Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
JOURNAL OF COMPUTER AND SYSTEM SCIENCES
Journal of Computer and System Sciences 73 (2007) 186–206
www.elsevier.com/locate/jcss

Holistic analysis of asynchronous real-time transactions with earliest deadline scheduling

Rodolfo Pellizzoni *, Giuseppe Lipari

Scuola Superiore Sant'Anna, Pisa, Italy

Received 8 June 2005; received in revised form 31 January 2006

Available online 13 November 2006

Abstract

In distributed real-time systems, an application is often modeled as a set of real-time *transactions*, where each transaction is a chain of precedence-constrained tasks. Each task is statically allocated to a processor, and tasks allocated on the same processor are handled by a single-processor scheduling algorithm. Precedence constraints among tasks of the same transaction are modeled by properly assigning scheduling parameters as offsets, jitters and intermediate deadlines.

In this paper we address the problem of schedulability analysis of distributed real-time transactions under the earliest deadline first scheduling algorithm. We propose a novel methodology that reduces the pessimism introduced by previous methods by explicitly taking into account the offsets of the tasks. Moreover, we extend the analysis to account for blocking time due to shared resources. In particular, we propose two kinds of schedulability tests, CDO-TO and MDO-TO, and show, with an extensive set of simulations, that they provides improved schedulability conditions with respect to classical algorithms. Finally, we apply the methodology to an important class of systems: heterogeneous multiprocessor systems, with a general purpose processor and one or more coprocessors (DSPs).
© 2006 Published by Elsevier Inc.

Keywords: Real-time systems; Scheduling algorithms; Heterogeneous multiprocessors

1. Introduction

Distributed real-time systems are widely used in many industrial areas. Notable examples can be found in factory automation, automotive systems, flight control, etc. Usually, a distributed real-time system is modeled as a set of real-time periodic *transactions*. Each transaction is a sequence of tasks that are periodically activated, where each task is statically allocated to one computational node. Tasks must execute in order, i.e. a task can start executing only after the preceding task in the transaction has completed. Each transaction is assigned an *end-to-end* deadline: the time between the activation of the transaction and the finishing time of the last task in the sequence must not exceed the transaction deadline. A set of transactions is said to be *schedulable* if all transactions complete before their deadlines. A *schedulability tests* is an algorithm that given the parameters of all transactions returns *true* if the set is schedulable.

* Corresponding author.
E-mail address: rodolfo@sssup.it (R. Pellizzoni).

0022-0000/$ – see front matter © 2006 Published by Elsevier Inc.
doi:10.1016/j.jcss.2006.04.002

If a *necessary and sufficient* test returns *false*, the system is not schedulable (i.e. a transactions can miss its deadline at some point). If a *sufficient* test returns *false*, the set may or may not be schedulable.

Schedulability analysis of distributed real-time systems is an important problem that has been studied for a long time by the research community. Solutions have been proposed both for fixed priority scheduling [1,2] and for earliest deadline first (EDF) [3,4]. Usually, the precedence constraints in the transaction are modeled by assigning each task an *initial offset* and a *maximum jitter* [1]. The *initial offset* ϕ_{ij} of a periodic task is the instant of the first activation of the task. Every successive activation is a multiple of the task period plus the initial offset. However, even if a periodic task is activated as some time t its *release* time (i.e. the time from which it can start executing) may be delayed due to the precedence constraint. In fact, a task belonging to a transaction may start only after it has been activated and the preceding task in the transaction has completed execution. Hence *maximum jitter* is the maximum time interval it can occur from the task activation until the completion time of the preceding task in the transaction.

By introducing offsets and jitters to model precedence constraints, the schedulability problem for a distributed system with P computational nodes is reduced to P single-node schedulability problems. On each node, we need to test the schedulability of a set of independent periodic tasks. In this problem, a very important role is played by task offsets. A set of periodic tasks is said to be *synchronous* if the first activation of every task is at the same time. Conversely, a set of tasks is said to be *asynchronous* if each task has an *initial offset*. In the case of transactions, on each computational node we must test the schedulability of an asynchronous task set. Unfortunately, any necessary and sufficient feasibility test for asynchronous tasks requires an exponential time to run [5]. Therefore, one interesting problem is to find an efficient yet tight schedulability condition for asynchronous task sets.

In a previous paper [6,7], we presented a new sufficient schedulability test for asynchronous task sets that we showed to be much tighter than previous existing tests. In this paper, we apply such method to the problem of EDF-schedulability analysis of distributed transactions. The goal is to obtain a less pessimistic analysis without losing too much on efficiency. This extension is not trivial, as we will show in Section 3. Previous works on such problem has been based on the *holistic analysis*, first proposed by Tindell and Clark [1] and later improved by Palencia and Gonzàlez [2,4]. In such analysis, the worst-case response time of each task is used to set the offset and the jitter of the successive task in the same transaction. Then, the computation of worst-case response times is iterated until a stable solution is found. If response times are bounded, the holistic method is guaranteed to converge to a solution. Unfortunately, as we will see in Section 3, by applying our method to the holistic analysis in a straightforward way, the resulting algorithm does not converge. Therefore, in this paper we propose a modification of the holistic analysis eliminating the jitter parameter. We propose two new algorithms, CDO and MDO, and prove their convergence.

To summarize, the main contribution of this paper are the following:

- First, in Section 3 we introduce a new methodology for worst-case response time analysis of distributed transactions that takes into account task and transaction offsets.
- Second, in Section 4 we present CDO and MDO, two new iterative algorithms for holistic analysis that make full use of our new methodology.
- In Section 5, we show that both algorithms are effective with an extensive set of simulations with synthetic transactions.
- In Section 6 we show how our algorithms can be augmented to account for resources shared among tasks.
- Finally, in Section 7 we also show how to apply our methodology to a specific yet important real case: a heterogeneous multiprocessor system with one general purpose processor and one or more dedicated digital signal processors (DSPs).

2. System model and notation

In this section, we introduce the notation and the model used throughout the paper. We consider the feasibility problem of a transaction set consisting of M real-time periodic transactions $\mathcal{T}_1, \ldots, \mathcal{T}_M$ and P processors $p_1, \ldots, p_P$.

2.1. Transactions

A transaction $\mathcal{T}_i$ is a sequence of N_i tasks, $\tau_{i1}, \ldots, \tau_{iN_i}$ with precedence constraints: a task τ_{ij}, $j \geqslant 2$, can start executing only after the preceding task $\tau_{i,j-1}$ has completed execution. We assume that all transaction and task

parameters are expressed by natural numbers. Time is divided into slots, starting from 0: $t \in \mathcal{N}$. In what follows, we will refer to a *busy period* $[t_1, t_2)$ for a processor p_i as an interval of time in which p_i is always busy.

Each transaction $\mathcal{T}_i$ is characterized by period T_i and offset ϕ_i, such that the kth instance of each transaction is activated at time $a_i^k = \phi_i + (k-1)T_i$. Each transaction is further characterized by an end-to-end relative deadline D_i, that is the maximum time between the activation of the transaction and the finishing time of the last task.

2.2. Tasks

Each task τ_{ij} is characterized by its assigned processor $p_{ij} \in p_1, \dots, p_P$, a worst-case computation time C_{ij}, an offset ϕ_{ij} and an activation delay δ_{ij}. The task period is equal to the period of the transaction the task belongs to. The offset ϕ_{ij} is the time at which the task is activated by the periodic timer relative to the transaction activation time: therefore, each task's job τ_{ij}^k has an activation time $a_{ij}^k = a_i^k + \phi_{ij}$. However, even if job τ_{ij}^k has been activated at time a_{ij}^k it cannot start executing until after a certain delay δ_{ij} from the completion time of the preceding job $\tau_{i,j-1}^k$. If we denote with R_{ij}^k the response time of job τ_{ij}^k relative to the transaction activation a_i^k, we have the following relationship for the release time s_{ij}^k of τ_{ij}^k:

$$\begin{aligned} s_{i1}^k &= a_i^k + \delta_{i1}, \\ s_{ij}^k &= a_i^k + R_{i,j-1}^k + \delta_{ij} \quad \forall 1 < j \leqslant N_i. \end{aligned} \tag{1}$$

Since a job must be activated before being released, for all jobs τ_{ij}^k it must clearly hold $a_{ij}^k \leqslant s_{ij}^k$. In practice, this is easily verified if the offset ϕ_{ij} is set to be equal to the minimum possible release time for jobs of τ_{ij}.

Palencia and Gonzàlez [2] showed that this model is useful for systems where tasks suspend themselves and for distributed or multiprocessor transactions; values δ_{ij} are particularly useful as they can be used to model both suspension times and transmission delays.

p_{ij} is the processor to which the task is statically bound; note that tasks pertaining to the same transaction may be executed on different processors. Also note that since no task migration is allowed, we do not make any assumption about the processors; in particular, they do not need to share a common memory architecture.

2.3. Critical sections

Tasks that are allocated on the same processor (pertaining to the same or to different transactions) can access critical sections of code on *shared resources*. The usage of critical sections ensures that all resources are accessed in exclusive mode. To simplify our presentation, only single-unit resources are considered, although there are ways to consider the case of multi-unit resources [8]. We assume that no resource is shared among tasks executed on different processors. We consider a set $\mathcal{R}$ of R shared resources $\rho_1, \dots, \rho_R$. Each task τ_{ij} may access η_{ij} different critical sections. Each critical section ξ_{ijk} is described by a 3-ple $(\rho_{ijk}, \psi_{ijk}, C_{ijk})$, where:

1. $\rho_{ijk} \in \mathcal{R}$ is the resource being accessed;
2. ψ_{ijk} is the earliest time, relative to the activation time of job τ_{ijk}, that the task can enter ξ_{ijk};
3. C_{ijk} is the worst-case computation time of the critical section.

Critical sections can be properly nested in any arbitrary way, as long as their earliest entry time and worst-case computation time is known. Our model, first proposed in [9], is actually slightly different from the classic one used in the literature in that it requires earliest entry time to be known. Note that if earliest entry times are unknown, they can simply be set to zero, although this leads to increased pessimism in the analysis.

2.4. Scheduling algorithm

Our scheduling algorithm of choice is earliest deadline first (EDF). On each processor, EDF schedules the job with the earliest absolute deadline.

Until now, we have defined deadline for transactions. To schedule a transaction-based system under EDF, we must assign a relative deadline to each task. For each task τ_{ij}, we define a global relative deadline D_{ij} as the deadline

relative to the activation time of $\mathcal{T}_i$, i.e. the kth job of task τ_{ij} is then assigned an absolute deadline $d_{ij}^k = a_i^k + D_{ij}$. For the sake of simplicity, we also define a deadline d_{ij} relative to the activation time of $\mathcal{T}_i$: $d_{ij} = D_{ij} - \phi_{ij}$. Obviously $D_{iN_i} = D_i$.

