CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

The Carousel-EDF Scheduling Algorithm for Multiprocessor Systems

Paulo Baltarejo Sousa
Pedro Souto
Eduardo Tovar
Konstantinos Bletsas

CISTER-TR-130401
Version:
Date: 04-04-2013

The Carousel-EDF Scheduling Algorithm for Multiprocessor Systems

Paulo Baltarejo Sousa, Pedro Souto, Eduardo Tovar, Konstantinos Bletsas

CISTER Research Unit

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: pbsousa@dei.isep.ipp.pt, , emt@dei.isep.ipp.pt, ksbs@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

We present Carousel-EDF, a new hierarchical scheduling algorithm for a system of identical processors, and its overhead-aware schedulability analysis based on demand bound functions. Carousel-EDF is an offshoot of NPS-F and preserves its utilization bounds, which are the highest among algorithms not based on a single dispatching queue and that have few preemptions. Furthermore, with respect to NPS-F, Carousel-EDF reduces by up to 50% the number of contextswitches and of preemptions caused by the high-level scheduler itself. The schedulability analysis we present in this paper is grounded on a prototype implementation of Carousel-EDF that uses a new implementation technique for the release of periodic tasks. This technique reduces the pessimism of theschedulability analysis presented and can be applied, with similar benefits, to other scheduling algorithms such as NPS-F.

The Carousel-EDF Scheduling Algorithm for Multiprocessor Systems

Paulo Baltarejo Sousa*†, Pedro Souto*‡, Eduardo Tovar*†, and Konstantinos Bletsas*†
*CISTER/INESC-TEC Research Center
† ISEP-Polytechnic Institute of Porto, Portugal
‡ University of Porto, FEUP-Faculty of Engineering, Portugal
Email: †{pbs, emt, ksbs}@isep.ipp.pt, ‡pfs@fe.up.pt

***Abstract*—We present Carousel-EDF, a new hierarchical scheduling algorithm for a system of identical processors, and its overhead-aware schedulability analysis based on demand bound functions. Carousel-EDF is an offshoot of NPS-F and preserves its utilization bounds, which are the highest among algorithms not based on a single dispatching queue and that have few preemptions. Furthermore, with respect to NPS-F, Carousel-EDF reduces by up to 50% the number of context switches and of preemptions caused by the high-level scheduler itself. The schedulability analysis we present in this paper is grounded on a prototype implementation of Carousel-EDF that uses a new implementation technique for the release of periodic tasks. This technique reduces the pessimism of the schedulability analysis presented and can be applied, with similar benefits, to other scheduling algorithms such as NPS-F.**

I. Introduction

Given the increasing use of multiple processing units in computing systems, the problem of scheduling tasks with real-time constraints on such systems is particularly relevant. In this paper, we present Carousel-EDF, a novel algorithm based on EDF that addresses this problem for systems composed of identical processors.

Multiprocessor systems introduce a key issue that does not arise in the context of uniprocessor scheduling: the possibility of task migration. According to this ability, multiprocessor scheduling algorithms are categorized as: *global*, *partitioned*, and *semi-partitioned*.

Partitioned scheduling algorithms [1] partition the set of tasks in the system such that all tasks belonging to a set of the partition are executed always on the same processor. Thus, after partitioning, the scheduling on a multiprocessor system reduces to the scheduling problem on a uniprocessor. The main issue of this class of algorithms is its low utilization. This is because an idle processor cannot be used to run ready tasks that have been allocated a different processor. It has been proven that for some "pathological cases" their utilization bound is 50%.

Global scheduling algorithms try to overcome this limitation by allowing tasks to migrate from one processor to another. At any time instant, the m highest-priority runnable tasks in the system can be selected for execution on the m processors. Some scheduling algorithms [2], [3] of this class present a utilization bound of 100%, but at a cost of many preemptions and migrations. Although recent work has made some progress towards reducing the number of preemptions and migrations [4] as well as scheduling overheads [5], the implementation of these algorithms requires the use of a global shared queue, which raises scalability issues because of the need to prevent race conditions in the access to that queue, or as stated by Brandenburg [6] "*it would be unrealistic to expect global algorithms to scale to tens or hundreds of processors*"

Semi-partitioned, or task-splitting, scheduling algorithms [7], [8], [9], [10], [11] try to address these issues by limiting task migration. Typically, under task-splitting scheduling algorithms, most tasks execute only on one processor as in partitioned algorithms, whereas a few *migratory* tasks may execute on several processors, as in global scheduling algorithms. This approach produces a better balance of the workload among processors than partitioning (and consequently presents a higher utilization bound). Importantly, the approach also reduces the contention on shared queues and the number of migrations (by reducing the number of migratory tasks). For a survey on real-time scheduling algorithms and related issues, please check [12].

Carousel-EDF does not easily fit in this taxonomy. It is a global algorithm in that it allows the migration of any task, yet it can be implemented with contention-free ready queues, like partitioned algorithms. Furthermore, it partitions the task set into subsets such that tasks from one subset do not directly contend for processing resources with tasks from other subsets. In many respects, Carousel-EDF is closer to Notional Processor Scheduling - Fractional capacity (NPS-F) [10], a semi-partitioned algorithm from which it evolved. The similarities between the two ensure that Carousel-EDF preserves NPS-F utilization bounds, which range from 75% to 100%, by trading off with preemptions. Furthermore, Carousel-EDF reduces the maximum number of preemptions, and therefore context switches, introduced by the scheduler itself by up to half of those induced by NPS-F.

The contribution of this paper is fivefold. First, we present Carousel-EDF, a new scheduling algorithm that cuts by up to half the number of preemptions due to NPS-F, while preserving its utilization bounds. Second, we develop an overhead-aware schedulability analysis that is grounded on

a prototype implementation of Carousel-EDF on Linux. Third, we present the algorithms for setting up Carousel-EDF on a real system that are based on the overhead-aware schedulability analysis. Fourth, we show that Carousel-EDF can be implemented efficiently. This implementation uses a technique for the release of periodic tasks that eliminates the interference on a server caused by the release of other servers' tasks. This implementation technique is essential to reduce the pessimism of the schedulability analysis. Fifth, we present some preliminary results of an experimental evaluation using the overhead-aware schedulability analysis. These results show that, for values of the overheads estimated from a prototype implementation, Carousel-EDF leads to lower overheads than NPS-F for some task sets.

The remainder of this paper is structured as follows. In the next section, we present the system model. Section III describes the Carousel-EDF scheduling algorithm. After that, in Section IV, we present some implementation details of Carousel-EDF and the overheads it may incur in a real-system. Sections V and VI describe the overhead-aware schedulability analysis. In Section VII, we present the algorithms used to setup the system. Section VIII presents preliminary results of an experimental evaluation of Carousel-EDF. Finally, in Section IX, we conclude.

II. System Model

We consider preemptive real-time systems composed by m identical physical processors, each with a unique identifier in the range $P_1 \dots P_m$.

The system also includes a task set τ composed by n tasks, each with a unique identifier in the range $\tau_1 \dots \tau_n$. Each task τ_i generates a potentially infinite number of *jobs* and is characterized by three parameters: C_i, the worst case execution time of any of its jobs, T_i, the minimum inter-arrival time between consecutive jobs, and D_i, the relative deadline of all its jobs. We assume that $0 \leq C_i \leq D_i$ and that $\forall i : D_i = T_i$, unless stated otherwise.

The utilization of task τ_i, denoted u_i, is defined as the ratio of C_i to T_i ($u_i = \frac{C_i}{T_i}$). The system utilization (U_s), normalized to m processors, is defined as $U_s = \frac{1}{m} \cdot \sum_{i=1}^{n} u_i$.

The j^{th} job of task τ_i, denoted $\tau_{i,j}$, with $j \geq 1$, becomes *ready* to be executed at arrival time $a_{i,j}$ and completes by *finishing* (or completion) time $f_{i,j}$. By definition of T_i, the time difference between two consecutive job arrivals must be at least equal to T_i, i.e. $a_{i,j+1} - a_{i,j} \geq T_i$. The *absolute deadline*, $d_{i,j}$, of job $\tau_{i,j}$ is given by $d_{i,j} = a_{i,j} + D_i$. A *deadline miss* occurs when $f_{i,j} > d_{i,j}$.

