Scheduling within Temporal Partitions: Response-time Analysis and Server Design

Luis Almeida
LSE-IEETA/DET
Universidade de Aveiro
Aveiro, Portugal
lda@det.ua.pt

Paulo Pedreiras
LSE-IEETA/DET
Universidade de Aveiro
Aveiro, Portugal
pedreiras@det.ua.pt

ABSTRACT
As the bandwidth of CPUs and networks continues to grow, it becomes more attractive, for efficiency reasons, to share such resources among several applications with the minimum level of interference. This can be achieved using temporal partitions, with each application assigned to its own partition and executing as if it was executing alone on a resource with lower bandwidth. The partitions are associated to servers that execute the application tasks according to a given application-level scheduler. On the other hand, the set of servers is scheduled by a system-level scheduler. This paper addresses the particular case of fixed priorities-based application-level schedulers together with a periodic server model at the system level. It starts with an adequate response time analysis based on the notion of server availability for a known server. Then it addresses the inverse problem of designing a server with minimum system-level resource requirements to fulfill the application time constraints. In this context, the paper shows that response time based schedulability tests with linear time bounds do not need to consider all tasks but just a small subset, which may lead to substantial speed-ups. The proposed method goes a step further with respect to other recent works in the literature by considering a more complete task model, effectively computing the server parameters and establishing a better trade-off concerning complexity and tightness.

Categories and Subject Descriptors
D.4.7 [**Operating Systems**]: Organization and Design – *Real-time systems and embedded systems.*

General Terms
Algorithms, Design, Theory.

Keywords
Real-time systems, real-time scheduling, hierarchical scheduling, response-time analysis.

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

1. INTRODUCTION
Hierarchical scheduling has been generating a considerable interest, recently, due to its ability to separate the concerns of scheduling at the system and application levels. It is a fundamental brick in the current trend towards higher integration and flexibility in embedded systems [8], which opens the way to higher efficiency and lower costs by means of resource sharing, as well as to higher resilience to hardware failures by means of dynamic reallocation of computing or communication entities [7].

Hierarchical scheduling is intimately connected with resource temporal partitioning according to which a shared resource, e.g. CPU or network, is used by several complex applications each of which is composed of a set of entities, e.g. tasks or streams. These entities must be scheduled internally to the application inside one specific resource partition to which they were allocated. At a higher level, all resource partitions are scheduled using a given system-level policy. The concept of server is well adapted to this level, supporting temporal isolation among partitions (Figure 1).

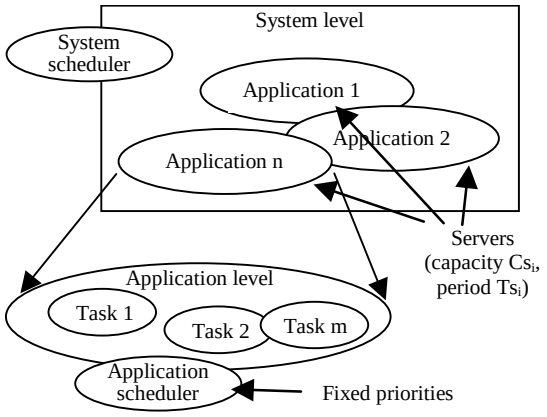


Figure 1. Hierarchical scheduling framework.

However, two problems arise: how to derive real-time guarantees for the applications running within each server and, conversely, how to design the server for a given application so that it fulfils the application requirements with the least resource utilization.

This paper addresses both problems. It relates directly to recent work available in the literature and constitutes a further contribution to the analysis and design of temporally partitioned

systems. A preliminary work-in-progress version was presented in [10]. The related work is discussed in section 2, which also highlights the contributions of this paper, section 3 presents the task model, section 4 describes the response-time analysis and section 5 shows the server design approach, with an example in section 6. Finally, section 7 extends the previous results including intra-server blocking and section 8 concludes the paper.

2. RELATED WORK

The problem of hierarchical scheduling bears many resemblances with other scheduling problems that have been tackled in the past, such as those regarding exclusions [5] and inserted idle-time [9][2]. In fact, looking to the system from the perspective of one application executing within a server, i.e. a temporal partition, the periods of time in which the respective server cannot execute, e.g. because another server has been scheduled at the system level, can be seen as exclusion periods or periods of inserted idle-time.

Moreover, within specific scopes, such as real-time communication over shared media, some forms of hierarchical scheduling/ temporal partitioning have long been used. This is the case with TDMA (as in TTP/C), in which each node has one or more dedicated slots in the TDMA round to transmit its traffic, as well as with the multi-phased cyclic framework (as in WorldFIP, FlexRay or FTT-CAN [1]), with several phases used in sequence within a micro-cycle, each for a given type of traffic (Figure 2). In both cases, either slots or phases can be taken as periodic servers within which several message streams must be scheduled.

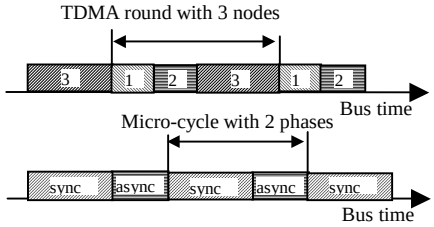


Figure 2. Common partitioning schemes for shared buses used in real-time communication .

