Component-based analysis and synchronization of distributed transactions scheduled by EDF

Nicola Serrelli
ReTiS Lab.
Scuola Superiore Sant’Anna

A thesis submitted for the degree of

Doctor of Philosophy

Supervisor: Prof. Giuseppe Lipari

January 22nd 2010
Thanksgiving

Thanks to my family and all friends.

Thanks to Giuseppe Lipari and Enrico Bini for their support.

Thanks to Michael Gonzalez Harbour for corrections and suggestions.

Thanks to Marko Bertogna for suggestion about speedup the execution of dbf algorithm.
abstract

In this thesis we address the problem of designing and analyzing distributed real-time systems modeled as sets of real-time transactions. Real-time transactions are often used to model distributed applications in many fields such as multimedia soft real-time systems, and embedded real-time systems. Transactions are sequences of tasks activated periodically. All tasks must complete before the transaction end-to-end deadline set by the designer. In this thesis, we propose to analyze each transaction in isolation, as they are components of the system. The advantages of such approach are manyfold: the analysis is based only on parameters related to the single transaction under study, then any change in another component of the system does not affect the validity of results of current transaction analysis; also the analysis of every single component, followed an aggregation of the analysis outcomes, is usually simpler than the study of the whole system.

In this thesis we make three distinct novel contribution to the state of art: 1) sporadic dbf: we found that the scenario of strictly periodic activations is not the worst when the transactions are activated sporadically. For this reason we extend the demand bound function (dbf) to sporadic transactions and we propose a suitable schedulability analysis. 2) IDSP: we propose algorithm IDSP (Implicit Deadline Synchronization Protocol) to assign the absolute deadlines to jobs at run-time. The protocol does not require synchronization between nodes and uses only local information. We guarantee that the demand that can be generated at run-time is always bounded by the sporadic dbf computed off-line. 3) ORDER: investigate a new algorithm (called ORDER) to assign intermediate deadlines in order to minimizes the resource requirement of a single transaction, presenting both theoretical and experimental results.
4.5. Simulations 80
Chapter 5. Conclusions 87
Bibliography 89
List of Figures

1.1 An example of distributed transactions 11
1.2 An example of virtual nodes, using \((\alpha, \Delta)\) servers 12
1.3 Notation for tasks. 18
1.4 moving left a deadline. 21
1.5 moving right a job. 21
1.6 moving right a transaction. 22

2.1 Example of demand bound function. 24
2.2 Parameters for the example 26
2.3 Example of sporadic transaction. 26
2.4 An example of sporadic dbf. 28
2.5 Algorithm for computing the dbf. 28
2.6 Algorithm for enumerating intervals. 31
2.7 Interval for offset of past instances of transaction. 32
2.8 Algorithm for generating \(\Gamma^-\) and \(\Gamma^+\). 36
2.9 Simulations with 4 processors (times in seconds) 38
2.10 Simulations with 8 processors (times in seconds) 39
2.11 Example of relation among \(\Phi^\ell, t_0\) and \(t_1\), for future instances. 41
2.12 Algorithm for overestimate a subtree. 42
2.13 Execution time for computing the dbf on 4 processors (times in seconds) with a small optimization. 42
2.14 Fast execution times on 4 CPUs. 43

3.1 Precedence set and relation among tasks. 46
3.2 Example of computation of the precedence set. 48
3.3 Example of computation of the precedence set. 49
3.4 Minimize the distance between deadline and activation of each job. 55
3.5 Induction on the smallest index of transaction. 56
3.6 Induction on a generic index of transaction (part 1). 58
3.7 Induction on a generic index of transaction (part 2). 60
3.8 Example of computation of the precedence set. 61
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.9</td>
<td>Chain among moved jobs.</td>
<td>62</td>
</tr>
<tr>
<td>3.10</td>
<td>Enforce distance among transaction activations.</td>
<td>63</td>
</tr>
<tr>
<td>4.1</td>
<td>The optimization algorithm.</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>The minimization of the demand bound function.</td>
<td>69</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of dbf, in the arbitrary deadline case.</td>
<td>72</td>
</tr>
<tr>
<td>4.4</td>
<td>Behavior of ORDER, NORM and PURE on the first example.</td>
<td>77</td>
</tr>
<tr>
<td>4.5</td>
<td>Behavior of ORDER, NORM and PURE on the second example.</td>
<td>78</td>
</tr>
<tr>
<td>4.6</td>
<td>Parameters of the example</td>
<td>80</td>
</tr>
<tr>
<td>4.7</td>
<td>ORDER on 2 processors, with various transaction length</td>
<td>81</td>
</tr>
<tr>
<td>4.8</td>
<td>ORDER vs PURE on 2 processors with 6 tasks</td>
<td>81</td>
</tr>
<tr>
<td>4.9</td>
<td>ORDER vs. PURE on 4 processors with 8 tasks</td>
<td>83</td>
</tr>
<tr>
<td>4.10</td>
<td>ORDER vs Simulated annealing on 2 processors</td>
<td>83</td>
</tr>
<tr>
<td>4.11</td>
<td>Example: optimizing with different weights</td>
<td>84</td>
</tr>
<tr>
<td>4.12</td>
<td>First simulation experiment.</td>
<td>86</td>
</tr>
<tr>
<td>4.13</td>
<td>Second simulation experiment.</td>
<td>86</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

1.1. Transactions

Distributed real-time systems are often modeled as a set of real-time transactions. The transaction model was first proposed by Tindell et al. in [TBW94]. Each transaction is a chain of real-time tasks, and each task is allocated on a (possibly different) processing node. The first task in the transaction is activated periodically, or by external events characterized by a minimum inter-arrival time. The other tasks are activated upon the completion of the preceding one. All tasks in the transaction must complete within an end-to-end deadline relative to the activation time of the transaction. We allow the end-to-end deadline to be larger than the period. This situation is quite common in real applications. For example, in multimedia streaming, the period at which video frames are generated and processed may be lower than the end-to-end deadline for delivering the frames to the user.

![Figure 1.1. An example of distributed transactions](image)

1.2. Analysis

An important problem is to check the schedulability of the system, i.e. to test if all transactions will complete before their end-to-end deadline under worst-case conditions. In fixed priority systems, the holistic analysis [TBW94, PGH98, PEP02, MTN08] consists in reducing the overall distributed schedulability problem into p single-node problems that can be solved using classical schedulability analysis. Each task is assigned a priority, and task parameters like offsets, jitters, response times are calculated so that the precedence constraints are automatically guaranteed. Since all schedulability problems depend on one another (i.e. the activation of an intermediate task, and hence its jitter, depends on the response time of the preceding task), the analysis is iterated until either a fixed-point solution is found or the set is deemed not schedulable. Similar techniques have been applied to EDF scheduling [Spu96, PH03, PL07, RGR08]. In this case, each
task must be assigned an intermediate deadline instead of a fixed priority. Holistic analysis also allows to mix different schedulers on different nodes, as long as the designer is able to compute the worst-case response time of every task.

1.3. Component-based design

The holistic analysis is global, in the sense that it can only be performed once the designer knows the parameters of all transactions. Moreover, any variation of one parameter of a transaction (computation time, priority of an intermediate task, . . . ) can influence the temporal behavior of any other transaction.

In a component-based approach, instead, it is desirable to perform the analysis in two steps: in the first step (local) we analyze each transaction in isolation, summarizing its temporal behavior with a (possibly small) set of temporal parameters. In the second step (integration), we must verify that the overall system is schedulable by only considering the temporal parameters derived in the first step. Such an approach facilitates sensitivity analysis, increases robustness of the solution, allows to easily substitute one component with another, reduces the complexity of dynamic on-line admission control, etc.

At run-time, an appropriate scheduler ensures that each transaction is temporally isolated from the (mis-)behaviors of the others. For example, aperiodic server algorithms and hierarchical scheduling [FM02, SL03, LB05] can successfully cope with this problem. In these systems, each application (or component) is handled by a different server. The model has been extended to distributed systems in [LLB06].

In distributed systems, a natural choice is to consider a transaction as a basic block of the component-based methodology. Therefore, the designer could be interested in designing, implementing and analyzing each transaction independently of the others. In our model, an application is scheduled on a set of virtual nodes. Each virtual node is assigned a temporal partition on one processor. Of course, it is important to minimize the computational resources required by each temporal partition so to maximize the likelihood of fitting more partitions into one physical processor.

![Figure 1.2. An example of virtual nodes, using (α, Δ) servers](image)

Usually, designers set the end-to-end deadline $D$ of a transaction, while the relative deadlines of all intermediate tasks are computed according to some heuristic. This problem is referred as the deadline assignment problem [DNS94, JS96]. It is clear that such assignment may influence the schedulability of the entire system: if a task is assigned a large intermediate deadline, it may finish too late leaving too short time for the following tasks to complete before the end-to-end deadline.
1.4. Analysis in isolation

When we analyze a transaction in isolation, it is not possible to use the response times of the tasks, because they depend on the presence of all other transactions. Therefore, in this thesis we use the slicing approach [DNS94]; each task is assigned an execution window, using appropriate offsets and deadlines, and the execution windows of any two tasks of the same transaction are not overlapping.

Following the slicing method, under EDF the temporal characteristics of each transaction are summarized by a set of demand bound functions (dbfs) [BHR90], one per each node. The integration analysis then consists, for every node, on summing all the dbfs related to each transaction and check that the resulting function never exceeds the computational power of the node.

However, two problems need to be solved to apply the slicing methodology to EDF-scheduled transactions. The first problem concerns with extending the analysis to sporadic transactions. In fact, unlike single tasks scheduled on uniprocessors, if the end-to-end deadline is larger than transaction’s period, then the worst case of sporadic transaction is not in the strictly periodic case, as it will explained in Section 2.3.

The other problem concerns with the run-time support. When scheduling transactions with EDF, we must assign an activation offset and an absolute deadline to each job. These, in turn, depend on the activation of the transaction, which may happen on a different node. Moreover, if activations are sporadic, we cannot predict the activation window of future jobs. Therefore, it may seem that a precise global clock synchronization protocol is needed. Fortunately, the global clock synchronization can be avoided in fixed priority systems by using the Release Guard Protocol [SL96]. The idea is to release the constraint on separating the execution windows of jobs residing on different nodes, guaranteeing instead only the correct separation between two jobs on the same processor. The protocol is simple and effective, but it was conceived for fixed priority schedulers only. In Chapter 3 we will show how to extend this protocol in order to work under EDF, without violate the result of the off-line analysis done with sporadic dbf.

1.5. Related work

1.5.1. Schedulability analysis of transactions. A possible strategy to tackle the schedulability analysis of transactions is to renounce to optimality altogether and simplify the problem. The most viable solution is to use one of the sufficient tests for schedulability analysis proposed in [PH03, PL07]. For example, once intermediate deadlines have been assigned, Pellizzoni and Lipari [PL07] propose to transform the distributed transaction problem into \( p \) single-node problems. To this end, the problem is over-constrained by imposing that all tasks must complete before their deadlines.

In classic holistic analysis, the best case response time is needed to estimate either the task jitter [TBW94, PGH98] or the task offset [PL07]. Several techniques have been proposed by for the best-case response time by Redell and Sanfridson [RS02] and Bril et al. [BCGG09].

An alternative method was proposed by Rahni et al. [RGR08]. The method consists in splitting the overall analysis in two steps: in the first step the aggregate demand bound function of a transaction is computed; in the second step, all the aggregate demand bound functions are added to test the overall schedulability.
The use of the demand bound function was initially proposed by Baruah et al., for testing the schedulability of set of tasks scheduled by EDF on single processors [BHR90]. This methodology is also known as “Processor Demand Criterion” [But04].

The computation of the dbf was later extended to more complex task models, such as the generalized multi-frame tasks [BCGM99]. Recently, Zhang and Burns [ZB08] proposed a technique to reduce the number of points to check during analysis based on demand bound function.

The processor demand criterion has been extended to the analysis of distributed real-time transactions by Rahni et al. [RGR08]. However, their methodology is still based on the holistic analysis: the activation time of a task is set equal to the finishing time of the previous task in the transaction.

In [DNS94] authors proposed a methodology to analyze the schedulability of task graphs. The methodology also computes intermediate deadlines by using an heuristic approach, and it is based on the slicing approach: each task is assigned a slice that does not overlap with the slices of other tasks. Later [Jia06] uses time slices to decouple the schedulability analysis of each node, reducing the complexity of the analysis. Such an approach improves the robustness of the schedule, and allows to analyze each transaction in isolation from the others, because the activation of a task does not depend on the completion time of the previous task, which in turn depends on the presence of other transactions and their parameters.

A notable alternate analysis was proposed by Jayachandran and Abdelzaher [JA08], who developed several transformations to reduce the analysis of a distributed system to the single processor case. However, in their analysis, the isolation between transactions is not ensured.

Unfortunately, when some task is not schedulable, again we have no clue on how to modify intermediate deadlines to make the task schedulable. Classical algorithms for checking schedulability on every node have pseudo-polynomial complexity [BHR90,PL05] and return a yes/no response. Then, it would be necessary to explore the entire space of possible deadline assignment.

Several authors have proposed methods for performing sensitivity analysis on task deadlines [BRC06,HBJK06,BB07]. Balbastre et al. [BRC06] and Hoang et al. [HBJK06] proposed algorithms for deriving the maximum admissible variation of task deadlines that can be sustained by the task set. Bini and Buttazzo [BB07] described the geometry of the space of all feasible deadlines. However, these methodologies are valid only for independent tasks and not for transactions.

We remark that, similarly to what it happens in multiprocessor scheduling [BP09], activating the transactions as early as possible (i.e. periodically) is not the worst-case for the activation pattern. In Section 2.3 we show this by an example.

### 1.5.2. Time synchronization

Regarding the run-time support for scheduling transactions in distributed systems, Gantman et al. [GGLR98] presented a survey on synchronization protocols for real-time distributed systems. Among the many algorithms presented in the survey, the Release Guard Protocol (originally proposed in [SL96]) achieves a smaller average end-to-end response time, greatly reduces start-time jitter, and does not require a global clock synchronization. The protocol uses only local information regarding the minimum separation time between instances of the same task, and
appropriately delays future instances so to guarantee that higher priority tasks do not interfere too much with lower priority tasks. However, the Release Guard Protocol only works with fixed priority schedulers, and assumes that the end-to-end deadline does not exceed the period. The protocol has been enhanced by Zhang et al. [ZKG⁺08] to deal with sporadic transactions, again on fixed priority schedulers.

1.5.3. Initial deadline assignment. In [DNS94,JS96], the authors propose to use separate windows of execution for the tasks in a transaction. Each task is assigned a release time coincident with the deadline of the previous task in the transaction, and all tasks are required to complete before their absolute deadline. By using this technique (also called slicing) the problem is over-constrained with respect to the original model used in holistic analysis. However, as reported in [JS96], it also eliminate release jitter and allows independent analysis of different transactions. Therefore, we will follow this approach in our work.

On the deadline assignment problem, two papers [DNS94,KGM97] independently proposed two similar assignment strategies. The first idea is divide the end-to-end deadline proportionally to the computation time of all tasks, as follows

\[ D_i = D \frac{C_i}{\sum_j C_j} \]

This method is called NORM in [DNS94] and it is the most widely used in the literature.

A slightly different method is based on the distribution of the laxity equally among all tasks:

\[ D_i = C_i + \frac{D - \sum_j C_j}{n} \]

This method is called PURE in [DNS94]. However, as we will show, both these assignments require a larger amount of resources than our proposed technique.

1.6. Research goals and contributions of this thesis

In this work, we approach the problem of deadline assignment from a different perspective. When designing an embedded system, it is common practice to divide the system into components, and then design and develop each component independently. In distributed systems, a natural choice is to consider a transaction as a basic block of the component-based methodology. Therefore, the designer could be interested in designing, implementing and analyzing each transaction independently of the others.

As a matter of facts, optimizing schedulability (thus maximizing the processor utilization in the different nodes) is not always the main concern of the designer. In addition, it would be important to investigate other system properties, like the robustness to variation of the parameters (execution times, inter-arrival times), the possibility to reduce overall power consumption, etc.

From this point of view, independent analysis of each transaction separately gives some significant advantages over global analysis:

- by analyzing each transaction in isolation, it is easier to understand the contribution of the transaction to the load of the system;
• the system designer has the possibility to “guide” the deadline assignment problem to reach a different goal;
• it is easier to identify the “problem node”, i.e. the node that acts as a bottleneck, and modify the deadline assignment so to avoid the problem.

The original contributions of the presented research are:
• an algorithm to correctly compute the dfbfs of sporadic transactions on each node.
• Algorithm IDSP (Implicit Deadline Synchronization Protocol) to assign absolute deadlines to jobs in an EDF scheduler. Our protocol guarantees that, under certain conditions, the run-time demand of the jobs on each node never exceeds the dfb computed off-line.
• a new method to assign intermediate deadline for a single transaction, in order to minimize the resource requirements of that transaction. We show also that this method, called ORDER ensures a higher resource usage than the existing ones and works also with sporadic transactions. Moreover, We used simulated annealing to find the optimum deadline assignment that minimizes the resource requirement of a transaction in isolation and we compared that result with the global approach of holistic analysis.

1.7. System model

A distributed real-time application is modeled by a set of $m$ transactions $\{T_1, \ldots, T_m\}$. The transactions are non-concrete, meaning that the relative offsets between the transactions is not known a priori. To simplify the presentation, since our work investigates each transaction in isolation, throughout this thesis we drop the index of the transactions.

Transaction $T$ is composed by a set of $n$ tasks $\{\tau_1, \ldots, \tau_n\}$. Each task $\tau_i$ has a computation time $C_i$, while $C = \sum_{i=1}^{n} C_i$ is the overall computation time of the transaction.

The first task $\tau_1$ of the $\ell^\text{th}$ instance of the transaction is activated at $\Phi_\ell$, that is called absolute activation, while other tasks $\tau_i$ with $i > 1$ are activated upon the completion of the preceding one. We denote by $\tau_\ell^i$ the $\ell^\text{th}$ instance of the task $\tau_i$. We consider sporadic transactions with minimum inter-arrival time $T$. Hence we have

$$\Phi_\ell - \Phi_{\ell-1} \geq T. \quad (1.3)$$

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

The successive instances will be denoted by positive indexes $\ell > 0$, and their absolute activations by $\Phi^1, \Phi^2, \ldots$ Similarly, the previous instances will be denoted by negative indexes $\ell < 0$, and their absolute activations by $\Phi^{-1}, \Phi^{-2}, \ldots$

The following vector represents the sporadic activation pattern:

$$\overline{\Phi} = (\Phi^{-k_0}, \ldots, \Phi^{k_1}) \quad (1.4)$$

where $k_0$ and $k_1$ depend on the number of instances we need to consider in the analysis of the instance 0, i.e. the maximum number of instances of transaction that may overlap
1.7. SYSTEM MODEL

with the interval under study (for more details see Section 2.4 and Section 2.7). Finally, \( \Gamma \) is the set of all possible sporadic activation patterns.

The utilization of the transaction is defined as \( U = \frac{C}{T} \). Each transaction \( T \) has an end-to-end deadline \( D \) that is the maximum tolerable time from the activation of the first task \( \tau_1 \) to the completion of the last task \( \tau_n \). Since the analysis of the constrained deadline \( (D \leq T) \) is a straightforward extension of the classic analysis, throughout the work we always assume \( D > T \). In such a case, it may happen that a task is activated before its previous instance has completed. In the presented research, we assume that the different activations of each task are served in a FIFO order.

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

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

Each task is assigned an intermediate deadline \( \overline{D}_i \), that is the interval of time between the activation of the transaction and the absolute deadline of the task. Hence, using the notation introduced so far, the absolute deadline of the \( \ell \)-th instance of \( \tau_i \), is

\[
(1.5) \quad d_i^\ell = \Phi^\ell + \overline{D}_i.
\]

We enforce the precedence relationship between tasks by the slicing technique [DNS94]: for each task we set the activation offset \( \phi_i \), relative to the activation of the transaction \( \Phi^\ell \), equal to the intermediate deadline of the preceding one:

\[
(1.6) \quad \phi_1 = 0, \quad \phi_i = \overline{D}_{i-1} \quad i = 2, \ldots, n
\]

Clearly, the task absolute activation is

\[
(1.7) \quad \phi_i^\ell = \Phi^\ell + \phi_i.
\]

Moreover, we define the task relative deadline \( D_i \) as

\[
D_i \overset{\text{def}}{=} \overline{D}_i - \phi_i.
\]

The relationship between activation offsets and relative deadlines is depicted in Figure 1.3. Clearly,

\[
(1.8) \quad \sum_{i=1}^{n} D_i = D
\]

In the analysis, we will assume that the \( \ell \)-th instance of task \( \tau_i \), denoted by \( \tau_i^\ell \), is allowed to execute only within \( [\Phi^\ell + \phi_i, \Phi^\ell + \overline{D}_i] \). We will show in Section 2.2 how to relax this constraint.

The slicing assumption (Equation 1.6) allows us to decouple the design and analysis of each different transaction, since the activations depend only on parameters of the
same transaction (deadlines), and not on the completion times, which in turn depend on the interference from other transactions.

