs due to the execution of τ_1 are marked red, the jobs postponed due to the execution of previous jobs of τ_2 that are not executed in the right period are marked orange. The yellow job is postponed due to the orange job in the same period but can still finish its execution on time.

Fig. 2: An example illustrates different system behaviors according to the original design and the enhanced overrun handling.

time *Period()* is called at the beginning of the loop, i.e., when the expired task is finished, the *Block_while_expired()* function records the marked state of the expired period to update the system statistic routine and releases the next job immediately while updating the timer with the current system tick plus its period. Figure 1 illustrates the flowchart of the RMS manager. However, such an overrun handling mechanism is not able to keep the periodicity of the task, since the system tick at which the delayed task instance finishes is normally not at an integer multiple of the period of the task. Also this moment does not necessarily have to happen during or at the end of the first period after the deadline expired, but could be in any period after the originally expired one. In such a scenario, all the postponed jobs that would have been initialized in the time interval between the expired deadline and the newly released task instance is created are just gone.

To illustrate the behavior of the original design and the behavior after the enhancement, we provide an example with two implicit-deadline sporadic[1] tasks in Figure 2: τ_1 with $C_1 = 6$ and $T_1 = 10$, and τ_2 with $C_2 = 1$ and $T_2 = 2$, where τ_1 is given a higher priority than τ_2. We let τ_1 only release two jobs (at 0 and 10) and τ_2 has a phase of 6, i.e., the first job of τ_2 is released at 6 but its follow-up jobs arrive with a period of 2.

The behavior using the current version of RTEMS is shown in Figure 2a. We see that τ_2 can not be executed in the interval $[10; 16]$ as τ_1 has higher priority, resulting in 3 expired periods. Using the original overrun handling the job of τ_2 released at 10 (colored red) is finished at 17 which leads to a new release of τ_2 at 17, as τ_2 is marked to have deadline misses. This results in a shift of the periodicity of τ_2 by 1 for this job and all the following jobs. The job that should be released at 16 is released at 17 now (orange), and the following jobs are all shifted as well. it is worth noting that the jobs that should be released at 12 and 14 are never released to the system. Overall, we can observe that the original design in RTEMS does not match the expectation for overrun handling and periodicity in most applications and researches, as two jobs do not enter the system at all and the period of the τ_2 is shifted due to the deadline miss of the job started at 10.

[1]For the simplicity of presentation, the sporadic task model here is more clear to show the effects rather than using the strict periodic release pattern.

C. Enhancement

After discussing the original design, we now explain how we enhanced the original implementation to handle the deadline overrun correctly. Based on the previous observations, the main ideas of our enhancement can be summarized as follows:

- correcting the deadline assignment errors, i.e., keep the periodicity of the task,
- tracking the number of postponed jobs, and
- providing two modes of job releasing depending on the situation, i.e., normal and postponed release.

The flowchart is provided in Figure 3. In the rest of this subsection, we explain more details about our implementation.

Correcting the deadline assignment: To keep the correct deadline assignment after a job misses its deadline, we add an additional variable called *latest_deadline* in the *Control* structure that records the latest deadline assigned by the period watchdog. This variable is used to call an additional function named *Renew_deadline()* in the *Timeout()* function, making sure that the watchdog updates the timer to the next absolute deadline. The next deadline will be recorded in the variable *latest_deadline*. Due to this enhancement, the

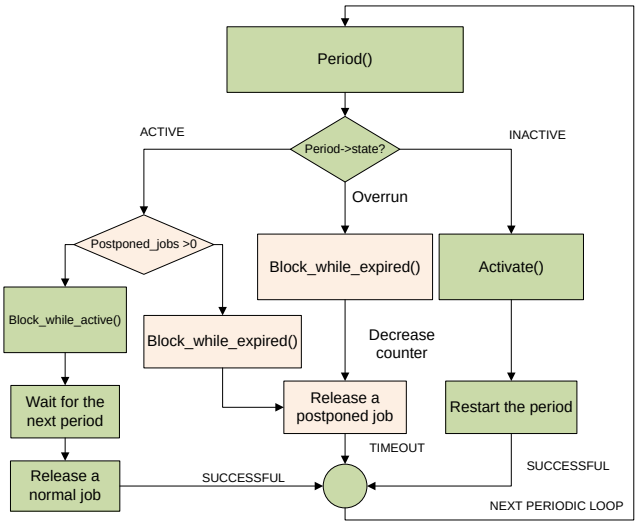


Fig. 3: Flowchart of the enhanced RMS manager. The light background blocks are involved in the enhancement.

