Task Splitting and Load Balancing of Dynamic Real-Time Workloads for Semi-Partitioned EDF

Daniel Casini, *Member, IEEE*, Alessandro Biondi, *Member, IEEE*, and Giorgio Buttazzo, *Fellow, IEEE*

Abstract—Many real-time software systems, such as those commonly found in the context of multimedia, cloud computing, robotics, and real-time databases, are characterized by a dynamic workload, where applications can join and leave the system at runtime. Global schedulers can transparently support dynamic workload without requiring any off-line task-allocation phase, thus providing advantages to the system designer. Nevertheless, such schedulers exhibit poor worst-case performance when compared to semi-partitioned schedulers, which instead can achieve near-optimal schedulability performance when used in conjunction with smart task splitting and partitioning techniques, and they are also lighter in terms of run-time overhead. This article proposes an approach to efficiently schedule dynamic real-time workloads on multiprocessor systems by means of semi-partitioned scheduling. A linear-time approximation scheme for the C=D splitting algorithm under partitioned EDF scheduling is proposed. Then, a load-balancing algorithm is presented to admit new real-time workloads with a limited number of re-allocations. The article finally reports on a large-scale experimental study showing that (i) the linear-time approximation is characterized by a very limited utilization loss compared with the corresponding exact approach (that has a much higher complexity), and that (ii) the whole approach allows achieving considerable improvements with respect to global and partitioned EDF scheduling.

Index Terms—Real-time systems, dynamic workloads, semi-partitioned scheduling, schedulability analysis, load balancing, partitioning

1 Introduction

Several time-sensitive applications include computational activities (tasks) that may join and leave the system at runtime, for instance, to respond to specific events in their operating environment. This is common in multimedia software systems [1] (including those widely available in smartphones and tablets), cloud computing [2], real-time databases, robotics systems, and open environments, in which some components may change while the rest of the system continue to operate. To name a concrete example, the applications developed for the Robotic Operating System (ROS) [3], a popular middleware layer for the rapid prototyping, development, and deployment of robots, are software systems in which the workload can be dynamic. Indeed, ROS allows reacting to the occurrence of specific environmental conditions (e.g., the sudden occurrence of an obstacle in front of the robot) by creating or killing computational nodes, while the other nodes remain operational [4].

Furthermore, the most popular real-time operating systems, e.g., VxWorks, QNX and Linux (with the `SCHED_DEADLINE` scheduling class), provide specific system calls (e.g., `taskSpawn()` in VxWorks) to create and activate tasks at runtime. Most commonly, these systems implement *global* scheduling policies such as *global fixed-priority* (G-FP) and *global earliest-deadline first* (G-EDF), which have the benefit of providing automatic and application-transparent load balancing across the available processors. This benefit likely determined the popularity of such schedulers; however, they have been demonstrated to be not optimal and to exhibit poor worst-case performance due to several issues that have been identified in the literature [5]. Optimal multiprocessor scheduling algorithms, such as RUN [6], U-EDF [7], QPS [8], and LLREF [9], have been proposed, but they are generally more complex (and hence more difficult to implement) and more expensive in terms of run-time overhead when compared to G-FP and G-EDF.

Partitioned and *semi-partitioned* scheduling represent effective alternatives to global schedulers. Partitioned scheduling relies on a static task-to-processor mapping, which for static workloads is typically determined with an off-line design phase. Notably, Sun and Di Natale [10] and Biondi and Sun [11] proved that the most popular analysis techniques for global schedulers such as G-FP and G-EDF can deem schedulable only task sets that are also schedulable under partitioned scheduling, hence recommending the use of the latter. However, in the presence of dynamic workloads, partitioned scheduling requires facing with on-line task allocation issues that may not be easy to solve if a "good" schedulability performance is desired. Furthermore, partitioned scheduling tend to lead to poor schedulability performance in the presence of high-utilization tasks.

Semi-partitioned scheduling builds upon partitioned scheduling by allowing some of the tasks to be *split* among multiple processors, i.e., being subject to a controlled (and limited) migration at specific time instants

• *Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo are with the TeCIP Institute and the Department of Excellence in Robotics & AI of the Scuola Superiore Sant'Anna, 56124 Pisa, Italy. E-mail: {daniel.casini, alessandro.biondi, giorgio.buttazzo}@santannapisa.it.*

Manuscript received 1 Sept. 2020; revised 4 Nov. 2020; accepted 10 Nov. 2020. Date of publication 16 Nov. 2020; date of current version 8 Nov. 2021. (Corresponding author: Daniel Casini.)
Recommended for acceptance by B. Parhami.
Digital Object Identifier no. 10.1109/TC.2020.3038286

during their execution. In this way, the performance of partitioned scheduling is improved by distributing the load generated by some tasks across multiple processors. As in the case of partitioned scheduling, semi-partitioned scheduling algorithms typically come with an off-line task allocation strategy, and are hence not suitable to be used on-line with the purpose of supporting dynamic workloads. Nevertheless, Brandenburg and Gül [12] showed how semi-partitioned EDF scheduling via the C=D splitting algorithm [13], when used in conjunction with smart task partitioning techniques, can guarantee near-optimal performance, while being a much simpler and lighter (in terms of run-time overhead) approach with respect to global schedulers. As most of the papers targeting multiprocessor real-time scheduling, their work focused on static task sets only. However, the relevance of such a result suggests that also dynamic workloads may benefit of semi-partitioned scheduling.

Nonetheless, supporting C=D semi-partitioning scheduling of dynamic workloads gives rise to some non-trivial challenges. Specifically, the C=D splitting algorithm has a high computational complexity, which would lead to high overheads if executed on-line, thus resulting being not suitable for dynamic workloads. Furthermore, load-balancing algorithms are needed to support the dynamic allocation and splitting of incoming workloads.

Contribution. This paper makes the following three contributions. First, it proposes a linear-time approximate algorithm for efficiently splitting workload under C=D semi-partitioned scheduling, which enables making practically viable online scheduling decisions. Second, it presents load-balancing algorithms to admit new workload while performing limited re-allocations to facilitate the admission of future workloads. Third, it reports on two large-scale experimental studies that have been conducted to assess the performance of the proposed methods.

Paper Structure. The rest of the paper is organized as follows. Section 2 introduces the system model, reviews the essential background, and presents the adopted notation. Section 3 proposes a linear-time algorithm for performing the C=D splitting. Section 4 presents a set of load-balancing algorithms for admitting new workload and performing limited workload re-allocations. Section 5 reports on the experimental results. Section 6 discusses the related work. Finally, Section 7 concludes the paper and illustrates some future work.

This paper extends a preliminary conference version of this work [14] by: (i) proposing a new mathematical formulation of the linear-time method for C=D splitting, which extends the previous one adopted in [14] allowing to split constrained-deadline reservations instead of considering implicit deadlines only as in [14], (ii) simplifying and clarifying the load balancing strategies, (Section 4), (iii) discussing how to handle scheduling transients under semi-partitioned scheduling (Section 4.4), and (iv) reporting new experimental results to explore the empirical performance of the new contributions of this paper.

2 System Model and Background

This work addresses the problem of scheduling a dynamic workload composed of *reservation servers* upon m identical processors. A reservation r_i is characterized by a tuple (C_i, D_i, T_i), where C_i is the execution time *budget*, T_i is the minimum inter-replenishment time of the budget, and D_i is the relative constrained deadline[1] $D_i \leq T_i$. Reservations may dynamically require to join and leave the system. Upon each join request, a schedulability-based *acceptance test* (detailed later) is performed to determine whether the reservation can be accepted. Reservations that do not pass the test are rejected (i.e., ignored). At any point in time, $\mathcal{R}$ denotes the set of currently admitted reservations. Reservations are considered to be independent (i.e., they do not share resources other than the processors). Each reservation can be used for manifold purposes, including (i) serving the execution of a single periodic/sporadic real-time task; (ii) implementing a *hierarchical scheduling framework* [16], i.e., managing a local scheduler upon the reservation that in turn manages a set of real-time tasks; and (iii) serving the execution of non-real-time (i.e., best-effort) workload.

Each reservation server $r_i \in \mathcal{R}$ releases a potentially-infinite number of *instances*. During each instance, the server executes for at most C_i time units and then is descheduled. An instance of the server starts when the budget is refilled and r_i has pending workload to execute. An instance terminates either (i) when the budget is exhausted or (ii) the server does not have anymore pending workload to execute. Note that the release times of the instances follow a *sporadic* pattern.

The results presented in this work are not limited to a specific reservation algorithm, but the server behavior has to comply with the runtime requirements discussed in Section 2.1.

A reservation r_i is said to be *schedulable* if it can execute its entire budget C_i before its relative deadline for any instance of r_i. The acceptance test must guarantee that *all* the reservations in $\mathcal{R}$ are always schedulable. In this work, the acceptance test adopts an on-line *load balancing* algorithm that allocates the reservations to the processors, which is later presented in Section 4.

The adoption of reservation servers allows guaranteeing *temporal isolation* of the workload, thus providing a protection mechanism against tasks' overruns or processor-eager, best-effort computational activities. This feature is particularly suited for systems running dynamic workloads, for which—conversely to static, safety-critical real-time systems—accurate estimates of the tasks' *worst-case execution time* (WCET) are often not available. Such a computational model is also of practical relevance, as it is analogous to the one supported by the `SCHED_DEADLINE` scheduling class of Linux, today available in the main distribution of the kernel and hence present in billions of machines and devices around the world.

In this paper, reservations are assumed to be managed under *semi-partitioned* EDF scheduling with the C=D splitting scheme [12], [13], which is briefly reviewed in the next section.

1. Although most of the algorithms for implementing reservation servers consider an implicit deadline, the interest toward deadline-constrained reservations recently arose in the context of ongoing developments of the `SCHED_DEADLINE` scheduling class of the Linux kernel (see https://lkml.org/lkml/2017/2/10/611), also finding responses from the scientific community [15].

2.1 C=D Semi-Partitioned Scheduling of Reservations

Semi-partitioned scheduling improves the schedulability performance of partitioned scheduling when valid static reservation-to-processor allocations cannot be found or simply do not exist. This is done by allowing some reservations to be *split* across multiple processors, thus involving the migration of the workload executing in such reservations. More specifically, the budget of semi-partitioned reservations is divided into multiple portions (i.e., time chunks) that are executed on different processors with precedence constraints.

The C=D scheme proposed by Burns *et al.* [13] has been found to be a particularly effective method to split the budget. According to this approach, the budget is split into $n \geq 2$ chunks, each to be executed on a different processor. That is, each instance of a reservation with split budget starts executing the first chunk of budget on a processor, then migrates to another processor to executed the second chunk of budget, and so on until the budget is finished. When executing the first $n-1$ chunks, the reservation is scheduled with C=D, i.e., with a relative deadline equal to the corresponding duration of the portion. In this way, such chunks have always zero laxity. Conversely, when executing the last chunk of budget, the reservation is scheduled with a deadline greater than or equal to the duration of the chunk ($D \geq C$). Brandenburg and Gül [12] proposed an extension of the original Burns *et al.*'s approach where the deadline assignment is reversed. This approach allows taking advantage of slack reclamation, which in turn provides the benefit of reducing the number of migrations in the average case. The latter scheme is the one considered in this paper as the run-time scheduling mechanism.