In Chapter 3 we study the runtime behavior and compare it with the off-line analysis. To do that, we define the actual absolute activation time $a_i^\ell$ as the instant when $\tau_i^{\ell}$ is allowed to execute, while $f_i^\ell$ identifies the actual absolute finishing time of job $\tau_i^{\ell}$ and clearly, if the system is schedulable, $f_i^\ell \leq \Phi_i^\ell + \overline{D}_i$. To satisfy the order among jobs of the same instance of transaction and respect at the same time the off-line analysis we allow $a_i^\ell$ to be:

$$a_i^\ell = \Phi_i^\ell \quad i = 1$$

$$a_i^\ell \in \left[ f_{i-1}^\ell, \Phi_i^\ell + \phi_i \right] \quad i > 1.$$

The values of $T, \Phi, D, C, \overline{D}, \phi_i$ (and related symbols like $f_i^\ell$) are all real numbers. Finally, we use the notation $(\cdot)_0 \overset{\text{def}}{=} \max\{0, \cdot\}$.

### 1.8. Notation

Here we summarize, for case of reference, the notation used throughout the document.

- $p$ number of processing nodes
- $m$ number of transactions
- $\{T_1, \ldots, T_m\}$ all transactions in the systems
- $U_{\text{tot}}$ total utilization
- $T$ the transaction under analysis
- $n$ number of tasks in $T$
- $T_k$ subset of tasks in $T$ mapped on node $k$
- $n_k$ number of tasks in $T_k$
- $D, D', D^*$ end to end deadline of $T$
- $C$ overall computation time of $T$
\( C^k \) overall computation time of \( \mathcal{T}_k \)
\( T \) minimum inter-arrival time of \( \mathcal{T} \)
\( U \) utilization of \( \mathcal{T} \)
\( U_k \) utilization of \( \mathcal{T}_k \)

\( \tau_i \) \( i \)th task in \( \mathcal{T} \)
\( x_i \) processing node of \( \tau_i \)
\( C_i \) computation time of \( \tau_i \)

\( D_i, D'_i \) relative deadline of \( \tau_i \) (relative to task activation)
\( \overline{D}_i \) intermediate deadline of \( \tau_i \) (relative to transaction activation)
\( \phi_i \) activation offset of \( \tau_i \) (relative to transaction activation)

\( w, w' \) value of \( \text{dbf} \) for node \( k \) and a specific interval \( t \)
\( S_k, S'_k \) scheduling points related to \( \text{dbf} \) for node \( k \)
\( S^p_k \) finite periodic part of \( S_k \)
\( F_k \) finite transient part of \( S_k \)

\( \Phi^\ell \) absolute activation time of \( \ell \)th instance of \( \mathcal{T} \)
\( a^\ell \) actual absolute activation time of \( \ell \)th instance of \( \mathcal{T} \)
\( \Phi^\ell_f \) equivalent absolute activation time of \( \ell \)th instance of \( \mathcal{T} \)

\( -k_0 \) the most negative index of instance of \( \mathcal{T} \) considered by analysis
\( k_1 \) the largest index of instance of \( \mathcal{T} \) considered by analysis
\( k_2 \) the maximum number of future instances of \( \mathcal{T} \) considered by analysis

\( \Phi, \Phi' \) sporadic activation pattern, \( \{ \Phi^{-k_0}, \ldots, \Phi^{k_1} \} \)
\( \Gamma \) set of all possible \( \Phi \)
\( \Gamma^- \) subset of \( \Gamma \) related only to past instances of \( \mathcal{T} \), i.e. with negative indexes
\( \Gamma^+ \) subset of \( \Gamma \) related only to future instances of \( \mathcal{T} \), i.e. with positive indexes

\( \tau_i^\ell \) \( \ell \)th instance of \( \tau_i \)
\( \phi_i^\ell \) off-line absolute activation time of \( \ell \)th instance of \( \tau_i \)
\( d_i^\ell, d_i'^\ell \) off-line absolute deadline time of \( \ell \)th instance of \( \tau_i \)
\( a_i^\ell, a_i'^\ell \) actual absolute activation time of \( \ell \)th instance of \( \tau_i \)
\( f_i^\ell, f_i'^\ell \) actual absolute finishing time of \( \ell \)th instance of \( \tau_i \)
\( \phi_i^\ell_f \) equivalent absolute activation time of \( \ell \)th instance of \( \tau_i \)
\( d_i^\ell_f \) equivalent absolute deadline time of \( \ell \)th instance of \( \tau_i \)
\( d_{\text{max}} \) the biggest value among a set of equivalent absolute deadline time (i.e. one of \( d_i^\ell_f \))

\( t_0, t'_0 \) start of the interval under analysis
\( t_1, t'_1 \) end of the interval under analysis
\( t, t', t^* \) \hspace{1cm} \text{length of the interval under analysis}
\( \xi_k \) \hspace{1cm} \text{constant used to reduce the slope } \alpha_k
\( \mathcal{P}_i^\ell \) \hspace{1cm} \text{precedence set of job } \tau_i^\ell
\( \mathcal{P}_i^{\prime \ell} \) \hspace{1cm} \text{reduced precedence set of job } \tau_i^\ell
\( \mathcal{J} \) \hspace{1cm} \text{subset of jobs in } T_k \text{ completely inside an interval } [t_0, t_1]
\( \mathcal{J}^\ell \) \hspace{1cm} \text{subset of jobs in } \mathcal{J} \text{ related to the instance of transaction } \ell
\( \mathcal{C}_i^\ell \) \hspace{1cm} \text{sequence of jobs in } \mathcal{J} \text{ that forces the minimum value of } \underline{d}_k^\ell

\( s_k(i) \) \hspace{1cm} \text{a function that sorts the tasks in } T_k \text{ by increasing computation time}
\( \alpha(S_k), \alpha_k \) \hspace{1cm} \text{slope of a line passing from the origin that can upper bound } S_k, \text{ i.e. } \text{dbf}_k(t)
\( \Delta(S_k), \Delta_k \) \hspace{1cm} \text{horizontal displacement of a line with slope } \alpha_k \text{ that can upper bound } \text{dbf}_k(t)
\( Z(t) \) \hspace{1cm} \text{function } \alpha_k(t - \Delta_k)0 \text{ that is a linear upper bound for } \text{dbf}_k(t)
\( h(\cdot), h(\alpha_1, \ldots, \alpha_p, \Delta_1, \ldots, \Delta_p) \) \hspace{1cm} \text{cost function used to optimize deadlines}

\( (\cdot)_0 \) \hspace{1cm} \text{max}\{0, \cdot\}
\( \text{df}_k(t_0, t_1) \) \hspace{1cm} \text{demand function for node } k \text{ in the interval } [t_0, t_1]
\( \text{dbf}_k(t) \) \hspace{1cm} \text{demand bound function, } \text{dbf}, \text{ for node } k \text{ in an interval of length } t
\( \text{dbf}_k(T, t) \) \hspace{1cm} \text{demand bound function, } \text{dbf}, \text{ for node } k \text{ in an interval of length } t, \text{ for a generic } T
\( \text{step}(x) \) \hspace{1cm} \text{step function}
\( \text{jobln}_i^\ell(t_0, t_1) \) \hspace{1cm} \text{binary-valued function that returns 1 if the job } \tau_i^\ell \text{ has both activation and deadline in } [t_0, t_1]

Here and there, we use move left and move right, referred to an absolute activation (or an absolute deadline), as short way to say, respectively, to decrease and increase the value of that absolute activation (or absolute deadline). As extension, we use move left and move right, referred to a job (or an instance of transaction), as short way to say, respectively, to decrease and increase both absolute activation(s) and absolute deadline(s) related to that job (or instance of transaction) of the same value. This notation descends directly from those figures where we try to place jobs in a time-line, with small values on the left side and big values in the right side of the figure.

As example, in figure 1.4 we decrease the value of \( d_3^0 \) from 10 to 9, without change \( \phi_4^0 \).

In figure 1.5 we move right \( \tau_4^0 \) of 1, than the new value of absolute activation and deadlines are

\( \phi_4^0 = 11 \quad d_4^0 = 15. \)

In this particular example, in order to keep valid the rule \( \phi_i^\ell = d_i^{\ell-1} \), we increase also the value of \( \phi_3^0 = 15 \) and \( d_3^0 = 11. \)
1.8. NOTATION

Before

After

Figure 1.4. moving left a deadline.

Before

After

Figure 1.5. moving right a job.

While, figure 1.6 shows an example of moving right an entire instance of transaction. In fact, in this case, we increase by 2 all $\phi^1_i$ and $d^1_i$. Notice that, if we want achieve a sporadic activation pattern, we have to ensure that the reduced difference $\Phi_2 - \Phi_1$ is still greater or equal to $T$ (if necessary, we have to moving right also the $2^{th}$ and following instances, like a domino effect).
Before

After

Figure 1.6. moving right a transaction.
Analysis of sporadic transactions

2.1. Periodic demand bound function

In this Section, we recall the concept of demand bound function for a transaction that is strictly periodic (i.e. \( \forall \ell, \Phi^\ell = \ell T \)). In the next Section we extend the demand bound function to the sporadic case.

The computational requirement of the subset \( T_k \) of tasks allocated on node \( k \) is modeled by its demand bound function (dbf).

**Definition 1.** The demand function on node \( k \), denoted by \( df_k(t_0, t_1) \), is the total computation time of all the instances of the tasks in \( T_k \), having activation time and deadline within \([t_0, t_1]\).

For a periodic transaction, the demand function can be computed as follows [BHR90]:

\[
(2.1) \quad df_k(t_0, t_1) \overset{\text{def}}{=} \sum_{\tau_i \in T_k} \left( \left\lfloor \frac{t_1 - \bar{D}_i}{T} \right\rfloor - \left\lceil \frac{t_0 - \phi_i}{T} \right\rceil + 1 \right) C_i
\]

As suggested by Rahni et al. [RGR08], the overall demand bound function of \( T_k \) in an interval of length \( t \), is defined as:

\[
(2.2) \quad \text{dbf}_k(t) \overset{\text{def}}{=} \max_{t_0} df_k(t_0, t_0 + t)
\]

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

\[
(2.3) \quad \forall k = 1, \ldots, p \quad \forall t > 0 \quad \sum_{T} \text{dbf}_k(T, t) \leq t
\]

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

Since the demand bound function \( \text{dbf}_k(t) \) is right-continuous, piecewise constant, and increasing, it can be represented by the values at each step. For this purpose we introduce the set of scheduling points as follows:

\[
(2.4) \quad S_k \overset{\text{def}}{=} \{(t; w) : \text{dbf}_k(t) = w, \lim_{x \to t^-} \text{dbf}_k(x) < w\}
\]

Notice that the set \( S_k \) contains infinite scheduling points.
However, it has been proved in [RGR08] that after some initial transient points, the function repeats the same sequence of steps every period $T$, in the sense that it exists an instant $t^*$ after which the transient has finished, such that:

$$\forall (t; w) \in S_k, t > t^* + T \quad (t - T; w - \sum_{\tau_i \in T_k} C_i) \in S_k$$

Hence the set $S_k$ can be finitely represented by its finite transient part $S^t_k$ and its finite periodic part $S^p_k$ as follows:

$$S_k = S^t_k \cup \{(t; w) : t = t' + rT, w = w' + r \sum_{\tau_i \in T_k} C_i, (t'; w') \in S^p_k, r \in \mathbb{N}\}$$

Notice that the subset of transient scheduling points $S^t_k$ may eventually be empty. Instead $S^p_k$ is never empty. Moreover, Eq. 2.5 can be rewrite as:

$$\forall t > t^* + T \quad df_k(t) = df_k(t - T) + \sum_{\tau_i \in T_k} C_i.$$ 

In Figure 2.1 we illustrate the definitions introduced in this Section by an example. Consider a transaction whose parameters are: period $T = 5$, end-to-end deadline $D = 8$, task deadlines $D_1 = 2$ and $D_2 = 6$, computation time $C_1 = 1$ and $C_2 = 3$. Both tasks are assigned to a single node. In the lower part of Figure 2.1, we show three consecutive instances of the transaction on three different lines. In the upper part, we show the values of 3 functions: the demand in $[0, t]$; the demand in $[2, 2 + t]$; and the demand bound function. We represent the points where the $dbf$ has a step by a thick dot. Notice that the set of transient scheduling points $S^t_1$ is composed only by point $(2; 1)$. The set of periodic scheduling points is $S^p_1 = \{(6; 4), (8; 5)\}$. The steps are tightly related to task deadlines. For example in the figure, the points $p_1, p_2, p_3$ depend on the deadlines of task $\tau_1$, while the points $p_4, p_5$ depend on the deadlines of $\tau_2$.

Since $D_1 + D_2$ is constant and equal to the end-to-end deadline $D$, increasing the deadline $D_1$ implies a reduction of $D_2$. In general, when we modify a deadline, many scheduling points may vary. Notice that in Figure 2.1 points $p_2$ and $p_3$ are not part of the $dbf$ because they do not contribute to the maximum computation time in the
corresponding interval. However, if we modify some deadlines, it may happen that points \( p_2, p_3 \) move to the left until they start to contribute to the \( \text{dbf} \). For example, by reducing \( D_1 = 1 \) and enlarging \( D_2 = 7 \), point \( p_2 \) becomes \((6, 2)\), while point \( p_4 \) becomes \((7, 4)\).

From the above example, it is clear that the \( \text{dbf} \) represents the most important scheduling properties of a transaction in a compact way.

### 2.2. Relaxing the offset assumption

In the analysis, we assume that each task is activated at the deadline of the preceding one, by setting \( \phi_i \) according to 1.6. This is necessary for our analysis, but it would require a clock synchronization between the nodes.

Here we show how it is possible to relax this assumption and let a task start at the completion time of the preceding one, but with absolute deadline still computed as in the off-line case.

Let us denote the activation of the current transaction instance is \( \Phi^\ell \). Then, every job \( \tau_i^\ell \) is assigned an absolute deadline \( d_i^\ell = \Phi^\ell + D_i \). The activation offset can be anticipated at the completion time of the preceding task in the transaction. This approach slightly simplifies the implementation, although it still requires a global notion of clock in the distributed system. In addition, for all intervals of length \( t \), the \( \text{dbf}_k(t) \) cannot increase by anticipating the activation of the tasks; on the contrary, it may happen that for some job whose activation and deadline were both entirely contained in an interval of length \( t \), at run time the distance between its activation and its deadline increases, so that it may be larger than \( t \). Therefore, it is likely that the actual run-time computational demand of the tasks on a node \( k \) is much lower than the demand computed off-line using the activation offset as in Equation 1.6.

More formally:

**Lemma 1.** Anticipating the activation of a task without modifying its absolute deadline does not increase the \( \text{dbf} \).

**Proof.** Let \( \phi_i^\ell = \Phi^\ell + \phi_i \) be the activation of job \( \tau_i^\ell \), \( d_i^\ell \) its absolute deadline and let \( a_i^\ell < \phi_i^\ell \) be its actual starting time. Then, \( d_i^\ell - a_i^\ell > D_i \).

Let \( t \geq D_i \), and let \( t_0 \) be such that \( \text{df}_k(t_0, t_0 + t) \) is maximal; \( k \) is the index of the node where \( \tau_i \) is mapped. If the interval contains \( \phi_i^\ell \) and \( d_i^\ell \) but not \( a_i^\ell \), then the \( \text{dbf} \) may decrease. In all other cases, the \( \text{dbf} \) in \( t \) remains the same. \( \square \)

Notice that this lemma is valid for both, periodic and sporadic activations of instances of transaction.

While for the purpose of the analysis we impose that the activation of a task is equal to the deadline of the previous task, thanks to Lemma 1 at run-time we can activate a job at the completion of the previous job, as long as the deadlines are correctly set.

### 2.3. Example of sporadic transaction

Unfortunately, for sporadic transactions, the worst case does not occur with periodic activations. Consider the following transaction with 3 tasks on 2 processors. The transaction has period \( T = 5 \) and end-to-end deadline \( D = 12 \). The task parameters are reported in Table 2.2.
2. ANALYSIS OF SPORADIC TRANSACTIONS

<table>
<thead>
<tr>
<th>Task</th>
<th>( C_i )</th>
<th>proc.</th>
<th>( D_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_1 )</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>( \tau_2 )</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>( \tau_3 )</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

**Figure 2.2.** Parameters for the example

In Figure 2.3, we show two possible activation patterns. The first one corresponds to a periodic activation (\( \Phi_1 = T \)): in this case, it is easy to see that the maximum demand on processor 1 in any interval of length 5 is at most 3 units of computation.

In the second activation pattern we delay the activation of the second instance by 2 units of time (\( \Phi_1 = T + 2 \)). As a consequence, the demand in interval \([7, 12]\) becomes 4 units of time, because one extra instance of \( \tau_1 \) enters the interval. Thus, delaying an instance can increase the demand.

This problem occurs when transactions have end-to-end deadline greater than the period. In this case, jobs of different instances can overlap on the same window. Therefore, it is not true anymore (as it is in the uniprocessor case) that a periodic activation pattern leads to the worst-case demand in every interval.

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

### 2.4. Definition of sporadic dbf

A job \( \tau_1^\ell \in T_k \), runs inside interval \([t_0, t_1]\) if its absolute deadline \( d_1^\ell \) is smaller or equal to \( t_1 \)

\[
t_1 \geq d_1^\ell = \Phi_1 + D_i
\]

and its activation is not earlier than \( t_0 \)

\[
t_0 \leq \phi_1^\ell = \Phi_1 + \phi_i
\]

By introducing the function

\[
\text{step}(x) = \begin{cases} 
0 & \text{if } x < 0 \\
1 & \text{if } x \geq 0 
\end{cases}
\]

\[
(2.10)
\]
we can define the following binary-valued function

\[ \text{jobIn}_i^\ell(t_0, t_1) \overset{\text{def}}{=} \text{step}(t_1 - D_i - \Phi^\ell) \cdot \text{step}(\phi_i + \Phi^\ell - t_0) \]

that returns 1 if the job \( \tau_i^\ell \) has both activation and deadline in \([t_0, t_1]\), and it returns 0 otherwise.

Hence, the demand of all the tasks belonging to the transaction \( T_k \) can be expressed as:

\[ df_k(t_0, t_1) \overset{\text{def}}{=} \max_{\tau_i \in T_k} \left\{ \sum_{\ell=-k_0}^{k_1} \sum_{\tau_i} \text{jobIn}_i^\ell(t_0, t_1)C_i \right\} \]

where \( k_0 \) and \( k_1 \) are indexes of transaction instances (later determined in Eq. 2.22) that may have an effect on the demand in \([t_0, t_1]\). We introduced \( k_0 \) and \( k_1 \) because instances of transaction too far in the past and too far in the future cannot overlap with the interval \([t_0, t_1]\), then we reduce the sum to only those instances that may have jobs inside the interval: \(-k_0\) identifies the most negative and \( k_1 \) the most positive.

The sum on all the transaction instances can be split in three parts: the first part is the sum over the indexes corresponding to the past instances (from \(-k_0\) to \(-1\)); the second part is the current instance (with \( \ell = 0 \)), and the third part is the sum over the future instances (from 1 to \( k_1 \)). Hence Equation 2.12 becomes

\[(2.13)\]

\[ df_k(t_0, t_1) = \max_{\Phi \in \Gamma^-} \left\{ \sum_{\ell=-k_0}^{-1} \sum_{\tau_i} \text{jobIn}_i^\ell(t_0, t_1)C_i + \sum_{\tau_i} \text{jobIn}_i^0(t_0, t_1)C_i + \sum_{\ell=1}^{k_1} \sum_{\tau_i} \text{jobIn}_i^\ell(t_0, t_1)C_i \right\} \]

where \( \Gamma^- \) and \( \Gamma^+ \) are the sets of the possible activation patterns of the past and the future instances respectively. Although Eq. 2.13 is apparently more complex than Eq. 2.12, it will be more useful for our purposes because it has the advantage of decoupling the dependency on past and future instances (see Sections 2.6 and 2.7). In fact, one of future instances cannot influence the offset of past instances and vice versa, then we can simplify the original problem and study these two groups of instances separately.

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

\[ dbf_k(t) \overset{\text{def}}{=} \max_{t_0} df_k(t_0, t_0 + t) \]

Figure 2.4 shows that, for the same parameters of figure 2.2, the sporadic dbf computed from Eq. 2.14 is larger than the periodic dbf (Eq. 2.2).
2. ANALYSIS OF SPORADIC TRANSACTIONS

Equation 2.14 is a nice and compact expression of the dbf. It is however unclear how such a dbf should be practically computed: how many instances $ℓ$ of the transaction should we consider in the sum of Eq. 2.13? How many values of $t_0$ should we consider in the maximum of Eq. 2.14?

We follow a strategy similar to the one used for computing the dbf of periodic transaction. The strategy is described in the algorithm reported in Figure 2.5. First (at line 2), we compute the list intervalSet of all the significant intervals $[t_0, t_1]$, i.e. the intervals such that $∀ε > 0$ both the demands $df_k(t_0 + ε, t_1)$ and $df_k(t_0, t_1 - ε)$ are strictly less than $df_k(t_0, t_1)$. In Section 2.6 we describe the procedure STOREINTERVALS for performing this step. After sorting the intervals $[t_0, t_1]$ in intervalSet by increasing $t_1 - t_0$ (at line 3), we search for the activation pattern $Φ$ of past and future instances, i.e. the

---

**Figure 2.4.** An example of sporadic dbf.

1: $\text{intervalSet} ← ∅$ \> initialize the set of intervals
2: $\text{storeIntervals}$ \> store intervals in intervalSet
3: sort intervalSet by increasing $t_1 - t_0$
4: lastDBFval ← 0
5: for each $[t_0, t_1] ∈ \text{intervalSet}$ do \> loop on all intervals
6: $Γ^−, Γ^+ ← ∅$ \> init sets of past and future patterns
7: $\text{SPANPATTERNS}(t_0, t_1)$ \> store all patterns
8: curDBFval ← $df_k(t_1, t_0)$ \> Eq. 2.13
9: if curDBFval $>$ lastDBFval then \> Eq. 2.14
10: store the point $(t_1 - t_0, \text{curDBFval})$
11: else
12: do nothing (dominated by previous point)
13: end if
14: end for

**Figure 2.5.** Algorithm for computing the dbf.
instances before and after the one under analysis, that maximizes the demand in \([t_0, t_1]\). In Section 2.7 we describe the procedure \texttt{SPANPATTERNS} that computes the set \(\Gamma\) of all possible activation patterns.

