The Demand Bound Function Interface of Distributed Sporadic Pipelines of Tasks Scheduled by EDF

Nicola Serreli, Giuseppe Lipari, Enrico Bini
Scuola Superiore Sant'Anna, Pisa, Italy
Email: {n.serreli,g.lipari,e.bini}@sssup.it

***Abstract*—In distributed real-time embedded systems (DRE), it is common to model an application as a set of task chains. Each chain is activated cyclically and must complete before an end-to-end deadline. Each task of the chain is bound to execute on a particular processing element.**

The complexity of designing and analyzing a DRE can be reduced by applying a component-based methodology: each pipeline can be seen as a component with its temporal characteristic summarized in its *interface*. Analysis can be carried out in two different steps: 1) derivation of the *temporal interface* of a component pipeline; 2) analysis of the whole system by integrating the temporal interfaces of the components.

In this paper, we propose to describe the temporal interface of a task pipeline by a set of *demand bound functions* (dbf), one per each node on which the pipeline executes, and we describe an algorithm for computing the dbfs. First, we show that the scenario of strictly periodic activations is not the worst when the pipelines are sporadically activated. Then, we propose an exact algorithm for computing the dbfs. We show by experimental analysis that the computation time of the algorithm on pipelines with reasonable size is below one second on common PCs. Finally, we estimate the pessimism introduced by our analysis with respect to holistic analysis by an extensive set of simulations.

I. Introduction

Today's applications are often developed by different vendors, each one providing separate components. As the application is distributed over several processing elements, components are of distributed nature as well. For example, this is the typical scenario in the automotive context [1], [2].

In the analysis of such a system it is of key importance to preserve the following properties:

1) each vendor provides only a synthetic information on the developed component (called component interface);
2) the integration of the components is made only on the information contained in the interface.

In real-time systems, a component is equipped also with a *temporal interface* that contains information related to the amount of computational resource required by the component over time. The analysis is then performed in two steps: in the first step, each component is analyzed in isolation, summarizing its temporal behavior with a (possibly small) set of temporal parameters. Such temporal parameters will be part of the component interface along with the functional and behavioral parameters. In the second step (integration), we must verify that the overall system is schedulable by integrating the temporal interfaces derived in the previous step.

In distributed real-time embedded (DRE) systems, a component is often modeled as a chain of tasks (also called transaction or pipeline) [3]. Each task of the pipeline is allocated on a (possibly different) processing node. The first task in the pipeline is activated periodically, or by external events characterized by a minimum interarria521(e)-338.91(l and processed may be much lower than the EE deadline for delivering the frames to the user.

When integrating the components (that we assume to be modeled by pipelines) it is important to check the willomplete before their EE deadline under worste -case conditions. In fixed priority systems, the *holistic analysis* [3], [4] consists in reducing the overall distributed schedulability problem into p single-node problems that can be solved using classical schedulability analysis. Each task is assigned a priority, and task parameters like offsets, jitters and response times are calculated so that the prece- schedulability problems depend on one another (i.e. the activation of an intermediate task, and
is found or the set is deemed not schedulable. Similar deadline instead of a fixed priority. Holistic analysis also allows to mix different schedulers on different nodes, as long as the designer is able to compute the worste -case

Unfortunately, the holistic analysis is unfit for component-based analysis, since it requires to know the parameters of the tasks of all pipelines (of the other components as well).n fact, at each step it is not possible depend on the presence of all other pipelines, so we cannot set the offsets and the jitters of intermediate tasks. In other words, following the holistic model we cannot easily compute the temporal interface of the component.

A different approach consists in fixing the offset and the relative deadline of intermediate tasks to appropriate values, so that precedence constraints are respected. In this way, each task can be treated independently of the others. This approach is called *slicing* in [9]. In practice, the EE deadline is sliced into non-overlapping *execution windows* for the task.

Following the slicing method, under EDF the temporal characteristics of the pipelines are abstracted by a set of *demand bound functions* (dbfs) [10], one for each node. Then, the first step consists in computing the dbf of every pipeline on every processor. This set of dbf (or an approximation of them) form of the temporal interface of the pipeline. Then, the integration analysis consists in summing all the dbfs for every node, and check that the resulting function never exceeds the computational power of the node.

This method is pessimistic with respect to the holistic analysis. In fact, fixing a-priori the offset of every task to the deadline of the preceding one adds additional constraints to the problem, reducing the chance of a system to result schedulable. However, in our opinion, the advantages of a component-based design methodology overcomes the loss of schedulability (that is however experimentally evaluated in this paper).

In this paper, we propose an algorithm to exactly compute the dbf of periodic and sporadic pipelines. We first show that, when the EE deadline is larger than the period, the worst case arrival pattern for a sporadic pipeline is not necessarily the periodic one. Then, we describe our algorithm, and compute its complexity, which is exponential in the number of tasks and in the ratio between the end-to-end deadline and the period. However, we show by experimental analysis that the computation time of the algorithm on pipelines with reasonable size is below one second on common PCs, which is acceptable for all practical uses. Finally, we estimate the pessimism introduced by our analysis with respect to the state of the art holistic algorithm in the literature.

II. Related work

The classic method for guaranteeing the EE deadline of chains of distributed tasks is the holistic analysis [3], [4], [6], [7]. In these approaches the run-time behavior of the tasks is iteratively simulated until it converges to a fixed point. As widely discussed earlier this approach is not well quited for component-based analysis.

Other approaches [1], [11] are based on the propagation of sequences of events [12] and analyze the system through the real-time calculus [13]. To best of our knowledge, however, the correlation between activation of consecutive activation of the same pipeline is lost introducing some pessimism in the analysis.

The use of the demand bound function was initially proposed by Baruah et al., for testing the schedulability of set of tasks scheduled by EDF on single processors [10]. This methodology is also known as "Processor Demand Criterion" [14]. The computation of the dbf was later extended to more complex task models, such as the generalized multiframe tasks [15]. Recently, Zhang and Burns [16] proposed a technique to reduce the number of points to check during analysis based on demand bound function.

The processor demand criterion has been extended to the analysis of distributed real-time pipelines by Rahni et al. [8]. However, their methodology is still based on the holistic analysis: the activation time of a task is set equal to the finishing time of the previous task in the pipeline.

In [9] authors proposed a methodology to analyze the schedulability of task graphs. The methodology also computes intermediate deadlines by using an heuristic approach, and it is based on the *slicing* approach: each task is assigned a slice that does not overlap with the slices of other tasks. Later [17] uses time slices to decouple the schedulability analysis of each node, reducing the complexity of the analysis. Such an approach improves the robustness of the schedule, and allows to analyze each pipeline in isolation. We recently proposed a heuristic algorithm for assigning intermediate task deadlines based on the slicing approach [18]. Our methods enables a component-based analysis.

