Rate-Based Resource Allocation Models for Embedded Systems[*]

Kevin Jeffay
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175 USA
jeffay@cs.unc.edu

Steve Goddard
Computer Science & Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588-0115 USA
goddard@cse.unl.edu

Abstract: Run-time executives and operating system kernels for embedded systems have long relied exclusively on static priority scheduling of tasks to ensure timing constraints and other correctness conditions are met. Static priority scheduling is easy to understand and support but it suffers from a number of significant shortcomings such as the complexity of simultaneously mapping timing and importance constraints onto priority values. Rate-based resource allocation schemes offer an attractive alternative to traditional static priority scheduling as they offer flexibility in specifying and managing timing and criticality constraints. This paper presents a taxonomy of rate-based resource allocation and summarizes the results of some recent experiments evaluating the real-time performance of three allocation schemes for a suite of intra-kernel and application-level scheduling problems encountered in supporting a multimedia workload on FreeBSD UNIX.

1. Introduction

Run-time executives and operating system kernels for embedded systems have long relied on static priority scheduling of tasks to ensure timing constraints and other correctness conditions are met. In static priority scheduling tasks are assigned an integer priority value that remains fixed for the lifetime of the task. Whenever a task is made ready to run (*e.g.*, when the arrival of an interrupt releases a waiting task), the active task with the highest priority commences or resumes execution, preempting the currently executing task if need be. There is rich literature that analyzes static priority scheduling and demonstrates how timing and synchronization constraints can be met using static priority scheduling (see [14] for a good summary discussion). For these and other reasons virtually all commercial real-time operating systems, including VxWorks [27], VRTX [17], QNX [22], pSOSystem (pSOS) [21], and LynxOS [16], support static priority scheduling.

However, despite the popularity and simplicity of static priority scheduling, the method has significant shortcomings. As discussed in greater detail in Section 2, static priority scheduling suffers from a number of problems including:

- An inability to directly map timing or importance constraints into priority values,
- The problem of dealing with tasks whose execution time is either unknown or may vary over time,
- The problem of dealing with tasks whose execution time or execution rate deviates significantly at run-time from the behavior expected at design-time,
- The problem of degrading system performance gracefully in times of overload, and

[*] This work supported in parts by grants from the National Science Foundation (grants CDA-9624662, ITR-0082870, and ITR-0082866), and the IBM and Intel corporations.

- The problem of ensuring full utilization of the processor or other system resources in tightly resource constrained systems.

As a solution to these and other problems, we are investigating the use of rate-based resource allocation methods for real-time and embedded systems. In a rate-based system a task is guaranteed to make progress according to a well-defined rate specification such as "process *x* samples per second," or "process *x* messages per second where each message consists of 3-5 consecutive network packets."

Recently a number of rate-based resource allocation paradigms have been developed and reported in the real-time systems and multimedia computing literature. These include:

- The "constant-bandwidth" abstraction for a server algorithm for executing aperiodic workloads [2, 23, 24],
- The Lottery [28], *SMART* [19], *SFQ* [6], *EEVDF* [25], and *BERT* [3] variants of proportional share real-time resource allocation in UNIX, and
- A series of rate-based extension to the Liu and Layland theory of real-time scheduling [5, 7, 9, 29].

This paper summarizes recent developments in rate-based resource allocation and informally demonstrates how rate-based methods can provide a framework for the natural specification and realization of timing constraints in embedded and real-time systems. To structure the discussion, three dimensions of the basic resource allocation problem are considered:

- *The type of resource allocation problem.* To fully explore the utility of rate-based resource allocation three scheduling problems are considered: application-level scheduling (*i.e.*, scheduling of user programs or tasks/threads), scheduling the execution of system calls made by applications ("top-half" operating system-level scheduling), and scheduling asynchronous events generated by devices ("bottom-half" operating system-level scheduling). This treatment is motivated by the logical structure of traditional, monolithic real-time (and general purpose) operating systems and kernels with hardware enforced protection boundaries. Moreover, this simple taxonomy of scheduling problems emphasizes that in real systems one must consider issues of intra-kernel resource allocation and scheduling as well as application task scheduling.
- *The type or class of rate-based resource allocation method*. We consider three broad classes of rate-based resource allocation paradigms: allocation based on a fluid-flow paradigm, allocation based on a polling or periodic server paradigm, and allocation based on a generalized Liu and Layland paradigm. For an instance of the fluid-flow paradigm the proportional share scheduling algorithm *earliest eligible virtual deadline first* (*EEVDF*) [25] is considered. For an instance of the polling server paradigm, scheduling based on the *constant bandwidth server* (*CBS*) server concept [2] is considered. Finally, for the generalized Liu and Layland paradigm a rate-based extension to the original Liu and Layland task model called *rate-based execution* (*RBE*) [9] is considered.
- *The characteristics of the workload generated.* For each rate-based allocation scheme above, the likely expected real-time performance of an application is considered under a set of execution environments where the applications execute at various rates. Specifically, cases are considered wherein the applications execute at "well-behaved," constant rates, at bursty rates, and at uncontrolled "misbehaved" rates. For well-behaved workloads, the three rate-based schemes considered (and indeed virtually any real-time scheduling scheme) can execute a given workload in real-time. However, the rate-based schemes perform quite differently when applications need to be scheduled at bursty or uncontrolled rates.

The goal is demonstrate how rate-based methods naturally solve a variety of resource allocation problems that traditional static priority scheduling methods are inherently poorly suited to. However, it will also be argued that “one size does not fit all.” One rate-based resource allocation scheme does not suffice for all scheduling problems that arise within the layers of a real system. While one can construct an execution environment wherein all of the rate-based schemes considered here perform well, for more realistic environments that are likely to be encountered in practice, the best results are likely to be achieved by employing different rate-based allocation schemes at different levels in the operating system.

To be sure, rate-based resource allocation methods are not a panacea. There are other scheduling architectures based on extensions to (or indirect uses of) static priority scheduling that can also provide effective solutions for many of the problems considered herein (see [14]), however, because of space constraints, in this paper only rate-based methods are considered.

The following section describes the shortcomings of static priority scheduling in more detail. Section 3 presents a taxonomy of rate-based resource allocation methods. Three classes of rate-based resource allocation are described and the strengths and weaknesses of each approach are discussed. The results of some recent experiments performed to evaluate specific instances of rate-based schedulers are reviewed in Section 4. Section 5 proposes and evaluates a hybrid scheme that combines different forms of rate-based resource allocation within different layers of the operating system and application. Section 6 summaries the results and concludes with a discussion of directions for further study in rate-based resource allocation.