Run-Time Scheduling Mechanism. As soon as a server is admitted, its budget is immediately replenished to maximum value C_i. If an instance of a server r_i starts at a time t, the next budget replenishment is set at time $t + T_i$. Each instance of r_i beginning at time t is scheduled with absolute deadline $t + D_i$. The servers execute without self-suspensions: i.e., the budget is discharged if the server has pending workload that is not ready to execute and is depleted when the server stops having pending workload. For each processor P, at each point in time the reservation allocated to P that has (i) a pending instance and (ii) the earliest absolute deadline is selected for being executed.

Under semi-partitioned scheduling, some reservations servers never migrates across processors, i.e., they are statically partitioned. Such reservations are referred to as *partitioned reservations*, while the others are referred to as *semi-partitioned reservations*.

Given a semi-partitioned reservation r_i whose budget C_i is split into two portions, say C_t and C_h such that $C_i = C_t + C_h$, the first portion of budget is scheduled on a processor P' with relative deadline $D_h = D_i - C_t$ and minimum inter-replenishment time T_i, while the second one is scheduled on a different processor $P'' \neq P'$ with relative deadline $D_t = C_t$ and minimum inter-replenishment time T_i. This split gives rise to two sub-reservations, denoted as *head reservation* and *tail reservation*, respectively.

At run-time, the execution of the workload executing upon a semi-partitioned reservation r_i is subject to the following rules. Consider an instance of r_i released at time t and suppose that the server has always pending workload to execute. The first C_h units of budget of r_i are served by its head reservation, i.e., on processor P'. Then, every time the budget C_h is exhausted, the workload executing upon r_i is *migrated* to processor P'', where it will be served by the tail reservation of r_i. If the head reservation is schedulable within its relative deadline D_h, this event is guaranteed to happen at a time $t' \leq t + D_h$. The head reservation is de-scheduled and its budget will be replenished at time $t + T_i$.

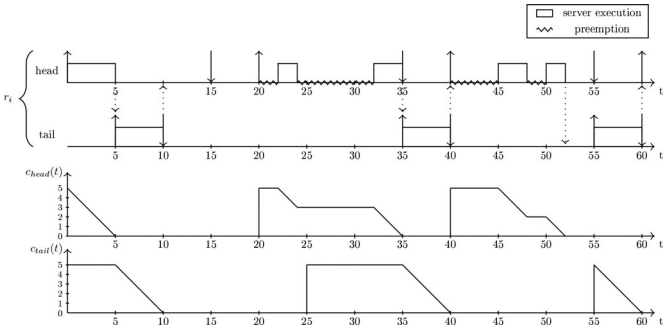

Fig. 1. Example of semi-partitioned scheduling of a reservation r_i ($C_i = 10, D_i = T_i = 20$) under C=D splitting. The budget of r_i is split into two budget chunk of 5 time units each, to be executed on two processors. Up-arrows denote the beginning of an instance of the servers. Down-arrows denote the absolute deadlines of each instance. Dotted arrows denote the migration of the workload executing upon r_i across the two processors.

If the tail reservation is schedulable within its relative deadline $D_t = C_t$, the C=D approach [12] ensures that C_t units of time are served before time $t + D_i$, thus guaranteeing the schedulability of r_i. Once the budget of the tail reservation is exhausted, also this server is de-scheduled and its budget will be replenished at time $t'' + T_i$, where t'' is the starting time of its last instance. The pending workload upon r_i will then be able to restart the execution from processor P' (thus involving another migration) at time $t + T_i$. Note that, although the two sub-reservations have the same minimum inter-replenishment time, their replenishment times are generally not synchronized.

The approach generalizes to the case in which the budget is divided in more than two parts by splitting a reservation into one head reservation and *multiple* tail reservations.

Example. Consider a reservation r_i with $C_i = 10$ and $T_i = D_i = 20$ that is split into: (i) one head reservation configured with $C_h = 5$ and $D_h = 15$; (ii) one tail reservation configured with $C_t = 5$, $D_t = 5$. A possible schedule of such sub-reservations is illustrated in Fig. 1, together with the evolution of their budgets over time (indicated by functions $c_{head}(t)$ and $c_{tail}(t)$, respectively).

It is worth observing that the C=D approach implicitly poses the limitation that *no more than one* tail reservation can be allocated on each processor.

*How to Split and Allocate the Reservations?*The two main issues with semi-partitioned scheduling consists in (i) deciding how to size the budget portions of semi-partitioned reservations, i.e., selecting a splitting algorithm, and, (ii) determining how to allocate reservation, e.g., by means of bin-packing heuristics (such as variants of first-fit and worst-fit). Previous work assumed a static workload and leveraged an off-line design phase to solve this problem.

The next section briefly reviews the C=D splitting algorithm proposed by Burns *et al.* [13], which has also been adopted by Brandenburg and Gül in [12].

2.2 Burns *et al.*'s C=D Splitting Algorithm

Whenever a reservation r_i cannot be statically allocated to a single processor, Burns *et al.* [13] proposed to accomplish the splitting with the following two-phase approach:

i) Given a processor P_k, an algorithm is used to compute the *maximum* $C_t < C_i$ for which a tail reservation with budget C_t, deadline $D_t = C_t$, and minimum inter-replenishment time T_i can be allocated to P_k *such that all the reservations running on* P_k *are schedulable*.
ii) The remaining portion of budget $C_h = C_i - C_t$ is then allocated to another processor $P_x \neq P_k$ following a bin-packing heuristic (or is in turn selected for being split).

The core of their proposal consists in the algorithm adopted in phase (i). Such an algorithm starts from the value of C_i'' for which the selected processor P_k is fully utilized (i.e., such that $\sum_{r_i \in \mathcal{R}_k} C_i/T_i = 1$) after allocating the tail reservation; then, it allocates the tail reservation to P_k and applies the following steps:

1) Perform the Quick convergence Processor-demand Analysis (QPA) [17] to determine whether the set of reservations allocated to P_k is schedulable.
2) If not, recompute a reduced value of C_t by means of a *fixed-point iteration* based on the failure point of the QPA (please refer to [13] for further details). Then, re-iterate the procedure from step 1 until the QPA does not fail.
3) If, at any iteration, the computed value of C_t reduces to 0, then the tail reservation cannot be allocated to processor P_k.

This algorithm is optimal, in the sense that it finds the maximum value of C_t for which a tail reservation can be safely allocated to processor P_k. However, it suffers from a high computational complexity. The QPA has a pseudo-polynomial time complexity when the utilization of the analyzed processor is strictly lower than one, while has exponential complexity in the case of a fully-utilized processor. Note that the latter case corresponds to the starting condition of the algorithm and that the QPA is applied multiple times. In addition, it requires the execution of fixed-point iterations that further increase the algorithm complexity. To the best of our knowledge, the actual complexity of this algorithm is unknown: anyway, it is clearly *unsuitable for performing on-line decisions* concerning the splitting of the reservations, especially if multiple alternatives for the splitting must be evaluated by a load balancing algorithm—which is the primary objective of this work.

2.3 Notation and Table of Symbols

The m processors are referred to as $P_1, P_2, \ldots, P_m$. The set of n_k reservations allocated to processor P_k (both statically or resulting from a split) is denoted by $\mathcal{R}_k$, with $\bigcap_{k=1}^{m} \mathcal{R}_k = \emptyset$. The utilization of a reservation r_i is denoted as $U_i = C_i/T_i$. Two functions $tail(P_k) = \{true, false\}$ and $head(P_k) = \{true, false\}$ are used to indicate whether a tail and a head reservation is allocated to P_k, respectively. If $tail(P_k) = true$, then $r_{t,k}$ denotes the (only) tail reservation allocated to P_k. Similarly, if $head(P_k) = true$, then $r_{h,k}$ denotes the head reservation allocated to P_k with the largest utilization. For the sake of simplicity, the subscript k is omitted when referring to an arbitrary processor or it is clear from the context.

The set of n_k^P partitioned reservations allocated to P_k is denoted as $\mathcal{R}_k^P \subseteq \mathcal{R}_k$. Given a tail reservation $r_{t,k}$ (resp., head reservation $r_{h,k}$), the *father* reservation that has been split is denoted as $\mathcal{F}(r_{k,t})$ (resp., $\mathcal{F}(r_{k,h})$). Finally, $\mathcal{P}(r_i)$ denotes the processor to which reservation r_i is allocated to. The main notation adopted throughout the paper is summarized in Table 1.

TABLE 1
Main Notation Adopted Throughout the Article

Symbol	Description
$\mathcal{R}$	set of reservations admitted into the system
P_k	k-th processor
$\mathcal{R}_k$	set of reservations allocated to processor P_k
$\mathcal{R}_k^P$	set of partitioned reservations allocated to processor P_k
n_k	number of reservations allocated to processor P_k
n_k^P	number of partitioned reservations allocated to processor P_k
r_i	i^{th} reservation
C_i	budget of r_i
T_i	minimum inter-replenishment time of r_i
D_i	relative deadline of r_i
U_i	utilization of r_i
$r_{h,k}$	head reservation allocated to P_k
$r_{t,k}$	tail reservation allocated to P_k
$\mathcal{F}(r_i)$	father reservation of a tail or head reservation r_i
$\mathcal{P}(r_i)$	processor in which r_i is allocated

3 An Approximate Algorithm For C=D Splitting

This section proposes a new algorithm for computing a lower-bound to the maximum zero-laxity (C=D) portion of budget that can be allocated to a processor, thus allowing to compute an approximate solution to the C=D splitting discussed in Section 2.2. The algorithm has been designed to have a *linear time* complexity in order to be efficiently applied for on-line load balancing.

First, an approximate sensitivity analysis is presented in Section 3.1. Then, Section 3.2 shows how the sensitivity analysis can be leveraged to design an algorithm that splits the budget of a reservation. Finally, Section 3.3 discusses some implementation issues and the algorithm complexity.

3.1 Approximate Sensitivity Analysis

The method proposed in this paper is based on the *processor-demand criterion* (PDC) proposed by Baruah *et al.* [18]. The PDC analysis is based on the notion of *demand bound function* and provides an exact schedulability test for a set of constrained-deadline sporadic tasks executing upon a single processor under EDF scheduling. Since the reservation servers considered in this work behave as sporadic tasks [12], the schedulability of the reservations allocated to a given processor P_k can be verified by checking the PDC as $\forall t \geq$

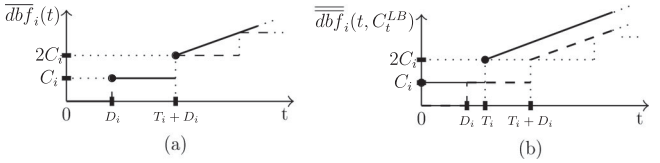

Fig. 2. Illustrations of the demand bound functions introduced in Section 3 (solid lines) with $\nu_i = 1$. The dashed lines in inset (a) depict the functions $dbf_i(t)$, while the dashed line in inset (b) depicts function $\overline{dbf}_i(t)$. Inset (b) considers function $\overline{\overline{dbf}}_i(t, C_t^{LB})$ with $C_t^{LB} = 0$.

