A Fixed-Priority-Driven Open Environment for Real-Time Applications*

Tei-Wei Kuo and Ching-Hui Li
{ktw,lch86}@cs.ccu.edu.tw
Real-Time and Embedded System Laboratory
Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi, Taiwan 621, ROC

Abstract

This paper extends the useful concept of open systems proposed by Liu, et al. [2, 3] in scheduling real-time applications and non-real-time applications, where the schedulability of each real-time application can be validated independently of other applications in the system. We replace the underlying earliest-deadline-first OS scheduler of the open system architecture in [2, 3] with a rate-monotonic OS scheduler. The motivation behind this work is that many existing operating systems may not support the earliest deadline first scheduling very well. We propose to use the idea of sporadic servers [14] to preserve CPU cycles for applications. We also develop schedulability tests for real-time applications which adopt the rate monotonic scheduling algorithm, the earliest deadline first scheduling algorithm [11], the priority ceiling protocol [18], and the stack resource policy [1]. We allow tasks in each application to share local and global non-preemptable resources. A global resource synchronization mechanism is proposed. This paper provides a fixed-priority-based alternative for the important open system architecture.

1 Introduction

Rate-based scheduling has been an active research topic in the past few years. Researchers proposed various rate-based scheduling algorithms for periodic and sporadic tasks based on the notion of General Processor Sharing (GPS) [7, 9, 16, 19, 20]. The idea of GPS-based scheduling is very different from common disciplines such as priority-driven scheduling [1, 11, 12, 18] and time-driven scheduling [5]. The GPS-based scheduling is a work-conserving scheduling mechanism. The schedulability of each task in GPS-based systems is guaranteed with an assigned CPU service rate, independent of the demands of other tasks. The enforcement of a guaranteed CPU service rate for a task must rely on certain admission control mechanism to manage the total workload of the system [15].

In the past decades, there was an increasing demand for application systems with response-time requirements. Complex application systems may be developed independently and then run together on a computer with any combination. The notion of open system is proposed by Liu, et al. [2, 3] in scheduling real-time applications and non-real-time applications such that the schedulability of each real-time application can be validated independently of other applications in the system. A two-level hierarchical scheduling scheme which resembles the GPS scheduling scheme in [19] is proposed to provide fair sharing of a processor among applications running on the processor. Applications are executed by either a total bandwidth server [16] or a constant utilization server [2, 3], depending on their characteristics. Servers are then scheduled by an earliest-deadline-first (EDF) OS scheduler [11] with the reserved CPU capacity. Schedulability tests for real-time applications which share global resources are proposed [3].

This paper is to develop a new two-level hierarchical scheduling scheme for an open system with the motivation that many existing operating systems may not support the earliest deadline first (EDF) scheduling very well. We shall follow the important open system architecture developed by Liu, et al. [2, 3] and replace the underlying OS scheduler with a fixed-priority scheduler, such as the rate monotonic (RM) scheduler [11]. We propose to use sporadic servers [14] to execute applications in the open system because of the incompatibility of the constant utilization (and total bandwidth) servers and the underlying RM OS scheduler.

*Supported in part by a research grant from the National Science Council under Grant NSC87-2213-E-194-002

We shall show that the schedulability of any real-time applications which adopt EDF, RM, the priority ceiling protocol (PCP) [18], or the stack resource policy (SRP) [1] can be validated independently of other applications. We develop a global resource synchronization mechanism for the two-level hierarchical scheduling scheme such that tasks in different applications can share global non-preemptable resources. Efficient schedulability tests for applications which adopts EDF, RM, PCP, and SRP scheduling algorithms are developed. The scheme can also be extended by incorporating the idea of the rate monotonic analysis (RMA) proceedure [10, 17] and the concept of fundamental frequency [8] in developing more accurate schedulability tests.

There are two major contributions in this paper: (1) We propose a fixed-priority-based alternative for the powerful open system architecture proposed by Liu, et al. [2, 3]. An entire scheduling framework is proposed to execute real-time and non-real-time applications in the open system. A rate-monotonic OS scheduler and the idea of sporadic servers [14] are used to preserve CPU cycles for applications. (2) We develop schedulability tests for real-time applications which adopt the rate monotonic scheduling algorithm, the earliest deadline scheduling algorithm, the priority ceiling protocol, and the stack resource policy. We allow tasks in each application to share local and global non-preemptable resources. A global synchronization mechanism is proposed.

The rest of this paper is organized as follows: Section 2 describes the open system architecture and the proposed scheduling hierarchy. Section 3 develops the schedulability tests for real-time applications which adopt RM and EDF scheduling. Section 4 considers the synchronization of local resources. We develop the schedulability tests for real-time applications which adopt PCP and SRP scheduling. Section 5 proposes a global resource synchronization mechanism and develops the schedulability tests for real-time applications which adopt PCP and SRP scheduling. Section 6 is the conclusion.

2 Open System Architecture

2.1 System Architecture

This section is meant to propose a new two-level hierarchical scheme for scheduling independently developed real-time and non-real-time applications in an open environment. We shall follow the definitions and the open system architecture in [2, 3]. Distinct from the previous work, we are interested in a fixed-priority-based hierarchical scheduling scheme in supporting the open system architecture. We propose to adopt a fixed-priority scheduling algorithm RM for the underlying OS scheduler in the two-level hierarchical scheme, where Liu, et al. adopts a dynamic-priority scheduling algorithm EDF for the underlying OS scheduler in the previous work [2, 3]. This research is motivated by the observation that many existing operating systems can support fixed-priority scheduling, such as the rate monotonic (RM) scheduling, better than dynamic-priority scheduling, such as EDF, where RM is an optimal fixed-priority scheduling algorithm which assigns tasks priorities inversely proportional to their periods, and EDF is an optimal dynamic-priority scheduling algorithm which schedules tasks in the order of their deadlines.

