Evaluation of Scheduling Algorithms for Embedded FreeRTOS-based Systems

Gessé Oliveira
dept. Computer Science
Federal University of Bahia - UFBA
Salvador, Brasil
gesse_oliveira@hotmail.com

George Lima
dept. Computer Science
Federal University of Bahia - UFBA
Salvador, Brasil
gmlima@ufba.br

***Abstract*—Dynamic priority real-time scheduling (DPS), such as Earliest-Deadline First (EDF), offers high levels of system schedulability, implying that under this scheduling policy system processing capacity can be utilized in an optimized way. However, this schedulability gain can be compromised due to overheads associated with managing dynamic priority queues, making the use of DPS less appealing in low processing capacity embedded systems. In this paper we assess the overheads associated with two different implementations of EDF in FreeRTOS running on an ARM-M4 architecture, comparing them against Rate-Monotonic scheduling (RMS), a classic fixed-priority policy. The two EDF implementations differ from each other by the manner priority queues are implemented, based either on min-heap (EDF-H) or on multiple linked lists (EDF-L). Runtime overheads and schedulability are taken into consideration for different types of task sets and system loads. Results indicate that the higher overheads of EDF-H may lead to poor performance with respect to RMS in some scenarios. Even presenting slightly higher overheads than RMS, EDF-L was shown to perform consistently better in all considered experiments.**

***Index Terms*—FreeRTOS, Real-time Systems, Embedded Systems, Microcontroller, EDF, Scheduler**

I. Introduction

Scheduling is key for ensuring real-time correctness. Classical scheduling policies are based on selecting at each scheduling instant the highest priority ready task to run. Depending on how priorities are assigned, different scheduling overheads are observed. When priorities are assigned in an offline manner, a simple vector can be used for ordering ready tasks. In this case, low overheads are expected since each priority is represented by a fixed position in the priority vector. This scenario differs from when task priorities are assigned online and may change over time. In this case, reordering ready task queue is necessary, which incurs extra overheads. This is one of the reasons fixed-priority scheduling (FPS) is the usual choice when implementing real-time embedded systems, which are often systems with low computing resources. However, dynamic priority schedulers (DPS) offer high system schedulability when compared to FPS. This means that there are systems for which a set of tasks cannot be feasibly scheduled under FPS while it can be under DPS [1], [2].

When designing a real-time embedded system, designers must choose the type of necessary execution infra-structure, either based on DPS or FPS. Depending on the architecture, there may be gains in choosing DPS over FPS since the former may compensate higher overheads with better resource usage. Examples of Earliest Deadline First (EDF), a DPS solution in existing operating systems (OS) can be easily found, mainly in Linux-based OS. This is the case of SCHED EDF patch [3]. Since standard Linux version 3.14 EDF scheduling has been included in the kernel. The ready task queue implementation is based on the red-black binary tree (RB-Tree), which keeps insertion and deletion for n tasks in $O(\log n)$. EDF was also considered at use-space level in the first version of Mapping Real-Time to POSIX (MARTOP) [4]. Broader discussions about EDF and FPS are available [5].

In small scale embedded systems, the use of EDF is less often seen although efforts towards using EDF in this domain, showing that scheduling overhead can be affordable are found. For example, using a specific kernel called ERIKA Enterprise, mechanisms for time representation and EDF scheduling were shown to be effective even when small microprocessors are considered [6], [7].

Our focus in this paper is on the use of DPS and FPS in FreeRTOS [8] for supporting real-time applications. This is an open source RTOS ideally suited to deeply embedded real-time systems implemented in architectures equipped with microcontrollers or small microprocessors. This type of application normally includes tasks with real-time requirements. Native FreeRTOS, however, does not implement EDF. Its scheduler is basically a FPS configured so as to offer round-robing scheduling per priority level.

Modifying FreeRTOS to add new scheduling functionalities can be done at user space levels [9], [10] but this kind of implementation exhibits high overheads. Experimental results in a kernel-space implementation has also been reported [11]. A recent study aimed at multi-mode applications running over a Raspiberry Pi platform [12]. DPS policies over ARM-M4 LPC2138 hardware have been considered [11]. Our study also aims at architectures with low computing resources. More specifically, our goal is to verify whether (and to what extent) EDF is advantageous when supporting embedded systems built on top of ARM-M4 STM32F407 hardware [13]. This hardware infra-structure offers relatively low computing resources and it is very popular for implementing embedded systems [14]. For carrying out our evaluation, two different implementations of EDF were considered. In summary, our contribution can be stated as follows:

- We describe two implementations of EDF for FreeRTOS at kernel space level for supporting real-time systems composed of periodic real-time tasks. Several functions and data structures were implemented. The considered implementations were inspired by previous work: priority queues were implemented by multiple linked lists [15] (EDF-L); the other use the classical min-heap data structure [16] (EDF-H) similarly to other studies [16], [12].
- Although both implementations of EDF have been considered before, to the best of our knowledge the evaluation of EDF-L has not been done in an actual OS and assessment has been done based only on simulation. Moreover, EDF-H, the classical implementation of dynamic priority queues, is the one usually taken for reporting EDF performance, with very few studies based on FreeRTOS. Experimental results on low computing capacity platforms is hardly seen.
- Our assessment takes several possible scenarios into consideration with varying processor utilization loads and task set configurations. Results from our experiments serve to characterize EDF against FPS in low processing capacity systems. This kind of information is useful for guiding designers of embedded systems.

