Scheduling Imprecise Tasks in Real-Time Distributed Systems

Rômulo Silva de Oliveira, Joni da Silva Fraga and Jean-Marie Farines
Universidade Federal de Santa Catarina
LCMI-DAS-UFSC, Caixa Postal 476
Florianópolis-SC, 88040-900, Brasil
(romulo, fraga, farines)@lcmi.ufsc.br

Abstract

The Imprecise Computation technique has been proposed as an approach to the construction of real-time systems that are able to provide both guarantee and flexibility. This paper analyzes the utilization of Imprecise Computation in the scheduling of distributed real-time applications. First we discuss the main problems associated with that goal, and we define a task model suitable to distributed environments. Then, this paper presents an approach to the scheduling of distributed imprecise tasks. Finally, we describe specific scheduling algorithms to be used within the proposed approach.

1. Introduction

A basic problem found in the construction of real-time distributed systems is the allocation and scheduling of tasks on the available resources. In many ways, real-time systems require the simultaneous achievement of two fundamental goals [4]: to guarantee that the results will be produced at the specified time and to provide flexibility into the system so it can adapt itself to a dynamic environment and increase its utility.

There are scheduling solutions that assume a fixed group of tasks to be executed. These solutions reserve resources for the worst-case scenario regarding the task load. Worst-case scenario in this context means that all tasks will arrive, simultaneously, with maximum frequency, presenting their maximum execution time, characterizing the worst possible task load to the processors at a time instant.

Solutions based on worst-case as in [25] are able to guarantee that, in any task arrival scenario, all tasks will meet their time constraints. However, this approach results in systems with lack of flexibility and under-utilization of resources. There are also scheduling solutions that do not provide an off-line guarantee for deadlines. Although resources are used fully and the resulting system is quite flexible, the lack of a previous guarantee for its temporal behavior makes this solution unfeasible for the class of applications with critical temporal requirements [20].

The Imprecise Computation technique [16] divides each application task into a mandatory part and an optional part. The mandatory part is able to generate a minimum quality result that is necessary to maintain the system in a safe state. The optional part refines this result until it reaches maximum quality. A mandatory part is said to generate an imprecise result, while the result of the mandatory plus optional parts is said to be precise. A task is called imprecise if it is possible to decompose it into mandatory and optional parts. This technique tries to achieve both fundamental objectives mentioned before: flexibility and off-line guarantee.

Few works exist in the literature about Imprecise Computation in distributed systems. Most of those works ([10], [12], [26]) analyze specific problems and particular cases. It does not exist in the literature a wide study about "how to solve the scheduling problem when real-time distributed systems are built by using the Imprecise Computation concept". The general goal of this paper is to show how real-time applications that use Imprecise Computation can be scheduled in distributed systems.

Section 2 presents the task model assumed in this work. An approach for the scheduling problem is presented in section 3. Section 4 of this paper discusses the problem of scheduling imprecise tasks in real-time distributed systems and describes suitable algorithms. Concluding comments appear in section 5.

2. Task model

We consider the execution of an application in a distributed system formed by a set $\mathbf{L}$ of $\boldsymbol{l}$ processors. Processors are connected through some communication network where a message transfer between two different processors is characterized by a maximum latency Δ. It is supposed that an auxiliary processor will execute the communication protocol which does not compete with application tasks for processor time. The delay associated with a message transfer between two tasks on the same processor is not considered in the scheduling analysis.

A distributed application in this model is expressed as a set of activities. Each activity is represented by a directed aciclic graph related to a group of tasks with precedence relationships. Each node in the graph represents a task and each arc, a precedence relation. In this model, a direct precedence relation corresponds to an explicit message transfer (dynamic release of tasks). An application is a set **A** of ***m*** activities. Each activity can be periodic or sporadic. For each activity A_j, $1 \leq j \leq m$, belonging to the set **A,** an infinite sequence of activations can occur. A periodic activity A_j is characterized by a period P_j and a sporadic activity A_j by a minimum time interval P_j between activations.

The whole application is associated to a set **T** of ***n*** tasks, that is, the set that includes all tasks that belong to all activities of set **A**. Mutual exclusion relationships do not exist among tasks. Any task can be preempted at any time. Each task T_i is characterized by a maximum release jitter J_i, a global priority $\rho(T_i)$ and a deadline D_i relative to the beginning of its activity. For each task T_i, $1 \leq i \leq n$, if $T_i \in A_j$ then we have $D_i \leq P_j$. Each task receives a unique priority in the application. Task T_i can not begin its execution until receiving a message from each one of its direct predecessors.