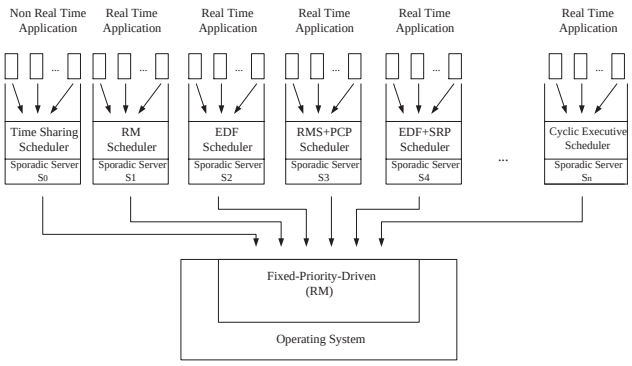


Figure 1: An Open system Architecture

Figure 1 provides the open system architecture supported by the two-level fixed-priority hierarchical scheme. The system has a single processor whose speed is one. There are real-time and non-real-time applications executing in the open environment. Each application A_i is executed by a sporadic server S_i [14] with a CPU budget c_i and a period p_i[1], where a sporadic server was originally proposed by Sprunt, Sha, and Lehoczky [14] to execute sporadic task requests. Each sporadic server is associated with a ready queue contains ready tasks of the application executing on the server. Each server has a scheduler associated with it. The server scheduler uses the scheduling algorithm $\sum_i$, such as EDF or RM, chosen for the application A_i to schedule tasks and order tasks in the ready queue of sporadic server S_i. Figure 1 shows that server S_0 uses a time sharing algorithm to schedule tasks in application

[1] The open system architecture supported by the two-level hierarchical scheme proposed by Liu, et al. [2, 3] services applications by either a total bandwidth server [16] or a constant utilization server [2, 3], depending on the characteristics of applications.

A_0.

The scheduler provided by the operating system is called the *OS scheduler*. The OS scheduler maintains all servers in the system [2, 3]. The OS scheduler replenishs the server budget for every server according to the definitions of sporadic server [14]. A server is ready if its ready queue is not empty. The OS scheduler schedules all the ready servers according to a fixed-priority scheduling algorithm. Since the rate monotonic (RM) scheduling algorithm is an optimal fixed-priority scheduling algorithm, we shall use RM as the scheduling algorithm for the underlying OS scheduler in this paper. Tasks of the scheduled ready server can execute until they run out of the budget of the server, or a higher-priority ready server arrives. Tasks of the scheduled ready server execute in the order defined by the chosen scheduling algorithm $\sum_i$ of the server.

The selection of a CPU budget c_i and a period p_i for a new application A_i and the admission control will be discussed in the next section. The operations of the OS scheduler are defined as follows:

Initiation of an application:
- Create a sporadic server S_i with a CPU budget c_i and a period p_i for a new application A_i if A_i passes the admission control.

Maintenance of each server S_i:
- The budget replenishment mechanism is done according to the definitions of sporadic server [14]. /* The budget replenishment mechanism will be summarized in the next section */

Interaction between tasks and server scheduler:
- The scheduler of each server S_i schedules tasks according to the chosen algorithm $\sum_i$.
- The scheduled task of each server S_i executes under the CPU budget of S_i.

Scheduling of servers:
- The OS scheduler schedules the ready server with the highest priority in the system. /* global resource synchronization will be discussed in Section 5. */

Termination of an application:
- Destroy the corresponding sporadic server.

2.2 Scheduling Hierarchy and Admission Control

2.2.1 Sporadic Servers

Sporadic servers were originally proposed for servicing sporadic task requests [14]. In this paper, a sporadic server is used to execute periodic or sporadic tasks in an application. The budget replenishment mechanism of a sporadic server is summarized as follows:

Each sporadic server is associated with a CPU budget c_i and a period p_i. Suppose that the system is scheduled by a fixed-priority scheduler. Let P_s denote the priority of the task which is executing. A priority level P_i is *active* if $P_s \geq P_i$. A priority level is *idle* if it is not active. Let RT_i denote the replenishment time for a sporadic server executing at priority level P_i. The repleshment time RT_i of server S_i (with priority level P_i and period p_i) is set as follows:

- If S_i has a non-zero remaining CPU budget, and P_i becomes active at time t, then $RT_i = t + p_i$.
- If the CPU budget of S_i is exhausted, and the CPU budget of S_i is replenished at time t, then $RT_i = t + p_i$.

The replenishment amount is determined when P_i becomes idle, or when the remaining CPU budget of S_i becomes zero. The replenishment amount is equal to the amount of server execution time consumed since the last time at which the status of P_i changes from idle to active. As shown in [14], a periodic task set that is schedulable with a periodic task τ_i is also schedulable if τ_i is replaced with a sporadic server with the same period and CPU budget. The schedulability analysis of sporadic servers is equivalent to that of periodic tasks. We refer interested readers to [14] for details.

2.2.2 Scheduling and Admission Control

Since the rate monotonic (RM) scheduling algorithm is an optimal fixed-priority scheduling algorithm, we adopt RM for the underlying OS scheduler with the requirements that each server has a distinct priority: The priorities of servers are inversely proportional to their periods. The server with the smallest period has the highest priority in the system. Every server is associated with a distinct priority level. When two servers have the same period, their priority order is determined arbitrarily. That is, if S_i and S_j are two sporadic servers in the system, and the period of S_i is larger than the period of S_j, then the priority of S_i is smaller than the priority of S_j. If the period of S_i is equal to the period of S_j, then the priority order of S_i and S_j is arbitrary.

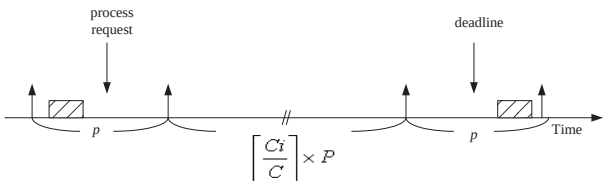


Figure 2: The relationship between timing constraints and server budget

When a sporadic server is used to execute a non-real-time application, the period and the CPU budget of the server can be set arbitrarily. However, the period should be a small number to make the application responsive to user requests. The amount of the CPU budget determines the amount of the CPU capacity reserved for the non-real-time application. When a sporadic server is used to execute a real-time application, the period of the server must be no larger than $d_i/(2 + \lceil \frac{c_i}{c} \rceil)$ for any task τ_i with CPU requirement c_i and relative deadline d_i in the real-time application, where c is the CPU budget of the server. It is because the sporadic server is only guaranteed to receive c units of CPU cycles within every p units of time, as shown in Figure 2.

