Technical Report

Unified Overhead-aware Schedulability Analysis for Slot-based Task-splitting

Paulo Baltarejo Sousa
Konstantinos Bletsas
Eduardo Tovar
Pedro Souto
Benny Akesson

CISTER-TR-130201
Version:
Date: 02-01-2013

Unified Overhead-aware Schedulability Analysis for Slot-based Task-splitting

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

CISTER Research Unit

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

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

Abstract

Hard real-time multiprocessor scheduling has seen, in recent years, the flourishing of semi-partitioned scheduling algorithms. This category of scheduling schemes combines elements of partitioned and migrative scheduling for the purposes of achieving efficient utilisation of the system's processing resources with strong schedulability guarantees and with low dispatching overheads. The sub-class of slot-based "task-splitting" scheduling algorithms, in particular, offers very good trade-offs between schedulability guarantees (in the form of high utilisation bounds) and the number of preemptions/migrations involved.However, so far there did not exist unified scheduling theory for such algorithms; each one was formulated in its own accompanying analysis. This article changes this fragmented landscape by formulating a unified schedulability theory for slot-based semi-partitioning in applicable to all known algorithms of this class. This new theory is based on exact schedulability tests, thus also overcoming many sources of pessimism in existing analysis. In turn, since schedulability testing guides the task assignment under the schemes in consideration, we also formulate an improved task assignment procedure. As the other main contribution of this article, and as a response to the fact thatmany unrealistic assumptions, present in the original theory, tend to undermine the theoretical potential of such scheduling schemes, we identified and modelled into the new analysis all overheads incurred by the algorithms in consideration. The outcome is a new overhead-aware schedulability analysis that permits increased efficiency and reliability. The merits of new theory are evaluated by an extensive set of experiments.

Noname manuscript No.
(will be inserted by the editor)

Unified Overhead-aware Schedulability Analysis for Slot-based Task-splitting

Paulo Baltarejo Sousa · Konstantinos Bletsas · Eduardo Tovar · Pedro Souto · Benny Åkesson

Received: date / Accepted: date

Abstract Hard real-time multiprocessor scheduling has seen, in recent years, the flourishing of semi-partitioned scheduling algorithms. This category of scheduling schemes combines elements of partitioned and global scheduling for the purposes of achieving efficient utilization of the system's processing resources with strong schedulability guarantees and with low dispatching overheads. The sub-class of slot-based "task-splitting" scheduling algorithms, in particular, offers very good trade-offs between schedulability guarantees (in the form of high utilization bounds) and the number of preemptions/migrations involved. However, so far there did not exist unified scheduling theory for such algorithms; each one was formulated in its own accompanying analysis.

This article changes this fragmented landscape by formulating a more unified schedulability theory covering the two state-of-the-art slot-based semi-partitioned algorithms, S-EKG and NPS-F (both fixed job-priority based).

This new theory is based on exact schedulability tests, thus also overcoming many sources of pessimism in existing analysis. In turn, since schedulability testing guides the task assignment under the schemes in consideration, we also formulate an improved task assignment procedure. As the other main contribution of this article, and as a response to the fact that many unrealistic assumptions, present in the original theory, tend to undermine the theoretical potential of such scheduling schemes, we identified and modelled into the new analysis all overheads incurred by the algorithms in consideration. The outcome is a new overhead-aware schedulability analysis that permits increased

Paulo Baltarejo Sousa · Konstantinos Bletsas · Eduardo Tovar · Benny Åkesson
CISTER/INESC-TEC, ISEP,
Polytechnic Institute of Porto, 4200-072 Porto, Portugal
E-mail: pbs@isep.ipp.pt

Pedro Souto
ISR-Porto, FEUP,
University of Porto, Portugal E-mail: pfs@fe.up.pt

efficiency and reliability. The merits of this new theory are evaluated by an extensive set of experiments.

Keywords Multiprocessor Systems · Slot-based Task-Splitting Algorithms · Schedulability Analysis · System Overheads

1 Introduction

The advent of *multicore* chips has drawn the interest of the research community to real-time scheduling on multiprocessors[1] in order to allow efficient use of the processing capacity offered by such systems. However, many challenges exist, because, unlike real-time scheduling theory for uniprocessor systems, which is considered mature, real-time scheduling theory for multiprocessor systems is still a rapidly developing research field. One of the reasons behind many research challenges is that multiprocessor systems introduce an additional dimension to the scheduling problem, which is that of *task migration*. According to the degree of migration, multiprocessor scheduling algorithms have traditionally been categorized as *global* or *partitioned*.

Global scheduling algorithms store all tasks in one global queue, shared by all processors. At any time instant, the m highest-priority tasks among those are selected for execution on the m processors. Tasks can migrate from one processor to another during the execution; that is, an execution of a task can be stopped (preempted) in one processor and resumed on another processor. Some scheduling algorithms [9,3] of this class present a *utilization bound* (a metric for evaluating scheduling algorithms, defined as a threshold for the task set workload such that all tasks meet their deadlines when the task set workload does not exceed that threshold) of 100%, at the cost of many preemptions and migrations. However, the global shared queue imposes the use of some locking mechanism to serialize the access to that queue, which may become a bottleneck. Additionally, the high number of preemptions and migrations can cause numerous cache misses.

In contrast, partitioned scheduling algorithms partition the task set such that all tasks in a partition are assigned to the same processor. Tasks are not allowed to migrate from one processor to another. This class of scheduling algorithms presents a utilization bound of at most 50%. However, it transforms a multiprocessor system, composed by m processors, into m uniprocessor systems, thus simplifying the scheduling problem.

The partitioned scheduling schemes require two algorithms: an *off-line* task-to-processor assignment algorithm and a *run-time* task-dispatching algorithm. The first one assigns tasks to processors and the second one schedules tasks at run-time to execute on the processor(s). Assigning tasks to processors is a bin-packing problem, which is known to be NP-hard. The main goal of a bin-packing algorithm [20] is to pack a collection of items with different sizes

[1] We use the term multiprocessor rather than multicore, because a lot of that work applies not only to multicore but also to other multiprocessor systems.

into the minimum number of fixed-size bins such that the total weight (or volume) in each bin does not exceed some maximum value. In the context of real-time scheduling algorithms, each item is a task from the task set, the size of each item is the *utilization of the task* (defined as the ratio between the execution requirement and the period or the minimal inter-arrival time of a task), each bin is a processor and the size of each bin is the processing capacity of one processor, usually assumed as 100%. There exist several heuristics for these types of problems; examples include Next-Fit (NF) and First-Fit (FF). NF assigns tasks one by one to the current processor and if one task does not fit on the current processor it leaves the current processor behind and continues packing on the next processor. FF assigns a task to the first (lowest indexed) processor that can accept the task. The task-dispatching algorithm schedules the statically assigned tasks using a uniprocessor scheduling algorithm, such as the Earliest-Deadline-First (EDF) [30], which assigns the highest priority to the ready task with earliest absolute deadline.

Recently, real-time researchers have developed *semi-partitioned* or *task-splitting* scheduling algorithms for multiprocessor systems to solve or reduce the drawbacks and limitations presented by global and partitioned scheduling algorithms. Typically, under task-splitting scheduling algorithms, most tasks (called *non-split tasks*) execute on only one processor (as in partitioning) while a few tasks (called *split tasks*) use multiple processors (as in global scheduling). Contrary to what the name may suggest, the code of such tasks is not split; what is split is the execution requirement of such tasks. This approach produces a better workload balance among processors than partitioning (and makes it possible to construct algorithms with a higher utilization bound). Additionally, semi-partitioning may be used to reduce (or remove [38]) the need for a locking mechanism (e.g. by avoiding global shared queues) and it has the potential to reduce the number of migrations, compared to global scheduling (by reducing the number of migrating tasks).

This article focuses on *slot-based* task-splitting scheduling algorithms. These scheduling algorithms present the highest utilization bound among scheduling algorithms that do not share a global queue. Such algorithms subdivide the time into (typically) equal-duration time slots. Each time slot on every processor is composed by one or more *time reserves*, which are time windows (of a fixed respective length) used to execute one or more tasks. Reserves for split tasks, which execute on two or more processors, must be carefully positioned within the time slots in order to avoid their overlapping in time. The three main contributions of the article are: (i) the formulation of a unified, processor demand-based and overhead-aware, schedulability analysis applicable to slot-based task-splitting algorithms S-EKG and NPS-F[2]; (ii) an improved task-assignment algorithm, taking advantage of the new analysis; and (iii) the identification of the overheads associated with slot-based task-splitting scheduling schemes. Apart from the theoretical value of the aforementioned contributions,

[2] Specifically, we only cover the main variant of NPS-F, which splits tasks between no more than two processors.

they are also important because of the following real-world considerations. First, the higher processor utilization, resulting from the improved schedulability testing and task assignment, allows cost savings by enabling fewer (or slower) processors to schedule a given task set. Second, the overhead-aware nature of the analysis permits greater reliability, because it takes into account the overheads incurred by tasks when running in a real system. By contrast, analysis that ignore overheads may deem schedulable a task set, whose tasks may miss their deadlines when running in a real system because of the system overheads. This is an important step towards the use of task-splitting-based scheduling for higher-criticality applications (for which, a missed deadline may have serious real-world consequences).

1.1 Historical perspective and related work

Semi-partitioning was born out of the desire to avert the occurrence of pathological cases when partitioned scheduling performed particularly inefficiently, such as in the following example.

Example: Consider m processors and $n = m + 1$ tasks, each of which arrives every 2 time units and needs to execute for $1.0 + \epsilon$ time units until its next arrival. With partitioning, there is no way to assign tasks without one processor being assigned two (or more) tasks. In turn, this means that on that processor, under whichever scheduling algorithm, it is impossible for more than one task to meet all its deadlines. The implication is that (for $m \to \infty$ and $\epsilon \to 0^+$) deadlines can be missed even though the system is utilized barely above 50%.

Yet, researchers observed [2,6] that, in many cases, if the execution time of a task could be "split" into two pieces (assigned to different processors), then it would be possible to meet deadlines. In the context of the above example, all tasks except the last one could be assigned to one respective processor but the last task could use two processors (any two) in the following manner: after each arrival, execute for $(1.0 + \epsilon)/2$ time units on its first processor and the remaining $(1.0 + \epsilon)/2$ time units on its second processor. Provided that the intervals for execution of this task on the two processors do not overlap in time, this would allow all deadlines to be met.

Many recent algorithms are based on this idea and they differ in: (i) how tasks are assigned to processors and split at design time; and (ii) how tasks (in particular, split tasks) are dispatched at run-time. In just a few years, the landscape of semi-partitioning already comprises many diverse approaches to scheduling. For example, see [2,6,26–28,4,5,29,15–18] and also the survey by Davis and Burns [21]. However, as mentioned before, this article focuses solely on the subset of slot-based task-splitting scheduling algorithms:

In 2006, Andersson and Tovar [6] introduced the first slot-based task-splitting scheduling algorithm called EKG (nowadays often retroactively referred to as "Periodic EKG" or "the original EKG"). EKG was limited to the scheduling of periodic tasks only. Under this scheme, time is divided into *time* *slots* of unequal (in the general case) duration, with the time boundaries of a given time slot corresponding to the time instants of two consecutive job arrivals (possibly by different tasks) in the system. Most tasks are partitioned but at most $m-1$ tasks (with m being the number of processors) are split – each between a corresponding pair of successively indexed processors. Within each time slot, the first piece of a split task is executed at the end of the time slot on the first processor utilized by that task, and the second piece is executed at the start of the time slot on the other processor. All other tasks are executed under EDF on their respective processors. The basic form of the algorithm has a utilization bound of 100%. Clustered variants of EKG divide the system into m/k clusters of k processors each – hence the name EKG, stands for "**E**DF with task splitting and **k** processors in a **g**roup". Such clustering may be used to trade-off utilization bound for fewer preemptions and migrations.

However, the original EKG suffered from the limitation that, by design, it could not handle sporadically arriving tasks. This was because split task budgets in each time slot were proportional to the task utilization and the time slot length. However, given that time slots were formed between successive job arrivals, it was necessary to know the time of next job arrival in order to compute these budgets. With periodic tasks, this is not a problem, since arrival times are deterministic and may be computed in advance. However, with sporadic arrivals, this information is neither known in advance nor predictable.

This is why, in 2008, Andersson and Bletsas came up with an adapted design that came to be known as Sporadic EKG (S-EKG). In order to accommodate sporadic tasks, this algorithm "decouples" the time slot boundaries from the time instants of job arrivals. Rather, all time slots are of equal length. However, given that tasks can now arrive at "unfavorable" offsets relative to the time slot boundary, there is a penalty to be paid in terms of utilization bound: in order to ensure schedulability, processors can no longer be filled up to their entire processing capacity. Via a designer-set parameter, which controls the time slot length, S-EKG can be configured for a utilization bound from 65% to arbitrarily close to 100%, at the cost of more preemptions and migrations. Later in the same year, Andersson et al. came up with a version of S-EKG, named EDF-SS [5]. EDF-SS can handle arbitrary-deadline tasks (whereas its predecessor was formulated in the context of implicit-deadline tasks). However, due to different task assignment heuristics, one version does not dominate the other. Moreover, in part due to this "break" from the previous variant, no utilization bound above 50% has been proven for EDF-SS.

The three EKG variants discussed share a basic design: at most $m-1$ tasks are split, each between two successively indexed processors – the first piece of a split task executes at the end of the time slot on the first processor used by that task and the second piece is executed at the start of the time slot on the other processor. However, a less prescriptive approach to splitting the execution time of tasks between processors, while at the same time maintaining a slot-based dispatching, was soon devised:

In 2009, Bletsas and Andersson presented NPS [15], rapidly superseded entirely by NPS-F [16,17]. This algorithm (and its short-lived predecessor) employ a server-based approach. Each server (termed *notional processor* in the context of that work) serves one or more tasks employing an EDF scheduling policy. Under NPS-F (that stands for Notional Processor Scheduling - Fractional capacity), it is the execution time of these servers which is split – not directly that of the underlying tasks served. In principle, this allows improved efficiency in the utilization of a multiprocessor system. NPS-F has a utilization bound of 75% configurable up to 100% at cost of more preemptions and migrations. Compared to S-EKG, for corresponding configurations characterized by roughly the same number of preemptions, NPS-F has a higher utilization bound. However, a downside to splitting servers instead of tasks is that the number of migrating tasks is not bounded *a priori* and typically exceeds $m-1$.

1.2 Contribution of this article

As a general pattern, not specific to semi-partitioned scheduling, scheduling theory tends to be originally formulated together with a set of simplifying assumptions that have little correspondence with a real system. In the context of slot-based task-splitting scheduling algorithms, although some practical works [35,36,38,13,39] have clearly demonstrated that these scheduling schemes are practical to implement in a real system, the practice also shows some performance degradation compared to what is theoretically achievable, due to various sources of overheads often unaccounted for by the theory. For instance, most theoretical works assume that task switching is instantaneous, but, in practice, it is time consuming because the operating system has to save the state of one task and restore that of another task. In this work, we bridge the gap between theory and practice by adapting the schedulability theory so that it accounts for the overheads that these algorithms incur in a real system. However, the contributions of the article are wider, and are outlined as follows:

1. We formulate a new and comprehensive scheduling theory for slot-based semi-partitioning. Although this theory is not specific to any particular scheme, we show how it can be applied to the specific algorithm under consideration (S-EKG or NPS-F). The fact that this new theory employs exact, processor demand-based, schedulability tests makes it inherently more efficient than the original analysis for the respective algorithms, which employed utilization-based tests. In the absence of overheads, the new analysis dominates its predecessors.
2. We identify and model into the new analysis all types of scheduling overheads manifested under the scheduling algorithms in consideration. This renders the new, unified schedulability analysis *overhead-aware.*
3. We develop a sophisticated off-line task assignment algorithm, which is guided by the new overhead-aware analysis. This brings increased efficiency and reliability to slot-based task-splitting scheduling algorithms.

4. We experimentally derive estimates of the various respective overheads using a real Linux-based multiprocessor system. Using these estimates we validate the efficiency and reliability of the new theory, by applying it to different scenarios.

Note however that, in this paper, we had to balance expressive completeness with presentation. Hence, strictly speaking, we only cover one of the two variants ("flat-mapped") of NPS-F, which, like S-EKG, the other algorithm covered, splits tasks between no more than two processors. Covering the general case (splitting over any number of processors) would add little practical value, since both NPS-F variants have the same theoretical properties, at the cost of considerable complexity. Another limitation of this work is that it does not consider the need for task synchronization (i.e. to access shared resources).

1.3 Organization of this article

The remainder of this article is structured as follows. Section 2 discusses basic assumptions and the system model considered. A generic description of slot-based task-splitting algorithms is presented in Section 3 that ends with a summary of most of the notation used in this article. The purpose of this section is to provide the reader with the necessary background to understand the new demand-based and overhead-aware schedulability analysis presented in Section 4. This analysis is then used to develop a new task to processor assignment algorithm in Section 5. In Section 6, the new schedulability analysis is evaluated and compared to the original schedulability analysis of slot-based task-splitting scheduling algorithms. Additionally, extensive experimental results are provided and discussed. Finally, in Section 7 conclusions are drawn.

2 Assumptions and system Model

2.1 Assumptions about the architecture

We assume a multiprocessor system consisting of identical processors, all of which always execute at the same frequency. This means that the *execution speed* of a processor does not depend on activities on another processor (e.g. whether the other processor is busy or idle or which task it is busy executing) nor does it change at run-time. This work is therefore only applicable to systems with Simultaneous MultiThreading (SMT) and Dynamic Voltage and Frequency Scaling (DVFS) features disabled. In state-of-the-art hardware it is possible to disable both these hardware features via the BIOS and/or software.

We assume that each processor has a *local timer* that keeps track of real-time (not calendar time) and provides a function for reading its value. Furthermore, we assume that it is able to generate an interrupt at x time units in the future (x being configurable). These facilities are rather common. For example, on Linux they are provided by the high-resolution timers framework.

2.2 System model

We consider preemptive real-time systems composed by m physical processors. Each physical processor is uniquely indexed in the range $P_1 \cdots P_m$. The system also includes a task set τ composed by n *independent* tasks, each of which is uniquely indexed in the range $\tau_1 \cdots \tau_n$. Each task τ_i is characterized by its worst-case execution time C_i, by its minimum inter-arrival time T_i and by the time span that can elapse since its arrival until its execution is completed, the relative deadline D_i. We assume $0 \leq C_i \leq D_i$. Note that we also assume *arbitrary deadlines*, i.e. it may be that $D_i < T_i$, $D_i = T_i$ or $D_i > T_i$.

The utilization of task τ_i, denoted u_i, is defined as:

$$u_i = \frac{C_i}{T_i} \tag{1}$$

and the (normalized) system utilization, U_s, is defined as:

$$U_s = \frac{1}{m} \cdot \sum_{i=1}^{n} u_i \tag{2}$$

Each task τ_i generates a potentially infinite number of *jobs* and each job $\tau_{i,j}$ (that is the j^{th} job of task τ_i with $j \leq 1$) becomes *ready* to be executed at arrival time ($a_{i,j}$) and continues until *finishing* (or completion) time ($f_{i,j}$). The *absolute deadline* ($d_{i,j}$) of job $\tau_{i,j}$ is computed as $d_{i,j} = a_{i,j} + D_i$ and a deadline miss occurs when $f_{i,j} > d_{i,j}$. By definition of T_i, the time difference between any two consecutive job arrivals must be at least equal to T_i. Figure 1 illustrates the relation among the timing parameters of job $\tau_{i,j}$. The execution of job $\tau_{i,j}$ is represented by a gray rectangle and the sum of all execution chunks ($c^x_{i,j}$) must be less or equal than C_i.

