A Comparative Study of the Realization of Rate-Based Computing Services in General Purpose Operating Systems*

Kevin Jeffay
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175 USA

Gerardo Lamastra
ReTiS Lab
Scuola Superiore di Studi Universitari e Perfezionamento S.Anna, Pisa, Italy

Abstract: *Scheduling architectures that support a rate abstraction are becoming increasingly popular for realizing real-time services in general-purpose operating systems. While many rate-based schemes have been proposed, there has been little discussion of the relative merits of each approach. We study the performance of a set of multimedia applications under three different rate-based scheduling schemes implemented in the FreeBSD operating system: a proportional share scheme (Earliest Eligible Virtual Deadline First scheduling), a polling, server-based scheme (the Constant Bandwidth Server), and a rate-based extension to the original Liu and Layland task model (Rate-Based Execution). Furthermore, we consider three specific scheduling problems: scheduling application level tasks, scheduling system calls, and scheduling the kernel-level processing of data input from devices such as network interfaces. Based on empirical evidence, we conclude that "one size does not fit all" — that no one rate-based resource allocation scheme suffices for all scheduling problems along the data path from the device to an application. Rather, we achieve the best performance for our multimedia workload when we apply different rate-based scheduling policies at different layers of the operating system such as proportional share scheduling of system calls and application tasks, and rate-based Liu and Layland scheduling of device processing.*

1. Introduction

Rate-based models of real-time execution have recently become popular for scheduling real-time workloads on general purpose operating systems. This is because the processing requirements of soft real-time applications such as multimedia conferencing are typically better expressed in terms of an average processing rate such as "display 30 frames per second," than as a set of event response time requirements such as "process every video frame within 33 *ms* of its arrival." That is, in the world of real-time computing on the desktop, real-time processing requirements are frequently expressed as higher-level, more abstract, and softer application-level requirements. Moreover, these requirements often are not directly supported by the traditional Liu and Layland style periodic task models [14] that have been the cornerstone of real-time systems research.

To support this new class of rate-based communication and computation services, numerous algorithms and scheduling frameworks have been developed. These include:

- The resource reservation abstraction in Real-Time Mach [16, 21, 26] and the Rialto operating system [11],
- The vertically integrated nature of the Nemesis [12], and RED-Linux operating systems [29],
- The "constant-bandwidth" abstraction for a server algorithm for executing aperiodic workloads [1, 22, 23],
- The Lottery [27, 28], SMART [19], SFQ [7], and EEVDF [24] variants of proportional share real-time resource allocation in UNIX, and
- The rate-based extension to the Liu and Layland theory of real-time scheduling [8, 9].

All of these works have demonstrated the utility of a particular paradigm of rate-based resource allocation for meeting the requirements of soft and hard real-time applications under a given set of conditions. While these absolute demonstrations of success have been compelling, we are interested in understanding the relative merits of using one rate-based scheme over another for the various resource allocation problems that arise in a general purpose operating system such as FreeBSD UNIX. That is, while rate-based resource allocation schemes have been shown to be superior to traditional Liu and Layland schemes, to the best of our knowledge, rate-based schemes have never been compared to one another.

Towards this end, we have conducted an implementation study to compare the relative merits of existing rate-based resource allocation schemes such as those listed above. We have two goals in this effort. First, we seek to understand the implementation costs and application/operating system performance under the various scheduling schemes. Second, we believe that it is not likely to be the case that one resource allocation scheme will suffice for all scheduling problems that arise in a general purpose operating system. We therefore also seek to understand the appropriateness of applying various classes of rate-based scheduling schemes within the various layers of a traditional monolithic UNIX operating system kernel.

In this paper we present the results of an empirical investigation of these problems. We provide a comparison of the relative costs and performance improvements under various rate-based scheduling schemes. There are three dimensions to our study:

* Work supported by grants from the National Science Foundation (grants CCR-9510156, ITR-0082870, & ITR-0082866) and the Intel Corporation.

In: *Proceedings of the 7th IEEE International Conference on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, December 2000, pages 81-90.

