Resource Partition for Real-Time Systems

Aloysius K. Mok, Xiang (Alex) Feng
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
{*mok,xf*}@cs.utexas.edu

Deji Chen
Fisher-Rosemount Systems, Inc.
8627 Mopac North
Austin, TX 78759
Deji.Chen@frco.com

Abstract

We investigate an approach to implement the open system environment idea by means of temporal resource partitions. In this approach, application task groups with hard timing constraints may share the same physical resource and yet be free from the interference of one another. Each resource partition uses only a fraction of the time on the resource. Partitions are specified by two models, a static partition model and a bounded-delay partition model. Both models achieve a clean separation of concerns between task group level scheduling and resource level partition scheduling. The schedulability problems for both preemptive fixed priority and dynamic priority scheduling policies are analyzed.

Keywords: *Resource Partition, Real-Time Task Scheduling.*

1 Introduction

Until recent years, the study of real-time scheduling problems has been primarily concerned with allocating dedicated resources to service a set of application programs which are characterized by a real-time system model. Since the first real-time system model was introduced by Liu and Layland in 1973 [25], there have been many variations proposed to model real-time systems, e.g., the sporadic model [26], the pinwheel model [11]. The schedulability analysis of these models always assumes that the resource to be allocated is made available at a uniform rate and accessible exclusively by the tasks under consideration. This assumption no longer holds in the environment of *open systems* [10] where a physical resource must be time-shared by different task groups and each task group is scheduled as if it has exclusive access to a resource, without interference from other task groups. Two reasons why open systems are considered are: (1) resource sharing is more economical than dedicated resources; (2) in case of hardware failures, an open system environment would allow task groups to be relocated by co-existing with other task groups on the diminished pool of shared resources.

The sharing of resources in open systems poses new difficulties. Since the scheduling policy of each task group assumes exclusive access to a resource, the scheduling policies of the different task groups may conflict with one another. These conflicts may be resolved by a second-level scheduler which coordinates the access to the shared resource by the different task groups. One of the tenets of the open system approach is to avoid performing a global schedulability analysis that considers the timing requirements of all the tasks in all task groups. Ideally, each task group can be analyzed by itself for schedulability. This may be possible if, for example, the shared resource can be time-shared by infinite time-slicing; the net effect is as if each task group has exclusive access to the resource that is made available at a fraction of the actual rate. However, infinite time-slicing is impractical because of the context switching overhead costs and because of resource-specific constraints that may impose a lower bound on the time-slice size. For example, a communication bus cannot be infinitely time-sliced if a bus cycle must be at least as long as the signal propagation latency across the bus.

Practical implementation of the open system approach may be accomplished by customizing the second-level scheduler to take advantage of the common real-time system model of the task groups so as to minimize context switching between task groups. This is the approach in recent work [10, 14]. We shall take a broader view. The general idea is to view each task group as accessing a virtual resource that operates at a fraction of the rate of physical resource shared by the group but the rate varies with time during execution. In a later section, we shall characterize the rate variation of each virtual resource by means of a delay bound D that specifies the maximum extra time the task group may have to wait in order to receive its fraction of the physical resource over any time interval starting from *any* point in time. This way, if we know that an event e will occur within x time units from another event e' assuming that the virtual resource operates at a uniform rate and event occurrence depends only on resource consumption (i.e., virtual time progresses uniformly), then e and e' will be apart by at most $x + D$ time units in real time. If infinite time-slicing is possible, the delay bound is zero. In general, the delay bound of a virtual resource will be task-group-specific. The characterization of virtual resource rate variation by means of the delay bound will allow us to better deal with more general types of timing constraints such as jitter. We call virtual resources whose rate of operation variation is bounded real-time virtual resources.

The rate variation and therefore the delay bound of a real-time virtual resource is in general a function of the scheduling policy used to allocate the shared physical resource among the task groups. One approach to construct real-time virtual resources that is especially amenable to delay bound determination is through temporal resource partitioning. In this approach, the second-level scheduler is responsible only for assigning partitions (collection of time slices) to the task groups and does not require information on the timing parameters of the individual tasks within each task group. The schedulability analysis of tasks on a partition depends only on the partition parameters. This enforces the desired separation of concerns; scheduling at the resource partition (task group) level and at the task level are isolated at run time. Resource partitioning is advocated in a system design concept arising in the avionics industry that is known as Integrated Modular Avionics (IMA) [1, 30]. In IMA, a single computer system with internal replication provides a common computing resource to numerous subsystems or functions. Currently, most avionics systems are implemented by a federated architecture whereby subsystems and functions are loosely coupled in order to minimize the fault propagation. One obvious disadvantage of the federated approach is the profligate usage of resources. The overall objective of IMA is to accomplish the same fault tolerance requirement as the federated approach and yet maximize resource utilization. A key idea of realizing this goal is to use temporal resource partitioning of the single computer system to ensure fault containment within each function which is inherent to the federated architecture that IMA aims at replacing.

We shall propose two resource partition models and investigate both task level and task group (resource partition) level scheduling problems as well as output jitter concerns. We assume that tasks are periodic and can start at any time, although the period can be interpreted as the minimum separation time in the sporadic task model without invalidating the results in this paper. Throughout the paper it is assumed that time values have the domain of the non-negative real numbers. All the results will still hold if the domain of time is the set of non-negative integers. Preemptive scheduling is assumed, i.e., a task executing on the shared resource can be interrupted at any instant in time, and its execution can be resumed later. Although a resource can be a processor, a communication bus, etc., we shall talk about a single processor as the resource to be shared.

Definition 1 *A task T is defined as (c, p), where c is the (worst case) execution time requirement, p is the period of the task.*

Even though we do not specify a per-period deadline explicitly, we shall define deadlines when they are relevant to the result in this paper.

Definition 2 *A task group τ is a collection of n tasks that are to be scheduled on a real-time virtual processor (a partition), $\tau = \{T_i = (c_i, p_i)\}_{i=1}^{n}$.*

We use the term task group to emphasize its difference from the term task set in that a task set is to be scheduled on a dedicated resource while a task group is scheduled on a partition of the shared physical resource.

The rest of this paper is organized as follows. Sections 2 defines the static partition model and Section 3 proposes a bounded-delay model. Each section first defines the model and derives some properties from the model. Both fixed priority scheduling and dynamic scheduling are applied to the model. The resource level scheduling is also discussed. In Section 4 we compare these two models and discuss the jitter problems. We review the related work in Section 5, and end wih the conclusion in Section 6.