0, $\sum_{r_i \in \mathcal{R}_k} dbf_i(t) \leq t$, where $dbf_i(t)$ is the demand bound function of r_i and is defined as

$$dbf_i(t) = \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i. \tag{1}$$

To design the approximate splitting algorithm, the demand bound function of each reservation is first approximated by an upper bound. Following the results in [19], the demand bound function of any reservation $r_i \in \mathcal{R}_k$ is upper-bounded by

$$\overline{dbf}_i(t) = \begin{cases} dbf_i(t) & \text{if } t < \nu_i T_i + D_i \\ C_i + U_i(t - D_i) & \text{otherwise.} \end{cases} \tag{2}$$

where the parameter $\nu_i \geq 0$ denotes the number of steps of the original demand bound function that are retained in the approximation. Such a function is illustrated in Fig. 2a. Leveraging this bound, it is possible to formulate a sufficient PDC-based condition to verify the schedulability of the reservations allocated to a processor, which is provided by the following theorem.

Theorem 1 (From [19]). *A set of reservations $\mathcal{R}_k$ is EDF-schedulable on a single processor if $\sum_{r_i \in \mathcal{R}_k} U_i \leq 1$ and*

$$\forall t \in \bigcup\nolimits_{r_i \in \mathcal{R}_k} \xi(r_i), \quad \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) \leq t \tag{3}$$

where

$$\xi(r_i) = \{sT_i + D_i\}, \ s = 0, \ldots, \nu_i \tag{4}$$

With the above theorem in place, it is possible to formulate the following optimization problem.

Problem Definition. Consider a set of reservations $\mathcal{R}_k$ allocated to a processor P_k that *does not* already include a tail reservation. By Theorem 1, a safe budget C_t for a tail reservation r_t with minimum inter-replenishment time T_t, such that r_t can be safely allocated to P_k, can be computed by solving the following optimization problem:

$$\begin{aligned} &\text{maximize} C_\text{t} \\ &\text{subject to} \sum_{r_i \in \mathcal{R}_k} \frac{C_i}{T_i} + \frac{C_\text{t}}{T_\text{t}} \leq 1 \\ &\qquad \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf}_t(t) \leq t, \\ &\qquad \forall t \in \bigcup\nolimits_{r_i \in \{\mathcal{R}_k \cup r_t\}} \xi(r_i). \end{aligned}$$

This optimization problem can be manipulated to obtain a sub-optimal solution in a *closed form*. Given a lower-bound $C_\text{t}^\text{LB} \geq 0$ to the zero-laxity budget of r_t, the problem is rewritten by means of $J+1$ constraints of the form $C_\text{t} \leq V_j(\mathcal{R}_k, T_t, C_\text{t}^\text{LB})$ (with $j = 0, \ldots, J$), whose left-hand side terms are *independent* of C_t, so that the solution can be easily computed as $C_\text{t} = min_{j=0,\ldots,J} \{V_j(\mathcal{R}_k, T_\text{t}, C_\text{t}^\text{LB})\}$. In other words, given the parameters of the reservations in set $\mathcal{R}_k$ and the minimum inter-replenishment time T_t of the tail reservation, the expressions $V_j(\mathcal{R}_k, T_t, C_\text{t}^\text{LB})$ must be *constant terms*.

First of all, note that the constraint $\sum_{r_i \in \mathcal{R}_k} \frac{C_i}{T_i} + \frac{C_\text{t}}{T_\text{t}} \leq 1$ (corresponding to a very simple necessary condition for feasibility) originates a trivial upper bound on the value of C_t, that is

$$C_\text{t} \leq C_\text{t}^\text{MAX} = \left(1 - \sum_{r_i \in \mathcal{R}_k} U_i\right) T_\text{t}. \tag{5}$$

Leveraging the bound C_t^MAX, the terms $V_j(\mathcal{R}, T_t, C_\text{t}^\text{LB})$ can be derived by considering the constraints originated by the PDC check-points in the set $\bigcup_{r_i \in \{\mathcal{R}_k \cup r_t\}} \xi(r_i)$. First, note that functions $\overline{dbf}_i(t)$ are piece-wise defined in intervals that depend on the check-point t. Also, by looking at Equation 4, observe that the check-points of the tail reservation depend on the optimization variable $C_\text{t} = D_\text{t}$.

Therefore, when considering any of the check-points t of the tail reservations (i.e., those in the set $\xi(r_t)$), the value of functions $\overline{dbf}_i(t)$ for the other reservations cannot be expressed in a closed form as their value depend on C_t, which is unknown. This issue introduces a sort of circular dependency in the equations that is solved via approximations by the following lemma.

Lemma 1. *If the conditions*

$$\begin{cases} C_t \leq min_{r_i \in \mathcal{R}_k}\{D_i\} - \epsilon & (a) \\ \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(jT_\text{t} + C_\text{t}^\text{MAX}) + (j+1)C_t \leq jT_t + D_t, \\ \text{for } \ j = 1, \ldots, \nu_i & (b) \end{cases} \tag{6}$$

hold (with $\epsilon > 0$ arbitrary small), then

$$\forall t \in \xi(r_t), \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf}_t(t) \leq t. \tag{7}$$

Proof. Each of the conditions above corresponds to one element of the set $\xi(r_t)$. Condition (a) verifies the constraint $\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(D_\text{t}) + \overline{dbf}_t(D_\text{t}) \leq D_\text{t}$. If the tail reservation (configured with $C_\text{t} = D_\text{t}$) does not have the smallest deadline among the reservations allocated to P_k, then it may be preempted, thus inevitably missing its deadline. Therefore, a solution exists only if $C_\text{t} = D_\text{t} < min_{r_i \in \mathcal{R}_k}(D_i)$, which then gives $\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(D_\text{t}) = 0$ and the constraint for point D_t is implicitly verified. Condition (b) verifies the constraint $\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf}_t(t) \leq t$ for points $t = jT_t + D_t$ with $j = 1, \ldots, \nu_i$. Since functions $\overline{dbf}_i(t)$ are monotonic non-decreasing in t and $C_\text{t} \leq C_\text{t}^\text{MAX}$, then $\overline{dbf}_i(jT_\text{t} + D_\text{t}) \leq \overline{dbf}_i(jT_t + C_\text{t}^\text{MAX})$. The lemma follows by noting that, for points $t = jT_t + D_t$, the value of $\overline{dbf}_t(t)$ corresponds to $(j+1)C_\text{t}$. □

Before proceeding with the constraints originated by the check-points of the head and partitioned reservations, it is

TABLE 2
List of Terms $V_j(\mathcal{R}_k, T_\mathrm{t}, C_\mathrm{t}^{\mathrm{LB}})$ $(j = 0, \ldots, J)$ for Theorem 2, Where $J = |\bigcup_{\{r_i \in \mathcal{R}_k} \xi(r_i)| + |\xi(r_t|)$

Constraint for point $t = D_\mathrm{t}$
$V_0 = min\left\{C_\mathrm{t}^{\mathrm{MAX}}, min_{r_i \in \mathcal{R}_k}\{D_i\} - \epsilon\right\}$
Constraints for points $t = sT_\mathrm{t} + D_\mathrm{t}$, $s = 1, \ldots, \nu_t$
$V_s(\mathcal{R}_k, T_\mathrm{t}) = T_\mathrm{t} - \frac{1}{s}\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i\left(sT_\mathrm{t} + C_\mathrm{t}^{\mathrm{MAX}}\right)$
Constraints for points $t = sT_\mathrm{i} + D_i, s = 0, \ldots, \nu_i, \forall r_i \in \mathcal{R}_k$ **with** $i = 1, \ldots, n_k$
$V_{z+s}(\mathcal{R}_k, T_\mathrm{t}, C_\mathrm{t}^{\mathrm{LB}}) = \begin{cases} \frac{1}{j+1}(t - \sum_{r_a \in \mathcal{R}_k} \overline{dbf}_a(t)) & \text{if } jT_\mathrm{t} + C_\mathrm{t}^{\mathrm{LB}} \le t < (j+1)T_\mathrm{t} + C_\mathrm{t}^{\mathrm{LB}}, \\ & j = 0, \ldots, \nu_t - 1 \\ \frac{T_\mathrm{t}}{t + T_\mathrm{t} - C_\mathrm{t}^{\mathrm{LB}}}\left(t - \sum_{r_a \in \mathcal{R}_k} \overline{dbf}_a(t)\right) & \text{if } t \ge \nu_t T_\mathrm{t} + C_\mathrm{t}^{\mathrm{LB}}, \end{cases}$
where $z = (\nu_t + 1) + \sum_{x=1}^{i-1}(\nu_x + 1)$

necessary to introduce a new demand bound function $\overline{\overline{dbf}}_t(t)$, which is explicitly conceived to deal with the contribution originated by the tail reservation. This function is illustrated in Fig. 2b and allows removing the circular dependency that would have been introduced by the use of $\overline{dbf}_t(t)$, which again depends on the value of the (unknown) optimization variable $D_\mathrm{t} = C_\mathrm{t}$.

Lemma 2. *It holds*

$$\forall t \ge 0,\ \overline{\overline{dbf}}_t(t, C_\mathrm{t}^{\mathrm{LB}}) \ge \overline{dbf}_t(t),$$

where

$$\overline{\overline{dbf}}_t(t, C_\mathrm{t}^{\mathrm{LB}}) = \begin{cases} 0 & \text{if } t < C_\mathrm{t}^{\mathrm{LB}} \\ (j+1)C_t & \text{if } t \ge C_\mathrm{t}^{\mathrm{LB}} + jT_t \wedge \\ & t < C_\mathrm{t}^{\mathrm{LB}} + (j+1)T_t \\ C_t + U_t(t - C_\mathrm{t}^{\mathrm{LB}}) & \text{if } t \ge \nu_t T_t + C_\mathrm{t}^{\mathrm{LB}}, \end{cases} \tag{8}$$

for $j = 0, \ldots, \nu_t - 1$, and $C_\mathrm{t}^{\mathrm{LB}}$ is a lower bound to C_t.

Proof. Let us separately consider the three cases in which $\overline{\overline{dbf}}_t(t, C_\mathrm{t}^{LB})$ is defined.

Case $t < C_\mathrm{t}^{\mathrm{LB}}$. As $\overline{dbf}_t(t) = 0$ for $t < C_\mathrm{t} = D_\mathrm{t}$, being $C_\mathrm{t}^{\mathrm{LB}} \le C_\mathrm{t}$, then also $\overline{\overline{dbf}}_t(t, C_\mathrm{t}^{\mathrm{LB}}) = 0$.

Case $C_\mathrm{t}^{\mathrm{LB}} + jT_t \le t < C_\mathrm{t}^{\mathrm{LB}} + (j+1)T_t$, $(j = 0, \ldots, \nu_t - 1)$. Being $C_\mathrm{t}^{\mathrm{LB}} \le C_\mathrm{t} = D_\mathrm{t}$, in this case $\overline{dbf}_t(t)$ can be either equal to jC_t, when $t < D_t + jT_t$, or $(j+1)C_\mathrm{t}$, when $t \ge D_t + jT_t$. Hence, $\overline{dbf}_t(t)$ is always upper-bounded by $(j+1)C_\mathrm{t}$.