- *The type of resource allocation problem.* We consider both application-level and operating system-level resource allocation. Specifically, we consider the performance of rate-based resource allocation schemes for three resource allocation problems: scheduling user programs (application-level scheduling), 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).
- *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 each class we implemented a representative instance from the literature. For an instance of the fluid-flow paradigm we implemented the proportional share scheduling algorithm *earliest eligible virtual deadline first* (*EEVDF*) [24]. For an instance of the polling server paradigm we implemented a scheduler based on the *constant bandwidth server* (*CBS*) server concept [1]. For the generalized Liu and Layland paradigm we implemented a rate-based extension to the original Liu and Layland task model called *rate-based execution* (*RBE*) [10].
- *The characteristics of the workload generated.* For each rate-based allocation scheme above, we compare the real-time performance of a set of distributed multimedia applications under a set of execution environments where the applications execute at various rates. Specifically, we consider cases wherein the applications execute at "well-behaved," constant rates, at bursty rates, and at uncontrolled "misbehaved" rates.

Our results show that for well-behaved workloads, each rate-based allocation scheme we consider executes the workload in real-time. There are differences in the overheads associated with each scheme, however, these differences are likely strongly influenced by our implementation and may or may not be fundamental. The rate-based schemes perform quite differently, however, when applications need to be scheduled at bursty or uncontrolled rates. The Liu and Layland *RBE* extension does not isolate well-behaved tasks from the effects of misbehaving tasks and leads to lower throughput for the multimedia applications. Moreover, applications miss numerous deadlines when scheduled by a *CBS* server. The proportional share scheme performs slightly better but results in significantly poorer response times for those tasks that do miss a deadline. That is, although fewer deadlines are missed under the proportional share scheme than under the *CBS* scheme, those tasks that miss a deadline in the proportional share scheme complete later than tasks missing a deadline under *CBS*.

Our results also confirm our expectations that "one size does not fit all." One resource allocation scheme does not suffice for all scheduling problems that arise within the layers of a general purpose UNIX operating system. While one can construct an execution environment wherein all of the rate-based schemes we consider perform well, for more realistic environments that are likely to be encountered in practice, the best results are achieved by employing different rate-based allocation schemes at different levels in the operating system. Specifically, we find that for scheduling device drivers and low-level kernel processing such as network packet and protocol processing, the rate-based extensions to the original Liu and Layland model provide lower latency and fewer deadline misses than other schemes. For scheduling user applications as well as the system calls made by these applications, the proportional share schemes give the best results.

In total, our results show that while rate-based scheduling schemes remain a good solution for soft real-time computing, the straightforward reduction to practice of the theory and the universal application of a single scheduling policy does not provide the optimum results. In this manner this work contributes to our understanding of how rate-based services can be realized in a general purpose operating system.

The following section describes the problem we are addressing and our approach in more detail. Section 2 also reviews the rate-based resource allocation literature in more detail. Section 3 describes our experimental evaluation environment and the experiments we perform. Section 4 presents results for the three classes of rate-based resource allocation we are considering. Section 5 proposes and evaluates a hybrid scheme that combines different forms of rate-based resource allocation within different levels of the operating system. The results are summarized in Section 6.

2. Background and Related Work

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 [14] or Mok's sporadic task model [18].

With the advent of multimedia computing and other soft-real-time problems, 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, 7, 9, 11, 27]. 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) [3]), were concerned with allocating network bandwidth to connections ("flows") such that for a particular definition of fair, 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, 24, 27]. 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 [25].

Server-based allocation derives from the problem of scheduling aperiodic processes 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 [1, 22, 23]. 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, [30], 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 [10].

For our study we will chose one algorithm from the literature from each class of rate-based allocation methods. The choice is motivated by the prior work of the authors, specifically our ability to gain access to high-quality implementations of each algorithm. As we comment in Section 6, we do not believe the results we ultimately show are a function of our choice of individual algorithms but are a function of the class of algorithms we use. For an instance of the fluid-flow paradigm we implemented the proportional share scheduling algorithm *earliest eligible virtual deadline first* (*EEVDF*) [24]. For an instance of the polling server paradigm we implemented a scheduler based on the *constant bandwidth server* (*CBS*) server concept [1]. For the generalized Liu and Layland paradigm we implemented the *rate-based execution* (*RBE*) model [10]. The following describes each algorithm in more detail.[1]

2.1 Earliest Eligible Virtual Deadline First

EEVDF is a proportional share resource allocation policy that allocates the CPU in fixed size quanta. The fundamental concept in proportional share allocation is that at all times, each task receives a precise *share* of the CPU. The share of the CPU a task is to receive is a function of the task's *weight*; a system defined parameter. If $A(t)$ represents the set of tasks active in the system at time t, and w_i is the weight of task i, then the share of the CPU that task i should receive at time t is