However, recent work has brought to light new results that are more general and abstract away the specific application scope. In [7] the authors use the concepts of virtual resource, virtual time, bounded-delay resource partitioning and server supply function to deduce real-time guarantees for hierarchical partition scheduling. In [6] the focus is on hierarchical resource partitioning using fixed priorities local schedulers and both deferrable and sporadic servers. It presents a response-time analysis as well as utilization bounds. In [4], the authors deduce a tighter schedulability test for partitions using fixed priorities local schedulers and address the issue of server design in order to meet the application time constraints. In [12], the authors present a response time analysis for a composed model in which EDF local scheduling is executed within a system-level fixed priorities framework. The task model considers several practical issues such as release jitter and inter-task synchronization blocking. Finally, in [11] the authors present an analysis for hierarchical partition scheduling, considering both fixed priorities and EDF local scheduling, together with a generic periodic resource model. A general scheduling interface is also presented that facilitates the composition of partitions and derivation of real-time guarantees. The paper also addresses the inverse problem of defining a partition in order to meet a given set of real-time requirements.

This paper relates closely to the works referred above. It is interesting to note that these works are very recent and some of them were submitted in parallel, leading to overlapping of short parts. Our motivation was to extend the work started in [13] and later improved in [1], in which we developed a response-time analysis for the asynchronous traffic within a double phase cyclic framework as depicted in Figure 2. In this paper we use the same reasoning but with a more general model that fits well in task scheduling. We will consider a fixed priorities local scheduler together with a periodic server model to manage a resource partition. Then, we derive upper bounds to the worst-case response time of tasks executing within a given server. We will also use the concept of server supply function as in [7], or resource supply in [11], but we will refer to it as server availability function for coherence with our previous work. The response-time analysis we propose is similar to the one for fixed priorities presented in [11] but we extend it to cover a more realistic task model that includes deadlines shorter than or equal to periods, release jitter and synchronization blocking. Our response-time analysis is also equivalent, despite different, to the one in [6] but we believe ours is more flexible since it can be easily adapted to irregular partitions as in [1] and [14], by describing them analytically in the respective server availability function (see further on).

The server design problem is addressed in both [4] and [11]. When compared to the former one, our method is simpler but less tight and, as shown later, may represent a more favorable compromise between tightness and complexity in certain circumstances. On the other hand, [11] follows the same reasoning as we did in [13] of matching the application demand with the server supply to derive the worst-case response time. The corresponding result presented in [11] is equivalent to the one we presented in [10]. In this paper we extend that work by generating the server period that minimizes the server bandwidth together with the overhead caused by context switching at the system level, using the cost function proposed in [4]. Finally, we consider release jitter as well as intra-server blocking in our task model, which none of the other approaches does.

In summary, this paper builds upon the work in [13],[1], [4] and [11] and proposes the following:

- extension of the task model to a more realistic one;
- alternative way of deducing the solution space for the server parameters;
- new method to search the server solution space based on the new concept of application external points;
- full computation of the server parameters that minimize a given cost function.

3. TASK MODEL

In this work, we consider that a given active resource, say a CPU, is used to execute a set of independent applications. Each application Ω is composed by a set Γ_Ω of N_Ω tasks, $\Gamma_\Omega \equiv \{\tau_i(C_i,T_i,D_i,J_i,P_i),\ i=1..N_\Omega\}$, in which each task, at this point, will be considered independent and fully preemptive. Each task τ_i

is characterized by a period T_i, a worst-case execution time C_i, a relative deadline D_i that is shorter than or equal to the respective period, a maximum release jitter of J_i and a fixed priority P_i that may possibly be derived from the period, the deadline, or any sub-optimal criterion. Also, the set of tasks Γ_Ω executes by priority order within a periodic server S_Ω, which is characterized, at a system level, by a period T_S and a capacity C_S. The server can be of any type as long as, under continuous demand, it behaves like a periodic task executing for C_S time units within every T_S time units interval. This is also the server model considered in [11] and [4] ([6] considers deferrable and sporadic models, only).

The system-level scheduler that schedules partitions, and the current system load, determine when the server is available to execute application tasks. We thus define the server *availability function*, $A_S(t)$, which returns for each instant t the cumulative CPU time available for the application to execute since an arbitrary time origin. For static system level scheduling such as TDMA (Figure 2), $A_S(t)$ may be exactly characterized a priori. However, if an on-line system-level scheduling policy is used, the specific pattern of $A_S(t)$ may be difficult to predict. Therefore, for the sake of independence with respect to the system-level scheduling policy and load, it is helpful to use a lower bound $A\_{S}(t)$ that assures that the cumulative CPU time available for an application is never smaller than a given value.

In order to determine a lower bound to the availability function, we can use the same reasoning as in [4] or [11]. Basically, it considers the worst-case server availability pattern with respect to the arbitrary time origin in which the server suffers maximum latency (Δ) in the beginning and then follows a periodic pattern with its capacity available at the end of each periodic instance (Figure 3). The value of Δ depends directly on the maximum finishing jitter[1] of the server execution, as determined by the system scheduler. In certain cases, e.g. in regular TDMA schedules, such jitter is eliminated because the server executes in a strict periodic fashion, leading to $\Delta = T_S - C_S$. Thus, Δ will vary between this best-case value and the worst-case depicted in Figure 3 in which $\Delta = 2*(T_S - C_S)$. For the sake of generality we will consider such initial latency as given by Equation 1.

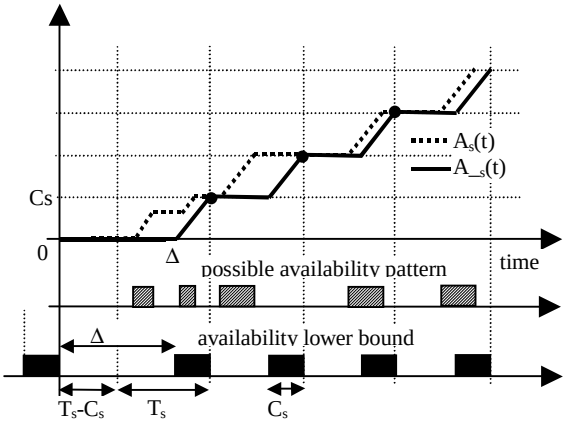


Figure 3. Server availability functions $A_S(t)$ and $A\_{S}(t)$.