The worst-case global relative response time R_{ij} of task τ_{ij} is the maximum possible response time R_{ij}^k for any job τ_{ij}^k, i.e. it is the greatest difference between the completion time of some job τ_{ij}^k and the activation time a_i^k of its transaction. A real-time transaction system is thus schedulable if and only if for all tasks of all transactions, the worst-case global relative response times are less than or equal to the corresponding global relative deadlines. Based on offsets, we can also define for each task τ_{ij} a worst-case response time r_{ij} relative to the activation time of the task: $r_{ij} = R_{ij} - \phi_{ij}$.

The deadline of the last task in the transaction is often called the *end-to-end deadline* and it is a physical constraint given by the application. The deadline of the intermediate tasks are not proper constraints but *free parameters* used by the scheduling algorithm and by the schedulability analysis. The designer can freely assign and modify such deadlines in order to make schedulability easier or according to some global optimality function.[1]

Unfortunately, the problem of optimally assigning these intermediate deadlines can be proven to be NP-Hard. The proof is a trivial reduction from 3-partition along the line of [10]. The best known heuristic so far is a modification of the one used by Palencia and Gonzàlez [4], and consists of assigning each task a deadline proportional to its computation time, subtracting from the end-to-end deadline the delay times:

$$D_{ij} = \left(D_i - \sum_{1 \leqslant k \leqslant N_i} \delta_{ik}\right) \frac{\sum_{1 \leqslant k \leqslant j} C_{ik}}{\sum_{1 \leqslant k \leqslant N_i} C_{ik}} + \sum_{1 \leqslant k \leqslant j} \delta_{ik}. \tag{2}$$

Note that since the deadline heuristic is not optimal, there are transaction sets that are schedulable for some deadline assignments but not for the one given by Eq. (2). In this case, it could make sense to explore the space of the different assignments trying to find one that guarantees schedulability. However, devising a good search algorithm over the space of deadline assignments is difficult in the general case, since it is not easy to understand how a change in the deadline of a task affects the response times of tasks of different transactions. Therefore, in the general case we will only consider the heuristic provided by Eq. (2), while in Section 7.3 we will propose more sophisticated heuristics for a specialized case.

3. Holistic analysis

The system model presented in Section 2 implies a precedence constraint among tasks pertaining to the same transaction. However, precedence constraints are hard to consider in any schedulability analysis. Therefore, task jitters are introduced in order to enforce the precedence constraints.

The release jitter J_{ij} of τ_{ij} is the maximum difference between the activation time a_{ij}^k of a job τ_{ij}^k and its release time s_{ij}^k. The model with offsets and jitters is exemplified in Fig. 1. Once release jitters have been defined, we can enforce the precedence constraints by setting offsets and jitters so that a job τ_{ij}^k suffering maximum jitter is always released at least δ_{ij} time units after the previous job $\tau_{i,j-1}^k$ has finished. Hence, we can apply one of the response time analysis that are introduced in Sections 3.1 and 3.2 in order to compute the worst-case response times and thus

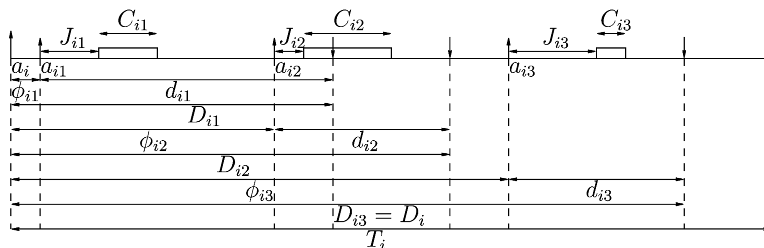

Fig. 1. Model of a transaction $\mathcal{T}_i$ consisting of 3 tasks $\tau_{i1}, \tau_{i2}, \tau_{i3}$, with their parameters.

[1] It is worth noticing that in EDF intermediate deadlines play the same role as the tasks priorities in fixed priority scheduling.

determine if the transaction set is feasible. Unfortunately, offsets and jitters depend on task response times. To solve this problem, we use variations of the holistic analysis first developed by Tindell and Clark [1]. The main idea is to use an iterative method that at each step, starting from some initial offsets and jitters, first computes the response times and then updates offsets and jitters.

We proceed as follows. In the following Sections 3.1 and 3.2 we describe two different response time analyses for transaction sets scheduled by EDF. The first one, called NTO, was introduced by Palencia and Gonzàlez [4], while the second one (TO) is our original contribution to the problem. Both algorithms assume that tasks are independent, i.e. no resources other than processors are shared among tasks. Afterward, in Section 4 we show how these response time analyses can be used in holistic methods. In Section 6 we extend the TO analysis to account for shared resources.

3.1. Response time analysis under EDF

Given a set of periodic, independent tasks to be scheduled by EDF on a single processor, Spuri [3] proposed an algorithm for computing an upper bound on the worst-case response time of a task. His algorithm, however, does not consider task offsets. This means that the analysis proposed by Spuri is still valid even in the case of tasks with offsets, but the results may be pessimistic.

A first approach to the problem of computation of worst-case relative response times for transaction-based systems would be to apply Spuri's method, considering each task to be independent from other tasks of the same transaction. However, this approach is extremely pessimistic. Palencia and Gonzàlez [4] introduced a new method that is much less pessimistic than Spuri's one by taking into consideration the offsets among tasks of the same transaction. We now briefly recall the fundamental ideas behind their method. In this section, we assume for simplicity of exposition that each transaction set is scheduled on a single processor. However, results can be immediately extended to transactions running on P different processors. In fact, when we compute the response time of a task τ_{ij}, tasks executed on a different processor than p_{ij} do not contribute in any way to its response time once the precedence constraints have been enforced using offsets and jitters. We can thus adopt single processor analysis by simply considering a new set of transactions $\mathcal{T}_1', \dots, \mathcal{T}_M'$ composed only of the tasks of $\mathcal{T}_1, \dots, \mathcal{T}_M$, respectively, that run on the same processor as τ_{ij}.

Palencia and Gonzàlez's analysis is based on the following theorem:

Theorem 1. [4] *The worst-case relative response time r_{ab} of a task τ_{ab} can be found in a busy period such that for each transaction $\mathcal{T}_i, i \neq a$, there is a task τ_{ij}, which we call the starting task for $\mathcal{T}_i$, that is released exactly at the beginning of the busy period after having experienced its maximum release jitter.*

Note that releasing task τ_{ab} at the beginning of the busy period may not lead to its worst-case response time. The following theorem limits the complexity of the analysis by limiting the activation times that may lead to the worst-case response time:

Theorem 2. [11] *The worst-case relative response time r_{ab} of a task τ_{ab} corresponds to the response time of some job τ_{ab}^k executed inside a busy period such that either the absolute deadline of τ_{ab}^k corresponds to the absolute deadline of a job of a task of another transaction (executed inside the busy period) or a job of a task of transaction $\mathcal{T}_a$ (possibly τ_{ab} itself) is released at the beginning of the busy period after having experienced maximum jitter.*

Proof. By contradiction, suppose that the absolute deadline of τ_{ab}^k does not correspond to the absolute deadline of a job of a task of another transaction (executed inside the busy period) nor a job of a task of transaction $\mathcal{T}_a$ (possibly τ_{ab} itself) is released at the beginning of the busy period after having experienced maximum jitter. If τ_{ab}^k is not the first task to be released in the busy period we can then increase its response time by moving its activation time a_{ab}^k, and thus the activation time of $\mathcal{T}_a$, to occur earlier until one of the two conditions holds. Since moving a_{ab}^k in such way does not change the set of jobs with higher priority than τ_{ab}^k executed inside the busy period, the finishing time of τ_{ab}^k does not change as well. Therefore, since a_{ab}^k is moved to occur earlier, the relative response time of τ_{ab}^k will increase, which contradicts the hypothesis.

If τ_{ab}^k is the first task to be released in the busy period, we could obtain a worse response time by moving the activation time of all other transactions to occur earlier so that τ_{ab} is released at the beginning of the new busy period after having experienced maximum jitter. □

To compute r_{ab}, we need to compute the worst-case response time for each possible activation time of a job τ_{ab}^k, as explained in Theorem 2, and take the maximum. In particular, we must compute the maximum contribution of every transaction $\mathcal{T}_i$ to the finishing time of τ_{ab}^k. The contribution is the interference imposed by $\mathcal{T}_i$ on τ_{ab}^k, and can be computed as the sum of the execution times of all the jobs of the transaction that are released inside the busy period with absolute deadline less than or equal to that of τ_{ab}^k. Palencia and Gonzàlez showed how to compute the worst-case contribution $W_{ij}(t, D)$ of $\mathcal{T}_i$ in a busy period of length t and greatest absolute deadline D, assuming that τ_{ij} is the starting task. $W_{ij}(t, D)$ can be computed as the sum of the contributions of all tasks in $\mathcal{T}_i$:

$$W_{ij}(t, D) = \sum_{1 \leqslant k \leqslant N_i} W_{ikj}(t, D). \tag{3}$$

To compute $W_{ikj}(t, D)$, we need the distance ρ_{ikj} between the first activation time of a job of task τ_{ik} inside the busy period and the busy period itself, considering τ_{ij} as the starting task. It can be shown [4] that:

$$\rho_{ikj} = \bigl(T_i - (\phi_{ij} + J_{ij} - \phi_{ik})\bigr) \mod T_i.$$

Hence, $W_{ikj}(t, D)$ can be computed as follows:

$$W_{ikj}(t, D) = \max\left(\left\lfloor \frac{J_{ik} + \rho_{ikj}}{T_i} \right\rfloor + \min\left(\left\lceil \frac{t - \rho_{ikj}}{T_i} \right\rceil, \left\lfloor \frac{D - \rho_{ikj} - d_{ik}}{T_i} \right\rfloor + 1\right), 0\right) C_{ik}. \tag{4}$$

Unfortunately, Theorem 1 does not tell us which task is effectively the starting task for $\mathcal{T}_i$. Therefore, if we want to run an exact analysis, we need to consider every possible task of each transaction as the starting task for that transaction, and compute the response time of τ_{ab}^k for every possible combination of starting tasks. Unfortunately, this would lead to an intractable analysis since we would need to consider $\prod_{i=1}^{M} N_i$ cases. In order to obtain a tractable analysis some pessimism is introduced by considering an upper bound $W_i(t, D)$ to the worst-case contribution of $\mathcal{T}_i$ as the maximum among all possible starting tasks: $W_i(t, D) = \max_{1 \leqslant j \leqslant N_i} W_{ij}(t, D)$.