### 2.5. Periodicity

Here we show that also the sporadic \texttt{dbf} has a periodic pattern of steps after a transient that is long at most \(D + T\).

**Lemma 2.** For large values of \(t\), the \(\texttt{dbf}_k(t)\) has a periodic pattern. More formally:

\[
\forall t > D + T \quad \texttt{dbf}_k(t + T) = \texttt{dbf}_k(t) + \sum_{\tau_i \in T_k} C_i.
\]

**Proof.** Let \(t_0\) and \(\Phi\) be the instant and activation pattern that give the value of \(\texttt{dbf}_k(t)\) in Equations 2.14 and 2.13 respectively, and let us set \(t_1 = t_0 + t\).

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

\[
(2.15) \quad \forall h \geq \ell \quad \Phi^h - \Phi^{h-1} = T
\]

otherwise, we could anticipate all \(\Phi^h\) with \(h \geq \ell\) without removing any job from the interval. On the contrary, the deadline of a job may enter the interval, and the worst-case activation pattern cannot be \(\Phi\) anymore, causing a contradiction.

From (2.15) and the definition of \(\ell\), we notice that the instance \(\ell\) of the transaction ends earlier than \(t_1\). Clearly this is also true for all instances before \(\ell\). Formally

\[
\Phi^{\ell-1} \leq t_0 \implies \Phi^{\ell} \leq t_0 + T
\]

\[
\Phi^{\ell} + D \leq t_0 + T + D < t_1.
\]

From 2.15, it follows that any interval of length \(T\) starting after \(\Phi^{\ell} + D\) contains exactly one activation and one deadline of each task. Hence the demand generated in the interval \([t_0, t_1 + T]\) increases by one job for all tasks in \(T_k\), i.e. \(C_k\).

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

Since, thanks to the lemma, the transient part of the \texttt{dbf} lasts for no longer than \(D + T\) and the periodic part is long \(T\), it is possible to compute the \texttt{dbf} only for lengths of intervals less than \(D + 2T\).

### 2.6. Enumerating the intervals

The first stage requires to enumerate all the intervals \([t_0, t_1]\). The pseudo-code of this stage is reported in Figure 2.6. First, we claim that \(t_0\) must coincide with the activation of some job. In fact, if this does not happen then we could increase \(t_0\) achieving a shorter interval with the same demand. Hence we set \(t_0\) equal to the activation of the job \(\tau^0_i\), i.e. \(t_0\) spans on \(\{\phi_i : \tau_i \in T_k\}\) (see line 4 of the algorithm). Notice that, without loss of generality, we label by 0 the transaction instance which this job belongs to.
Regarding the possible values of \( t_1 \), it is easy to see that it is sufficient to test only the absolute deadlines \( d_{i,j} \). In fact only if \( t_1 = d_{i,j} \) for some task \( \tau_j \in T_k \) and some transaction instance \( h \), then a reduction of \( t_1 \) by an arbitrary small amount \( \varepsilon \) will decrease the demand \( df \) by at least \( C_j \). However, the main difficulty here is that the absolute activations are not fixed, hence we do not know where the deadlines are until we fix the sporadic activation pattern \( \Phi \).

First, we list the values of \( t_1 \) associated with the absolute deadlines of the instance 0 (see lines 5–9 of figure 2.6). Then we invoke the recursive procedures \textsc{futureDeadline} and \textsc{pastDeadline} that list the absolute deadlines of the future and past instances, respectively.

These two procedures explore the possible activation patterns \( \Phi \) such that task activations are aligned with \( t_0 \). For each pattern the values of absolute deadlines are recorded as candidate values for \( t_1 \).

Now, we give more details on the steps. We start by setting \( \Phi^\ell = \ell T, \forall \ell \). At this point, we list all intervals \([t_0, d_{i,j}]\), that are intervals starting with the job under analysis \( \tau_i^0 \) (the one with \( \phi_i = t_0 \)) and ending at the deadline of some other job (of the same, or of other transaction instances). We only check intervals with size \( d_{i,j} - t_0 \leq D + 2T \), because the \textsc{dbf} repeats the same pattern for larger intervals (see Lemma 2). We insert in the list \textsc{intervalSet} all the intervals \([t_0, d_{i,j}]\). After this step the intervals inserted in \textsc{intervalSet} correspond to the periodic activation scenario only.

Since the worst case for a sporadic transaction may not occur with periodic activations, as we showed in the example of Figure 2.3, we investigate whether a variation in the activation pattern \( \Phi \) can increase the demand or not. Hence, we vary some absolute activations \( \Phi^\ell \) (for \( \ell \neq 0 \)) and we study if intervals \([t_0, d_{i,j}]\) with the same demand becomes shorter for some activation pattern \( \Phi \).

**Future instances.**

We first investigate the effect of a delaying future instances. The idea is to delay future instances of transaction in order to make intervals \([t_0, t_1]\) big enough to put a new job in, but such that if we reduce \( t_1 \) or increase \( t_0 \), the value of \textsc{dbf} decrease. A solution is to explore values of \( \Phi^\ell \) to enforce each \( \phi_i = t_0 \) and then store all \( t_1 = d_{i,j} \), for all \( j \).

Clearly we have to keep valid the minimum inter-arrival time and save only deadlines bigger than \( t_0 \), i.e.:

\[
\Phi^\ell \geq \Phi^{\ell-1} + T \quad d_{i,j} > t_0
\]

In order to study also the periodic pattern of activation and to allow more freedom to following instances, we enforce the case \( \Phi^\ell = \Phi^{\ell-1} \), even if \( \tau_1 \) is not mapped to the current cpu.

For all found values of \( \Phi^\ell \), we have to investigate following instances, in a recursive way. In fact each value may have a different effect, because reduces the interval of values that following offsets may assume. We start from \( \ell = 1 \) until \( \ell < \left\lceil \frac{t_0 + D + 2T}{T} \right\rceil \), because after that we are sure that all possible intervals are bigger than \( D + 2T \).

Then, the set of values for a generic \( \Phi^\ell \) is

\[
\Phi^\ell \in \{ \Phi^{\ell-1} + T \} \cup \{ t_0 - \phi_i : t_0 - \phi_i > \Phi^{\ell-1} + T, \tau_i \in T_k \}
\]
Figure 2.6. Algorithm for enumerating intervals.
With reference to figure 2.6, the loop at row 16 studies each value of set 2.16, while the inner loop check if deadlines related to current instance are good candidate as $t_1$ and store possible intervals $[t_0, d^h_i]$ in the repository intervalSet. The iteration on following instances is done by row 24.

**Past instances.**

For past instances we follow a slightly different strategy. The goal is to reduce the size of $[t_0, d^h_i]$ and maximize the number of jobs ($h < 0$ is a generic instance in the past and $\tau_i \in T_k$). A way to reduce the size of the interval $[t_0, d^h_i]$ it to move activations to the past, until the absolute activation of one job coincide with $t_0$. Then “freeze” absolute activations $\Phi$ and store all deadline $d^h_i > t_0$.

**Figure 2.7.** Interval for offset of past instances of transaction.

We start observing that instances further in the past, limits the range of values for activations of instances that arrive after. Formally:

$$\forall \ell < 0, \quad \Phi^\ell = \gamma + \ell T, \quad \gamma \leq 0$$

$$\Rightarrow \gamma + hT \leq \Phi^h \leq hT, \quad \ell < h < 0.$$  

This descends directly from the property of minimum distance between two consecutive instances of transaction. In figure 2.7 we show an example, where $\ell = -4$ and $h = -3$.

Let call $\ell$ the first instance in the past that has the end-to-end deadline after $t_0$.

$$\Phi^{\ell - 1} + D \leq t_0$$

$$\Phi^\ell + D > t_0.$$  

Clearly $\Phi^\ell$ can move freely to the past. Then interesting values of $\Phi^\ell$ are:

$$\Phi^\ell \in \{\ell T\} \cup \{t_0 - \phi_i : t_0 - \phi_i < \ell T, \tau_i \in T_k\}.$$  

For others instance $h > \ell$, the rule is

$$\Phi^\ell \in \{\ell T, \Phi^{\ell - 1} + T\} \cup \{t_0 - \phi_i : \Phi^{\ell - 1} + T < t_0 - \phi_i < \ell T, \tau_i \in T_k\}.$$  

Because we are slowly moving all instances to the past, initially the first instance to solve Equations 2.17 is $\ell = -((D-t_0)T - 1) = -k_0$. But, after recursively test all value
in Equations 2.19–2.20, we have to set the first instance in the past as \( \ell = -k_0 + 1 \), and repeats the same steps, until we give the possibility to all instance in the past to be free to chose their absolute activation and impose restrictions to other instances.

Instead compute all \( \Phi \) and then compute all deadlines, it is possible to do that, step by step, every time we choose a \( \Phi^\ell \):

\[
(2.21) \quad t_1 \in \{ \Phi^\ell + \overline{D}_i : \Phi^\ell + \overline{D} > t_0, \tau_i \in T_k \}.
\]

Notice that Equation 2.19 is equivalent to Equation 2.20 with an \( \Phi^{\ell-1} = -\infty \), i.e. an offset big enough to move all past instances in the past so that them end before \( t_0 \), like \(-2k_0(D + T)\).

Now we present an algorithm for computing the activation patterns that determine the maximum demand in a given interval \([t_0, t_1]\).

### 2.7. Enumerating the activation patterns

In this Section we explain the procedure \( \text{spanPatterns}(t_0, t_1) \) (see line 7 of the algorithm in Figure 2.5) that checks all possible sporadic activation patterns of past and future instances that may have an impact on the interval \([t_0, t_1]\). Therefore, we are interested only in transaction instances that may overlap with the interval \([t_0, t_1]\). The indexes of these transactions are from \(-k_0\) to \(k_1\), where

\[
(2.22) \quad k_0 = \left\lfloor \frac{D - t_0}{T} \right\rfloor - 1 \quad k_1 = \left\lceil \frac{t_1}{T} \right\rceil - 1.
\]

Hence the sum of transactions instances of Eq. 2.13 has to be made for \( \ell = -k_0, \ldots, k_1 \):

\[
(2.23) \quad \ell \in [-\left( \left\lfloor \frac{D - t_0}{T} \right\rfloor - 1 \right), \left\lceil \frac{t_1}{T} \right\rceil - 1].
\]

For the example of figure 2.2 (see also Figure 2.4 for a time line representation of the instances), if we set \( t_0 = \phi_1 = 0 \) and \( t_1 = 13 \), we find \( k_0 = 2 \) and \( k_1 = 2 \), meaning that in the analysis of the demand in the interval \([5, 12]\) we consider the instances \(-2(= -k_0), -1, 0, 1, 2(= k_1)\) of the transaction.

In the exploration of the activation patterns we distinguish between future instances (with index \( \ell > 0 \)) and past instances (with index \( \ell < 0 \)). The guideline for the exploration of absolute activations of future instances is to increase slowly the value of \( \Phi^\ell \) starting from \( \Phi^{\ell-1} \) and examine those values that may increase the demand. It is easy to see that it happens when we reach a value of \( \Phi^\ell \) such that \( \phi^\ell_i = t_0 \). This is possible by setting

\[
(2.24) \quad \Phi^\ell = t_0 - \phi_i.
\]

But the demand may also decrease if a deadline becomes bigger than \( t_1 \). However, Eq. 2.23 is a valid absolute activation only if it respects the constraints of being a sporadic transaction with minimum inter-arrival \( T \); that is

\[
(2.24) \quad \Phi^\ell \geq \Phi^{\ell-1} + T.
\]

This condition introduces a recurrent dependency between all the values \( \Phi^0, \Phi^1, \Phi^2, \ldots, \Phi^{k_1} \).

The same rationale is applied to past instances (the ones with index \( \ell < 0 \)). In this case however, we start from the periodic activation pattern and slowly increase the distance among \( \Phi^\ell \), moving left all past instances. Then the event that increase the
demand is when an absolute deadline is decreased enough to became equal to $t_1$. Clearly, the demand may decrease if an absolute offset became smaller than $t_0$.

The full algorithm that explores the activation patterns is reported in Figure 2.8, with both procedures \textsc{computeFuture} and \textsc{computePast} to compute $\Gamma^+$ and $\Gamma^-$.

What values do we try for $\Phi = \left\{ \Phi_{-k_0}, \ldots, \Phi_{k_1} \right\}$? The following lemma provides an initial upper bound to the number of values.

**Lemma 3.** The number of vectors $\Phi \in \Gamma$ that we need to consider is upper bounded by $2^N$, with $N = (k_0 + k_1 + 1) \cdot n_k$.

**Proof.** If we fix $t_0$ and $t_1$, (2.11) can only have two values: 0 or 1. Therefore, all possible sporadic activation patterns can be divided in two subsets for each task: activation for which (2.11) has value equal to 0, and activations for which (2.11) has value equal to 1. For each task, the number of instances to be considered is $k_0 + k_1 + 1$. Hence the number of possible partitions is $2^{(k_0 + k_1 + 1) \cdot n_k}$, as required. \hfill $\Box$

In Lemma 3, we established an upper bound to all the activation patterns to be tested. However, the number of such partitions can be much lower. In fact, jobs are not independent of each other. For example, if a job activation is earlier than $t_0$, because of a specific value of $\Phi$, its preceding jobs are also outside the interval of interest. Similarly, if a job has deadline greater than $t_1$, all successive jobs are also outside of the interval.

Notice also that the values of the absolute activations are influenced by each other. For example, if $\Phi^1$ is large, the instance $\ell = 1$ arrives late and the following instances may be completely outside the interval of interest. Similarly, if $\Phi^{-1} \ll -T$, then the instance $\ell = -1$ arrived early, and previous instances $-2, -3, \ldots$ may not overlap with the interval of interest. In other words, some of the Equations 2.11 are linked with each other in complex ways.

Finally, observe that all $\Phi$ inside a partition are equivalent for our purposes, since the values of Equations 2.11 do not change. Therefore, we can choose one representative vector for each partition.

**Future instances.**

Let us start by analyzing future instances. The first absolute activation $\Phi^1$ can assume values in the following set:

$$\Phi^1 \in \{T\} \cup \left\{ t_0 - \phi_i : t_0 - \phi_i > T, \tau_i \in \mathcal{T}_k \right\}.$$  

In fact, we want to investigate all $\Phi^1$ such that some job $\tau^1_1$ has activation coincident with $t_0$. In this way, it may happen that a job $\tau^\ell_1$, whose activation is outside the interval $[t_0, t_1]$, is moved inside the interval, thus increasing the $\text{dbf}$. We do not know which of these jobs maximizes the $\text{dbf}$, thus we need to check all of them.

Notice that, if $\Phi^1$ is set to a value between any two consecutive elements of the previous set, the number of activations inside interval $[t_0, t_1]$ will not increase, because each element of the set of Eq. 2.25 identifies the point where a new job activation comes inside the interval; on the contrary, increasing $\Phi^1$ may lead a deadline to be greater than $t_1$. Therefore, we choose the minimum possible value that increases the number of activations.
After fixing the absolute activation \( \Phi^1 \), we proceed with all possible values for \( \Phi^2 \) in a similar way. The general equation for \( \Phi^\ell \) is:

\[
\Phi^\ell \in \{ \Phi^{\ell-1} + T \} \cup \{ t_0 - \phi_i : t_0 - \phi_i > \Phi^{\ell-1} + T, \tau_i \in \mathcal{T}_k \}
\]

**Past instances.**

A similar reasoning applies to past instances. We start by computing the possible values for \( \Phi^{-1} \) as follows:

\[
\Phi^{-1} \in \{-T\} \cup \{ t_1 - \overline{D}_i : t_1 - \overline{D}_i < -T, \tau_i \in \mathcal{T}_k \}.
\]

In fact, we want to move all the deadlines of instance \( \ell = -1 \) that are after \( t_1 \) to be coincident with \( t_1 \), in turn.

Again, if we set \( \Phi^{-1} \) to a value between any two consecutive elements of the previous set, the number of deadlines in \([t_0, t_1]\) will not increase. Therefore, we choose the minimum possible value that increases the number of deadlines.

The general Equation for \( \Phi^\ell \), when \( \ell < 0 \), is:

\[
\Phi^\ell \in \{ \Phi^{\ell+1} - T \} \cup \{ t_1 - \overline{D}_i : t_1 - \overline{D}_i < \Phi^{\ell+1} - T, \tau_i \in \mathcal{T}_k \}.
\]

In the example of Figure 2.4, if we assume \( t_0 = 0 \) then \( \Phi^1 \) should be tested with the values of 5 (\( = T \)). Instead, if \( t_0 = \phi_3 = 7 \) then \( \Phi^1 \) is checked both when it is 5 and when it is \( t_0 - \phi_1 = 7 \), meaning that we align the activation of the instance 1 with the offset \( \phi_3 = t_0 = 7 \).

### 2.8. Complexity analysis

We start by analyzing the complexity of procedure STOREINTERVALS. The outer loop (line 3 in figure 2.6) is executed \( n_k \) times. After adding the intervals for instance 0, procedures FUTUREDEADLINE and PASTDEADLINE are invoked.

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

Procedure PASTDEADLINE is very similar. The number of instances is \( k_0 \) (see Eq. 2.22). The maximum number of elements generated for each combination of past activations is \( n_k k_0 \). Finally, the maximum number of combinations (line 29) is \( (n_k + 2)^{k_0} \).

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

\[
s = k_2 n_k \left\lfloor \frac{s}{k} \right\rfloor + n_k k_0 (n_k + 2)^{k_0}
\]

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

Notice that, while generating the the values of \( t_1 \), it is quite common to obtain many times always the same values. In average, we expect that the final size of the list is much smaller than its upper bound \( s \).
1: procedure spanPatterns\((t_0, t_1)\)
2: \(k_1 = \left\lceil \frac{t_1}{T} \right\rceil - 1\) \(\triangleright\) see Eq. 2.22
3: computeFuture\((1, (0, \ldots, 0))\)

4: \(k_0 = \left\lceil \frac{D-t_0}{T} \right\rceil - 1\) \(\triangleright\) see Eq. 2.22
5: computePast\((-1, (0, \ldots, 0))\)

6: end procedure

7: procedure computeFuture\((\ell, (\Phi^1, \ldots, \Phi^{k_1}))\)
8: if \(\ell > k_1\) then \(\triangleright\) the exit condition
9: store \((\Phi^1, \ldots, \Phi^{k_1})\) in \(\Gamma^+\) \(\triangleright\) \(\Gamma^+\) is global
10: else
11: \(\Phi^0 \leftarrow 0\)
12: for all \(\Phi^\ell \in \{\Phi^{\ell-1} + T\} \cup \{t_0 - \phi_i : t_0 - \phi_i > \Phi^{\ell-1} + T, \tau_i \in (T_k)\}\) do
13: computeFuture\((\ell + 1, (\Phi^1, \ldots, \Phi^{k_1})\))
14: end for
15: end if
16: end procedure

17: procedure computePast\((\ell, (\Phi^{-k_0}, \ldots, \Phi^{-1}))\)
18: if \(\ell < -k_0\) then
19: store \((\Phi^{-k_0}, \ldots, \Phi^{-1})\) in \(\Gamma^-\)
20: else
21: \(\Phi^0 \leftarrow 0\)
22: for all \(\Phi^\ell \in \{\Phi^{\ell+1} - T\} \cup \{t_1 - \overline{D}_i : t_1 - \overline{D}_i < \Phi^{\ell+1} - T, \tau_i \in (T_k)\}\) do
23: computePast\((\ell - 1, (\Phi^{-k_0}, \ldots, \Phi^{-1}))\)
24: end for
25: end if
26: end procedure

Figure 2.8. Algorithm for generating \(\Gamma^-\) and \(\Gamma^+\).

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

\[O((n_k + 1)^{k_0} + (n_k + 1)^{k_1}).\]
Finally, the complexity of computing the \( \text{dbf} \) is:

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

We are aware that the proposed algorithm is very complex. Most of the complexity lies in the sporadicity of the transaction that requires to check all possible scenarios. In this thesis, we focused on the exact analysis regardless of its complexity. We leave to future investigations the development of simplified algorithms as well as Fully Polynomial Time Approximation Schemes (FPTAS).

2.9. Simulations

In order to test the practical complexity of our algorithm, we wrote an implementation and conducted several runs on synthetic transaction sets with different number of tasks, cpu and deadline/period ratio, measuring the time needed to compute the \( \text{dbf} \). The result is that, in spite of the exponential nature of the problem, the algorithm is usable for systems with a reasonable size.

The most important parameters are the ratio between end to end deadline \( D \) and the period \( T \), and the number of tasks on each cpu. For ratio greater than 25 the time required to compute the \( \text{dbf} \) begins to be quite long, while, for values of \( \frac{D}{T} \) of 10 or less, it is possible to put 20 tasks on each cpu and still receive an answer in a reasonable time.