[1] Finishing jitter refers to the absolute jitter that affects the instant in which the server exhausts its capacity within each period.

$$\Delta = (1+\beta)*(T_S - C_S) \quad (1)$$

with

$$\beta = (R_S - C_S) / (T_S - C_S)$$

Notice that β ($0 \le \beta \le 1$) is a normalized measure of the maximum server finishing jitter, while R_S ($C_S \le R_S \le T_S$) is the maximum relative finishing instant of all server instances. To maintain independency from the system scheduling policy $\beta = 1$ should be considered (worst-case).

For simplicity, in the remainder of the paper we will use the same expression *availability function* for the lower bound function, and refer to it as $A\_{S}(t)$ (Equation 2). In [4] this function is called server characteristic function, in [7] server supply function and in [11] resource supply bound function.

$$A\_{S}(t) = \begin{cases} 0, & t < \Delta \\ t - (\Delta + k*(T_S - C_S)), & \Delta + k*T_S \le t < \Delta + k*T_S + C_S \\ (k+1)*C_S, & \Delta + k*T_S + C_S \le t < \Delta + (k+1)*T_S \end{cases} \quad (2)$$

$$k = \lfloor (t-\Delta)/T_S \rfloor$$

4. RESPONSE-TIME ANALYSIS

A relatively simple but effective way of upper bounding the response-time for each task τ_i within a given server S is to use the same reasoning explained in [13] and [1]. The fact that we are now using a fully preemptive task model together with a periodic server further simplifies the analysis therein presented, which was based on non-preemption with inserted idle-time together with a background server.

Therefore, we compute for each task τ_i and for each instant t the maximum load submitted to the server by the task itself after its release together with all higher priority tasks. We call this the *level i submitted load function*, $H_i(t)$ (Equation 3). It can be determined by the usual methods in fixed-priorities response time analysis [3] since the critical instant for each task is not changed by the presence of the server [6]. Equation 3 considers Γ_Ω sorted by decreasing priorities, $\forall_{i,j}\ i<j \Leftrightarrow P_i>P_j$.

$$H_i(t) = \sum_{j=1}^{i} \left\lceil (t + J_j)/T_j \right\rceil * C_j \quad (3)$$

The worst-case response time for task τ_i, referred to as R_i, can thus be obtained as expressed in Lemma 1.

Lemma 1. Given the task set Γ_Ω executed within a server S with availability function $A\_{S}(t)$, the worst-case response time R_i for task $\tau_i \in \Gamma_\Omega$ is obtained by determining the earliest instant in which the maximum level i submitted load $H_i(t)$ matches the least server availability $A\_{S}(t)$.

Proof: Lemma 1 can be proved by considering the definitions of both $A\_{S}(t)$ and $H_i(t)$. In fact, $A\_{S}(t)$ stands for the minimum execution time that the server can deliver to the application counted from t=0. On the other hand, $H_i(t)$ stands for the maximum execution time required to execute τ_i to completion, when released at t=0 and considering the maximum interference it may suffer by higher priority tasks within the application. Thus, the worst-case response time is given by the instant when the least availability is just enough to cover the longest requested

execution time (Figure 4). If the server, in one or more instances, executes before than considered in $A\__S(t)$ the response time can only be shorter. ♦

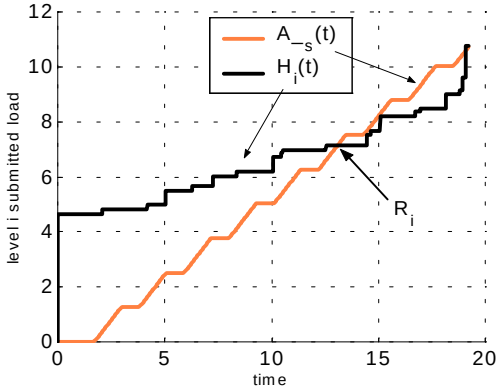


Figure 4. Worst-case response time of task τi.

With Lemma 1, we can perform a trivial schedulability test as stated in theorem 1.

Theorem 1. The task set Γ_Ω executed within a server S with an availability function $A\__S(t)$ is schedulable if (and only if) $\forall \tau_i \in \Gamma_\Omega$ $R_i \le D_i$

Proof: The *if* is trivially proved but the *only if* requires R_i to be an accurate value. This depends on the accuracy of the availability lower bound $A\__S(t)$. In practice, this lower bound will be pessimistic because $A_S(t)$ will not suffer maximum delay in all instances after startup, thus there will always be a given load for which some exact worst-case response times are lower than the computed R_i. In this case, the test in Theorem 1 will be sufficient, only. However, in particular situations, such as regular TDMA schedules, $A\__S(t) = A_S(t)$ and thus the test will be necessary (this is why we kept the *only if* within parenthesis). ♦

The value of R_i can be determined using Equation 4 or, more efficiently, the equivalent Equation 5, which makes use of the inverse of the availability function $A\__S(t)$, referred to as $A_S^{inv}(t)$. This is formalized in Equation 6 (it is referred to as service time bound function in [11]).

$$R_i = \text{earliest } t\text{: } A\__S(t) = H_i(t) \qquad (4)$$

$$\Leftrightarrow R_i = \text{earliest } t\text{: } t = A_S^{inv}(H_i(t)) \qquad (5)$$

Formally, inverting $A\__S(t)$ requires specifying the value of the inverse corresponding to the flat segments of the original function. We consider the lowest values of such intervals, as indicated in Figure 3, which result in the lowest availability.

