Efficient Feasibility Analysis of Real-Time Asynchronous Task Sets

Rodolfo Pellizzoni
Università di Pisa and Scuola Superiore S. Anna, Pisa, Italy
rodolfo@sssup.it

Abstract

Several schedulability tests for real-time periodic task sets scheduled under the Earliest Deadline First algorithm have been proposed in literature, including analyses for precedence and resource constraints. However, all available tests consider synchronous task sets only, that are task sets in which all tasks are initially activated at the same time. In fact, every necessary and sufficient feasibility condition for asynchronous task sets, also known as task sets with offsets, is proven to be NP-complete in the number of tasks. We propose a new schedulability test for asynchronous task sets that, while being only sufficient, performs extremely better than available tests at the cost of a slight complexity increase. The test is further extended to task sets with resource constraints, and we discuss the importance of task offsets on the problems of feasibility and release jitter. We then show how our methodology can be extended in order to account for precedence constraints and multiprocessor and distributed computation applying holistic response time analysis to a real-time transaction-based model. This analysis is finally applied to asymmetric multiprocessor systems where it is able to achieve a dramatic performance increase over existing schedulability tests.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time systems are computing systems that must react within precise time constraints to external events. Real-time systems are gaining more and more importance in our society since an increased number of control systems relies on computer control. Examples of such applications include production processes, automotive applications, telecommunication systems, robotics and military systems. The correct behaviour of a real-time system depends not only on computation results but also on the time at which the result is provided [32, 9]. Moving an actuator at the wrong time can be as disastrous as moving it in the wrong direction.

It is important to note that real-time computing does not correspond to fast computing. While the objective of fast computing is to minimize the average response time of each task, the objective of real-time computing is to meet the timing constraints of each task in the system. The most typical timing constraint for a task is a *deadline*, that is the maximum time at which the task must complete execution. In particular, a real-time system is said to be *hard* if missing a deadline may cause catastrophic consequences on the environment under control. Obviously, the average response time of the system has no effect on its correct behaviour in a real-time system.

A key world in real-time system theory is *predictability* [33]. In other words, we must be able to predict, based on hardware and software specifications, the evolution of each task. Specifically, a hard real-time system must guarantee that all tasks remains feasible (meet their deadlines) even in the worst possible scenario. Unfortunately, most optimization techniques used for fast computing, especially on the hardware side, do not work well on real-time systems since they affect predictability. While deep processor pipelining and caching enhance a task's execution speed on average, in

the worst case they may actually introduce further delays. Furthermore, an increase of computational power or a decrease of execution time do not necessarily improve feasibility, at least in multiprocessor systems [17].

A real-time system typically consists of a set of concurrent tasks that compete over processor time. The system must thus implement a *scheduling algorithm* that decides which task must be scheduled (executed) at each time slot. In order to achieve predictable guarantees, *schedulability tests* must be developed for each scheduling algorithm. A schedulability test is an algorithm that given a task set returns a positive answer if the task set can be feasibly scheduled under its associated scheduling algorithm. This work introduces new schedulability tests for the well-know Earliest Deadline First (EDF) scheduling algorithm [24].

Chapter 2 introduces the system models that we will use in our work and presents a brief survey of available scheduling algorithms. We present both the standard periodic task model used in literature and the transaction model introduced in [25].

Chapter 3 presents our new schedulability test for asynchronous periodic task sets. Our main idea is the exploitation of *task offsets*; the offset of each task is the first time instant at which it is activated in the system. As we show by means of experimental evaluations, our technique clearly outperforms the available tests in literature, although its computation complexity is also slightly higher.

Chapter 4 provides an extension of the test developed in Chapter 3 to the case of tasks sharing resources. In order to preserve predictability, a real-time system must implement a *real-time resource access protocol* that gives guarantees on the maximum time that a task may be blocked waiting for a resource. Our discussion is based on the widely used Stack Resource Protocol (SRP) [2].

Chapter 5 provides more insight into the relation between task offsets and feasibility. We presents a new offset representation and try to develop heuristics to choose task offsets in a quasi-optimal way. However, we debate that solving the latter problem is probably impossible, at least in polynomial time.

Chapter 6 presents a relevant problem in the field of real-time control systems, that of task output jitter. The test developed in Chapter 3 is then applied to the problem and compared to existing approaches.

In Chapters 7 and 8 we extend the offset methodology applied in Chapter 3 to the transaction model. This system model is particularly suitable to

represent systems in which a task may suspend itself for a certain time, and for distributed and multiprocessor systems. Since the schedulability test for this system model is quite complex, we split the discussion into two chapters. Chapter 7 covers the problem of response time analysis for transaction systems scheduled under EDF, while Chapter 8 is about the holistic analysis of transaction sets with dynamic offsets. In both chapters we present our original contribution to the problem and in Chapter 8 we even show experimental results.

In Chapter 9 the transaction-based test is applied to the case of *heterogeneous multiprocessor systems*, or *asymmetric multiprocessors*. Experimental evaluations show how our test, combined to suitable heuristics, is able to dramatically outperforms all other analysis known so far.

Finally, in Chapter 10 we offer some conclusive thoughts about our major results and possible future work.

Chapter 2

System Model

We will start our discussion by introducing system models and general definitions that will be needed in the rest of our work. We will basically use two different models: a *simple task model*, which is very similar to the standard literature real-time task set model, and a *transaction model*, which was first introduced in [25] and is suitable to represent a variety of situations in which dependencies between tasks arises. These two models will be covered in Section 2.1 and 2.3, while in Section 2.2 we will provide a quick overview of the main results about feasibility for both fixed and dynamic priority scheduling.

2.1 Simple task model

We will now introduce our basic task model. We assume that a task set $\mathcal{T}$ made of N different tasks $\{\tau_1, \ldots, \tau_N\}$ must be scheduled in a hard real-time system. Barring exceptions, we will suppose that all tasks are executed on a single processor.

Each task τ_i consists of an infinite series of jobs $\{\tau_{i1}, \ldots, \tau_{ij}, \ldots\}$. Each job τ_{ij} consists of a thread of execution that must run for at most C_i time units; such value is called the *worst-case computation time* of task τ_i, and we suppose that it does not change between jobs of the same task. A task is said to be *periodic* if each successive job τ_{ij} is *activated* (i.e., presented to the system) at a fixed time interval T_i, which is called the task's *period*. A task is said to be *aperiodic* if it is not periodic. Note that we can't provide any guarantee about aperiodic tasks since no bound on the interval between successive activations of an aperiodic task is provided; any number

of jobs may be activated inside a time interval of any length. Therefore, in order to introduce aperiodic tasks in a hard-real time system we must execute them inside a special periodic task that is typically called a *server*. We will not be concerned with servers in the remainder of this work. A special case of aperiodic tasks worth mentioning is that of *sporadic* tasks. Successive jobs of a sporadic task are activated at intervals that are at least equal to a given *minimum inter-arrival time* T_i. In this case the lower bound on the interval permits to develop real-time guarantees on the system, therefore we are not forced to execute sporadic tasks inside a server; note, however, that in general a sporadic task performs worse than a periodic task of period equal to the minimum inter-arrival time of the sporadic task, meaning that analyses developed for periodic tasks (such as those introduced in the following chapters) cannot be always extended to sporadic tasks. In what follows we will be mainly interested in the study of periodic task sets.

A task is characterized by a tuple $(\phi_i, C_i, T_i, D_i, J_i)$, where:

- ϕ_i is the task's *offset*. The offset is the first time at which a job of the task is activated.
- C_i is the worst-case execution time of the task; note, however, that in most practical applications computing C_i is far from being easy.
- T_i is either the task's period (for periodic tasks) or the task's minimum inter-arrival time (for sporadic tasks).
- D_i is the task's *relative deadline*. A job is feasible only if it finishes at most D_i time units after its activation.
- J_i is the task's *release jitter*.

Each job τ_{ij} is thus activated at time $a_{ij} = \phi_i + jT_i$ and must finish before or at its *absolute deadline* $d_{ij} = a_{ij} + D_i$. If we let f_{ij} be the time at which job τ_{ij} finishes execution (called the *finishing time* or the *completion time* of the job), the feasibility condition for the task set becomes: $\forall 1 \leq i \leq N, \forall j \geq 1, f_{ij} \leq d_{ij}$. Each task τ_i can further experience a release jitter J_i. If the release jitter is not zero, then the *release time* of job τ_{ij} (i.e., the time at which the task is ready to be scheduled) is different from its activation time, being comprised between a_{ij} and $a_{ij} + J_i$. If not told otherwise, we will suppose that all release jitters are equal to zero, and thus the release time of a job corresponds to its activation time; in this case we will use the two terms interchangeably.

A task set is said to be *incomplete* if the task offsets are not specified. This basically means that we are required to make a pessimistic assumption on offsets. A task set is said to be *complete* if task offsets are given; it is also called a *task set with offsets*. In this case, the task set is said to be *synchronous* if all offsets are equal to zero. A task said is *asynchronous* if it is not synchronous. Note that given an asynchronous task set $\mathcal{T}$ we can always define the corresponding synchronous task set as $\mathcal{T}' = \{\tau'_1, \ldots, \tau'_N\}$, where for every task $\tau'_i : \phi'_i = 0, C'_i = C_i, T'_i = T_i, D'_i = D_i, J'_i = J_i$.

We will suppose that all task parameters are expressed by integer numbers. It can be proven that in this case, if a feasible schedule exists, then a feasible *integer schedule* (i.e. a schedule in which all preemption times are expressed by integer numbers) exists too [4]; this property is known as the *integral boundary constraint*. We can thus restrict ourselves to consider only integer schedules. A schedule will thus be a function $\sigma : \mathbb{N} \to \mathcal{T} \cup \{\emptyset\}$ that assigns to each time slot t a task τ_i, or the symbol $\emptyset$ to indicate that the processor is idle. Note that considering all task parameters and thus all activation and preemption times to be integer is not a major limitation for at least two reasons. The first one is that, if some parameters were originally expressed by rational numbers, we can always reduce them to integers multiplying them by their common denominator; there seems to be no reason to choose irrational values. The second one is that all time values inside the system, due to practical implementation, must be multiples of a basic clock tick.

We further define:

1. $U_i = \frac{C_i}{T_i}$ is the *utilization* of task τ_i; the utilization is a measure of how much computation time the task requires.

2. $U = \sum_{i=1}^{N} U_i$ is the *total utilization* of task set $\mathcal{T}$.

3. $\Phi = \max\{\phi_1, \ldots, \phi_N\}$ is the largest offset.

4. function $\gcd(T_i, T_j)$ is the greatest common divisor between two periods T_i and T_j; $\gcd(T_i, \ldots, T_j)$ is the same among periods $T_1, \ldots, T_j$.

5. function $\mathrm{lcm}(T_1, T_j)$ is the least common multiple between two periods T_1, T_j, and $\mathrm{lcm}(T_i, \ldots, T_j)$ is the same among periods $T_1, \ldots, T_j$.

6. $H = \mathrm{lcm}\{T_1, \ldots, T_N\}$ is the *hyperperiod* of $\mathcal{T}$.

7. $\eta_i(t_1, t_2) = \left(\left\lfloor \frac{t_2 - \phi_i - D_i}{T_i} \right\rfloor - \left\lceil \frac{t_1 - \phi_i}{T_i} \right\rceil + 1 \right)_0$ is the number of jobs of task

τ_i with release time greater than or equal to t_1 and deadline less then or equal to t_2 [5].

Also, notation $(x)_0$ will be used as an abbreviation of $\max(x, 0)$.

An essential concept is that of *busy period*. A busy period $[t_1, t_2)$ is an interval of time in which the processor is always busy, that is: $\forall t \in [t_1, t_2), \sigma(t) \neq \emptyset$.

In all the figures showing a schedule, we represent the execution of each task on a separate horizontal line. Upward arrows represent release times while downward arrows represent absolute deadlines.

2.1.1 Resource usage

In our simple task model, tasks can be synchronized using *shared resources*. This synchronization model is commonly used in shared memory systems, as many real-time systems and most embedded systems are. We consider a set $\mathcal{R}$ of R shared resources $\rho_1, \ldots, \rho_R$. To simplify our presentation, only single-unit resources are considered, although there are ways to consider the case of multi-unit resources [2].

Any task τ_i is allowed to access shared resources only through mutually exclusive *critical sections*. Each critical section ξ_{ij} is described by a 3-ple $(\rho_{ij}, \phi_{ij}, C_{ij})$, where:

1. ρ_{ij} is the resource being accessed;
2. ϕ_{ij} is the earliest time, relative to the release time of job τ_{ij}, that the task can enter ξ_{ij};
3. C_{ij} 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. Note that our model, which was first proposed in [23], is actually slightly different from the classic one in literature in that it requires earliest entry time to be known. We feel, however, that such a request should not pose too great a problem in practical applications.

Since critical sections must be executed in a mutually exclusive way to guarantee synchronization, a task can't enter a critical section using a resource ρ_k if another task is using the same resource. This means that a task can be forced to wait until another one finishes executing a critical section before restarting execution. Such a task is said to be *blocked* by the task holding the resource.

2.2 Scheduling algorithms

Two classes of scheduling algorithms have been proposed in literature: *fixed priority scheduling* (FPS) and *dynamic priority scheduling* (DPS). Note that all the following results are valid if no resource synchronization is considered.

In fixed priority scheduling, each task τ_i is assigned a fixed priority value. At each time instant, the scheduler schedules the active task with the highest priority (a task is said to be active if its current job has been released but has not finished yet and it is not blocked). Under the *Rate Monotonic* algorithm (or briefly RM), each task is assigned a priority that is inversely proportional to its period. Rate Monotonic is proven to be optimal (meaning that if a task set is schedulable under any algorithm, it is schedulable under RM) among all fixed priority algorithms in the case where task deadlines are equal to task periods [24]. In the case in which deadlines are less than or equal to the periods, the *Deadline Monotonic* algorithm (or DM), where each task is assigned a priority inversely proportional to its relative deadline, is optimal instead [22].

In dynamic priority scheduling, each task is also assigned a priority value, but such value can be dynamically adjusted. For example, under the Earliest Deadline First algorithm (or briefly EDF), each task is assigned a priority inversely proportional to the absolute deadline of its current job; priorities are thus changed each time a new job is activated. EDF is proven to be optimal among all scheduling algorithms [13].

If deadlines are equal to the periods, than the condition $U \leq 1$ is a necessary a sufficient feasibility condition under EDF. Such a condition is only sufficient if deadlines are less than the periods. Under RM, two similar feasibility bounds on utilization can be provided [24, 8]:

$$U \leq N(2^{\frac{1}{N}} - 1)$$

and

$$\prod_{1 \leq i \leq N} (U_i + 1) \leq 2$$

but these conditions are only sufficient.

Necessary and sufficient feasibility conditions exists for both DM and EDF when deadlines are less than or equal to the periods and the task set is synchronous [4, 1]. The EDF schedulability test, known as *processor demand criterion*, will be analyzed in Chapter 3. While these tests are clearly more comprehensive than the bounds proposed before, they are also

more extensive in terms of computation time. The computation complexity of a schedulability test is usually computed as a function of the number of tasks. The sufficient and necessary tests for both DM and EDF have *pseudo-polynomial* complexity, meaning that they have complexity polynomial in the number of tasks but also proportional to some task parameters. Intuitively, a pseudo-polynomial complexity is somehow worse than a polynomial one but better than an exponential one.

Scheduling is much more complex when tasks are allowed to share resources. In fact, if plain EDF or RM/DM are applied, a higher priority task can be blocked indefinitely by lower priority tasks. In order to provide real-time guarantees, the scheduling algorithm must be modified to include a *resource access protocol*. The protocol usually gives an upper bound B_i, called the *maximum blocking time*, to the time a task τ_i can be blocked waiting for lower priority tasks. Many different resource synchronization protocols have been proposed for both EDF and RM/DM [2, 12, 19, 29].

Given a maximum blocking time B_i for each task τ_i, the previous schedulability tests can be updated to include the effect of blocking times. The feasibility condition based on utilization for EDF becomes:

$$\forall 1 \leq i \leq N, \sum_{1 \leq j \leq i} U_j + \frac{B_i}{T_i} \leq 1$$

where tasks are assumed to be ordered by increasing relative deadline. The same holds for RM, except that the bound is not 1 but $i(2^{\frac{1}{i}} - 1)$. In Chapter 4 we will see how the processor demand criterion can be applied to the case of resource usage.

In general, DPS offers better results over FPS in term of feasibility. The main drawback of dynamic priority scheduling is in its increased conceptual complexity over fixed priority scheduling. This is the main reasons why most if not all commercially available real-time systems today implement only FPS. Secondary DPS problems include a slightly increased scheduling overhead and worse predictability in case of overloading (an overload occurs if for some reason a task executes longer than its worst-case computation time). However, dynamic priority scheduling is progressively gaining importance and most new research operating systems typically offer a choice between RM/DM and EDF scheduling.

In the remainder of this work, EDF is assumed as the scheduling algorithm.

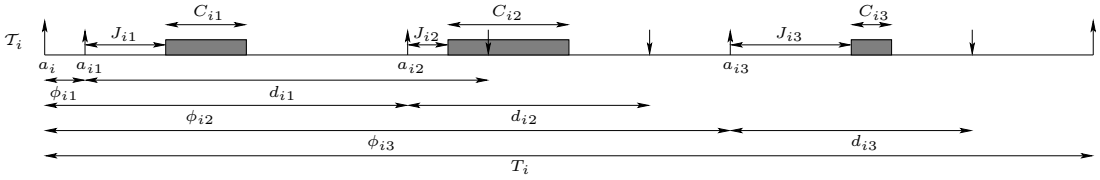

Figure 2.1: Transaction model.

2.3 Transaction model

In this section we introduce a general transaction model that will be used in Chapter 7 and in Chapters 8 and 9 with small changes.

We will consider a real-time system made of M sets of periodic tasks that we will name *transactions*. Each transaction $\mathcal{T}_i$ is characterized by period T_i and offset ϕ_i such that the k^{th} instance of each transaction is activated at time $a_i^k = \phi_i + kT_i$. Furthermore, each transaction $\mathcal{T}_i$ is composed of N_i tasks $\tau_{i1}, \ldots, \tau_{iN_i}$. Each task τ_{ij} is characterized by an offset ϕ_{ij}, a computation time C_{ij}, a relative deadline d_{ij} and a release jitter J_{ij}. The k^{th} instance (that is, the k^{th} job) of task τ_{ij}, that we will denote τ_{ij}^k, has an activation time $a_{ij}^k = a_i^k + \phi_{ij}$, but its release time can be delayed up to a maximum release jitter J_{ij}. The job takes up to C_{ij} units of computation times to be completed and must be finished before its absolute deadline $a_{ij}^k + d_{ij}$. We will further define D_{ij} to be the global relative deadline of task τ_{ij}, that is the deadline relative to the activation time of transaction $\mathcal{T}_i$: $D_{ij} = d_{ij} + \phi_{ij}$. We will assume that tasks are statically assigned to be scheduled on P different processors: in particular, tasks pertaining to the same transaction may be executed on different processors. Figure 2.1 shows the model for a transaction $\mathcal{T}_i$.

The worst-case relative response time r_{ij} of a task is the greatest difference between the finishing time of a job τ_{ij}^k and its activation time a_{ij}^k. The worst-case global response time R_{ij} is the greatest difference between the finishing time of a job of τ_{ij} and the activation time of its transaction, thus $R_{ij} = r_{ij} + \phi_{ij}$. Note that a real-time transaction system is feasible if and only if for all tasks of all transactions, the worst-case relative response time is less than or equal to the relative deadline.

Tasks are allowed to share resources in mutually exclusive way using critical section as addressed in Section 2.1.1. Each critical section ξ_{ijk} is described by a 3-ple $(\rho_{ijk}, \phi_{ijk}, C_{ijk})$, where:

1. ρ_{ijk} is the resource being accessed;
2. ϕ_{ijk} is the earliest time, relative to the activation time of τ_{ijk}, that the task can enter ξ_{ijk};
3. C_{ijk} is the worst-case computation time of the critical section.

The effect of resource usage is that a task running on one processor can be blocked by both lower priority tasks running on the same processor and by tasks running on a different processor; the maximum blocking time for task τ_{ij} is denoted B_{ij}.

Chapter 3

Feasibility analysis for asynchronous task sets

3.1 Introduction

As we said in the previous Chapter 2, in single processor systems the Earliest Deadline First scheduling algorithm is optimal [13], in the sense that if a task set is feasible, then it is schedulable by EDF. Therefore, the feasibility problem on single processor systems can be reduced to the problem of testing the schedulability with EDF.

The feasibility problem for a set of independent periodic tasks to be scheduled on a single processor has been proven to be co-NP-complete in the strong sense [21, 5]. Leung and Merril [21] proved that it is necessary to analyze all deadlines from 0 to $\Phi + 2H$. Baruah et al. [5] proved that, when the system utilization U is strictly less than 1, the Leung and Merril's condition is also sufficient.

Under certain assumption, the problem becomes more tractable. For example, if deadlines are equal to period, a simple polynomial test has been proposed by Liu and Layland in their seminal work [24]. If the deadlines are less than or equal to the periods and the task set is *synchronous* (i.e. all tasks have initial offset equal to 0), then a pseudo-polynomial test has been proposed by Baruah et al. [4, 5].

In the case of asynchronous periodic task sets, any necessary and sufficient feasibility test requires an exponential time to run. However, it is possible to obtain a sufficient test by ignoring the offsets and considering the task set as synchronous. Baruah et al. [5] showed that, given an asyn-

chronous periodic task set $\mathcal{T}$, if the corresponding synchronous task set $\mathcal{T}'$ (obtained by considering all offsets equal to 0) is feasible, then $\mathcal{T}$ is feasible too. However, if $\mathcal{T}'$ is not feasible, no definitive answer can be given on $\mathcal{T}$. In some case this sufficient test is quite pessimistic, as we will show in Section 3.5.

The basic idea behind Baruah's result is based on the concept of *busy period*. A busy period is an interval of time where the processor is never idle. If all tasks start synchronously at the same time t, the first deadline miss (if any) must happen in the longest busy period starting from t. Unfortunately, when tasks have offsets, it may not be possible for them to start at the same time. Hence, in case of asynchronous task sets, we do not know where the deadline miss might happen in the schedule.

In this chapter, a new sufficient pseudo-polynomial feasibility test for asynchronous task set is proposed. Our idea is based on the observation that the patterns of arrivals of the tasks depend both on the offsets and on the periods of the tasks. By computing the minimum possible distance between the arrival times of any two tasks, we are able to select a small group of critical arrival patterns that generate the worst-case busy period. Our arrival patterns are pessimistic, in the sense that some of these patterns may not be possible in the schedule. Therefore, our test is only sufficient. However, experiments show that our test greatly reduces pessimism with respect to previous sufficient tests.

3.1.1 Motivation

The problem of feasibility analysis of asynchronous task sets can be found in many practical applications. For example, in distributed systems a *transaction* consists of a chain of tasks that must execute one after the other and each task can be allocated to a different processor. A transaction is usually modelled as a set of tasks with offsets, such that the first task in the chain has offset 0, the second task has an offset equal to the minimal response time of the first task, and so on. Furthermore, each task is assigned a non-zero release jitter. In this way, the problem of feasibility analysis of the entire system is divided into the problem of testing the feasibility on each node. The holistic analysis [34, 31] iteratively computes the worst-case response time of each task and updates the start time jitter of the next task in the chain, until the method converges to a result. The methodology has been recently extended by Palencia and González Harbour [26], and will be further explored in Chapter 7 and 8.

Another important field of application is concerned with the problem of minimizing the *output jitter* of a set of periodic tasks. The output jitter is defined as the distance between the response time of two consecutive instances of a periodic task. Minimizing the output jitter is a very important issue in control systems. Baruah et al. [3] presented a method for reducing the output jitter consisting in reducing as much as it is possible the relative deadlines of the tasks without violating the feasibility of the task set. However, tasks are considered synchronous. In Chapter 6 we will extend the method to asynchronous task sets.

3.2 System model

The simple task model introduced in Section 2.1 is used in this chapter. Task set are supposed to be complete with no release jitter and deadlines less than or equal to the periods; no resource constraint is considered. The analysis proposed in this chapter will be extended to the case of resource usage in the following Chapter 4.

3.3 Feasibility analysis

In this section, we will first show the fundamental results for the problem of feasibility analysis of periodic task sets on single processor systems. Then we present our idea and prove it correct.

Our analysis is based on the *processor demand criterion* [5, 9]. The *processor demand function* is defined as

$$df(t_1, t_2) = \sum_{i=1}^{N} \eta_i(t_1, t_2) C_i.$$

It is the amount of time demanded by the tasks in interval $[t_1, t_2)$ that the processor must execute to ensure that no task misses its deadline. Intuitively, the following is a necessary condition for feasibility:

$$\forall\, 0 \leq t_1 < t_2 : df(t_1, t_2) \leq t_2 - t_1.$$

In plain words, the amount of time demanded by the task set in any interval must never be larger than the length of the interval.

Now, we report two fundamental results on the schedulability analysis of a periodic task set with EDF. The proofs of these results are not the original

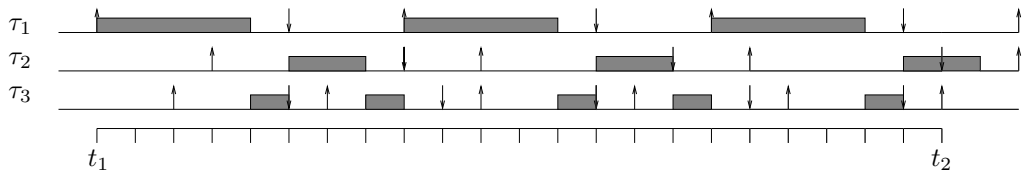

Figure 3.1: Example of busy period.

ones. They have been rewritten for didactic purposes. In fact, by following the proofs, the reader will understand the basic mechanism underlying our methodology.