watchdog updates the timer correctly instead of doing nothing in the original design. The recorded deadline in the variable *latest_deadline* is prepared for the next call of the *Timeout()* function, allowing to set the deadline to the sum of latest deadline and period for the next deadline assignment directly.

Tracking the number of postponed jobs: In addition to the correct deadline assignment, we also implemented a tracking mechanism for the number of the postponed jobs to ensure that the postponed jobs are correctly released. To deploy this idea, we add an additional variable in the *Control* structure called *postponed_jobs* and initialize it while the periodic timer is created. When the watchdog detects a deadline miss (a period timeout while the related task is not finished), this variable is increased by one immediately in the *Timeout()* function. Conversely, every time a postponed job is released by *Release_postponedjob()*, this variable is decreased by one immediately.

Two modes of job releasing: As we mentioned before, normally the execution of periodic tasks is correct, if there is no deadline miss. When there is an overrun, the watchdog of the expired period detects such an overrun and the next call of *Period()* will immediately release one job and set the expired period state back to the normal state by using the function *Block_while_expired()*. In the original facility, the above behavior is handled by the function *Release_job()*. In the enhancement, we replaced it with an additional function *Release_postponedjob()* and let the watchdog handle the deadline assignment individually. We implemented it similar to the original job releasing routine, but it does not assign the new deadline to the timer while releasing a postponed job. Since they already missed their deadline and are tracked by the watchdog in the *Timeout()* function, it is meaningless to assign an already expired deadline to the watchdog. Every time a postponed job is released, the variable *postponed_jobs* is decreased by one. When the variable *postponed_jobs* is not 0, the scheduler is in the **postponed mode**.

Based on the original design in RTEMS, every time *Period()* is called at the beginning of the loop, it checks the state of the current period. When the state is EXPIRED, in fact the period might be expired many times already. In this block, now it calls the enhanced *Block_while_expired()* to release the postponed jobs without assigning a new deadline. We added one more condition when checking if the state of the period is ACTIVE, in which the task is blocked and waits for the next period. The additional condition checks if the variable *postponed_jobs* is greater than 0, which means the scheduler is still in the **postponed mode**. Otherwise, when the variable *postponed_jobs* is 0, the scheduler is in the **normal mode** and the job release is the same as it was in the original design.

We use the same example as before to illustrate the effect of our enhancement in Figure 2b, which is now matching the expectation of most applications and researches. At time 10 τ_1 is executed, as it has higher priority than τ_2, which leads to three expired periods of τ_2. This means, the watchdog increases the variable *postponed_jobs* to 3. When task τ_2 can be executed, it starts to release the postponed jobs in the postponed mode. Moreover, the three postponed jobs of τ_2 that should be released at time units 10, 12, and 14 due to the execution of τ_1, marked red, are released at time units 16, 17, and 18, respectively. However, the period from 16 to 18 and from 18 to 20 also expired while executing the three previously postponed jobs and the job postponed at 16, marked orange. At 20 the job postponed at 18 (marked orange as well) is released and is finished at 21, at which time the job postponed at 20 is released (yellow) and finishes at 22. After all the postponed jobs are finished, the release of τ_2 turns back according to the original pattern again.

We provide an additional example in Figure 2c as well to demonstrate how the enhancement works for dynamic real-time guarantees [12]. Detailed notation can be found in Section III or in [12]. Suppose that task τ_1 requires a full timing guarantee with the abnormal execution time $C_1^A = 6$, and the normal execution time $C_1^N = 1$. Task τ_2 is a timing tolerable task with $C_2^A = C_2^N = 1$. In Figure 2c, the second job of task τ_1 needs 6 time units for its execution time. We can see that after all the postponed jobs of task τ_2 are finished at 24, the release of task τ_2 turns back according to the original periodic pattern again.

