Partition Scheduling In APEX Runtime Environment for Embedded Avionics Software

Yann-Hang Lee and Daeyoung Kim
Real Time Systems Research Laboratory
CISE Department, University of Florida
Gainesville, FL 32611, USA
{yhlee, dkim}@cise.ufl.edu

Mohamed Younis and Jeff Zhou
Advanced System Technology Group
AlliedSignal
Columbia, MD 21045, USA
{younis, zhou}@batc.allied.com

Abstract

Advances in the computer technology encouraged the avionics industry to replace the federated design of control units with an integrated suite of control modules that share the computing resources. The new approach, which is called integrated modular avionics (IMA), can achieve substantial cost reduction in the development, operation and maintenance of airplanes. A set of guidelines has been developed by the avionics industry to facilitate the development and certification of integrated systems. Among them, a software architecture is recommended to address real-time and fault-tolerance requirements. According to the architecture, applications are classified into partitions supervised by an operating system executive. A general-purpose application/executive (APEX) interface is defined, which identifies the minimum functionality provided to the application software of an IMA system. To support the temporal partitioning between applications, APEX interface requires a deterministic cyclic scheduling of partitions at the O/S level and a fixed priority scheduling among processes within each partition. In this paper, we propose a scheduling scheme for partitions in APEX. The scheme determines the frequency that each partition must be invoked and the assignment of processor capacity on every invocation. Then, a cyclic schedule at the O/S level can be constructed and all processes within each partition can meet their deadline requirements.

1. Introduction

Advancements in technology have enabled the avionics industry to develop new design concepts, which result in highly integrated software-controlled digital avionics. The new approach, referred to as Integrated Modular Avionics (IMA), introduces methods that can achieve high levels of reusability and cost effectiveness compared to the existing federated implementations of avionics [2]. The IMA approach, based on the concept of partitioning, utilizes standardized modules in building functional components of avionics systems. While integration enables resource sharing, some boundaries need to be enforced to maintain system predictability and to prevent bringing down the whole system because of a failure of a single function, either due to generic faults or due to missed deadlines.

Failure containment is very crucial for the integrated environment to guarantee that a faulty component cannot cause other components to fail and to risk generating a total system failure. For instance, in a passage jet, a miss of task deadlines in the entertainment subsystem must not negatively influence any critical flight control subsystems such as the flight manager. An IMA system is called strongly partitioned if the boundaries between the integrated functions are clearly defined so that a faulty function cannot interfere with or cause a failure in any other functions. Strong functional partitioning facilitates integration, validation, and certification. Following the IMA guidelines, the cost of both development and maintenance is expected to decrease because of mass production of the building blocks, lower levels of spares, and reduced certification costs.

A software architecture is recommended among the IMA guidelines to address real-time and fault-tolerance constraints for the integrated environment [4]. In the architecture, applications are classified into partitions supervised by an operating system executive. Within each partition, a partition executive manages application tasks and intra-partition task communication. To communicate across partition boundaries, a message mechanism provided by the operating system executive must be invoked. The operating system executive manages processor sharing among partitions. A general-purpose application/executive (APEX) interface is defined in [4], which identifies the minimum functionality provided to the application software of an IMA

system. To support the temporal partitioning between applications, APEX interface requires a deterministic cyclic scheduling of partitions at the operating system level and a fixed priority scheduling among tasks within each partition. Thus, the tasks in one partition can only be executed during the fixed partition window allocated to the partition and a task overrun cannot cross the partition window boundaries. This scheduling approach not only restrains a task failure within the partition, but also facilitates system upgrade and the integration of additional functions without the need to reconfigure the whole system.

Apparently, task execution in each application partition is affected by the two-level scheduling structure of APEX that consists of a low-level cyclic schedule at the operating system level and a high level fixed priority schedule at partition executive level. A different two-level hierarchical scheduling scheme was proposed by Deng and Liu in [7]. The scheme allows real- time applications to share resources in an open environment. The scheduling structure has an earliest-deadline-first (EDF) scheduling at the operating system level. The second level scheduling within each application can be either time-driven or priority-driven. For acceptance test and admission of a new application, the scheme analyzes the application schedulability at a slow processor. Then, the server size is determined and server deadline of the job at the head of ready queue is set at run-time. Since the scheme does not rely on fixed allocation of processor time or fine-grain time slicing, it can support various types of applications, such as release time jitters, non-predictable scheduling instances, and stringent timing requirements.