2. Traditional Static Priority Scheduling

Traditional models of real-time resource allocation are based on the concept of a discrete but recurring event, such as a periodic timer interrupt, that causes the release of task. The task must be scheduled such that it completes execution before a well-defined deadline. For example, most real-time models of execution are based on the Liu and Layland periodic task model [15] or Mok’s sporadic task model [18]. In these models each execution of a task must complete before the next instance of the same task is released. The challenge is to assign priority values to tasks such that all releases of all tasks are guaranteed to complete execution before their respective deadlines when scheduled by a preemptive priority scheduler. Common priority assignment schemes include the *rate-monotonic* scheme [15] wherein tasks that recur at high rates have priority over tasks that recur at low rates (*i.e.*, a task’s priority is equal to its recurrence period), and the *deadline monotonic* scheme [13] wherein a task’s priority is related to its response time requirement (it’s deadline).[1] In either case, static assignment of priority values to tasks leads to a number of problems.

2.1. Mapping performance requirements to priorities

Simple timing constraints for tasks that are released in response a single event, such as a response time requirement for a specific interrupt, can be easily specified in a static priority system. More complex constraints are frequently quite difficult to map into priority values. For example, consider a signal processing system operating on video frames that arrive over a LAN from a remote acquisition device. The performance requirement may be to process 30 frames/second and hence it would be natural to model the processing as a periodic task and trivial to schedule using a static

[1] We assume throughout that low priority values indicate high scheduling priority.

priority scheduler. However, it is easily the case that each video frame arrives at the processing node in a series of network packets that must be reassembled into a video frame and it is not at all obvious how the network packet and protocol processing should be scheduled. That is, while there is a natural constraint for the application-level processing, there is no natural constraint (*e.g.*, no unique response time requirement) for the processing of the individual events that will occur during the process of realizing the higher-level constraint. Nonetheless in a static priority scheduler the processing of these events must be assigned a single priority value.

The problem here is that the system designer implicitly creates a response time requirement when assigning a priority value to the network processing. Since there is no natural unique value for this requirement a conservative (*e.g.*, short response time) value is typically chosen. This conservative assignment of priority has the effect of reserving more processing resources for the task than will actually ever be consumed and ultimately limits the either the number or the complexity of tasks that can be executed on the processor.

2.2. Managing "misbehaved" tasks

In order to analyze any resource allocation problems, assumptions must be made about the environment in which the task executes. In particular, in virtually all real-time problems the amount of resources required for the execution of the task (*e.g.*, the amount of processor time required to execute the task) is assumed to be known in advance. A second problem with static priority resource allocation occurs when assumptions such as these are violated and a task "misbehaves" by consuming more resources than expected. The problem is to ensure that a misbehaving task does not compromise the execution of other "well-behaved" tasks in the system.

An often-touted advantage of static priority systems is that if a task violates its execution assumptions, higher priority tasks are not affected. While it is true that higher priority tasks are not affected by misbehaving lower priority tasks (unless higher priority tasks share software resources with the lower priority tasks), all other tasks have no protection from the misbehaving task. For example, a task that is released at a higher rate than expected can potentially block *all* lower priority tasks indefinitely.

The issue is that static priority scheduling fundamentally provides no mechanism for isolating tasks from the ill-effects of other tasks other than the same mechanism that is used to ensure response time properties. Given that isolation concerns typically are driven by the relative importance of tasks (*e.g.*, a non-critical task should never effect or interfere with the execution of a mission-critical task), and importance and response time are often independent concepts, attempting to manage both concerns with a single mechanism is inappropriate.

2.3. Providing graceful/uniform degradation

Related to the task isolation problem is that of providing graceful performance degradation under transient (or persistent) overload conditions. The problem of graceful degradation can be considered a generalization of the isolation problem; a set of tasks (or the environment that generates work for the tasks) misbehaves and the processing requirements for the system as a whole increase to the point where tasks miss deadlines. In these overload situations it is again useful to control which tasks' performance degrades and by how much.

Static priority scheduling again has only a single mechanism for managing importance and response time. If the mission-critical tasks also have the smallest response-time requirements then they will have the highest priority and will continue to function. However, if this is not the case then there is no separate mechanism to control the execution of important tasks. Worse, even if the priority structure matches the importance structure, in overload conditions under static priority scheduling only one form of degraded performance is possible: high priority tasks execute normally while the lowest priority task executes at a diminished rate if at all. That is, since a task with priority p will *always* execute to completion before any pending task with priority less than p commences or resumes execution, it is impossible to control how tasks degrade their execution. (Note however, as frequently claimed by advocates of static priority scheduling, the performance of a system under overload conditions is predictable.)

2.4. Achieving full resource utilization

The rate-monotonic priority assignment scheme is well known to be an optimal static priority assignment. (Optimal in the sense that if a static priority assignment exists which guarantees periodic tasks have a response time no greater than their period, then the rate-monotonic priority assignment will also provide the same guarantees.) One drawback to static priority scheduling, however, is that the achievable processor utilization is restricted. In their seminal paper, Liu and Layland showed that a set of n periodic tasks will be schedulable under a rate-monotonic priority assignment if the processor utilization of a task set does not exceed $n(2^{1/n} - 1)$ [15]. If the utilization of a task set exceeds this bound then the tasks may or may not be schedulable. (That is, this condition is a sufficient but not necessary condition for schedulability). Moreover, Liu and Layland showed that in the limit the utilization bound approached ln 2 or approximately 0.69. Thus 69% is the least upper bound on the processor utilization that guarantees feasibility. A least upper bound here means that this is the minimum utilization of all task sets that fully utilize the processor. (A task set fully utilizes the processor if increasing the cost of any task in the set causes a task to miss a deadline.) If the utilization of the processor by a task set is less than or equal to 69% then the tasks are guaranteed to be schedulable.

Lehoczky, Sha, and Ding formulated an exact text for schedulabilty under a rate-monotonic priority assignment and showed that ultimately, schedulability is not a function of processor utilization [12]. However, nonetheless, in practice the utilization test remains the dominant test for scheduability as it is both a simple and an intuitive test. Given this, a resource allocation scheme wherein scheduability was more closely tied to processor utilization (or a similarly intuitive metric) would be highly desirable.