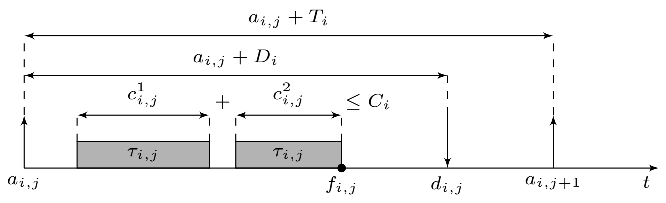

Fig. 1: Job timing parameters.

We also consider a set of k *servers*, which are equivalent to physical processors in terms of processing capacity, indexed in the range $\tilde{P}_1 \cdots \tilde{P}_k$. The set of tasks that can be assigned to a server $\tilde{P}_q$ (denoted by $\tau[\tilde{P}_q]$) is limited by its processing capacity that is equal to 1.0 (100%). The utilization of a server $\tilde{P}_q$ ($U[\tilde{P}_q]$) is given by:

$$U[\tilde{P}_q] = \sum_{\tau_i \in \tau[\tilde{P}_q]} u_i \tag{3}$$

3 Slot-based task-splitting

This section provides background on slot-based semi-partitioning that is essential to understand the demand-based and overhead-aware schedulability analysis presented in the next section. We start by describing the basic concepts and a generic scheduling algorithm. We then show that both S-EKG[3] and NPS-F can be formulated as instances of this generic algorithm.

From this point onwards, we will not consider the (original) EKG scheduling algorithm [6], whose applicability is limited to periodic task sets with implicit deadlines, and whenever we refer to slot-based task-splitting algorithms we mean S-EKG and NPS-F.

3.1 Generic scheduling algorithm

A key concept of the generic scheduling algorithm is that of a *server*. A server is a logical entity that provides computation services to tasks and has a maximum capacity equal to that of the underlying physical processors. Thus, in the generic algorithm, a task is first mapped to a server, which is then allocated one or two processors. A processor may be allocated to at most three servers, but at *any time* a processor is allocated to only one server and one server is served by at most one processor. A *time reserve* is a time window during which a processor is exclusively reserved to a server, i.e. executes tasks of only that server. Therefore, time reserves on a processor are non-overlapping. Furthermore, given the sporadic nature of the tasks in a server, time reserves are periodic and we call their period, which is the same for all reserves, the *time slot*. In the generic scheduling algorithm, in any time slot, a processor has one time reserve per server it is allocated to.

The scheduling of a set of tasks in the generic algorithm comprises two procedures, one that is performed off-line and another that is executed at run-time. The off-line procedure maps tasks to servers, determines the computation capacity of each server and allocates reserves on the processors in order to ensure that each server has the required capacity. The run-time procedure should be a scheduling algorithm that runs on each processor and that uses EDF to choose the task of the server associated to the currently active time reserve.

We now describe the off-line procedure. The generic algorithm specifies a procedure composed of four steps and what is performed in each step, but it does not prescribe any algorithm for any of the steps. This is up to the specific scheduling algorithms.

To illustrate the generic algorithm, we use an example. The figures illustrating its application were obtained by using the algorithms specified for the NPS-F, later described in Section 3.3. The task set (τ) in our example is com-

[3] We focus on S-EKG and not in EDF-SS, because latter is a version of the former that explores different bin-packing heuristics for assigning task-to-processors.

prised of seven tasks, τ_1 to τ_7. Inset (a) of Figure 2 represents each task in that set by a rectangle whose height represents that task's utilization.

The first step of the off-line procedure is mapping tasks to servers, which we denote $\tilde{P}_q$. The generic algorithm does not prescribe how tasks are mapped to servers. Each specific scheduling algorithm can use its own mapping. Inset (b) of Figure 2 shows the task-to-server mapping obtained by applying NPS-F's first step algorithm.

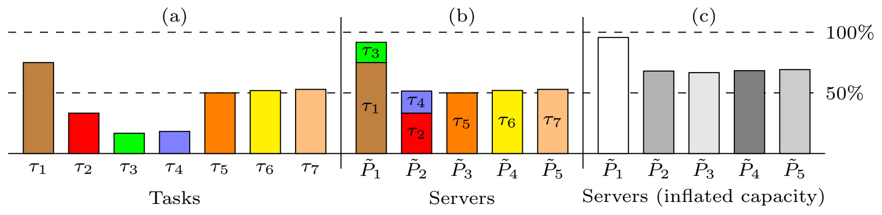

Fig. 2: Task-to-server mapping.

The second step of the off-line procedure is to determine the (computation) capacity of each server. This is obtained by *inflating* the sum of the utilization's of the server's tasks. Capacity inflation is required to compensate for time intervals during which a server may have ready tasks, but none of them can be executed. Such a scenario may arise because none of the server's time reserves are active, and a processor executes tasks of only the server associated to its current time reserve. Several methods can be used to determine by how much to inflate a server capacity. In Section 4, we present one method in the context of the new schedulability analysis. At this point, we assume that such a method exists, and illustrate its application in Inset (c) of Figure 2.

The third step of the off-line procedure is to allocate processors to servers. Again, the generic algorithm does not prescribe how this allocation is done. Each specific algorithm can specify its own. Figure 3 illustrates the server-to-processor assignment obtained by applying the algorithm used in NPS-F to our running example. Servers $\tilde{P}_1$ and $\tilde{P}_4$ are assigned to only one processor each, and are, hence, classified as *non-split servers*; whereas servers $\tilde{P}_2$, $\tilde{P}_3$, and $\tilde{P}_5$ are *split servers* because they are assigned to two processors each.

The fourth and last step of the off-line procedure is to define the reserves for each processor. Again, the generic algorithm does not prescribe how this is done. Figure 4 illustrates the reserves determined by the application of an algorithm used by NPS-F to our running example. In this case, all processors synchronize at the beginning of each time slot. However, other schemes are possible, as shown in Section 4. On each processor P_p, the time slot can be divided into three reserves, at most: $x[P_p]$, $y[P_p]$, and $N[P_p]$. The $x[P_p]$ reserve occurs at the beginning of the time slot and it is reserved for the split server shared by processor P_p and processor P_{p-1}, if any. The $y[P_p]$ reserve occurs at the end of the time slot and it is reserved for the split server shared by processor

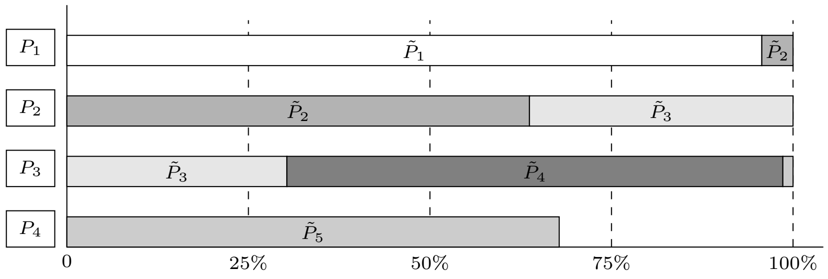

Fig. 3: Server-to-processor assignment

P_p and processor P_{p+1}, if any. The remaining part, $N[P_p]$, is reserved for the non-split server assigned to processor P_p.

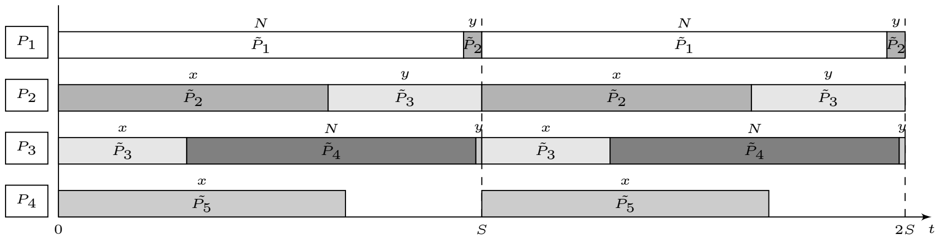

Fig. 4: Processor reserves and time slot.

At run-time, the dispatching inside each reserve is performed according to an Earliest-Deadline First (EDF) policy: the active job with the earliest deadline, among those served by the reserve is executed.

3.2 S-EKG

The S-EKG algorithm shares many features with the generic algorithm. Both are slot-based; both use an off-line procedure to map tasks to processors and a run-time algorithm that uses EDF to choose the running task. A major difference between the two is that S-EKG, as described in its original publication [4], does not use the concept of server, instead it assigns tasks to processors directly, employing a procedure similar to the NF bin-packing heuristic that we describe next.

In S-EKG, the task-to-processor mapping procedure strives to ensure that the utilization of each processor is equal to $\mathrm{UB}_{\mathrm{S-EKG}}$ (a theoretical utilization bound of the algorithm). It iterates over the set of tasks. If a task has a utilization that exceeds $\mathrm{UB}_{\mathrm{S-EKG}}$, it assigns the task to a dedicated processor. Otherwise, it assigns the task to the next available processor whose utilization is lower than $\mathrm{UB}_{\mathrm{S-EKG}}$. In this case, if task τ_i cannot be integrally assigned to the current processor, P_p, without exceeding that bound, it is split between that processor and the next one, P_{p+1}, so that P_p ends up utilized exactly by $\mathrm{UB}_{\mathrm{S-EKG}}$ and P_{p+1} receives the remaining share of τ_i. Consequently, the number of split tasks is at most $m-1$ and there is at most one task split between each pair of successively indexed processors P_p and P_{p+1}. Furthermore, in a schedulable system, the utilization of every non-dedicated processor (except possibly the last one) is exactly $\mathrm{UB}_{\mathrm{S-EKG}}$.

S-EKG uses a designer-set integer parameter δ, which determines the length of the time slot according to Equation 4.

$$S = \frac{1}{\delta} \min_{\tau_i \in \tau}(T_i) \tag{4}$$

This parameter also affects the utilization bound ($\mathrm{UB}_{\mathrm{S-EKG}}$) and the inflation factor (α), which is used to inflate the utilization, as follows:

$$\mathrm{UB}_{\mathrm{S-EKG}} = 4 \cdot (\sqrt{\delta \cdot (\delta + 1)} - \delta) - 1 \tag{5}$$

$$\alpha = \frac{1}{2} - \sqrt{\delta \cdot (\delta + 1)} + \delta \tag{6}$$

Depending on the chosen value for δ, $\mathrm{UB}_{\mathrm{S-EKG}}$ varies from 65% (with δ equal to one) to arbitrarily close to 100% (for $\delta \to \infty$). Therefore, the value of δ can be used to trade-off the target utilization bound against preemptions and migrations.

Although, the original description of S-EKG [4] does not use the concept of server, it is straightforward to map tasks to servers, which are then allocated time reserves as done in the generic algorithm, in such a way that each task is allocated the same set of processors as in S-EKG. The rules to apply are as follows: (i) each task assigned to a dedicated processor is mapped to a server, which is then allocated exclusively the same dedicated processor as in S-EKG; (ii) all non-split tasks that are assigned to one processor are mapped to a non-split server, which is then allocated the same processor as in S-EKG; (iii) each split task is mapped to a server that is split between the same processors that split task is assigned to in S-EKG.

With respect to the inflation of servers, under the original approach [4], each server is (safely, but inefficiently) inflated by the same amount $2 \cdot \alpha$ – in other words:

$$U^{infl:orig}_{S-EKG}[\tilde{P}_q] = U[\tilde{P}_q] + 2 \cdot \alpha \tag{7}$$

with α calculated according to Equation 6.

3.3 NPS-F

It is rather straightforward to formulate NPS-F as an instance of the generic algorithm. Indeed, NPS-F is based on the same concepts as the generic algorithm, and these concepts even have the same name, except for the servers, which were called "notional processors", and gave the name to NPS-F. Furthermore, NPS-F's off-line procedure comprises exactly the same four steps.

Next, we summarize the algorithms used by NPS-F for each step of the off-line procedure. These are the algorithms that were used in the running example in Section 3.1 to illustrate the generic algorithm.

In the first step, the mapping of tasks to servers, NPS-F uses any bin-packing heuristic so that the utilization of each server is smaller or equal to that of a processor. Inset (b) of Figure 2, in Section 3.1, shows the task-to-server mapping obtained with NPS-F by employing the FF bin-packing heuristic.

In the second step, the original paper on NPS-F used the following expression to inflate the capacity of each of the servers obtained in the first step:

$$U^{infl:orig}_{NPS-F}[\tilde{P}_q] = \frac{(\delta + 1) \cdot U[\tilde{P}_q]}{U[\tilde{P}_q] + \delta} \tag{8}$$

where δ is an integer designer-set parameter, which is also used to set the length of the time slot like in S-EKG (see Equation 4).

The algorithm used by NPS-F to allocate processors to servers, the third step, just iterates over the set of servers and assigns each server to the next processor that has yet some available capacity. If the processor's available capacity cannot accommodate the processing requirements of a server, the server is *split*. That is, the current processor's available capacity is allocated to partially fulfil the server's requirements, whereas the server remaining requirements are fulfilled by the next processor.

Finally, the algorithm used by NPS-F in the fourth and last step is also straightforward. For each processor, it allocates one reserve per server. Furthermore, the duration of each reserve is proportional to the processor capacity used by the corresponding server and is such that each server is periodic with a period equal to the time slot, S.

We end this subsection with the utilization bound determined by the original schedulability analysis:

$$\mathrm{UB_{NPS-F}} = \frac{2 \cdot \delta + 1}{2 \cdot \delta + 2} \tag{9}$$

which ranges from 75% (for δ equal to one, which is the most preemption- and migration-light setting) to arbitrarily close to 100% (for $\delta \to \infty$). Because δ controls the length of the time slot, S, (see Equation 4), its value can be used to trade-off the target utilization bound against preemptions and migrations like in S-EKG.

3.4 Notation

For ease of reference, Table 1 provides a summary of most of the notation used in this article.

Table 1: Notation

Symbol	**Interpretation**	**Constraint/ Definition**
τ	A task set	
τ_i	The i^{th} task	
C_i	The worst-case execution requirement of task τ_i	
T_i	The minimum inter-arrival time of task τ_i	$T_i \geq C_i$
D_i	The relative deadline of task τ_i	$D_i \geq C_i$
u_i	The utilization of task τ_i	$u_i = C_i/T_i$
n	The number of tasks of τ	
$\tau_{i,j}$	The j^{th} job of task τ_i	
$a_{i,j}$	The arrival time of job $\tau_{i,j}$	
$d_{i,j}$	The absolute deadline of job $\tau_{i,j}$	$d_{i,j} = a_{i,j} + D_i$
P_p	The p^{th} processor	
$x[P_p]$	The processor P_p's x reserve length	See Equation 39
$N[P_p]$	The processor P_p's N reserve length	See Equation 39
$y[P_p]$	The processor P_p's y reserve length	See Equation 39
$U[P_p]$	The utilization of processor P_p	
m	The number of processors	
$\tilde{P}_q$	The q^{th} server	
$\tau[\tilde{P}_q]$	The set of tasks assigned to the server $\tilde{P}_q$	
$U[\tilde{P}_q]$	The utilization of server $\tilde{P}_q$	See Equation 3
$U^{infl}[\tilde{P}_q]$	The inflated utilization of server $\tilde{P}_q$	See Algorithm 2
$U_x^{infl}[\tilde{P}_q]$	The x part of the inflated utilization of server $\tilde{P}_q$	See Equation 39
$U_y^{infl}[\tilde{P}_q]$	The y part of the inflated utilization of server $\tilde{P}_q$	See Equation 39
k	The number of servers	
S	The time slot length	See Equation 4
δ	A designer-set integer parameter controlling the migration frequency of split tasks	
α	The inflation factor of S-EKG	See Equation 4
$\varOmega$	The time interval between the two split server reserves	See Equation 38
$RelJ$	An upper bound for the release jitter	See Figure 5
$RelO$	An upper bound for the release overhead	See Figure 5
$ResL$	An upper bound for the reserve latency	See Figure 7
$CtswO$	An upper bound for the context switch overhead	
$IpiL$	An upper bound for inter-processor interrupt latency	See Figure 8
$CpmdO$	An upper bound of the cache-related preemption/migration delay overhead	
C^{fake}	Execution time of the fake task modelling a reserve	See Equation 27 and Equation 40
D^{fake}	Deadline of the fake task	See Equation 27 and Equation 40
T^{fake}	Minimal inter-arrival time of the fake task	See Equation 27 and Equation 40
U_s	Utilization of the system	See Equation 2
$\text{UB}_{\text{S-EKG}}$	Utilization bound of S-EKG	See Equation 5
$\text{UB}_{\text{NPS-F}}$	Utilization bound of NPS-F	See Equation 9

4 New demand-based and overhead-aware schedulability analysis

The original schedulability analysis for slot-based task-splitting scheduling algorithms was based on utilization. While this simplifies the derivation of utilization bounds, it also entails pessimism. In [5], the move towards processor-demand based analysis was not carried out in a way that would preserve the most useful theoretical properties (namely, the utilization bound) of previous work (S-EKG). Therefore, in [37], the authors present a schedulability analysis based on processor demand specific for the S-EKG scheduling algorithm.

In this article, a new schedulability analysis, based on processor demand, is introduced that can be applied to both S-EKG and NPS-F. This new schedulability analysis supersedes all previous utilization-based analyses. Further, it defines new schedulability tests that incorporate all real-world overheads incurred by implementations of the S-EKG and NPS-F algorithms [38,39].

The schedulability analysis that we develop in this section has two stages, which correspond to the two main stages of the task-to-processor mapping algorithm presented in the previous section. In the first stage, the analysis focuses on the schedulability of the tasks assigned to each server, assuming that each server is executed in isolation on a processor. The second stage examines whether there is enough capacity to accommodate all servers in the system.

We present each stage of the new demand-based overhead-aware schedulability analysis in its own subsection, but before that we provide an overview of the overheads that may be incurred by this class of scheduling algorithms.

4.1 Overheads

In order to carry out an overhead-aware schedulability analysis, we first need to identify the overheads that may be incurred at run-time because of the mechanisms used in the implementation of the scheduling algorithms. In this subsection, we provide an overview of the overheads that may arise in an implementation of a slot-based task-splitting scheduling algorithm. This overview is based on implementations [38,39] of S-EKG and NPS-F in the Linux kernel for the x64 architecture.

The overheads that a system may incur because of a scheduling algorithm are related to the following five mechanisms: (i) interrupts; (ii) timers; (iii) ready queues; (iv) context switching; and (v) caches. We examine the overheads of each mechanism in turn.

Most real-time systems interact with their environment and use interrupts whenever they need to react to external events. We assume that the interrupt handlers, or interrupt service routines, are implemented as tasks, as supported in the PREEMPT-RT Linux kernel [32]. Nevertheless, the occurrence of an interrupt suspends the execution of the currently running task to release a task that will service this interrupt. Furthermore, depending on the deadline of the released task, it may preempt the currently running task. A special kind of interrupt is the inter-processor interrupt (IPI). As its name suggests, these interrupts are generated by one processor and handled on another, and may be used by a processor to notify another of the occurrence of events. The processing of an IPI by the target processor is similar to that of an interrupt generated by the environment. Our algorithms use the IPI in the implementation of split servers, more specifically when a job, whose priority is higher than that of all ready jobs of its server, arrives on a processor at a time instant that falls within the reserve of that server in the other processor. In this case, the newly arrived job should immediately start execution in the server's reserve on the other processor. We denote the delay incurred by the use of IPI in the dispatching of a task the *IPI Latency*, *IpiL*.

Timers are a per-processor mechanism of the Linux kernel designed to schedule computations some time in the future. Our algorithms use timers to release tasks and also to trigger "server-switches" at the end of each time reserve. Timers are implemented using a priority queue and interrupts generated by some timer/counter device, therefore they incur overheads related to the handling of these interrupts as well. Timer interrupts are different from other interrupts in that they are not handled by separate tasks, but immediately upon occurrence of the interrupt. Thus, the expiration of a timer suspends the execution of the current task on that processor. Another "imperfection" associated with timers is that they cannot be used to measure time intervals precisely. We denote the delay incurred in the release of periodic tasks because of these imperfections the *Release Jitter*, *RelJ*.