Given an activation time A for τ_{ab}^k, relative to the beginning of the busy period, an upper bound to its finishing time can be computed by iterating over the following recurrence until it converges to a fixed point, or $w_{ab} > A + d_{ab}$:

$$w_{ab} = W_{ab}^A(w_{ab}, D) + \sum_{1 \leqslant i \leqslant M, i \neq a} W_i(w_{ab}, D), \tag{5}$$

where $D = A + d_{ab}$ and $W_{ab}^A(t, D)$ is the contribution of transaction $\mathcal{T}_a$, based on the activation at time A of τ_{ab}^k; $W_{ab}^A(t, D)$ can be computed similarly to $W_{ij}(t, D)$ (see [4,11] for complete equations). If (5) converges, an upper bound to the relative response time is $r_{ab} = w_{ab} - A$.

3.2. Taking offsets into account

If the transaction offsets are known a priori, Theorem 1 gives us a pessimistic condition since there may be no time in which $M - 1$ tasks are released simultaneously.

An improvement can be obtained by taking the transaction offsets explicitly into account. In [6,7], we showed how to perform a schedulability analysis for EDF-scheduled task sets when tasks have offsets. We will now extend our methodology to the response time analysis of transaction sets of the type analyzed in the previous section. For simplicity, we will suppose that tasks experience no release jitter; this is not a major concern since our new response time analysis will be used in Section 4.2 by iterative methods that do not use jitter to enforce the precedence constraints. The main idea behind our methodology is that of *minimum activation time distance*. Whenever transaction offsets are considered, it may be impossible for tasks pertaining to different transactions to be activated simultaneously. However, we can always compute the minimum distance between activations of any two tasks, as the following lemma explains.

Lemma 3. *The minimum time distance between any activation time of task τ_{pq} and the successive activation time of task τ_{ij} is equal to*:

$$\Delta_{pqij} = (\phi_i + \phi_{ij} - \phi_p - \phi_{pq}) \mod \gcd(T_p, T_i).$$

Proof. Note that for any two jobs τ_{pq}^{x} and τ_{ij}^{y}:

$$a_{ij}^{y} - a_{pq}^{x} = \phi_i + \phi_{ij} - \phi_p - \phi_{pq} + (y-1)T_i - (x-1)T_p.$$

This also implies:

$$\forall x \geqslant 0,\ \forall y \geqslant 0,\ \exists z \in \mathbb{Z}, \quad a_{ik}^{y} - a_{pq}^{x} = \phi_i + \phi_{ik} - \phi_p - \phi_{pq} + z \gcd(T_i, T_p)$$

thus the minimum difference corresponds to the thesis. □

Once Δ_{pqij} has been defined, we can modify Theorem 1 in order to obtain tighter worst-case response times.

Theorem 4. *The worst-case relative response time r_{ab} of task τ_{ab} corresponds to the response time of some job τ_{ab}^{k} activated inside a busy period where some task τ_{pq} (possibly τ_{ab} itself), which we will call the starting task, is activated at the beginning of the busy period and for each other transaction $\mathcal{T}_i, i \neq a, p$, there is a task τ_{ij} that is activated Δ_{pqij} time units after the beginning of the busy period.*

Proof. If job τ_{ab}^{k} is activated inside a busy period $[t_1, t_2)$, we can always choose t_1 such that the processor is not busy at $t_1 - 1$. Therefore, there is surely at least one task that is released at the beginning of the busy period, say τ_{pq} (note that it can be $a = p$). The worst-case response time r_{ab} can be found when every transaction offers its worst-case contribution to the finishing time of τ_{ab}^{k}. Suppose that the worst-case contribution for transaction $\mathcal{T}_i$, $i \neq a, p$ in any busy period starting with the activation of a job of τ_{pq} can be found when some task τ_{ij} is the first task of $\mathcal{T}_i$ to be released inside the busy period. Now, if we move the activation pattern of transaction $\mathcal{T}_i$ so that the activation of the first job of τ_{ij} inside the busy period occurs earlier, but still inside the busy period, the contribution of $\mathcal{T}_i$ increases. In fact, new jobs of tasks of $\mathcal{T}_i$ may now contribute to the finishing time of τ_{ab}^{k} (either because their activation is moved inside the busy period or because their absolute deadline becomes less than or equal to the one of τ_{ab}^{k}). From Lemma 3, Δ_{pqij} is the minimum possible distance between an activation of τ_{pq}, and thus the beginning of the busy period, and any activation of τ_{ij}. Hence, the theorem follows. □

By using Theorem 4, we can develop a new response time computation method along the line of Palencia and Gonzàlez's one. Once a starting task τ_{pq} and an activation time for a job τ_{ab}^{k} have been fixed, we can compute a new term $W_{ij}^{pq}(t, D)$ for the contribution of transaction $\mathcal{T}_i$, supposing that task τ_{ij} is activated at its minimum time distance Δ_{pqij} from τ_{pq}. In particular, the distance ρ_{ikj}^{pq} between the first activation of any task τ_{ik} inside the busy period and the busy period itself can be computed as follows:

$$\rho_{ikj}^{pq} = (\Delta_{pqij} + \phi_{ik} - \phi_{ij}) \mod T_i.$$

$W_{ikj}^{pq}(t, D)$ and $W_{ij}^{pq}(t, D)$ can then be computed by applying equations similar to (4) and (3):

$$W_{ikj}^{pq}(t, D) = \max\left(\left\lfloor \frac{J_{ik} + \rho_{ikj}^{pq}}{T_i} \right\rfloor + \min\left(\left\lceil \frac{t - \rho_{ikj}^{pq}}{T_i} \right\rceil, \left\lfloor \frac{D - \rho_{ikj}^{pq} - d_{ik}}{T_i} \right\rfloor + 1\right), 0\right) C_{ik},$$
$$W_{ij}(t, D) = \sum_{1 \leqslant k \leqslant N_i} W_{ikj}(t, D).$$

Since we do not know which task τ_{ij} leads to the maximum contribution, we use an upper bound $W_i^{pq}(t, D) = \max_{1 \leqslant j \leqslant N_i} W_{ij}^{pq}(t, D)$ to obtain a tractable analysis.

As for the possible activation times of τ_{ab}, it suffices to note that the first possible activation time lies at Δ_{pqab} time units after the beginning of the busy period; successive activations are spaced out by $\gcd(T_p, T_a)$ time units.

An upper bound to the response time for task τ_{ab} can then be computed by using a recurrence similar to the one in Eq. (5):

$$w_{ab}^{pq} = W_{ab}^{A}\left(w_{ab}^{pq}, D\right) + \sum_{1 \leqslant i \leqslant M, i \neq a} W_i^{pq}\left(w_{ab}^{pq}, D\right)$$

and considering the maximum w_{ab}^{pq} over all possible starting tasks τ_{pq}. The exact form of the final algorithm can be easily derived from the previous equations. For space constraints, we omit the mathematical development of such equations. A complete analysis can be found in [11].

In the remainder of the paper, NTO (Non-Transaction Offsets) will be used to refer to the original Palencia–Gonzàlez method, while TO (Transaction Offsets) will be used for our new method. We conclude by formally proving that TO provides tighter results than NTO.

Theorem 5. *Given a transaction set $\mathcal{T}$ where task jitters are zero and a task τ_{ab} of $\mathcal{T}_a$, the worst-case relative response time r_{ab} computed by TO is less than or equal to the worst-case relative response time computed by NTO.*

Proof. While checking the activation times for τ_{ab} as described in Theorem 2 yields the same worst-case response time as checking all possible activation times inside the busy period [4], the TO analysis limits the activation times to be checked to a subset of the busy period. Therefore, it suffices to prove that once an activation time for a job τ_{ab}^k has been fixed, the response time of τ_{ab}^k computed by TO is less than or equal to the one computed by NTO. It now suffices to prove that for each transaction $\mathcal{T}_i$, $i \neq a$:

$$\forall p,\ 1 \leqslant p \leqslant M,\ \forall q,\ 1 \leqslant q \leqslant N_p,\ \forall t, D \in \mathbb{N}: \quad W_i^{pq}(t, D) \leqslant W_i(t, D). \tag{6}$$

In fact, both TO and NTO compute the response time using a recurrence over the sum of the upper bounds to the contributions; moreover, the contribution $W_{ab}^A(t, D)$ of $\mathcal{T}_a$ does not change between TO and NTO. Hence, the theorem follows directly from (6).

It now remains to prove (6). Let us start by considering the contributions $W_{ij}^{pq}(t, D)$ and $W_{ij}(t, D)$ of TO and NTO, respectively. Since release jitters are equal to zero, the only difference in computing $W_{ij}^{pq}(t, D)$ with respect to $W_{ij}(t, D)$ is that the activation time of τ_{ij} is deferred for a time Δ_{pqij} from the beginning of the busy period. Therefore, both activation times and absolute deadlines of all jobs of $\mathcal{T}_i$ are deferred in $W_{ij}^{pq}(t, D)$ and thus $W_{ij}^{pq}(t, D) \leqslant W_{ij}(t, D)$, since some jobs may not be scheduled in the busy period due to a deferred activation time or may not count due to a deferred absolute deadline. But since $W_i(t, D) = \max_{1 \leqslant j \leqslant N_i} W_{ij}(t, D)$ and $W_i^{pq}(t, D) = \max_{1 \leqslant j \leqslant N_i} W_{ij}^{pq}(t, D)$, it follows that $W_i(t, D)^{pq} \leqslant W_i(t, D)$. □

4. Iterative algorithms

In the previous Section 3 we presented methods to compute task response times for transaction sets where task offsets and jitters have been used to enforce precedence constraints. In this section we show how we can effectively assign offsets and jitters based on the computed worst-case global response times, in order to obtain iterative algorithms that converge to a stable solution that satisfies all precedence constraints. We will present three algorithms. The first one, called WCDO and presented in Section 4.1, was proposed by Palencia and Gonzàlez. It updates task jitters at each iteration step and is unable to take advantage of transaction offsets. The second and third one, algorithms CDO and MDO, are our original contribution to the problem and are introduced in Sections 4.2 and 4.3. Both update task offsets at each step and are specifically designed to use our TO response time analysis, which is able to exploit transaction offsets to provide tighter worst-case response times.

4.1. Original holistic analysis

In this section we describe the holistic analysis proposed by Palencia and Gonzàlez in [2], algorithm WCDO (Worst-Case Dynamic Offsets), which is an extension of the original analysis by Tindell and Clark [1]. In algorithm WCDO, the offset of each task is initially set to the minimum possible completion time of the previous task plus the transmission delay:

$$\phi_{ij} = \sum_{1 \leqslant k < j} (\delta_{ik} + C_{ik}) + \delta_{ij} \quad \forall 1 \leqslant j \leqslant N_i. \tag{7}$$

Task jitters are initially set to 0, and then the worst-case global response time R_{ij} is computed for each task, using the NTO analysis. At this point, jitters are modified as follows:

$$J_{i1} = 0,$$
$$J_{ij} = R_{i,j-1} + \delta_{ij} - \phi_{ij} \quad \forall 1 < j \leqslant N_i, \tag{8}$$

while offsets never change. Note that this basically means that at each step jitters are modified so that each task τ_{ij} is released at worst δ_{ij} time units from the completion time of the immediately preceding task $\tau_{i,j-1}$.

