Implementation and Evaluation of Multi-Mode Real-Time Tasks under Different Scheduling Algorithms

Anas Toma, Vincent Meyers and Jian-Jia Chen
Department of Computer Science
TU Dortmund University
Dortmund, Germany
firstname.lastname@tu-dortmund.de

***Abstract*—Tasks in the multi-mode real-time model have different execution modes according to an external input. Every mode represents a level of functionality where the tasks have different parameters. Such a model exists in automobiles where some of the tasks that control the engine should always adapt to its rotation speed. Many studies have evaluated the feasibility of such a model under different scheduling algorithms, however, only through simulation. This paper provides an empirical evaluation for the schedulability of the multi-mode real-time tasks under fixed- and dynamic-priority scheduling algorithms. Furthermore, an evaluation for the overhead of the scheduling algorithms is provided. The implementation and the evaluation were carried out in a real environment using Raspberry Pi hardware and FreeRTOS real-time operating system. A simulation for a crankshaft was performed to generate realistic tasks in addition to the synthetic ones. Unlike expected, the results show that the Rate-Monotonic algorithm outperforms the Earliest Deadline First algorithm in scheduling tasks with relatively shorter periods.**

I. Introduction

In modern automotive systems, *Electronic Control Units* (ECUs) are used to control and improve the functionalities, the performance and the safety of various components. These embedded systems are in continuous interaction with various parts of the automobile such as the doors, the wipers, the lights and most importantly the engine [14]. In order to guarantee a correct behavior, the embedded system should react within a specific amount of time, i.e. the deadline. The timing correctness in these systems is very important, because a delayed reaction can result in a faulty behavior and then affect the reliability and safety of the automobile.

The software of an automotive application can be modeled as a set of recurrent tasks with timing constraints, i.e. periodic real-time tasks. For instance, to control the engine of an automobile, an *angular* task may release jobs depending on the engines speed. Such a task is linked to the rotation of specific devices such as crankshaft, gears or wheels. It could be responsible for calculating the time at which the spark signal should be fired, adjusting the fuel flow, or minimizing fuel consumption and emissions [9]. The period of this task, i.e. the time between the release of two consecutive jobs, is inversely proportional to the speed of the crankshaft. With an increasing rotation speed, the time available for the task to execute all of its functions may not be long enough, which results in deadline misses. This could lead to catastrophic consequences in hard real-time systems [6].

TABLE I: An example of a multi-mode task with three different execution modes.

Rotation Speed (rpm)	Mode Type	Executed Functions
$[0, 3000]$	A	$f1$, $f2$ and $f3$
$(3000, 6000]$	B	$f1$ and $f2$
$(6000, 9000]$	C	$f1$

In order to meet the timing constraints and prevent a potential system failure, the job has to react before the next job is released. Therefore, the task might have to drop some of its functions, the non-critical ones, to meet its deadline. This can be achieved by using tasks with different execution modes, i.e. *multi-mode tasks*, to adapt to the changing environment [15]. In some cases, tasks may react differently according to an external input and thus switch into different modes accordingly. In our example of the automobile's engine, the input is the engine speed and the functionalities of the tasks are part of the fuel injection system. Every time the crankshaft finishes a rotation, the tasks have to execute their respective functions. If the engine speeds up, the tasks may need to use another algorithm or functions to achieve their goal and avoid deadline misses. In other cases, the engine may be more stable at higher rotation speeds, but requires additional functions to be executed at lower speeds to keep it stable. Consequently, these functions are not required to be executed at higher speeds, which can be exploited to reduce the execution time of the tasks [7]. Table I shows an example of a multi-mode task with 3 types of execution modes: A, B and C. The selection of the mode depends on the rotation speed, where the task executes different functions in each mode. The rotation speed of the engine is measured in *revolutions per minute* (rpm).