The kernel keeps the released tasks that are ready to run in queues, known as ready queues. Therefore, when a task is released, the scheduler moves the task to a ready queue, and the dispatcher is invoked to select the next task to run, which may be either the task that was running before the release of the task, the released task or any other task that is ready to run. In the case of the slot-based task-splitting algorithms considered, all these data structures are either private to some processor or shared by two processors. Nevertheless, the release of a task requires some processing, which we call the *Release Overhead*, *RelO*.

A context switch occurs whenever the dispatcher decides to change the running task on a processor. This entails saving the state of the processor to some operating system data structure associated with the task being evicted, and restoring the state of the processor to the contents of the corresponding data structure associated with the task that was allocated the processor. We use the *Context switch Overhead*, *CtswO*, to account for this overhead.

The worst-case execution time of a task is very hard to estimate for processors with caches. For this reason, if any memory caching mechanism is used *at all*, locked caches or scratchpads are often used instead in the embedded domain, for better predictability [8,33]. Still, in this work we assume the use of conventional caches, as in general-purpose processors. For such architectures, the worst-case execution time of a task is typically computed assuming that the task is executed without being preempted. However, when a task is preempted, it may incur additional costs when it is resumed because the cache lines with its data may have been evicted by other tasks and need to be fetched again from main memory, or from higher cache levels. Likewise, migrating one task from one processor to another requires the destination processor to fetch anew the cache footprint of the task. These costs are known as *cache-related preemption and migration delays* (CPMD). To incorporate the CPMD, we pessimistically assume that every preemption incurs the worst-case CPMD cost, *CpmdO*. Furthermore, we do not distinguish between job preemption and job migration events. This simplification is not as pessimistic as it may seem because there is evidence [12,14] to suggest that, in a heavily loaded system, the

CPMD costs of preemptions and migrations can be similar. Another simplification is that we do not differentiate between tasks when applying CPMD costs; we use the same cost *CpmdO* irrespective of the preempted or the preempting task. Although some works [25,31] allow for estimating CPMD more precisely, they rely on detailed knowledge of the program code of each task and the memory layout.

Although in this subsection we have identified the different sources of overheads associated with slot-based task-splitting scheduling algorithms, in the analysis devised in the subsequent subsections, we sometimes lump together overheads of different sources that occur in sequence. The reasons for this are two-fold. First, this leads to shorter expressions. Second, it simplifies the experimental measurement of the overheads and often leads to more precise experimental estimates of these overheads.

4.2 New demand-based schedulability test for mapping tasks to servers

In this section, we derive a schedulability test for the tasks mapped to a server based on demand-bound functions that takes into account the overheads described in the previous subsection. This leads to a new task-to-server mapping algorithm. For the purpose of the mapping of tasks to servers, we consider that a server is allocated a processor exclusively, i.e. it runs on a single processor that it does not share with any other server. Hence, we treat each server as a uniprocessor system.

Our analysis is based on the concept of *demand-bound function* (dbf) [10], which specifies an upper bound on the aggregate execution requirements of all jobs (of $\tau[\tilde{P}_q]$) over any possible interval of length t. Therefore the demand-based schedulability test for a server $\tilde{P}_q$ is given by:

$$\text{dbf}^{\text{part}}(\tilde{P}_q, t) \leq t, \forall t > 0 \tag{10}$$

We use the word "part", which stems from "partitioned", as a superscript of all the dbfs of this stage to distinguish them from functions of the second stage.

Ignoring all overheads and assuming sporadic task sets with arbitrary deadlines, the $\text{dbf}^{\text{part}}(\tilde{P}_q, t)$ can be computed as:

$$\text{dbf}^{\text{part}}(\tilde{P}_q, t) = \text{dbf}^{\text{part}}(\tau[\tilde{P}_q], t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \cdot C_i \tag{11}$$

Next, we proceed by incorporating each source of overhead into the new overhead-aware schedulability analysis, one at a time. First, we consider the overheads caused by the release of tasks. We assume that all tasks are periodic, because it corresponds to the worst case. For periodic tasks we need to take into account not only the release overhead, but also the release jitter caused by timers. Therefore, the effects of timers and task release will be considered together. Next, we consider the effects of context switching and CPMD. Finally, we incorporate the effect of interrupts other than those caused by timers.

Scheduling algorithms use timers to trigger the release of periodic tasks. Therefore, the release of periodic tasks is affected by two of the overheads discussed in the previous section: the release overhead, and the release jitter. Figure 5 graphically shows these two overheads for job $\tau_{i,j}$. (In all figures, the execution of a job is graphically represented by a rectangle labelled with the job identifier.) As illustrated, the effects of these two overheads are different. Whereas both overheads, the release jitter of job $\tau_{i,j}$, $RelJ_{i,j}$, and the release overhead of job $\tau_{i,j}$, $RelO_{i,j}$, reduce the amount of time that job $\tau_{i,j}$ has to complete its execution, only the release overhead actually requires processing time. Thus, we model the effect of these two overheads differently.

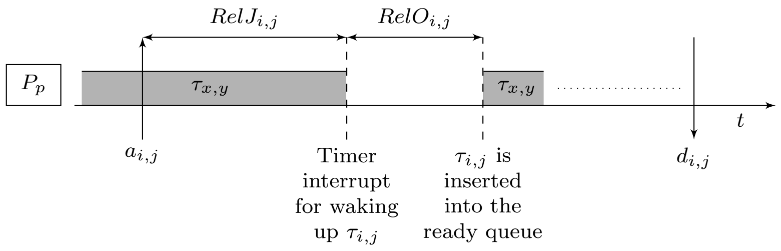
$RelJ_{i,j}$
$RelO_{i,j}$
P_p
$\tau_{x,y}$
$\tau_{x,y}$
t
$a_{i,j}$
Timer interrupt for waking up $\tau_{i,j}$
$\tau_{i,j}$ is inserted into the ready queue
$d_{i,j}$

Fig. 5: Illustration of the release jitter and release overhead. In this example, $\tau_{x,y}$ is the currently executing job that incurs an execution penalty due to the release overhead $RelO_{i,j}$ of job $\tau_{x,y}$.

Let $RelJ$ and $RelO$ be the upper bounds on the release latency and on the release overhead, respectively. As shown in Figure 5, the release latency decreases the amount of time available to complete a task, i.e., in the worst case, τ_i has $D_i - RelJ$ time units to complete. Therefore, we modify the $\text{dbf}^{\text{part}}(\tau[\tilde{P}_q], t)$ to:

$$\text{dbf}^{\text{part}}(\tau[\tilde{P}_q], t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - (D_i - RelJ)}{T_i} \right\rfloor + 1\right) \cdot C_i \qquad (12)$$

Concerning the release overhead, one way of modelling it could be by increasing the execution demand of a task accordingly. However, that approach does not work properly when multiple tasks are released too close together in time. The reason is that the release overhead contributes "immediately" to the processor demand – meaning that to model the processor demand correctly, it should be increased by $RelO$ time units at the time of the release, not at the deadline of the task released. Therefore, we instead model the release overhead as higher-priority interfering workload (as it is in reality). This way, we may compute the execution demand for releasing all jobs of $\tau[\tilde{P}_q]$ in a time interval $[1,\, t)$ as:

$$\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q], t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \left\lceil \frac{t + RelJ}{T_i} \right\rceil \cdot RelO \tag{13}$$

Modifying accordingly $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t)$, we get:

$$\begin{aligned} &\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t) = \\ &\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q], t) + \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i + RelJ}{T_i} \right\rfloor + 1\right) \cdot C_i \end{aligned} \tag{14}$$

We now consider the context switching overhead, which is common to all schedulers. Every job causes *at most* two context switches: when it is released and when it completes – but not every job release causes a context switch. Therefore the number of context switches over a time interval of length t is upper bounded by twice the number of job releases during that interval. Let $CtswO$ be an upper bound on the context switch overhead. We amend the derivation of the $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t)$, by increasing the execution demand of each job by twice $CtswO$, to:

$$\begin{aligned} &\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t) = \\ &\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tau[\tilde{P}_q], t) + \\ &\sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i + RelJ}{T_i} \right\rfloor + 1\right) \cdot (C_i + 2 \cdot CtswO) \end{aligned} \tag{15}$$

In order to incorporate the cache-related overheads, i.e. the CPMD, we assume that every preemption incurs the worst-case CPMD cost, $CpmdO$. Furthermore, we compute an upper bound on the number of preemptions for server $\tilde{P}_q$ in a time interval of length t as:

$$\mathrm{nr}^{\mathrm{part}}_{\mathrm{pree}}(\tilde{P}_q, t) = \sum_{\tau_i \in \tau[\tilde{P}_q]} \left\lceil \frac{t + RelJ}{T_i} \right\rceil \tag{16}$$

That is, we assume that every task that may be released by a timer in a time interval of length t, causes a preemption. Thus, the cumulative cost of CPMD over one interval of length t is:

$$\mathrm{dbf}^{\mathrm{part}}_{\mathrm{CpmdO}}(\tilde{P}_q, t) = \mathrm{nr}^{\mathrm{part}}_{\mathrm{pree}}(\tilde{P}_q, t) \cdot CpmdO \tag{17}$$

Because this increases the server execution demand, we amend the expression of the $\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t)$ (Equation 15) to:

$$\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t) = \mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{CpmdO}}(\tilde{P}_q, t) \tag{18}$$

In contrast with the other overheads, the cache related overheads cannot be assigned to a particular task. Indeed, the jobs of some tasks may never be preempted, whereas the jobs of other tasks may be preempted several times. This is the reason why we do not incorporate the CPMD overheads in $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t)$.

Finally, we consider the interrupt overheads. We assume that interrupt service tasks have higher priority than "normal" tasks. Thus, we model each sporadic interrupt as a task with worst-case execution time equal to C_i^{Int}, minimum inter-arrival time equal to T_i^{Int} and zero laxity ($C_i^{Int} = D_i^{Int}$). Periodic interrupts are also modelled as zero-laxity tasks, but T_i^{Int} represents their period and they are also characterized by a release latency L_i^{Int}, which accounts for deviations from strict periodicity. For sporadic interrupts, we let L_i^{Int} equal to zero, since any variability in their arrival pattern is already accounted for by T_i^{Int}. Thus the interrupt execution demand for n^{Int} interrupts is then given by:

$$\mathrm{dbf}^{\mathrm{part}}_{\mathrm{IntO}}(\tilde{P}_q, t) = \sum_{i=1}^{n^{Int}} \max\left(0, \left\lfloor \frac{t - D_i^{Int} + L_i^{Int}}{T_i^{Int}} \right\rfloor + 1\right) \cdot C_i^{Int} \tag{19}$$

Because the interrupt overhead increases the execution demand of a server, the $\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t)$, incorporating all the overheads, becomes:

$$\mathrm{dbf}^{\mathrm{part}}(\tilde{P}_q, t) = \mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{CpmdO}}(\tilde{P}_q, t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{IntO}}(\tilde{P}_q, t) \tag{20}$$

Equation 20 can be used in a new schedulability test by the algorithm that maps tasks to servers. Algorithm 1 shows the pseudo-code of this algorithm. The algorithm iterates over the set of all tasks and, for each task τ_i, it checks whether it fits in one of the opened servers (subject to the constraints of the bin-packing heuristics used, e.g., NF or FF). For each server $\tilde{P}_q$ checked (q being the server index), it provisionally adds task τ_i to it, then it computes the length of the testing time interval t (computed as twice the least-common multiple of the T_i of tasks in $\tau[\tilde{P}_q]$)[4], and finally, it applies the new schedulability test, by invoking the `dbf_part_check` function.

If the test succeeds for some server $\tilde{P}_q$, then task τ_i is permanently mapped to it, otherwise, a new server is opened and task τ_i is added to it. The task set is considered unschedulable whenever the schedulability test fails for a server with only one task.

This new algorithm is not applicable to S-EKG. In that case, for reasons that will be explained later, the task to server mapping and the server to processor assignment are performed in a single step using the algorithm that is outlined in Section 5.2.1.

To summarize, in this subsection we have developed a new overhead-aware analysis for schedulability testing in the task-to-server mapping stage. However, this test considers each server in isolation and it does not encompass all

[4] Approaches exist for considerably reducing the length of the testing interval t [22,40,34,23] in order to speed up the schedulability test, but would have required some amendments, in the presence of the scheduling overheads considered.

Algorithm 1 Pseudo-code of the new task-to-server mapping algorithm.

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

$k \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    $scheduled \leftarrow 0$
    for $q \leftarrow 1$ to $k$ do
        add_task_to_server($\tau_i, \tilde{P}_q$)
        $t \leftarrow 2 \cdot$ lcm_T($\tilde{P}_q$)
        if dbf_part_check($\tilde{P}_q, t$) then
            $scheduled \leftarrow 1$
            break
        else
            remove_task_from_server($\tau_i, \tilde{P}_q$)
        end if
    end for
    if $scheduled = 0$ then
        $k \leftarrow k + 1$ {add a new server}
        add_task_to_server($\tau_i, \tilde{P}_k$)
        $t \leftarrow 2 \cdot$ lcm_T($\tilde{P}_q$)
        if not dbf_part_check($\tilde{P}_k, t$) then
            $k \leftarrow 0$
            break {failure}
        end if
    end if
end for
```

the scheduling overheads that may be incurred by servers when they share a processor with other servers. In the next subsection, we develop a new schedulability analysis for the processor-to-server assignment step.

4.3 New demand-based schedulability test for assigning servers to processors

To fully model all the overheads incurred by the use of periodic reserves, it is necessary to assign each server to one or more processors. Precisely modelling the impact of these overheads allows us to determine the *exact* processing capacity requirements of each server. In turn, this allows us to test whether or not all servers can be accommodated on the m physical processors.

With the server-to-processor assignment described in Section 3, non-split servers are allocated just one processor reserve whereas split-servers must be allocated two reserves. Because, each type of server incurs different overheads, we deal with each type of server separately.

4.3.1 Non-split servers

The approach we follow to check the schedulability of a server is to verify that the execution demand by all jobs assigned to a server (computed using the dbf) does not exceed the amount of time (computed using the *supply-bound* *function* (sbf)) that the system can provide for their execution, for every time interval of length t. Formally, we can express this schedulability test as:

$$\text{dbf}^{\text{sb:non-split}}(\tilde{P}_q, t) \leq \text{sbf}^{\text{sb:non-split}}(\tilde{P}_q, t), \forall t > 0 \tag{21}$$

where we use the superscript "sb" (an abbreviation for "slot based") to distinguish the functions/variables used in this subsection from similar functions/variables used in the previous subsection. This superscript may be suffixed with either ":non-split" or ":split", depending on whether the function/variable applies to non-split servers or to split servers, respectively.

We develop an analysis that allows us to apply the schedulability test in Equation 21 to non-split servers in two steps. First, we revisit the analysis developed in Section 4.2 to take into account the effect of the reserve mechanism on the computing demand of a non-split server. Second, we factor into our analysis the effect of the reserve mechanism on the computing supply of a non-split server.

In Equation 20, we decomposed the demand of a server, $\text{dbf}^{\text{part}}(\tilde{P}_q, t)$, into three components. The first, $\text{dbf}^{\text{part}}(\tau[\tilde{P}_q], t)$, comprises the execution requirements induced by each task mapped to server $\tilde{P}_q$, including not only its execution time, but also overheads that may arise because of mechanisms used by the scheduling algorithm, i.e. timers, task releases and context switches. Clearly, these requirements are not affected by the use of reserves. However, now we also need to take into account the *Release Interference*, $\text{dbf}^{\text{sb:non-split}}_{\text{RelI}}(\tilde{P}_q, t)$, i.e. the overhead incurred by the release of tasks mapped to other servers that share the processor with $\tilde{P}_q$. Furthermore, as we explain below, the other two components are also affected by the use of reserves. Hence, in a first approximation, we have:

$$\begin{aligned}
&\text{dbf}^{\text{sb:non-split}}(\tilde{P}_q, t) = \\
&\quad \text{dbf}^{\text{part}}(\tau[\tilde{P}_q], t) + \text{dbf}^{\text{sb:non-split}}_{\text{CpmdO}}(\tilde{P}_q, t) \\
&\quad + \text{dbf}^{\text{sb:non-split}}_{\text{IntO}}(\tilde{P}_q, t) + \text{dbf}^{\text{sb:non-split}}_{\text{RelI}}(\tilde{P}_q, t)
\end{aligned} \tag{22}$$

We now proceed with the development of the analytical expressions for the $\text{dbf}^{\text{sb:non-split}}$ parameters on the right-hand side of Equation 22.

The CPMD overheads now comprise not only the preemptions caused by tasks in the server, but also the preemptions incurred due to the reserve mechanism. In the worst case, the reserve mechanism preempts the last job that executes in the server's reserve. Thus, during an interval of duration S, a non-split server incurs at most one additional preemption due to the use of reserves:

$$\text{nr}^{\text{sb:non-split}}_{\text{pree}}(\tilde{P}_q, t) = \left\lceil \frac{t + ResL}{S} \right\rceil + \text{nr}^{\text{part}}_{\text{pree}}(\tilde{P}_q, t) \tag{23}$$

where $ResL$, the reserve latency, is an overhead akin to the release overheads that occurs at the beginning of a reserve and is explained later in this subsection.

Accordingly, the worst-case overall CPMD cost for that server in a time interval of length t is given by:

$$\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{CpmdO}}(\tilde{P}_q, t) = \mathrm{nr}^{\mathrm{sb:non-split}}_{\mathrm{pree}}(\tilde{P}_q, t) \cdot CpmdO \tag{24}$$

Taking into account interrupts with reserves is somewhat harder than in the case of a uniprocessor. Indeed, whereas on a uniprocessor a sporadic interrupt can be modelled as a sporadic interfering task, this is not the case with reserves. This is because reserve boundaries behave like temporal firewalls, and therefore an interrupt affects only the reserve that was active at the time the interrupt task is executed. Hence, each interrupt has to be modelled as a bursty periodic task. Given the complexity of such a formulation, we deal with it in Appendix A. Let $\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{IntO}}(\tilde{P}_q, t)$ denote the amount of time required for executing all fired interrupts inside the reserves of $\tilde{P}_q$ in a time interval of length t, as determined in Appendix A.

Finally, we consider the release overhead, i.e. the processor time required to handle the release of jobs. On slot-based task-splitting algorithms, a server's tasks share the processor with other tasks whose servers are assigned to the same processor. Consistent with implementation [38,39] we assume that all jobs of a task are released on the processor(s) to which the task is assigned. As shown in Figure 6, non-split server $\tilde{P}_q$ can incur not only the release overheads of its own jobs, but also the release overheads of the jobs of both its immediate neighbor servers ($\tilde{P}_{q-1}$ and $\tilde{P}_{q+1}$).

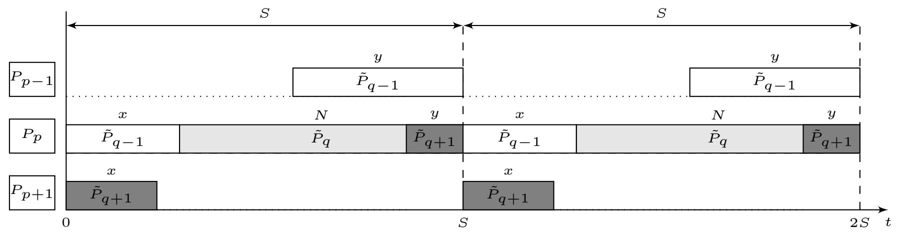

Fig. 6: Illustration of the release interference for non-split servers. In this example, server $\tilde{P}_q$ may suffer the interference from the release of tasks mapped to $\tilde{P}_{q-1}$ and $\tilde{P}_{q+1}$, if these releases occur within $\tilde{P}_q$'s reserve.

Recall that the release overhead cost of all jobs of $\tau[\tilde{P}_q]$ in a time interval of length t is already accounted for in the derivation of $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t)$ (see Equation 14). Therefore, what remains is to incorporate the release interference, $\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{RelI}}(\tilde{P}_q, t)$, the release overhead cost from neighboring servers, i.e. servers sharing the same processor:

$$\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{RelI}}(\tilde{P}_q, t) = \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q-1}, t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q+1}, t) \tag{25}$$

where $\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_q, t)$ (see Equation 13) denotes the amount of time required to release all jobs of server $\tilde{P}_q$ in a time interval of length t.

We now consider the effect of the reserve mechanism on the amount of time supplied to the execution of the tasks of a non-split server. In comparison with the analysis in Section 4.2, the amount of time supplied to the execution of a non-split server is reduced because of two factors. The first is the sharing of the processor with other servers. The second is the imprecision of the timers used to measure the duration of the reserves. We analyze the effect of each of these factors in turn.

In slot-based task-splitting algorithms, a non-split server $\tilde{P}_q$ is confined to execute within a single periodic reserve of length $Res^{len}[\tilde{P}_q]$, which is available every S time units:

$$Res^{len}[\tilde{P}_q] = U^{infl}[\tilde{P}_q] \cdot S \tag{26}$$

where $U^{infl}[\tilde{P}_q]$ represents the inflated processing capacity of server $\tilde{P}_q$, which is computed by Algorithm 2 presented at the end of this subsection. Thus, for any time interval of length t, only a fraction of such interval is supplied for the execution of a server. We model the unavailability of the reserve as an interfering *fake task* with attributes:

$$\begin{aligned} C^{fake} &= S - Res^{len}[\tilde{P}_q] \\ T^{fake} &= S \\ D^{fake} &= C^{fake} \end{aligned} \tag{27}$$

Hence, the supply-bound function for non-split servers can be expressed, in a first approximation, as follows:

$$\mathrm{sbf}^{\mathrm{sb:non-split}}(\tilde{P}_q, t) = t - \max\left(0, \left\lfloor \frac{t - D^{fake}}{T^{fake}} \right\rfloor + 1\right) \cdot C^{fake} \tag{28}$$

The second source of the reduction in the amount of time supplied to the execution of a non-split server is the processing time required to switch from one reserve to the next, which also includes the execution of the scheduler. Furthermore, the switch of reserves is also associated with a delay between the time at which the current reserve should end and the time at which it actually ends, for example because the processor is executing a non-preemptible code segment. To facilitate the experimental measurement of this overhead, we decide to group these three parameters in a single one that we call *reserve latency*. This is illustrated in Figure 7, which also shows that this parameter includes the time required to switch to the first job of the new reserve.

We model this reduction in the supply of processing time to the reserve as an increase in the execution demand of the fake task. Let $ResL$ be an upper bound for the reserve latency. The expression for $\mathrm{sbf}^{\mathrm{sb:non-split}}(\tilde{P}_q, t)$ then becomes:

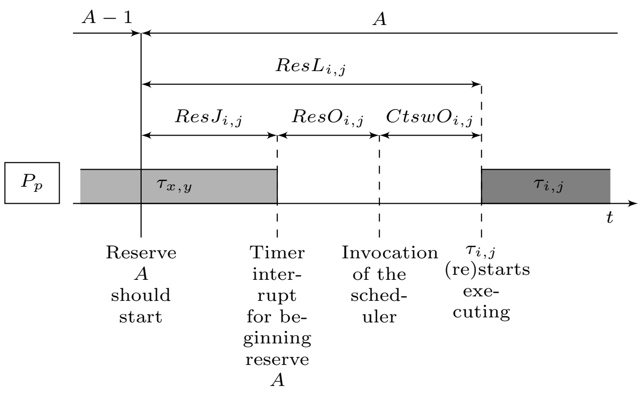

Fig. 7: Illustration of the reserve overhead. The execution of job $\tau_{i,j}$ of server mapped to reserve A, is delayed by $ResL$ with respect to the instant the reserve should start.

$$\text{sbf}^{\text{sb:non-split}}(\tilde{P}_q, t) = t - \max\left(0, \left\lfloor \frac{t - D^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) \tag{29}$$

By replacing this expression in Inequality 21 and moving some terms from the right-hand side to the left-hand side, we obtain the following schedulability test for non-split servers:

$$\text{dbf}^{\text{sb:non-split}}(\tilde{P}_q, t) + \text{dbf}^{\text{sb:non-split}}_{\text{Fake}}(\tilde{P}_q, t) \leq t, \forall t > 0 \tag{30}$$

where $\text{dbf}^{\text{sb:non-split}}(\tilde{P}_q, t)$ is given by Equation 22 and $\text{dbf}^{\text{sb:non-split}}_{\text{Fake}}(\tilde{P}_q, t)$ is given by:

$$\text{dbf}^{\text{sb:non-split}}_{\text{Fake}}(\tilde{P}_q, t) = \max\left(0, \left\lfloor \frac{t - D^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) \tag{31}$$

To complete the analysis of non-split servers, we provide an algorithm to compute the inflated utilization of server $\tilde{P}_q$, $U^{infl}[\tilde{P}_q]$. Indeed, evaluating $\text{dbf}^{\text{sb:non-split}}_{\text{Fake}}(\tilde{P}_q, t)$ depends on $U^{infl}[\tilde{P}_q]$, via $Res^{len}[\tilde{P}_q]$ and C^{fake} (see Equations. 26 and 27). Furthermore, $\text{dbf}^{\text{sb:non-split}}_{\text{IntO}}(\tilde{P}_q, t)$ also depends on $U^{infl}[\tilde{P}_q]$, as shown in Appendix A.

In order to achieve the highest possible schedulability, we are interested in determining the minimum inflated utilization required for server $\tilde{P}_q$. We use the schedulability test developed in this section to determine an interval that is guaranteed to include the inflated utilization. This interval can be arbitrarily small. We start with the interval $[U[\tilde{P}_q], 1.0]$. Then, like in the bisection method, we successively halve this interval in such a way that the inflated utilization is guaranteed to be in every generated interval. Algorithm 2 shows the pseudo-code for the `inflate_sb_non_split` function. In each iteration, it computes the current interval's midpoint and then applies the schedulability test, implemented in the `dbf_sb_non_split_check` function, to that utilization value. If the outcome of the test is positive, i.e. the server is schedulable with that utilization, the midpoint value computed becomes the upper bound of the interval in the next iteration, otherwise it becomes the lower bound. The algorithm converges rather rapidly, and in ten iterations, it generates an interval that is less than 0.1% wide that contains the minimum inflated capacity of the server required for the server to be schedulable, according to the schedulability test in Inequality 30. In Section 6, we provide some details on the implementation of the `dbf_sb_non_split_check` function.

Algorithm 2 Pseudo-code algorithm of the `inflate_sb_non_split` function.