In order to better utilize the CPU cycles, we assume that the periods of all sporadic servers in the system are harmonically related. In other words, if a period of a server can not divide the period of another server, then the later can divide the former. As shown in [14], a periodic task set that is schedulable with a periodic task τ_i is also schedulable if τ_i is replaced with a sporadic server with the same period and CPU budget. The schedulability analysis of sporadic servers is equivalent to that of periodic tasks. Since Kuo and Mok [8] shows that the achievable utilization factor of any periodic task set with periods being harmonically related is 100%, the achievable utilization factor of a collection of sporadic servers with periods being harmonically related is 100%. Note that our requirement in assigning each sporadic server with a distinct priority level does not invalidate the above schedulability test because servers with the same period have consecutive priority levels. The admission control mechanism is as follows:

Let U be the sum of utilization factors of all sporadic servers in the system, where the utilization factor of a server is the ratio of its CPU budget and period. A new application A_i with a sporadic server S_i is admitted to enter the system if $U + U_i \leq 100\%$, where U_i is the utilization factor of S_i.

The considerations of global resource utilization will be made in Section 5.

3 Independent Applications over Sporadic Servers

3.1 Overview

This section is meant to derive the achievable utilization factor for the schedulability analysis of independent applications in the open environment, where tasks in independent applications do not share global or local non-preemptable resources. We will derive a sufficient condition for tasks in a real-time application which adopts the rate monotonic (RM) scheduling algorithm and a sufficient and necessary condition for tasks in a real-time application which adopts the earliest deadline first (EDF) scheduling algorithm [11]. The schedulability of each application A_i with a reserved CPU capacity $\frac{C}{P}$ can be validated independently of other applications, where the CPU budget and the period of the corresponding server S_i are C and P, respectively. We shall delay the discussions of local and global resource sharing to Sections 4 and 5, respectively.

As assumed in [11], let a real-time application A_i consist of n independent periodic tasks $\tau_1, \tau_2, \cdots, \tau_n$, where each task τ_j has a CPU requirement c_j and a period p_j, and the relative deadline d_j of each task τ_j be equal to its period p_j. Tasks share only preemptable resources such as CPU.

Suppose that a sporadic server S_i with a CPU budget C and a period P is used to execute application A_i, and the reserved capacity of A_i is $\frac{C}{P}$. Let the period P of the server be the greatest common divisor (GCD) or a divisor of the GCD of all of the periods of tasks in the application, and the initial phase of each task occur at a time point which is a multiple of P. Note that the CPU requirements of each periodic task in A_i can still be much less than P. P will be at least no less than the time granularity of the system timer. For example, in many modern operating systems such as Microsoft Windows, the timer granularity is usually set as $10ms$, and the granularity of CPU requirements/consumption can be much much less than $10ms$.

In the following sections, we shall derive a sufficient condition for tasks in a real-time application which adopts the rate monotonic (RM) scheduling algorithm and a sufficient and necessary condition for those which adopt the earliest deadline first (EDF) scheduling algorithm [11].

3.2 RM Server Scheduler

Let a real-time application A_i consist of n independent periodic tasks $\tau_1, \tau_2, \cdots, \tau_n$, where each task τ_j has a CPU requirement c_j and a period p_j, and a sporadic server S_i with a CPU budget C and a period P be used to execute application A_i. We shall follow and revise the terminologies and the main theorems in [11] in deriving the achievable utilization factor for tasks in A_i, independently of other applications in the open environment. Note that A_i only reserves $\frac{C}{P}$ of the entire system CPU cycles (the system speed is equal to one).

The CPU requirement c_j of each task τ_j is measured in terms of the system with speed equal to one. Let server S_i adopt the RM scheduling algorithm.

Definition 1 *A critical instant for a task is defined as an instant at which a request for the task will need to take the maximum number of sporadic server periods to complete.*

Note that Liu and Layland [11] defined that a critical instant for a task is an instant at which a request for the task will have the largest response time, where tasks run on a system whose speed is one. Since tasks in each application A_i now execute under the CPU budget of a sporadic server S_i, and a sporadic server is only guaranteed to receive C time units of CPU cycles within every P time units which may happen near the beginning or the end of a server's period, the definition of critical instant must be re-defined according to the scheduling behavior of sporadic servers by the OS scheduler.

Definition 2 *[11] A real number α is the achievable utilization factor of a scheduling algorithm $\sum$ if any task set with a utilization factor no larger than α is schedulable by $\sum$.*

Theorem 1 *A critical instant for any task in A_i occurs whenever the task is requested simultaneously with requests for all higher priority tasks in A_i.*

Proof. The proof of this theorem follows the same argument of the corresponding theorem, i.e., Theorem 1, in [11]. That is, any advancing of the request of a higher-priority task toward the request time of any task τ_j will not speed up the completion of τ_j in terms of the number of sporadic server periods. □

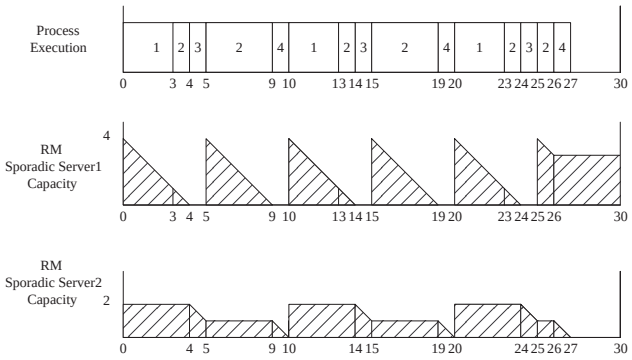
Theorem 2 *Let the ratio between any two task periods in A_i be less than 2. The achievable utilization factor of the RM scheduling algorithm for tasks in A_i is $U = \frac{C}{P} n(2^{1/n} - 1)$.*

Proof. The proof of the corresponding theorem, i.e. Theorem 4, in [11] can be revised to prove this theorem. The main idea is to show that the achievable utilization factor of the n tasks in application A_i is minimized when the CPU requirement c_j of each task τ_j in A_i is set as $\frac{C}{P}(p_{j+1} - p_j)$, for $j < n$, and $c_n = \frac{C}{P}(p_n - 2(c_1 + c_2 + \cdots + c_{n-1}))$. □

Theorem 3 *The achievable utilization factor of the RM scheduling algorithm for server S_i is $U = \frac{C}{P} n(2^{1/n} - 1)$.*

Proof. The correctness of this proof directly follows from the arguments of the corresponding theorem in [11], i.e., Theorem 5 in [11]: Suppose that for some j, $p_n = q * p_j + r$, $q > 1$ and $r \geq 0$. Let us replace task τ_j with a new task τ'_j such that $p'_j = q * p_j$ and $c'_j = c_j$, and increase c_n by the amount needed to again fully utilize the CPU budget of the server. This increase is at most $c_j(q-1)$. Similar to the arguments in Theorem 5 in [11], the replacement of τ_j with τ'_j will decrease the achievable utilization factor. Thus, this theorem follows directly from Theorem 2. □