After setting the jitters, new worst-case response times are computed using NTO, jitters are modified again and so on until the system converges to a stable result or diverges. In the latter case, it is usually possible to stop the iteration after $R_{ij} > D_{ij}$ for some task τ_{ij}, since this means that we cannot prove that the system is schedulable.

More formally, we define $\mathbf{R}^k$ as the response time vector $\{R^k_{11}, \ldots, R^k_{1N_1}, \ldots, R^k_{M1}, \ldots, R^k_{MN_M}\}$ of worst-case global response times computed at some step k of the algorithm, and the $\leqslant$ operator over the space of response time vectors as follows:

$$\mathbf{R}' \leqslant \mathbf{R}'' \quad \Leftrightarrow \quad \forall 1 \leqslant i \leqslant M,\ \forall 1 \leqslant j \leqslant N_i\colon \quad R'_{ij} \leqslant R''_{ij}.$$

Note that the $\nleqslant$ operator is consequently defined as follows:

$$\mathbf{R}' \nleqslant \mathbf{R}'' \quad \Leftrightarrow \quad \exists 1 \leqslant i \leqslant M,\ \exists 1 \leqslant j \leqslant N_i\colon \quad R'_{ij} > R''_{ij}.$$

We shall further introduce function wcdo($\mathbf{R}$) as the function that, given the worst-case global response times at some step k, evaluates new response times by computing jitters as in Eqs. (8) and running the NTO analysis. Algorithm WCDO can then be expressed as an iteration of the type $\mathbf{R}^{k+1} = \mathrm{wcdo}(\mathbf{R}^k)$, starting from the best-case global response time vector:

$$\mathbf{R}^0 = \left\{ R_{ij} = \sum_{1 \leqslant k \leqslant j} (C_{ik} + \delta_{ik}) \right\}. \tag{9}$$

If the response times do not diverge, algorithm WCDO is proven to converge to a fixed value because function wcdo($\mathbf{R}$) is monotonic, as it follows from this theorem:

Theorem 6. [11] *The worst-case global response times computed by NTO are monotonically non-decreasing in the jitters.*

4.2. Algorithm CDO

We now introduce algorithm CDO (Cycling Dynamic Offsets), our original contribution to the problem. The basic idea is to modify Palencia and Gonzàlez algorithm to take into account offsets between tasks of different transactions. However, such extension is not immediate, because our TO analysis does not consider jitters. Modifying TO for taking jitters into account would unnecessarily increase the complexity of the analysis.

Instead, we believe that a way to simplify both the model and the problem is to eliminate the jitter variable from the holistic analysis. As demonstrated in Theorem 7, this simplification leads to tighter global response times even if when we use the NTO analysis.

The idea is the following. We use an iterative algorithm similar to WCDO, using the same starting offsets as in Eq. (7) and zero jitters. However, instead of updating the jitters at each step, we update the offsets, based on the worst-case global response times computed at the step before, as follows:

$$\phi_{i1} = \delta_{i1},$$
$$\phi_{ij} = R_{i,j-1} + \delta_{ij} \quad \forall 1 < j \leqslant N_i, \tag{10}$$

while jitters remains zero.

Since jitters are always zero, we can use either the NTO or the TO analysis to compute response times at each step. We will initially use the NTO analysis to compare our approach with the WCDO algorithm; later in this section, we will move to the TO analysis. We denote function $\mathrm{f_{NTO}}$ as the function that, given the worst-case global response times at some step k, evaluates new response times by computing offsets as in Eqs. (10) and running the NTO analysis. The algorithm can then be expressed as an iteration over $\mathbf{R}^{k+1} = \mathrm{f_{NTO}}(\mathbf{R}^k)$, starting from $\mathbf{R}^0$ as defined in Eq. (9).

In order to help understand the differences between WCDO and CDO, Fig. 2 shows how offsets and jitters are set in the two cases, given computed worst-case global response times (for the sake of simplicity, we show a single transaction with $\delta_{ij} = 0$ for every task τ_{ij}). Note that the two models are not equivalent from a scheduling point of

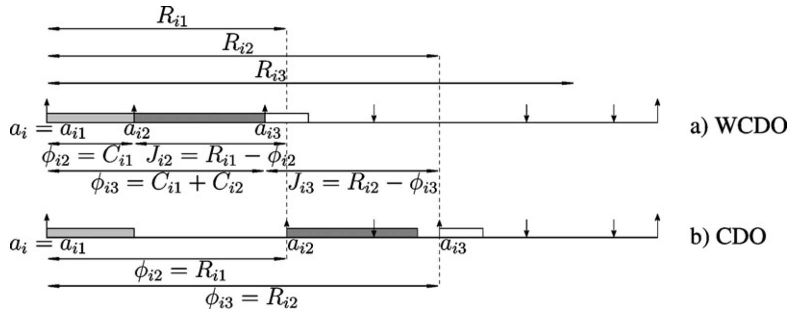

Fig. 2. Setting offsets and jitters.

view, in the sense that in our algorithm we are explicitly prohibiting a job τ_{ij}^k to be released before its activation times $a_{ij}^k = a_i^k + R_{i,j-1} + d_{ij}$, while in the WCDO model job τ_{ij}^k may be released before if the previous job $\tau_{i,j-1}^k$ experiences a response time less than the worst-case.

The programming model of our application changes as well. In standard holistic analysis, each task in the transaction (except the first one) is blocked waiting for an explicit activation from the preceding task, with a signal, a semaphore or a message (in case of tasks located on different nodes). The first task in a transaction is periodically activated by a timer event. In our model, instead, all the tasks in a transaction are periodically activated at their respective activations, that are spaced by their respective offsets.

One might think that our algorithm gives more pessimistic results than WCDO, because the release times of the tasks are delayed most of the times. In fact, the contrary is effectively true: at each step, our algorithm provides tighter worst-case global response times than WCDO, as the following theorem proves.

Theorem 7. *Given a response time vector* $\mathbf{R}$, $f_{\mathrm{NTO}}(\mathbf{R}) \leqslant \mathrm{wcdo}(\mathbf{R})$.

Proof. Let $\mathbf{R}' = \mathrm{wcdo}(\mathbf{R})$ and $\mathbf{R}'' = f_{\mathrm{NTO}}(\mathbf{R})$. Furthermore, let ϕ'_{ij} and J'_{ij} be the offset and jitter for each task τ_{ij} as computed by wcdo (using Eqs. (7) and (8) with response time $\mathbf{R}$) and let ϕ''_{ij} be the offset as computed by f_{NTO} (using Eqs. (10)). Then it suffices to prove that $\forall 1 \leqslant a \leqslant M, \forall 1 \leqslant b \leqslant N_i$: $R''_{ab} \leqslant R'_{ab}$.

Using Eqs. (8) we easily obtain $\phi''_{ij} = J'_{ij} + \phi'_{ij}$ for each task τ_{ij}. This means that assuming the same activation time a_i^k, the activation time of any job τ_{ij}^k in f_{NTO} coincides with the release time of τ_{ij}^k in wcdo after having experienced the maximum jitter. Therefore, once we select a starting task τ_{ij} for a transaction $\mathcal{T}_i$ in order to compute its contribution $W_{ij}(t, D)$ according to Theorem 1, the transaction activation time is the same for f_{NTO} and wcdo. Also note that while the activation times of τ_{ab} to be considered as described in Theorem 2 do change, the associated activation times of transaction $\mathcal{T}_a$ remain the same since the global relative deadlines of the tasks are modified by neither f_{NTO} nor wcdo.

Therefore, to show that $R''_{ab} \leqslant R'_{ab}$, it suffices to prove that once the activation times of all transactions have been set, the contribution $W_{ij}(t, D)$ of each transaction in f_{NTO} is less than or equal to the one in wcdo. The only difference in the contribution is that some jobs that are activated inside the busy period in wcdo may be activated outside the busy period in f_{NTO} due to a deferred activation time, and thus do not contribute to the response time of τ_{ab}. Hence, the theorem is proved. □

We just proved that, applying one step of the two algorithms on the same vector $\mathbf{R}$, our algorithm gives tighter response times. However, we have no guarantees, using either the TO or the NTO analysis, that response times are monotonic in the offsets. In fact, we found transaction sets in which increasing the offsets leads to tighter response times. This means that the iterative algorithm is not guaranteed to converge, even if the response times do not diverge, since it could fall into a limit cycle.

Figure 3 illustrates the problem. It shows a possible evolution of the response times computed at each step.[2] The response time vectors computed at each step are numbered from $\mathbf{R}^0$ to $\mathbf{R}^8$; arrows represent the application of

[2] For the sake of simplicity, we show only two response times in the figure. However, the reader must consider that the response time vector is defined over a multi-dimensional space.

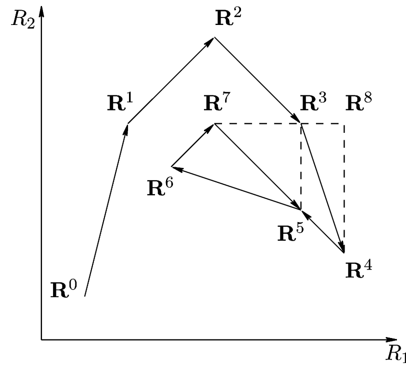

Fig. 3. A limit cycle.

1. Given $\mathbf{R}^k$, compute $\mathrm{f_{NTO}}(\mathbf{R}^k)$.
2. If $\mathrm{f_{NTO}}(\mathbf{R}^k) \leqslant \mathbf{R}^k$, stop the algorithm with final response times $\mathbf{R}^k$.
3. Otherwise compute $\bar{k} = \mathrm{cycle}(\mathbf{R}^k)$.
   (a) If $\bar{k} = -1$, then end the iteration step with $\mathbf{R}^{k+1} = \mathbf{R}^k$.
   (b) Otherwise compute $\mathbf{R}'^k = \max \mathrm{R}(\bar{k}, k)$ and $\bar{k}' = \mathrm{cycle}(\mathbf{R}'^k)$, then go back to step 3(a) considering $\mathbf{R}'^k$ and $\bar{k}'$ instead of $\mathbf{R}^k$ and $\bar{k}$, respectively.

Fig. 4. CDO iteration step.

function $\mathrm{f_{NTO}}$ at each step. In the example, if we apply the algorithm described above, the iteration enters a limit cycle at step 5.

We need to identify cycles and find a way of exiting. In Fig. 4 we report the final CDO algorithm. Each iteration step k in algorithm CDO is done as follows. First, if $\mathrm{f_{NTO}}(\mathbf{R}^k) \leqslant \mathbf{R}^k$, we can immediately stop the algorithm with final response times $\mathbf{R}^k$. In fact, this means that using the offsets computed at step k we obtain response times $\mathbf{R}^k$ that are compatible with the deadlines and the offsets, so we can stop the algorithm.