$$A_S^{inv}(u) = \Delta + m * T_S + (u - m*C_S)\,, \quad m = \lceil u/C_S \rceil - 1$$
$$\Leftrightarrow A_S^{inv}(u) = (\beta + \lceil u/C_S \rceil) * (T_S - C_S) + u \qquad (6)$$

Equation 5 can be solved iteratively with

$$R_i^0 = H_i(0) \quad \text{and} \quad R_i^{n+1} = A_S^{inv}(H_i(R_i^n))$$

that either converges to $R_i = R_i^{n+1} = R_i^n$ or grows beyond the deadline D_i, within finite iterations. This can be easily demonstrated by the fact that, from iteration to iteration, the increment in $H_i(t)$ is lower bounded by $min_{j=1..i}(C_j)$. If the whole CPU is available to execute the application, then $A\__S(t) = A_S^{inv}(t) = t$ and Equations 4 and 5 reduce to the usual response time analysis for fixed priorities scheduling for a similar task model [3].

A simpler but less tight upper bound R_i' for the response time of each task can be obtained considering a linear lower bound to the availability function, also proposed in [4] and [11], herein referred to as *linear availability function* $A\_'_S(t)$ (Equation 7). This function is depicted in Figure 5, with an initial latency of Δ, such as $A\__S(t)$, and then grows linearly with slope $\alpha = C_S / T_S$, i.e. the server bandwidth.

$$A\_'_S(t) = (t - \Delta) * \alpha, \quad \text{for} \quad t > \Delta \text{ and 0 otherwise} \qquad (7)$$

The response time upper bounds can be obtained rewriting Equation 5 using the inverse of $A\_'_S(t)$ (Equation 8).

$$R_i' = \text{earliest } t\text{: } t = \Delta + H_i(t)/\alpha \qquad (8)$$

This allows us to state Theorem 2, which will be particularly helpful in the following section.

Theorem 2. The task set Γ_Ω, executed within a server S with initial latency Δ and bandwidth α, is schedulable if $\forall \tau_i \in \Gamma_\Omega$ $R_i' \le D_i$.

Proof: This can be proved noticing that, for every task τ_i the intersection between $H_i(t)$ and $A\_'_S(t)$ (i.e. R_i'), is always later than or coincident with that between $H_i(t)$ and $A\__S(t)$ (i.e. R_i). Thus, $\forall \tau_i \in \Gamma_\Omega$ $R_i \le R_i'$, proving the theorem. ♦

5. SERVER DESIGN

In this section we tackle the opposite problem of the previous one, i.e. given an application Ω, which parameters (C_S, T_S), or equivalently (α, Δ), should the server S_Ω have so that it requires the least system resources and still meets the application time constraints? In order to address this problem we will start by noticing that such a server should be as tight as possible concerning fulfilling the application time constraints, otherwise it would over consume system resources. Therefore, using the approach presented in the previous section for Theorem 1, we will consider that, for each task τ_i the respective R_i is just on the deadline D_i. This means that the availability function $A\__S(t)$ should be such that intersects $H_i(t)$ exactly at $t=D_i$. To achieve this, we start by defining the set of *deadline points* $DP_\Omega \equiv \{DP_i(D_i, H_i(D_i)),\ i=1..N_\Omega\}$ (Figure 5), which represent the lowest availability required for meeting all application deadlines. Our purpose, then, is to define $A\__S(t)$ so that it is higher than but as close as possible to the set of such points.

In [11] (theorem 5), given a fixed server period T_S, the authors suggest performing an extensive search for the minimum server capacity C_S that still allows meeting the tasks deadlines. However, the search method is not specified. We suggest using Binary Search in the interval $[0, T_S]$, which is relatively efficient. For example, if a resolution of 1/256 of T_S is enough, only 8 iterations are needed.

Correspondingly, we can use the approach suggested in Theorem

2 to find the lowest linear bound $A\_'_S(t)$ that passes above or through all deadline points DP_i. This is also proposed in [11] (theorem 6), where C_S is determined as a function of the period T_S so that $A\_'_S(t)$ touches at least one deadline point. Notice that all task deadlines, i.e. all deadline points, are tested against the linear lower bound.

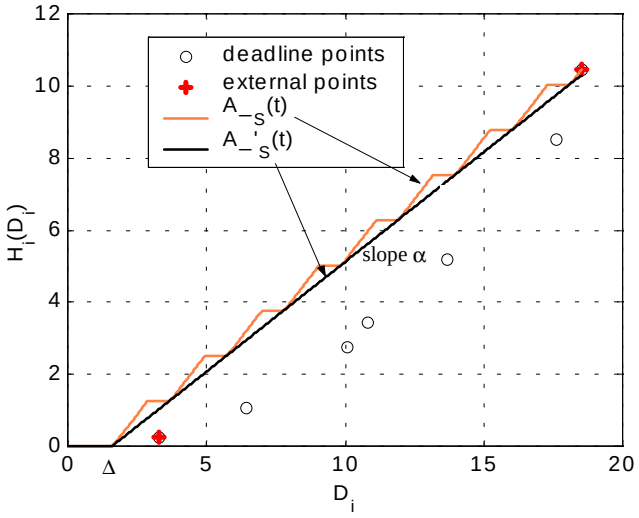


Figure 5. Determining the availability function A_S(t).

However, we are not interested in determining just C_S as a function of T_S but rather in analyzing the space of possible solutions (C_S, T_S) to choose the one that minimizes the use of system resources. For this purpose we determine the subset of *external points*, $E_\Omega \equiv \{E_j(x_j,y_j), j=1..N_E\} \subset DP_\Omega$ (Figure 5), that is the subset of deadline points through which a linear bound (α, Δ) can be drawn that fulfills Theorem 2. Necessarily, such linear bound must also respect its boundary constraints namely $\alpha \leq 1$ (the server cannot use more than the total CPU bandwidth) and, for each E_j, $\alpha \geq y_j/x_j$ (otherwise, $\Delta<0$ and, since $\alpha>0$, it would lead to $T_S<0$ and $C_S<0$). This directly leads to Theorem 3.