Case $t \ge \nu_t T_t + C_\mathrm{t}^{\mathrm{LB}}$. Analogously as for the previous case, in this one $\overline{dbf}_t(t)$ can be either equal to **(i)** $\nu_t C_t$, when $t < \nu_t T_t + D_\mathrm{t}$, or **(ii)** $C_t + U_t(t - D_\mathrm{t})$, when $t \ge \nu_t T_t + D_\mathrm{t}$. Note that in both the sub-cases $\overline{dbf}_t(t)$ is upper-bounded by $C_t + U_t(t - D_\mathrm{t})$. Being, $C_\mathrm{t}^{\mathrm{LB}} \le C_\mathrm{t} = D_\mathrm{t}$, it holds $C_t + U_t(t - C_\mathrm{t}^{\mathrm{LB}}) \ge C_t + U_t(t - D_\mathrm{t})$.

Hence the lemma follows. □

Thanks to this upper bound, it is now possible to remove the circular dependency in the constraints originated by the check-points of the head and partitioned reservations.

Lemma 3. *If the inequality*

$$\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{\overline{dbf}}_t(t, C_\mathrm{t}^{LB}) \le t, \tag{9}$$

holds $\forall t \in \{\bigcup_{r_i \in \mathcal{R}_k} \xi(r_i)\}$, then

$$\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf}_t(t) \le t,$$

also holds $\forall t \in \{\bigcup_{r_i \in \mathcal{R}_k} \xi(r_i)\}$.

Proof. The lemma directly follows from Lemma 2 and the definition of the set $\xi(r_i)$. □

Finally, the results of Lemma 1 and Lemma 3 are combined in the following theorem, which provides a *closed-form expression* for computing a safe bound on C_t.

Theorem 2. *A set of reservations $\{\mathcal{R}_k \cup r_t\}$ composed of n_k partitioned and/or head reservations, and one tail reservation r_t with minimum inter-replenishment time T_t, can be safely EDF-scheduled on a single processor P_k if*

$$C_\mathrm{t} = D_\mathrm{t} = min_{j=0,\ldots,J}\left\{V_j(\mathcal{R}_k, T_\mathrm{t}, C_\mathrm{t}^{\mathrm{LB}})\right\},$$

where $V_0(\mathcal{R}_k, T_\mathrm{t}, C_\mathrm{t}^{\mathrm{LB}}), \ldots, V_J(\mathcal{R}_k, T_\mathrm{t}, C_\mathrm{t}^{\mathrm{LB}})$ are defined as in Table 2, and $0 \le C_\mathrm{t}^{\mathrm{LB}} \le C_\mathrm{t}$.

Proof. The set of reservations $\{\mathcal{R}_k \cup r_t\}$ is schedulable if the conditions of Theorem 1 hold. Note that Lemma 1 and Lemma 3 can be combined to obtain sufficient conditions for which Theorem 1 holds. The terms in Table 2 are obtained by simple algebraic transformations of the conditions of such lemmas, which have been reformulated in the form $\forall j = 0, \ldots, J,\ C_\mathrm{t} \le V_j(\mathcal{R}_k, T_\mathrm{t}, C_\mathrm{t}^{\mathrm{LB}})$. Specifically, each check-point in the set $\bigcup_{r_i \in \{\mathcal{R}_k \cup r_t\}} \xi(r_i)$ originates a constraint $V_j(\mathcal{R}_k, T_\mathrm{t}, C_\mathrm{t}^{\mathrm{LB}})$. All of such constraints are verified if $C_\mathrm{t} = min_{j=0,\ldots,J}\{V_j(\mathcal{R}_k, T_\mathrm{t}, C_\mathrm{t}^{\mathrm{LB}})\}$. The algebraic transformations to obtain such constraints are the following.

Constraint for Point $t = D_\mathrm{t}$.It directly follows from Equation (5) and condition (a) in Lemma 1.

Constraints for Points $t = sT_\mathrm{t} + D_\mathrm{t}$: Following condition (b) of Lemma 1, the inequality

$$\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(sT_\mathrm{t} + C_\mathrm{t}^{\mathrm{MAX}}) + (s+1)C_t \le sT_t + D_t$$

needs to be satisfied. Recalling that $C_\mathrm{t} = D_\mathrm{t}$ and solving with respect to C_t, the inequality can be rewritten as

$$sC_\mathrm{t} \le -\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(sT_\mathrm{t} + C_\mathrm{t}^{\mathrm{MAX}}) + sT_t,$$

1: **procedure** APPROXIMATEC=D($\mathcal{R}_k, T_t, \lambda$)
2: $C_{\mathrm{t}}^{\mathrm{LB}} \leftarrow 0$;
3: **for** $i = 1, \ldots, \lambda + 1$ **do**
4: Compute constraints $V_j(\mathcal{R}_k, T_t, C_{\mathrm{t}}^{\mathrm{LB}})$ by Table 2
5: $C_{\mathrm{t}}^{\mathrm{LB}} \leftarrow \min_{j=0,\ldots,J} \{V_j(\mathcal{R}_k, T_t, C_{\mathrm{t}}^{\mathrm{LB}})\}$;
6: **end for**
7: **return** $C_{\mathrm{t}}^{\mathrm{LB}}$;
8: **end procedure**

Fig. 3. Pseudo-code for computing a lower bound for C_{t} with iterative refinement.

and then as,

$$C_{\mathrm{t}} \leq T_t - \frac{1}{s} \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(sT_{\mathrm{t}} + C_{\mathrm{t}}^{\mathrm{MAX}}).$$

Constraints for Points $t = sT_i + D_i$.By Lemma 3, the following inequality needs to be satisfied:

$$\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{\overline{dbf}}_t(t) \leq t.$$

The three cases in which $\overline{\overline{dbf}}_t(t, C_{\mathrm{t}}^{LB})$ is defined are individually discussed next.

Case $t < C_{\mathrm{t}}^{\mathrm{LB}}$. Since $\overline{\overline{dbf}}_t(t, C_{\mathrm{t}}^{LB}) = 0$, no constraint is needed in this interval.

Case $jT_t + C_{\mathrm{t}}^{\mathrm{LB}} \leq t < (j+1)T_{\mathrm{t}} + C_{\mathrm{t}}^{\mathrm{LB}},\ j = 0, \ldots, \nu_t - 1$. Since $\overline{\overline{dbf}}_t(t, C_{\mathrm{t}}^{LB}) = (j+1)C_{\mathrm{t}}$, the inequality becomes

$$\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + (j+1)C_{\mathrm{t}} \leq t,$$

which by solving with respect to $C_{\mathrm{t}} = D_{\mathrm{t}}$ can be rewritten as

$$D_{\mathrm{t}} \leq \frac{1}{j+1} \left(t - \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) \right).$$

Case $t \geq \nu_t T_{\mathrm{t}} + C_{\mathrm{t}}^{\mathrm{LB}}$. Since $\overline{\overline{dbf}}_t(t, C_{\mathrm{t}}^{LB}) = C_{\mathrm{t}} + U_t(t - C_{\mathrm{t}}^{\mathrm{LB}})$, the inequality becomes

$$\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + C_{\mathrm{t}} + U_t(t - C_{\mathrm{t}}^{\mathrm{LB}}) \leq t.$$

Recalling that $U_t = C_{\mathrm{t}}/T_{\mathrm{t}}$ and by solving with respect to C_{t}, the inequality can be rewritten as

$$C_{\mathrm{t}} \left(\frac{T_{\mathrm{t}} + t - C_{\mathrm{t}}^{\mathrm{LB}}}{T_{\mathrm{t}}} \right) \leq t - \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t),$$

and then as

$$C_{\mathrm{t}} \leq \frac{T_{\mathrm{t}}}{T_{\mathrm{t}} + t - C_{\mathrm{t}}^{\mathrm{LB}}} \left(t - \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) \right).$$

The right-hand sides of such inequalities are the terms reported in Table 2. Hence the theorem follows. □

With Theorem 2 in place, an algorithm to compute a *refined* bound for the zero-laxity budget C_{t} is proposed next.

3.2 Algorithm for Approximate C=D

The results of Theorem 2 can be used to implement an algorithm that efficiently computes a safe value for C_{t} via iterative refinements, which is is reported in Fig. 3.

The algorithm inputs the set $\mathcal{R}_k$ of reservations already allocated to processor P_k, the period T_t of the tail reservation to be allocated on P_k, and a number λ of refinement iterations. As a first step, the lower bound $C_{\mathrm{t}}^{\mathrm{LB}}$ is initialized to zero, which is clearly a safe value. Then, at each iteration, the constraints $V_j(\mathcal{R}_k, T_t, C_{\mathrm{t}}^{\mathrm{LB}})$ of Table 2 are computed and a new lower bound for C_{t} is obtained by leveraging Theorem 2. This lower bound can then be used as a new value for $C_{\mathrm{t}}^{\mathrm{LB}}$ to further refine the bound provided by Theorem 2, which is monotone non-decreasing with $C_{\mathrm{t}}^{\mathrm{LB}}$. Hence, the algorithm also generates a non-decreasing sequence of lower bounds for C_{t}. Surprisingly, the experiments reported in Section 5.1 show that just two refinement iterations ($\lambda = 2$) provide a significant improvement with respect to the adoption of Theorem 2 with $C_{\mathrm{t}}^{\mathrm{LB}} = 0$ ($\lambda = 0$, no bound refinement).

3.3 Implementation and Complexity

In this work, the methods proposed in the previous sections were derived to be used *on-line* for admitting a new reservation by means of C=D splitting. Therefore, considering the case in which a set of reservations $\mathcal{R}_k$ is already allocated to P_k, the value of C_{t} has to be computed for evaluating the possibility of allocating a tail reservation to P_k. In this case, the approach presented in Section 3.1 allows implementing a *linear-time* algorithm for computing the C=D splitting. In fact, all the terms in the constraints $C_{\mathrm{t}} \leq V_j(\mathcal{R}_k, T_{\mathrm{t}}, C_{\mathrm{t}}^{\mathrm{LB}})$ (see Table 2) that *do not depend* on T_{t} can be pre-computed and stored in a table each time a reservation (partitioned or head) is allocated to P_k (e.g., as using dynamic programming): this operation can be done in $\mathcal{O}(\sum_{r_i \in \{\mathcal{R}_k \cup r_t\}}(\nu_i + 1)) = \mathcal{O}(n_k)$ time, which is linear in the number of tasks as soon ν_i is constant $\forall r_i \in \{\mathcal{R}_k \cup r_t\}$. Then, to implement Theorem 2, it is required to compute (i) the upper bound $C_{\mathrm{t}}^{\mathrm{MAX}}$, which can be done in constant time, (ii) the sum of demand bound functions in $V_1(\mathcal{R}_k, T_{\mathrm{t}}, C_{\mathrm{t}}^{\mathrm{LB}}), \ldots, V_{\nu_t}(\mathcal{R}_k, T_{\mathrm{t}}, C_{\mathrm{t}}^{\mathrm{LB}})$, which can be done in $\mathcal{O}(n_k)$ time, and (iii) the minimum among the constraints, which can be done in $\mathcal{O}(n_k)$ time.

The APPROXIMATEC=D algorithm computes the bound of Theorem 2 for $\lambda + 1$ times. Hence, the algorithm has complexity $\mathcal{O}((\lambda + 1)n_k)$, which is again linear until λ is constant.