The remainder of this paper is structured as follows. Section II gives background information on scheduling and FreeRTOS. It also presents previous work on implementing EDF in FreeRTOS. Our implementations of EDF are detailed in Section III while the assessment of them is given in Section IV. Section V concludes this paper.

II. Background and related work

A. Task model and motivation

The type of application we aim at is composed of periodic tasks to be scheduled on a single processor. Although the considered implementations of EDF can deal with other task models, our evaluation focuses on the periodic implicit deadline tasks as this kind is what is typically found in a large spectrum of embedded real-time systems (*e.g.,*sensor polling, control loops). We denote this type of application as a set τ with n independent real-time tasks τ_i, $i = 1, 2, \ldots, n$. Each task $\tau_i \in \tau$ is characterized by its worst-case execution time and its period, denoted respectively by C_i and T_i. Each task τ_i is assumed to release a (possible infinite) sequence of periodic jobs, each of which executes for no more than C_i time units and must finish execution by T_i time units after its release time. That is, a job of task τ_i released at instant r must finish by its (absolute) deadline $r + T_i$. The utilization of each task τ_i is defined as $U_{\tau_i} = C_i/T_i$ and system utilization is given by $U_\tau = \sum_{i=1}^{n} U_{\tau_i}$.

We consider two classical priority-based scheduling policies, RMS and EDF. Under both schemes, the scheduler selects the highest priority ready job to execute at any scheduling instant. No constraint on preemption is assumed, which means that at any time the scheduler may preempt an executing job in favor of another. For RMS, task priorities are given by the inverse of their periods. The lower the task period, the higher the priority of its jobs. This means that job priorities are known (and fixed) at design time, contrasting with EDF, for which priorities are defined by jobs' deadlines. The earliest the deadline of a ready job, the higher its priority. These characteristics imply that overheads observed in RMS schedulers tend to be smaller than those found in EDF since the latter must deal with re-ordering ready task queues as their priorities depend on their jobs' deadlines. Priority queues under RMS can be represented by a bit-vector and the existence of a ready job at a given priority level can be indicated by the corresponding position in the vector.

For the assumed system model, it is well known that under EDF, there is no deadline misses if and only if [1]

$$U_\tau \leq 1\,, \tag{1}$$

For RMS, a necessary and sufficient schedulability condition can be given via response time analysis [17]. This type of analysis computes the worst-case response time R_i for each task τ_i. The system is considered schedulable under RMS (or other FPS policy) if and only

$$\forall \tau_i \in \tau, R_i \leq T_i\,. \tag{2}$$

For the considered task model, the value of R_i can be computed as

$$r_i^k = r_i^{k-1} + \sum_{j \in \text{hp}(i)} \left\lceil \frac{r_i^{k-1}}{T_j} \right\rceil C_j\,, \tag{3}$$

with hp(i) representing the set of tasks with higher priority than that of τ_i. Recurrence (3) converges for any system that is feasibly scheduled via FPS. The recurrence starts with $r_i^0 = C_i$ and stops whenever $r_i^k = r_i^{k-1}$ or $r_i^k > T_i$. The latter case implies that τ_i may miss its deadline and so the system is considered unschedulable by RMS. In the former case, no task misses any of deadlines with R_i corresponding to the fixed-point solution of (3).

Note that there is no system feasibly schedulable by EDF or RMS if the respective conditions established by Equations (1) and (2) fail. As no system can be feasibly scheduled if it demands more than 100% of the system processing capacity, Equation (1) implies that under EDF the system can be fully utilized. This is not true for RMS. Figure 1 plots the percentage of randomly generated task sets considered schedulable by Equation (3). Each dot in the graph corresponds to $1\,000$ synthetic tasks sets each of which with 10 tasks. It can be seen that for systems with high processor utilization, schedulability under RMS rapidly drops whereas all generated task sets would be schedulable by EDF.

Figure 1, however, gives only a theoretical perspective as schedulability tests do not account for OS-related overheads, which includes costs associated with preemption, interrupt handlers, and managing priority queues. As DPS requires reordering of tasks in priority queues, EDF tends to be

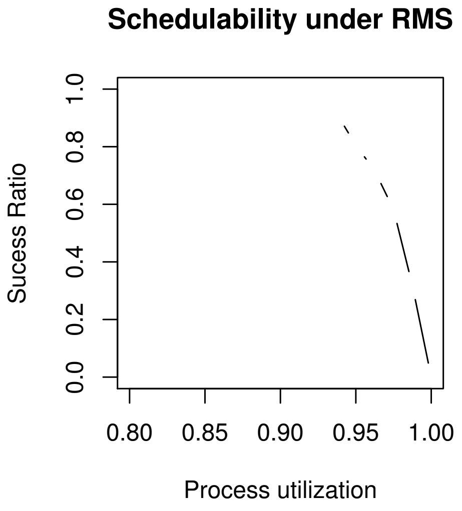


Fig. 1. Typical schedulability of systems scheduled via RMS, which contrasts with EDF that would deal with systems utilizing 100% of a processor.