We will now start by reproducing the following important lemma.

Lemma 1 ([5]) *Task set $\mathcal{T}$ is feasible on a single processor if and only if:*

1. $U \leq 1$, *and*
2. $\forall\, 0 \leq t_1 < t_2 \leq \Phi + 2H : df(t_1, t_2) \leq t_2 - t_1$.

Proof.

Both conditions are clearly necessary. By contradiction. Suppose both conditions hold but $\mathcal{T}$ is not feasible. Consider the schedule generated by EDF. It can be proven [5, 21] that since $U \leq 1$ and the task set is not feasible, some deadline in $(0, \Phi + 2H]$ is missed. Let t_2 be the first instant at which a deadline is missed, and let t_1 be the last instant prior to t_2 such that either no jobs or a job with deadline greater than t_2 is scheduled at $t_1 - 1$. By choice of t_1, it follows that $[t_1, t_2)$ is a busy period and all jobs that are scheduled in $[t_1, t_2)$ have arrival times and deadlines in $[t_1, t_2]$ (see Figure 3.1). It also follows that at least one job with deadline no later than t_2 must be released exactly at t_1, otherwise either a job with deadline greater than t_2 or no job would be scheduled at t_1. Since there is no idle time in $[t_1, t_2)$ and the deadline at t_2 is missed, the amount of work to be done in $[t_1, t_2)$ exceeds the length of the interval. By definition of df, it follows that $df(t_1, t_2) > t_2 - t_1$, which contradicts condition 2. □

By looking at the proof, it follows that it is sufficient to check the values of $df(t_1, t_2)$ for all times t_1 that corresponds to the release time of some job. In the same way, we can check only those t_2 that correspond to the absolute deadline of some job.

We will now prove that for a synchronous task set the first deadline miss, if any, is found in the longest busy period starting from $t_1 = 0$. Thus, it

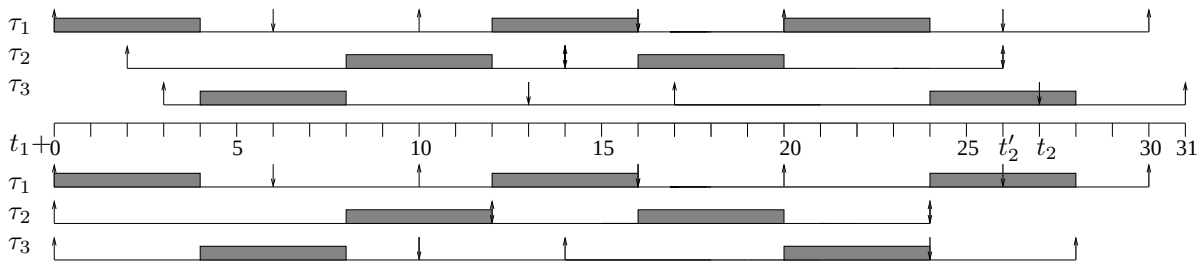

Figure 3.2: Example synchronous task set

suffices to check all deadlines from 0 to the first idle time. The idea is that, given any busy period starting at t_1, we can always produce a "worst-case" busy period "pulling back" the release times of all tasks that are not released at t_1, so that all tasks are released at the same time. Figure 3.2 shows the idea for a task set of $N = 3$ tasks with $\tau_1(C = 4, D = 6, T = 10), \tau_2(C = 4, D = 12, T = 12), \tau_3(C = 4, D = 10, T = 14)$ where task τ_1 is released at t_1. In the lower part of the figure, we show the situation where tasks τ_2 and τ_3 are "pulled back" until their first release time coincides with t_1.

Theorem 1 ([5]) *A synchronous task set $\mathcal{T}$ is feasible on a single processor if and only if:*

$$\forall L \leq L^\star, df(0, L) \leq L$$

where L is an absolute deadline and $L^\star$ is the first idle time in the schedule.

Proof. The condition is clearly necessary. By contradiction. Consider the schedule generated by EDF. Suppose that a deadline is missed, and let $[t_1, t_2)$ be a busy period as in the previous lemma. We already proved that there is at least one task that is released exactly at t_1. Let τ_i be one such task, so that $t_1 = a_{im}$ for some m, and τ_k be the task whose deadline d_{kp} is not met (note that it could be $i = k$). By following the same reasoning as in Lemma 1, we obtain $df(t_1, t_2) > t_2 - t_1$.

Now consider a task τ_j, $j \neq i, k$, and suppose that the first release time of a job of τ_j is $a_{jl} > t_1$. The new schedule generated by "pulling back" all releases of task τ_j of $a_{jl} - t_1$ is still unfeasible. In fact, since all absolute deadlines of task τ_j are now located earlier, the number of jobs of τ_j in $[t_1, t_2]$ could be increased: $\eta'_j(t_1, t_2) \geq \eta_j(t_1, t_2)$. Thus $\sum_{i=1, i\neq j}^{N} \eta_i(t_1, t_2)C_i + \eta'_j(t_1, t_2)C_j \geq \sum_{i=1}^{N} \eta_i(t_1, t_2)C_i > t_2 - t_1$ (see task τ_2 in Figure 3.2). Now consider task τ_k, and suppose that $k \neq i$. Let a_{kl} be the first release time

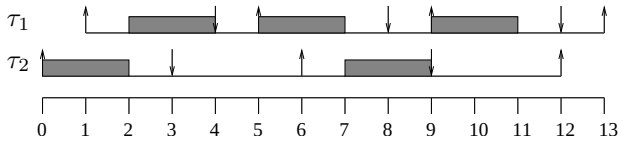

Figure 3.3: Example task set, $\tau_1(\phi = 1, C = 2, D = 3, T = 4), \tau_2(\phi = 0, C = 2, D = 3, T = 6)$

of τ_k after t_1. By moving all releases back of $a_{kl} - t_1$, we also move back its deadlines. Let $d'_{kp} \leq d_{kp}$ be the new deadline. We shall consider two possible cases. First, suppose that for each deadline $d_{jq} \leq d_{kp}, j \neq k$, it still holds $d_{jq} \leq d'_{kp}$. Then $df(t_1, t'_2 = d'_{kp}) = df(t_1, t_2) > t_2 - t_1 > t'_2 - t_1$ and the new task set is not feasible. Second, suppose that d_{jq} is the largest deadline in $[t_1, t_2]$ such that $d'_{kp} < d_{jq} \leq d_{kp}$. Consider the new busy period $[t_1, t'_2 = d_{jq})$. Then $df(t_1, t'_2) = df(t_1, t_2)$, but we obtain $t'_2 \leq t_2$ and thus the new task set is not feasible (see Figure 3.2, where $k = 3$ and $j = 1$).

Therefore, by moving back all tasks such that their first release time is at t_1 we obtain an unfeasible schedule where all tasks are released at t_1. Thus if a deadline is not met inside any busy period, then a deadline must not be met inside the busy period starting at 0. Since this contradicts the hypothesis, the theorem holds. □

Baruah et al. [5] showed that for $U < 1$ the analysis has complexity $O\Big(N \frac{U}{1-U} \max_{i=1}^{N}\{T_i - D_i\}\Big)$.

The previous theorem does not hold in the case of an asynchronous task set. It still gives a sufficient condition, in the sense that if the hypothesis holds for the corresponding synchronous task set, than the original asynchronous task set is feasible. However the condition is no longer necessary. Consider the feasible task set in Figure 3.3. It is easy to see that no instant t_1 exists such that both tasks are released simultaneously. We can still use a pessimistic analysis by considering the corresponding synchronous task set, but in the case of Figure 3.3 this wouldn't work since it can be easily seen that the corresponding synchronous task set is not feasible. Checking all busy periods in $[0, \Phi + 2H]$ is possible but would imply an exponential complexity.

Our main idea is as follow. Since there is always an initial task that is released at t_1, we build a new task set $\mathcal{T}'_i$ for each possible initial task $\tau_i, 1 \leq i \leq N$. Since τ_i is released at the beginning of the busy period, we fix $\phi'_i = 0$ in $\mathcal{T}'_i$ and check the busy period starting from 0 instead of t_1. We

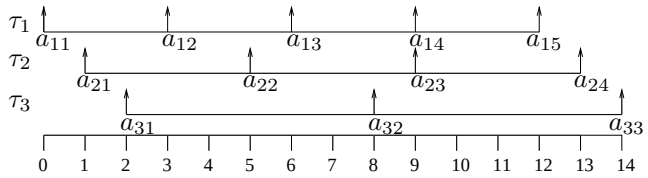

Figure 3.4: Example task set, $\tau_1(\phi = 0, T = 3), \tau_2(\phi = 1, T = 4), \tau_3(\phi = 2, T = 6)$

can then "pull back" each other task τ_j by setting ϕ'_j to the minimum time distance between any activation of τ_i and the successive activation of τ_j in the original task set $\mathcal{T}$. We will use the following Lemma:

Lemma 2 *Given two tasks τ_i and τ_j, the minimum time distance between any release time of task τ_i and the successive release time of task τ_j is equal to:*

$$\Delta_{ij} = \phi_j - \phi_i + \left\lceil \frac{\phi_i - \phi_j}{\gcd(T_j, T_i)} \right\rceil \gcd(T_j, T_i)$$

Proof. Note that for each possible job τ_{im} and τ_{jl}, $a_{jl} - a_{im} = \phi_j - \phi_i + lT_j - mT_i$. Thus $\forall l \geq 0, \forall m \geq 0, \exists K \in \mathbb{Z}, a_{jl} - a_{im} = \phi_j - \phi_i + K \gcd(T_j, T_i)$. By imposing $a_{jl} \geq a_{im}$, we obtain $K \geq \left\lceil \frac{\phi_i - \phi_j}{\gcd(T_j, T_i)} \right\rceil$. By simple substitution, we obtain the lemma. □

Definition 1 *Given task set $\mathcal{T}$, $\mathcal{T}'_i$ is the task set with the same tasks as $\mathcal{T}$ but with offsets:*

$$\begin{aligned} \phi'_i &= 0 \\ \phi'_j &= \Delta_{ij} \qquad \forall j \neq i, 1 \leq j \leq N \end{aligned}$$

Consider the example of Figure 3.4. By setting $i = 1$ we obtain $\phi'_1 = 0, \phi'_2 = a_{23} - a_{14} = 0$ and $\phi'_3 = a_{31} - a_{11} = 2$.

We will now prove that, to assess the feasibility of $\mathcal{T}$, it suffices to check that, for every task set $\mathcal{T}'_i$, all deadlines are met inside the busy period starting from time 0.

Theorem 2 *Given task set $\mathcal{T}$ with $U \leq 1$, scheduled on a single processor, if $\forall\ 1 \leq i \leq N$ all deadlines in task set $\mathcal{T}'_i$ are met until the first idle time, then $\mathcal{T}$ is feasible.*

Proof. By contradiction. Consider the schedule generated by EDF. Suppose that a deadline is not met for task set $\mathcal{T}$, and let $[t1, t2)$ be the busy period as defined in Lemma 1. We already proved that there is at least one task that is released at t_1, let it be τ_i. From Lemma 2, it follows that for every τ_j, $j \neq i$, the successive release time is $a_{jl} \geq t_1 + \Delta_{ij}$. By following the same reasoning as in Theorem 1, we can "pull back" every task so that its first release time coincides with its minimum distance from t_1, and the resulting schedule is still unfeasible. Let $\sigma_i(t)$ be the new resulting schedule.

Now, observe that, from t_1 on, the new schedule $\sigma_i(t)$ is coincident with the schedule $\sigma'_i(t)$ generated by task set $\mathcal{T}'_i$ from time 0: $\forall t \geq t_1 : \sigma_i(t) = \sigma'_i(t - t_1)$. Therefore, there is a deadline miss in the first busy period in the schedule generated by $\mathcal{T}'_i$, against the hypothesis. Hence, the theorem follows. □

Note that Theorem 2 gives us a less pessimistic feasibility condition that Theorem 1. As an example, consider the task set in Figure 3.3. According to Theorem 2 the task set is feasible, while Theorem 1 gives no result.

However Theorem 2 gives only a sufficient condition. For example, consider the following task set: $\tau_1(\phi = 0, C = 1, D = 2, T = 5), \tau_2(\phi = 1, C = 1, D = 2, T = 4), \tau_3(\phi = 2, C = 1, D = 2, T = 6)$. By analyzing the schedule, it can be seen that it is feasible, but Theorem 2 fails to give any result. The reason can be easily explained. When we "pull back" the tasks to their minimum distance from τ_i, we are not considering the cross relations between them. In other words, it may not be possible that the pattern of release times analyzed with Theorem 2 are found in the original schedule of $\mathcal{T}$. We are considering only N patterns, but they are pessimistic.

In order to reduce the pessimism in the analysis, we can generalize Theorem 2 in the following way. Instead of fixing just the initial task τ_i, we can also fix the position of other tasks with respect to τ_i and then minimize the offsets of all the remaining tasks with respect to the fixed ones. The following lemma provides more insight into the matter.

Lemma 3 *The time distance between any release time a_{il} of task τ_i and the successive release time a_{jp} of task τ_j assumes values inside the following set:*

$$\left\{ \Delta_{ij}(k) \mid \forall\, 0 \leq k < \frac{T_j}{\gcd(T_i, T_j)} \right\}$$

where

$$\Delta_{ij}(k) = \left\lceil \frac{\phi_i + kT_i - \phi_j}{T_j} \right\rceil T_j - (\phi_i + kT_i - \phi_j)$$

Proof. Consider release time a_{il} and let a_{jp} be the first successive release time of task τ_j. Since a_{jp} is greater than or equal to a_{il}, it must be $\phi_j + pT_j \geq \phi_i + lT_i$. But since a_{jp} is the first such release, we obtain $p = \left\lceil \frac{\phi_i + lT_i - \phi_j}{T_j} \right\rceil$ and $a_{jp} - a_{il} = \left\lceil \frac{\phi_i + lT_i - \phi_j}{T_i} \right\rceil T_j - (\phi_i + lT_i - \phi_j)$. To end the proof it suffices to note that the value of $a_{jp} - a_{il}$ as a function of l is periodic of period $\frac{T_j}{\gcd(T_i,T_j)}$. In fact: $\left\lceil \frac{\phi_i + \left(l + K\frac{T_i}{\gcd(T_i,T_j)}\right)T_i - \phi_j}{T_j} \right\rceil T_j - \left(\phi_i + \left(l + K\frac{T_j}{\gcd(T_i,T_j)}\right)T_i\right) + \phi_j = \left\lceil \frac{\phi_i + lT_i + K\frac{T_jT_i}{\gcd(T_i,T_j)} - \phi_j}{T_j} \right\rceil T_j - \left(\phi_i + lT_i + K\frac{T_jT_i}{\gcd(T_i,T_j)}\right) + \phi_j = a_{jp} - a_{il}$. □

Therefore, after fixing the first task τ_i we can fix another task τ_j to one of the values of Lemma 3. Now, if we want to fix a third task, we must be careful to select a time instant that is *compatible* with the release times of both τ_i and τ_j. The basic idea, explained in the following lemma, is to consider τ_i and τ_j as a single task of period $\mathrm{lcm}(T_i, T_j)$ and offset $\phi_i + kT_i$.

To generalize the notation, we denote with i_1 the index of the first task that is fixed, with i_2 the index of the second task, and so on, until i_M that denotes the index of the last task to be fixed.

In what follows, notation lcm_{ij} denotes the least common multiple among periods $T_i, \ldots, T_j$, and $\gcd_{ij}$ will be used in the same way for the greatest common divisor.

Lemma 4 *Let $a_{i_1 l_1}$ be any release time of task τ_{i_1}, and let $\Delta_{i_1 i_2}(k_1)$ be the distance between $a_{i_1 l_1}$ and the successive release time of task τ_{i_2}. The time distance between $a_{i_1 l_1}$ and the successive release time of task τ_{i_3} assumes values inside the following set:*

$$\left\{ \Delta_{i_1 i_2 i_3}(k_2) \mid \forall\, 0 \leq k_2 < \frac{T_{i_3}}{\mathrm{lcm}(T_{i_3}, \gcd(T_{i_1}, T_{i_2}))} \right\}$$

where

$$\Delta_{i_1 i_2 i_3}(k_2) = \left\lceil \frac{\phi_{i_1} + k_1 T_{i_1} + k_2 \mathrm{lcm}(T_{i_1}, T_{i_2}) - \phi_{i_3}}{T_{i_3}} \right\rceil T_{i_3} - (\phi_{i_1} + k_1 T_{i_1} + k_2 \mathrm{lcm}(T_{i_1}, T_{i_2}) - \phi_{i_3})$$

Proof. Since the time difference between $a_{i_1 l_1}$ and the successive release time $a_{i_2 l_2}$ of τ_{i_2} must be equal to $\Delta_{i_1 i_2}(k_1)$, not all values of l_1 are acceptable. Indeed, it must hold $l_1 \equiv k_1 \mathrm{mod}\left(\frac{T_{i_2}}{\gcd(T_{i_1}, T_{i_2})}\right)$.

Let $\tau_{i_1 i_2}$ be a task with period $T_{i_1 i_2} = T_{i_1} \frac{T_{i_2}}{\gcd(T_{i_1}, T_{i_2})} = \text{lcm}(T_{i_1}, T_{i_2})$ and offset $\phi_{i_1 i_2} = \phi_{i_1} + k_1 T_{i_1}$. All acceptable release times $a_{i_1 l_1}$ correspond to the release times of task $\tau_{i_1 i_2}$. We can then apply Lemma 3 to $\tau_{i_1 i_2}$ and τ_{i_3} obtaining $\Delta_{i_1 i_2 i_3}$. □

Lemma 5 *The time distance between any release time $a_{i_1 l_1}$ of task τ_{i_1} and the successive release time of task τ_{i_p}, given $a_{i_2 l_2} - a_{i_1 l_1} = \Delta_{i_1 i_2}(k_1), \ldots, a_{i_{p-1} l_{p-1}} - a_{i_1 l_1} = \Delta_{i_1 \ldots i_{p-1}}(k_{p-2})$, assumes values inside the following set:*

$$\left\{ \Delta_{i_1 \ldots i_p}(k_{p-1}) \mid \forall\ 0 \leq k_{p-1} < \frac{T_{i_p}}{\text{lcm}(T_{i_p}, \text{gcd}_{i_1 i_{p-1}})} \right\}$$

where

$$\Delta_{i_1 \ldots i_p}(k_{p-1}) = \left\lceil \frac{\phi_{i_1} + \sum_{q=1}^{p-1} k_q \text{lcm}_{i_1 i_q} - \phi_{i_p}}{T_{i_p}} \right\rceil T_{i_p} - (\phi_{i_1} + \sum_{q=1}^{p-1} k_q \text{lcm}_{i_1 i_q} - \phi_{i_p})$$

Proof. The proof can be obtained by induction, reasoning in the same way as in Lemma 4. □

Lemma 6 *The minimum time distance between any release time $a_{i_1 l_1}$ of task τ_{i_1} and the successive release time of task τ_j, given $a_{i_2 l_2} - a_{i_1 l_1} = \Delta_{i_1 i_2}(k_1), \ldots, a_{i_M l_M} - a_{i_1 l_1} = \Delta_{i_1 \ldots i_M}(k_{M-1})$, is equal to:*

$$\Delta_{i_1 \ldots i_M j} = \phi_j - \phi_{i_1} - \sum_{q=1}^{M-1} k_q \text{lcm}_{i_1 i_q} + \left\lceil \frac{\phi_{i_1} + \sum_{q=1}^{M-1} k_q \text{lcm}_{i_1 i_q} - \phi_j}{\gcd(T_j, \text{lcm}_{i_1 i_M})} \right\rceil \gcd(T_j, \text{lcm}_{i_1 i_M})$$

Proof. Reasoning in the same way as in Lemma 4 and 5, all acceptable release times $a_{i_1 l_1}$ must correspond to the release times of a task $\tau_{i_1 \ldots i_M}$ with period $T_{i_1 \ldots i_M} = \text{lcm}_{i_1 i_M}$ and offset $\phi_{i_1 \ldots i_M} = \phi_{i_1} + \sum_{q=1}^{M-1} k_q \text{lcm}_{i_1 i_q}$. We can then apply Lemma 2 to $\tau_{i_1 \ldots i_M}$ and τ_j obtaining $\Delta_{i_1 \ldots i_M j}$. □

Following the same line of reasoning as in Theorem 2, we now define task set $\mathcal{T}'_{i_1 \ldots i_M k_1 \ldots k_{M-1}}$ in the same way as $\mathcal{T}'_i$ before.

Definition 2 *Given task set $\mathcal{T}$, $\mathcal{T}'_{i_1 \ldots i_M k_1 \ldots k_{M-1}}$ is the task set with the same tasks as $\mathcal{T}$ but with offsets:*

$$\phi'_{i_1} = 0$$

$$\begin{aligned}
\vdots \quad &\vdots \quad \vdots \\
\phi'_{i_p} \quad &= \quad \Delta_{i_1 \ldots i_p}(k_{p-1}) \\
\vdots \quad &\vdots \quad \vdots \\
\phi'_{i_M} \quad &= \quad \Delta_{i_1 \ldots i_M}(k_{M-1}) \\
\phi'_j \quad &= \quad \Delta_{i_1 \ldots i_M j} \qquad \forall j \neq i_1, \ldots, i_M, 1 \leq t \leq N
\end{aligned}$$

Finally, we generalize Theorem 2 to the case of M fixed tasks.

Theorem 3 *Given task set $\mathcal{T}$ with $U \leq 1$, to be scheduled on a single processor, let M be a number of tasks, $1 \leq M < N$. If $\forall \tau_{i_1}, 1 \leq i_1 \leq N, \forall \tau_{i_2} \neq \tau_{i_1}, 1 \leq i_2 \leq N, \ldots, \forall \tau_{i_M} \neq \tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_{M-1}}, 1 \leq i_M \leq N, \forall k_1, 0 \leq k_1 < \frac{T_{i_2}}{\gcd(T_{i_1}, T_{i_2})}, \forall k_2, 0 \leq k_2 < \frac{T_{i_3}}{\gcd(T_{i_3}, \mathrm{lcm}(T_{i_1}, T_{i_2}))}, \ldots, \forall k_{M-1}, 0 \leq k_{M-1} < \frac{T_{i_M}}{\gcd(T_{i_M}, \mathrm{lcm}_{i_1 i_{M-1}})}$, all deadlines in task set $\mathcal{T}'_{i_1 \ldots i_M k_1 \ldots k_{M-1}}$ are met until the first idle time, then $\mathcal{T}$ is feasible.*

Proof. By contradiction. Consider the schedule generated by EDF. Suppose that a deadline is missed and let $[t_1, t_2)$ be the busy period as in the proof of Lemma 1. We choose i_1 such that $t_1 = a_{i_1 l_1}$ for some l_1. Next, we choose any distinct indexes $i_2, \ldots, i_M$ and any values $k_1, \ldots, k_{M-1}$ as in Lemmas 3, 4, 5, and compute the corresponding distances $\Delta_{i_1 i_2}(k_1), \ldots, \Delta_{i_1 \ldots i_M}(k_{M-1})$ from t_1. These tasks are fixed and will not be "pulled back". For the remaining tasks, we "pull back" their release times as much as it is possible: for every non-fixed task τ_j we set the distance from t_1 equal to $\Delta_{i_1 \ldots i_M j}$. By following the same reasoning as in Theorem 2, it can be easily proven that a deadline is still missed in the new generated schedule $\sigma(t)$. Note that, from t_1 on, the schedule $\sigma(t)$ is coincident with the schedule $\sigma'(t)$ generated by task set $\mathcal{T}'_{i_1 \ldots i_M k_1 \ldots k_{M-1}}$ from time 0: $\forall t \geq t_1 : \sigma(t) = \sigma'(t - t_1)$. Hence, a deadline is missed in the first busy period of $\sigma'(t)$, against the hypothesis. □

3.4 Algorithm

Theorem 2 gives us a new feasibility test for asynchronous task sets with $U < 1$ on single processor systems. For each initial task τ_i we first compute the minimal offset ϕ_j for each $j \neq i$ and the length $L^\star$ of the busy period.

```
for each i = 1 ... N
    φ_i = 0
    for each j = 1 ... N, j ≠ i
        compute φ_j (according to Lemma 2)
    next j
    L* = C_i
    while L* changes
        L* = Σ_{i=1}^{N} ⌈ (L* − (φ_i + D_i)) / T_i ⌉
    repeat
    for each deadline L ≤ L*
        if df(0, L) > L return unknown
    next L
next i
return feasible
```

Figure 3.5: Sample code, 1 fixed task

Then we check that each deadline L less than or equal to $L^\star$ is met. The pseudo code is given in Figure 3.5.

Note that the recurrence over the length of the busy period $L^\star(t+1) = \sum_{i=1}^{N} \left\lceil \frac{L^\star(t)-(\phi_i+D_i)}{T_i} \right\rceil$ converges in pseudo-polynomial time if $U < 1$ [30].

Since we must execute the algorithm for each initial task τ_i, the test has a computational complexity that is N times that of Barauh's synchronous test: $O\left(N^2 \frac{U}{1-U} \max_{i=1}^{N}\{T_i - D_i\}\right)$.