III. The Carousel-EDF Scheduling Algorithm

Carousel-EDF partitions task set into subsets of tasks we call *servers*. Furthermore, it associates each server with a *reserve*, a time interval during which a processor is exclusively allocated to a server. The duration of a reserve may vary from server to server, but the duration of a server's reserve does not change over time.

Carousel-EDF uses a two-level hierarchical run-time algorithm. The top-level scheduler allocates processors to servers. This allocation is done in a round robin fashion; that is, a processor is allocated to the first server for the duration of its reserve, then to the second server and so on until the last server. This sequence is then repeated indefinitely. Each processor is allocated to all the servers in the same order, but the schedule of each processor is staggered from that of the previous one by a *time slot*, S, i.e. the schedules of each processor have a different phase, so that at any time at most one processor is allocated to each server. As a result, each server is allocated a different processor every time slot and its tasks can run during the corresponding reserve, i.e. a server's reserves are periodic with period S.

The low-level scheduler uses the EDF algorithm to schedule tasks in a server during the respective reserves. Because the top-level scheduler ensures that at any time at most one processor is allocated to a server, the low-level scheduler operates on a local basis, i.e. there is no need for coordination among the low-level schedulers of the different processors.

A. Off-line Procedure

In order to ensure the schedulability of a task set Carousel-EDF uses an off-line procedure that comprises three steps: (i) the partitioning of the task set into servers, (ii) the sizing of the periodic reserve of each server and (iii) the phasing of each processor, i.e. determining the server and the size of a processor's first reserve.

Servers have unique identifiers in the range $\tilde{P}_1 \dots \tilde{P}_k$. The set of tasks in server $\tilde{P}_q$ is denoted $\tau[\tilde{P}_q]$. We define the *utilization of server* $\tilde{P}_q$, $U[\tilde{P}_q]$, the sum of the utilization of its tasks, i.e. $U[\tilde{P}_q] = \sum_{\tau_i \in \tau[\tilde{P}_q]} u_i$. Because at any time a server can have at most one processor allocated to it, a server's utilization must not exceed 100%. Thus, the assignment of tasks to servers can be done by applying any bin-packing heuristic.

Fig. 1 illustrates this first step. Fig. 1(a) shows the task set (τ), which is composed by 7 tasks. Each task is represented by a rectangle, whose height is proportional to its utilization. Fig. 1(b) shows the assignment of these tasks to 5 servers, by applying the first-fit bin-packing heuristic.

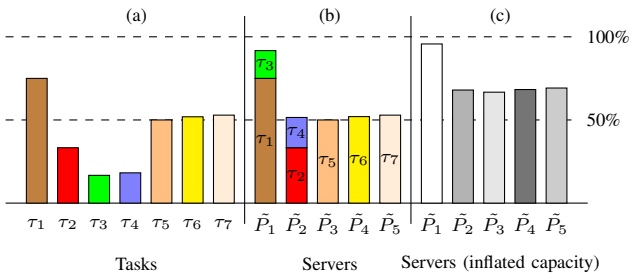

Figure 1: Task to server assignment.

Once the servers have been defined, the procedure computes the size of each server's reserve. Given that the period of all servers is the time slot, S, the size of each reserve ($Res^{len}[\tilde{P}_q]$) is given by $S \cdot U^{infl}[\tilde{P}_q]$, where $U^{infl}[\tilde{P}_q]$ denotes the *inflated utilization* of the server. We use the inflated server utilization rather than the server utilization because the tasks of a server can run only when a processor is allocated to that server. Thus, a server may have ready tasks but cannot execute them. By inflating the utilization of a server [10], we ensure that the tasks deadlines are met in spite of the unavailability of processing resources caused by the use of reserves. Fig. 1(c) shows the outcome of the this step.

The last step in the off-line procedure is determining the phase of each processor, i.e. the server and the size of the first reserve of each processor. Because the (high-level) schedule of each processor is staggered by S from that of the previous processor, the phase of all processors can be determined from the schedule of processor P_1, whose first reserve is that of server $\tilde{P}_1$. As shown in Fig. 2(a), the schedule of processor P_i begins at time $i \cdot S$ of P_1's schedule. Alg. 5 in Sec. VII presents the algorithm for computing the first reserve.

Fig. 2(a) illustrates the schedule for the example shown in Fig. 1. In this example, we have 4 processors and 5 servers. At time 0, processor P_1 is allocated to server $\tilde{P}_1$. When that server's reserve expires, processor P_1 is allocated to server $\tilde{P}_2$, and so on until it is allocated to the last server $\tilde{P}_5$. Processor P_1 is allocated again to server $\tilde{P}_1$ at time $4S$, i.e. at the next time instant multiple of S. This ensures not only that every reserve has period S, but also that at any time there is at most one processor allocated to any server. Therefore, when the sum of the duration of all reserves is not a multiple of the time slot, there is a time gap between the end of the last reserve and the beginning of the first that we name *empty reserve*.

B. Preemptions and Utilization Bound

As already mentioned, NPS-F is at the genesis of Carousel-EDF. In particular, the concepts of server, reserve and time slot are also used in that algorithm with exactly the same names, except for server, which is known as *notional processor*, and gave the algorithm its name. Furthermore, all the steps but the last step of the off-line procedure are those used in NPS-F.

Thus, the main difference between the two algorithms is at the high-level scheduler. In NPS-F, servers are statically assigned to processors one after the other so as to ensure the full-utilization of each used processor. In other words, the last step of the off-line procedure described above is replaced by a step that iterates over the set of servers and assigns each server a reserve on the next processor that is not yet fully utilized. If the server inflated utilization is higher than the available capacity of the current processor, the processor demand not satisfied by the current processor is assigned to a reserve on the next processor. Thus such servers become *split servers*, because their reserve is split into two.

Fig. 2(b) shows the schedule that is generated by NPS-F for our running example. Initially all processors have utilization 0, no server has been assigned, and the current processor is P_1. We iterate over the set of servers in the order of their ids (although any order is possible). Thus, server $\tilde{P}_1$ is assigned to processor P_1. Next, we consider server $\tilde{P}_2$. Its inflated utilization is larger than that available on processor P_1, therefore the reserve is split so that the first reserve makes the utilization of processor P_1 equal to 1. $\tilde{P}_2$'s second reserve, for the remaining of $\tilde{P}_2$'s inflated utilization, is assigned to the next processor, P_2, which now becomes the current processor. The procedure is repeated until all servers are assigned reserves on the different processors.

In [10] it is proven that, for periodic reserves with period $S \leq \frac{\min_{\tau_i \in \tau}(T_i)}{\delta}$, where δ is a positive integer, the mapping of tasks to servers is done using the first-fit bin-packing heuristic, and the utilization $U[\tilde{P}_q]$ of each server $\tilde{P}_q$ is inflated to $U^{infl}[\tilde{P}_q] = \frac{(\delta+1) \cdot U[\tilde{P}_q]}{U[\tilde{P}_q]+\delta}$, NPS-F's normalized utilization bound is given by $U[\tilde{P}_q] = \frac{2 \cdot \delta + 1}{2 \cdot \delta + 2}$.

Although NPS-F has proven better upper bounds on the number of preemptions for the same guaranteed utilization bound than any other algorithm, the splitting of servers is responsible for up to half of these preemptions: every server may lead to a preemption, and so does every splitting, and all but one server may be split. The key observation that led to Carousel-EDF was that, if a server is not split at time slot boundaries, we can eliminate up to half of those preemptions and still satisfy the server's processing demand. It is clear from Fig. 2, that in the first time slot, the two algorithms lead to the same high-level schedules for all processors. The schedules of each processor in the second time slot are not the same, but, in that time slot, P_i's schedule in Carousel-EDF is equal to P_{i+1}'s schedule in NPS-F, and so on. Thus, at any time instant t, the *active* servers, i.e. the servers that have a processor allocated, are the same for both algorithms, although the processor that is allocated to each server may be different.