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

A share represents a fraction of the resource's capacity that is "reserved" for a process. That is, if the process's share remains constant during any time interval $[t_1, t_2]$, then the process is entitled to use the resource for $(t_2 - t_1)f_i(t)$ time units in the interval. Since the set of active tasks, competing for the resource at a given instant is a function of the number of tasks in the system, the denominator in (1) will change over time. Consequently, the actual service time that task i can expect to receive in any interval $[t_1, t_2]$ is

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

Equations (1) and (2) correspond to an ideal "fluid-flow" system in which the resource can be allocated in arbitrarily small units of time. In practice one can implement only a discrete approximation to the fluid system. However, when the resource is allocated in discrete time quanta it is not possible for a process to always receive exactly the service time it is entitled to in all time intervals. The difference between the service time that a process 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 process i becomes active, and let $s(t_0, t)$ be the service time process i receives in the interval $[t_0, t]$. Then if process i is active in the interval $[t_0, t]$, its lag at time t is defined as $lag_i(t) = S_i(t_0, t) - s_i(t_0^i, t)$.

It has been shown that if one can schedule a set of processes in a proportional share system such that the lag is bounded by a constant over all time intervals, then proportional share execution implies real-time execution [25]. Proportional share allocation is realized through a form of weighted round-robin scheduling. In [24] it has been shown that the *EEVDF* algorithm, a proportional share variant of deadline scheduling, provides optimal (*i.e.*, minimum possible) lag bounds and can hence be used for real-time computing. The

---

[1] Because of space considerations, these descriptions are simply meant to give the basic flavor of each algorithm. We assume the reader has some familiarity with these algorithms.

key result is that the lag bounds are a function of the quantum of allocation used, hence real-time guarantees in an *EEVDF* (and all proportional share) systems will have a $\pm q$ error term, where q is the quantum size.

2.2 Constant Bandwidth Server (*CBS*)

The *CBS* server algorithm is also a method of achieving rate-based allocation by a form of deadline scheduling. In *CBS*, and its related cousin the *total bandwidth server* (*TBS*) [22, 23], 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 kth aperiodic request arrives at time t_k, it will execute as a task with a deadline $d_k = \max(t_k, d_{k-1}) + c_k/U_S$ where c_k is the worst case execution time of the kth 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.

2.3 Rate-Based Execution (*RBE*)

The *RBE* paradigm is conceptually similar to the server-based algorithms and also uses a deadline-driven policy in the underlying scheduler. 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 complete execution before a relative deadline D. The actual deadline for processing of the jth event of task i is given by the recurrence:

$$D_i(j) = \begin{cases} t_{ij} + d_i & \text{if } 1 \le j \le x_i \\ \max(t_{ij} + d_i, D_i(j - x_i) + y_i) & \text{if } j > x_i \end{cases} \quad (3)$$

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 [10].

3. Experimental Method and Design

We compare the performance of a distributed multimedia workload on FreeBSD UNIX under various combinations of rate-based resource allocation policies described above. Here we describe the experimental setup and our performance metrics. We begin with a description of our workload.

3.1 Real-Time Workload

To compare the rate-based schedulers, we use three simple multimedia applications that we believe are indicative of the types of real-time and non-real-time processing that is likely to be performed on a general purpose workstation. 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).

Each of these programs consists of a simple main loop consisting of a *read()* operation on a UDP socket bound to a specific port followed by a computation phase with a known execution time. In addition to these three receiving processes we also ran another process that executed the Dhrystone benchmark program to simulate a (background) compute intensive program. Each of these programs was run as a separate process on the modified FreeBSD system described below.

In addition, we configured three separate machines to act as message generators and send messages with the desired size and rate to the corresponding receiving process on the experimental machine. Each machine was responsible for generating a stream of messages of one type (*e.g.*, audio messages), destined for a single process (*e.g.*, the audio phone process) on the machine running the rate-extensions to FreeBSD. The four machines (one for each multimedia data type plus the receiving machine) were all connected to an unloaded 10Mbps Ethernet. The experimental setup is illustrated in Figure 1.