3. A Taxonomy of Rate-Based Resource Allocation Models

The genesis of rate-based resource allocation schemes can be traced to the problem of supporting multimedia computing and other soft-real-time problems. In this arena it was observed that while one could support the needs of these applications with traditional real-time scheduling models, these models were not the most natural ones to apply [6, 8, 11, 28]. Whereas Liu and Layland models typically dealt with response time guarantees for the processing of periodic/sporadic events, the requirements of multimedia applications were better modeled as aggregate, but bounded, processing rates.

From our perspective three classes of rate-based resource allocation models have evolved: *fluid-flow allocation*, *server-based allocation*, and *generalized Liu and Layland style allocation*. Fluid-flow allocation derives largely from the work on fair-share bandwidth allocation in the networking community. Algorithms such as *generalized* *processor sharing* (GPS) [20], *packet-by-packet generalized processor sharing* (PGPS) [20] (better known as *weighted fair queuing* (WFQ) [4]), were concerned with allocating network bandwidth to connections ("flows") such that for a particular definition of fairness, all connections continuously receive their fair share of the bandwidth. Since connections were assumed to be continually generating packets, fairness was expressed in terms of a guaranteed transmission rate (*i.e.*, some number of bits per second). These allocation policies were labeled as "fluid flow" allocation because since transmission capacity was continuously available to be allocated, analogies were drawn between conceptually allowing multiple connections to transmit packets on a link and allowing multiple "streams of fluid" to flow through a "pipe."

These algorithms stimulated tremendous activity in both real-time CPU and network link scheduling. In the CPU scheduling realm numerous algorithms were developed, differing largely in the definition and realization of "fair allocation" [19, 25, 28]. Although fair/fluid allocation is in principle a distinct concept from real-time allocation, it is a powerful building block for realizing real-time services [26].

Server-based allocation derives from the problem of scheduling aperiodic tasks in a real-time system. The salient abstraction is that a "server process" is invoked periodically to service any requests for work that have arrived since the previous invocation of the server. The server typically has a "capacity" for servicing requests (usually expressed in units of CPU execution time) in any given invocation. Once this capacity is exhausted, any in-progress work is suspended until at least the next server invocation time. Numerous server algorithms have appeared in the literature; differing largely in the manner in which the server is invoked and how its capacity is allocated [2, 23, 24]. Server algorithms are considered to be rate-based forms of allocation as the execution of a server is not (in general) directly coupled with the arrival of a task. Moreover, server-based allocation has the effect of ensuring aperiodic processing progresses at a well defined, uniform rate.

Finally, rate-based generalizations of the original Liu and Layland periodic task model have been developed to allow more flexibility in how a scheduler responds to events that arrive at a uniform average rate but unconstrained instantaneous rate. Representative examples here include the (m, k) allocation models that requires only m out of every k events be processed in real-time [5], the *window-based allocation* (DWYQ) method that ensures a minimum number of events are processed in real-time within sliding time windows [29], and the *rate-based execution* (RBE) algorithm that "reshapes" the deadlines of events that arrive at a higher than expected rate to be those that the events would have had had they arrived at a uniform rate [9].

To illustrate the utility of rate-based resource allocation, one algorithm from the literature from each class of rate-based allocation methods will be discussed in more detail. The choice is motivated by the prior work of the authors, specifically our experience implementing and using these algorithms in production systems [7, 10]. For an instance of the fluid-flow paradigm the proportional share scheduling algorithm *earliest eligible virtual deadline first* (*EEVDF*) [25] will be discussed. For an instance of the polling server paradigm the *constant bandwidth server* (*CBS*) server concept [2] will be discussed. For the generalized Liu and Layland paradigm the *rate-based execution* (*RBE*) model [9] will be discussed. Although specific algorithms are chosen for discussion, ultimately the results presented are believed to hold for each algorithm in the same class as the algorithm discussed.

3.1. Fluid-Flow Resource Allocation: Proportional Share Scheduling

In a proportional share (*PS*) system each shared resource r is allocated in discrete quanta of size at most q_r. At the beginning of each time quantum a task is selected to use the resource. Once the task acquires the resource, it may use the resource for the entire time quantum, or it may release the resource before the time quantum expires. For a given resource, a *weight* is associated with each task that determines the relative *share* of the resource that the task should receive. Let w_i denote the weight of task i, and let $A(t)$ be the set of all tasks active at time t. Define the (instantaneous) share $f_i(t)$ of task i at time t as

$$f_i(t) = \frac{w_i}{\sum_{j \in A(t)} w_j} \tag{1}$$

A share represents a fraction of the resource's capacity that is "reserved" for a task. If the resource can be allocated in arbitrarily small sized quanta, and if the task's share remains constant during any time interval $[t_1, t_2]$, then the task is entitled to use the resource for $(t_2 - t_1)f_i(t)$ time units in the interval. As tasks are created/destroyed or blocked/released, the membership of $A(t)$ changes and hence the denominator in (1) changes. Thus in practice, a task's share of a given resource will change over time. As the total weight of tasks in the system increases, each task's share of the resource decreases. As the total weight of tasks in the system decreases, each task's share of the resource increases. When a task's share varies over time, the service time S that task i should receive in any interval $[t_1, t_2]$, is

$$S_i(t_1, t_2) = \int_{t_1}^{t_2} f_i(t)\, dt \tag{2}$$

time units.

Equations (1) and (2) correspond to an ideal "fluid-flow" system in which the resource can be allocated for arbitrarily small units of time. In such a system tasks make progress at a uniform rate as if they were executing on a dedicated processor with a capacity that is $f_i(t)$ that of the actual processor. In practice one can implement only a discrete approximation to the fluid system. When the resource is allocated in discrete time quanta it is not possible for a task to always receive exactly the service time it is entitled to in all time intervals. The difference between the service time that a task should receive at a time t, and the time it actually receives is called the service time lag (or simply lag). Let t_0 be the time at which task i becomes active, and let $s(t_0, t)$ be the service time task i receives in the interval $[t_0, t]$. Then if task i is active in the interval $[t_0, t]$, its lag at time t is defined as

$$lag_i(t) = S_i(t_0^i, t) - s_i(t_0^i, t). \tag{3}$$

Since the lag quantifies the allocation accuracy, it is used as the primary metric for evaluating the performance of *PS* scheduling algorithms. Previously we have shown that one can schedule a set of tasks in a *PS* system using a "virtual time" *earliest deadline first* rule such that the lag is bounded by a constant over all time intervals [25]. By using this algorithm, called *earliest eligible virtual deadline first* (*EEVDF*), a *PS* system's deviation from a system with perfectly uniform allocation is bounded and thus, as explained below, real-time execution is possible.