Based on these informal arguments, we claim that if tasks are assigned to servers using first-fit bin packing, server inflation and the time slot are determined as described above, Carousel-EDF preserves NPS-F utilization bounds, while reducing its upper bounds on the number of preemptions by up to half. Formal proofs can be found in Appendices A and B.

The improvement on the number of preemptions comes at the potential cost of additional migrations, i.e. preemptions where the preempted jobs are resumed at a different processor. In this respect, Carousel-EDF behaves essentially as a global scheduler, therefore the penalty is hard to quantify. Above all, it depends on the task set and on the processor

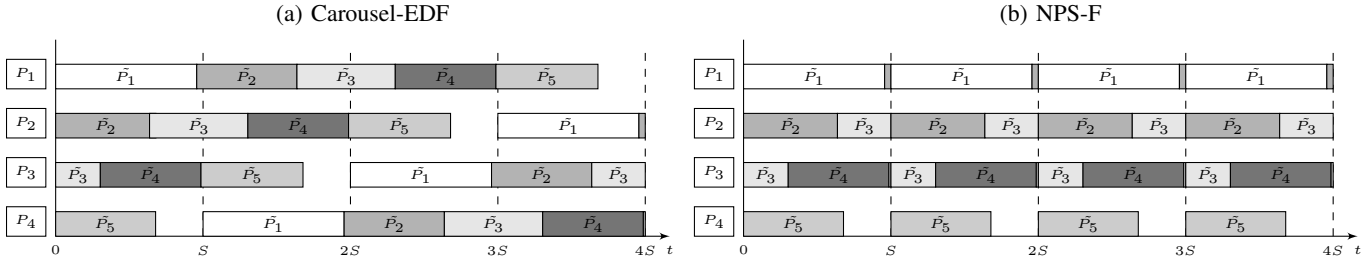

Figure 2: High level schedule.

characteristics. If these are such that the cache footprint of a preempted task is wiped out from the processor caches, the cost of a preemption will be as high as that of a migration [13], and therefore there is no penalty. In hard real-time applications, absent any guarantee that such worst case scenarios do not occur, an overhead-aware schedulability analysis has to assume that the cost of a preemption is as high as that of a migration.

IV. Scheduler implementation and overheads

In this section, we provide a few details regarding the implementation[1] of Carousel-EDF's top- and low-level schedulers, and identify the overheads it induces. This is important to better understand the overhead-aware schedulability analysis of Carousel-EDF developed in the following sections.

A. Scheduler Implementation

The basis of the top-level scheduler implementation is a circular queue of servers. For each processor, we use i) a pointer to the server it is currently allocated; and ii) a timer that measures the time left until the end of the current reserve. Whenever one of these timers expires, the top-level scheduler updates the corresponding processor pointer to the next server, starts a new timer for the new reserve, and triggers the invocation of the low-level scheduler so that the processor switches to a task belonging to the newly active server.

As usual, tasks that are ready to run are kept in a *ready queue*, ordered by their deadlines. However, rather than using a system wide ready queue, our implementation uses a per-server ready queue. Because Carousel-EDF guarantees that at any time there is at most one processor allocated to any server, this eliminates contention in the access to the ready queue.

In addition to a ready queue, our implementation uses also a *release queue* per server. This is similar to a ready queue, but, rather than keeping the tasks that are ready to run, it keeps the periodic tasks that are waiting to be released, ordered by their release times. This queue plays a crucial role in the feasibility of Carousel-EDF, as it allows us to reduce the pessimism of the schedulability analysis. Indeed, by using this queue and by the appropriate use of timers, we prevent the release of jobs belonging to a server from interfering with other servers.

The key observation is that it is of no use to release a task outside the reserves of the server to which it belongs. Therefore, our implementation uses timers for the release of periodic tasks only when the release time falls in the corresponding reserve. Otherwise, the release of the task is postponed until the beginning of the corresponding reserve. Thus, at the beginning of a reserve, and before invoking the low-level scheduler, the top-level scheduler traverses the release queue and moves all the tasks whose release time has passed to the ready queue of the corresponding server. Furthermore, it starts a timer for each task whose release time will occur during the new reserve.

The low-level scheduler implements EDF and is not aware that Carousel-EDF uses a per-server run queue. Furthermore, sporadic tasks continue to be released by interrupts (see Appendix C).

B. Overheads

In order to perform an overhead-aware schedulability analysis, we need to take into account all the delays a task may suffer from its arrival time until its completion. In this subsection, we consider only the delays other than the execution of tasks.

Grounded on our implementation of Carousel-EDF and with an eye towards an experimental evaluation of the algorithm, we have grouped these delays in five classes: task-release overheads; reserve switching overheads; context switching overheads; cache-related preemption and migration delays (CPMD); and interrupt overheads.

The task release mechanism introduces two overheads. The first overhead, the *Release Jitter*, denotes the time difference between the time instant when a job should be released, $a_{i,j}$, and the time when the job release actually starts. This overhead reflects the inability to precisely measure time intervals by the operating system. The second overhead, *Release Overhead*, is the time used by the processor to release a job, including moving the job to the ready queue.

The switch from one reserve to the next incurs also two overheads, the *Reserve Jitter* and the *Reserve Overhead*. The reserve jitter is analogous to the release jitter: it is the

[1] Available for download at http://webpages.cister.isep.ipp.pt/~pbsousa/retas/3_2_11-rt20-retas-rb.html

difference between the time instant when a reserve should begin and the time when the reserve actually starts. Likewise the reserve overhead is analogous to the release overhead: it represents the time during which the processor performs the actions required to switch from a reserve to another, e.g. switching the ready queue of a processor from that of the terminating reserve to that of the new reserve.

A context-switch occurs whenever the dispatcher allocates a processor to a task different from the one that is currently executing on that processor. Typically, a context-switch comprises the following operations: (i) saving the context of the current executing task; (ii) selecting the next task to be executed; and (iii) (re)storing the context of the new task. The *Context-switch Overhead* is the processor time required to perform these operations.

The *Cache-related Preemption/Migration Delay* (CPMD) overhead quantifies the cost of preemptions and migrations in the worst-case execution time of a task. The worst-case execution time of a task is typically derived assuming that its jobs execute continuously in time, without preemptions. With Carousel-EDF, a task may be preempted by another that has a deadline earlier than its own. As a result the cache lines of the preempted task may be evicted from the cache. When the preempted task later resumes execution, these cache lines may have to be brought back, either from higher-level caches or from memory, incurring additional execution delays.

Interrupts are an event-notification mechanism and usually trigger the execution of interrupt handlers (IH). We assume that these handlers are implemented as operating system tasks with higher priority than "normal" tasks. *Interrupt Overheads* represent the processor time used by IHs.

V. Task Set Partitioning

The first step in the off-line procedure is the partitioning of the task set into servers. Because at any time a server is allocated at most one processor, the processing demand of all tasks in a server, including overheads, must be lower than the processing that can be supplied by a processor. Thus, before presenting the partitioning algorithm, we present an overhead-aware schedulability analysis based on processor demand.

A. Server Schedulability Analysis

Assuming that each server executes on its own processor, the demand-based test for a server $\tilde{P}_q$ is given by:

$$\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t) \leq t, \forall t > 0 \tag{1}$$

where $\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t)$ is the demand-bound function [14], which provides a upper bound (over every possible time interval $[t_0,\ t_0 + t)$ of length t) on the aggregate execution requirement of all jobs of $\tilde{P}_q$ ($\tau[\tilde{P}_q]$) released at time t_0 or later and whose absolute deadlines lie at or before time $t_0 + t$.

Assuming sporadic task sets with arbitrary deadlines and ignoring overheads, $\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t)$ can be computed as:

$$\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t) = \mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \cdot C_i \tag{2}$$

Next, we consider each overhead identified in the previous section and incorporate it into the demand-based test.

Task Release. Fig. 3 illustrates graphically the overheads related to the release of job $\tau_{i,j}$.

