

# Ravenscar-EDF: Comparative Benchmarking of an EDF Variant of a Ravenscar Runtime

Paolo Carletto<sup>(✉)</sup> and Tullio Vardanega<sup>(✉)</sup>

Department of Mathematics, University of Padua, 35121 Padua, Italy  
carletto.paolo@gmail.com, tullio.vardanega@math.unipd.it

**Abstract.** Subsequent to the publication of the seminal work by Liu and Layland in 1973, researchers and practitioners alike started discussing which online scheduling algorithm was to be preferred between FPS and EDF. Results published in 2005 sustained the superiority of EDF, already proven in theory, also from an implementation perspective. With this work, we aim at digging deeper into the roots of those results. To this end, we took the first-ever instance of an Ada Ravenscar runtime, with its FPS scheduler, combined with its IPCP locking policy companion, and developed a variant of it that implements EDF scheduling coupled with DFP locking. In this manner, we were able to transparently attach those two runtime variants to a suite of synthetic benchmarks, which we used to perform an extensive quantitative comparison between those two runtimes, getting to the bottom of where one prevails on the other.

**Keywords:** Ravenscar profile · Earliest Deadline First · Deadline Floor Protocol · Analysis and development · Performance comparison

## 1 Introduction

The publication of the seminal work by Liu and Layland [6] back in 1973 sparked a great deal of interest on the question of which online (preemptive) scheduling policy for single-core processors was best. From that moment, the real-time systems community divided between two camps: one supporting Rate Monotonic (RM); the other championing Earliest Deadline First (EDF). From the usage perspective, this confrontation seems to have been won by the RM camp, as the most part of existing technology, whether general-purpose operating systems or real-time kernels implements Fixed Priority Preemptive Scheduling, hence RM. Arguably, this happens because RM is easier to implement on top of runtimes that do not support the notion of timing deadline natively. Implementation is simpler also because a fixed-priority constant value can be assigned per task and simply copied to each recurrent job of it, without the per-job dynamic computation that the deadline driven approach requires. A simplistic technique to implement a deadline-driven scheduler on top of a priority-based runtime directly maps absolute deadlines to the existing priorities. In that manner, any real-time kernel that supports priorities can also support deadline-driven scheduling, at

the cost of computing the deadline-to-priority mapping at any job release, and of resolving the conflicts that may arise when multiple deadlines map to the same priority. Some authors [7] suggested that this additional implementation burden and the runtime overhead stemming from dynamic priority management was the prime reason for EDF not being supported in commercial real-time kernels, in spite of it being known that EDF would maximise the total schedulable utilization of the processor [11].

The work we present here illustrates an empirical, quantitative comparison between concrete implementations of the RM and EDF variants of a real-time kernel embedded in the Ada Ravenscar runtime developed by AdaCore for the Leon processor<sup>1</sup> family.

Arguably, our work yields two distinct contributions. First, it makes a very fair comparison as the *sole* elements that change in the systems being confronted are the scheduling operations that implement RM and EDF in the corresponding runtimes. As the application stays unchanged, any performance difference is directly ascribable to the scheduling variant being used. Second, it stresses each system to the limits discussed in the literature [11] using exactly the same, unchanged, application software, as the switch of scheduling policy is completely transparent to it.

On those two premises, we have created a suite of synthetic benchmarks that aims to (and if fact does) single out the conditions under which one policy performs better than the other, to help appreciate why that happens more profoundly – we think – than discussed in [7].

## 2 The RM-to-EDF Transformation Process

### 2.1 The Ada Ravenscar Profile

The Ravenscar profile [3,4] is an important asset of the Ada programming language. Especially when used for embedded targets, it allows simple yet flexible real-time systems, fully analysable for their timing feasibility (aka schedulability), to be implemented on a runtime system that is itself lean, small and fast, fit for being engineered to the highest level of integrity. The profile is specified in the Ada standard (since its 2005 revision) via a collection of restrictions on the full language. It is defined to support applications that use a statically-defined set of library-level tasks scheduled by the fixed priority scheme known as “FIFO Within Priorities”. The Ada Ravenscar Profile is especially designed for those embedded applications that have tight timing and memory requirements, high-integrity (eg. safety-critical) constraints, and want to dispense with the heavy constraints of traditional cyclic scheduling.

The first-ever Ravenscar runtime to be released to industrial use was produced by AdaCore for the Leon processor family, and named *GNAT-2012-LEON-ELF-BIN*. That technology originated from a fork of the Open Ravenscar Real-Time Kernel (ORK+) developed by the Technical University of Madrid<sup>2</sup>.

<sup>1</sup> <http://www.adacore.com/gnatpro-safety-critical/platforms/erc32/>.

<sup>2</sup> <http://www.dit.upm.es/~ork/index.html/>.

The Leon processor that was targeted by that runtime is a 32-bit CPU micro-processor core, based on the SPARC-V8 RISC architecture and instruction set. It was originally designed by the European Space Research and Technology Centre (ESTEC), part of the European Space Agency (ESA), and subsequently developed, in synthesizable VHDL and maintained by Gaisler Research, now Cobham Gaisler.