4 LOAD BALANCING

This section presents a load balancing algorithm for managing the allocation and the splitting of the reservations under C=D semi-partitioned scheduling. The algorithm has been designed to be *as simple as possible* (to be practically used online) and employs a limited number of re-allocations of the reservations. At a high level, the algorithm reacts to two events: (i) the *arrival* of a new reservation, where its *admission* must be evaluated by finding a proper allocation; and (ii) the *exit* of a reservation, which consists in performing some re-allocations in order to favor the admission of future reservations.

The following two sections discuss how to handle the arrival and the exit of a reservation. Then, Section 4.3 discusses how it is possible to increase the performance of the load balancing algorithm, at the cost of performing a single

1: **procedure** MULTISPLIT(r_i, λ)
2: Let **S** be an ordered list of elements $(S_k.C, S_k.P)$
3: $\forall P_k$, $(k = 1, \ldots, m)$
4: $S_k.C \leftarrow$ APPROXIMATEC=D($\mathcal{R}_k, T_i, \lambda$)
5: $S_k.P \leftarrow P_k$
6: Sort **S** according to $S_k.C$ in decreasing order
7: $x \leftarrow \max\left\{x \mid \sum_{j=1}^{x} S_j.C < C_i \ \wedge \ x < m\right\}$
8: **for all** $j = 1, \ldots, x$ **do**
9: $r_t^j \leftarrow (S_j.C, S_j.C, T_i)$
10: Allocate r_t^j to $S_j.P$
11: **end for**
12: $C_h = C_i - \sum_{j=1}^{x} S_j.C$
13: $D_h = D_i - \sum_{j=1}^{x} S_j.C$
14: $r_h \leftarrow (C_h, D_h, T_i)$
15: Try to allocate r_h on a processor with a partitioning heuristic
16: **if** r_h has been allocated **then**
17: **return** SUCCESS
18: **else**
19: $\forall j = 1, \ldots, x$, remove r_t^j from $S_j.P$
20: **return** FAIL
21: **end if**
22: **end procedure**

Fig. 4. Pseudo-code of the algorithm for splitting and allocating reservations.

reallocation of an already-partitioned reservation. Finally, Section 4.4 discusses how to handle the scheduling transients originated by a reservation that leaves the system or that is reconfigured.

4.1 Admission of a New Reservation

Whenever the system receives a request for admitting a new reservation $r_i = (C_i, D_i, T_i)$, the following operations are performed:

1) First, the algorithm tries to find a static allocation of r_i to a processor (i.e., as with standard partitioned scheduling) by using a partitioning heuristic. In particular, according to our experiments, the *best-fit* heuristic has been found to perform best. If a valid allocation is found, then r_i is admitted into the system.
2) If step 1) fails, then r_i is split into a head reservation r_h and a number tail reservations $r_t^1, \ldots, r_t^x$, with $x < m$, by the algorithm reported in Fig. 4. The algorithm works as follows. First, for each processor P_k, a safe bound on the maximum budget of a tail reservation allocated to P_k is computed by means of the APPROXIMATEC=D algorithm (line 4). The pairs $(S_k.C, S_k.P)$ of budgets and processors are recorded in an ordered list **S**. Then, **S** is sorted in descending order with respect to the budget of each pair. Then, the maximum index x is computed such that the sum of the largest x elements in **S** does not exceed the budget C_i of the reservation to be split. Subsequently, for each of the x first elements in **S**, a tail reservation r_t^j is defined with budget and deadline $S_j.C$, period T_i (line 9), and allocated to processor $S_j.P$ (line 10), with $j = 1, \ldots, x$. The same partitioning heuristic used in step 1) is then applied to allocate the head reservation r_h: if the allocation succeeds, then the algorithm also succeeds; otherwise the algorithm fails and the reservation r_i is rejected.

Note that both the steps require evaluating whether a reservation can be safely allocated to a processor, which can be performed by leveraging Theorem 1 with computational cost $\mathcal{O}(n_k)$. As discussed in Section 3.3, algorithm APPROXIMATEC=D has $\mathcal{O}(n_k)$ complexity. Hence, the overall computational cost of the above operations is $\mathcal{O}(m \cdot n^{\text{MAX}})$, where $n^{\text{MAX}} = \max_{k=1,\ldots,m}\{n_k\}$. Note that this is the same complexity of the approach described in [14].

4.2 Handling the Exit of a Reservation

Whenever a partitioned reservation $r_i \in \mathcal{R}_k^P$ (i.e., allocated to processor P_k) leaves the system, if $tail(P_k) = true$ let $r_{t,k}$ be the (only) tail reservation allocated on P_k, and let $r_j = \mathcal{F}(r_{t,k})$. Then, the algorithm tries to allocate r_j to P_k after removing all head and tail reservations related to r_j from the corresponding processors. That is, the algorithm tries to re-assemble the semi-partitioned reservation r_j by allocating it as a partitioned reservation of P_k. Conversely, if $tail(P_k) = false$ but $head(P_k) = true$, let $r_{h,k}$ be head reservation with the highest utilization allocated on P_k. Then, the algorithm tries to re-assemble $r_j = \mathcal{F}(r_{h,k})$ on P_k as above. Whenever a semi-partitioned reservation $r_i \in \mathcal{R} \setminus \mathcal{R}_k^P$ leaves the system, let $\mathcal{P}(r_i)$ be the set of processors in which at least a tail or head reservation of r_i was allocated. Then, for each processor $P_k \in \mathcal{P}(r_i)$, the same procedure described above for the exit of a partitioned reservation is performed. These operations require checking at most m times (the maximum number of splits) whether a reservation can be allocated to a processor, which can be performed in $\mathcal{O}(m \cdot n^{\text{MAX}})$ time.

4.3 Re-Allocate Partitioned Reservations

Whenever the algorithm does not find a valid allocation for a new reservation r_i, the chances of admitting r_i can be increased by trying to re-allocate a previously-allocated partitioned reservation. In particular, the following heuristic has been found to be effective while employing minimal re-allocations limited to a *single* reservation.

For each processor P_k $(k = 1, \ldots, m)$, check if after de-allocating the partitioned reservation $r_j \in \mathcal{R}_k^P$ that has the highest utilization (i.e., $r_j \in \mathcal{R}_k^P \mid U_j = \max_{r_x \in \mathcal{R}_k^P} U_x$) it is possible to allocate r_i to P_k. If yes, then try to re-allocate r_j by following steps 1 and 2 in Section 4.1. When the first valid re-allocation is found, r_j is re-allocated, r_i is allocated to P_k, and the algorithm terminates. The computational complexity of this extension is $\mathcal{O}(m^2 \cdot n^{\text{MAX}})$ complexity.

4.4 Handling Scheduling Transients

It is worth observing that the admission of a new reservation may not be immediately performed when another reservation leaves the system or it is reconfigured during a re-allocation (so freeing some utilization bandwidth). This is because the leaving (or modified) reservation may have already affected the execution of the other reservations, and hence the system is subject to a *transient* (also referred to as mode-change by some authors). However, note that this issue is not specifically related to semi-partitioned scheduling, as it also occurs in uniprocessor systems [20], [21] (and hence under partitioned scheduling) and under global scheduling [22], [23]. Several solutions are available for analyzing such transients [20], [22], [24] by deriving a safe

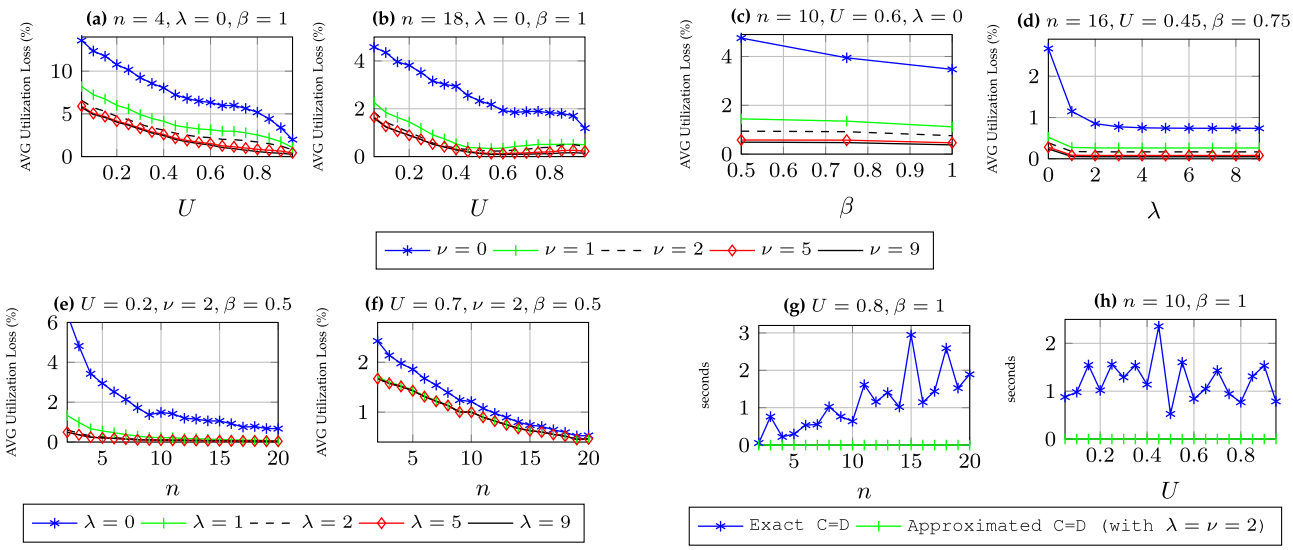

Fig. 5. Average utilization loss introduced by the approximate algorithms for C=D splitting (presented in Section 3) as a function of the task-set utilization (insets (a) and (b)), the parameters β and λ (insets (c) and (d)), the number of reservations n (insets (e) and (f)). Insets (g) and (h) evaluates the running times of the proposed methods against the one of the exact C=D splitting algorithm.

bound on the time that must be waited before admitting a new reservation or let re-allocations to take effect.

The design of efficient methods to handle scheduling transients that are tailored to C=D semi-partitioned scheduling is out of the scope of this paper and is left as future work. This section illustrates how to apply state-of-the-art results for partitioned scheduling to handle scheduling transients in the setting considered in this work.

When deadline misses cannot be tolerated, it is necessary to ensure that a newly admitted (or re-allocated) reservation is subject to an *admission delay*. For example, in the case of partitioned reservations, the AADT algorithm [24] (and the related protocol) may be used to compute such an admission delay in polynomial time. Nevertheless, it is designed for uniprocessor (or multiprocessor partitioned) scheduling, and hence it cannot be directly applied to semi-partitioned reservations. To fill this gap, Lemma 4 establishes a safe time to admit a semi-partitioned reservation without incurring scheduling transients.

Lemma 4. *Let r_i be a semi-partitioned reservation that has been admitted in the system at time t and split into $x < m$ sub-reservations $r_i^1, \ldots r_i^x$. Also, let $\mathcal{B}$ be the set of processors in which the sub-reservations of r_i are allocated to, i.e., $\mathcal{B} = \{P_k \mid \exists r_i^j \in \mathcal{R}_k \text{ s.t. } j = 1, \ldots, x \wedge k = 1, \ldots, m\}$.*

Then, r_i can be admitted without incurring scheduling transients at time

$$t + \max_{j=1,\ldots,x} \{d_i^j\}, \tag{10}$$

where d_j^i is a safe admission delay for sub-reservation r_i^j.