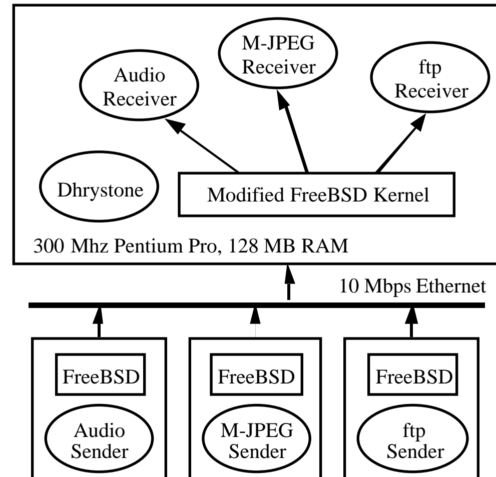

Figure 1: Experimental setup.

With this experimental setup we conducted a number of experiments where we investigated the performance of different rate-based resource allocation schemes under different workloads. The goal was to evaluate how each rate-based allocation scheme performs when the rates of processes to be scheduled varies from constant (uniform), to "bursty" (erratic instantaneous rate but constant average rate), to "misbehaved" (long-term deviation from average expected processing rate).

For each experiment, three variations of the traffic generated by the sending processes were used: (1) all three senders' message transmission rates were constant and uniform, (2) all three senders' message rates were made bursty by selecting a random inter-message delay from a Pareto distribution with a mean equal to the previous uniform constant rate, and (3) the audio and video senders message rates were constant as in (1), but the file sender "misbehaved" and sent messages at a rate of 1,000/second instead of 200/second.

3.2 Rate-Based Scheduling of Operating System Layers

Our experiments focus on the problem of processing inbound network packets and scheduling user applications to consume these packets. Figure 2 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 [31].) 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 (called *mbufs*) sufficient to hold the entire packet. This chain of *mbufs* is passed on a procedure call to a general interface input routine for a class of input devices (*e.g.*, Ethernet). This procedure determines which network protocol (*e.g.*, IP) 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.

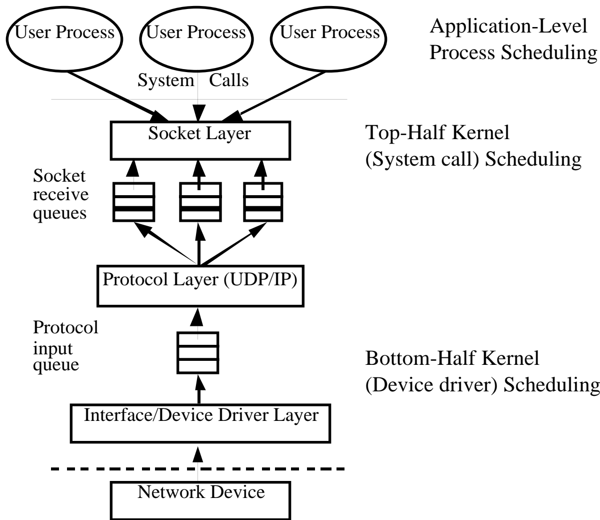

Figure 2: Architectural diagram of UDP protocol processing in FreeBSD.

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 *mbuf* chain at the head of the input queue is removed and processed by the appropriate routines for the transport protocol (*e.g.*, UDP). This results in the *mbuf* 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 *mbufs* remain on the protocol input queue.

The kernel socket layer code executes when a process 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 process's local buffers from the *mbuf* 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 user process.

We chose the problem of processing inbound network packets because it involves a range of resource allocation problems at different layers in the operating system. Specifically, we identify three 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 threads 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, one cannot determine 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, one can know the cost of device processing exactly as device drivers typically perform constant functions (such as placing a string of buffers representing a packet on a queue) while at the application-level one can only at best estimate the time required for an application to complete.

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

3.3 Performance Metrics

Following the experimental design used in [8], we compare the *EEVDF*, *CBS*, and *RBE* resource allocation schemes by examining the performance of network protocol processing under the network conditions described in Section 3.1. In each experiment we report performance measures at the device-driver level ("bottom-half" kernel performance), at the system-call/socket interface level ("top-half" kernel performance), and at the application level. Performance measures of interest are: the number of packets dropped at each level, the number of deadlines missed for the processing of individual packets, and the range on actual response times observed for packet processing. We collect data over one minute intervals. Ideally we would like to observe that:

- When data arrives for each application at a constant rate, all applications process data in real-time (*i.e.*, no data is lost and all application deadlines are respected). The Internet phone would receive 3,000 packets (50×60) in a measurement interval, the video player would receive 5,400 packets, and the file transfer would receive 12,000 packets. The deadlines for the processing of packets will be 20 *ms* for the phone application, 11.1 *ms* for the video application, and 5 *ms* for the *ftp* application.
- When data arrives for an application at a faster rate than the application can process it, the data is discarded at the lowest level of the operating system (the *receive livelock* effect does not occur [4, 17]) and the processing of data for other applications that is arriving at a uniform rate is minimally effected.