D. Helper Function

In our enhancement, we also provided a helper function called *Postponed_num()* to return the number of postponed jobs with the current period ID of tasks as function input. According to the expected behavior, the number of postponed jobs is only increased by the watchdog of the corresponding period; it is only decreased by the routine of postponed job releasing in RMS manager. This helper function is especially useful for on-line admission control and the system monitor design in terms of scheduling. In fact, it is already used in [12] for the system state analysis, where the overhead of the enhancement is negligible in our evaluation (see Section III).

REMARK: ARBITRARY DEADLINE

Up to this point, we have presented how to handle the overrun for implicit- ($D_i = T_i \ \forall i$) or constrained-deadline ($D_i \leq T_i \ \forall i$) task sets properly with our enhancement based on the original scheduler design in RTEMS. In the arbitrary-deadline task model, no general relation between D_i and T_i exists. Especially, for some tasks $D_i > T_i$ is possible. However, due to the limitation of the original design in the fixed-priority scheduler, the arbitrary-deadline task model is not supported yet in RTEMS as well. Since the deadline detection is originally embedded in the routine of the task periodicity, the deadline is expected to be less than or equal to their period without any overrun. However, this expectation is only true for implicit-deadline and constraint-deadline task models and applications where all deadlines are met.

From the schedulability analysis aspect, the detection of deadline misses should be separated as an individual feature. One potential solution is to set the deadline of a task explicitly as the input parameter while the period is initialized and update the deadline accordingly while every job is released. The detection routine for deadline misses should be revised for recording the number of deadline misses rather than the number of periods expired. By co-working with our enhancement, this solution could make the fixed-priority scheduler of RTEMS support more general real-time task models.

III. CASE STUDY: SYSTEMS WITH DYNAMIC REAL-TIME GUARANTEES

The need for the presented enhanced implementation for RTEMS was discovered during the work on the paper *Systems with dynamic real-time guarantees in uncertain and faulty execution environment* [12]. A *System with Dynamic Real-Time Guarantees* can be used to analyze and schedule systems where some tasks have two different worst-case execution times (WCET); a shorter WCET C_i^N for executions that happens more often (called normal execution), and a longer WCET C_i^A for some rare special cases (called abnormal execution). The general idea is that also all tasks in the system normally need to fulfill strict timing guarantees but in some special cases, i.e., a number of tasks with abnormal execution happen in a short period of time, for some not so important tasks (called *timing tolerable* tasks) rare deadline misses are tolerable while for the more important tasks (called *timing strict* tasks) deadline misses are allowed under no circumstances. In *Systems with Dynamic Real-Time Guarantees* fixed-priority scheduling is used.

A *System with Dynamic Real-Time Guarantees* assumes that at the beginning of a task's execution it is not possible to determine if the task is executed in a normal or an abnormal mode, i.e., abnormal executions happen randomly and do not follow a strict pattern. This is the case when we look at the fault tolerance enhanced to handle soft errors, i.e., the consequences of transient faults of the computing hardware or the memory subsystem, by using software-based solutions, e.g., re-execution [10] or checkpointing [6]. Such faults can either happen for each individual task instance with low probability or they can happen as a burst that affects (nearly) all tasks over a small time window. In both cases it would not be sensible to in general assume the longer C_i^N in the analysis if those faults occur rarely and some deadline misses can be tolerated, as it would lead to over dimensioning the system. The idea of *Systems with Dynamic Real-Time Guarantees* is to give *full timing guarantees*, i.e., all tasks meet all their deadlines, if tasks are executed normally, and maybe downgrade this guarantees to *limited timing guarantees* if some tasks are executed in the abnormal mode. When *limited timing guarantees* are given, only the *timing strict* tasks are guaranteed to meet their deadlines while the *timing tolerable* tasks may miss some deadlines but still bounded tardiness for these tasks is guaranteed. In addition, *Systems with Dynamic Real-Time Guarantees* provide a system monitor that analyzes if *full timing guarantees* can be given for all tasks, i.e., all tasks will meet their deadline if no faults occur, or if only *limited timing guarantees* can be given for some *timing tolerable* tasks. To determine this for each *timing tolerable* tasks an over estimation of the busy period is calculated, summing up the current carry-in workload by jobs with higher or identical priority and the workload created in the future under the assumptions of 1) a worst-case release pattern and 2) that no further faults occur. For details see Section 6 of [12].