associated with higher overheads. That is, less than what is established by these equations is actually available for running the system tasks. When looking at distinct scheduling policies, a natural question then arises: does the higher processor utilization in Equation (1) compensate for the imposed overheads associated with managing dynamic priority queues? The focus of this paper is on answering this question taking RMS and distinct implementations of EDF over FreeRTOS running on an ARM-M4 STM32F407 hardware [13]. As this hardware offers relatively low computing resources but it is very popular for implementing embedded systems [14], the type of evaluation presented in this paper provides useful information for designers.

B. FreeRTOS

FreeRTOS [8] is an open source real-time OS distributed under MIT license [18], which is ideally suited to deeply embedded real-time applications that uses microcontrollers. Its main features include: small footprint, low memory and processing requirements; widespread availability and support as it can be found in more 40 processor architectures [19], it is mostly written in C with small sections in assembler (PORT Layer). Applications can be implemented in FreeRTOS as a collection of independent threads.

FreeRTOS native scheduler is based on FPS. The number of priority levels is configured at compilation time. Each level is implemented as independent queues. Tasks assigned to the same queue are scheduled in a round robin fashion. That is, FreeRTOS can be easily configured to implement any FPS by assigning a single task per priority queue. In our experiments, we considered its native scheduler configured as RMS by assigning a task per priority level with priorities assigned according to RMS.

In order to customize the FreeRTOS kernel, as it is the case of implementing additional scheduling policies, some data structures and functions need to be understood:

- TCB. Tasks execute within their own context with no dependency on other tasks or the scheduler. Upon creation, each task is associated to a Task Control Block (TCB), which contains many task attributes. Information in TCB is to describe the task and its state, which includes task name, function pointer, priority, stack address among others.
- `pxCurrentTCB`. FreeRTOS has not a “running” list or state. The kernel maintains variable `pxCurrentTCB` to keep track of the thread id that is currently running.
- `xCreateTask()`. This function is responsible for both allocating a pointer and inserting the new task in the internal priority queue.
- `vTaskDelayUntil()`. It delays a task for a specific amount of time starting from a specified time instant.
- `vTaskStartScheduler()`. It starts the scheduler. From this function the Kernel will be managing all tasks of the system.
- `xTaskIncrementTick()`. This is an interrupt service routine (ISR) triggered by internal timer, configured as $1\,ms$ in this work. It is used to base time ticks through incrementing variable `xTickCount` indefinitely. Moreover, this function calls the scheduler after checking whether the priority of a new job is higher than job currently running.
- `vTaskSwitchContext()`. This is an interrupt handler in charge of performing context switches.

The aforementioned structure and functions were modified in some extent in our implementations of EDF.

C. Related work

As previously mentioned, FPS can be considered a native FPS scheduler in FreeRTOS. Although many implementations of EDF can be found in general, only a few of them target FreeRTOS. An implementation of EDF at user-space level has been described [9], [10]. Carrying out scheduling decisions at user-space level incurs extra overheads. To our knowledge, the only existing implementation of EDF over FreeRTOS that has been reported [12] target Raspberry Pi [12], a more powerful hardware platform. It is based on implementing priority queues as a classical *min-heap* [16]. This previous work deals with multi-mode real-time application. One of our implementations is also based on min-heap but our hardware platform provides much less computing resources. Further, this previous work found that FPS would be more effective for systems with high utilization. This conclusion is, to some extent, validated by our experiments. However, we founded that this does not hold in an alternative implementation of EDF we considered in this work, based on linked lists [15], [20]. Its evaluation has been carried out only via simulation. We provide here experimental results that sustain its good performance in an execution environment offering low level of processing resources. More details of this implementation is given in the next section.

III. Design and Implementation

This section summarizes our implementation. [1] We start by describing how we implemented the periodic task model and then present the considered EDF implementations.

A. Periodic tasks

In order to implement periodic tasks in FreeRTOS, some extensions of the TCB structure were needed to accommodate typical fields used in periodic real-time systems [21]. In addition to the original TCB attributes in FreeRTOS, the following ones with `uint64_t` data type were added:

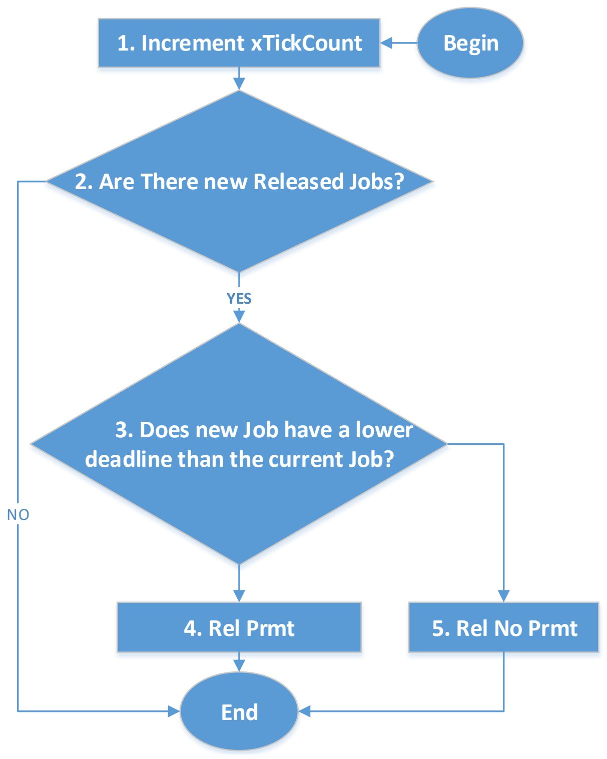
- `MsPeriod`. It represents the task period.
- `MsWCET`. This corresponds to the task worst-case execution time.
- `MsRelDeadline`. For the considered task model this should be equal to `MsPeriod` and represents the task relative deadline.
- `MsAbsDeadline`. This is the current job deadline. It is updated upon the release of a new job at time t as $\texttt{MsAbsDeadline} = t + \texttt{MsRelDeadline}$.
- `MsNextWakeTime`. The next waking time of a task, which is set to `MsPeriod` from the previous task release time.
- `MsNumberExecJob`. This variable was added for statistics purposes. It counts the number of released jobs from the time the system starts.

Furthermore, there are specific functions that must be modified and created so as to provide the required functionality related with periodic task activation and execution. The main ones are explained below.

- `xCreateTask()`. This function was modified to deal with the new TCB parameters. It enqueues the created task, as will be better explained soon in the next subsections.
- `vTaskStartScheduler()`. This piece of code is run to get the highest priority ready job. This is done before triggering the scheduler. The value of `pxCurrentTCB` is then updated accordingly to represent this job.
- `xTaskIncrementTick()`. An ISR timer was adapted to deal with `relprmt` and `relnoprmt` events. These events take place depending on whether preempting the current running job is or is not necessary, respectively. As this routine deals with the priority queue, its code depends on how the queue is implemented (Sections III-B and III-C). Figure 2 serves as a generic description.
- `xEndJob()`. This is a new function, introduced for notifying the scheduler upon the finishing of a job. Once the highest priority task is selected by the scheduler, a new task is scheduled to run. If there is no ready task, processor runs idle. In general, a task in our implementation have its code structured as follows.

```
void Task(void)
{
/*
Infinite loop to construct a "thread"
in FreeRTOS which implements a task
*/
 while(1)
 {
  /*Execute some action*/
  DoSomething();
  /*
  After task execution the kernel will
  be notified and the context switch will
  occur.
  */
  xEndJob();
 }
}
```

[1]Opensource implementation is available: https://github.com/gesseoliveira/FreeRTOS_EDF.



1. Increment xTickCount
Begin
2. Are There new Released Jobs?
YES
NO
3. Does new Job have a lower deadline than the current Job?
4. Rel Prmt
5. Rel No Prmt
End

Fig. 2. Flowchart of xTaskIncrementTick interrupt routine.

B. EDF queue based on min-heap (EDF-H)

Min-heap is a data structure usually employed in priority queue implementations. Our EDF queue implementation follows algorithms found in classical textbooks [16]. Its basic structure is given by the following code in C. The heap is made of a TCB vector with each entry for a ready task. An auxiliary variable (`count`) was considered to represent the current enqueued task jobs.

```
typedef struct
{
    TCB_t arr[HEAP_SIZE]; /*TCB array    */
    int count;  /* # jobs in the queue */
} Heap;
```

Three basic functions to manage the heap were implemented. Their prototypes are as follows.

```
int32_t getMin(Heap *);
TCB_t * extractMin(Heap *h, int16_t *);
void    insert(Heap *, TCB_t);
```

In a nutshell: `get_min()` returns the highest priority (*i.e.,*earliest deadline) of the currently enqueued jobs; `extract_min()` returns this job, removing it from the queue; and `insert()` enqueues a new released job. As the former function does not update the heap, it runs in $O(1)$ whereas the other two have complexity of $O(\log n)$ for n jobs.

C. EDF queue based on multiple linked lists (EDF-L)

An EDF queue can be implemented using a linked list with jobs enqueued in EDF order. However, for n tasks this would require $O(n)$ steps for inserting or searching. Aiming at reducing this cost, an implementation based on multiple linked lists was described [15]. This design is particularly interesting for small embedded systems, as it is the case considered in this work. We implemented this alternative approach.