We can obtain a less pessimistic test, at the cost of an increased computation time, by using Theorem 3. As the number of fixed tasks M increases, we can expect to obtain higher percentages of feasible task sets, but the computation complexity rises quickly. If we select M fixed tasks, the complexity is bounded by $O\left(N^{M+1} \max_{i,j=1}^{N} \left\{\frac{T_i}{\gcd(T_i,T_j)}\right\}^{M-1} \frac{U}{1-U} \max_{i=1}^{N}\{T_i - D_i\}\right)$. The pseudo code for $M = 2$ is given in Figure 3.6.

for each $i_1 = 1 \dots N$
  $\phi_i = 0$
  for each $i_2 = 1 \dots N, i_2 \neq i_1$
    for each $k_1 = 0 \dots \dfrac{T_{i_2}}{\gcd(T_{i_1}, T_{i_2})} - 1$
      compute ϕ_{i_2} (according to Lemma 3)
      for each $j = 1 \dots N, j \neq i_1, i_2$
        compute ϕ_j (according to Lemma 6)
      next j
      $L^\star = C_i$
      while $L^\star$ changes
        $L^\star = \sum_{i=1}^{N} \left\lceil \dfrac{L^\star - (\phi_i + D_i)}{T_i} \right\rceil$
      repeat
      for each deadline $L \leq L^\star$
        if $df(0, L) > L$ return *unknown*
      next L
    next k_1
  next i_2
next i
return *feasible*

Figure 3.6: Sample code, 2 fixed tasks

3.5 Experimental evaluation

In this section, we evaluate the effectiveness of the proposed tests against Baruah's sufficient synchronous test described by Theorem 1 and against the exponential test executed by checking the demand function for every deadline until $\Phi + 2H$. For each experiment, we generated 2000 synthetic task sets consisting of 6, 10 and 20 tasks, respectively, and with total utilization ranging from 0.8 to 1.

Each task set was generated in the following way. First, utilizations U_i were randomly generated according to a uniform distribution, so that the total utilization summed up to the desired value. Then periods were generated uniformly between 10 and 200 and the worst-case computation time of each task was computed based on utilization and period. Finally, relative deadlines were assigned to be either between 0.3 and 0.8 times the task's period or between half period and the period, and offsets were randomly generated between 0 and the period.

We experimented with two types of task sets. In the first case, we generated the periods so that the greatest common divisor between any two tasks were a multiple of 5. In the second case, we chose the *gcd*s as multiples of 10. The basic idea is that, if the *gcd* between two periods is 1, the distance between the release times of the two tasks can assume any value, 0 included. In the limit case in which all tasks' periods are relatively prime, it is possible to show that the synchronous test is necessary and sufficient also for asynchronous task sets. Note that in the real world, a situation in which the task periods are relatively prime is not very common.

In each experiment we computed the percentage of feasible tasks using Baruah's synchronous test, our test with one, two and three fixed tasks respectively, and the exponential test. In the following we will denote these tests with *sync*, *1-fixed*, *2-fixed*, *3-fixed* and *exponential*, respectively. The results are presented in Figures 3.7, 3.8, 3.9, 3.10, 3.11, 3.12.

Figures 3.7, 3.8 and 3.9 shows the results for 6 tasks, with a minimum gcd of 10 in Figures 3.7 and 3.8 and a minimum gcd of 5 in Figure 3.9. In the first two figures, 1-fixed accepts a number of task sets up to 10% higher then the sync test. Performances are clearly lower with gcd = 5, as shown in Figure 3.9. Also note that the 2-fixed and 3-fixed tests do not achieve significant improvements over the 1-fixed test. In fact, increasing the number of fixed tasks seems to be beneficial only if the number of fixed tasks M is comparable to the number of total task N, but in that case the test obviously becomes not tractable (note that for $M = N - 1$ the test is

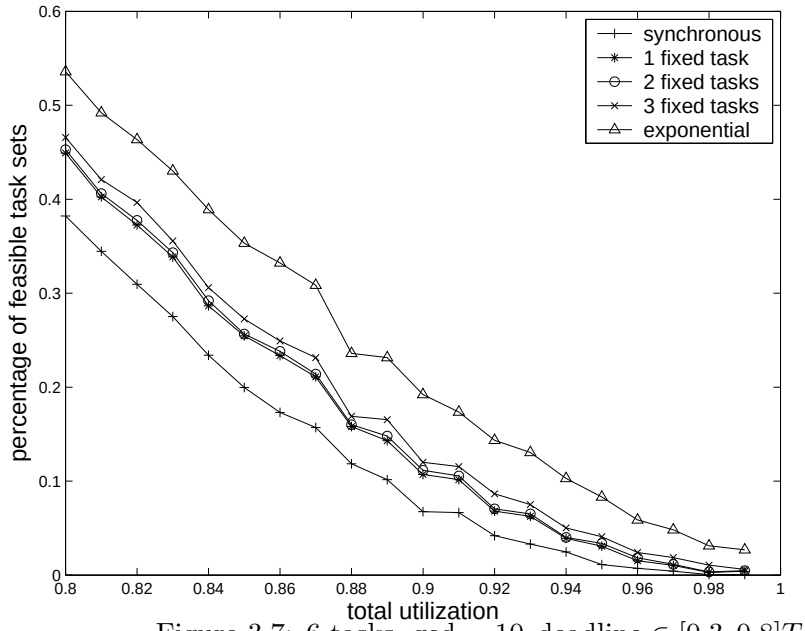

Figure 3.7: 6 tasks, gcd $= 10$, deadline $\in [0.3, 0.8]T$

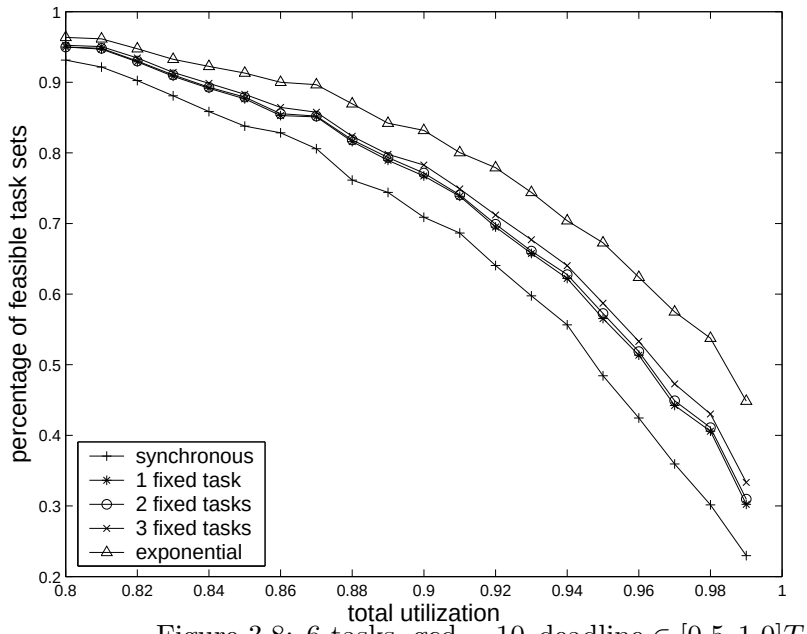

Figure 3.8: 6 tasks, gcd $= 10$, deadline $\in [0.5, 1.0]T$

	synchronous	1 fixed task	exponential
6 tasks	40	67	2233
10 tasks	122	387	461356
20 tasks	639	6341	42781200

Table 3.1: Mean number of cycles, gcd $= 10$, deadline $\in [0.3, 0.8]T$

equivalent to the exponential one). The same considerations can be applied to Figures 3.10 and 3.11 where we show the results for 10 tasks and gcd $= 10$. The 1-fixed test again achieves very good results. For high computational loads, the improvement over the sync test is more than 10%. The 2-fixed and 3-fixed tests seem even less beneficial. The same holds in Figure 3.12 for 20 tasks and gcd $= 10$.

We also computed the mean number of simulation cycles needed to test a task set. Table 3.1 shows the results for gcd $= 10$ and deadline between 0.3 and 0.8 times the period for the sync, 1-fixed and exponential test. Notice that the number of cycles needed for the 1-fixed test is only one order of magnitude greater than the sync test. As the number of tasks increases we can appreciate that the growth in the number of cycles for the 1-fixed test is still acceptable compared to the corresponding growth of the exponential test.

3.6 Conclusions

In this chapter, we presented a new sufficient feasibility test for asynchronous task sets and proved it correct. Our test tries to take into account the offsets by computing the minimum distances between the release times of any two tasks. By analyzing a reduced set of critical arrival patterns, the proposed test keeps the complexity low and reduces the pessimism of the synchronous sufficient test. We showed, with an extensive set of experiments, that our test outperforms the synchronous sufficient test.

As future work, we are planning to extend our test to the case of relative deadline greater than the period.

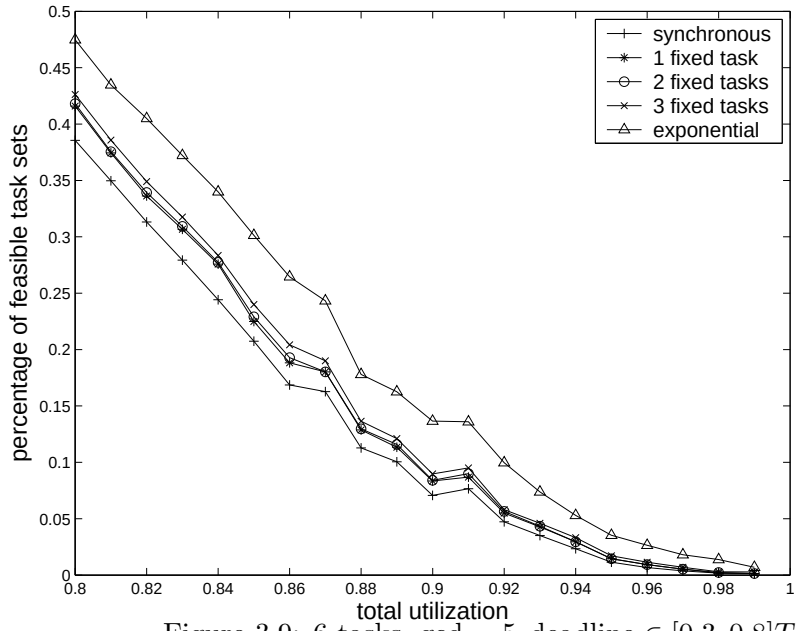

Figure 3.9: 6 tasks, $\gcd = 5, \text{deadline} \in [0.3, 0.8]T$

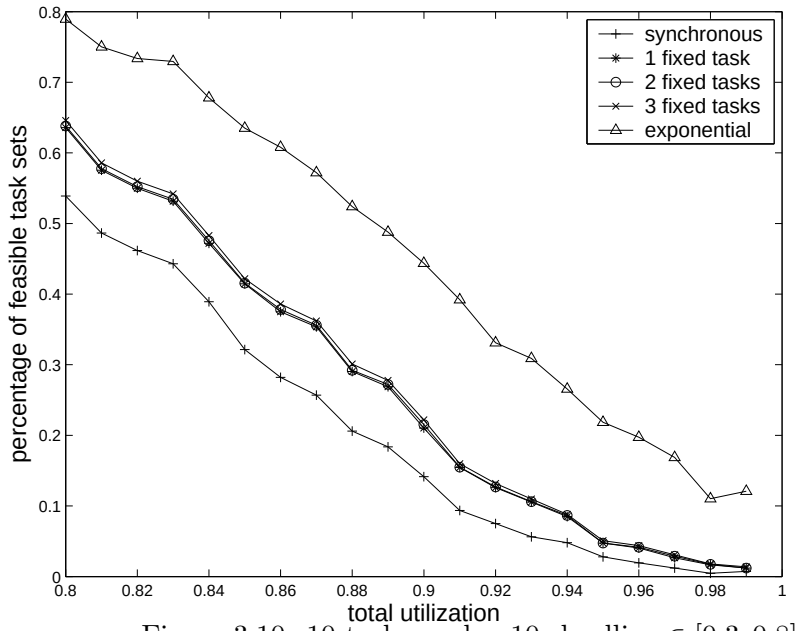

Figure 3.10: 10 tasks, $\gcd = 10, \text{deadline} \in [0.3, 0.8]T$

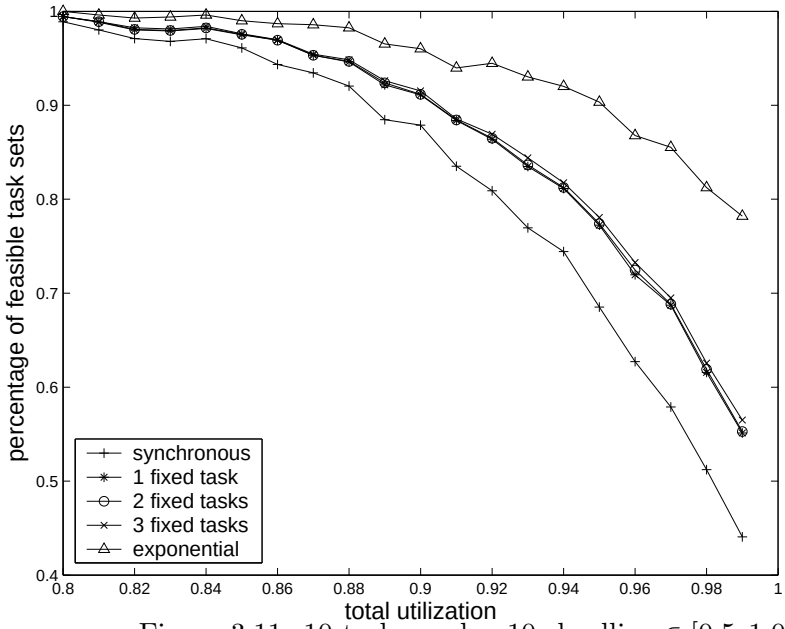

Figure 3.11: 10 tasks, gcd $= 10$, deadline $\in [0.5, 1.0]T$

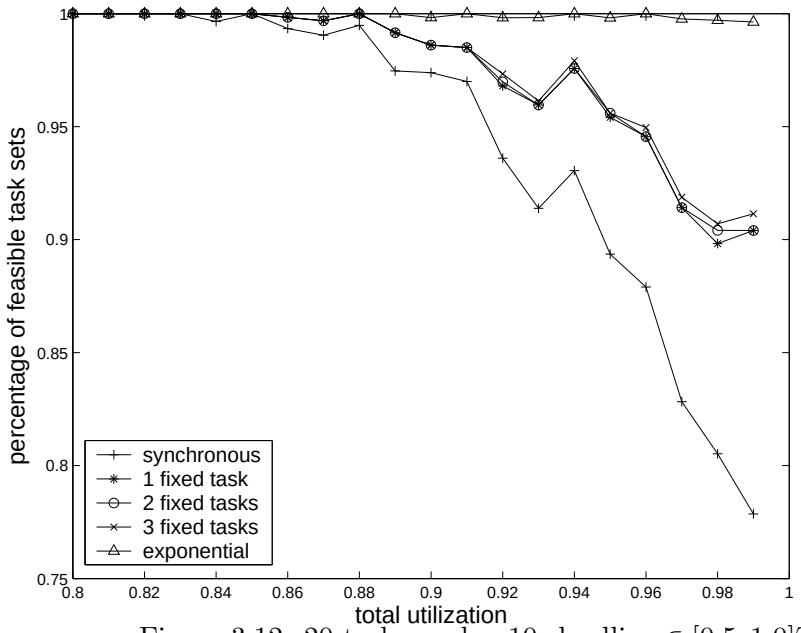

Figure 3.12: 20 tasks, gcd $= 10$, deadline $\in [0.5, 1.0]T$

Chapter 4

Resource usage extension

4.1 Introduction

In the previous Chapter 3, we developed sufficient schedulability tests for asserting the feasibility of asynchronous task sets under EDF. However, no resource constraints have been considered. In this chapter, we will extend the 1-fixed task test to cover the problem of resource usage.

The same task model as in the previous chapter is considered, but tasks are assumed to share resources as described in Section 2.1.1

Different resource access protocols have been proposed to bound the maximum blocking time of tasks due to mutual exclusion under EDF. We base our discussion on the Stack Resource Protocol proposed by Baker [2], which is the most frequently used. Under SRP, each task is assigned a static preemption level $\pi_i = \frac{1}{D_i}$. In addition, each resource ρ_k is assigned a static ceiling $\text{ceil}(\rho_k) = max_i\{\pi_i | \exists j, \rho_{ij} = \rho_k\}$. A dynamic system ceiling is then defined as follows:

$$\Pi_s(t) = max(\{\text{ceil}(\rho_k) | \rho_k \text{ is busy at time } t\} \cup 0)$$

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 manly interested in two of them:

Property 1 *Under SRP, a job can only be blocked before it starts execution; once started, it can only be preempted by higher priority jobs.*

Property 2 *A job can only be blocked be one lower priority job.*

In the following Section 4.2 we will introduce our modified test and prove it correct, while in Section 4.3 we will show how to compute a bound on the length of the busy period when resources are taken into account. The proposed analysis follows the one provided in [23].

4.2 Test extension

In order to prove our modified test, we now need to introduce some preliminary concepts.

Lemma 7 *Given two tasks τ_i and τ_j, the minimum time distance between any release time of task τ_i and the successive release time of task τ_j that is greater or equal to some value $q+1$ is equal to:*

$$\Delta_{ij}^{q} = \phi_j - \phi_i + \left\lceil \frac{\phi_i + q + 1 - \phi_j}{\gcd(T_j, T_i)} \right\rceil \gcd(T_j, T_i)$$

Proof. The proof is a simple extension of Lemma 2; it is sufficient to see that the condition on q is equivalent to considering τ_i being released q time units later. □

Definition 3 *Given task set $\mathcal{T}_i'$, we define the following dynamic preemption level:*

$$\pi_i(t) = min(\{\pi_j | \phi_j' + D_j \leq t\} \cup 2)$$

Note that instead of the constant 2 we could have used any numeric value that is strictly greater than any possible system ceiling (2 is clearly ok since no task can have a preemption level greater than 1).

Definition 4 *Given task set $\mathcal{T}_i'$, we define the following dynamic maximum blocking time:*

$$B_i(t) = max(\{C_{jk} - 1 | D_j > t + \Delta_{ji}^{\phi_{jk}} \wedge \text{ceil}(\rho_{jk}) \geq \pi_i(t)\} \cup 0)$$

We can now prove our theorem:

Theorem 4 *Given task set $\mathcal{T}$ with $U \leq 1$, if $\forall\ 1 \leq i \leq N, \forall\ L \leq L^\star$:*

$$df(0, L) + B_i(L) \leq L$$

where L is an absolute deadline of task set $\mathcal{T}_i'$ and $L^\star$ is the first idle time in the schedule of $\mathcal{T}_i'$, then $\mathcal{T}$ is feasible.

Proof. By contradiction. Consider the schedule generated by EDF. Suppose a deadline is not met for task set $\mathcal{T}$. Let t_2 be the first instant at which a deadline is not met, and let t_1 be the last instant prior to t_2 such that no job with deadline less than or equal to t_2 is active at $t_1 - 1$. Note that following this definition the busy period $[t_1, t_2)$ is the same as in Lemma 1 if there are no blocking times, since all tasks that execute inside $[t_1, t_2)$ must be released at or after t_1 and have deadlines at or before t_2. We will call $\mathcal{A}$ the set of all such tasks.

If blocking times are introduced, then it is possible for a single job τ_{jp} with deadline greater than t_2 to be executed inside $[t_1, t_2)$. For this to be possible, the job must be inside a critical section at time t_1, since it must block some other job in $\mathcal{A}$. Note that there can be only one such job; otherwise some job in $\mathcal{A}$ would be blocked by two lower priority jobs, which is impossible due to Property 2.

Now consider job τ_{jp}. Since it is inside a critical section at t_1, it must hold $a_{jp} + \phi_{jk} < t_1$ for some k, and $a_{jp} + D_j > t_2$ since its deadline is greater than t_2. Now, there is surely a task τ_i that is release at t_1. If the above conditions can hold, than they surely hold if we choose $t_1 - a_{jp} = \Delta_{ji}^{\phi_{jk}}$, since it is the minimum possible time difference. Note that in any case the maximum blocking time induced by critical section ξ_{jk} is equal to $C_{jk} - 1$, since τ_{jp} can always be delayed by other tasks so that it enters ξ_{jk} at $t_1 - 1$. τ_{jp} must also be able to block some job in $\mathcal{A}$, thus ceil(a_{jk}) must be at least equal to the minimum preemption level of tasks in $\mathcal{A}$. This proves that $Bi(t2 - t1)$ is indeed the worst-case blocking time.

To end the proof, it suffices to note that if we "pull back" all jobs in $\mathcal{A}$ as in Theorem 2, the resulting schedule is still unfeasible. In fact, the tasks in $\mathcal{A}$ do not change, while since the new deadline t_2' is less or equal than t_2, the maximum blocking time is greater or equal than before. Finally, note that a task τ_j is in $\mathcal{A}$ if and only if $\phi_j' + D_j \leq t_2'$, thus $\pi_i(t)$ is in fact the minimum preemption level of tasks in $\mathcal{A}$. □

4.3 Busy period length

Note that the recurrence over the length of the busy period for task set $\mathcal{T}_i'$:

$$L^\star = \sum_{j=1}^{N} \left\lceil \frac{L^\star - (\phi_j' + D_j)}{T_j} \right\rceil$$

is no longer valid when blocking times due are considered. In fact, a task τ_j whose offset is greater than or equal to the maximum length computed in this way, may still contribute to the busy period by adding a blocking time. The recurrence must thus be modified by adding the blocking time contribution:

$$L^\star = \sum_{j=1}^{N} \left\lceil \frac{L^\star - (\phi_j' + D_j)}{T_j} \right\rceil + B_j^\star(L^\star)$$

where

$$B_i^\star(t) = max(\{C_{jk} - 1 | \phi_j' \geq t \wedge \text{ceil}(\rho_{jk}) \geq \pi_i^\star(t)\} \cup 0)$$

and

$$\pi_i^\star(t) = min(\{\pi_j | \phi_j' < t\} \cup 2)$$

Chapter 5

Offset Space Analysis

5.1 Introduction

In Chapter 3 we introduced a new schedulability test, called 1-fixed, for asynchronous periodic task sets. Experimental evaluations showed that using 1-fixed we can achieve a substantial feasibility increase over the classic synchronous analysis. This means that asynchronous systems are in general easier to schedule than synchronous one.

In this chapter, we will suppose that there is no constraint on task offsets. This means that the system designer can set the offsets to any values he chooses. The problem is thus how to choose the offsets in an optimal way.

In Section 5.2 we will introduce the problem in a formal way and prove some basic facts, taken from [16]. Then in Sections 5.3 and 5.4 we will introduce and discuss a new offset representation, that we will call phase space representation, that offers some insight on the relation among task activation times. Finally, in Section 5.5 we propose an offset assignment that tries to make a unfeasible synchronous task set feasible.

5.2 Task model and basic facts

We will consider a periodic asynchronous task set $\mathcal{T}$ of the type defined in Section 2.1. Under this constraint, the activation patters of $\mathcal{T}$ depends on the task offsets and periods only. We can identify the task offsets using an N-tuple $\Phi = \{\phi_1, \ldots, \phi_N\}$, which we call the offset of the task set. We identify the i^{th} value of Φ (that is, ϕ_i) with Φ_i. It is not difficult to see that different offsets may lead to the same periodic behavior: thus, we are

interested in introducing the concept of *equivalence* between offsets.

Definition 5 *Given offsets Φ' and Φ'', we will say that Φ' and Φ'' are equivalent and write $\Phi' \equiv \Phi''$ iff $\exists t \in \mathbb{N}$, such that $\forall\ 1 \leq i \leq N$, the difference between the next activation of τ_i and time t in the scheduling constructed from offset Φ' is equal to the one int the scheduling constructed from Φ''_i.*

It can be easily seen that relation $\equiv$ is an equivalence relation. Thus, given offset Φ and relation $\equiv$, we can create an equivalence class $[\Phi]_{\equiv}$ as the class containing all offsets that are equivalent to Φ with respect to $\equiv$.

Theorem 5 ([16]) *We may restrict the offsets in such a way that:*

$$\forall\ 1 \leq i \leq N, \Phi_i \in [0, T_i - 1]$$

without emptying any equivalence class $[\cdot]_{\equiv}$.

Since we can restrict the offsets without losing generality, in what follows we will do so. We shall say that an offset Φ expressed in this way is defined over the *offset space* of $\mathcal{T}$.

Theorem 6 ([16]) *There are $\frac{\prod_{1 \leq i \leq N} T_i}{H}$ different equivalence classes $[\cdot]_{\equiv}$ for task set $\mathcal{T}$.*

Note that a task set is feasible for offset Φ if and only if it is feasible for each offset in $[\Phi]_{\equiv}$, so if we want to check if a task set if feasible for any offset, it suffices to check only one offset for each class $[\cdot]_{\equiv}$ (that is, $\frac{\prod_{1 \leq i \leq N} T_i}{H}$ different offsets); a formal proof is provided by Goossens in [16]. Selecting $\frac{\prod_{1 \leq i \leq N} T_i}{H}$ different offsets such that each pertains to a different class is not particularly difficult. A simple method to do it is also presented in [16]. Note, however, that the number of equivalence classes is not polynomial in the size of N, thus such a test remains untractable even if we select only one offset inside each class. Furthermore, the proposed method does not permit to understand how each equivalence class is related to the activation patterns of the task set. We will therefore introduce a characterization of equivalence classes $[\cdot]_{\equiv}$ that will help in the latter task.

Our idea is to represent each class with an M-tuple $\Gamma = \{\gamma_1, \ldots, \gamma_M\}$, such that for each $1 \leq i \leq M$ we will define a value G_i,depending only on periods $T_1, \ldots, T_N$, $0 \leq \gamma_i < G_i$. As before, we will identify γ_i with Γ_i. We

shall also say that a class Γ expressed in this way is defined over the *phase space* of task set $\mathcal{T}$, so that we will talk of a specific value Γ as a *phase*.