```
Inputs: $\tilde{P}_q$ {server to analyse}
        $\Delta$ {desired precision}
        $t$ {time interval for computing the demand-bound function}
Outupt: $U^{infl}[\tilde{P}_q]$ {minimum inflated utilization that ensures schedulability of $\tilde{P}_q$}

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

4.3.2 Split servers

In this subsection, we develop a schedulability analysis for split-servers similar to the one developed in the previous subsection. Again, we use a schedulability test based on the demand-bound and the supply-bound functions:

$$\mathrm{dbf}^{\mathrm{sb:split}}(\tilde{P}_q, t) \leq \mathrm{sbf}^{\mathrm{sb:split}}(\tilde{P}_q, t), \forall t > 0 \tag{32}$$

and we derive the expression for $\mathrm{dbf}^{\mathrm{sb:split}}(\tilde{P}_q, t)$, by revisiting the analysis developed in Section 4.2 to take into account the increase in the demand of processing time because of the reserve mechanism, and the expression for $\mathrm{sbf}^{\mathrm{sb:split}}(\tilde{P}_q, t)$, by accounting for the reduction in the amount of time supplied to the server because of the reserve mechanism.

Based on the arguments used in the previous subsection, we can express $\mathrm{dbf}^{\mathrm{sb:split}}(\tilde{P}_q, t)$ as follows:

$$\begin{aligned} &\mathrm{dbf}^{\mathrm{sb:split}}(\tilde{P}_q, t) = \\ &\quad \mathrm{dbf}^{\mathrm{sb:split}}(\tau[\tilde{P}_q], t) \\ &\quad +\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{CpmdO}}(\tilde{P}_q, t) + \mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{IntO}}(\tilde{P}_q, t) + \mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{RelI}}(\tilde{P}_q, t) \end{aligned} \tag{33}$$

That is, like with non-split servers, the preemptions and migrations of tasks, the interrupts and the release of tasks of servers that share processors with the split server also need to be taken into account, and amended specifically to split servers. However, unlike with non-split servers, we also need to amend $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t)$, i.e. the processor demand of the server's tasks, assuming that they are executed in their own processor and accounting for the overheads incurred by the timers, the release of the server's tasks and the context switches between server's tasks. This is because the release of tasks of a split server may use IPI, which, as we show below, affects the components of the demand accounted for in $\mathrm{dbf}^{\mathrm{part}}(\tau[\tilde{P}_q], t)$. We now develop an expression for each term in Equation 33.

As described in Section 4.1, slot-based task-splitting algorithms may use IPIs to notify the dispatcher in another processor of the release of a task. As a result, the dispatching of a task may incur an *IPI Latency.* (Note that this parameter does not include the time required for context switching, this is already accounted for, as it will occur whether or not the release is via an IPI.) Figure 8 illustrates such a case. The arrival of a job of task τ_i assigned to a split server shared between processors P_p and P_{p-1}, for instance, occurs at a time instant t and is handled on processor P_p, but this time instant t falls inside the reserve of that server on the other processor, P_{p-1}. If this job is the highest priority job of its server, P_p notifies P_{p-1} of the new arrival via an IPI. Clearly, the overhead caused by the IPI, $IpiL_{i,j}$, only delays the dispatch of job $\tau_{i,j}$ (and only if job $\tau_{i,j}$ is the highest priority job of its server).

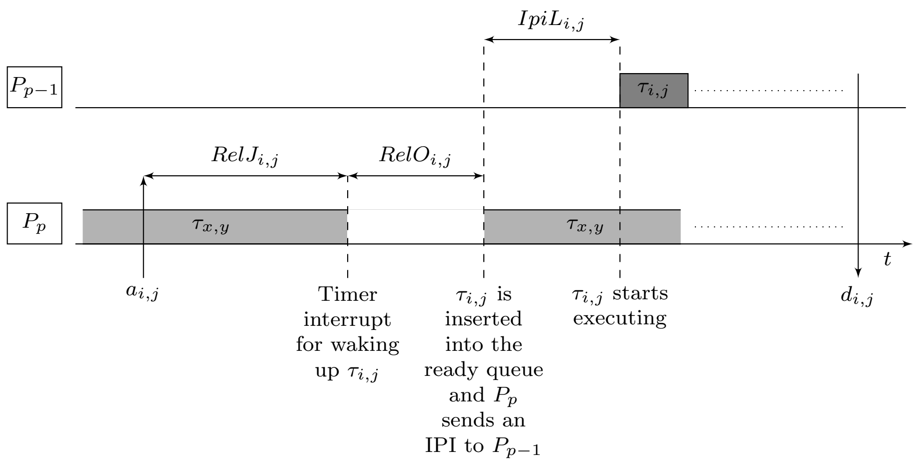
$IpiL_{i,j}$
P_{p-1}
$\tau_{i,j}$
$RelJ_{i,j}$
$RelO_{i,j}$
P_p
$\tau_{x,y}$
$\tau_{x,y}$
t
$a_{i,j}$
Timer interrupt for waking up $\tau_{i,j}$
$\tau_{i,j}$ is inserted into the ready queue and P_p sends an IPI to P_{p-1}
$\tau_{i,j}$ starts executing
$d_{i,j}$

Fig. 8: Illustration of the IPI latency at the release of a split job. It does not include the time for context switching.

Thus, the IPI latency has an effect similar to the release jitter and we take it into account by adding it to the release jitter in $\text{dbf}^{\text{part}}(\tau[\tilde{P}_q], t)$, see Equation 15:

$$\begin{aligned}
&\text{dbf}^{\text{sb:split}}(\tau[\tilde{P}_q], t) = \\
&\ \text{dbf}^{\text{part}}_{\text{RelO}}(\tau[\tilde{P}_q], t) \\
&\ + \sum_{\tau_i \in \tau[\tilde{P}_q]} \max\left(0, \left\lfloor \frac{t - D_i + RelJ + IpiL}{T_i} \right\rfloor + 1\right) \\
&\ \cdot (C_i + RelO + 2 \cdot CtswJ)
\end{aligned} \tag{34}$$

where $IpiL$ is an upper bound for the IPI latency.

The cost of the CPMD is more of a concern for split servers than for non-split servers, because tasks may actually migrate between two processors. Nevertheless, in our analysis, we assume a worst-case CPMD overhead, $CpmdO$, which accounts for both. Hence, compared with modelling CPMD overheads for non-split servers, the only difference is that other than EDF preemptions, split servers incur two additional preemptions per time slot (*vs.* one for non-split servers), one for each reserve they use. Accordingly, $\text{nr}^{\text{sb:split}}_{\text{pree}}(\tilde{P}_q, t)$ is calculated as follows:

$$\text{nr}^{\text{sb:split}}_{\text{pree}}(\tilde{P}_q, t) = 2 \cdot \left\lceil \frac{t + ResL}{S} \right\rceil + \text{nr}^{\text{part}}_{\text{pree}}(\tilde{P}_q, t) \tag{35}$$

and the cost of the CPMD over a time interval of length t is:

$$\text{dbf}^{\text{sb:split}}_{\text{CpmdO}}(\tilde{P}_q, t) = \text{nr}^{\text{sb:split}}_{\text{pree}}(\tilde{P}_q, t) \cdot CpmdO \tag{36}$$

The interrupt overhead for split servers is modelled as for non-split servers; that is, each interrupt is modelled as bursty periodic task. Given the complexity of such a formulation we deal with that in Appendix A. Let $\text{dbf}^{\text{sb:split}}_{\text{IntO}}(\tilde{P}_q, t)$ be an upper bound on the amount of time required for executing all fired interrupts inside the reserves of $\tilde{P}_q$ in a time interval of length t.

Finally, we consider the release interference by servers that execute on the same processor. As illustrated in Figure 9, a split server, $\tilde{P}_q$, can incur the release interference of, at most, the previous two ($\tilde{P}_{q-1}$ and $\tilde{P}_{q-2}$) and also, at most, the next two servers ($\tilde{P}_{q+1}$ and $\tilde{P}_{q+2}$).

Thus, the release interference on $\tilde{P}_q$ by its neighbor servers is computed as:

$$\begin{aligned}
&\text{dbf}^{\text{sb:split}}_{\text{RelI}}(\tilde{P}_q, t) = \\
&\ \text{dbf}^{\text{part}}_{\text{RelO}}(\tilde{P}_{q-1}, t) + \text{dbf}^{\text{part}}_{\text{RelO}}(\tilde{P}_{q-2}, t) \\
&\ + \text{dbf}^{\text{part}}_{\text{RelO}}(\tilde{P}_{q+1}, t) + \text{dbf}^{\text{part}}_{\text{RelO}}(\tilde{P}_{q+2}, t)
\end{aligned} \tag{37}$$

This concludes the analysis of the effect of the reserve mechanism on the processing demand by a split-server. Before we analyze the effect of the reserve

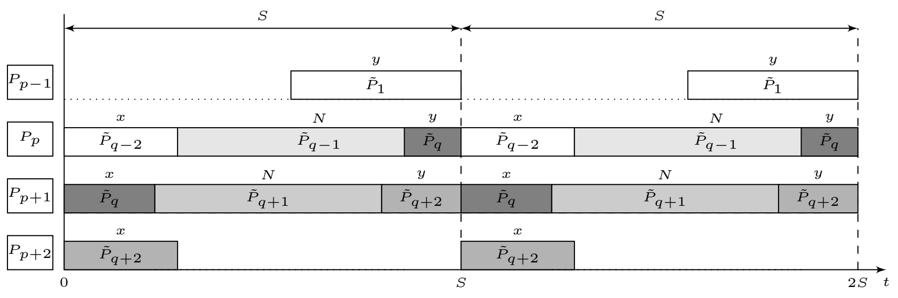

Fig. 9: Illustration of the potential release interference exerted by neighboring servers. In this example, server $\tilde{P}_q$ may suffer the release interference not only from the tasks in $\tilde{P}_{q-1}$ and $\tilde{P}_{q-2}$, if these tasks are released during $\tilde{P}_q$'s reserve on processor P_p, but also from tasks in $\tilde{P}_{q+1}$ and $\tilde{P}_{q+2}$, if these tasks are released during $\tilde{P}_q$'s reserve on processor P_{p+1}).

mechanism on the amount of time supplied to the server, $\text{sbf}^{\text{sb:split}}(\tilde{P}_q, t)$, we need to provide an implementation detail that we omitted in our short description of the assignment of reserves to processors in Section 3. In that description, the reserves of a split server $\tilde{P}_q$ on different processors P_p and P_{p+1} are temporally adjacent, as illustrated in Figure 10. In practice, because of the limitations in the measurement of the duration of a reserve, this layout requires explicit synchronization between the dispatchers on both processors to prevent simultaneous execution of the same task by both processors at the beginning of a time slot. This synchronization would lead to an additional overhead, which can be avoided by shifting the beginning of the time slot on processor P_{p+1} in time, i.e. by staggering the time slots in consecutive processors.

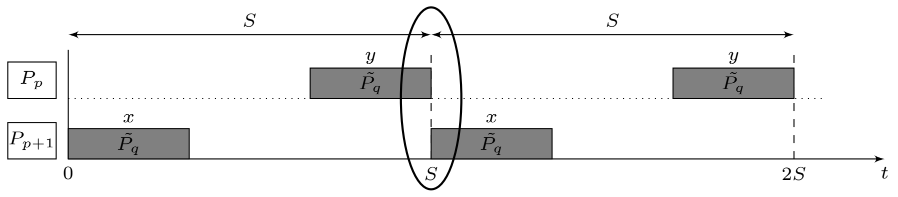

Fig. 10: Illustration of the adjacent time slots that cause the instantaneous migration problem.

In [17], the authors have shown that the time shift Ω given by

$$\Omega = (S - x[P_{p+1}] - y[P_p])/2 \tag{38}$$

is optimal [5] with respect to utilization for a split server $\tilde{P}_q$ whose reserves are $x[P_{p+1}]$ and $y[P_p]$. With this value, the end of $x[P_{p+1}]$ is also separated from the start of $y[P_p]$ by the same Ω time units, as illustrated in Figure 11. Therefore, Ω is also optimal with respect to the reserve overhead tolerated, i.e., it is the time shift that provides the maximum protection against race conditions caused by the reserve jitter that may arise among schedulers of processors with reserves that are mapped to the same server.

Although this result was formulated in the context of NPS-F, it applies to any slot-based task-splitting scheduling algorithm. Therefore in our analysis, we assume that the two reserves of a split-server $\tilde{P}_q$ are Ω apart of each other.

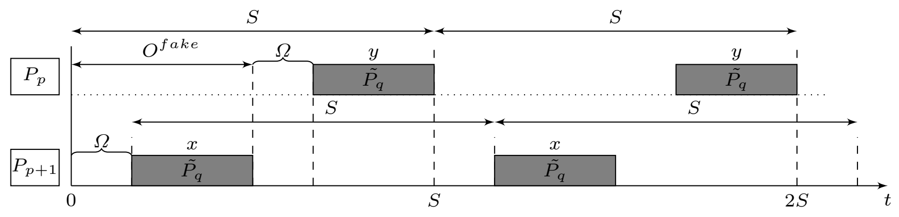

Fig. 11: The instantaneous migration problem can be prevented by shifting the time slot. This does not affect the processing time supplied to non-split servers, because their reserves are shifted as well.