In this paper, we are looking into the single processor scheduling issues for partitions in APEX environment. In [6], Audsley and Wellings discussed the characteristics of avionics applications under the APEX interface. They provide a recurrent solution to analyze task response time in an application domain and show that there is a potential for a large amount of release jitter. However, the paper does not address the issues of constructing cyclic schedules at the operating system level. To remedy the problem, our first step is to establish scheduling requirements for the lower level cyclic schedules such that task schedulability under the given high level fixed priority schedules in each partition can be ensured. The approach we adopt is similar to the one in [7] of comparing the task execution in APEX environment with that at a slow processor. The cyclic schedule then tries to allocate partition execution intervals by stealing task inactivity periods. This stealing approach resembles the slack stealer for scheduling soft-aperiodic tasks in fixed priority systems [9]. Once the schedulability requirements are obtained, suitable cyclic schedules can be constructed. Following the partitioning concept of IMA, the operating system level cyclic schedule is flexible to support system upgrade and integration. It is designed in a way that no complete revision of scheduling algorithms is required when the workload or application tasks in one partition are modified.

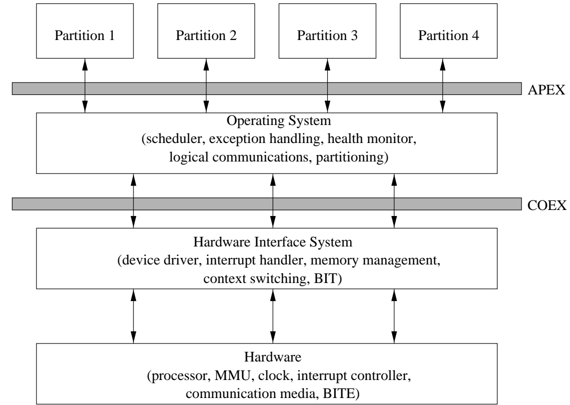


Figure 1. APEX software architecture

In the following section, we give a brief overview of the APEX software architecture on which applications are executed in separate partitions. In Section 3, we focus on the two-level scheduling mechanism and give the main theorem on the schedulability requirements. A numerical example is presented to illustrate the schedulability requirements with respect to various task characteristics. Then, in the same section, we show how to use the requirements to construct the cyclic schedules in the operating system level. Finally, a short conclusion follows in Section 4.

2. APEX Architecture: An Overview

As shown in Figure 1, the IMA software architecture consists of hardware interface system, operating system, and application software [4]. The hardware interface system is comprised of device driver, interrupt handler, memory management unit, and context switch function. Combining with the hardware in a cabinet, they form a core module to support operating system and application execution via the COEX (core-executive) interface. The operating system executive implements a set of services defined in the APEX (application-executive) interface such that application software can use the system resource and control the scheduling and communication between its internal processing tasks.

One of the objectives to be achieved with the APEX interface is portability. With the standardized APEX interface, application software can be developed without regard to hardware implementations and configurations. Also, a hardware change can be transparent to application software. The other important objective is to set up an execution environment such that the applications of multiple criticalities can be integrated. This requires a robust partition such that

an application cannot affect any other applications in terms of the content in their address spaces and the allocated execution times.

In order to support the spatial partition between applications, the tasks of an application are running in a predetermined area of memory. Any memory accesses (at least, write access) outside of a partition's defined area are prohibited by memory management unit. For temporal partition, the operating system maintains a major time frame during which each partition is activated in one or more scheduled partition windows. In other word, through a predefined configuration table of partition windows, the APEX impose a cyclic scheduling to ensure that each partition receives an fixed amount of processing time. This arrangement of a cyclic schedule can also allow message-driven executions. The IMA architecture assumes the existence of the ARINC 659 backplane bus [3] and the ARINC 629 data bus [1] for communications between core modules in a cabinet and between cabinets. Since both buses provide periodic message windows, message transmissions and partition activation windows can be synchronized to reduce end-to-end delays.