In the context of component-based analysis, Lorente et al. [19] proposed the holistic analysis onto a set of virtual processors, rather than fully available ones. However the component interface was not specified.

A notable alternate analysis was proposed by Jayachandran and Abdelzaher [20], who developed several transformations to reduce the analysis of a distributed system to the single processor case. However, in their analysis, the isolation between transactions is not ensured.

III. System model and notation

A distributed real-time application is modeled by a set of pipelines $\{\mathcal{T}_1, \ldots, \mathcal{T}_m\}$. To simplify the presentation, since our work investigates each pipeline in isolation, throughout the paper we drop the index of the pipelines.

Pipeline $\mathcal{T}$ is composed by a set of n tasks $\{\tau_1, \ldots, \tau_n\}$. Task τ_i has a computation time C_i.

The first task τ_1 of the ℓ^{th} instance of the pipeline is activated at ℓ, that is called *absolute activation*, while tasks τ_i, with $i > 1$, are activated upon the completion of the preceding one τ_{i-1}.

We denote by τ_i^ℓ the ℓ^{th} instance of the task τ_i, that we often call job in accordance with a commonly adopted terminology. We consider sporadic pipelines with minimum iterarrival time T. Hence we have

$$\ell - \ell-1 \geq T. \tag{1}$$

To describe a possible scenario of activations for the sporadic pipeline under analysis, we need to list the possible values of absolute activations ℓ. We label the instance of the pipeline under analysis by . Moreover, we operate a time translation, so to set the activation of this pipeline at time reference . Therefore, we set $^0 =$.

The *future instances* (w.r.t. the -the one under analysis) will be denoted by positive indexes $\ell >$, and their

absolute activations by ^{1}, ^{2},... Similarly, the *past instances* will be denoted by negative indexes $\ell <$, and their absolute activations by $^{-1}$, $^{-2}$,...

A sequence $\{\ ^\ell\}_{\ell\in\mathbb{Z}}$ represents a possible scenario of absolute activations of all the instances. However, when we are investigating the demand on a finite interval $[t_0, t_1]$, only a finite number of instances may overlap with the interval. Hence we represent the *sporadic activation pattern* as follows

$$\overline{\ } = (\ ^{-k_0}, \ldots, \ ^{k_1}) \tag{2}$$

where indexes of the instances are taken from $-k_0$ to k_1,

$$k_0 = \left\lceil \frac{D - t_0}{T} \right\rceil - 1 \quad k_1 = \left\lceil \frac{t_1}{T} \right\rceil - 1. \tag{3}$$

In Figure 1 we show the interpretation if the instance indexes $-k_0$ and k_1 compared with an interval $[t_0, t_1]$.

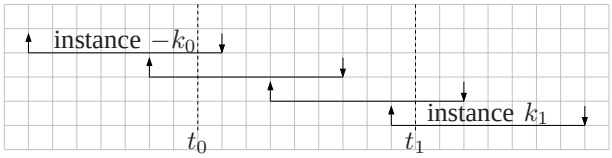

Figure 1: Example of calculation of k_0 and k_1.

We remark that, similarly to what it happens in multiprocessor scheduling [21], activating the pipelines as early as possible (i.e. periodically) **is not the worst-case** for the activation pattern. In Section V we show this by an example.

Each pipeline $\mathcal{T}$ has an *end-to-end deadline* D that is the maximum tolerable time from the activation of the first task τ_1 to the completion of the last task τ_n. Since the analysis of the constrained deadline ($D \leq T$) is a straightforward extension of the classic analysis, throughout the paper we always assume $D > T$. In such a case, it may happen that a task is activated before its previous instance has completed. In this paper, we assume that the activations of each task are served in a FIFO order.

The application is distributed across p processing nodes, and each task τ_i of the pipeline $\mathcal{T}$ is mapped onto computational node $x_i \in \{1, \ldots, p\}$. Hence, we define $\mathcal{T}_k = \{\tau_i \in \mathcal{T} : x_i = k\}$ as the subset of tasks in $\mathcal{T}$ mapped onto node k and n_k as the cardinality of $\mathcal{T}_k$.

The delay due to network communication can be easily taken into account by considering the network as a special processing node, and messages as tasks. The methodology presented in this paper is valid also when different scheduling policies are used on the processing nodes. However, to simplify the presentation, in this paper we make two assumptions: we neglect the delay due to network communication (for example, restricting to a multiprocessor system with shared memory); and we assume EDF as the only scheduling algorithm in the system.

Each task is assigned an *intermediate deadline* $\overline{D}_i$, that is the interval of time between the activation of the pipeline and the absolute deadline of the task. Hence, using the notation introduced so far, the absolute deadline of the ℓ^{th} instance of τ_i, is

$$d_i^\ell = \ ^\ell + \overline{D}_i. \tag{4}$$

We enforce the precedence relationship between tasks by the slicing technique [9]: for each task we set the *activation offset* ϕ_i, relative to the activation of the pipeline $^\ell$, equal to the intermediate deadline of the preceding one:

$$\phi_1 = \ , \quad \phi_i = \overline{D}_{i-1} \quad i = 2, \ldots, n \tag{5}$$

Clearly, the task absolute activation is

$$a_i^\ell = \ ^\ell + \phi_i. \tag{6}$$

Moreover, we define the task *relative deadline* D_i as

$$D_i \stackrel{\text{def}}{=} \overline{D}_i - \phi_i.$$

The relationship between activation offsets and relative

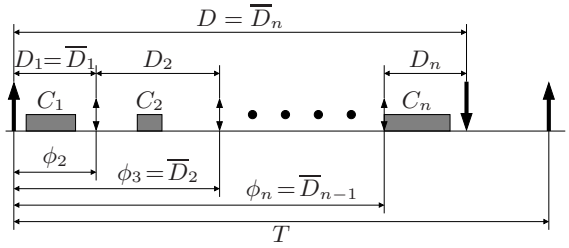

Figure 2: Notation for tasks.

deadlines is depicted in Figure 2. Clearly,

$$\sum_{i=1}^{n} D_i = D \tag{7}$$

The values of T, $^\ell, D, C_i, \overline{D}_i, D_i, \phi_i$ are all real numbers. Finally, we use the notation $(\cdot)_0 \stackrel{\text{def}}{=} \max\{\ , \cdot\}$.