We now proceed with the development of the reduction in the time supplied to execute the tasks of a split server because of the reserve mechanism.

Let $U_x^{infl}[\tilde{P}_q]$ and $U_y^{infl}[\tilde{P}_q]$ be the fractions of $U^{infl}[\tilde{P}_q]$ assigned to processors P_{p+1} and P_p, respectively. Then the duration of the $\tilde{P}_q$ reserves are given by:

$$\begin{aligned} U^{infl}[\tilde{P}_q] &= U_x^{infl}[\tilde{P}_q] + U_y^{infl}[\tilde{P}_q] \\ x[P_{p+1}] &= U_x^{infl}[\tilde{P}_q] \cdot S \\ y[P_p] &= U_y^{infl}[\tilde{P}_q] \cdot S \end{aligned} \tag{39}$$

As in the analysis for non-split servers, we model the unavailability of the processor outside the reserves with fake tasks, now two per time slot, each with the following parameters:

$$\begin{aligned} C^{fake} &= \Omega \\ T^{fake} &= S \\ D^{fake} &= C^{fake} \end{aligned} \tag{40}$$

[5] That proof assumed implicit-deadline tasks; proof for arbitrary deadlines has not yet been published. In any case, in this work, we set Ω accordingly.

Although the two fake tasks have the same arrival rate, they arrive at a relative offset. To account for the worst case, we assume that the first fake task arrives at t equal to zero and the second task arrives at an offset of:

$$O^{fake} = \Omega + \min(U_x^{infl}[\tilde{P}_q], U_y^{infl}[\tilde{P}_q]) \cdot S \tag{41}$$

Thus we can express the amount of time supplied to the execution of tasks of the split server, $\mathrm{sbf}^{\mathrm{sb:split}}_{\mathrm{Fake}}(\tilde{P}_q, t)$, as:

$$\begin{aligned} &\mathrm{sbf}^{\mathrm{sb:split}}(\tilde{P}_q, t) = \\ &\quad t \\ &\quad - \max\left(0, \left\lfloor \frac{t - D^{fake}}{T^{fake}} \right\rfloor + 1\right) \cdot C^{fake} \\ &\quad - \max\left(0, \left\lfloor \frac{t - D^{fake} - O^{fake}}{T^{fake}} \right\rfloor + 1\right) \cdot C^{fake} \end{aligned} \tag{42}$$

Like the reserve of a non-split server, each of the two reserves of a split server incurs the reserve overhead. Let $ResL$ be an upper bound for the reserve latency. Thus, to take into account this overhead, we do just as in the case of non-split servers, i.e. we add $ResL$ to the execution demand of each of the two fake tasks, and $\mathrm{sbf}^{\mathrm{sb:split}}(\tilde{P}_q, t)$ becomes:

$$\begin{aligned} &\mathrm{sbf}^{\mathrm{sb:split}}(\tau[\tilde{P}_q], t) = \\ &\quad t \\ &\quad - \max\left(0, \left\lfloor \frac{t - D^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) \\ &\quad - \max\left(0, \left\lfloor \frac{t - D^{fake} - O^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) \end{aligned} \tag{43}$$

Replacing this expression in Inequality 32 and moving some terms from the right-hand side to the left-hand side, we obtain the following schedulability test for split servers:

$$\mathrm{dbf}^{\mathrm{sb:split}}(\tilde{P}_q, t) + \mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Fake}}(\tilde{P}_q, t) \leq t, \forall t > 0 \tag{44}$$

where $\mathrm{dbf}^{\mathrm{sb:split}}(\tilde{P}_q, t)$ is given by Equation 33 and $\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Fake}}(\tilde{P}_q, t)$ is given by:

$$\begin{aligned} &\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Fake}}(\tilde{P}_q, t) = \\ &\quad \max\left(0, \left\lfloor \frac{t - D^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) \\ &\quad + \max\left(0, \left\lfloor \frac{t - D^{fake} - O^{fake} + ResL}{T^{fake}} \right\rfloor + 1\right) \cdot (C^{fake} + ResL) \end{aligned} \tag{45}$$

To complete the analysis of split servers, we provide an algorithm to compute the inflated utilization of server $\tilde{P}_q$, $U^{infl}[\tilde{P}_q]$. Indeed, evaluating $\text{dbf}^{\text{sb:split}}_{\text{Fake}}(\tilde{P}_q, t)$ depends on $U^{infl}[\tilde{P}_q]$, via $x[P_{p+1}]$, $y[P_{p+1}]$, Ω, C^{fake} and O^{fake}(see Equations 38, 39, 40, and 41). Furthermore, $\text{dbf}^{\text{sb:split}}_{\text{IntO}}(\tilde{P}_q, t)$ also depends on $U^{infl}[\tilde{P}_q]$, as shown in Appendix A.

In order to achieve the highest possible schedulability, we are interested in determining the minimum inflated utilization required for server $\tilde{P}_q$. The algorithm we use for split servers is similar to that used for non-split servers, presented in Algorithm 2, except that it uses the function `dbf_sb_split_check`, which implements the schedulability test in Equation 44, rather than function `dbf_sb_non_split_check`. In Section 6, we provide some details on the implementation of these functions.

5 New server-to-processor assignment procedure

The application of the schedulability tests developed in the previous section raises two main issues. First, computing the inflation of the utilization of each server requires knowledge of whether or not the server is split, and of which servers it shares the processor with. However, this depends on the inflated utilization of the server. In other words, there is a circular dependency between server inflation and the assignment of servers to processors. Second, when a server is split between two processors, the length of the reserves may be either too short or too long, for example larger than S. To prevent this undesirable outcome, we specify two assignment rules, which further exacerbate the first issue. Thus, in this section, we start by describing the assignment rules. After that, we address how to resolve the circularity associated with the first issue.

5.1 Assignment rules

To prevent reserves too short to be useful, we add the following rule to the assignment algorithms that are presented below:

A1: Whenever a server would be split between two processors P_p and P_{p+1} in such a way that the length of the second reserve (i.e. on P_{p+1}) would be larger than the length of its only reserve had it been assigned to a single reserve on P_{p+1}, then the server should not be split, but rather assigned as non-split to P_{p+1}.

Figures 12 illustrates rule A1 using aligned time slots for reasons of clarity.

Clearly, if the size of the non-split reserve is smaller than that of the second reserve, not splitting the server will lead to a lower computation demand by the server in both the first and the second processor. This means that there will be more computation resources for the remaining servers in the second processor. Although the computation resources not used on the first processor

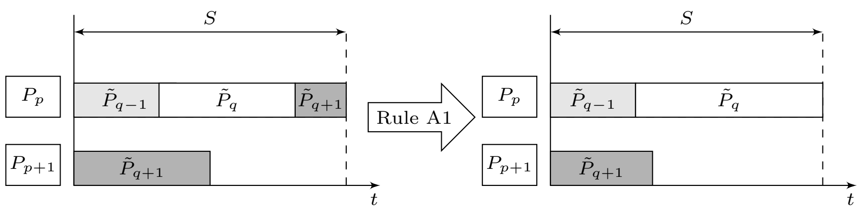

Fig. 12: Illustration of assignment rule A1.

will not be used to satisfy the demand of the task set to schedule, they can be used by other (non-real time) tasks.

On the other hand, if the second reserve of the split server is shorter than the single reserve required if the server were not split, it must be the case that the first reserve is used for satisfying the demand of the server's tasks, and therefore, for the sake of improving the schedulability, the server should be split.

Another issue concerns the case when the two reserves of a split server (possibly after application of rules A1) add up to almost S, or even surpass it. As a result, the schedulers on two processors might attempt to run the same task simultaneously. To prevent such a scenario, we specify the following rule:

A2: In cases where a server would be split such that $(U_x^{infl}[\tilde{P}_q] + U_y^{infl}[\tilde{P}_q]) \cdot S > S - ResL$, the server should instead become a *single server*.

A single server is assigned to a processor, utilizing its entire processing capacity, without being confined in a time reserve. This arrangement amounts to partitioning. Figure 13 illustrates rule A2.

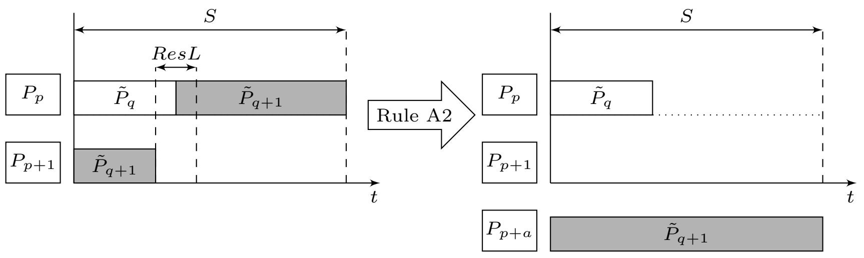

Fig. 13: Illustration of assignment rule A2.

5.2 Assignment procedure

Section 3 suggests that server-to-processor assignment is straightforward once the servers have been inflated. However, with the schedulability tests developed in the previous section, this is not so. The challenge is that server inflation depends on the assignment of servers to processors, because the release interference overhead depends on which servers are allocated the same processor. Therefore, we have a circularity issue: inflation depends on the assignment, and the assignment depends on the inflation. For example, when we first inflate a server $\tilde{P}_{q-1}$, we do not yet know the servers that it will share a processor with. We can assume that the next server, $\tilde{P}_q$, will share the processor with the server currently being analyzed, but later, because of the application of rule A2, server $\tilde{P}_q$ may be allocated its own processor (as a single server), and therefore server $\tilde{P}_{q-1}$ will share the processor not with that server but with the one that follows it, i.e. server $\tilde{P}_{q+1}$, and it will have to be re-inflated.

The approach we use to overcome this issue is backtracking. To limit the amount of backtracking, we merge several steps of the generic algorithm in a single step. In the next two subsubsections, we illustrate the application of this approach to S-EKG and to NPS-F, respectively.

5.2.1 Task-to-processor assignment procedure in S-EKG

The distinctive feature of S-EKG is that the split servers, if any, have only one task. To ensure this, we merge the four steps of the generic algorithm in a single one. The full algorithm is somewhat complex, therefore, we just provide an overview of its main steps, which are illustrated in Figure 14.

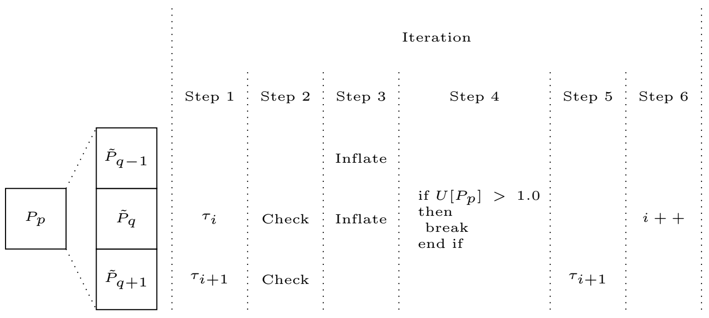

Fig. 14: New S-EKG task-to-processor mapping algorithm.

The algorithm starts by assigning empty servers to the processors. All processors are assigned a non-split server, one split server per predecessor processor and one split server per successor processor, so that the first and the last processors are assigned only two servers, whereas the other processors are assigned three servers. Then, it iterates over the set of tasks, two tasks at time, if available, and it assigns the tasks to the servers in an attempt to maximize the utilization of each server, subject to the constraint that each split server has at most one task. In the first step (Step 1), it provisionally assigns tasks τ_i and τ_{i+1} to $\tilde{P}_q$, the non-split server, and $\tilde{P}_{q+1}$, the split-server shared with the next processor, respectively, by invoking the `add_task_to_server` function. Then, it checks (Step 2) the schedulability of each server by invoking the `dbf_part_check` function. If some server with only one task is not schedulable, then the task set is also not schedulable. Otherwise, if the non-split server is not schedulable, the algorithm backtracks and assigns τ_i to $\tilde{P}_{q+1}$, and moves to the next iteration (where it will map tasks τ_{i+1} and τ_{i+2} to servers $\tilde{P}_{q+2}$ and $\tilde{P}_{q+3}$, respectively, and check their schedulability). If both servers are schedulable, it proceeds by inflating (Step 3) the capacity of the previous, $\tilde{P}_{q-1}$, and the current, $\tilde{P}_q$, servers by invoking the `inflate_sb_split()` and `inflate_sb_non_split()` functions, respectively. It then checks (Step 4), if $U[P_p](= U_x^{infl}[\tilde{P}_{q-1}] + U^{infl}[\tilde{P}_q])$ is larger than 1.0. If yes, then it proceeds as in Step 2, when the non-split server is not schedulable. Otherwise, (Step 5) it assigns τ_i permanently to $\tilde{P}_q$, removes τ_{i+1} from $\tilde{P}_{q+1}$ server, and moves to the next iteration (Step 6), in which it will attempt to map task τ_{i+1} to server $\tilde{P}_q$ and task τ_{i+2} to server $\tilde{P}_{q+1}$.

For sake of simplicity, in this description we omitted many details, including those related to the application of rules A1 and A2.

5.2.2 New server-to-processor assignment for NPS-F

In the case of NPS-F, to limit the amount of backtracking, we keep the first step of the generic algorithm, i.e. the mapping of tasks to servers, separated and merge the remaining steps in a single one. The mapping of tasks to servers is performed in a first step, as described in Algorithm 1, and is never undone. The backtracking can affect only the assignment of servers to processors, and therefore their inflation and the definition of the reserves.

Algorithm 3 shows the pseudo-code of the new merged step. It assigns servers to processors (employing a NF bin-packing heuristic) and maps processor reserves to servers. The algorithm iterates over the set of servers created by the task mapping algorithm. First, it tries to assign each server as a non-split server. For that purpose, it inflates the current server by invoking the `inflate_sb_non_split` function, which considers the interference of the previous and the next server. If $U[P_p]$ (the utilization of the current processor already assigned to other servers) plus $U^{infl}[\tilde{P}_q]$ (the inflated utilization of the current server) is smaller than or equal to 1.0 (100%), the current server $\tilde{P}_q$ is assigned (non-split) to the current processor P_p and the algorithm moves to the next server. Otherwise, it will try to assign the current server, $\tilde{P}_q$, as a split server. Thus, it computes the inflation of the server by invoking the `inflate_sb_split` function, which considers the interference of the previous two and also the next two servers. If rule A1 applies, then the server is assigned as a non-split server to the next processor, and the algorithm moves to the next server. If rule A2 does not apply, then the current server $\tilde{P}_q$ becomes a split server and is assigned to both the current and the next processor, and the algorithm moves to the next server. Otherwise, i.e. if rule A2 applies, the server is classified as a single server, moved to the end of the server list (and the servers renumbered, for ease of description of the algorithm), so that it is later allocated a dedicated processor. Furthermore, the algorithm is restarted, because servers that have already been assigned to a processor may have to be re-inflated. For example, server $\tilde{P}_{q-1}$, which was inflated assuming that $\tilde{P}_q$ would share the processor with it, will now share the processor with $\tilde{P}_{q+1}$. However, this then entails the possibility that $\tilde{P}_q$ was not sufficiently inflated (since the release interference from tasks on $\tilde{P}_{q+1}$ might be greater than what the schedulability test assumed).

Thus backtracking is performed only when rule A2 is applied. Furthermore, the number of times the algorithm backtracks is bounded by the number of servers. This is because application of rule A2 determines that the server will become a single server, and therefore will no more be subject to application of rule A2.

For the sake of ease of understanding, Algorithm 3 does not include some improvements that could make it more efficient or that could reduce the pessimism in the server inflation for some task sets. For example, when the algorithm applies rule A2 to a server, it moves it to the end of the servers list and restarts the assignment from the beginning. However, there is no need to backtrack all the way back to the beginning: it would be enough to backtrack until the highest numbered processor whose y-reserve mapping is not affected. Therefore, the amount of work that has to be redone can be limited by slightly changing the algorithm. Yet another improvement on the speed of the algorithm is to prevent attempting assignments that will surely fail. For example, if the current processor has already been assigned a non-split server, the current server cannot be assigned as non-split in that processor. Therefore, in this case, the algorithm should try immediately to assign the server as a split server. Yet another example is the case where the sum of the size of the x-reserve, in terms of utilization, and the uninflated utilization of the server under analysis is larger than 1.0. Clearly, that server cannot be assigned to the N-reserve, and therefore the algorithm should try immediately to assign the server as a split server.

Algorithm 3 takes a pessimistic stance and considers that a non-split server always shares the processor with two other servers, and that a split server always shares the processors with four other servers, but this is the worst case. In the best-case scenario, a non-split server may share the processor with only one more server, and a split server with two other servers. Thus, by assuming the best-case, it is possible to eliminate any pessimism from the algorithm (all pessimism is included in the functions that inflate the servers). However, this comes at the cost of additional backtracking, whenever an assumption is proved wrong. Still, it is possible to reduce the pessimism without adding backtracking by taking into account previous assignment decisions. For example, when inflating a non-split server and the x-reserve of the current processor is empty, the algorithm need not consider the interference of the previous server, because they do not share processors.

