Semi-Partitioned Hard Real-Time Scheduling with Restricted Migrations upon Identical Multiprocessor Platforms

François Dorin, Patrick Meumeu Yomsi, Joël Goossens, Pascal Richard

Semi-Partitioned Hard Real-Time Scheduling with Restricted Migrations upon Identical Multiprocessor Platforms

François Dorin[1]*
dorinfr@ensma.fr

Patrick Meumeu Yomsi[2]†
patrick.meumeu.yomsi@ulb.ac.be

Joël Goossens[2]
joel.goossens@ulb.ac.be

Pascal Richard[1]
richardp@ensma.fr

Abstract

Algorithms based on semi-partitioned *scheduling have been proposed as a viable alternative between the two extreme ones based on* global *and* partitioned *scheduling. In particular, allowing migration to occur only for few tasks which cannot be assigned to any individual processor, while most tasks are assigned to specific processors, considerably reduces the runtime overhead compared to* global *scheduling on the one hand, and improve both the schedulability and the system utilization factor compared to* partitioned *scheduling on the other hand.*

In this paper, we address the preemptive scheduling problem of hard real-time systems composed of sporadic constrained-deadline tasks upon identical *multiprocessor platforms. We propose a new algorithm and a scheduling paradigm based on the concept of semi-partitioned scheduling with restricted migrations in which jobs are not allowed to migrate, but two subsequent jobs of a task can be assigned to different processors by following a* periodic strategy.

1. Introduction

In this work, we consider the preemptive scheduling of hard real-time sporadic tasks upon *identical* multiprocessors. Even though the *Earliest Deadline First* (EDF) algorithm [13] turned out to be no longer optimal in terms of schedulability upon multiprocessor platforms [7], many alternative algorithms based on this scheduling policy have been developed due to its optimality upon uniprocessor platforms [13]. Again, the primary focus for designers is at improving the worst-case system utilization factor with guaranteeing all tasks to meet deadlines. Unfortunately, most of the algorithms developed in the literature suffer from a trade-off between the theoretical schedulability and the practical overhead of the system at run-time: achieving high system utilization factor leads to complex computations. Up to now, solutions are still widely discussed.

In recent years, *multicore architectures*, which include several processors upon a single chip, have been the preferred platform for many embedded applications. This is because they have been considered as a solution to the "thermal roadblock" imposed by single-core designs. Most chip makers such as Intel, AMD and Sun have released dual-core chips, and few designs with more than two cores have been released as well. However given such a platform, the question of scheduling hard real-time systems becomes more complicated, and thus, has received considerable attention [3, 8]. For such systems, most results have been derived under either *global* or *partitioned* scheduling techniques. While both approaches might be viable, each has serious drawbacks on this platform and none will likely utilize the system very well while considering EDF scheduling policy.

In *global* scheduling [6], all tasks are stored in a single priority-ordered queue. The global scheduler selects for execution the highest priority tasks from this queue. Unfortunately, algorithms based on this technique may lead to runtime overheads that are prohibitive. This is due to the fact that tasks are allowed to migrate from one processor (CPU) to another in order to complete their executions and each migration cost may not be negligible. Moreover, Dhall et al. showed that global EDF may cause a deadline to be missed if the total utilization factor of a task set is slightly greater than 1 [7].

In *partitioned* scheduling [5], tasks are first assigned statically to processors. Once the task assignment is done, each processor uses independently its local scheduler. Unfortunately, algorithms based on this technique may lead to task systems that are schedulable if and only if some tasks are *not* partitioned. Moreover, Lopez et al. showed that the total utilization factor of a schedulable system using this technique is at most $0.5(m+1)$, where m is the number of processors in the system.

These two scheduling techniques are *incomparable* — at least for priority driven schedules —, that is, there are

*PhD candidate of the LISI - ENSMA, France.
†Postdoctoral researcher of the FNRS, Belgium.
[1]LISI - ENSMA - Université de Poitiers – 1 Rue C. Ader, B.P. 40109 – 86961 Chasseneuil du Poitou - France.
[2]Université Libre de Bruxelles (ULB) – 50 Avenue F. D. Roosevelt, C.P. 212 – 1050 Brussels - Belgium.

systems which are schedulable with partitioning and not by global and conversely. It thus follows that the effectiveness of a scheduling algorithm depends not only on its runtime overhead, but also its ability to schedule feasible task systems.

Recent work [2, 10, 11] came up with a novel and promising technique called *semi-partitioned scheduling* with the main objectives of reducing the runtime overhead and improving both the schedulability and the system utilization factor. In semi-partitioned scheduling techniques, most tasks, called *non-migrating tasks*, are fixed to specific processors in order to reduce runtime overhead, while few tasks, called *migrating tasks*, migrate across processors in order to improve both the schedulability and the system utilization factor. However, the migration costs depend on the instant each migrating task is migrated during execution: migrating a task during the execution of one of its jobs is more time consuming than migrating it at the instant of its activation. As such, we can distinguish between two levels of migration: *job migration* where a job is allowed to execute on different processors [2] and *task migration* where task migration is allowed, but job migration is *forbidden*. Between the two techniques, the task migration is the one which minimizes the migration costs. To the best of our knowledge, only one solution using the latter semi-partitioned scheduling technique was established but it is only applicable to soft real-time systems [1].

In this paper, we study the scheduling of hard real-time systems composed of sporadic constrained-deadline tasks upon *identical* multiprocessor platforms. For such platforms, all the processors have the same computing capacities.

Related work. On the road to solve the above mentioned problem, sound results using semi-partitioned scheduling techniques have been obtained by S. Kato in terms of both schedulability and system utilization factor. However, these results are all based on "*job-splitting*" strategies [11], and consequently, may still lead to prohibitive runtime overheads for the system. Indeed, the main idea of S. Kato consists in using a "*job-splitting*" (i.e., job migration is allowed) strategy based on a specific algorithm for tasks which cannot be scheduled by following the *First Fit Decreasing* (FFD) algorithm [12]. Figure 1 illustrates that the entire portion of job τ_k is not partitioned but it is split into $\tau_{k,1}$, $\tau_{k,2}$ and $\tau_{k,3}$ which are partitioned upon CPUs π_1, π_2 and π_3, respectively. The amount of each share is such a value that fills the assigning processor to capacity without timing violations.

Due to such a "*job-splitting*" strategy, it is necessary to have a mechanism which ensure that each job cannot be executed in parallel upon different processors. Such a mechanism has been provided by the introduction of *execution windows*[3]. Using this mechanism, each share can execute only during its execution window. Hence, multiple executions of the same job upon several processors at the same time are prohibited by guaranteeing that the execution windows do not overlap [12].

[3] An execution window for a job is a time span during which it is allowed to execute.