The environment used to perform these tests is based on a Linux distribution (Ubuntu) with kernel 2.6.24-16-generic. The HW is a notebook with an Intel T2400 (a Centrino duo at 1.8GHz) and 2GByte of ram.

To get the time for each test, we use the Linux call

\[
\text{int clock_gettime(clockid_t clock_id, struct timespec *tp)}
\]

with value \( \text{CLOCK_THREAD_CPUTIME_ID} \), in order to get the CPU-time given to our thread. The program is implemented in a single thread, it does not contain optimizations and respect the algorithm proposed in Figure 2.5, 2.6 and 2.8.

In Figure 2.9 and Figure 2.10 we report, in seconds, the time required by our implementation of the algorithm to compute the \( \text{dbf} \) for transactions with 20–120 tasks uniformly distributed among 4 and 8 CPUs, i.e. if we have 40 tasks and 4 CPUs then there are 10 tasks mapped on each CPU. We vary the value of \( \frac{D}{T} \) from 5 to 20. Computation time \( C_i \), relative deadline \( D_i \) and mapping \( x_i \) are randomly generated. As we can see, the algorithm resolves small systems in few seconds, while medium systems require some minutes. Graphical representations in Figures 2.9 and 2.10 show the slope almost exponential of each curve (y-axis is in logarithmic scale).

If we compare results of the case with 4 CPU against results related to tests with 8 CPUs, we notice that when we increase the number of CPU, the number of task on each CPU decrease and then the algorithm has to test less offsets, decreasing the time required.

We tried also to run the test with a large system consisting of 40 tasks, 4 CPUs and \( \frac{D}{T} = 25 \). It took around one day to complete the computation of the \( \text{dbf} \). Notice that a deadline period ratio of 25 means that we can have 24 instances of transaction running before the first ends.
2. ANALYSIS OF SPORADIC TRANSACTIONS

<table>
<thead>
<tr>
<th>n</th>
<th>D</th>
<th>min</th>
<th>average</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>5</td>
<td>0.0038</td>
<td>0.008687</td>
<td>0.01968</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>0.1497</td>
<td>0.2002</td>
<td>0.2950</td>
</tr>
<tr>
<td>60</td>
<td>5</td>
<td>0.6551</td>
<td>1.0892</td>
<td>1.4081</td>
</tr>
<tr>
<td>80</td>
<td>5</td>
<td>2.8609</td>
<td>4.4397</td>
<td>5.5224</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>9.8498</td>
<td>16.5764</td>
<td>22.8148</td>
</tr>
<tr>
<td>120</td>
<td>5</td>
<td>99.8150</td>
<td>131.7730</td>
<td>172.7335</td>
</tr>
</tbody>
</table>

| 20  | 10| 0.0148 | 0.0390 | 0.07311 |
| 40  | 10| 3.1423 | 6.7326 | 12.0010 |
| 60  | 10| 47.0424 | 106.5302 | 138.9140 |
| 80  | 10| 415.6527 | 672.8660 | 1100.9191 |

| 20  | 15| 0.09068 | 0.18023 | 0.3284 |
| 40  | 15| 7.8494 | 20.0923 | 36.6787 |
| 60  | 15| 264.5299 | 399.1137 | 597.4694 |

Figure 2.9. Simulations with 4 processors (times in seconds)

2.10. Optimizing the algorithm

Despite the fact that the algorithm gives a response in a reasonable time for small and medium transactions, the execution time could be reduced by applying some code optimization.
<table>
<thead>
<tr>
<th>$n$</th>
<th>$\frac{D_T}{T}$</th>
<th>$\text{min}$</th>
<th>$\text{average}$</th>
<th>$\text{max}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>5</td>
<td>0.00035590</td>
<td>0.003888</td>
<td>0.01236</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>0.01150</td>
<td>0.02149</td>
<td>0.04102</td>
</tr>
<tr>
<td>60</td>
<td>5</td>
<td>0.06871</td>
<td>0.1034</td>
<td>0.1460</td>
</tr>
<tr>
<td>80</td>
<td>5</td>
<td>0.2765</td>
<td>0.4115</td>
<td>0.6262</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>0.8970</td>
<td>1.2746</td>
<td>1.7832</td>
</tr>
<tr>
<td>120</td>
<td>5</td>
<td>2.2320</td>
<td>3.2095</td>
<td>4.0526</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>0.002730</td>
<td>0.007563</td>
<td>0.01531</td>
</tr>
<tr>
<td>40</td>
<td>10</td>
<td>0.05001</td>
<td>0.1179</td>
<td>0.1826</td>
</tr>
<tr>
<td>60</td>
<td>10</td>
<td>0.4926</td>
<td>1.2065</td>
<td>2.1588</td>
</tr>
<tr>
<td>80</td>
<td>10</td>
<td>3.4278</td>
<td>6.2470</td>
<td>8.9823</td>
</tr>
<tr>
<td>20</td>
<td>15</td>
<td>0.007386</td>
<td>0.01656</td>
<td>0.03480</td>
</tr>
<tr>
<td>40</td>
<td>15</td>
<td>0.2779</td>
<td>0.3893</td>
<td>0.6214</td>
</tr>
<tr>
<td>60</td>
<td>15</td>
<td>2.7329</td>
<td>7.6004</td>
<td>12.666</td>
</tr>
<tr>
<td>80</td>
<td>15</td>
<td>24.3439</td>
<td>45.6743</td>
<td>68.706</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>0.01183</td>
<td>0.03168</td>
<td>0.05448</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>0.5934</td>
<td>1.3678</td>
<td>2.7353</td>
</tr>
<tr>
<td>60</td>
<td>20</td>
<td>16.9809</td>
<td>27.7639</td>
<td>41.7258</td>
</tr>
<tr>
<td>80</td>
<td>20</td>
<td>127.4887</td>
<td>233.0104</td>
<td>330.9758</td>
</tr>
</tbody>
</table>

Figure 2.10. Simulations with 8 processors (times in seconds)
**Parallelism**

A simple technique is based on exploiting the inherent parallelism of many steps of the algorithm. The effective number of parallel threads can be tuned depending to the grade of parallelism of the system that run the test, like a multi processor or a cluster. Two good places where introduce the parallelism are:

- row 2 in figure 2.6: we can use more than one thread to compute independently the set of intervals that derives by each \( t_0 = \phi_i \).
- row 4 in figure 2.5: the demand of each single interval is independent from others intervals.

To reduce wait-states, it is possible to start computing the demand of intervals before procedure `storeIntervals` ends. Clearly there are some shared structure, like the list of intervals and the `dbf`.

**Caching**

Another possibility is to use caching. For example, during the computation of demand of future instances for an interval \([t_0, t_1]\), it may happen that in one point of the tree there is \( \Phi^\ell = t_0 - \phi_i \) and in another point we find \( \Phi^h = t_0 - \phi_i \). Clearly the maximum contribution of these two subtrees is exactly the same. Moreover, this is true even if we start from a different \( t_0 \), when the interval has the same length and is still true that \( \phi_i \) coincide with the start of the interval. Hence, we can think to cache the demand of each value \( \phi_i \) for each length \( t_1 - t_0 \). Obviously, some intervals have more chances to be reused and they can be found checking the length all intervals given by procedure `storeIntervals`. Also, it is also possible to tune this method in order to reduce the memory requirement, for example caching only the demand of most used intervals or removing caches when we are sure that they are not more usefull (also in this case we can use the set of intervals computed by procedure `storeIntervals`).

**Branch and bound**

The last technique presented here is based on branch and bound. In fact, while we are computing the demand of an interval, we explore a tree of possible sporadic activations patterns, then it can be useful to have a way to overestimate the contribution of a subtree. At every point of the tree, we are able to know the range of values of the absolute activation \( \Phi^\ell, t_0 \) and \( t_1 \). Figure 2.11 shows an example related to the analysis of future instances.

When analysing future instances, \( \Phi^\ell \geq \Phi^\ell-1 + T \), and we are able to know the maximum number of instances of transaction that may overlap with the interval: this happens when the activation pattern is periodic. Now we need a way to estimate the contribution of each instance. We propose a demand bound function that covers only jobs of the first instance, with task index from 1 to \( i \):

\[
(2.28) \quad \text{dbf}^i_k(t) \overset{\text{def}}{=} \max_{t_0 \leq \phi_i} \sum_{\tau_j \in T_k, j \leq i} \text{jobln}^0_j(t_0, t_0 + t)C_j
\]

Clearly, \( \text{dbf}^i_k(t) \geq \text{dbf}^j_k(t) \) if and only if \( i \geq j \). We know that moving \( \Phi^\ell \) to the rigth, we may have a contribution from instance \( \ell \) that is given by an interval of length...
2.10. OPTIMIZING THE ALGORITHM

\[ \min(t_1 - t_0, t_1 - \Phi^{\ell-1} + T) \] placed somewhere between \( \Phi^{\ell} \) and \( t_1 \). Then, using \( \text{dbf}_k(t) \) we can easily find this max. To understand which index \( i \) must be used, we can note that all jobs with absolute deadline smaller than \( t_1 \) may contribute to the demand:

\[ d_i = \max_{\tau_j \in T_k} \{d_j : \Phi^{\ell} + D_j < t_1\} \]

The algorithm reported in Figure 2.12 contains a small optimization. In fact, every time we increase the index of transaction \( \ell \) we reduce the difference \( t_1 - \Phi^{\ell} \), then if the last deadline before \( t_1 \) is related to \( \tau^{\ell}_i \), clearly in the following instance of transaction \( \ell + 1 \) the last deadline will be \( d_{j+1}^{\ell+1} \) with \( j \leq i \).

A similar reasoning can be done for past instances.

**Optimized simulations**

We redo some simulations using a modified version of the algorithm that compute the demand of each interval using branch and bound during the exploration of sporadic activation pattern. In this way, we achieve a reduction of time required to compute the \( \text{dbf} \) to 50% of original time. For large system the required time is much smaller, around 10% of initial time, and the difference became larger increasing the size of systems. Only in few cases, when the systems are really small, the required time is almost unchanged. This because the pruned sub-tree are small and the advantages are comparable with the time required to test if a sub-tree is interesting or not. We report the numerical results in Table 2.13 and plot them in Figure 2.14.

---

**Figure 2.11.** Example of relation among \( \Phi^{\ell} \), \( t_0 \) and \( t_1 \), for future instances.
1: procedure OVERESTIMATE\((t_0, t_1, \Phi^{\ell-1})\)
2:  value = 0
3:  \(\Phi^{\ell} = \Phi^{\ell-1} + T\)
4:  \(i = n;\)
5:  while \(\Phi^{\ell} < t_1\) do
6:      length = min\((t_1 - t_0, t_1 - \Phi^{\ell});\)
7:      while \(\Phi^{\ell} + D_i > t_1\) or \(x_i \neq k\) do
8:          \(i− = 1;\)
9:          if \(i < 1\) then
10:             return value;
11:      end if
12:  end while
13:  value += \(\text{dbf}_k(t_1 - \Phi^{\ell});\)
14:  \(\Phi^{\ell+} = T\)
15: end while
16: return value;
17: end procedure

Figure 2.12. Algorithm for overestimate a subtree.

<table>
<thead>
<tr>
<th>CPU</th>
<th>(n)</th>
<th>(D/T)</th>
<th>(\text{min})</th>
<th>\text{average}</th>
<th>\text{max}</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
<td>5</td>
<td>0.0148</td>
<td>0.0177</td>
<td>0.0227</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>5</td>
<td>0.1237</td>
<td>0.1546</td>
<td>0.2048</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>5</td>
<td>0.5075</td>
<td>0.6577</td>
<td>0.7730</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>5</td>
<td>5.1757</td>
<td>8.0756</td>
<td>11.5889</td>
</tr>
</tbody>
</table>

|        |        |        |   0.0323   | 0.0397 | 0.0514 |
| 4   | 20   | 10     | 0.0323     | 0.0397 | 0.0514 |
| 4   | 40   | 10     | 0.3420     | 0.5943 | 0.8360 |
| 4   | 60   | 10     | 3.7987     | 5.5338 | 7.6891 |
| 4   | 100  | 10     | 105.1929   | 142.2384 | 192.2207 |

|        |        |        |   0.0431   | 0.0654 | 0.0895 |
| 4   | 20   | 15     | 0.0431     | 0.0654 | 0.0895 |
| 4   | 40   | 15     | 1.5671     | 2.3917 | 4.0547 |
| 4   | 60   | 15     | 22.4993    | 28.9635 | 36.5755 |
| 4   | 80   | 15     | 109.9120   | 158.1644 | 203.6692 |

Figure 2.13. Execution time for computing the \(\text{dbf}\) on 4 processors (times in seconds) with a small optimization.
2.10. OPTIMIZING THE ALGORITHM

Figure 2.14. Fast execution times on 4 CPUs.
CHAPTER 3

Implicit Deadline Synchronization Protocol

In order to implement a system that uses the transaction model presented in this thesis, we must modify the part of OS related to set the absolute activation time and absolute deadline for each job, under the assumption that the system is schedulable (see eq. 2.3, eq. 2.14 and Eq. 2.13), all tasks execute for less or equal than their worst-case execution time $C_i$ and that two consecutive instances of the same transaction are spaced at least by $T$, so that the scheduler EDF is able to ensure that:

1. in every interval, the sporadic demand of every transaction is always less than or equal to the sporadic demand computed off-line;
2. every transaction always respects its end-to-end deadline.

At first glance, it may seem that we need a strict clock synchronization protocol to guarantee that the activations and the deadlines are correctly computed. In fact, in a distributed system the timing information are obtained on each node by reading local timer hardware interfaces, and different timers can have different offsets and different speeds. Therefore, a global clock synchronization protocol is often used to synchronize the different timing views to the one taken as reference.

In this thesis, instead, we show that it is possible to remove the need for a common time reference. However, we still assume that there is no drift among the timers of different nodes, i.e. all timers use the time unit to measure exactly the same amount of time. In this way the measured length of any interval of time is the same for all timers, during the whole execution of our system.

Our idea is similar to the Release Guard Protocol (RGP) [SL96] by Sun and Liu. RGP has been thought for reducing the start-time jitter and guaranteeing a minimum separation time between two task activations in a fixed priority system. RGP works by delaying the activations of a task so that the distance between two consecutive instances is never less than the minimum inter-arrival time.

The algorithm we propose uses a similar idea to impose a minimum distance between deadlines.

We remind that we are considering one single transaction at a time and that, once we assign the correct deadline to a job $\tau_i^T$, we can also let it start before its offset $\phi_i^T$ thanks to lemma 1.

The second observation is that the dbfs of different nodes are not related to each other. If we restrict our attention to a node $k$, as long as the system is schedulable and all tasks meet their deadlines, we can use the activation of some task in $T_k$ as a reference time to compute all other parameters.
3. IMPLICIT DEADLINE SYNCHRONIZATION PROTOCOL

3.1. The precedence set

Before describing the protocol, we need an additional definition.

**Definition 2.** We define as precedence set $P^\ell_i$ of job $\tau^\ell_i$ the set of all jobs $\tau^h_j$ of tasks $\tau_j \in T_k$, with $h \in \{\ell, \ell - 1, \ell - 2, \ldots \ell - k_0\}$, that have absolute deadline $d^h_j < d^\ell_i$ and absolute activation $\phi^h_j < \phi^\ell_i$, under all possible sporadic activation patterns $\Phi \in \Gamma$.

The number $k_0$ of instances in the past that we have to consider is:

$$k_0 = \left\lfloor \frac{D}{T} \right\rfloor - 1.$$  

The precedence sets impose a partial order on the set of jobs belonging to previous instances. In practice, the deadline $d^\ell_i$ of job $\tau^\ell_i$ must necessarily follow all the deadlines of jobs $P^\ell_i$ under all possible sporadic activation patterns $\Phi$.

Figure 3.1 show graphically these relations and, as we will show in lemma 4, we can just check them when $\Phi^\ell - \Phi^h = (\ell - h)T$. Clearly, cases a and d of figure 3.1 include the possibility that there is not any overlap between $\tau^\ell_i$ and $\tau^h_j$.

**Lemma 4.** Let $\ell$ and $h$ two indexes of instance of transaction, with $\ell > h$. If

- all instance of transaction from $h^{th}$ to $\ell^{th}$ are spaced exactly of $T$
- $d^h_j < d^\ell_i$
- $\phi^h_j < \phi^\ell_i$

then, under all possible sporadic activation patterns $\Phi \in \Gamma$,

$$d^h_j < d^\ell_i \quad \text{and} \quad \phi^h_j < \phi^\ell_i$$
3.1. THE PRECEDENCE SET

**Proof.** The proof follows from the definition of absolute activation and deadline:

\[ d_i^\ell = \Phi_i^\ell + D_i. \]

\[ \phi_i^\ell = \Phi_i^\ell + \phi_i. \]

Under the assumption of periodic activation, \( \Phi_i^\ell = \Phi_i^h + (\ell - h)T \), while a sporadic activation pattern leads to \( \Phi_i^\ell \geq \Phi_i^h + (\ell - h)T \). Then, if we compare periodic deadline \( d_i^\ell \) with sporadic deadline \( d_i'^\ell \):

\[ d_j^h < d_i^\ell = \Phi_i^\ell + D_i \]
\[ = \Phi_i^h + (\ell - h)T + D_i \]
\[ \leq \Phi_i'^\ell + D_i = d_i'^\ell. \]

The same steps are true for activation

\[ \phi_j^h < \phi_i^\ell = \Phi_i^\ell + \phi_i \]
\[ = \Phi_i^h + (\ell - h)T + \phi_i \]
\[ \leq \Phi_i'^\ell + \phi_i = \phi_i'^\ell. \]

\( \square \)

Set \( P_i^\ell \) can be very large, but, as we will show in lemma 8, it is enough to only compute and consider a smaller subset \( P_i^{\ell*} \) that contains at most one job per past instance. Let us show how to compute such reduced subset:

- Set \( P_i^{\ell*} \) contains at most one job per each instance \( \ell, \ell - 1, \ldots, \ell - k_0 \). Initially, \( P_i^{\ell*} \) is empty.
- Considering instance \( h = \ell \), we only need to know the task \( \tau_j \in T_k \) that immediately precedes \( \tau_i \) in the transaction. Then, \( \tau_i^\ell \) is added to \( P_i^{\ell*} \). If \( \tau_i \) has no preceding task in \( T_k \), we skip this step.
- Then, we enter a cycle in which we compute the job for instance \( h \in \ell - 1, \ell - 2, \ldots, \ell - k_0 \). Here we assume that the distance of two consecutive instances of transactions, with index from \( h \) to \( \ell \), is equal to the period. In fact, this is the worst condition for a job \( \tau_j^h \) to achieve absolute deadline \( d_j^h \) smaller than \( d_i^\ell \) and absolute activation \( \phi_j^h \) less than \( \phi_i^\ell \).

If \( P_i^{\ell*} \) is empty, we consider the latest job \( \tau_j^h \in J^h \) that has absolute deadline smaller than \( d_i^\ell \) and absolute activation \( \phi_j^h \) less than \( \phi_i^\ell \). If such a job exists, its deadline \( d_j^h \) is added to \( P_i^{\ell*} \). Else, we skip to the next instance.

If instead, \( P_i^{\ell*} \) contains one or more jobs, let \( d_u^z \) be the greatest deadline among the jobs already in \( P_i^{\ell*} \), related to job \( \tau_u^z \). Consider the latest job \( \tau_j^h \in J^h \) that has absolute deadline in interval \( (d_u^z, d_i^\ell) \) and absolute activation less than \( \phi_i^\ell \). If such a job exists, its deadline \( d_j^h \) is added to \( P_i^{\ell*} \). Otherwise we look for jobs that are in the precedence set of \( \tau_i^\ell \) but not in the one of \( \tau_u^z \); this means that we have to search the latest job \( \tau_j^h \) that has absolute deadline smaller than \( d_i^\ell \) (it will be smaller also than \( d_u^z \)) and absolute activation in interval \( (\phi_u^z, \phi_i^\ell) \).
If such a job exists, its deadline $d^h_j$ is added to $P^*_i$. Else, we skip to the next instance.

- We iterate until instance $k = \ell - k_0$.

An example of the procedure is shown in Figure 3.2. Consider a transaction having 6 tasks, with period $T = 10$ and end-to-end deadline $D = 25$. Hence, we need to consider $k_0 = 2$ instances. The intermediate deadlines are respectively, 3, 6, 10, 14, 21, 25. We assume that tasks $\tau_1, \tau_3, \tau_5$, are allocated on processor $k = 1$, while task $\tau_2, \tau_4, \tau_6$ are allocated on processor $k = 2$. We want to compute the precedence set of job $\tau_2^{\ell}$ (the second task in the bottom line of Figure 3.2). By setting all preceding activations at distance equal to $T$, we have the activation pattern shown in the figure. The precedence set of $\tau_2^{\ell}$ contains:

1. no job of instance $\ell$, because there is no task preceding $\tau_2$ on processor 2;
2. job $\tau_1^{\ell-1}$;
3. job $\tau_5^{\ell-2}$.

According to rule 3, we must check that $d_2^\ell \geq d_4^{\ell-1} + 2$ and $d_2^\ell \geq d_6^{\ell-2} + 1$. Therefore, $d_2^\ell$ can be computed as:

$$d_2^\ell = \max\{a_2^\ell + 25, d_2^{\ell-1} + 10, d_4^{\ell-1} + 2, d_6^{\ell-2} + 1\}.$$ 

In general, the number of elements to maximize is upper bounded by $\min\{k_0, n_k - 1\} + 2$. It depends on the ratio between end-to-end deadline and period, but in no case is greater than $n_k + 1$.