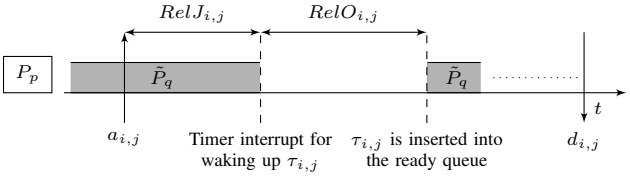

Figure 3: Illustration of the release jitter and release overhead of a job released by a timer.

Let $RelJ$ be the upper bound on the release jitter. As shown in Fig. 3, the release jitter decreases the amount of time available to complete a task, i.e., in the worst case, τ_i has $D_i - RelJ$ time units to complete. Therefore, we modify the $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t)$ to:

$$\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - (D_i - RelJ)}{T_i} \right\rfloor + 1\right) \cdot C_i \tag{3}$$

We model the release overhead as a higher-priority interfering workload, rather than as an increase in the execution demand of a task, because the release overhead contributes "immediately" to the processor demand. Thus, the processing demand for releasing all jobs of $\tau[\tilde{P}_q]$ in a time interval of length t: is:

$$\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q], t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \left\lceil \frac{t + RelJ}{T_i} \right\rceil \cdot RelO \tag{4}$$

where $RelO$ is a upper bound on release overhead.

Incorporating these overheads in $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t)$, we get:

$$\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t) = \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q], t) + \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i + RelJ}{T_i} \right\rfloor + 1\right) \cdot C_i \tag{5}$$

Context Switching. The number of context switches over a time interval of length t is upper bounded by twice the number of job releases during that interval. This is because the lower-level scheduler schedules the tasks in each

server according to the EDF policy and, under EDF, context switches occur either when a job is released or when a job completes, but not every job release will cause a context switch. Let $CtswO$ be an upper bound for the context-switch overhead. We amend the derivation of the $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t)$ to take into account context switching by increasing the execution demand of each job by twice $CtswO$:

$$\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t) = \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q], t) + \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i + RelJ}{T_i} \right\rfloor + 1\right) \cdot (C_i + 2 \cdot CtswO) \quad (6)$$

CPMD. Determining the CPMD is a challenging research problem. For the state-of-the-art, see [15]. In our analysis, we use a rather crude but safe model to account for CPMD: we assume that every preemption leads to the worst case delay for the preempted task. In the worst case, every task release leads to a preemption. Therefore, the cumulative cost of the CPMD over one interval of length t is computed multiplying the number of task preemptions for server $\tilde{P}_q$ by the upper bound of CPMD, denoted by $CpmdO$:

$$\mathrm{dbf}^{\mathrm{part}}_{\mathrm{CpmdO}}(\tilde{P}_q, t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \left\lceil \frac{t + RelJ}{T_i} \right\rceil \cdot CpmdO \quad (7)$$

This cost represents an increase in the server's processing demand, therefore we amend $\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t)$ to:

$$\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t) = \mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{CpmdO}}(\tilde{P}_q, t) \quad (8)$$

Interrupts. We model each interrupt Int_i as a sporadic task with worst-case execution time equal to C_i^{Int}, minimal inter-arrival time equal to T_i^{Int} and maximum jitter J_i^{Int}, and also with zero laxity ($C_i^{Int} = D_i^{Int}$). Furthermore, we assume that each interrupt is bound to a specific processor so as to balance the interrupt overhead on all processors. Thus the interrupt processing demand for the n_p^{Int} interrupts on processor P_p is given by:

$$\mathrm{dbf}^{\mathrm{part}}_{\mathrm{IntO}}(P_p, t) = \sum_{i=1}^{n_p^{Int}} \max\left(0, \left\lfloor \frac{t - (D_i^{Int} - J_i^{Int})}{T_i^{Int}} \right\rfloor + 1\right) \cdot C_i^{Int} \quad (9)$$

Because on Carousel-EDF servers execute on all processors, we safely consider the maximum demand on all processors:

$$\mathrm{dbf}^{\mathrm{part}}_{\mathrm{IntO}}(\tilde{P}_q, t) = \max\left(\mathrm{dbf}^{\mathrm{part}}_{\mathrm{IntO}}(P_1, t) \ldots \mathrm{dbf}^{\mathrm{part}}_{\mathrm{IntO}}(P_m, t)\right) \quad (10)$$

This interrupt accounting method is *server-centric* and lies between the task-centric and the processor-centric methods described in [16].

Finally, incorporating all overheads, we get:

$$\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t) = \mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{CpmdO}}(\tilde{P}_q, t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{IntO}}(\tilde{P}_q, t) \quad (11)$$

B. Task Set Partitioning Algorithm

The algorithm uses the schedulability test developed in the previous subsection to ensure that the demand of all tasks mapped to a server does not exceed the capacity of a processor. It determines both the number of servers to use and the tasks that are mapped to each server.

Alg. 1 shows the pseudo-code. It iterates over the task set τ and for each task τ_i it checks whether it fits in one of the opened servers. For each opened server $\tilde{P}_q$, it adds provisionally task τ_i to it, and applies the schedulability test (Ineq. 1), by invoking the `dbf_part_check` function. If the test succeeds, then task τ_i is permanently assigned to server $\tilde{P}_q$. If the test fails for all opened servers, it creates a new server and adds task τ_i to it. The task set is considered unschedulable whenever the schedulability test fails for a server with only one task. Although Alg. 1 uses a first-fit heuristic, it could use a different bin packing heuristic.

Algorithm 1: Task set partitioning.

Input: set of n tasks τ_i, with $1 \leq i \leq n$
Output: set of k servers, with $k \geq 0$ ($k = 0$ means failure)

```
k ← 0
for i ← 1 to n do
    scheduled ← 0
    for q ← 1 to k do
        if is_open(q) then
            add_task_to_server(τ_i, P̃_q)
            t ← 2 · lcm_T(P̃_q)
            if dbf_part_check(P̃_q, t) then
                scheduled ← 1
                break
            else
                remove_task_from_server(τ_i, P̃_q)
            end if
        end if
    end for
    if scheduled = 0 then
        k ← k + 1 {add a new server}
        add_task_to_server(τ_i, P̃_k)
        t ← 2 · lcm_T(P̃_q)
        if not dbf_part_check(P̃_k, t) then
            k ← 0
            break {failure}
        end if
    end if
end for
```

VI. Server Inflation

The task set partitioning algorithm presented limits the processing demand of a server to that of a processor. To increase the system utilization, it is advantageous that, when a server does not use the full capacity of a processor, the remaining capacity is used by other servers. However, in that case, a server's ready task may not be able to run, even if it is the task with the earliest deadline. To ensure

that despite this processor unavailability a server tasks are still schedulable, we may have to inflate the server demand. This inflation reduces the processor unavailability to the maximum tolerated so that the server tasks are schedulable.

Before presenting the algorithm used to inflate a server demand, we develop a new schedulability analysis for servers that takes into account the fraction of the time slot that is not available for a server. This analysis is used by a schedulability test invoked by the reserve inflation algorithm.

A. Schedulability Analysis

Carousel-EDF guarantees that each server is allocated a processor for the duration of its reserve every time slot (whose size is S time units). That is, the tasks of each server can execute only within a periodic reserve of length:

$$Res^{len}[\tilde{P}_q] = U^{infl}[\tilde{P}_q] \cdot S \tag{12}$$

where $U^{infl}[\tilde{P}_q]$ represents the inflated processing demand of server $\tilde{P}_q$. Hence, given a time interval of length t, only a fraction of t is supplied for the execution of a server.

Rather than modelling the remaining interval in a time slot as a reduction in the processing capacity available for the server tasks, we model it as a *fake task* with attributes:

$$C^{fake} = D^{fake} = S - Res^{len}[\tilde{P}_q] \tag{13}$$

$$T^{fake} = S \tag{14}$$

This approach has the advantage that the right hand side of the schedulability test (Ineq. 15) continues to be t:

$$\mathrm{dbf}^{\mathrm{res}}(\tilde{P}_q, t) \leq t, \forall t > 0 \tag{15}$$

where $\mathrm{dbf}^{\mathrm{res}}(\tilde{P}_q, t)$ is given by:

$$\mathrm{dbf}^{\mathrm{res}}(\tilde{P}_q, t) = \mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t) + \max\left(0, \left\lfloor \frac{t - D^{fake}}{T^{fake}} \right\rfloor + 1\right) \cdot C^{fake} \tag{16}$$

where $\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t)$ is given by Eq. 11.