Within each application partition, the basic execution unit is a process or task. Tasks in a partition share the resources allocated to the partition and can be executed concurrently with other tasks of the same partition. A task comprises the executable program, data and stack areas, process state, and entry point. Its attributes also include base and current priorities, time capacity, and deadline. The scheduling algorithm within each partition is based on priority preemption. When the operating system determines to initiate a new partition window according to it cyclic schedule, the executing task in the old partition window is preempted. On the other hand, in the newly activated partition, the ready task with the highest current priority among those in the partition is selected to run. In addition, the operating system provides messaging mechanisms for inter-partition communication, and time management mechanisms to release periodic tasks and to detect missed deadlines.

3. Partition Scheduling in APEX Architecture

In this section, we consider the scheduling requirements for a partition server, S_k, which executes the tasks of an application partition A_k according to a fixed priority preemptive scheduling algorithm and shares the processing capacity with other partition servers in the operating system level. Let the application A_k consist of $\tau_1, \tau_2, \ldots, \tau_n$ tasks . Each task τ_i is invoked periodically with a period T_i and takes a worst-case execution time (WCET) C_i. Thus, the total processor utilization demanded by the application is $\rho_k = \sum_{i=1}^{n} \frac{C_i}{T_i}$. Also, upon each invocation, the task τ_i must be completed before its deadline period D_i, where $C_i \leq D_i \leq T_i$.

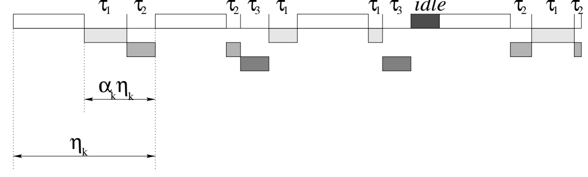


Figure 2. Task/partition scheduling in APEX

At the system level, the partition server S_k is scheduled periodically with a fixed period. We denote this period as the *partition cycle*, η_k. For each partition cycle, the server can execute the tasks in A_k during an interval $\alpha_k \eta_k$ where $\alpha_k \leq 1$ and is called *partition capacity*. For the remaining interval of $(1 - \alpha_k)\eta_k$, the server is blocked. It is our objective to find a pair of parameters α_k and η_k such that the tasks of A_k meet their deadlines. In Figure 2, we show an example execution sequence of a partition that consists of three tasks. During each partition cycle, the tasks are scheduled to be executed for a period of $\alpha_k \eta_k$. If there is no active task in the partition, the processor is idle and cannot run any active tasks from other partitions.

3.1. Partition Schedulability Requirements

As required in the APEX specification [4], the processes (or tasks) of each partition are running under a fixed priority preemptive scheduling. Suppose that there are n tasks in A_k listed in priority ordering $\tau_1 < \tau_2 < \cdots < \tau_n$ where τ_1 has the highest priority and τ_n is the lowest. In order to evaluate the schedulability of the partition server S_k, let's consider the case that A_k is executed at a dedicated processor of speed α_k, normalized with respect to the processing speed of S_k. Based on the necessary and sufficient condition of schedulability in [10, 8], task τ_i is schedulable if there exists a $t \in H_i = \{lT_j | j = 1, 2, \ldots, i; l = 1, 2, \ldots, \lfloor D_i/T_j \rfloor\} \cup \{D_i\}$, such that

$$W_i(\alpha_k, t) = \sum_{j=1}^{i} \frac{C_j}{\alpha_k} \lceil \frac{t}{T_j} \rceil \leq t.$$

The expression $W_i(\alpha_k, t)$ indicates the worst cumulative execution demand on the processor made by the tasks with a priority higher than and equal to τ_i during the interval $[0, t]$. We now define $B_i(\alpha_k) = \mathbf{max}_{t \in H_i}\{t - W_i(\alpha_k, t)\}$ and $B_0(\alpha_k) = \mathbf{min}_{i=1,2,\ldots,n} B_i(\alpha_k)$. Note that, when τ_i is schedulable, $B_i(\alpha_k)$ represent the total period in the interval $[0, D_i]$ that the processor is not running any tasks with a priority higher than or equal to that of τ_i. It is equivalent to the level-i inactivity period in the interval $[0, D_i]$ [9].

