Semi-Partitioned Scheduling of Dynamic Real-Time Workload: A Practical Approach Based on Analysis-Driven Load Balancing

Daniel Casini[1], Alessandro Biondi[2], and Giorgio Buttazzo[3]

1 **Scuola Superiore Sant'Anna, Pisa, Italy**
daniel.casini@santannapisa.it

2 **Scuola Superiore Sant'Anna, Pisa, Italy**
alessandro.biondi@santannapisa.it

3 **Scuola Superiore Sant'Anna, Pisa, Italy**
giorgio.buttazzo@santannapisa.it

Abstract

Recent work showed that semi-partitioned scheduling can achieve near-optimal schedulability performance, is simpler to implement compared to global scheduling, and less heavier in terms of runtime overhead, thus resulting in an excellent choice for implementing real-world systems. However, semi-partitioned scheduling typically leverages an off-line design to allocate tasks across the available processors, which requires a-priori knowledge of the workload. Conversely, several simple global schedulers, as global earliest-deadline first (G-EDF), can transparently support dynamic workload without requiring a task-allocation phase. Nonetheless, such schedulers exhibit poor worst-case performance.

This work proposes a semi-partitioned approach to efficiently schedule dynamic real-time workload on a multiprocessor system. A linear-time approximation for the C=D splitting scheme under partitioned EDF scheduling is first presented to reduce the complexity of online scheduling decisions. Then, a load-balancing algorithm is proposed for admitting new real-time workload in the system with limited workload re-allocation. A large-scale experimental study shows that the linear-time approximation has a very limited utilization loss compared to the exact technique and the proposed approach achieves very high schedulability performance, with a consistent improvement on G-EDF and pure partitioned EDF scheduling.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases semi-partitioned scheduling, dynamic workload, real-time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.13

1 Introduction

Many real real-time systems are characterized by a dynamic workload where computational activities (tasks) can join and leave the system, e.g., depending on the occurrence of specific events in their operating environment. Representative examples are modern multimedia software systems [17] (including those widely available in smartphones and tablets), cloud services [28], real-time databases, and open environments in which software components may join the system while the rest of the components continue to operate. Indeed, the possibility to spawn tasks at runtime is given by several real-time operating systems, including VxWorks, QNX and Linux. Such operating systems offer *global* scheduling policies such as *global fixed-priority* (G-FP) and *global earliest-deadline first* (G-EDF), which have the precious benefit of providing an automatic load balancing across the available processors, thus providing to the application designer a simple and application-transparent scheduling mechanism. This benefit likely determined the popularity of such schedulers; however, they have been demonstrated being not optimal and generally exhibit poor worst-case performance due to several difficulties that have been identified in the literature [18]. For this reason, numerous efforts have been spent in studying and analyzing different techniques for scheduling real-time workload on multiprocessor systems. In particular, several optimal multiprocessor scheduling algorithms have been proposed, such as RUN [37], U-EDF [33], QPS [32] and LLREF [16], which are generally more complex (and hence more difficult to implement) and more expensive in terms of run-time overhead with respect to G-FP and G-EDF. Besides global schedulers, alternative approaches have been proposed based on *partitioned* and *semi-partitioned* scheduling. The former class of schedulers relies on a static allocation of the workload to the processors, which is generally suitable for hard real-time systems with fixed task sets. Semi-partitioned scheduling allows improving the performance of partitioned schedulers when a valid workload allocation cannot be found or simply does not exist. While some tasks are statically allocated to the processors, others are *split* across multiple processors, i.e., they are subject to a controlled (and limited) migration at specific time instants during their execution. As for partitioned schedulers, semi-partitioned schedulers typically leverage an off-line phase for allocating the tasks, which makes them less prone to support dynamic workload. Recently, Brandenburg and Gül [12] demonstrated that by clever task partitioning, semi-partitioned EDF scheduling with C=D splitting [13] allows achieving 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 workload may benefit of semi-partitioned scheduling. Nonetheless, considerable challenges arise when aiming at supporting the C=D semi-partitioned scheduling of dynamic workload. In particular, the C=D splitting algorithm has a high computational complexity and therefore it cannot be adopted on-line without incurring in high overheads. Furthermore, load-balancing algorithms are needed to support the dynamic allocation and splitting of incoming workload.

Contribution. This paper makes the following three contributions. First, it proposes linear-time approximate methods for performing the C=D splitting, which enable making practically viable online scheduling decisions. Second, it presents load-balancing algorithms for C=D semi-partitioned scheduling to admit new workload, and performing limited re-allocations to facilitate the admission of future workload. 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 three linear-time approximate methods for performing the C=D splitting. Section 4 presents some 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.

2 System Model and Background

This paper considers the problem of scheduling dynamic workload consisting of an arbitrary number of *reservation servers* upon m identical processors. A reservation r_i is characterized by a *budget* of execution time C_i and a minimum inter-replenishment time T_i. Such reservation servers can arbitrarily enter or leave the system. However, each of them must pass an *acceptance test* (based on parameters C_i and T_i) before being admitted into the system; those that do not pass the test are rejected (i.e., ignored). At any point in time, $\mathcal{R}$ denotes the set of reservations that are currently admitted for execution. Each reservation server $r_i \in \mathcal{R}$ generates a potentially infinite number of *instances*: in each instance, the server executes for at most C_i time units and then is de-descheduled. An instance of the server begins when (i) the budget of the server is replenished and (ii) the server 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 beginning times of the instances follow a *sporadic* pattern. Reservations are considered to be independent (i.e., they do not share resources other than the processors). The results presented in this work are not limited to a specific reservation algorithm, but the server behavior has to comply with some requirements that are discussed in the next section. We say that a reservation r_i is *schedulable* if, in every instance of the server, the system is able to guarantee the execution of its entire budget C_i (used to serve pending workload running upon the server) before the time at which the budget will be replenished. The goal of the acceptance test is to ensure that *all* the reservations into the set $\mathcal{R}$ are always schedulable. In this work, the acceptance test employs an on-line *load balancing* algorithm that allocates the reservations to the processors, which will be presented in Section 4. 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* [39], 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. Note that the adoption of reservation servers also provides the benefit of guaranteeing a *temporal isolation* of the workload, protecting the system from tasks' overruns or processor-eager best-effort computational activities. This feature is particularly suited for systems running dynamic workload, 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. Nonetheless, the approach proposed in this paper is also valid for sporadic tasks with implicit deadlines. In this work, the reservations that are admitted for execution are scheduled under *semi-partitioned* EDF scheduling with the C=D splitting scheme [12, 13], which is briefly reviewed in the next section.

2.1 C=D Semi-partitioned Scheduling of Reservations

Semi-partitioned scheduling allows improving the schedulability performance of pure partitioned scheduling when valid static allocations of the reservations cannot be found or simply do not exist. With this approach, some reservations are statically allocated to processors, while others are *split* across multiple processors, thus involving the migration of the workload executing upon such reservations. More specifically, the budget of some reservations is divided into multiple portions (i.e., chunks) that are executed on different processors with precedence constraints. Among the various methods that one may imagine to split the budget, the so called C=D splitting scheme has been found to perform particularly well. One of the first proposals of this method is due to Kato and Yamasaki [26]: the authors assumed partitioned fixed-priority scheduling as a baseline scheduling algorithm while ensuring that the split portions of budget are executed with the highest priority on each processor. Since a reservation scheduled at the highest priority does not suffer temporal interference from the other reservations allocated on the same processor, it is guaranteed that its budget C will be always consumed within a deadline of $D = C$ time units from its release. The authors exploited this property to facilitate the splitting phase. Later, Burns et al. [13] proposed an improved C=D scheme under partitioned EDF scheduling, which exploits scheduling deadlines to guarantee the system schedulability in the presence of splitting. Following their approach, a budget is split into n portions, each allocated on n different processors. The first $n - 1$ portions are scheduled with a scheduling deadline equal to the corresponding duration of the portion – i.e., they have always zero laxity. Differently from [26], the last portion is scheduled with a deadline greater than or equal to the duration of the portion, hence it may suffer temporal interference from other reservations. Recently, Brandenburg and Gül [12] proposed an extension of the Burns et al.'s approach where the execution order of the portions of budget is flipped. This approach allows taking advantage of slack reclamation, which in turn provides the benefit of reducing the number of migrations in the average-case. In this paper, the latter splitting scheme is considered for the run-time scheduling mechanism.