Scheduling to Minimize Lag

The goal in proportional share scheduling is to minimize the maximum possible lag. This is done by conceptually tracking the lag of tasks and at the end of each quantum, considering only tasks whose lag is positive [25]. If a task's lag is positive then it is "behind schedule" compared to the perfect fluid system — it should have accumulated more time on the CPU than it has up to the current time. If a task's lag is positive it is considered eligible to execute. If its lag is negative, then the task has received more processor time than it should have up to the current time and it is considered ineligible to execute

When multiple tasks are eligible, in *EEVDF* they are scheduled earliest deadline first, where a task's deadline is equal to its estimated execution time cost divided by its share of the CPU, $f_i(t)$. This deadline represents a point in the future when the task should complete execution if it receives exactly its share of the CPU. For example, if a task's weight is such that its share of the CPU at the current time is 10% and it requires 2 *ms* of CPU time to complete execution, then its deadline will be 20 *ms* in the future. If the task actually receives 10% of the CPU, over the next 20 *ms* it will execute for 2 *ms*.

Proportional share allocation is realized through a form of weighted round-robin scheduling wherein in each round the task with the earliest deadline is selected. In [25] it was shown that the *EEVDF* algorithm provides optimal (*i.e.*, minimum possible) lag bounds.

Realizing Real-Time Execution

In principle, there is nothing "real-time" about proportional share resource allocation. Proportional share resource allocation is concerned solely with *uniform* allocation (often referred to in the literature as "fluid" or "fair" allocation). A *PS* scheduler achieves uniform allocation if it can guarantee that tasks' lags are always bounded.

Real-time computing is achieved in a *PS* system by (*i*) ensuring that a task's share of the CPU (and other required resources) remains constant over time, and by (*ii*) scheduling tasks such that each task's lag is always bounded by a constant. If these two conditions hold over an interval of length t for a task i, then task i is guaranteed to receive $(f_i \times t) \pm \varepsilon$ units of the resource's capacity, where f_i is the fraction of the resource reserved for task i, and ε is the allocation error, $0 \le \varepsilon \le \delta$, for some constant δ (for *EEVDF* δ = the quantum size q) [25]. Thus, although real-time allocation is possible, it is not possible to provide hard and fast guarantees of adherence to application-defined timing constraints. Said another way, all guarantees have an implicit, and fundamental, "$\pm\,\varepsilon$" term. In FreeBSD-based implementations of *EEVDF*, ε has been a tunable parameter and was most commonly set to 1 *ms* [7].

The deadline-based *EEVDF* proportional share scheduling algorithm ensures that each task's lag is bounded by a constant [25] (condition (*i*)). To ensure a task's share remains constant over time (condition (*ii*)), whenever the total weight in the system changes, a "real-time" task's weight must be adjusted so that its initial share (as given by equation (1)) does not change. For example, if the total weight in the system increases (*e.g.*, because new tasks are created), then a real-time task's weight must increase by a proportional amount. Adjusting the weight to maintain a constant share is simply a matter of solving equation (1) for w_i when $f_i(t)$ is a constant function. (Note that w_i appears in both the numerator and denominator of the right-hand side of (1).) If the sum of the processor shares of the real-time tasks is less than 1.0 then all tasks will execute in real-time (*i.e.*, under *EEVDF* real-time scheduablity is a simple function of processor utilization).

3.2. Liu and Layland Extensions: Rate-Based Execution (*RBE*)

The traditional Liu and Layland model of periodic real-time execution has been extended in a number of directions to be more flexible in way in which real-time requirements were modeled and realized. For example, all of the traditional theory assumes a minimum separation in time between releases of instances of the same task. This requirement does not map well in actual systems that, for example, receive inputs over a network. For example, in a video processing application, video frames may be transmitted across an internetwork at precise intervals but arrive at a receiver with arbitrary spacing in time because of the store-and-forward nature of most network switches. Although there is explicit structure in this problem (frames are generated at a precise rate), there is no way to capture this structure in a Liu and Layland model.

The *RBE* paradigm is one extension to the Liu and Layland model to address this problem. In *RBE*, each task is associated with three parameters (*x*, *y*, *d*) which define a rate specification. In an *RBE* system, each task is guaranteed to process at least x events every y time units, and each event j will be processed before a relative deadline d. The actual deadline for processing of the j^{th} event for task i is given by the following recurrence. If t_{ij} is the time of the arrival of the j^{th} event, then the instance of task i servicing this event will complete execution before time:

$$D_i(j) = \begin{cases} t_{ij} + d_i & \text{if } 1 \le j \le x_i \\ \max(t_{ij} + d_i, D_i(j - x_i) + y_i) & \text{if } j > x_i \end{cases} \qquad (4)$$

The deadline for the processing of an event is the larger of the release time of the job plus its relative deadline, or the deadline of the x^{th} previous job plus the y parameter (the averaging interval) of the task. This deadline assignment function confers two important properties on *RBE* tasks. First, up to x consecutive jobs of a task may contend for the processor with the same deadline and second, the deadlines for processing events j and $j+x$ for task i are separated by at least y time units. Without the latter restriction, if a set of events for a task arrive simultaneously it would be possible to saturate the processor. However, with the restriction, the time at which a task must complete its execution is not wholly dependent on its release time. This is done to bound processor demand. Under this deadline assignment function, requests for tasks that arrive at a faster rate than x arrivals every y time units have their deadlines postponed until the time they would have been assigned had they actually arrived at the rate of exactly x arrivals every y time units [9].

The *RBE* task model derives from the *linear bounded arrival process* (*LBAP*) model as defined and used in the DASH system [1]. In the *LBAP* model, tasks specify a desired execution rate as the number of messages to be processed per second, and the size of a buffer pool used to store bursts of messages that arrive for the task. The *RBE* model generalizes the *LBAP* model to include a more generic specification of rate and adds an independent response time (relative deadline) parameter to enable more precise real-time control of task executions.