The way M and G_i are chosen is quite tricky and is the subject of the next section. We will also show how to compute Γ for an offset Φ, effectively performing a conversion from offset-space to phase-space, and how to compute an offset pertaining to a given phase. Finally, in section 5.4 we will prove that the phase-space representation is correct, meaning that each phase is related to one and only one offset equivalence class.

5.3 Phase-space construction and conversions

The phase-space can be constructed by following these steps:

1. Factorize each period T_i. Consider the crescent ordered set of prime factors that appear at least in two different periods, with any multiplicity. Let p_i be the i^{th} such factor, $1 \leq i \leq P$, and m_i the number of periods that it divides.

2. M is equal to $\prod_{1 \leq i \leq P}(m_i - 1)$.

3. An ordered list of values $G_1, \ldots, G_M$ can be constructed be using the following steps in order:

   (a) For each factor p_i, discard one of the periods in which the factor appears with higher multiplicity (it could be only one of course). For each other period in which the factor appears, in decreasing order of multiplicity, add to the list of values p_i power its multiplicity for that period.

   (b) Repeat for next p_i in the prime factor list.

An example will help make things clearer. Let's consider a task set $\mathcal{T}$ of $N = 4$ tasks with periods $T_1 = 9 = 3^2, T_2 = 45 = 3^2 \cdot 5, T_3 = 15 = 3 \cdot 5, T_4 = 25 = 5^2$. We get $p_1 = 3, p_2 = 5, m_1 = 3, m_2 = 3$. M is thus equal to $(3-1) \cdot (3-1) = 4$. The periods with higher multiplicity are T_1 or T_2 for $p_1 = 3$ and T_4 for $p_2 = 5$. Thus $G_1, \ldots, G_4 = 9, 3, 5, 5$.

You can immediately note that there are $9 \cdot 3 \cdot 5 \cdot 5 = 675$ different possible phases. In fact, it should be easy to see from the procedure outlined above that such number is equal to $\frac{\prod_{1 \leq i \leq N} T_i}{H}$, that we also proved equal to the number of different equivalence classes in Theorem 6.

We will now show how to obtain the phase relative to an offset Φ. In order to do so we will need to compute the minimum time distance Δ_{ij} between the successive activations of two tasks τ_i and τ_j in the scheduling constructed from offset Φ. In Section 3.3 we proved that $\Delta_{ij} = (\phi_j - \phi_i) \bmod \gcd(T_i, T_j)$. Δ_{ij}^K will be used as a shortcut to $\Delta_{ij} \bmod K$. Now supposed you already factorized periods T_i as above, you can construct an ordered list of values $\gamma_1, \ldots, \gamma_M$ following these steps in order:

1. For each factor p_i, consider the tasks whose periods have the highest multiplicity; if there is a task whose period has multiplicity higher than every other, consider it as having multiplicity equal to the second highest. Suppose the number of such tasks is k; order them from the smallest index to the largest, and let's call them $\tau_{p_{i1}}, \ldots, \tau_{p_{ik}}$. Let K be p_i power its multiplicity for the tasks.
2. Add to the γ-list $\Delta^K_{p_{i1}p_{i2}}, \Delta^K_{p_{i2}p_{i3}}, \ldots, \Delta^K_{p_{ik-1}p_{ik}}$.
3. Consider the multiplicity immediately lower, and again order the k tasks having such multiplicity using their index (note that now k can even be 1, which is not possible for the highest multiplicity). We will call these tasks $\tau_{p_{i1}}, \ldots, \tau_{p_{ik}}$ as before, but now let $\tau_{p_{i0}}$ be the last task with the immediately previously considered multiplicity. K is again p_i power its multiplicity for tasks $\tau_{p_{i1}}, \ldots, \tau_{p_{ik}}$.
4. Add to the γ-list $\Delta^K_{p_{i0}p_{i1}}, \Delta^K_{p_{i1}p_{i2}}, \ldots, \Delta^K_{p_{ik-1}p_{ik}}$.
5. Repeat from step 3 until you reach the lowest multiplicity of p_i, then proceed to the next prime factor.

You should be able to note that since $\Delta_{ij} \in [0, \gcd(T_i, T_j) - 1]$, value γ_i is effectively comprised in $[0, G_i - 1]$. Let's go back to our example. We have $\Gamma = \{\Delta^9_{12}, \Delta^3_{23}, \Delta^5_{23}, \Delta^5_{34}\}$. Also note that $\Delta^9_{12} = \Delta_{12}$ since the gcd between the periods of the tasks is 9 and $\Delta^5_{34} = \Delta_{34}$ for similar reasons. Given for example $\Phi = \{6, 37, 5, 17\}$, we obtain $\Gamma = \{4, 1, 3, 2\}$.

Since Γ represents an entire equivalence class, it should be easy to see that we can obtain H different offsets from it [16]. We will show how to obtain one; the others can simply be found by considering the scheduling constructed from the given offset, advancing time from 1 until $H - 1$ and considering at each step the distance to the next activation of each task.

In order to compute the offset, we need to briefly introduce the theory of congruence systems. A congruence system is an equivalence system of the

type:

$$\begin{array}{rcl} x & \equiv & a_1 \mathrm{mod} b_1 \\ \vdots & \vdots & \vdots \\ x & \equiv & a_K \mathrm{mod} b_K \end{array}$$

The generalized Chinese remainder theorem states that such a congruence system admits solution if and only if $\forall\ 1 \leq i, j \leq K, \gcd(b_1, b_K)$ divides $a_1 - a_K$, and that the solution is unique module $\mathrm{lcm}_{1 \leq i \leq K} b_i$.

Now suppose that you have already associated to each γ_i tasks τ_j, τ_k and power K so that $\gamma_i = \Delta_{jk}^K$ as detailed above. We will construct a congruence system for each task so that the final result of the system will hold the offset value. In order to do so it is sufficient to follow these steps from γ_1 to γ_M:

1. For each γ_i. Suppose that $\gamma_i = \Delta_{jk}^K$ in the sense detailed above. If γ_i is the first γ-value relative to a prime factor p, add $\phi_j \equiv 0 \mathrm{mod} K$ to the congruence system of τ_j. Suppose that $\bar{\phi}_j$ is the result of the congruence system constructed for τ_j so far (clearly 0 if γ_i is the first γ-value for p).

2. Add to the congruence system for τ_k the following equivalence: $\phi_k \equiv (\bar{\phi}_j + \gamma_i) \mathrm{mod} K$.

3. Repeat for next γ_i.

Note that since at most one equivalence is added to the congruence system of each task for each prime factor, the gcd between the modules is always 1 and thus the generalized Chinese remainder theorem surely holds. Also note that we only need to solve congruence systems of two equivalences, since we can simply solve the congruence system for a task once we have two equivalences and then carry just the result.

To make things clearer, we take a look again at our example. Suppose $\Gamma = \{4, 1, 3, 2\}$ as before. We will outline the steps made in computing an associated offset Φ'. Each time a new equivalence is added to a congruence system, we solve the system immediately and show its solution, separated

by a horizontal bar.

$$\begin{array}{lcll} \phi_1 & \equiv & 0(\text{mod}9) & \text{from } \gamma_1 \\ \phi_2 & \equiv & 4(\text{mod}9) & \text{from } \gamma_1 \\ \phi_3 & \equiv & 5(\text{mod}3) = 2(\text{mod}3) & \text{from } \gamma_2 \\ \phi_2 & \equiv & 0(\text{mod}5) & \text{from } \gamma_3 \\ \hline \phi_2 & \equiv & 40(\text{mod}45) & \\ \phi_3 & \equiv & 43(\text{mod}5) = 3(\text{mod}5) & \text{from } \gamma_3 \\ \hline \phi_3 & \equiv & 8(\text{mod}15) & \\ \phi_4 & \equiv & 10(\text{mod}5) = 0(\text{mod}5) & \text{from } \gamma_4 \end{array}$$

We obtain $\Phi' = \{0, 40, 8, 0\}$. It should be easy to check that converting Φ' to phase-space we indeed obtain the original $\Gamma = \{4, 1, 3, 2\}$; this should be quite obvious since the transformation from phase to offset-space maintains the Δ-values. If you wish to check that Φ' and the previous $\Phi = \{6, 37, 5, 17\}$ do pertain to the same equivalence class, please note that at time $t = 183$ in the scheduling constructed from Φ' the time difference between t and the successive activation of each task is exactly equal to Φ.

Finally, note that our way of defining the phase-space is not clearly unique, in the sense that it depends upon a chosen order for both the prime factors and the tasks. Clearly any possible order can be chosen as long as the transformations from offset-space to phase-space and vice versa are defined accordingly.

5.4 Phase-space representation correctness

In this section, we prove that the phase-space is a correct representation for equivalence classes $[\cdot]_{\equiv}$. We wish to prove that the relation between phases and equivalence classes derived from the conversion between offset to phase-space (meaning that each class is related to any phase for which a conversion exists from an offset pertaining to that class) is a bijective function. Since we showed that each offset can be converted to a phase and each phase can be converted to an offset, it follows that every phase is related to at least one class and that each class is related to at least one phase. And since the dimension of the phase-space is exactly equal to the number of different classes, either the relation is a bijective function or there is a class that is related to at least two phases and there is a phase that is related to at least two classes. We will show that each class is related to one and only one phase, thus proving the correctness of the representation.

Lemma 8 *In the relation derived from the conversion from offset to phase-space, each equivalence class is related to one and only one phase.*

Proof. Since each offset converts to one and only one phase, each class is related to at least one phase. In order to prove that a class cannot be related to more than one phase, we must show that given any two offsets Φ' and Φ'' of that class, the two offsets correspond to the same phase.

The phase-space representation associates some value Δ_{jk}^{K} to each γ_i. Now, $\Delta_{jk} = (\phi_k - \phi_j) \bmod \gcd(T_j, T_k)$ does not change between two offsets Φ' and Φ'' pertaining to the same equivalence class. Note in fact that given the definition of relation $\equiv$, Φ'' can be surely obtained from Φ' by multiple applications of the two following operations:

1. adding or subtracting 1 from all components ϕ_i of Φ';
2. adding or subtracting T_i from one component ϕ_i.

Adding or subtracting 1 from all offsets does not change $\phi_k - \phi_j$. Adding or subtracting the task period from ϕ_j or ϕ_k does not change Δ_{jk} either since its module is the gcd of both T_j and T_k.

But since Δ_{jk} does not change between Φ' and Φ'', γ_i remains the same as well for every i and thus both offsets relates to same phase. □

We have thus proved the following theorem:

Theorem 7 *The phase-space representation is a correct representation of equivalence classes* $[\cdot]_{\equiv}$.

5.5 Feasibility applications

Given a phase Γ it is not difficult to compute the minimum distance Δ_{ij} for all $1 \leq j \leq N$, after fixing an initial task i. This can be done to apply the 1-fixed schedulability analysis developed in Chapter 3. It is sufficient to apply the steps outlined in the conversion from phase-space to offsets-space, with the following differences:

1. Ignore any prime factor p_k that does not divide τ_i.
2. Treat each task with a multiplicity relative to p_k higher than that of τ_i as having multiplicity equal to that of τ_i.

3. For each prime factor, start by adding the equivalence $\phi_i \equiv 0 \mathrm{mod} K$ (where K is the prime factor power its multiplicity in T_i), than proceed as detailed before for all tasks whose index follows ϕ_i. For the tasks that precede τ_i, proceed in reverse order, taking into account that $\Delta_{ij}^K = (K - \Delta_{ji}^K) \mathrm{mod} K$.

After solving the congruence systems, it suffices to impose $\Delta_{ij} = \phi_j$ for all $1 \leq j \leq N$.

Going back to our example, let's compute $\{\Delta_{3j}\}_{1 \leq j \leq N}$ for $\Gamma = \{4, 1, 3, 2\}$.

ϕ_3	$\equiv$	$0(\mathrm{mod}3)$	
ϕ_2	$\equiv$	$2(\mathrm{mod}3)$	reversing γ_2
ϕ_1	$\equiv$	$4(\mathrm{mod}3) = 1(\mathrm{mod}3)$	reversing γ_1
ϕ_3	$\equiv$	$0(\mathrm{mod}5)$	
ϕ_3	$\equiv$	$0(\mathrm{mod}15)$	
ϕ_2	$\equiv$	$2(\mathrm{mod}5) = 3(\mathrm{mod}5)$	reversing γ_3
ϕ_2	$\equiv$	$2(\mathrm{mod}15)$	
ϕ_4	$\equiv$	$2(\mathrm{mod}5)$	from γ_4

And so we obtain $\Delta_{31} = 1, \Delta_{32} = 2, \Delta_{33} = 0, \Delta_{34} = 2$.

Note that although the Δ-values for some τ_i can be computed easily (for example for τ_1) so that we may get an easy dependence of Δ from Γ, in general the dependence is not particularly easy to visualize. The above examples shows that Δ_{32} depends both on γ_2 and on γ_3. Since we must solve a congruence system to get the result, the dependence is not particularly easier than that between the Δ-values and the offsets Φ (in this latter case difficulties arise from the presence of the module in the equation used to compute Δ).

Also note that if we want to apply the 1-fixed test, we need to check all different phases, since each phase yields different Δ-values; thus the 1-fixed analysis treats each phase pessimistically yet differently from all other phases (see also Chapter 3).

In his paper [16], Goossens proposed an heuristic, the *dissimilar offsets assignment*, that tries to choose the offsets in order to achieve feasibility. The heuristic is based on the assumption that in order to ease the scheduling of the task set we should try to make its activation patterns as dissimilar as possible in respect to the ones of its associated synchronous task set. This is done by trying to maximize both Δ_{ij} and Δ_{ji} at the same time for each i, j. However, since Goossens considers only the offset-space representation, his practical algorithm is far from ideal. Its overall result is similar to choosing

a phase $\Gamma = \{G_1/2, \ldots, G_M/2\}$. This is clearly not good in the general case since if there are at least three tasks sharing the same factor p in their period, the offset's difference between the first and the third task will be minimized with respect to p. A better heuristic should thus take into account the number of tasks sharing each factor p. We can define a new heuristic, the *improved offsets assignment*, that tries to maximize the difference between offsets in respect to each prime factor p in the task periods. The improved offsets assignment is effectively defined on the phase space, since we are really interested in choosing an offset equivalence class.

Suppose that we already associated to each γ_i tasks τ_j, τ_k and power K so that $\gamma_i = \Delta_{jk}^K$ as detailed in Section 5.3. Also suppose that for a given power K, $\gamma_i, \ldots, \gamma_{i+q-1}$ are the $\gamma - values$ relative to that K. Then:

1. if K has the greatest multiplicity for its prime factor p, use the following phase assignment: $\gamma_i = 0, \gamma_{i+1} = \frac{K}{q}, \gamma_{i+2} = 2\frac{K}{q}, \ldots, \gamma_{i+q-1} = (q-1)\frac{K}{q}$;

2. otherwise: $\gamma_i = \frac{K}{q+1}, \gamma_{i+1} = 2\frac{K}{q+1}, \ldots, \gamma_{i+q-1} = q\frac{K}{q+1}$.

In order to see the effectiveness of our heuristic, we have conducted experimental evaluations. Random task sets were generated with the same procedure as the one described in Section 3.5. Periods are chosen between 10 and 200, with a minimum greatest common divisor between any two periods equal to 10. Each task set consists of 5 tasks. Each deadline is randomly chosen either between 0.3 and 0.8 times the task's period or between half period and the period.

Figures 5.1 and 5.2 show the results in term of percentage of schedulable task sets versus utilization. In all figures, *synch* is the standard synchronous test; *random* is the 1-fixed test executed on random offset; *optimal* is the 1-fixed test executed on all possible equivalence classes (meaning that a task set is schedulable under optimal if it is schedulable for at least one offset); *dissimilar* is the 1-fixed test applied to Goossens's dissimilar offsets assignment and finally *improved* is the 1-fixed test applied to our improved offsets assignment.

As you can see, results are quite dissatisfactory. The random test performs well, being able to schedule a mean 36% of all task sets schedulable under optimal but not synch; on the contrary, our heuristic gives no better results than random, and the dissimilar offset assignment performs even worse. This basically means that both heuristic are completely useless. We could thus ask ourself if a better heuristic exists. Unfortunately, we strongly

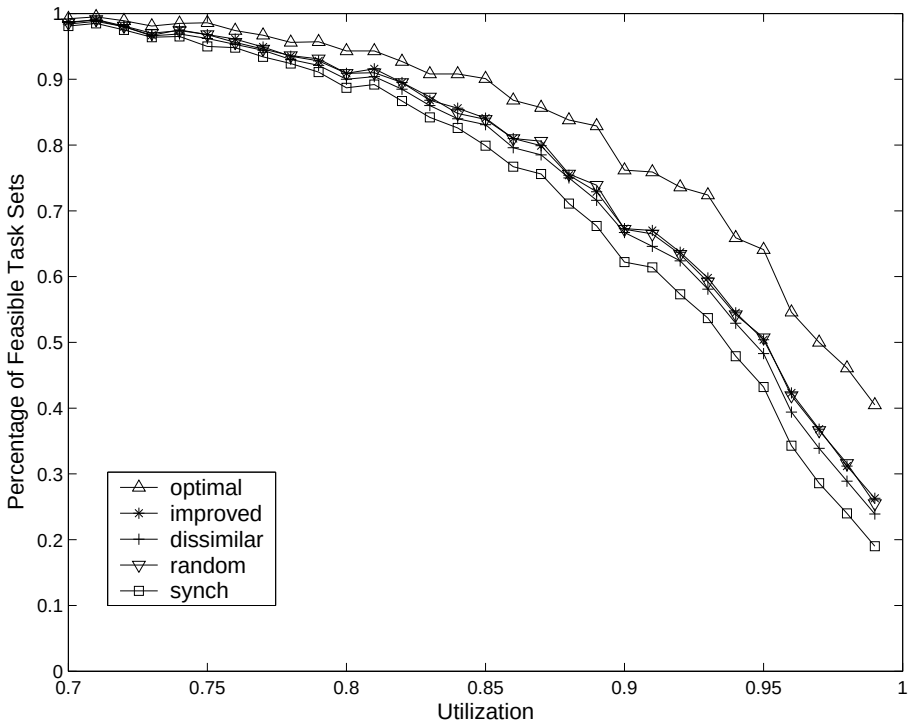

Figure 5.1: 5 tasks, deadline $\in [0.5, 1.0]T$

think that it does not. First, the basic idea beyond both the dissimilar and the improved offset assignment seems very natural and we doubt that a better idea could be exploited. Second and more important, by looking at the pattern of offsets for which a task set is schedulable, it can be seen that such offsets are arranged in very complex ways that depend on the interactions between the modules studies in Section 5.3. Intuitive heuristic are thus very likely to be doomed to failure, and finding a good polynomial algorithm is probably very difficult if not impossible.

Furthermore, just choosing the offset in a random way can be sufficient: trying one random offset we can schedule 36% of all task sets that are schedulable under optimal but not synch, trying two we schedule a mean 59%, and trying 3 we go up to 74%. The 90^{th} percentile can be reached by choosing 6 offsets and the 99^{th} by choosing 11 offsets.

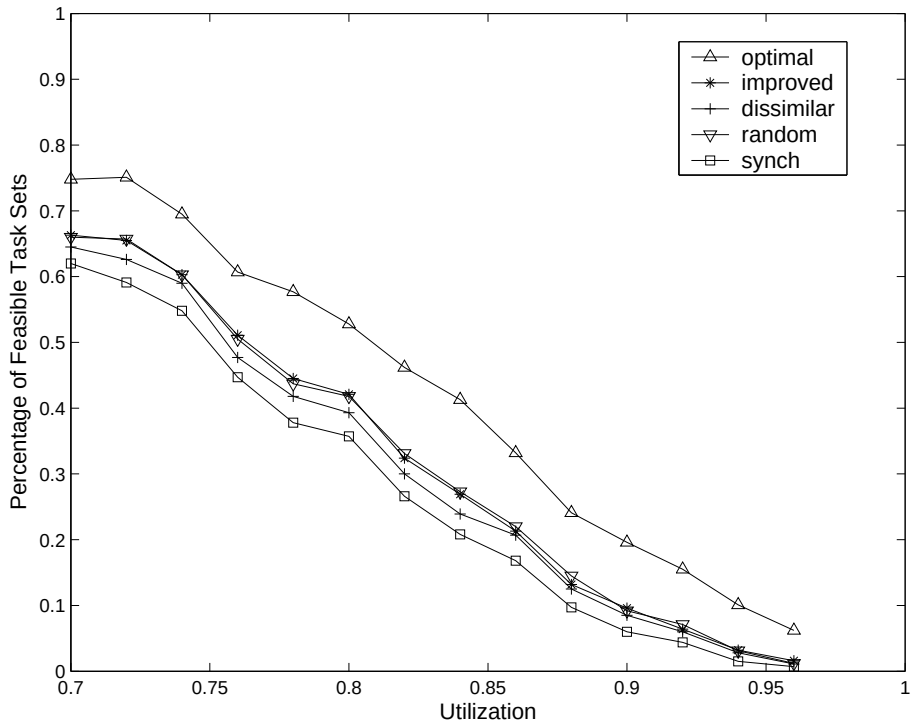

Figure 5.2: 5 tasks, deadline $\in [0.3, 0.8]T$

5.6 Conclusions

In this chapter we introduced an alternative representation of task offsets, called the phase space representation. This representation is useful since it maps each class of different offsets in one phase and vice versa. It also shows more clearly the relation between offsets and patterns of activation times, although the relation is by no way simple. We also tried to devise heuristics to choose the offsets in a nearly optimal way, but we showed that such attempt is probably fruitless. On the contrary, trying one or more random offsets can give good results.

Chapter 6

Minimization of output jitter

6.1 Introduction

In this section, we will use the 1-fixed test developed in Chapter 3 in order to address a relevant problem in the design of real-time control algorithms, that of output jitter.

Assuming that a task set is schedulable, we may be interested in adjusting our scheduling algorithm in order to optimize some secondary objective. For example, we may be interested in minimizing the number of context switches, or, as we will assume in the rest of this chapter, in minimizing the output jitter of some tasks, that is the variation in the finishing times (that we will also call inter-finishing time) for successive jobs of the same task. More formally, given a feasible schedule for a periodic task set $\mathcal{T}$ of the model defined in Section 2.1 with zero release jitters, let

$$\begin{aligned} p_i^{\min} &= \min_{k \geq 0}\{f_{ik+1} - f_{ik}\} \\ p_i^{\max} &= \max_{k \geq 0}\{f_{ik+1} - f_{ik}\} \end{aligned}$$

That is, $p_i^{\min}$ is the minimum difference between the finishing times of two successive jobs of τ_i while $p_i^{\max}$ is the maximum such difference. We can then define an *absolute jitter* for τ_i as follows:

$$\text{AbsJitter}_i = \max(p_i^{\max} - p_i, p_i - p_i^{\min})$$

The absolute jitter is thus the maximum variation of the inter-finishing time from the task's period. A measure relative to the task's period is often more

useful, thus leading to the definition of *relative jitter*:

$$\text{RelJitter}_i = \frac{\text{AbsJitter}_i}{T_i}$$

Finally, we define the relative jitter of the entire task set:

$$\text{RelJitter} = \max_{1 \leq i \leq N} \{\text{RelJitter}_i\}$$

Note that we may be interested in minimizing the output jitter not of the entire task set but of some tasks only. In this case, it is clearly sufficient to consider only the relative jitters of those tasks.

A task set is said to be *jitter-free* if RelJitter = 0. Note that under normal EDF it is extremely improbable (although not impossible) for a task set to be jitter-free. However, minimizing the output jitter in periodic task system is in a certain way a non-issue. In fact, given a feasible schedule we could obtain a virtually jitter-free system by simply postponing the execution of the last chunk of length ϵ of each task until its deadline. Assuming $\epsilon \to 0$, we can always maintain feasibility in this way.

However, the above solution is still far from ideal. Apart from its optimality, EDF possesses many other features that are desirable in real-time systems. Among them, EDF bounds the total number of context switches to a reasonable number (twice the number of tasks N), and satisfies the integral boundary constraint. Such features are clearly not preserved by the modification presented before. In general, we would like any new scheduling algorithm to preserve the properties of EDF; in particular, this means that we do not accept delaying a job ready to execute nor inserting idle time in the schedule.

Furthermore, delaying the finishing time until the deadline is unacceptable for control algorithms. Minimizing the output jitter of control tasks is particularly important since the quality of the control strongly depends on the output jitters. Unfortunately, since the quality of the control is also dependent on the finishing time of the task, delaying it is not a good idea.

In the following section 6.2 we will first recall two polynomial time bounds on the relative jitter of task sets. These bounds are taken from [3]. In the following section 6.3 we will show how we can minimize the output jitter by moving the deadlines and checking feasibility with either the synchronous or the 1-fixed test from Chapter 3. Finally, in Section 6.4 we will provide simulation results.

In all sections, we will initially suppose that all task deadlines are equal to the periods. This hypothesis usually makes sense since setting a stricter

deadline is usually done in order to reduce the output jitter of the task, and that is precisely the method we will use.

6.2 Polynomial time bounds

First of all, note that a job of τ_i can surely finish no sooner than C_i and no later than T_i. An easy bound on the relative jitter is thus:

$$\text{RelJitter}_i \leq \frac{T_i - C_i}{T_i} = 1 - U_i$$

However, this bound can be easily improved as shown in [3] to the following:

$$\text{RelJitter}_i \leq \max_i \{U - U_i\} \tag{6.1}$$

The main idea used in the proof is that the latest finishing time of a task is actually bounded by the total utilization of the system. Unfortunately, even this bound is not very accurate. We could obtain a lower bound using one of the response time analyses that we will introduce in Chapter 7, but it would require pseudo-polynomial time. Instead, we will use this bound to introduce a polynomial time minimization algorithm for output jitters.