Run-time scheduling mechanism. As soon as a server is admitted, its budget is immediately replenished. If an instance of a server r_i begins at a time t, the next budget replenishment is set at time $t + T_i$. As typical for EDF scheduling, each reservation server r_i is assigned a *relative deadline* D_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. Following semi-partitioned scheduling, some reservations servers are statically allocated to processors (i.e., they never migrate across processors) – for this reason they are referred to as *partitioned reservations*. Partitioned reservations have a relative deadline equal to their minimum inter-replenishment time, that is $D_i = T_i$. Other reservations are split across multiple processors and are referred to as *semi-partitioned reservations*. Consider a semi-partitioned reservation r_i whose budget C_i is split into two portions, say C_i' and C_i'' such that $C_i = C_i' + C_i''$. Following the approach proposed in [12], the first portion of budget is scheduled on a processor P' with relative deadline $D_i' = T_i - C_i''$ and minimum inter-replenishment time T_i, while the second one is scheduled on a different processor $P'' \neq P'$ with relative deadline $D_i'' = C_i''$ 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. Suppose that an instance of r_i begins at time t and that the server has continuously pending workload to execute. The first C_i' units of budget of r_i are served by its head reservation, i.e., on processor P'. Then, every time the budget C_i' is exhausted, the workload executing upon r_i is *migrated* to processor P'', where it will be served by the r_i's tail reservation. If the head reservation is schedulable within its relative deadline D_i', this event is guaranteed to happen at a time $t' \leq t + D_i'$. Contextually, the head reservation is de-scheduled and its budget will be replenished at time $t + T_i$. If the tail reservation is schedulable within its relative deadline $D_i'' = C_i''$, the C=D approach [12] guarantees that C_i'' units of time are served before time $t + T_i$, 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 arrival 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

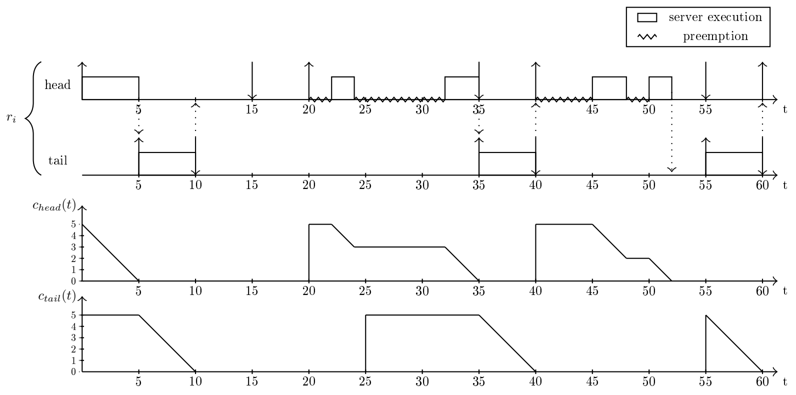

Figure 1 Example of semi-partitioned scheduling of a reservation r_i ($C_i = 10, T_i = 20$) under C=D splitting. The budget of r_i is split into two portions of length 5 time units, executing on two processors. Up-arrows denote the beginning of an instance of the servers. Down-arrows denote the absolute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

time, their replenishment times are generally not synchronized. The approach can be further generalized by considering budget splits in more than two portions: in this case, a reservation is split into one head reservation and *multiple* tail reservations. For each processor P, at each point in time the system selects for execution the reservation allocated to P that has (i) a pending instance and (ii) the earliest absolute deadline. To better clarify the scheduling mechanism, consider a reservation r_i with $C_i = 10$ and $T_i = 20$ that is split into: (i) one head reservation configured with $C'_i = 5$ and $D'_i = 15$; (ii) one tail reservation configured with $C''_i = 5$, $D''_i = 5$. A possible schedule of such sub-reservations is illustrated in Figure 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. For the sake of simplicity, in this work we also pose this limitation for head reservations: this is reflected only in a restriction of the possible allocation configurations. One of the main issues with semi-partitioned scheduling consists in splitting and allocating the reservations. Previous work assumed a static workload and leveraged an off-line design phase to solve this problem. This phase typically consists in the combination of (i) bin-packing heuristics (such as variants of first-fit and worst-fit) to allocate the reservations and (ii) a splitting algorithm to decide how to size the budget portions of the semi-partitioned reservations. 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* $C''_i < C_i$ for which a tail reservation with budget C''_i, deadline $D''_i = C''_i$ 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 $C'_i = C_i - C''_i$ is then allocated to another processor $\neq P_k$ 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) [42] to determine whether the set of reservations allocated to P_k is schedulable.
2. If not, recompute a reduced value of C''_i 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''_i reduces to 0, then the tail reservation cannot be allocated to processor P_k.

This algorithm is optimal, in the sense that it founds the maximum value of C''_i 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 $r_{tail,k} \in \mathcal{R}_k$ denotes the tail reservation allocated to P_k. Similarly, if $head(P_k) = true$, then $r_{head,k} \in \mathcal{R}_k$ denotes the head reservation allocated to P_k. 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 $r_{tail,k}$ (resp., head reservation $r_{head,k}$), the *father* reservation from which the split has been originated is denoted as $\mathcal{F}(r_{k,tail})$ (resp., $\mathcal{F}(r_{k,head})$). The notation adopted in this paper is summarized in Table 1.

3 An Approximated Algorithm For C=D Splitting

This section presents a new approach for computing the C=D splitting discussed in Section 2.2. The proposed algorithm provides an approximate solution to compute a safe lower-bound on the maximum zero-laxity portion of budget that can be allocated to a processor. The algorithm has been designed to have a *linear time* complexity in order to be efficiently applied for on-line load balancing. The baseline approach is first presented in Section 3.1. Then, two possible extensions are proposed in Section 3.2 to improve the algorithm precision. Finally, Section 3.3 discusses some implementation issues and the algorithm complexity.

3.1 The Baseline Approach

The method proposed in this paper is based on the *processor-demand criterion* (PDC) proposed by Baruah et al. [6]. The PDC analysis is based on the notion of *demand bound*

Table 1 Main notation adopted throughout the paper.

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
$r_{head,k}$	head reservation allocated to P_k
$r_{tail,k}$	tail reservation allocated to P_k
$\mathcal{F}(r_i)$	father reservation of a tail or head reservation r_i

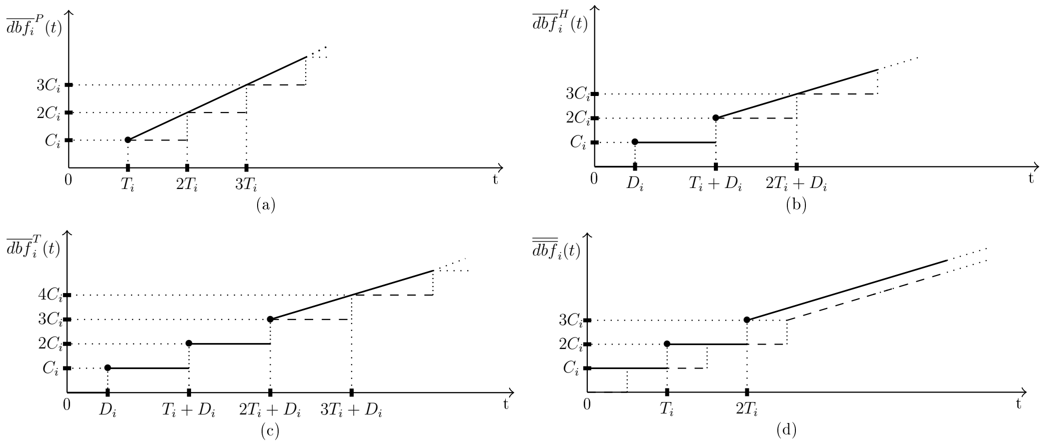

Figure 2 Illustrations of the demand bound functions introduced in Section 3.1 (solid lines). The dashed lines in insets (a), (b) and (c) depict functions $dbf_i(t)$, while the dashed line in inset (d) depicts function $\overline{dbf_i}^T(t)$.

