Hierarchical Reservation Support in Resource Kernels

Saowanee Saewong and Ragunathan (Raj) Rajkumar
Real-time and Multimedia Systems Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213
{*ssaewong, raj+*}*@ece.cmu.edu*

Abstract

Reservation-based operating systems provide applications with guaranteed and timely access to system resources. One of the chief benefits of the approach is temporal isolation, which prevents the timing mis-behavior of one task from hurting other tasks. Such a benefit is appealing enough that applications desire to recursively apply this reservation model to each of their components [2, 8]. In this paper, we study such a hierarchical reservation model in the context of fixed-priority scheduling as used in systems such as the Resource Kernel [13]. We perform detailed analyses under both deferrable-server and sporadic-server replenishment schemes using a hierarchical deadline-monotonic scheduler. We also provide a least upper scheduling bound for a hierarchical rate-monotonic scheduler. Finally, we describe the implementation of this hierarchical scheme in CMU's Linux/RK and demonstrate its functional behavior of providing hierarchical temporal isolation.

1 Introduction

OS resource management for real-time and multimedia systems has been an active research topic for decades. A variety of real-time scheduling policies with corresponding schedulability analyses and admission control tests has been developed. Resource reservation schemes [10] have recently gained popularity since they can not only provide timeliness guarantees to real-time tasks, but also provide temporal isolation across tasks. Strictly speaking, the timing misbehavior of one task cannot hurt the ability of another task to make forward progress or meet its timing constraints. However, in a multi-user or multi-threaded application environment, temporal isolation among tasks is not sufficient. The abstraction of resource partitioning should be extended to any application in recursive fashion, so that any application or component can support temporal isolation among its various components. In other words, an application can create "child reservations" from its parent and specify an appropriate real-time or non-real-time scheduling policy to manage its guaranteed share of the underlying resource.

With such support, the admission control for the timeliness guarantees of each real-time reserve can be done locally at each level of the hierarchy. For example, a child reservation cannot use more resources than it reserves if there is no unclaimed resource left. However, if some child reservations underuse their resource allocation, the unused resource must be preferably assigned to their siblings. In this case, the parent reservation always gets its full quota of resource grants even though some of its children may misbehave. Such a hierarchical reservation model can be applied to applications such as multimedia servers where QoS guarantees must be provided to multiple classes of clients.

We now provide a brief overview of the Resource Kernel approach to provide context and background for the design and implementation of our hierarchical reservation model.

1.1 Resource Kernel

A Resource Kernel (RK) [13] allows applications to specify only their resource and timeliness demands leaving the kernel to satisfy those demands using (hidden) resource management schemes. This separation of resource specification from resource management allows OS-subsystem-specific customization by extending, optimizing or even replacing resource management schemes. Since an application can use multiple resource reservations simultaneously, those reserves are aggregated in a *resource set* that in turn are associated to application tasks. The reservation specification uses the $\{C, T, D\}$ model for the reservation of C units of resource time every recurring time interval T before a relative deadline D. To provide the temporal isolation among tasks, the resource kernel monitors and enforces actual resource used by each reserve. Resource reservation was implemented at CMU first in μkernels and later ported to monolithic kernels [12], in which it was demonstrated that the resource management subsystem can be built in a "portable" OS-independent fashion. Linux-RK [12] is the modified CMU version of Linux using the Resource Kernel techniques. It can provide timeliness guarantees of shared resources like, CPU, network bandwidth and disk bandwidth. In the rest of this paper, we use the term "reserve" and "reservation" interchangeably.

1.2 Related Work

The Open System architecture developed by Liu, et al. [5, 4] proposes a *two-level* hierarchical scheduling scheme in order to schedule real-time applications and non-real-time applications which may use different scheduling policies. The schedulability of each real-time application can be validated in isolation from other applications. The Earliest Deadline First (EDF) scheduling policy is used as the lower-level scheduler (OS level which is the root scheduler in our context). This architecture assumes that the initial phases of tasks served by the server scheduler (a child scheduler in our context) are known in advance and used by the lower-level scheduler to adjust the server budget for every new replenishment. The Open System architecture assumes that each scheduler is for one single application. In our architecture, the parent scheduler does not need to know any information about its

child schedulers. The requirements we need to know from any reserve are just the 3-tuple $\{C, T, D\}$, and similar to the classical real-time task model, no specific task phasing is required. Finally, we can support an *arbitrarily* deep hierarchy with any number of levels.

Kuo and Li [6] extended the Open System architecture to use the Rate-Monotonic OS scheduler instead of the earliest-deadline-first scheduler. There are two additional restrictions in their architecture. First, the period of the server has to be the greatest common divisor (GCD) or a divisor of the GCD for all of the periods of tasks served by the server. Secondly, the initial phase of each task *must* occur at a time-point which is a multiple of the server period. These restrictions may not be ideal in a multi-user environment where the user can spawn many real-time applications with different timing requirements. Unyelioglu and Rajkumar [17] proposed an initial approach for 2-level hierarchical reservation on RT-Mach, a microkernel using Resource Kernel techniques.

Lipari and Baruah [8] have recently analyzed a hierarchical extension to the Constant Bandwidth Server (CBS) framework[1]. While this work and ours share very similar goals, the reservation models are rather different, and we analyze a fixed-priority scheduling policy while they use the EDF policy. We also go farther with a fully functional implementation on a reservation-based real-time OS, Linux/RK and perform a system evaluation.

1.3 Organization of the Paper

The rest of this paper is organized as follows. In Section 2, we present the requirements of a hierarchical reservation scheme. The overview of our hierarchical reservation model is also presented. In Section 3, we outline the analysis of hierarchical reservation with deadline-monotonic scheduling policy and the assumptions we have made. The details of the analysis are presented in Section 4 and Section 5. Section 6 focuses on the implementation of the hierarchical reservation model and its performance evaluation. Finally, we present our concluding remarks outlining our research contributions and future work in Section 7.

2 The Hierarchical Reservation Model

In this section, we define the user-view of the hierarchical reservation model and define some terminology.

2.1 Terminology

The following terms will be used.

- The term *resource principal* [11] is used to refer to the reserve owners which can be a thread, an application, or a group of those.

- The term *reserve domain* of a reserve is used to represent all child reserves of that particular reserve including the reserve itself. (see Figure 1.)