B. Reserve Overhead

So far we have ignored the overheads associated with the use of reserves. These overheads comprise not only the costs related to the implementation of reserves, but also other costs that would not occur if a server was allocated its own processor. Fig. 4 illustrates the overheads associated with the switching from one reserve to another.

The similarity with the task release jitter and overhead shown in Fig. 3 is striking, and stems from the fact that both task releases and reserve switching rely on the use of timers. However, in contrast to the release jitter, we model the reserve jitter like the reserve overhead because both reduce the time supplied to the reserve.

Although our implementation handles the release of tasks that occur outside of a reserve upon a reserve switch, the reserve overhead needs not include the time required for the handling of these releases, because it is already accounted for in $\mathrm{dbf}^{\mathrm{part}}_{\mathrm{CpmdO}}(\tilde{P}_q, t)$ (see Eq. 7).

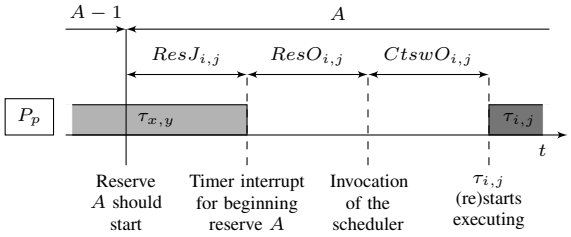

Figure 4: Illustration of the reserve jitter.

In addition to the direct costs of implementing reserves, the use of reserves may add two overheads per reserve switch that would not occur if a server was allocated its own processor. Upon a reserve switch, if the processor is executing a job of the old reserve, this job will be preempted and will be resumed later in a subsequent reserve of its server. This preemption leads to an additional context switch and possibly to a CPMD. Again, we model these overheads by increasing the processing demand by the corresponding values:

$$ResL = ResJ + ResO + CtswO + CpmdO \tag{17}$$

where $ResJ$ and $ResO$ are upper bounds for the reserve jitter and the reserve overhead respectively.

This delay that occurs at the beginning of every reserve can be viewed as extending the fake task, therefore we amend the demand bound function of a reserve in Eq. 16 as follows:

$$\begin{aligned}\mathrm{dbf}^{\mathrm{res}}(\tilde{P}_q, t) = & \\ & \mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t) \\ & + \max\left(0, \left\lfloor \frac{t - (D^{fake} - RelL)}{T^{fake}} \right\rfloor + 1\right) \\ & \cdot (C^{fake} + RelL)\end{aligned} \tag{18}$$

C. Server Inflation Algorithm

We use the schedulability test in Ineq. 15, with $\mathrm{dbf}^{\mathrm{res}}(\tilde{P}_q, t)$ as given by Eq. 18, to determine an interval that is guaranteed to include the inflated utilization. This interval can be arbitrarily small. We start with the interval $[U[\tilde{P}_q], 1.0]$. Then we successively halve this interval using the bisection method. Alg. 2 shows the pseudo-code for the `res_inflate` function, which is used for server inflation. The algorithm converges rather rapidly, and in 10 iterations, it generates an interval that is less than 0.001 wide, that contains the minimum inflated capacity of the server required for the server to be schedulable, according to the schedulability test in Ineq. 15. In Subsection VI-D we provide some details on the implementation of the `dbf_res_check` function.

Algorithm 2: Reserve inflation (`res_inflate`) function.

Inputs: $\tilde{P}_q$ {server to analyse}
ϵ {accuracy of desired estimate}
t {time interval for computing the demand bound function}
Outupt: $U^{infl}[\tilde{P}_q]$ {minimum inflated utilization, with an error smaller than ϵ, that ensures schedulability of $\tilde{P}_q$}

$U_{min} \leftarrow U[\tilde{P}_q]$
$U_{max} \leftarrow 1.0$
while $U_{max} - U_{min} > \epsilon$ **do**
  $U^{infl}[\tilde{P}_q] \leftarrow (U_{min} + U_{max})/2$
  if dbf_res_check$(\tilde{P}_q, t)$ **then**
    $U_{max} = (U_{min} + U_{max})/2$
  else
    $U_{min} = (U_{min} + U_{max})/2$
  end if
  $U^{infl}[\tilde{P}_q] \leftarrow U_{max}$
end while

Alg. 2 may lead to an inflated utilization of 100% or even higher. This means that the overheads incurred by the use of reserves are higher than the unused processor capacity, and therefore the server is allocated a dedicated processor, becoming a *single* server with an inflated utilization of 1.

We can now specify a simple schedulability test for Carousel-EDF as:

$$\sum_{q=1}^{k} U^{infl}[\tilde{P}_q] \leq m \tag{19}$$

where m is the number of processors in the system.

D. Schedulability check functions

Both the task set partitioning algorithm and the reserve inflation algorithm check a server's schedulability by invoking two functions: `dbf_part_check` and `dbf_res_check`, respectively. These tests succeed if for any time interval in the range [0, t), the width of that interval is larger or equal to the computing demand of the server in that interval. In the case of Carousel-EDF we need to check the schedulability for a value of t that is twice the least common multiple (LCM) of the periods of the tasks in each server. For servers with a large number of tasks whose periods are not multiples of each other, the value of t may be very large. To speed up the execution, these functions use Quick Processor-demand Analysis (QPA) [17]. Alg. 3 shows the pseudo-code of these functions. $\mathrm{dbf}^{\mathrm{xxx}}(\tilde{P}_q, t)$ stands for $\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t)$ in the case of the `dbf_part_check` function, and for $\mathrm{dbf}^{\mathrm{res}}(\tilde{P}_q, t)$ in the case of the `dbf_res_check` function.

VII. Carousel Generation

The last step in the off-line procedure is the generation of the carousel, i.e. the high-level schedule. It comprises determining the size of the reserves and their sequence and also the initial phasing of the different processors.

Alg. 4 shows the algorithm that generates the sequence of servers. It has a single loop in which it iterates over the set of servers created by the task set partitioning algorithm (Alg. 1). For each server it inflates its utilization,

Algorithm 3: QPA-based schedulability test functions.

Input: $\tilde{P}_q$ {server to analyse}
Returns: **true** if $\tilde{P}_q$ is schedulable, **false** otherwise

$t \leftarrow 2 \cdot \mathrm{lcm\_T}(\tilde{P}_q)$
$d_{min} \leftarrow \min_{\tau_i \in \tau[\tilde{P}_q]}(D_i)$
while $\mathrm{dbf}^{\mathrm{xxx}}(\tilde{P}_q, t) \leq t$ **and** $\mathrm{dbf}^{\mathrm{xxx}}(\tilde{P}_q, t) > d_{min}$ **do**
  if $\mathrm{dbf}^{\mathrm{xxx}}(\tilde{P}_q, t) < t$ **then**
    $t \leftarrow \mathrm{dbf}^{\mathrm{xxx}}(\tilde{P}_q, t)$
  else
    $t \leftarrow t - 1$
  end if
end while
return $\mathrm{dbf}^{\mathrm{xxx}}(\tilde{P}_q, t) \leq d_{min}$ {**true** if $\tilde{P}_q$ is schedulable, **false** otherwise}

identifies single servers and assigns a sequence number to each server that determines the order of the server in the carousel. Sequence number 0 is assigned to single servers. The algorithm returns true, if the set of servers is schedulable over m processors, and false otherwise.

Algorithm 4: Reserve sequence generator.