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

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

To design the approximate splitting algorithm, the demand bound function of each reservation is first approximated by an upper bound, which is a particular case of the one proposed by Fisher et al. [20]. In particular, three types of upper bounds are distinguished depending on whether a reservation is partitioned, head or tail.

Partitioned reservation. The demand bound function of a partitioned reservation r_i is upper-bounded with function $\overline{dbf_i}^P(t)$, defined as

$$\overline{dbf_i}^P(t) = \begin{cases} 0 & \text{if } t < T_i, \\ U_i t & \text{if } t \geq T_i. \end{cases}$$

Head reservation. The demand bound function of a head reservation r_i is upper-bounded with function $\overline{dbf_i}^H(t)$, defined as

$$\overline{dbf_i}^H(t) = \begin{cases} 0 & \text{if } t < D_i, \\ C_i & \text{if } D_i \leq t < T_i + D_i, \\ 2C_i + U_i(t - T_i - D_i) & \text{if } t \geq T_i + D_i. \end{cases}$$

Tail reservation. Finally, the demand bound function of a tail reservation r_i is upper-bounded with function $\overline{dbf_i}^T(t)$, defined as

$$\overline{dbf_i}^T(t) = \begin{cases} 0 & \text{if } t < D_i, \\ kC_i & \text{if } (k-1)T_i + D_i \leq t < kT_i + D_i,\ k = 1, 2 \\ 3C_i + U_i(t - 2T_i - D_i) & \text{if } t \geq 2T_i + D_i. \end{cases}$$

Graphical representations of these three functions are reported in Figures 2(a), 2(b) and 2(c). To keep a compact notation, the following function is also defined:

$$\overline{dbf_i}(t) = \begin{cases} \overline{dbf_i}^P(t) & \text{if } r_i \text{ is partitioned}, \\ \overline{dbf_i}^H(t) & \text{if } r_i \text{ is head}, \\ \overline{dbf_i}^T(t) & \text{if } r_i \text{ is tail}. \end{cases}$$

Based on these bounds, it is possible to formulate a sufficient PDC-based condition to verify the schedulability of the reservations allocated to a processor, which is expressed by the following theorem.

▶ **Theorem 1.** *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*

$$\forall t \in \bigcup_{r_i \in \mathcal{R}_k} \xi(r_i), \quad \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(t) \leq t$$

where

$$\xi(r_i) = \begin{cases} \{T_i\} & \textit{if } r_i \textit{ is partitioned}, \\ \{D_i, T_i + D_i\} & \textit{if } r_i \textit{ is head}, \\ \{D_i, T_i + D_i, 2T_i + D_i\} & \textit{if } r_i \textit{ is tail}. \end{cases}$$

Proof. By construction, $\forall t \geq 0, \overline{dbf_i}(t) \geq dbf_i(t)$. Hence, if $\forall t \geq 0, \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(t)) \leq t$ holds, then also the original PDC condition is satisfied. Note that $\overline{dbf_i}(t)$ is a stepwise not-decreasing function composed by either constant segments or linear segments, so also $W(t) = \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(t)$ has the same shape with discontinuities in correspondence to the discontinuities of functions $\overline{dbf_i}(t)$. Let $\Delta_1, \Delta_2, \ldots, \Delta_j, \ldots$ be the ordered sequence of the points t in which function $W(t)$ has a discontinuity. Consider one of such points $t = \Delta_j$. If $W(t)$ is constant in $[\Delta_j, \Delta_{j+1})$ (constant segment), then it is sufficient to check that $W(\Delta_j) \le \Delta_j$ to guarantee that $\forall t \in [\Delta_j, \Delta_{j+1})$, $W(t) \le t$. Now, consider the other case in which $W(t)$ has a linear segment in $[\Delta_j, \Delta_{j+1})$. By the hypothesis $\sum_{r_i \in \mathcal{R}_k} U_i \le 1$, it follows that every linear segment of $W(t)$ has a slope that cannot be larger than 1. Hence, it is again sufficient to check that $W(\Delta_j) \le \Delta_j$. Overall, the condition $W(t) \le t$ must be checked only in the discontinuities of functions $\overline{dbf}_i(t)$, which occur for the points expressed by the set $\xi(r_i)$. Hence, the theorem follows. ◀

With the above theorem in place, the considered optimization problem can be defined. 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_{tail} for a tail reservation r_{tail} with minimum inter-replenishment time T_{tail}, such that r_{tail} can be safely allocated to P_k, can be computed by solving the following optimization problem:

$$\begin{aligned} &\text{maximize} \quad C_{\text{tail}} \\ &\text{subject to} \quad \sum_{r_i \in \mathcal{R}_k} \frac{C_i}{T_i} + \frac{C_{\text{tail}}}{T_{\text{tail}}} \le 1 \\ &\qquad\qquad\quad \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf}_i^{\,T}(t) \le t, \ \forall t \in \bigcup_{r_i \in \{\mathcal{R}_k \cup r_{tail}\}} \xi(r_i) \end{aligned}$$

This optimization problem can be manipulated to obtain a sub-optimal solution in a **closed-form**. To this end, the problem is rewritten by means of $J+1$ constraints $C_{\text{tail}} \le V_j(\mathcal{R}_k, T_{tail})$ (with $j = 0, \ldots, J$) in which the functions $V_j(\mathcal{R}_k, T_{tail})$ are *independent* of C_{tail}, so that the solution can be easily computed as $C_{\text{tail}} = min_{j=0,\ldots,J}\ V_j(\mathcal{R}, T_{\text{tail}})$. In other words, given the parameters of the reservations in set $\mathcal{R}_k$ and the minimum inter-replenishment time T_{tail} of the tail reservation, the expressions $V_j(\mathcal{R}_k, T_{tail})$ must result in *constant terms*. First of all, note that the constraint $\sum_{r_i \in \mathcal{R}_k} \frac{C_i}{T_i} + \frac{C_{\text{tail}}}{T_{\text{tail}}} \le 1$ (corresponding to a very simple necessary condition for feasibility) originates a trivial upper bound on the value of C_{tail}, that is

$$C_{\text{tail}} \le C_{\text{tail}}^{\texttt{MAX}} = \left(1 - \sum_{r_i \in \mathcal{R}_k} U_i\right) T_{\text{tail}}.$$

Leveraging the bound $C_{\text{tail}}^{\texttt{MAX}}$, the terms $V_j(\mathcal{R}, T_{tail})$ can be derived by considering the constraints originated by the check-points in the set $\bigcup_{r_i \in \{\mathcal{R}_k \cup r_{tail}\}} \xi(r_i)$. First, note that functions $\overline{dbf}_i(t)$ are piece-wise defined in intervals that depend on the check-point t. When considering the check-points of the tail reservations (i.e., those in the set $\xi(r_{tail})$), the value of functions $\overline{dbf}_i(t)$ for the partitioned and the head reservations cannot be expressed in a closed-form as their value depend on the optimization variable C_{tail} (that is unknown), thus introducing a sort of circular dependency in the equations. The following lemma allows overcoming this issue.