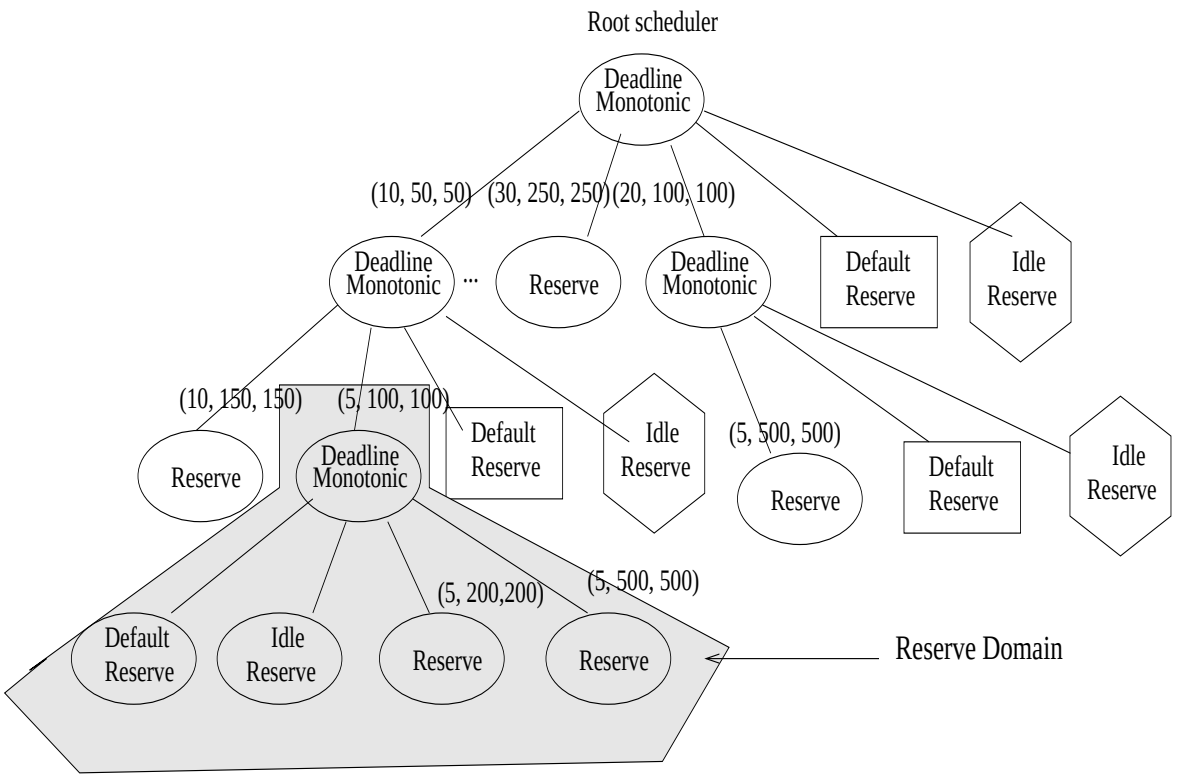


Figure 1: A Sample of Reservation Hierarchy

- *A depleted reserve* is a reserve that has used up its reserve within the current periodic interval.
- *A hard reservation* or *hard reserve* is a reserve which has consumed the guaranteed allocation in the current period and cannot be scheduled until it is replenished.
- *A soft reservation* or *soft reserve* is a reserve that can be scheduled at background priority (within its parent's reserve domain) after depletion if there is any available resource left in the domain.

2.2 Design Goals

We characterize the challenges of a hierarchical reservation model as a set of goals that a system should aim to satisfy:

- **Heterogeneity of Resource Scheduling Policies:** An application should be able to choose its own resource scheduling policies which can be real-time, non-real-time or combination of

those in its own resource partition. Thus, the system needs to have a mechanism to avoid scheduling mismatch among schedulers. Because timeliness guarantees are required in any real-time scheduler, schedulability analysis should be done *locally* without global knowledge of other schedulers.

- **Uniform Reserve Specification:** In order to maintain schedulability analysis locally in each layer of the hierarchy, a uniform reserve specification for both real-time and non-real-time schedulers is needed. This should also abstract away the details of scheduling policies.
- **Enforcement and Protection:** In addition to providing the timing guarantee to a lower layer (child reserve) , its upper layer (parent reserve) must enforce the resource usage so that any timing mis-behavior in the lower layer cannot hurt other resource principals.
- **Hierarchical temporal isolation:** If one or more child reserves underuse the resource, the parent reserve should still be able to get its entire reserved amount on the resource. The management of the unclaimed resources of its children should be done by the parent reserve.

2.3 The Functionality of Hierarchical Reservation

The functionality that is added by hierarchical reservations relative to "flat" resource reservations is partly illustrated in Figure 1 and can be summarized as follows.

- Each reserve can create child reserves to form a sub-tree.
- Parent reserves can use different scheduling policies specified by their resource principals.
- Each reserve uses the $\{C, T, D\}$ model for resource usage and timeliness specifications. The $\{C, T, D\}$ model can represent bandwidth (C/T) and delay (D) for real-time applications or time-slices for non-real-time requirements.
- The guaranteed resource allocation of a reserve is allocated and checked by its parent reserve.
- Each parent reserve (scheduler) ensures timeliness by dynamically monitoring and enforcing actual resource usage of its children. This is done recursively. Even though some resource principals can deploy a scheduler that lacks this support, temporal isolation is still maintained in its parent layers.
- A corresponding *default reserve* is activated when a parent reserve has available resources but there is no child reserve claiming that resource. This default reserve is meant for soft reserves and non-real-time applications.
- A corresponding *idle reserve* is activated when the parent reserve has available resources but there is no application (even non-real-time ones) trying to use the resource. This information

is useful to detect over-reservation and can be used as a feedback control for adaptive resource management.

- To maintain composability of the analysis, a hierarchical priority assignment scheme has been used to avoid scheduling mismatch among schedulers. Strictly speaking, a child reserve from a higher-priority parent will always be considered as a higher-priority reserve than any child reserve from a lower-priority parent.
- Starting at the root scheduler, the highest-priority child reserve eligible to execute is scheduled to run. This decision is then applied recursively.

2.4 Admission Control

The parent reserve is responsible for determining the schedulability of its children based on its

- **own reserve specification**: how much resource it gets, and how frequently,
- **a scheduling policy**: how it manages its resource and
- **children's requirements**: how much resource each child needs, and how frequently.