2 The Static Resource Partition Model

In this section, we formalize the resource partition concept. Informally, a (temporal) partition is simply a collection of time intervals during which the physical resource is made available to the task group being scheduled on the partition. In this section, we investigate the problem of scheduling task groups for a given partition whose time intervals are explicitly specified by a list. By making use of the technique of supply functions, we shall analyze both fixed and dynamic priority schedulers with respect to this partition model. The schedulability results are obtained based on the key idea of the critical partition. Finally we discuss the (second-level) scheduling problem of the partitions themselves.

It will be seen from this section that the schedulability test for both fixed and dynamic priority assignment is comparable with traditional models in complexity. However, we no longer have the utilization bound of 1.0 for periodic tasks.

2.1 The Resource Partition

Definition 3 *A resource partition Π is a tuple (Γ, P), where Γ is an array of N time pairs $\{(S_1, E_1), (S_2, E_2), \ldots, (S_N, E_N)\}$ that satisfies $(0 \leq S_1 < E_1 < S_2 < E_2 < \ldots < S_N < E_N \leq P)$ for some $N \geq 1$, and P is the partition period. The physical resource is available to a task group executing on this partition only during time intervals $(S_i + j \times P, E_i + j \times P), 1 \leq i \leq N, j \geq 0$.*

We shall refer to the intervals where the processor is unavailable to a partition *blocking time* of the partition. In traditional models where resources are dedicated to a task group, there is no blocking time and we may consider this as a special case corresponding to the partition $\Pi = (\{(0, P)\}, P)$.

Example 1 $\Pi_1 = (\{1, 2), (4, 6)\}, 6)$ *is a resource partition whose period is 6 with available resource time from time 1 to time 2 and from time 4 to time 6 every period. See Figure 1.*

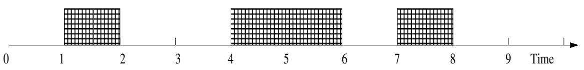


Figure 1. Timing Diagram of Partition Π_1

Definition 4 *We call a resource partition where $N = 1$ a Single Time Slot Periodic Partition (*STSPP*). A partition is otherwise a Multi Time Slot Periodic Partition (*MTSPP*).*

Definition 5 *The availability factor of a resource partition Π is $\alpha(\Pi) = (\sum_{i=1}^{n}(E_i - S_i))/P$.*

The availability factor of Π_1 in Example 1 is $\alpha(\Pi_1) = ((2-1) + (6-4))/6 = 0.5$.

Definition 6 *The Supply Function $S(t)$ of a partition Π is the total amount of time that is available in Π from time 0 to time t.*

From the definition we can easily prove some properties of $S(t)$.

- $S(0) = 0$
- $S(t)$ is a monotonically non-decreasing function for t $(t \geq 0)$.
- $S(u) - S(v) \leq u - v$ for $u > v \geq 0$.
- $S(t+P) - S(t) = S(P)$ $(t \geq 0)$ (Because the partition is periodic.)
- $S(t) = \lfloor \frac{t}{P} \rfloor \times S(P) + S(t - P \lfloor \frac{t}{P} \rfloor)$ for $t > 0$.

2.1.1 Fixed Priority Scheduling

Given a partition, we now analyze the schedulability problem of a task group that may execute only during the available time of the partition. For both STSPP and MTSPP, the classical utilization bound no longer holds, as can be seen by a contradiction. Suppose there exists a utilization bound U. Consider a task whose period P is equal to the longest blocking time of the partition, and whose execution time is smaller than $U \times P$. Obviously the task is not schedulable on the partition if it requests at the beginning of the longest blocking time, although its utilization factor is smaller than U. Therefore, there is no non-zero utilization bound.

The lack of a utilization bound leads us to reconsider the concept of critical instances. Recall that Liu and Layland [25] defined critical instances to be the time when the task is requested simultaneously with requests of all higher priority tasks. The essential idea of the definition is to construct a worst-case scenario. If the task system is schedulable in the worst case scenario it is definitely schedulable at any time. Therefore, we need to know what the worst case scenario is. An intuitive answer might be the longest blocking time slot. Although this is true for STSPP, it is not the case for MTSPP. For example, consider the schedule of tasks $T_1 = (1, 3)$ and $T_2 = (1, 4)$ in Π_1 in Example 1. The task relative deadlines are equal to the corresponding periods. The longest interval for which the resource is unavailable starts from time 2 and ends at time 4. Suppose the "critical instance" starts at time 2. The first requests of both tasks will finish before the deadlines, but the second request of T_2 misses the deadline, as shown in Figure 2. Hence, the longest blocking time slot is not necessarily the worst case. This is because the supply after the so defined "critical instance" is not necessarily larger than the supply inside the "critical instance". In general, we need to test for schedulability at more than one "critical instance".

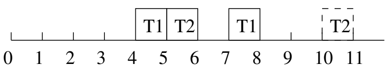


Figure 2. Example in Traditional Critical Instance Test

Definition 7 *We call $E_1, E_2, \ldots, E_N$ of a partition Π $(\{(S_1, E_1), (S_2, E_2), \ldots, (S_N, E_N)\}, P)$ interval-based critical points (*IBCP*s). If a task is requested simultaneously with all higher priority tasks at an* IBCP*, it is called an interval-based critical instance (*IBCI*).*

Theorem 1 *For fixed priority assignment and a task group whose relative deadlines are no more than the periods, a* *task is schedulable in a partition* Π *if and only if its first request is schedulable in all* IBCI*s.*

Proof. We only need to prove that if a task $T = \{c, p\}$ with relative deadline $d \leq p$ is unschedulable, it will fail on one of the IBCIs.

Let t to be the time T misses the deadline but all its previous requests are successfully scheduled. Searching from t backward, let t_0 be the first time when no outstanding execution time of all higher priority tasks and T exists. t_0 always exists because 0 or the finish time of T's previous request are such time points. Only the last request of T exists in (t_0, t) and only higher priority tasks and T are scheduled in (t_0, t). For any higher priority task, we move it backward so that the first request after t_0 now request at t_0. This makes T's situation worse, so T still misses deadline at t. Then we move T backward so that T requests at t_0. In the new schedule T will miss deadline at time $t_0 + d$.