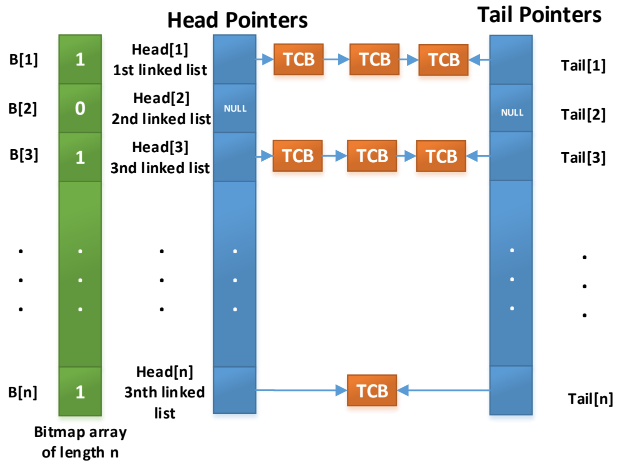


Fig. 3. Basic structure for implementing an EDF queue using multiple linked lists. Illustration taken from [15].

Figure 3 gives an overview of the considered implementation. A set of n linked lists stores the enqueued jobs for the n tasks with the following characteristics: (a) a list can hold more than one job and there may be empty lists; (b) the lists' nodes keep TCB pointers for the corresponding tasks; (c) jobs in a list k have deadlines greater than those in lists l for any $l > k$; (d) no job in list k at time t has deadline greater than $t + T_k$. Note that if jobs had fixed priorities, properties (c)-(d) would clearly hold since there would be a job per list. Nonetheless, even for the EDF priority assignment, the expected number of jobs per list is small.

The reasoning of EDF-L is that dealing with several small lists might be better than a single long list if the search space can be efficiently reduced. The bitmap vector, seen in the figure, plays a role in this regard. It indicates whether there is some enqueued job in the corresponding list, which can be done in $O(1)$ [15], [20]. This allows the scheduler to skip empty lists in an efficient manner. Another way of speeding up the search is via the tail pointer vector from which the greatest deadline job belonging to a given list can be easily accessed. Once the list to enqueue a job is found, a new released job can be inserted into the list (in its first position) in $O(1)$ in most cases.

Upon job releases at instant t, there are basically two scenarios for job insertions in the EDF-L queue, depending on the deadline of a newly released job J. If its priority is higher than that of the job J' executing at t, then it preempts J'. Assume that J' is an instance of task τ_k. As the deadline of the preempted job is guaranteed to be the second shortest deadline in the system at time t (due to EDF ordering), it can be inserted in the list in $O(1)$ as follows. First, the list l on which J' will be inserted is found, which can be done in $O(1)$ using the bitmap vector. The value of l corresponds to the least value for which the bit indicates a non-empty list among the lists that can hold J', that is, those lists such that $l \geq k$.

The other job insertion scenario is more costly. If the deadline of the new job J of a task τ_k released at time t is greater than the currently executing job, it must be enqueued. However, in this case, inserting J into list k is not enough since there might be previously enqueued jobs in lists $l > k$ with deadlines earlier than $t+T_k$. To comply with property (d) previously stated this operation requires removing such jobs from lists l and inserting them into list k, which takes linear time. Again, as the number of jobs per list is expected to be low, such operation is likely to be fast.

Finally, when a job finishes executing, the corresponding list is updated accordingly in $O(1)$. Likewise, a new job can be selected to execute in constant time as jobs are in EDF order in the lists. If processor is made idle upon finishing executing a job, the bitmap vector is updated accordingly, also in $O(1)$.

The basic structure for implementing each of the n lists is given below.

```
typedef struct {
    int     Qnt   ;
    TCB_t  *Head;
    TCB_t  *Tail;
} PList_t;
```

The prototypes of the basic functions that carry out the previously described operations are given below. In brief: `list_add_head()` and `list_rm_head()` inserts and removes a TCB in a list, respectively; and `next_list()` returns the id l of the non-empty list that immediately follows a list k of interest. For a list k this latter function returns the id $l > k$ for which there is some job.

```
unsigned int
   list_add_head(unsigned int, TCB_t *, PList_t *);

unsigned int
   list_rm_head(unsigned int, TCB_t *, PList_t *);

unsigned int next_list(uint64_t, uint64_t);
```

IV. Experimental Evaluation

For our evaluation, task sets were synthetically generated using the UUniFast algorithm [22]. According to this procedure, utilization values of each task follows a Uniform distribution and sum up to the desired task set utilization. Once the utilization of a task was generated, its period was randomly chosen within the desired interval and its execution time was then computed. Task execution time was implemented as a loop according to its generated execution time value. The hardware platform for the experiments consisted of an ARM CORTEX-M4 micro-controller with a 168 MHz CPU, 512 KB flash memory and 128 KB of RAM memory. As this platform is mostly used for small embedded systems, in our evaluation we considered systems with no more than 60 tasks.

Several task set configurations were considered. We highlight here results obtained for two types of task sets, called hereafter *easy* and *hard*. The former corresponds to task sets containing tasks with periods varying from 2 to 1 000 clock ticks whereas tasks in the latter type have periods in between 2 and 50 clock ticks. The system was configured such that each clock tick takes 1 ms. Note that hard task sets cause a high number of scheduling events, possibly leading to higher overheads when compared to easy task sets. During the experiments, time was measured as clock cycles by means of the clock cycle counter (CYCCNT), a feature available in Cortex-M4 as part of the DWT (Data Watchpoint and Trace).