▶ **Lemma 2.** *If the three conditions*

$$\begin{cases} C_{tail} \le min_{r_i \in \mathcal{R}_k}(D_i) - \epsilon & (a) \\ \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(T_{tail} + C_{tail}^{MAX}) + 2C_{tail} \le T_{tail} + D_{tail} & (b) \\ \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}(T_{tail} + C_{tail}^{MAX}) + 3C_{tail} \le 2T_{tail} + D_{tail} & (c) \end{cases}$$

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

$$\forall t \in \xi(r_{tail}), \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf}_i^{\,T}(t) \le t.$$

Proof. Each of the three conditions corresponds to an element of the set $\xi(r_{tail})$. Condition (a) is needed for verifying the constraint $\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(D_{\text{tail}}) + \overline{dbf}_i^T(D_{\text{tail}}) \leq D_{\text{tail}}$. If the tail reservation (configured with $C_{\text{tail}} = D_{\text{tail}}$) 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 $C_{\text{tail}} = D_{\text{tail}} < min_{r_i \in \mathcal{R}_k}(D_i)$, which then gives $\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(D_{\text{tail}}) = 0$ and the constraint for point D_{tail} is implicitly verified. Conditions (b) and (c) verify the constraint $\sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf}_i^T(t) \leq t$ for points $t = T_{tail} + D_{tail}$ and $t = 2T_{tail} + D_{tail}$. Since functions $\overline{dbf}_i(t)$ are monotonic non-decreasing and $C_{\text{tail}} \leq C_{\text{tail}}^{\texttt{MAX}}$, then $\overline{dbf}_i(T_{\text{tail}} + D_{\text{tail}}) \leq \overline{dbf}_i(T_{tail} + C_{\text{tail}}^{\texttt{MAX}})$. Similarly, also $\overline{dbf}_i(2T_{\text{tail}} + D_{\text{tail}}) \leq \overline{dbf}_i(2T_{tail} + C_{\text{tail}}^{\texttt{MAX}})$. The lemma follows by noting that, in the two points, the value of $\overline{dbf}_i^T(t)$ corresponds to $2C_{\text{tail}}$ and $3C_{\text{tail}}$, respectively. ◀

Before proceeding with the constraints originated by the check-points of the head and partitioned reservations, it is necessary to introduce a new demand bound function $\overline{\overline{dbf}}_{tail}(t)$, which is explicitly conceived to deal with the contribution originated by the tail reservation. This function is illustrated in Figure 2(d) and allows removing the circular dependency that would have been introduced by the use of $\overline{dbf}_i^T(t)$.

▶ **Lemma 3.**

$$\forall t \geq 0, \; \overline{\overline{dbf}}_{tail}(t) \geq \overline{dbf}_i^T(t),$$

where

$$\overline{\overline{dbf}}_{tail}(t) = \begin{cases} C_{tail} & \text{if } t < T_{tail} \\ 2C_{tail} & \text{if } T_{tail} \leq t < 2T_{tail} \\ 3C_{tail} + U_{tail}(t - 2T_{tail}) & \text{if } t \geq 2T_{tail} \end{cases}$$

Proof. Let us consider separately the three cases in which $\overline{\overline{dbf}}_{tail}(t)$ is defined. If $t < T_{\text{tail}}$, then $\overline{dbf}_i^T(t)$ can be either equal to 0 or C_{tail}; hence $\overline{dbf}_i^T(t) \leq C_{\text{tail}}$. Since $0 < D_{\text{tail}} < T_{\text{tail}}$, if $T_{\text{tail}} \leq t < 2T_{\text{tail}}$, then $\overline{dbf}_i^T(t)$ can be either equal to C_{tail} or $2C_{\text{tail}}$; hence $\overline{dbf}_i^T(t) \leq 2C_{\text{tail}}$. For the same reason, if $t \geq 2T_{\text{tail}}$, then $\overline{dbf}_i^T(t)$ can be either equal to $2C_{\text{tail}}$ or $3C_{\text{tail}} + U_{tail}(t - 2T_{\text{tail}} - D_{\text{tail}})$. Since for $t \geq 2T_{\text{tail}}$ we have $U_{tail}(t - 2T_{\text{tail}}) \geq 0$, then $\overline{dbf}_i^T(t) \leq 3C_{\text{tail}} + U_{tail}(t - 2T_{\text{tail}})$. 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 reservation (i.e., those in the set $\xi(r_{head,k})$).

▶ **Lemma 4.** *If the two conditions*

$$\begin{cases} \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(D_{head}) + \overline{\overline{dbf}}_{tail}(D_{head}) \leq D_{head} \\ \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(T_{head} + D_{head}) + \overline{\overline{dbf}}_{tail}(T_{head} + D_{head}) \leq T_{head} + D_{head} \end{cases}$$

hold, then

$$\forall t \in \xi(r_{head,k}), \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf}_i^T(t)(t) \leq t.$$

Proof. The lemma directly follows from Lemma 3 and the definition of the set $\xi(r_i)$. ◀

Similarly, the same bound can be applied to the constraints originated by the check-points of the partitioned reservations.

▶ **Lemma 5.** *If*

$$\forall t \in \left\{T_i \mid r_i \in \mathcal{R}_k^P\right\}, \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(T_i) + \overline{\overline{dbf}}_{tail}(T_i) \le T_i$$

holds, then

$$\forall t \in \bigcup_{r_i \in \mathcal{R}_k^P} \xi(r_i), \sum_{r_i \in \mathcal{R}_k} \overline{dbf}_i(t) + \overline{dbf_i}^T(t) \le t.$$

Proof. The lemma directly follows from Lemma 3 and the definition of the set $\xi(r_i)$. ◀

Finally, the results of Lemma 2, Lemma 4 and Lemma 5 are combined in the following theorem, which provides a *closed-form expression* for computing a safe bound on C_{tail}.

▶ **Theorem 6.** *A set of reservations $\mathcal{R}_k$ composed of n_k^P partitioned reservations, at most one head reservation and a tail reservation with minimum inter-replenishment time T_{tail}, can be safely EDF-scheduled on a single processor P_k if*

$$C_{tail} = D_{tail} = min_{j=0,\dots,J}\{V_j(\mathcal{R}_k, T_{tail})\}$$

where $V_0(\mathcal{R}_k, T_{tail}), \dots, V_J(\mathcal{R}_k, T_{tail})$ are defined as in Table 2.

Proof. The set of reservations $\mathcal{R}_k$ is schedulable if the conditions of Theorem 1 hold. Lemma 2, 4 and 5 provide 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 $\forall j = 0, \dots, J,\ C_{\text{tail}} \le V_j(\mathcal{R}_k, T_{\text{tail}})$.[1] All of such constraints are verified if $C_{\text{tail}} = min_{j=0,\dots,J}\{V_j(\mathcal{R}_k, T_{\text{tail}})\}$. Hence the theorem follows. ◀

To avoid incurring in algebraic errors, the derivation of the equations reported in Table 2 has also been mechanized with the Wolfram Mathematica tool: the corresponding file is publicly available on-line [15].

3.2 Extensions

The method presented in the previous section can be extended to further increase the precision of the solution provided by Theorem 6. Two of such extensions are presented here, which are not discussed in details due to lack of space: a detailed discussion is available in an on-line appendix of this paper [15].

Extension 1. As argued in Lemma 3, the bound $\overline{\overline{dbf}}_{tail}(t)$ on the demand bound function of the tail reservation has been derived by considering zero as a lower-bound on C_{tail}. Leveraging a different lower bound C_{tail}^{LB}, it is possible to obtain a demand bound function tighter than

[1] In particular, terms V_1 and V_2 are derived from Lemma 2, terms $V_3, \dots, V_{2+n_k^P}$ are derived from Lemma 5, and terms $V_{3+n_k^P}$ and $V_{4+n_k^P}$ are derived from Lemma 4.

Table 2 List of terms $V_j(\mathcal{R}_k, T_{\text{tail}})$ $(j = 0, \dots, J)$ for Theorem 6, where $J = n_k + 2$ if $head(P_k) = false$ or $J = n_k + 3$ if $head(P_k) = true$.