- If t_0 is one of the IBCPs, T fails the corresponding IBCI.
- If $S_i + j \times P \leq t_0 < E_i + j \times P, 1 \leq i \leq N, j \geq 0$, we shift the task requests together so that all those requesting at t_0 now request at E_i. The resource availability between T's request and its deadline decreases by $(E_i + j \times P) - t_0$ in the front and increases by no more than $(E_i + j \times P) - t_0$ in the end. This is an IBCI at E_i and T is not schedulable.
- If t_0 is not in any of the intervals where the resource is available, we shift the task requests together so that all those requesting at t_0 now request at the closest point $E_j + j \times P$ satisfying $1 \leq j \leq N, j \geq 0$ before t_0. The resource availability between T's request and its deadline does not increases in the front and may decreases in the end. This is an IBCI at E_j and T is not schedulable.

So there is always an IBCI that T will fail if T is unschedulable on Π. ■

Following the response-time analysis method of the traditional models [2, 12], we can give the following algorithm as the schedulability test.

Algorithm 1
Test $T = (c, p)$ *with deadline* d. G *is the set of higher priority tasks:*

```
for (i = 0; i ≤ N; i++) /* each loop tests an IBCI */ {
  r = c;
  for (;;) {
    r' = c + Σ_{T_i∈G}(⌈r/p_i⌉c_i);
    r'' = x where r' == S(E_i + x) − S(E_i); }
    If (r'' > d) {deadline missed; return fail; }
    If (r'' == r) {deadline met for this IBCI; break; }
    r = r'';
  }
}
return success;
```

Also using similar analysis techniques for traditional task models, we can prove the following corollary from Theorem 1.

Corollary 1 *For the preemptive fixed priority scheduling discipline, a task group whose relative deadlines are no bigger than the periods is schedulable on a partition with the rate/deadline-monotonic priority assignment (*RMA/DMA*) if it is schedulable on the partition by some priority assignment.*

2.1.2 Dynamic Priority Scheduling

We now turn to the schedulability of task groups on partitions by using dynamic priority schedulers. It turns out that the earliest deadline first (EDF) scheduling algorithm is still optimal in this case. In other words, if there exists a schedule for a task group on a resource partition, the task group is also schedulable using EDF. Same applies to the least slack first scheduling (LSF) algorithm. We could apply the proof techniques in [25, 26] to prove the the next result.

Theorem 2 *If a task group* τ *is schedulable in partition* Π *by a scheduling policy, it is also schedulable by* EDF *or* LSF.

For a periodic task group whose relative deadlines are equal to the corresponding periods, the utilization bound of EDF is 1.0 if the resource is always available. If the group is scheduled on a resource partition, however, the simple bound no longer applies. In the following, we call the recurring instances of a periodic task the *jobs* of the task.

Definition 8 ([6]) *Let* T *be a task, and* t *a positive real number. The demand bound function* $\mathsf{dbf}(T, t)$ *denotes the maximum cumulative execution requirement of the jobs of* T *that have both arrival times and deadlines within any time interval of duration* t.

Theorem 3 *A task group* G *is infeasible on a partition* Π *if and only if* $\Sigma_{T \in G} \mathsf{dbf}(T, t) > S(t_0 + t) - S(t_0)$ *for some positive real numbers* t_0 *and* t.

The proof of Theorem 3 is not shown here as it is almost the same as that in [6] except that the resource is not always available. Theorem 3 is computationally difficult to convert into a feasibility test algorithm. We shall derive a better result next.

2.2 Critical Partition

The lack of critical instance and the complexity of EDF testing motivate this subsection, where we shall define the least supply function, the critical partition, critical instance, and a better EDF feasibility testing algorithm.

We shall allow a task to start issuing requests at any time. As such, we may without loss of generality assume that time 0 as the start time for each partition. For the same partition, we may get different representations of the partition by choosing different time instants at which to start making the resource available, but all of these representations are equivalent for the purpose of the following analysis.

2.2.1 Least Supply Function

Definition 9 *The Least Supply Function (*LSF*) $S^*(t)$ of a resource partition Π is the minimum of $(S(t+d) - S(d))$ where $t, d \geq 0$.*

Definition 10 *A Least Supply Time Interval (u, v) is an interval that satisfies $(S(v) - S(u)) = S^*(u - v)$.*

Intuitively, $S^*(t)$ is the smallest amount of resource time available to a partition in any interval of length of t.

Some relevant properties of $S^*(t)$ are:

- $S^*(P) = S(P)$

  Proof: By definition $S^*(P) = \min(S(t+P) - S(t)) = S(P)$ ■

- $S^*(t+a) - S^*(t) \geq S^*(a), t \geq 0, a \geq 0$

  Proof. Let $S^*(t+a) = S((t+a)+b) - S(b)$. We have $S^*(t+a) = (S(t+(a+b)) - S(a+b)) + (S(a+b) - S(b)) \geq S^*(t) + S^*(a)$. ■

- $S^*(t+P) - S^*(t) = S^*(P), t \geq 0$

  Proof. From previous property, we have $S^*(t+P) - S^*(t) \geq S^*(P)$. Suppose $S^*(t+P) - S^*(t) > S^*(P)$. Let $S^*(t) = S(t+b) - S(b)$. $S(b+t+P) - S(b) = S(b+t) + S(P) - S(b) = S^*(t) + S^*(P) < S^*(t+P)$. This contradicts the definition of $S^*(t+P)$. Therefore $S^*(t+P) - S^*(t) = S^*(P)$. ■

- $S^*(t)$ is a monotonically non-decreasing function for $(t \geq 0)$.

  Proof. Suppose there exist t_1, t_2, $t_1 < t_2$ and $S^*(t_1) > S^*(t_2)$. Let $S^*(t_2) = S(a+t_2) - S(a)$, then $S^*(t_2) = S(a+t_2) - S(a) > S(a+t_1) - S(a) \geq S^*(t_1)$. Contradiction. ■

- $S^*(t) = \lfloor \frac{t}{P} \rfloor \times S^*(P) + S(t - P\lfloor \frac{t}{P} \rfloor)$ for $t > 0$.

Starting from time 0, $S(t)$ is a step function with N steps every P time units and repeats every P time units; $S^*(t)$ is also a step function with at most $N \times N$ steps every P time units and repeats every P time units.

Note that $S^*(t)$ may be regarded as a special case of supply functions; all the properties of $S(t)$ also hold for $S^*(t)$. Next, we show how to compute the LSF for a partition.

Lemma 1 *Any time interval where a partition Π receives the least resource time has an equal amount of available time in an interval that starts with an* IBCP *point of* Π.