To assess the overhead of the various implementations, we measure the number of Dhrystone iterations completed in a measurement interval. This is at best an indirect measure of overhead, however we feel it is nonetheless an insightful measure.

4. Performance Results I: Universal application of each rate-based schemes

We modified the FreeBSD system (2.2.2-RELEASE) to support the *EEVDF*, *CBS*, and *RBE* scheduling algorithms, and ran a suite of experiments to assess the impact of rate-based scheduling on packet and network protocol processing. Our experiments were conducted on a 300Mhz Pentium Pro with 128 MB of memory. The network interface was a 3Com 3C595 (*vx0*) 10/100 Ethernet adapter running at 10Mbps. To begin, we compare the performance of our application workload when all resource allocation in the kernel and at the application level is performed using the same rate-based allocation scheme. In the next section we look at the performance when different rate-based allocation schemes are employed at different levels in the kernel.

As an aside, we note that from an engineering perspective, there are numerous interesting issues concerning the evolution of an operating system with a monolithic structure that uses static priority scheduling, to one that performs integrated rate-based resource allocation (where integration here refers to integration between application layer scheduling and the scheduling of all other layers of the operating system). Some of these issues were discussed in [8]; in this work, we defer a broader discussion of these issues to a future paper.

4.1 Performance Under Proportional Share Allocation

Our first implementation uses *EEVDF* proportional share scheduling at the device, protocol processing, and the application layers. Each real-time task is assigned a share of the CPU equal to its expected utilization of the processor. For example, the Internet phone application requires 5% of the CPU (see Section 3.1) and hence is assigned a weight of 0.05. Non-real-time tasks (the Dhrystone) 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. Each incoming packet stream is serviced according to its expected rate of arrival, using a WFQ-like scheme [3, 8]. The network device driver enqueues packets for different flows in different queue (*i.e.*, it includes a packet classifier). This is done to enable us to bound the ill-effects of "misbehaved" arrival processes. In our implementation of *EEVDF*, based on the results reported in [8], we use a scheduling quantum of 1 *ms*.

Table 1 gives the packet processing performance for the universal application of *EEVDF* scheduling. 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*. This can be seen by noting that 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. In this manner the receive livelock problem is also avoided. The overhead of packet processing in this case goes up (*e.g.*, the time spent demultiplexing packets at the device layer to determine if they should be dropped) and this can be seen in the decrease in Dhrystone iterations. Performance is 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.

Table 2 gives the corresponding response time and deadline miss statistics. Real-time performance is achieved (no deadlines are missed) when traffic is generated at a constant rate. When the file transfer sender misbehaves the overhead of

processing these packets results in an increase in the average response time of all applications and a modest number of deadline misses. Performance is poor under the bursty sender case because of the number of dropped packets and the longer queue lengths at the IP and socket layers.

Table 1: EEVDF performance — throughput and loss rates.

	Constant Rate Senders			Misbehaved *ftp*			Bursty (Pareto) Senders		
	Drops at IP	Drops at socket	Packets Processed	Drops at IP	Drops at socket	Packets Processed	Drops at IP	Drops at socket	Packets Processed
Phone	0	0	2,993	5	0	2,997	1,585	0	1,342
ftp	0	0	11,961	17,999	0	11,902	5,315	0	5,408
M-JPEG	0	0	5,346	56	0	5,390	2,705	0	2,498
Dhrystone	N/A	N/A	5.1E+6	N/A	N/A	3.7E+6	N/A	N/A	14.2E+6

Table 2: Response time results (*ms*) and deadline miss ratios.

	Constant Rate Senders				Misbehaved *ftp*				Bursty Senders			
	Min	Avg.	Max	%Miss	Min	Avg.	Max	%Miss	Min	Avg.	Max	%Miss
Phone	1.9	7.6	12.1	0%	1.9	9.8	31.4	2%	2.6	17.3	56.2	47.8%
ftp	1.9	2.4	4.2	0%	1.9	73.4	172.3	99%	1.9	17.9	100.8	49.6%
M-JPEG	5.2	6.1	9.9	0%	5.2	9.5	21.0	10%	5.8	14.1	41.1	97.2%

4.2 Performance Under Server Based Allocation

The second implementation uses a *CBS* server task for processing at the device, protocol processing, and the application layers. Each task has a capacity equal to its CPU utilization and period equal to the expected interarrival time of packets. The Dhrystone task is again assigned to a server with capacity equal to the unreserved capacity of the CPU. This implementation also uses a packet classifier in the device driver for early demultiplexing of packets into separate IP queues.