Proof. Reservation r_i does not incur in scheduling transients if each sub-reservation waits for its corresponding admission delay. By definition, each delay d_j^i is a safe time for admitting the sub-reservation r_i^j in the corresponding processor. Hence, waiting for the maximum admission delay among all sub-reservation provides a safe condition for admitting r_i. □

Lemma 4 provides a simple and safe way to extend state-of-the-art results for dealing with scheduling transients to admit reservations that require to be split. Note that Lemma 4 can also be used in conjunction with other methods providing safe admission delays for each of the sub-reservations (i.e., working under partitioned scheduling).

Nonetheless, waiting for an admission delay may not be always acceptable. In these circumstances, a simple and effective strategy consists in allowing the allocation of a reservation r_i only if the corresponding admission delay is zero. For example, this method may be suitable for reservations that need to be re-allocated in a different processor to favor the admission of new workloads.

5 Experimental Results

This section presents the results of two large-scale experimental studies that have been conducted to evaluate the approach presented in this paper. The first study, discussed in Section 5.1, has been carried out to assess the performance of the approximate C=D splitting algorithm presented in Section 3 with respect to the exact algorithm proposed by Burns *et al.* in [13]. Furthermore, the space of parameters ν_i and λ (see Section 3) has been explored to determine suitable configurations to be used at run time. The second study, discussed in Section 5.2, has been carried out to evaluate the performance of the load balancing algorithms presented in Section 4 (adopted in conjunction with the C=D splitting algorithm of Section 3), comparing them to G-EDF and partitioned EDF scheduling under different settings.

5.1 C=D Splitting: Approximate Versus Exact

The objective of this experimental study is to evaluate the utilization loss introduced by the approximate C=D splitting algorithm presented in Section 3 with respect to the exact Burns *et al.*'s [13] method. Specifically, the study considers a single processor on which a set of reservations is already allocated and is based on computing with the two methods the maximum zero-laxity budget of a tail reservation to be allocated on the considered processor.

Reservation Set Generation. Given n reservations and a target utilization $U = \sum_{i=1}^{n} U_i$, the individual utilizations U_i of the n reservations are generated with the UUnifast algorithm [25]. For each reservation, the minimum inter-replenishment time T_i is randomly generated in the range [1, 1000] ms with uniform distribution and the budget is then computed as $C_i = U_i T_i$. The relative deadline of each reservation r_i is then randomly generated with uniform distribution in the interval $[C_i + \beta(T_i - C_i), T_i]$, with $\beta \in [0, 1]$. Intuitively, the use of larger values for β tend to generate deadlines closer to the corresponding periods, where $\beta = 1$ implies $D_i = T_i$.

Experiments. The utilization U has been varied in the range $[0.05, 0.95]$ with step 0.05, and the number n of reservations has been varied from 2 to 20.[2]The number of additional steps ν_i of the approximate demand bound function of each task and the parameter λ (see Algorithm 3) have been varied in the interval $[0, 9]$ with step one. For the sake of simplicity, the parameter ν_i is set to the same value ν for all the tasks. The parameter β, which controls the relative deadline of the tasks, has been varied in the set $\{0.5, 0.75, 1\}$. For each combination of these parameters, 5000 reservation sets have been tested, for a total of almost 600 million reservation sets. For each reservation set $\mathcal{R}$, the period T_t of a tail reservation r_t has been randomly generated in the range [1, 1000] ms with uniform distribution. Then, the value of C_t such that the set of reservations $\{r_t \cup \mathcal{R}\}$ can be safely EDF-scheduled on a single processor has been computed by both the exact method from [13] and the approximate method proposed in this paper. The two methods have been compared in terms of *utilization loss*: that is, given the exact value C_t^{EXA} (by [13]) and an approximate value $C_t^{\mathrm{APP}} \leq C_t^{\mathrm{EXA}}$, the utilization loss introduced by the approximate method is defined as $\left(C_t^{\mathrm{EXA}}/T_t\right) - \left(C_t^{\mathrm{APP}}/T_t\right)$.

The experimental results for six representative configurations are reported in Fig. 5. Figs. 5a and 5b, show that the improvement provided by increasing the value of ν becomes very small for $\nu \geq 2$, and that the utilization loss decreases as the utilization U increases. Fig. 5c shows that utilization loss slightly decreases as the parameter β increases, and Figs. 5e and 5f illustrate the dependency of the utilization loss on the number of tasks n, which improves as n increases. As for parameter ν, also the improvement achieved by increasing the number of refinement iterations λ becomes smaller for $\lambda \geq 2$. Overall, the results show that configuring the approximate C=D algorithm with $\nu = \lambda = 2$ provides an empirical utilization loss always below 3 percent. This is an important result because such low values for parameters ν and λ determine very short running times of the approximate C=D algorithm.

*Running Times.*Another experiment has been carried out to evaluate the running times of the proposed methods against the one of the exact C=D splitting algorithm. The tests have been executed on a machine equipped with an Intel Core i7-6700K @ 4.00 GHz. The Microsoft VC++2015 compiler has been used to compile literal implementations (i.e., not designed for being extremely efficient) of the algorithms. The approximate C=D algorithm has been configured with $\nu = \lambda = 2$. The running times of both methods have been collected using the Windows API for measuring the wall clock. Despite the experiments having been performed on a dedicated processor, measurements may include some additional overhead (e.g., execution of services of the operating system). A preliminary experiment excluded the possibility of using the API that measures the execution time of the process only, as the offered precision is comparable with the running time of Algorithm 3. As showed in Figs. 5g and 5h, the exact C=D splitting algorithm exhibited maximum running times in the order of a few seconds, with an increasing trend as a function of the utilization and the number of reservations, whereas the running time of the approximate algorithm always resulted below 30 microseconds. The maximum running times of Algorithm 3 showed a slightly increasing trend with respect to the number of tasks, ranging from 16 μs for $n = 2$ tasks, to 29 μs for $n = 20$ tasks.

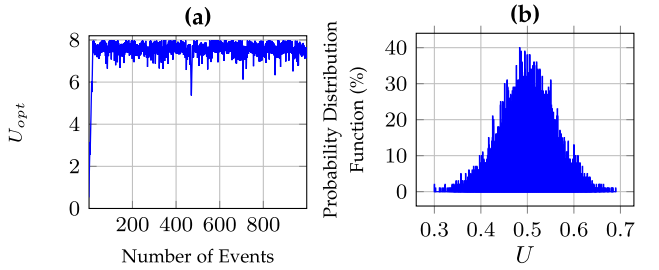

Fig. 6. Analysis of the workload generated with $m = 8, U_{\mathrm{AVG}} = 0.5, U_\sigma = 0.3$, and $\psi = 0.9$. Inset (a) reports U_{opt} as a function of the number of occurred events, whereas inset (b) shows the empirical probability distribution function of the generated utilizations.

5.2 Proposed Approach Versus G-EDF and P-EDF

A second experimental study has been performed to evaluate the performance of C=D semi-partitioned scheduling managed by the load balancing algorithms presented in Section 4 (that make use of the approximate splitting algorithm of Section 3) against G-EDF and partitioned EDF (P-EDF) scheduling. For G-EDF scheduling, a relatively favorable condition has been considered in which the acceptance test is performed by combining four state-of-the-art polynomial-time tests (suitable for being executed on-line), which are: GFB [26], BAK [27], a polynomial-time approximation of LOAD [28], [29], and I-BCL [30] (configured with 3 iterations, as suggested by the authors). In other words, if *any* of these tests is passed, then a new reservation is admitted. For P-EDF, three common partitioning heuristics have been tested: first-fit, best-fit, and worst-fit (the latter with respect to the utilizations of the reservations). The study is based on synthetic dynamic workloads, which have been generated as follows.

Generation of Dynamic Workload. A *sequence* of N^E events is generated, where each event can be of type `ARRIVAL` or `EXIT`. An `ARRIVAL` event consists in a new reservation r_i that is tried to be admitted into the system. The *beta distribution* [31] has been adopted to control the statistical validity

2. In the special case of a single reservation ($n = 1$), the computation of the *exact* maximum zero-laxity budget that can be safely allocated to a processor can be computed by solving a simple equation (the details are available in Appendix A). The number of tasks has been limited to 20 because the results show that the error introduced by the proposed approximation decreases as the number of tasks increases, approaching very low values for more than 20 tasks.

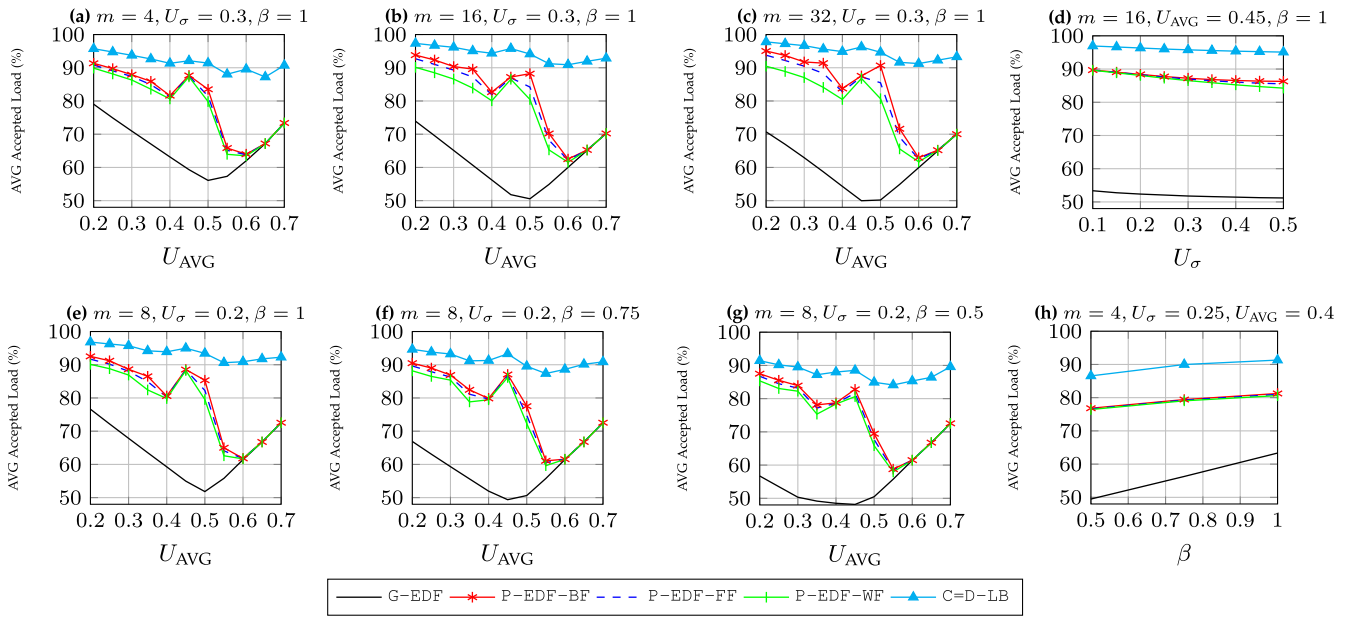

Fig. 7. Average accepted load obtained by different scheduling approaches as a function of the average utilization U_{AVG} (insets (a), (b), (c), (e), (f) and (g)), U_σ (inset (d)), and β (inset (h)). The results are related to eight representative configurations identified by the fixed parameters reported in the caption above the graphs.

of the utilizations of the reservations, generating the utilization values in a fixed range $[U_{\min}, U_{\max}] = [0.01, 0.9]$. In each experiment, the beta distribution has been configured with two parameters U_{AVG} and U_σ, controlling the average and the variance of the generation, respectively. The minimum inter-replenishment time T_i of each reservation was generated in the range [1, 1000] ms with uniform distribution, and the budget was then computed as $C_i = U_i T_i$. As in Section 5.1, the relative deadline of each reservation r_i has been randomly generated with uniform distribution in the interval $[C_i + \beta(T_i - C_i), T_i]$, with $\beta \in [0, 1]$.