Otherwise, we must check if we incurred in a limit cycle. This can be done with the following function:

$$\mathrm{cycle}\big(\mathbf{R}^k\big) = \max\big(\big\{\bar{k} \in \mathbb{N} \mid \mathrm{f_{NTO}}\big(\mathbf{R}^k\big) = \mathbf{R}^{\bar{k}}\big\} \cup -1\big). \tag{11}$$

Cycle($\mathbf{R}^k$) returns -1 if no cycle can be found or $\bar{k}$ if a limit cycle is found starting at step $\bar{k}$. If $\mathrm{cycle}(\mathbf{R}^k) = -1$, then we simply set $\mathbf{R}^{k+1} = \mathrm{f_{NTO}}(\mathbf{R}^k)$. If $\mathrm{cycle}(\mathbf{R}^k) = \bar{k} \geqslant 0$, we *jump out* of the limit cycle by selecting a new response time vector as the maximum between all response times in the limit cycle. To this purpose, we define a function $\max \mathrm{R}$ as follows:

$$\max \mathrm{R}(k_1, k_2) = \begin{pmatrix} \max_{k \in \{k_1 \ldots k_2\}}(R_{11}^k) \\ \vdots \\ \max_{k \in \{k_1 \ldots k_2\}}(R_{MN_M}^k) \end{pmatrix}.$$

Then, the new response time vector can be computed as $\max \mathrm{R}(\bar{k}, k)$.

Unfortunately, when we jump out of a limit cycle we could incur in another cycle. An example is presented in Fig. 3. Suppose we are at step $k = 7$. When we jump out of cycle $\{\mathbf{R}^5, \mathbf{R}^6, \mathbf{R}^7\}$, we find point $\mathbf{R}^3$ that has already been visited. If we simply set $\mathbf{R}^8 = \max \mathrm{R}(5, 7)$, we incur in the new limit cycle $\{\mathbf{R}^3, \ldots, \mathbf{R}^7\}$. In order to prevent this problem, each time we jump out of a cycle we must check again if we incur in a new limit cycle. This can be done by using function (11) and possibly jump out of this new cycle too. Of course, the problem can be found again, recursively. However, every time we jump out of a cycle, we can only incur in a cycle including more points. Therefore, sooner or later we must find a point which is not part of any cycle.

The following theorem proves that algorithm CDO is indeed correct and that it provides tighter response times with respect to WCDO.

Theorem 8. *Given a transaction set $\mathcal{T}$, if WCDO converges to response times $\bar{\mathbf{R}}$, then CDO converges to response times $\mathbf{R}' \leqslant \bar{\mathbf{R}}$ in a finite number of steps*; *furthermore, algorithm CDO is correct, in the sense that if offsets are set according to Eqs.* (10) *with response times $\mathbf{R}'$, $\mathbf{R}'$ are upper bounds to the worst-case global response times for $\mathcal{T}$ respecting task precedence constraints.*

Proof. Since WCDO converges to $\bar{\mathbf{R}}$, $\mathrm{wcdo}(\bar{\mathbf{R}}) = \bar{\mathbf{R}}$. Moreover, since wcdo is monotonic, $\forall \mathbf{R} \leqslant \bar{\mathbf{R}}, \mathrm{wcdo}(\mathbf{R}) \leqslant \bar{\mathbf{R}}$. From Theorem 7 we also obtain: $\forall \mathbf{R} \leqslant \bar{\mathbf{R}}, \mathrm{f_{NTO}}(\mathbf{R}) \leqslant \bar{\mathbf{R}}$. Clearly $\mathbf{R}^0 \leqslant \bar{\mathbf{R}}$. CDO can never reach at any step a point $\mathbf{R} \nleqslant \bar{\mathbf{R}}$. If this was possible, we could surely find a step k so that $\forall k' \leqslant k$, $\mathbf{R}^{k'} \leqslant \bar{\mathbf{R}} \wedge \mathbf{R}^{k+1} \nleqslant \bar{\mathbf{R}}$. However, this is impossible. In fact, $\mathbf{R}^{k+1}$ can be obtained from $\mathbf{R}^k$ by application of either function $\mathrm{f_{NTO}}(\mathbf{R}^k)$ or function $\max \mathrm{R}(k_1, k_2)$ with $k_1, k_2 \leqslant k$, and neither of them can give a response time vector that is not less than or equal to $\bar{\mathbf{R}}$.

Since the number of points $\mathbf{R} \leqslant \bar{\mathbf{R}}$ is finite, to prove the first part of the theorem it now suffices to show that CDO never passes through the same point twice before stopping. It is impossible that, for any step k, $\exists k' \leqslant k$, $\mathbf{R}^k = \mathbf{R}^{k'}$, since the iterative step of CDO only ends when function (11) returns -1, meaning that no such k' can be found. Therefore, algorithm CDO visits a new point at each step and thus must stop in a finite number of steps with response times $\mathbf{R}' \leqslant \bar{\mathbf{R}}$.

Furthermore, if algorithm CDO converges to response times $\mathbf{R}'$, $\mathbf{R}'' = \mathrm{f_{NTO}}(\mathbf{R}') \leqslant \mathbf{R}'$; otherwise, applying the iterative step described in Fig. 4 to $\mathbf{R}'$ we would obtain a new response time vector $\mathbf{R}''' \nleqslant \mathbf{R}'$. Therefore, if we set the offsets of $\mathcal{T}$ according to Eqs. (10) with response times $\mathbf{R}'$, we obtain $\forall 1 \leqslant i \leqslant M$, $1 < j \leqslant N_i$, $R''_{i,j-1} + d_{ij} \leqslant \phi_{ij}$ and therefore the precedence constraints are met. □

If we now want to use the TO analysis instead of the NTO one, we can simply define a new function $\mathrm{f_{TO}}(\mathbf{R})$ that computes the response times in the same way as $\mathrm{f_{NTO}}(\mathbf{R})$ but using the TO analysis instead of the NTO one, and then substitute $\mathrm{f_{NTO}}$ with $\mathrm{f_{TO}}$ in the iterative step described in Fig. 4. From Theorem 5 it is trivial to prove that Theorem 7 and consequently Theorem 8 still hold. To differentiate the two methods, we call them CDO-NTO and CDO-TO, respectively.

4.3. Simplifying the algorithm

While Theorem 8 proves that CDO is always better than WCDO, the definition of CDO is actually quite complex. We can define a simpler algorithm, that we call MDO (Maximum Dynamic Offsets). The idea is to get rid of the cycles altogether by simplifying the iterative step using the following equation:

$$\mathbf{R}^{k+1} = \begin{pmatrix} \max(R^k_{11}, \mathrm{f_{NTO}}(R^k_{11})) \\ \vdots \\ \max(R^k_{MN_M}, \mathrm{f_{NTO}}(R^k_{MN_M})) \end{pmatrix}.$$

In other words, algorithm MDO always jumps out to the maximum between the previously computed response times and the newly computed ones. By using algorithm MDO, the response time iteration clearly evolves monotonically.

Theorem 7 shows that at each step function $\mathrm{f_{NTO}}$ gives us a tighter result with respect to WCDO. Since MDO is using the same function, it can be easily proven that MDO converges to better response times then WCDO and that it is correct along the line of Theorem 8.

Intuitively, we expect that algorithm CDO performs better than MDO. However, this does not always happen, in the sense that in some rare cases MDO can actually give tighter results than CDO. In Section 5 we show by means of experimental evaluation that the difference between the two is negligible.

Since MDO can also be applied using either $\mathrm{f_{NTO}}$ or $\mathrm{f_{TO}}$, we will differentiate between algorithms MDO-NTO and MDO-TO.

5. Evaluation

We now present performance comparison between the original holistic analysis and our methodology. The comparison has been made by conducting a series of simulation experiments. For each experiment, we generated 1000 synthetic sets of transactions, each one consisting of 5 transactions with either 5 or 10 tasks, executed on either 2 or 4 processors, respectively.

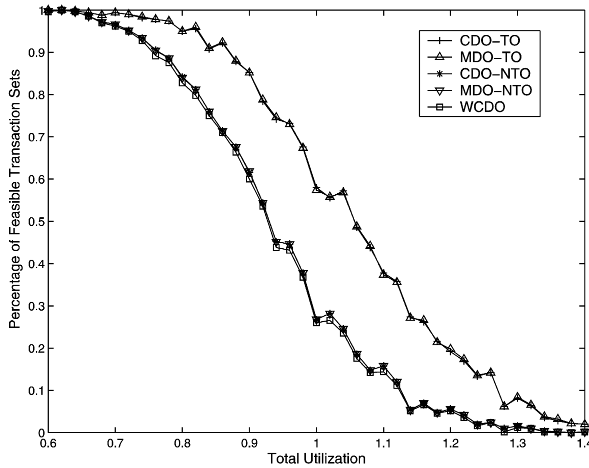

Fig. 5. 5 transactions, 5 tasks per transaction, 2 processors.

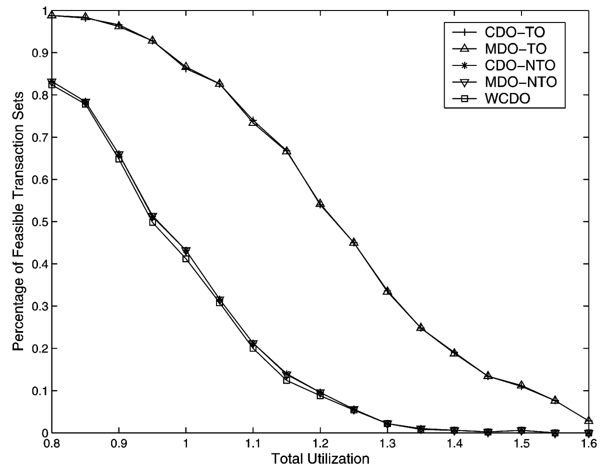

Fig. 6. 5 transactions, 10 tasks per transaction, 4 processors.

Table 1
Mean number of iteration steps, 5 transactions

	MDO-TO	MDO-NTO	WCDO
5 tasks, 1 proc	5.19	5.34	5.18
5 tasks, 2 proc	6.16	7.58	7.78
10 tasks, 4 proc	10.97	15.77	16.11

Each transaction was generated in the following way. First, a transaction utilization $U_i = \frac{1}{T_i}\sum_{j=1}^{N_i} C_{ij}$ was randomly generated according to a uniform distribution, so that the total utilization $U = \sum_{i=1}^{M} U_i$ summed up to a desired value. Transaction periods were uniformly generated between 20 and 400. The total worst-case computation time of each transaction $C_i = \sum_{j=1}^{N_i} C_i$ was computed based on utilization and period. An end-to-end relative deadline between half period and the period was assigned to each transaction, and the offset was randomly generated between 0 and the period. Afterward, computation times of tasks were also generated according to a uniform distribution, so that their sum were equal to their transaction total computation time. No delay times among tasks were considered. Finally, task deadlines were assigned as in Eq. (2), and each task was randomly assigned to a different processor.