We assume the use of multiple versions to compose each imprecise task T_i with a mandatory part and an optional part. Worst-case execution time of both parts are known off-line and denoted by M_i and O_i, respectively.

We also suppose that each task T_i is associated with a nominal value V_i, representing the usefulness of task T_i. However, it does not take into account dependence between tasks. This paper introduces the concept of effective value. Each release $T_{i,k}$ of task T_i has an effective value $V_{i,k}$ defined at run-time. The effective value is calculated from the nominal value, considering task dependences. The value added by $T_{i,k}$ to the system will be zero in the case of an imprecise execution or $V_{i,k}$ when its optional part is executed. The scheduling of optional parts tries to maximize the total system value.

Intra-task dependence between releases $T_{i,k}$ and $T_{i,k+1}$, for a given task T_i is modeled by assuming that an imprecise execution of $T_{i,k}$ will increase the effective value of a precise execution of release $T_{i,k+1}$. The effective value of release $T_{i,k+1}$, denoted by $V_{i,k+1}$, is:

V_i when the execution of $T_{i,k}$ is precise;

$V_i+(\alpha_i \,.\, V_{i,k})$ when the execution of $T_{i,k}$ is imprecise;

where α_i is called the recovery rate of task T_i. Although the concept of intra-task dependence was already implicit in [6], the formulation presented in this paper, based on recovery rate, is different.

Inter-task dependence is associated to the fact that input data with better quality reduces the execution time of some tasks. The inter-task dependence is modeled assuming that a precise execution of a predecessor $T_{j,k}$ will reduce the worst-case execution time of sucessor $T_{i,k}$ mandatory and optional parts. The worst-case execution time of $T_{i,k}$ will be:

$M_i + O_i$ when the execution of $T_{j,k}$ is imprecise;

$\beta_{j,i} \,.\, M_i + \gamma_{j,i} \,.\, O_i$ when the execution of $T_{j,k}$ is precise;

where $\beta_{j,i}$ and $\gamma_{j,i}$, $0< \beta_{j,i} \leq 1$, $0< \gamma_{j,i} \leq 1$, are reduction factors linking T_j to T_i. This formulation for inter-task dependence is an adaptation of the one used in [9].

It is important to observe that none of the two types of dependence defined here is able to increase the mandatory load in the system. Intra-task dependence may increase the effective value of some optional parts. Inter-task dependence may reduce the worst-case execution time of some tasks. Anyway, the mandatory load of the system does not increase. This property is important because, otherwise, any off-line schedulability test would lose its validity when the system mandatory load increased.

3. Scheduling approach

This section describes our approach to schedule imprecise tasks in distributed systems. Figure 1 illustrates the approach composed by four algorithms. Algorithm 1 is responsible for the allocation of tasks on processors; algorithm 2 is able to verify that mandatory parts will always be concluded before their respective deadline; algorithm 3 is used to select optional parts that, at a given instant, should be considered for execution; algorithm 4 determines if a given optional part can be executed without jeopardizing the execution of mandatory parts.

The first algorithm breaks the initial task set into ***l*** subsets (***l*** processors). Since a successful allocation needs all mandatory parts being guaranteed, the allocation algorithm must use algorithm 2 to verify this property. The allocation algorithm also tries to obtain load balance to increase the odds of an optional part being executed.

The second algorithm must be able to verify that no deadline will be missed when only mandatory parts are executed. Since fixed priorities will be used, it is necessary to determine a priority assignment policy and an appropriate schedulability test. The complexity of the problem increases due to tasks on different processors that communicate. A processor schedule may be influenced by delays due to messages from other processors.

The third algorithm evaluates at run-time if a given optional part should be considered for execution. The goal is to implement an admission policy that discards optional parts whose importance is very low. Obviously, the minimum level of importance for an optional part to be admitted depends on the processor load at each instant.

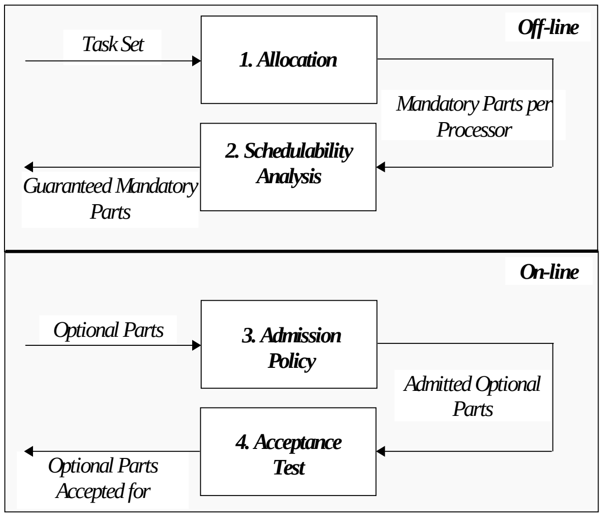

Figure 1. General approach.