An application conforming to the Ada Ravenscar Profile comprises  $N$  tasks that are due to execute concurrently on the same processor core. All such tasks are defined to have a period (denoted by the symbol  $T$ ) that is the minimum time span that elapses between two subsequent releases of it, a relative deadline ( $D$ ), and a worst-case execution time demand ( $C$ ). For the system to be feasible, any task  $\tau_i$  arriving at time  $t$  must be able to execute for its maximum computation time ( $C_i$ ) by its absolute deadline, which falls at time  $t + D_i$ .

With fixed priority scheduling, each task is assigned a static priority ( $P$ ), which is attached at release to all of its recurring jobs. For best schedulability results, the task priority is derived from its relative deadline (or equivalently, its rate, when  $D = T$ ). Two tasks with relative deadlines  $D_i$  and  $D_j$ , such that  $D_i < D_j$ , will be assigned priorities such that  $P_i > P_j$ .

Under the Ravenscar Profile, tasks may contend for exclusive access to shared resources that are enclosed within protected objects. To warrant predictable arbitration of such contention, protected objects are assigned a static ceiling priority, and access to the protected object is controlled by the priority ceiling protocol (PCP) [9]. The form of PCP assumed in the Ravenscar Profile is the “immediate” version (IPCP) of it, in which the contending task’s priority is raised to the resource ceiling immediately upon access to the resource.

The Ada runtime that we used in this work does not support the 2012 version of the language, and therefore does not allow the user to directly represent relative deadlines in the program code. We circumvented that limitation by providing an ad-hoc API, which is only used during task elaboration and had no impact on our comparative performance evaluation.

To perform our experiment, we modified the original Ravenscar runtime to support Earliest Deadline First for scheduling [2], and the Deadline Floor Protocol [5] for locking. Thanks to the substantial semantic equivalence between FPS with IPCP and EDF with DFP, we were able to compare those two runtime variants, which only differ in a small number of (important) scheduling operations.

## 2.2 Turning Priorities into Deadlines

Earliest deadline first (EDF) is a dynamic scheduling algorithm that places tasks in a ready queue sorted by absolute deadline in increasing order. Whenever a scheduling event occurs (job completion, job release, synchronization lock released), the task with the shortest absolute deadline is dispatched to execution.

The Deadline Floor Protocol (DFP) used in an EDF-scheduled system when tasks contend for shared resources, is structurally equivalent to the Immediate

Priority Ceiling Protocol (IPCP) used in a system scheduled under fixed priorities. Under the DFP, every resource is assigned a relative deadline equal to the shortest relative deadline of the tasks that may use it. The relative deadline attribute of a shared resource is called its *deadline floor*, a pun to the sought symmetry with the *priority ceiling* defined for all priority ceiling protocols. The key idea in the DFP is that the absolute deadline of a task might be temporarily shortened while accessing a shared resource, increasing its preemption privilege under EDF. Given a task with absolute deadline  $d$  that accesses a resource with deadline floor  $D_F$  at time  $t$ , the absolute deadline of the task becomes  $d := \min(d, t + D_F)$  while holding the resource.

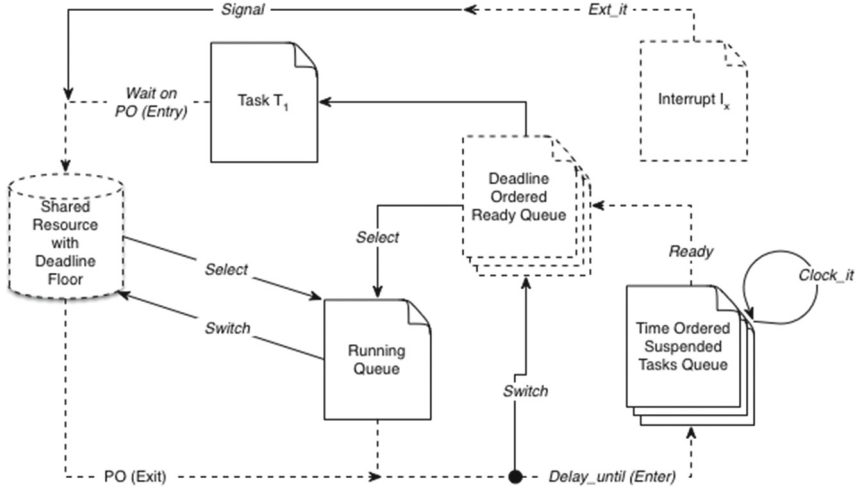
Given these definitions, a Ravenscar-EDF Profile [1,2] with DFP presents two main differences to the original one:

1. A different Task Dispatching Policy: the default “*FIFO Within Priorities*” scheduling policy is replaced by “*EDF*”, while retaining the logic that jobs with identical deadlines (a much rarer event than having identical priorities) would have FIFO ordering in the ready queue. As a Ravenscar runtime has only one scheduling policy, we changed the fixed priority default to EDF directly inside the source code of the Ravenscar-EDF runtime. Notably, since our EDF system variant works only in an “*EDF*” mode we did not need to follow the Ada standard “*EDF Within Priorities*” scheduling which was designed to allow it to coexist with other dispatching policies;
2. A different Locking Policy for shared resources: the default locking policy, IPCP [9] is replaced by DFP [5], designed for EDF scheduling. As a Ravenscar runtime can only have one locking policy, we changed IPCP to DFP directly in the source code of the Ravenscar-EDF runtime.