Such a task model was presented by Buttazzo et al. [7]. They also provide schedulability analysis under Earliest Deadline First (EDF) algorithm. Furthermore, another analysis under Rate Monotonic (RM) algorithm is provided in [9], in addition to simulation for the effectiveness of the proposed test. However, non of the studies above performed the evaluation of the system in a real environment. In this paper, we provide an empirical evaluation of multi-mode tasks under EDF and RM algorithms. The evaluation was performed on a real hardware

Published in the 14th annual workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2018), Barcelona, Spain, July 2018.

running a real-time operating system. The contribution of this paper can be summarized as follows:

- Modifying the FreeRTOS real-time operating system to consider the periodic and multi-mode real-time tasks. Furthermore, several cost functions were implemented for a comprehensive evaluation[1].
- Implementing the EDF and RM scheduling algorithms in FreeRTOS which can be used in further studies and researches[1].
- Empirical evaluation for the schedulability of the multi-mode tasks under EDF and RM algorithms in a real environment, i.e. FreeRTOS running on Raspberry Pi. Moreover, overhead evaluation of both algorithms is provided in this work.

II. Background and Literature Review

A. FreeRTOS

In this Subsection, we introduce the FreeRTOS and its main components that were modified in our implementation [4]. FreeRTOS is a real-time operating system kernel that supports about 35 microcontroller architectures. It is a widely used and relatively small application consisting of up to 6 C files [3]. FreeRTOS can be customized by modifying the configuration file *FreeRTOSConfig.h*, e.g. turning preemption on or off, setting the frequency of the system tick, etc. Tasks in FreeRTOS execute within their own context with no dependency on other tasks or the scheduler. Upon creation, each task is assigned a *Task Control Block* (TCB) which contains the stack pointer, two list items, the priority, and other task attributes. Tasks can have priorities from 0 (the lowest) to *configMAX_PRIORITIES*-1 (the highest), where *configMAX_PRIORITIES* is defined in *FreeRTOSConfig.h*. A task in FreeRTOS can be in one of the following four states:

- Running: The task is currently executing.
- Ready: The task is ready for execution but preempted by an equal or a higher priority task.
- Blocked: The task is waiting for an event. The task will be unblocked after the event happens or a predefined timeout.
- Suspended: The task is blocked but does not unblock after a timeout. Instead the task enters or exits the suspended state only using specific commands.

The following are the main functions and data structures in FreeRTOS which will be mentioned in the following sections:

- *xTaskCreate()*: Creates a task and add it to the ready list.
- *prvInitialiseTCBVariables()*: Initialize the fields of the TCB.
- *vTaskDelayUntil()*: Delays a task for a specific amount of time starting from a specified reference of time.
- *vTaskStartScheduler()*: Starts the FreeRTOS scheduler.
- *pxReadyTasksLists*: An array of doubly linked lists with size of *configMAX_PRIORITIES* that contains the ready tasks according to their priorities. Each array element and a corresponding list represents a level of priority.
- *uxTopReadyPriority*: A pointer to the task with the highest priority in the ready list.

[1]The implementation is available on https://github.com/Anas-Toma/multi-mode

The scheduler in FreeRTOS is responsible for deciding which task executes at a specific time. It is triggered by every system tick interrupt and schedules the task with the highest static priority in the ready list for execution. It loops the ready list from the pointer *uxTopReadyPriority* to the lowest priority that has a non-empty list. If two tasks have the same priority, the share the CPU and switch the execution for every system tick.

B. Scheduling the Multi-Mode Tasks

Buttazzo et al. [7] provide analysis for the feasibility of multi-mode tasks under the EDF algorithm. Furthermore, a method is provided to determine the switching speed that keep the utilization of the tasks below a predefined threshold. On the contrary, Huang and Chen [9] present a feasibility test for such a task model under RM algorithm. Furthermore, they show the advantages of using the fixed-priority scheduling over the dynamic-priority scheduling. Both of the studies above evaluated their approaches by simulation.

III. Real-Time Multi-Mode Task Model

Multi-mode tasks are periodic tasks that can be executed in several modes [9]. Given a set $\mathcal{T}$ of n independent real time tasks. Each task i (for $i = 1, 2, \ldots, n$) has m_i execution modes, i.e., $\tau_i = \{\tau_i^1, \tau_i^2, \ldots, \tau_i^{m_i}\}$. In each mode τ_i^j, the task has different *worst-case execution time* (WCET) C_i^j, *period* T_i^j and *relative deadline* D_i^j. The task consists of an infinite sequence of identical instances, called jobs. T_i^j represents the time interval between the release of two consecutive jobs of the same task. Once a job is released, it should be executed within the deadline D_i^j. The mode of the task may change based on an external interrupt or any other event, which can be used to change the execution time of the tasks and then the total utilization accordingly. If the mode is changed during the runtime, it will take effect in the next period.