Constraint for point $t = D_{\textbf{tail}}$
$V_0(\mathcal{R}_k, T_{\text{tail}}) = min\left\{C_{\text{tail}}^{\texttt{MAX}}, min_{r_i \in \mathcal{R}_k}\{D_i\} - \epsilon\right\}$
Constraints for points $t = zT_{\textbf{tail}} + D_{\textbf{tail}}$, z = 1,2
$V_z(\mathcal{R}_k, T_{\text{tail}}) = \frac{1}{z}\left(zT_{\text{tail}} - \sum_{r_i \in \mathcal{R}_k^P} \overline{dbf_i}\left(zT_{\text{tail}} + C_{\text{tail}}^{\texttt{MAX}}\right)\right)$
Constraints for points $t = T_{\mathbf{i}}$, $\forall r_i \in \mathcal{R}_k^P$ $(z = 1, \dots, n_k^P)$
$V_{2+z}(\mathcal{R}_k, T_{\text{tail}}) = \begin{cases} \frac{1}{2}\left(t - \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}\,(t)\right) & \text{if } T_{\text{tail}} \leq t < 2T_{\text{tail}} \\ \frac{T_{\text{tail}}}{t + T_{\text{tail}}}\left(t - \sum_{r_i \in \mathcal{R}_k} \overline{dbf_i}\,(t)\right) & \text{if } t \geq 2T_{\text{tail}} \end{cases}$
Constraints for points $t = zT_{\textbf{head}} + D_{\textbf{head}}$, z = 1,2
$V_{2+n_k^P+z}(\mathcal{R}_k, T_{\text{tail}}) = \begin{cases} \frac{1}{2}\left(t - zC_{\text{head}} - \sum_{r_i \in \mathcal{R}_k^P} \overline{dbf_i}\,(t)\right) & \text{if } T_{\text{tail}} \leq t < 2T_{\text{tail}} \\ \frac{T_{\text{tail}}}{t + T_{\text{tail}}}\left(t - zC_{\text{head}} - \sum_{r_i \in \mathcal{R}_k^P} \overline{dbf_i}\,(t)\right) & \text{if } t \geq 2T_{\text{tail}} \end{cases}$

$\overline{\overline{dbf}}_{tail}(t)$, thus increasing the precision of the result provided by Theorem 6. Such an improved function can be expressed as follows (its derivation is analogous to Lemma 3):

$$\overline{\overline{dbf}}_{tail}\left(t, C_{\text{tail}}^{LB}\right) = \begin{cases} 0 & \text{if } t < C_{\text{tail}}^{LB} \\ C_{\text{tail}} & \text{if } C_{\text{tail}}^{LB} \leq t < T_{\text{tail}} + C_{\text{tail}}^{LB}, \\ 2C_{\text{tail}} & \text{if } T_{\text{tail}} + C_{\text{tail}}^{LB} \leq t < 2T_{\text{tail}} + C_{\text{tail}}^{LB}, \\ 3C_{\text{tail}} + U_{tail}(t - 2T_{\text{tail}} - C_{\text{tail}}^{LB}) & \text{if } t \geq 2T_{\text{tail}} + C_{\text{tail}}^{LB} \end{cases}$$

A sequence of safe lower bounds C_{tail}^{LB} can be obtained in an iterative fashion. That is, it is possible to design an algorithm that starts with $C_{\text{tail}}^{LB} = 0$, use Theorem 6 to compute a value of C_{tail} and then sets $C_{\text{tail}}^{LB} = C_{\text{tail}}$. This latter value can in turn be used to repeat the computation of a new (possibly higher) value of C_{tail} by means of Theorem 6, and so on for a desired number λ of times or until converging to a desired tolerance. Surprisingly, the experiments reported in Section 5.1 show that just two iterations ($\lambda = 2$) provide a significant improvement.

Extension 2. The demand bound functions introduced in the previous section approximate the exact functions by considering a limited number of discontinuities and then a linear upper-bound. Specifically one, two, and three discontinuities have been adopted for the demand bound functions of the partitioned, head, and tail reservations, respectively. The precision of the method can be increased by refining such approximations, i.e., considering additional x discontinuities for each of such functions. Then, it is sufficient to iterate the number x from zero up to a desired value β applying Theorem 6 at each iteration, and finally taking the maximum value obtained for C_{tail}. Also in this case, the experiments reported in Section 5.1 show that just two iterations ($\beta = 2$) provide a significant improvement.

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_{tail} has to be computed for evaluating the possibility of allocating a tail reservation to P_k. In this case, the baseline 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 $C_{\text{tail}} \leq V_j(\mathcal{R}_k, T_{\text{tail}})$ (see Table 2) that *do not depend* on T_{tail} 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 $\mathcal{O}(n_k)$ time. Then, the resulting splitting algorithm only consists in computing (i) the upper bound $C_{\text{tail}}^{\texttt{MAX}}$, which can be done in constant time, (ii) the sum of demand bound functions in $V_1(\mathcal{R}_k, T_{\text{tail}})$ and $V_2(\mathcal{R}_k, T_{\text{tail}})$, which can be done in $\mathcal{O}(n_k)$ time, and (iii) the minimum required by Theorem 6, which can be done in $\mathcal{O}(n_k)$ time. Extension 1 discussed in Section 3.2 consists in repeating the baseline approach λ times, hence has complexity $\mathcal{O}(\lambda n_k)$: fixing a constant number of iterations λ, the resulting algorithm has again linear-time complexity. The same applies to Extension 2 with respect to the number of iterations β.

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 minimal 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. It is worth observing that the admission of a new reservation cannot be generally done immediately 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 [14, 36] (and hence under partitioned scheduling) and under global scheduling [35, 29]. Several solutions are available for analyzing the transient [14, 35], which allow deriving a safe bound on the time that must be waited before admitting a new reservation or let re-allocations to take effect. The design of improved methods that are tailored to C=D semi-partitioned scheduling is out of the scope of this paper and is left as future work. The following two sections discuss how to handle the arrival and the exit of a reservation. Then, Section 4.3 discusses some extensions that allow improving the performance of the load balancing algorithm, but at the cost of increasing its computational complexity.

4.1 Admission of a New Reservation

Whenever the system receives a request for admitting a new reservation r_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, in 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_{head} and a tail reservation r_{tail}. The method presented in Section 3 is used for computing the value of C_{tail} for each processor

P_k ($k = 1, \ldots, m$): the *maximum* of such values is selected as the budget of r_{tail} and the tail reservation is allocated to the corresponding processor. Then, the head reservation r_{head} is configured with budget $C_{head} = C_i - C_{\text{tail}}$ and relative deadline $D_{head} = T_i - C_{\text{tail}}$. Finally, the algorithm tries to allocate r_{head} to a processor following the same strategy used in step 1. If the allocation of the head reservation fails, then r_i is *rejected*; otherwise it is admitted into the system.

Note that both the steps require evaluating whether a reservation can safely be allocated to a processor. If a processor P_k contains only partitioned reservations, then a simple utilization test is adopted: this operation can be performed in constant time (storing the processor utilization in an incremental fashion). Otherwise, Theorem 1 is used, whose cost is $\mathcal{O}(n_k)$. As discussed in Section 3.3, the computation of C_{tail} for a processor P_k has $\mathcal{O}(n_k)$ complexity. Hence, the overall computational cost of the above operations is $\mathcal{O}(mn^{\text{MAX}})$, where $n^{\text{MAX}} = \max_{k=1,\ldots,m}\{n_k\}$.

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, then the following operations are performed:

1. If $tail(P_k) = true$, let $r_j = \mathcal{F}(r_{k,tail})$ and try to allocate r_j to P_k after removing $r_{k,tail}$. That is, the algorithm tries to re-assemble the semi-partitioned reservation r_j. If r_j cannot be allocated to P_k, then the method presented in Section 3 is used for re-computing the value of C_{tail} for processor P_k to inflate the budget of $r_{k,tail}$, contextually decreasing the budget of the corresponding head reservation of r_j.
2. If $head(P_k) = true$, let $r_j = \mathcal{F}(r_{k,head})$ and try to allocate r_j to P_k after removing $r_{k,head}$.

Whenever a semi-partitioned reservation $r_i \in \mathcal{R}$ leaves the system, let $r_{k,head}$ (allocated to P_k) and $r_{z,tail}$ (allocated to P_z) be its head and tail reservations, respectively. Then, $r_{k,head}$ is removed from $\mathcal{R}_k$ and step 1 is performed on P_k. Also, $r_{z,tail}$ is removed from $\mathcal{R}_z$ and step 2 is performed on P_z. These operations require (i) checking at most twice whether a reservation can be allocated to a processor, which costs $\mathcal{O}(n^{\text{MAX}})$ time, and (ii) computing C_{tail} for a single processor (see step 1), which can also be performed in $\mathcal{O}(n^{\text{MAX}})$ time.

4.3 Extensions

This section describes three extensions that have been found to be effective for improving the performance of the load balancing algorithm.

(TAS) – Try all possible splits. Step 2 in Section 4.1 can be improved to increase the chances of admitting a new reservation by means of splitting. The algorithm can be modified as follows: for each processor P_k ($k = 1, \ldots, m$) in decreasing order with respect to their utilization $U^{(k)} = \sum_{r_i \in \mathcal{R}_k} U_i$, (i) compute the value of C_{tail} and use it for configuring the tail reservation, and (ii) try to allocate the resulting head reservation to a processor $\neq P_k$ (following the same strategy used in step 1). Since for each processor the algorithm tries to allocate the head reservation on the other processors, this approach increases the computational cost of the load balancing algorithm, which results $\mathcal{O}(m^2 n^{\text{MAX}})$.