To implement the new scheduling model and the new locking policy of the Ravenscar-EDF runtime [2], we had to modify some fundamental data structures in the original runtime; specifically, those related with the handling of tasks and protected objects, as shown in Fig. 1. Let us now illustrate the changes we applied in some detail.

First, we needed to support `pragma Relative_Deadline` in place of the original `pragma Priority`, to attach the EDF scheduling attribute to application tasks. The attribute value declared by the user is used at initialization time to set the new `Base_Relative_Deadline` attribute, added to the Ada Task Control Block, which never changes during program execution. In turn, this value serves to maintain two new task attributes:

1. **Active\_Relative\_Deadline**: to represent the task’s relative deadline, which the runtime must consider for the purposes of scheduling. The value of this attribute may change because of DFP, which may temporarily lower it when the task acquires a protected object. If this attribute were missing, DFP would overwrite the `Base_Relative_Deadline` attribute thereby preventing correct restoration of the original base relative deadline of the task when leaving the protected object.



**Fig. 1.** An outline of the internals of the EDF runtime. Dotted lines represent the data structures that have to be changed to support EDF and DFP.

2. **Active\_Absolute\_Deadline**: to determine the task position in the ready queue, at every dispatching point. The absolute deadline of a job of  $\tau_i$  released at time  $t$  is  $t + D_i$  where  $D_i$  is the task's relative deadline.

Next, we had to introduce a new directive `pragma Deadline_Floor` to replace `pragma Priority` for setting the locking attribute of protected objects.

The biggest changes obviously concerned the scheduling policy, which required replacing every priority based criteria in use to manage the ready queue with a new deadline based policy.

This change entailed a full revamp of the queueing system in the runtime. Whereas the original version compared static priority values to determine one task's position in the ready queue, the EDF version compares absolute time values, which are computed at release for every new job. To avoid unduly biasing the evaluation, we chose to retain in the EDF runtime exactly the same linked-list organization that was implemented in the original version. A linked list that needs linear traversal for positioning a task in the ready queue is obviously not the most efficient choice of runtime structure: the FPS solution should rather use an array of per-priority queues; The EDF solution instead a binary tree. Yet, we did not go that way, to make the comparison fair.

In contrast with [2], we decided to retain the `Delay_until` API for the EDF runtime, so that application tasks would have an identical API to invoke across the two runtime variants. Again, this choice may seem to deflect from the Ada standard, but in a Ravenscar-EDF runtime, the EDF semantics of `Delay_until` is very straightforward and allows the application to stay unchanged on the switch of runtime. To this end, we moved to the `Wakeup_Expired_Alarms` procedure the operation of computing the new absolute deadline on task resumption

and placing it in the ready queue accordingly, so that `Delay_until` only had to assert the preemption variable when the running task is taken off the CPU. Notably, this simplification eased our refactoring of interrupt handling.

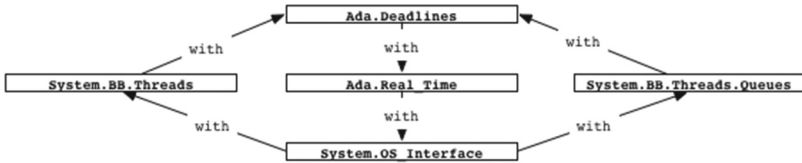
Further changes were needed to support DFP, but they turned out to be easy because DFP works in exactly the same way as IPCP once the runtime replaces priorities by deadlines. Both the original (IPCP) and the new version (DFP) modify the corresponding attribute of the task that gains access to a protected object (PO). IPCP raises the task's current priority to the priority ceiling of the PO; DFP lowers the relative deadline of the task to the deadline floor of the PO.

To implement DFP, we had to add a new `Floor` attribute to the `Protection` record of the PO, which stores the deadline floor attribute assigned to the protected object at declaration. We then added a `Caller_Relative_Deadline` attribute to store the relative deadline of the task that acquires the PO, to allow restoring the task's original relative deadline on leaving the PO. We changed the `Initialize_Protection` procedure that is called when `pragma Deadline_Floor` is encountered by the main program during initialization, to set the value of `Floor` attribute in the PO. Finally, we changed the `Lock` and `Unlock` procedures so that they update the relative deadline of the task on access to the PO and restore the original one on exit, respectively.

### 2.3 Implementation Challenges

Implementing the changes described in the previous section caused some development problems, which may be worth recalling to illustrate the bottom-up repercussions of top-down pressure of language changes.

First of all, we incurred the circular dependency shown in Fig. 2 when, following the suggestions in [5], we included the package `Ada.Deadlines` as a dependent of `Ada.Real_Time`. Using the “`limited with`” clause did not help, since it does not apply to subtypes, which deadlines are in fact.



**Fig. 2.** Circular dependency caused by the introduction of package `Ada.Deadlines`.

We solved that problem by moving all the relevant contents of `Ada.Real_Time` into `System.BB.Time`, so that introducing `System.BB.Deadlines` as a child unit to it did not cause any visibility issues. Not a clean solution, indeed, though effective for the particular internal organization of the runtime we used.

A much bigger and more fundamental problem we incurred had to do with the handling of interrupts. The crux of it is that interrupt handling intrinsically assumes priorities, which – in principle – do not belong in an EDF system.