Suppose that instead of its utilization U_i, we reserve to each task τ_i a *processor share* $\lambda_i \geq U_i$. We could then assign a stricter deadline $d'_i = \frac{C_i}{\lambda_i}$, and, supposing that $\Lambda = \sum_i \lambda_i \leq 1$, the task set will remain feasible under EDF. With the new assigned deadlines it can be proven that the bound on the relative jitter becomes:

$$\text{RelJitter}_i \leq \Lambda \frac{U_i}{\lambda_i} - U_i$$

It can be proven [3] that if we set the processor shares as follows:

$$\lambda_i = \max\left(U_i, \frac{U_i}{U_i + J}\right) \tag{6.2}$$

then, supposed that the task set is feasible, the relative jitter of the task set is bounded by J.

We can thus use a binary search to determine an accurate bound on the relative jitter, by searching the greatest processor shares that make the task set feasible. The pseudocode in Figure 6.1 shows the algorithm, where J_{init} can be the bound computed in Equation 6.1.

```
J_high ← J_init
J_mid ← 0
do
    J_mid ← (J_high + J_high) / 2
    Compute λ_i, 1 ≤ i ≤ N (according to 6.2)
    if (Σ_{i=1}^{N} λ_i ≤ 1)
        J_high ← J_mid
    else
        J_low ← J_mid
while(J_high − J_low > thres)
return J_mid
```

Figure 6.1: Sample code, polynomial time method

6.3 Pseudo-polynomial time minimization

A stricter bound on the relative jitter of the task set can be obtained by directly modifying the task deadlines and using a pseudo-polynomial analysis to test feasibility. If we set a deadline $d_i \leq T_i$ for task τ_i, the maximum finishing time of any of its job becomes d_i, and thus its maximum release jitter is now equal to:

$$\text{RelJitter}_i = \frac{d_i - C_i}{T_i} = \frac{d_i}{T_i} - U_i$$

If we assign to each task a new deadline $C_i + JT_i$ and the task set remains feasible, then the relative jitter of the task set will be at most equal to J. We can thus use a binary search algorithm like the one used in the previous section to search the minimum J, by checking the feasibility of the task set at each step using a processor demand criterion based test. If the task set is asynchronous, using the 1-fixed test instead of the synchronous one is beneficial to the jitter analysis, as we will show in the next section.

Note that instead of a feasibility analysis, we could run a response time analysis of the type discussed in Chapter 7. While this would lead to better results once all deadlines are fixed, it would also require a much more complicated study since it gives no direct requirements on how the deadlines should be modified.

6.4 Experimental evaluation

We conducted a series of experiments in order to evaluate the performance of the pseudo-polynomial time minimization in respect to the other methods introduced in the previous two sections.

Random task sets were generated with the same procedure as the one described in Section 3.5. Periods were chosen between 10 and 200, with a minimum greatest common divisor between any two periods equal to 10. Each task set consists of either 5 or 10 tasks. Deadlines are initially assumed to be equal to periods.

Figures 6.2, 6.3, 6.4 and 6.5 shows the experimental results, in term of mean relative jitter versus total utilization. Note that since deadlines are equal to periods, every randomly generated task set is feasible. Also, we can expect relative jitter to rise with the utilization, as it becomes progressively more difficult to reduce each task's output jitter as the interference from other tasks rises.

Figures 6.2, 6.3 and 6.4 shows the results for task sets composed of 10 tasks. In Figure 6.2 we tried to minimize the output jitter of all 10 tasks, in Figure 6.3 of 5 tasks out of 10, and in Figure 6.4 of just 3 of them. Finally, Figure 6.5 gives the results for task sets of 5 tasks, where we tried to minimize the jitter of all 5. In all figures, *bound* stands for the simple bound from Section 6.2, *poly* is the polynomial time minimization, and *sync* and *async* are the pseudo-polynomial time minimizations using the synchronous and 1-fixed test respectively.

Some conclusions that we are able to draw from the figures:

- Bound performance are typically quite low compared to the minimization methods, which is quite a good hint at the fact that spending time to implement methods for reducing the output jitter can be remunerative.
- In the same way, the most cost-intensive methods (sync and async) clearly outperforms poly.
- All the minimization methods performs quite better as the ratio between the number of task in the task set and the number of jitters to minimize increases. This should be no surprise since the higher the ratio, the more freedom we have to adjust the scheduling in order to decrease the relative jitter.
- The async method achieves a practically constant improvement over

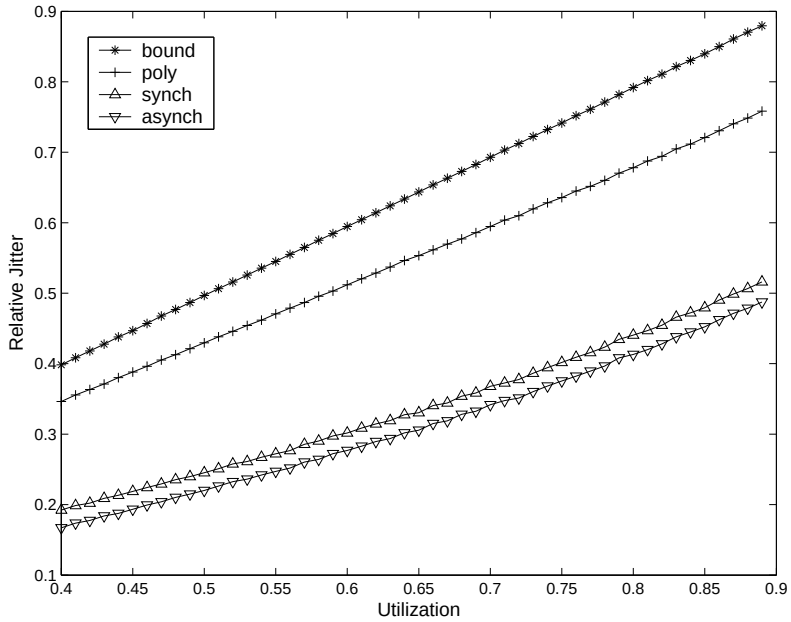

Figure 6.2: 10 tasks, 10 minimized jitters

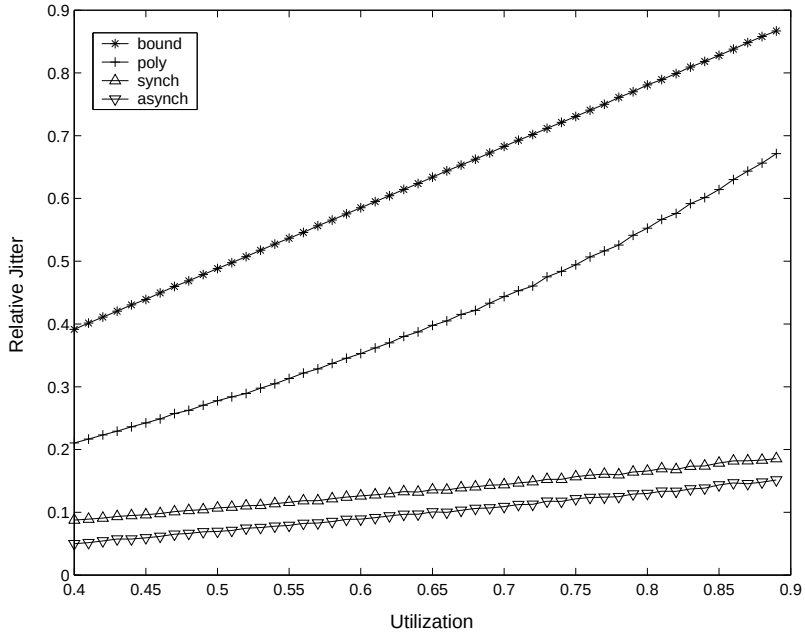

Figure 6.3: 10 tasks, 5 minimized jitters

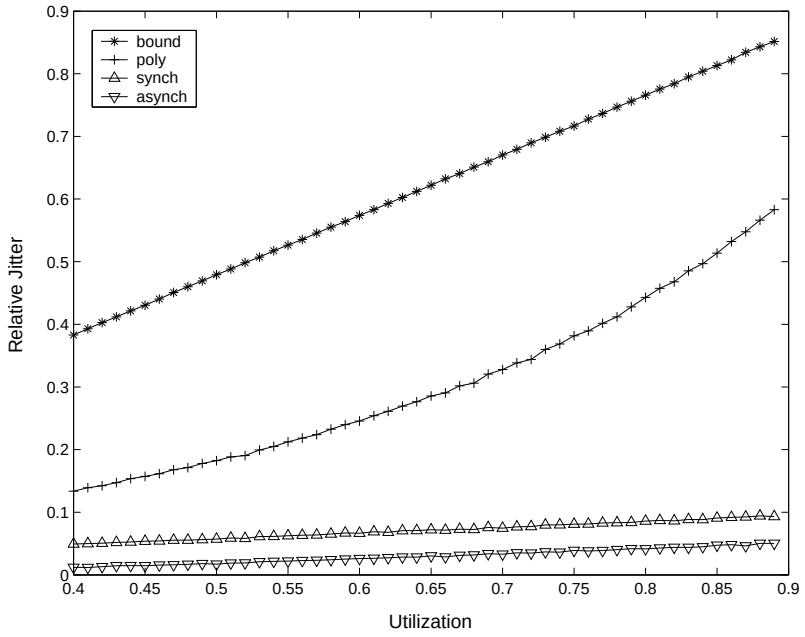

Figure 6.4: 10 tasks, 3 minimized jitters

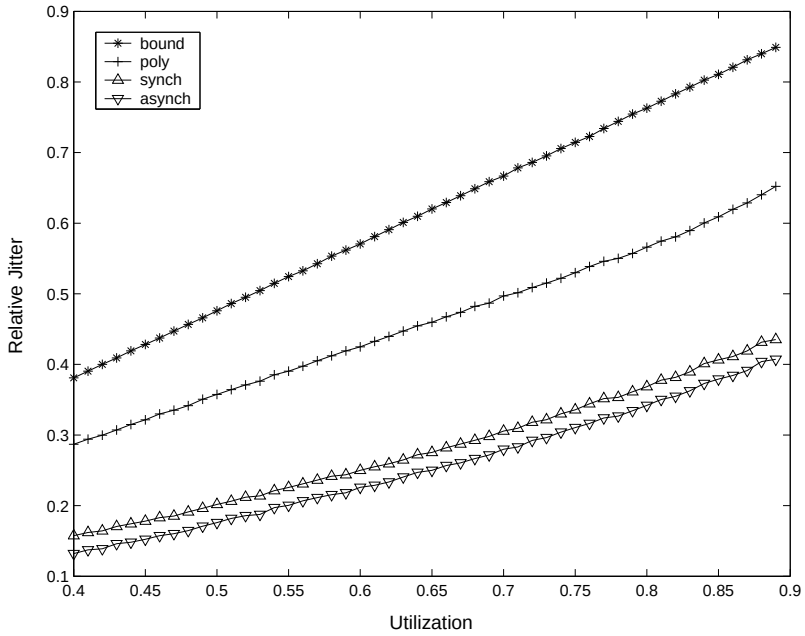

Figure 6.5: 5 tasks, 5 minimized jitters

the sync one. Unfortunately, such improvement is relatively small in respect to how 1-fixed compared to the synchronous test in Section 3.5.

6.5 Conclusions

We have briefly shown how our improved processor demand analysis for asynchronous task sets can be used to address the problem of output jitter. Such problem is particularly important in the case of control tasks, in which the output jitter directly impacts on the quality of the control. We have voluntarily kept the discussion easy, addressing the problem from a strictly scheduling-based point of view. However, a clear understanding of the control-based point of view may be necessary to better understand the problem [10] [11].

Chapter 7

Response time analysis for EDF

7.1 Introduction

In Chapter 3, we saw how initial offsets can be exploited to improve a task set's schedule. We developed a new algorithm, called 1-fixed, which constitutes a sufficient feasibility test for task sets with offset scheduled under EDF.

In this chapter, our attention will be shifted on the problem of response time computation. Spuri [30] solved the problem for the synchronous simple task model from Section 2.1. We could show how to extend his method to asynchronous task, but we are more interested in extending our technique to the transaction model described in Section 2.3. This analysis will in fact be extremely useful in the following Chapters 8 and 9.

Palencia and Gonzàlez developed a worst-case response time analysis for the transaction model in [25]; however, they considered *incomplete transactions*, in the sense that the transaction offsets are not specified. We will present a variation of their original algorithm in Section 7.2. The proposed method is actually slightly less pessimistic and faster than the one previously presented and above all way simpler to understand. In Section 7.3 we will instead introduce our new analysis that is able to take into account transaction offsets.

In what follows, we will assume for simplicity of exposition that each transaction set is scheduled on a single processor. However, results can be quickly adapted to the case of multi-processor systems. It suffices to note

that when we compute the response time of a task τ_{ij}, tasks executed on a different processor do not contribute in any way to the finishing time of the task under analysis. We can thus adopt single-processor analyses by simply considering a new set of transactions $\mathcal{T}'_1, \ldots, \mathcal{T}'_M$ composed only of the tasks of $\mathcal{T}_1, \ldots, \mathcal{T}_M$ respectively that run on the same processor as τ_{ij}.

7.2 Variation of the Palencia-Gonzàlez's method

For this method, we will assume that the transaction offsets are unknown, that is, any offset values $\phi_1, \ldots, \phi_M$ are possible, provided that $\phi_1, \ldots, \phi_M$ are natural numbers. Goossens proved that choosing a granularity for the offsets smaller than the rest of the systems does not help feasibility in any way [16]. Under this constraint, it is easy to prove the following theorem:

Theorem 8 ([26]) *The worst-case response time of a task τ_{ab} can be found in a busy period starting at some time t_0, such that for each transaction $\mathcal{T}_i, i \neq a$, there is a task τ_{ij} that is released exactly at t_0 after having experienced its maximum release jitter.*

Note that if the transaction offsets are set, than the theorem above does not hold anymore as there may not be any time t_0 in which $M - 1$ tasks are released simultaneously. Also note that releasing task τ_{ab} at t_0 may not lead to its worst-case response time. To determine the response time of τ_{ab} we need to compute the total contribution of all tasks to τ_{ab}, that is the total processor time demanded by tasks executed inside the busy period with deadlines less than or equal to that of τ_{ab}. If we move the activation patter of τ_{ab} to occur earlier (in other world, we "pull back" the activation times of τ_{ab}), we cause its deadline to be earlier too, and this may imply that some deadlines of other tasks that previously occurred before the deadline of τ_{ab} now occur after, decreasing the total contribution. We thus use the following strategy: for each possible activation time A of τ_{ab}, such that τ_{ab} is executed inside the busy period (eventually after having experienced some jitter, provided that it is less than or equal to its maximum), we compute its finishing time and thus its response time, and then we take the overall maximum. In order to reduce the number of activation times that needs to be checked, we use the following theorem:

Theorem 9 *The worst-case response time of a task τ_{ab} corresponds to the response time of one of its jobs 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 same busy period) or a job of a task of transaction $\mathcal{T}_a$ (eventually τ_{ab} itself) is released at the beginning of the busy period after having experienced maximum jitter.

Proof. Consider job τ_{ab}^k, and suppose that neither its deadline corresponds to the deadline of a job of a task of another transaction (executed inside the busy period) nor a job of a task of transaction $\mathcal{T}_a$ is released at the beginning of the busy period after having experienced maximum jitter. We can then "pull back" the activation time of τ_{ab}^k, and thus the activation time of $\mathcal{T}_a$, until one of the two above conditions becomes true, without decreasing the response time of τ_{ab}^k. In fact, note that all jobs that contributed to delaying the finishing time of τ_{ab}^k still counts, while since a_{ab}^k is pulled back as well, the response time will actually increase.

Finally, note that the above consideration may not hold if τ_{ab}^k is the first task to be released in the busy period. In this case, though, we could obtained a worst response time by "pulling back" all other transactions so that τ_{ab} is released at the beginning of the new busy period after having experienced maximum jitter. □

To compute the contribution of all tasks to the finishing time of τ_{ab}^k we use a recurrence equation over the worst-case contribution of all tasks to a busy period of length t and deadline D, where D is the deadline (relative to t_0) of τ_{ab}^k, and t is subject to the recurrence. To simplify the notation, from now on we will assume without loss of generality $t_0 = 0$.

Let's start by computing the worst-case contribution to a busy period of length t and deadline D of a transaction $\mathcal{T}_i, i \neq a$; we will call it $W_i(t, D)$. We know that a task τ_{ik} must be released at the beginning of the busy period, but unfortunately we do not know which. Trying out every possible combination for each transaction would lead to exponential complexity. We thus take a pessimistic approach and compute the contributions $W_{ik}(t, D)$ for each possible starting task (release at the beginning of the busy period) τ_{ik}, then take the maximum. Thus:

$$W_i(t, D) = max_{1 \leq k \leq N_i} W_{ik}(t, D)$$

In order to compute $W_{ik}(t, D)$, we first need to compute the distance ρ_{ijk} between the first activation time of a job of task τ_{ij} inside the busy period and the busy period itself, considering τ_{ik} as the starting task. Palencia and Gonzàlez showed that:

$$\rho_{ijk} = (T_i - (\phi_{ik} + J_{ik} - \phi ij) \mathrm{mod} T_i) \mathrm{mod} T_i$$

Given this equation, the contribution $W_{ijk}(t, D)$ of task τ_{ij} to the busy period considering τ_{ik} as the starting task can be computed as follows:

$$W_{ijk}(t, D) = \left(\left\lfloor \frac{J_{ij} + \rho_{ijk}}{T_i} \right\rfloor + \min \left(\left\lceil \frac{t - \rho_{ijk}}{T_i} \right\rceil, \left\lfloor \frac{D - \rho_{ijk} - d_{ij}}{T_i} \right\rfloor + 1 \right) \right)_0 C_{ij} \tag{7.1}$$

Clearly now

$$W_{ik}(t, D) = \sum_{1 \leq j \leq N_i} W_{ijk}(t, D) \tag{7.2}$$

Now let's compute the contribution of transaction $\mathcal{T}_a$. Let's call A the activation time of the instance of τ_{ab} we are checking. Then the first activation of task τ_{ac} after A if equal to $A + \phi_{ac} - \phi_{ab}$ if $\phi_{ac} \geq \phi_{ab}$, or $A + T_i + \phi_{ab} - \phi_{ac}$ if $\phi_{ac} < \phi_{ab}$, thus it is always equal to $A + (\phi_{ac} - \phi_{ab}) \mathrm{mod} T_i$. We can now compute the difference ρ^A_{acb} between the first activation of an instance of τ_{ac} inside the busy period and the busy period itself, supposing task τ_{ab} is activated at A, in the following way, considering that τ_{ac} is activated every T_a time units and that the busy periods starts at 0:

$$\rho^A_{acb} = (A + (\phi_{ac} - \phi_{ab}) \mathrm{mod} T_a) \mathrm{mod} T_a = (A + \phi_{ac} - \phi_{ab}) \mathrm{mod} T_a \tag{7.3}$$

We can then compute the contribution $W^A_{acb}(t, D)$ for task τ_{ab} and the total contribution $W^A_{ab}(t, D)$ for transaction $\mathcal{T}_a$ in the same way we computed W_{ijk} and W_{ik} in Equations 7.1 and 7.2, by simply substituting ρ^A_{acb} for ρ_{ijk}.

The finishing time w^A_{ab}, relative to the beginning of the busy period, can then be computed with the following recurrence:

$$w^A_{ab} = B_{ab} + W^A_{ab}(w^A_{ab}, D) + \sum_{1 \leq i \leq M, i \neq a} W_i(w^A_{ab}, D) \tag{7.4}$$

where

$$D = A + d_{ab}$$

It now remains to be seen how we can compute the set ψ_{ab} of starting times A that need to be checked. We start by computing an upper bound to the length of the busy period L. This can be done with the following recursion, considering the contributions without the deadline limitation:

$$L = \sum_{i \leq i \leq M} W_i(L, \infty) \tag{7.5}$$

Note that, due to the pessimistic nature of the computation, the value of L may growth to infinity even if the total system utilization $U = \sum_{1 \leq i \leq M, 1 \leq j \leq N_i} \frac{C_{ij}}{T_i}$

is less then 1. In this case, an upper bound to L equal to $\text{lcm}(T_i, \ldots, T_M)$ can be used; if a busy period were longer than this bound, in fact, then surely $U > 1$ and the system is not feasible from the beginning. We will provide a stricter bound in Section 8.3.

Using Theorem 9, we now consider the set ψ of all activations for τ_{ab} in the following way. We first add all the possible activation times whose absolute deadline correspond to the deadline of some task of another transaction, scheduled inside the busy period. Suppose that we want to do so for a task τ_{ij} of transaction $\mathcal{T}_i$. Since we do not know a priori which is the starting task for transaction $\mathcal{T}_i$, we must consider all possible starting tasks τ_{ik}. Palencia and Gonzàlez [25] proved that the set of activation times for jobs of τ_{ij} that can be scheduled inside the busy period, eventually after having experienced some jitter, is equal to:

$$\{a_{ijk} | \exists p, p_{0,ijk} \leq p \leq p_{L,ijk}, a_{ijk} = \rho_{ijk} + (p-1)T_i\}$$

where

$$p_{0,ijk} = -\left\lfloor \frac{J_{ij} + \rho_{ijk}}{T_i} \right\rfloor + 1 \tag{7.6}$$

$$p_{L,ijk} = \left\lceil \frac{L - \rho_{ijk}}{T_i} \right\rceil \tag{7.7}$$

Thus we need to consider the following set of activation times for τ_{ab}:

$$\psi' = \bigcup_{1 \leq i \leq M, i \neq a, 1 \leq k \leq N_i, 1 \leq j \leq N_i, p_{0,ijk} \leq p \leq pL,ijk} \rho_{ijk} + (p-1)T_i + d_{ij} - d_{ab}$$

We then need to check all possible activation times that correspond to some task τ_{ac} of $\mathcal{T}_a$ being released at the beginning of the busy period after having experienced maximum jitter. This is simply achieved by using the following set:

$$\psi'' = \bigcup_{1 \leq c \leq N_a, p_{0,abc} \leq p \leq p_{L,abc}} \rho_{abc} + (p-1)T_a$$

Finally, considering that we only need to check those activation times a_{ab}^k such that τ_{ab}^k is released inside the busy period starting at 0, we find:

$$\psi = \{A | A \in \psi' \cup \psi'', A + J_{ab} \geq 0\} \tag{7.8}$$

Given set ψ we finally obtain:

$$r_{ab} = \max_{A \in \psi}(w_{ab}^A - A)$$

The main difference between the proposed analysis and Palencia and Gonzàlez' original one is in the fact that in the original analysis the authors fixed a starting task for transaction $\mathcal{T}_a$ as well. This leads to a sometimes better evaluation of the upper bound L on the length of the busy period, but it adds unnecessary pessimism to the contribution of tasks of transactions $\mathcal{T}_a$ different from τ_{ab} and τ_{ac}. It is also $\max_{1 \leq i \leq M} N_i$ times slower than the complexity of our method.

7.3 New offset method

In this section we introduce a second method for computing the worst-case response times. To the contrary of the previous section, we now suppose that the transaction offsets are fixed, but that tasks experience no release jitter. As we said before, under these circumstances Theorem 8 does not hold anymore. We therefore apply ideas similar to the one used in Chapter 3. The worst-case response time for a task τ_{ab} can be surely found in a busy period in which some task τ_{pq} is activated (and released, since is suffers no release jitter) at the beginning of the busy period. Our idea is to fix this starting task τ_{pq} and compute the worst-case response time in this case. The overall worst-case response time for task τ_{ab} can then be found by taking the maximum for every possible starting task τ_{pq}.

As we have done before, we will now compute the worst-case contribution $W_{ik}^{pq}(t, D)$ of transaction $\mathcal{T}_i, i \neq a, p$ to the finishing time of task τ_a, supposing that task τ_{ik} is the first to be activated inside the busy period. Since the transaction offsets are fixed, it may not be possible for task τ_{ik} to be activated precisely at the beginning of the busy period. We thus use the same strategy as in Chapter 3: the worst-case contribution is surely to be found when the activation of task τ_{ik} is nearer to the beginning of the busy period. We can compute the minimum distance Δ_{pqik} between any activation of task τ_{pq} in the scheduling and the successive activation of task τ_{ik} as follows.