Table 3 gives the packet processing performance for the universal application of *CBS* scheduling. Performance is again excellent in the well-behaved senders case. In the case of the misbehaved file transfer, *CBS* also does a good job of isolating the other well-behaved processes. The excess *ftp* packets are dropped at the IP layer and thus receive livelock does not occur. In the case of bursty senders *CBS* scheduling outperforms proportional share scheduling. Under *CBS*, throughput results for the bursty case are nearly identical to the constant rate senders case. This is because *CBS* tasks are event driven and hence can respond quicker to the arrival of packets. That is, 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" [3]). 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; parameters that are directly related to application requirements. For the choices we made for quantum size for *EEVDF*, and server period for *CBS*, we get good performance under *CBS* and poor performance under *EEVDF*. 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. Table 4 shows that considerably more deadlines are missed under *CBS*. Even when senders are well behaved, a significant number of packets are processed late. This is problematic since in this case 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 our implementation the capacity of a *CBS* task is updated only when the task sleeps or is awaken by the kernel, hence many other kernel related functions that interrupt servers (*e.g.*, Ethernet driver execution) are inappropriately charged to 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).

4.3 Performance Under Generalized Liu and Layland Allocation

The third implementation uses *RBE* scheduling for processing at the device, protocol, and application layers. Each task has 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. This has the effect of ensuring that the *RBE* tasks will have the same main scheduling parameters (period and relative deadline parameters) as they did in the *CBS* experiments.

Table 3: CBS performance — throughput and loss rates.

	Constant Rate Senders			Misbehaved *ftp*			Bursty (Pareto) Senders		
	Drops at IP	Drops at socket	Packets Processed	Drops at IP	Drops at socket	Packets Processed	Drops at IP	Drops at socket	Packets Processed
Phone	0	0	2,977	0	0	2,978	0	0	2,938
ftp	2	0	11,914	17,880	0	12,120	5	0	10,760
M-JPEG	0	0	5,388	0	0	5,391	0	0	3,192
Dhrystone	N/A	N/A	3.4E+6	N/A	N/A	2.4E+6	N/A	N/A	4.12E+6

Table 4: Response time results (*ms*) and deadline miss ratios.

	Constant Rate Senders				Misbehaved *ftp*				Bursty Senders			
	Min	Avg.	Max	%Miss	Min	Avg.	Max	%Miss	Min	Avg.	Max	%Miss
Phone	1.1	3.1	6.6	0%	1.2	3.9	30.0	1.9%	1.09	42.2	170.3	68.4%
ftp	1.1	4.1	18.8	34.3%	142	242.0	391.5	100%	1.07	21.3	95.8	82.5%
M-JPEG	5.8	10.0	17.8	35.8%	5.9	21.4	197.1	26.3%	5.8	10.9	36.1	32.2%

Table 5: RBE performance — throughput and loss rates.

	Constant Rate Senders			Misbehaved *ftp*			Bursty (Pareto) Senders		
	Drops at IP	Drops at socket	Packets Processed	Drops at IP	Drops at socket	Packets Processed	Drops at IP	Drops at socket	Packets Processed
Phone	0	0	3,000	0	0	2,998	0	0	3,027
ftp	0	0	11,944	0	9,052	20,794	0	0	10,778
M-JPEG	0	0	5,443	0	0	5,444	0	0	5,287
Dhrystone	N/A	N/A	1.2E+6	N/A	N/A	0.4E+3	N/A	N/A	1.47E+6

Table 6: Response time results (*ms*) and deadline miss ratios.