RBE tasks can be scheduled by a simple *earliest-deadline-first* rule so long as the combined processor utilization of all tasks does not saturate the processor. (Hence there are no undue limits on the achievable processor utilization.) Although nothing in the statement of the *RBE* model precludes static priority scheduling of tasks, it turns out that the *RBE* model points out a fundamental distinction between deadline-based scheduling methods and static priority based methods. Analysis of the *RBE* model has shown that under deadline-based scheduling, feasibility is solely a function of the distribution of task deadlines in time and is independent of the rate at which tasks are invoked. In contrast, the opposite is true of static priority schedulers. For *any* static priority scheduler, feasibility is a function of the rate at which tasks are invoked and is independent of the deadlines of the tasks [9]. Said more simply, the feasibility of static priority schedulers is solely a function of the periodicity of tasks, while the feasibility of deadline schedulers is solely a function of the periodicity of the occurrence of a task's deadlines. Given that it is often the operating system or application that assigns deadlines to tasks, this means that the feasibility of a static priority scheduler is a function of the behavior of the external environment (*i.e.* arrival processes), while the feasibility of a deadline driven scheduler is a function of the implementation of the operating system/application. This is a significant observation as one typically has more control over the implementation of the operating system than they do over the processes external to the system that generate work for the system. For this reason deadline based scheduling methods have a significant and fundamental advantage over static priority based methods when there is uncertainty in the rates at which work is generated for a real-time system, such as is the case in virtually all distributed real-time systems.

3.3. Server-Based Allocation: The Constant Bandwidth Server (*CBS*)

The final class of rate-based resource allocation algorithms are server algorithms. At a high-level, the *CBS* algorithm combines aspects of both *EEVDF* and *RBE* scheduling (although it was developed independently of both works). Like *RBE* it is an event based scheduler, however, like *EEVDF* it has a notion of a quantum. Like both, it achieves rate-based allocation by a form of deadline scheduling

In *CBS*, and its related cousin the *total bandwidth server* (*TBS*) [23, 24], a portion of the processor's capacity, denoted U_S, is reserved for processing aperiodic requests of a task. When an aperiodic request arrives it is assigned a deadline and scheduled according to the *earliest-deadline- first* algorithm. However, while the server executes, its capacity linearly decreases. If the server's capacity for executing a single request is exhausted before the request finishes, the request is suspended until the next time the server is invoked.

A server is parameterized by two additional parameters C_S and T_S, where C_S is the execution time available for processing requests in any single server invocation and T_S is the inter-invocation period of the server ($U_S = C_S/T_S$). Effectively, if the k^{th} aperiodic request arrives at time t_k, it will execute as a task with a deadline

$$d_k = \max(t_k, d_{k-1}) + c_k/U_S \qquad (5)$$

where c_k is the worst case execution time of the k^{th} aperiodic request, d_{k-1} is the deadline of the previous request from this task, and U_S is the processor capacity allocated to the server for this task.

CBS resource allocation is considered a rate-based scheme because deadlines are assigned to aperiodic requests based on the rate at which the server can serve them and not (for example) on the rate at which they are expected to arrive. Note that like *EEVDF* and *RBE*, scheduability in *CBS* is solely a function of processor utilization. Any real-time task set that does not saturate the processor is guaranteed to execute in real-time under any of these three algorithms.

4. Using Rate-Based Scheduling

To see how rate-based resource allocation methods can be used to realize real-time constraints and overcome the shortcoming of static priority scheduling, the three algorithms above were used to solve various resource allocation problems that arose in FreeBSD UNIX when executing a set of interactive multimedia applications. The details of this study are reported in [10]. The results are summarized here.

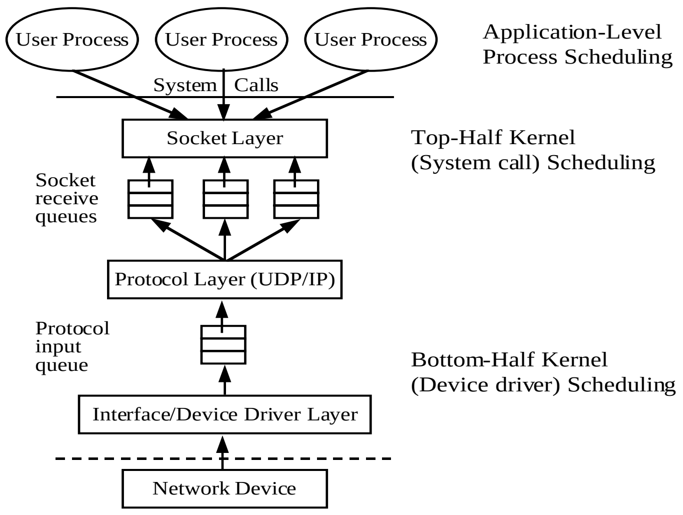

Figure 1: Architectural diagram of UDP/IP protocol processing in FreeBSD.

4.1. A Sample Real-Time Workload

To compare the rate-based schedulers, three simple multimedia applications were considered. These applications were chosen because they illustrate many of the essential resource allocation problems found in distributed real-time and embedded applications. The applications are:

- An Internet telephone application that handles incoming 100 byte audio messages at a rate of 50/second and computes for 1 millisecond on each message (requiring 5% of the CPU on average),
- A motion-JPEG video player that handles incoming 1,470 byte messages at a rate of 90/second and computes for 5 milliseconds on each message (requiring 45% of the CPU on average), and
- A file transfer program that handles incoming 1,470 byte messages at a rate of 200/second and computes for 1 millisecond on each message (requiring 20% of the CPU on average).

The performance of different rate-based resource allocation schemes was considered under varying workload conditions. The goal was to evaluate how each rate-based allocation scheme performed when the rates of tasks to be scheduled varied from constant (uniform), to “bursty” (erratic instantaneous rate but constant average rate), to “mis-behaved” (long-term deviation from average expected processing rate).

4.2. Rate-Based Scheduling of Operating System Layers

Our experiments focused on the problem of processing inbound network packets and scheduling user applications to consume these packets. Figure 1 illustrates the high-level architecture of the FreeBSD kernel. Briefly, in FreeBSD, packet processing occurs as follows. (For a more complete description of these functions see [30].) When packets arrive from the network, interrupts from the network interface card are serviced by a device-specific interrupt handler. The device driver copies data from buffers on the adapter card into a chain of fixed-size kernel memory buffers sufficient to hold the entire packet. This chain of buffers is passed on a procedure call to a general interface input routine for a class of input devices. This procedure determines which network protocol should receive the packet and enqueues the packet on that protocol's input queue. It then posts a software interrupt that will cause the protocol layer to be executed when no higher priority hardware or software activities are pending.