We generated the transaction periods so that the greatest common divisor between any two periods were a multiple of 20. The greater is the gcd between two transaction periods, the larger is the minimum time distance between two successive activations of tasks of the two transactions and the smallest is the contribution of one transaction to finishing time of the tasks of the other. In particular, we showed that if a period is prime with all others, then the TO analysis is equivalent to the NTO one. Note that in real world applications, transaction periods are rarely prime with each other.

Figure 5 shows the percentage of tasks that are proved to be feasible by algorithm WCDO, MDO-NTO, MDO-TO, CDO-NTO and CDO-TO for a system of 5 transactions with 5 tasks each, running on two processors, with total utilizations ranging from 0.6 to 1.4. All 95% confidence intervals are within 5% of the mean. A first observation is that algorithms MDO-NTO and MDO-TO perform basically the same as CDO-NTO and CDO-TO, respectively, and may thus be preferable due to their simplicity. While algorithm MDO-NTO achieves a small gain over WCDO, MDO-TO achieves an improvement up and beyond 20% for utilizations around 0.75. Also, the response times computed by algorithm WCDO were 36% longer then those computed by algorithm MDO-TO on average.

Figure 6 shows the case with 5 transactions and 10 tasks per transaction, running on 4 processors. This time, algorithm MDO-TO is able to prove feasible over 50% more total transaction sets than MDO-NTO at utilizations around 1.1. The benefit of the TO approach seems to go up as the parallelism of the system increases.

Finally, Table 1 shows the mean number of iteration steps needed to achieve convergence by algorithms MDO-TO, MDO-NTO and WCDO in the cases analyzed before. As you can see, the number of steps is low in all cases and similar for the three algorithms, except for the fact that algorithm MDO-TO seems to perform better than the others under increased parallelism.

6. Shared resources

In this section, we will extend our TO analysis to cover the problem of blocking times and synchronization on shared resources. To bound the maximum blocking time experienced by tasks due to mutual exclusion a *resource access protocol* must be introduced. Many resource access protocols have been proposed in literature [12,13]; we base our discussion on the Stack Resource Protocol (SRP) [8]. Note that SRP only works for uniprocessor systems and considers no jitter. However, we assume that no resource is shared among tasks pertaining to different processors, so we can safely extend it to our transaction model once task offsets have been fixed under the TO analysis.

In the remainder of this section, we briefly introduce the SRP and some related properties. We then present the extension to the TO analysis.

6.1. SRP

Under SRP, each task is assigned a static preemption level $\pi_{ij} = \frac{1}{D_{ij}}$ in addition to its dynamic priority defined by EDF. The following fundamental property holds:

Property 1. *Task τ_{ij} can preempt task τ_{lk} only if $\pi_{ij} > \pi_{lk}$.*

To ease further definitions, we also define an additional preemption level π_s as a preemption level that is strictly greater than the preemption level of every task.

Each resource ρ_p is assigned a static ceiling $\text{ceil}(\rho_p) = \max_{ij}\{\pi_{ij} \mid \exists \xi_{ijk}, \rho_{ijk} = \rho_p\}$. A dynamic system ceiling is then defined as follows:

$$\Pi_s(t) = \max\bigl(\bigl\{\text{ceil}(\rho_p) \mid \rho_p \text{ is busy at time } t\bigr\} \cup 0\bigr).$$

The scheduling rule is the following: a job is not allowed to start execution until its priority is the highest among the active jobs and its preemption level is strictly higher then the system ceiling.

Among the many useful properties of SRP, we are mainly interested in two of them:

Property 2. [8] *Under SRP, a job can only be blocked before it starts execution*; *once started, it can only be preempted by higher priority jobs.*

Property 3. [8] *A job can be blocked only once by one lower priority job.*

Property 3 can be extended to groups of tasks executing in a busy period as proved by the following lemma:

Lemma 9. *Consider a busy period $[t_1, t_2)$, where t_3 is the greatest absolute deadline of any task that is completely executed in $[t_1, t_2)$ and t_1 is the last instant prior to t_2 such that either no jobs or a job with deadline greater than t_3 executes. Then tasks that are completely executed in $[t_1, t_2)$ may be blocked only once by a single lower priority job.*

Proof. Without blocking times, all jobs completely executed inside the busy period must be released at or after t_1; furthermore, one job must be activated exactly at t_1. We will call $\mathcal{A}$ the set of such jobs. However, when blocking times are introduced, it is possible for a job of some task τ_{ij} with deadline greater than t_3 to be executed inside the busy period. For this to be possible, the job must be inside a critical section at time t_1, since it must block some higher priority job in $\mathcal{A}$. However, there can only be one such job; otherwise, some job in $\mathcal{A}$ would be blocked by at least two lower priority jobs, which is impossible due to Property 3. □

6.2. TO extension

We extend the TO analysis by considering an added term due to blocking time in the response time computation. Due to Lemma 9, it makes sense to define a *maximum dynamic blocking time* $B^{pq}(t, D)$ as the maximum blocking time that can be experienced by any task inside a busy period of length t and maximum deadline D where τ_{pq} is activated at the beginning of the busy period. In order to define $B^{pq}(t, D)$ we first need to introduce some preliminary definitions.

First of all, note that the blocking task τ_{ij} must be activated at least $\psi_{ijk}+1$ time units before the beginning of the busy period to be able to block any task in $\mathcal{A}$; however, its activation time is further constrained by offsets relations. In order to capture this behavior, we need to compute a new minimum activation time distance between tasks.

Lemma 10. *Given two tasks τ_{pq} and τ_{ij}, the minimum time distance between any activation time of task τ_{pq} and the successive activation time of task τ_{ij} that is greater or equal to some value $k+1$ is equal to*:

$$\Delta_{pqij}^{k} = (\phi_i + \phi_{ij} - \phi_p - \phi_{pq} - k - 1) \mod \gcd(T_p, T_i) + k + 1.$$

Proof. The proof is a simple extension of the one of Lemma 3. □

Second, we define a new *minimum dynamic preemption level* which is the minimum preemption level of any task completely executed in the busy period.

Definition 1. Given an initial task τ_{pq}, we define the following minimum dynamic preemption level:

$$\pi_{pq}(t, D) = \min_{ij}\left(\left\{\pi_{ij} \mid \Delta_{pqij} + d_{ij} \leqslant D \wedge \Delta_{pqij} < t\right\} \cup \pi_s\right).$$

We can finally define the maximum dynamic blocking time for transaction systems:

Definition 2. Given an initial task τ_{pq}, the maximum dynamic blocking time is defined as:

$$B^{pq}(t, D) = \max_{ijk}\left(\left\{C_{ijk} - 1 \mid d_{ij} > D + \Delta_{ijpq}^{\psi_{ijk}} \wedge \operatorname{ceil}(\rho_{ijk}) \geqslant \pi_{pq}(t, D)\right\} \cup 0\right).$$

Lemma 11. *$B^{pq}(t_2 - t_1, t_3 - t_1)$ is an upper bound to the maximum blocking time experienced by tasks completely scheduled in a busy period $[t_1, t_2)$, where t_3 is the maximum absolute deadline of such tasks and t_1 corresponds to an activation time of task τ_{pq}.*

Proof. As in Lemma 9, let $\mathcal{A}$ be the set of jobs that are released at or after t_1 and have deadline at or before t_3. Since jobs in $\mathcal{A}$ can only be blocked by a single lower priority job, the maximum blocking time can be no longer than the length of some critical section $C_{ijk} - 1$; in fact, the blocking job can enter ξ_{ijk} at worst at $t_1 - 1$.

Furthermore, since the job cannot enter ξ_{ijk} before ψ_{ijk} time units have elapsed since its activation, and its deadline must be greater than t_3, it must also hold $d_{ij} + t_1 - \psi_{ijk} - 1 > t_3$. However, since we know that τ_{pq} is activated at time t_1, then the more restrictive condition $d_{ij} > t_3 - t_1 + \Delta_{ijpq}^{\psi_{ijk}}$ must also hold. Finally, resource ρ_{ijk} must be able to block some job in $\mathcal{A}$, thus $\operatorname{ceil}(\rho_{ijk})$ must be at least equal to the minimum preemption level of tasks in $\mathcal{A}$.

To end the proof it now suffices to prove that the minimum dynamic preemption level $\pi_{pq}(t, D)$ is indeed a lower bound to the minimum preemption level of tasks in $\mathcal{A}$. However this is obvious since every task τ_{ij} in $\mathcal{A}$ has a deadline less than or equal to t_3, therefore $\Delta_{pqij} + d_{ij} \leqslant t_3 - t_1$, and must be activated before t_2, therefore $\Delta_{pqij} < t_2 - t_1$. □

Theorem 12. *Given initial task τ_{pq} and release time A, an upper bound to the response time for τ_{ab} can be computed by using the following recursion*:

$$w_{ab}^{pq} = W_{ab}^{A}\left(w_{ab}^{pq}, D\right) + \sum_{1 \leqslant i \leqslant M, i \neq a} W_i^{pq}\left(w_{ab}^{pq}, D\right) + B^{pq}(t, D).$$

Proof. Note that Theorem 1 still holds when blocking time by a single lower priority job is considered as long as a job of τ_{pq} is released at the beginning of the busy period. Since we proved that $B^{pq}(t, D)$ is an upper bound to the blocking time experienced in any busy period of length t and maximum deadline D, the theorem follows. □

7. Schedulability analysis of heterogeneous multiprocessor system

The improved holistic analyses introduced in Section 4 can be used to provide better schedulability conditions for multiprocessor and distributed systems. A special case that, in our opinion, is susceptible of further inquiry is that of heterogeneous multiprocessor systems (also known as asymmetric multiprocessors).

An heterogeneous multiprocessor system is composed by a general purpose CPU and one or more specialized CPUs like, for example, a digital signal processor (DSP). The utility of DSPs as hardware accelerators has already been investigated [14,15]. The Texas Instruments TM320C8x, for example, is a single-chip MIMD processor integrating a 32-bits RISC processor and four 32-bits floating point DSPs.

The specialized CPUs are typically used as hardware accelerators, or coprocessors: every task runs on the general purpose CPU, and occasionally may request some computation to be performed on a coprocessor. Usually, this "communication" is synchronized, in the sense that the task is suspended until the coprocessor returns the results. A task that actually requests a coprocessor is called a *DSP task*. In this paper, for simplicity we will assume that each DSP task requires a single fixed coprocessor.

We assume that each DSP task is composed by three computation chunks: the first and the third one are executed on the general purpose processor, while the second one runs on a coprocessor. We assume that synchronization between the processor and the coprocessor is done in zero time; we could, however, account for transmission delays by introducing suspension times between the first and the second chunk of a task and between the second and the third.