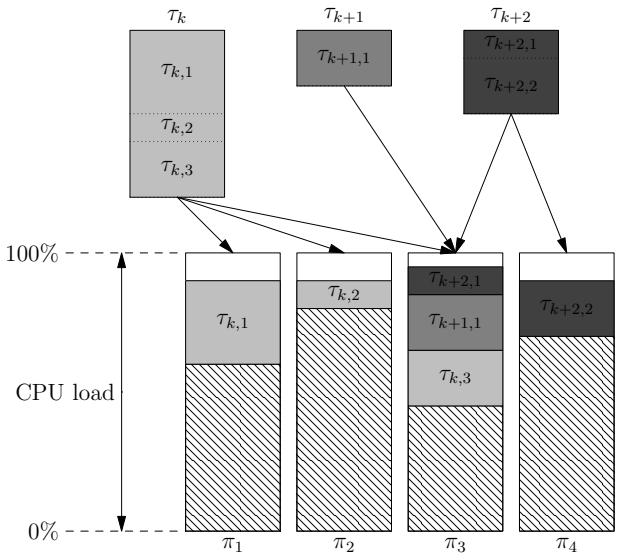

Figure 1. S. Kato's algorithm [12].

This research. In this paper, we propose a new algorithm and a scheduling paradigm based on the concept of semi-partitioned scheduling with *restricted migrations*: jobs are not allowed to migrate, but two subsequent jobs of a task can be assigned to different processors by following a *periodic strategy*. Our intuitive idea is to limit the runtime overhead as it may still be prohibitive for the system when using a *job-splitting* strategy. The algorithm starts with a classical step where tasks are partitioned among processors using the FFD algorithm for sake of comparison of our results to those obtained by S. Kato. Then, for the remaining tasks (i.e., those whose all the jobs cannot be assigned to one processor by following FFD without exceeding its capacity), a semi-partitioning scheduling technique with restricted migrations is used by following a periodic strategy detailed later on.

Paper organization. The remainder of this paper is structured as follows. Section 2 presents the task model and the platform that are used throughout the paper. Section 3 provides the principles of the proposed algorithm. Section 4 elaborates the conditions under which the system is schedulable. Section 5 presents experimental results. Finally, Section 6 concludes the paper and proposes future work.

2. System model

We consider the preemptive scheduling of a hard real-time system $\tau \stackrel{\text{def}}{=} \{\tau_1, \tau_2, \ldots, \tau_n\}$ comprised of n tasks upon

m identical processors. In this case, all the processors have the same computing capacities. The k^{th} processor is denoted π_k. Each task τ_i is a sporadic constrained-deadline task characterized by three parameters (C_i, D_i, T_i) where C_i is the Worst Case Execution Time (WCET), $D_i \leq T_i$ is the relative deadline and T_i is the minimum inter-arrival time between two consecutive releases of τ_i. These parameters are given with the interpretation that task τ_i generates an infinite number of successive jobs $\tau_{i,j}$ with execution requirement of at most C_i each, arriving at time $a_{i,j}$ such that $a_{i,j+1} - a_{i,j} \geq T_i$ and that must completes within $[a_{i,j}, d_{i,j})$ where $d_{i,j} \stackrel{\text{def}}{=} a_{i,j} + D_i$, the absolute deadline of job $\tau_{i,j}$.

The *utilization factor* of task τ_i is defined as $u_i \stackrel{\text{def}}{=} C_i/T_i$ and the *total utilization factor* of task system τ is defined as $U_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{i=1}^{n} u_i$. A task system is said to *fully utilize* the available processing capacity if its total utilization factor equals the number of processors (m). The *maximum* utilization factor of any task in τ is denoted $u_{\max}(\tau)$. A task system is *preemptive* if the execution of any of its jobs may be interrupted and resumed later due to the execution of another job with a higher priority. In this paper, we consider preemptive scheduling policies and we place no constraints on total utilization factor.

We consider that tasks are scheduled by following the *Earliest Deadline First* (EDF) scheduler with *restricted migrations*. That is, on the one hand, the shorter the absolute deadline of a job the higher its priority. On the other hand, tasks can migrate from one processor to another but when a job has started upon a given processor then it must complete without any migration. Such a strategy is a compromise between task partitioning and global scheduling strategies.

We assume that all the tasks are independent, that is, there is no communication, no precedence constraint and no shared resource (except for the processors) between tasks. We assume that the jobs of a task cannot be executed in parallel, that is, for any i and j, $\tau_{i,j}$ cannot be executed in parallel on more than one processor. Moreover, this research assumes that the costs of preempting and migrating tasks are included in the WCET, which makes sense since we limit migrations at task arrival instants and preemptions at task arrival or completion instants (EDF is the local scheduler).

3. Principle of the algorithm

In this section, we provide to the reader the main steps of our algorithm. Its design is based on the concept of semi-partitioned scheduling with *restricted migrations* and consists of two phases:
(1) *The assigning phase*: here, each non-migrating task is assigned to a specific CPU by following the FFD algorithm and the jobs of each migrating tasks are assigned to CPUs by following a *periodic strategy*.
(2) *The scheduling phase*: here, each migrating task is modeled upon each CPU as a *multiframe* task and thus the schedulability analysis focuses on each CPU individually, using the rich and extensive results from the uniprocessor scheduling theory.

3.1. Semi-partitioning scheduling algorithm

This section describes our semi-partitioning algorithm based on the concept of semi-partitioned scheduling. Like traditional partitioned scheduling algorithms, the schedulers have the same scheduling policy upon each CPU, that is EDF in this case, but each of them is not completely independent, because several CPUs may share a migrating task. The principle of the algorithm is as follows.

- Sort the tasks in decreasing order of their utilization factor.
- For each task considered individually:
  - Select one CPU and set it as the one with the smallest index.
  - If the current task is schedulable upon the selected CPU while using the FFD algorithm (i.e. all jobs meet their deadlines), then assign the task to this CPU.
  - If the current task is neither schedulable upon the selected CPU and nor upon the others by using the FFD algorithm, then determine the number of jobs that can be scheduled upon the selected CPU w.r.t. an EDF scheduler. Assign those jobs to the selected CPU by using the multiframe task approach defined earlier. Then, move to the next CPU. Repeat this process until all the jobs of the task are assigned to a CPU.

In the end, the intuitive idea behind our algorithm is to consider one task at each step and defines the minimum number of CPUs required to execute all its jobs (this takes at most m iterations, where m is the number of CPUs). At the beginning, the algorithm performs using an FFD algorithm. Then, whenever a task cannot be assigned to a particular CPU, its jobs are assigned to a subset of CPUs belonging to the platform (at most m). Thus, the algorithm requires at most $O(nm)$ schedulability tests.