This does *not* depend on what the upper scheduling policies are (i.e. how its grandparent manages the resource and how much of the resource the parent gets compared to its siblings) and what lower scheduling policies their children have (i.e. how their children manage their resource share).

3 Hierarchical Deadline-Monotonic (HDM) Overview

In this paper, we primarily focus on using the deadline-monotonic scheduling policy. Being a generalization, this can be easily adapted to the rate-monotonic scheduling policy.

3.1 Assumptions

This section presents the overview of schedulability analysis for hierarchical schedulers using the deadline-monotonic (DM) policy. We will show later that the schedulability of each task/thread with $\{C, T, D\}$ requirement can be validated locally by its parent. We make the following assumptions to obtain these analytical results (similar to the classical real-time requirements of Liu and Layland [9]):

(A1) Tasks with hard deadlines are periodic and independent of other tasks.

(A2) The next request of each task occurs only after the previous request is completed.

(A3) After sending a request, a task is always ready to run without blocking itself or waiting for another resource.

(A4) We assume that tasks are preemptable at any time and the cost of preemption is negligibly small.

(A5) A task attaches with a child reserve which has the same capacity, period and deadline as its timing requirement, and synchronizes with the reserve.

(A6) A child reserve has an arbitrary phasing and may therefore be synchronized or unsynchronized with its parent reserve.

(A7) The period and deadline of a child reserve are equal to or greater than those of its parent reserve.

3.2 Notation

We use the following notation. A parent reserve (R^p) has a reserve specification of $\{C^p, T^p, D^p\}$ and has n child reserves: $R_1, R_2, ..., R_n$ with the request of $\{C_1, T_1, D_1\}$ to $\{C_n, T_n, D_n\}$ respectively. By convention, $P(\tau_1) > P(\tau_2) > ... > P(\tau_n)$ where $P(\tau_i)$ represents the priority of τ_i. For the Deadline-Monotonic policy, this means that $D_1 < D_2 < ... < D_n$. There are n tasks, $\tau_1, \tau_2, ..., \tau_n$, associated with reserves, $R_1, R_2, ..., R_n$, respectively.

Schedulability analysis is done on one reserve domain at a time, ignoring all other tasks outside the reserve domain to achieve the locality of schedulability analysis. This analysis of a reserve domain can be applied to any node in the hierarchy. During analysis, the Default and Idle reserves are *not* included in the child reserve set because they have the lowest priorities compared to other child reserves in the domain.

3.3 Deferrable Server and Sporadic Server

To analyze the schedulability of Hierarchical Deadline-Monotonic (HDM) scheduling, we consider a parent reserve as a server that serves requests from its child reserves. Even though all child reserves are periodic, they can have different initial phases which can produce *back-to-back execution* (*double hit* [16]) as in Deferrable Server (DS) which serves aperiodic requests [7]. This back-to-back execution can be avoided using the Sporadic Server (SS) technique [15]. In this paper, we analyze both the Deferrable Server and Sporadic Server replenishment schemes.

3.4 Two-step Schedulability Test

The schedulability test for HDM needs to be done in two steps:

1. **The schedulability of a parent reserve itself:** This is done by a "grandparent" reserve. The introduction of a multi-level hierarchy introduces a potential jittery behavior on each reserve. An undepleted reserve may have ready tasks to run during some periodic intervals, and no ready tasks in others. This behavior can force other sibling reserves to be able to execute

sooner sometimes and later at other times. Bernet and Burns [3] propose a general form of the worst-case response-time formulation under such jitter conditions:

$$\omega^{n+1} = C_i + \sum_{j<i} \lceil \frac{\omega^n + J_j}{T_j} \rceil C_j \tag{1}$$

J_j is a jitter form that in our general case is given by $J_j = D_j - C_j$ since an admitted reserve is guaranteed to get its C_j computation time by its deadline D_j, and this is the maximum jitter that can occur under these conditions. This formula has to be solved recursively starting with $\omega^0 = C_i$ and finishing when $\omega^{n+1} = \omega^n$ or when $\omega^{n+1} > D_i$ on failure.

Note: The jitter term above J_j will be used in subsequent theorems and is equal to $D_j - C_j$.

2. **The schedulability of all child reserves:** This analysis is dependent on the policy a parent reserve uses to schedule its children. In this paper, we focus on the Deadline-Monotonic (DM) scheduling policy. To analyze the schedulability of task sets that are associated with child reserves, the critical instant and the critical zone have to be determined.

3.5 Critical Instant and Critical Zone

Definition 1. *A critical instant for a task in the hierarchical reservation model is defined as an instant at which a request from the task associated with a child reserve will have the largest response time using the resource from its parent reserve.*

This is a refinement of the "critical instant" definition by Liu and Layland [9] where tasks run on a system where the resource is always available to the "parent reserve" for scheduling purposes. In our model, since the resource of child reserves is obtained from its parent reserve, tasks can run only if the parent reserve is also available. For example, consider a task attaching to the child reserve R_1 with (C_i, T_i) pair of (5,10) and its parent reserve R^p (4,5). Even though the task is ready to run and no other tasks run in the system, its execution is broken into 4 units every 5 units and, hence, takes 6 units of time to finish its job.[1]

For the critical zone, we use the same definition as presented by Liu and Layland [9].

4 Schedulability Analysis for HDM with Sporadic Server

Definition 2. *The "pre-used resource" (PR) of a task τ_i is the maximum amount of resource in the current period T^p which any lower-priority tasks may have claimed when τ_i arrives.*

[1] This description corresponds to the behavior of a hard reservation.

When an instance of task τ_i arrives, it may find that its parent reserve has been consumed by other lower-priority tasks. The maximum amount of this "pre-used" resource for any task τ_i, $PR(\tau_i)$ varies between $[0, C^p]$ depending on the existence of lower-priority tasks. Clearly, if τ_i is the lowest priority task, $PR(\tau_i) = 0$. More than one lower-priority task can show up and grab the resource. Therefore, we have

$$PR(\tau_i) = min(\sum_{j>i} C_j, C^p) \tag{2}$$

We now present the following theorem.