Here there are some strong properties:

**Lemma 5.** all possible sporadic activation patterns $\Phi \in \Gamma$, the distance between $d_i^\ell$ and the deadline $d_h^b$ of any of the jobs in $P_i^\ell$ cannot be less than

$$d_i^\ell - d_h^b \geq (\ell - h)T + D_i - D_j \tag{3.1}$$

**Proof.** Follows directly from the definitions. In fact

$$d_i^\ell - d_j^h = \Phi^\ell + D_i - \Phi^h - D_j$$

$$\geq \Phi^h + (\ell - h)T + D_i - \Phi^h - D_j$$

$$= (\ell - h)T + D_i - D_j$$

**Lemma 6.** If we have a chain of $N$ jobs

$$\tau_{i_1}^{\ell_1}, \tau_{i_2}^{\ell_2}, \ldots, \tau_{i_N}^{\ell_N}$$
3.1. THE PRECEDENCE SET

where each couple of consecutive jobs are connected by a precedence constraint, i.e.
$\tau_{\ell_{j}} \in \mathcal{P}_{i_{j+1}}$ or $\tau_{\ell_{j+1}} \in \mathcal{P}_{i_{j}}$, and each job is at the minimum distance to the next one,
then the distance between the first and the last job of the chain is:

$$d_{\ell_{N}} - d_{\ell_{1}} = (\ell_{N} - \ell_{1})T + D_{i_{N}} - D_{i_{1}}.$$  

**Proof.** First we add and subtract all deadlines of intermediates jobs, then substitute each couple with the minimum distance.

$$d_{\ell_{N}} - d_{\ell_{1}} = (d_{\ell_{N}} - d_{\ell_{N-1}}) + (d_{\ell_{N-1}} - d_{\ell_{N-2}}) + \ldots + (d_{\ell_{2}} - d_{\ell_{1}})$$

$$= (\ell_{N} - \ell_{N-1})T + D_{i_{N}} - D_{i_{N-1}}$$

$$+ (\ell_{N-1} - \ell_{N-2})T + D_{i_{N-1}} - D_{i_{N-2}}$$

$$+ \ldots$$

$$+ (\ell_{2} - \ell_{1})T + D_{i_{2}} - D_{i_{1}}$$

$$= (\ell_{N} - \ell_{1})T + D_{i_{N}} - D_{i_{1}}.$$  

Notice that lemma 6 does not require a precedence constraint between two non consecutive jobs, in particular, between the first and the last. Figure 3.3 shows an example of chain with length $N = 4$: $\tau_{\ell_{1}} \in \mathcal{P}_{i_{2}}$, $\tau_{\ell_{3}} \in \mathcal{P}_{i_{2}}$ and $\tau_{\ell_{3}} \in \mathcal{P}_{i_{4}}$. Clearly, $\ell_{2} = \ell_{3}$.

**Corollary 1.** If we have three jobs, $\tau_{\ell_{1}}$, $\tau_{\ell_{j}} \in \mathcal{P}_{i_{2}}$, $\tau_{\ell_{3}} \in \mathcal{P}_{i_{2}}$ and $\tau_{\ell_{3}} \in \mathcal{P}_{i_{4}}$. Clearly, $\ell_{2} = \ell_{3}$.

**Corollary 1.** If we have three jobs, $\tau_{\ell_{1}}$, $\tau_{\ell_{j}}$, $\tau_{\ell_{3}}$, such that

$$\tau_{\ell_{j}} \in \mathcal{P}_{i_{2}}$$

$$\tau_{\ell_{3}} \in \mathcal{P}_{i_{2}}$$

$$\tau_{\ell_{3}} \in \mathcal{P}_{i_{4}}$$

then, under all possible activation patterns, it is true that the deadline of $\tau_{\ell_{3}}$ is before the deadline of $\tau_{\ell_{j}}$, or in other words

$$d_{\ell_{j}} - d_{\ell_{3}} \geq d_{\ell_{j}} - d_{\ell_{3}}$$  

\[\Box\]
Proof. Follows directly from the definitions and the fact that the distance between \(d^\ell_i\) and \(d^s_i\) is always the sum of distances \(d^\ell_i - d^h_j\) and \(d^\ell_i - d^s_i\).

The next property is useful, in particular for implementations, because it reduces the number of all required \(P^\ell_i\) to one for all tasks.

Lemma 7. If \(\ell\) and \(h\) are both bigger than \(k_0\), \(P^\ell_i\) is exactly equivalent to \(P^h_i\), in the sense that for each \(r^{(\ell-z)}_u \in P^h_i\) there is a \(r^{(\ell-z)}_u \in P^\ell_i\) and vice versa.

Moreover,

\[d^\ell_i - d^{(\ell-z)}_u = d^h_i - d^{(h-z)}_u\]

Proof. From definition of \(P^\ell_i\) and \(P^h_i\) (and lemma 4) we are under the assumption that all instances of transaction between \((\ell - k_0)\)th and \(\ell\)th and between \((h - k_0)\)th and \(h\)th are spaced exactly by \(T\). Then, if we take \(\Phi^\ell\) and \(\Phi^h\) as references of the two groups of instances of transaction (and related jobs) its easy to see that for each activation (deadline) in one group there is an equivalent activation (deadline) in the other group at the fixed distance of \(|\Phi^\ell - \Phi^h|\). As examples:

\[\Phi^{\ell-k_0} - \Phi^{h-k_0} = (\Phi^\ell - k_0T) - (\Phi^h - k_0T) = \Phi^\ell - \Phi^h\]

\[\phi^{\ell-k_0} - \phi^{h-k_0} = (\Phi^\ell - k_0T + \phi_u) - (\Phi^h - k_0T + \phi_u) = \Phi^\ell - \Phi^h\]

\[d^{\ell-k_0}_u - d^{h-k_0}_u = (\Phi^\ell - k_0T + D_u) - (\Phi^h - k_0T + D_u) = \Phi^\ell - \Phi^h\]

From these steps:

\[d^{(\ell-z)}_u < d^\ell_i\]

\[\Phi^\ell - zT - D_u < \Phi^\ell - D_i\]

\[\Phi^\ell - zT - D_u - (\Phi^\ell - \Phi^h) < \Phi^\ell - D_i - (\Phi^\ell - \Phi^h)\]

\[\Phi^h - zT - D_u < \Phi^h - D_i\]

\[d^{(h-z)}_u < d^h_i\]

\[\phi^{(\ell-z)}_u < \phi^\ell_i\]

\[\Phi^\ell - zT - \phi_u < \Phi^\ell - \phi_i\]

\[\Phi^\ell - zT - \phi_u - (\Phi^\ell - \Phi^h) < \Phi^\ell - \phi_i - (\Phi^\ell - \Phi^h)\]

\[\Phi^h - zT - \phi_u < \Phi^h - \phi_i\]

\[\phi^{(h-z)}_u < \phi^h_i\]

we see that \(r^{(\ell-z)}_u \in P^\ell_i\) if and only if \(r^{(h-z)}_u \in P^h_i\).
With a similar reasoning:
\[
d^\ell_i - d^{(\ell-z)}_u = (\Phi^\ell_i - D_i) - (\Phi^\ell_i - zT - D_u)
\]
\[
= (\Phi^\ell_i - D_i - (\Phi^\ell_i - \Phi^h_i)) - (\Phi^\ell_i - zT - D_u - (\Phi^\ell_i - \Phi^h_i))
\]
\[
= (\Phi^h_i - D_i) - (\Phi^h_i - zT - D_u)
\]
\[
= d^h_i - d^{(h-z)}_u.
\]
\[
\square
\]

Clearly, if \( \ell > k_0 \) and \( h < k_0 \), \( P^h_i \) contains jobs with index of transaction in \( [0,h] \) and is equivalent to a subset of \( P^{\ell*}_i \) that contains only jobs with index of transaction in \( [\ell-h,\ell] \).

### 3.2. The protocol

We now present the IDSP algorithm to compute the absolute deadline of a job \( \tau^\ell_i \) at the instant of its activation \( a^\ell_i \). The protocol consists of three simple rules.

**Rule 1** The separation between activation and deadline of \( \tau^\ell_i \) must always be greater than its relative deadline:
\[
d^\ell_i \geq a^\ell_i + D_i
\]

**Rule 2** The second rule mandates a minimum separation between the deadlines of the jobs of the same task:
\[
d^\ell_i \geq d^{\ell-1}_i + T.
\]

**Rule 3** The distance between \( d^\ell_i \) and any job in \( P^\ell_i \) must not be less than the minimum possible distance as computed by Lemma 5. In formula:
\[
\forall \tau^h_j \in P^\ell_i, \quad d^\ell_i \geq d^h_j + (\ell-h)T + D_i - D_j
\]

At run-time, it may happen that a job \( \tau^\ell_i \) is activated before a job of its precedence set, due to the fact that the end-to-end deadline can be larger than the period and previous jobs may complete much earlier than their deadline. In such a case, according to Rule 3, the job is suspended because its deadline cannot be computed until we have computed the deadlines of all the jobs in its precedence set.

From the previous rules, the job deadline \( d^\ell_i \) can simply be computed as the maximum among the RHS of the three inequalities. Notice that we only use parameters that are local to each node (\( a^\ell_i, d^\ell_i \)), or statically known (\( D_i, T, D_i \) and one \( P^{\ell*}_i \) for each task).

Notice also that a different instance of the protocol must be applied for every single transaction.

### 3.3. Proof of correctness

Rule 3 mandates that all the deadlines in the precedence set \( P^\ell_i \) must be computed before we can compute deadline \( d^\ell_i \). The following lemma proves that at run-time it is sufficient to only consider \( P^{\ell*}_i \).

**Lemma 8.** If all the jobs in \( P^{\ell*}_i \) have been assigned a deadline at run-time, then all the jobs in \( P^\ell_i \) have been assigned a deadline.
Proof. By contradiction. Suppose that a job \( \tau^h_j \in P^f_i - P^s_i \) has not been assigned a deadline, and let \( \tau^f_i \) be the first job for which this happens at run-time.

Since \( d^f_j < d^s_i \) but \( \tau^h_j \) does not belong to \( P^s_i \), from rules used to build \( P^s_i \), it must exist a job \( \tau^z_u \in P^s_i \) such that \( h \leq s \leq \ell \) and \( d^f_j < d^z_u < d^f_i \) and \( \phi^h_j < \phi^z_u \). Then, \( \tau^z_u \in P^s_u \).

Since \( \tau^z_u \) has been assigned a deadline at run-time, according to rule 3, \( \tau^h_j \) must have been assigned a deadline, against the hypothesis. \( \Box \)

The following lemma proves that the absolute deadlines assigned by algorithm IDSP never exceed the absolute deadlines assigned by an algorithm that uses global time.

Lemma 9. Let \( a^\ell \) be the activation of the \( \ell \)-th instance of transaction \( T \). Under the assumption that the transaction is schedulable, the absolute deadline \( d^\ell_i \) of every job \( \tau^\ell_i \), computed dynamically using Equations 3.4–3.6, is never larger than \( a^\ell + D_i \).

Proof. We want to prove that

\[
\forall \ell, i \quad d^\ell_i \leq a^\ell + D_i
\]

We prove it by induction on both \( \ell \) (the instance index) and \( i \) (the task index).

We start proving the first instance, when \( \ell = 0 \). Notice that since \( \ell = 0 \) then only rules 3.4 and 3.6 can be applied. By induction on \( i \). If \( i = 1 \) then we have to prove that

\[
d^0_1 \leq a^0 + D_1
\]

In this case only rule 3.4 applies. Hence

\[
d^0_1 = a^0 + D_1 = a^0 + D_i
\]

as required. Now we prove it for any \( i \) assuming it is true for all smaller task indexes.

Notice that \( P_{i}^{0*} \) contains at most one job, related to the same instance, the first one. If \( d^0_i \) has been computed according to Eq. 3.4, then

\[
d^0_i = a^0 + D_i = f^0_{i-1} + D_i \leq d^0_{i-1} + D_i \leq a^0 + D_{i-1} + D_i = a^0 + D_i
\]

as required. If, instead rule 3.6 applies then \( P_{i}^{0*} \) contains one job, \( \tau^0_j \) and we have

\[
d^0_i = d^0_j + D_i - D_j \leq a^0 + D_i + D_j - D_j = a^0 + D_i
\]

which concludes the proof for the instance of transaction \( \ell = 0 \).

Now we prove by induction on \( \ell \), hence assuming Eq. 3.7 true for \( \ell - 1 \). We also prove it by induction on \( i \). If \( i = 1 \) then we have to prove that

\[
d^\ell_1 \leq a^\ell + D_1
\]

If rule 3.4 is applied, then

\[
d^\ell_1 = a^\ell + D_1 = a^\ell + D_i
\]

as required. If, instead rule 3.5 is applied, then

\[
d^\ell_1 = d^{\ell-1}_1 + T \leq a^{\ell-1}_1 + D_1 + T \leq a^\ell_1 + D_1
\]

as required. If rule 3.6 is applied, means that

\[
d^\ell_1 = \max_{\tau^h_j \in P^f_i} \left\{ d^h_j + (\ell - h)T + D_1 - D_j \right\}
\]
Suppose that \( \tau^z_u \) is the job that gives the max, then

\[
d^\ell_i = d^z_u + (\ell - z)T + \overline{D}_1 - \overline{D}_u \leq a^z + \overline{D}_a + (\ell - z)T + \overline{D}_1 - \overline{D}_u \leq a^\ell + \overline{D}_1
\]

We conclude by proving it for any \( i \), assuming it true for the preceding ones. If \( d^\ell_i \) has been computed according to Eq. 3.4, then

\[
d^\ell_i = a^\ell + D_i = f^\ell_{i-1} + D_i \leq d^\ell_{i-1} + D_i \leq a^\ell + \overline{D}_{i-1} + D_i = a^\ell + \overline{D}_i
\]
as required. If rule 3.5 applies then

\[
d^\ell_i = d^\ell_{i-1} + T \leq a^{\ell-1} + \overline{D}_i + T \leq a^\ell + \overline{D}_i
\]

As before, if rule 3.6 is applied, means that

\[
d^\ell_i = \max_{\tau^h_j \in P^\ell_i} \{ d^h_j + (\ell - h)T + \overline{D}_i - \overline{D}_j \}
\]

Suppose that \( \tau^z_u \) is the job that gives the max, then we have

\[
d^\ell_i = d^z_u + (\ell - z)T + \overline{D}_i - \overline{D}_u \leq a^z + \overline{D}_u + (\ell - z)T + \overline{D}_i - \overline{D}_u \leq a^\ell + \overline{D}_i
\]

which concludes the proof.

Lemma 9 guarantees that, if the transaction is locally schedulable on each node, then no task misses the deadlines that a global algorithm would have assigned on each node. From it descends also:

**Corollary 2.** Let \( a^\ell \) be the activation of the \( \ell \)-th instance of transaction \( T \). Under the assumption that the transaction is schedulable, the actual absolute activation \( a^\ell_i \) of every job \( \tau^\ell_i \), computed dynamically using Equations 3.4–3.6, is never larger than \( a^\ell + \phi_i \):

\[
(3.9) \quad \forall \ell, i \quad a^\ell_i \leq a^\ell + \phi_i.
\]

**Proof.** If we combine Eq. 3.4 with the result of lemma 9, i.e. Eq. 3.7, we obtain:

\[
\forall \ell, i \\

a^\ell_i + D_i \leq d^\ell_i \leq a^\ell + \overline{D}_i \\
a^\ell_i \leq a^\ell + \overline{D}_i - D_i = a^\ell + \phi_i
\]

that is exactly what we want to prove.

Now we prove that the protocol assigns deadlines so that the sporadic dbf, as computed in Section 2.7 and 2.6, is always respected.

**Lemma 10.** Let \( \tau^h_j \) be any job in \( P^\ell_i \). Under the assumption that the transaction is schedulable and all deadlines are assigned using Equations 3.4–3.6, the distance between \( d^h_j \) and \( d^i_i \) is never smaller than the distance as computed in Equation (3.1).

**Proof.** For \( \tau^h_j \in P^\ell_i \), the lemma follows directly from Rule 3. For the other jobs, it is easy to see that we can apply a similar reasoning to the one in Lemma 8 to derive the thesis.
Now the main theorem.

**Theorem 1.** Under the assumption that all tasks execute for less than their WCET, and that for any interval the sum of the sporadic dbfs computed off-line never exceeds the length of the interval, then the dbf computed on-line by IDSP is always less or equal to the sporadic dbf.

**Proof.** The proof is by contradiction. Let $[t_0, t_1]$ be the first and smaller interval in which the demand computed on-line exceeds the sporadic dbf. Smaller means that if we increase $t_0$ of any amount, or decrease $t_1$, the demand of this new interval is smaller. Then, $t_0$ must be coincident with a task activation and $t_1$ with a deadline. We will show that by moving deadline and activations in a conservative way (i.e. without decreasing the on-line demand in $[t_0, t_1]$), we reach one of the situations enumerated by the algorithms in Figures 2.8 and 2.6. Hence we obtain a contradiction, and the thesis is proved.

Notice that it is not important to consider jobs with absolute run-time activation smaller than $t_0$ or absolute run-time deadline bigger than $t_1$. In fact, after all changes, they may or may not came inside the inside the interval but in both cases they cannot decrease the demand. Then we will consider only jobs that are fully contained inside the interval, i.e. jobs with both absolute run-time activation after $t_0$ and absolute run-time deadline before $t_1$, and ensure that they will always stay inside the interval.

The idea is to increase all absolute activation $a_\ell^i$ of jobs until $d_\ell^i - a_\ell^i = D_i$. Then, starting from instance of transaction with smallest index and then increasing the index, we move all jobs to the left, one by one starting from smaller indexes, until for each job the absolute activation is greater or equal to $t_0$ and rules 2 and 3 are still true. If happens that the one of inequalities of rule 3 is verified as equality with one job of previous instances of transaction, than we move all jobs with the same transaction index and smaller task index, to the right, until the rule 3 is verified as equality also for jobs of the same instance of transaction. Notice that moving to the right, cannot make any absolute deadline bigger than $t_1$. After all jobs are moved, the distance between two jobs $\tau_\ell^i, \tau_\ell^j$ of the same instance of transaction is exactly $D_i - D_j$. Then, for each instance of transaction, it is possible to find an absolute activation $a_\ell^i$. The distance between two consecutive activations is greater or equal to the period. This prove that it is possible to find a pattern of activations of transaction, enumerated by 2.8, that produces a demand greater or equal to the one computed at run-time.

From here, we go deeper and more formally in the proof.

We define $J$ as the set of jobs that are fully inside the interval:

$$J = \left\{ \tau_\ell^i : \quad a_\ell^i \geq t_0 \quad \text{and} \quad d_\ell^i \leq t_1 \right\}.$$

It is also useful to define the sub-set of jobs of a specific instance of transaction as:

$$J^\ell = \left\{ \tau_\ell^i : \quad a_\ell^i \geq t_0 \quad \text{and} \quad d_\ell^i \leq t_1 \right\}.$$

We will use the symbol $\phi_\ell^i$ and $d_\ell^i$ to identify the modified activation and deadline respectively, while $a_\ell^i$ and $d_\ell^i$ are their on-line versions. Initially, $\phi_\ell^i = a_\ell^i$ and $d_\ell^i = d_\ell^i$, then, step by step, we change the value of $\phi_\ell^i$ and $d_\ell^i$. 
3.3. PROOF OF CORRECTNESS

The rule 1 ensure that, on-line, the distance \( d^\ell_i - a^\ell_i \geq D_i \), than if we increase \( \phi^\ell_i \), without change \( d^\ell_i \), there is no way to move the activation outside the interval, then if we set

\[ \forall i \quad \phi^\ell_i = d^\ell_i - D_i \]

we cannot change the demand in the interval. Notice that the change of a job activation has no effect on other jobs. Figure 3.4 shows an example.

By induction, we will show that it is possible to obtain, for all jobs in \( J \)

\[ \forall \ell, i, j \quad \tau^\ell_i \in J^\ell, \tau^\ell_j \in J^\ell \]
\[ d^\ell_i - d^\ell_j = D_i - D_j. \]

Before each step, the condition is that all activations are bigger or equal to \( t_0 \) and all deadlines are small or equal to \( t_1 \):

\[ \forall i, \ell \quad \phi^\ell_i \geq t_0 \quad \text{and} \quad d^\ell_i \leq t_1. \]
We will ensure that this is true after each step. After the induction, we will see also how achieve:

$$\forall \ell, h \quad \ell > h \quad J^\ell \neq \emptyset, \ J^h \neq \emptyset$$

$$\Phi^\ell - \Phi^h \geq (\ell - h)T.$$  

(3.12)

If we obtain a set of activations and deadlines that satisfy eq. 3.10, eq. 3.11 and eq. 3.12, then we build a scenario that contains all jobs in $J$ and that is identified by a sporadic activation pattern $\Phi$ compatible to the ones studied by sporadic dbf. But this lead to a contradiction and prove the theorem.

*Induction on the smallest index of transaction.*

Let $z$ the smallest index of transaction that has some jobs fully inside the interval $[t_0, t_1]$, and let $u$ the smallest job index in $J^z$. Then we can move this job to the left, until its activation coincide with $t_0$:

$$\phi^z_u = t_0$$

$$d^z_u = t_0 + D_u.$$
Notice that also this movement has no effect on other jobs, because if there is a relation through rules 2 and 3, the on-line deadline was equal or bigger than $t_0 + D_u$ and then the distance between the two deadlines cannot decrease and invalidate one rule. Obviously Eq. 3.11 is respected.