3.2. The assigning phase

In this phase, each task τ_k is assigned to a particular processor π_j by following the FFD algorithm, as long as the task does not cause the system not to be schedulable upon π_j. That is, $\text{Load}(\tau^{\pi_j}) \stackrel{\text{def}}{=} \sup_{(t \geq 0)} \frac{\text{DBF}(\tau^{\pi_j}, t)}{t} \leq 1$, where τ^{π_j} denotes the subset of tasks assigned to CPU π_j and DBF is the classical Demand Bound Function [4], $\text{DBF}(\tau^{\pi_j}, t) \stackrel{\text{def}}{=} \sum_{\tau_k \in \tau^{\pi_j}} \left\lfloor \frac{t+T_k-D_k}{T_k} \right\rfloor \cdot C_k$. Such a task is classified into a *non-migrating task*. Now, if $\text{Load}(\tau^{\pi_j}) > 1$, that is, all the jobs of a task cannot be assigned to the same processor, then the task is classified into a *migrating*

Algorithm 1: Semi-Partitioning Algorithm

```
Input: m, τ = {τ_1, ..., τ_n}, parameter K.
Output: True and A if τ is schedulable, False otherwise.
Function SemiPart (in m, in K, in τ, out A)
begin
    for (i=1 ... n) do
        Sched ← False;
        /*We try to schedule the task using FFD */
        for (j=1 ... m) do
            if τ_i schedulable on π_j then
                τ_i is assigned to π_j;
                Sched ← True;
        if (Sched == False) then
            /*Task τ_i is a migrating task. We perform
            the job assignment among the CPUs by
            calling Algorithm 3 */
            if Algo2(τ_i, K, A) == False then
                /*τ_i is not schedulable */
                return False;
    return True;
```

task. The jobs are assigned to several processors by following a *periodic strategy* and are executed upon these by using a semi-partitioned scheduling with restricted migrations. Details on this strategy, obviously chosen for sake of simplicity during the implementations, are provided in the Section 3.3.

Example. In order to illustrate this strategy for a system with a single *migrating task* task $\tau_i = (C_i, D_i, T_i)$, we assume that τ_i has been assigned to a set of CPUs containing at least π_1 and π_2 with the periodic sequence $\sigma \stackrel{\text{def}}{=} (\pi_1, \pi_2, \pi_1)$. This means, the first job of τ_i will be assigned to π_1, the second one to π_2, the third one to processor π_1 and from the fourth job of τ_i, this very same process will repeat *cyclically* upon processors π_1 and π_2. Figure 2 depicts this job assignment.

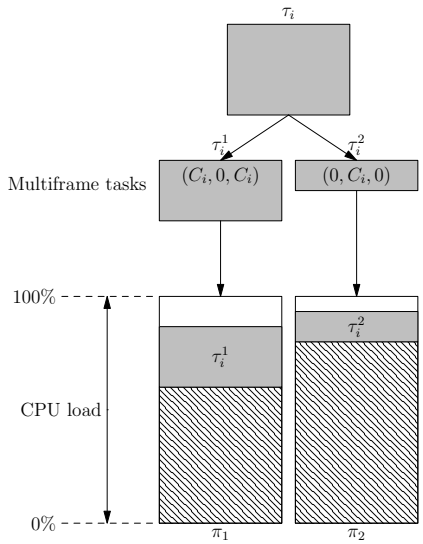

Figure 2. Periodic job assignment.

From the schedulability point of view, task τ_i will be duplicated upon CPUs π_1 and π_2 and according to the scheduling phase, it will be seen as two multiframe tasks (see [14] for further details on multiframe tasks). These multiframe tasks will be denoted by τ_i^1 and τ_i^2 and will have the following parameters: $\tau_i^1 \stackrel{\text{def}}{=} ((C_i, 0, C_i), D_i, T_i)$ and $\tau_i^2 \stackrel{\text{def}}{=} ((0, C_i, 0), D_i, T_i)$. As such, we will be able to perform the schedulability analysis upon each CPU individually, using uniprocessor approaches, already well understood.

3.3. Periodic job assignment strategy

Since each task consists of an infinite number of jobs, a potentially infinite number of periodic job assignment strategies can be defined for assigning migrating tasks to the CPUs. In this section, we propose two algorithms to achieve this: The *most regular-job pattern algorithm* and the *alternative-job pattern algorithm*.

In this paper, since each migrating task is modeled as a multiframe task upon each CPU it has at least one job assigned to, we assume that the number of frames obtained from each *migrating task* is equal to a non-negative integer K which is known beforehand for sake of readability. This assumption will be relaxed in future work.

Let $A[1 \cdots n, 1 \cdots m]$ be a matrix of integers where $A[i, j] \stackrel{\text{def}}{=} x$ (with $1 \leq i \leq n$ and $1 \leq j \leq m$) indicates that x jobs among K consecutive jobs of task τ_i will be executed upon processor π_j. In the following, we provide the reader with details on two algorithms used to initialize matrix $A[1 \cdots n, 1 \cdots m]$, while guaranteeing for every task τ_i that $\sum_{j=1,\ldots,m} A[i, j] = K$.

Before going any further in this paper, it is worth noticing that if $K = 1$, then task migrations are forbidden. Thus, our model extends the classical partitioning scheduling model. If the $A[i, j]$ are known beforehand, then the *most regular-job pattern algorithm* is considered, otherwise we consider the *alternative-job pattern algorithm*.

Most regular-job pattern algorithm

A *uniform assignment* of jobs of each migrating task τ_i among the subset of CPUs $\mathcal{S}_i$ upon which τ_i will execute at least one job seems to be a good idea at first glance (but we have no theoretical result proving that). For this reason we introduce the principle laying behind such a strategy [9]. In $\mathcal{S}_i$, we assume that the CPUs are ranged in a non-decreasing index-order. If $\mathcal{S}_i = \emptyset$, then the system is clearly not schedulable.

The job assignment is performed according to the following two steps for task τ_i.

- *Step 1.* The matrix A is computed thanks to Algorithm 1 (see Section 3.1 for details).
- *Step 2.* The assignment sequence σ of task τ_i is defined through K sub-sequences as follows.

$$\sigma \stackrel{\text{def}}{=} (\sigma_0, \sigma_1, \ldots, \sigma_\ell, \ldots, \sigma_{K-1}) \qquad (1)$$

where the $(\ell+1)^{th}$ sub-sequence σ_ℓ (with $\ell = 0,\ldots,K-1$) is given in turn by the following m-tuple:

$$\sigma_\ell \stackrel{\text{def}}{=} (\sigma_\ell^1, \sigma_\ell^2, \ldots, \sigma_\ell^m) \quad (2)$$

To define sub-sequence σ_ℓ using the uniform assignment pattern, there is at most one job per CPU each time w.r.t. Equation 3. The ℓ^{th} job of τ_i will be assigned to π_j if and only if:

$$\sigma_\ell^j \stackrel{\text{def}}{=} \left\lceil \frac{\ell+1}{K} \cdot A[i,j] \right\rceil - \left\lceil \frac{\ell}{K} \cdot A[i,j] \right\rceil = 1 \quad (3)$$

The goal is to assign $A[i,j]$ jobs among K consecutive jobs of task τ_i to CPU π_j by following a step-case function. For the ℓ^{th} job, σ_ℓ^j yields 1 when the job is assigned to π_j and 0 otherwise. Figure 3 illustrates Equation 3 for job assignment of τ_i to CPU π_j when $K = 11$ and $A[i,j] = 4$.