In the original Ravenscar runtime (as well as in the Ada standard), interrupts have their own set of priority values, defined by the `Interrupt_Priority` type, at the top of the interrupt range, from 241 to 255 for our processor target. The intent is that interrupt handlers go directly to the top of the ready queue and concur solely with other interrupts as described in the left of Fig. 3. This mechanism is very natural for a priority based system, but it does not fit well in a deadline-based runtime as long as they have no deadline attribute (as reported in the center of Fig. 3).

The solution that we adopted reserves a fictitious position at the top of the ready queue for the current interrupt handler. If an interrupt handler is active, that position is used and the deadline-based part of the queue is frozen. If no interrupt is running, that position is not in use and cannot be contended. We unlock the queue when the handler exits, thereby enabling normal tasks to execute again. This solution does *not* support interrupt nesting, but it could be extended to it by making the top position point to a priority-ordered queue reserved for interrupts. The right part of Fig. 3 shows the reengineering of the ready queue from the original version to the EDF one with support for interrupts.

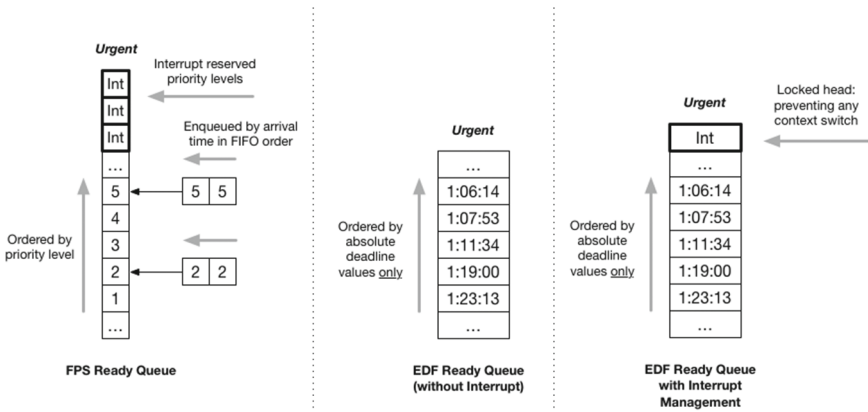


Fig. 3. Different ready queue organizations.

To implement these mechanisms, we changed the `Change_Priority` procedure to use a persistent boolean flag, to tell whether an interrupt handler is running or not. Asserting that variable effectively inhibits context switch and enables interrupt handlers to execute undisturbed.

The `Change_Priority` procedure is called inside `Interrupt_Wrapper`, the container that enables interrupt handlers to execute on their own stack, transparently to normal tasks.

The not-very-elegant nature of the solution that we devised for interrupt handling was one of the two major integration problems that we encountered. The other arose in evaluating the `Default_Relative_Deadline` attribute, which

mirrors the `Default.Priority` value of the FPS runtime, assigned to all priority attributes that lack explicit user setting. The FPS attribute is set, arbitrarily, to the value that best approximates the medium point in the standard priority range, excluding the top subrange reserved for interrupts. In our FPS runtime, the chosen value was 120. In a deadline based system, there is no sound value to choose. A small value penalizes urgent tasks that have been set an explicit relative deadline by the user. A large value may penalize the “defaulted” task if it happened to live in a system with many urgent tasks.

Since none of our synthetic tasks had uninitialized deadlines, we were free to arbitrarily set the `Default.RelativeDeadline` attribute value to zero and let it be overwritten by the relative deadline that the program assigned to the task at declaration. A better solution should be defined for general use.

### 3 Evaluation Benchmark

In keeping with the empirical nature of our experiment, we based our evaluation approach on the generation, categorization and execution of a large number of test scenarios designed to thoroughly stress both runtime variants.

To this end, we defined three types of synthetic tasks - Short, Mid and Long -, each with corresponding magnitude of period  $P$  and worst-case execution time  $C$ . We then composed those tasks into tasksets with different cardinality (which ranged from 30 to 180 concurrent tasks) and a variety of CPU utilization scenarios between 75% and 125%.

We further duplicated the tasksets into one version with *implicit* deadlines and the other with *constrained* deadlines, using Rate Monotonic or Deadline Monotonic assignments respectively for the FPS benchmarks.

Figure 4 depicts the automation engine that we constructed to generate, build, execute and record the run of 5.438 tasksets. In the first step, the engine composes tasksets, incurring a bound on their maximum cardinality determined by the 4MB limit of the target processor’s limit. Subsequently, it tests their feasibility using Response Time Analysis (RTA) for FPS [10] and the equivalent criterion for EDF. Since those tests are exact and accurate, we were able to have fine-grained control over the worst-case utilization scenarios that we wanted to generate.