4.1 Critical Instant of SS-HDM

Theorem 1. *Under the Hierarchical Deadline-Monotonic scheduler with Sporadic Server replenishment, a critical instant for a task associated with a child reserve occurs whenever the task is requested simultaneously with requests for all higher-priority tasks and lower-priority tasks have used the resource quota of the current replenishment for $PR(\tau_i)$ units before the task shows up.*

Proof.
We will divide this proof into two parts. First, we prove why the response time is the longest if requests of all higher-priority tasks and the task come in at the same time. Secondly, the worst case due to the resource "pre-usage" by lower-priority tasks is derived.

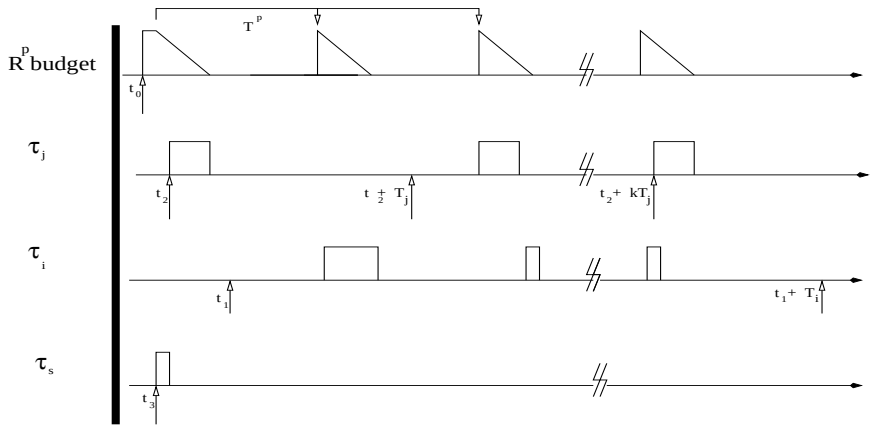


Figure 2: The Critical Instant of SS-HDM.

We shall follow the same approach as that of Liu and Layland [9]. However, in our model, the child reserves (the task set) in a reserve domain can only be scheduled when its parent reserve is eligible to execute: C^p units of time every T^p with a relative deadline of D^p. (In the Liu and Layland case, the

task set can be scheduled 100% of the time.)

We first determine the phasing of a task τ_i that will make its response time the longest with respect to higher-priority tasks. Consider any task τ_j with higher priority than τ_i. Suppose τ_j arrived earlier than τ_i. There are two cases. One, τ_j could execute for some time since its parent reserve was available. Then, moving τ_j's arrival time towards τ_i's arrival time can only make τ_i's response time longer or stay the same. Two, τ_j could not execute since the parent reserve was depleted. In this case, moving τ_i's arrival time towards τ_j's can only make τ_i's response time longer. Now, suppose that task τ_j arrived later than task τ_i. Moving task τ_j's arrival time towards that of τ_i cannot make the response time for τ_i shorter. Using this argument across all higher-priority tasks, the longest response time for task τ_i occurs when it arrives together with its higher-priority tasks.

Now, consider any task τ_s with lower priority than τ_i. Its request occurs at time t_3 as shown in Figure 2. In addition to the preemption by higher-priority tasks, unlike the analysis in the classical Liu and Layland model, τ_s can cause τ_i's response time (and that of higher priority tasks) to be longer by consuming the parent reserve's resource before τ_i and its higher-priority tasks arrive. The maximum amount of time that can be consumed by all lower-priority tasks is given by $PR(\tau_i)$.

We now analyze when the pre-used resource, $PR(\tau_i)$, should occur to produce the critical instant. With the SS replenishment, the resource is always replenished T^p units from when it is used. Therefore, if we advance the $PR(\tau_i)$ to be $t_3 - \phi$, $\phi \geq 0$, the next replenishment will be advanced too and, hence, task τ_i can use its resource earlier. Therefore, we can conclude that the critical instant of task τ_i happens when all of its higher-priority tasks request the resource at the same time, and its lower-priority tasks have pre-used the resource for $PR(\tau_i)$ at $PR(\tau_i)$ time units before τ_i request. □

4.2 The Worst-Case Response Time of SS-HDM

If a child reserve τ_i is admitted into the system, it is guaranteed by its parent reserve that it will get its requested time within its current period (or the current deadline of this replenishment). But the actual execution intervals can be at the start or the end of the guaranteed time-frame. In other words, if there are tasks associated with "another" higher-priority parent reserve, the grandparent scheduler will schedule that task before task τ_i. Otherwise, τ_i will get the resource immediately. Therefore, the worst-case completion time of task τ_i can be obtained by the combination of the following:

- The worst-case phasing of τ_i occurs within its parent reserve. This corresponds to the critical instant defined in Theorem 1.
- The parent reserve of a task τ_i can get *its* guaranteed C^p value at the end of its guaranteed time-frame T^p (or D^p when $D^p < T^p$). Note that this is a pessimistic assumption because the parent reserve may be the highest-priority task in the system and can get the resource right away. However, this pessimistic assumption is a necessary cost to accomplish the locality of scheduling

analysis, without the need for any information from its grandparent (what scheduling policy it uses, what the parent's priority is, etc.). This means that even the highest-priority task in this reserve domain may have to wait for $T^p - C^p$ (or $D^p - C^p$ when $D^p < T^p$) to use the resource after its arrival.

Figure 3 shows the critical zone of task τ_i which we have analyzed so far. The first time-line shows the current available resource of the parent reserve. The shadow boxes show the execution time of tasks τ_j, τ_i and τ_s which produce the worst-case scenario as discussed above.

The last line shows the critical zone of τ_i. The execution time of each task from previous time-lines is projected to the last line. We will formalize this execution time pattern in the proof of the next theorem.

Theorem 2. *For the Hierarchical Deadline-Monotonic Scheduler, the worst-case response time for a task τ_i that shares the parent reserve with the reserve specification as $\{C^p, T^p, D^p\}$ with tasks τ_1, τ_2, ..., τ_n where the $P(\tau_1) < P(\tau_2) < ... < P(\tau_n)$ is the smallest solution to the following equation:*

$$\omega^{n+1} = C_i + \sum_{j<i, R_j \in child(R_p)} \lceil \frac{\omega^n + J_j}{T_j} \rceil C_j + \lceil \frac{\omega^n + PR(\tau_i)}{T^p} \rceil (T^p - C^p) \tag{3}$$

This formula has to be solved recursively starting with $\omega^0 = (T^p - C^p) + C_i$ and finishing when $\omega^{n+1} = \omega^n$ or when $\omega^{n+1} > D_i$ on failure.