Algorithm 3 Pseudo-code of the new server-to-processor assignment algorithm.

```
Input: set of $k$ servers
Output: reserves for the $m$ processors, allocating them to the input servers

$restart \leftarrow$ true
while $restart$ do
    $restart \leftarrow$ false
    $p \leftarrow 1$ {processor index}
    $q \leftarrow 1$ {server index}
    $U[P_p] \leftarrow 0$
    while $q \leq k$ and $Type[\tilde{P}_q] \neq$ SINGLE do
        $Type[\tilde{P}_q] \leftarrow$ NON-SPLIT
        $t \leftarrow 2 \cdot \text{lcm\_T}(\tilde{P}_{q-1}, \tilde{P}_q, \tilde{P}_{q+1})$
        $U^{infl}[\tilde{P}_q] \leftarrow \text{inflate\_sb\_non\_split}(\tilde{P}_q, t)$
        if $U[P_p] + U^{infl}[\tilde{P}_q] \leq 1.0$ then
            {server need not be split – we need not check rule A1, at this point}
            $U[P_p] \leftarrow U[P_p] + U^{infl}[\tilde{P}_q]$
            add_server_to_processor_N($P_p, \tilde{P}_q, U^{infl}[\tilde{P}_q]$)
        else
            {Server cannot be NON-SPLIT on $P_p$: try to split it}
            $U^{infl}_{tmp} \leftarrow U^{infl}[\tilde{P}_q]$ {Note that we are considering the interference by $\tilde{P}_{q-1}$, but
            that is not needed}
            $t \leftarrow 2 \cdot \text{lcm\_T}(\tilde{P}_{q-2}, \tilde{P}_{q-1}, \tilde{P}_q, \tilde{P}_{q+1}, \tilde{P}_{q+2})$
            $U^{infl}[\tilde{P}_q] \leftarrow \text{inflate\_sb\_split}(\tilde{P}_q, t)$
            $U^{infl}_y[\tilde{P}_q] \leftarrow 1.0 - U[P_p]$
            $U^{infl}_x[\tilde{P}_q] \leftarrow U^{infl}[\tilde{P}_q] - U^{infl}_y[\tilde{P}_q]$
            if $U^{infl}_x[\tilde{P}_q] \geq U^{infl}_{tmp}$ then
                {Rule A1 – note that inflate_sb_non_split() always considers 3 servers, but in
                this case we need only consider 2}
                adjust_reserves($P_p$) {the y reserve becomes empty}
                $p \leftarrow p + 1$
                $U^{infl}[\tilde{P}_q] \leftarrow U^{infl}_{tmp}$ {non-split server inflated utilization, prev. computed}
                add_server_to_processor_N($P_p, \tilde{P}_q, U^{infl}[\tilde{P}_q]$)
                $U[P_p] \leftarrow U^{infl}[\tilde{P}_q]$
            else
                if $(U^{infl}_x[\tilde{P}_q] + U^{infl}_y[\tilde{P}_q]) \geq 1.0 - ResL/S$ then
                    {Rule A2}
                    $Type[\tilde{P}_q] \leftarrow$ SINGLE
                    move_to_last($\tilde{P}_q$) {so that split servers are assigned “neighbor” processors}
                    $restart \leftarrow$ true
                    break {start all over: inflation of other servers may have been affected}
                else
                    $Type[\tilde{P}_q] \leftarrow$ SPLIT
                    add_server_to_processor_Y($P_p, \tilde{P}_q, U^{infl}_y[\tilde{P}_q]$)
                    $U[P_p] \leftarrow U[P_p] + U^{infl}_y[\tilde{P}_q]$
                    $p \leftarrow p + 1$
                    add_server_to_processor_X($P_p, \tilde{P}_q, U^{infl}_x[\tilde{P}_q]$)
                    $U[P_p] \leftarrow U^{infl}_x[\tilde{P}_q]$
                end if
            end if
        end if
        $q \leftarrow q + 1$
    end while
    if $restart$ then
        continue
    end if
    $p \leftarrow p + 1$
    while $q \leq k$ do
        {handle SINGLE servers, if any}
        add_server_to_processor($P_p, \tilde{P}_q$)
        $U[P_p] \leftarrow 1.0$
        $p \leftarrow p + 1$
        $q \leftarrow q + 1$
    end while
end while
```

5.2.3 Effect of assignment rules on the schedulability analysis

As shown in Algorithm 3, the introduction of assignment rule A2 may lead to backtracking. Although, as we have argued, backtracking is limited, it can nevertheless be undesirable for some task sets, because the increase in execution time may be deemed excessive. In such cases, one can avoid backtracking at the cost of some pessimism, by amending Equations 25 and 37 (employed by the schedulability test), respectively, to:

$$\begin{aligned}&\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{RelI}}(\tilde{P}_q, t) = \\ &\quad \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(prev(\tilde{P}_q), t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q_A(q,t)}, t)\end{aligned} \tag{46}$$

and

$$\begin{aligned}&\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{RelI}}(\tilde{P}_q, t) = \\ &\quad \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(prev(\tilde{P}_q), t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(prev(prev(\tilde{P}_q)), t) \\ &\quad +\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q_A(q,t)}, t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q_B(q,t)}, t)\end{aligned} \tag{47}$$

wherein $prev(\ )$ denotes the previous server (not assigned a dedicated processor) and the server indexes q_A and q_B are computed as:

$$\begin{aligned}&q_A(q,t) \in \{q+1, \ \ldots, \ k\} : \\ &\quad \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q_A(q,t)}, t) \geq \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_p, t) \ \forall p \in \{q+1, \ \ldots, \ k\}\end{aligned} \tag{48}$$

and

$$\begin{aligned}&q_B(q,t) \in \{q+1, \ \ldots, \ k\} \setminus \{q_A\} : \\ &\quad \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q_B(q,t)}, t) \geq \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_p, t) \ \forall p \{q+1, \ \ldots, \ k\} \setminus \{q_A(q,t)\}\end{aligned} \tag{49}$$

That is, when inflating a server, rather than considering the release interference from the next server, we consider the maximum release interference that any of the servers not yet assigned may cause, thus taking a worst-case approach. Similarly for split servers, but in this case we need to consider the two largest values of the release interference that any of the servers not yet assigned may cause.

Note that the values of indexes $q_A(q,t)$ and $q_B(q,t)$ may change with the values of t. However, since both $\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q_A(q,t)}, t)$ and $\mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q_A(q,t)}, t) + \mathrm{dbf}^{\mathrm{part}}_{\mathrm{RelO}}(\tilde{P}_{q_B(q,t)}, t)$ are non-decreasing functions of t, Quick-Processor Demand (QPA) EDF analysis, which is discussed in the next section, is still applicable.

6 Applying the new schedulability theory

In this section, we apply the schedulability theory developed so far to two studies. In both studies we compare the new theory with the utilization-based schedulability theory proposed originally. In the first study, we consider the efficiency of processor utilization. In the second study, we analyze the reliability of the schedules generated. Before presenting these studies, we address two issues related to the application of the new schedulability theory. First, we discuss implementation issues of the schedulability tests. Second, we discuss the experimental evaluation of the different parameters used by the new theory.

6.1 Implementation issues of the schedulability test functions

As discussed earlier, in slot-based task-splitting algorithms, overhead-aware schedulability testing has to be done at two stages: (i) during the task-to-server mapping and (ii) during the server-to-processor assignment. In our code, this testing is, respectively, performed by the C functions (i) `dbf_part_check` (implementing Equation 10) and (ii) `dbf_sb_non_split_check` or `dbf_sb_split _check` (implementing Equations 30 and 44, respectively).

All these functions check whether, at every instant within a time interval $[1, t)$, where t is an argument of the functions, the supply of processor time satisfies the demand. Unlike in conventional uniprocessor EDF scheduling, where certain techniques allow the safe use of much shorter intervals [22,40,34,23], in our case, it is necessary to set t to twice the least common multiple of all T_is of the tasks of the server under consideration (which can be a very big number), and therefore the length t of this testing interval can be exceptionally long. This raises two difficulties. First, the value for t may exceed the range of a 64-bit integer. To overcome this limitation, we used the GNU Multiple Precision Arithmetic C-Library[6]. Second, a longer testing interval means many more iterations, in order to test for all integer values in the range $[1, t)$. To speed up the analysis, we implemented the schedulability testing using Quick Processor-demand Analysis (QPA) [41], which overcomes the need to test for all values in the interval $[1, t)$. This technique works by identifying large sub-intervals within which no deadline misses may occur, and skipping them during testing. This way, for most cases, the analysis is significantly sped up. Algorithm 4 shows, in pseudo-code, how the QPA technique can be used with each of the schedulability tests we defined earlier (where dbf^{xxx} stands for any of them).

6.2 Quantification of overheads

In order to account for the effect of scheduling overheads using the new theory, worst-case estimates for the various overheads themselves are required as input to the analysis. However, upper bounds on the worst-case values of the previously identified overheads cannot be determined via a purely analytical approach, because they depend in complex ways on the characteristics of both the hardware and software, including the operating system, that are rarely

[6] Available online at `http://gmplib.org/`

Algorithm 4 Pseudo-code algorithm of the schedulability test functions.

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

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

documented with sufficient detail. Therefore, our approach was to experimentally measure (and log) the overheads of 100 randomly generated task sets, scheduled during 1000 seconds each, first under S-EKG and then under NPS-F. The corresponding maximum recorded values were then rounded up and the values thus obtained were treated as safe upper bounds for the respective overheads. Although, arguably, there is always the possibility that worse values might be observed if the experiment ran for more time, we deem this level of accuracy sufficient for our purposes in this study. For a more detailed study, or in practice, the number of required measured values will likely vary and depend on such factors as the variability of the measured parameters or the level of safety required. For instance, in [19] a comparative study of global and partitioned algorithms is presented, using empirical data obtained using the LITMUS$^{\mathrm{RT}}$ framework. In that work, some of the overhead costs collected vary a lot from our measurements.

The 24-core platform used in our experiments is built from 1.9 GHz AMD Opteron 6168 chips [24] running at a frequency of 1.9 GHz. Each Opteron 6168 module has 12 cores and occupies one socket on the motherboard. The operating system was the modified 2.6.31 Linux kernel [38].

All parameters were determined in a way consistent to their definition in Section 4. The context switch overhead is measured from the time instant the scheduler starts executing until the time instant when it calls the assembly routine that switches from the current executing job to the new one. To determine the release jitter, we measured the time interval between the (theoretical) job arrival time and the time instant when the timer actually expires, i.e., when the timer callback is invoked. The release overhead was determined by measuring the time interval between the time instant when the timer callback is invoked and a task removed from the release queue is inserted into the ready queue. The reserve latency was estimated by measuring the time interval from the time at which a reserve should (theoretically) start until the time instant when a ready job (if one exists) starts to execute within the reserve. Finally, we measured the IPI latency as the time interval between the generation of the interrupt (by the emitting processor) and the time instant the corresponding handler starts executing (on the other processor).

Table 2 presents the values of these parameters determined experimentally and the estimates derived from those values that were used as input to our experimental evaluation of the overhead-aware analysis. Essentially, we took a pessimistic stance and derived the estimates by rounding up the maximum values measured for each of the parameters.

Table 2: Experimentally-derived estimates for the various scheduling overheads (in μs).

	RelJ	*RelO*	*ResL*	*CtswO*	*IpiL*
Measured	17.45	8.56	30.24	35.21	19.30
Estimates	20.00	10.00	40.00	40.00	20.00

Other than the various overheads identified earlier, we also collected measurements for the *tick interrupt*, which occurs on every processor. This is a periodic interrupt used by the operating system kernel for triggering various operations such as the invocation of the scheduler. The worst-case execution time measured for this interrupt was 8.06 μs. Although its periodicity (approximately 1 ms in our setup) can be configured via the Linux kernel macro `HZ`, in practice this interrupt suffers from jitter. We estimated this jitter, by comparing the recorded inter-arrival times with the reference period, as 177 μs. These values were obtained with a Linux kernel compiled with both the tickless option (for suppressing the tick interrupts during idle intervals) and the CPU frequency scaling features disabled.

We did not derive estimates for overheads from any interrupts other than the tick interrupt because all other interrupts can be configured to be managed by one specific processor (preferably, the least utilized one). Hence, we deemed that, even if we would have gone through that effort, their inclusion would not meaningfully change the overall picture. However, our analysis still allows the overheads related to any interrupt to be specified as input and factored in.

Determining CPMD is a challenging research problem that exceeds the scope of this work. For the state-of-the-art, see [11,1]. Nevertheless, our new schedulability theory allows the incorporation of their effects. In the study with overheads we report below, we performed a sensitivity analysis with respect to the CPMD overhead, by assuming a few values for its maximum value.

Although, in a strict sense, the measurement-based estimates characterize only the system in which the measurements were made, we believe that this particular contribution is important for the following reasons. First, it shows the feasibility of the new analysis, which in turn further validates the slot-based task-splitting approach for multiprocessor scheduling as practical and efficient. Second, by documenting how we derived the measurement-based estimates in a manner consistent with the earlier definitions of the respective overheads, it is possible to re-use the same approach in order to derive estimates for the overheads in different systems.

6.3 Task set generation

In our studies, we consider different types of task sets. We characterize each task set by its normalized utilization and by the characteristics of its tasks. Because, we use a synthetic load, generated randomly using an unbiased random number generator, rather than specifying a single value for the task set normalized utilization, we use an interval with minimum value $U_{s:min}$ and width inc, $[U_{s:min}, U_{s:min} + inc)$. With respect to the characteristics of the tasks, the period of each task T_i, is uniformly distributed over $[T_{i:min}, T_{i:max})$. All tasks generated are implicit-deadline ($D_i = T_i$) in order to allow comparisons with the original analysis. The worst-case execution time of a task, C_i, is derived from T_i and the task's utilization, u_i, which is also uniformly distributed over $[u_{i:min}, u_{i:max})$.

Algorithm 5 shows the task generation procedure. It takes as inputs the minimum normalized system utilization, $U_{s:min}$, the granularity of the normalized system utilization of each task set, inc, the number of task sets, n^{τ}, the minimum and maximum values of the utilization of each task in all task sets, $u_{i:min}$ and $u_{i:max}$, respectively, the minimum and maximum values of the period of each task in all task sets, $T_{i:min}$ and $T_{i:max}$, respectively, and the number of processors in the system, m. The output of this procedure are n^{τ} task sets which are put in array Γ. The normalized system utilization of task set $\Gamma[i]$ (for i between 1 and n^{τ}) is in the range $[U_{s:min} + (i-1) \cdot inc$, $U_{s:min} + i \cdot inc)$, and the parameters of each task in these sets satisfy the values specified in the inputs of the procedure.

In all experiments, we used $U_{s:min}$ equatl to 0.75, inc equal to 0.001 and n^{τ} equal to 250, allowing us to evaluate the algorithms for a fairly loaded system, i.e. whose load has normalized utilization between 75% and 100%. Indeed, for systems with a lighter load, we would expect no major differences, as all task sets would most likely be schedulable. To evaluate the effect of different types of tasks, we consider four classes of task sets according to the utilization of their tasks:

- *Heavy*: Tasks whose u_i is in the range [0.65, 0.95).
- *Medium*: Tasks whose u_i is in the range [0.35, 0.65).
- *Light*: Tasks whose u_i is in the range [0.05, 0.35).
- *Mixed*: Tasks whose u_i is in the range [0.05, 0.95).

Independently of their utilization, the periods of all tasks of all task sets are uniformly distributed in the range [5 ms, 50 ms], with a resolution of 1 ms.

Finally, in all experiments we set m to 24, the number of processors in the system we used to measure the overheads.

6.4 Evaluation of the new analysis in the absence of overheads

As a first step in the evaluation of the new analysis, we compare it to the original analysis published for both algorithms, so as to evaluate the improvements in processor utilization that stem from less pessimism in the new analysis.

Algorithm 5 Pseudo-code algorithm of the task set generator.

```
Input: $U_{s:min}$, minimum normalized system utilization
       $inc$, granularity of the normalized system utilization of the task sets
       $n^{\tau}$ number of task sets to generate
       $[u_{i:min}, u_{i:max}]$ range of the utilization of each task to generate
       $[T_{i:min}, T_{i:max}]$ range of the period of each task to generate
       $m$ number of processors/cores
Output: $\Gamma[n^{\tau}]$ set of generated task sets

$U_{s:max} \leftarrow U_{s:min} + inc$
for $j \leftarrow 1$ to $n^{\tau}$ do
  $\Gamma[j] \leftarrow \emptyset$
  $i \leftarrow 0$
  $\tau \leftarrow \emptyset$
  $U_s \leftarrow 0$
  $in\_range \leftarrow$ false
  while $U_s < U_{s:max}$ do
    {generate a task}
    $u_i \leftarrow$ uniform$(u_{i:min}, u_{i:max})$
    $T_i \leftarrow$ uniform$(T_{i:min}, T_{i:max})$
    $C_i \leftarrow T_i \cdot u_i$
    $\tau_i \leftarrow$ create_task$(T_i, C_i)$
    {add task to task set}
    add_task_to_taskset$(\tau_i, \tau)$
    $U_s \leftarrow U_s + u_i/m$
    $i \leftarrow i + 1$
    if $U_{s:min} \leq U_s < U_{s:max}$ then
      {Done with this task set: it has the target utilization}
      add_taskset_to_set$(\tau, \Gamma[j])$
      $in\_range \leftarrow$ true
      break
    end if
  end while
  if $\neg in\_range$ then
    {Abort current task set generation: utilization is above target}
    $j \leftarrow j - 1$ {Try again}
  else
    {Generated task set successfully. Update target range utilization for next task set}
    $U_{s:min} \leftarrow U_{s:max}$
    $U_{s:max} \leftarrow U_{s:min} + inc$
  end if
end for
```

Given the goals of this study, we have chosen as metric the normalized inflated system utilization, which is defined as follows:

$$U_s^{infl} = \frac{1}{m} \cdot \sum_{q=1}^{k} U^{infl}[\tilde{P}_q] \tag{50}$$

where m is the number of processors in the system, k is the number of servers, and $U^{infl}[\tilde{P}_q]$ is the inflated utilization of server $\tilde{P}_q$. A schedulability analysis is more efficient than another, if its normalized system inflated utilization is lower.

Because the original, utilization-based, analysis assumes no scheduling overheads, the results presented in this subsection were obtained considering all overheads equal to 0.

6.4.1 Experiments for the S-EKG scheduling algorithm

In order to apply the new analysis to S-EKG, we employed the task-to-processor mapping algorithm outlined in Section 5.2.1. A major difference between this algorithm and the original S-EKG algorithm is that it does not cap the utilization of each processor to the theoretical utilization bound ($\mathrm{UB}_{\mathrm{S-EKG}}$), but rather uses the new schedulability tests presented in Section 4.

In our study, we considered the effect of the S-EKG design parameter δ, in addition to the workload itself, because in the original analysis this parameter has a major influence on the system utilization. Thus, for each workload, i.e. task set, we computed the normalized utilization for each of the following δ values: 1, 2, 4, and 8.

Figure 15 provides a comparison between the original (utilization-based) and the new (processor demand-based) theory, for task sets generated under the "mixed" setup ($u_{i:min} = 0.05$, $u_{i:max} = 0.95$) for different values of δ and with the time slot length S selected in each case according to Equation 4. Each point in the plots shown in this section represents an average of the normalized utilization for 100 randomly generated task sets, satisfying the corresponding parameter values.

As shown in Inset (a), many task sets of relatively low utilization are not schedulable according to the original analysis even with higher values for δ. The results are completely different when we apply the new schedulability test (see Inset (b)), with almost all task sets being schedulable even with δ equal to one (the most preemption-light setting). Furthermore, the effect of δ on the schedulability of the task sets is much lower than in the original analysis. Indeed, the original S-EKG schedulability test fails for all task sets with δ equal to one. The explanation is that the original S-EKG task-to-processor assignment algorithm caps the utilization of each processor to the theoretical utilization bound ($\mathrm{UB}_{\mathrm{S-EKG}}$), and for δ equal to one, $\mathrm{UB}_{\mathrm{S-EKG}} \approx 0.65$, which is less than the lowest U_s (0.75) of any task set used in the experiments. In fact, the only task sets with $U_s > \mathrm{UB}_{\mathrm{S-EKG}}$ deemed schedulable by the original schedulability test are some task sets with one or more tasks with $u_i > \mathrm{UB}_{\mathrm{S-EKG}}$ (which then get assigned to dedicated processors).

Figure 16 further highlights the benefits of the new schedulability analysis. It compares the results of the new analysis with those of the original analysis for task sets generated according to the "heavy", "medium" and "light" parameter setup. The same conclusions as before apply. The new analysis clearly improves the inflation efficiency, in all cases. The improvement is so large that the inflated utilization is, at all points in the graph, very close to the uninflated utilization even for δ equal to one.

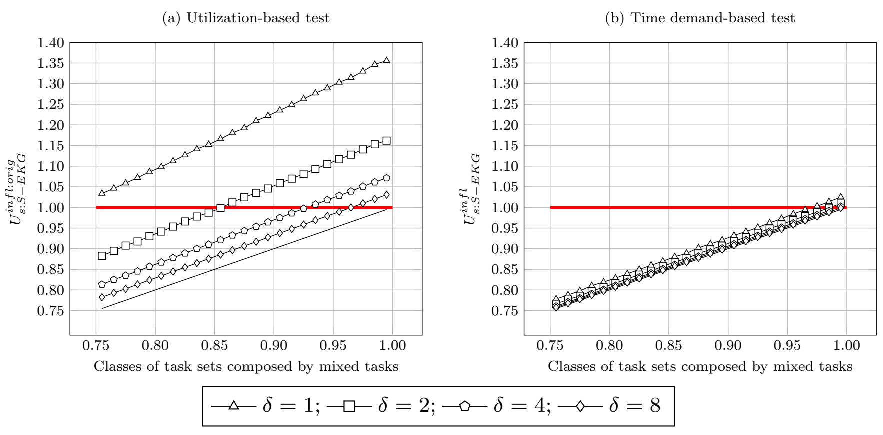

Fig. 15: Comparison between the original S-EKG and the new schedulability analysis for task sets composed by mixed ($u_i \in [0.05, 0.95)$) tasks.

6.4.2 Experiments for the NPS-F scheduling algorithm

We performed the same set of experiments using NPS-F rather than S-EKG. Figure 17 compares the original (Inset(a)) and the new (Inset(b)) schedulability analysis for task sets generated under the "mixed" setup ($u_{i:min} = 0.05$, $u_{i:max} = 0.95$), which demonstrates a considerable improvement in mapping efficiency. In fact, using the new analysis, the points for the inflated and uninflated utilization almost coincide in the graph (even for δ equal to one). These observations also apply to the experiments with the "light", "medium" and "heavy" task utilization setup, shown in Figure 18.