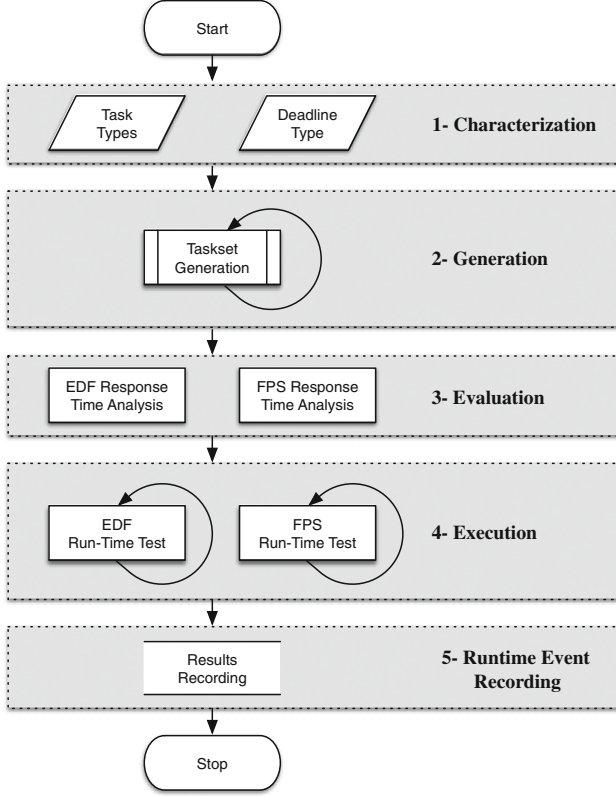
For the FPS case, our engine uses a simpler variant of the fine-grained high-accuracy version of the classic RTA equation presented in [12]:

$$R_i^{n+1} = CS1 + C_i + \sum_{j \in hp(i)}^n \left\lceil \frac{R_j^n}{T_j} \right\rceil \cdot (CS1 + C_j + CS2) \quad (1)$$

With Eq. 1, a taskset scheduled with FPS, is feasible if and only if  $R_i \leq D_i \forall i$ . Let us briefly recall the meaning of the cost factors that appear in it.

1.  $CS1$  is the context switch experienced by task  $\tau_i$  when it preempts another task on access to the CPU.





**Fig. 4.** Automated engine generation and evaluation steps.

2.  $C_i$  is the highest computation time demand of task  $\tau_i$ ;
3.  $T_j$  is the period of task  $\tau_j$ ;
4.  $CS2$  is the dual of  $CS1$  and accounts for the cost of cleaning the context up when task  $\tau_j$  releases the CPU;

The feasibility of the same tasksets was then evaluated for EDF with the quantitative method proposed in [11] for constrained-deadline systems, using Eq. 2:

$$h(t) = \sum_{j=1}^i \max \left( 0, \left\lfloor \frac{t - D_j}{T_j} \right\rfloor + 1 \right) \cdot (CS1 + C_j + CS2) \quad (2)$$

which stipulates that a taskset is schedulable under EDF if and only if the worst-case CPU load  $L$  does not exceed 1:

$$L = \max_{\forall t} \left( \frac{h(t)}{t} \right) \leq 1 \quad (3)$$

The subsequent step was the core part of our automation engine: in it, we compile, build and execute the benchmark tasks for both runtimes, recording each occurrence of 3 events of interest:

- **Regular Completion:** when a task’s job completes execution within the assigned deadline;
- **Deadline Missed:** when a task’s job completes after its assigned deadline;
- **Preemption:** when the current task is preempted by a newly release job.

We run our benchmarks on an evaluation version of the *TSIM/LEON SPARC simulator*, which limits the longest span of execution to  $2^{32} = 4.294.967.296$  clock cycles. To overcome that limitation we used an approach derived from [8], which suggests how to generate bounded hyperperiods using a composition of bases and exponents. As we needed to contain the highest hyperperiod to  $2^{32}$ , we set an artificial upper-bound to  $2^7 \cdot 3^6 \cdot 5^3 \cdot 7^3 = 4.000.752.000 < 2^{32}$ . Adding this constraint to the taskset generation algorithm, we ensured that all their hyperperiods would fully execute with our simulator. Moreover, using prime numbers as the basis of the calculation, we yielded a sufficient quantity of not harmonic<sup>3</sup> tasksets.

## 4 Evaluation Results

Citation [7] arguably is the most famous discussion of a structured quantitative performance comparison of EDF vs FPS. Acknowledging it, we decided to follow its same overall logic, setting the following evaluation criteria:

1. **Highest Schedulable Utilization:** Which tasksets achieved the highest schedulable utilization in each runtime variant? How did the corresponding values relate to the theoretical ratios discussed in [11]?
2. **runtime Overhead:** Do the less preemptions and context switches that EDF incurs justify the higher costs of its scheduling operations?
3. **Resilience to Overload Situations:** What happens to EDF and FPS under overload conditions, when the CPU utilization exceeds 100%?
4. **Locking Policy:** How does DFP perform compared to IPCP?

Question 1 reflects the intent to seek empirical evidence in relation to the theoretical results presented in [11]. That work in fact shows that the performance of EDF is 1,44269 better than FPS for Implicit Deadlines, 1,76322 for Constrained Deadlines and 2 for Arbitrary Deadlines.

Table 1 presents the results we obtained in response to criterion 1. Interestingly, they are much less slanted in favour to EDF than they were in [11].