Example 1 *RM Server Scheduler:*



Sporadic Server 1 (5,4) capacity 0.8

Process	Exec Time	Period	Utilization
1	3	10	0.3
2	6	15	0.4

Sporadic Server 2 (10,2) capacity 0.2

Process	Exec Time	Period	Utilization
3	1	10	0.1
4	2	20	0.1

Figure 3: RM server schedulers

Let a system consist of two real-time applications which adopt a RM scheduler. The CPU budget and the period of the first sporadic server S_1 are 4 and 5, respectively. Two periodic tasks τ_1 and τ_2 execute on S_1. The CPU requirements and the period of $\tau_1(/\tau_2)$ are 3 and 10 (6 and 15), respectively. The CPU budget and the period of the second sporadic server S_2 are 2 and 10, respectively. Two periodic tasks τ_3 and τ_4 execute on S_2. The CPU requirements and the period of $\tau_3(/\tau_4)$ are 1 and 10 (/ 2 and 20), respectively. Since the OS scheduler adopts RM scheduling, the priority P_1 of S_1 is higher than the priority P_2 of S_2. Since S_1 and S_2 also adopt RM scheduling, the priority of τ_1 is higher than the priority of τ_2, and the priority of τ_3 is higher than the priority of τ_4.

Figure 3 shows the executions of τ_1, τ_2, τ_3, and τ_4 on servers S_1 and S_2 from time 0 to time 30. At time 0, all tasks arrives. τ_1 on server S_1 starts execution. Since the priority levels P_1 and P_2 both become active, the replenishment time RT_1 of S_1 is set as 5, and the replenishment time RT_2 of S_2 is set as 14. Since the CPU budget of S_1 is 4, τ_2 starts execution at time 3

after τ_1 finishes its execution. At time 4, the CPU budget of S_1 is exhausted, and τ_3 on S_2 starts execution. At time 4, the priority level P_1 becomes idle, and the priority level P_2 becomes active. The replenishment amount of S_1 is set as 4.

At time 5, the CPU budget of S_1 is replenished such that S_1 preempts S_2. τ_2 on S_1 resumes its execution. The replenishment time RT_1 of S_1 is set as 10. At time 9, the CPU budget of S_1 is exhausted again. Although τ_2 has not finished its execution, τ_2 must stop. τ_4 on S_2 starts its execution. At time 10, the CPU budget of S_1 is replenished such that S_1 preempts S_2. Since the second request of τ_1 arrives, τ_1 starts its execution at time 10 although S_2 has not finished its execution. Note that S_1 adopts RM scheduling. □

Example 1 provides an interesting insight in the schedulability analysis of independent tasks of a real-time application: The total utilization factor of the first application is 0.7 which is higher than the achievable utilization factor $0.8 * 2(2^{1/2} - 1) = 0.664$, as shown in Theorem 3. Apparently, the first application which consists of τ_1 and τ_2 is schedulable. As astute readers may point out, with the same arguments adopted in the proofs of Theorem 1 and 3, the rate monotonic analysis (RMA) procedure [10, 17] can be applied here. When a RMA schedulability test is used, the schedulability of each application must be verified in an off-line fashion. Furthermore, the total utilization factor of the second application is 0.2 which is higher than the achievable utilization factor $0.2 * 2(2^{1/2} - 1) = 0.166$, as shown in Theorem 3. Since τ_3 and τ_4 are belonging to the same fundamental frequency, their achievable utilization factor should be $0.2 * 1(2^{1/1} - 1) = 0.2$ according to the fundamental-frequency-based schedulability analysis proposed in [8] and the arguments in the proofs of Theorem 1 and 3. In other words, the schedulability analysis of RM server schedulers in this paper can be further generalized by incorporating the ideas of RMA and fundamental frequency.

3.3 EDF Server Scheduler

Let a real-time application A_i consist of n independent periodic tasks $\tau_1, \tau_2, \cdots, \tau_n$, where each task τ_j has a CPU requirement c_j and a period p_j, and a sporadic server S_i with a CPU budget C and a period P be used to execute application A_i. The CPU requirement c_j of each task τ_j is measured in terms of the system with speed equal to one. Let server S_i adopt the EDF scheduling algorithm. We shall follow and revise the terminologies and the main theorems in [11] in deriving the achievable utilization factor for tasks in A_i independently of other applications in the open environment.

Definition 3 *[11] An overflow occurs at time t if there exists a task which misses its deadline at time t.*

Definition 4 *A server is idle at time t if the priority level of the server is idle, and the CPU budget of the server is not exhausted.*

Let P_s denote the priority of the server which is executing. A priority level P_i is *active* if $P_s \geq P_i$. A priority level is *idle* if it is not active.

Lemma 1 *When EDF is used to schedule tasks in application A_i, there is no idle time for server S_i prior to an overflow.*

Proof. This lemma can be proved in an analogous way as the corresponding theorem, i.e., Theorem 6, in [11]: The main idea is as follows: If there exists an idle period before an overflow, then moving all requests of any task, after the idle period, toward the end time of the idle period will not result in an idle period and may only result in an overflow. This is a contradiction to the assumption that there is a server idle time prior to an overflow. □

Theorem 4 *For the set of n periodic tasks serviced by a sporadic server S_i with a period P and a CPU budget C, EDF is feasible if and only if $\sum \frac{c_j}{p_j} \leq \frac{C}{P}$.*

Proof. The corresponding theorem, i.e., Theorem 7, in [11] can be revised to prove this theorem. To show the necessity, we show that the total demand of computation time by all tasks between time 0 and time $p_1 p_2 \cdots p_n$, i.e.,