Similar behavior occurs in Mixed-Criticality Systems [11] witch have two modes, a high- and a low-criticality mode where tasks have a longer execution time in the high-criticality mode. It is often assumed that low-criticality tasks can be abandoned when the system switches from the low-criticality to high-criticality mode to ensure that the high-criticality tasks will still meet their deadlines. However, this assumption has been criticised recently [4, 7, 8]. The problem is tackled when a *System with Dynamic Real-Time Guarantees* is used, as the low-criticality tasks are seen as *timing tolerable* tasks and the system gives *limited timing guarantees*, i.e., guarantees bounded tardiness instead of abandoning the task.

To analyze the behavior of *Systems with Dynamic Real-Time Guarantees*, a QEMU emulator under *Real-Time Executive for Multiprocessor Systems (RTEMS)* [1] was used where the number of cores was set to 1 for the simulation. In the situation it was assumed that transient faults happen randomly, i.e., a given rate of faults per millisecond, and at the moment a task instance finished its normal execution a random draw, based on the probability of faults per millisecond and C_i^N, determined if the execution was prolonged to run up to C_i^A or not. The system monitor was used to determine the amount of time when only *limited timing guarantees* could be provided.

As the *timing tolerable* tasks are not abandoned those tasks may miss their deadlines. However, they should be executed after the deadline as the result may still be useful. In addition, it may happen that not only one job of a task misses the deadline but also that the execution of the task may be postponed for more than one period. In these cases more than one job of a task may be ready to execute at a given time, another situation previously not covered in RTEMS [1]. The enhancement presented in Section II was necessary to ensure that the release pattern was still correct when a task missed its deadline. The number of the postponed releases was determined using the helper function. The system monitor framework also adopts the helper function when it calculates

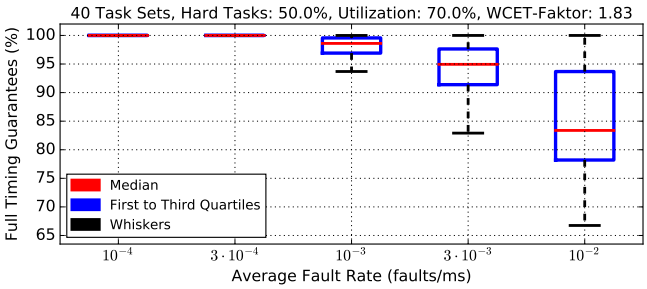


Fig. 4: Percentage of Time where *Full Timing Guarantees* can be given for task sets with utilization 70% in the normal mode under different fault rates.

the carry-in workload for each task.

The scheduling algorithm in [12] can only schedule 44.4% of the task sets when the task sets are randomly generated with 10 tasks, 50% of these tasks are randomly chosen to be *timing strict* tasks, the total utilization in the normal mode is 70%, and $C_i^A = 1.83 \cdot C_i^N \forall i$, i.e., the total utilization in the abnormal mode is $\approx 128.1\%$.

We used 40 of those randomly created tasks sets under the given setting that are schedulable according to the scheduling algorithm in [12] if the bounded tardiness condition for the *timing tolerable* tasks is dropped. Obviously, for utilization $> 100\%$ in the abnormal mode, bounded tardiness for the *timing tolerable* tasks can only be guaranteed if the fault rate is not too high as a high fault rate will lead to more overruns.

We let the system simulate a run for one hour under different fault rates for each of those task sets, i.e., on average 10^{-4}, $3 \cdot 10^{-4}$, 10^{-3}, $3 \cdot 10^{-3}$ and 10^{-2} faults per millisecond (f/ms). For each executed task instance we decided if the instance was faulty or not by a random draw. The results, i.e., the percentage of time the system was running with *full timing guarantees*, are shown in Figure 4 (which is Fig 8 in [12]). The median of those 40 sets is colored red. The blue box represents the interval from the first to the third quartile, while the black whiskers show the minimum and maximum of all of the data.