By comparing the task executions at server S_k and at a dedicated processor of speed α_k, we can obtain the following theorem:

Theorem 1 *The application A_k is schedulable at server S_k that is with a partition cycle η_k and a partition capacity α_k, if*

a) A_k is schedulable at a dedicated processor of speed α_k, and

b) $\eta_k \leq B_0(\alpha_k)/(1-\alpha_k)$

Proof: The task execution at server S_k can be modeled by tasks $\tau_1, \tau_2, \ldots, \tau_n$ of A_k and an extra task τ_0 that is invoked every period η_k and has an execution time $C_0 = (1-\alpha_k)\eta_k$. The extra task τ_0 is assigned with the highest priority and can preempt other tasks. We need to show that, given the two conditions, any task τ_i of A_k can meet its deadline even if there are preemptions caused by the invocations of task τ_0. According to the schedulability analysis in [10, 8], task τ_i is schedulable at server S_k if there is a $t \in H_i \cup G_i$, such that

$$\sum_{j=1}^{i} C_j \lceil \frac{t}{T_j} \rceil + C_0 \lceil \frac{t}{\eta_k} \rceil \leq t$$

, where $G_i = \{l\eta_k | l = 1, 2, \ldots, \lfloor D_i/\eta_k \rfloor\}$.

If τ_i is schedulable on a processor of speed α_k, there exists a $t_i^* \in H_i$ such that $B_i(\alpha_k) = t_i^* - W_i(\alpha_k, t_i^*) \geq B_0(\alpha_k) \geq 0$ for all $i = 1, 2, \ldots, n$. Note that $W_i(\alpha_k, t_i)$ is a non-decreasing function of t_i. Assume that $t_i^* = m\eta_k + \delta$, where $\delta < \eta_k$. If $\delta \geq B_0(\alpha_k)$,

$$\begin{aligned}
\sum_{j=1}^{i} C_j \lceil \frac{t_i^*}{T_j} \rceil + C_0 \lceil \frac{t_i^*}{\eta_k} \rceil &= \alpha_k W_i(\alpha_k, t_i^*) + (m+1)C_0 \\
&\leq \alpha_k(t_i^* - B_0(\alpha_k)) + (m+1)C_0 \\
&= \alpha_k(t_i^* - B_0(\alpha_k)) + (m+1)(1-\alpha_k)\eta_k \\
&\leq \alpha_k(t_i^* - B_0(\alpha_k)) + (1-\alpha_k)(t_i^* - \delta) + B_0(\alpha_k) \\
&= t_i^* + (1-\alpha_k)(B_0(\alpha_k) - \delta) \\
&\leq t_i^*
\end{aligned}$$

The above inequality implies that all tasks τ_i are schedulable at server S_k. On the other hand, if $\delta < B_0(\alpha_k)$, then, at $t_i^{'} = m\eta_k < t_i^*$, we have

$$\begin{aligned}
\sum_{j=1}^{i} C_j \lceil \frac{t_i^{'}}{T_j} \rceil + C_0 \lceil \frac{t_i^{'}}{\eta_k} \rceil &\leq \alpha_k(t_i^* - B_0(\alpha)) + mC_0 \\
&\leq \alpha_k(t_i^* - \delta) + m(1-\alpha_k)\eta_k \\
&= \alpha_k t_i^{'} + (1-\alpha_k)t_i^{'} \\
&= t_i^{'}
\end{aligned}$$

Since $t_i^{'} \in G_i$, the application A_k is schedulable at server S_k. □

When we compare the execution sequences at server S_k and at the dedicated processor, we can observe that, at the end of each partition cycle, S_k has put the same amount of processing capacity to run the application tasks as the dedicated processor. However, if the tasks are running at the dedicated processor, they are not blocked and can be completed earlier within each partition cycle. Thus, we need an additional constraint to bound the delay of task completion at server S_k. This bound is set by the second condition of the Theorem and is equal to the minimum inactivity period before each task's deadline.