IV. Periodic demand bound function

First, we recall the concept of demand bound function for a pipeline that is strictly periodic (i.e. $\forall \ell$, $^\ell = \ell T$). Then, in the next section we extend the demand bound function to the sporadic case.

The computational requirement of the subset $\mathcal{T}_k$ of tasks allocated on node k is modeled by its *demand bound function* (dbf).

Definition 1: The *demand function* on node k, denoted by $\mathsf{df}_k(t_0, t_1)$, is the total computation time of all the instances of the tasks in $\mathcal{T}_k$, having activation time and deadline within $[t_0, t_1]$.

For periodic pipeline, the demand function can be computed as follows [10]:

$$\mathsf{df}_k(t_0, t_1) \stackrel{\text{def}}{=} \sum_{\tau_i \in \mathcal{T}_k} \left(\left\lfloor \frac{t_1 - \overline{D}_i}{T} \right\rfloor - \left\lceil \frac{t_0 - \phi_i}{T} \right\rceil + 1 \right)_0 C_i \tag{8}$$

As suggested by Rahni et al. [8], the overall *demand bound function* of $\mathcal{T}_k$ in an interval of length t, is defined as:

$$\mathsf{dbf}_k(t) \stackrel{\text{def}}{=} \max_{t_0} \mathsf{df}_k(t_0, t_0 + t) \tag{9}$$

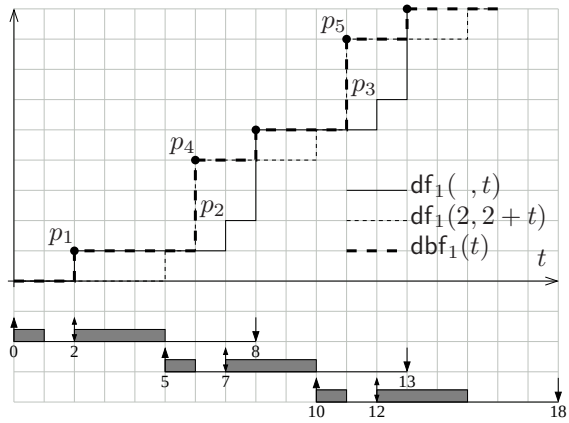

Figure 3: Example of demand bound function.

A necessary and sufficient schedulability test for non-concrete pipelines (i.e. periodic pipelines with free initial offset), scheduled by EDF consists in checking that the demand never exceeds the length of the interval on every processor

$$\forall k = 1, \ldots, p \ \ \forall t > \qquad \sum_{\mathcal{T}} \mathsf{dbf}_k(\mathcal{T}, t) \leq t \qquad (10)$$

where the sum is made over all the pipelines in the system, and $\mathsf{dbf}_k(\mathcal{T}, t)$ denotes the demand bound function of $\mathcal{T}$ on node k. In this case, first the dbf is computed for each pipeline and for each node (applying the max operator), and then we sum all the dbf together to compute the overall computational requirementdbf on node k.

In Figure 3 we illustrate the definitions introduced in this section by an example. Consider a pipeline whose parameters are: period $T = 5$, end-to-end deadline $D = 8$, task deadlines $D_1 = 2$ and $D_2 = 6$, computation time $C_1 = 1$ and $C_2 = 3$. Both tasks are assigned to a single node. In the lower part of Figure 3, we show three consecutive instances of the pipeline on three different lines. In the upper part, we show the values of 3 functions: the demand in $[\ ,t]$; the demand in $[2, 2+t]$; and the demand bound function. We represent the points where the dbf has a step by a thick dot. The steps are tightly related to task deadlines. For example in the figure, the points p_1, p_2, p_3 depend on the deadlines of task τ_1, while the points p_4, p_5 depend on the deadlines of τ_2.

To compute the dbf of a periodic pipeline, it is sufficient to consider the value of the demand functions obtained on the intervals that start with the activation of a task, as shown in [8]. Also, the dbf has a periodic pattern: its value for a generic large interval t can be computed as $\mathsf{dbf}(t') + jC$, where $C = \sum_{\tau_i \in \mathcal{T}_k} C_i$, $j \geq$ and $t' = t - jT$ (see Section 4.1 in [8]).

V. Demand bound function of sporadic pipelines

Unfortunately, for sporadic pipelines, the worst case does not occur with periodic activations. Consider the following pipeline with 3 tasks on 2 processors. The

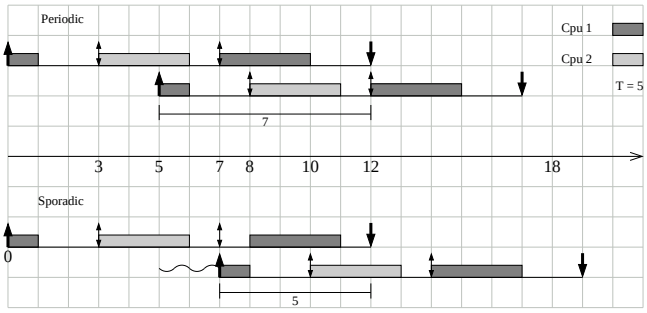

Figure 4: Example of sporadic pipeline.

pipeline has period $T = 5$ and end-to-end deadline $D = 12$. The task parameters are reported in Table I.

Task	C_i	proc.	D_i
τ_1	1	0	3
τ_2	3	1	4
τ_3	3	0	5

Table I: Parameters for the example

In Figure 4, we show two possible activation patterns. The first one corresponds to a periodic activation ($^1 = T$): in this case, it is easy to see that the maximum demand on processor in any interval of length 5 is at most 3 units of computation.

In the second activation pattern, the activation of the second instance is delayed by 2 units of time ($^1 = T+2$). As a consequence, the demand in interval $[7, 12]$ becomes 4 units of time, because one extra instance of τ_1 enters the interval. Thus, delaying an instance can increase the demand.

Hence, the analysis based on the classic periodic demand bound function is not applicable if pipelines are sporadic. One of the contributions of this paper is to extend the demand bound function to sporadic pipelines.

A job τ_i^ℓ in $\mathcal{T}_k$, runs inside interval $[t_0, t_1]$ if its absolute deadline d_i^ℓ is not later than t_1

$$t_1 \geq d_i^\ell = \overline{D}_i + \quad ^\ell \qquad (11)$$

and its activation is not earlier than t_0

$$t_0 \leq a_i^\ell = \phi_i + \quad ^\ell \qquad (12)$$

By introducing the function

$$\mathsf{step}(x) = \begin{cases} \quad \text{if } x < \\ 1 \quad \text{if } x \geq \end{cases} \qquad (13)$$

we can define the following binary-valued function

$$\mathsf{jobIn}_i^\ell(t_0, t_1) \stackrel{\text{def}}{=} \mathsf{step}(t_1 - \overline{D}_i - \quad ^\ell) \cdot \mathsf{step}(\phi_i + \quad ^\ell - t_0) \qquad (14)$$

that returns 1 if the job τ_i^ℓ has both activation and deadline in $[t_0, t_1]$, and it returns otherwise.