Now we move all jobs in $\mathcal{J}^z$, starting from the smallest. Let $e$ the smallest index that is still waiting to be examined. Also in this case we try to move left, until is possible to respect the rules, i.e. when the distance between the deadline of current job and the previous deadline is equal to the difference of intermediate deadlines.

\[
\phi^z = d^z_u + D_e - D_u - D_e
\]

\[
d^z_e = d^z_u + D_e - D_u
\]

Also in this case, the movement has no effect on other jobs $\tau^h_j$, in the same instance (i.e. $\tau^h_j \in \mathcal{J}, j > e$ and $h = z$) and in future instances (i.e. $\tau^h_j \in \mathcal{J}, h > z$). In fact combining the fact that the new reference $d^z_u$ is equal or smaller than $d^z_u$ and that the lemma 10 applied to $\tau^u_z$ and $\tau^z_u$ imposes $d^z_e - d^z_u \geq (z - z)T + D_e - D_u$, then the value of $d^z_e$ cannot be greater than $d^z_e$. Hence the distance between the two deadlines, $d^z_i - d^z_j$, cannot decrease and invalidate one rule. Moreover, Eq. 3.11 is respected because we reduce the initial value of activation and deadline without allowing them to be smaller than the deadline of previous job in $\mathcal{J}^z$, that is bigger than $t_0$.

After we compute the new activation and deadline for all jobs in $\mathcal{J}^z$ it is easy to see that all deadlines are space out exactly as in the off-line case, i.e. :

\[
\forall i, j \quad \tau^z_i \in \mathcal{J}^z, \tau^z_j \in \mathcal{J}^z
\]

\[
d^z_i - d^z_j = D_i - D_j.
\]

Then we can set the corresponding absolute activation of transaction equal to

\[
\Phi^z = d^z_u - D_u.
\]

It is important to notice that the same value can be found applying the same equality to all jobs in $\mathcal{J}^z$:

\[
\forall i \quad \tau^z_i \in \mathcal{J}^z, \quad \Phi^z = d^z_i - D_i.
\]

This conclude the first step of the induction.

\textit{Induction on a generic index of transaction (part 1).}

The next step is try to make the same steps for all instances of transactions, starting from small indexes and then increasing until we cover all instances. Let $\ell$ the current instance of transaction and suppose that Eq. 3.10 and Eq. 3.11 are true for all instances $h < \ell$.

We move all jobs in $\mathcal{J}^\ell$, starting from the smallest and then, step by step, increasing the job index. Let $u$ the smallest index that is still waiting to be examined. Also in this case we try to move left, until is possible to respect the IDSP protocol rules 1 and 3, without going outside the interval. This means that the new absolute activation and
absolute deadline must respect these Equations:

\[
\begin{align*}
\phi_u^\ell & = \phi_u^\ell + D_u \\
\phi_u^\ell & \geq t_0 \\
d_u^\ell & \geq d_h^\ell + \overline{D}_u - \overline{D}_e \\
d_u^\ell & \geq (\ell - h)T + d_h^h + \overline{D}_u - \overline{D}_e \\
\end{align*}
\]

(3.13)

Figure 3.6. Induction on a generic index of transaction (part 1).
There are three possibilities:

- the activation of the job coincide with $t_0$:
  \[ \phi_\ell^u = t_0 \]

  This may happen if there is not any job in $P^u_\ell$ that is also in $J^\ell$. This implies that $\tau^\ell_u$ is the job with smallest index in $J^\ell$. Here we set
  \[ \phi_\ell^u = t_0 \]
  \[ d_\ell^u = t_0 + D_u. \]

- there is another job $\tau^e_e \in J^\ell \cap P^u_\ell$, with $e < u$ (then $\tau^\ell_e$ is already examined) and with a modified deadline such that
  \[ d_\ell^u - d_\ell^e = D_u - D_e. \]

  Here we set
  \[ \phi_\ell^u = d_\ell^e + D_u - D_e - D_j \]
  \[ d_\ell^u = d_\ell^e + D_u - D_e. \]

- there is another job $\tau^h_i \in J \cap P^u_\ell$, with $h < \ell$ (then $\tau^h_i$ is already examined) and with a modified deadline such that
  \[ d_\ell^u - d_\ell^i = (\ell - h)T + D_u - D_i. \]

  Here we set
  \[ \phi_\ell^u = d_\ell^i + D_u - D_i - D_j \]
  \[ d_\ell^u = d_\ell^i + D_u - D_i. \]

Notice that with this assignment, we do not ensure the minimum distance among deadlines of the same instance of transaction. For an example, see figure 3.6.

As before, the movement has no effect on other jobs: the minimum distance with already examined jobs is ensured by construction and for other jobs in $J$ the minimum distance cannot decrease, because a reasoning similar to the one above lead to $d_\ell^u \leq d_\ell^i \leq t_1$. Moreover, by construction, $\phi_\ell^u \geq t_0$. Then Eq. 3.11 is respected.

*Induction on a generic index of transaction (part 2)*.

After we check (and eventually change) all jobs in $J^\ell$, we made another iteration on jobs indexes, in order to increase deadlines until 3.10 is true also for these jobs. Let $\tau^\ell_i$ the last job in $J^\ell$, i.e. the one with the biggest index (and deadline), the we set

\[ \forall j \quad \tau^j_j \in J^\ell \]

(3.14)

\[ \phi^j_j = d^j_j - D_i + D_j - D_j \]

\[ d^j_j = d^j_j - D_i + D_j. \]

Now there is the problem. In fact, may happen we have to increase both absolute activation and absolute deadline to satisfy Eq. 3.14, but if we increase a deadline, then we have to increase also other absolute deadlines connected to this one by precedence.
set, in order to maintain the minimum distance among deadlines, i.e. to not break the three rules of IDSP. Nevertheless we can prove that the value of an absolute deadline never became bigger than $t_1$. Obviously the new value for absolute activation remains bigger than $t_0$:

$$\phi^\ell_{-j} \geq t_0.$$
3.3. PROOF OF CORRECTNESS

Whit reference to figure 3.7, let $\tau_j^\ell$ the job that we move to the right (i.e. we increase $d_j^\ell$) and let $\tau_u^z$ the job that is moved in order to verify the rule 3 (i.e. $\tau_j^\ell \in P_u^z$). $\tau_j^\ell$ cannot be the last job in $J^\ell$, $\tau_i^\ell$, because $\tau_i^\ell$ does not move in this phase.

Moreover, $\tau_i^\ell \notin P_u^z$, otherwise the minimum distance required between $d_u^z$ and $d_j^\ell$ is constrained by the distance $d_u^z$ and $d_j^\ell$. But $d_j^\ell$ did not move, then $d_u^z - d_j^\ell$ is already bigger than $(z - \ell)T + D_u - D_j$, against the hypothesis.

Now we move $\tau_j^\ell$ and $\tau_u^z$ in order to be in the condition where there is the minimum distance between $\tau_j^\ell$ and $\tau_i^\ell$, and between $\tau_j^\ell$ and $\tau_u^z$. There are three possible possibility, as show in figure 3.8:

1) $d_j^\ell \geq d_u^z$ and $\phi_j^\ell > \phi_u^z$
2) $d_j^\ell > d_u^z$ and $\phi_j^\ell \leq \phi_u^z$
3) $d_j^\ell < d_u^z$ and $\phi_j^\ell > \phi_u^z$.

The first and second case are easy, in fact $t_1 \geq d_j^\ell \geq d_u^z$.

The third case, is quite complex. First, notice that $\phi_u^z > t_0$. In fact, we know that

- $\phi_j^\ell \geq t_0$
- $\tau_j^\ell \in P_u^z$
- from the definition of $P_u^z$

$$\forall \tau_e^u \in P_u^z \quad a_e^u < a_u^z.$$
Notice also that, in this phase, \( \tau^i \) cannot be moved to the left, i.e. decrease both absolute activation and deadline, because \( d^i \) is already at the minimum distance with one or more jobs in \( J \cap P^i \) and, if we follow the sequence of minimum distances imposed during previous steps of iteration by Eq. 3.13 and Eq. 3.14, we build a chain of constraints that arrives to \( t_0 \). See figure 3.9a and figure 3.9b for two examples of chain.

Let \( C^i \) this chain, i.e. all jobs in \( J \) that, with their precedence constraints, forced the minimum value for \( d^i \). For sure, the smallest absolute job activation in this set coincide with \( t_0 \), otherwise we can move all jobs to the left, until the smallest absolute job activation coincide with \( t_0 \), without broke any precedence constraint.

It is interesting study the maximum absolute job deadline in \( C^i \):

\[
d_{max} = \max_{\tau_v^i \in C^i} \{d_v^i\}.
\]

If \( d_{max} \geq d_u^z \), everything is fine, because

\[
d_u^z \leq d_{max} \leq t_1
\]

If instead \( d_{max} < d_u^z \), then the job ( \( \tau_z^v \) ) with the smallest absolute job activation must be in the precedence set of \( \tau^z_u \), because:

- \( \phi_z^v = t_0 < \phi_z^u \) and \( d_u^z \leq d_{max} < d_u^z \)
- we are in the condition used to compute the precedence set: \( \Phi_z^z - \Phi_v^v = (z - v)T \).

This result derives from the lemma 6 applied to the chain that connect \( \tau_u^z \) to \( \tau^z_e \).

But, if \( \tau_z^v \in P_u^z \) then it is not possible that the previous value of the deadline of \( \tau_u^z \) was smaller than \( d_u^z \), because that value is already the minimum that satisfy the rule 3 of IDSP.

With this, we conclude that the movement may have effect on other jobs, but all deadlines remains smaller or equal than \( t_1 \). Then Eq. 3.11 is respected. Moreover also 3.10 is respected for jobs in \( J^f \) (see eq. 3.14) and conclude the induction.
As before, we can use any deadline of jobs in $J^\ell$ to compute the corresponding absolute activation of transaction:

$$\forall i \in J^\ell, \Phi^\ell = d^\ell_i - D_i.$$  

*Enforce distance among transaction activations.*

**Figure 3.10.** Enforce distance among transaction activations.

Now we have to ensure that two consecutive instance of transactions that has some jobs $J$ are correctly spaced. Then, if this is true for all couples of consecutive activations of transaction, is easy to see that is true for all couples, as required by eq. 3.12, because changes may affect following instances of transaction the best way is to start from smaller indexes and increases step by step.
Let \( \ell \) and \( h \) the index of transaction under test, with \( \ell > h \). If exist a job \( \tau_{\ell}^i \in J^\ell \) and a job \( \tau_h^j \in J^h \) such that \( \tau_h^j \in P_{\ell}^i \), then, from the definition of \( P_{\ell}^i \), must be \( \Phi_{\ell} - \Phi_h^h \geq (\ell - h)T \).

If there is not any jobs that enforce a precedence constraint, the \( \Phi_{\ell} - \Phi_h^h \) can be smaller, equal and greater than \( (\ell - h)T \). Obviously we have to study the case smaller, like the example in figure 3.10.

Then we increase \( \Phi_{\ell} \) until we reach the value:

\[
\Phi_{\ell} = \Phi_h^h + (\ell - h)T. \tag{3.15}
\]

Then we update absolute activations and deadlines of jobs in \( J^\ell \):

\[
\phi_{\ell}^i = \Phi_{\ell} + D_i - D_i^d^\ell,
\]

\[
\underline{d}_{\ell}^i = \Phi_{\ell} + \overline{D}_i.
\]

Also in this case it is possible that one or more changed deadline \( \underline{d}_{\ell}^i \) is in the precedence set of a following job \( \tau_u^z \) and that is necessary to update \( \underline{d}_{\ell}^i \) in order to maintain the minimum distance between these deadlines. Moreover, if we want preserve what we do in previous steps with Eq. 3.14, i.e. minimum distance between deadlines of jobs of the same instance of transactions, we have to increase all activations and deadlines in \( J^z \) of the same quantity. Without loss generality, we can suppose that \( \underline{d}_{\ell}^i \) is the biggest deadline in \( J^z \).

As before, if we update \( \underline{d}_{\ell}^i \) means that there is a chain of minimum distances and precedence constraints. Let \( \phi_{\ell}^i \) the first job of instance of transaction \( \ell \). Then the activation of \( \tau_{\ell}^i \) coincide with \( t_0 \) or is greater. If \( \phi_{\ell}^i > t_0 \) means that there is a sequence of chains of minimum distances (Eq. 3.14) and movements of instance activations done in this phase (Eq. 3.15) that enforces the distance between \( \tau_{\ell}^i \) and a job \( \tau_v^e \) with \( \phi_v^e = t_0 \), otherwise is possible to move \( \tau_{\ell}^i \) and all jobs connected to the left without make any activation smaller than \( t_0 \). All these constraints impose a distance between transaction activations equal to the periodic distance:

\[
\Phi_z^z - \Phi_v^z = (z - v)T.
\]

Cannot be \( \underline{d}_{\ell}^i > \underline{d}_{v}^e \), otherwise \( \tau_v^e \in P_{\ell}^z \) and we go against the hypothesis that we increase the value of \( \underline{d}_{\ell}^i \) because this value is the smallest that it can assume without violate the minimum distance from \( \underline{d}_{v}^e \):

\[
d_{\ell}^i - d_{v}^e = \Phi_z^z + \overline{D}_u - \Phi_v^v - \overline{D}_e = (z - v)T + \overline{D}_u - \overline{D}_e.
\]

Then \( d_{\ell}^i \leq d_{v}^e \leq t_1 \). Note that the update of all deadlines in \( J^z \) may lead to increase also deadlines of following jobs (i.e. with a bigger index of instance of transaction) but we can apply the same reasoning to prove that these deadlines remains smaller than \( t_1 \). Obviously, this sequence of updates cannot be infinite because each change can only affect following jobs, without any kind of loops.

At the end, after we repeat these steps for all couples of consecutive activations of transaction, we obtain that Eq. 3.10 and Eq. 3.12 are respected for all jobs in \( J \). As told before, we build a sporadic activation pattern that is studied by the sporadic dbf without remove any job form the interval, then cannot be that online demand is bigger than the off line one, and we obtain the contradiction that proves the thesis. \( \square \)
We prove that IDSP works correctly with sporadic demand bound function. But it is possible to extend this result also to holistic analysis:

**Theorem 2.** Under the assumption that all tasks execute for less than their WCET, and that the node is computed schedulable by holistic analysis, then no task in the system will miss its absolute deadline computed by IDSP.

**Proof.** First of all, we recall that holistic analysis bases its results on the response time of each task, that is used to check if all jobs end before their absolute deadline. Clearly response times are computed within busy periods, i.e. intervals of time where there is always at least one job ready to be scheduled.

We want to make a proof by contradiction and we suppose that at $t'_0$ starts the first busy period that leads to a deadline miss, at $t'_1$. Clearly may happen that $t'_0$ does not coincide with an activation and $t'_1$ does not coincide with a deadline. We call $\tau^z_{u}$ the job that miss the deadline $d^z_{u}$.

The first step is build a new interval $[t_0, t_1]$, internal to $[t'_0, t'_1]$, that contains all jobs that have to run in the original interval, i.e. we do not remove any contribution that lead to the deadline miss. Notice that is important that the new interval contains only jobs with deadline smaller than $d^z_{u}$, because others cannot force $\tau^z_{u}$ to miss its deadline.

We set $t_0$ equal to the first offset $\phi^b_i$ bigger than $t'_0$, in order to ensure that the sub-interval $[t_0, t_1]$ contains all significant activations. Notice that $\tau^b_i$ may be or not the job that starts the busy period $[t'_0, t'_1]$.

Because we are not interested on jobs with a priority lower than the one that miss the deadline, we set $t_1 = d^z_{u}$.

Repeating the same steps described in the proof of Theorem 1 on interval $[t_0, t_1]$, we obtain the same set of jobs spaced as in an off-line sporadic activation pattern. Then this case should be studied by holistic analysis and is not possible that $\tau^z_{u}$ miss its deadline. That conclude the proof.

□

### 3.4. Improving response time

It is possible to reduce the response time, allowing the execution of a job $\tau^b_i$ as soon as it arrives, even if the job should be suspended to ensure the minimum distance of deadline required by rule 3. Notice that, if we set a deadline that is smaller than the one computed after all jobs in $P^b_i$ arrives, it may happen that this job steals processor time to jobs with deadline within the temporary and the correct deadlines, leading those penalized jobs to miss their deadline.

But, if we allow $\tau^b_i$ to run as background job, i.e. when there is not any job with a deadline set ready to run, we cannot steal processor time and neither we can decrease the performance of $\tau^b_i$. Moreover, we can allow $\tau^b_i$ to run even if there are some jobs with a very large deadline $d^h_j$ and we are sure that $d^b_i$ cannot be bigger than $d^h_j$, because this means that $\tau^b_i$ has an higher priority than $\tau^h_j$. Unfortunately there is no way to estimate well the temporary deadline without knowing the absolute (and global) actual activation time $a^\ell$ of the instance of transaction $\ell$, but if we suppose that $a^\ell$ coincides with the arrival time of $\tau^b_i$ we can apply the lemma 9, and then impose a temporary intermediate deadline equal to $\overline{D}_i$. Obviously the actual absolute activation time of the
transaction \( \ell \) is smaller, then also the correct deadline is furthermore smaller. Notice, lemma 9 works if there are no deadline misses in the system.

Because \( D_i \) can be bigger than any \( D_i \) in the system (except for few tasks, like the ones with small index \( i \)), usually this temporary deadline is used to order background jobs (the ones that should be suspended), giving more priority to jobs with small index \( i \).

Another interesting value, used to overestimate the value of absolute deadline, is the end-to-end deadline \( D \), that, used for all jobs, enforce a FCFS queue among background jobs of the same transaction: \( d_i^\ell = a_i^\ell + D = f_i^\ell + D \).

Notice that, if \( \tau_j^\ell (i \neq j) \) is the first job related to the instance of transaction \( \ell \) activated on the current node, it is possible to compute the temporary deadline of \( \tau_i^\ell \) adding \( D_i \) to the arrival time of \( \tau_j^\ell \), reducing the difference between the temporary and the correct deadlines.
CHAPTER 4

Assign deadlines

4.1. Optimization goals

In this Section, we examine some possible metrics to evaluate an assignment of task’s intermediate deadlines in a transaction.

Ideally, we would like to make the total dbf on each node, given by the sum of the dbfs of all transactions on the same node, as low as possible. We will present our definition of “minimal dbf” later on. Right now, we compare two possible strategies for minimizing the demand bound function: minimizing separately the dbf$_k(t)$ of each transaction or minimizing the sum dbf$_k(T, t)$ of all transactions together.

We can make the following observations:

1. Minimizing each single dbf$_k(t)$ is much less complex. In fact dbf$_k(t)$ has a periodic pattern of steps with a period equal to $T$, while, as happens for periodic tasks, the period of the pattern of steps related to $\sum_T$ dbf$_k(T, t)$ is the hyper-period $H = \text{lcm}_T\{T\}$, i.e. the least common multiplier among periods of all transactions in the system. Clearly, this value can be extremely large compared with each single period.

2. From the designer point of view, separating the time requirement of each transaction makes it easier to modify the behavior of each transaction independently from the others, guaranteeing that the rest of the system will not be influenced. Instead, if the minimization is made globally, then any time a modification occurs in a transaction the optimal deadline assignment would need to be recomputed. Moreover, it is not easy to understand how a modification would impact on the rest of the system.

3. However, the global minimal dbfs always smaller than the sum of the $n$ solution coming from the minimization of the single transactions. Therefore, minimizing each transaction separately may not lead to a global optimal solution.

We believe that the benefits coming from analyzing each transaction separately make it worth to follow this path. In addition, we will gain a better understanding in the mathematical structure of the problem, which may lead to further developments in the future.

4.1.1. Single node goal. To simplify the presentation, without loss of generality, we first assume to work on a single processor (identified by index 1). Given a dbf and the corresponding set of scheduling points $S_1$ we introduce:

\[
\alpha(S_1) \overset{\text{def}}{=} \sup_{(t; w) \in S_1} \frac{w}{t}
\]
that is the minimum slope of a line passing from the origin that can upper bound \( \text{dbf} \). Basically \( \alpha(S_1) \) is the minimum speed (or bandwidth) of a processor that can feasibly schedule a transaction whose scheduling points are in \( S_1 \). Hence it is quite straightforward to set our primary target as finding the deadline assignment that minimizes \( \alpha(S_1) \). However there is a (trivial) lower bound to this minimization.

**Lemma 11.** Given any transaction \( T \), let \( S_1 \) be the set of scheduling points. Then

\[
\alpha(S_1) \geq U_1 = \frac{C_1}{T}
\]

**Proof.** Let \((t'; w')\) be one scheduling point in the set of periodic scheduling points \( S_p \) (we know it exists because \( S_p \neq \emptyset \)). Then

\[
\alpha(S_1) = \sup_{(t; w) \in S_1} \frac{w}{t} \geq \sup_{b \in \mathbb{N}} \frac{w' + bC}{t' + bT} \geq \frac{C}{T} = U.
\]

as required. \( \square \)

Lemma 11 sets a lower bound on the value of \( \alpha(S_1) \). We call \( \text{minSlope}(T) \) the procedure that takes the transaction as input and assigns the deadlines that minimize the slope \( \alpha(S_1) \).