Proof.

As shown in Figure 3, at the critical instant, the resource is first consumed by $PR(\tau_i)$ at time $PR(\tau_i)$ before the request of τ_i starts. The first replenishment in the critical zone of the resource hence begins at time $t_1 - PR(\tau_i) + T^p$. Before the first replenishment time, τ_i may get the resource if there is any resource left from $PR(\tau_i)$. After the first replenishment time, τ_i and its higher-priority tasks continuously get their parent reserve's resource until τ_i finishes its job. The resource is then shaped into a periodic form with parameter C^p every T^p. We can think of the waiting time to get the resource, shown as brick boxes in the figure, as a *pseudo-task* having the capacity $T^p - C^p$ and the period T^p. The preemption from the pseudo-task is given by

$$\lceil \frac{\omega + PR(\tau_i)}{T^p} \rceil (T^p - C^p) \tag{4}$$

where ω is the time that takes τ_i to finish its job. The remainder of the resource after excluding the preemption of the pseudo-task are shared by τ_i and its higher-priority tasks. The maximum preemption by higher-priority tasks τ_j, which have a jitter form, J_j, defined in equation (1), is given

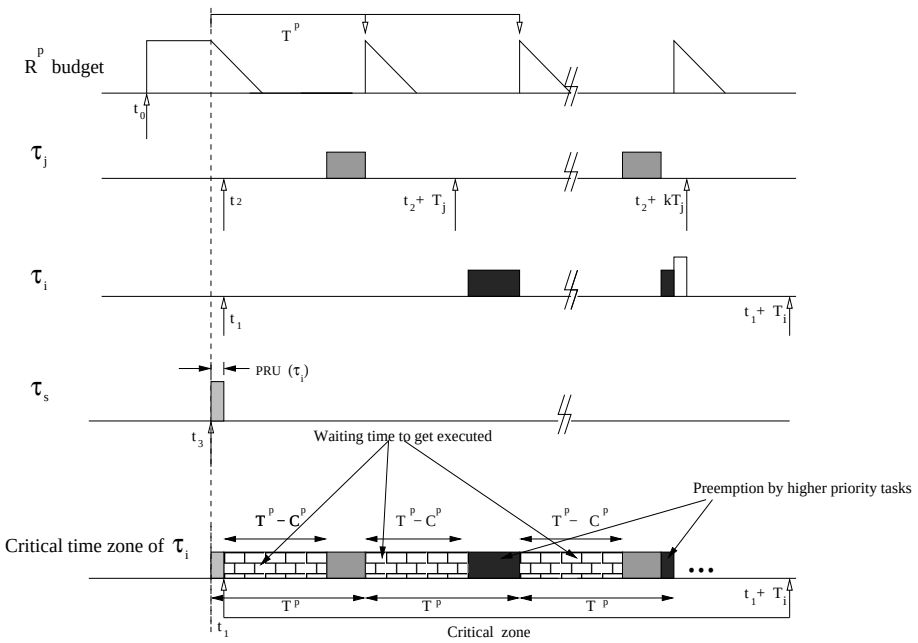


Figure 3: The critical zone of task τ_i

by

$$\sum_{j<i} \lceil \frac{\omega + J_j}{T_j} \rceil C_j \tag{5}$$

where ω is the time that τ_i takes to finish its job.

In order for τ_i to get its entire resource requirement of C_i, the total preemption experienced by τ_i is given by the sum of equations (4) and (5). Therefore, τ_i's completion time is given by

$$\omega^{n+1} = C_i + \sum_{j<i} \lceil \frac{\omega^n + J_j}{T_j} \rceil C_j + \lceil \frac{\omega^n + PR(\tau_i)}{T^p} \rceil (T^p - C^p) \tag{6}$$

As shown in the beginning of this proof, τ_i has to wait at least $T^p - C^p$ to get the resource. Therefore, in the best case, the finishing time is $T^p - C^p + C_i$ which is ω^0. We can then recursively find the finish-time of τ_i which covers all possible preemptions.[2] □

5 Schedulability Analysis for Deferrable Server HDM

In this section, we now analyze the schedulability of Hierarchical Deadline-Monotonic scheduler using the deferrable server. The performance comparison between DS and SS is beyond the scope of this

[2] In the case that $D^p < T^p$, we need to substitute $T^p - C^p$ with $D^p - C^p$ in equation (6).

paper, and can be found in [7] and [3].

5.1 Critical Instant of DS-HDM

Theorem 3. *Under the hierarchical deadline-monotonic scheduler with deferrable server replenishment:*
(a) If $PR(\tau_i) < C^p$, the critical instant for a task associated with a child reserve occurs whenever (1) its request occurs exactly C^p time units after the replenishment and (2) the task is requested simultaneously with requests for all higher-priority tasks.
(b) If $PR(\tau_i) = C^p$, the critical instant happens when additionally, lower-priority tasks have preclaimed the resource for C^p units right before the task request.

Proof.
Unlike the sporadic server, the deferrable server (DS) performs the replenishment at fixed time instants. Let us assume that the parent reserve is replenished at time $t_0 + kT^p$ for $k \geq 0$. We now consider the critical instant of task τ_i that requests the resource C_i at time $t_0 + \phi$. Let us first assume that τ_i is the highest priority. The parent reserve can guarantee only that each C_i, $C_i > C^p$, will be given to τ_i before $t_0 + \phi + T^p$. Two cases arise:

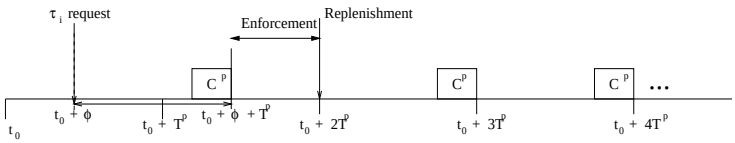


Figure 4: The effect of unsynchronized phase between child and parent reserves

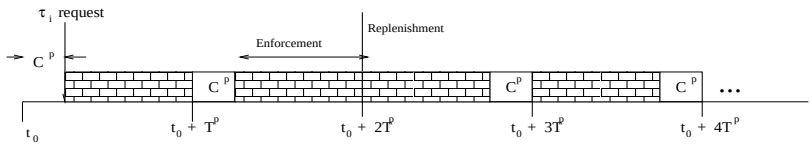
Case 1. If $\phi > C^p$ and τ_i gets its resource after time $t_0 + T^p$, the parent reserve will be enforced by its grandparent after resource C^p is depleted. The replenishment will start again at time $t_0 + 2T^p$. If τ_i always gets the resource as late as possible, it will have the largest possible completion time for a given ϕ as shown in Figure 4. Now, let us vary ϕ. As can be seen, the enforcement during time $(t_0+\phi+T^p, t_0+2T^p)$, which represents time the task has to wait to get the newly replenished resource, equals to $T^p - \phi$. Clearly, increasing ϕ reduces the waiting time and, hence, reduces its completion time. Therefore, task τ_i has the largest completion time when $\phi = C^p$.
Case 2. If $\phi < C^p$, even though τ_i gets the resource right before the guarantee time $t_0 + \phi + T^p$, the parent reserve still has the resource left for $C^p - \phi$. If we assume that τ_i gets the rest of the resource right before the next replenishment $t_0 + 2T^p$, it, hence, gets the resource $2C^p - \phi$, which is monotonically decreasing in ϕ. The largest possible value of ϕ, which equals to C^p, produces the maximum completion time.

Combining both two cases, the critical instant of task τ_i occurs when its request starts at time C^p after the replenishment.

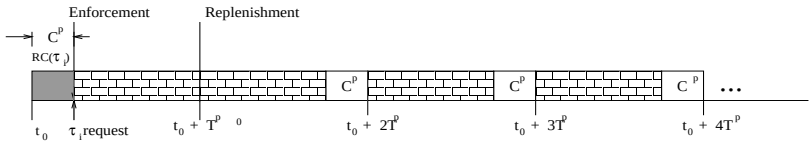
We now relax the assumption that τ_i has the highest priority. Suppose there is τ_j, $j < i$. It can cause the maximum preemption to task τ_i when their requests occur simultaneously. The detail of this analysis is the same as shown in the previous section.

Considering the effect of resource pre-usage by lower-priority tasks, we have shown that if the task shows up C^p after the replenishment, it can get the resource in the worst case after the next replenishment time whether the resource is pre-used or not. However, if lower-priority tasks come to get the resource $PR(\tau_i)$ right before τ_i shows up and $PR(\tau_i) = C^p$, the parent reserve is depleted, τ_i will have to wait for the next replenishment and may get the resource at the end of the guaranteed time-frame time as shown in Figure 5(b). The completion time of τ_i due to the pre-used resource is larger than the previous case (the C^p phase offset) if τ_i takes time less than $2T^p - \phi$ to finish its job. Otherwise, τ_i finishes at the same time in both cases. □

.



(a) Effects of the unsynchronized phase



(b) Effects from $PR(\tau_i) = C^p$

Figure 5: Critical Instant

Theorem 4. *For a given set of n tasks with $\sum_{i=2}^{n} C_i \geq C^p$, the hierarchical deadline-monotonic scheduling algorithm with deferrable server replenishment is feasible if and only if all real-time tasks sharing a parent reserve have period greater than $2(T^p - C^p)$.*

Proof.

The result of Theorem 3 shows that if $PR(\tau_i) = C^p$, task τ_i has to wait for $2(T^p - C^p)$. For any task τ_j, $PR(\tau_i) = C^p$ if and only if $\sum_{j>i} C_j \geq C^p$. Let us consider the highest-priority task τ_1

(which has the shortest deadline). If $\sum_{j>2} C_j \geq C^p$, it has to wait for the resource at least until time $2(T^p - C^p)$. The scheduling thus cannot guarantee any timing requirement of τ_1 if its period is less than $2(T^p - C^p)$. □

The constraints in Theorem 4 strongly suggest that child reserves should have their period larger than $2(T^p - C^p)$. In practice, the parent reserve capacity C^p is more likely to be changed than its period T^p. Changing C^p can affect the schedulability of previously guaranteed child reserves. To maintain the stability of our reserve guarantees, child reserves should have their periods greater than two times the period of the parent reserve ($2T^p$).

Note: For a given set of n tasks with $T_i \geq 2T^p$, $1 \leq i \leq n$, task τ_i schedulability is the same after time $2T^p - C^p$. Therefore, as long as the period of a task τ_i is greater than $2T^p$, one can use the worst-case completion time formula of either case for a schedulability test.

5.2 The Worst-Case Response Time of DS-HDM

Theorem 5. *For the Hierarchical Deadline-Monotonic scheduling using the deferrable server replenishment and enforcement, if $T_i > 2T^p$ for all i, the schedulability of task τ_i that shares the parent reserve $\{C^p, T^p, D^p\}$ with tasks τ_1, τ_2, ..., τ_n where the $P(\tau_1) < P(\tau_2) < ... < P(\tau_n)$ can be determined by the following procedures:*

- *Find the smallest solution of the equation:*

$$\omega^{n+1} = C_i + \sum_{j<i} \lceil \frac{\omega^n + J_j}{T_j} \rceil C_j + \lceil \frac{\omega^n + C^p}{T^p} \rceil (T^p - C^p) \tag{7}$$

  This formula has to be solved recursively starting with $\omega^0 = (T^p - C^p) + C_i$ and finishing when $\omega^{n+1} = \omega^n$ or $\omega^{n+1} > D_i$. [3]

Proof.

The critical zone of the task, which shown in Figure 5(b) has the same pattern as one in Sporadic Server case where $PR(\tau_i) = C^p$. Substituting the value of $PR(\tau_i)$ in equation (3), we obtain the theorem. □

[3] Note again that if $D^p < T^p$, we have to change the term $T^p - C^p$ to $D^p - C^p$

5.3 Utilization Bound of HDM

The utilization bound of the hierarchical rate-monotonic scheduler on a parent reserve with utilization U^p, which is C^p/T^p, is given by

$$U = n((\frac{3-U^p}{3-2U^p})^{\frac{1}{n}} - 1) \tag{8}$$

where n is the number of its children reserves. As n approaches ∞, the utilization bound becomes

$$\lim_{n\to\infty} U = ln(\frac{3-U^p}{3-2U^p}) \tag{9}$$

The proof of this bound is beyond the scope of this paper and is shown in our CMU technical report [14]. It is to be noted that the Liu and Layland bound [9] is obtained when $U^p = 1$.