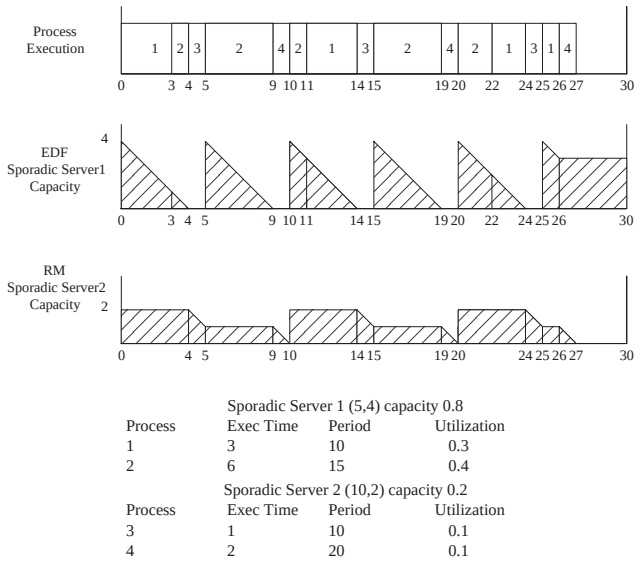
$$(p_2 p_3 \cdots p_n)c_1 + (p_1 p_3 \cdots p_n)c_2 + \cdots + (p_1 p_2 \cdots p_{n-1})c_n,$$

should not be more than $(p_1 p_2 \cdots p_n)\frac{C}{P}$. To show the sufficiency, similar arguments in the corresponding theorem, i.e., Theorem 7, in [11] are applied. The main idea is to show that the total demand of computation time between 0 and t being larger than $t\frac{C}{P}$, if an overflow occurs at time t, will be a contradiction to the assumption $\sum \frac{c_j}{p_j} \leq \frac{C}{P}$. □

Example 2 *EDF Server Scheduler:*

Let an open system consist of two real-time applications, as illustrated in Example 1, except that the first application now adopts an EDF server scheduler. The second application still adopts a RM server scheduler. Since the underlying OS scheduler adopts RM scheduling, the priority P_1 of the first server S_1 is higher than the priority P_2 of the second server S_2.

Figure 4 shows the executions of τ_1, τ_2, τ_3, and τ_4 on servers S_1 and S_2 from time 0 to time 30. At time



Sporadic Server 1 (5,4) capacity 0.8

Process	Exec Time	Period	Utilization
1	3	10	0.3
2	6	15	0.4

Sporadic Server 2 (10,2) capacity 0.2

Process	Exec Time	Period	Utilization
3	1	10	0.1
4	2	20	0.1

Figure 4: RM and EDF server schedulers

0, all tasks arrives. τ_1 on server S_1 starts execution because it has a deadline earlier than that of τ_2. Since the priority levels P_1 and P_2 both become active, the replenishment time RT_1 of S_1 is set as 5, and the replenishment time RT_2 of S_2 is set as 10. When τ_1 finishes its execution at time 3, τ_2 starts its execution since the remaining CPU budget of S_1 is one. At time 4, the CPU budget of S_1 is exhausted, and τ_3 on S_2 starts execution. Note that the priority of τ_3 is higher than the priority of τ_4 because the second application adopts RM scheduling. At time 4, the priority level P_1 becomes idle, and the priority level P_2 becomes active. The replenishment amount of S_1 is set as 4.

At time 5, the CPU budget of S_1 is replenished such that S_1 preempts S_2 (since the underlying OS scheduler is a RM scheduler). τ_2 on S_1 resumes its execution. The replenishment time RT_1 of S_1 is set as 10. At time 9, the CPU budget of S_1 is exhausted again. Although τ_2 has not finished its execution, τ_2 must stop. τ_4 on S_2 starts its execution. At time 10, the CPU budget of S_1 is replenished such that S_1 preempts S_2. Since an EDF server scheduler is adopted for S_1, τ_2 on S_1 resumes its execution, although the second request of τ_1 arrives.

As shown in Theorem 4, the achievable utilization bound of the first application is $\frac{4}{5} = 0.8$. Since the total utilization factor of the first application is $(0.3 + 0.4) = 0.7$, the first application is schedulable, as shown in Figure 4. □

4 Local Resource Synchronization

The previous section provide efficient schedulability tests for real-time applications which adopt the RM or EDF scheduling algorithm. The schedulability of real-time applications which consist of independent tasks can be validated independently of other applications in the open environment. This section is meant to extend the two-level hierarchical scheme proposed in the previous sections in local resource sharing among tasks of in the same application.

Let a real-time application A_i consist of n independent periodic tasks $\tau_1, \tau_2, \cdots, \tau_n$, where each task τ_j has a CPU requirement c_j and a period p_j, and the relative deadline d_j of each task τ_j be equal to its period p_j. Tasks in A_i may share non-preemptable resources, such as semaphores, belonging to application A_i. No non-preemptable global resources are shared among tasks in different applications, where a global resource is a resource, such as a system data structure or a device, which is belonging to the system and accessible to all applications. The discussion of global resource synchronization will be delayed until Section 5.

Suppose that a sporadic server S_i with a CPU budget C and a period P is used to execute application A_i, and the reserved capacity of A_i is $\frac{C}{P}$. Because all applications and their corresponding servers only share preemptable resources such as CPU among themselves, each server can still be guaranteed with the reserved amount of CPU budget within every period. The schedulability tests proposed in the previous section for applications which adopt EDF and RM remain the same.

When an application adopts a real-time resource synchronization algorithm such as the *priority ceiling protocol* (PCP) [18] or the *stack resource policy* (SRP) [1], we shall show as follows that the schedulability tests for PCP and SRP remain the same in terms of the CPU capacity of the corresponding server. Sha, Rajkumar, and Lehoczky [18] proposed PCP in which tasks can inherit the higher priority of a task they block. The priority ceiling of a resource is the priority of the highest priority task which may use the resource. A task's resource request is blocked if its priority is no larger than the priority ceiling of any resource which has been grabbed by another task but has not yet been released. SRP proposed by Baker [1] is extended from PCP based on the same concept of implicit locking. SRP reduces the maximum number of context switchings to no more than twice the number of jobs in the system and can handle certain dynamic priority assignments, such as EDF.

The schedulability tests of PCP and SRP for a n-task real-time application A_i executing on a sporadic server S_i with a CPU budget C and a period P are as follows: Let tasks in application A_i scheduled by PCP or SRP be listed in the increasing order of their periods, where PCP adopts the rate monotonic priority assignment scheme, and SRP adopts the earliest deadline first priority assignment scheme [11].