If some deadline assignment has \( \alpha(S_1) = U_1 \), it is reasonable to set a secondary target in our process of minimizing the \( \text{dbf} \). For this purpose we define

\[
\Delta(S_1) \overset{\text{def}}{=} \inf_{(t; w) \in S_1} \left\{ t - \frac{w}{U_1} \right\}
\]

that is the minimum horizontal displacement \( \Delta_1 \) such that function \( Z(t) = U_1(t - \Delta_1) \) is an upper bound to \( \text{dbf}(t) \). We call \( \text{maxDelta}(T) \) the procedure that takes the transaction as input and assigns the deadlines that maximizes \( \Delta(S_1) \).

This selection of the secondary target of the \( \text{dbf} \) minimization has a strong motivation in the theory of the hierarchical scheduling. In fact, if \( \Delta(S_1) \geq 0 \), then the \( \text{dbf} \) can be upper bounded by the linear function \( \alpha_1(t - \Delta_1) \) and the transaction can be feasibly scheduled on a server that provides a bandwidth of \( U_1 \) and a delay of \( \Delta_1 \) (the \( (\alpha, \Delta) \)-service curves have been introduced by Feng et al. [FM02] and have been used to model the service of virtual resources by other authors [LB05, SL03]).

In Figure 4.2 we depict the two goals of an optimization procedure: the arrow labeled with “1” represents the objective of minimizing the slope of the upper bound line; only if the slope is equal to the transaction utilization, it makes sense to try to maximize the horizontal displacement \( \Delta_1 \) (arrow labeled with “2”).

As final step of the \( \text{dbf} \) minimization we try to maximize the minimum deadline of the transaction simply defined as

\[
\minDL(S_1) = \min_{(t; w) \in S_1} t
\]

keeping the constraints on \( \alpha(S_1) \) and \( \Delta(S_1) \) as found in the previous two steps. The procedure that accomplishes this last step is called \( \text{pushMinDL}(T) \).

Figure 4.1 reports the pseudo-code of the complete optimization procedure also depicted in Figure 4.2. In step one (line 2 in the code) we depict the procedure \( \text{minSlope} \). If
1: \textbf{procedure} \textsc{optimAssignDL}(\mathcal{T})
2: \quad \text{minSlope}(\mathcal{T})
3: \quad \text{if } \alpha(S_1) = U_1 \text{ then}
4: \quad \quad \text{maxDelta}(\mathcal{T})
5: \quad \text{end if}
6: \quad \text{pushMinDL}(\mathcal{T})
7: \textbf{end procedure}

\textbf{Figure 4.1.} The optimization algorithm.

the found assignment has \(\alpha(S_1) = U_1\), then we proceed also with the step two \text{maxDelta}. Finally we conclude by \text{pushMinDL}.

\textbf{Figure 4.2.} The minimization of the demand bound function.

\subsection*{4.1.2. Multi node goal.} For a distributed transaction (i.e. a transaction whose tasks are allocated to distinct nodes), we must consider one \text{dbf}_k(t) for each node \(k\). Let \(S_k\) be the set of scheduling points of \text{dbf}_k(t). Let \(U_k\) be the utilization of the transaction under analysis on node \(k\), defined as \(U_k = \sum_{\tau_j \in T_k} \frac{C_j}{T} \). We want to minimize:

\[
\alpha_k = \sup_{(t,w) \in S_k} \frac{w}{t}.
\]

However, the sum of all tasks deadlines must be equal to the transaction end-to-end deadline. As a consequence, if we minimize the slope on node \(k\), the slope on another node may increase. In other words, it is not possible to minimize every \text{dbf}_k(t) independently of the others. Therefore, for a distributed transaction the goal is to minimize some function that combines the slopes and the delta of all the \text{dbfs}. Let \(h(\cdot)\) be this function.
We can write the optimization goal as:

$$\text{minimize } h(\alpha_1, \ldots, \alpha_p, \Delta_1, \ldots, \Delta_p)$$

subject to $\sum_{j=1}^{n} D_j = D$

where $\Delta_k \overset{\text{def}}{=} \Delta(S_k)$. By choosing the most appropriate $h(\cdot)$ function, it is possible to select a different optimization goal. The optimization goals proposed in this thesis will be described in Section 4.3.3.

4.2. Single node

The single-node problem has no striking practical applicability. In fact, it is not clear why a designer should prefer to use a transaction on a single node when it is possible to achieve the same outcome by a unique task obtained by gluing together all the tasks of the transaction. Nonetheless, some of the results are interesting and unexpected and may help to improve our understanding of the problem.

In the special case when $D \leq T$, the following theorem holds.

**Theorem 3.** Given a transaction $\mathcal{T}$, if $D \leq T$ then the optimal deadline assignment is

$$D_j = C_{j} \frac{D}{C_1}.$$  

For this deadline assignment, if we denote by $S_1$ the scheduling points of $\mathcal{T}$ on node 1, we have:

$$\alpha(S_1) = \frac{C_1}{D}.$$  

**Proof.** Since $D \leq T$, two consecutive activations of the transaction $\mathcal{T}$ do not overlap in time. The scheduling point $(D; C_1)$ belongs to $S_1$, since $D$ is the shortest interval length that can accommodate a demand of $C_1$ time units. In fact when $t_0 = 0$ and $t_1 = D$ we have $\text{dbf}(t_0, t_1) = C_1$. The other pairs $(t_0, t_1)$ that can make the demand equal to $C_1$ are $(\phi_j, \phi_j + T)$ which are separated by $T$ that is not less than $D$.

Since $(D; C_1) \in S_1$ then hence $\alpha(S_1) \geq \frac{C_1}{D}$ for any assignment, and the best we can hope is to find an assignment where the inequality holds with the equal sign. We now find the unique optimal deadline assignment. If it exists $D_1, \ldots, D_n$ such that $\alpha(S_1) = \frac{C_1}{D}$, we have:

$$\forall j = 1, \ldots, n \quad \frac{C_1}{D} = \alpha(S_1) \geq \frac{C_j}{D_j} \Rightarrow \frac{D_j}{D} \geq \frac{C_j}{C_1}$$

because $(D_j; C_j) \in S_1$. However none of previous $n$ inequalities can hold strictly, otherwise by adding all of them we would find the false assertion $1 > 1$. Hence the unique deadline assignment that minimizes $\alpha(S_1)$ is

$$D_j = C_{j} \frac{D}{C_1}.$$  

as required. \(\square\)
4.2. SINGLE NODE

We now investigate the deadline assignment problem when the end-to-end deadline $D$ is larger than the transaction period $T$.

First we have the following result asserting that postponing the end-to-end deadline cannot worsen the demand bound function.

**Lemma 12.** Let $T$ be a transaction, with end-to-end deadline $D$ and intermediate deadlines $\{D_j\}_{j=1}^n$ such that

\[
\alpha(S_1) = U_1.
\]

Then, for any end-to-end deadline $D' \geq D$, the following deadline assignment

\[
\begin{align*}
D'_j &= D_j & j &= 1, \ldots, n-1 \\
D'_n &= D_n + D' - D
\end{align*}
\]

also has

\[
\alpha(S'_1) = U_1.
\]

**Proof.** Since in $T'$ the absolute deadline of the last task is postponed and all the other absolute deadlines and activations are unchanged, we have:

\[
dbf_k(T', t) \leq dbf_k(T, t) \implies \alpha(S'_1) \leq \alpha(S_1)
\]

If we recall Lemma 11, we have:

\[
U_1 = \alpha(S_1) \geq \alpha(S'_1) \geq U_1
\]

which gives us $\alpha(S'_1) = U_1$ as required.

Finally the following Corollary gives us a deadline assignment with $\alpha(S_1) = U_1$.

**Corollary 3.** Given a transaction $T$ with $D \geq T$, if we assign the intermediate deadlines as

\[
\begin{align*}
D_j &= \frac{C_j}{U_1} \\
D_n &= \frac{C_n}{U_1} + D - T
\end{align*}
\]

and we denote by $S_1$ the scheduling points of $T$, then $\alpha(S_1) = U_1$.

**Proof.** It follows from Theorem 3 and Lemma 12.

Corollary 3 asserts that in the arbitrary deadline case we can always find a deadline assignment such that $\alpha(S_1) = U_1$. However it does not say if this assignment is unique or not. Hence, there may be space for increasing the horizontal displacement $\Delta_1$ without increasing the slope $\alpha_1$.

Unfortunately, in this case we are unable to find a compact and elegant result to assign the optimal deadlines. To confirm this difficulty we show an example where the result obtained by maximizing $\Delta(S_1)$ is counterintuitive. Consider a (very simple) transaction $T$ with period $T = 4$ and end-to-end deadline $D = 8$, composed by two tasks whose computation times are $C_1 = C_2 = 1$. Then the utilization is $U_1 = \frac{1}{2}$.

If we assign the deadlines by the NORM or by the PURE algorithms (see Equations 1.1 and 1.2) and we denote by $S_1$ the scheduling points of the resulting dbf, we find that
ASSIGN DEADLINES

\[ D_1 = D_2 = 4 \]
\[ D_1 = 2, D_2 = 6 \]
\[ D_1 = 3, D_2 = 5 \]

Figure 4.3. Example of dbf, in the arbitrary deadline case.

\[ \Delta(S_1) = 0, \] as illustrated also in Figure 4.3. Another reasonable attempt is to assign deadlines according to Corollary 3 that guarantees at least \( \Delta(S_1) \geq 0 \). Hence if we set \( D_1 = 2 \) and \( D_2 = 6 \) as indicated in the hypothesis of the corollary, and we plot the resulting dbf, we realize that \( \Delta(S_1) = 0 \) for this assignment as well.

However if we assign \( D_1 = 3 \) and \( D_2 = 5 \) (the corresponding dbf is plot in thick black in the figure), we get \( \Delta(S_1) = 1 \), which is a better deadline assignment than the previous ones.

4.3. Multi node

In this Section we assume again that the tasks of the transaction are mapped onto \( p \) different computational nodes. A node can be either a physical processor or a virtual processor that is characterized by its supply function [APF02, LB05, SL03].

In Section 4.2 we showed that, in the single node case, it is possible to find a deadline assignment such that \( \alpha(S_1) = U_1 \) if and only if \( D \geq T \). Unfortunately this result is not valid in the multi-node case.

We now prove that, if the end-to-end deadline is larger than or equal to a value \( D^* \), then it is possible to find a deadline assignment such that for every node \( k \), \( \alpha_k = U_k \). However, this result is only sufficient: if \( D < D^* \), it may still be possible to find such a deadline assignment.

We prove the result in two steps: we first study the case when the transaction \( T \) can use 100% of the computation time, even if the Utilization of the transaction on a specific CPU is less than one, i.e. we allow \( \alpha_k = 1 \). Later we will remove this limitation, giving an \( o(n) \) algorithm that assigns intermediate deadlines in order to guarantee a slope of related dbf \( \alpha_k \in [U_k, 1] \). Then, from these intermediate deadlines, it is easy to compute the end-to-end deadline.

Please note that, if the real end-to-end deadline is greater than \( D^* \), the result of this algorithm can be directly used to derive a deadline assignment, or as a starting point for the optimization procedure that try to maximize the \( \Delta_k \) factors.
4.3.1. Initial assignment. We start by proving the following theorem.

**Theorem 4**. Let $T$ be a distributed transaction on $p$ nodes. Let $n_k$ be number of
tasks in $T_k$ mapped on node $k$ and $U_k$ the utilization of $T_k$.

Furthermore, let $s_k : 1, \ldots, n_k \rightarrow 1, \ldots, n$ be a mapping that sorts the tasks in $T_k$ by
increasing computation time. More formally:

\[
\forall \ell, j = 1, \ldots, n_k \quad \tau_{s_k(\ell)}, \tau_{s_k(j)} \in T_k
\]

\[
\ell \leq j \iff C_{s_k(\ell)} \leq C_{s_k(j)}.
\]

Then, if the deadlines are assigned according to the following rule:

\[
\forall k, \forall j = 1, \ldots, n_k \quad D_{s_k(j)} = \sum_{\ell=1}^{j} C_{s_k(\ell)}
\]

and $\forall k \quad U_k \leq 1$, we have

\[
\forall k, \forall t \geq 0 \quad \text{dbf}_k(t) \leq t
\]

**Proof.** Since the dbf$_k$ are step functions, we only need to show that the thesis holds
for all $t$ where the functions change value, that is all intervals of length $t$ that start with
the activation of a task, and end with the deadline of a task.

Let us start by observing that $\forall j, \quad D_j \leq T$. In fact, on every processor $k$, the largest
deadline, associated by construction to the task with the largest computation time, is

\[
D_{s_k(n_k)} = \sum_{\ell=1}^{n_k} C_{s_k(\ell)} = U_k T \leq T.
\]

Now, consider processor $k$. We split the proof in the two cases: $t \leq T$ and $t > T$.

**Case A**, $t \leq T$. We have two sub-cases. The interval starts and ends with the
activation and the deadline of the same task, or it starts and ends with the activation
and the deadline of different tasks. In both cases, the interval can contain zero or one
instance of tasks with deadline less or equal than $t$. Let $\tau_{s_k(j)}$ be the task with the largest
deadline among those. By construction only those tasks $\tau_{s_k(i)}$ that are before $\tau_{s_k(j)}$ in
the $s_k()$ mapping have $D_{s_k(i)} < D_{s_k(j)} \leq t$.

Therefore,

\[
\text{dbf}_k(t) = \text{dbf}_k(D_{s_k(j)}) \leq \sum_{i=1}^{j} C_{s_k(i)} = D_{s_k(j)} \leq t.
\]

**Case B**, $t > T$. In this case, we are sure that the interval contains at least one instance
of all tasks allocated to processor $k$. Let $\tau_l$ be the task with the largest deadline in the
interval. Clearly, $\ell = s_k(n_k)$. Let $q$ be the number of instances of $\tau_l$ in the interval:

\[
q = \left\lfloor \frac{t - D_l}{T} \right\rfloor + 1.
\]
Finally, let \( \tau_{s_k(j)} \) be the task with the largest deadline less than or equal to \( t - qT \), if it exists. The maximum total amount of computation time in any interval of length \( t \) is:

\[
C = q \sum_{i=1}^{n_k} C_{s_k(i)} + \sum_{i=1}^{j} C_{s_k(i)} \\
\leq q U_k T + D_{s_k(j)} \\
\leq q U_k T + t - qT \leq t.
\]

Notice that, if there is no task with deadline less than or equal to \( t - qT \), the second summation is empty. However, \( q U_k T \leq t \), thus the theorem holds. \( \square \)

From Theorem 4, it is possible to compute the end-to-end deadline as follows:

\[
D = \sum_{k=1}^{p} \sum_{i=1}^{n_k} D_{s(i)} = \sum_{k=1}^{p} \sum_{i=1}^{n_k} \sum_{j=1}^{i} C_{s(j)}
\]

Now, we analyze the special case in which all tasks on processor \( k \) have the same computation time \( U_k n_k T \). In this case, the computation time of task \( \tau_i \), allocated on processor \( k \) is \( C_i = \). Since every task has the same computation time, the order is not relevant. So, let us consider one of possible orders \( s_k(\cdot) \). Then we can set:

\[
\forall i = 1 \ldots n_k, \quad D_{s_k(i)} = \sum_{j=1}^{i} C_{s_k(j)} = i \frac{U_k}{n_k} T
\]

It follows that:

\[
D^* = \sum_{k=1}^{p} \sum_{i=1}^{n_k} D_{s_k(i)} = \sum_{k=1}^{p} \sum_{i=1}^{n_k} \left( i \frac{U_k}{n_k} T \right)
= \sum_{k=1}^{p} n_k \frac{(n_k + 1) U_k}{2} \frac{T}{n_k} = T \sum_{k=1}^{p} \frac{U_k (n_k + 1)}{2}
\]

Notice that this \( D^* \) is the biggest end to end deadline that we can find using Theorem 4 varying only computation times, i.e. without change the number of tasks, nodes and the mapping between tasks and nodes.

Since this method does not care about offsets among tasks, works for both periodic and sporadic transactions.

4.3.2. Optimization problem. The proposed deadline assignment (Eq. 4.12) is built such that the slopes \( \alpha_k \) on each node are all equal to 1. In fact, on all nodes, the set of scheduling points \( S_k \) contains \( (C_{s_k(1)}; C_{s_k(1)}) \) which constrains \( \alpha_k \) to be equal to 1. However the asymptotic slope is clearly \( U_k \). For long intervals the df starts to have a periodic pattern of steps, and every \( T \) it can increase only of \( \sum_i C_{s_k(i)} \), with an asymptotic slope of \( U_k \).

However, we can reduce the slope \( \alpha_k \) on the \( k \)-th node by multiplying the deadline of every task in \( T_k \) by an appropriate constant \( \xi_k \). For example, if we choose \( \xi_k = \frac{1}{U_k} \)
the deadlines becomes assigned according to

\[(4.16) \quad \forall i = 1, \ldots, n_k, \quad D_{s_k(i)} = \xi_k \sum_{j=1}^{i} C_{s_k(j)} = \frac{1}{U_k} \sum_{j=1}^{i} C_{s_k(j)}\]

and we obtain a dbf with slope equal to \(U_k\). Notice that, after the multiplication, the largest deadline on any processor is always equal to the period:

\[D_{s_k(n_k)} = \frac{1}{U_k} \sum_{j=1}^{n_k} C_{s_k(j)} = \frac{1}{U_k} U_k T = T.\]

This means that is possible to apply the same proof to show that the dbf is always less than or equal to \(U_k\) and the end-to-end deadline for the special case of Eq. 4.15 becomes:

\[(4.17) \quad D^* = T \sum_k n_k + 1 \geq T(n + p)\]

where we remind that \(n\) denotes the number of tasks and \(p\) the number of nodes.

Therefore, by assigning the deadlines as indicated in Eq. 4.16, the end-to-end deadline \(D\) must be at least \(p\) times the transaction period \(T\). This result is confirmed by the results of the simulations described in Section 4.5.1.

However, if we want to set the slope \(\alpha_k\) to a value higher than the utilization \(U_k\), we must multiply all deadlines by \(\xi_k = \frac{1}{\alpha_k}\). Also, if we want to introduce an horizontal displacement \(\Delta_k\), one possible method is to sum term \(\Delta_k\) to all deadlines of the tasks on processor \(k\).

**Corollary 4.** If intermediate deadlines are assigned as follows:

\[(4.18) \quad \forall k, \forall i = 1, \ldots, n_k, \quad D_{s_k(i)} = \frac{1}{\alpha_k} \sum_{j=1}^{i} C_{s_k(j)} + \Delta_k\]

with \(U_k \leq \alpha_k \leq 1\), and \(\Delta_k \geq 0\), then:

\[\forall k, \forall t, \quad \text{dbf}_k(t) \leq (\alpha_k(t - \Delta_k))_0\]

**Proof.** The proof is a straightforward extension of the proof of Theorem 4.

We simply observe that scaling the deadlines by a common factor \(\frac{1}{\alpha_k}\) and adding them a common value \(\Delta_k\) does not change the relation \(\ell \leq j \iff D_{s_k(\ell)} \leq D_{s_k(j)}\). Then it is possible to follow the same steps in the proof of Theorem 4 to check that for all possible intervals \(\text{dbf}_k(t) \leq (\alpha_k(t - \Delta_k))_0\).

Case 0, \(t < D_{s_k(1)}\). The smallest deadline is greater than \(\Delta_k\), then:

\[\text{dbf}_k(t) = 0 \leq (\alpha_k(t - \Delta_k))_0.\]

Case A, \(D_{s_k(1)} \leq t \leq T + \Delta_k\). We obtain

\[\text{dbf}_k(t) = \text{dbf}_k(D_{s_k(j)}) \leq \sum_{i=1}^{j} C_{s_k(i)} = \alpha_k(D_{s_k(j)} - \Delta_k) \leq (\alpha_k(t - \Delta_k))_0.\]

Note that \(\alpha_k(t - \Delta_k) > 0\) is always true because we are considering \(t > \Delta_k\).
4. ASSIGN DEADLINES

Case B, \( t > T + \Delta_k \). We obtain:

\[
\text{dbf}_k(t) \leq q \sum_{i=1}^{n_k} C_{s_k(i)} + \sum_{i=1}^{j} C_{s_k(i)} \\
\leq q U_k T + \alpha_k(D_{s_k(j)} - \Delta_k) \\
\leq \alpha_k(q T + D_{s_k(j)} - \Delta_k) \leq (\alpha_k(t - \Delta_k))_0.
\]

Also in this case \( \alpha_k(t - \Delta_k) > 0 \) is always true.

From Equation 4.18, it is clear that we can also reverse the process. Instead of computing the end-to-end deadline \( D \) that guarantees a slope \( \alpha_k \) on all processors, we can find a set of multiplication constants \( \alpha_k \) and of horizontal displacements \( \Delta_k \) such that the sum of all deadlines does not exceed the desired end-to-end deadline, and a certain objective function \( h(\cdot) \) is minimized. The problem in its generality can be expressed as follows:

\[
\begin{align*}
\minimize & \quad h(\alpha_1, \ldots, \alpha_p, \Delta_1, \ldots, \Delta_p) \\
\text{subject to} & \quad \\
& \quad \sum_k \sum_i D_{s_k(i)} \leq D \\
& \quad U_k \leq \alpha_k \leq 1 \\
& \quad \Delta_k \geq 0.
\end{align*}
\]

4.3.3. Solution of the optimization problem. Until now we have proposed a general formulation of the optimization problem. In the general case, this problem can be solved numerically.