Input: set of k servers $\tilde{P}_q$, with $1 \leq i \leq k$
ϵ {accuracy in server inflation}
m {number of processors}
Outupt: **boolean** {**true** if schedulable}
$\Gamma[]$ {server sequence}

$S \leftarrow \frac{1}{\delta} \cdot \min_{\tau_i \in \tau}(T_i, D_i)$ {time slot}
$s \leftarrow 1$ {sequence order}
$U \leftarrow 0$ {system utilization}
for $q \leftarrow 1$ to k **do**
  {inflate servers}
  $t \leftarrow 2 \cdot \mathrm{lcm\_T}(\tilde{P}_q)$
  $U^{infl}[\tilde{P}_q] \leftarrow \mathrm{res\_inflate}(\tilde{P}_q, \epsilon, t)$
  if $U^{infl}[\tilde{P}_q] \geq 1.0$ **then**
    $U^{infl}[\tilde{P}_q] \leftarrow 1.0$
    $Tp[\tilde{P}_q] \leftarrow$ SINGLE
    $\Gamma[q] \leftarrow 0$ {0 means not in sequence}
  else
    $\Gamma[q] \leftarrow s$
    $s \leftarrow s + 1$
  end if
  $U \leftarrow U + U^{infl}[\tilde{P}_q]$
end for
return $U \leq m$

To complete the generation of the high-level schedule, we need to determine the first reserve for each processor. Let r be the number of processors used in the carousel. As illustrated in Fig. 2, the schedule of each processor must be such that processor i is S time units ahead of that of processor $i-1$, modulo r. Therefore, for each of the r processors, we need to determine the server of its first reserve and its duration. Alg. 5 shows the pseudo-code of an algorithm that computes these parameters. It takes as inputs the servers sequence, including the server parameters. The algorithm has one single loop in which it iterates over the servers (and also the r processors used to run the carousel). It then determines the servers that are active in the first processor at each multiple of the time slot S, and the time left at that point until the end of the reserve, and assigns these values as the parameters of the first reserve of the

corresponding processor.

Algorithm 5: First reserve generator

Input: set of k servers $\tilde{P}_q$, with $1 \leq i \leq k$
$\Gamma[]$ {server sequence}
Outupt: $\rho[]$ {server of first reserve of each processor}
$\Phi[]$ {duration of first reserve of each processor}

$p \leftarrow 1$ {processor index}
$S \leftarrow \min_{\tau_i \in \tau}(T_i, D_i)$ {time slot}
$t \leftarrow 0$ {accumulated schedule time}
for $q \leftarrow 1$ to k **do**
  if $\Gamma[q] <> 0$ **then**
    {only servers that belong to the carousel}
    $t \leftarrow t + U^{infl}[\tilde{P}_q] \cdot S$ {expiration time of this reserve}
    if $t \geq (p-1) \cdot S$ **then**
      {reserves crosses time slot boundary}
      $\rho[p] \leftarrow q$ {assign first server}
      $\Phi[p] \leftarrow t - (p-1) \cdot S$ {duration of first reserve}
      $p \leftarrow p + 1$
    end if
  end if
end for

VIII. Experimental Results

In this section, we report on a preliminary experimental evaluation of Carousel-EDF by comparing it with NPS-F. The results presented are not meant as evidence that Carousel-EDF has better schedulability than NPS-F, but only to show that there are workloads that, for the bounds considered and based on the overhead-aware schedulability analysis of each algorithm, are schedulable under Carousel-EDF but not under NPS-F.

A. Overhead parameters bounds

The values used for the different overhead parameters bounds are shown in Table I. The values of all parameters in this table, with exception of the $CpmdO$, were determined by experimentally measuring the overheads on a 24-core computer built from two 1.9 GHz AMD Opteron 6168 chips, each with with 12 cores, running a modified 2.6.31 Linux kernel with an NPS-F implementation, as described in more detail in [18]. The maximum values thus obtained were then rounded up. The value for $ResD$ represents the measured bound for the entire reserve delay, comprising the reserve jitter, the reserve overhead and the context switch. Although these values were not measured on our prototype implementation of Carousel-EDF, we do not expect them to be different. Furthermore, they are platform dependent and therefore should be seen only as values that may occur on a real system, as of the time of writing.

Table I: Experimentally-derived values for the various scheduling overheads (in μs).

	$RelJ$	$RelO$	$CtswO$	$ResD$	$CpmdO$
Measured	17.45	8.56	35.21	30.24	
Used values	20.00	10.00	40.00	40.00	100.00

The $CpmdO$ is highly dependent on the load and its estimation is a research problem in itself. Given the preliminary

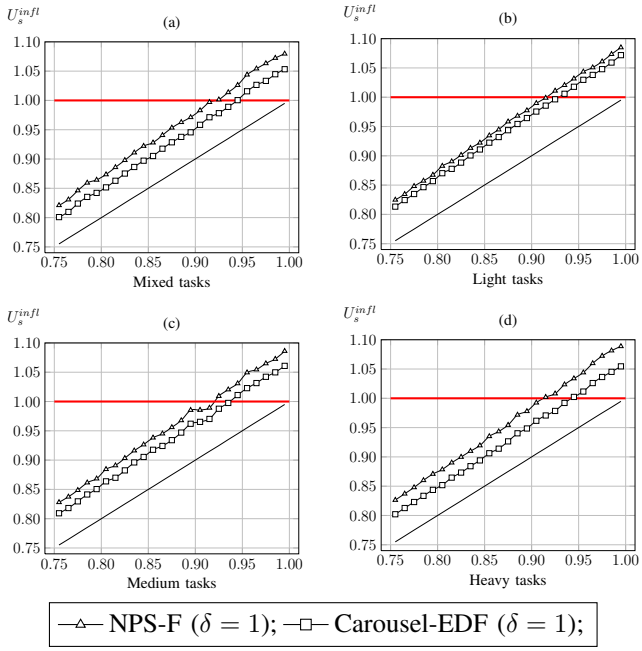

Figure 5: Normalized inflation: NPS-F *vs.* Carousel-EDF.

nature of these experiments, we set it to $100\mu s$, in order not to penalize too heavily tasks whose C is small.

Although our analysis allows to factor in interrupt overheads, we only consider tick interrupts. We did not consider other interrupts, as they are highly dependent on the load and they should not affect significantly the overall picture.

B. Task set generation

The task sets used in the experiments were generated randomly. The utilization of each task was uniformly distributed in a given range. We considered four ranges, to model different types of tasks: *light, medium, heavy*, and *mixed* with utilization in the ranges $[0.05, 0.35)$, $[0.35, 0.65)$, $[0.65, 0.95)$ and $[0.05, 0.95)$, respectively. The period of all tasks was uniformly distributed in the range $[5, 50]$ ms, with a resolution of 1 ms. All tasks generated were implicit deadline ($D_i = T_i$). The worst-case execution time of the tasks was derived from the utilization and the period of the task ($C_i = u_i \cdot T_i$).

C. Experiment

In order to compare the schedulability of Carousel-EDF with that of NPS-F, for each of the 4 types of tasks, we generated 100 task sets for each value of normalized utilization ranging from 0.75 to 1.0, with a granularity of 0.1. The number of processors in the system was 24. For each algorithm, and each task set, we determined the normalized inflated utilization using the respective overhead-aware schedulability analysis. In the case of NPS-F, we used the schedulability analysis presented in [18] modified to eliminate the release interference among servers as described in Section IV.

Fig. 5 shows, for each algorithm, the average normalized inflated system utilization, U_s^{infl}, as a function of the value of the normalized utilization of the task set. The normalized inflated system utilization under Carousel-EDF is smaller than that under NPS-F, independently of the task set type considered. This was expected, because NPS-F incurs more overheads than Carousel-EDF. However, the pessimism of the analysis may be larger for NPS-F than for Carousel-EDF. Indeed, based on experimental evidence [13], the analysis assumes that the worst case cache-related overheads caused by preemptions are the same whether or not there is a migration. On average, however, the costs incurred by preemptions with migrations are likely to be higher than without, and preemptions with migrations are more likely in Carousel-EDF than in NPS-F.