The fourth algorithm determines if a given optional part, previously admitted by algorithm 3, can be accepted for execution. It takes into account all assumptions of algorithm 2. An optional part can be accepted if it does not alter the conditions supposed by algorithm 2 in such a way to invalidate the off-line schedulability test.

The general goal of this work is to show how real-time applications based on Imprecise Computation concepts can be scheduled in the case of distributed systems. Once we defined the approach to be used, it remains to solve the four scheduling problems.

4. Scheduling algorithms

This section explains how the four problems described in the previous section can be solved. We do not present here a complete description of the algorithms due to space limitation. A more rigorous formalization, as well as quantitative analysis, can be found in [17], [18] and [19].

4.1 Allocation

Real-time scheduling in a distributed context is generally solved in two stages. In the first stage tasks are allocated to processors. The second stage corresponds to the local scheduling of each individual processor. The local scheduling considers as load all tasks allocated in the first stage. Migration of tasks is usually not allowed in real-time applications due to the cost of such operation, both in terms of resources and time. The fact that each task is allocated permanently to a processor increases the importance of an appropriate allocation.

The primary goal of allocation is to guarantee that all tasks can always conclude its respective mandatory parts before theirs deadlines. In order to accomplish that, an allocation algorithm proposes several solutions. With regard to the primary objective, any allocation solution that allows an off-line guarantee for the mandatory part of every imprecise task is considered satisfactory.

Allocation algorithms should also try, as secondary objective, to increase the chances of all optional parts to execute. Consider two allocation solutions X and Y, both equally good from the point of view of the primary objective. That is, mandatory parts can be guaranteed so much in X as in Y. It is possible that allocation X is better balanced with regard to allocation Y, so that the execution of optional parts will be better in X than in Y. Therefore, from the point of view of the secondary objective, allocation X is better than allocation Y.

The task allocation problem in distributed systems is very hard. Most cases require a prohibitive processing cost in order to obtain an optimal solution. Therefore, approximate algorithms are normally used. In this work we use a well-known method called simulated annealing ([13], [23]).

The primary objective of allocation consists to guarantee task deadlines when we consider only mandatory parts. The secondary objective will be to provide a reasonable load balance to avoid localized overloads and to increase the odds of optional parts to execute. Simulated annealing requires the definition of an energy function **E** that evaluates the quality of a given solution. A *low energy* is associated with a good solution. We define this energy function as:

$$E = K_e \, . \, E_e + K_b \, . \, E_b$$

where E_e and E_b represent the energy associated with task schedulability and load balance, respectively. Values K_e and K_b are so that the aspect *schedulability* receives greater importance than the aspect *load balance*. A solution with good load balance but not schedulable will always have a greater energy than a solution with severe load balance but with all deadlines guaranteed.

Like in [23], value E_e measures by how much application deadlines may be lost. By assuming that the schedulability test calculates the maximum response time R_i for each task T_i with deadline D_i, energy is given by:

$$E_e = \sum_{i=1}^{n} Max(0, R_i - D_i)$$

Value E_b measures how well the average system workload is balanced. The average utilization U_i is computed for each processor L_i. The average utilization of a processor is the sum of the utilization of all tasks allocated to that processor. The utilization of a task if defined to be its average execution time C_i divided by its average repetition rate I_i. The average repetition rate corresponds to the period of a periodic task and to the average time between activation of a sporadic task. Energy E_b is given by:

$$E_b = \sum_{j=1}^{l} \left| U - U_j \right|,$$

where $\boldsymbol{l}$ is the number of processors and $\boldsymbol{U}$ represents the average system utilization given by:

$$U = \left(\sum_{i=1}^{n} \frac{C_i}{I_i} \right) \Big/ l .$$

A simple way to find a good allocation solution would be to balance system load taking into account the maximum execution time of each task. But, it is more realistic for the sake of load balancing to consider the average execution time of each task. Sporadic tasks are treated in worst-case analysis as periodic tasks with period equal to the minimum interval between activations. This treatment is too pessimistic for load balance analysis. It is more realistic to treat sporadic tasks as periodic tasks with period equal to the average time interval between activations. If each task has a fixed nominal value it is possible to use these nominal values to judge the load balance quality. Two situations are undesirable: to have too many optional parts in the same processor and to have very valuable tasks concentrated on the same processor. This analysis is not possible in applications where the nominal value of a task varies along the time.