IV. Design and implementation

This section covers the implementation of the multi-mode task model and both scheduling algorithms in FreeRTOS (A ported version to Raspberry Pi [1]).

A. Multi-Mode Task Model

1) Periodic Real-Time Tasks: It is necessary to have a periodic task model in order to implement the multi-mode tasks. Therefore, the tasks in FreeRTOS were modified by expanding the *task control block* (TCB) structure with the typical fields used in periodic real-time systems [6]. In addition to the original TCB attributes in FreeRTOS, the following ones with *portTickType* data type were added:

- *uxPeriod*: Period.
- *uxWCET*: Worst-case execution time.
- *uxDeadline*: Relative deadline.
- *uxPreviousWakeTime*: The previous wake time of the task.

The absolute deadline of a task can then be calculated as $D = uxDeadline + uxPreviousWakeTime$. Those attributes were also added to the parameters of the *xTaskGenericCreate()*, *xTaskCreate()* and *prvInitialiseTCBVariables()* functions to be initialized upon task creation. To guarantee the

periodicity of the tasks, i.e. constant execution frequency, the *vTaskDelayUntil()* function is used to delay the task for the specified period of time T_i^j starting from the arrival time captured by the *xTaskGetTickCount()* function and stored in *uxPreviousWakeTime* variable.

2) Modes: Now, we have a periodic task model and it will be modified to have different execution modes. To achieve that, the TCB attributes described in Subsection IV-A1 should have many values corresponding to the modes of the task. Since the number of the modes are fixed and known upon system setup, an array data structure is used to store the several values of the same attribute. The TCB fields were modified as follows:

- `portTickType *uxPeriods;`
- `portTickType *uxWCETs;`
- `portTickType *uxDeadlines;`

Additional attributes were added to store the number of the modes and determine threshold values for each mode level as follows:

- `unsigned int uxNumOfModes`: The number of the modes.
- `unsigned int *uxModeBreaks`: The range for each mode.

uxModeBreaks contains the maximum value of each mode level. For example, the first mode (indexed by 0) will be chosen if the external input is between 0 and *uxModeBreaks*[0]. Similarly, the range of the second mode is (*uxModeBreaks*[0],*uxModeBreaks*[1]]. The parameters were also added to the corresponding functions as described in Subsection IV-A1.

To switch to the corresponding mode during the runtime, the function *vUpdateMode()* was implemented. It chooses the appropriate mode based on an external input and the defined mode ranges in the array *uxModeBreaks*. The value of the external input is stored in a global variable named *externalInput* with type *volatile unsigned int*. It is declared as volatile, because its value might change at any moment during the runtime. So, any application can change the mode easily by updating this variable according to an external input or any other event. The *externalInput* is initialized to 0, which means that the first mode is the default one. According to the definition of the multi-mode tasks in Section III, tasks do not change their mode once a mode change request is arrived, even if they are blocked. Any changes will be applied starting from the next release. Therefore, the mode is updated in our implementation right before the next wake-up time. This was done by calling *vUpdateMode()* at the start of the function prvAddTaskToReadyQueue().

B. Rate-Monotonic Scheduler

According to the RM algorithm, the priorities of the tasks are assigned statically before the execution according to their periods, i.e., the tasks with a shorter period has a higher priority [12]. We reserve the priority level 1 in FreeRTOS and the corresponding ready list *pxReadyTasksLists*[1] for the tasks to be scheduled under RM algorithm. All of these tasks are assigned to priority 1 upon creation temporarily. Then, their priorities are assigned according to RM algorithm before the scheduler is started. A new function named *vAssignPriorities()* was implemented and is called in *vTaskStartScheduler*() function after the creation of the idle task to assign those priorities. Another attribute, *unsigned int *uxPriorities*, was added to the TCB to store the priorities of the same task for all the corresponding modes. Moreover, the following doubly linked list was created to sort the tasks according to their periods in all of their modes:

```
1 struct doublyLinkedListNode {
2     unsigned int value;
3     void *task;
4     int mode;
5     volatile struct doublyLinkedListNode *↩
          prev;
6     volatile struct doublyLinkedListNode *↩
          next;
7 };
```

Where *value* and *task* store the period of each mode and a pointer to the corresponding task's TCB respectively. The tasks in *pxReadyTasksLists[1]* are inserted into the doubly linked list and sorted according to their periods. Then, the priorities are assigned for each task for all the modes by filling the *uxPriorities* array. Finally, the tasks are moved to their corresponding ready lists according to the new assigned priorities.