Theorem 3. The task set Γ_Ω, executed within a server S with initial latency Δ and bandwidth α, is schedulable if $\forall \tau_i$: $DP_i \in E_\Omega$, $R_i' \leq D_i$.

Proof: Informally, Theorem 3 says that it is sufficient to test the schedulability of the external points to guarantee that the task set is schedulable (Figure 5). It can be proved just by observing the definition of external points. If these points are below a given linear availability function, then all other deadline points are and thus the whole set is schedulable. ♦

The determination of the E_Ω subset is carried out using a simple algorithm that goes through all deadline points in ascending deadline order, starting with the assumption that the first deadline point is an external point. Notice that the slopes of the segments that join every two consecutive external points must be monotonically decreasing from the maximum $\alpha=1$ to the minimum $\alpha = y_{N_E}/x_{N_E}$. Therefore, the algorithm removes from the set of deadline points all those that would violate this decreasing slope pattern and the referred limits. At the end, the points that remain are the external points. The time complexity of the algorithm varies with the characteristics of the task set between N_Ω and N_Ω^2. Also, in general, the number of external points N_E is substantially smaller than the number of deadline points N_Ω (i.e. the number of tasks), which contributes to decrease the complexity of the remaining part of the process.

Figure 6 plots the number of external points against the number of tasks for 1500 randomly generated sets, each constrained to a total bandwidth smaller than but close to 50% and with up to 120 tasks. It is surprising to see how small the number of external points normally is, between 1 and 5, and that there is still a tendency for reduction as the number of tasks grows (for large task sets, the most frequent number of external points is 1 !).

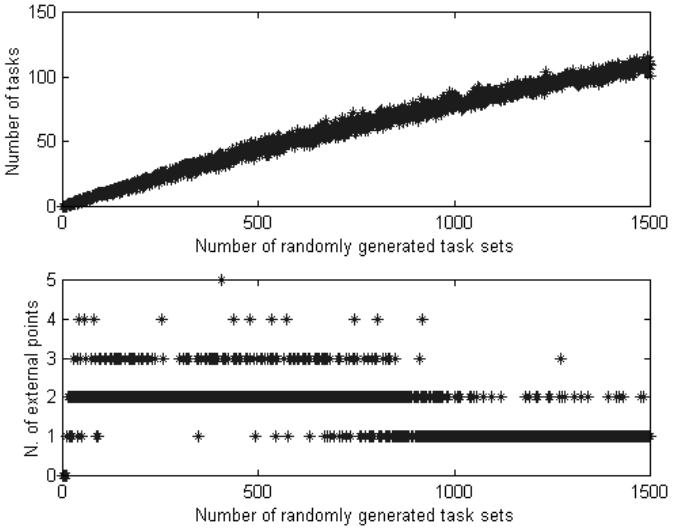


Figure 6. Number of external points in random sets.

Hence, the use of external points may accelerate substantially the execution of repetitive response time based schedulability tests, for example in optimization processes, in which the task set remains unchanged and several server parameters are tested. This is depicted in Figure 7 for several implementations of the test corresponding to theorem 6 in [11], to determine C_S for 200 different values of T_S, and using random task sets of different sizes (each point corresponds to the average of 10 sets with the same size). We used a direct implementation in Matlab ([11](th.6)) and an improved version ([11] (th.6)_imp), as well as both cases using external points, (with ext.pt.) and (with ext.pt._imp), respectively. The benefits of using external points are clear for task sets with more than 8 tasks.

The set of external points defines a solution space for the server design problem. In fact, all linear availabilities $A\_'_S(t)$ that touch at least one external point with a valid slope are possible solutions. Valid slopes are those in between the slopes of the segments that join each point with the previous and with the next, considering the absolute maximum and minimum for the points in the extremes. This can be represented by the union of the following N_E subintervals:

$$\alpha \in \{[1\ \alpha(E_1,E_2)] \cup [\alpha(E_1,E_2)\ \alpha(E_2,E_3)] \cup \ldots \cup [\alpha(E_{N_E-1},E_{N_E})\ y_{N_E}/x_{N_E}]\}$$

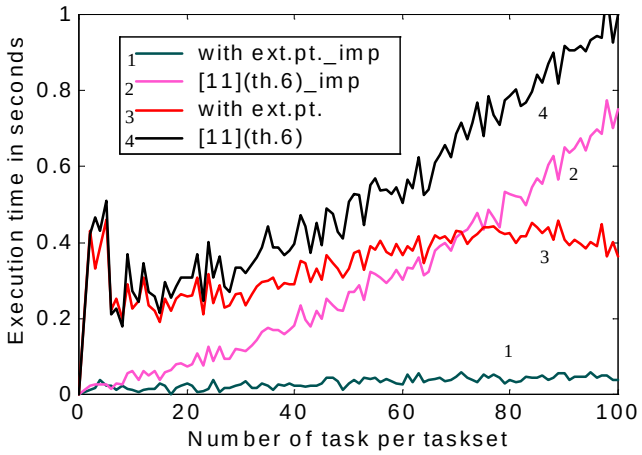


Figure 7. Speed up by using external points.