Hence, the demand of all the tasks belonging to the pipeline $\mathcal{T}_k$ can be expressed as:

$$\mathsf{df}_k(t_0, t_1) \stackrel{\text{def}}{=} \max_{\Phi \in \Gamma} \sum_{\ell=-k_0}^{k_1} \sum_{\tau_i \in \mathcal{T}_k} \mathsf{jobIn}_i^\ell(t_0, t_1) C_i \qquad (15)$$

where k_0 and k_1 are indexes of pipeline instances, defined in Eq. (3), that may have an effect on the demand in $[t_0, t_1]$.

The sum on all the pipeline instances ℓ can be split in three parts: the first part is the sum over the indexes corresponding to the *past instances* (from $-k_0$ to -1); the second part is the *current instance* (with $\ell =$), and the third part is the sum over the *future instances* (from 1 to k_1). Hence Equation (15) becomes

$$\begin{aligned}\mathsf{df}_k(t_0,t_1) = &\sum_{\tau_i} \mathsf{jobIn}_i^0(t_0,t_1)C_i \\ &+ \max_{(\Phi^{-k_0},\ldots,\Phi^{-1})\in\Gamma^-} \sum_{\ell=-k_0}^{-1} \sum_{\tau_i} \mathsf{jobIn}_i^\ell(t_0,t_1)C_i \\ &+ \max_{(\Phi^{1},\ldots,\Phi^{k_1})\in\Gamma^+} \sum_{\ell=1}^{k_1} \sum_{\tau_i} \mathsf{jobIn}_i^\ell(t_0,t_1)C_i\end{aligned} \quad (16)$$

where $^-$ and $^+$ are the sets of the possible activation patterns of the past and the future instances respectively. Although Eq. (16) is apparently more complex than Eq. (15), it is computationally more efficient, because it has the advantage of decoupling the dependency on past and future instances.

Finally, as for the periodic dbf, the sporadic dbf is the maximum among all the sporadic demand functions computed on intervals with the same length:

$$\mathsf{dbf}_k(t) \stackrel{\text{def}}{=} \max_{t_0} \mathsf{df}_k(t_0, t_0 + t) \quad (17)$$

Figure 5 shows that, for the same parameters of Table I, the sporadic dbf computed from Eq. (17) is larger than the periodic dbf (Eq. (9)).

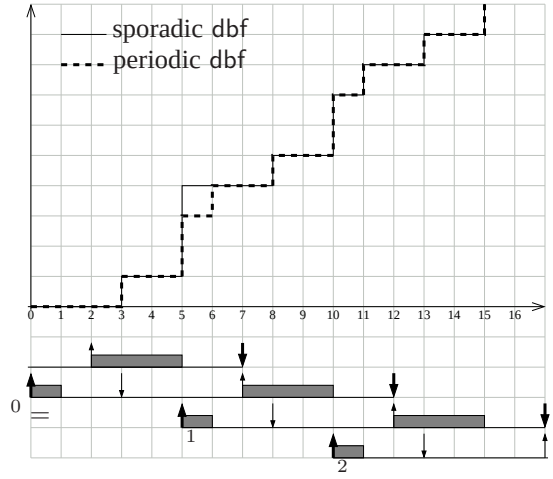

Figure 5: An example of sporadic dbf.

Equation (17) is a nice and compact expression of the dbf. It is however unclear how such a dbf should be practically computed.

We follow a strategy similar to the one used for computing the dbf of periodic pipeline. The strategy consists in the algorithm reported in Figure 6. First (at line 2), we compute the list intervalSet of all the *significant* intervals

```
1: intervalSet ← ∅                          ▷ initialize the set of intervals
2: STOREINTERVALS                           ▷ store intervals in intervalSet
3: sort intervalSet by increasing t1 − t0
4: lastDBFval ←
5: for each [t0, t1] ∈ intervalSet do       ▷ loop on all intervals
6:       ^−, ^+ ← ∅ ▷ init sets of past and future patterns
7:     SPANPATTERNS(t0, t1) ▷ store patterns in ^−, ^+
8:     curDBFval ← df_k(t1, t0)             ▷ Eq. (16)
9:     if curDBFval > lastDBFval then       ▷ Eq. (17)
10:        store the point (t1 − t0, curDBFval)
11:    else
12:        do nothing (dominated by previous point)
13:    end if
14: end for
```

Figure 6: Algorithm for computing the dbf.

$[t_0, t_1]$, i.e. the intervals such that $\forall \varepsilon >$ both the demands $\mathsf{df}_k(t_0 + \varepsilon, t_1)$ and $\mathsf{df}_k(t_0, t_1 - \varepsilon)$ are **strictly less** than $\mathsf{df}_k(t_0, t_1)$. In Section V-A we describe the procedure STOREINTERVALS for performing this step. After sorting the intervals $[t_0, t_1]$ in intervalSet by increasing $t_1 - t_0$ (at line 3), we search for the activation pattern $\bar{}$ that maximizes the demand in $[t_0, t_1]$. In Section V-B we describe the procedure SPANPATTERNS that stores all possible activation patterns of future instances in $^+$ and those ones related to past instances in $^-$.

A. Enumerating the intervals

The first stage requires to enumerate all the intervals $[t_0, t_1]$. The pseudocode of this stage is reported in Figure 7. First, we claim that t_0 must coincide with the activation of some job. In fact, if this does not happen then we could increase t_0 achieving a shorter interval with the same demand. Hence we set t_0 equal to the activation of the job τ_i^0, i.e. t_0 spans on $\{\phi_i : \tau_i \in \mathcal{T}_k\}$ (see line 4 of the algorithm). Notice that, without loss of generality, we label by the pipeline instance which this job belongs to.

Regarding the possible values of t_1, it is sufficient to test only the absolute deadlines d_j^h. In fact if $t_1 = d_j^h$ for some task $\tau_j \in \mathcal{T}_k$ and some pipeline instance h, then a reduction of t_1 by an arbitrary small amount ε will decrease the demand df by at least C_j. However, the main difficulty here is that the absolute activations are not fixed, hence we do not know where the deadlines are until we fix the sporadic activation pattern $\bar{}$.