An immediate extension of Theorem 1 is to include the possible blocking delay due to synchronization and operating system overheads. Assume that the tasks in the partition adopt a priority ceiling protocol to access shared objects [12]. The blocking time is bounded to the longest critical section in the partition that accesses the shared objects. Similarly, additional delays caused by the operating system level can be considered. For instance, the partition may be invoked later than the scheduled moments since the proceeding partition just enters an O/S critical section. In this case, we can use the longest critical section in the operating system level to bound this scheduling delay. These delay bounds can be easily included into the computation of $W_i(\alpha_k, t_i)$.

3.2. A Scheduling Example

Theorem 1 provides a solution to determine how frequent a partition server must be scheduling at the O/S level and how much processor capacity it should use during its partition cycle. It is easy to see that $B_0(\alpha_k)$ and η_k are increasing functions of α_k. This implies that if more processor capacity is assigned to a partition during its partition period, the tasks can still meet their deadlines even if the partition cycle increases. To illustrate the result of Theorem 1, we consider an example in Table 1 in where four application partitions are allocated in a core module. Each partition consists of several periodic tasks and the corresponding parameters of (C_i, T_i) are listed in the Table. Tasks are set to have deadlines equal to their periods and are scheduled within each partition according to a rate-monotonic algorithm [11]. The processor utilization demanded by the 4 partitions, ρ_k, are 0.25, 0.15, 0.27, and 0.03, respectively.

Following Theorem 1, the minimum level-i inactivity period is calculated for each partition and for a given capacity assignment α_k, i.e., $B_0(\alpha_k) = \mathbf{min}_i \mathbf{max}_{t \in H_i}(t - \sum_{j=1}^{i} \frac{C_j}{\alpha_k} \lceil \frac{t}{T_j} \rceil)$. The resulting inactivity periods are plotted in Figure 3 for the four partitions. It is easy to see that, when α_k is slightly larger than the processor utilization, the tasks with a low priority (and a long period) just meet their deadlines, and thus have a small inactivity period. On the other hand, when α is much larger than the processor uti-

	Partition 1 (util.=0.25)	*Partition 2 (util.=0.15)*	*Partition 3 (util.=0.27)*	*Partition 4 (util.=0.03)*
tasks	(4,100)	(2,50)	(7,80)	(1,80)
(C_i, T_i)	(9,120)	(1,70)	(9,100)	(2,120)
	(7,150)	(8,110)	(16,170)	
	(15,250)	(4,150)		
	(10,320)			

Table 1. Task parameters for the example partitions

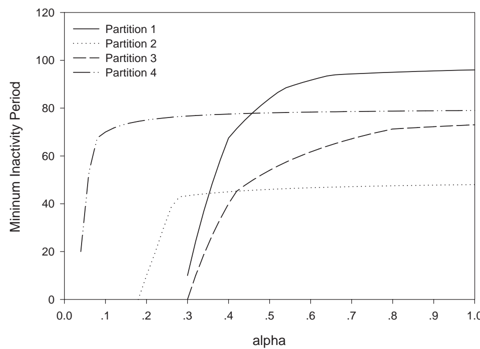


Figure 3. Inactivity periods of the exmple partitions

lization of the partition, the inactivity period is bounded to the smallest task period in each partition. This is due to the fact that the tasks with a short period cannot accumulate more inactivity period before their deadlines. The curves in the figure also show that an increase of α_k after the knees wouldn't make the inactivity periods significantly longer.