C. Earliest Deadline First Algorithm

The EDF algorithm assigns the highest priority to the job with the earliest absolute deadline among of the ready jobs [13]. Before implementing the EDF algorithm, task creation functions were modified, so the tasks can be scheduled dynamically. The static priority parameter in the *xTaskCreate()* function is discarded by setting it always to 1. The FreeRTOS uses an array of linked lists to store the ready tasks according to their priorities. The array size can be defined by the variable *configMAX_PRIORITIES*. However, it is not suitable to use an array with a fixed size for dynamic priority assignment. Of course this array can still be used, but either it should be big enough for any eventual number of tasks, or its size should be always reallocated. To avoid such an overhead, we replaced the the ready list *pxReadyTasksLists* with a doubly linked list that has the same name to maintain all the ready tasks. We apply a binary heap on the ready tasks to find the one with the highest priority. Every time a task is added to the ready list by calling the *prvAddTaskToReadyQueue()* function, the absolute deadline is calculated, as shown in Subsection IV-A1, and the task with the earliest absolute deadline is scheduled for execution.

D. Additional Modifications

1) Shared Processor Behavior: In this subsection, we present the additional modifications to the system in order to improve the overall performance and make our EDF implementation work appropriately. In the FreeRTOS, the tasks share the processor equally if they have the same priority. The processor executes the tasks in a round-robin behavior, which results in a context switching for every system tick and then additional overhead. The actual cost of switching between two tasks is approximately $4\mu s$ per every context switch according to our measurements. Even if the ready list has just one task or only one task has the highest priority, the FreeRTOS performs

context switching on the same task for every system tick. This includes saving the state of the task and restoring it every system tick which results in a high overhead. We solved such a problem by performing context switching only if we have a new task with a higher priority or if the current task under execution is moved to the blocked state. Context switching is then only conducted when necessary. Tasks with the same priority are scheduled according to their insertion order in the ready list.

2) Performance and Evaluation metrics: For system evaluation, we implemented the following cost functions to measure the performance of the implemented schedulers [6]:

- System overhead: The time required to handle all mechanisms other than executing jobs such as scheduling decisions, context switching and system tick interrupts.
- Success ratio: The percentage of the schedulable task sets among the total number of the task sets.
- Average response time:

$$t_r^j = \frac{1}{n}\sum_{i=1}^{n}(f_i - a_i)$$

  where a_i and f_i are the arrival time and the finishing time of task execution respectively.
- Maximum lateness:

$$L_{max} = \max_i(f_i - d_i)$$

3) Configurations and Definitions: Several configuration parameters were added to the system which are required for the evaluation or visualization of the scheduling. The following definitions were added to the file *FreeRTOSConfig.h*:

- *configANALYSE_METRICS*: Trace the data of the tasks for the evaluation metrics (1: Enabled, 0: Disabled).
- *configANALYSE_OVERHEAD*: Measure the total time consumed by the tick interrupts (1: Enabled, 0: Disabled).
- *configPLOTTING_MODE*: Trace tasks at the context switches (1: Enabled, 0: Disabled).
- *configTICKS_TO_EVAL*: The period time in milliseconds for any of above modes to run.
- *configEVAL_THRESHOLD*: The time between evaluation rounds. It must be long enough for tasks to delete themselves.
- *configUSE_TASK_SETS*: Consider more than one task set for evaluation. (1: Multiple task sets, 0: Only one task set).
- *configSET_SIZE*: The number of the task sets used in the evaluation.
- *configNUMBER_OF_TASKS*: The number of the tasks per a task set.

Python scripts for the evaluation metrics, the overhead and the plotting were also implemented.