Theorem 5 *[18] A task τ_i scheduled by PCP on a system with speed equal to one will always meet its deadline if $(\sum_{j<i} \frac{c_j}{p_j}) + \frac{c_i+b_i}{p_i} < i(2^{1/i} - 1)$, where b_i is the blocking time of τ_i by lower-priority tasks.*

Theorem 6 *A task τ_i in application A_i which adopts PCP will always meet its deadline if $(\sum_{j<i} \frac{c_j}{p_j}) + \frac{c_i+b_i}{p_i} < \frac{C}{P} i(2^{1/i} - 1)$, where b_i is the blocking time of τ_i by lower-priority tasks, and C and P are the CPU budget and the period of the server for A_i, respectively.*

Proof. By considering the blocking time b_i as the extra computation time for τ_i, the correctness of this theorem follows directly from the same arguments in Theorem 2, 3 and 5. □

Theorem 7 *[1] Tasks scheduled by SRP on a system with speed equal to one are schedulable if $\forall_{k=1,\cdots,n} k, (\sum_{i=1}^{k} \frac{c_i}{p_i}) + \frac{b_k}{p_k} \leq 1$, where b_k is the execution time of the longest critical section of any task whose relative deadline, i.e., period, is less than that of τ_k.*

Theorem 8 *Tasks in application A_i which adopts SRP are schedulable if $\forall_{k=1,\cdots,n} k, (\sum_{i=1}^{k} \frac{c_i}{p_i}) + \frac{b_k}{p_k} \leq \frac{C}{P}$, where b_k is the execution time of the longest critical section of any task whose relative deadline, i.e., period, is less than that of τ_k, and C and P are the CPU budget and the period of the server for A_i, respectively.*

Proof. By considering the blocking time b_k as the extra computation time imposed on τ_k, the correctness of this theorem follows directly from the same arguments in Theorem 4 and 7. □

5 Global Resource Synchronization

5.1 Scheduling Mechanism

The purpose of this section is to further extend the two-level hierarchical scheme in global resource sharing among tasks in different applications in the open environment. The objective of this section is to provide a uniform mechanism for global resource sharing and propose the corresponding schedulability tests.

Let a real-time application A_i consist of n periodic tasks $\tau_1, \tau_2, \cdots, \tau_n$, where each task τ_j has a CPU requirement c_j and a period p_j, and the relative deadline d_j of each task τ_j be equal to its period p_j. Tasks in A_i may share non-preemptable local or global resources, where a local resource is belonging to A_i, and a global resource is a resource, such as a system data structure, belonging to the system and accessible to all applications. A critical section of a task is *global* if the task is accessing a global resource in the critical section. A critical section is *local* if it is not global. Suppose that a sporadic server S_i with a CPU budget C and a period P is used to execute application A_i.

The global resource synchronization among applications is handled in a restricted way to provide a uniform mechanism with bounded priority inversion time for applications sharing global resources: We assume that critical sections are properly nested, as required by PCP [18].

We adopt the idea of the kernelized monitor model [12] to share global resources among applications. Let the system have a unique sporadic server S_g which is responsible to reserving a CPU budget in servicing all global critical sections. The period P_g of S_g is the GCD of all applications which might access global resources. Since the underlying OS scheduler adopts RM scheduling, let S_g have the highest priority in the system. The decision of the CPU budget C_g of S_g will be discussed in the next section. P_g will be at least no less than the time granularity of the system timer.

When a task τ_i in application A_i requests for a global resource, the request is always granted, and τ_i is moved to the ready queue of S_g. S_g will become ready to execute the global critical section of τ_i until τ_i leave the corresponding critical section. Note that S_g executes at the highest priority level in the system and will immediately start executing τ_i when τ_i enters the global critical section. τ_i may lock any local and global resources during the execution of the global critical section. Since S_g executes at the highest priority level in the system, S_g will only service global critical sections of tasks in different applications one by one, and no other servers may preempt S_g. When τ_i leaves the global critical section, τ_i goes back to the ready queue of the original server S_i for execution, and S_g stops execution because of no further service requests. The CPU time consumed by τ_i executing on S_g will also be subtracted from the remaining CPU budget of S_i by the system. This double count is to prevent tasks on S_i from overrunning S_g. If the remaining CPU budget of S_i decreases to zero during subtraction, it remains zero. Note that tasks in A_i will only consume more CPU cycles than their reserved capacity. In other words, the

subtraction will not affect the schedulability tests in the previous section. The replenishment procedure of S_g follows the definitions of sporadic server [14]. We assume that S_g always has a sufficient CPU budget for executing global critical sections. The decision of the CPU budget C_g of S_g will be discussed in Section 5.2.2.

The operations of the OS scheduler defined in Section 2 are revised as follows: Let the system have a unique sporadic server S_g which is responsible to reserving a CPU budget in servicing all global critical sections. S_g has the highest priority in the system.

Initiation of an application:
- Create a sporadic server S_i with a CPU budget c_i and a period p_i for a new application A_i if A_i passes the admission control.

/* the corresponding admission control will be discussed in the following section. */

Maintenance of each server S_i and S_g:
- The budget replenishment mechanism is done according to the definitions of sporadic server [14].

Interaction between tasks and server scheduler:
- The scheduler of each server S_i schedules tasks according to the chosen algorithm $\sum_i$.
- When a task τ_i in application A_i requests for a global resource, the request is always granted, and τ_i is moved to the ready queue of S_g.
- When a task τ_i in application A_i releases all of its global resources, τ_i is moved back to the ready queue of S_i, and the CPU time consumed by τ_i on S_g will be also subtracted from the remaining CPU budget of S_i by the system.
- The scheduled task of each server S_i executes under the CPU budget of S_i.

Scheduling of servers:
- The OS scheduler schedules the ready server with the highest priority in the system.

Termination of an application:
- Destroy the corresponding sporadic server.

Note that the global resource synchronization mechanism proposed in [2, 3] may require both the server and the task executing on the server to become non-preemptable when the task and the server accesses any global resource. It may impose a strong impact on applications which do not share global resources. Under the new two-level hierarchical scheme proposed in this paper, applications which do not access global resource will always receive its reserved CPU service independently of other applications. However, a portion of CPU cycles must be reserved for servicing global critical sections under the new scheme.

5.2 Schedulability Analysis, Implementation Consideration, and Admission Control

5.2.1 Schedulability Analysis and Implementation Consideration

This section is meant to discuss the implementation issues of servers which adopt PCP and SRP in local and global resource synchronization in the open environment. It is obvious that any sporadic server which does not have any task requesting any global resource will still receive its reserved CPU budget within every period. The schedulability tests of applications which adopt RM, EDF, PCP with only local resource synchronization, and SRP with only local resource synchronization remain the same. In the following, we shall propose schedulability tests for applications which adopt PCP or SRP and access global resources.