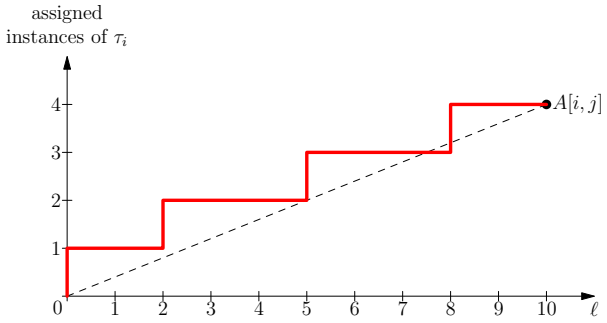

Figure 3. Job assignment sequence to π_1.

Thanks to these rules, it follows, depending on the value of parameter K, that a direct advantage of this cyclic job assignment is its ability to considerably reduces the number of task migrations.

Example. In order to illustrate our claim, consider a platform comprised of 3 identical CPUs π_1, π_2, π_3. We assume that $K = 11$. For a migrating task τ_i, let us consider that $A[i,1] = 4$, $A[i,2] = 2$ and $A[i,3] = 5$, meaning that 4 jobs of τ_i are assigned to CPU π_1, 2 jobs to CPU π_2 and 5 jobs to CPU π_3. The computations for defining the job assignment sub-sequences using Equation 3 are summarized in Table 1.

ℓ	0	1	2	3	4	5	6	7	8	9	10	total
π_1	1	0	1	0	0	1	0	0	1	0	0	4
π_2	1	0	0	0	0	1	0	0	0	0	0	2
π_3	1	0	1	0	1	0	1	0	1	0	0	5

Table 1. Most regular-job assignment sequence

At step $\ell = 5$ for instance, one job of τ_i will be assigned to π_1, then one job to π_2, and no job will be assigned to π_3. Whenever $\sigma_k^j = 0$ $(\forall j)$, as this is case at steps 1, 3, 7 and 10 in Table 1, then none of the jobs is assigned to none of the CPUs. Such a sub-sequence can be ignored while defining the cyclic assignment controller. In this example, the complete assignment sequence, that will be used to define the cyclic assignment controller for jobs of task τ_i is derived as follows.

$$\begin{aligned} \sigma &= (\sigma_0, \sigma_1, \ldots, \sigma_{K-1}) \\ &= (\pi_1, \pi_2, \pi_3, \pi_1, \pi_3, \pi_3, \pi_1, \pi_2, \pi_3, \pi_1, \pi_3) \end{aligned}$$

While using this strategy, even though this algorithm considerably reduces the number of task migrations, we have a recursion problem since to determine the job-pattern we need actually to know the job-pattern. Indeed, in order to know that $A[i,j]$ jobs of task τ_i are assigned to CPU π_j, we need to perform a schedulability test. However, such a test needs the pattern of the multiframe tasks assigned to each CPU to be known beforehand. A solution to this recursion problem is to use a schedulability test which requires *only* the number of jobs. That is, the one based on a worst-case scenario, which necessarily introduces a lot of pessimism.

The next job-pattern determination technique will address this drawback and consequently schedules a larger number of task systems.

Alternative job-pattern algorithm

This cyclic job assignment algorithm has been designed in order to overcome the drawback highlighted in the previous one (see algorithms 2–3). It considers each CPU individually and compute the matrix A incrementally. Initially, $A[i,1]$ is initialized at K. If the obtained multiframe task is not schedulable upon π_1, then $A[i,1]$ is decremented to $K-1$. The decrement is repeated until we get the largest number jobs of τ_i which are schedulable upon π_1. In the previous example, integers between 11 and 5 have been tested without success, but $A[i,1] = 4$ succeeded, then the multiframe task which will be considered upon this CPU is $\tau_i^1 = ((C_i,0,C_i,0,0,C_i,0,0,C_i,0,0),D_i,T_i)$ using *Step 2* of the previous method (Equation 3 in particular). In addition, instead of directly considering a multiframe task with K frames each time, this alternative algorithm allows us to temporarily consider multiframe tasks with a number of frames equal to the number of remaining jobs.

In order to determine the multiframe task τ_i^2 that will be executed upon CPU π_2, we temporarily consider a multiframe task $\tau_i'^2$ with $(J = K - A[i,1])$ frames (corresponding to the number of remaining jobs for τ_i^2). This temporal multiframe task $\tau_i'^2$ is built by using the *Step 2* of the previous method (Equation 3 in particular), and then we determine its pattern by considering $\tau_i'^2 \stackrel{\text{def}}{=} ((\sigma_1^2 C_i, \ldots, \sigma_J^2 C_i), D_i, T_i)$ with $\sigma_\ell^2 \in \{0,1\}, 1 \le \ell \le J$. After this operation has been performed, the multiframe task τ_i^2 is provided with K frames by using τ_i^1 and $\tau_i'^2$, based on the following two rules: (*i*) If the j^{th} frame of τ_i^1 is nonzero, then the j^{th} frame of τ_i^2 equals zero. (*ii*) If the j^{th} frame of τ_i^1 is zero and corresponds to the q^{th}

zero-frame of τ_i^1, then the j^{th} frame of τ_i^2 is determined by using the value of the q^{th} frame of $\tau_i'^2$.

These rules can be generalized very easily. Suppose our aim is determining the multiframe task τ_i^k corresponding to τ_i upon CPU π_k. The steps to perform are as follows. We first determine the amount of remaining jobs $J \stackrel{\text{def}}{=} K - \sum_{q=1}^{k-1} A[i,q]$. Then, we compute $\tau_i'^k$ thanks to Equation 3, by using J rather than K. Finally, we determine τ_i^k by using the pseudo-code of Algorithm 2 where $\tau_i^\ell(j)$ denotes the j^{th} frame of τ_i^ℓ. That is, if $\tau_i^\ell = ((C_i,0,C_i),D_i,T_i)$, then $\tau_i^\ell(1) = C_i$, $\tau_i^\ell(2) = 0$ and $\tau_i^\ell(3) = C_i$.