The highest utilization (obtained without incurring deadline misses) reached by our EDF runtime was only 3,72% (first two lines of Table 1) better than achieved by FPS for the same tasksets. As expected in point of theory, EDF prevailed because it generated a lower number of preemptions, which yielded room

---

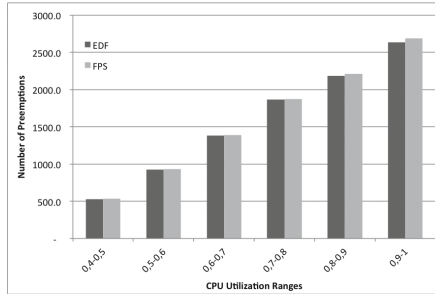
<sup>3</sup> A task system has harmonic rates if and only if the periods of its tasks are pairwise divisible (for each  $i, j$  one has  $p_i | p_j$  or  $p_j | p_i$ ) with no remainder.

**Table 1.** Highest schedulable utilization achieved by EDF over FPS (line 1 & 2) and vice-versa (line 3 & 4). **RC** stands for Regular Completions; **DM** for Deadline Misses; **PR** for Preemptions

Taskset type	Task types	Delta schedulable utilization	Max CPU load	EDF			FPS		
				RC	DM	PR	RC	DM	PR
Constrained	Short & mid	<b>2,89%</b>	105,50%	30.714	0	3.637	29.850	415	6.202
Implicit	Mid only	<b>3,72%</b>	102,63%	18.691	0	837	18.021	673	2.040
Constrained	All	<b>0,05%</b>	104,06%	24.398	0	5.131	24.409	0	5.211
Implicit	All	<b>5,22%</b>	100,85%	24.935	953	6.309	26.236	0	5.715

for higher schedulable utilization, but surprisingly less markedly for constrained-deadline tasksets, and more visibly – but still marginally – for implicit-deadline tasksets. In our experiments, tasks’ execution times are short enough to be sensitive to the overhead of runtime procedures, making the different complexity of the two runtimes more manifest.

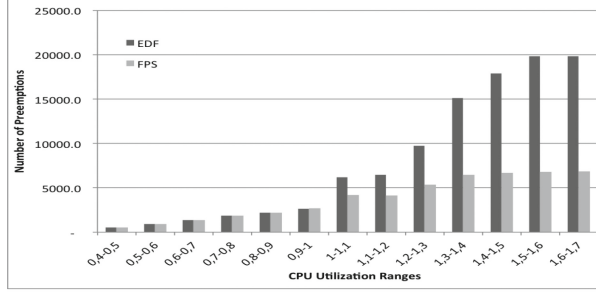
The good relative performance of FPS presented in the bottom half of Table 1 can be explained in two ways, depending on the type of experiment that yielded it: when the number of preemptions spared by EDF with respect to FPS is small, then the marginal gain in schedulable CPU utilization also becomes small; conversely, when a taskset overloads the CPU, EDF may “blow up” and cause an inordinate number of vacuous preemptions, many of which lead to deadline miss.



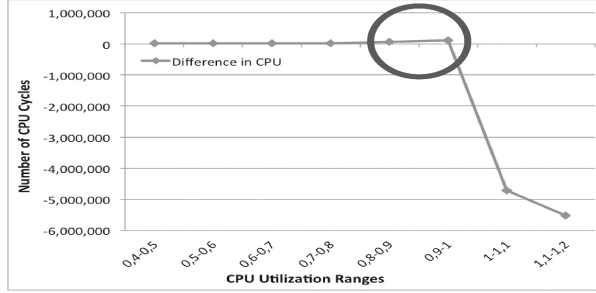
**Fig. 5.** Average number of preemptions in both runtimes with utilization  $\leq 100\%$ .

The different preemption behaviour of the two runtimes leads to Question 2. As the cost of individual context switch operations is nearly the same (ca. 2,389 CPU cycles) in both runtimes, answering that question required considering the cumulative cost incurred in the respective executions. Figure 5 shows the average benefit gained by EDF from lesser recourse to preemption for schedulable utilizations under 100%. The total quantity of CPU time that the application

tasks could earn from that benefit in a hyperperiod scarcely exceeded 120000 CPU cycles, very little indeed, considering that the smallest (short) tasks in our experiments run for 750000 cycles. This quantity however must be considered with care, since it is an average value, which balances out best- and worst-case situations, where the two runtimes may perform rather differently.



(a) Total preemptions in the full utilization spectrum.

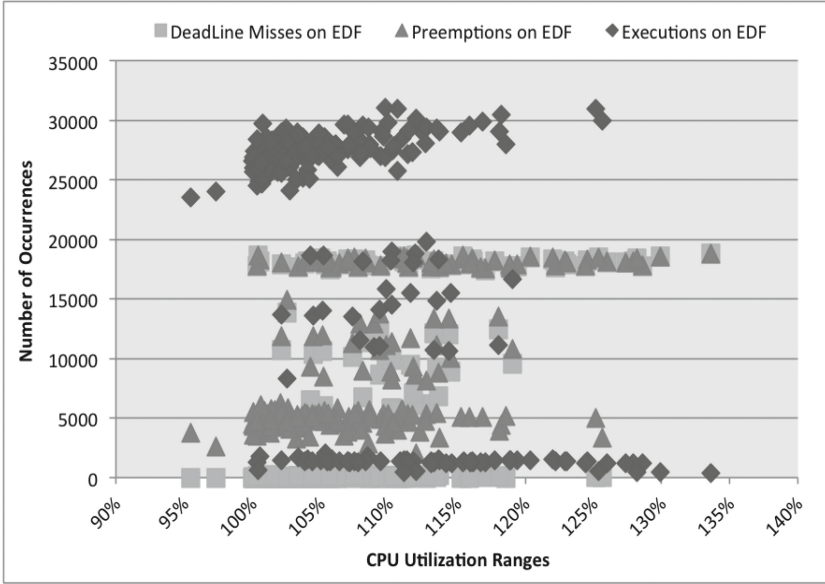


(b) Average cumulative differences in clock cycles over a full hyperperiod for CPU loads that increase from 0 to severe overload situations.