PCP with global resource synchronization:

Let S_g be a unique sporadic server with a period P_g and a CPU budget C_g to service global critical sections of tasks in all applications. Suppose that a real-time application A_i adopts PCP to schedule all tasks in A_i. Tasks in A_i may access global resources. The priority ceiling of any global resource is set as ∞, i.e., the highest possible priority level in A_i. When a task τ_i locks a global resource, τ_i automatically inherits the highest possible priority level ∞ in the system from a non-existing task τ_∞, where the system can assume that a non-existing task τ_∞ with a priority equal to ∞ also tries to lock the global resource, and the task only consumes an infinite small amount of CPU time. Let a sporadic server S_i be used to execute tasks in A_i, and the CPU budget and the period of S_i be C and P, respectively.

Lemma 2 *The blocking time of τ_i by a lower-priority task in application A_i can be determined based on the tasks in A_i plus τ_∞ independently of other applications.*

Proof. The correctness of this lemma follows directly from the fact that global critical sections of applications other than A_i only consume the CPU budget of S_g, and the global critical sections of applications other than A_i will not impose any blocking time on any task τ_i in A_i in terms of the CPU budget of S_i. □

Theorem 9 *A task τ_i in application A_i which adopts PCP will always meet its deadline if $(\sum_{j<i} \frac{c_j}{p_j}) + \frac{c_i+b_i}{p_i} < \frac{C}{P} i(2^{1/i} - 1)$, where b_i is the blocking time of τ_i by lower-priority tasks.*

Proof. The correctness of this theorem follows directly from Theorem 6 and Lemma 2. □

SRP with global resource synchronization:

Suppose that a real-time application A_i adopts SRP to schedule all tasks in A_i. Tasks in A_i may access global resources. Let the preemption level $\lceil GR \rceil_0$ for any global resource GR_i in the system be equal to ∞, i.e., the highest possible level, and each global resource has only one instance.

Lemma 3 *The execution time of the longest critical section of any task whose relative deadline, i.e., period, is less than that of τ_k in application A_i can be determined based on the tasks in A_i independently of other applications.*

Proof. The correctness of this lemma follows directly from the fact that global critical sections of applications other than A_i only consume the CPU budget of S_g, and the global critical sections of applications other than A_i will not impose any blocking time on any task τ_i in A_i in terms of the CPU budget of S_i. □

Theorem 10 *Tasks in application A_i which adopts SRP are schedulable if $\forall_{k=1,\cdots,n} k, (\sum_{i=1}^{k} \frac{c_i}{p_i}) + \frac{b_k}{p_k} \leq \frac{C}{P}$, where b_k is the execution time of the longest critical section of any task whose relative deadline, i.e., period, is less than that of τ_k.*

Proof. The correctness of this theorem follows directly from Theorem 8 and Lemma 3. □

In the beginning of this paper, we assume that the period P of a sporadic server S_i is the greatest common divisor (GCD) or a divisor of the GCD of all of the periods of tasks in the application. We also assume that periods of all sporadic servers in the system are harmonically related. The assumption on the harmonic relationship of server periods is merely to maximizing the entire system utilization. We must emphasize that when the periods of sporadic servers in the system are not harmonically related, the correctness of theorems proposed in this paper remain. However, the period P of a sporadic server S_i must be the greatest common divisor (GCD) or a divisor of the GCD of all of the periods of tasks in the application. Otherwise, the schedulability tests proposed in this paper must be more conservative in including extra overheads in CPU time because each period of the server may not fit well into the period of any task executing on the server. The overheads might be around two CPU budget cycles of a server.

5.2.2 Admission Control

This section is meant to propose the admission control for the open system architecture when tasks in different applications may share global resources.

The global synchronization mechanism adopts the idea of the kernelized monitor model [12]. A unique sporadic server S_g is responsible to reserving a CPU budget C_g in servicing all global critical sections for every period P_g. The period P_g of S_g is the GCD of all applications which might access global resources. Because an open system may consist of a dynamic set of applications in the system, we may assume that P_g is equal to a sufficiently small number, such as the time granularity of the system timer. Note that in many modern operating systems, the timer granularity is usually set as $10ms$.

When a real-time application A_i arrives, a sporadic server with a CPU budget C_i and a period P_i may be requested. Let B_i be the sum of the durations of all global critical sections in A_i which might be possibly requested with the reserved CPU budget C_i within a period. Note that P_g is no larger than the periods of all applications which might access any global resource. If no tasks in A_i may access any global resource, $B_i = 0$. Application A_i will pass the admission control if

- $(\sum_{\text{existing application } A_j} B_j) + B_i \leq C_g$
- $(\sum_{\text{existing application } A_j} \frac{C_j}{P_j}) + \frac{C_g}{P_g} + \frac{C_i}{P_i} \leq 1$

Note that all P_i and P_g are belonging to the same fundamental frequency. B_i for an application A_i must be determined before A_i is admitted to run. In the worst case, B_i can be close to C_i. However, it rarely happens that a global critical section in an application is very long. Suppose that the duration of the global critical section of any task in an application A_i is limited by a small number ϵ_i, and there are n_i periodic tasks in A_i. Since the period of the sporadic server of A_i is a multiple of P_g, B_i is no larger than $n_i * \epsilon_i$. Let the system consist of m applications, C_g must be no less than $\sum_{\text{existing application } A_j} n_j * \epsilon_j$. In other words, if we want to allow more applications which share global resources to run in the open environment, C_g must be enlarged. However, the server S_g which services global critical sections will reserve a larger amount of CPU budget C_g for every period P_g. It then reduces the number of applications and the amount of CPU budgets which can be reserved by the applications in the system.

6 Conclusion

This paper proposes a fixed-priority-based alternative for the important open system architecture proposed by Liu, et al. [2, 3]. We replace the underlying

earliest-deadline-first (EDF) OS scheduler of the original open system architecture with a rate-monotonic (RM) OS scheduler. The motivation behind this work is that many existing operating systems may not support the earliest deadline first scheduling very well. We propose to use the idea of sporadic servers [14] to preserve CPU cycles for applications, where the total bandwidth servers and constant utilization servers that are deadline-driven GPS-based servers in the original open system architecture can not be used with a fixed-priority OS scheduler. An entire scheduling framework is proposed in this paper to execute real-time and non-real-time applications in the open system. We develop schedulability tests for real-time applications which adopt the rate monotonic scheduling algorithm, the earliest deadline first scheduling algorithm, the priority ceiling protocol, and the stack resource policy. We allow tasks in each application to share local and global non-preemptable resources. A global synchronization mechanism is also proposed. The schedulability of each real-time applications can be validated independently of other applications in the system.