	Constant Rate Senders				Misbehaved *ftp*				Bursty Senders			
	Min	Avg.	Max	%Miss	Min	Avg.	Max	%Miss	Min	Avg.	Max	%Miss
Phone	2.1	2.4	17.8	0%	1.24	1.3	1.4	0%	1.0	8.6	63.0	10.7%
ftp	1.1	1.3	1.7	0%	73.7	78.0	84.4	100%	1.0	1.1	8.3	0.1%
M-JPEG	6.5	8.3	10.5	0%	5.75	5.9	22.9	0.06%	5.7	6.9	10.4	0%

Table 5 gives the packet processing performance for the universal application of *RBE* scheduling. Performance is excellent in the well-behaved and bursty senders case but poorer in the case of the misbehaved file transfer application. On the one hand, RBE appears to provide good isolation between the file transfer and the other real-time applications, however, this isolation comes at the expense of the non-real-time Dhrystone application. All of the excess *ftp* packets are dropped higher-up in the kernel at the socket layer and the extra time required to process these packets up through the socket layer means there is less time available for the Dhrystone application. In fact, unlike *CBS* or *EEVDF*, *RBE* as an algorithm 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 (packet arrivals in our case) 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. In our case, this means that packets are enqueued at the socket layer but with deadlines that are so large that processing is so delayed processed 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, receive livelock could be possible under *RBE* scheduling. Worse yet, had the real-time workload consumed a larger cumulative fraction of the CPU we would not have seen isolation between the well-behaved and misbehaved real-time applications. (That is, the fact that we observe isolation in this experiment is an artifact of our specific real-time workload.)

Because the *RBE* scheduler assigns deadline to all packets, and because our system is not overloaded, we observe the smallest response times under *RBE*. However, given the number of packets that are dropped at the socket layer, these response time figures are less meaningful. Dhrystone performance is the worst under *RBE* as this task pays the penalty for all of the unnecessary packet processing.

5. Performance Results II: Hybrid rate-based scheduling

The results of applying a single rate-based resource allocation policy to the problems of device, protocol, and application processing are mixed. When processing occurs at rates that match the underlying scheduling model (*e.g.*, the constant rate senders case), all the policies we have considered achieve 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 good 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 hence 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 (unnecessary) decreased throughput for the non-real-time activities. In total, we conclude that there is utility in applying different rate-based resource allocation schemes in different layers of the kernel. We conjecture that the best performing system will result from mixing rate-based schemes.

To test this conjecture we constructed two hybrid rate-based FreeBSD systems. For application and system call level processing we use *EEVDF* scheduling. We make this choice 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. It is easy to implement and to control precisely 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 we consider both *CBS* and *RBE* scheduling. 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.

Tables 7-10 give the throughput, loss, and deadline miss statistics for these *CBS+EEVDF* and *RBE+EEVDF* systems. In the case of constant rate senders, both perform flawlessly. The overhead of the schemes, as measured by the Dhrystone performance, is comparable and only slightly worse than the best performing constant-rate senders system in Section 4 (the *EEVDF* system). In the misbehaved *ftp*

application case, both implementations provide good isolation, comparable to the best systems in Section 4. However, in both the hybrid approaches, response times and deadline miss ratios are now much improved. In the case of bursty senders, all packets are eventually processed and although many deadlines are missed, both hybrid schemes miss fewer deadlines than the systems in Section 4. 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*.

Table 7 *CBS+EEVDF* performance — throughput and loss rates.

	Constant Rate Senders			Misbehaved *ftp*			Bursty (Pareto) Senders		
	Drops at IP	Drops at socket	Packets Processed	Drops at IP	Drops at socket	Packets Processed	Drops at IP	Drops at socket	Packets Processed
Phone	0	0	2,869	0	0	2,797	0	0	2,988
ftp	0	0	11,722	17,898	0	11,545	0	0	10,340
M-JPEG	0	0	5,343	0	0	5,398	0	0	4,951
Dhrystone	N/A	N/A	3.1E+6	N/A	N/A	2.9E+6	N/A	N/A	4.6E+6

Table 8: Response time results (ms) and deadline miss ratios.

	Constant Rate Senders				Misbehaved *ftp*				Bursty Senders			
	Min	Avg.	Max	%Miss	Min	Avg.	Max	%Miss	Min	Avg.	Max	%Miss
Phone	1.2	7.6	14.7	0%	2.5	9.8	24.2	0.03%	1.2	14.2	184.4	19.8%
ftp	1.2	1.3	2.8	0%	239	249.6	258.5	100%	1.2	5.4	89.1	30.4%
M-JPEG	5.7	7.9	23.7	0.2%	6.5	8.5	12.3	0.3%	5.7	8.9	43.3	14.6%

6. Summary, Conclusions, and Future Work

Rate-based resource allocation schemes are a good fit for providing real-time services in a general purpose operating system. 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. We have considered three such rate-based schemes: 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.

Table 9: *RBE+EEVDF* performance — throughput and loss rates.