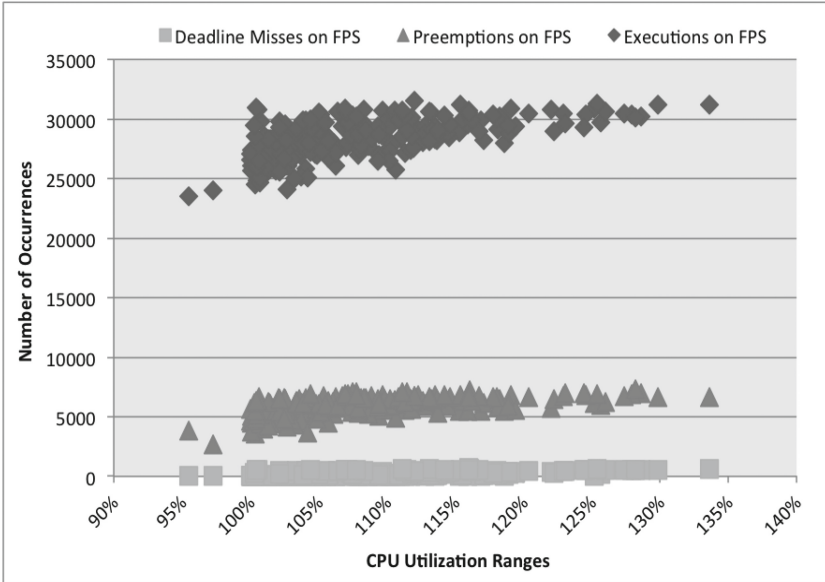
**Fig. 6.** Effect of differences in average number of preemptions for varying CPU loads.

When the CPU utilization exceeds 100%, the prevalence of EDF over FPS inverts radically: EDF incurs a much greater number of preemptions (cf. Fig. 6a), for a massive loss of application performance, which shadows to the modest gain achieved near 100%. Figure 6b contrasts the gain to the loss.

Question 3 delves deeper into the issue of what happens under overload conditions. Figure 7 plots the graph of dispersion for EDF and FPS, which helps highlight the greater resilience of FPS. When FPS operates in overload conditions in fact, the number of completed executions, deadline misses and preemptions are linear to one another (cf. Fig. 7b). This happens because only tasks with lower priorities (the “long” ones) are delayed indefinitely, without this affecting those with higher priority. EDF, instead, has a radically different behavior: beyond 100% utilization, its performance immediately starts to deteriorate and



(a) Overload conditions under EDF.



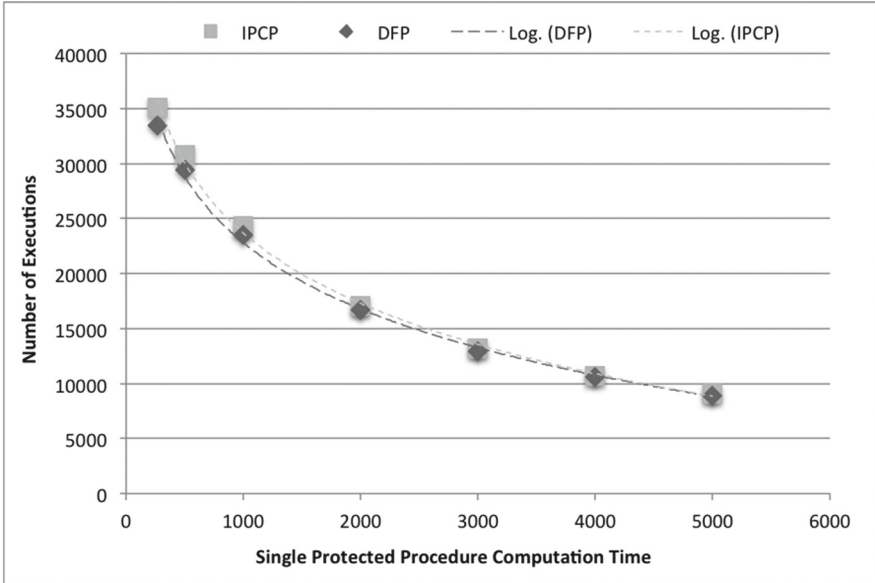
(b) Same taskset, same overload conditions but different behavior under FPS.

**Fig. 7.** Overload conditions under EDF and FPS.

two kinds of extreme behavior emerge, with a blurred zone in between them as shown in Fig. 7a. When the overload condition is transient (or rather the excess load is modest), EDF still allows a high number of completed executions (higher darkest diamonds in Fig. 7a) with a low number of misses and a relative small number of preemptions (respectively lower darker triangles and squares in the same graph). When the overload conditions are more marked, they generate a sort of domino effect, which causes the number of preemptions and misses to increase dramatically (higher darker triangles and squares), while the number of completed executions drops equally fast (lower darkest diamonds in Fig. 7a).