The `EXIT` event corresponds to the exit of a reservation that is *randomly* selected among those that are currently admitted in the system. Each sequence s of events is generated as follows: a random real number $x \in [0, 1]$ is generated N^E times with uniform distribution; each time, if $x \in [0, \Lambda]$, an `ARRIVAL` event is generated and enqueued to s, else an `EXIT` event is generated and enqueued to s. The term Λ is a variable *threshold* that controls the generation and has been set to $\Lambda = (1 - U_{opt}/m) + \psi(U_{opt}/m)$ with the following interpretation. The parameter U_{opt} is the current utilization that the system would have if an *optimal* scheduling algorithm would have been used to process the previously-generated events. The first term in the definition of Λ is provided to increase the probability of generating an `ARRIVAL` event when the system load is low. The second term depends on a parameter $\psi \in [0, 1]$, which is used to control the tendency of a sequence to load the processors; i.e., the larger ψ the larger the average load demanded by a sequence. Fig. 6 reports some details about the generated workload. Inset (a) shows the utilization U_{opt} provided by the generator as a function of the number of occurred events, when the generator is configured with $m = 8, U_{\text{AVG}} = 0.5, U_\sigma = 0.3$, and $\psi = 0.9$. Under this configuration, the generator is able to maintain a fairly high U_{opt}, and hence is capable of properly loading the system to stimulate the tested methods. Inset (b) corroborates the statistical validity of the generated utilizations by showing the *empirical* probability distribution function of the generated utilizations in a sequence of 10000 events.

Experiments. The average utilization U_{AVG} of the generated reservations has been varied in the range [0.2, 0.7] with step 0.05, whereas the variance U_σ has been varied in the range [0.10, 0.50], with step 0.05. The parameter β, which regulates the relative deadlines, has been varied in the set $\{0.5, 0.75, 1\}$. The number of processors m has been varied in the set {4, 8, 16, 32} and the parameter ψ in the set {0.6, 0.7, 0.8, 0.9}. For each combination of the varied parameters, 1000 sequences of 10000 events have been generated. Each generated sequence has been tested with G-EDF, P-EDF, and the approaches proposed in this paper, measuring the *average* load accepted by each algorithm across the whole sequence. This measure is subsequently normalized to the hypothetical average load that would have been accepted by an optimal scheduling algorithm. This index expresses the quality of an algorithm in terms of acceptance rate (the higher the better and 100 percent corresponds to the performance of an optimal algorithm).[3]

Fig. 7 reports the results for eight representative configurations with $\psi = 0.9$. The labels `P-EDF-FF`, `P-EDF-WF`, and `P-EDF-BF` in the legend indicate first-fit, worst-fit, and best-fit partitioning, respectively; `C=D-LB` indicates the proposed approach based on load balancing presented in Section 4 configured with $\nu = \lambda = 2$. As it can be observed from Figs. 7a, 7b, 7c, 7d, 7e, 7f, 7g, the performance of the algorithms is significantly affected by the utilization of the tested reservations (as also previously observed in other

3. Note that the typical schedulability ratio metric makes little sense in the presence of dynamic workload, as the behavior of the different algorithms may significantly differ depending on the previous workload. For instance, an algorithm may reject a lot of "small" (low utilization) reservations because it previously accepted a "heavy" (high utilization) reservation.

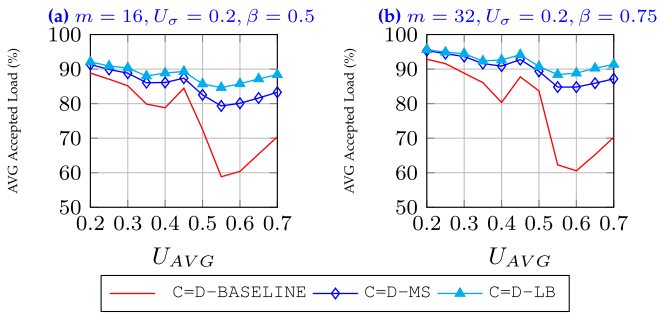

Fig. 8. Average accepted utilization obtained by different semi-partitioned scheduling methods as a function of the average task utilization.

works). The proposed approach allows achieving high performance, keeping the average accepted load above the 87 percent when $D_i = T_i$ (i.e., $\beta = 1$), even in the presence of several reservations with high utilization. In particular, it allows achieving a performance improvement up to 40 and 30 percent over G-EDF and P-EDF, respectively. In the case of constrained deadlines ($\beta = 0.5, 0.75$), the performance of all the various approaches decreases: nevertheless, C=D-LB still allows keeping the average accepted load above 84 percent. The algorithms based on P-EDF show relatively good performance up to values of U_{AVG} that are close to 0.5. Basic partitioned scheduling with simple heuristics has been found to always outperform G-EDF. Fig. 7d shows the dependency on the variance U_σ of the utilizations: the average accepted load slightly decreases as U_σ increases. Finally, Fig. 7h shows the dependency on the parameter β: for all the tested approaches, the average accepted load decreases as β decreases.

It is worth observing that the curves tend to show a non-monotonic behavior for the following reason. Under large values for U_{AVG}, the acceptance or the rejection of a reservation corresponds to a significant difference in terms of instantaneous accepted load. Since this phenomenon also occurs in the case of an optimal scheduling algorithm (to which the performance is normalized to), the processors tend to be less loaded across a sequence, independently of the tested algorithm, which is a situation that favors non-optimal algorithms. The non-monotonic behavior of the performance of G-EDF has been found to also depend on the combination of multiple acceptance tests; in particular, the I-BCL test tends to perform better than the others for larger values of U_{AVG}. Fig. 8 compares different approaches to perform semi-partitioned scheduling of dynamic workloads: `C=D-BASELINE`, i.e., the approaches of Sections 4.1 and 4.2 but limiting semi-partitioned reservations to be split in at most two chunks and without leveraging the re-allocate partitioned reservation extensions discussed in Section 4.3, as in the baseline load-balancing approach of [14]; `C=D-MS`, which extends `C=D-BASELINE` allowing semi-partitioned reservations to be split into multiple chunks (at most m); `C=D-LB`, the complete approach proposed in this paper also considered in Fig. 7. These three approaches represent three different configurations a system designer may want to explore as a trade-off to balance complexity and performance (in terms of average-accepted load). `C=D-BASELINE` and `C=D-MS` have both $\mathcal{O}(m \cdot n^{\text{MAX}})$ complexity for admitting a new reservation, but `C=D-BASELINE` may have a lower run-time overhead due to a lower number of migrations (the lower the number of chunks, the lower the maximum number of migrations), whereas `C=D-LB` has $\mathcal{O}(m^2 \cdot n^{\text{MAX}})$ complexity. As expected, `C=D-LB` reports the highest average accepted load, but also `C=D-MS` shows a good performance, thus representing an interesting compromise when an approach with lower complexity is required.

6 Related Work

The problem of scheduling real-time workload on a multicore platform has been extensively investigated. A detailed discussion of all the results proposed in the literature is too vast to fit in the space available in this paper and readers interested in the topic can refer to the survey written by Davis and Burns [5]. For this reason, this section focuses on techniques based on semi-partitioned scheduling, which are more relevant to the proposed approach. Semi-partitioned scheduling has been first introduced by Anderson *et al.* [32] in 2005. Later, numerous semi-partitioned scheduling algorithms have been presented, including the proposals of Andersson *et al.* [33] and Kato *et al.* [34], [35], [36].

The idea of having split tasks (i.e., reservations) executing at zero-laxity has been originally proposed by Kato and Yamasaki [36] in the context of fixed-priority scheduling, and later extended by Burns *et al.* [13] to EDF, who proposed the C=D scheme on which this paper builds upon. Kato and Yamasaki [36] ensured split reservations to be executed with the highest priority on each processor, thus guaranteeing their budget C to be always consumed within $D = C$ time units from their release time.

In 2011, Bastoni *et al.* [37] presented a thorough comparison of several semi-partitioned scheduling algorithms, illustrating their benefits with respect to other scheduling approaches. The method described in this paper has been motivated by a recent development due to Brandenburg and Gül [12], who showed that, by adopting clever task-allocation heuristics, the C=D splitting algorithm proposed by Burns et al. [13] allows achieving a near-optimal performance in the presence of static real-time workload. As in [12], the proposed approach also combines C=D scheduling with processor reservations, but in a more dynamic environment where reservations can be created and destroyed at runtime. Brandenburg and Gül also reports on a solid evaluation of the overhead introduced by C=D scheduling demonstrating its practical effectiveness. An overhead-aware analysis for semi-partitioned scheduling algorithms has been also proposed by Souto *et al.* [38]. Maia *et al.* [39] considered the problem of applying semi-partitioned scheduling to fork-join tasks on a multicore platform. George *et al.* [40] considered a different approach of semi-partitioned scheduling, where different jobs of the same task can be released on different processors, but each job executes on a single processor only. Very recently, Hobbs *et al.* [41] presented approaches for semi-partitioned scheduling in the context of soft real-time systems, aimed at guaranteeing a bounded tardiness.

The problem of taking online scheduling decisions for real-time workload has been investigated in many works. In particular, the difficulty of the problem has been discussed in the seminal work of Dertouzos and Mok [42] and by Fisher *et al.* [43]. Lee and Shin [23] and Nelis *et al.* [22] proposed techniques for analyzing the effect of system

transients under global scheduling. Block and Anderson [44] and Block *et al.* [45] addressed dynamic workload in the context of task reweighting under partitioned and P-Fair scheduling, respectively.

To the best of our knowledge, this work proposes the first method to perform online admission control under semi-partitioned scheduling and hard deadline constraints.

7 Conclusion and Future Work

This paper proposed methods to enable C=D semi-partitioned scheduling for dynamic workloads consisting of reservation servers. Reservation servers can arbitrarily join and leave the system, but each of them is subject to an admission test before being admitted into the system.

The presented approach allows performing C=D splitting in linear-time, thus drastically reducing the computational complexity with respect to prior (but exact) methods [13] characterized by very high computational complexity.

The approximate C=D splitting method has been then leveraged to design a load balancing algorithm, which allows dynamically allocating and splitting incoming reservations at runtime. A method for extending state-of-art results on scheduling transients to semi-partitioned scheduling has also been discussed.