Each of these subintervals corresponds to $A\_'_S(t)$ lines that pass through one external point $E_j(x_j,y_j)$, j=1..N_E, which can be characterized by the straight-line Equation 9 (left).

$$y=\alpha_j\,(x-x_j)+y_j \quad and \quad \Delta_j = x_j - y_j/\alpha_j \tag{9}$$

Equation 9(right) shows a hyperbolic relationship between α and Δ around each point E_j and also allows determining the respective server period T_S using Equation 10.

$$Ts_j=\Delta_j/((1+\beta)(1-\alpha_j)) \tag{10}$$

The complete solution space in terms of the server parameters (α, Δ) can be obtained by calculating the following intersection for all N_E external points:

$$\cap_{j=1..N_E} \quad 0 \le \Delta_j \le x_j - y_j/\alpha_j \ \text{ and } \ \alpha_j \le 1 \tag{11}$$

This solution space can equivalently be expressed in terms of the server parameters (C_S,T_S) by inserting Δ_j from Equation 9 (right) into 10 and replacing α_j with Cs_j/Ts_j. The result is the following intersection for all N_E external points:

$$\cap_{j=1..N_E} \ Cs_j \le Ts_j \text{ and}$$
$$Cs_j \ge \frac{-(x_j-(1+\beta)Ts_j)+\sqrt{(x_j-(1+\beta)Ts_j)^2+4(1+\beta)y_jTs_j}}{2(1+\beta)} \tag{12}$$

Condition 12 is equivalent to the result presented in [11] (theorem 6). Recall that $x_j = D_j$ and $y_j = H_j(D_j)$.

Finally, in order to find one specific solution, we use the cost function proposed in [4], which considers both the bandwidth directly requested by the server, $\alpha=C_S/T_S$, as well as the overhead bandwidth implicitly used by the server in context switching between applications at the system level. This latter factor can be roughly computed as C_O/T_S where C_O is a system parameter representing the context switching time. The cost function can thus be expressed as $F=\alpha+C_O/T_S$. Minimizing F corresponds to finding the balance between minimizing α and maximizing T_S.

Again inserting Δ from Equation 9 (right) into Equation 10 we obtain $Ts_j=Ts_j(\alpha_j)$, for each α_j subinterval, i.e. each external point. Then, using $Ts_j(\alpha_j)$ within the cost function results in $F=F(\alpha_j)$. The α_j of minimum cost for each subinterval j (referred to as $\alpha_{j,min}$) can be easily determined with a closed formula (Equation 13) obtained by differentiating $F(\alpha_j)$ with respect to α_j and calculating the respective root. Whenever $\alpha_{j,min}$ lies outside the respective subinterval, the closest extreme is considered for minimum. After having determined $\alpha_{j,min}$ for all subintervals (j=1..N_E) it is then just a matter of selecting the one that generates the absolute minimum cost (α_{min}). It is also necessary to identify to which subinterval the α_{min} value belongs to, in order to determine Δ (Equation 9 right) and then T_S (Equation 10).

$$\alpha_{j,\min} = y_j/x_j * \left(1+\sqrt{1-\frac{(y_j-(1+\beta)*C_O)/(x_j-(1+\beta)*C_O)}{y_j/x_j}}\right) \tag{13}$$

The server parameters generated this way are not optimal due to several factors such as the pessimism included in the server initial latency Δ, the successive approximations of the effective server availability function $A_S(t)$ and the use of the deadline points that may lead to worse than necessary bandwidth requirements. However, concerning the $A\_{S}(t)$ approximation by $A\_'_S(t)$, i.e. the linear bound, it is possible to carry out a simple final improvement step. The fact that $A\_'_S(t)$ touches at least one deadline (external) point according to the design method presented above, it does not imply that the corresponding $A\_{S}(t)$ function also touches one, because $\forall_t\, A\_{S}(t) \ge A\_'_S(t)$. Hence, we may allow $A\_{S}(t)$ to actually enter the area bounded by $A\_'_S(t)$ until it also touches one deadline point (this assures that Theorem 1 is still met). This can be done decreasing C_S, increasing T_S or a combination of both.

For simplicity and efficiency according to tests with random sets, we propose increasing T_S, according to Equation 14, which corresponds to maintaining the height of the steps in $A\_{S}(t)$ while expanding the function to the right. This also leads to a further reduction in α. The amount of benefit, however, depends on the task set. Using 1000 random task sets with uniformly distributed periods, we achieved a best improvement of 8% increase in T_S and 7.4% reduction in α. However, on average, the benefits were lower, with 1.8% increase in T_S and 1.7% reduction in α.

$$T_{S,imp} = T_S + \min_{i=1..N_\Omega}\left(\frac{D_i - A_S^{inv}(H_i(D_i))}{\lfloor (D_i+(1+\beta)*C_S)/T_S \rfloor}\right) \tag{14}$$

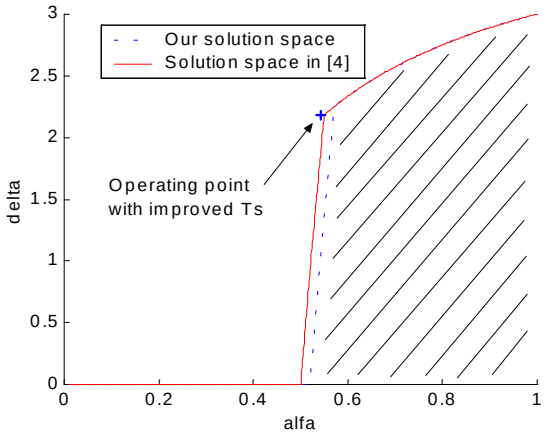
This improvement can be carried out over any non-optimal result obtained by any periodic server based method.