6 System Implementation and Evaluation

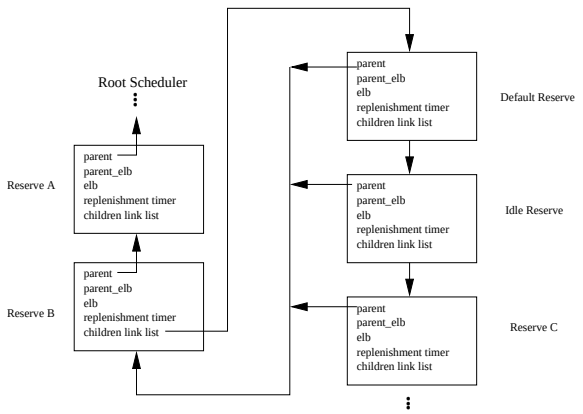


Figure 6: The Core Reservation Structure in our Hierarchical Reservation Implementation on Linux/RK

We have implemented the Hierarchical Deadline-Monotonic reservation model on Linux/RK, CMU's version of Linux with Resource Kernel (RK) functionality. In this implementation, each reserve in Resource Kernel (RK) has its own replenishment timer. A budget of the reserve is replenished every period. We extend this reserve structure to have the parent reference and the link list of its own children as illustrated in Figure 6. The eligibility of this reserve is determined by (1) whether the reserve has its available budget (`elb` for *eligible flag*) and (2) whether all of its ancestors (`parent_elb`) have an available budget to use. The enforcement of this scheme uses only *one* global enforcement timer which is used by the *current* executing reserve, a reserve that is associated with the *current* process. This enforcement timer is set to the minimum of the available budget among the reserve

generations: itself and its ancestor reserves. For example, assuming a parent hard-reserve (20, 50) has only one child hard-reserve (40, 500), when a task attached to the child reserve is first executed at t_0, the enforcement timer is set to min(20, 40) = 20. The task is halted after it executes for 20 time units, $t_0 + 20$. It will be able to run after the parent reserve is replenished, $t_0 + 50$. The task, hence, finishes its job after it gets the second replenishment and will be able to use the resource again after time $t_0 + 500$ due to the depletion of the child reserve.

6.1 Resource Sharing among Default Reserves

If a reserve underuses its resource, its unclaimed resource is transferred to the default reserve of its parent reserve. In addition to collecting the unclaimed resource, the default reserve is responsible for "collecting" all unreserved resources in the parent reserve domain. To achieve this functionality, (1) the same replenishment and enforcement mechanisms used in regular reserves are used, and (2) all tasks using the default reserve are treated as the lowest-priority tasks compared to other tasks associated with its sibling reserves. Therefore, tasks associated with the default reserve will be executed when all of its ancestor reserves are eligible and no other task associated with an eligible reserve is ready. We adapt the *goodness* mechanism implemented in the Linux Kernel to use the `rt_priority` to discriminate the resource among default reserves. The `rt_priority` of a reserve is determined relative to its parent reserve's priority. Therefore, tasks associated with the default reserve of a higher-priority parent reserve is always executed before tasks associated with a default reserve of a lower-priority parent reserve as long as both are eligible to run. It is worth noting that with this mechanism, we also provide provide fairness among non-real-time applications who share the same default reserve.

6.2 The API

A synopsis of the application programming interface we use in hierarchical reservation is presented below. The names of these system calls make them self-explanatory.

```
*resource_set rk_resource_set_create(name);
*rsv rk_CPU_reserve_create(*parent_rsv, *resource_set, cpu_reserve_attr);
*resource_set rk_get_default_rs(*parent_rsv);
*resource_set rk_get_idle_rs(*parent_rsv);
int rk_resource_set_attach_process(*resource_set, *task);
int rk_resource_set_destroy(*resource_set);
```

6.3 Performance Evaluation

In this section, we report the evaluation of our hierarchical reservation scheme implemented on Linux/RK based on linux 2.2.14. All measurements were performed on a Pentium II 400 MHz laptop

Table 1: The reservation parameters used in the experiments

Task	Reserve	Own reserve				The parent reserve			
		C(ms)	T(ms)	D(ms)	U(%)	C(ms)	T(ms)	D(ms)	U(%)
$task1$	R1	10	50	50	20	-	-	-	-
$task2$	R4	60	250	250	24	40	100	100	40
$task3$	R5	70	800	800	8.75	20	100	100	20
$task4$	D_RSV1	-	-	-	-	40	100	100	40
$task5$	D_RSV2	-	-	-	-	20	100	100	20

with 64 MB of memory. We illustrate the following basic points in these experiments:

1. Temporal isolation among multiple levels of hierarchy.

2. Fair bandwidth-sharing among default reserves.

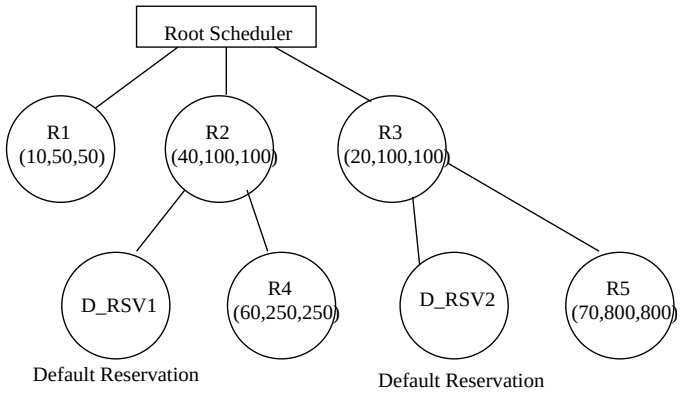


Figure 7: The Hierarchical Reserves used in the Experiment.

In the experiments, five reservations were created as shown in Figure 7. The default reserve D_RSV1 (and D_RSV2) is created automatically when the reserve $R4$ ($R5$) is created from the parent reserve $R2$ ($R3$). Five tasks, τ_1 to τ_5 were spawned and associated with reserves listed in Table 1. We set all tasks to be continuous-loop CPU-consuming tasks because we want to show the effectiveness of our resource enforcement. Figure 8 shows the snapshot of the execution schedule 10 seconds after the tasks start executing. As we expected, the CPU usage of $task1, task2$ and $task3$ are shaped into periodic forms as specified in their associated reserves (its own reserve and its parent reserve). For example, $task2$ starts executing at time 10002 and is enforced by its parent reserve, RSV_1, after executing for $40ms$, at time 10060 due to the enforcement of its parent reserve, etc. Figure 9(a) shows the CPU utilization of $task1$, $task2$ and $task3$. All tasks get resources exactly as they requested and obtained guarantees for.