Proof. We can prove this by using a time-slice swapping argument. Given a least supply time interval (u, v) If u is an IBCP point obviously the lemma holds, otherwise let E_i denote the latest IBCP point before u, E_{i+1} the earliest point after u and S_{i+1} the starting point of the available time interval between E_i and E_{i+1}. There are two cases:

1. If $u < S_{i+1}$, then $S^*(v-u) = S(v) - S(u) = S(v) - S(E_i) \geq S(v - (u - E_i)) - S(E_i)$. Because (u, v) is an least supply time interval only the equal sign can be true. Therefore, (u, v) and $(E_i, (v - u + E_i))$ have the same amount of available time.

2. If $u \geq S_{i+1}$, then $S^*(v-u) = S(v) - S(u) = S(v) - (S(E_{i+1}) - (E_{i+1} - u)) = (S(v) + E_{i+1} - u) - S(E_{i+1}) \geq S(v + E_{i+1} - u) - S(E_{i+1})$. For the same reason as in 1, (u, v) and $(E_{i+1}, (v - u + E_{i+1}))$ have the same amount of available time. ■

To compute LSF, first locate all the IBCP points; then for each IBCP point compute S(t) from time 0 to time P taking the IBCP point as the starting point, i.e. time 0. For every t, the minimum of the supply functions S(t) so computed yields the $S^*(t)$ we need.

Figure 3 shows how to compute the $S^*(t)$ for Partition $\Pi_1 = (\{(1,2),(4,6)\}, 6)$ in Example 1. There are two IBCP points, time 0 and time 2. Therefore two supply functions are generated and are shown in the figure. The LSF is the bottom envelop of the two $S(t)$s. The algorithm to compute the LSF for a partition is attached in the appendix.

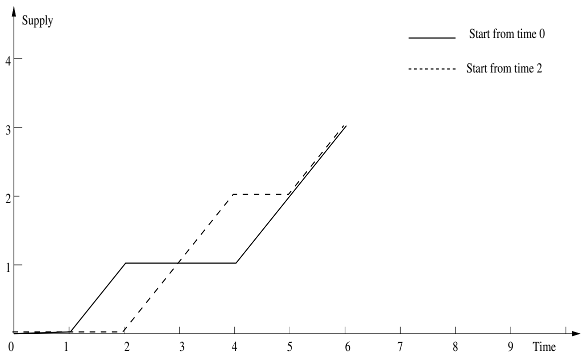


Figure 3. Computing Supply Function

2.2.2 Critical Partition

Definition 11 *A critical partition of a resource partition $\Pi = (\Gamma, P)$ is $\Pi^* = (\Gamma^*, P)$ where Γ^* has time pairs corresponding to the steps in $S^*(t)$ such that Π^*'s supply function equals $S^*(t)$ in $(0, P)$.*

Corollary 2 *Π^*'s supply function equals $S^*(t)$ for $t \geq 0$.*

The critical partition of Π_1 in Example 1 is $\Pi_1^* = (\{(2,3),(4,6)\}, 6)$.

Theorem 4 *A task group τ is feasible in a partition Π if and only if it is feasible in its critical partition Π^*.*

Proof: (i) If τ is infeasible in Π, then according to Theorem 3, there exist u and v such that $\mathsf{dbf}(u - v) > S(u) - S(v)$. Suppose π is the request pattern of τ satisfying the inequality. Let us discard all task requests before v and shift π backward so that v is aligned up at time 0, then try to schedule this request pattern on Π^*. The total request with deadlines before $u-v$ is $\mathsf{dbf}(u-v) > S(u)-S(v) > S^*(u-v)$, the available resource before $u - v$. So τ is infeasible in Π^*.

(ii) If τ is infeasible in Π^*, according to Theorem 3, there is u and v such that $\mathsf{dbf}(u - v) > S^*(u) - S^*(v)$. Suppose π is the request pattern of τ satisfying the inequality. Let's discard all task requests before v and shift π backward so that v is aligned up at time 0. The total request with deadlines before $u - v$ is $\mathsf{dbf}(u - v) > S^*(u) - S^*(v) > S^*(u - v)$, the available resource before $u - v$. Now let $S^*(u - v) = S((u - v) + a) - S(a)$. We shift π again from 0 to a and try to schedule π on Π. The request of π after a that must be finished before $(t - t_0) + a$ is $\mathsf{dbf}(u - v) > S^*(u - v) = S((u - v) + a) - S(a)$. So τ is infeasible in Π. ■

2.2.3 Fixed Priority Scheduling

Definition 12 *The critical instance of a task on partition Π is when it is requested simultaneously with all higher priority tasks at time 0 on the critical partition Π^*.*

Theorem 5 *Suppose the preemptive fixed priority scheduling policy is used to schedule a task group on a partition by some priority assignment where all deadlines are no bigger than the corresponding periods. If a task's first request is schedulable in a partition's critical instance, then the task is schedulable in the partition.*

Proof: We show that if a task is unschedulable, it will fail in the critical instance. If a task $T = (c, p)$ with deadline $d \leq p$ is unschedulable, it will fail in one of the IBCI. Let the IBCP to be E. We compare the resource supply between IBCI and the critical instance. By definition $S(E + x) - S(E) \geq S^*(x), 0 \leq x \leq d$. The resource supply in the critical instance always lacks behind that in the IBCI. So the schedule sequence of all tasks does not change in the critical instance. In other words, T could not be finished before d due to early resource supply before some higher priority task requests come in. So T will fail in the critical instance. ■

We may modify Algorithm 1 to test only the critical instance. Note that Theorem 5 is only a sufficiency test. There may be task groups that are schedulable but fail to pass the critical instance test of Algorithm 1.

Example 2 *Partition $\Pi_2 = (\{(1,2),(4,6),(7,8)\}, 8)$. Its critical partition is $\Pi_2^* = (\{(2,3),(4,5),(6,8)\}, 8)$. Task group $\{T_1 = (1,4), T_2 = (1,6)\}$ is schedulable on Π_2 with* RMA. *This can be checked with Algorithm 1. However, in the critical instance, T_2 misses deadline at 6. Figure 4 is the supply functions of* IBCI*s and the critical instance.*

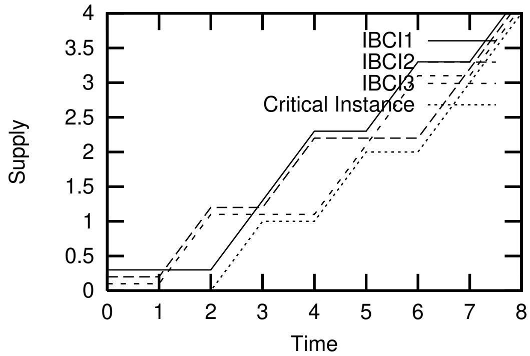


Figure 4. Critical Instance Test

2.2.4 Dynamic Priority Scheduling

From Theorems 3 we have the following corollary.