Several experiments were conducted to determine appropriate simulated annealing parameters for the problem and to observe the consequences of using load balance as secondary objective. In most applications there is a positive feedback between values E_e and E_b. A better scheduling solution, most of the time, also results in a better load balancing. In the same way, a better load balancing facilitates system scheduling.

It is important to note the simulated annealing efficiency concerning secondary objective. There were experiments where the same set of tasks was allocated using the proposed energy function and using an energy function that takes into account only the scheduling of mandatory parts (value E_e) while ignoring load balance (value E_b). In general, when simulated annealing has load balancing as its secondary goal, system value is 5% greater than when it considers only system schedulability. Details of these experiments are described in [18].

4.2 Schedulability test

The imprecise computation approach requires an off-line guarantee that each mandatory part will be finished within its respective deadline. In order to analyze this aspect of the problem it is possible to ignore the existence of optional parts and to consider each task made exclusively by its mandatory part.

Many scheduling solutions exist in the literature to guarantee deadlines when a group of tasks execute in a single processor. Similar solutions for distributed systems exist in a smaller number than for the single processor case. Deterministic predictability for mandatory parts can be obtained, for example, through scheduling based on fixed priorities [15]. In this approach, tasks receive priorities according to some specific policy and a schedulability test is executed off-line. The schedulability test must be compatible with the priority assignment policy. An on-line preemptive scheduler produces the execution schedule using the priorities previously assigned.

Precedence relationships among tasks are created by needs of synchronization and/or transmissions of data, and usually appear in distributed systems. Although this also happens in centralized systems, the fact that tasks are distributed among different processors increases the dificulties of scheduling. Any schedulability analysis for distributed systems should be able to work with precedence relations.

The work in [1] shows that it is possible to use offsets to implement precedence relations among tasks. By establishing an offset between the release of two tasks it is possible to guarantee that the successor task will only begin its execution after the predecessor task has concluded. This technique is sometimes called *static release of tasks* [21], because the relative release time of tasks is previously defined in terms of offsets. The original system is translated into an equivalent system where tasks are independent but with offsets that enforce the precedence relations.

It is also possible to implement precedence relations through an explicit message from the predecessor task to the successor task. This message informs that the predecessor task is concluded and the successor task can be released. The message can also contain some data. This technique is sometimes called *dynamic release of tasks* [21], because the successor release time depends on the instant the predecessor task finishes; this value is only known at run-time. The uncertainty regarding the successor release time can be modeled as a release jitter [24]. For the sake of analysis, the original system is transformed in an equivalent system where tasks are independent but with a release jitter. In this case schedulability tests are applied to the equivalent system.

Most published work use fixed priorities and static task release (offsets) to implement precedence relations ([1], [24]). Dynamic release is used in [22] and [24] for distributed systems analysis where there are only linear precedence relations. The only work known by the authors that uses dynamic task release to implement arbitrary precedence relations in distributed systems appears in [5]. The analysis presented in [5] is such that precedence relations are first transformed in release jitter and, after that, all tasks can be analyzed as independent tasks.

We propose in this work the use of fixed priorities and dynamic task releases in the scheduling of critical real-time distributed systems. We assume that tasks receive priorities following the deadline monotonic policy (DM, [14]). Deadline monotonic is a simple and fast way to assign priorities. DM is not optimal in systems with precedence relations among tasks. However, DM is optimal in the class of all policies that generate decreasing priorities within the precedence graphs [11].

The schedulability test used in this work is described in [19]. It studies the schedulability of real-time distributed applications where tasks may present arbitrary precedence relations. It assumes that tasks are periodic or sporadic, and have deadlines equal to or less than their period.

Our work in [19] develops transformation rules that start from a task set with precedence relations and create an equivalent set of independent tasks with release jitter. By eliminating all precedence relations in the task set we can apply any available schedulability test valid for independent tasks. The equivalent task set is such that the maximum response time of task T_i in the new set is equal to or greater than the maximum response time of the same task in the original set. A schedulability test based on the transformation rules will be sufficient but not necessary.

For each task of the original set, the schedulability analysis is resolved in three steps:

1. Apply transformation rules to create an equivalent set;
2. Compute the maximum response time for the task in the equivalent set;
3. Compare the computed maximum response time with task deadline.

In [19] each transformation rule is presented and proved as a theorem. Rules 1 to 3 are applied when a task T_i, single task of activity A_x, receives interference from tasks of another activity A_y that completely resides on the same processor. Figure 2.a illustrates rule 3 where the A_y interference is reduced to the tasks with higher priority than T_i (T_8 in figure 2a) and it is equivalent to a task with execution time corresponding to the addition of execution times of T_5, T_6 and T_7 ($i<j$ implies that $\rho(T_i)>\rho(T_j)$).