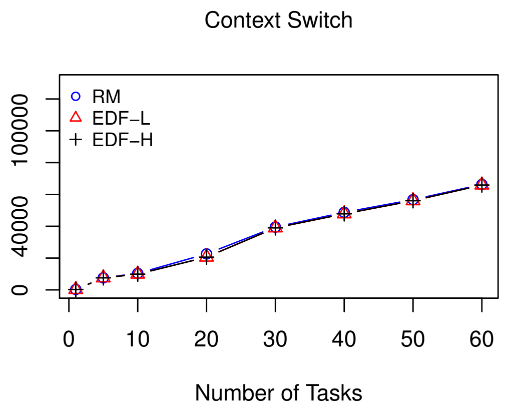
The following sections compare EDF-H and EDF-L against RMS. Recall that FreeRTOS offers native support for RMS. Observed effects due to context switching, scheduling overhead and schedulability are reported. Experimental setups were defined for easy and hard task sets, varying the number of tasks in the sets and the task set utilization. For each setup, reported results correspond to the average of 30 different task sets, each of which executed during 10 seconds.

A. Context switches

As RMS and EDF behave differently in terms of scheduling decisions, it is interesting to observe whether there are significant effects in terms of generated context switches. If it were, this could imply (or partially explain) their differences in observed scheduling overheads. Figure 4 shows the found results for task sets with utilization of 85%. As expected, the number of context switches varies for different task set sizes and is much greater for hard task sets (about 50% more). However, differences between RMS and EDF are marginal. Similar trends were observed in other experiments with different task set utilization setups. In conclusion, differences in context switches cannot play a central role in explaining possible differences of scheduling overheads for RMS and EDF.

B. Scheduling overhead

Scheduling overhead is measured in this section by computing the overall time the system spent on running scheduling related functions as compared to the total of 10 seconds each experiment ran. Results are given in Figure 5. As expected, RMS presents the lowest overhead when compared to both EDF implementations. EDF-L performs better than EDF-H. As the number of tasks grows, overhead proportionally increases for all approaches. Further, the overall overhead for scheduling



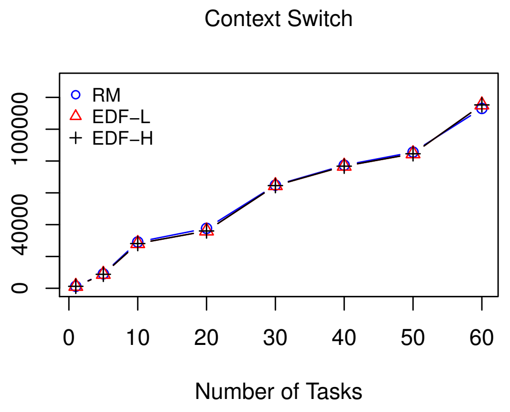


Fig. 4. Observed context switches for easy (on top) and hard task sets. All generated tasks sets tau had utilization equal to $U_\tau = 0.85$.

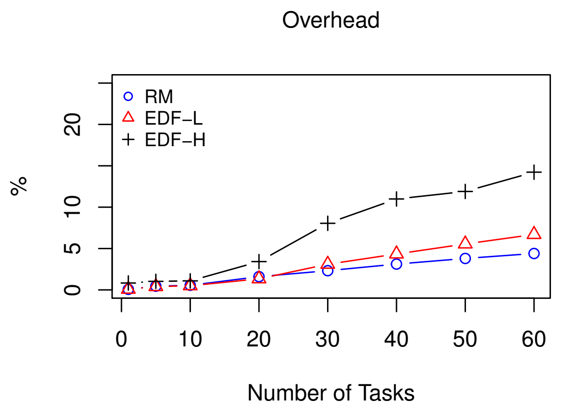
hard task sets is up to 50% higher than what was observed for easy task sets. The difference follows the same trend observed for context switches, as shown in the previous section. This means that the overhead associated with managing priority queues is what causes most differences between the considered scheduling implementations in terms of overheads.

C. Schedulability

As usual, we measure schedulability by verifying whether some job misses its deadline during execution. That is, schedulability is being measured by

$$\text{Success Ratio} = \frac{\text{\# task sets with no missed deadlines}}{\text{\# generated task sets}}$$

Figure 6 shows the found results per task set size for task sets with 85% utilization. For easy task sets, all three scheduling approaches can successfully manage task sets with up to 30 tasks. After that the performance of RMS drops. Interestingly, RMS performs better than EDF-H for sets with more than 50 tasks. That is, the scheduling overhead for managing the min-heap becomes (recall Figure 5) a bottleneck for EDF-H. In this case, the theoretical advantage of EDF