Corollary 3 *A task group G is infeasible on a partition Π if and only if $\Sigma_{T \in G} \mathsf{dbf}(T, t) > S^*(t)$ for some positive real number t.*

Since $S^*(t)$ is the supply function of the critical partition, we can devise a pseudo-polynomial feasibility test algorithm for task group scheduling on a partition similar to the ones developed for traditional feasibility test, e.g., in [29]. We shall not delve into the details here.

2.3 Resource Level Scheduling

The preceding schedulability results pertain to determining whether a task group can be scheduled on an explicitly given partition. Consistent with the tenets of an open system, the schedulability tests require only the attributes of the task group and the specification of the partition it runs on. More importantly, the second-level scheduler which implements (allocates resource time to) the partitions does not require information about the attributes of the task group running on the partitions, thus providing for a clean separation of concerns.

Given a static partition model where all the time intervals are defined, there is not much the run-time system can do other than shifting the partition as a whole along the time line, as tasks may start requesting at any time other than time 0. The static resource model is more motivated by the scenario where a single resource is already divided into a set of partitions and the goal is to schedule task groups in the given partitions. This is nonetheless an important technique for designing high-criticality applications since the static model is very amenable to timing correctness certification, and also because it gives the designer control over context switch overheads if the designer is also free to select the time intervals of a static partition.

We shall introduce in the next section a bounded-delay partition model that facilitates second-level scheduling when the time intervals of a partition are not explicitly specified.

3 The Bounded-Delay Resource Partition Model

In this section we shall first start with a few preliminary definitions then we shall define the bounded-delay resource partition model and prove a schedulability theorem.

Definition 13 *For any $v \geq 0$, $S'(v)$ is the smallest t such that $S(t) = v$.*

$S'(v)$ is similar to the definition of the inverse function of $S(t)$. However, because $S(t)$ is not a function with one-to-one mapping, we embed a minimization operation into the definition.

Definition 14 *The partition delay Δ of Partition Π is the smallest d so that for any t_0 and $v > 0$, $S'(S(t_0) + v) - (t_0 + v/\alpha(\Pi)) \leq d$.*

Definition 15 *Let h denote the execution rate of the resource where partition Π is implemented. The Normalized Execution of partition Π is an allocation of resource time to Π at a uniform, uninterrupted rate of $(\alpha(\Pi) \times h)$.*

Intuitively, Δ is the maximum delay of the available time interval of any length in the partition Π relative to its normalized execution regardless of the starting point. In the definition, t_0 is the starting point. $S'(S(t_0) + v)$ is the first time point that has accumulated v execution time units since time t_0. $(t_0 + v/\alpha(\Pi))$ is the time point that has accumulated v execution time units running on the normalized execution since t_0.

Definition 16 *A bounded delay resource partition Π is a tuple (α,Δ) where α is the availability factor of the partition and Δ is the partition delay.*

Note that the definition actually defines a set of partitions because there are many different partitions in the static partition model that may satisfy this requirement.

Theorem 6 *Given a task group τ and a bounded delay partition $\Pi = (\alpha, \lambda_n)$, let S_n denote a valid schedule of τ on the normalized execution of Π, S_p the schedule of τ on Partition Π according to the same execution order and amount as S_n. Also let λ denote the largest amount of time such that any job on S_n is completed at least λ time before its dealine. S_p is a valid schedule if and only if $\lambda \geq \lambda_n$.*

Proof: We show the necessity by contradiction.

We first construct a static partition $\Pi' = (\{(\lambda_n, P)\}, P)$ where $P = \frac{\lambda_n}{1-\alpha}$. Π' is an STSPP where the only time slot is at the end of the period. The partition delay is λ_n, and $\alpha(\Pi') = \frac{P-\lambda_n}{P} = \alpha$. So Π' is one of the partitions $\Pi(\alpha, \lambda_n)$. Note that any time point on Π''s normalized execution is mapped to Π' at a no-earlier time point.

Let us suppose $\lambda < \lambda_n$. According to the definition of λ there is at least one job J with deadline d_j that is completed on S_n at time $d_j - \lambda$ that is later than time $t_0 = d_j - \lambda_n$. During the construction of S_p we align up t_0 with the start time of one partition period of Π'. Therefore the execution part of J after t_0 on S_n will be mapped to the time slot starting from $t_0 + \lambda_n$ that is d_j on S_p, thus J misses its deadline. Therefore, S_p is invalid.

To show the sufficiency, we construct a mapping of the execution time from S_n to S_p. If we can find a mapping that satisfies the following conditions, we can guarantee that τ is schedulable on Π.

Condition 1: There is no time interval in S_p that is earlier than its corresponding interval in S_n. This condition guarantees that every job in the interval executed in S_n is also available to be executed in the corresponding interval in S_p. If this condition does not hold, it is possible that there exists a job which has been released to execute in S_n but not yet in S_p in the corresponding interval.

Condition 2: There is no time interval in S_p that is over λ_n time later than its corresponding interval in S_n. This condition guarantees that no job in S_p will miss its deadline since every job is finished λ_n time before its deadline in S_n.

The following procedure describes how we construct the mapping:

1. Take the interval from time 0 to time L where L is the least common multiple (LCM) of the periods of every task

and the period of the partition in the task group τ from both schedules.

2. Compute the supply functions of both schedules, let $S_n(t)$denote the supply function of S_n and $S_p(t)$ denote the supply function of S_p.

3. Find the maximum $S_p(t) - S_n(t)$ $(0 \le t < P)$ and let d denote the value. Note that we only need to compute the interval from time 0 up to the period of the partition.

4. Given a time interval $(t_1, t_2)(t_1 < t_2)$ in S_n, map it into $(S'_p(S_n(t_1) + d)$, $S'_p(S_n(t_2) + d))$ in S_p.

Let us show why this mapping satisfies the two conditions above.