Algorithm 2: Computation of τ_i^k

Input: K, k, $\tau = (\tau_i^1, \ldots, \tau_i^{(k-1)})$, $\tau_i'^k$
Output: Multiframe task τ_i^k
Function `Compute` (in $\tau_i'^k$, in K, in τ)
begin
  $q \leftarrow 1$;
  for *(j = 1 ... K)* **do**
    Free $\leftarrow$ True;
    $\zeta \leftarrow 1$;
    while *(*$\zeta < k$ **and** Free*)* **do**
      if $\tau_i^\zeta(j) = 0$ **then** $\zeta \leftarrow \zeta + 1$;
      else Free $\leftarrow$ False;
    if Free **then**
      $\tau_i^k(j) \leftarrow \tau_i'^k(q)$;
      $q \leftarrow q + 1$;
    else $\tau_i^k(j) = 0$;
  return τ_i^k ;

Algorithm 3: Jobs assignment for migrating task τ_i

Input: K, task τ_i
Output: True and A if τ_i is schedulable, False otherwise
Function `Algo2` (in τ_i, in K, out A)
begin
  RemJobs $\leftarrow K$; */* Remaining jobs*/*
  for *(*$k = 1 \cdots m$*)* **do**
    for *(*$j =$ RemJobs $\cdots 1$*)* **do**
      Compute $\tau_i'^k$ by using Equation 3 where K is replaced by RemJobs ;
      $\tau_i^k = \texttt{Compute}(\tau_i'^k, K, (\tau_i^1, \ldots, \tau_i^{(k-1)}))$;
      if *(*τ_i^k is schedulable on π_k*)* **then**
        $A[i,k] \leftarrow j$;
        RemJobs $\leftarrow$ RemJobs $- j$;
        Exit;
    if RemJobs $= 0$ **then return** True ;
  return False

Example. Consider the same task τ_i as in the previous example. Assuming only 4 jobs can be assigned to π_1, then the multiframe task τ_i^1 upon π_1 is $\tau_i^1 = ((C_i,0,C_i,0,0,C_i,0,0,C_i,0,0),D_i,T_i)$ by using Algorithm 2. If τ_i^1 is schedulable upon π_1, then we consider π_2 and we repeat the same process. If not, we try to assign jobs to π_1, now assuming only 3 jobs can be assigned this time.

In this example, we assume that 4 jobs have successfully been assigned to π_1. We now assume that neither 4, nor 3 jobs of τ_i cannot be assigned to π_2, but 2 jobs can. Computing the intermediate task $\tau_i'^2$ using $J = 11 - 4 = 7$ leads us to $\tau_i'^2 = ((C_i,0,0,C_i,0,0,0),D_i,T_i)$. By applying Algorithm 2 again, we obtain $\tau_i^2 = ((0,C_i,0,0,0,0,C_i,0,0,0,0),D_i,T_i)$.

We repeat the same process for CPU π_3. When trying to assign the 5 remaining jobs, computing $\tau_i'^3$ with $J = 5$ gives $\tau_i'^3 = ((C_i,C_i,C_i,C_i,C_i),D_i,T_i)$. Again, applying Algorithm 2, we obtain $\tau_i^3 = ((0,0,0,C_i,C_i,0,0,C_i,0,C_i,C_i),D_i,T_i)$.

4. Schedulability analysis

This section derives the schedulability conditions for our algorithm. Because each migrating task is modeled as a multiframe task on each CPU upon which it has a job assigned to, the schedulability analysis is performed on each CPU individually, using results from the uniprocessor scheduling theory. As tasks are scheduled upon each CPU according to an EDF scheduler, a specific analysis is necessary only for CPUs executing at least one multiframe task. For such a CPU, classical schedulability analysis approaches such as "*Processor Demand Analysis*" cannot applied, unfortunately. This is due to migrating tasks. We consider two scenarios to circumvent this issue: the *packed scenario* and the *scenario with pattern* and we develop a specific analysis based on an extension of the *Demand Bound Function* (DBF) [3,4] to multiframe tasks. We recall that a natural idea to cope with each migrating task is to duplicate it as many time as the number of CPUs it has a job assigned to and consider a multiframe task upon each CPU.

4.1. Packed scenario

As each migrating task τ_i is modeled as a collection of at most m multiframe tasks τ_i^1, τ_i^2, ..., τ_i^m, where τ_i^j is to be executed upon CPU π_j (m is the number of CPUs), this scenario considers the *worst-case*. This occurs when, for each multiframe task associated to τ_i, all nonzero execution requirements are at the beginning of the frames and the remaining execution requirements are set to zero. That is τ_i^k is modeled as $\tau_i^k \stackrel{\text{def}}{=} ((C_i,C_i,\ldots,C_i,0,\ldots,0),D_i,T_i)$, where C_i is the execution requirement of task τ_i. (see Appendix for the proof).

In order to define $\widehat{\text{DBF}}$ at time t (that is, our adapted Demand Bound Function for systems such as those considered in this paper), we need to define the contribution of each task in the time interval $[0,t)$. As K denote the number of execution requirements in a multiframe task τ_i^k associated to τ_i, let ℓ_i^k denote the number of nonzero execution requirements in τ_i^k. When using the packed scenario,

τ_i^k can be modeled as $\tau_i^k = ((\underbrace{C_i,\ldots,C_i}_{\ell_i^k \text{ elements}},0,\ldots,0),D_i,T_i)$. The challenge is to take into account that only ℓ_i^k jobs of task τ_i^k will contribute to the DBF in the time interval $[0,K\cdot T_i)$.

Based on Figure 4, the contribution to $\widehat{\text{DBF}}$ at time t for the multiframe task τ_i^k is determine as follows.

- First, consider the number of intervals of length $K\cdot T_i$ by time t. Letting s denote that number, we have:

$$s \stackrel{\text{def}}{=} \left\lfloor \frac{t}{K\cdot T_i} \right\rfloor \tag{4}$$

  As such, the contribution of τ_i^k to the DBF is at least $s\cdot \ell_i^k \cdot C_i$.

- Second, consider the time interval $[s\cdot K\cdot T_i,t)$, where several jobs may have their deadlines before time t. Assuming that all the jobs are assigned to the considered CPU, the number of jobs ("a") is given by:

$$a \stackrel{\text{def}}{=} \max\left(0, \left\lfloor \frac{(t \bmod K\cdot T_i) - D_i}{T_i} \right\rfloor + 1\right) \tag{5}$$

  Since at most ℓ_i^k jobs over K will be executed upon that CPU, then the exact contribution of τ_i^k in interval $[s\cdot K\cdot T_i,t)$ is given by: $\min(\ell_i^k,a)\cdot C_i$.