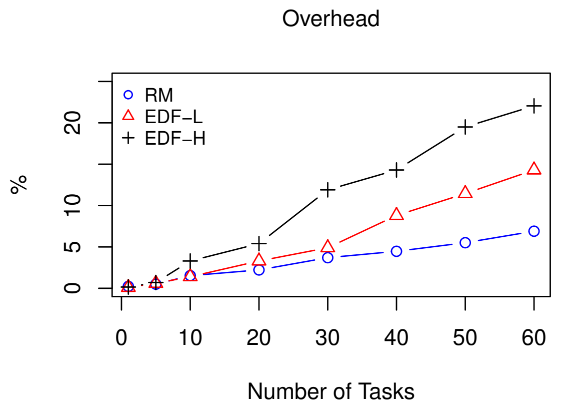
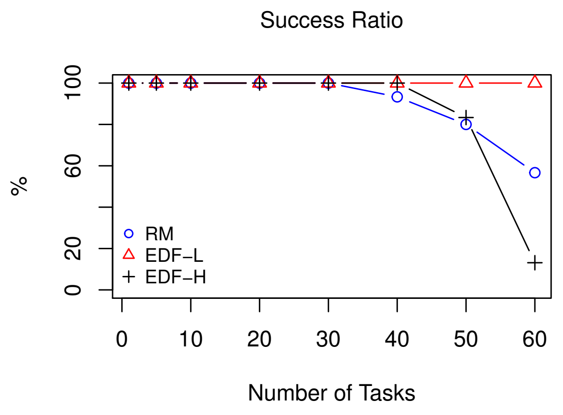


Fig. 5. Scheduling overhead for easy (on top) and hard task sets. All generated tasks sets tau had utilization equal to $U_\tau = 0.85$.



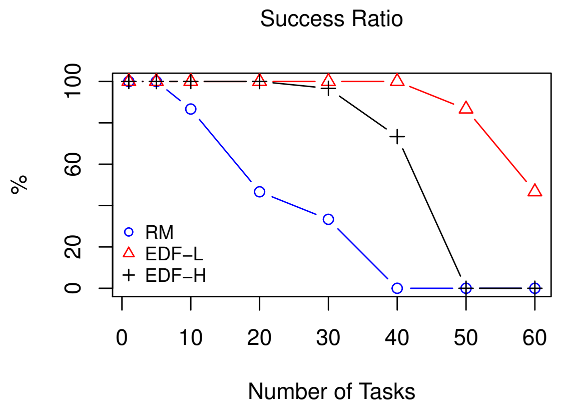


Fig. 6. Schedulability for easy (on top) and hard task sets for several sizes of task sets, all of them with utilization 0.85.

(illustrated in Figure 1) over RMS does not hold for heap-based priority queues. EDF-L, on the other hand, is capable of scheduling all easy task sets.

The relative good performance of all three implementations observed for easy task sets does not hold when hard task sets are considered. As can be seen in Figure 6, deadline misses under RMS starts to appear in task sets whose size is as low as 10. Once again, EDF-H performs worse than EDF-L. While EDF-L was able to successfully schedule half of the generated tasks sets with 60 tasks, the success ratio of RMS and EDF-H is zero when the number of tasks is 40 and 50, respectively. In conclusion, the theoretical schedulability of EDF partially compensates for the higher overheads in managing dynamic priority queues although the use of heap is shown to consistently perform worse than the implementation using multiple linked lists.

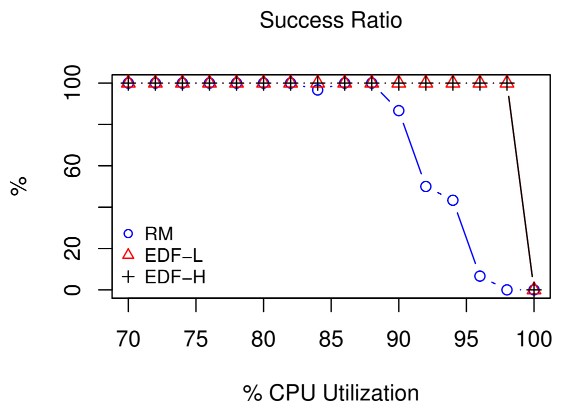
Similar conclusions can be drawn from Figure 7, which shows the schedulability trend observed for easy and hard task sets per utilization. Two extreme scenarios were considered, namely easy task sets with 10 tasks and hard ones with 60. Note that for the former case, EDF-H performs well w.r.t. RMS. For stressed systems (*i.e.,*hard and large task sets), the opposite trend is observed. Nonetheless, EDF-L consistently shows to be the best implementation choice in all considered scenarios.

V. Conclusion

We experimentally compared EDF against RMS aiming at characterizing these policies for small embedded real-time systems based on FreeRTOS. As this application domain usually employs hardware with low computing resources, the experimental characterization we provided serve to support system designers.

Two implementations of EDF queue were considered, based on the classical min-heap data structure (EDF-H) or on multiple linked lists (EDF-L). EDF-L was shown to consistently perform better than EDF-H in all experimental setups. Further, although RMS was shown to induce the lowest scheduling overheads, this was shown not to compensate for its poorer schedulability performance when compared to EDF-H.