Processing by the protocol layer occurs asynchronously with respect to the device driver processing. When the software interrupt posted by the device driver is serviced, a processing loop commences wherein on each iteration the buffer chain at the head of the input queue is removed and processed by the appropriate routines for the transport protocol. This results in the buffer chain enqueued on the receive queue for the destination socket. If any process is blocked in a kernel system call awaiting input on the socket, it is unblocked and rescheduled. Normally, software interrupt processing returns when no more buffers remain on the protocol input queue.

The kernel socket layer code executes when an application task invokes some form of receive system call on a socket descriptor. When data exists on the appropriate socket queue, the data is copied into the receiving task's local buffers from the buffer chain(s) at the head of that socket's receive queue. When there is sufficient data on the socket receive queue to satisfy the current request, the kernel completes the system call and returns to the application task.

The problem of processing inbound network packets was chosen for study because it involves a range of resource allocation problems at different layers in the operating system. Specifically, there are three distinct scheduling problems: scheduling of device drivers and network protocol processing within the operating system kernel, scheduling system calls made by applications to read and write data to and from the network, and finally the scheduling of user applications. These are distinct problems because the schedulable work is invoked in different ways in different layers. Asynchronous events cause device drivers and user applications to be scheduled but synchronous events cause system calls to be scheduled. Systems calls are, in essence, extensions of the application's thread of control into the operating system. Moreover, these problems are of interest because of the varying amount of information that is available to make real-time scheduling decisions at each level of the operating system. At the application and system call-level it is known exactly which real-time entity should be "charged" for use of system resources while at the device driver-level one cannot know which entity to charge. For example, in the case of inbound packet processing, it cannot be determined which application to charge for the processing of a packet until the packet is actually processed and the destination application is discovered. On the other hand, the cost of device processing can be known exactly as device drivers typically perform simple, bounded-time functions (such as placing a string of buffers representing a packet on a queue). This is in contrast to the application-level where often one can only estimate the time required for an application to complete.

The challenge is to allocate resources throughout the operating system so that end-to-end system performance measures (*i.e.*, network interface to application performance) can be ensured.

4.3. Workload Performance Under Proportional Share Allocation

A version of FreeBSD was constructed that used *EEVDF* proportional share scheduling (with a 1 *ms* quantum) at the device, protocol processing, and the application layers. In this system each real-time task is assigned a share of the CPU equal to its expected utilization of the processor (*e.g.*, the Internet phone application requires 5% of the CPU and hence is assigned a weight of 0.05). Non-real-time tasks are assigned a weight equal to the unreserved capacity of the CPU. Network protocol processing is treated as a real-time (kernel-level) task that processes a fixed number of packets when it is scheduled for execution.

In the well-behaved senders case all packets are moved from the network interface to the socket layer to the application and processed in real-time. When the file transfer sender misbehaves and sends more packets than the *ftp* receiver can process given its CPU reservation, *EEVDF* does a good job of isolating the other well-behaved processes from the ill-effects of *ftp*. Specifically, the excess *ftp* packets are dropped at the lowest level of the kernel (at the IP layer) before any significant processing is performed on these packets. These packets are dropped because the kernel task associated with their processing is simply not scheduled often enough (by design) to move the packets up to the socket layer. In a static priority system (such as the unmodified FreeBSD system), network interface processing is the second highest priority task. In this system, valuable time would be “wasted” processing packets up through the kernel only to have them later discarded (because of buffer overflow) because the application wasn’t scheduled often enough to consume them.

The cost of this isolation comes in the form of increased packet processing overhead as now the device layer must spend time demultiplexing packets (typically a higher-layer function) to determine if they should be dropped. Performance is also poorer when data arrives for all applications in a bursty manner. This is an artifact of the quantum-based allocation nature of *EEVDF*. Over short intervals, data arrives faster than it can be serviced at the IP layer and the IP interface queue overflows. With a 1 *ms* quantum, it is possible that IP processing can be delayed for upwards of 8-10 *ms* and this is sufficient time for the queue to overflow in a bursty environment. This problem could be ameliorated to some extent by increasing the length of the IP queue, however, this would also have the effect of increasing the response time for packet processing.

4.4. Workload Performance Under Generalized Liu and Layland Allocation

When *RBE* scheduling was used for processing at the device, protocol, and application layers, each task had a simple rate specification of (1, *p*, *p*) (*i.e.*, one event will be processed every *p* time units with a deadline of *p*) where *p* is the period of the corresponding application or kernel function.

Perfect real-time performance is realized in the well-behaved and bursty arrivals cases but performance is significantly poorer than *EEVDF* in the case of the misbehaved file transfer application. On the one hand, RBE provides good isolation between the file transfer and the other real-time applications, however, this isolation comes at the expense of the performance of non-real-time/background applications. Unlike *EEVDF*, as an algorithm, *RBE* has no mechanism for directly ensuring isolation between tasks as there is no mechanism for limiting the amount of CPU time an *RBE* task consumes. All events in an *RBE* system are assigned deadlines for processing. When the work arrives at a faster rate than is expected, the deadlines for the work are simply pushed further and further out in time. Assuming sufficient buffering, all will eventually be processed.

In the FreeBSD system, packets are enqueued at the socket layer but with deadlines that are so large that processing is delayed such that the socket queue quickly fills and overflows. Because time is spent processing packets up to the socket layer that are never consumed by the application, the performance of non-real-time applications suffers. Because of this, had the real-time workload consumed a larger cumulative fraction of the CPU, *RBE* would not have isolated the well-behaved and misbehaved real-time applications. (That is, the fact that isolation was observed in these experiment is an artifact of the specific real-time workload.)

Nonetheless, because the *RBE* scheduler assigns deadline to all packets, and because the system under study was not overloaded, *RBE* scheduling results in the smaller response times for real-time events than seen under *EEVDF* (at the cost non-real-time task performance).

4.5. Workload Performance Under Server Based Allocation

When *CBS* server tasks were used for processing throughout the operating system and at the application layers, each server task was assigned a capacity equal to its application's/function's CPU utilization. The server's period was made equal to the expected interarrival time of data (packets). Non-real-time tasks were again assigned to a server with capacity equal to the unreserved capacity of the CPU.