First, we list the values of t_1 associated with the absolute deadlines of the instance (see lines 5–9). Then we invoke the recursive procedures FUTUREDEADLINE and PASTDEADLINE that list the absolute deadlines of the future and past instances, respectively.

These two procedures explore the possible activation patterns $\bar{}$ such that task activations are aligned with t_0. Specifically, the call FUTUREDEADLINE$(t_0, \ell, \ ^{\ell-1})$ explores all the activation patterns of instances starting from the ℓ-th one, assuming that the absolute activation of the $(\ell - 1)$-th instance is $^{\ell-1}$ and the start of the

```
1: procedure STOREINTERVALS
2:     intervalSet ← ∅                                  ▷ initialize
3:     for each τ_i ∈ T_k do                            ▷ loop on t_0
4:         t_0 ← φ_i
5:         for τ_j ∈ T_k do
6:             if D̄_j > t_0 then
7:                 store [t_0, D̄_j] in intervalSet
8:             end if
9:         end for
10:        FUTUREDEADLINE(t_0, 1,  )
11:        k_0 ← ⌈(D − t_0)/T⌉ − 1
12:        PASTDEADLINE(t_0, −k_0, −2k_0(D + T))
13:    end for
14: end procedure

15: procedure FUTUREDEADLINE(t_0, ℓ,  ^{ℓ−1})
16:    for all  ^ℓ ∈ {  ^{ℓ−1} + T} ∪ {t_0 − φ_i : t_0 − φ_i >
       ^{ℓ−1} + T, τ_i ∈ T_k} do
17:        for each τ_i ∈ T_k do
18:            t_1 =  ^ℓ + D̄_i
19:            if t_1 > t_0 then
20:                store [t_0, t_1] in intervalSet
21:            end if
22:        end for
23:        if ℓ < ⌈(D + 2T)/T⌉ then
24:            FUTUREDEADLINE(t_0, ℓ + 1,  ^ℓ)
25:        end if
26:    end for
27: end procedure

28: procedure PASTDEADLINE(t_0, ℓ,  ^{ℓ−1})
29:    for all  ^ℓ ∈ {ℓT,  ^{ℓ−1} + T} ∪ {t_0 − φ_i :  ^{ℓ−1} +
       T < t_0 − φ_i < ℓT, τ_i ∈ T_k} do
30:        for each τ_i ∈ T_k do
31:            t_1 =  ^ℓ + D̄_i
32:            if t_1 > t_0 then
33:                store [t_0, t_1] in intervalSet
34:            end if
35:        end for
36:        if ℓ < −1 then
37:            PASTDEADLINE(t_0, ℓ + 1,  ^ℓ)
38:        end if
39:    end for
40: end procedure
```

Figure 7: Algorithm for enumerating intervals.

interval is t_0. The key statement of the procedure is at line 16, where $^{\ell}$ is assigned the possible values. The possible choices for $^{\ell}$ are also illustrated in Figure 8. In the situation depicted in the figure, $^{\ell}$ can assume three possible values: $^{\ell-1}+T$ when the ℓ-th instance starts as early as possible, $t_0-\phi_2$ when $^{\ell}+\phi_2$ coincides with t_0, or t_0 when the job τ_1^{ℓ} is activated exactly at t_0. The procedure PASTDEADLINE works similarly.

For each pattern the values of absolute deadlines are recorded as candidate values for t_1.

We conclude the section by showing that, after a transient that is long at most $D+T$, the dbf becomes periodic.

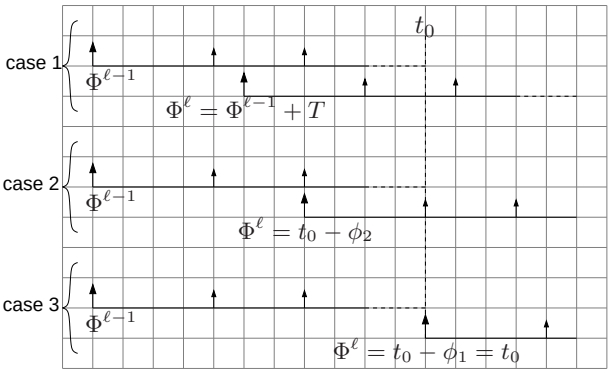

Figure 8: Exploring the absolute activations.

Lemma 1: For large values of t, the $\mathsf{dbf}(t)$ has a periodic pattern. More formally:

$$\forall t > D+T \quad \mathsf{dbf}_k(t+T) = \mathsf{dbf}_k(t) + C^k.$$

where $C^k = \sum_{\tau_i \in \mathcal{T}_k} C_i$.

Proof: Let t_0 and $\underline{\ }$ be the instant and activation pattern that give the value of $\mathsf{dbf}_k(t)$ in Equations (17) and (16) respectively, and let us set $t_1 = t_0 + t$.

We identify with ℓ the first pipeline instance with activation $^{\ell} > t_0$, hence $^{\ell-1} \leq t_0$. Since we are in the worst case and $^{\ell} > t_0$, then

$$\forall h \geq \ell \qquad ^{h} - \ ^{h-1} = T \tag{18}$$

otherwise, we could move all ^{h} with $h \geq \ell$ early without removing any job from the interval. On the contrary, the deadline of a job may enter the interval, and the worst-case activation pattern cannot be $\overline{\ }$ anymore, causing a contradiction.

From (18) and the definition of ℓ, we notice that the instance ℓ of the pipeline ends earlier than t_1. Clearly this is also true for all instances before ℓ. Formally

$$^{\ell-1} \leq t_0 \quad \Rightarrow \quad ^{\ell} \leq t_0 + T$$
$$^{\ell} + D \leq t_0 + T + D < t_1.$$

From (18), it follows that any interval of length T starting after $^{\ell} + D$ contains exactly one activation and one deadline of each task. Hence the demand generated in the interval $[t_0, t_1 + T]$ increases by one job for all tasks in $\mathcal{T}_k$, i.e. C^k.

Suppose by absurd that $\mathsf{dbf}_k(t+T) > \mathsf{dbf}_k(t) + C^k$. Then, it exists an interval $[t'_0, t'_0+t+T]$ with demand larger than $\mathsf{dbf}_k(t) + C^k$. Let $\underline{\ }'$ be its activation pattern, and let us call ℓ' the first instance with $^{\ell'} > t'_0$. Following the same reasoning as above, the demand in $[t'_0, t'_0 + t]$ decreases by C^k. However, this is absurd because we obtain a new interval with the same length t but with demand higher than in $[t_0, t_0 + t]$. ■

Since, thanks to the lemma, the transient part of the dbf lasts for no longer than $D+T$ and the periodic part is long T, it is possible to compute the dbf only for lengths of intervals less or equal to than $D + 2T$.