(MS) – Multi-splitting. The splitting schemes discussed above consider the splitting in only two sub-reservations (one head and one tail). In the presence of several reservations that have a heavy utilization (i.e., greater than 0.5), the performance of the load balancing algorithm can be improved by employing an enhanced splitting scheme that considers multiple tail reservations. The algorithm can be modified as follows. First, try to admit the new reservation r_i by splitting it into two sub-reservations. If this fails, let ${C_{\mathrm{tail}}}^{(1)}, \dots, {C_{\mathrm{tail}}}^{(m)}$ be the sequence of the values of C_{tail} for each processor P_k $(k = 1, \dots, m)$ in *decreasing* order. Then, find the maximum index $x < m$ such that $\sum_{j=1}^{x} {C_{\mathrm{tail}}}^{(j)} < C_i$ and configure a head reservation with budget $C_{head} = C_i - \sum_{j=1}^{x} {C_{\mathrm{tail}}}^{(j)}$ and relative deadline $D_{head} = T_i - \sum_{j=1}^{x} {C_{\mathrm{tail}}}^{(j)}$. Subsequently, configure x tail reservations with budgets ${C_{\mathrm{tail}}}^{(1)}, \dots, {C_{\mathrm{tail}}}^{(x)}$, allocate them to the corresponding processors and finally try to allocate the head reservations on one of the remaining processors. If this strategy fails, check also if $\sum_{j=1}^{x+1} {C_{\mathrm{tail}}}^{(j)} \geq C_i$: in this case, the first x tail reservations are allocated as previously discussed and the head reservation is configured with $C_{head} = D_{head} = C_i - \sum_{j=1}^{x} {C_{\mathrm{tail}}}^{(j)}$.[2] This approach does not increase the asymptotic complexity of the load balancing algorithm, but it increases the run-time overhead due the multiple migrations incurred by the multi-split reservations.

(RPR) – 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 $(k = 1, \dots, m)$, 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 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 depends on the technique selected for splitting r_j. If the baseline approach (presented in Section 4.1) is used, then the algorithm has $\mathcal{O}(m^2 n^{\mathrm{MAX}})$ complexity. Conversely, if this extension is adopted in conjunction with the TAS extension, then the algorithm has $\mathcal{O}(m^3 n^{\mathrm{MAX}})$ complexity, while has $\mathcal{O}(m^2 n^{\mathrm{MAX}})$ complexity if it is adopted in conjunction with the MS extension.

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 algorithms presented in Section 3 with respect to the exact algorithm proposed by Burns et al. in [13]. 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 approximation of Section 3), comparing them to G-EDF and partitioned EDF scheduling under different configurations.

5.1 C=D splitting: Approximated vs. Exact

A first experimental study has been carried to evaluate the utilization loss introduced by the approximate C=D splitting algorithms presented in Section 3 with respect to the exact Burns et al.'s [13] method. The study considers a single processor on which a set of reservations is

[2] This is a special case where a head reservation is allocated as if it would be a tail reservation, which allows overcoming the limitation that only at most one head reservation can be allocated to a processor.

already allocated and aims at computing the maximum zero-laxity budget (with different approaches) to allocate a tail reservation on the considered processor.

Reservation set generation. Given n reservations and a target utilization $U = \sum_{i=1}^{n} U_i$, the utilizations U_i of the n reservations are generated with the UUnifast algorithm [9]. 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 $C_i = U_i T_i$. Among the n reservations, one, say r_i, is randomly selected to be the head reservation: the relative deadline of r_i is then randomly generated with uniform distribution in the interval $[C_i + \beta(T_i - C_i), T_i]$, with $\beta = 0.9$ (that makes the interval representative of the configurations generated by C=D splitting). The remaining $n - 1$ reservations are partitioned and hence are configured with implicit deadline.

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. [3] For each combination of these two parameters, 5000 reservation sets have been tested, for a total of almost 2 million reservation sets. For each reservation set $\mathcal{R}$, a random period T_{tail} for a tail reservation r_{tail} has been randomly generated in the range [1, 1000] ms with uniform distribution. Then, the value of C_{tail} such that the set of reservations $r_{tail} \cup \mathcal{R}$ can be safely EDF-scheduled on a single processor has been computed by (i) the exact method of [13] and (ii) the approximate methods proposed in this paper under four different approaches (all of them have *linear-time complexity*). Specifically, `BASELINE` is adopted to refer the approach presented in Section 3.1; `EXT1` to refer Extension 1 of Section 3.2 configured with $\lambda = 2$; `EXT2` to refer Extension 2 of Section 3.2 configured with $\beta = 2$; and `EXT1+EXT2` to refer `EXT1` and `EXT2` applied in conjunction. The approximate methods have then been compared to the exact method in terms of *utilization loss*: that is, given the exact value C_{tail}^{EXA} (by [13]) and an approximate value $C_{tail}^{\mathrm{APP}} \leq C_{tail}^{\mathrm{EXA}}$, the utilization loss introduced by the approximation is defined as $\left(C_{tail}^{\mathrm{EXA}}/T_{tail}\right) - \left(C_{tail}^{\mathrm{APP}}/T_{tail}\right)$.

The experimental results for four different configurations are reported in Figure 3. The complete set of results is available online [15]. As it can be observed from Figures 3(a) and 3(b), the utilization loss introduced by all the tested approaches decreases as the number of reservation increases, approaching values lower than 1% for $n > 18$. In particular, the combination of `EXT1` and `EXT2` (denoted as `EXT1+EXT2`) originates a very limited average utilization loss that is always lower than about 2%. Figures 3(c) and 3(d) show the dependency of the results on the utilization U: `EXT1` is particularly effective for low values of utilization, while `EXT2` increases its effectiveness as U increases. Also varying the utilization, the `EXT1+EXT2` approach exhibits a very good performance. By looking at the complete results collected in this study, it is possible to derive some guidelines for designing an efficient algorithm that – empirically speaking – introduces an average utilization loss *lower than 3%*, that is: use `EXT1+EXT2` for $n \in \{2, 3\}$, use `EXT1` for $n \geq 4$ and $U \leq 0.45$, and use `EXT2` for $n \geq 4$ and $U > 0.45$. Furthermore, `BASELINE` can be adopted in the presence of a higher number of reservations (e.g., $n > 12$).

[3] In the special case of a single reservation ($n = 1$), the *exact* maximum portion of zero-laxity budget that can be safely allocated to a processor can be computed by solving a simple equation (the details are available in an on-line appendix of this paper [15]). The number of reservations has been limited to 20 because the results show that the error introduced by the proposed approximation decreases as the number of reservations increases, approaching very low values for more than 20 reservations.

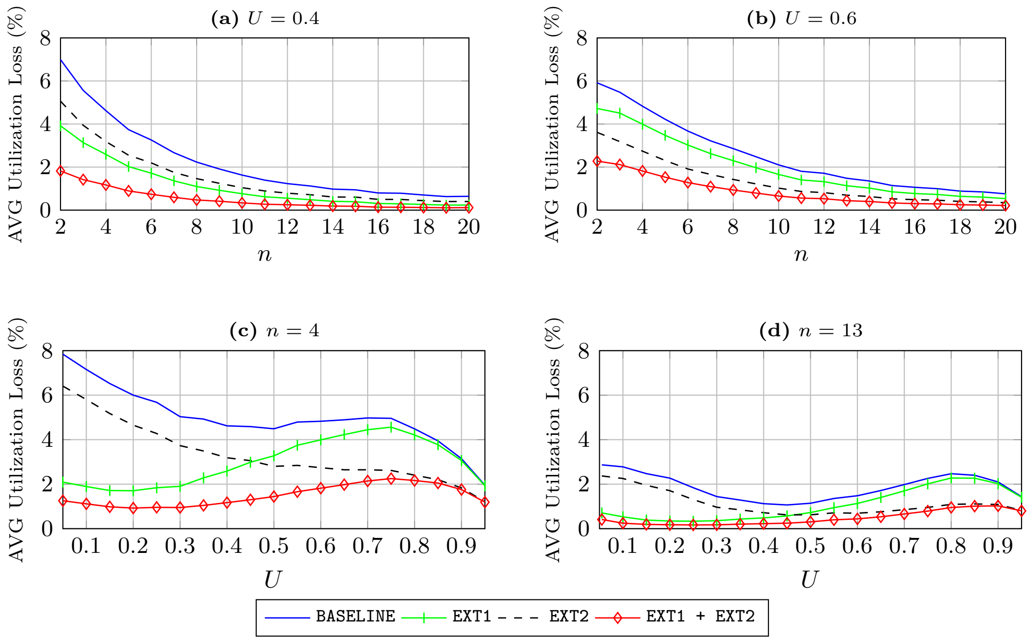

Figure 3 Average utilization loss introduced by the approximate algorithms for C=D splitting (presented in Section 3) as a function of the number of reservations n (insets (a) and (b)) and the utilization U (insets (c) and (d)). The results are related to four representative configurations identified by the fixed parameter reported in the caption above the graphs.