6.5 Reliability of the task assignment

The lower efficiency of the utilization-based analysis provides a safety margin to compensate for overheads that occur in real systems, which are not accounted for in the analysis. However, there is no guarantee that this over-provisioning is sufficient. It may well be the case that the utilization-based test considers a task set as schedulable, when it really is not because of the overheads incurred in real systems.

To better evaluate this possibility, we carried out a study in which we assessed whether the task sets deemed schedulable using the utilization-based analysis were unschedulable according to the new demand-based and overhead-aware schedulability analysis. Therefore the metric we used in this study was:

$$\frac{N^{util}_{sched} - N^{oa\_db|util}_{sched}}{N^{util}_{sched}} \tag{51}$$

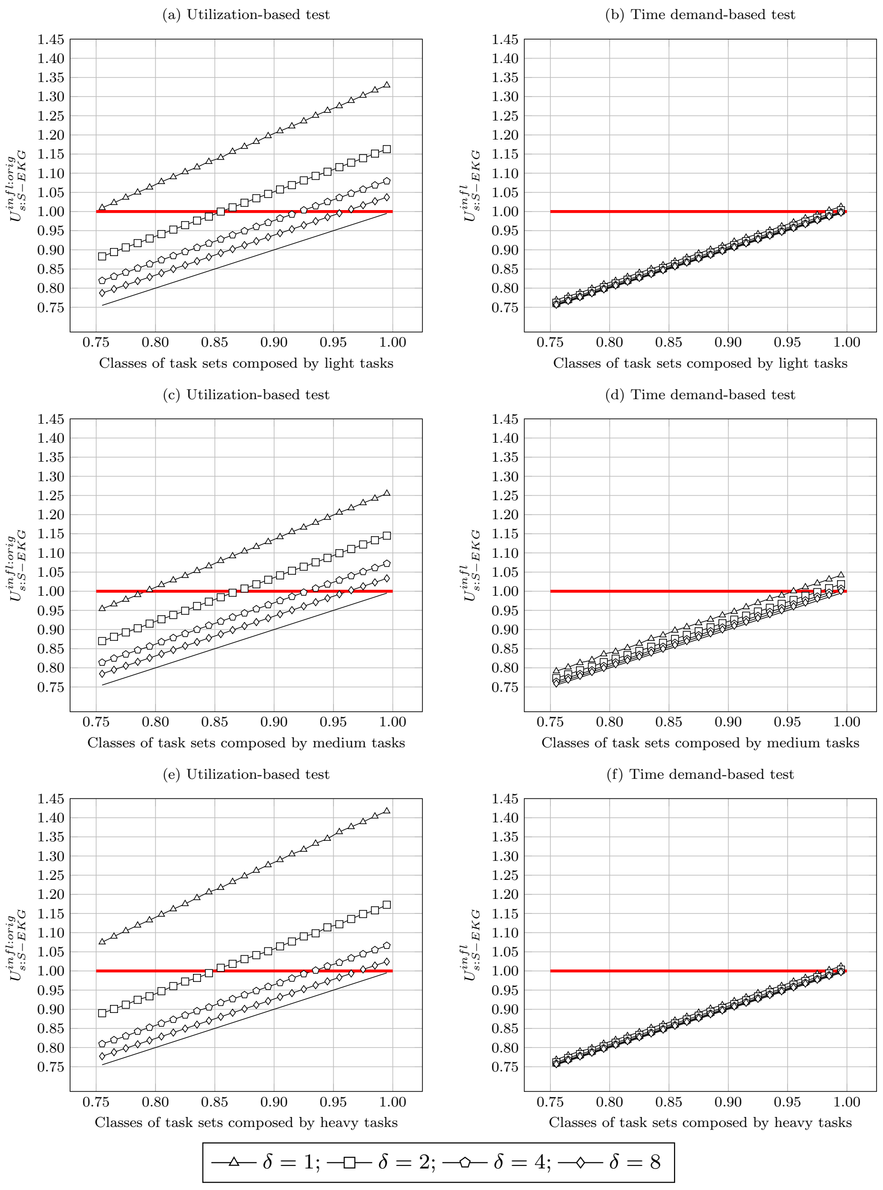

Fig. 16: Comparison between the original S-EKG and the new schedulability analysis considering task sets composed by light ($u_i \in [0.05,0.35)$), medium ($u_i \in [0.35,0.65)$), and heavy tasks ($u_i \in [0.65,0.95)$).

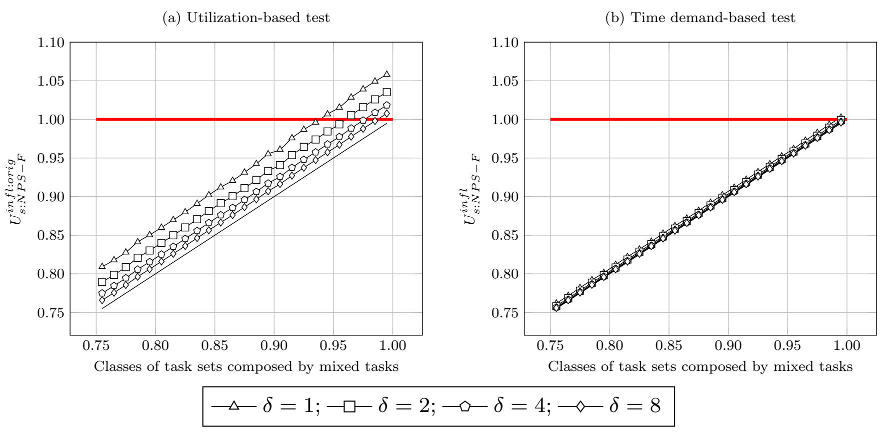

Fig. 17: Comparison between the original NPS-F and the new schedulability analysis considering task sets composed by mixed ($u_i \in [0.05, 0.95)$) tasks.

where N^{util}_{sched} is the number of task sets deemed schedulable according to the utilization-based schedulability analysis and $N^{oa\_db|util}_{sched}$ is the number of these task sets that are also schedulable according to the overhead-aware demand-based analysis. Because of space limitations, we consider only NPS-F.

In all experiments of this study, we kept all overheads constant using the values presented in Table 2, that are based on measurements on implementations of the NPS-F and the S-EKG algorithms. For the *CpmdO*, given the dependence of this parameter on the load, we chose to perform a sensitivity analysis and used three values for this parameter 0, 100 and 500 μs. The zero value represents a best case for the utilization-based analysis; the lower the overheads the more likely the inflated utilization to be enough to make up for them. The 500 μs value corresponds to a rather high value for the CPMD, taking into account that the minimum task period, and consequently the slot duration, in any task set is not much higher than five milliseconds. For some light tasks, 500 μs may be larger than the task computation time, itself, therefore we used a third value of 100 μs, which should not penalize as much lighter tasks. As in the previous study, for each task set generated according to Algorithm 5, we used all the values of the design parameter δ considered. Furthermore, we ignored all the interrupts except the local timer interrupts. (This is tantamount to assume that interrupt handling is performed by a dedicated processor.)

Figure 19 summarizes the results of this study. Each inset shows the results for a different value of the *CpmdO* parameter. The value of this parameter has a major effect, although it may not be that apparent at first sight, because the ranges in the y axis are different. As expected, the higher the value of the *CpmdO* the higher the fraction of tasks deemed schedulable according to the utilization-based analysis, but not schedulable by the new overhead-aware and

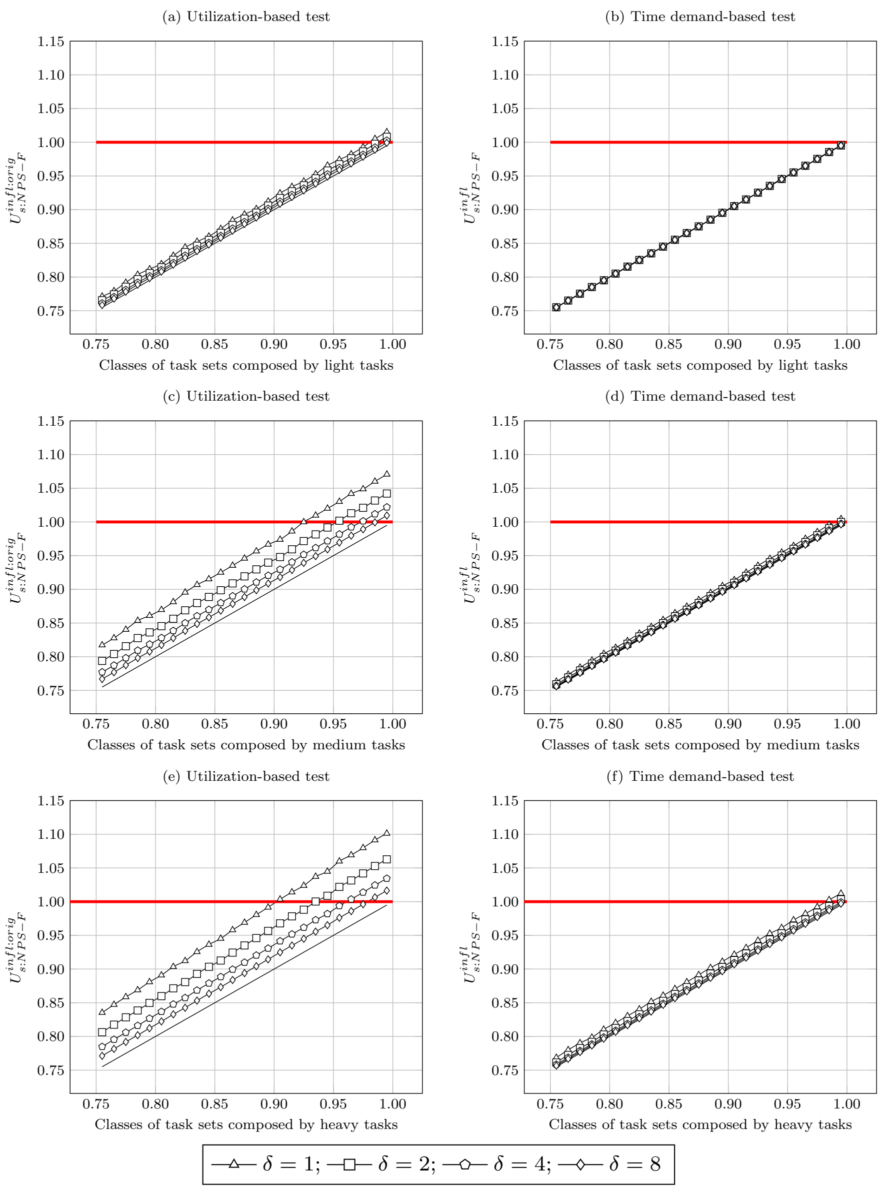

Fig. 18: Comparison between the original NPS-F and the new schedulability analysis considering task sets composed by light ($u_i \in$ [0.05,0.35)), medium ($u_i \in$ [0.35,0.65)), and heavy tasks ($u_i \in$ [0.65,0.95)).

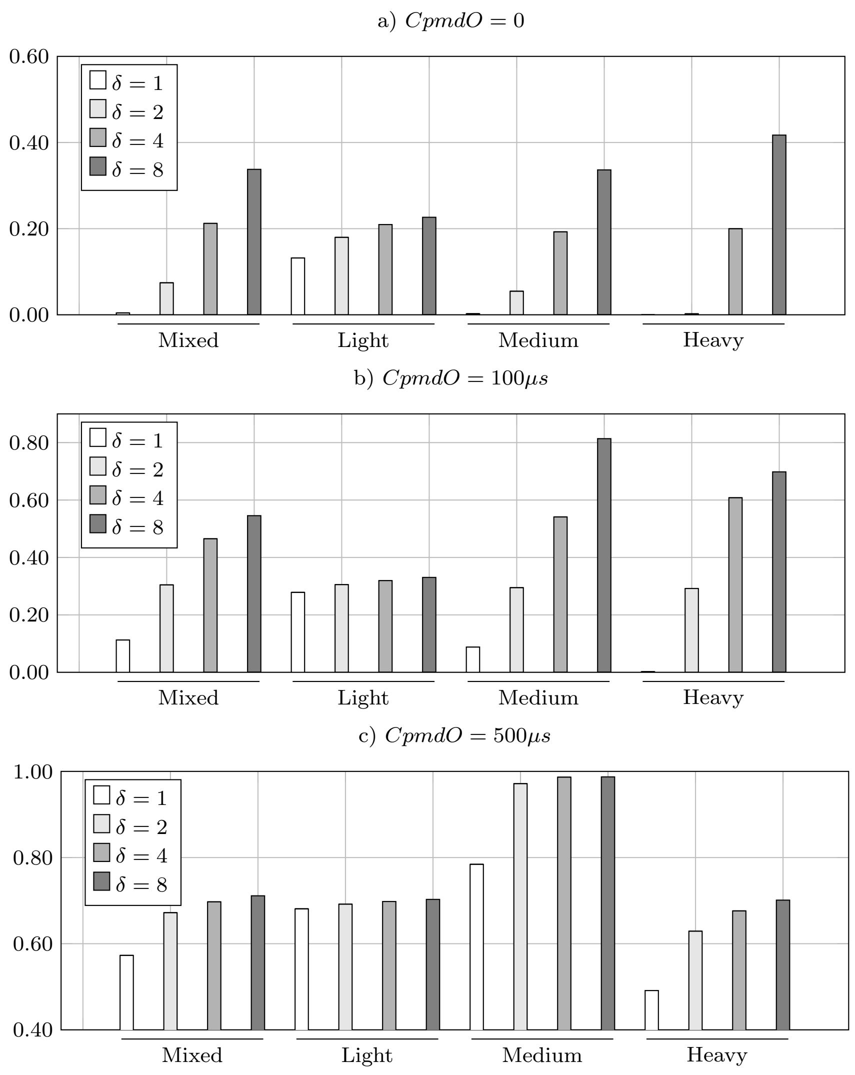

Fig. 19: Fraction of task sets schedulable according to the utilization-based analysis, but not schedulable according to the new overhead-aware and demand-based analysis, as a function of the type of tasks of the task set and of the design parameter δ.

demand-based analysis, for the parameter values considered. Also clear is the effect of the design parameter δ. The higher the value of this parameter, the higher the fraction of tasks that are not schedulable.

Figure 19 shows the fraction of non schedulable task sets ignoring the utilization of the task set, to make the dependence on the factors considered more clear. Inset (a) of Figure 20 shows the dependence on the utilization of the task sets, for the mixed task sets with *CpmdO* equal to zero. As shown,

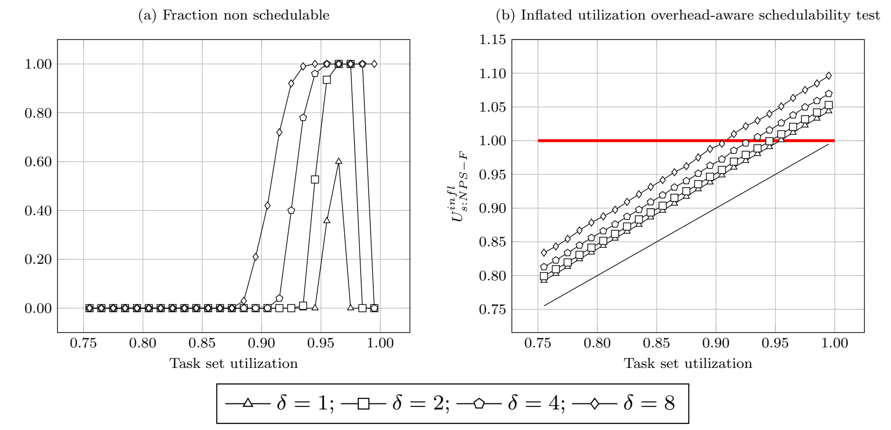

Fig. 20: Fraction of mixed task task sets that are considered schedulable by the original schedulability analysis but are not by the schedulability with overheads ($CpmdO$ equal to zero)

for utilizations lower than a certain value, which depends on the value of δ, all task sets are schedulable according to both analyses. However, at a given point the fraction of non-schedulable task sets rises sharply to 1, and remains around 1 until a point, which also depend on δ, when it then drops more or less sharply to 0. As shown, the value of δ determines the width of the plateau where the fraction is equal to 1: the higher the value of δ the earlier the fraction rises to 1, and the later it drops back to 0. For this parameter settings, for δ equal to one, the fraction of unschedulable task sets never reaches 1, rather increases up to around 0.60 and then drops back to zero. In any case, the pattern is clear and applies also to other types of task sets and different values of $CpmdO$, and can be easily explained with the help of Inset (a) of Figure 17 and Inset (b) of Figure 20, which show the average inflated utilization respectively for mixed tasks task sets for the utilization-based analysis and for the overhead-aware and demand-based analysis with $CpmdO$ equal to zero. Consider a given value of δ, say 4. For task sets whose utilization is below 0.91, the overheads are small enough that virtually all task sets are considered schedulable by both analyses. As the task set utilization increases from 0.91 to 0.95, the average inflated utilization according to the new analysis increases and becomes higher than 1, see Inset (b) of Figure 20, so that virtually all task sets are deemed unschedulable. On the other hand in that range, for δ equal to four, the inflated utilization according to the utilization-based analysis, see Inset (a) of Figure 17, is still below about 0.97, and many task sets are still deemed schedulable. Therefore, in that interval the fraction of non-schedulable task sets raises from 0 to 1, and remains 1 until it drops sharply for task sets whose utilization is in the range $[0.99, 1.0)$, which are all deemed unschedulable also by the utilization-based analysis.

Inset (b) of Figure 20 also shows that, contrary to what is predicted by the utilization based analysis, when we take into account overheads, increasing the value of δ decreases the schedulability rather than increasing it. This pattern also holds for higher values of $CpmdO$, as we would expect, and for other types of task sets, and confirms an observation already made in [13].

Even though the new overhead-aware demand-based analysis is conservative, i.e. is based on worst-case assumptions, and therefore it may be that a task set it considers non-schedulable is actually schedulable, the parameter values we used are all values we measured in a real system, except the values for the CPMD overheads. For the latter we assumed several values including zero, and even in this case, which is rather optimistic, the utilization-based analysis may consider a given task set schedulable, when some tasks may miss their deadline. This is unacceptable in safety-critical hard-real time systems, where the consequences of missing a deadline may be catastrophic. The overhead-aware and demand-based analysis we developed allows to account for all overheads incurred by an implementation, does so in a conservative way, and therefore ensures that its results are reliable, as long as the values of its parameters are valid.

7 Conclusions

In this article, as a main contribution, we formulated a new demand-based and overhead-aware schedulability analysis for slot-based task-splitting algorithms. This new scheduling analysis, which guides the task assignment and splitting process, produces a better schedule than the previous analyses in terms of both efficiency and reliability. The new theory, applicable to both S-EKG and NPS-F (the two slot-based semi-partitioned algorithms with high utilization bounds), allow both algorithms to target arbitrary-deadline tasks and, importantly, takes the real-world overheads incurred by these kinds of scheduling algorithms into account. Interestingly, the experimental results (obtained for estimates of the various overheads derived via testing on a real system) show that, in practice, the configurations (in terms of time slot length) that afford the best schedulability are not the ones deemed as such by the analysis ignoring overheads available so far.

In the near future, we plan to use this new theory to carry out an exhaustive evaluation of slot-based task-splitting algorithms, of the most appropriate values for their parameters, e.g. the design parameter δ or the time slot size, S, best suited for different classes of tasks, as well as of heuristics for assigning tasks to processors. Yet another research direction that we plan to pursue is to evolve the new theory to reduce the pessimism with respect to CPMD costs. This is motivated by our experimental measurements that suggest that the CPMD overheads used in other works may be more than one order of magnitude larger than the other overheads.