In this Section, we choose one relevant optimization function for which we provide an analytical solution to the problem of Eq. 4.19:

\[
h(\alpha_1, \ldots, \alpha_k) = \max_k \alpha_k U_k.
\]

In this case, we are interested in minimizing the maximum ratio between the slopes \( \alpha_k \) of our dbfs and the minimum utilization \( U_k \) that are achievable on each node \( k \). We also assume that all \( \Delta_k = 0 \), thus we do not consider the horizontal displacement of the dbfs.

This function is relevant for our goals of component based analysis: in fact, we would like to set the intermediate deadlines so that the worst-case utilization of each transaction on processor \( k \) be as close as possible to the minimum utilization \( U_k \). The optimal solution tells the designer how much utilization must be allocated on each processor; thus the designer can perform an early allocation of the bandwidth of the processors to each transaction in the system.

The solution of this problem can be computed by the following simple algorithm. By changing variable \( \xi_k = \frac{\alpha_k U_k}{U_k} \), the problem can be rewritten as:

\[
\begin{align*}
\minimize & \quad \max_k \xi_k \\
\text{subject to} & \quad \\
& \quad \sum_k \beta_k \xi_k \leq D \\
& \quad 1 \leq \xi_k \leq \frac{1}{U_k}.
\end{align*}
\]
4.3. MULTI NODE

\[
\begin{array}{|c|c|c|c|c|}
\hline
& D_1 & D_2 & \alpha_1 & \alpha_2 & \xi \\
\hline
\text{NORM} & 6.66 & 12.33 & 0.15 & 0.15 & 3 \\
\text{PURE} & 9.5 & 10.5 & 0.105 & 0.19 & 2.1 \\
\text{ORDER} & 10 & 10 & 0.1 & 0.2 & 2 \\
\hline
\end{array}
\]

**Figure 4.4.** Behavior of ORDER, NORM and PURE on the first example.

where

\[
\beta_k = \frac{1}{U_k} \sum_{i=1}^{n_k} \sum_{j=1}^{i} C_{s_k(j)}.
\]

If we ignore for the moment the last constraint, this problem has solution when all \(\xi_k\) are equal to the same constant \(\xi\). Therefore,

\[
\xi = \sum_k \beta_k \quad \alpha_k = \xi U_k
\]

If for some \(k \alpha_k > 1\), then we can set \(\alpha_k = 1\), and iterate the previous solution to find the other \(\alpha_k\). The number of iterations, in the worst case, is \(p^2\) to compute \(\beta_k\) we need to sum \(n_k\) variables; thus the complexity is \(O(p^2 \text{ max } n_k)\).

The algorithm just described is called ORDER, and it is the one we propose to assign intermediate deadlines to transactions. The value of the optimization function \(h(\cdot)\) is called energy of the solution.

A solution of the more general formulation of Equation 4.19 depends on the choice of \(h(\cdot)\).

**4.3.4. Examples and considerations.** We first consider the simple situation in which each task of a given transaction is allocated on a different node. Consider a transaction consisting of 2 tasks with computation time \(C_1 = 1\) and \(C_2 = 2\), allocated on 2 processors. The transaction period and end-to-end deadline are \(T = 20\) and \(D = 20\), respectively.

We first compute \(\beta_1 = \frac{C_1}{U_1} = T = 20\) and \(\beta_2 = \frac{C_2}{U_2} = T = 20\). Then, we compute

\[
\xi = \frac{\beta_1 + \beta_2}{D} = \frac{2T}{D} = 2.
\]

Then, \(\alpha_1 = \xi U_1 = 0.1\) and \(\alpha_2 = \xi U_2 = 0.2\). Finally, the deadlines are:

\[
D_1 = \frac{C_1}{\alpha_1} = \frac{T}{\xi} = \frac{D}{2} = 10 \quad D_2 = \frac{C_2}{\alpha_2} = 10.
\]

Notice that, in this case, the value of the energy \(\xi\) does not depend on the values of the execution times. In fact, if we multiply all computation times by a constant, \(\xi\) and \(D_i\) do not change.

In the general case of \(n\) tasks on \(n\) processors, it is easy to see that the energy can be computed as \(\xi = \max\{1, \frac{nT}{D}\}\), and the relative deadlines are \(D_i = \frac{D}{n}\).

In figure 4.4, we report the comparison between the results of ORDER, NORM and PURE algorithms. The latter two algorithms are described by Equations 1.1 and 1.2, respectively. Notice that, for both NORM and PURE, the actual values of the energy and of the deadlines depend on the computation times, and the energy is higher than
4. ASSIGN DEADLINES

<table>
<thead>
<tr>
<th></th>
<th>$D_1$</th>
<th>$D_2$</th>
<th>$D_3$</th>
<th>$\alpha_1$</th>
<th>$\alpha_2$</th>
<th>ENERGY</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORM</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>0.27</td>
<td>0.2</td>
<td>2</td>
</tr>
<tr>
<td>PURE</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0.36</td>
<td>0.2</td>
<td>2</td>
</tr>
<tr>
<td>ORDER</td>
<td>3.33</td>
<td>13.33</td>
<td>13.33</td>
<td>0.3</td>
<td>0.15</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Figure 4.5. Behavior of ORDER, NORM and PURE on the second example.

the energy obtained with ORDER. This means that, with the same transaction, the distribution of the utilization among the two processors is more unbalanced.

Now, let us consider a slightly more complex example of 3 tasks on two processors. The three tasks have computation times $C_1 = 1$, $C_2 = 2$, $C_3 = 3$, and are allocated on the first, second and first processor, respectively. The period is $T = 20$ and the end-to-end deadline is $D = 30$.

We start by computing $\beta_1 = C_1 + C_1 + C_3 U_1 = 25$ and $\beta_2 = C_3 U_2 = T = 20$. Then $\xi = \frac{\beta_1 + \beta_2}{D} = \frac{45}{30} = 1.5$. Then, $\alpha_1 = \xi U_1 = 0.3$ and $\alpha_2 = \xi U_2 = 0.15$. Finally, the deadlines are:

$$D_1 = \frac{C_1}{\alpha_1} = 3.33 \quad D_2 = \frac{C_2}{\alpha_2} = 13.33$$
$$D_3 = \frac{C_1 + C_3}{\alpha_1} = 13.33$$

Once again, note that the energy does not depend on the values of the computation times: if we multiply all computations times by a constant $K$, also the utilization is multiplied by the same constant, so $\beta_1$ and $\beta_2$ do not change. Rather, $\xi$ depends on how the computation times are distributed among the two processors. However, such dependency is not very strong, as we will show in Section 4.5.1. The comparison with the other algorithms is shown in figure 4.5.

Notice that once again ORDER achieves a lower energy. This means that, if we use NORM or PURE to assign deadlines, we have to allocate two times the minimum possible utilization on the second processor (0.2 instead of 0.1). If we use ORDER, instead, we only have to assign 50% more utilization on both processors.

Also, by performing an extensive set of experiments (see Section 4.5.1, we have seen that the energy has a weak dependency on the actual values of the computation times and from their distribution on the processors. Thus, by using ORDER, the designer can have an early assessment of the amount of utilization that will be needed on each processor to support a single transaction (i.e. a component in our model).

4.4. Optimization Algorithm

The assignment algorithm presented in the previous Section guarantees that $\alpha_k = U_k$ for any node $k$ if the end-to-end deadline is not smaller than $D^*$, as given by Equation 4.14. However, the value of $D^*$ can be very large. Moreover, even if the end-to-end deadline is large enough, the assignment may not be optimal with respect to the goals we have presented in Section 4.1. Therefore, in the general case it is necessary to find an
algorithm to derive an optimal deadline assignment. To the best of the authors’ knowledge, no efficient polynomial-time algorithm has been proposed to solve the problem of Equation 4.5.

In this research, we use Simulated Annealing [DNS95] to explore the space of possible deadline assignments. Unlike the work in [DNS95], we apply the optimization procedure to a single transaction.

As discussed in Section 4.1, we must balance the deadline assigned to the tasks belonging to different nodes by using an appropriate goal function $h(\cdot)$. If we lower the $dbf$ on one node (for example by increasing the deadline of the tasks allocated on that node), probably the $dbf$ on another node may increase. It is then necessary to balance the contrasting requirements of the different nodes by designing an appropriate $We start by observing that $\alpha_k \geq U_k$, in other words the slope of the $dbf$ can never go below the utilization of the transaction on the corresponding node (see Lemma 11), we denote as $ds_k = \alpha_k - U_k \geq 0$. Then, we need a way to weight the allocation of deadlines among different nodes. In facts, the designer may want to control the optimization algorithm by weighting the allocation of the $\alpha_k$ on the different nodes. For example, the designer may want to push on the optimization on one particular node, and be more relaxed on other nodes. As another example, we may weight the $\alpha_k$ reduction with the total utilization of all transactions on node $k$. We denote as $w_k$ the weight of node $k$, with $0 \leq w_k \leq 1$. Finally, in case $\alpha_k = U_k$, we want to further optimize the $dbf_k$ by maximizing the $\Delta_k$.

From the requirements described above, we decided to use the following function:

$$h(\cdot) = \sum_k h_k$$

where:

$$h_k = \begin{cases} e^{\frac{-K}{w_k} ds_k} & \text{if } ds_k > 0 \\ e^{\frac{-K}{w_k} \Delta_k} & \text{if } ds_k = 0. \end{cases}$$

and $K$ is an appropriately large constant\(^1\). In practice, the algorithm will try to first reduce those terms corresponding to nodes with large weights; only if the slopes of all nodes are close to their target utilization $U_k$, the $\Delta_k$ component starts to have a moderate effect on the goal function $h(\cdot)$.

### 4.4.1. An example.

Consider a transaction consisting of 3 tasks on 2 nodes. The transaction period is 50 and the end-to-end deadline is 90. The other task parameters are reported in figure 4.6: the 2\(^{nd}\), 3\(^{rd}\) and 4\(^{th}\) columns report the computation times, the task allocation, and an assignment of deadlines proportional to computation times, respectively.

By applying Equations 4.14 and 4.16, we obtain $D^* = 120$, which is larger than the associated deadline. Next, we use our optimization algorithm. In the first run, we set equal weights $w_0 = w_1 = 0.5$. The resulting deadlines are reported in the 5\(^{th}\) column. The $dbfs$ for node 1 and node 2 are depicted in Figures 4.11.(a) and 4.11.(b) respectively, with label $opt_1$. As you can see, we obtained a good result on node 1 with $\alpha_2 = U_2 = 0.12$, while for node 0 the slope $\alpha_1 = 0.25$ is still quite high against $U_1 = 0.2$.

\(^1\)In the simulation experiments, we set $K = 5$.\

---

\(\sum\)
4. ASSIGN DEADLINES

<table>
<thead>
<tr>
<th>Task</th>
<th>$C_j$</th>
<th>$x_{i,j}$</th>
<th>$D_j$ (prop)</th>
<th>$D_j$ (opt1)</th>
<th>$D_j$ (opt2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>4</td>
<td>1</td>
<td>22.50</td>
<td>15.82</td>
<td>20.09</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>6</td>
<td>2</td>
<td>33.75</td>
<td>50.45</td>
<td>19.81</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>6</td>
<td>1</td>
<td>33.75</td>
<td>23.72</td>
<td>50.09</td>
</tr>
</tbody>
</table>

**Figure 4.6.** Parameters of the example

Notice that applying Equation 4.16 and multiplying by $\frac{1}{\alpha_k}$ (instead of $\frac{1}{U_k}$) we obtain deadlines equal to 16, 50 and 40, respectively, for a total of $D^* = 106$.

If the designer is not satisfied with this result, he may want to do the opposite: reduce the dbf$_1$, paying the price of increasing the dbf$_2$. Therefore we run again the optimization procedure with weights $w_0 = 0.9$ and $w_1 = 0.2$. This time, the deadlines are reported in the 6th column, while the dbf$_1$ and dbf$_2$ are reported in the Figures 4.11.(c) and 4.11.(d) respectively, with label opt$_2$. This time $\alpha_1$ is equal to $U_1$, whereas $\alpha_2 = 0.3 > U_2$. Applying Equation 4.16 and multiplying by $\frac{1}{\alpha_k}$ we obtain deadlines equal to 20, 20 and 50 respectively, for a total of $D^* = 90$.

4.5. Simulations

4.5.1. ORDER algorithm. In this Section, we compare algorithm ORDER against algorithms NORM and PURE [DNS94]. We randomly generated all transactions with constant period equal to 1000 units and constant total utilization equal to 10% (that is, the sum of the computation times of all tasks is 100, regardless of the processor they are allocated on). In fact, as explained in Section 4.3.4, the results for algorithm ORDER do not depend on the actual value of the utilization, but only on the distribution of the computation times among the tasks. Also for algorithm NORM there is a very weak dependency on the value of the utilization. The computation time of each task is randomly generated according to method UUniFast described in [BB05], so that their sum is equal to 100.

In the experiments, we varied the number of processors $p$ from 2 to 4. For each value of $p$, we generated transactions with variable number of tasks between $p$ and $3p$. We generated the transaction so that no consecutive tasks were allocated on the same processor. Finally, we varied the ratio between end-to-end deadline and period from 1 to $3p$. For each combination of these three parameters, we randomly generated 30 transactions, and we applied all three algorithms to each transaction, computing the resulting energy $h(\cdot)$. In the Figures, we report for each point the average energy and the 95% confidence interval.

In Figure 4.7 we show the value of the energy for algorithm ORDER by varying the number of tasks in the transaction. On the $x$ axis we report the ratio between end-to-end deadline and period. Notice that the energy grows proportionally to the number of tasks. Also, consider that for 2 tasks the energy reaches its minimum value 1 at ratio equal to 2, and increasingly more for longer transactions. This confirms the intuition that partitioning the transaction into a smaller number of tasks gives better results in terms of utilization, and hence of schedulability.
Finally, notice that the confidence interval is very small, showing that in reality the results are almost independent of the random distribution of computation times among the tasks.
In Figure 4.8, we show the comparison between ORDER and PURE for the case of 2 processors and 6 tasks. Since we imposed that two consecutive tasks cannot be allocated on the same processor, in this case the topology is unique: the first, the third and the fifth task are allocated on one processor, while the second, the fourth and the sixth tasks are allocated on the other. In this case ORDER always perform better than PURE. Also, notice that PURE has a strange behavior around values 1.5, 3 and 4.5 of the deadline-period ratio. This is due to the fact that algorithm PURE does not take into account the number of tasks allocated on the same processor. If the three tasks on one processor do not interfere much (for example, when the deadline-period ratio is around 2 their “slices” are not overlapping), then the deadline assignment made by PURE is a “good” assignment. Instead, when the slices overlap (as in the case of $D_T = 3$, then the demand increases. Algorithm ORDER has not such a problem as it directly accounts for tasks on the same processor.

The results are very similar for different number of processors. We only report similar plots for the case of 4 processors and 8 tasks in Figure 4.9.

We did not report the values for algorithm NORM in the figures. In fact, we were not able to derive meaningful statistical values from this algorithm with our experimental setting. While the average value of the energy is always much bigger than PURE and ORDER, the statistical confidence interval was very large. The reason is due to the fact that the values of $\alpha_k$ obtained with the deadline assignment made by NORM strongly depend on the value of the tasks computation times, which are randomly generated.

One important consideration can be made. The results of the experiments show that the ratios between $\alpha_k$ and $U_k$ that we obtain with algorithms ORDER and PURE do not depend on the actual values of the computation times, and very little on their distribution. Hence, the designer of a distributed embedded system can use the graphs of Figures 4.7-4.9 to estimate how much extra bandwidth is needed on each processor for the desired end-to-end deadline/period ratio and the selected number of tasks. This result can be very useful in the early stages of the development, when the designer has to understand how to partition the work among the tasks, and how many tasks must be used.

Finally, we have run experiments to compare the energy of the solutions provided by ORDER against solutions obtained by Simulated Annealing. The Simulated Annealing algorithm randomly tries different deadline assignments to select the best one. In Figure 4.10 we show the results on 2 processors and each point is the average of 30 experiments with various transaction lengths. We can see that the difference between the two algorithms is very small, despite the great different in complexity and execution time.

4.5.2. Full optimization. We applied the optimization algorithm to sets of randomly generated transactions. The goal is to evaluate the improvement in schedulability with respect to a standard deadline assignment. In the literature, experiments are done by using a proportional deadline assignment: each task is assigned a relative deadline proportional to its computation time.

We generated a set of $m$ transactions, each one consisting of a number of tasks between $n_{\text{min}}$ and $n_{\text{max}}$. The sum of all utilizations on all processor was set equal to $U_{\text{tot}}$. Transaction periods are randomly selected $[100, 500]$. The ratio between end-to-end deadline and period is randomly selected between $R_{\text{min}}$ and $R_{\text{max}}$. Tasks are randomly
allocated on $p$ processors. For each value of $U_{\text{tot}}$ we generated 500 such transaction sets. Initially, tasks are assigned deadlines proportional to their computation times.
Figure 4.11. Example: optimizing with different weights

For each set we perform three different schedulability tests. The first one is the classical Palencia-Gonzalez test [PH03], denoted as WCDO, which is an extension of the holistic analysis for EDF. The algorithm iteratively computes the response times for each task, and updates the starting jitters to the finishing times of the preceding tasks. The iteration is stopped when it converges to a fixed point, or when a deadline is missed.

The second test, proposed by Pellizzoni and Lipari [PL07], is denoted as MDO. It is an extension of WCDO which uses the offsets instead of the jitters. Again, response times are computed at each iteration by using the results of [PL05], but instead of updating the jitters, the offsets are updated instead. MDO is superior to WCDO with respect to number of accepted transaction sets.

Finally, the third test, denoted as DBF, is based on the demand bound functions computation. This one is the less effective as it assumes that a job is not allowed to start until its activation offset $\phi_i$, which is a pessimistic assumption.

After running the three tests, if the set is not schedulable with one of the tests, we execute the optimization algorithm of Section 4.4 on each transaction separately. Each node is assigned a weight $w_k$ equal to its total load: in this way, we push more on the
optimization on those node that are more loaded. After the optimization, we run the
three tests again to see if the set is schedulable with the new deadline assignment.

In Figure 4.12 we show the results of the first experiment, where $m = 6$, $n_{\text{min}} = 2,$
$n_{\text{max}} = 5$, $R_{\text{min}} = 0.5$, $R_{\text{max}} = 1.5$, $p = 2$. As you can see, even if our optimization algo-
rithm performs a local optimization on each transaction (which is not globally optimal),
we still obtain some improvement. Clearly, the DBF test is the least effective because
of the inherent pessimism in computing the dbf. However, notice that the optimization
algorithm brings benefits not only to the DBF test, but also to the other holistic
tests. This is a clear indication of the benefit of assigning according to the optimization
algorithm of Section 4.4.

It is important to notice that the goal of our optimization procedure is to reduce the
dbf of each transaction on each node. In computing the dbf, we are forced to do the
pessimistic assumption that tasks are activated at the deadline of the previous task in
the transaction. If we assume instead that tasks are activated at the finishing time of
the previous task in the transaction, the corresponding dbf becomes much lower. This
is why the DBF is the least effective of all three schedulability tests.

The second experiment has been performed with the following settings: $m = 10,$
$n_{\text{min}} = 5$, $n_{\text{max}} = 10$, $R_{\text{min}} = 1.5$, $R_{\text{max}} = 3$, $p = 4$. The results are plotted in Figure
4.13. In this case, the improvement is very small. In our opinion, it depends on the fact
that the problem is already over-constrained. In facts, the result of Equation 4.17 tell us
that the end-to-end deadline is somewhat proportional to the sum of the number of tasks
and of the number of nodes. In this setting, we have up to 10 tasks on 4 processors, but
the end-to-end deadline can be as low as 1.5 times the period. In this case, any deadline
assignment will probably have a bad effect on the values of $U_k$. Notice that the MDO
test is strikingly successful in this case.

We have performed many more experiments that cannot show due to space con-
straints, but the behavior is very similar to the ones that we have reported. Also note
that, since our approach is locally optimal, but not globally, it may happen that a trans-
action set that is schedulable when deadlines are assigned proportional to the task’s
computation times, becomes non schedulable after our optimization algorithm. We be-
lieve that this is part of the game: a local optimization procedure cannot be globally
optimal. However, the advantages of isolating the design of each transaction are impor-
tant and useful.
Figure 4.12. First simulation experiment.

Figure 4.13. Second simulation experiment.
CHAPTER 5

Conclusions

In this thesis we addressed the problem of analyzing the schedulability of sporadic transactions on a distributed system scheduled by EDF, and how to support our methodology at run-time. Our methodology is based on computing the Processor Demand Criterion [But04], the demand bound function [BHR90] and slicing technique [DNS94]. We proposed an algorithm to compute off-line the sporadic dbf on each node. We also proposed the IDSP protocol that assigns appropriate deadlines to jobs using only local information, and guaranteeing that the dbf computed on-line never exceeds the dbf computed off-line.

On the same bases, we investigated the problem of assigning intermediate deadlines to real-time transactions. Instead of a global optimization method, we presented a possible metric for local optimization of a single transaction and some interesting analytical results that give more insight in the problem.

Following the idea of local optimization, we propose a simple algorithm to assign intermediate deadlines that performs better than similar algorithms presented in the literature, and can be used effectively for component-based distributed systems. Thanks to its low complexity, it can be used as it is or as the first step of an optimization process.

In the last part of this thesis we propose an algorithm to perform local optimization and compare it with other (global) optimization algorithms described in literature.
Bibliography