As expected, performance is good when work arrives in a well-behaved manner. In the case of the misbehaved file transfer, *CBS* also does a good job of isolating the other well-behaved tasks. The excess *ftp* packets are dropped at the IP layer and thus little overhead is incurred. In the case of bursty senders *CBS* scheduling outperforms *EEVDF* scheduling. This is because like *RBE*, scheduling of *CBS* tasks is largely event driven and hence *CBS* tasks respond quicker to the arrival of packets. Under *EEVDF* the rate at which the IP queue can be serviced is a function of the quantum size and the number of processes currently active (which determines the length of a scheduling "round" [4]). In general these parameters are not directly related to the real-time requirements of applications. Under *CBS* the service rate is a function of the server's period which is a function of the expected arrival rate and thus is a parameter that is directly related to application requirements. For the choices of quantum size for *EEVDF*, and server period for *CBS*, good performance under *CBS* and poor performance under *EEVDF* results. However, we conjecture that is likely the case that these parameters could be tuned to reverse this result.

Although *CBS* outperforms *EEVDF* in terms of throughput, the results are mixed for response times for real-time tasks. Even when senders are well behaved some deadlines are missed under *CBS*. This results in a significant number of packets being processed later than with *EEVDF* or *RBE* scheduling. This is problematic since in these experiments the theory predicts that no deadlines should be missed. The cause of the problem here relates to the problem of accounting for the CPU time consumed when a *CBS* task executes. In the implementation of *CBS*, the capacity of a *CBS* task is updated only when the task sleeps or is awaken by the kernel. Because of this, many other kernel related functions that interrupt servers (*e.g.*, Ethernet driver execution) are inappropriately charged to *CBS* tasks and hence bias scheduling decisions. This accounting problem is fundamental to the server-based approach and cannot be completely solved without significant additional mechanism (and overhead).

5. Hybrid rate-based scheduling

The results of applying a single rate-based resource allocation policy to the problems of device, protocol, and application processing were mixed. When processing occurs at rates that match the underlying scheduling model (*e.g.*, when work arrives at a constant rate), all the policies considered achieved real-time performance. When work arrives for an application that exceeds the application's rate specification or resource reservation, then only the *CBS* server-based scheme and the *EEVDF* proportional share scheme provide true isolation between well-behaved and misbehaved applications. When work arrives in a bursty manner, the quantum-based nature of *EEVDF* leads to less responsive protocol processing and more (unnecessary) packet loss. *CBS* performs better but suffers from the complexity of the CPU-time accounting problem that must be solved. *RBE* provides the best response times but only at the expense of decreased throughput for the non-real-time activities. The obvious question is whether or not there is utility in applying different rate-based resource allocation schemes in different layers of the kernel to better match the solution to a particular resource allocation problem to the characteristics of the problem.

To test this conjecture two hybrid rate-based FreeBSD systems were constructed. For application and system call level processing *EEVDF* scheduling was used. This choice was made because the quantum nature of *EEVDF*, while bad for intra-kernel resource allocation, is a good fit given the existing round-robin scheduling architecture in FreeBSD (and many other operating systems such as Linux). It is easy to implement and to precisely control and gives good real-time response when schedulable entities execute for long periods relative to the size of a quantum. For device and protocol processing inside the kernel both *CBS* and *RBE* scheduling were considered. Since the lower kernel layers operate more as an event driven system, a paradigm which takes into account the notion of event arrivals is appropriate. Both of these policies are also well-suited for resource allocation within the kernel because, in the case of *CBS*, it is easier to control the levels and degrees of preemption within the kernel and hence it is easier to account for CPU usage within the kernel (and hence easier to realize the results predicted by the *CBS* theory). In the case of *RBE*, processing within the kernel is more deterministic and hence *RBE*'s inherent inability to provide isolation between tasks that require more computation than they reserved is less of a factor.

The forms of hybrid rate-based resource allocation described here remains the topic of active study. Preliminary results show that when work arrives at well-behaved rates both *CBS+EEVDF* and *RBE+EEVDF* systems perform flawlessly. (Thus hybrid allocation performs no worse than the universal application of a single rate-based scheme throughput the system.) In the misbehaved *ftp* application case, both hybrid implementations provide good isolation, comparable to the best single-policy systems. However, in both hybrid approaches, response times and deadline miss ratios are now much improved. In the case of bursty arrivals, all packets are eventually processed and although many deadlines are missed, both hybrid schemes miss fewer deadlines than did the single-policy systems. Overall the *RBE+EEVDF* system produces the lowest overall deadline miss ratios. While we do not necessarily believe this is a fundamental result (*i.e.*, there are numerous implementation details to consider), it is the case that the polling nature of the *CBS* server tasks increases response times over the direct event scheduling method of *RBE*.

6. Summary, Conclusions, and Future Work

Rate-based resource allocation schemes are a good fit for providing real-time services in distributed real-time and embedded systems. Allocation schemes exist that are a good fit for the scheduling architectures used in the various layers of a traditional monolithic UNIX kernel such as FreeBSD. Three such rate-based schemes were considered: the *earliest eligible virtual deadline first* (*EEVDF*) fluid-flow paradigm, the *constant bandwidth server* (*CBS*) polling server paradigm, and the generalization of Liu and Layland scheduling known as *rate-based execution* (*RBE*). We compared their performance for three scheduling problems found in FreeBSD: application-level scheduling of user programs, scheduling the execution of system calls made by applications in the "top-half" of the operating system, and scheduling asynchronous events generated by devices in the "bottom-half" of the operating system. For each scheduling problem we considered the problem of network packet and protocol processing for a suite of canonical multimedia applications. We tested each implementation under three workloads: a uniform rate packet arrival process, a bursty arrival process, and a misbehaved arrival process that generates work faster than the corresponding application process can consume it.

The results were mixed. When work arrives at rates that match the underlying scheduling model (the constant rate senders case), all the policies we considered achieve real-time performance. When work arrives that exceeds the application's rate specification, only the *CBS* server-based scheme and the *EEVDF* proportional share scheme provide isolation between well-behaved and misbehaved applications. When work arrives in a bursty manner, the quantum-based nature of *EEVDF* gives less responsive protocol processing and more packet loss. *CBS* performs better but suffers from CPU-time accounting problems that result in numerous missed deadlines. *RBE* provides the best response times but only at the expense of decreased throughput for the non-real-time activities.

We next investigated the application of different rate-based resource allocation schemes in different layers of the kernel and considered *EEVDF* proportional share scheduling of applications and system calls combined with either *CBS* servers or *RBE* tasks in the bottom half of the kernel. The quantum nature of *EEVDF* scheduling proves to be well suited to the FreeBSD application scheduling architecture and the coarser-grained nature of resource allocation in the higher-layers of the kernel. The event driven nature of *RBE* scheduling gives the best response times for packet and protocol processing. Moreover, the deterministic nature of lower-level kernel processing avoids the shortcomings observed when *RBE* scheduling is employed at the user-level.