Rules 4 and 5 are applied to distributed activities that interfere with an independent task T_i. They eliminate all remote dependence of these distributed activities so tasks in the same processor of T_i no longer depend on tasks executed in other processors. Rules 6 to 11 are used when a task T_i, subject to precedence relations within its own activity A_x, receives interference from tasks of other activities. These rules transform the activity that contains task T_i until T_i becomes an independent task. Figure 2.b shows rule 8, which transforms a remote precedence to a jitter. A task with two precedences is reduced by rule 11 (figure 2.c) to a task presenting only one precedence, corresponding to the worst path in the precedence graph.

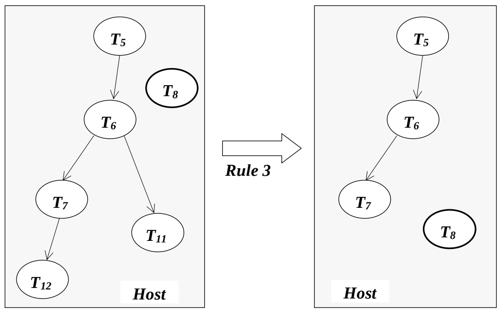

Figure 2.a. Transformation Rule 3

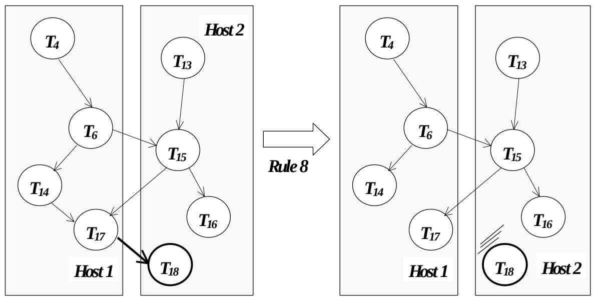

Figure 2.b. Transformation Rule 8

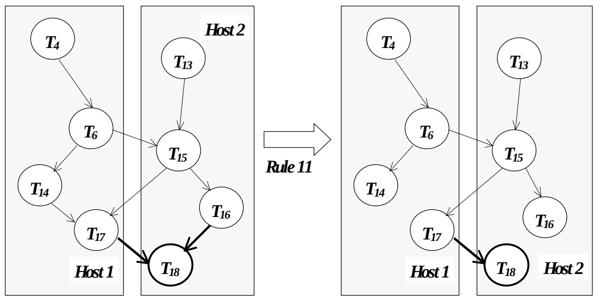

Figure 2.c. Transformation Rule 11

The transformation rules are organized according to an algorithm that computes the maximum response time of each task. This algorithm is based on successive transformations of the application to eliminate all precedence relations, applying the rule set. Finally, the algorithm computes the maximum response time of the independent task, for example, we can use the algorithm described in [2]. The algorithm presented in [19] has a complexity given by $\mathbf{O(n^2+n.E)}$, where **E** represents the complexity of the schedulability test used at the end for sets of independent tasks.

The method described in [19] has been evaluated by simulation. Its performance has been compared with the performance of a simple and direct transformation as used in [5], where every precedence relation is transformed into release jitter. Both methods transform the original task set into a set of independent tasks that can be analyzed by the algorithm presented in [2]. The performance in both tests perform is similar when processor utilization is 40% or less. When processor utilization is 50% or higher the test in [19] is less pessimistic than that in [5].

4.3 Admission policy

It is possible that the spare capacity identified off-line is not big enough to execute all optional parts available. In this case it is necessary to choose among optional parts available for execution at a certain instant which ones should be executed and which ones should be discarded. This choice should consider the importance of each task.

In many systems the importance of executing any task depends on the past behavior of the system. A common situation consists in periodic tasks where the importance of a precise execution increases when that task has been imprecisely executed in past activations. It is desirable to execute the task precisely to interrupt its accumulation of imprecision. This kind of dependence is called intra-task [17] and examples appear in applications such like radar systems and control systems [6].

Tasks that introduce a producer-consumer relationship also present dependence with regard to the importance of a precise execution. A producer task generates an output data that will be used as input by the consumer task. Many algorithms present a smaller execution time when they receive an input data of better quality. So, by precisely executing a producer task it will decrease the execution time of the consumer task. This type of dependence is called inter-task [17] and examples appear in voice recognition, radar systems and image processing [9].