In Figure 4, the maximum partition cycles are depicted with respect to the assigned capacity α_k. If the points below the curve are chosen to set up cyclic scheduling parameters for each partition at O/S level, then the tasks in the partition are guaranteed to meet their deadlines. For instance, the curve for partition 2 indicates that, if the partition receives 28% of processor capacity, then its tasks are schedulable as long as its partition cycle is less than or equal to 59 time units. Note that the maximum partition cycles increase as we assign more capacity to each partition. This increase is governed by the accumulation of inactivity period when α_k is small. Then, the growth follows by a factor of $1/(1-\alpha_k)$ for a larger α_k.

Figure 4 suggests possible selections of (α_k, η_k) for the 4 partitions subject to a total assignment of processor capacity

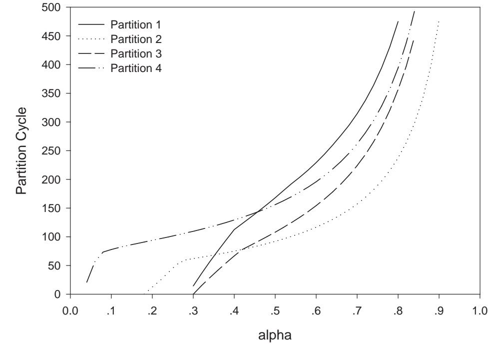


Figure 4. The maximum partition cycles for different processor capacity assignments

not greater than 1. From the Figure, a feasible assignment for (α_k, η_k) is (0.32, 36), (0.28, 59), (0.34, 28), and (0.06, 57), respectively. In the following subsection, we shall discuss the approaches of using the feasible pairs of (α_k, η_k) to construct cyclic schedules in the APEX O/S level.

To reveal the properties of the parameters (α_k, η_k) under different task characteristics, we present two evaluations on the task set of Partition 1 in Table 1. We first alter the task deadlines such that D_i is equal to $1.0T_i, 0.8T_i, 0.6T_i$, and $0.4T_i$ for all tasks. The tasks are then scheduled according to the deadline monotonic algorithm within the partition [5]. The maximum partition cycles are plotted in Figure 5. As the deadlines become tighter, the curves shift to the low-right corner. This change suggests that either the partition must be invoked more frequently or should be assigned with more processor capacity. For instance, if we fix the partition cycle at 56 time units and reduce the task deadlines from $1.0T_i$ to $0.4T_i$, then the processor capacity must be increased from 0.34 to 0.56 to guarantee the schedulability. The result of demanding more processor capacity for tasks with tight deadlines is similar to the channel bandwidth reservation for real-time messages in ATM network. If messages of a channel have a tight deadline, a significant bandwidth must be reserved while the message transmission is multiplexed with other channels.

The second experiment, shown in Figure 6, concerns with the partition schedulability under different task execution times. Assume that we upgrade the avionics cabinets with a high speed processor. Then, the application partitions are still schedulable even if we allocate a less amount of processor capacity or extend partition cycles. In Figure 6, we show the maximum partition cycles for Partition 1 of Table 1. By changing task execution times proportionally, the processor utilizations are set to 0.25, 0.20, 0.15 and 0.1. It

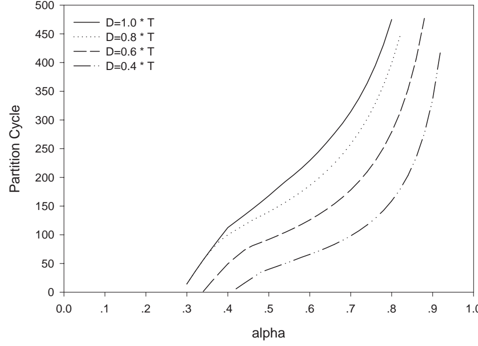


Figure 5. The maximum partition cycles for partition 1 under different task deadlines