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 @ 4.00GHz. The Microsoft VC++2015 compiler has been used to compile literal implementations (i.e., not designed for being extremely efficient) of the algorithms. 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. The proposed approximate methods showed running times under the precision offered by the Windows API for measuring the time executed by a process with performance counters.

5.2 Proposed Approach vs. G-EDF and P-EDF

A second experimental study has been done to evaluate the performance of C=D semi-partitioned scheduling managed by the load balancing algorithms presented in Section 4 – that in turn make use of the approximate splitting algorithms 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 [24], BAK [3], a polynomial-time approximation of LOAD [5, 22] and I-BCL [8] (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 study is based on synthetic dynamic workload, 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 utilization of each reservation has been generated with the *beta distribution* [4], which allows ensuring a given average U_{AVG} and a given variance U_σ, thus controlling the statistical validity across all the generated values. The distribution has been configured for generating the utilization of each reservation in the fixed range $[U_{min}, U_{max}] = [0.01, 0.9]$.

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 $C_i = U_i T_i$. The `EXIT` event corresponds to the exit of a *random* reservation among those previously generated. Each sequence s is generated as follows: a random real number $x \in [0, 1]$ is generated N^E times with uniform distribution, then 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 utilization accepted by an *optimal* scheduling algorithm that has been stimulated by 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 loading the processors; i.e., the higher ψ the higher the average load demanded by a sequence.

Experiments. The average U_{AVG} of the utilizations of the generated reservations has been varied in the range [0.1, 0.7] with step 0.05, whereas the variance U_σ has been varied in the range [0.05, 0.50], with step 0.05. 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, for a total of more than 4 billion events. 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% corresponds to the performance of an optimal algorithm).[4] Figure 4 reports the results for six representative configurations with $\psi = 0.9$. The complete set of results is available online [15]. 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 with no extensions enabled; `C=D-LB+EXT` refers to `C=D-LB` applied in conjunction with all the extensions presented in Section 4.3. As can be observed from Figures 4(a), 4(b) and 4(c), the performance of the algorithms is significantly affected by the utilization of the tested reservations (as also previously observed in other works). The `C=D-LB+EXT` approach allows achieving high performance, keeping the average accepted load above the 87% in all the configurations, even in the presence of several reservations with high utilization. In particular, it allows achieving a performance improvement up to 40% over G-EDF and up to 25% over P-EDF. The algorithms based on P-EDF show relatively

[4] 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.

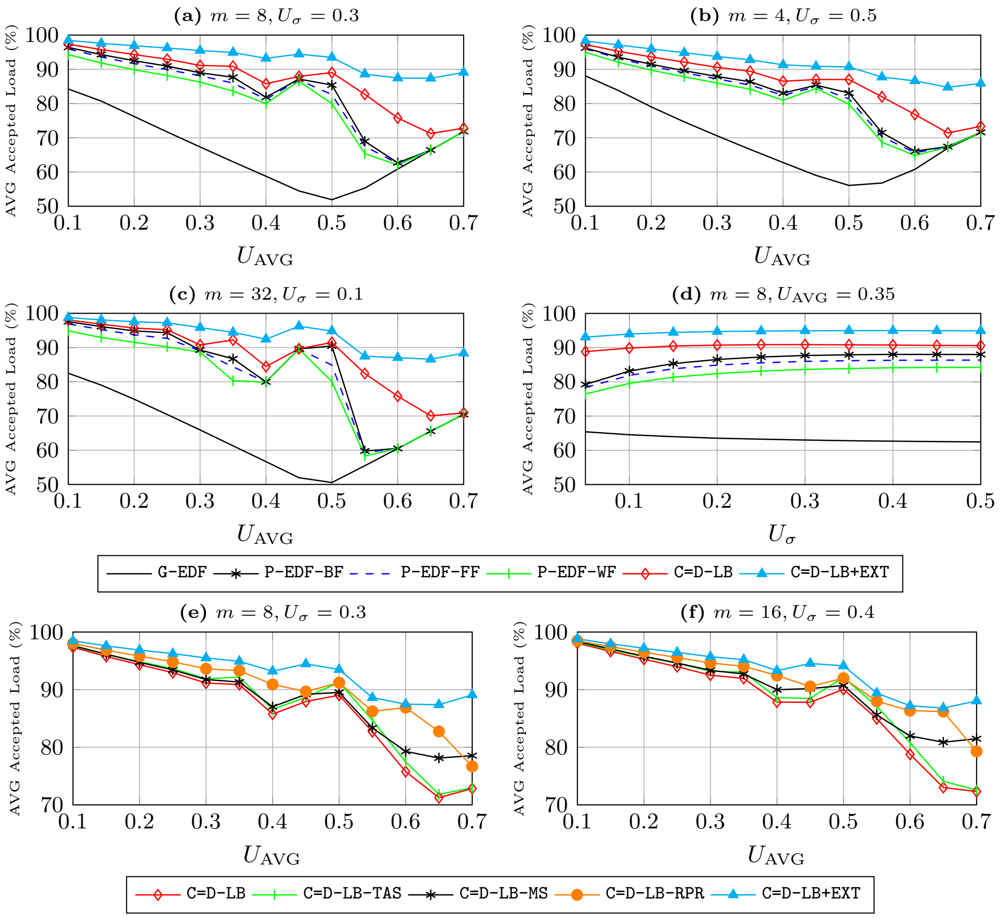

Figure 4 Average accepted load obtained by different scheduling approaches as a function of the average U_{AVG} of the utilizations of the generated reservations (insets (a), (b), (c), (e) and (f)) and the variance U_σ (inset (d)). The results are related to six representative configurations identified by the fixed parameters reported in the caption above the graphs.

good performance up to values of U_{AVG} that are close to 0.5. Surprisingly, basic partitioned scheduling with simple heuristics has been found to always outperform G-EDF. Figure 4(d) shows the dependency on the variance U_σ of the utilizations, which is found to be limited for the `C=D-LB+EXT`. In general, the proposed approach has been found to be robust to the presence of reservations with heterogeneous utilization. Finally, Figures 4(e) and 4(f) show the performance of the baseline load-balancing approach (`C=D-LB`) in conjunction with a single extension. As it can be observed from the graphs, the adoption of the RPR extension provides the highest performance. It is worth observing that the curves tend to show a non-monotonic behavior for the following reason. In the presence of high values of 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), 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 depend on the combination of multiple acceptance tests; in particular, the I-BCL test tends to perform better than the others at high values of U_{AVG}.

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 [18]. 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 firstly introduced by Anderson et al. [1] in 2005. Later, numerous semi-partitioned scheduling algorithms have been presented, including the proposals of Andersson et al. [2] and Kato et al. [25, 26, 27]. In 2011, Bastoni et al. [7] 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. [40]. 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 Deterzous and Mok [19] and by Fisher et al. [21]. Lee and Shin [29] and Nélis et al. [35] proposed techniques for analyzing the effect of system transients under global scheduling, which may also be very useful C=D scheduling. Block and Anderson [11] and Block et al. [10] addressed dynamic workload in the context of task reweighting under partitioned and P-Fair scheduling, respectively. Manimaran and Murthy [23] proposed a scheduling algorithm for parallel and divisible real-time workload, however the authors did not provide any analysis and assessed the algorithm performance only by means of scheduling simulations. Mamat et al. [31, 30] addressed the problem of performing load balancing of aperiodic tasks with known arrival times in clustered-based computing. Other authors addressed the same problem in the context of uniprocessor systems: most relevant to us are the works by Stoimenov et al. [41], Santinelli et al. [38] and Nie et al. [34].