Most studies on imprecise computation consider that tasks are not affected by the behavior of other tasks or by what happened to their previous releases. In [9] the task model includes inter-task dependence. Error in the input data of a task can extend the worst-case execution time of its own mandatory and/or optional parts. In [6] the task model includes intra-task dependence. It is supposed that all tasks should be precisely executed at least once every few releases. In [6] and [9] the error model is such that an optional part may become mandatory.

Algorithms that select optional parts for execution should contemplate the aspects described above. Intra-task and inter-task dependence should be considered when comparing the importance of the precise execution of different tasks. Since these algorithms are executed on-line, they should present a small computational cost.

The approach described in section 3 includes the execution of optional parts when the processor is not executing mandatory parts. The aim of an admission policy is to discard optional parts that present a too small gain for the system. System overhead is reduced because the acceptance test does not take into account optional parts that were refused by the admission policy. This scheme was presented before in [8] where two heuristics, FCFS and AVDT, have been described as admission policies for systems with optional components: when policy FCFS is used, all optional components are always admitted; when policy AVDT is used, only optional components with a value density greater than the average system value density are considered. The value density ($\lambda_{i,k}$) is obtained by the division of the task value by its worst-case execution time. Task model used in [8] assumes independent tasks.

In [17] two heuristics are proposed as admission policy for imprecise tasks. The two policies, CVDT and INTER, consider the existence of intra-task and inter-task dependence. They are analyzed through simulation and the results compared to those obtained for FCFS and AVDT. The simulation goal is to determine if they are able to increase total value of the system. CVDT and INTER are adopted in this work since they present better results in the presence of dependence between tasks.

Policy "Compensated Value Density Threshold" (CVDT) applies a principle similar to that of AVDT. Only optional parts that have a value density $\lambda_{i,k}$ greater than a certain threshold will be considered for execution. The threshold ζ used by CVDT is given by:

$$\zeta = \Lambda_p(t) \times \mathrm{Min}\left(5 \times \pi_p(t), 1.1\right),$$

where **p** is the processor which executes T_i and $\pi_p(t)$ represents the acceptance test rejection rate for the processor **p**, that is, the number of rejected optional parts divided by the number of optional parts offered to the acceptance test at processor **p**. The average value density $\Lambda_p(t)$ of processor **p** is multiplied by a correction factor that takes into account the acceptance test rejection rate. When the rejection rate is 20%, value $\Lambda_p(t)$ is used as threshold. A rejection rate greater than 20% indicates that admission policy is too loose and the threshold should increase. If the rejection rate is smaller than 20% then admission policy is too tight and the threshold should decrease. Constant 1.1 appears in the equation to establish an upper limit for the correction factor. The values of the correction factor upper limit (1.1) and the ideal rejection rate (20%) has been set from several experiments.

Policy "Compensated Value Density Threshold for Inter-Task Dependencies" (INTER) is similar to CVDT in the sense that it compares a value density with a threshold. INTER applies the same threshold ζ as defined before. When activation $T_{i,k}$ is going to start its execution, threshold ζ is compared with the sum of its value density $\lambda_{i,k}$ and correction factor $\varepsilon_{i,k}$. The optional part of $T_{i,k}$

will be admitted only when $\lambda_{i,k} + \varepsilon_{i,k} \geq \zeta$. The correction factor $\varepsilon_{i,k}$ considers that a precise execution of $T_{i,k}$ reduces the mandatory execution time of its successor tasks. In [17] the computation of $\varepsilon_{i,k}$ appears in detail.

4.4 Acceptance test

In order to guarantee off-line that all mandatory parts will be concluded before their respective deadline, it is necessary to reserve resources for the worst-case scenario. Worst-case scenario, as used here, refers so much to execution times as to the worst possible combination of task releases. This scenario is tipically much more pessimist than the average case. Since resources were reserved for worst-case, every time a task has a behavior with less demand, it generates some spare capacity. There are several on-line factors able to generate spare capacity with respect to system resources (mandatory parts are concluded with fewer resources than it was reserved; sporadic tasks are activated with a smaller frequency than the maximum frequency established at design time, etc). This spare capacity is used to execute optional parts.

Optional parts must use only the spare capacity so they will not jeopardize the execution of previously guaranteed mandatory parts. Algorithms for the detection of spare capacity tend to be simple and not very efficient or efficient but very complex. Also, algorithms that solve it must be used on-line, when they dispute resources (processor time) with the optional parts themselves. An optimal acceptance test is defined as able to detect the whole spare capacity in the system as soon as it is generated. Optimal solutions are unfeasible in practice, due to the complexity of its algorithms.