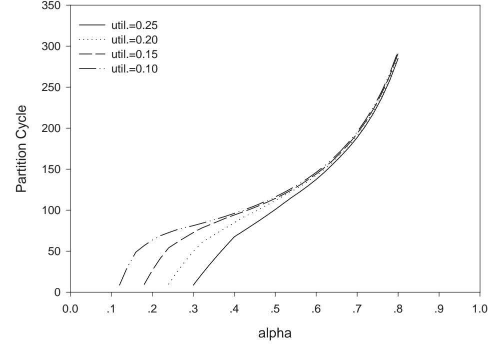


Figure 6. The maximum partition cycles for partition 1 under different processor utilizations

is interesting to observe the improvement of schedulability when α_k is slightly larger than the required utilization. For instance, assume that the partition cycle is set to 56. Then, for the 4 different utilization requirements, the processor capacities needed for schedulability can be reduced from 0.34, to 0.28, 0.212, and 0.145, respectively. On the other hand, once a more than sufficient capacity is assigned to the partition, the maximum partition cycle is relatively independent of the processor utilization, and is mainly affected by task deadlines and periods.

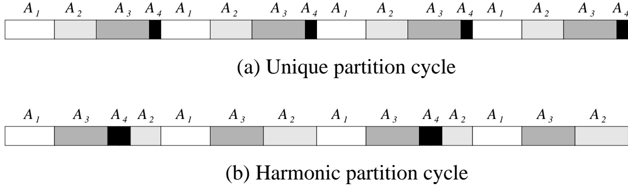
3.3. Cyclic Schedule at the APEX O/S Level

Given the schedulability requirement of (α_k, η_k) for each partition server S_k, a cyclic schedule must be constructed at the APEX O/S level. Notice that the pair of parameters (α_k, η_k) indicates that the partition must receive an α_k amount of processor capacity at least every η_k time units. The execution period allocated to the partition needs not to be continuous, or to be restricted at any specific instance of a scheduling cycle. This property makes the construction of the cyclic schedule extremely flexible. In the following, we will use the example in Table 1 to illustrate two simple approaches.

1. Unique partition cycle approach: In this approach, the O/S schedules every partition in a cyclic period equal to the minimum of η_k and each partition is allocated an amount of processor capacity that is proportional to α_k. For instance, in the example of Table 1, a set of feasible assignment of (α_k, η_k) can be transferred from $\{$(0.32, 36), (0.28, 59), (0.34, 28), (0.06, 57)$\}$ to $\{$(0.32, 28), (0.28, 28), (0.34, 28), (0.06, 28)$\}$. The resulting cyclic schedule is shown in Figure 7(a) where the O/S invokes each partition every 28 time units and allocates 8.96, 7.84, 9.52, and 1.68 time units to partitions 1, 2, 3, and 4, respectively.

2. Harmonic partition cycle approach: When partition cycles are substantially different, we can adjust them to form a set of harmonic cycles in which η_j is a multiple of η_i, if $\eta_i < \eta_j$ for all i and j. Then, the O/S cyclic schedule runs repeatedly every major cycle which is equal to the maximum of η_k. Each major cycle is further divided into several minor cycles with a length equal to the minimum of η_k. In Figure 7(b), the cyclic schedule for the example is illustrated where the set of (α_k, η_k) is adjusted to $\{$(0.32, 28), (0.28, 56), (0.34, 28), (0.06, 56)$\}$. The major and minor cycles are 56 and 28 time units, respectively. Note that, partitions 2 and 4 can be invoked once every major cycle. However, after the processing intervals for partitions 1 and 3 are assigned in every minor cycle, there doesn't exist a continuous processing interval of length 0.28x56 time units in a major cycle. To schedule the application tasks in partition 2, we can assign an interval of length 0.22x28 in the first minor cycle and the other interval of length 0.34x28 in the second minor cycle to the partition. This assignment meets the requirement of allocating 28% of processor capacity to the partition every 56 time units.

Comparing Figures 7(a) and 7(b), there are a less number of context switches in the harmonic partition cycle approach. The reduction could be significant if there are many partitions with different partition cycles. In such a case, an optimal approach of constructing a set of harmonic cycles and assigning processing intervals should be sought in order to minimize the number of context switches. The other modification we may consider is that, when a partition cycle



(a) Unique partition cycle

(b) Harmonic partition cycle

Figure 7. Example cyclic schedules at the O/S level

is reduced to fit in the cyclic schedule, the originally allocated capacity becomes more than sufficient. We can either redistribute the extra capacity equally to all partitions or to keep the same allocation in the partition for future extensions.