V. Experimental Evaluations

Two evaluation methods were conducted in our work. In the first one, we implemented a python script to generate tasks synthetically. In the second evaluation method, we generated task sets with timing characteristics similar to the tasks in a real-world automotive software system. The first and the second types of tasks are called *synthetic* and *realistic* task sets respectively.

Period	Share
1 ms	3 %
2 ms	2 %
5 ms	2 %
10 ms	25 %
20 ms	25 %
50 ms	3 %
100 ms	20 %
200 ms	1 %
1000 ms	4 %
angle-synchronous ms	15 %

TABLE II: Task distribution among periods

Mode	0	1	2	3	4	5
Min. Max	0 1000	1001 2000	2001 3000	3001 4000	4001 5000	5001 6000
Period	30	15	10	7.5	6	5

TABLE III: 6 modes ranging from 0 to 6000 rpm with their periods in milliseconds.

A. Setup

The FreeRTOS was used as a real-time operating system to implement the multi-mode tasks and both scheduling algorithms on Raspberry Pi B+ board [1, 2]. The hardware board has ARM1176JZF-S 700 MHz processor and 512 MB of RAM. The UART interface of the Raspberry was used to generate an external interrupt. The corresponding interrupt service routine sets the global variable *externalInput* to the number of the mode determined by the evaluation script used in each respective evaluation method. The function *setupUARTInterrupt()* was implemented in the file *uart.c* located in the drivers directory in order to set up the UART interface.

Two types of task sets were generated: (1) synthetic and (2) realistic. For the synthetic tasks, a set of utilization values were generated in the range of 10% to 100% with a step size of 10 according to the *UUniFast* algorithm [5]. The approach in [8] was used to generate periods in the range of 1 to 100 ms with an exponential distribution. The WCET C_i^j of each task was calculated by $T_i * U_i$. The deadlines are implicit, i.e. equal to the period. A proportion p of those tasks were converted to multi-mode tasks with M modes. Note that the normal periodic tasks are multi-mode tasks with only one mode $M = 1$. The generated values above were assigned for the first mode of all the tasks. For the multi-mode tasks, the values for the remaining modes were scaled by the factor of 1.5, i.e., $C_i^{m+1} = 1.5 * C_i^m$, $T_i^{m+1} = 1.5 * T_i^m$. For each multi-mode task, one of the modes was then chosen to have the highest utilization while the WCETs of the other modes were reduced by multiplying them with random values between 0.75 and 1. According to the configurations above, 100 task sets were generated with 50% multi-mode tasks and cardinality of 10, i.e. the number of the tasks per a task set. The number of modes used in the evaluation are 5, 8 and 10. Each task set was assigned 10 seconds for execution and 5800 ms to delete itself.

Furthermore, realistic tasks that share the characteristics of

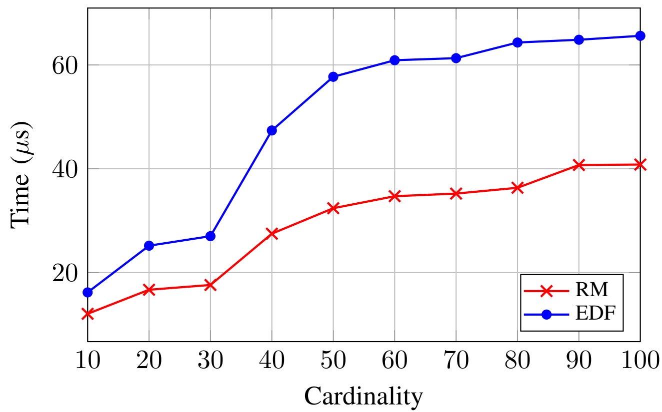
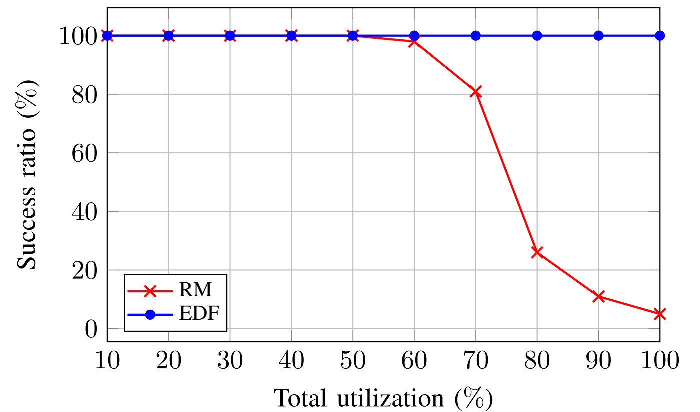


Fig. 1: The overhead of RM and EDF scheduling algorithms.