6. EXAMPLE

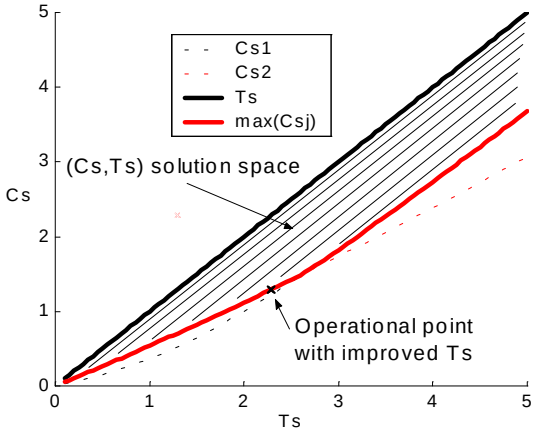
To illustrate the design methodology presented above we make use of the example suggested in [4]. This example consists of an application Ω, with the task set Γ_Ω={(C_i,T_i)= (*1,4*), (*1,10*), (*3,25*) and (D_i=T_i, J_i=*0*, P_i=*1/i*)}. Firstly, we obtain the sets of deadline points DP_Ω={(*4,1*), (*10,4*), (*25,13*)} and external points E_Ω={(*4,1*), (*25,13*)}. Then, using the external points we can derive the α subintervals and using expressions 11 and 12 we can establish the (α,Δ) and (C_S,T_S) solution spaces, respectively (Figure 8).

In the (α,Δ) figure we can see that our solution space is contained in the one derived in [4] and thus it is worse, although for a small difference (less than 4% in the low values of α). On the other

hand, the worst-case time complexity of the method in [4] may reach $2^{N\Omega}$-1, contrasting with the simplicity of our method with a worst-case time complexity of N_Ω^2. Executing the final improvement step with Equation 14 considering C_O =0.1016 generates an operating point (Figure 8) that is slightly better than both solution spaces, with (α,Δ) = (0.544, 2.182) or equivalently (C_S,T_S) = (1.300, 2.391). Just for comparison, the corner in the solid line of Figure 8-a) corresponds to (α,Δ) = (0.55, 2.182). Figure 9 shows the obtained availability functions (top) and variation of the cost function F with respect to T_S (bottom).



a) (α,Δ) solution space

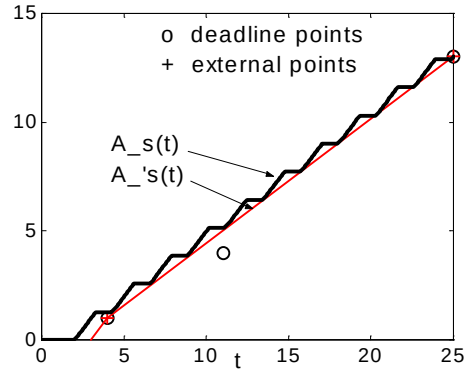


b) (C_s,T_s) solution space

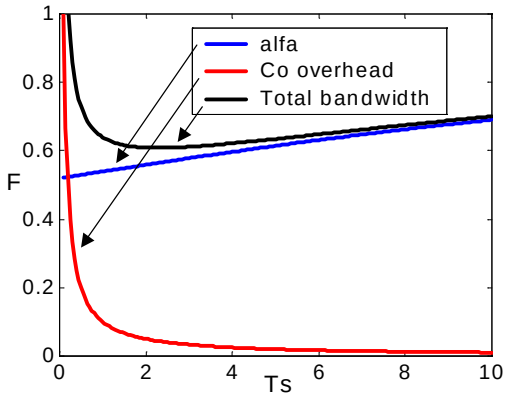
Figure 8. Solution spaces for the given example

7. SYNCHRONIZATION AMONG TASKS

In many practical applications there is the need for synchronization in the access to shared resources that constitute critical sections. There are several protocols specifically developed for real-time systems that allow bounding priority inversions and blocking intervals as well as preventing deadlocks. Our issue here is to discuss whether and how can such protocols be used within the temporally partitioned framework considered in this paper, with fixed priorities local schedulers.



a) Availability functions plus deadline and external points



b) cost function with respect to Ts

Figure 9. Availability and cost functions for the example

In this paper, we consider intra-server synchronization, only, i.e. synchronization among tasks of the same application (Figure 10), which execute within the same partition and require access to resources that are exclusive to that application (*application resources*). These tasks eventually cause blocking to one another, the duration of which depends on the specific synchronization protocol used. Such blocking (*application level blocking* - B_i^a) consists on the execution of critical sections of lower priority tasks, i.e. the blocking tasks, while there are higher priority ones pending, i.e. the blocked tasks. Therefore, the application level blocking B_i^a causes interference to the blocked tasks, delaying them, and consequently must be considered in the respective level i submitted load $H_i(t)$, which must be updated accordingly (Equation 15).

$$H_i(t) = B_i^a + \sum_{j=1}^{i} \left\lceil (t + J_j)/T_j \right\rceil * C_j \qquad (15)$$

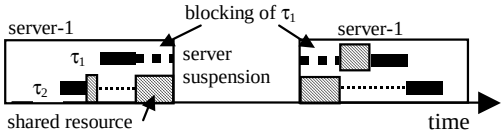


Figure 10. Intra-server blocking.

Moreover, we can state Lemma 2 with respect to the properties of the synchronization protocols.

Lemma 2. All properties of any synchronization protocol applied to shared application resources within a server S, remain the same as established in a non-partitioned processing system, namely the number and duration of priority inversions and deadlock avoidance.

Proof: To prove Lemma 2 notice that semaphore locking and unlocking and the related blocking take place within the application, only, during the execution of the respective server. When the server is suspended until the next periodic instance, the status of all semaphores stays unaltered. Therefore, from the point of view of the synchronization protocol, whichever it is, these periods of time have no impact on the protocol properties, except on the inflation of the blocking that, obviously, can now extend across consecutive server instances. However, this extra delay due to server suspension is the same that the task would suffer just because of executing within a partition and that is already accounted for in the availability function of the server and should not be considered blocking. Thus, excluding this suspension delay, the application level blocking (B_i^a) that a task can suffer is the same as it would be if the respective application was executed in a dedicated system, i.e. non-partitioned. ♦

Lemma 2 plus the updated load function lead to Theorem 4:

Theorem 4. The response time analysis and the server design method proposed previously in this paper apply equally when the task model considers application level blocking as long as:

- the updated load function is used (Equation 15);
- the application tasks can always be suspended at any point of their execution when the server capacity is exhausted or the server is preempted as determined by the system scheduler.