Although the debate on the use of dynamic priority versus fixed-priority scheduling is not new, it is usually based on theoretical or simulated results. To the best of our knowledge, this is the first work to consider an actual implementation of



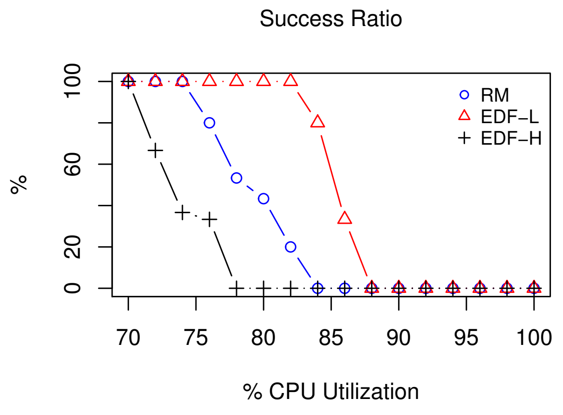


Fig. 7. Schedulability for easy (on top) and hard task sets varying utilization. Easy and hard task sets are of size 10 and 60, respectively.

EDF-L in a RTOS.

As EDF makes a better utilization of the processor, spare processing capacity can be used for other purposes. Future steps of this research will include implementing and evaluating energy-saving mechanisms under EDF for small embedded systems.

Acknowledgment

This work was partially funded by FAPESB grant number APP0042/2016.

References

1 Liu, C. L. and Layland, J. W., "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.
2 Baruah, S. K., Rosier, L. E., and Howell, R. R., "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems*, vol. 2, no. 4, pp. 301–324, 1990.
3 Faggioli, D., Trimarchi, M., and Checconi, F., "An implementation of the earliest deadline first algorithm in linux," in *ACM Symposium on Applied Computing*, 2009, pp. 1984–1989.
4 Stahlhofen, A. and Zobel, D., "Linux sched deadline vs. martop-edf," in *IEEE International Conference on Embedded and Ubiquitous Computing*, 2015, pp. 168–172.
5 Buttazzo, G. C., "Rate monotonic vs. EDF: Judgment day," *Real-Time Systems*, vol. 29, no. 1, pp. 5–26, 2005.
6 Buttazzo, G., "Efficient edf implementation for small embedded systems," *Workshop on Operating Systems Platforms for Embedded Real-Time applications*, 2006.
7 Cho, K.-M., Liang, C.-H., Huang, J.-Y., and Yang, C.-S., "Design and implementation of a general purpose power-saving scheduling algorithm for embedded systems," in *IEEE International Conference on Signal Processing, Communications and Computing*, 2011, pp. 1–5.
8 Barry, R., *Mastering the FreeRTOS^{TM} Real Time Kernel*. Wiley, 2016.
9 Paez, F. E., Urriza, J. M., Cayssials, R., and Orozco, J. D., "Freertos user mode scheduler for mixed critical systems," in *IEEE Argentine Conference on Embedded Systems*, 2015, pp. 37–42.
10 Kase, R., "Efficient scheduling library for freertos," Master's thesis, Kth Royal Institute of Technology, Stockholm, Sweden, 2016.
11 Belagali, R., Kulkarni, S., Hegde, V., and Mishra, G., "Implementation and validation of dynamic scheduler based on lst on freertos," in *IEEE International Conference on Electrical, Electronics, Communication, Computer and Optimization Techniques*, 2016, pp. 325–330.
12 Anas Toma, V. M. and Chen, J.-J., "Implementation and evaluation of multi-mode real-time tasks under different scheduling algorithms," *Operating Systems Platforms for Embedded Real-Time applications*, 2018.
13 STMicroelectronics. (2016) Datasheet ARM-M4 STM32F4 family. Rev.B. [Online]. Available: https://www.st.com/resource/en/datasheet/dm00037051.pdf
14 ——. (2020) Who we are. [Online]. Available: https://www.st.com/content/st_com/en/about/st_company_information/who-we-are.html
15 Pathan, R. M., "Design of an efficient ready queue for earliest-deadline-first (edf) scheduler," in *Design, Automation Test in Europe Conference Exhibition*. Operating Systems Platforms for Embedded Real-Time applications, 2016, pp. 293–296.
16 Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to Algorithms*, 3rd ed. Massachusetts Institute of Technology, 2009.
17 Audsley, N., Burns, A., Richardson, M., Tindell, K., and Wellings, A., "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.
18 Barry, R. (2020) License freertos. [Online]. Available: https://www.freertos.org/a00114.html
19 STmicroelectronics. (2020) FreeRTOS support architectures. [Online]. Available: https://www.st.com/en/embedded-software/freertos-kernel.html
20 Pathan, R. M., "Unifying fixed- and dynamic-priority scheduling based on priority promotion and an improved ready queue management technique," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 209–220.
21 Buttazzo, G. C., *Hard Real-Time Computing Systems*, 3rd ed. Springer US, 2011.
22 Emberson, P., Stafford, R., and Davis, R., "Techniques for the synthesis of multiprocessor tasksets," *WATERS*, pp. 6–11, 2010.