Now we present an algorithm for computing the activation patterns that determines the maximum demand in a given interval $[t_0, t_1]$.

B. Algorithm for enumerating the activation patterns

In this section we explain the procedure SPANPATTERNS(t_0, t_1) (see line 7 of the algorithm in Figure 6) that checks all possible sporadic activation patterns of past and future instances that may have an impact on the interval $[t_0, t_1]$. Therefore, we are interested only in pipeline instances that may overlap with the interval $[t_0, t_1]$.

In the exploration of the activation patterns we distinguish between future instances (with index $\ell >$) and past instances (with index $\ell <$). The guideline for the exploration of absolute activations of future instances is to align some task activation $a_i^\ell =$ $^\ell + \phi_i$ with t_0. This is possible by setting

$$^\ell = t_0 - \phi_i. \tag{19}$$

However, this is a valid absolute activation only if it respects the constraints of being a sporadic pipeline with minimum interarrival T, that is

$$^\ell \geq \ ^{\ell-1} + T. \tag{20}$$

This condition introduces a recurrent dependency between all the values ^{0}, ^{1}, $^2, \ldots,$ ^{k_1}. The procedure COMPUTEFUTURE for testing future instances is reported in Figure 7.

The same rationale is applied to past instances (the ones with index $\ell <$). In this case however, we aim at finding the absolute activation $^\ell$ such that some absolute deadline is aligned with t_1. The full algorithm that explores the activation patterns is reported in Figure 9.

In the example of Figure 5, if we assume $t_0 =$ then ^{1} should be tested with the values of $5 (= T)$. Instead, if $t_0 = \phi_3 = 7$ then ^{1} is checked both when it is 5 and when it is $t_0 - \phi_1 = 7$, meaning that we align the activation of the instance 1 with the offset $\phi_3 = t_0 = 7$.

C. Complexity analysis

We start by analyzing the complexity of procedure STOREINTERVALS. The outer loop (line 3) is executed n_k times. After adding the intervals for instance 0, procedures FUTUREDEADLINE and PASTDEADLINE are invoked.

Procedure FUTUREDEADLINE explores a number of instances at most equal to $k_2 = \left\lceil \frac{D+2T}{T} \right\rceil - 1$. Of this, the first $\left\lfloor \frac{t_0}{T} \right\rfloor$ instances may vary their activation time, while for the successive ones, the worst-case corresponds to interarrival times equal to T. The number of possible combinations of activations (line 16) is then $n_k^{\left\lfloor \frac{t_0}{T} \right\rfloor}$. For each combination, $n_k k_2$ deadlines are generated.

Procedure PASTDEADLINE is very similar. The number of instances is k_0 (see Eq. (3)). The maximum number of elements generated for each combination of past activations is $n_k k_0$. Finally, the maximum number of combinations (line 29) is $(n_k + 2)^{k_0}$.

```
1: procedure SPANPATTERNS(t_0, t_1)
2:     k_1 = ⌈(t_1)/T⌉ − 1                                   ▷ see Eq. (3)
3:     COMPUTEFUTURE(1, ( , ..., ))   [k_1]
4:     k_0 = ⌈(D − t_0)/T⌉ − 1                               ▷ see Eq. (3)
5:     COMPUTEPAST(−1, ( , ..., ))   [k_0]
6: end procedure

7: procedure COMPUTEFUTURE(ℓ, ( ^1, ..., ^{k_1}))
8:     if ℓ > k_1 then                                       ▷ the exit condition
9:         store ( ^1, ..., ^{k_1}) in ^+                    ▷ ^+ is global
10:    else
11:        ^0 ←
12:        for all ^ℓ ∈ { ^{ℓ−1} + T} ∪ {t_0 − φ_i : t_0 − φ_i >
       ^{ℓ−1} + T, τ_i ∈ T_k} do
13:            COMPUTEFUTURE(ℓ + 1, ( ^1, ..., ^{k_1}));
14:        end for
15:    end if
16: end procedure

17: procedure COMPUTEPAST(ℓ, ( ^{−k_0}, ..., ^{−1}))
18:    if ℓ < −k_0 then
19:        store ( ^{−k_0}, ..., ^{−1}) in ^−
20:    else
21:        ^0 ←
22:        for all ^ℓ ∈ { ^{ℓ+1} − T} ∪ {t_1 − D̄_i : t_1 − D̄_i <
       ^{ℓ+1} − T, τ_i ∈ T_k} do
23:            COMPUTEPAST(ℓ − 1, ( ^{−k_0}, ..., ^{−1}));
24:        end for
25:    end if
26: end procedure
```

Figure 9: Algorithm for generating $^-$ and $^+$.

Each generated interval must be inserted in a ordered list, an operation that takes logarithmic time in the size of the list. The size of the list at the end of the procedure is:

$$s = k_2 n_k^{\lfloor \frac{t_0}{T} \rfloor + 1} + n_k k_0 (n_k + 2)^{k_0}$$

and the complexity is $O(\sum_{i=1}^{s} \log(i))$.

Notice that, while generating the the values of t_1, it is quite common to obtain many times always the same values. In average, we expect that the final size of the list is much smaller than its upper bound s.

Regarding procedure SPANPATTERNS, we apply a similar reasoning. We address separately future and past instances. Procedure COMPUTEFUTURE builds a tree in which at level 1 sets the value of ^{1}, at level 2 sets the value of ^{2}, and so on. There will be k_1 levels. Each node has at most $n_k + 1$ children. Hence, the number of leafs of such a tree is $(n_k + 1)^{k_1}$. Each leaf corresponds to a different value of ($^1, \ldots,$ ^{k_1}). A similar tree can be built for past instances. Thus the complexity of enumerating all activation patterns is

$$O((n_k + 1)^{k_0} + (n_k + 1)^{k_1}).$$

Finally, the complexity of computing the whole dbf is

$$O\left(\sum_{i=1}^{s}\log(i)+sn_k\left((n_k+1)^{k_0}+(n_k+1)^{k_1}\right)\right).$$

We are aware that the proposed algorithm is very complex, although it runs in acceptable time for very realistic settings as it will be shown in the experiments. Most of the complexity lies in the sporadic nature of the pipeline that requires to check all possible scenarios. In this paper, we focused on the exact analysis regardless of its complexity. We leave to future investigations the development of simplified algorithms as well as some approximation schemes.

VI. Experimental Results

A. Computation of the dbf

In order to test the run-time of our algorithm, we conducted several runs on synthetic chains of tasks. In each experiment we varied the number of tasks, CPU and deadline/period ratio, measuring the time needed to compute the sporadic dbf according to the procedures described in Section V.