Many works in the literature try to give an on-line guarantee to aperiodic tasks. Although such algorithms have not been specifically created for imprecise computation, they can be used as acceptance test. From the point of view of scheduling, an optional part can be considered as an aperiodic task that was not guaranteed off-line and needs a dynamic guarantee.

Dynamic slack stealing has been presented in [7]. This algorithm calculates the spare capacity of the system at run-time. The dynamic slack stealing algorithm results in excellent efficiency with regard to the identification of spare capacity. Meanwhile, the overhead of this algorithm is very high and consequently not feasible.

It is proposed in [3] a modification of the dynamic slack stealing algorithm called dynamic approximate slack stealing (DASS). This approach is such that every spare capacity identified really exists. DASS is a compromise between efficiency in the identification of spare capacity and computational cost of the algorithm. The algorithm DASS was selected as the acceptance test for the approach presented in this work. DASS was originally created for applications without precedence relations, and executed in monoprocessors. It is necessary to modify DASS before using it in our task model. We will describe a modified DASS (MDASS) introduced in [18].

In our task model, when a task is going to start its execution there is an automatic request of guarantee for its optional part. The request is for an execution time O_i and a deadline equal to deadline D_i of the mandatory part. Algorithm DASS keeps slack counters $S_i(t)$ for each priority level **i**. These slack counters are checked to determine if the optional part of task T_i can be accepted or not. DASS computes the extra interference each task can receive without increasing its maximum response time beyond its deadline. By applying the original DASS, the maximum response time of a task may increase, but it will never become greater than the task deadline. In the modified DASS (MDASS), an optional part is accepted only if its execution will not modify the maximum response time of local tasks with remote successors.

Section 2 has described inter-task dependence. It can be used to increase the efficiency of MDASS in the identification of slack. Suppose there is a precedence relation between T_j and T_i such that a precise execution of T_j decreases the execution time of T_i. The knowledge of this fact is used to increase the efficiency of MDASS.

Algorithm MDASS has been implemented and many experiments conducted. Data structures necessary to implement MDASS occupy a small amount of memory. They have to keep task properties that are static (period, maximum execution time, etc), the amount of slack $S_i(t)$ of each task and how much processor time each task has already used in its present activation. It is also necessary to keep the instant of last activation of sporadic tasks.

5. Conclusions

In this work we have analyzed some questions related with real-time scheduling of imprecise tasks in distributed systems. The problem analysis was divided in four aspects: allocation of tasks to processors, off-line guarantee for mandatory parts, on-line identification of spare capacity and on-line selection of optional parts for execution when it is not possible to execute everyone. Once we have defined the basic task model to be used, the adopted approach was described, that is, how to schedule imprecise tasks in distributed systems. We have proposed algorithms for the four specific scheduling problems.

The approach proposed in this work simultaneously provides the necessary off-line guarantee for critical systems and increases system utility through execution of optional parts. The cost of a critical system is reduced because only the critical functions and properties receive a guarantee for its behavior in a worst-case situation. Scheduling of optional parts is based on best-effort. The

Imprecise Computation technique can be seen as a mechanism capable of tolerating overloads without compromising its critical timing constraints.

We have done many simulations that provided an opportunity to use the complete solution proposed in this work. Initially, applications have been generated at random, but within certain mandatory and optional utilization patterns. Then tasks have been allocated according to the solution based on simulated annealing described in section 4.1. As part of the allocation problem, schedulability of the task set was tested by the algorithm indicated in section 4.2. Once allocation has been completed, the application execution has been simulated. The solution indicated in section 4.4 has been used during the execution simulation to identify spare capacity and to establish an acceptance test. Finally, admission policies described in section 4.3 have been included in the simulation.

The contribution of this paper should be understood as the presentation of a complete approach for scheduling imprecise tasks in distributed systems. Although the scheduling solutions appear summarized in this text, they have been formalized, analyzed and implemented. Theorems and simulations used for this purpose can be found in [17], [18] and [19].

References

[1] N. Audsley, K. Tindell, A. Burns. "The End of the Line for Static Cyclic Scheduling?" Proceedings of the Fifth Euromicro Workshop on Real-Time Systems, IEEE Computer Society Press, pp. 36-41, june 1993.

[2] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, A. J. Wellings. "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling." Software Engineering Journal, Vol. 8, No. 5, pp.284-292, 1993.

[3] N. C. Audsley, R. I. Davis, A. Burns. "Mechanisms for Enhancing the Flexibility and Utility of Hard Real-Time Systems." Proc. of the IEEE Real-Time Systems Symp., pp. 12-21, San Juan, Puerto Rico, december 1994.