References

1. Altmeyer, S., Davis, R.I., Maiza, C.: Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. Real-Time Systems **48**(5), 499–526 (2012)
2. Anderson, J., Bud, V., Devi, U.: An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In: proc. of the 17th IEEE Euromicro Conference on Real-Time Systems (ECRTS'05), pp. 199–208. Palma de Mallorca, Balearic Islands, Spain (2005)
3. Anderson, J., Srinivasan, A.: Mixed pfair/erfair scheduling of asynchronous periodic tasks. Journal Computer and System Sciences **68**(1), 157–204 (2004)
4. Andersson, B., Bletsas, K.: Sporadic multiprocessor scheduling with few preemptions. In: proc. of the 20th IEEE Euromicro Conference on Real-Time Systems (ECRTS'08), pp. 243–252. Prague, Czech Republic (2008)
5. Andersson, B., Bletsas, K., Baruah, S.: Scheduling arbitrary-deadline sporadic tasks on multiprocessors. In: proc. of the 29th IEEE Real-Time Systems Symposium (RTSS'08), pp. 385–394. Barcelona, Spain (2008)
6. Andersson, B., Tovar, E.: Multiprocessor scheduling with few preemption. In: proc. of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Application (RTCSA'06), pp. 322–334. Sydney, Australia (2006)
7. Audsley, N., Burns, A., Richardson, M., Tindell, K., Wellings, A.J.: Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal **8**(5), 284–292 (1993)
8. Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In: proc. of the 10th ACM International Symposium on Hardware/Software Codesign (CODES'02), pp. 73–78. Estes Park, Colorado (2002)
9. Baruah, S., Cohen, N., Plaxton, G., Varvel, D.: Proportionate progress: A notion of fairness in resource allocation. Algorithmica **15**, 600–625 (1994)
10. Baruah, S., Mok, A., Rosier, L.: Preemptively scheduling hard-real-time sporadic tasks on one processor. In: proc. of the 11st IEEE Real-Time Systems Symposium (RTSS'90), pp. 182–190. Lake Buena Vista, Florida, USA (1990)
11. Bastoni, A.: Towards the integration of theory and practice in multiprocessor real-time scheduling. Ph.D. thesis, University of Rome "Tor Vergata" (2011)
12. Bastoni, A., Brandenburg, B., Anderson, J.: Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In: proc. of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'10), pp. 33–44. Brussels, Belgium (2010)
13. Bastoni, A., Brandenburg, B., Anderson, J.: Is semi-partitioned scheduling practical? In: proc. of the 23rd IEEE Euromicro Conference on Real-Time Systems (ECRTS'11), pp. 125–135. Porto, Portugal (2011)
14. Bastoni, A., Brandenburg, B.B., Anderson, J.H.: An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In: proc. of the 31st IEEE Real-Time Systems Symposium (RTSS'10), pp. 14–24. IEEE Computer Society, San Diego, CA, USA (2010)
15. Bletsas, K., Andersson, B.: Notional processors: an approach for multiprocessor scheduling. In: Proc. of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09), pp. 3–12. San Francisco, CA, USA (2009)
16. Bletsas, K., Andersson, B.: Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In: Proc. of the 30th IEEE Real-Time Systems Symposium (RTSS'09), pp. 385–394. Washington, DC, USA (2009)
17. Bletsas, K., Andersson, B.: Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. Real-Time Systems **47**(4), 319–355 (2011)
18. Burns, A., Davis, R.I., Wang, P., Zhang, F.: Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme. Real-Time Syst. **48**(1), 3–33 (2012)
19. Calandrino, J., Leontyev, H., Block, A., Devi, U., Anderson, J.: Litmus^{rt} : A testbed for empirically comparing real-time multiprocessor schedulers. In: proc. of the 27th IEEE Real-Time Systems Symposium (RTSS'06), pp. 111–126. Rio de Janeiro, Brazil (2006)

20. Coffman, E., Garey, M., Johnson, D.: Approximation algorithms for NP-hard problems. chap. Approximation algorithms for bin packing: a survey, pp. 46–93. PWS Publishing Co., Boston, MA, USA (1997)
21. Davis, R., Burns, A.: A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. Technical report YCS-2009-443, University of York, Department of Computer Science (2009)
22. George, L., Rivierre, N., Spuri, M.: Preemptive and nonpreemptive real-time uniprocessor scheduling. Tech. Rep. 2966, INRIA, France (1996)
23. Hoang, H., Buttazzo, G., Jonsson, M., Karlsson, S.: Computing the minimum EDF feasible deadline in periodic systems. In: proc. of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06), pp. 125–134 (2006)
24. Inc., A.: AMD Opteron Processor. http://products. amd.com/en-us/OpteronCPUDetail.aspx?id=645
25. Ju, L., Chakraborty, S., Roychoudhury, A.: Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In: proc. of the IEEE Design, Automation Test in Europe Conference Exhibition (DATE'07), pp. 1–6. Nice, France (2007)
26. Kato, S., Yamasaki, N.: Real-time scheduling with task splitting on multiprocessors. In: proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07), pp. 441–450. Daegu, Korea (2007)
27. Kato, S., Yamasaki, N.: Portioned EDF-based scheduling on multiprocessors. In: proc. of the 8th ACM/IEEE International Conference on Embedded Software (EMSOFT'08), pp. 139–148. Atlanta, GA, USA (2008)
28. Kato, S., Yamasaki, N.: Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: proc. of the 21st Euromicro Conference on Real-Time Systems (ECRTS'09), pp. 239–248. Dublin, Ireland (2009)
29. Lakshmanan, K., Rajkumar, R., Lehoczky, J.: Partitioned fixed-priority preemptive scheduling for multi-core processors. In: proc. of the 21st Euromicro Conference on Real-Time Systems (ECRTS 09), pp. 239–248. Dublin, Ireland (2009)
30. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM **20**(1), 46–61 (1973)
31. Lunniss, W., Altmeyer, S., Maiza, C., Davis, R.: Integrating cache related pre-emption delay analysis into EDF scheduling. In: proc. of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13), pp. 75–84. Philadelphia, PA, USA (2013)
32. PREEMPT-RT: Real-time Linux wiki (2012). URL `https://rt.wiki.kernel.org/`
33. Puaut, I., Pais, C.: Scratchpad memories vs locked caches in hard real-time systems: A quantitative comparison. In: proc. of the Conference on Design, Automation and Test in Europe (DATE'07), pp. 1484–1489. Nice, France (2007)
34. Ripoll, I., Crespo, A., Mok, A.: Improvement in feasibility testing for real-time tasks. Real-Time Systems **11**(1), 19–39 (1996)
35. Sousa, P.B., Andersson, B., Tovar, E.: Implementing slot-based task-splitting multiprocessor scheduling. Technical report HURRAY-TR-100504, CISTER, Polytechnic Institute of Porto (ISEP-IPP) (2010)
36. Sousa, P.B., Andersson, B., Tovar, E.: Implementing slot-based task-splitting multiprocessor scheduling. In: proc. of 6th IEEE International Symposium on Industrial Embedded Systems (SIES'11), pp. 256–265. Vasteras, Sweden (2011)
37. Sousa, P.B., Bletsas, K., Andersson, B., Tovar, E.: Practical aspects of slot-based task-splitting dispatching in its schedulability analysis. In: proc. of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11), pp. 224–230. Toyama, Japan (2011)
38. Sousa, P.B., Bletsas, K., Tovar, E., Andersson, B.: On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems. In: proc. of the 13th Real-Time Linux Workshop (RTLWS'13), pp. 207–218. Real-Time Linux Foundation, Prague, Czech Republic (2011)
39. Sousa, P.B., Pereira, N., Tovar, E.: Enhancing the real-time capabilities of the Linux kernel. In: 24th Euromicro Conference on Real-Time Systems (ECRTS'12) – Work-in-Progress Session. Pisa, Italy (2012)

40. Spuri, M.: Analysis of deadline scheduled real-time systems. Tech. rep., INRIA (1996)
41. Zhang, F., Burns, A.: Improvement to Quick Processor-Demand Analysis for EDF-Scheduled Real-Time Systems. In: proc. of the 21st IEEE Euromicro Conference on Real-Time Systems (ECRTS' 09), pp. 76–86 (2009)

Appendix

A Interrupts

Interrupts in an operating system are raised by any hardware or software component when it wants the processor's attention. Basically, when a processor receives an interrupt, it stops the execution of the current task to execute the interrupt service routine (ISR) associated with the received interrupt. We model each sporadic interrupt Int_i as a sporadic interfering task with minimum inter-arrival time of T_i^{Int} and an execution time equal to C_i^{Int}, which runs at a higher priority than normal tasks. Some periodic interrupts (for example, the periodic tick) are also characterised by an arrival jitter J_i^{Int}. We assume that C_i^{Int} is much smaller than S ($C_i^{Int} \ll S$) and the number of distinct types of interrupts is limited to n^{Int}. Modelling interrupts in this manner allows safely upper-bounding the cumulative execution demand by interrupts using conventional analysis for sporadic tasks (which we next formulate in detail). However, specifically for interrupts with $T_i^{Int} < S$, modelling such interrupts as bursty periodic tasks[7] is sometimes less pessimistic. Intuitively, this is because under slot-based task-splitting scheduling algorithms, interrupts only exert overhead when present inside the reserve(s) of the server under consideration. Outside its reserve(s), an interrupt contributes to the processor-demand of some other server instead. Therefore, for each interrupt with $T_i^{Int} < S$, we consider both models and pick the least pessimistic value.

Next, we present in detail how the processor demand of interrupts is bounded under our analysis. Note that depending on the server type (split or non-split), we model the execution demand of interrupts in a slightly different way. First, let us consider non-split servers:

A non-split server executes in a single reserve of length $Res^{len}[\tilde{P}_q]$ (see Equation 26). The cumulative execution demand by all interrupts on the server can be upper-bounded as

$$\mathrm{dbf}_{\mathrm{IntO}}^{\mathrm{sb:non-split}}(\tilde{P}_q, t) = \sum_{i=1}^{n^{Int}} \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:non-split}}(Int_i, \tilde{P}_q, t) \tag{52}$$

where $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:non-split}}(Int_i, \tilde{P}_q, t)$ is the respective upper bound on the processor demand by interrupt Int_i.

For every interrupt Int_i (irrespective of whether $T_i^{Int} < S$ or $T_i^{Int} \geq S$), an upper bound for $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:non-split}}(Int_i, \tilde{P}_q, t)$ can be (pessimistically) computed as:

$$\mathrm{dbf}_{\mathrm{Int}}^{(\mathrm{continuous})}(Int_i, t) = \left\lceil \frac{t + J_i^{int}}{T_i^{Int}} \right\rceil \cdot C_i^{Int} \tag{53}$$

The pessimism in this derivation lies in that even interrupts raised outside the reserves of the server in consideration are treated as interfering. However, for interrupts with $T_i^{Int} < S$, an alternative derivation of an upper bound for $\mathrm{dbf}_{\mathrm{Int}}^{\mathrm{sb:non-split}}(Int_i, \tilde{P}_q, t)$ is possible, using the bursty periodic model, as explained earlier.Namely:

$$\begin{aligned} &\mathrm{dbf}_{\mathrm{Int}}^{(\mathrm{non-split:bursty})}(Int_i, \tilde{P}_q, t) = \\ &\quad \mathrm{nr}^{\mathrm{S}}(t) \cdot \mathrm{dbf}_{\mathrm{Int}}^{\mathrm{N}}(Int_i, \tilde{P}_q) + \mathrm{dbf}_{\mathrm{IntO}}^{\mathrm{tail:N}}(Int_i, \tilde{P}_q) \ \forall i, T_i^{Int} < S \end{aligned} \tag{54}$$

[7] The bursty periodic arrival model was introduced in [7]

where $\mathrm{nr}^{\mathrm{S}}(t)$ is an upper bound on the number of time slots fully contained in the time interval under consideration (of length t) and $\mathrm{dbf}^{\mathrm{N}}_{\mathrm{Int}}(Int_i, \tilde{P}_q)$ is an upper bound on the demand by Int_i inside the reserve (of length $Res^{len}[\tilde{P}_q]$) of server $\tilde{P}_q$ in a single time slot (of length S). Similarly for $\mathrm{dbf}^{\mathrm{tail:N}}_{\mathrm{Int}}(Int_i, \tilde{P}_q)$, but over the remaining time interval (i.e. the "tail") of length t^{tail}. These terms, in turn, are derived as:

$$\mathrm{nr}^{\mathrm{S}}(t) = \left\lfloor \frac{t + ResL}{S} \right\rfloor \tag{55}$$

$$t^{tail} = t - \mathrm{nr}^{\mathrm{S}}(t) \cdot S \tag{56}$$

$$\mathrm{dbf}^{\mathrm{N}}_{\mathrm{Int}}(Int_i, \tilde{P}_q) = \mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, Res^{len}[\tilde{P}_q]) \tag{57}$$

and

$$\mathrm{dbf}^{\mathrm{tail:N}}_{\mathrm{Int}}(Int_i, \tilde{P}_q) = \mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, \min(t^{tail}, Res^{len}[\tilde{P}_q])) \tag{58}$$

Often, though not always, $\mathrm{dbf}^{\mathrm{(non-split:bursty)}}_{\mathrm{Int}}(Int_i, \tilde{P}_q, t)$ provides a less pessimistic estimate than $\mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, t)$, for interrupts with $T^{Int}_i < S$. Hence in the general case $\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{Int}}(Int_i, t)$ is computed as:

$$\mathrm{dbf}^{\mathrm{sb:non-split}}_{\mathrm{Int}}(Int_i, \tilde{P}_q, t) = \begin{cases} \mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, t) & \text{if } T^{Int}_i \geq S \\ \min\left(\mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{IntO}}(Int_i, t),\ \mathrm{dbf}^{\mathrm{(non-split:bursty)}}_{\mathrm{Int}}(Int_i, \tilde{P}_q, t)\right) & \text{if } T^{Int}_i < S \end{cases} \tag{59}$$

Next, we deal with split servers. For convenience we define:

$$\begin{aligned} Res^{len}_x[\tilde{P}_q] &= x[P_{p+1}] = U^{infl}_x[\tilde{P}_q] \cdot S \\ Res^{len}_y[\tilde{P}_q] &= y[P_p] = U^{infl}_y[\tilde{P}_q] \cdot S \end{aligned} \tag{60}$$

As mentioned before, a split server $\tilde{P}_q$ executes on two reserves (of length $Res^{len}_x[\tilde{P}_q]$ and $Res^{len}_y[\tilde{P}_q]$) separated by Ω time units. That is, it is idle during Ω time units, next it executes during $x[P_{p+1}]$ on processor P_{p+1}, then it is idle again during Ω time units, and finally it executes during $y[P_p]$ on processor P_p (see Figure 11).

The cumulative execution demand by all interrupts on the server reserves can be upper-bounded as

$$\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{IntO}}(\tilde{P}_q, t) = \sum_{i=1}^{n^{Int}} \mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Int}}(Int_i, \tilde{P}_q, t) \tag{61}$$

where $\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Int}}(Int_i, \tilde{P}_q, t)$ is the respective upper bound on the processor demand by interrupt Int_i.

For every interrupt Int_i (irrespective of whether $T^{Int}_i < S$ or $T^{Int}_i \geq S$), an upper bound for $\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Int}}(Int_i, \tilde{P}_q, t)$ can be (pessimistically) computed by $\mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, t)$. As for non-split servers, the pessimism in this derivation lies in that even interrupts raised outside the reserves of the server in consideration are treated as interfering. Then, for interrupts with $T^{Int}_i < S$ it is possible to employ the bursty periodic model that may reduce the pessimism:

$$\begin{aligned}&\mathrm{dbf}^{\mathrm{(split:bursty)}}_{\mathrm{Int}}(Int_i, \tilde{P}_q, t) = \\ &\quad \mathrm{nr}^{\mathrm{S}}(t) \cdot (\mathrm{dbf}^{\mathrm{X}}_{\mathrm{Int}}(Int_i, \tilde{P}_q) + \mathrm{dbf}^{\mathrm{Y}}_{\mathrm{Int}}(Int_i, \tilde{P}_q)) \\ &\quad + \mathrm{dbf}^{\mathrm{tail:X}}_{\mathrm{Int}}(Int_i, \tilde{P}_q) + \mathrm{dbf}^{\mathrm{tail:Y}}_{\mathrm{Int}}(Int_i, \tilde{P}_q)\ \forall i, T^{Int}_i < S\end{aligned} \tag{62}$$

where $\mathrm{dbf}^{\mathrm{X}}_{\mathrm{Int}}(Int_i, \tilde{P}_q)$ and $\mathrm{dbf}^{\mathrm{Y}}_{\mathrm{Int}}(Int_i, \tilde{P}_q)$ are upper bounds on the demand by Int_i inside the reserves (of length $Res^{len}_x[\tilde{P}_q]$ and $Res^{len}_y[\tilde{P}_q]$) of server $\tilde{P}_q$ in a single time slot (of length S). Similarly for $\mathrm{dbf}^{\mathrm{tail:X}}_{\mathrm{Int}}(Int_i, \tilde{P}_q)$ and $\mathrm{dbf}^{\mathrm{tail:Y}}_{\mathrm{Int}}(Int_i, \tilde{P}_q)$, but over the remaining time interval (i.e. the "tail") of length t^{tail}. These terms, in turn, are derived as:

$$\mathrm{dbf}^{\mathrm{X}}_{\mathrm{Int}}(Int_i, \tilde{P}_q) = \mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, Res^{len}_x[\tilde{P}_q]) \tag{63}$$

$$\mathrm{dbf}^{\mathrm{Y}}_{\mathrm{Int}}(Int_i, \tilde{P}_q) = \mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, Res^{len}_y[\tilde{P}_q]) \tag{64}$$

$$\begin{aligned}&\mathrm{dbf}^{\mathrm{tail:X}}_{\mathrm{Int}}(Int_i, \tilde{P}_q) = \\ &\begin{cases} 0 & \text{if } t^{tail} \le \varOmega \\ \mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, t^{tail} - \varOmega) & \text{if } \varOmega < t^{tail} \le \varOmega + Res^{len}_x[\tilde{P}_q] \\ \mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, Res^{len}_x[\tilde{P}_q]) & \text{if } \varOmega + Res^{len}_x[\tilde{P}_q] < t^{tail} \le S \end{cases}\end{aligned} \tag{65}$$

$$\begin{aligned}&\mathrm{dbf}^{\mathrm{tail:Y}}_{\mathrm{Int}}(Int_i, \tilde{P}_q) = \\ &\begin{cases} 0 & \text{if } t^{tail} \le 2 \cdot \varOmega + Res^{len}_x[\tilde{P}_q] \\ \mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, t^{tail} - 2 \cdot \varOmega + Res^{len}_x[\tilde{P}_q]) & \text{if } 2 \cdot \varOmega + Res^{len}_x[\tilde{P}_q] < t^{tail} \le S \end{cases}\end{aligned} \tag{66}$$

As for non-split servers, often, though not always, $\mathrm{dbf}^{\mathrm{(split:bursty)}}_{\mathrm{Int}}(Int_i, \tilde{P}_q, t)$ provides a less pessimistic estimate than $\mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, t)$, for interrupts with $T^{Int}_i < S$. Hence in the general case $\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Int}}(Int_i, t)$ is computed as:

$$\begin{aligned}&\mathrm{dbf}^{\mathrm{sb:split}}_{\mathrm{Int}}(Int_i, \tilde{P}_q, t) = \\ &\begin{cases} \mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, t) & \text{if } T^{Int}_i \ge S \\ \min\left(\mathrm{dbf}^{\mathrm{(continuous)}}_{\mathrm{Int}}(Int_i, t),\ \mathrm{dbf}^{\mathrm{(split:bursty)}}_{\mathrm{Int}}(Int_i, \tilde{P}_q, t)\right) & \text{if } T^{Int}_i < S \end{cases}\end{aligned} \tag{67}$$