Putting all of these expressions together, the DBF of τ_i^k at time t is defined as follows.

$$\widehat{\text{DBF}}(\tau_i^k,t) \stackrel{\text{def}}{=} s\cdot \ell_i^k \cdot C_i + \min\left(\ell_i^k,a\right)\cdot C_i \tag{6}$$

Schedulability Test 1. Let $\tau = \{\tau_1,\tau_2,\ldots,\tau_n\}$ be a set of n constrained-deadline sporadic tasks to be scheduled upon an identical multiprocessor platform. For each selected CPU π, if $\mathcal{N} \subseteq \tau$ denotes the subset of *non-migrating tasks* upon π, then a *sufficient condition* for the system to be schedulable upon π is given by

$$\sum_{\tau_j\in\mathcal{N}} \text{DBF}(\tau_j,t) + \sum_{\tau_i^k\in\pi\setminus\mathcal{N}} \widehat{\text{DBF}}(\tau_i^k,t) \le t \quad \forall t \tag{7}$$

In Equation 7, τ_j is a *non-migrating task*, DBF is the classical Demand Bound Function and τ_i^k is a multiframe task assigned to π.

4.2. Scenario with pattern

This scenario, in contrast to the packed one which considers the worst-case, takes the pattern of job assignment provided by our algorithm into account. Indeed, implementations showed that the schedulability analysis based on the packed scenario may be too pessimistic, unfortunately.

The schedulability test developed in this section is similar to the one developed previously for packed scenarios in that it is also based on the DBF. However the pattern of job assignment to CPUs is taken into account. Therefore, the only noticeable difference comes from the second term of Equation 6 which is replaced by:

$$\max_{c=0}^{K-1}\left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K}\right) \tag{8}$$

In Expression 8, $nb_i(t) \stackrel{\text{def}}{=} \left\lfloor \frac{(t \bmod K\cdot T_i)-D_i}{T_i} \right\rfloor + 1$ and denotes the number of jobs of τ_i^k in the time interval $[s\cdot K\cdot T_i,t)$. This is, we compute the processor demand by considering that the "critical instant" coincide with the first job of the pattern, then the second and so on, until the K^{th} job. After that, we consider the maximum processor demand in order to capture the worst-case. Hence we obtain:

$$\widehat{\widehat{\text{DBF}}}(\tau_i^k,t) \stackrel{\text{def}}{=} s\cdot \ell_i^k \cdot C_i + \max_{c=0}^{K-1}\left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K}\right) \tag{9}$$

Schedulability Test 2. Let $\tau = \{\tau_1,\tau_2,\ldots,\tau_n\}$ be a set of n constrained-deadline sporadic tasks to be scheduled upon an identical multiprocessor platform. For each selected CPU π, the *sufficient condition* for the system to be schedulable upon π is as the previous one except that in Expression 7 the value of $\widehat{\text{DBF}}$ is now replaced by $\widehat{\widehat{\text{DBF}}}$.

Note that Lemma 1 in the Appendix provides a proof of the "*dominance*" of Schedulability Test 2 over Schedulability Test 1. That is, all systems that are schedulable using the Test 1 are also schedulable using Test 2.

5. Experimental results

In this section, we report on the results of experiments conducted using the theoretical results presented in Sections 3 and 4. These experiments help us to evaluate the performances our algorithms relative to both the FFD and the S. Kato algorithms. Moreover, they help us to point out the influence of some parameters such as the value of parameter K and the number of CPUs in the platform. We performed a statistical analysis based on the following characteristics: (*i*) the number of CPUs is chosen in the set $M \stackrel{\text{def}}{=} \{2,4,8,16,32,64\}$ for practical purpose. Indeed multicore platforms often have a number of cores which is a power of 2, (*ii*) the system utilization factor for each CPU varies between 0.70 and 1 using a step of 0.05, as the results are identical when the CPU utilization is less than 0.7.

During the simulations, 10.000 runs have been performed for each configuration of the pair (number of CPUs in the platform, system utilization factor). In the figures displayed below, a "*success ratio*" of 2% of an algorithm $\mathcal{A}$ over an algorithm $\mathcal{B}$ means that $\mathcal{A}$ leads to a schedulability ratio of y%, where $\mathcal{B}$ leads to a schedulability ratio $(y-2)$%.

For comparison reasons, the same protocol as the one described in [12] by S. Kato for generating task systems has been considered. That is:

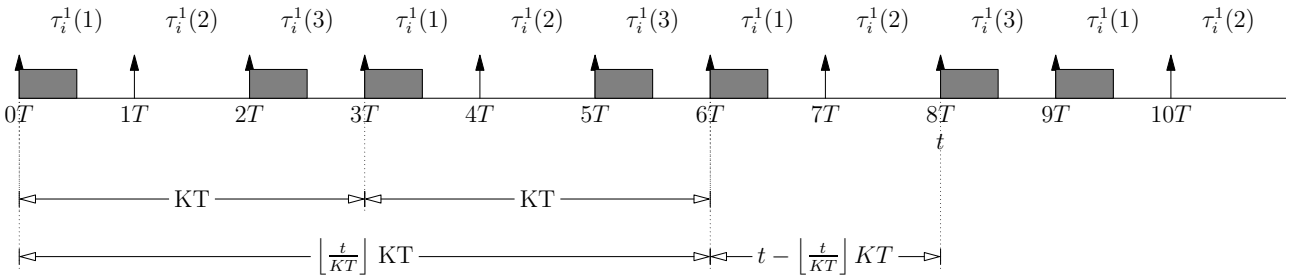

Figure 4. Illustration of the multiframe task $\tau_i^1 = ((C_i, 0, C_i), T_i, T_i)$.

1. the utilization factor of each task is randomly generated in the range $[0,1]$ with the constraint that the sum of utilization factors of all tasks must be equal to the utilization factor of the whole system,
2. the period T_i of task τ_i is randomly chosen in the range $[100, 3000]$,
3. the relative deadline D_i of task τ_i is set equal to its period,
4. the WCET C_i of task τ_i is computed from its period T_i and its utilization factor u_i as $C_i \stackrel{\text{def}}{=} u_i \cdot T_i$.

It is worth noticing that we have no control of the number of tasks composing the system. This number depends on the utilization factors of the tasks.

Figure 5 depicts the curve of the success ratio relative to parameter K for "*packed scenarios*". As we can see, the lower the value of K, the higher the success ratio. This result may be surprising and counter-intuitive at first glance as we would expect the contrary, that is, increasing K would improve the success ratio. A reason to this is provided by the pessimism introduced when packing all the nonzero frames at the beginning of each multiframe task. Indeed in this case, the processor demand is over-estimated at time $t = 0$ for each CPU upon which there is a multiframe task. To illustrate our claim, when $K = 2$, each migrating task is assigned to at most 2 CPUs, thus increasing the pessimism for these CPUs. Consequently, the larger the value of K, the larger the number of CPUs for which the processor demand will be over-estimated.