7 Conclusions and Future Work

This work addressed the problem of scheduling real-time dynamic workload upon a symmetric multiprocessor platform. The workload executes upon reservation servers that can arbitrarily join and leave the system, but each of them must pass an admission test before being admitted for execution. The reservations are scheduled under C=D semi-partitioned scheduling. A set of linear-time approximate methods for performing the C=D splitting have been presented to reduce the complexity of online scheduling decisions. Then, load balancing algorithms have been proposed for admitting new real-time workload in the system and performing limited workload re-allocation for facilitating the admission of future reservations. Both the contributions have been evaluated with large-scale experimental studies. The linear-time approximate splitting methods have been shown to originate a very limited utilization loss with respect to the exact technique previously proposed by Burns et al. [13]. In particular, a combination of such methods originates an average utilization loss that is below the 3%. The proposed scheduling approach based on C=D semi-partitioning and load balancing algorithms allows achieving very high schedulability performance, with a consistent improvement over G-EDF and partitioned EDF scheduling with different bin-packing heuristics. As a representative result, the proposed approach allows keeping the average system load above the 87% in most of the tested scenarios, even in the presence of reservations with very high utilizations, with an improvement up to 40% over G-EDF and up to 25% over P-EDF. Considered the simplicity and the limited overhead of C=D semi-partitioned scheduling, as identified in [12], the results concluded in this work suggest its usage even in the presence of dynamic workload. Future work include the derivation of methods that are tailored to C=D semi-partitioned scheduling for handling system transients, the support for elastic reservations [14] to favor the admission of new workload and the consideration of synchronization issues. Moreover, additional research on load balancing algorithms may allow to further increase the performance of the proposed approach.

References

1 J. Anderson, V. Bud, and U.C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS 05)*, Palma de Mallorca, Spain, July 6-8 2005.

2 B. Andersson, K.Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In *Real-Time Systems Symposium, 2008*, Barcelona, Spain, Nov 30 – Dec 3 2008.

3 T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *24th IEEE International Real-Time Systems Symposium (RTSS 03)*, Cancun, Mexico, Dec, 3-5 2003.

4 N. Balakrishnan and V. B. Nevzorov. *A Primer on Statistical Distributions.* Wiley, 2003.

5 S. Baruah and T. P. Baker. Global EDF schedulability analysis of arbitrary sporadic task systems. In *Euromicro Conference on Real-Time Systems (ECRTS 08)*, Prague, Czech Republic, July, 2-4 2008.

6 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 systems*, 2(4):301–324, 1990.

7 A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Is semi-partitioned scheduling practical? In *23rd Euromicro Conference on Real-Time Systems*, Porto, Portugal, July, 5-8 2011.

8 M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, April 2009.

9 E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, May 2005.

10 A. Block, J. H. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, Hong Kong, China, July, 17-19 2005.

11 A. Block and J. H. Anderson. Accuracy versus migration overhead in real-time multiprocessor reweighting algorithms. In *12th International Conference on Parallel and Distributed Systems – (ICPADS'06)*, Minneapolis, USA, July, 12-15 2006.

12 B. Brandenburg and M. Gül. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016)*, Porto, Portugal, November 29 – December 2 2016.

13 A. Burns, R. Davis, P. Wang, , and F. Zhang. Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme. *Real-Time Systems*, 48:3–33, 2012.

14 G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, March 2002.

15 D. Casini, A. Biondi, and G. Buttazzo. Semi-partitioned scheduling of dynamic real-time workload: A practical approach based on analysis-driven load balancing, online material. URL: `https://retis.sssup.it/~d.casini/sp-dyn`.

16 H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, Rio de Janeiro, Brazil, 5-8 December 2006.

17 T. Cucinotta, L. Abeni, L. Palopoli, and G. Lipari. A robust mechanism for adaptive scheduling of multimedia applications. *Journal ACM Transactions on Embedded Computing Systems*, 10(4):1–24, Nov. 2011.

18 R. Davis and A. Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4):35:1–35:44, 2011.

19 M. L. Dertouzos and A. K. Mok. Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, Dec. 1989.

20 N. Fisher, T. P. Baker, and S. Baruah. Algorithms for determining the demand-based load of a sporadic task system. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, Sydney, Australia, Aug, 16-18 2006.

21 N. Fisher, J. Goossens, and S. Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1):26–71, June 2010.

22 N. W. Fisher. *The Multiprocessor Real-Time Scheduling of General Task Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2007.

23 G.Manimaran and C. S. R. Murthy. An efficient dynamic scheduling algorithm for multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):312–319, Mar. 1998.

24 J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2):187–205, Sept. 2003.

25 S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, Miami, Florida, USA, April, 14-18 2008.

26 S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2009)*, San Francisco, CA, USA, April 13-16 2009.

27 S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *21st Euromicro Conference on Real-Time Systems*, Dublin, Ireland, July, 1-3 2009.

28 K. Konstanteli, T. Cucinotta, K. Psychas, and T. Varvarigou. Admission control for elastic cloud services. In *2012 IEEE Fifth International Conference on Cloud Computing*, Honolulu, HI, USA, June, 24-29 2012.

29 J. Lee and K. G. Shin. Schedulability analysis for a mode transition in real-time multi-core systems. In *Proceedings of the 2013 IEEE 34th Real-Time Systems Symposium (RTSS 2013)*, Washington, DC, USA, December 2013.

30 A. Mamat, Y. Lu, J. Deogun, and S. Goddard. An efficient algorithm for real-time divisible load scheduling. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, Stockholm, Sweden, April, 12-15 2010.

31 A. Mamat, J. Deogun Y. Lu, and S. Goddard. Real-time divisible load scheduling with advance reservations. In *2008 Euromicro Conference on Real-Time Systems*, Prague, Czech Republic, July, 2-4 2008.

32 E. Massa, G. Lima, P. Regnier, G. Levin, and S. Brandt. Quasi-partitioned scheduling: optimality and adaptation in multiprocessor real-time systems. *Real-Time Systems*, 52(5):566–597, 2016.

33 G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic. U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In *24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, Pisa, Italy, July 11-13 2012.

34 W. Nie, S. Zhou, K. J. Lin, and S. D. Kim. An on-line capacity-based admission control for real-time service processes. *IEEE Transactions on Computers*, 63(9):2134–2145, Sept. 2014.

35 V. Nélis, J. Marinho, B. Andersson, and S. M. Petters. Global-EDF scheduling of multimode real-time systems considering mode independent tasks. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011)*, Porto, Portugal, July 6-8 2011.

36 J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, March 2004.

37 P. Regnier, G. Lima, E. Massa, G. Levin, , and S. Brandt. Multiprocessor scheduling by reduction to uniprocessor: an original optimal approach. *Real-Time Systems*, 49(4):436–474, 2013.

38 L. Santinelli, G. Buttazzo, and E. Bini. Multi-moded resource reservations. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, Chicago, Illinois, USA, April, 11-13 2011.

39 I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *Journal ACM Transactions on Embedded Computing Systems*, 7(3):1–39, April 2008.

40 P. Souto, P. B. Sousa, R. I. Davis, K. Bletsas, and E. Tovar. Overhead-aware schedulability evaluation of semi-partitioned real-time schedulers. In *21st International Conference on Embedded and Real-Time Computing Systems and Applications*, Hong Kong, China, August 19-21 2015.

41 N. Stoimenov, L. Thiele, L. Santinelli, and G. Buttazzo. Resource adaptations with servers for hard real-time systems. In *10th International Conference on Embedded Software (EMSOFT 2010)*, Scottsdale, Arizona, USA, October, 24-29 2010.

42 F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Trans. Computers*, 58(9):1250–1258, 2009.