Lemma 9 *The minimum time distance between any activation time of task τ_{pq} and the successive activation time of task τ_{ik} is equal to:*

$$\Delta_{pqik} = (\phi_i + \phi_{ik} - \phi_p - \phi_{pq}) \bmod \gcd(T_p, T_i)$$

Proof. Note that for each possible job τ_{pq}^x and τ_{ik}^y, $a_{ik}^y - a_{pq}^x = \phi_i + \phi_{ik} - \phi_p - \phi_{pq} + yT_i - xT_p$. Thus $\forall x \geq 0, \forall y \geq 0, \exists z \in \mathbb{Z}, a_{ik}^y - a_{pq}^x =$

$\phi_i + \phi_{ik} - \phi_p - \phi_{pq} + z\,\gcd(T_i, T_p)$ and the minimum difference corresponds to the thesis. □

Given Δ_{pqik}, we can compute the distance ρ_{ijk}^{pq} between the first activation of τ_{ij} inside the busy period and the busy period itself as in Equation 7.3, substituting Δ_{pqik} for A:

$$\rho_{ijk}^{pq} = (\Delta_{pqik} + \phi_{ij} - \phi_{ik}) \mathrm{mod} T_i$$

Finally, given ρ_{ijk}^{pq} we use Equations 7.1 and 7.2 to compute $W_{ik}^{pq}(t, D)$. The overall correctness of this procedure is more formally proven by the following theorem:

Theorem 10 *The worst-case response time of task τ_{ab} corresponds to the response time of one of its job τ_{ab}^k activated inside a busy period where task τ_{pq} 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. Let's consider the response time of a job of τ_{ab} activated inside a busy period; there is surely at least one task that is released at the beginning at the busy period, say τ_{pq} (note that it can be $a = p$). We do not know which starting task for transaction $\mathcal{T}_i, i \neq a, p$, gives the worst-case contribution to the finishing time of τ_{ab}; suppose it is τ_{ik}. Now, if we move the activation pattern of τ_{ik} to occur earlier inside the busy period, we won't surely decrease its contribution, since the deadlines of tasks of $\mathcal{T}$ will be "pulled back" as well, and this means that new jobs may be activated inside the busy period. But since we proved that 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_{ik}, the theorem follows. □

Note that Theorem 10 still gives us a pessimistic condition, since the activation times of the starting task of all transactions $\mathcal{T}_i, i \neq a, p$, are simultaneously minimized with respect to the activation time of τ_{pq}, while this may not be possible in the actual schedule. In fact, when we "pull back" the tasks to their minimum distance from τ_{pq}, we are not considering the cross relations between them. We may devise a more complex condition, in order to reduce pessimism, by "fixing" more than one starting task, but as we showed in Chapter 3, this idea is not likely to achieve significantly better results while it substantially increases the complexity of the algorithm.

As before, we now need to compute the set ψ^{pq} of activation times for task τ_{ab} that need to be checked. We start be computing the upper bound

L^{pq} on the length of the busy period, given by the following recurrence:

$$L^{pq} = W^{pq}_{pq}(L^{pq}, \infty) + \sum_{1 \leq i \leq M, i \neq p} W^{pq}_{i}(L^{pq}, \infty) \tag{7.9}$$

Note that as before, L^{pq} may growth to infinity even if the total utilization is less then 1, so that an upper bound is required.

We now need to compute the total set of possible activations of τ_{ab} inside the busy period, that we denote ψ^{pq}. Following the same line as in Lemma 9, it should be easy to see that any activation of τ_{ab} may only occurs at time $\Delta_{pqab} + k \gcd(T_p, T_a)$ for some $k \in \mathbb{Z}$. Since the activation must occur inside the busy period we get:

$$\psi^{pq} = \{A | \exists k, A = \Delta pqab + k \gcd(T_p, T_a), A < L^{pq}\} \tag{7.10}$$

The recurrence over the finishing time for τ_{ab}, given an activation time A, is now:

$$w^{pqA}_{ab} = B_{ab} + W^{pq}_{pq}(w^{pqA}_{ab}, D) + W^{A}_{ab}(w^{pqA}_{ab}, D) + \sum_{1 \leq i \leq M, i \neq a, p} W^{pq}_{i}(w^{pqA}_{ab}, D) \tag{7.11}$$

if $p \neq a$, or otherwise:

$$w^{aqA}_{ab} = B_{ab} + W^{aq}_{aq}(w^{aqA}_{ab}, D) + \sum_{1 \leq i \leq M, i \neq a} W^{aq}_{i}(w^{aqA}_{ab}, D) \tag{7.12}$$

We want to briefly underline two significant considerations. First, note that if $\gcd(T_p, T_a) = 1$, we will be basically forced to check each possible activation time for τ_{ab} inside the busy period. Fortunately, there is a way to bound the number of activation times to check. We could use the analogue of Theorem 9 and compute a new set of activation times ψ' in the same way as in Section 7.2 (but taking into consideration the minimum activation time Δ for the starting task of each transaction). Now given two values $A', A'' \in \psi, A' < A''$, if there are no values of ψ' in the interval $(A', A'']$, then only A' needs to be checked. In this way, no more activation times that those of the previous method need to be checked.

Second, this method can be used even if release jitters are not zero, but its complexity becomes greater. In fact, we cannot simply suppose that task τ_{pq} is released at the beginning of the busy period after having experienced maximum jitter, since the contribution of tasks of other transactions will now depend both on the transaction offsets and on jitters. This basically means that we will be forced to try every possible release jitter for task τ_{pq}.

Chapter 8

Improved holistic analysis

8.1 Introduction

The response time analyses developed in the previous Chapter 7 can be used to solve the schedulability problem for dynamic offset transactions [26] [25]. We consider the same transaction model as in Chapter 7, but we now suppose that task offsets and jitters are not initially known; on the contrary, we want to impose a precedence constraint among tasks of the same transaction so that each task τ_{ij} can begin execution only after task τ_{ij-1} has finished at least δ_{ij} time units before. In [25] Palencia and Gonzàlez showed that this model is useful for systems where tasks suspend themselves and for distributed multiprocessor transactions.

For example, a task may execute for some time, and then suspend itself to read some data from the disk. We could model this task as a transaction $\mathcal{T}_i$ composed of two tasks: task τ_{i1} corresponds to the code before the suspension, and τ_{i2} to the code after the suspension. δ_{i2} is used to model the maximum suspension time.

In multiprocessor and distributed systems it is usual that the system can be modelled with transactions composed of several tasks. For example, consider a client-server architecture where a client task issues service requests to several servers, possibly executing on different processors. The client task can be modelled as a transaction where pieces of code between request are individual tasks. Each portion of execution on a server is too modelled as a task of the same transaction, running on a different processor. Communication delays can be modelled using δ values, or, if a more precise model is required, each network node can be modelled as an individual processor,

accounting the non-preemptability of message packets as blocking times [18] (see also Chapter 9).

Since precedence constraints are hard to consider, we want to change the task offsets and jitters so that each task can always be released after the previous one has finished. Offsets and jitters thus depend on task response times. But unfortunately, in order to compute response times we need to know both offsets and jitters. To solve the problem we will use variations of the holistic analysis first developed by Tindell and Clark [35].

This chapter is organized as follows. In Section 8.2 we will introduce different holistic techniques, including our original contribution to the problem. Then in Section 8.3 we will discuss implementation issues for the presented techniques and in Section 8.4 we will present experimental evaluations. Finally, Section 8.5 introduces the deadline problem that will be further examined in the following Chapter 9.

8.2 Holistic Analyses

In this section we use a modified version of the transaction model introduced in Section 2.3 and used in the previous Chapter 7. We suppose that for each transaction $\mathcal{T}_i$, task τ_{i1} is activated at the same time of the transaction, and that each subsequent task τ_{ij} must be released some time δ_{ij} after the finishing time of τ_{ij-1}.

Palencia and Gonzàlez [26] used the following algorithm (a refinement of Tindell and Clark original holistic analysis [35]), called WCDO (Worst-case Analysis for Dynamic Offsets), to solve the problem of response time analysis for the above model. Each transaction is transformed into a transaction of the type described in Section 2.3 using the following offsets:

$$\phi_{i1} = 0 \tag{8.1}$$

$$\phi_{ij} = \sum_{1 \le k < j} C_{ik} + \delta_{ik+1} \qquad \forall 1 < j \le N_i \tag{8.2}$$

and relative deadlines $d_{ij} = D_{ij} - \phi_{ij}$. The task jitters are initially set to 0, and then the worst-case response time R_{ij} is computed for each task. At this point, jitters are modified as follows:

$$J_{i1} = 0 \tag{8.3}$$

$$J_{ij} = R_{ij-1} - \phi_{ij} + \delta_{ij} \qquad \forall 1 < j \le N_i \tag{8.4}$$

After setting the jitters, new response times are computed for the tasks with the response time analysis described in Section 7.2 (NTO analysis), jitters are modified again and so on until the system converges to a stable result or diverges, in which case the iteration is usually stopped after $R_{ij} > D_{ij}$ for some task τ_{ij}, since this means that we cannot prove that the system is schedulable.

Note that this basically means that the offsets are first set equal to the minimum possible response time of each task (supposing that the relative response time of each task before is exactly equal to its computation time) and then at each step jitters are modified so that each task τ_{ij} is released in the worst-case after waiting δ_{ij} time units from the worst-case finishing time of the immediately previous task τ_{ij}.

If the response times are bounded from above, then the algorithm is proved to converge because the worst-case response times, as computed in Section 7.2, are monotonically non decreasing in the jitters, as the following theorem proves:

Theorem 11 *The worst-case response times computed in Section 7.2 are monotonically non decreasing in the jitters.*

Proof. We will prove that given any two tasks τ_{ij} and τ_{ab} (possibly the same task), if we substitute jitter J_{ij} with a new jitter value $J'_{ij} \geq J_{ij}$, then the new worst-case response time r'_{ab} is greater than or equal to the previous one r_{ab}.

We need to consider different cases. First, suppose $i \neq a$. Then we have two possibilities. If τ_{ij} is not the starting task of $\mathcal{T}_i$, then r'_{ab} is surely increased or equal to r_{ab} since the increased jitter J'_{ij} may cause a new job of τ_{ij}, activated before the beginning of the busy period, to be released inside it. If τ_{ij} is the starting task, since it is released after having experienced maximum jitter due to Theorem 9, the deadlines of all tasks of $\mathcal{T}_i$ are "pulled back" a number of time units equal to $J'_{ij} - J_{ij}$ and thus new jobs may contribute to the finishing time of τ_{ab}. However, since all activation times of tasks of $\mathcal{T}_i$ are pulled back as well, jobs that were released inside the busy period may now not be released inside it, and thus the total contribution of $\mathcal{T}_i$ may be less than before. Suppose that this happens and let τ_{ik} be the task of one such job, such that all tasks that are released after the beginning of the busy period with jitter J_{ij} and starting task τ_{ij} are released as well after the beginning of the busy period with starting task τ_{ik} and jitter J'_{ij}. Then even if the contribution of $\mathcal{T}_i$ is less than before with starting task τ_{ij}, the contribution is higher or equal selecting τ_{ik} as the starting task (see

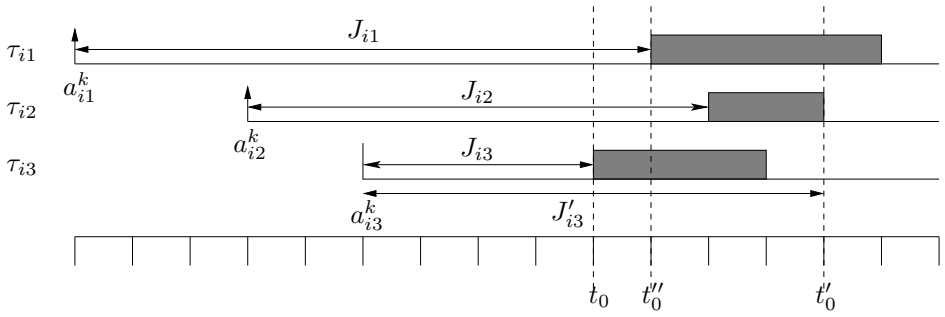

Figure 8.1: Example of jitter extension.

Figure 8.1 where $j = 3$ and $k = 1$; the worst-case busy period starts not at t'_0 but at t''_0).

Second, suppose that $i = a$. If $j \neq b$, then the release time is greater or equal then before for the same reasons as above. If $j = b$, then the increased jitter may: increase the number of jobs of τ_{ab} scheduled inside the busy period, permit new values inside ψ (those for which the condition $A + J_{ab} \geq 0$ didn't hold before), and "pull back" the deadlines of all tasks of $\mathcal{T}_a$ in case τ_{ab} is the starting task. Notice that in this case, the deadline of τ_{ab} is pulled back as well, so that D is decreased and the contribution of tasks of other transactions may decrease. However, since we always check all values of A so that the deadlines of τ_{ab} corresponds to the deadline of some jobs of another transaction, the worst-case response time of τ_{ab} cannot decrease even in this case. □

We shall now introduce a new refined algorithm for this same problem, which we will call algorithm CDO (Cycling Dynamic Offsets). The basic idea is to modify Palencia and Gonzalez algorithm to take into account offsets between tasks of different transactions, as we did in the response time analysis of Section 7.3 (TO analysis). However, such extension is not immediate, because our TO analysis does not consider jitters. Modifying TO for taking jitters into account is not trivial, as the TO algorithm would become too complex.

Therefore, we decided to follow a different approach: to eliminate jitters from Palencia and Gonzalez analysis. How it will become evident in Section 8.4, by eliminating the jitters and by using the TO analysis, we are able to provide much less pessimistic response times.

We will thus use the following idea. We follow an iterative approach similar to that of WCDO, but instead of updating the jitters at each step,

we modify the offsets, based on the response times computed at the step before, as follows:

$$\phi_{i0} = 0 \tag{8.5}$$

$$\phi_{ij} = R_{ij-1} + \delta_{ij} \qquad \forall 1 < j \leq N_i \tag{8.6}$$

while jitters are always 0.

Roughly speaking, the iterative algorithm can be described as follows. Starting from the best-case response times $\{R_{ij} = C_{ij}\}$, we evaluate the offsets according to Equations 8.5 and 8.6. At step k, we compute the response times given the offsets computed at the previous step $k-1$; given these response times, we set the new offsets; and so on. The algorithm should stop when we obtain the same response times as in the previous step.

Unfortunately, the algorithm just described does not work since we have no guarantees, using either the TO or the NTO analysis, that response times are monotonic in the offsets. This means that the iteration might not converge even if the response times are bounded from above, since it could reach a limit cycle.

To highlight the problem and explain our solution, we need to introduce some additional notation. We shall use $\mathbf{R}^k$ as the response time vector $\{R^k_{11}, \ldots, R^k_{1N_1}, \ldots, R^k_{M1}, \ldots, R^k_{MN_M}\}$ of global response times computed at step k of the algorithm.

We define the $\leq$ operator over the space of response time vectors as follows:

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

We shall further introduce function f as the function that, given the response times at some step k, evaluates new response times by computing offsets as in Equations 8.5 and 8.6 and running the response time analysis presented in Section 7.2 (NTO). We denote with f' the function that does the same thing using the analysis presented in Section 7.3 (TO).

The algorithm can then be expressed as an iteration over $\mathbf{R}^{k+1} = f(\mathbf{R}^k)$, starting from $\mathbf{R}^0 = \{C_{11}, \ldots, C_{1N_1}, \ldots, C_{M1}, \ldots, C_{MN_M}\}$.

Figure 8.2 illustrates the problem of the limit cycle. It shows a possible evolution of the response times computed at each step (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). The response time vectors computed at each step are numbered from $\mathbf{R}^0$ to $\mathbf{R}^8$; arrows represent the application of function f at

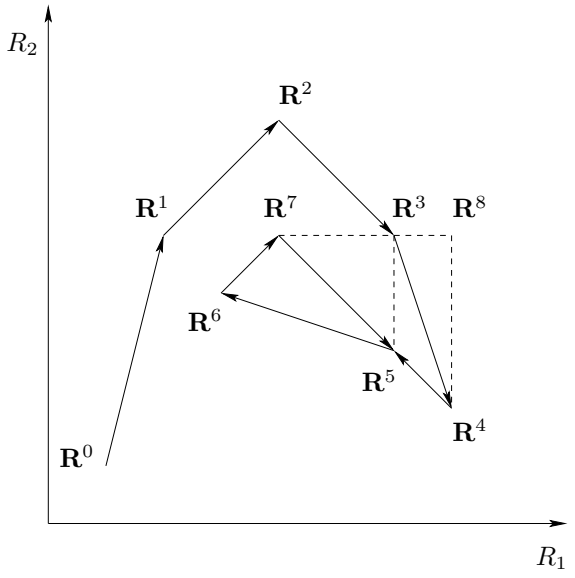

Figure 8.2: Holistic analysis example.

each step. In the example, if we applies the algorithm described above, the iteration enters a limit cycle at step 5.

We will thus need to modify the iteration step in order to achieve convergence. In Figure 8.3 we report the CDO algorithm. Each iteration step k in algorithm CDO is done as follows. First, if $f(\mathbf{R}^k) \leq \mathbf{R}^k$, we can immediately stop the algorithm with final response times $\mathbf{R}^k$. In fact, this means that, by 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 cycle($\mathbf{R}^k$):

$$\text{cycle}(\mathbf{R}^k) = \max(\{\bar{k} | f(\mathbf{R}^k) = \mathbf{R}^{\bar{k}}\} \cup -1).$$

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 cycle($\mathbf{R}^k$) $= -1$, then we simply set $\mathbf{R}^{k+1} = f(\mathbf{R}^k)$. If cycle($\mathbf{R}^k$) $= \bar{k} \leq 0$, 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. We define a function maxR as follows:

$$\text{maxR}(k_1, k_2) = \begin{pmatrix} \max_{k \in \{k_1 \ldots k_2\}}(R_{11}^k) \\ \vdots \\ \max_{k \in \{k_1 \ldots k_2\}}(R_{MN_M}^k) \end{pmatrix}$$

Then, the new response time can be computed as $\text{maxR}(\bar{k}, k)$.

Unfortunately, when we jump out of a limit cycle we could incur in another cycle. An example is presented in Figure 8.2. Suppose we are at step $k = 7$. When we jump out of cycle $\{\mathbf{R}^5, \mathbf{R}^6, \mathbf{R}^7\}$, we find the already visited point $\mathbf{R}^3$. If we simply set $\mathbf{R}^{k+1} = \text{maxR}(5, 7)$, 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 again function cycle 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 jump in a cycle including more points. Therefore, sooner or later we must find a point which is not part of any cycle.

The following fundamental theorems prove that algorithm CDO is indeed correct and that it provides better response times with respect to WCDO; g is the function that, given the response times at some step k, evaluates new response times according to the WCDO algorithm, that is, by computing new jitters as in Equations 8.3 and 8.4 and running the NTO analysis with the offsets given by Equations 8.1 and 8.2.

Theorem 12 *Given response times $\mathbf{R}^k$ at step k, the new response times $\mathbf{R}^{k+1}$ computed by function f are not greater than the response times computed by function g.*

Proof. Let $\mathbf{R}' = g(\mathbf{R}^k)$ and $\mathbf{R}'' = f(\mathbf{R}^k)$. Furthermore, let ϕ'_{ij} and J'_{ij} be the offset and jitter of task τ_{ij} computed at step $k+1$ by WCDO and let ϕ''_{ij} be the offset computed by CDO. Then it suffices to prove that $\forall\, 1 \leq i \leq M, \forall\, 1 \leq j \leq N_i :\ R'_{ij} \geq R''_{ij}$.

Using Equations 8.4 and 8.6 we easily obtain $\phi''_{ij} - \phi'_{ij} = J'_{ij}$ for each task τ_{ij}. Note that this basically means that, assuming the same transaction activation times, in algorithm CDO each task activation time coincides with the release time of the same task in WCDO after having experienced the maximum jitter.

We know prove that the response time R''_{ab} of task τ_{ab} computed by the NTO analysis in the CDO case with offsets $\{\phi''_{ij}\}$ and zero jitters is less than or equal to the response time R'_{ab} computed by NTO in the WCDO case with offsets $\{\phi'_{ij}\}$ and jitters $\{J'_{ij}\}$. Let's start by considering the contribution of a transaction $\mathcal{T}_i, i \neq a$, to the finishing time of a task τ_{ab}. Suppose τ_{ij} is a starting task in both cases. Then because of the considerations from the previous paragraph, the activation time of $\mathcal{T}_i$ does not change between CDO

and WCDO. This means that all jobs that can be released after the beginning of the busy period in WCDO, are still activated after the beginning of it in CDO, while tasks that are activated after the end of the busy period in algorithm CDO, may be activated inside it in WCDO due to the jitter. Thus, the total contribution of $\mathcal{T}_i$ in CDO is less than or equal to the one in WCDO.

Finally, consider transaction $\mathcal{T}_a$. For tasks $\tau_{ac}, c \neq b$, the same considerations as above hold. For task τ_{ab}, note that since D_{ab} is not changed by the algorithms, τ_{ab} is actually always activated in CDO at the same time in which it would be released in WCDO should it suffers maximum jitter. Therefore the only effect is that some instances of τ_{ab} that could be release inside a busy period of maximum length in WCDO are activated after it in CDO, and thus they are not considered for response time evaluation. This means that even the contribution of transaction $\mathcal{T}_a$ in CDO is less than or equal to the one in WCDO and thus the theorem holds. □

Theorem 13 *Given a transaction set $\mathcal{T}$, if WCDO converges to response times $\bar{\mathbf{R}}$, then CDO converges to response times $\mathbf{R} \leq \bar{\mathbf{R}}$ in a finite number of steps.*

Proof. Since WCDO converges to $\bar{\mathbf{R}}$, $g(\bar{\mathbf{R}}) = \bar{\mathbf{R}}$. Also note that since WCDO is monotonically non decreasing in the jitters, then function g is also monotonic, and thus $\forall\, \mathbf{R} \leq \bar{\mathbf{R}}, g(\mathbf{R}) \leq \bar{\mathbf{R}}$. Therefore because of Theorem 12 we also obtain: $\forall\, \mathbf{R} \leq \bar{\mathbf{R}}, f(\mathbf{R}) \leq \bar{\mathbf{R}}$. Since clearly $\mathbf{R}^0 \leq \bar{\mathbf{R}}$, it follows that CDO can never reach at any step a point $\mathbf{R} \not\leq \bar{\mathbf{R}}$. If this was possible, we could surely find a step k so that $\forall\, k' \leq k, \mathbf{R}^{k'} \leq \bar{\mathbf{R}} \wedge \mathbf{R}^{k+1} \not\leq \bar{\mathbf{R}}$. However, this is impossible. In fact, $\mathbf{R}^{k+1}$ can be obtained from $\mathbf{R}^k$ by application of either function f or of function maxR, but neither of them can give a result that is not less than or equal to $\bar{\mathbf{R}}$.

Since the number of points $\mathbf{R} \leq \bar{\mathbf{R}}$ are finite, it now suffices to prove that CDO never passes through the same point twice before stopping. Note that it is impossible that for any step k, $\exists k' \leq k, \mathbf{R}^k = \mathbf{R}^{k'}$, since the iterative step of CDO only ends when function cycle 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 $\mathbf{R} \leq \bar{\mathbf{R}}$. □

Note that although CDO is proven to converge in finite time if a stable point exists, it can be untractable in the worst-case, since it may require to pass trough each state $\mathbf{R} \leq \bar{\mathbf{R}}$. Simulation results show that this event

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

Figure 8.3: CDO iteration step.

is very unlikely to happen, and that algorithm CDO usually converges in a number of steps no greater that those of algorithm WCDO; however, since the possibility of a cycle remains, we must check at each step if any cycle is present.

We can define a simpler algorithm, that we call MDO (Maximum Dynamic Offsets), using the following iteration step:

$$\mathbf{R}^{k+1} = \begin{pmatrix} \max(R_{11}^k, f(R_{11}^k)) \\ \vdots \\ \max(R_{MN_M}^k, f(R_{MN_M}^k)) \end{pmatrix}$$

In other words, in algorithm MDO at each step we always "jump out" to the maximum between the previously computed release times and the newly computed ones. Since algorithm MDO is clearly monotonic, if the response times are bounded that it surely converges. Furthermore, note that because of Theorem 12 the response times computed by MDO at each step are less than or equal to those computed by WCDO. Given this results, it is trivial to prove that Theorem 15 still holds, in the sense that MDO converges to a stable point that is less or equal to that of WCDO in a finite number of steps.

While at first glance algorithm MDO may seem excessively pessimistic in confront to CDO, simulations showed that the decrease in performance is negligible. Therefore, algorithm MDO will be our preferred holistic method int the remainder of this work since it is way simpler than CDO.

Since under both MDO and CDO tasks have always zero jitter, the TO

response time analysis developed in Section 7.3 comes in handy. In order to use that method, though, we need to prove that Theorems 15 still holds. This is a consequence of the following theorem:

Theorem 14 *Given a transaction system of the type described in Section 2.3, where all release jitters are equal to zero, the response times computed by the TO analysis are no greater than the response times computed by the NTO analysis.*

Proof. Given a task τ_{ab}, we will prove that the total contribution of every transaction to its worst-case response time in the TO analysis, considering any initial task τ_{pq}, is less than or equal to the contribution in the NTO analysis.

Let's start by considering the total contribution of transaction $\mathcal{T}_i, i \neq a$, to the finishing time of a task τ_{ab}. Since release jitters are equal to zero, the only effect of applying TO with respect to NTO is that the activation time of the starting task of $\mathcal{T}_i$, say 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 TO and thus the contribution is surely less or equal than the one in NTO, since some jobs may not be scheduled in the busy period due to a deferred activation time or deadline.

For what concerns transaction $\mathcal{T}_a$, note that the contribution of tasks of $\mathcal{T}_a$ does not change between the two analyses once an activation time for τ_{ab} has been fixed. Therefore, it suffices to recall that while checking the activation times for τ_{ab} in set ψ from Section 7.2 (NTO) is equivalent to checking all possible activation times inside the busy period, the TO analysis limits the activation times to be checked to a subset of the busy period. Therefore, the worst-case contribution of transaction $\mathcal{T}_a$ in TO is surely less than or equal to the one in NTO. □

Given function f, f' and g and response time vector $\mathbf{R}^k$ at step k we obtain from Theorems 12 and 14: $g(\mathbf{R}^k) \leq f(\mathbf{R}^k) \leq f'(\mathbf{R}^k)$. It is then trivial to prove the following theorem:

Theorem 15 *Given a transaction set $\mathcal{T}$, if WCDO converges to response times $\bar{\mathbf{R}}$, both CDO and MDO, using function f' instead of f at each iteration step, converge to response times $\mathbf{R}' \leq \bar{\mathbf{R}}$ and $\mathbf{R}'' \leq \bar{\mathbf{R}}$ in a finite number of steps.*

Proof. The proof is an immediate extension of Theorem 15. □

Since both CDO and MDO can be used either with the NTO or the TO response time analysis, in the remainder of our work we will use names CDO-NTO and MDO-NTO to refer to algorithms CDO and MDO applied to NTO, and names CDO-TO and MDO-TO to refer to the holistic methods applied to TO.