Condition 1 requires $S'_p(S_n(t_1) + d) \ge t_1$ and $S'_p(S_n(t_2) + d) \ge t_2$. We show it by contradiction. Consider t_1 first and suppose $S'_p(S_n(t_1) + d) < t_1$, then we have $S_p(S'_p(S_n(t_1) + d)) < S_p(t_1)$ because $S_p(t)$ is a non-decreasing function. Therefore, $S_n(t_1) + d < S_p(t_1)$, then $d < S_p(t_1) - S_n(t_1)$ that contradicts with the definition of d. For the same reason, $S'_p(S_n(t_2) + d) \ge t_2$. Hence, Condition 1 is satisfied.

Condition 2 requires $S'_p(S_n(t_1) + d) \le t_1 + \lambda_n$ and $S'_p(S_n(t_2) + d) \le t_2 + \lambda_n$. Let us consider t_1 first. We have $S_p(t) - S_n(t) \le d$ and $S_n(t) = t \times \alpha(\Pi)$. Let t' denote the largest time point earlier than t_1 such that $S_p(t') - S_n(t') = d$. From the definition of partition delay, we have $S'_p(S_p(t') + v) - (t' + v/\alpha(\Pi)) \le \lambda_n$, let $v = S_n(t_1) - S_n(t')$ then we have $S'_p(S_p(t') + S_n(t_1) - S_n(t')) - (t' + (S_n(t_1) - S_n(t'))/\alpha(\Pi)) \le \lambda_n$, $S'_p(S_n(t_1) + S_p(t') - S_n(t')) - t_1 \le \lambda_n$. Therefore $S'_p(S_n(t_1) + d) \le t_1 + \lambda_n$. For the same reason $S'_p(S_n(t_2) + d) \le t_2 + \lambda_n$, thus Condition 2 is satisfied.

Overall, the interval $(0, L)$ could be mapped to $(S'(d), L + S'(d))$. Furthermore, any hyperperiod interval $(n * L, (n+1) * L)$ could be mapped to $(n * L + S'(d), (n + 1) * L + S'(d))$. Therefore, the sufficiency is proven. ■

Figure 5 shows the procedure for performing the mapping from S_n to S_p.

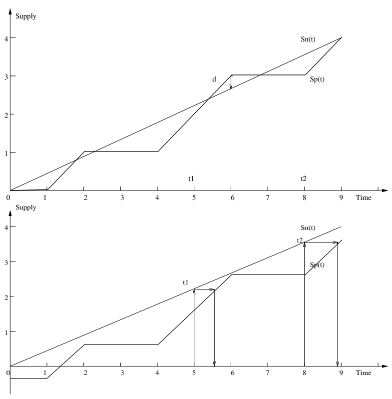


Figure 5. Mapping from S_n to S_p

Note that in the proof above we first schedule a task group in either normalized execution or on a partition. Then the resulting schedule is fixed and mapped to the other execution as if the task group were still scheduled in the original execution. However, this may not preserve the scheduling policy used in the original schedule. Regardless of the mode of execution the task group is scheduled on, the release time and the deadline of every job remain the same. For example, suppose part of a lower priority job J_1 is scheduled before J_2 on the normalized execution simply because J_2 is not started yet. If that J_1' execution time is mapped to a time after J_2's start time in the partition, then in the partition lower priority job J_1 is scheduled instead of higher priority job J_2.

Still, Theorem 6 provides a practical way to schedule a task group on a partition. If we could find a schedule on the normalized execution and the smallest λ is no less than λ_n, we could use this schedule on the partition and be guaranteed that no deadline will be missed on the partition. The schedule on the normalized execution is the same as the traditional task schedule, for which there are many known techniques.

Since EDF is an optimal scheduling policy with or without a partition, we have:

Corollary 4 *A task group is feasible on a partition $\Pi = (\alpha, \lambda_n)$ if and only if its* EDF *schedule on the normalized execution has the property that all requests finish at least λ_n before the deadlines.*

We point out that determing whether the smallest λ is no less than λ_n on the normalized execution is equivalent to testing if the task set is schedulable on the normalized execution where a task (c, p)'s deadline is set to be $p - \lambda_n$.

Definition 17 *The jitter-tolerance of a periodic task system τ is defined to be the largest Δ such that even if every job is released Δ time units late, all the tasks in τ should still be able to meet their deadlines.*

Let S_{EDF} denote the EDF schedule of a task group τ in a normalized execution. Let λ_{EDF} denote the largest λ such that all the tasks in the task group complete execution at least λ before their deadlines in S_{EDF}. Then the jitter-tolerance of τ is exactly equal to λ_{EDF}. A task system τ is feasible in a partition Π (α, Δ) if $U(\tau) \le \alpha$ and the jitter-tolerance of τ is no bigger than Δ.

3.1 Resource Level Scheduling

Given the resource requirement of (α_k, Δ_k) for each partition S_k, a schedule must be constructed at the resource level. Note that the pair of parameters (α_k, Δ_k) indicates only that the partition must receive an α_k amount of processor capacity with the partition delay no greater than Δ.

It does not impose any restriction on the execution time and period. This property makes the construction of the schedule extremely flexible. In this subsection we suggest both static and dynamic scheduling algorithms for partitions.

1. Static schedule:

In this approach, the resource schedules every partition cyclically with period equal to the minimum of $T_k = (\Delta_k/(1-\alpha_k))$ and each partition is allocated an amount of processor capacity that is proportional to α_k. If the T_k of the partitions are substantially different, we may adjust them conservatively to form a harmonic chain in which T_j is a multiple of T_i, if $T_i < T_j$ for all i and j. This way, the static resource schedule is repeated every major cycle which has length equal to the maximum of T_k. Each major cycle is further divided into several minor cycles with a length equal to the minimum of T_k. This would reduce the number of context switches substantially[17, 15].

2. Dynamic schedule:

In this approach, the resource schedules every partition using the Earliest Deadline Schedule with a period of $T_k/2$ and an execution time of $T_k \times \alpha_k/2$. The division of the period in the static schedule by 2 is because we need to guarantee the maximum separation of two executions in two continuous periods to be less than the partition delay Δ so as to meet the requirement of partition delay.

Another way to achieve this is to separate the deadline and the period of the EDF scheduling. The period of a partition is assigned to be more than $T_k/2$ while its corresponding relative deadline is assigned to be less than $T_k/2$. However, the sum of the period and relative deadline is always equal to T_k.[33]

4 Discussion

From the software engineering point of view, the static partition model and the bounded-delay partition model are different ways to implement the open system approach. The main difference is in the way the partition is specified. The static partition model is explicit but as an interface specification, it is more cumbersome and less flexible than the bounded-delay partition model. However, a higher processor utilization may be possible if the static partitions can be customized to the timing attributes of the tasks in the task groups.