	Constant Rate Senders			Misbehaved *ftp*			Bursty (Pareto) Senders		
	Drops at IP	Drops at socket	Packets Processed	Drops at IP	Drops at socket	Packets Processed	Drops at IP	Drops at socket	Packets Processed
Phone	0	0	2,873	0	0	2,789	0	0	2,954
ftp	0	0	11,802	17,872	0	11,647	0	0	10,437
M-JPEG	0	0	5,324	0	0	5,393	0	0	4,956
Dhrystone	N/A	N/A	3.1E+6	N/A	N/A	2.9E+6	N/A	N/A	4.5E+6

Table 10: Response time (ms) and deadline miss ratios.

	Constant Rate Senders				Misbehaved *ftp*				Bursty Senders			
	Min	Avg.	Max	%Miss	Min	Avg.	Max	%Miss	Min	Avg.	Max	%Miss
Phone	1.2	7.3	15.8	0%	2.2	10.1	26.2	0.1%	1.2	13.6	78.3	12.3%
ftp	1.2	1.3	2.9	0%	189	237.6	289.5	100%	1.2	5.3	59.1	24.1%
M-JPEG	5.7	7.8	25.7	0.32%	6.3	8.2	12.8	0.3%	5.7	9.6	43.4	13.1%

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.

7. References

[1] 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.

[2] M. Borriss, H. Härtig, *Design and Implementation of a Real-Time ATM-Based Protocol Server*, Proc. of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998, pp. 242-251.

[3] 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.

[4] P. Druschel, G. Banga: Lazy Receiver Processing, *A Network Subsystem Architecture for Server Systems*, Proc., USENIX Symp. on Operating System Design and Implementation, Seattle, WA, Oct. 1996, pp. 261-275.

[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] H. Härtig, R. Baumgartl, M. Borriss, C. J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, J. Wolter, DROPS-OS Support for Distributed Multimedia Applications; In the proceedings of the 8th ACM SIGOPS European Workshop, Sintra, Portugal, Sept. 1998.

[7] 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.

[8] 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.

[9] 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.

[10] K. Jeffay, S. Goddard, *A Theory of Rate-Based Execution*, Proc. 20th IEEE Real-Time Systems Symposium, Dec. 1999, pp. 304-314.

[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] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, E. HydenM, *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications*, IEEE J. on Selected Areas in Communications, Sept. 1996, pp. 1280-1297.

[13] J. Liedtke, *On micro-kernel construction*, Proc. of the 15th ACM SOSP, Orcas Island, WA, Dec. 1995, pp. 237-250.

[14] 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.

[15] M. K. McKusick, K. Bostic, M.J. Karels J. S. Quarterman, *The Design and Implementation of the 4.4BSD UNIX Operating System*, Addison-Wesley, 1996.

[16] C.W. Mercer, S. Savage, H. Tokuda, *Processor Capacity Reserves: Operating System Supportfor Multimedia Applications*, IEEE Intl. Conf. on Multimedia Computing and Systems, Boston, MA, May 1994, pp. 90-99.

[17] J. Mogul, K. Ramakrishnan, *Eliminating Receive Livelock in an Interrupt-Driven Kernel*, ACM Transactions on Computer Systems, Vol. 15, No. 3, August 1997, pp. 217-252.

[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] R. Rajkumar, K. Juvva, A. Molano and S. Oikawa, *Resource Kernels: A Resource-Centric Approach to Real-Time Systems*, Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking, January 1998, pp. 150-164.

[22] M. Spuri, G. Buttazzo, *Efficient Aperiodic Service Under the Earliest Deadline Scheduling*, Proc. 15th IEEE Real-Time Systems Symp., Dec. 1994, pp. 2-11.

[23] 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.

[24] 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.

[25] 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.

[26] H. Tokuda, T. Nakajima, P., Rao, *Real-Time Mach: Towards a Predictable Real-Time System*, Proc. Of the USENIX Mach Workshop, Burlington, VT, October 1990, pp. 73-82.

[27] 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.

[28] C.A. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, Ph.D. Thesis, MIT, Laboratory for CS, September 1995.

[29] Y. C. Wang, K. J. Lin, *Enhancing the Real-Time Capability of the Linux Kernel*, Proc. of the 5th International Conference of Real-Time Computing Systems and Applications, Hiroshima, Japan, November 1998.

[30] 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.

[31] G.R. Wright, W.R. Stevens, *TCP/IP Illustrated, Volume 2, The Implementation*, Addison-Wesley, Reading MA, 1995.