The scheduling problem for such a system has been fully analyzed under fixed priority [15,16]. However, to the best of our knowledge, no convincing solution has been proposed so far for EDF. The holistic analysis offers a nice solution to the problem, since it is possible to treat each DSP task as a transaction $\mathcal{T}_i$ with three different tasks τ_{i1}, τ_{i2} and τ_{i3} corresponding to the three computation chunks of the DSP task. We can thus refer to DSP transactions instead of DSP tasks.

We discuss and show experimental results for three different cases. In the first one, we suppose that each transaction executes on a different exclusive DSP. In the second case, we assume a single preemptive DSP shared by all transactions. In the third, most realistic case, all transactions share a single non-preemptive DSP.

For each experiment, we generated 1000 transaction sets with 5 or 10 DSP transactions each and periods within 20 and 400 in the same way as in Section 5, in the sense that the computation time of each DSP transaction was divided according to a uniform distribution into its three tasks.

7.1. Multiple coprocessors

In this section, we discuss the case in which each DSP transaction executes on a different coprocessor. In this case, the coprocessor execution can be simply treated as a suspension time; that is, each DSP transaction $\mathcal{T}_i$ consists of two tasks τ_{i1} and τ_{i3} only, but a delay time $\delta_{i3} = C_{i2}$ is added.

Without using a transaction-based analysis there is no way to account for suspension times that do not occur before a job starts execution. Gai [15] gives good reasons why accounting for suspension times in EDF scheduled tasks is a difficult problem to solve. This means that if we want to use a task-based analysis, such as the processor demand criterion introduced by Baruah in [17] for synchronous task sets or our improved 1-fixed test for asynchronous task sets [6,7], we must consider each DSP transaction as a single task with worst-case execution time equal to $C_{i1} + C_{i3} + \delta_{i3}$.

In the case of fixed priority scheduling, In-Guk Kim et al. in [16] proposed an effective schedulability analysis. However, their test considers a synchronous task model, thus it fails to take task offsets into account.

Figures 7 and 8 show simulation results for the system, expressed as a percentage of schedulable transaction sets in respect to the total system utilization, for transaction sets with 5 and 10 DSP transactions, respectively. *Kim* is the analysis developed in [16] under deadline monotonic scheduling, MDO-TO is our transaction-based holistic analysis and 1-fixed is the schedulability analysis for the EDF-scheduled task-based model described above. MDO-TO achieves a dramatic performance increase over 1-fixed at utilization around 1.0: the schedulability percentage for 1-fixed quickly drops to 0, whereas with the transaction analysis we are able to guarantee almost every task. The performance shown by Kim is much better than 1-fixed, but degrades around utilization 1.2, whereas the transaction analysis has still a schedulability ratio of 50%. Finally, MDO-TO seems to work better as the number of tasks increases, while the other tests remain unchanged.

7.2. Preemptive coprocessor

We now suppose that the system offers one single preemptive coprocessor. In this case, each DSP transaction $\mathcal{T}_i$ consists of tasks τ_{i1} and τ_{i3} executed on the processor and of task τ_{i2} executed on the coprocessor. Note that the *Kim* analysis cannot be used in this case, since it does not consider the response time of the chunk executed on the DSP.

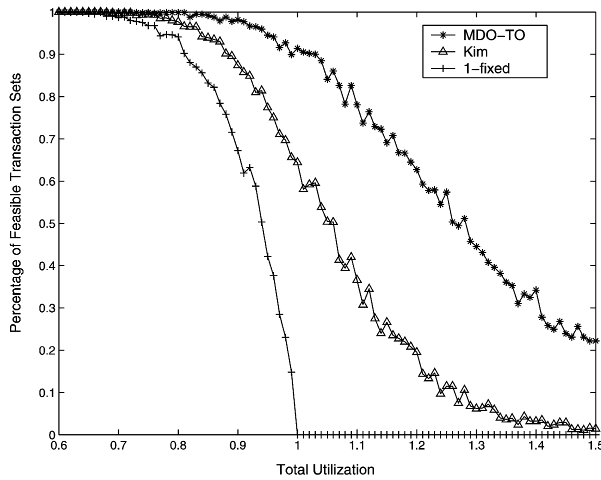

Fig. 7. 5 DSP transactions, dedicated coprocessors.

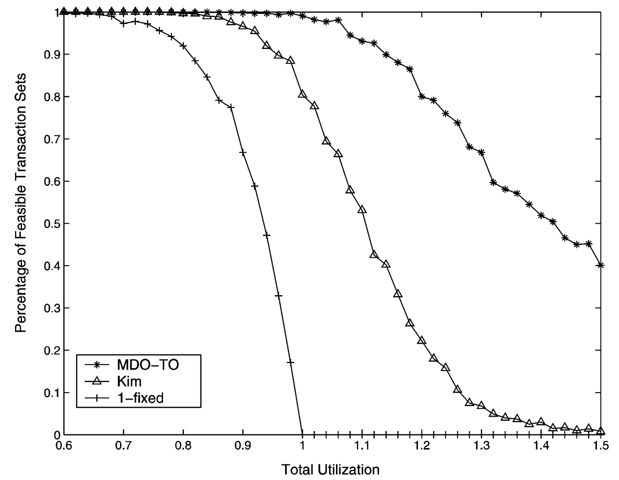

Fig. 8. 10 DSP transactions, dedicated coprocessors.

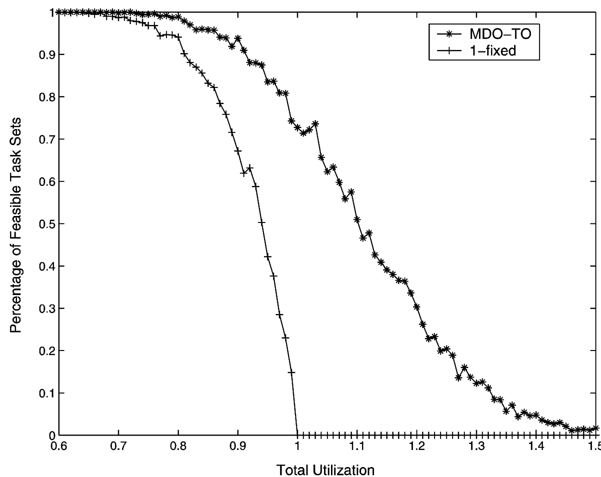

Fig. 9. 5 DSP transactions, shared preemptive coprocessor.

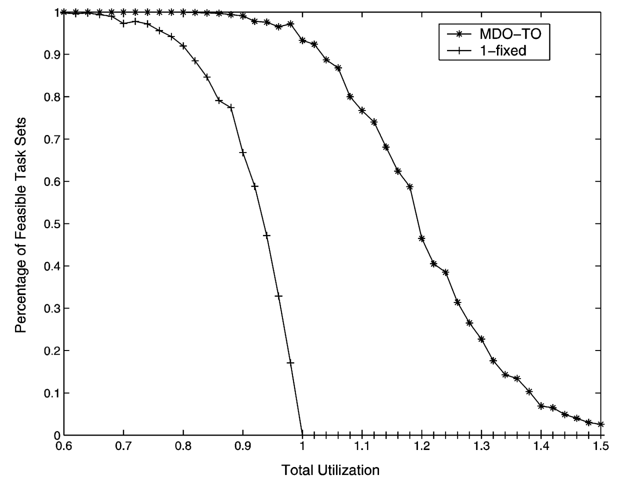

Fig. 10. 10 DSP transactions, shared preemptive coprocessor.

If we want to use 1-fixed, we must do the same assumption as in the previous case: when some code is executed on the coprocessor, the processor remains idle. Therefore, it is easy to see that in this case every DSP transaction can be modeled as a single task with execution time $C_{i1} + C_{i2} + C_{i3}$.

Figures 9 and 10 show the simulation results for transaction sets with 5 and 10 DSP transactions, respectively. Once again, the performance of MDO-TO is extremely superior for utilizations around 1.0.

7.3. Non-preemptive coprocessor

In this final case, the DSP is assumed to be non-preemptive. We can model this situation by supposing that each computation chunk on the DSP is executed inside a mutually exclusive critical section of length C_{i2}. We must then introduce blocking times to take care of the fact that a higher priority task can be blocked by a lower priority one simply because the lower priority task has taken control of the coprocessor before the activation of the higher priority one. Extensions to the 1-fixed test to account for blocking times have been introduced in [7].

The transaction-based analysis needs some in-depth considerations. If we use critical sections as detailed above, only the second task of each transaction may experience blocking. Since all such tasks execute on the same processor, the extension to the TO analysis detailed in Section 6 can be used to account for blocking times. We would like,

however, to stress an important fact. The blocking time analysis assumes that task offsets have been fixed. Since the offset based holistic analyses (CDO and MDO) change the offsets at each step, it follows that the blocking times can change at each step too. Therefore, Theorem 7 does not hold anymore and thus we cannot prove that CDO performs better than WCDO. However, we can say that algorithm MDO is correct along the line of Theorem 8 since it is still monotonic.

A second issue consists in deadline assignment. Using Eq. (2) to assign deadlines D_{i1} and D_{i2} does not constitute a good heuristic anymore, since we must take into account the blocking time. This means that if we were to use Eq. (2), the probability of τ_{i2} missing its deadline would be much higher then τ_{i1} and τ_{i3}. A simple yet much more efficient heuristic is the following:

$$D_{i1} = \frac{C_{i1}}{C_{i1} + C_{i2} + C_{i3}} D_i,$$
$$D_{i2} = \left(\frac{C_{i1} + C_{i2}}{C_{i1} + C_{i2} + C_{i3}} + \left(1.0 - \frac{C_{i1} + C_{i2}}{C_{i1} + C_{i2} + C_{i3}} \right) p \right) D_i.$$

This modified heuristic increases D_{i2} proportionally to a factor p; in particular, for $p = 0$ the above equation is equal to Eq. (2). We found through synthetic simulations that $p = 0.8$ tends to provide good results and will be consequently used in the following experiments, but even the case where $p = 1.0$ performs much better than $p = 0$.

We also designed a deadline search algorithm that, starting from the above heuristic, searches the deadline space to find an assignment that makes the task set feasible. The algorithm is detailed in Appendix A. Estimating its performance with respect to optimal assignment is difficult, due to the complexity of an optimal algorithm for practical transaction sets. However, from results obtained applying the methodology to very small transaction sets, we feel that our search algorithm could be able to find a solution in most cases in which a feasible deadline assignment exists.

Figures 11 and 12 show simulation results for transaction sets with 5 and 10 DSP tasks, respectively, where *MDO-TO*, *search* stands for the transaction analysis performed using the deadline search algorithm, and *MDO-TO*, *heuristic* for the transaction analysis performed using the improved heuristic. Once again, results for the transaction analysis are much better as the number of tasks increases. Under low utilization values, 1-fixed actually performs better than both MDO-TO, search and MDO-TO, heuristic, although the difference is negligible. This is because the effect of blocking time is worst for the transaction analysis than for plain EDF processor demand criterion. As utilization rises, the benefit of being able to reuse the coprocessor time becomes more significant and the transaction analysis becomes better than 1-fixed. The search algorithm also becomes beneficial, being able to schedule up to 10% more task sets compared to the heuristic.