The two models are compatible in the sense that one can be converted to another. From the static model to the bounded-delay model, we can first compute the critical partition, and then derive the maximum delay Δ from its LSF and hence $(\alpha(\Pi), \Delta)$, the representation of bounded-delay model. In the other direction, there is an infinite number of partitions in the static model that may correspond to one partition (α, Δ) in the bounded-delay model as long as the supply function of the critical partition falls between $\alpha \times t$ and $\alpha \times (t - \Delta)$. As a matter of fact, the bounded-delay model does not even require the partition to be periodic.

One important issue during design is jitter concern. Jitter is the variation between the inter-arrival or completion times (called input jitter and output jitter respectively) of successive jobs of the same task. Besides meeting all the deadlines, a good scheduling algorithm is also supposed to minimize output jitter. The predictability in both models may be exploited to significantly minimize the overall jitter. Take the bounded-delay model as an example. Given an output jitter requirement of J for a task group τ in partition (α, Δ), we may schedule the task group using normalized execution and obtain the jitter tolerance λ. The jitter requirement is met as long as $\lambda + \Delta \leq J$.

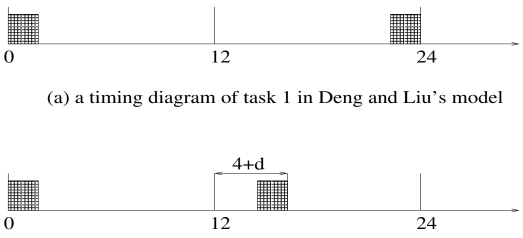
5 Related Work

The open system environment first proposed by Deng and Liu [10] allows real-time and non-real-time tasks not only to coexist but also to be able to join and leave the system dynamically. Therefore, the admission test on a real-time task needs to be independent of any other task in the system and a global schedulability analysis is out of the question. This concept was first discussed based on an EDF kernel scheduler and was later extended to the fixed priority scheduler as kernel scheduler[14]. It was further extended to parallel and distributed systems [13]. We take a broader approach in that we do not base our kernel scheduler on any particular scheduling policy; we start out with descriptions of a partition and we investigate whether application-specific task models such as the Liu and Layland periodic task systems can be scheduled on a partition.

Because in an open real-time environment the parameters of real-time tasks are no longer required to be known *a priori*, efficient online scheduling algorithms are needed [3, 32]. Also needed are practical mechanisms to provide isolation among tasks. One interesting approach is to assign each task a server with certain parameters [4, 22]. However, the interaction between tasks and the higher-level scheduler may increase the unpredictability in task execution and hence make other requirements such as output jitter bound difficult to realize. We believe that a clean separation between the scheduling of tasks within partitions and scheduling partitions on resources is more consistent with the tenet of open system environment. To wit, even if the application task groups are not all specified in one common system model such as Liu and Layland periodic tasks, our partition models can still be used. We only need to figure out the schedulability conditions of the new system model on partitions. The effect of the partition scheduling on task group scheduling is captured by the partition parameter Δ in the bounded-delay model.

Recently, [24] proposed a framework for achieving inter-application isolation. In [24] an PShED (Processor Sharing with Earliest Deadlines First) algorithm is used to schedule partitions (called servers in [24]) in order to isolate the problem of scheduling tasks within a partition. This approach has the nice property that the task scheduling within a partition is the same as traditional task scheduling. However, this property totally depends on the PShED algorithm, which is the only way a partition could be scheduled. In addition, the dynamic deadlines of each partition, which is used to decide which partition to schedule, depends on the particularity of the tasks within the partition. Because of the inter-relations between how the partitions are scheduled and how the tasks are scheduled within a partition, jitter is hard to analyzed in this approach. Finally, this approach may not be able to handle resource-specific constraints such as the rigidity of time slots in a communication bus.

To highlight the better predictability of the static partition model, consider the following example.



(a) a timing diagram of task 1 in Deng and Liu's model

(b) a timing diagram of task 1 in this paper

Figure 6. Comparison of Deng and Liu's model and this paper's model

Suppose tasks $T_1 = (1, 12)$ and $T_2 = (1, 4)$ are scheduled using the earliest-deadline-first scheduler on a partition with capacity of 1/2. As shown in Figure 6, in the model of [10] when other partitions are fully loaded the longest response time for T_1 could get close to 12. Since the shortest response time could be only 1 the execution consistency of T_2 is not that desirable which might be crucial for some tasks with tight jitter requirements. In contrast the bounded-delay partition model bounds the relative delay of partitions comparing with traditional environment. Hence, the two tasks could be assigned to a partition so that the longest response time of T_1 is bounded by a value that is sufficiently close to 4 which is the longest response time of T_1 as they are running on a virtual CPU with 1/2 speed of the original one. Once the parameters of a partition are determined the longest response time of a task inside this partition will be affected by only other tasks in the same partition. Hence better isolation among partitions is accomplished.

Finally, our work also differentiates from Proportional Share in [31]. The lag in Proportional Share holds only for intervals starting from the same time point while in this paper the partition delay applies to any interval regardless of the starting point. This difference is crucial since the partition delays are most useful for bounding the separation between event pairs.

Compared with application architectural concepts such as IMA, the work in this paper provides the scheduling-theoretic foundation for those architectures. It has been pointed out that there are some significant issues that remain unsolved in the resource partition problem. First, IMA was found to have a large amount of output jitter [1]. Because the available time of a partition cannot in general be evenly distributed the completion time of a certain job of a task is affected not only by outstanding jobs of other tasks but also by the fluctuation of the partition. Second, IMA has been considered only for usage with STSPP [17]. In STSPP partitions have only one continuous time slot within each period and this need not be the case. The results in this paper provide answers to some of these issues.

6 Conclusion

In this paper, we propose two resource partition models. We analyze them with respect to the schedulability of both the preemptive fixed priority scheduling and dynamic priority scheduling policies. We also discuss the second-level scheduling of partitions and the bounding of jitter in our models.

Highlights of the results in this paper are:

- We proposed an approach to open system environment with a clean separation of concerns between task group scheduling and partition scheduling. Task group scheduling depends only on the partition parameters and is independent of how the partitions are scheduled.

- Both fixed and dynamic priority schedulability test within a static partition are analyzed and proven to be comparable with the traditional schedulability test.

- The bounded-delay partition model gives more flexibility on partition level schedule. Scheduling of a real-time task group in a bounded-delay partition can be reduced to traditional scheduling on a normalized execution of the partition with preperiod deadlines. Guaranteed jitter can be achieved with the bounded-delay partition model.