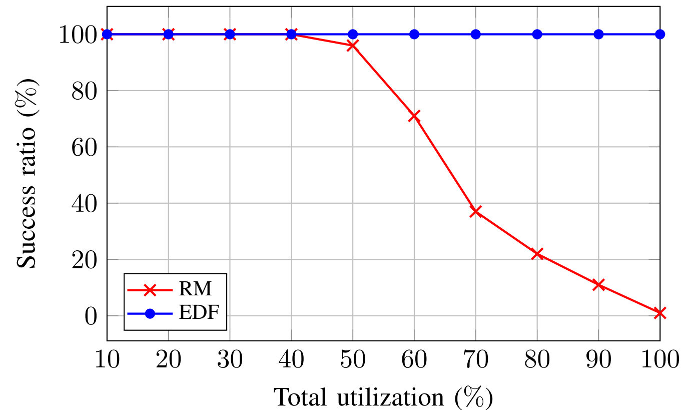
an automotive software system were generated as presented by Kramer et al. [10]. These characteristics cover the distribution of the tasks among the periods, the typical number of the tasks, the average execution time of the tasks and factors for determining the best- and worst-case execution times. Table II shows the distribution of the tasks among the periods [10]. The angle-synchronous tasks, which take 15% of all the tasks, are converted to multi-mode tasks as their worst-case execution time needs to adapt to their reduced period. In our case, the maximum engine speed is 6000 rpm with 4 available cylinders. For the conversion to multi-mode tasks, the engine speed was divided into 6 intervals and the periods were calculated by the upper bound of each mode as shown in Table III. The WCET of the tasks was assigned to the lowest mode. For the remaining modes, it was calculated based on the utilization of the first mode, i.e. $C_i = T_i * U_1$ and $U_1 = \frac{C_1}{T_1}$. Moreover, we implemented a crankshaft simulation that starts at an angular speed of 1 rpm and increases by 1000 rpm over 500 ms, and sends a signal every time the piston reaches the maximum position. This happens every one full rotation of the crankshaft. Once the simulated crankshaft reaches its highest speed of 6000 rpm, it will slow back down to 1 rpm. The acceleration/deceleration is steady during the whole execution. 100 task sets were generated per each utilization level from 10% to 100% with a step size of 10.

B. Results

The success ratio of the tasks and the overhead of the algorithms used in this subsection are defined in Subsection IV-D2. Figure 1 provides an evaluation for the overhead of both scheduling algorithms. As expected, the EDF algorithm has a higher overhead than the RM algorithm due to the dynamic priority assignment, where the priority of the jobs may change during the runtime. The EDF algorithm should always keep tracking of the absolute deadlines of the jobs, whilst the priorities according to RM algorithm are fixed prior to the execution, and the algorithm should just pick the next task in the ready list. We also observe that the overhead of both algorithms increases as the cardinality (i.e. the number of the tasks per a task set) increases. The increase of cardinality results in a longer ready list, which explains the growth in the overhead.



(a) M = 5



(b) M = 10

Fig. 2: Percentage of the schedulable task sets for 5 and 10 modes using the synthetic tasks.

Figures 2 and 3 show the impact of task utilization on the success ratio of the synthetic and realistic tasks respectively. They also compare between RM and EDF algorithms. What can be clearly seen in Figure 2 is that the EDF algorithm was able to find more feasible schedules than the RM algorithm. All the task sets with a utilization of up to 100% and up to 10 modes were feasibly scheduled under EDF. However, the RM algorithm could only achieve that for a utilization of up to 50% and 40%, and for a configuration of 5 and 10 modes respectively. After those levels of utilization, the success ratio of the RM algorithm decreases significantly. This is due to the fact that the EDF algorithm has a higher utilization bound than the RM algorithm.