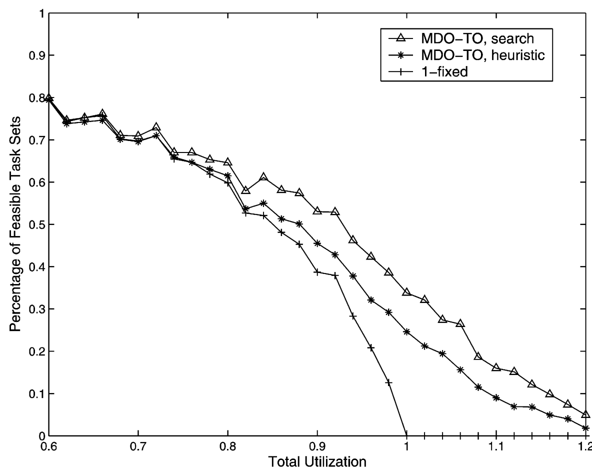

Fig. 11. 5 DSP transactions, shared preemptive coprocessor.

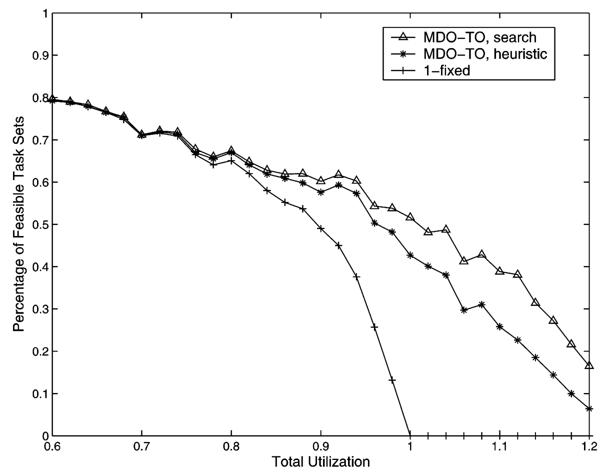

Fig. 12. 10 DSP transactions, shared preemptive coprocessor.

8. Conclusions and future work

In this paper we presented a set of algorithms for schedulability analysis of a set of distributed real-time transactions. By taking into account the offsets of the transactions and of the tasks in an efficient way, we improved over existing schedulability tests, in the sense that our algorithms provide much higher acceptance ratios and tighter worst-case response times. We also applied our algorithm MDO-TO to heterogeneous multiprocessor systems, with one general purpose processor and one or more coprocessors (DSPs). We showed that our methodology, based on the transaction model, provides better results than existing schedulability tests.

Deadline assignment remains a major problem. Although we have provided some insights on how deadline variations affect schedulability for the specific case of DSP transactions in Section 7.3, we plan to further research the issue as part of our future work.

A second unresolved issue regards *offset free* systems, i.e. systems in which the designer is free to choose the transaction offsets. Although the problem of optimally selecting offsets has been proven to be NP-hard [18], little work exists on finding suitable heuristics. We also feel that more work is needed in this direction, in order to provide designers with efficient methodologies and tools.

Appendix A. Deadline search algorithm

In this section we detail the deadline search algorithm used in Section 7.3. Starting from the heuristic of Eq. (12), the deadline search algorithm tries to find an assignment for each deadline D_{i1}, D_{i2} that guarantees the schedulability of the transaction set using algorithm MDO-TO modified as in Section 6 to take blocking times into consideration.

The algorithm iterates over the space of possible deadline assignments, choosing a new assignment at each step. Note that using a general optimum search algorithm such as simulated annealing is possible but inconvenient because at each step we can choose the next assignment based on simple yet good heuristics, while designing a suitable cost function does not seem easy. Algorithm MDO-TO is then run until one of the following occurs:

1. MDO-TO converges to a feasible solution;
2. at some step $\exists 1 \leqslant i \leqslant M$, $R_{i1} > D_{i1} \vee R_{i2} > D_{i2} \vee R_{i3} > D_{i3}$.

In the first case, the algorithm ends returning the current assignment. In the second, the algorithm first checks which deadlines are missed in the last step by MDO-TO and then updates all deadlines using heuristics that try to make the tasks that missed their deadlines schedulable. The heuristics used are detailed below in decreasing order of importance:

1. If τ_{i2} is not feasible, increase D_{i2} to give it more time to finish execution. Also, decrease D_{i1}, in order to have τ_{i1} finish earlier decreasing ϕ_{i2} as well.
2. If τ_{i1} is not feasible, increase its deadline D_{i1}.
3. If τ_{i3} is not feasible, decrease D_{i1}, so that both τ_{i1} and τ_{i2} should finish earlier. Also decrease D_{i2}, but of a value less than the one for D_{i1}, since τ_{i2} is more sensible to deadline variations due to the experienced blocking time.
4. If τ_{i2} is not feasible, increase of a small amount the deadline D_{j2} of every task τ_{j2}, $j \neq i$, as this helps τ_{i2} finish earlier.
5. If τ_{i1} is not feasible, increase of a small amount the deadline D_{j1} of every task τ_{j2}, $j \neq i$.

In order to better search the solution space, a random factor is applied to the update of each deadline. Also, the update value is scaled by a temperature which is decreased over time, helping achieving convergence. The final pseudo code is given below:

```
double temperature=initTemp;
double deadlines[NumTrans][2];
initialHeuristic(deadlines);        //set deadlines to initial heuristic
bool fail[NumTrans][3];             //if true, the corresponding
                                    //task missed its deadline
while(temperature>finalTemp) {
```

```
updateModel(deadlines);       //update model to current deadlines
if(MDOTOBlocking(failed))  //executes MDO-TO, update failed
    return true;                  //return true if task set is feasible
int numFailed[3]=0,0,0;       //counts the number of failed tasks
for(int i=0;i<numTrans;i++)
    for(int c=0;c<3;c++)
        if(fail[i][c]) numFailed[c]++;
for(int i=0;i<numTrans;i++)  //update deadlines
                              //according to heuristics
    if(fail[i][1]) {
        deadlines[i][0]-=deadlines[i][0]*N(temperature);
        deadlines[i][1]+=deadlines[i][1]*N(temperature*(1.0+p));
}
else if(fail[i][0])
    deadlines[i][0]+=deadlines[i][0]*N(temperature);
else if(fail[i][2]) {
    deadlines[i][0]-=deadlines[i][0]*N(temperature);
            deadlines[i][1]-=deadlines[i][1]*N(temperature*(1.0-p));
        }
        else {
            deadlines[i][1]+=deadlines[i][1]
                *N(temperature*(1.0+p)*numFailed[1]/numTrans);
            deadlines[i][0]+=deadlines[i][1]
                *N(temperature*numFailed[0]/numTrans);
        }
    boundAbove(deadlines);   //ensures that task deadlines do
                              //not exceed the transaction one
    temperature*=coolrate;   //update temperature
}
return false;
```

In the pseudo code, $N(\epsilon)$ stands for the normal distribution with mean ϵ and standard deviation $\sigma = 0.5\epsilon$. Note that both the mean and the standard deviation are always scaled by the temperature, therefore as the algorithm progresses the updates become smaller and more predictable. Factor p is used to differentiate the size of the update among different heuristics. We used the value $p = 0.4$.

Finally, *initTemp*, *finalTemp* and *coolrate* are used to control the temperature and thus the maximum number of steps, which is equal to $\lceil \log_{coolrate} \frac{finalTemp}{initTemp} \rceil$. We found by simulation that the algorithm gives good results even with a low number of steps. In the experiments we used values $initTemp = 0.5$, $coolrate = 0.94$, $finalTemp = 0.08$, which correspond to 30 maximum steps.

References

[1] K. Tindell, J. Clark, Holistic schedulability analysis for distributed hard real-time systems, Microproc. Microprog. 50 (1994) 117–134.
[2] J. Palencia, M.G. Harbour, Schedulability analysis for tasks with static and dynamic offsets, in: Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, 1998.
[3] M. Spuri, Analysis of deadline scheduled real-time systems, Tech. Rep. RR-2772, INRIA, France, January 1996.
[4] J. Palencia, M.G. Harbour, Offset-based response time analysis of distributed systems scheduled under EDF, in: Proceedings of the 15th Euromicro Conference on Real-Time Systems, Porto, Portugal, 2003.
[5] S. Baruah, L. Rosier, R. Howell, Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor, Real-Time Syst. 2 (1990) 301–324.
[6] R. Pellizzoni, G. Lipari, A new sufficient feasibility test for asynchronous real-time periodic task sets, in: Proceedings of the 16th Euromicro Conference on Real-Time Systems, Catania, Italy, 2004.
[7] R. Pellizzoni, G. Lipari, Feasibility analysis of real-time periodic tasks with offsets, Real-Time Syst. 30 (1) (2005) 105–128.
[8] T. Baker, Stack-based scheduling of real-time processes, Real-Time Syst. 3 (1) (1991) 67–99.
[9] G. Lipari, G. Buttazzo, Schedulability analysis of periodic and aperiodic tasks with resource constraints, J. Syst. Architect. 46 (2000) 327–338.
[10] P. Richard, On the complexity of scheduling real-time tasks with self-suspensions on one processor, in: Proceedings of the 15th Euromicro Conference on Real-Time Systems, Porto, Portugal, 2003.

[11] R. Pellizzoni, Efficient feasibility analysis of real-time asynchronous task sets, Master's thesis, Università di Pisa, 2004.
[12] M. Chen, K. Lin, Dynamic priority ceilings: A concurrency control protocol for real-time systems, Real-Time Syst. 2 (1990) 325–346.
[13] K. Jeffay, Scheduling sporadic tasks with shared resources in hard-real-time systems, in: Proceedings of the 13th IEEE Real-Time Systems Symposium, Phoenix, 1992, pp. 89–99.
[14] R. Baumgartl, H. Hartig, DSPs as flexible multimedia accelerators, in: Second European Conference on Real-Time Systems, Paris, France, 1998.
[15] P. Gai, G.C. Buttazzo, Multiprocessor DSP scheduling in system-on-a-chip architecture, in: Proceedings of the 14th Euromicro Conference on Real-Time Systems, 2002.
[16] I.-G. Kim, K.-H. Choi, S.-K. Park, D.-Y. Kim, M.-P. Hong, Real-time scheduling of tasks that contain the external blocking intervals, in: Real-Time Computing Systems and Applications, 1995, pp. 54–59.
[17] S. Baruah, A. Mok, L. Rosier, Preemptively scheduling hard-real-time sporadic tasks on one processor, in: Proceedings of the 11th IEEE Real-Time Systems Symposium, 1990, pp. 182–190.
[18] J. Goossens, Scheduling of offset free systems, Real-Time Syst. 5 (1997) 1–26.