- We also discussed the partition level scheduling for both models.

There are still many open issues to be investigated. For example, our jitter bound is not exact and tighter bounds may be possible. The possibility of interaction between tasks from different partitions may be pursued if the partition independence is not violated. Utilization bounds may be possible under some conditions.

References

[1] N. Audsley and A. Wellings. Analysing apex applications. In *IEEE Real-Time Systems Symposium*, pages 39–44, December 1996.

[2] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *8th IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[3] S. Baruah. Overload tolerance for single-processor workloads. In *Real-Time Technology and Applications Symposium*, pages 2–11, 1998.

[4] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari. Scheduling periodic task systems to minimize output jitter. In *The 6th International Conference on Real-Time Computing Systems and Applications*, 1999.

[5] S. Baruah and S. Lin. Improved scheduling of generalized pinwheel task systems. In *The 4th International Workshop on Real-Time Computing Systems and Applications*, pages 73–79, 1997.

[6] S. K. Baruah, D. Chen, S. Gorinsky, and A. K. Mok. Generalized multiframe tasks. *Real-Time Systems Journal*, 17(1):5–22, July 1999.

[7] S. K. Baruah, D. Chen, and A. K. Mok. Jitter concerns in periodic task systems. In *IEEE Real-Time Systems Symposium*, 1997.

[8] M. Chan and F. Chin. General schedulers for the pinwheel problem based on double-integer reduction. *IEEE Transactions on Computers*, 41(6):755–768, June 1992.

[9] D. Chen. *Real-Time Data Management in the Distributed Environment*. PhD thesis, The University of Texas at Austin, 1999.

[10] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium*, pages 308–319, December 1997.

[11] R. Holte, A. Mok, L. Rosier, I. Tulchinsky, and D. Varvel. The pinwheel: A real-time scheduling problem. In *22th Hawaii International Conference on System Sciences*, January 1989.

[12] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, October 1986.

[13] T. Kuo, K. Lin, and Y. Wang. An open real-time environment for parallel and distributed systems. In *20th International Conference on Distributed Computing Systems*, pages 206–213, 2000.

[14] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open system architecture for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 256–267, 1999.

[15] T. W. Kuo and A. K. Mok. Load adjustment in adaptive real-time systems. In *IEEE Real-Time Systems Symposium*, pages 160–170, 1991.

[16] J. L. L. Zhang and Z. Deng. Hierarchical scheduling of periodic messages in open system. In *IEEE Real-Time Systems Symposium*, pages 350–359, December 1999.

[17] Y. Lee, D. Kim, M. Younis, and J. Zhou. Partition scheduling in apex runtime environment for embedded avionics software. In *The 5th International Conference on Real-Time Computing Systems and Applications*, pages 103–109, 1998.

[18] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *IEEE Real-Time Systems Symposium*, December 1990.

[19] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm - exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, December 1989.

[20] J. Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

[21] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[22] G. Lipari and S. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Real-Time Technology and Applications Symposium*, pages 166–175, December 2000.

[23] G. Lipari and G. Buttazzo. Scheduling real-time multi-task applications in an open system. In *Euromicro Conference on Real-Time Systems*, pages 234–241, June 1999.

[24] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. In *IEEE Real-Time Systems Symposium*, pages 217–226, 2000.

[25] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1), January 1973.

[26] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, MIT, 1983.

[27] A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transaction on Software Engineering*, 1997.

[28] P. Richardson and S. Sarkar. Adaptive scheduling: Overload scheduling for mission critical systems. In *Real-Time Technology and Applications Symposium*, pages 14–23, December 1999.

[29] I. Ripoll, A. Crespo, and A. K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11:19–39, 1996.

[30] J. Rushby. *Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance*. NASA Contractor Report 209347. SRI International, Menlo Park, CA, 1999.

[31] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE Real-Time Systems Symposium*, pages 288–299, 1996.

[32] Y. L. T. Kuo and K. Lin. Efficient on-line schedulability tests for priority driven real-time systems. In *Real-Time Technology and Applications Symposium*, pages 4–13, 2000.

[33] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley. Scheduling transactions with temporal constraints: exploiting data semantics. In *IEEE Real-Time Systems Symposium*, pages 240–251, 1996.

[34] L. Zhou and K. Shin. Rate-monotonic scheduling in the presence of timing unpredictability. In *Real-Time Technology and Applications Symposium*, pages 22–27, December 1998.

	A=0 B=0	A=0 B=1	A=1 B=0	A=1 B=1
$V_a > V_b$	R	if($t_i - t + V_b) > V_a$ R[t,t+$V_a - V_b$],N[t+$V_a - V_b$,t_i] else R	R	R
$V_a \leq V_b$	N	N	if($t_i - t + V_a) > V_b$ N[t,t+$V_b - V_a$],R[t+$V_b - V_a$,t_i] else N	N

Table 1. Computing Table

Appendix

Algorithm to get $S^*(t)$: We generate the critical pattern, from which $S^*(t)$ can be easily obtained.

Input: $W = (\{(S_0, E_0), (S_1, E_1), \ldots, (S_N, E_N)\}, P)$

Calculation_of_Critical_Partition $\{$
  $\delta \leftarrow P - E_N$; // offset
  // Align to the first IBCP point.
  $W \leftarrow ((S_0 + \delta, E_0 + \delta), (S_1 + \delta, E_1 + \delta), \ldots, (S_N + \delta, E_N + \delta), P)$;
  $V \leftarrow W$;
  for $(i = 1; i \leq N; i++)$ $\{$
    $V \leftarrow$ ShiftToNextIBCP(V);
    Merge(W, V);
  $\}$
  return W;
$\}$

Merge(A, B) $\{$
// assume A is the generated partition so far, B is a partition based on an IBCP point.
  $T \leftarrow$ Merge all the time points of A and B;
  $V_a \leftarrow 0$;
  $V_b \leftarrow 0$;
  $t \leftarrow 0$;
  Traverse $T\{$
    $t_i \leftarrow$ next element of T;
// Execute the update according to A is increasing or not and B is increasing or not.

Update values according to Table 1 where R means Replace $[t, t_i]$ in A with B and N means nothing need to be done.

    if(A=1) $V_a += t_i - t$;
    if(B=1) $V_b += t_i - t$;
    if(t_i belongs to A) A=A xor A;
    B=B xor B;
  $\}$
  return T;
$\}$

The time complexity of the algorithm is $O(N^2)$.