The simulation platform is based on a Linux distribution (Ubuntu) with kernel 2.6.24-16-generic. The HW is a notebook with an Intel T2400 (a Centrino duo at 1.8GHz) and 2GByte of RAM. To get the time for each test, we use the Linux call

```
int clock_gettime(clockid_t clock_id,
                  struct timespec *tp)
```

with value `CLOCK_THREAD_CPUTIME_ID`, in order to get the CPU-time given to our thread. The algorithm if Figures 6, 7 and 9 is implemented in a single thread, without any code optimization (some possible improvements are discussed in Section VI-B).

The numerical results are reported in Table II. In Figure 10 we plot the case with 4 CPUs, while in Figure 11 the case with 8. The pipeline has a number of tasks $n = \{2\ ,4\ ,\ldots,12\ \}$, which are uniformly distributed over the available CPUs, i.e. if we have 4 tasks and 4 CPUs then there are 1 tasks on each CPU. We vary the value of $\frac{D}{T}$ from 5 to 2 . Computation time C_i, relative deadlines D_i and mapping x_i are randomly generated.

In spite of the exponential nature of the problem, the algorithm is usable for systems with a reasonable size. The most important parameters are the ratio between EE deadline and period D/T, and the number of tasks on each CPU. For ratio larger than 25 the time required to compute the dbf starts to become too large, whereas for values of $\frac{D}{T}$ of 1 or less, it is possible to put 20 tasks on each CPU and still receive an answer in a reasonable time.

B. Improving the computation of the sporadic dbf

Despite the fact that the algorithm computes the dbf in a reasonable time for small and medium dimension of the problem, the execution time can be further reduced with some code optimization that we highlight here.

CPU	n	D/T	min	average	max
4	20	5	0.0038	0.0089	0.0197
4	40	5	0.1497	0.2002	0.2950
4	60	5	0.6551	1.0892	1.4681
4	100	5	9.8498	16.5764	22.8148
4	20	10	0.0148	0.0390	0.0731
4	40	10	3.1423	6.7326	12.0010
4	60	10	47.0424	106.5302	138.9140
4	20	15	0.0907	0.18023	0.3284
4	40	15	7.8494	20.0923	36.6787
4	60	15	264.5299	399.1137	597.4694
8	20	15	0.0074	0.0166	0.0348
8	60	15	2.7329	7.6004	12.6663
8	80	15	24.3439	45.6743	68.7062
8	20	20	0.01183	0.03168	0.05448
8	60	20	16.9809	27.7639	41.7258
8	80	20	127.4887	233.0104	330.9758

Table II: Execution time for computing the dbf on 4 and 8 processors (times in seconds).

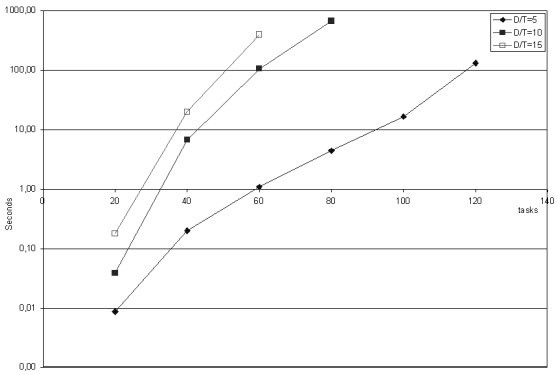

Figure 10: Execution times on 4 CPUs.

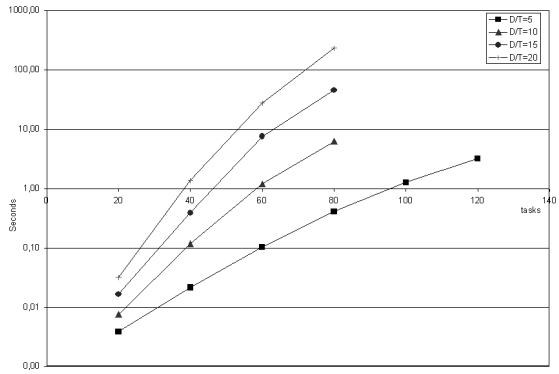

Figure 11: Execution times on 8 CPUs.

CPU	n	D/T	min	average	max
4	20	5	$0,0148$	$0,0177$	$0,0227$
4	40	5	$0,1237$	$0,1546$	$0,2048$
4	60	5	$0,5075$	$0,6577$	$0,7730$
4	100	5	$5,7577$	$8,0756$	$11,5889$
4	20	10	$0,0323$	$0,0397$	$0,0514$
4	40	10	$0,3420$	$0,5943$	$0,8360$
4	60	10	$3,7987$	$5,5338$	$7,6891$
4	100	10	$105,1929$	$142,2384$	$192,2207$
4	20	15	$0,0431$	$0,0654$	$0,0895$
4	40	15	$1,5671$	$2,3917$	$4,0547$
4	60	15	$22,4993$	$28,9635$	$36,5755$
4	80	15	$109,9120$	$158,1644$	$203,6692$

Table III: Execution time for computing the dbf on 4 processors (times in seconds) with a small optimization.

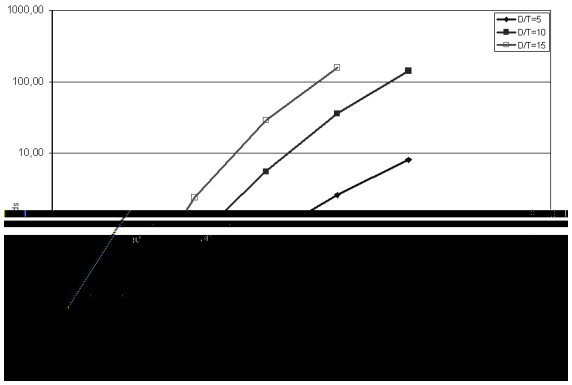

Figure 12: Fast execution times on 4 CPUs.

Parallelism: A simple technique is based on exploiting the inherent parallelism exposed by the algorithm. We highlight two places where the introduction of parallel threads can be fruitfully used.

1) Each iteration of the for loop of line 5 in Figure 6 is independent of each other, hence parallel threads could take fully advantage of this.
2) Each iteration of the for loop at line 3 in Figure 7 is independent of each other. More than one thread can be used to compute independently the set of intervals that derives for each $t_0 = \phi_i$.
3) To reduce wait-states, it is possible to start computing the demand of intervals before procedure STOREINTERVALS ends. Clearly there are some shared structure, like the list of intervals and the dbf.