Figure 6 depicts the curve of the success ratio relative to parameter K for the "*scenario with pattern*". Here, in contrast to the results obtained for "*packed scenarios*", the curves behave as we would intuitively expect them. That is, increasing the value of K leads to a higher success ratio. This is due to the fact that a higher value of K allows the jobs of a migrating task to be assigned to a larger set of CPUs, thus reducing the global computation requirement upon each CPU.

Now, it is worth noticing that both the packed scenario and the scenario with pattern lead to the same algorithm when $K = 2$. In fact, while considering a scenario with pattern, the multiframe execution requirements which are allowed for a migrating task τ_i are $(C_i, 0)$ and $(0, C_i)$. Hence, from the schedulability analysis view point, the one that will be considered is $(C_i, 0)$, which corresponds to the packed scenario.

Figure 7 compares the performances of the algorithms which provided the best results using our approach during the simulations, that is, the one using packed scenarios when $K = 2$ and the one using scenarios with pattern when $K = 20$, and those of both the FFD and the S. Kato algorithms.

We can see that our algorithms always outperform the FFD algorithm and are a bit behind the one proposed by S. Kato in terms of success ratio. However, the results provided by our algorithms strongly challenge those produced by the S. Kato one for large number of CPUs. Note that our algorithms in contrast to the S. Kato one do not allow job migrations, thus limiting runtime overheads which may be prohibitive for the system. Hence, we believe that our approach is a promising path to go for more competitive algorithms and for practical use.

6. Conclusion and Future work

In this paper, the scheduling problem of hard real-time systems comprised of constrained-deadline sporadic tasks upon identical multiprocessor platforms is studied. A new algorithm and a scheduling paradigm based on the concept of semi-partitioned scheduling with restricted migrations has been presented together with its schedulability analysis. The effectiveness of our algorithm has been validated by several sets of simulations, showing that it strongly challenges the performances of the one proposed by S. Kato. Future work will address two issues. The first issue is relaxing the constraint on parameter K as we think that is possible to define a value K for each task such that the number of frames are kept as small as possible. The second issue is solving the optimization problem taking into account the number of migrations.

Acknowledgment. The authors would like to thank Ahmed Rahni from LISI/ENSMA for his insightful comments on the schedulability tests of multiframe tasks.