[4] A. Burns, A. J. Wellings. "Criticality and Utility in the Next Generation." The Journal of Real-Time Systems, Vol. 3, correspondence, pp. 351-354, 1991.

[5] A. Burns, M. Nicholson, K. Tindell, N. Zhang. "Allocating and Scheduling Hard Real-Time Tasks on a Point-to-Point Distributed System." Proceedings of the Workshop on Parallel and Distributed Real-Time Systems, pp. 11-20, 1993.

[6] J. Y. Chung, J. W. S. Liu, K. J. Lin. "Scheduling Periodic Jobs that Allow Imprecise Results." IEEE Trans. on Computers, Vol.39, No.9, pp.1156-1174, sep 1990.

[7] R. I. Davis, K. W. Tindell, A. Burns. "Scheduling Slack Time in Fixed Priority Pre-emptive Systems." Proc. IEEE Real-Time Syst. Symp., pp.222-231, 1993.

[8] R. Davis, S. Punnekkat, N. Audsley, A. Burns. "Flexible Scheduling for Adaptable Real-Time Systems." Proceedings of the IEEE Real-Time Technology and Applications Symposium, pp. 230-239, may 1995.

[9] W. Feng, J. W.-S. Liu. "Algorithms for Scheduling Tasks with Input Error and End-to-End deadlines." Un. of Ill. at Urbana-Champaign, TR #1888, 1994.

[10] W. Feng, J. W.-S. Liu. "Performance of a Congestion Control Scheme on an ATM Switch." Proc. of the Int. Conference on Networks, Florida, jan 1996.

[11] M. G. Harbour, M. H. Klein, J. P. Lehoczky. "Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems." IEEE Transactions on Software Engineering, Vol. 20, No. 1, pp. 13-28, january 1994.

[12] X. Huang, A. Cheng. "Applying Imprecise Algorithms to Real-Time Image and Video Transmission." Proc. IEEE Workshop on Real-Time App. Chicago, may 1995.

[13] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi. "Optimization by Simulated Annealing." Science(220), pp.671-80, 1983.

[14] J. Y. T. Leung, J. Whitehead. "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks." Perf. Evaluation, 2(4), pp.237-250, dec. 1982.

[15] C. L. Liu, J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." Journal of the ACM, Vol. 20, No.1, pp.46-61, jan 1973.

[16] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, J.-Y. Chung. "Imprecise Computations." Proc. of the IEEE, Vol. 82, No. 1, pp. 83-94, january 1994.

[17] R. S. Oliveira, J. S. Fraga. "Scheduling Imprecise Computation Tasks with Intra-Task / Inter-Task Dependence." Proc. of the 21st IFAC/IFIP Workshop on Real Time Prog., Gramado-Brasil, pp.51-56, nov. 1996.

[18] R. S. Oliveira. "Escalonamento de Tarefas Imprecisas em Sistemas Distribuídos." Tese de Doutorado, Dep. de Eng. Elét., Univ. Fed. Santa Catarina, fevereiro 1997.

[19] R. S. Oliveira. J. S. Fraga. "Fixed Priority Scheduling of Tasks with Arbitrary Precedence Constraints in Distributed Hard Real-Time Systems." Journal of Systems Architecture (The EUROMICRO Journal), vol. 46, no. 11, pp. 991-1004, september/2000.

[20] K. Ramamritham, J. A. Stankovic, W. Zhao. "Distributed Scheduling of Tasks with Deadlines and Resource Requirements." IEEE Transactions on Computers, Vol. 38, No. 8, pp. 1110-1123, august 1989.

[21] J. Sun, J. W.-S. Liu. "Bounding the End-to-End Response Time in Multiprocessor Real-Time Systems." Proc. Workshop on Parallel and Distributed Real-Time Systems, pp.91-98, Santa Barbara, CA, april 1995.

[22] J. Sun, J. W.-S. Liu. "Synchronization Protocols in Distributed Real-Time Systems." Proc. 16th Int. Conf. on Distributed Computing Systems, may 1996.

[23] K. W. Tindell, A. Burns, A. J. Wellings. "Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy." Real-Time Systems., Vol.4, No.2, jun 1992.

[24] K. Tindell, J. Clark. "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems." Microprocessors and Microprogramming, 40, 1994.

[25] J. Xu, D. L. Parnas. "On Satisfying Timing Constraints in Hard-Real-Time Systems." IEEE Transactions on Software Engineering, Vol.19, No.1, pp.70-84, Jan 1993.

[26] A. C. Yu, K.-J. Lin. "A Scheduling Algorithm for Replicated Real-Time Tasks." IEEE IPCCC'92, pp. 395-402, 1992.