IX. Conclusions

This paper presented Carousel-EDF a novel scheduling algorithm that relies on most of the mechanisms of NPS-F, from which it evolved. By preventing the split of reserves, Carousel-EDF reduces by up to 50% the number of preemptions caused by the reserve mechanism, while preserving NPS-F utilization bounds, which were up to now the highest, among algorithms that do not use a single dispatching queue, for a given number of preemptions. The implementation of Carousel-EDF relies on a hierarchical two-level scheduler that runs in each processor. The top-level scheduler schedules servers in a round-robin fashion, whereas the low-level scheduler schedules the tasks in each server using the EDF policy. The paper also presented a schedulability analysis that is used both to partition the task set in servers and to inflate the server reserves so as to ensure that all tasks in a server meet their deadlines. This schedulability analysis was then used in an experimental study that confirmed that for overhead bounds based on values measured on a real-system, Carousel-EDF is able to schedule, on average, more task sets than NPS-F. Although the results are encouraging, there are still many issues that deserve further study. A more complete experimental evaluation considering a wider range of task sets and comparing it with other scheduling algorithms will be considered in future work.

Acknowledgements

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within projects Ref. FCOMP-01-0124-FEDER-022701 (CISTER) and FCOMP-01-0124-FEDER-020447 (REGAIN); also by FCT and the EU ARTEMIS JU, within JU grant nr.333053 (CONCERTO).

References

[1] S. Dhall and C. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, pp. 127–140, 1978.

[2] S. Baruah, N. Cohen, G. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, pp. 600–625, 1996.

[3] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *proc. of the 32nd Real-Time Systems Symposium (RTSS'11)*, Vienna, Austria, 2011, pp. 104–115.

[4] K. Funaoka, S. Kato, and N. Yamasaki, "Work-conserving optimal real-time scheduling on multiprocessors," in *proc. of the 20th IEEE Euromicro Conference on Real-Time Systems (ECRTS'08)*, Prague, Czech Republic, 2008, pp. 13–22.

[5] S. Kato and N. Yamasaki, "Global edf-based scheduling with laxity-driven priority promotion," *J. Syst. Archit.*, vol. 57, no. 5, pp. 498–517, May 2011.

[6] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[7] J. Anderson, V. Bud, and U. Devi, "An edf-based scheduling algorithm for multiprocessor soft real-time systems," in *proc. of the 17th IEEE Euromicro Conference on Real-Time Systems (ECRTS'05)*, Palma de Mallorca, Balearic Islands, Spain, 2005, pp. 199–208.

[8] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemption," in *proc. of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Application (RTCSA'06)*, Sydney, Australia, 2006, pp. 322–334.

[9] S. Kato and N. Yamasaki, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *proc. of the 21st Euromicro Conference on Real-Time Systems (ECRTS 09)*, Dublin, Ireland, 2009, pp. 239–248.

[10] K. Bletsas and B. Andersson, "Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound," in *proc. of the 30th IEEE Real-Time Systems Symposium (RTSS'09)*, Washington, DC, USA, 2009, pp. 385–394.

[11] A. Burns, R. I. Davis, P. Wang, and F. Zhang, "Partitioned edf scheduling for multiprocessors using a c=d task splitting scheme," *Real-Time Syst.*, vol. 48, no. 1, pp. 3–33, Jan 2012.

[12] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011.

[13] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers," in *proc. of the 31st IEEE Real-Time Systems Symposium (RTSS'10)*. San Diego, CA, USA: IEEE Computer Society, 2010, pp. 14–24.

[14] S. Baruah, A. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *proc. of the 11st IEEE Real-Time Systems Symposium (RTSS'90)*, Lake Buena Vista, Florida, USA, 1990, pp. 182–190.

[15] A. Bastoni, "Towards the integration of theory and practice in multiprocessor real-time scheduling," Ph.D. dissertation, University of Rome "Tor Vergata", 2011.

[16] B. Brandenburg, H. Leontyev, and J. Anderson, "An overview of interrupt accounting techniques for multiprocessor real-time systems," *J. Syst. Archit.*, vol. 57, no. 6, pp. 638–654, 2011.

[17] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[18] P. B. Sousa, K. Bletsas, E. Tovar, P. Souto, and B. Akesson, "Unified overhead-aware schedulability analysis for slot-based task-splitting," CISTER, Polytechnic Institute of Porto (ISEP-IPP), Technical report CISTER-TR-130201, 2013. [Online]. Available: https://www.cister.isep.ipp.pt/docs/

APPENDIX A.
UTILIZATION BOUND

By design, Carousel-EDF offers, for the same value of corresponding parameter δ, the same utilization bound as the original NPS-F algorithm. This bound is given by the following expression:

$$\frac{2 \cdot \delta + 1}{2 \cdot \delta + 2}$$

The following reasoning establishes this.

Lemma 1. *A set of k servers is schedulable on m processors under Carousel-EDF if and only if*

$$\sum_{q=1}^{k} U^{infl}[\tilde{P}_q] \leq m$$

Proof: This follows from the fact that Carousel-EDF successfully schedules all servers if and only if it provides $U^{infl}[\tilde{P}_q] \cdot S$ time units to each server $\tilde{P}_q$ within every time window of length S (equal to the time slot). This processing time is provided to the k servers from the m processors (by design, in such a manner that each server only receives processing time from at most one processor at any time instant). ■

Lemma 2. *After the bin-packing stage of Carousel-EDF, if there exist $k \geq 2$ utilised bins, it then holds that $\frac{1}{k}\sum_{q=1}^{k} U[\tilde{P}_q] > \frac{1}{2}$, for the k utilized bins (servers).*

Proof: By contradiction, if the claim were false then there would exist (at least) one pair of servers $\tilde{P}_q$ and $\tilde{P}_r$ such that $U[\tilde{P}_q] + U[\tilde{P}_r] \leq 1$. But this is impossible under the bin-packing scheme employed. ■

Lemma 3. *: For any server $\tilde{P}_q$, it holds that $U^{infl}[\tilde{P}_q] \leq \text{inflate}(U[\tilde{P}_q])$.*

Proof: This holds because Carousel-EDF inflates each server $\tilde{P}_q$ to the minimum $U^{infl}[\tilde{P}_q]$ sufficient for schedulability, according to an exact schedulability test. In contrast, the inflate function used by NPS-F potentially over-provisions. ■

Now we can prove the utilisation bound of Carousel-EDF. The following theorem and its proof are analogous to Theorem 3 from [10], which proved the utilisation bound of NPS-F.

Theorem 1. *The utilization bound of Carousel-EDF is $\frac{2\cdot\delta+1}{2\cdot\delta+2}$.*

Proof: An equivalent claim is that every task set with utilisation not greater than $\frac{2\cdot\delta+1}{2\cdot\delta+2} \cdot m$ is schedulable under Carousel-EDF. We will prove this equivalent claim.

Let us distinguish two complementary cases, depending on the number k of bins (servers) with tasks assigned to them, after the bin-packing stage: (i) $k = 1$; (ii) $k \geq 2$.

In the first case ($k = 1$), even if the system has just one processor ($m = 1$), the task set is always schedulable, because the reserve needed to implement the server corresponding to that single bin can always be accommodated on a single processor. Therefore, let us focus on the other case ($k \geq 2$):

From Lemma 1, Carousel-EDF successfully maps all k servers to a feasible online schedule if and only if

$$\sum_{q=1}^{k} U^{infl}[\tilde{P}_q] \leq m$$

Applying Lemma 3 to the above, a sufficient condition for the schedulability of all k servers is

$$\sum_{q=1}^{k} \text{inflate}(U[\tilde{P}_q]) \leq m \tag{20}$$

For the function $\text{inflate}(U) = \frac{(\delta+1)\cdot U}{U+\delta}$, it holds that $\frac{d}{dU}\text{inflate}(U) > 0$ and $\frac{d^2}{dU^2}\text{inflate}(U) < 0$ over the interval [0,1]. Therefore, from Jensen's Inequality [**?**] we have:

$$\sum_{q=1}^{k} \text{inflate}(U[\tilde{P}_q]) \leq k \cdot \text{inflate}(\bar{U})$$

where

$$\bar{U} = \frac{1}{k} \sum_{q=1}^{k} U[\tilde{P}_q])$$

Hence, combining this with the sufficient condition of Inequality 20, a new sufficient condition for the successful mapping of the k servers is

$$k \cdot \text{inflate}(\bar{U}) \leq m \tag{21}$$

Now, let the function $\alpha(U)$ denote the expression $\text{inflate}(U) - U$. Then, $\alpha(U) = \frac{U \cdot (1+U)}{U+\delta}$ and Inequality 21 can be rewritten as

$$k \cdot \left(\bar{U} + \alpha(\bar{U})\right) \leq m \tag{22}$$

Given that $\alpha(U)$ is a decreasing function of U over $[\frac{1}{2}, 1]$ it holds that

$$\frac{\alpha(U)}{U} < \frac{\alpha(\frac{1}{2})}{\frac{1}{2}} = \frac{1}{2\delta+1}, \ \forall U \in (\frac{1}{2}, 1] \Rightarrow \alpha(U) < \frac{1}{2\delta+1} \cdot U \ \forall U \in (\frac{1}{2}, 1]$$