We now consider the behavior of $task4$ and $task5$ which is associated with the default reserves D_RSV1 and D_RSV2 respectively. Figure 9(b) shows the CPU utiization each task gets in each

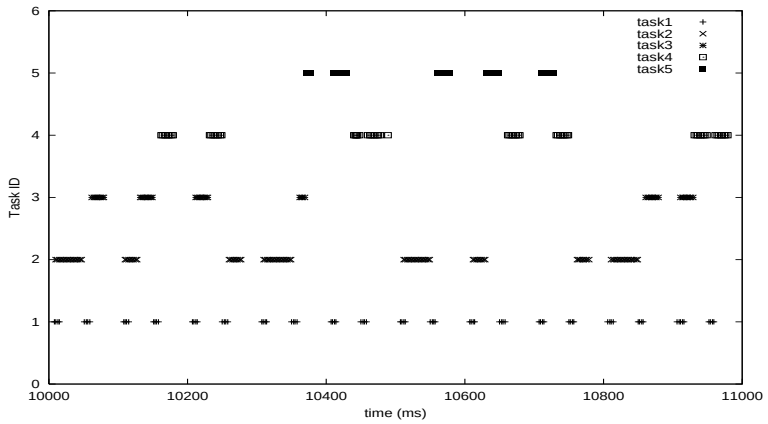
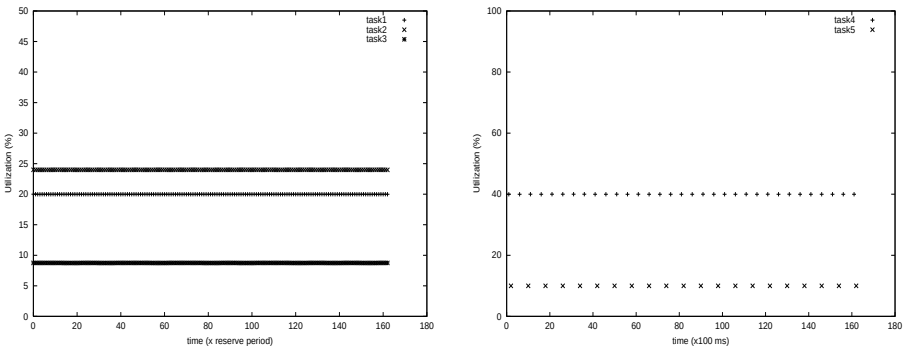


Figure 8: Snapshot of task scheduling



(a) tasks with regular reserves

(b) tasks with default reserves

Figure 9: The CPU Utilization of tasks in the experiment

parent period, $100ms$. As expected, $task4$ and $task5$ get the leftovers from $task2$ and $task3$ which cannot be greater than their parent reserves' quota (40% and 20%) respectively. Note that this utilization is respect to the parent reserve's period ($RSV2$ and $RSV3$) and not the child reserves'. Therefore, during some parent periods, $task4$ can get zero utilization if all resources are dedicated to $task2$, etc.

7 Concluding Remarks

Reservation-based real-time operating systems provide applications with guaranteed and timely access to resources. One of their chief benefits is obtaining temporal isolation across applications. Each resource reservation can be considered to be a virtual resource on which, in turn, the reservation paradigm can be recursively applied. In this paper, we defined, analyzed, implemented and evaluated

such a hierarchical reservation scheme based on the deadline-monotonic scheduling policy. Each node in this hierarchy can also be used to support a different scheduling policy, as well as both real-time and non-real-time applications simultaneously.

Reservation schemes use a replenishment scheme, much like aperiodic servers that are designed to service aperiodic tasks. Our analysis of a hierarchical reservation scheme is performed on replenishment policies defined by both deferrable servers and sporadic servers. A least upper bound of processor utilization under hierarchical rate-monotonic scheduling is also provided. We have also designed and implemented this hierarchical reservation model in Linux/RK, a real-time version of Linux with Resource Kernel (RK) functionality. The set of new system calls added and the new OS structures needed are described briefly. An evaluation of the system shows that the model behaves as predicted, with hierarchical temporal isolation properties and fair sharing of resource bandwidth across default reserves. This model needs to be extended along multiple fronts including support for the sharing of logical resources, and support for multiple resource types including disk bandwidth and network bandwidth. Extensive performance and overhead measurements will also be required to determine the maximum height of the reservation hierarchy that can be acceptable in practice.

References

[1] L. Abeni, G. Lipari, and G. Buttazzo. Constant bandwidth vs proportional share resource allocation. In *Proceedings of the IEEE International Conference on Mutimedia Computing and Systems*, Florence, Italy, June 1999.

[2] Jon C.R. Bennett and Hui Zhang. wf^2q: Worst-case fair weighted fair queueing. In *INFOCOM*, San Francisco, California, March 1996.

[3] Guillem Bernat and Alan Burns. New results on fixed priority aperiodic servers. In *IEEE Real Time System Symposium*, Phoenix, Arizona, December 1999.

[4] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open envirovment. In *IEEE Real-Time Systems Symposium*, December 1997.

[5] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Ninth Euromicro Workshop on Real-Time Systems*, 1997.

[6] Tei-Wei Kuo and Ching-Hui Li. A fixed-priority-driven open environment for real-time applications. In *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.

[7] John P. Lehoczky, Lui Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-ime environment. In *IEEE Real Time System Symposium*, San Jose, California, December 1987.

[8] Giuseppe Lipari and Sanjoy Baruah. A hierarchical extension to the constant bandwidth server framework. In *The 7th IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, May 2001.

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

[10] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *In Proceedings of the IEEE International Conference on Multimedia Computing and Systems Computing System*, May 1994.

[11] Akihiko Miyoshi and Ragunathan Rajkumar. Protecting resources with resource control list. In *The 7th IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, May 2001.

[12] Shui Oikawa and Raj Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Vancouver, June 1999.

[13] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[14] Saowanee Saewong and Raj Rajkumar. Hierarchical reservations in resource kernels. Technical report, Electrical and Computer Engineering CMU, 2001.

[15] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1:27–60, 1989.

[16] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1), January 1995.

[17] Konur Unyelioglu and Raj Rajkumar. Hierarchical reservations. Technical report, Information Networking Institute CMU, 1997.