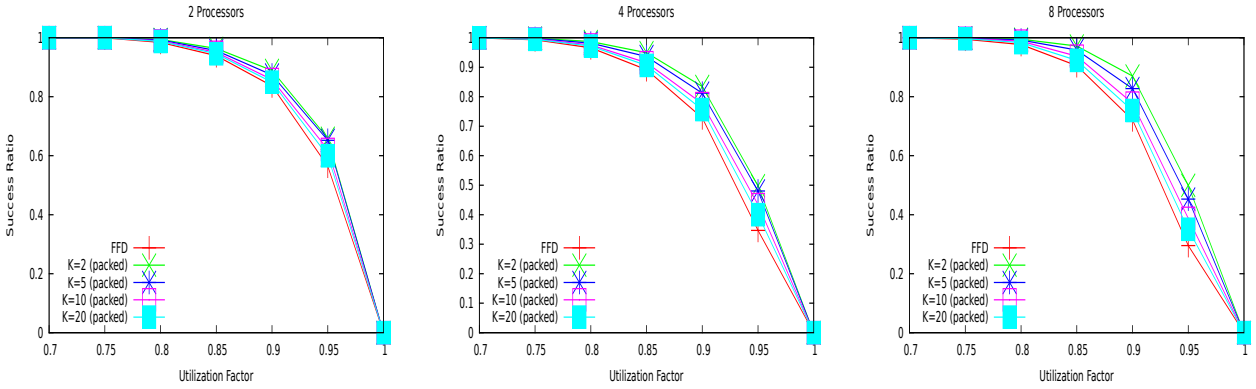

Figure 5. Success ratio relative to parameter K for packed scenarios

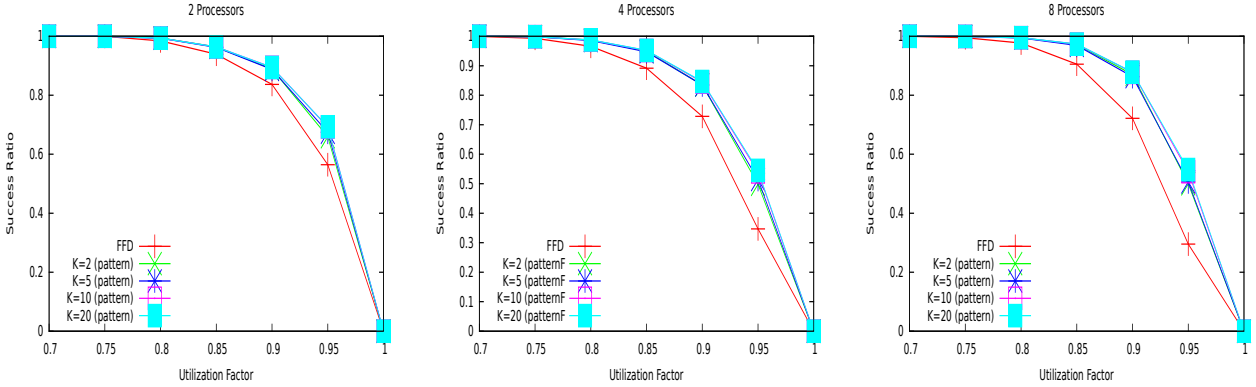

Figure 6. Success ratio relative to parameter K for scenarios with pattern

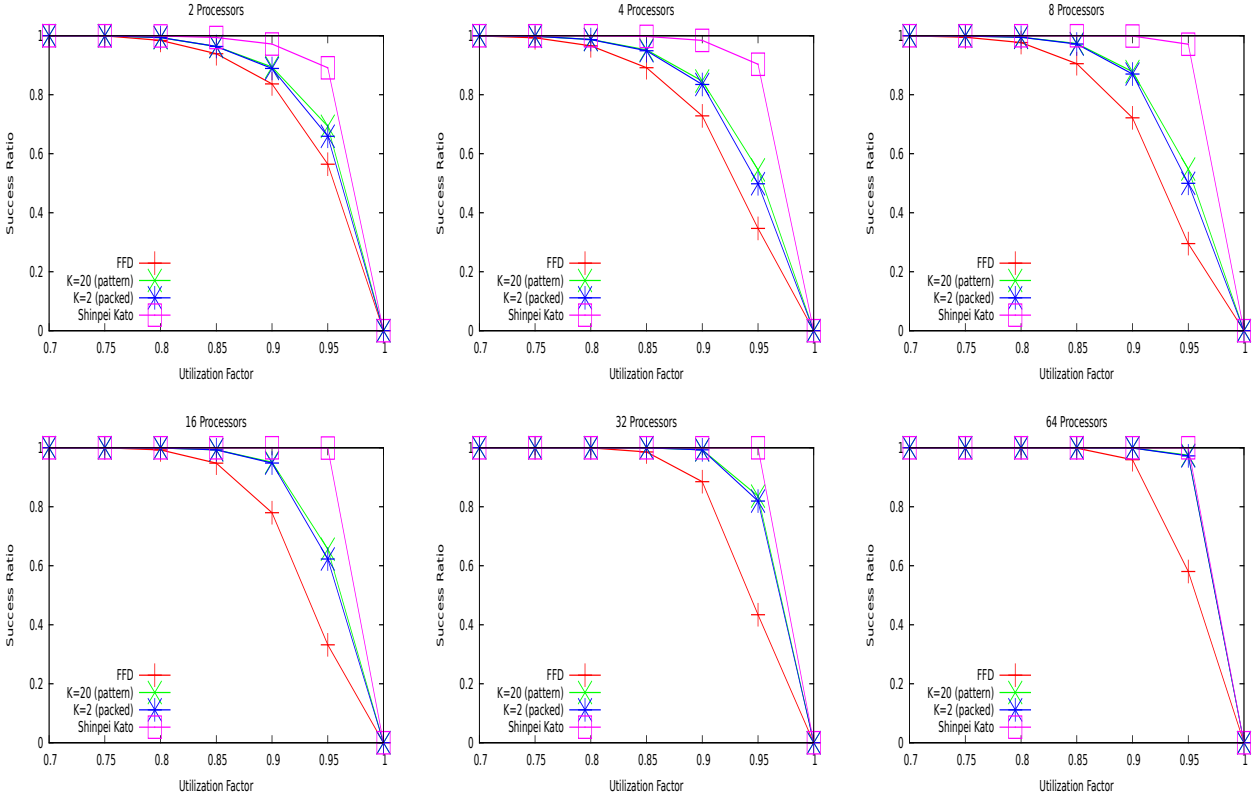

Figure 7. Comparison of FFD, packed- (K=2), pattern- (K=20) and S. Kato's algorithms

References

[1] J. H. Anderson, V. Bud, and U. C. Devi. An EDF-based scheduling for multiprocessor soft real-time systems. *In Proc. of the 17th Euromicro Conf. on Real-Time Systems*, pages 199–208, 2005.

[2] B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemptions. *In Proc. of the Euromicro Conference on Real-Time Systems*, pages 243–252, 2008.

[3] T. P. Baker and S. K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Realtime and Embedded Systems*. CRC Press, 2007.

[4] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Time-Critical Computing Systems*, 17:5–22, 1999.

[5] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Trans. on Computers 55 (7)*, pages 918–923, 2006.

[6] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Trans. Parallel Distrib. Syst.*, 20(4):553–566, 2009.

[7] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26:127–140, 1978.

[8] J. Goossens, S. Funk, and S. Baruah. Real-time scheduling on multiprocessors. *In Proc. of the 10th International Conference on Real-Time Systems*, pages 189–204, 2002.

[9] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967.

[10] S. Kato and N. Yamasaki. Real-time scheduling with task splitting on multiprocessors. *In Proc. of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 441–450, 2007.

[11] S. Kato and N. Yamasaki. Portioned EDF-based scheduling on multiprocessors. *In Proc. of the ACM International Conference on Embedded Software*, 2008.

[12] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. *In Proc. of the Euromicro Conference on Real-Time Systems*, pages 249–258, 2009.

[13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[14] A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Trans. on Software Engineering*, 23(10):635–645, 1997.

A. Appendix

Let τ_i be a multiframe task with a pattern $\sigma \stackrel{\text{def}}{=} (\sigma_1,\ldots,\sigma_K)$, that is, $\tau_i = ((\sigma_1 C_i,\ldots,\sigma_K C_i), D_i, T_i)$ with $\sigma_j \in \{0,1\}, 1 \le j \le K$. Let ℓ_i denotes the number of nonzero execution requirements. Let $\tau_i^{(p)}$ be the packed version of τ_i that is: $\sigma_j = 1$ if $j \le \ell_i$ and $\sigma_j = 0$, otherwise.

Lemma 1

$$\widehat{\widehat{\text{DBF}}}(\tau_i,t) \le \widehat{\widehat{\text{DBF}}}(\tau_i^{(p)},t) \tag{10}$$

Proof By definition:

$$\widehat{\widehat{\text{DBF}}}(\tau_i,t) = s\cdot\ell_i\cdot C_i + \max_{c=0}^{K-1}\left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K}\right)$$

In this Equation, the max term can be rewritten as:

$$\max_{c=0}^{K-1}\left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K}\right) = \max_{c=0}^{K-1}\left(C_i \sum_{j=c}^{c+nb_i(t)-1} \sigma_{i,j \bmod K}\right) \tag{11}$$

Note that since $\sigma_j \in \{0,1\}$ and we perform the sum of $nb_i(t)$ terms.

$$\sum_{j=c}^{c+nb_i(t)-1} \sigma_{j \bmod K} \le \min(\ell_i, \max(0, nb_i(t))) \tag{12}$$

Because we have at most ℓ_i nonzero execution requirements, so, we take the minimum between ℓ_i and $nb_i(t)$. As such,

$$\max_{c=0}^{K-1}\left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K}\right) \le \max_{c=0}^{K-1}(C_i \min(\ell_i, \max(0, nb_i(t)))) \tag{13}$$

leading to

$$\max_{c=0}^{K-1}\left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K}\right) \le C_i \min(\ell_i, \max(0, nb_i(t))) \tag{14}$$

Finally, $\widehat{\widehat{\text{DBF}}}(\tau_i,t) \le s\cdot\ell_i\cdot C_i + C_i\min(\ell_i,\max(0,nb_i(t)))$. That is, $\widehat{\widehat{\text{DBF}}}(\tau_i,t) \le \widehat{\widehat{\text{DBF}}}(\tau_i^{(p)},t)$. The lemma follows. ■

Theorem 1 *Let τ_i be a multiframe task. The worst-case pattern for τ_i is the packed pattern.*

Proof A pattern $\mathcal{A}$ is said worse than a pattern $\mathcal{B}$ if and only if $\mathcal{A}$ schedulable implies $\mathcal{B}$ also schedulable. If the packed pattern is schedulable and thanks to the previous lemma, then $\widehat{\widehat{\text{DBF}}}(\tau_i,t) \le \widehat{\widehat{\text{DBF}}}(\tau_i^p,t) \le t, \forall t$, that is, the pattern Σ is schedulable, and thus, the packed pattern is worse than Σ. Since Σ represent any pattern, the packed pattern is the worst-case pattern. The theorem follows. ■