Please note that while at each step algorithm MDO-TO performs better than MDO-NTO, this is no proof that MDO-TO will converge to a result that is less or equal than that of MDO-NTO. In fact, we have no proof that the response times computed from a vector $\mathbf{R}^k$ at step k will be less or equal to the response times computed from another vector $\mathbf{R}'^k \geq \mathbf{R}^k$. Indeed, using the following feasible single processor system: $\mathcal{T}_1(\phi_1 = 13, T_1 = 30, C_{11} = 2, C_{12} = 1, C_{13} = 2, C_{14} = 2, D_{11} = 6, D_{12} = 9, D_{13} = 16, D_{14} = 22), \mathcal{T}_2(\phi_2 = 49, T_2 = 70, C_{21} = 2, C_{22} = 4, C_{23} = 3, C_{24} = 3, D_{21} = 9, D_{22} = 26, D_{23} = 38, D_{24} = 51), \mathcal{T}_3(\phi_3 = 112, T_3 = 120, C_{31} = 2, C_{32} = 1, C_{33} = 2, C_{34} = 1, D_{31} = 37, D_{32} = 56, D_{33} = 93, D_{34} = 112), \mathcal{T}_4(\phi_4 = 121, T_4 = 140, C_{41} = 1, C_{42} = 4, C_{43} = 2, C_{44} = 6, D_{41} = 9, D_{42} = 47, D_{43} = 65, D_{44} = 121)$, it can be seen that using MDO-NTO R_{24} converges to a value of 25 while using MDO-TO we get a value of 26.

8.3 Implementation Issues

Figures 8.4 and 8.5 shows the pseudo-code for algorithm WCDO and MDO, where *rt-analysis* is either the NTO or the TO response time analysis; they are detailed in Figures 8.6 and 8.7.

Although algorithm NTO and TO may appear complex, their overall complexity is not too great compared to that of EDF schedulability analyses such as Baruah processor demand criterion [4] or the author's 1-fixed test from Chapter 3. In each algorithm, we compute the response times for all tasks, that is $N = \sum_{1 \leq i \leq M} N_i$ response times. For each task τ_{ab} and each of its possible activation times, we need to compute the contribution of each other transaction ($M - 1$ transactions). For each transaction $\mathcal{T}_i$, we must compute the worst-case contribution of all tasks by selecting each one in turn as a possible starting task: that is, computing $W_i(t, D)$ has a complexity of $O(N_i^2)$. Thus, the complexity of algorithm NTO amounts to $O(N^2 \max_{1 \leq i \leq M} N_i \; pseudopoli)$, and the complexity of NTO to $O(N^3 \max_{1 \leq i \leq M} N_i \; pseudopoli)$ (since we must also check all possible initial tasks τ_{pq}); this is also the complexity of the original Palencia-Gonzàlez analysis. $O(pseudopoli)$ is a pseudo-polynomial complexity that measure the number of activations time to be checked inside

compute $\forall\, i,j : \phi_{ij}$ (8.1,8.2)
$\forall\, i,j : R_{ij} = C_{ij}$
do
    compute $\forall\, i,j : R_{ij}$ using *NTO-analysis*
    compute $\forall\, i,j : J_{ij}$ (8.3,8.4)
    if $\exists\, i,j : R_{ij} > D_{ij}$
        return *unknown*
while R_{ij} changes
return *feasible*

Figure 8.4: Sample code, algorithm WCDO

compute $\forall\, i,j : \phi_{ij}$ (8.1,8.2)
$\forall\, i,j : R_{ij} = C_{ij}$
do
    compute $\forall\, i,j : R'_{ij}$ using *rt-analysis*
    compute $\forall\, i,j : \phi_{ij}$ (8.5,8.6)
    $\forall\, i,j : R_{ij} = \max(R_{ij}, R'_{ij})$
    if $\exists\, i,j : R_{ij} > D_{ij}$
        return *unknown*
while R_{ij} changes
return *feasible*

Figure 8.5: Sample code, algorithm MDO

```
compute L (7.5)
for each a = 1 ... M
    for each b = 1 ... N_a
        R_ab = C_ab
        compute ψ (7.8)
        for each A in ψ^pq
            compute w_ab^A (7.4)
            if w_ab^A − A > R_ab
                R_ab = w_ab^A − A
            end if
        next A
    next b
next a
```

Figure 8.6: Sample code, algorithm NTO

```
for each a = 1 ... M
    for each b = 1 ... N_a
        R_ab = C_ab
        for each p = 1 ... M
            for each q = 1 ... N_p
                compute L^pq (7.9)
                compute ψ^pq (7.10)
                for each A in ψ^pq
                    compute w_ab^pqA (7.11,7.12)
                    if w_ab^pqA − A > R_ab
                        R_ab = w_ab^pqA − A
                    end if
                next A
            next q
        next p
    next b
next a
```

Figure 8.7: Sample code, algorithm TO

the busy period and depends on tasks' parameters; note that since there cannot be more activations that the length of the busy period, any upper bound on the maximum length of the busy period constitutes an upper approximation for $O(pseudopoli)$ as well. A simple bound, equal to $\frac{\sum_{i,j} C_{ij}}{1-U}$, can be found in the following way by reasoning of the busy period length L. Surely in any case $L \leq \sum_{i,j} \lceil \frac{L}{T_i} \rceil C_{ij}$, since this condition means that all tasks of all transactions start at the same time with no offset. Then: $L < \sum_{i,j} (\frac{L}{T_i} + 1) C_{ij} = LU + \sum_{i,j} C_{ij}$, and by solving for L we get our bound. Also note that supposing for example $N_i = M$ for each i, the polynomial complexity of both NTO and TO is only $\sqrt{N}$ higher that that of Baruah's processor demand criterion (where N is the number of tasks) and of 1-fixed respectively

The worst-case complexity of holistic analysis is very high, since theoretically even for WCDO at each step we could have an increase in of just one time unit in one component of the response time vector. However, simulation shows that all algorithms typically converges (provided that they do so) in a small number of steps (see next section).

8.4 Experimental evaluation

We will now evaluate the performance of the described analysis. For each experiment, we generated 1000 synthetic transaction sets consisting of 5 transaction with 5 or 10 tasks.

Each transaction was generated in the following way. First, a transaction utilization was randomly generated according to a uniform distribution, so that the total utilization summed up to the desired value (Bini showed how to efficiently obtain such as result in [7]). Then periods were generated uniformly between 10 and 200 for transactions with 5 tasks and between 20 and 400 for transactions with 10 tasks, and the total worst-case computation time of each transaction was computed based on utilization and period. Relative deadlines were assigned to each transaction to be between half period and the period, and transaction offsets were 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 computation time. For simplicity we assumed all δ-values to be equal to 0, and that no resource constraint was present. Finally, deadlines were assigned proportional to each task's computation time, so that the deadline of the last task corresponded to the transaction

deadline (note that if we can set the task deadlines freely, the latter can not be the best solution: we will return on that in the following Section 8.5).

We generated the transaction periods so that the greatest common divisor between any two periods were a multiple of 1/20 of the maximum period (10 for transactions with 5 tasks and 20 for transactions with 10 tasks). As we showed in [14], the greater is the gcd between two transaction periods, the larger is the minimum distance between two successive activations of tasks of the two transactions and thus the smallest is the contribution of one transaction to the tasks of the other. In particular, we could show that if a period were prime with all others, then the TO analysis would be equal to the NTO one. Note that fortunately this kind of situation is not very common in real applications.

Figure 8.8 shows the percentage of feasible transaction sets scheduled by algorithms WCDO, MDO-NTO and MDO-TO, for a system of 5 transaction with 5 tasks each, running on a single processor, with utilizations ranging from 0.6% to 0.94%. While algorithm MDO-NTO achieves a small gain over WCDO (but it is about 5 times faster), algorithm MDO-TO achieves an improvement up and beyond 20% for utilizations around 0.75%. Also, the response times computed by algorithm WCDO are 36% longer then those computed by algorithm MDO-TO in mean.

Figure 8.9 shows the percentage of feasible transaction sets scheduled for the same system as before but running on two processors with utilization between 0.6% and 1.4%. Once again algorithm MDO-NTO does not achieve any significant benefit over WCDO, but algorithm MDO-TO performs even better than before achieving an improvement over 30% around 0.95% utilization; also, the improvement on the response times was around 46%.

Figure 8.10 shows the case with 5 transactions and 10 tasks per transaction, running on 4 processors. This time algorithm MDO-TO is able to schedule about 50% more total transaction sets than MDO-NTO at about 1.1% utilization. The benefit of our transaction offsets approach clearly seems to go up as the parallelism of the system increases.

Figure 8.11 uses the same system as Figure 8.9, but shows a comparison between algorithm MDO-TO and CDO-TO and between MDO-NTO and CDO-NTO. As we said in Section 8.2, there seems to be no real gain in preferring CDO over the simpler MDO. In particular, algorithm CDO-TO is able to schedule only 0.14% more tasks than MDO-TO, and there is no improvement for CDO-NTO over MDO-NTO.

Finally, Table 8.1 shows the mean number of steps needed for conver-

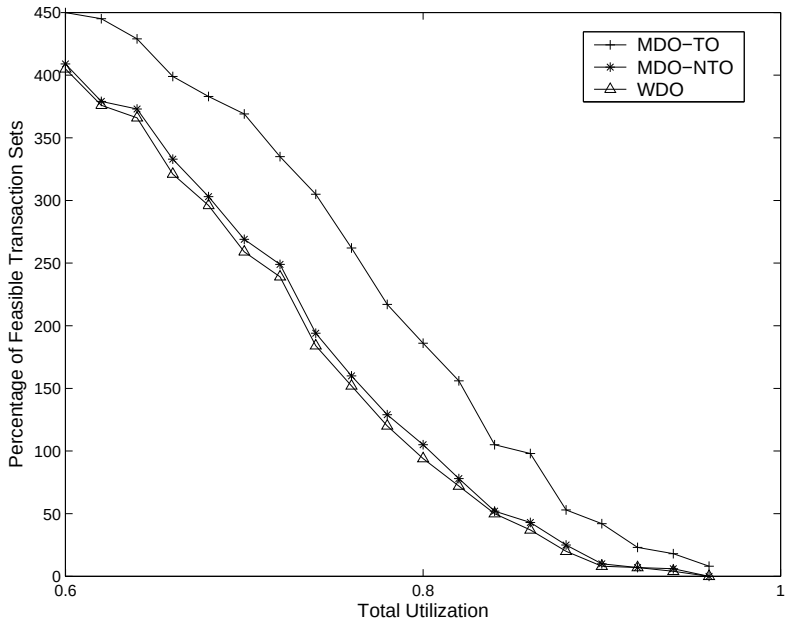

Figure 8.8: 5 transactions, 5 tasks per transaction, single processor

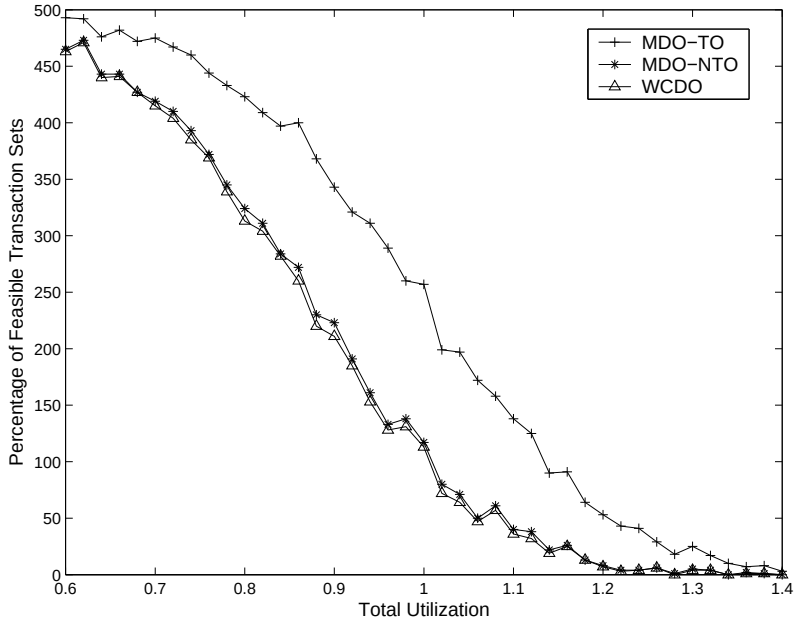

Figure 8.9: 5 transactions, 5 tasks per transaction, two processors

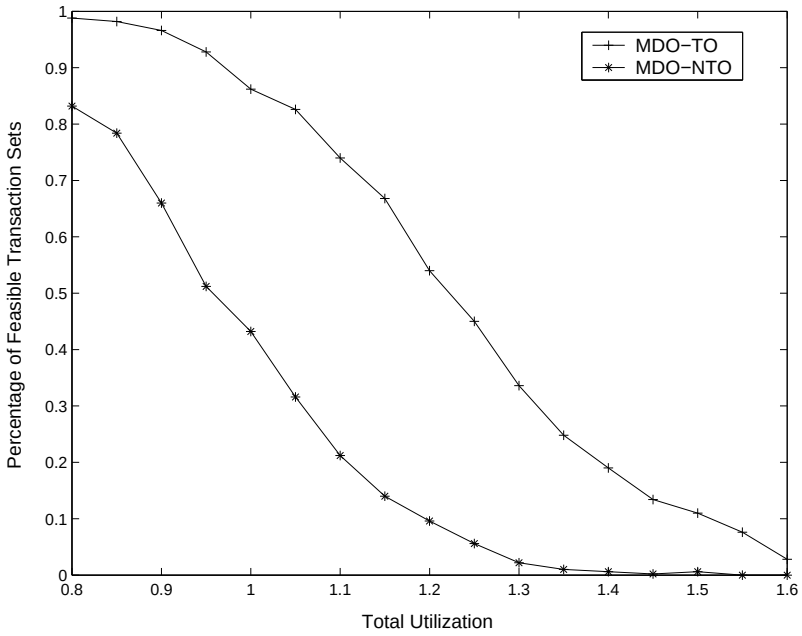

Figure 8.10: 5 transactions, 10 tasks per transaction, four processors

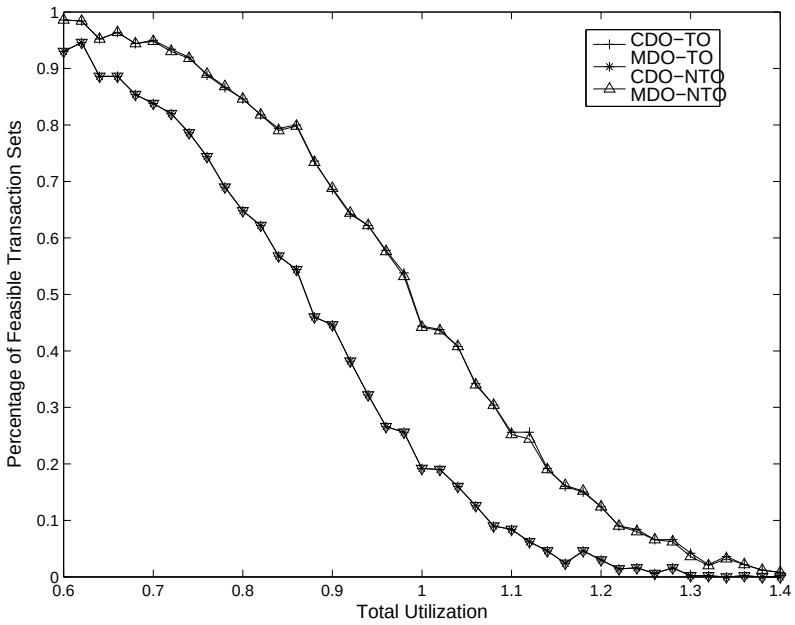

Figure 8.11: 5 transactions, 5 tasks per transaction, two processors

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

Table 8.1: Mean number of steps, 5 transactions

gence by algorithms MDO-TO, MDO-NTO and WCDO in the cases analyzed before. As you can see, the number of steps is indeed 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.

8.5 Deadline selection

In Section 8.2, we assumed that task deadlines were fixed. Although this can be the case with some distributed transactions, there are many cases in which we are only interested in that the last task of each transaction finishes before a prefixed transaction deadline. In other words, we are not really concerned with the finishing time of the non terminal task of a transaction provided that the last task meets the deadline. In this situation, we have some grades of liberty in our model, in the sense that we are free to choose the deadlines of all intermediate tasks. The easiest choice would be to assign to each task a deadline proportional to its computation time, that is, called D_i the relative deadline of transaction $\mathcal{T}_i$:

$$D_{ij} = D_i \frac{\sum_{1 \le k \le j} C_{ik}}{\sum_{1 \le k \le N_i} C_{ik}} \tag{8.7}$$

Extensive computations show that this simple heuristic is in fact good in the general case, in the sense that given no further information on the system it seems difficult to find a better one. If the δ-values are not zero, however, the following is usually a better heuristic:

$$D_{ij} = (D_i - \sum_{2 \le k \le N_i} \delta_{ik}) \frac{\sum_{1 \le k \le j} C_{ik}}{\sum_{1 \le k \le N_i} C_{ik}} + \sum_{2 \le k \le j} \delta_{ik} \tag{8.8}$$

It can easily be seen, however, that neither heuristics are optimal, in the sense that there exist transaction sets that are feasible under a certain deadline assignment but not under the one computed by our heuristic.

For example, in the following case: $\mathcal{T}_1(\phi_1 = 18, D_1 = 14, T_1 = 25, C_{11} = 5, C_{12} = 1, C_{13} = 2), \mathcal{T}_2(\phi_2 = 16, D_2 = 28, T_2 = 30, C_{21} = 5, C_{22} = 4, C_{23} = 4), \mathcal{T}_3(\phi_3 = 4, D_3 = 8, T_3 = 10, C_{31} = 1, C_{32} = 1, C_{33} = 1)$, using the deadlines computed by Equation 8.7 would not allow feasibility to be proven under MDO-TO, while changing the following deadlines: $D_{11} = 7, D_{21} = 13, D_{31} = 2$ would make the system schedulable.

Another solution would be the development of an algorithm that iterates over the space of possible deadline assignments in search of a feasible one. However, holistic analysis makes it difficult to understand why deadlines have been missed, and we feel that developing an efficient algorithm for the general case is extremely difficult. We will instead provide a better heuristic and a reasonably fast search algorithm for the system model introduced in the following Chapter 9 in Section 9.6.

Chapter 9

Heterogeneous multiprocessor systems

9.1 Introduction

In the previous Chapters 7 and 8 we saw how the response time analysis for transaction systems can be used to provide improved 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. The specialized CPUs are typically used as hardware accelerators, or coprocessors: every task runs on the general purpose CPU but may suspend itself for a certain time, sending a computation chunk to a coprocessor and waiting for the coprocessor to end before resuming operations on the general purpose processor. A task that actually request a coprocessor is called a *DSP task*; for simplicity, we will assume that each DSP task requires a single fixed coprocessor. A common type of coprocessor is in fact a DSP, and its utility as an hardware accelerator has already been investigated [6, 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. However, we do not want to restrict ourselves to the case of DSPs only; other units, such as programmable video controllers, may be considered as coprocessors.

Each DSP task τ_i of our system model is represented by a period T_i, a relative deadline D_i, offset ϕ_i and three computation times C_i^a, C_i^b, C_i^c

with the following semantic: first a computation chunk of length C_i^a must be executed on the processor, then a computation chunk C_i^b is executed on the coprocessor (or a fixed coprocessor if there are more than one), and finally execution resumes on the processor for a time C_i^c. We assumes that synchronization between the processor and the coprocessor is done in zero time; we could, however, account for transmission delays by introducing delay values δ_i^{ab} and δ_i^{bc} between the finishing time of C_i^a and the release time of C_i^b and between the finishing time of C_i^b and the release time of C_i^c as in Section 8.2.

The scheduling problem for such a system has been fully analyzed under fixed priority [15], but no convincing solutions have been proposed so far for EDF to our knowledge. The distributed response time analysis, on the contrary, offers a nice solution to the problem, since it is possible to treat each original DSP task τ_i as a new transaction $\mathcal{T}_i$ with three different tasks τ_{ia}, τ_{ib} and τ_{ic} with execution times C_i^a, C_i^b and C_i^c respectively. Note that we are free to choose deadlines D_{ia} and D_{ib}.

We will discuss and show simulation results for three different cases. In the first one, we will suppose that each DSP task executes on a different coprocessor, that is, the number of coprocessors in the system is equal to the number of DSP tasks. In the second case, we assume a single preemptible coprocessor in the system, and in the third one, we will use a non-preemptible coprocessor; this is typically the case of DSP coprocessors. These cases are covered in Section 9.2, 9.4 and 9.3.

For each experiment, we generated 1000 task sets with 5 or 10 DSP tasks each and periods within 20 and 400 in the same way as in section 8.4, in the sense that the computation time of each task was divided according to a uniform distribution into the three components C_i^a, C_i^b, C_i^c. Whenever a transaction model is applied, we used algorithm MDO-TO to solve the holistic problem; for EDF processor demand analysis, we used the 1-fixed algorithm presented in Chapter 3.

Sections 9.5 and 9.6 cover further details while Section 9.7 offers some final thoughts.

9.2 Multiple Coprocessors

In this section, we will discuss the case where each DSP task executes on a different coprocessor. In the case where deadlines are less than or equal to the periods, whenever the coprocessors are preemptible or not does not

matter since no coprocessor can be simultaneously requested by more than one job. In this case, in fact, the coprocessor execution can be simply treated as a suspension time; that is, each DSP task τ_i first executes for C_i^a time units, then suspends itself for C_i^b time units, and after that executes for C_i^b. The transaction model for τ_i thus consists simply of tasks τ_{ia} and τ_{ic}, but with an added delay time $\delta_{ic} = C_i^b$. We can use Equation 8.8 to choose a good heuristic for D_{ia}.

Unfortunately, there is no way under EDF, apart from using the transaction model, to account for suspension times that do not occur before a job starts execution. Gai [15] gives good reasons why this is a difficult problem to solve. In fact, the only known way to prove feasibility for such a system under plain EDF is to consider the suspension time as part of the execution time of the task, which is clearly extremely pessimistic. A good feasibility analysis exists instead under fixed priority scheduling, and is shown in [20]. However, this analysis considers a synchronous task model, thus it fails to take the task offsets into account.

Figure 9.1 and 9.2 shows the simulation results for the system, expressed as a percentage of feasible task sets in respect to the total system utilization, for task sets with 5 and 10 tasks respectively. *1-fixed* is the 1-fixed processor demand analysis from Chapter 3, while *Kim* is the analysis developed in [20] under deadline monotonic scheduling. MDO-TO achieves a dramatic performance increase over 1-fixed at utilization around 1.0: the feasibility percentage for 1-fixed quickly drops to 0 while with the transaction analysis we are able to schedule almost every task. The performance under Kim is instead much better, but degrades around utilization 1.2 while MDO-TO has still a feasibility ratio of 50%; note that in fact, the 1-fixed analysis is so pessimistic that the results under fixed priority are much better, even if the Kim analysis does not take the task offsets into account. Finally, MDO-TO seem to work better as the number of tasks increases, while the other tests remain unchanged; this outcome should be fairly predictable from the experimental results of Section 8.4.