Proof: The proof of this theorem can be easily established by noting that, from a response time analysis point of view, and given Lemma 2, the blocking just represents extra load that is accounted for in the updated load function. Thus, all the reasoning behind the previously shown analysis that is based on the matching between server availability and submitted load still applies. The second requirement guarantees that there is no coupling between the application level synchronization protocol and the system level scheduling. This assures that a server does not execute more than its capacity in any instance, not causing/suffering extra interference on/from other servers running in the system, thus behaving according to our periodic model. ♦

The second requirement of Theorem 4 may conflict with synchronization protocols based on non-preemption. Thus, either preemptive protocols are used, e.g. Priority Inheritance (PIP), Priority Ceiling (PCP) and Stack Resource Protocols (SRP), or the execution platform must be able to separate preemption within the application from preemption at the system level and assuring that the latter is always possible, even when the server is executing a local non-preemptive application section.

8. CONCLUSION

This paper considered the case in which an application composed by several tasks executes within a periodic server with a fixed priorities local scheduling policy. Two main results are presented, the response time analysis for such tasks and the design of the server to allow fulfilling the application time constraints using the least system resources.

The former contribution is a generalization of the well-known worst-case response time analysis for fixed-priority systems [3] that copes with the limited processor availability delivered by a server and it is based on the analysis previously developed for traffic scheduling within the asynchronous messaging system of FTT-CAN [1]. It is also equivalent to the one in [11] but includes a more complete task model with release jitter, deadlines earlier than periods and synchronization blocking.

The latter contribution goes in the same direction as that of [4] but presents a different method that leads to a more favorable compromise between tightness of the solution (slightly lower) and complexity of the process (substantially lower). Again, the presented method also bears similarities with the one in [11] but, not only it includes a more complete task model as referred above, as it also presents an heuristic to deduce the server parameters that minimize resource utilization taking into account the context switch overhead at the system level. Moreover, we present a final optimization step that is applicable to our method as well as to [4] and [11], which reduces the required server utilization when a linear bound to the server service is used.

A spin-off result that seems to have potential for more generalized use is the utilization of external points in the response time analysis. Further work will address the case of inter-server synchronization blocking as well as non-preemption at the application level and its impact at the system level.

9. ACKNOWLEDGMENTS

The authors would like to thank E. Bini and G. Lipari for the fruitful discussions concerning the server design problem that helped in improving the respective part of this paper.

10. REFERENCES

[1] Almeida L., P. Pedreiras, J. A. Fonseca, The FTT-CAN Protocol: Why and How, IEEE Transactions on Industrial Electronics, 49(6), December 2002.

[2] Almeida L., J. Fonseca. Analysis of a Simple Model for Non-Preemptive Blocking-Free Scheduling. Proc. of ECRTS'01 (EUROMICRO Conf. on Real-Time Systems). Delft, Holland. June 2001.

[3] Audsley, N., A. Burns, M. Richardson, K. Tindell and A. Wellings. Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling. Software Engineering Journal, 8(5): 285-292, 1993.

[4] Lipari G. and E. Bini. Resource Partitioning among Real-Time Applications. Proc. of ECRTS'03 (EUROMICRO Conf. on Real-Time Systems). Porto, Portugal. July 2003.

[5] Xu, J., D.L. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. IEEE Trans. on Software Engineering, 16:360-369 March 1990.

[6] Saewong, S., R. Rajkumar, J.P. Lehoczky, M.H. Klein. Analysis of hierarchical fixed priority scheduling. Proc. of ECRTS'02 (EUROMICRO Conf. on Real-Time Systems). Vienna, Austria. June 2002.

[7] Mok, A., X. Feng. A model of hierarchical real-time virtual resources. Proc. of RTSS'02 (IEEE Real-Time Systems Symposium). Austin, USA. December 2002.

[8] Rushby, J., A Comparison of Bus Architectures for Safety-Critical Embedded Systems, CSL Technical Report, SRI International, September 2001.

[9] Howell, R. and M. Venkatrao. On Non-Preemptive Scheduling of Recurring Tasks Using Inserted Idle Times. Information and Computation, **117**, 1995.

[10] Almeida, L. Response-Time Analysis and Server Design for Hierarchical Scheduling. Proc. of the Work-in-Progress session of RTSS'03 (IEEE Real-Time Systems Symposium). Cancun, Mexico. December 2003.

[11] Shin, I. and I. Lee. Periodic Resource Model for Compositional Real-Time Guarantees. Proc. of RTSS'03 (IEEE Real-Time Systems Symposium). Cancun, Mexico. Dec 2003.

[12] Harbour M. and J. Palencia. Response-Time Analysis for Tasks Scheduled under EDF within Fixed Priorities. Proc. of RTSS'03 (IEEE Real-Time Systems Symposium). Cancun, Mexico. December 2003.

[13] Pedreiras, P. and L. Almeida. Combining Time and Event-triggered Traffic in FTT-CAN. Proc. of WFCS'00 (IEEE Work. Factory Communication Systems). Porto, Portugal. Sept 2000.

[14] Mok, A., X. Feng. Real-time virtual resource: A Timely Abstraction for Embedded Systems. Proc. of EmSoft'02 (2nd Int. Conf. on Embedded Software). Grenoble, France. October 2002.