The concept of open systems provides application engineers a very useful way in developing real-time applications. Distinct from the past work [2, 3], we exploit fixed-priority-based scheduling for the open system architecture. The schedulability tests of applications which adopt RM, EDF, PCP, and SRP are proposed. Although the implementation of applications which adopts time-driven scheduling is not discussed in this paper, it is obvious that the open system architecture can also support time-driven applications. For example, a time-driven application can be assigned a sporadic server with the highest priority in the system to have a predictable behavior on its execution. The architecture can also support any $n > 1$ time-driven applications by providing them sporadic servers with the highest n priorities in the system, where the periods of the servers should be belonging to the same fundamental frequency to have predictable execution behaviors (Note their priorities should be higher than the priority of the server servicing global critical sections.) The overheads of the proposed two-level hierarchical scheme mainly come from the maintenance of sporadic servers, whereas the major overheads of the EDF-based two-level hierarchical scheme might come from the underlying EDF OS scheduling and the maintenance of total bandwidth servers and constant utilization servers. The server budgets and deadlines of the EDF-based two-level hierarchical scheme may need to be maintained virtually for every event occurrence. We must emphasize the budget replenishment mechanism of sporadic servers in this paper may not be executed more frequently than the maintenance of server budgets and deadlines of the two-level hierarchical scheme in [2, 3]. The evaluation of the performance and run-time overheads of the proposed approach is planned on RED-Linux, an real-time embedded operating system based on Linux [21][2]

For future research, we shall further explore the open system architecture in multiprocessor systems and develop middleware to support the architecture on existing operating systems. We will also further explore the issues of global resource sharing in open systems to support more robust and flexible mechanisms for global resource synchronization. We believe that more research in the open system architecture may be very rewarding in the building and integration of complex real-time applications.

References

[1] T.P. Baker, "A Stack-Based Resource Allocation Policy for Real Time Processes," *IEEE 11th Real-Time Systems Symposium*, December 4-7, 1990.

[2] Z. Deng, J. W.-S. Liu, and J. Sun, "A Scheme for scheduling Hard Real-Time Applications in Open System Environment," *Proceeding of the 9th Euromicro Workshop on Real-Time Systems,* pp. 191-199, June 1997.

[3] Z. Deng and J. W.-S. Liu, "Scheduling Real-Time Applications in an Open Environment," *IEEE 18th Real-Time Systems Symposium*, December 1997.

[4] C.-W. Hsueh and K.-J. Lin, "An Optimal Pinwheel Scheduler Using the Single-Number Reduction Technique," *IEEE Real-Time Systems Symposium*, December 1996, pp. 196-205.

[5] C.-W. Hsueh and K.-J. Lin, "On-Line Schedulers for Pinwheel Tasks Usingthe Time-Driven Approach," *the 10th Euromicro on Real-Time Systems*, June 1998, pp. 180-187.

[6] K. Jeffay, F.D. Smith, A. Moorthy, J. Anderson, "Proportional Share Scheduling of Operating System Services for Real-Time Applications," *IEEE 19th Real-Time Systems Symposium*, December 2-4, 1998.

[2] Note that when periods of sporadic servers in the rate-monotonic-based open system architecture are very small, the number of context switchings in the system may increase significantly in certain cases. On the other hand, an EDF-based open system architecture (with a total bandwidth server or a constant utilization server) might be restricted in allocating CPU budget for a task (during the deadline assignment) because, as indicated in [2, 3], the server must not run out of CPU budget when a higher-priority task with a small relative deadline arrives. It may also result in the increasing of the number of context switches significantly if it is not carefully implemented.

[7] J. Jehuda, G. Koren, and D.M. Berry, "A Time Sharing Architecture for Complex Real-Time Systems," *IEEE 1st International Conference on Engineering of Complex Computer Systems,* pp.9-16, 1995.

[8] T.-W. Kuo and A.K. Mok, "Incremental Reconfiguration and Load Adjustment in Adaptive Real-Time Systems," *IEEE Transactions on Computers,* Vol 46, No 12, pp. 1313-1324, December 1997.

[9] T.-W. Kuo, W.-R. Yang, and K.-J. Lin, "EGPS: A Class of Real-Time Scheduling Algorithms Based on Processor Sharing," *Proceeding of the 10th Euromicro Workshop on Real-Time Systems,* pp. 27-34, June 1998.

[10] J.P. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithms: - Exact Characterization and Average Behavior," *IEEE 10th Real-Time Systems Symposium*, December 1989.

[11] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *JACM*, Vol. 20, No. 1, January 1973, pp. 46-61.

[12] A.K. Mok, "Fundamental Design Problems for the Hard Real-Time Environment," MIT Ph.D. Dissertation, Cambridge, MA, 1983.

[13] A.K. Parekh and R.G. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single Node Case," *Proceedings of IEEE INFOCOM*, 1992.

[14] B. Sprunt, "Aperiodic Task Scheduling for Real-Time Systems," Ph.D. dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1990.

[15] M. Spuri, G. Buttazzo, and Sensini, "Scheduling Aperiodic Tasks in Dynamic Scheduling Environment," *IEEE Real-Time Systems Symposium*, 1995.

[16] M. Spuri and G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *IEEE 17th Real-Time Systems Symposium,* pp. 179-210, December 1996.

[17] L. Sha, "Distributed Real-Time System Design Using Generalized Rate Monotonic Theory," lecture note, Software Engineering Institute, CMU, 1992.

[18] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization,¡" *Technical Report* CMU-CS-87-181, Dept. of Computer Science, CMU, November 1987. *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990.

[19] I. Stoica, H. Abdel-Wahab, K. Jeffay, S.K. Baruah, J.E. Gehrke, and C.G. Plaxton, "A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems," *IEEE Real-Time Systems Symposium*, 1996, pp. 288-299.

[20] C.A. Waldspurger, "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management," Ph.D. Thesis, Technical Report, MIT/LCS/TR-667, Laboratory for CS, MIT, September 1995.

[21] Y.-C. Wang and K.J. Lin, "Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel," *IEEE 20th Real-Time Systems Symposium*, Dec 1999.