9.3 Preemptive Coprocessor

We will now suppose that the system offers a single coprocessor, but that such a coprocessor is preemptible. In this case, the transaction model consists of tasks τ_{ia}, τ_{ib} and τ_{ic} as in 9.1; deadlines are set according to Equation 8.7. The 1-fixed analysis is still valid in this case, and once again there is no valid method to reuse the coprocessor time on the main processor, meaning

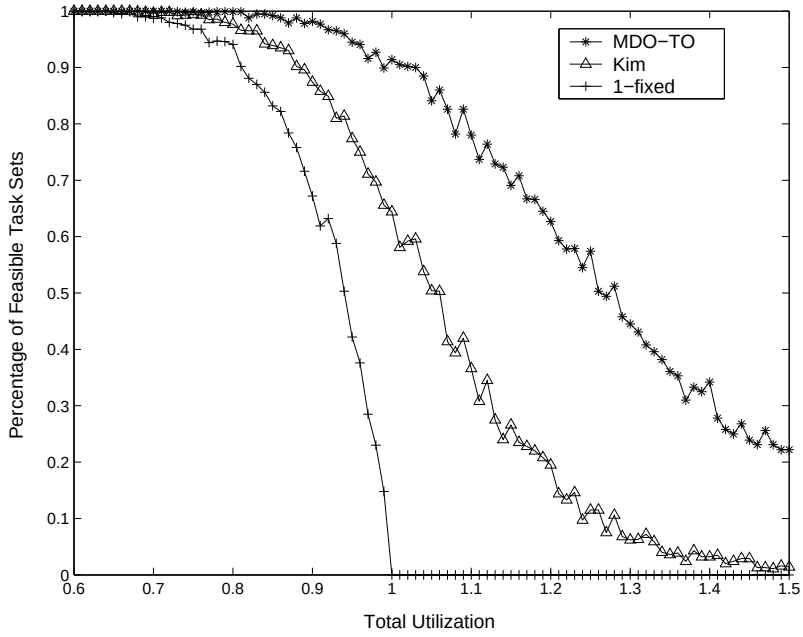

Figure 9.1: 5 DSP tasks, dedicated coprocessors

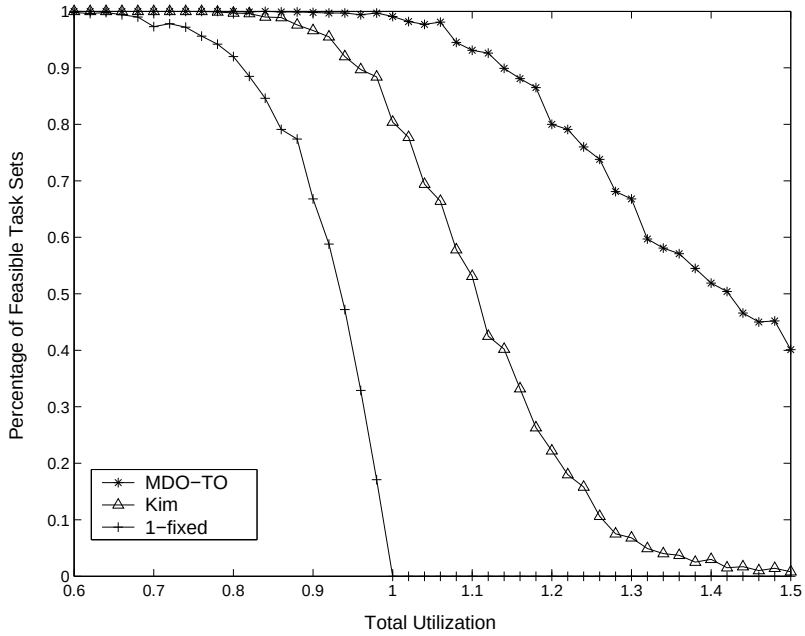

Figure 9.2: 10 DSP tasks, dedicated coprocessors

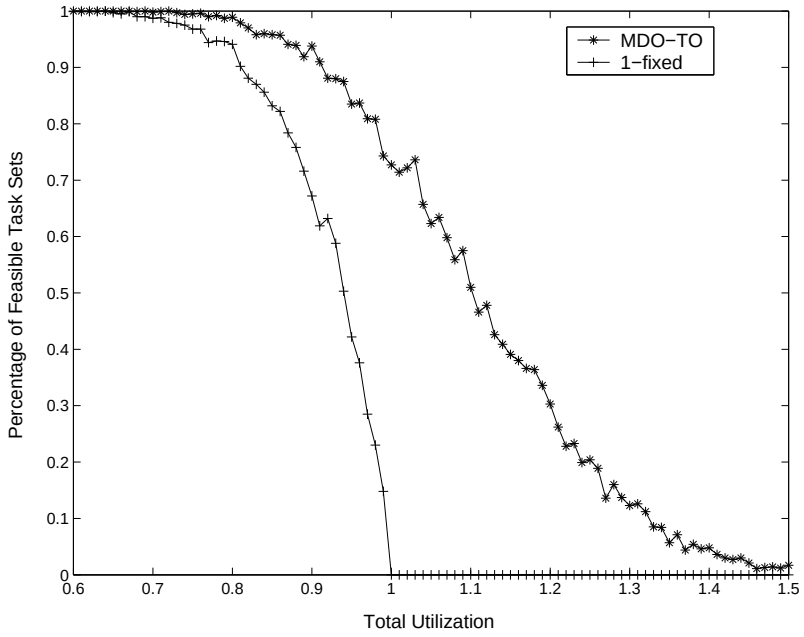

Figure 9.3: 5 DSP tasks, shared preemptible coprocessor

that we must treat each DSP task as a single task with computation time $C_{ia} + C_{ib} + C_{ic}$.

Figure 9.3 and 9.5 shows the simulation results for task sets with 5 and 10 tasks respectively; note that the 1-fixed test is the same as the one in the previous section. Once again, the performance of MDO-TO is extremely superior for utilizations around 1.0, but this time the feasibility percentage drops to 0 for utilizations around 1.5, while in the previous case, for 10 DSP tasks, the percentage was still at 40%.

9.4 Non Preemptive Coprocessor

In this final case, the coprocessor is assumed to be non preemptible. We can take care of this situation by supposing that each computation chunk on the coprocessor is executed inside a mutually exclusive critical section of length C_i^b. 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.

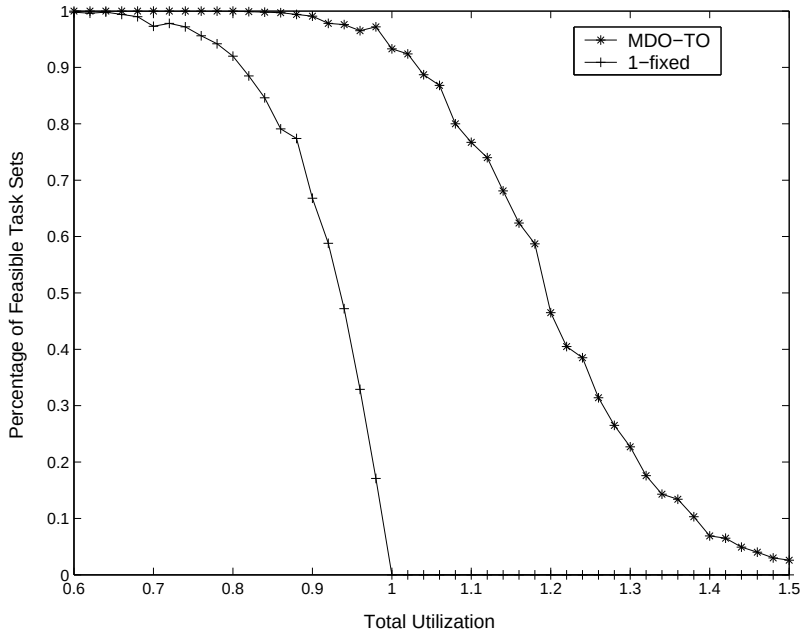

Figure 9.4: 10 DSP tasks, shared preemptible coprocessor

Under the simple task model, an extension of the 1-fixed test for resource usage has been introduced in Chapter 4.

The transaction analysis needs some in-depth considerations. If we use critical section as detailed above, only the second chunk of each transaction may experience blocking. Under this constraint, the blocking guarantees used in Section 7.2 and 7.3 are too pessimistic. In fact, since resource usage happens on the coprocessor only, we can develop guarantees similar to the ones used in the simple task model. This extension is detailed in Section 9.5. We would like, however, to stress an important fact here. The blocking times considered in 9.5 depends on task offsets. Since the offset based holistic analyses (CDO and MDO) change the offsets at each step, it follows that the blocking times change at each step too. This means that Theorem 12 does not hold anymore and thus we cannot prove that CDO performs better than WCDO, although this is very likely to happen. However, we can surely say that algorithm MDO converges if the response times are bounded since it is still monotonic.

A second issue consists in deadline placement. Using Equation 8.7 to choose deadlines D_{ia} and D_{ib} does not constitute a good heuristic anymore,

since we must take into account the blocking time. This means that if we were to use Equation 8.7, τ_{ib} would miss its deadline much more easily then τ_{ia} and τ_{ic}. A simple yet much more efficient heuristic is the following:

$$\begin{aligned} D_{ia} &= \frac{C_{ia}}{C_{ia}+C_{ib}+C_{ic}} D_i \\ D_{ib} &= \left(\frac{C_{ia}+C_{ib}}{C_{ia}+C_{ib}+C_{ic}} + \left(1.0 - \frac{C_{ia}+C_{ib}}{C_{ia}+C_{ib}+C_{ic}}\right)p\right) D_i \end{aligned}$$

This basically means that D_{ib} is increased of a number of time units proportional to some factor p; for $p = 0$ the equation is equal to the one in 8.7. We saw by simulations that $p = 0.8$ usually gives good results and will be consequently used in the following experiments, but even $p = 1.0$ is much better than $p = 0$.

We also tried to design a search algorithms that, starting from our improved heuristic, tries to find a suitable deadline assignment that makes the task set feasible. The algorithm uses some simple heuristics to move inside the search space and is quite fast: it is detailed in 9.6. Note that estimating the algorithm performance is quite hard, since computing, given a task set, if there is at least one deadline assignment that makes it schedulable under our transaction analysis takes way too much time for practical task sets; however, from results obtained from very small task sets we feel that our algorithm should be able to capture most task sets for which a feasible deadline assignment exists.

Figure 9.5 and 9.6 shows the simulation results for task 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 than the heuristic.

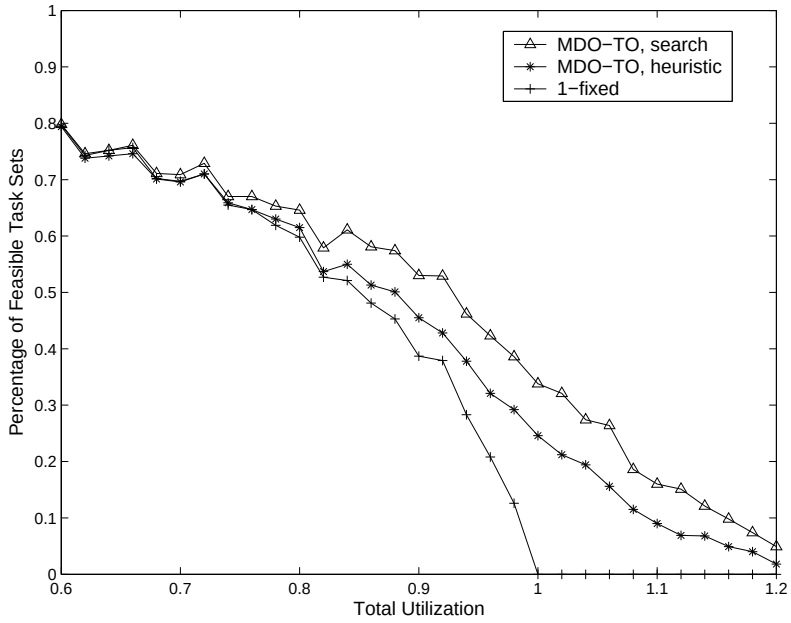

Figure 9.5: 5 DSP tasks, shared non preemptible coprocessor

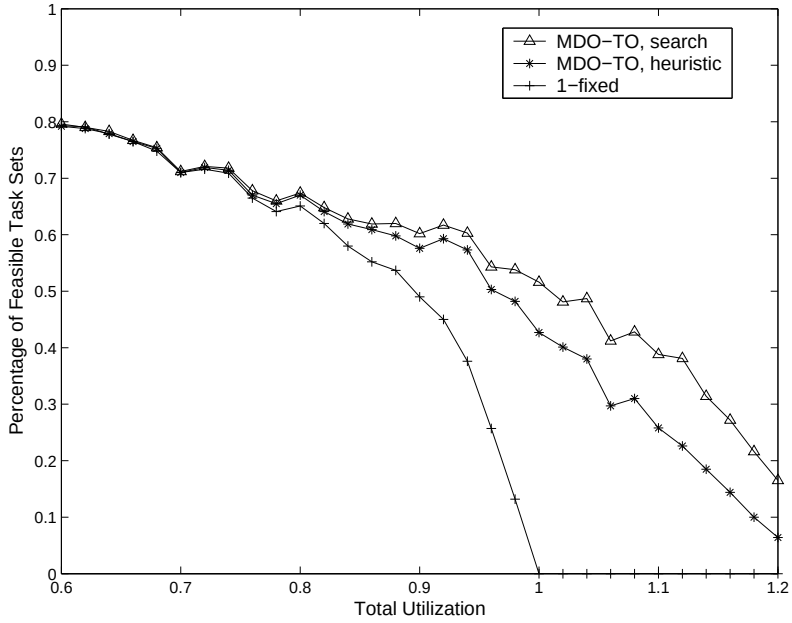

Figure 9.6: 10 DSP tasks, shared non preemptible coprocessor

9.5 Resource usage

In this section, we will extend the TO response time analysis to cover the problem of resource usage; our discussion will follow the same line as in Chapter 4. Tasks are assumed to share resources as detailed in Section 2.3.

We would like to use SRP as our resource access protocol. However, note that SRP, as all resource access protocols proposed for EDF, only works for uniprocessor systems. We will thus suppose that no resource is shared among tasks executed on different processors. Under this constraint, we can compute the worst-case response times by analyzing each processor independently as we proposed in Section 7.1.

Note that since no hypothesis is made on the offsets, the main properties of SRP remain true even for our transaction model if no release jitter is considered.

As in Section 4.2 we will first give some definitions and then prove our main theorem.

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

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

Proof. The proof is a simple extension of the one from Lemma 9. □

Definition 6 *Given an initial task τ_{pq}, we define the following dynamic preemption level:*

$$\pi_{pq}(t, D) = min(\{\pi_{ij} | \Delta_{pqij} + d_j \leq D \wedge \Delta_{pqij} < t\} \cup 2)$$

Definition 7 *Given an initial task τ_{pq}, we define the following dynamic maximum blocking time:*

$$B_{pq}(t, D) = max(\{C_{ijk} - 1 | d_{ij} > D + \Delta^{\phi_{ijk}}_{ijpq} \wedge \mathrm{ceil}(\rho_{ijk}) \geq \pi_{pq}(t, D)\} \cup 0)$$

Theorem 16 *The recurrence over the finishing time w^{pqA}_{ab} of task τ_{ab}, given initial task τ_{pq} and release time A for τ_{ab}, can be computed according to Equations 7.11 and 7.12 by the following substitutions:*

$$B_{ab} = B_{pq}(t, D)$$

Proof. The proof is an extension of both Theorem 4 and of Theorem 10. When blocking times are considered, the worst-case response time for task τ_{ab} can still be computed using the assumptions of Theorem 10, given that the release time of the initial task τ_{pq}, which we will consider to be 0 without loss of generality, corresponds not to the beginning of the busy period but to the last time that no job with deadline less or equal than D is active at the instant before. Let $\mathcal{A}$ be the set of jobs with deadline less than or equal to D that execute inside the busy period.

If blocking times are introduced, then it is possible for a single job τ_{ij}^l with deadline greater than D to be executed inside the busy period. For this to be possible, the job must be inside a critical section at time 0, since it must block some other job in $\mathcal{A}$. Note that there can only be one such job; otherwise some job in $\mathcal{A}$ would be blocked by two lower priority jobs, which is impossible.

Now consider job τ_{ij}^l. Since it is inside a critical section at 0, it must hold $a_{ij}^l + \phi_{ijk} < 0$ for some k, and $a_{ij}^l + d_{ij} > D$ since its deadline is greater than D. If the above conditions can hold, than they surely hold if we choose $a_{ij}^l = -\Delta_{ijpq}^{\phi_{ijk}}$, since it is the minimum possible time difference. Notice that in any case the maximum blocking time induced by critical section ξ_{ijk} is equal to $C_{ijk} - 1$, since τ_{ijl} can always be delayed by other tasks so that it enters ξ_{ijk} at -1. τ_{ijl} must be also able to block some job in $\mathcal{A}$, thus ceil(r_{ijk}) must be at least equal to the minimum preemption level of tasks in $\mathcal{A}$.

To end the proof, it now suffices to compute the minimum preemption level of tasks in $\mathcal{A}$. Note that if a task τ_{ij} is in $\mathcal{A}$, then its deadline must be less than or equal to D and thus $\Delta_{pqij} + d_{ij} \leq D$, and that the task must be included in the recurrence over t, that is $\Delta_{pqij} < t$. □

9.5.1 Busy period length

The dynamic maximum blocking time considered in the busy period length recurrence in Equation 7.9 must be slightly changed when blocking times are considered. Simply computing $B_{pq}(t, \infty)$ is incorrect since this would remove the possibility of any blocking time. Instead, B_{pq} must be adjusted to consider blocking time from tasks whose offset may be greater or equal to the computed length L at some step, that is:

$$B_{pq}(t,\infty) = max(\{C_{ijk}-1 | \exists k, \Delta_{pqik}+\phi_j-\phi_k \mathrm{mod} T_i \geq t \wedge \mathrm{ceil}(\rho_{ijk}) \geq \pi_{pq}(t,\infty)\} \cup 0)$$

Note that since we do not know which task is the starting one in transaction $\mathcal{T}_i$, we need to consider each possible starting task τ_{ik} to compute the first activation time of τ_{ij} inside the busy period.

9.6 Deadline search algorithm

Given a transaction set of the type used in Section 9.4, the deadline search algorithm tries to find an assignment for each deadline D_{ia}, D_{ib} that proves the feasibility of the set using algorithm MDO-TO modified as in Section 9.5 to take blocking times into consideration. Using a general optimum search algorithm such as simulated annealing is possible but inconvenient because we can make use of some good heuristics to move between solutions.

The algorithm iterates over the space of possible deadline assignments, choosing a new assignment at each step. Algorithm MDO-TO is then run until one of the following occurs:

1. MDO-TO converges to a solution where $\forall\ 1 \leq i \leq M, R_{ia} \leq D_{ia} \wedge R_{ib} \leq D_{ib} \wedge R_{ic} \leq D_{ic}$;
2. at some step $\exists\ 1 \leq i \leq M, R_{ia} > D_{ia} \vee R_{ib} > D_{ib} \vee R_{ic} > D_{ic}$

In the first case the deadline assignment is correct and the algorithm ends. In the second, the algorithm considers which deadlines are missed in the last step by MDO-TO, and then updates all deadlines using some simple heuristics that tries to make the tasks that missed their deadlines feasible. The heuristics used are detailed below in decreasing order of importance.

1. If τ_{ib} is not feasible, increase D_{ib} to give it more time to finish execution. Also, decrease D_{ia}, in order to have τ_{ia} finish earlier decreasing ϕ_{ib} as well.
2. If τ_{ia} is not feasible, increase its deadline D_{ia}.
3. If τ_{ic} is not feasible, decrease D_{ia}, so that both τ_{ia} and τ_{ib} should finish earlier. Also decrease D_{ib}, but do it with care, since τ_{ib} is sensible to blocking time.
4. If τ_{ib} is not feasible, increase the deadline D_{jb} of every task $j \neq i$, as this helps τ_{ib} finish earlier.
5. If τ_{ia} is not feasible, increase the deadline D_{ja} of every task $j \neq i$.

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. This helps achieving convergence preventing the algorithm from "swinging" back and forth between two different deadline assignments. 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;
```

$N(\epsilon)$ is a 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. This means that as the algorithm progresses the updates becomes smaller and more predictable.

Factor p is used to differentiate the 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.

9.7 Conclusions

In this section we introduced and discussed schedulability tests for heterogeneous multiprocessors based on the response time analysis developed in the previous chapters. We considered systems of DSP tasks that issue requests to either exclusive, shared preemptive or non-preemptive coprocessors. While actual systems with multiple coprocessors may be more complex, including non-DSP tasks, DSP tasks with exclusive access to a coprocessor and DSP tasks with shared access to either a preemptive or non preemptive coprocessor, the various techniques developed in each section provide a clear understanding of the underlying problems and may be composed to better analyze real systems. Extending the analysis to account for multiple coprocessor requests by each DSP task should be fairly simple.

As future work, we would like to extend our analysis making it possible for a task to issue a request not to a fixed coprocessor but instead to a pool of similar coprocessors. This case is more complex since it requires mixing our approach with conventional symmetric multiprocessor scheduling.

Chapter 10

Conclusions

In this work we introduced new schedulability tests for periodic task sets with offsets scheduled under EDF. We formally proved the correctness of our methods and provided experimental evaluations.

It is quite clear from our results that exploiting the offsets leads to better results in term of feasibility, output jitters and response times. Results are particularly significant for the schedulability problem of periodic task sets and especially for asymmetric multiprocessors of the type introduced in Chapter 9. In this letter case, the transaction-based analysis makes it possible to use a DSP (or another specialized processor) as a valid accelerator for real-time task sets scheduled under EDF.

Unfortunately, there seems to be no easy way to choose task offsets in an optimal or quasi-optimal way in an acceptable time. However, just trying a small number of randomly selected offsets for a task set seems to be enough to greatly improve its chance of being schedulable.

There are many areas of our work that may benefit from future work. We provide a brief list of the main ones.

- In Section 9.5 we supposed that all tasks sharing a given resource are executed on the same processor. In fact, while the problem of multiprocessor resource usage has been analyzed under Rate Monotonic [28, 27], no convincing solution exists for EDF to our knowledge. We would like to extend our transaction analysis to account for resources shared among tasks executed on different processors.

- The problem of deadline assignment for intermediate transaction tasks remain an open problem. We have provided both heuristics and a

search algorithm, but they had to be devised ad hoc for a specific system model. A more general approach may be useful.

- We have investigated how offsets can be assigned to obtain schedulability for simple task sets, but we have done nothing similar for the transaction model. Often the problem is not in determining whenever a system is schedulable or not but in selecting the tasks' parameters under suitable constraints. Unfortunately, the complexity of the holistic response time analysis could make such an attempt quite difficult.

- In the heterogeneous multiprocessor case, we supposed that each computation chunk was assigned to a fixed processor. This is clearly not always the case, since we could have a pool of similar coprocessor. The problem in this case is similar to that of symmetric multiprocessing, and usually requires the use of sub-optimal heuristics since the problem of processor assignment is proven to be NP-hard.

- It could be argued that our approach to DSP coprocessor scheduling is quite heavy from a computation point of view. Still, no method exists for exploiting the suspension time of tasks scheduled under EDF. An extension along the line of [15] could be useful, although it does not appear to be particularly easy to obtain.

- In Chapter 9 we saw how different coprocessor types (exclusive, shared preemptible and non-preemptible coprocessors) requires different analyses. Furthermore, the holistic response time analysis can be applied to a variety of distributed and multiprocessor systems. A verification tool could be developed in order to assess the schedulability problem for such systems in a simple way. We feel that a tool would be extremely useful to real-time systems developers.

Finally, note that Chapter 3 is also to appear in the proceedings of the 16^{th} Euromicro Conference on Real-Time Systems with the title: "A New Sufficient Feasibility Test for Asynchronous Real-Time Periodic Task Sets" [14].

Bibliography

[1] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard real-time scheduling: the deadline monotonic approach. In *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[2] T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

[3] S.K. Baruah, G.Buttazzo, S. Gorinsky, and G. Lipari. Scheduling periodic task systems to minimize output jitter. In *Proceedings of the International Conference on Real-Time Computing Systems and Applications*, pages 62–69, Hong Kong, December 1999.

[4] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182–190, December 1990.

[5] S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. *The Journal of Real-Time Systems*, 2, 1990.

[6] R. Baumgartl and H. Hartig. Dsps as fexible multimedia accelerators. In *Second European Conference on Real-Time Systems*, Paris, France, September 1998.

[7] E. Bini and G. C. Buttazzo. Biasing effects in schedulability measures. In *16th Euromicro Conference on Real-Time Systems*, Catania, Italy, June 2004.

[8] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe M. Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *Proceedings of the* 13^{th} *IEEE Euromicro Conference on Real-Time Systems*, pages 59–66, Delft, The Nederlands, June 2001.

[9] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.

[10] Anton Cervin. Improved scheduling of control tasks. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 4–10, York, UK, June 1999.

[11] Anton Cervin and Johan Eker. Control-scheduling codesign of real-time systems: The control server approach. *Journal of Embedded Computing*, 2004. To appear.

[12] M. Chen and K. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2, 1990.

[13] M. L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 1974.

[14] R. Pellizzoni e G. Lipari. A new sufficient feasibility test for asynchronous real-time periodic task sets. In *16th Euromicro Conference on Real-Time Systems*, Catania, Italy, June 2004. To appear.

[15] P. Gai and G. C. Buttazzo. Multiprocessor dsp scheduling in system-on-a-chip architecture. In *14th Euromicro Conference on Real-Time Systems*, June 2002.

[16] Joel Goossens. Scheduling of offset free systems. *Real-Time Systems Journal*, 5:1–26, 1997.

[17] R.L. Graham. Bounds on the performance of scheduling algorithms. In *Computer and Job Scheduling Theory*, pages 165–227. John Wiley and Sons, 1976.

[18] J. Lehoczky J. K. Strosnider, T. Marchok. Advanced real time scheduling using the IEEE 802.5 token ring. In *IEEE Real Time System Simposium*, Huntsville, Alabama, USA, 1988.

[19] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 89–99, Phoenix, December 1992.

[20] In-Guk Kim, Kyung-Hee Choi, Seung-Kyu Park, Dong-Yoon Kim, and Man-Pyo Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *Real-Time Computing Systems and Applications*, pages 54–59, October 1995.

[21] J.Y.-T. Leung and M.L. Merril. A note on preemptive scheduling of periodic real-time tasks. *Information Processing Letters*, 3(11), 1980.

[22] J.Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2, 1982.

[23] Giuseppe Lipari and Giorgio Buttazzio. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *System Architecture*, 46:327–338, 2000.

[24] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[25] J.C. Palencia and M. Gonzáles Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[26] J.C. Palencia and M. Gonzáles Harbour. Offset-based response time analysis of distributed systems scheduled under EDF. In *15th Euromicro Conference on Real-Time Systems*, Porto, Portugal, July 2003.

[27] R. Rajkumar. Synchronization in multiple processor systems. In *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[28] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real Time System Simposium*, 1988.

[29] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), September 1990.

[30] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report RR-2772, INRIA, France, January 1996.

[31] M. Spuri. Holistic analysis for deadline scheduled real-time distributed systems. Technical Report RR-2873, INRIA, France, April 1996.

[32] J.A. Stankovic. Real-time computing systems: The next generation. In J. Stankovic and K. Ramamritham, editors, *Tutorial on Hard Real-Time Systems*. IEEE Computer Society Press, 1988.

[33] J.A. Stankovic and K. Ramamritham. What is predictability for real-time systems? *Journal of Real-Time Systems*, 2, 1990.

[34] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real Time Systems*, 6(2):133–151, Mar 1994.

[35] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 50:117–134, April 1994.