Branch and bound: A second technique that we suggest is based on branch and bound. In the procedures COMPUTEFUTURE and COMPUTEPAST, while we are computing the demand in the interval $[t_0, t_1]$, we have to explore all possible sporadic activation patterns to find the one that maximizes Eq. (16). This operation is the exploration of a tree, where the depth corresponds to the the index ℓ of an instance of the pipeline, and at each node we assign a value to $^{\ell}$. Hence, if we are able to upper bound the contribution of a sub-tree, we can prune it without continuing the exploration.

For example, the contribution of an instance in an interval of length t could be upper bounded by the demand bound function of only one instance considered in isolation, and the contribution of all the remaining instances of the subtree can be set equal to the sum of each individual contribution.

We add branch and bound to our algorithm, reducing the required time. We report the numerical results in Table III and plot them in Figure 12.

C. Comparison against holistic analysis

As we discussed throughout all the paper, our proposed approach provides substantial benefit for software engineers by allowing the composition of pipelines that are developed and guaranteed independently of each other. However we incur in a loss ID schedulability compared with classic holistic analysis [4], [7]. In this experiment we evaluate this loss.

We assumed to have 2 or 4 computational nodes. On these nodes we distributed 6 pipelines, each one with a number of tasks between 6 and 12 and a period in $\{1 \quad , 2 \quad , \ldots , 9 \quad , 1 \quad \}$. For each setting we generated 200 sets of distributed pipelines. We plot the percentage of accepted sets as a function of the deadline-period ratio D/T of three schedulability tests: MDO proposed by Pellizzoni and Lipari [7], WCDO proposed by Palencia and González Harbor [4], and our proposed test based on the computation of the sporadic demand bound function. In Figure 13 we report the experiments, when the number

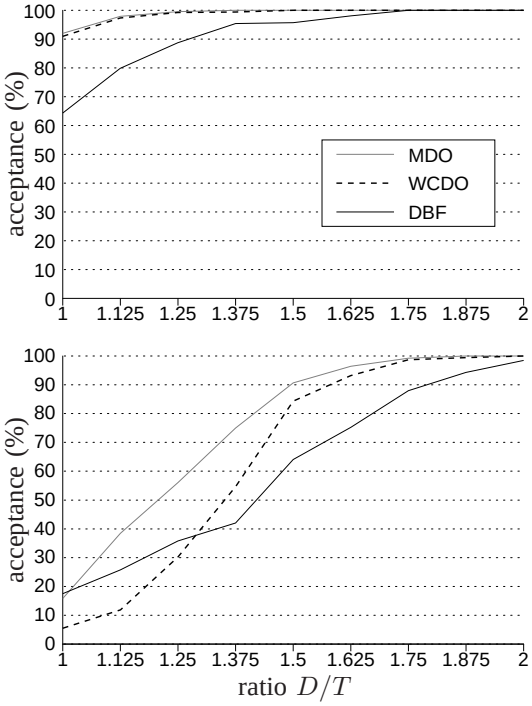

Figure 13: Performance of schedulability tests (2 CPUs).

of CPUs is 2. In the upper part of the figure we set the total utilization of each node equal to $.4$, while in the bottom part we set it equal to $.8$ (condition of heavy node utilization).

In Figure 13 we report the experiments, when the number of CPUs is 4. In the upper part of the figure we set the total utilization of each node equal to $.5$, while in the bottom part we set it equal to $.8$ (condition of heavy node utilization).

As expected the test based on the sporadic dbf is more pessimistic than the others. The source of pessimism comes from the assumption of releasing the tasks at the deadline of the preceding one, rather than at the completion. Even considering the pessimism of the analysis we still believe that the gain in terms of compositionality does dominate the incurred loss of schedulability.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we addressed the problem of analyzing the schedulability of sporadic pipelines on a distributed system

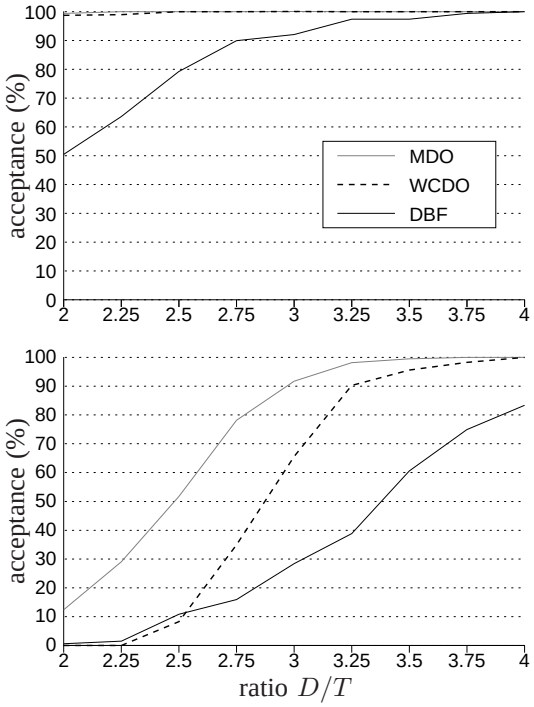

Figure 14: Performance of schedulability tests (4 CPUs).

scheduled by EDF, and how to support our methodology at run time. We proposed an algorithm to compute the *sporadic* dbf offline on each node.

As a future work, we plan to extend the methodology to task graphs. Also, we would like to study the effect of combining different scheduling strategies on different nodes, and the effects of network scheduling. Finally, we plan to study the possibility to schedule a pipeline in a time partition on each node, and study the effect of misbehaviors on the pipeline schedule, such as tasks exceeding their WCET, or events that do not respect the minimum interarrival time.

REFERENCES

[1] K. Richter, R. Racu, and R. Ernst, "Scheduling analysis integration for heterogeneous multiprocessor soc," in *Proceedings of the* 25^{th} *Real-Time Systems Symposium*, Cancun, Mexico, Dec. 2003, pp. 236–245.

[2] W. Zheng, Q. Zhu, M. Di Natale, and A. Sangiovanni-Vincentelli, "Definition of task allocation and priority assignment in hard real-time distributed systems," in *Proceedings of the* 28^{th} *IEEE Real-Time Systems Symposium*, Tucson, AZ, Dec. 2007, pp. 161–170.

[3] K. W. Tindell, A. Burns, and A. Wellings, "An extendible approach for analysing fixed priority hard real-time tasks," *Journal of Real Time Systems*, vol. 6, no. 2, pp. 133–152, Mar. 1994.

[4] J. C. Palencia and M. González Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 1998, pp. 26–37.

[5] M. Spuri, "Holistic analysis for deadline scheduled real-