In summary, we conclude that more research is needed on the design of rate-based resource allocation schemes that are tailored to the requirements and constraints of individual layers of an operating system kernel. All of the schemes we tested worked well for application-level scheduling (the problem primarily considered by the developers of each algorithm). However, for intra-kernel resource allocation, these schemes give significantly different results. By combining resource allocation schemes we are able to alleviate specific shortcomings, however, this is likely more accidental than fundamental as none of these policies were specifically designed for scheduling activities within the kernel. By studying these problems in their own right, significant improvements should be possible.

References

1. D. Anderson, Tzou, S., Wahbe, R., Govindan, R., Andrews, M., Support for Live Digital Audio and Video, Proc. 10th Intl. Conf. on Distributed Computing Systems, Paris, France, May 1990, pp. 54-61.
2. L. Abeni, G. Buttazzo, *Integrating Multimedia Applications in Hard Real-Time Systems*, Proc. of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998, pp. 4-13.

3. A. Bavier, and L.L. Peterson, *BERT: A Scheduler for Best Effort and Real-time Tasks*, Technical Report, Department of Computer Science, Princeton University, 2001.
4. A. Demers, S. Keshav, and S. Shenkar, *Analysis and Simulation of a Fair Queueing Algorithm*, *Jour. of Internetworking Research & Experience*, October 1990, pp. 3-12.
5. M. Hamdaoui and P. Ramanathan. *A dynamic priority assignment technique for streams with (m,k)-firm deadlines*, IEEE Transactions on Computers, April 1995.
6. P. Goyal, X. Guo, H. Vin, *A Hierarchical CPU Scheduler for Multimedia Operating Systems*, USENIX Symp. on Operating Systems Design & Implementation, Seattle, WA, Oct. 1996, pp. 107-121.
7. K. Jeffay, F. D. Smith, A. Moorthy, J. Anderson, *Proportional Share Scheduling of Operating Systems Services for Real-Time Applications*, Proc. of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998, pp. 480-491.
8. K. Jeffay, D. Bennett, *Rate-Based Execution Abstraction for Multimedia Computing*, Proc. of the Fifth Intl. Workshop on Network & Operating System Support for Digital Audio & Video, Durham, NH, April 1995, *Lecture Notes in Computer Science*, Vol. 1018, pp. 64-75, Springer-Verlag, Heidelberg.
9. K. Jeffay, S. Goddard, *A Theory of Rate-Based Execution*, Proc. 20th IEEE Real-Time Systems Symposium, Dec. 1999, pp. 304-314.
10. K. Jeffay, G. Lamastra, A Comparative Study of the Realization of Rate-Based Computing Services in General Purpose Operating Systems, Proceedings of the Seventh IEEE International Conference on Real-Time Computing Systems and Applications, Cheju Island, South Korea, December 2000, pages 81-90.
11. M.B. Jones, D. Rosu, M.-C. Rosu, *CPU Reservations & Time Constraints: Efficient, Predictable Scheduling of Independent Activities*, Proc., Sixteenth ACM Symposium on Operating Systems Principles, Saint-Malo, France, October 1997, pp. 198-211.
12. Lehoczky, J., Sha, L., Ding, Y., The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior, Proc. of the 10th IEEE Real-Time Systems Symp., Santa Monica, CA, December 1989, pp. 166-171.
13. J. Leung, and J. Whitehead, On the complexity of fixed-priority scheduling of periodic, real-time tasks, Performance Evaluation, 2, 1982, pp. 237-50.
14. J.W. S. Liu, *Real-Time Systems*, Prentice Hall, 2000.
15. C. L. Liu and J. W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, Journal of the ACM, Vol. 20, No. 1, January 1973, pp. 46-61.
16. LynuxWorks, *LynxOS and BlueCat real-time operating systems*, *http://www.lynx.com/index.html*.
17. Mentor Graphics, *VRTX Real-Time Operating System*.
18. A.K.-L., Mok, *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Thesis, MIT, Dept. of EE and CS, MIT/LCS/TR-297, May 1983.
19. J. Nieh, M.S. Lam, *Integrated Processor Scheduling for Multimedia*, Proc. 5th Intl. Workshop on Network and Operating System Support for Digital Audio &

Video, Durham, N.H., April 1995, Lecture Notes in Computer Science, T.D.C. Little & R. Gusella, eds., Vol. 1018, Springer-Verlag, Heidelberg.

20. A. K. Parekh and R. G. Gallager, *A Generalized Processor Sharing Approach To Flow Control in Integrated Services Networks-The Single Node Case*, *ACM/IEEE Transactions on Networking*, Vol. 1, No. 3, 1992, pp. 344-357.
21. *pSOS+TM/68K Real-Time Executive*, User's Manual, Motorola, Inc.
22. *QNX Operating System*, System Architecture and Neutrino System Architecture Guide, QNX Software Systems Ltd, 1999.
23. M. Spuri, G. Buttazzo, *Efficient Aperiodic Service Under the Earliest Deadline Scheduling*, Proc. 15th IEEE Real-Time Systems Symp., Dec. 1994, pp. 2-11.
24. M. Spuri, G. Buttazzo, F. Sensini, *Robust Aperiodic Scheduling Under Dynamic Priority Systems*, Proc. 16th IEEE Real-Time Systems Symp., Dec. 1995, pp. 288-299.
25. I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, C. Plaxton, *A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems*, Proc. 17th IEEE Real-Time Systems Symposium, Dec. 1996, pp. 288-299.
26. I. Stoica, H. Abdel-Wahab, K. Jeffay, *On the Duality between Resource Reservation and Proportional Share Resource Allocation*, Multimedia Computing & Networking '97, SPIE Proceedings Series, Vol. 3020, Feb. 1997, pp. 207-214.
27. *VxWorks Programmer's Guide*, WindRiver System, Inc., 1997.
28. C.A. Waldspurger, W.E. Weihl, *Lottery Scheduling: Flexible Proportional-Share Resource Management*, Proc. of the First Symp. on Operating System Design and Implementation, Nov. 1994, pp. 1-12.
29. R. West, K. Schwan, and C. Poellabauer, *Scalable scheduling support for loss and delay constrained media streams*, Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium, Vancouver, Canada, June 1999.
30. G.R. Wright, W.R. Stevens, *TCP/IP Illustrated, Volume 2, The Implementation*, Addison-Wesley, Reading MA, 1995.