We should clarify that the utilization factor computed in accord with [11] represents the highest CPU load that the system incurs during a hyperperiod. This value may be much higher than the average load that we were able to measure at run time. Hence, a very high max CPU load does not necessarily mean an unsustainable situation, but can rather be seen as a transient overload that both runtime variants can possibly cope with.

Question 4 reports a quantitative comparison between the two locking policies we implemented in our runtime variants. From our implementation, we learned that they are extremely close to one another in terms of runtime overhead.



**Fig. 8.** Number of regular execution completions incurred in the experiment.

To tell the performance differences between them, we made a number of runs of tasksets comprised of only one task whose sole activity was to seize a PO and execute a protected procedure in it, varying its computation time from small to large. With this setting, we counted how many task executions each

runtime was able to complete with the longest run of the simulator, without incurring deadline misses. Comparing the results, we saw that the applications running with IPCP completed more executions than with DFP. This difference decreases with a logarithmic converging progression as the computation time of the protected procedure increases (cf. Fig. 8). This result shows that DFP incurs more cumulative overhead than IPCP, evidently due to the need to read the clock in checking absolute deadlines.

## 5 Conclusions

In the past, the real-time systems community studied in earnest the relative benefits of EDF and FPS, with an eye to their use in industrial systems. Most of those studies were theoretical in nature, that is, they concentrated on the respective feasibility equations, showing that, in point of theory, EDF warranted much better performance.

In this work, we built an experimental framework where an EDF variant of an Ada Ravenscar runtime was developed and exposed to a quantitative comparison with the FPS original.

An interesting trait of our EDF runtime variant is that it changes the behavior of the smallest possible set of runtime primitives needed to support deadline-driven scheduling, without changing the API provided to the application. In that manner, we only had to build one (large) single set of benchmark applications and transparently bind them to the desired runtime.

Overall, our tests confirmed the theoretical conclusions reached by earlier works. Yet, we showed that the actual gain of EDF over FPS is far lower than anticipated even for CPU loads very close to 100%, where EDF was due to reap the best of its benefit. We also studied the behavior of EDF vs FPS scheduling under overload situations, where we experimentally observed the fragility of the former and contrasted to the resilience of the latter.

We hope to have provided milestone technology for the further study of this topic. Thanks to the high performance, modularity, and predictability of our Ravenscar runtimes, there is now room for further deeper investigations of how EDF (with DFP, which we also added to our implementation) behaves in comparison to FPS (with IPCP).

## References

1. Burns, A.: A deadline-floor inheritance protocol for EDF scheduled real-time systems with resource sharing. Technical report YCS- 2012-476, Department of Computer Science, University of York, UK (2012)
2. Burns, A.: An EDF runtime profile based on Ravenscar. *Ada Lett* **XXXII**(1), 24–31 (2013)
3. Burns, A.: The Ravenscar profile ACM. *Ada Lett*. **XIX**(4), 49–52 (1999)
4. Burns, A., Dobbing, B., Romanski, G.: The Ravenscar tasking profile for high integrity real-time programs. In: Asplund, L. (ed.) *Ada-Europe 1998*. LNCS, vol. 1411, pp. 263–275. Springer, Heidelberg (1998). doi:[10.1007/BFb0055011](https://doi.org/10.1007/BFb0055011)

5. Burns, A., Wellings, A.: The deadline floor protocol and Ada. *ACM SIGAda Ada Lett.* **36**(1), 29–34 (2016)
6. Liu, L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM* **20**(1), 46–61 (1973)
7. Buttazzo, G.: Rate monotonic vs EDF: judgment day. *Real Time Syst.* **29**, 5–26 (2005)
8. Goossens, J., Macq, C.: Limitation of the hyperperiod in real-time periodic task set generation. In: *Proceedings of the RTS Embedded System (RTS 2001)*, pp. 133–147 (2001)
9. Sha, L., Rajkumar, R., Lehoczky, J.: Priority inheritance protocols: an approach to real-time synchronisation. *IEEE Trans. Comput.* **39**, 1175–1185 (1990)
10. Audsley, N., Burns, A., Richardson, M., Tindell, K., Wellings, A.J.: Applying new scheduling theory to static priority pre-emptive scheduling. *Softw. Eng. J.* **8**(5), 284–292 (1993). doi:[10.1049/sej.1993.0034](https://doi.org/10.1049/sej.1993.0034)
11. Davis, R., Baruah, S., Rothvoss, T., Burns, A.: Quantifying the sub-optimality of uniprocessor fixed priority pre-emptive scheduling for sporadic tasksets with arbitrary deadlines. In: *RTNS 2009, Paris, ECE, 26–27 October 2009*
12. Vardanega, T., Zamorano, J., De La Puente, A.J.: On the dynamic semantics and the timing behavior of Ravenscar kernels. *Real Time Syst.* **29**, 59–89 (2005)



Reliable Software Technologies – Ada-Europe 2017  
22nd Ada-Europe International Conference on Reliable  
Software Technologies, Vienna, Austria, June 12-16,  
2017, Proceedings  
Blieberger, J.; Bader, M. (Eds.)  
2017, XIV, 251 p. 61 illus., Softcover  
ISBN: 978-3-319-60587-6