If we now turn to the realistic tasks, we observe that the EDF algorithm performs worse than the RM algorithm, which is unexpected. It was able to schedule all the task sets with a total utilization of only 10%. It failed to schedule any task set with a total utilization of more than 50%. However, the RM algorithm could schedule all the task sets with a total utilization of up to 40% and was still able to find feasible schedules for some of the task sets with a total utilization of up to 60%. This behavior is due to the high overhead of the EDF algorithm and the distribution of the tasks among the periods in this data set. The realistic data set has more tasks with shorter periods than

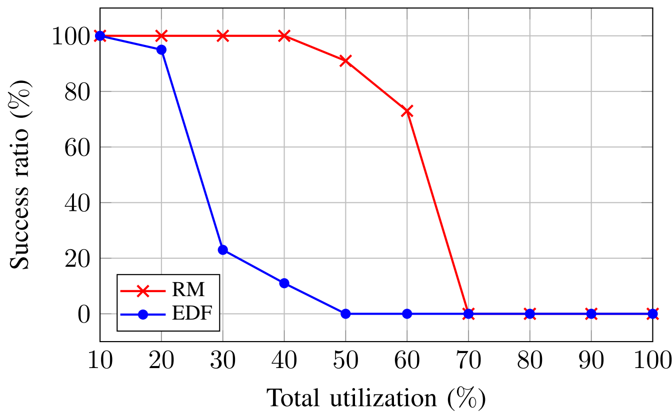


Fig. 3: Percentage of the schedulable task sets using the realistic tasks.

the synthetic one. For high workloads, the sum of the time required for the scheduling decision and the execution time of the job exceeds the relative deadline and then results in unschedulable task sets.

VI. Conclusion

In this paper, we evaluate the multi-mode tasks under the EDF and the RM scheduling algorithms in a real environment. To achieve that, the FreeRTOS real-time operating system was modified to implement this task model and both scheduling algorithms. Moreover, additional modifications were performed to provide configurable evaluation metrics. The experiments were performed on Raspberry Pi B+ board. Synthetic and realistic data sets were used in the evaluation. For the realistic data set, we generated angular tasks with periods tied to the rotation of a simulated crankshaft. The experiments confirmed that the EDF algorithm was in general able to find more feasible schedules than the RM algorithm for the synthetic task sets with high utilization values. However, it performed poorly when the realistic data set with relatively shorter periods was used, although a binary heap was used in the implementation to reduce the overhead of the scheduling decision. More feasible schedules were derived under the RM algorithm for this data set due to the low scheduling overhead.

VII. Future Work

Further work could usefully improve the implementation of the EDF algorithm by using a hardware accelerated binary heap to reduce the overhead caused by the dynamic scheduling [11]. However, such an implementation requires a special or additional hardware. Moreover, the system could be modified to handle task overruns.

VIII. Acknowledgements

This work is supported by the German Research Foundation (DFG) as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/) and as part of the Transregional Collaborative Research Centre Invasive Computing [SFB/TR 89].

References

[1] Freertos ported to raspberry pi. URL https://github.com/jameswalmsley/RaspberryPi-FreeRTOS.
[2] Raspberry Pi 1 Model B+. URL https://www.raspberrypi.org/documentation/hardware/.
[3] The FreeRTOS Kernel. URL http://www.freertos.org.
[4] An implementation of multi-mode real-time tasks, rate monotonic algorithm and earliest deadline first algorithm in FreeRTOS. URL https://github.com/Anas-Toma/multi-mode.
[5] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, 2005.
[6] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 2004.
[7] G. C. Buttazzo, E. Bini, and D. Buttle. Rate-adaptive tasks: Model, analysis, and design issues. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.
[8] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computers*, 57(9):1261–1276, 2008.
[9] W.-H. Huang and J.-J. Chen. Techniques for schedulability analysis in mode change systems under fixed-priority scheduling. *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 00:176–186, 2015. doi: doi.ieeecomputersociety.org/10.1109/RTCSA.2015.36.
[10] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free.
[11] N. C. Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H. Jones. Hardware-software architecture for priority queue management in real-time and embedded systems. *International Journal of Embedded Systems*, 6(4):319–334, 2014.
[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.
[13] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.
[14] N. Navet and F. Simonot-Lion. *Automotive embedded systems handbook*. CRC press, 2008.
[15] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, 1989.