Combining the above with the fact that (from Lemma 2) $\bar{U} > \frac{1}{2}$, we obtain from Inequality 21 the following new sufficient condition:

$$k \cdot \left(\bar{U} + \frac{1}{2\delta+1} \cdot \bar{U}\right) \leq m \tag{23}$$

In turn, this can be equivalently rewritten as:

$$k\bar{U} \leq \frac{2\delta+1}{2\delta+2} m \tag{24}$$

But the left-hand side of the above inequality, corresponds to the cumulative utilisation of all tasks in the system. Therefore, if $\sum_{\tau_i \in \tau} u_i \leq \frac{2\delta+1}{2\delta+2} m$, this is a sufficient condition for schedulability under Carousel-EDF. This proves the theorem. ■

APPENDIX B.
UPPER BOUNDS ON PREEMPTIONS

Under either of the two approaches to server mapping formulated in [10], for NPS-F, an upper bound $\mathrm{N}_{\mathrm{pre}}^{\mathrm{NPS-F}}$ on the preemptions (including migrations) related to the reserves within a time interval of length t is given by:

$$\mathrm{N}_{\mathrm{pre}}^{\mathrm{NPS-F}} < \left\lceil \frac{t}{S} \right\rceil \cdot 3 \cdot m \tag{25}$$

This follows from the fact that, within each time slot of length S, other that preemptions due to EDF, there occur: (a) one preemption per server (hence k in total), when its last (or only) reserve runs out; plus (b) $m-1$ (at most) migrations, corresponding to the case when some server migrates between a pair of successively indexed processors.

Therefore

$$\mathrm{N}_{\mathrm{pre}}^{\mathrm{NPS-F}} = \left\lceil \frac{t}{S} \right\rceil \cdot k + \left\lceil \frac{t}{S} \right\rceil \cdot (m-1) \tag{26}$$

and the bound of Inequality 25 is derived from the fact that $m-1 < m$ and also, as proven by Corollary 1 in [10], $k < 2 \cdot m$.

By comparison, Carousel-EDF generates within each timeslot, at most k preemptions/migrations (other than those due to EDF). Each of those corresponds to the ending of the reserve of the corresponding server. Therefore

$$\mathrm{N}_{\mathrm{pre}}^{\mathrm{Carousel-EDF}} = \left\lceil \frac{t}{S} \right\rceil \cdot k < \left\lceil \frac{t}{S} \right\rceil \cdot 2 \cdot m \tag{27}$$

By comparing the respective bounds (Eqs. 25 and 27), we observe that the upper bound for preemptions due to the time division under Carousel-EDF is 33.3% smaller than that under NPS-F.

However, the above upper bounds are pessimistic with respect to the value of k (i.e. assume that k is close to $2m$, which is the unfavorable scenario, in terms of preemptions/migrations generated). However the number of servers k may range from $m+1$ to $2m-1$; if it were less, we would use partitioning and it could not be $2m$ or higher from Lemma 2 in App. A. In many cases therefore, k could in fact be close to m. And, by inspection (see Eq. 26, for $k = m+1$ and $m \to \infty$, the reduction in the upper bound on reserve-related preemptions/migration when using Carousel-EDF, as compared to NPS-F, would tend towards 50%. Therefore, Carousel-EDF involves 33.3% to 50% fewer reserve-related preemptions and migrations than NPS-F.

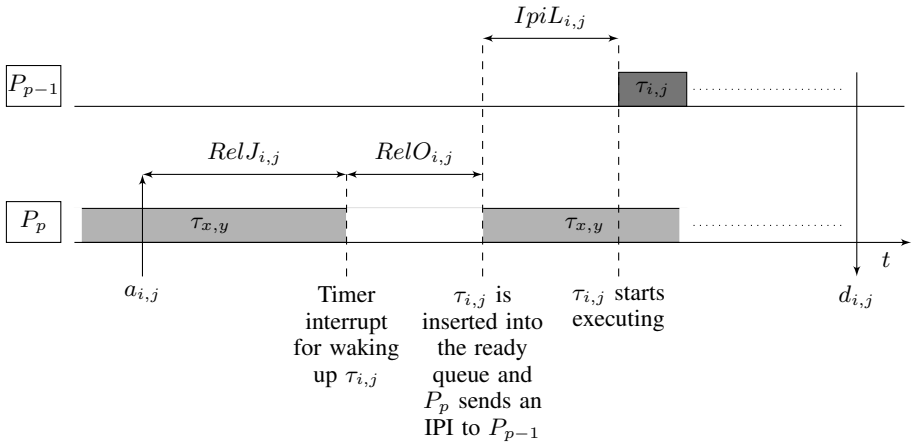

Figure 6: Illustration of the IPI latency at the release of a split job. It does not include the time for context switching.

APPENDIX C.
SUPPORTING OTHER JOB RELEASE MECHANISM

The Carousel-EDF scheduler can also support the release of jobs by interrupts. Using the criterion that jobs of a task can be released by any processor that execute such jobs. Then, the jobs can be released by any processor. In that case, we have to compute the execution demand of a server taking into account the release interference of the other servers. The release interference is computed as:

$$\mathrm{dbf}^{\mathrm{res}}_{\mathrm{RelI}}(\tilde{P}_q, t) = \sum_{i=1 \wedge i \neq q}^{k} \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_i, t) \tag{28}$$

Furthermore, in this case, it may require the use of Inter-Processor Interrupts (IPIs). The IPIs are used to notify the dispatcher in another processor of the release of a task. As a result, the dispatching of a task may incur an *IPI Latency*. (Note that this parameter does not include the time required for context switching, this is already accounted for, as it will occur whether or not the release is via an IPI.) Figure 6 illustrates such a case. The arrival of a job of task τ_i assigned to a split server shared between processors P_p and P_{p-1}, for instance, occurs at a time instant t and is handled on processor P_p, but this time instant t falls inside the reserve of that server on the other processor, P_{p-1}. If this job is the highest priority job of its server, P_p notifies P_{p-1} of the new arrival via an IPI. Clearly, the overhead caused by the IPI, $IpiL_{i,j}$, only delays the dispatch of job $\tau_{i,j}$ (and only if job $\tau_{i,j}$ is the highest priority job of its server). Thus, the IPI latency, has an effect similar to the release jitter.

Let $IpiL$ be an upper bound for the IPI latency. Then, for incorporating this release mechanism, the $\mathrm{dbf}^{\mathrm{res}}(\tilde{P}_q, t)$ (see Eq. 18) has to be amended to:

$$\begin{aligned}
&\mathrm{dbf}^{\mathrm{res}}(\tilde{P}_q, t) = \\
&\quad \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q], t) + \\
&\quad \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i + RelJ + IpiL}{T_i} \right\rfloor + 1\right) \cdot (C_i + 2 \cdot CtswO) + \\
&\quad \mathrm{dbf}^{\mathrm{part}}_{\mathrm{CpmdO}}(\tau[\tilde{P}_q], t) + \\
&\quad \max\left(0, \left\lfloor \frac{t - D^{fake} + ResL}{S} \right\rfloor + 1\right) \cdot (C^{fake} + RelL) + \\
&\quad \mathrm{dbf}^{\mathrm{res}}_{\mathrm{RelI}}(\tau[\tilde{P}_q], t)
\end{aligned} \tag{29}$$