IV. Related Implementations

FreeRTOS [9] is a well-known real-time operating system, which especially offers lighter and easier real time processing. By analyzing *timers.c* we can see that *prvProcessTimerOrBlockTask()* and *prvProcessExpiredTimer()* are responsible for the periodicity of the task. In fact the feature of *prvProcessTimerOrBlockTask()* is similar to the function *Period()* in RTEMS that determines if the task should be blocked or if the timer has expired and the overrun handling is required. In the function *prvProcessExpiredTimer()*, the expired timer is updated immediately with the next expiry time. To maintain the periodicity, all the tasks' timers are listed in the expiry time order, and the task which refers to the head of list expires first. Although the deadline of a task is assigned correctly in their timer, we are not aware of any tracking mechanism in the FreeRTOS scheduler that enforces the correct number of postponed jobs to be released.

Litmus-RT [3] is a popular real-time extension of the Linux kernel. The overrun handling for the fixed-priority scheduler can be found in *job_completion()* in *sched_pfp.c*. By tracking back to the common function *prepare_for_next_Period()* in *jobs.c* and *setup_release()*, we noticed that it also has implemented a counter called *job_no* to record how many jobs should be ideally released without overrun. With *sys_wait_for_job_release()* in *litmus.c*, the task is only going to sleep when its number of released job is greater than *job_no* by triggering *complete_job()*. This is similar to the case in *Period()* where RTEMS decides if the period should be blocked. By this implementation, the postponed jobs should be released consecutively until there is no postponed job, which is similar to our enhancement.

V. Conclusion and Future Work

The demand of overrun handling has emerged and it is widely used in practical and theoretical systems, e.g., in the design of soft real-time systems, Mixed-Criticality systems [11], and *Systems with Dynamic Real-Time Guarantees* [12][2]. In this work, we have enhanced the fixed-priority scheduler in the released version 4.11 of RTEMS with a generally expected overrun handling mechanism. The provided enhancement now is in the pending patch on RTEMS report ticket #2795 [5] including the motivational example mentioned in Section II. To avoid breaking the existing behavior for applications relying on the original RTEMS feature, the system-level integration for the new variant via some explicit function calls is planned for the future. In addition we consider supporting the arbitrary-deadline task model and a separated deadline assignment as future work.

References

[1] Rtems: Real-time executive for multiprocessor systems. http://www.rtems.com/, 2013.

[2] G. Bloom and J. Sherrill. Scheduling and thread management with rtems. *SIGBED Rev.*, 11(1):20–25, Feb. 2014.

[3] B. Brandenburg. LITMUS^{RT}: Linux testbed for multiprocessor scheduling in real-time systems. http://www.litmus-rt.org/, 2006.

[4] A. Burns and R. Davis. Mixed criticality systems-a review. Technical report, University of York, 2016. 7th edition.

[5] K.-H. Chen. #2795 ticket: Overrun handling for general real-time models. http://devel.rtems.org/ticket/2795, 2016.

[6] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[7] R. Ernst and M. D. Natale. Mixed criticality systems - A history of misconceptions? *IEEE Design & Test*, 33(5):65–74, 2016.

[8] A. Esper, G. Nelissen, V. Nélis, and E. Tovar. How realistic is the mixed-criticality real-time system model? In *RTNS*, pages 139–148, 2015.

[9] R. T. E. Ltd. Freertos. http://www.freertos.org/, 2016.

[10] F. Many and D. Doose. Scheduling analysis under fault bursts. In *RTAS*, pages 113–122, 2011.

[11] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*, pages 239–243, 2007.

[12] G. von der Brüggen, K.-H. Chen, W.-H. Huang, and J.-J. Chen. Systems with dynamic real-time guarantees in uncertain and faulty execution environment. In *Real-Time Systems Symposium, 2016. Proceedings., 37th*, 2016.

[2] *Systems with Dynamic Real-Time Guarantees* paper [12] can be found in the following url: http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/mixed_critical.pdf