The contributions have been evaluated with large-scale experimental studies. In particular, the linear-time approximation proposed to split reservations has been shown to originate a very limited (below the 3 percent) utilization loss with respect to the exact technique proposed by Burns *et al.* [13]. The adoption of task splitting and load balancing algorithms to manage dynamic workloads showed a notable schedulability performance, with improvements over global and partitioned EDF up to 40 and 30 percent, respectively. In most of the tested cases, the proposed method allows keeping the average system load above 87 percent, also in the presence of reservations with very high utilizations, which would result difficult to allocate using standard partitioned scheduling.

These results suggest the usage of semi-partitioned C=D scheduling also to handle dynamic workloads.

Interesting research lines for future research include the extension of the proposed approaches to parallel real-time tasks and elastic reservations [20]. Furthermore, developing new protocols (and extending the existing ones) for supporting lock-protected shared resources under C=D semi-partitioned scheduling is also a relevant direction for future work. In particular, the extension of protocols based on bandwidth inheritance among dependent reservations seems to be particularly promising. Another important direction for future extensions is the consideration of heterogeneous processors, where the splitting algorithm is required to account for multiple execution profiles related to different types of processors. Furthermore, additional load-balancing strategies may further improve schedulability.

References

[1] T. Cucinotta, L. Abeni, L. Palopoli, and G. Lipari, "A robust mechanism for adaptive scheduling of multimedia applications," *J. ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 4, pp. 1–24, Nov. 2011.

[2] K. Konstanteli, T. Cucinotta, K. Psychas, and T. Varvarigou, "Admission control for elastic cloud services," in *Proc. 5th Int. Conf. Cloud Comput.*, 2012, pp. 41–48.

[3] "ROS Website," 2020. [Online]. Available: http://www.ros.org/

[4] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-time analysis of ROS 2 processing chains under reservation-based scheduling," in *Proc. 31th Euromicro Conf. Real-Time Syst.*, 2019, pp. 6:1–6:23.

[5] R. 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.

[6] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "Multiprocessor scheduling by reduction to uniprocessor: An original optimal approach," *Real-Time Syst.*, vol. 49, no. 4, pp. 436–474, Nov. 2013.

[7] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic, "U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *Proc. 24th Euromicro Conf. Real-Time Syst.*, 2012, pp. 13–23,.

[8] E. Massa, G. Lima, P. Regnier, G. Levin, andS. Brandt,"Quasi-partitioned scheduling: Optimality and adaptation in multiprocessor real-time systems," *Real-Time Syst.*, vol. 52, no. 5, pp. 566–597, 2016.

[9] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *Proc. 27th Real-Time Syst. Symp.*, 2006, pp. 101–110.

[10] Y. Sun and M. Di Natale, "Pessimism in multicore global schedulability analysis," *J. Syst. Archit.*, vol. 97, pp. 142–152, Aug. 2019.

[11] A. Biondi and Y. Sun, "On the ineffectiveness of 1/m-based interference bounds in the analysis of global EDF and FIFO scheduling," *Real-Time Syst.*, vol. 54, no. 3, pp. 515–536, Jul. 2018.

[12] B. Brandenburg and M. Gül, "Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations," in *Proc. 37th Real-Time Syst. Symp.*, 2016, pp. 99–110.

[13] A. Burns, R. Davis, P. Wang, and F. Zhang, "Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme," *Real-Time Syst.*, vol. 48, pp. 3–33, Jan. 2012.

[14] D. Casini, A. Biondi, and G. Buttazzo, "Semi-partitioned scheduling of dynamic real-time workload: A practical approach based on analysis-driven load balancing," in *Proc. 29th Euromicro Conf. Real-Time Syst.*, 2017, 13:1–13:23.

[15] D. Casini, L. Abeni, A. Biondi, T. Cucinotta, and G. Buttazzo, "Constant bandwidth servers with constrained deadlines," in *Proc. 25th Int. Conf. Real-Time Netw. Syst.*, 2017, pp. 68–77.

[16] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *J. ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, Apr. 2008, Art. no. 1–39.

[17] F. Zhang and A. Burns, "Schedulability analysis for real-time systems with EDF scheduling," *IEEE Trans. Comput.*, vol. 58, no. 9, pp. 1250–1258, Apr. 2009.

[18] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Syst.*, vol. 2, no. 4, pp. 301–324, Oct. 1990.

[19] N. Fisher, T. P. Baker, and S. Baruah, "Algorithms for determining the demand-based load of a sporadic task system," in *Proc. 12th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2006, pp. 135–146.

[20] G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management," *IEEE Trans. Comput.*, vol. 51, no. 3, pp. 289–302, Mar. 2002.

[21] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-Time Syst.*, vol. 26, no. 2, pp. 161–197, Mar. 2004.

[22] V. Nélis, J. Marinho, B. Andersson, and S. M. Petters, "Global-EDF scheduling of multimode real-time systems considering mode independent tasks," in *Proc. 23rd Euromicro Conf. Real-Time Syst.*, 2011, pp. 205–214.

[23] J. Lee and K. Shin, "Schedulability analysis for a mode transition in real-time multi-core systems," in *Proc. IEEE 34th Real-Time Syst. Symp.*, 2013, pp. 11–20.

[24] D. Casini, A. Biondi, and G. Buttazzo, "Handling transients of dynamic real-time workload under EDF scheduling," *IEEE Trans. Comput.*, vol. 68, no. 6, pp. 820–835, Jun. 2019.

[25] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, no. 1, pp. 129–154, May 2005.

[26] J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-Time Syst.*, vol. 25, no. 2, pp. 187–205, Sep. 2003.
[27] T. Baker, "Multiprocessor EDF and deadline monotonic schedulability analysis," in *Proc. 24th Int. Real-Time Syst. Symp.*, 2003, pp. 120–129.
[28] S. Baruah and T. Baker, "Global EDF schedulability analysis of arbitrary sporadic task systems," in *Proc. 20th Euromicro Conf. Real-Time Syst.*, 2008, pp. 3–12.
[29] N. W. Fisher, "The multiprocessor real-time scheduling of general task systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2007.
[30] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 4, pp. 553–566, Apr. 2009.
[31] N. Balakrishnan and V. B. Nevzorov, *A Primer on Statistical Distributions*, Hoboken, NJ, USA: Wiley, 2003.
[32] J. Anderson, V. Bud, and U. Devi, "An EDF-based scheduling algorithm for multiprocessor soft real-time systems," in *Proc. 17th Euromicro Conf. Real-Time Syst.*, 2005, pp. 199–208.
[33] B. Andersson, K. Bletsas, and S. Baruah, "Scheduling arbitrary-deadline sporadic task systems on multiprocessors," in *Proc. 29th Real-Time Syst. Symp.*, 2008, pp. 385–394.
[34] S. Kato and N. Yamasaki, "Portioned static-priority scheduling on multiprocessors," in *Proc. 22nd Int. Symp. Parallel Distrib. Process.*, 2008, pp. 1–12.
[35] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *Proc. 21st Euromicro Conf. Real-Time Syst.*, 2009, pp. 249–258.
[36] S. Kato and N. Yamasaki, "Semi-partitioned fixed-priority scheduling on multiprocessors," in *Proc. 15th Real-Time Embedded Technol. Appl. Symp.*, 2009, pp. 23–32.
[37] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "Is semi-partitioned scheduling practical?" in *Proc. 23rd Euromicro Conf. Real-Time Syst.*, 2011, pp. 125–135.
[38] P. Souto, P. B. Sousa, R. I. Davis, K. Bletsas, and E. Tovar, "Overhead-aware schedulability evaluation of semi-partitioned real-time schedulers," in *Proc. 21st Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2015, pp. 110–121.
[39] C. Maia, P. M. Yomsi, L. Nogueira, and L. M. Pinho, "Real-time semi-partitioned scheduling of fork-join tasks using work-stealing," *EURASIP J. Embedded Syst.*, vol. 2017, no. 1, Sep. 2017, Art. no. 31.
[40] L. George, P. Courbin, and Y. Sorel, "Job versus portioned partitioning for the earliest deadline first semi-partitioned scheduling," *J. Syst. Archit.*, vol. 57, no. 5, pp. 518–535, May 2011.
[41] C. Hobbs, Z. Tong, and J. Anderson, "Optimal soft real-time semi-partitioned scheduling made simple (and dynamic)," in *Proc. 27rd Int. Conf. Real Time Netw. Syst.*, 2019, pp. 112–122.
[42] M. L. Dertouzos and A. K. Mok, "Multiprocessor on-line scheduling of hard-real-time tasks," *IEEE Trans. Softw. Eng.*, vol. 15, no. 12, pp. 1497–1506, Dec. 1989.
[43] N. Fisher, J. Goossens, and S. Baruah, "Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible," *Real-Time Syst.*, vol. 45, no. 1, pp. 26–71, Jun. 2010.
[44] A. Block and J. H. Anderson, "Accuracy versus migration overhead in real-time multiprocessor reweighting algorithms," in *Proc. 12th Int. Conf. Parallel Distrib. Syst.*, 2006, Art. no. 10.
[45] A. Block, J. H. Anderson, and G. Bishop, "Fine-grained task reweighting on multiprocessors," in *Proc. 11th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2005, pp. 429–435.

Daniel Casini (Member, IEEE) received the graduate (cum laude) degree in embedded computing systems engineering, the master degree jointly offered by the Scuola Superiore Sant'Anna of Pisa and the University of Pisa, and the PhD degree in computer engineering from the Scuola Superiore Sant'Anna of Pisa (with honors) working under the supervision of Prof. Alessandro Biondi and Prof. Giorgio Buttazzo. He is a post-doctoral researcher at the Real-Time Systems (ReTiS) Laboratory of the Scuola Superiore Sant'Anna of Pisa. In 2019, he was a visiting scholar at the Max Planck Institute for Software Systems (Germany). His research interests include software predictability in multi-processor systems, schedulability analysis, synchronization protocols, and the design and implementation of real-time operating systems and hypervisors.

Alessandro Biondi (Member, IEEE) received the graduate (cum laude) degree in computer engineering from the University of Pisa, Italy, within the excellence program, and the PhD degree in computer engineering from the Scuola Superiore Sant'Anna under the supervision of prof. Giorgio Buttazzo and prof. Marco Di Natale. He is an assistant professor with the Real-Time Systems (ReTiS) Laboratory of the Scuola Superiore Sant'Anna. In 2016, he was a visiting scholar at the Max Planck Institute for Software Systems (Germany). His research interests include design and implementation of real-time operating systems and hypervisors, schedulability analysis, cyber-physical systems, synchronization protocols, and component-based design for real-time multiprocessor systems. He was recipient of six best paper awards, one Outstanding Paper Award, ACM SIGBED Early Career Award 2019, and EDAA Dissertation Award 2017.

Giorgio Buttazzo (Fellow, IEEE) received the graduate degree in electronic engineering from the University of Pisa, in 1985, the MS degree in computer science from the University of Pennsylvania, in 1987, and the PhD degree in computer engineering from the Scuola Superiore Sant'Anna of Pisa, in 1991. He is full professor of computer engineering at the Scuola Superiore Sant'Anna of Pisa. He is editor-in-chief of *Real-Time Systems*, associate editor of the *ACM Transactions on Cyber-Physical Systems*. He has authored seven books on real-time systems and more than 200 papers in the field of real-time systems, robotics, and neural networks.