In addition to the flexible cycle schedules, the choice of α_k is adaptable as long as the sum of all α_k is less than or equal to 1. The parameter α_k must be select to ensure partition schedulability at a dedicated processor of speed α_k. For instance, if the task set is scheduled according to a rate monotonic algorithm, we can define a minimum capacity equal to $\rho_k / n(2^{1/n} - 1)$ which guarantees the partition schedulability. Additional capacity can be assigned to the partition such that the partition cycle can be prolonged. In the case that new tasks are added into the partition and the modified task set is still schedulable with the original capacity assignment, we may need to change the partition cycle and construct a new cyclic schedule at the O/S level. However, as long as the new cyclic schedule is subject to the requirement of (α_k, η_k) for each partition, no change to other partitions is necessary to ensure their schedulability.

4. Conclusion

The partition scheduling in APEX environment is described in this paper. To guarantee the schedulability of fixed priority algorithms at each partition, the necessary requirements for the cyclic schedule at the operating system level are established. Represented by partition cycle η_k and partition capacity α_k, the requirements can be easily used to construct flexible cyclic schedules. In addition, as long as the requirements can be satisfied for a partition, a modification of the task priority or task characteristics in any partition does not require a global change of scheduling algorithms.

For simplicity, we exclude several factors that may introduce additional delays to task execution in the derivation of α_k, and η_k . As in [6], the factors are listed as task blocking delay, partition delay (e.g., due to the critical sections at the O/S level), task release jitter, and operating system overheads (such as interrupt serving, context switch, etc.) Note that, since blocking delay and release jitter depend upon the tasks within the partition, their delay effect can be included in the computation of task inactivity periods and task deadlines. At the operating system level, the partition delay can be deducted from the allocated processor capacity. To count for the operating system overheads, we can model them as a linear function of processing interval if the minimum interoccurrence time is known. Then, a portion of the allocated processor capacity can be taken away from each partition.

We are currently extending the proposed scheme to address the scheduling issues of multiple processors and data transmission on the ARINC 659 backplane bus [3] and ARINC 629 inter-cabinet network [1]. The task is challenging as we aim at task and message scheduling to meet the end-to-end timing requirements of the system. The scheduling requirements devised in this paper can be used to define a promising base such that additional scheduling constraints can be incorporated.

References

[1] *Multi-transmitter Data Bus, ARINC Specification 629*. Aeronautical Radio Inc., Annapolis, MD, October 1990.

[2] *Design Guide for Integrated Avionics, ARINC report 651*. Aeronautical Radio Inc., Annapolis, MD, November 1991.

[3] *Backplane Data Bus, ARINC Specification 659*. Aeronautical Radio Inc., Annapolis, MD, December 1993.

[4] *Avionics Application Software Standard Interface, ARINC Specification 653*. Aeronautical Radio Inc., Annapolis, MD, January 1997.

[5] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard real-time scheduling: the deadline-monotonic approach. *Eighth IEEE Workshop on Real-time Operating Systems and Software*, pages 133–137, 1991.

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

[7] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. *Proc. IEEE Real-Time Systems Symposium*, pages 308–319, Dec 1997.

[8] J. Lehoczky. Fixed priority scheduling for periodic task sets with arbitrary deadlines. *Proc. IEEE Real-Time Systems Symposium*, pages 201–209, Dec. 1990.

[9] J. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. *Proc. IEEE Real-Time Systems Symposium*, pages 110–123, Dec. 1992.

[10] J. Lehoczky, L. Sha, and Y. Ding. The rate-monotonic scheduling algorithm: exact characteristics and average case behavior. *Proc. IEEE Real-Time Systems Symposium*, pages 166–171, Dec. 1989.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in hard real time environment. *J. Assoc. Comput. Mach.*, 20(1):46–61, 1973.

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, Sep. 1990.