Proportional Share Scheduling of Operating System Services for Real-Time Applications*

Kevin Jeffay, F. Donelson Smith, Arun Moorthy, James Anderson
University of North Carolina at Chapel Hill
Department of Computer Science
Chapel Hill, NC 27599-3175
{*jeffay,smithfd,moorthy,anderson*}@*cs.unc.edu*

Abstract

While there is currently great interest in the problem of providing real-time services in general purpose operating systems, the issue of real-time scheduling of internal operating system activities has received relatively little attention. Without such real-time scheduling, the system is susceptible to conditions such as *receive livelock* — a situation in which an operating system spends all its time processing arriving network packets, and application processes, even if scheduled with a real-time scheduler, are starved. We investigate the problem of scheduling operating system activities such as network protocol processing in a proportional share manner. We describe a proportional share implementation of the FreeBSD operating system and demonstrate that it solves the receive livelock problem. Packets are processed within the operating system only at the cumulative rate at which the destination applications are prepared to receive them. If packets arrive at a faster rate then they are discarded after consuming minimal system resources. In this manner the performance of "well behaved" applications is unaffected by "misbehaving" applications. We demonstrate this effect by running a set of multimedia applications under a variety of network conditions on a set of increasingly sophisticated proportional share implementations of FreeBSD and comparing their performance. This work contributes to our knowledge of the engineering of proportional share real-time systems.

1. Introduction

Applications such as interactive multimedia and immersive virtual environments, require real-time computation and communication services from the operating system in order to be effective. As applications such as these are increasingly being hosted on general purpose (rather than specialized real-time) operating systems, there is great interest in migrating real-time systems technology to desktop operating systems. A recent development in the area of real-time support in general purpose operating systems is the use of proportional share resource allocation for providing real-time services [3, 7, 9, 13]. In a proportional share system, processes make progress at a precise, uniform rate according to the share of system resources they are to receive. Processes appear to execute on a dedicated virtual processor whose capacity is a fraction of that of the actual processor. Proportional share resource allocation is particularly well-suited to the problem of providing real-time services within a general purpose operating system because its underlying scheduling mechanism is a quantum-based round-robin-like scheduler and because one can implement a proportional share system without introducing any new application-level concepts or mechanisms. This means that existing applications can be made to execute in a predictable, real-time manner without modifying the application.

Previous operating system work in proportional share resource allocation has considered only the problem of scheduling user processes. In particular, the problem of scheduling operating system activity such as network protocol processing has not been addressed. This is significant because much of the processing in the operating system occurs asynchronously with respect to system calls made by applications. If the execution of operating system activities is not managed carefully, the operating system may consume an inordinate amount of resources and nullify the benefits of real-time scheduling for application processes.

For example, consider the *receive livelock* problem described by Druschel and Banga, and by Mogul and Ramakrishnan [6, 23]. In most general purpose operating systems, the processing of inbound network packets is the highest priority activity after the processing of clock interrupts. This is the case because the network interface is arguably the most real-time device on a general purpose computer. Since one cannot typically control the rate at which packets arrive at a computer, when packets are not processed "fast enough," it is possible that packets may be lost at the network interface. Said another way, unlike most other devices attached to a computer, the operating system cannot force the network interface to stop generating service requests without running the risk of losing data. On modern high-speed networks, packets can arrive at

* Work supported by grants from the National Science Foundation (grants CCR-9510156, & CCR-9732916) and the IBM Corporation.

In: *Proceedings of the 19th IEEE Real-Time Systems Symposium*,
Madrid, Spain, December 1998, *to appear.*

high enough rates that the process of responding to network interrupts and performing the necessary protocol processing can saturate the system. The system will spend all of its time preparing packets to be received by applications and there will be no time for applications to actually receive and process any packets. Thus in the worst case, all inbound data is lost. Every packet is partially processed (by the operating system) while none are fully processed (by the destination application). The receive livelock problem has been observed on server machines such as web servers, file servers, and DNS name servers, that are attached to high-speed networks such as 100 Mbps FDDI rings [6].

The essence of the receive livelock problem is the static priority scheduling mechanisms employed in most operating systems (including most real-time operating systems). If the highest priority processing consumes all of the system's resources then all other processes starve. The solution therefore, is (1) to bound the resources consumed by the network interface, and (2) to ensure that if a packet is received at the network interface it will be processed by the destination application. An approach to the first problem investigated by Mogul and Ramakrishnan, is to poll for newly arrived network packets under times of high load. This technique worked well, however, by itself, polling could not ensure that packets received are eventually processed by the application. This is because ultimately, their system was not a real-time system and did not support integrated application and operating system processing. To address the second problem, Druschel and Banga developed a network subsystem architecture for processing packets according to application-level priorities. We adopt their architecture but implement it using proportional share technology. Moreover, we provide a real-time solution by integrating application and kernel-level scheduling. When combined with proportional share scheduling of user processes, we demonstrate that proportional share scheduling of packet and protocol processing provides a means for precisely controlling the resources consumed by the network subsystem. Network processing will occur at the rates at which applications are prepared to receive packets and hence all data received is eventually processed by the application. Moreover, our solution protects applications from "misbehaving" senders. If a remote sender transmits messages to an application on our machine at a higher rate than the receiving application can process, the "excess" messages are dropped at the network interface after only minimal processing. Thus applications whose senders are "well behaved" are unaffected by these errant processes.

While we are advocates of proportional share technology, we recognize that other solutions to the problems we outline herein are possible (*e.g.*, [23]). Whether or not one views a particular resource allocation approach as being intrinsically better or worse than any other will depend on factors such as the nature of real-time guarantees required by applications, the extent to which it is considered acceptable to modify the internal structures of the operating system or its API, and the extent to which it is considered acceptable to modify applications to take advantage of the new real-time services. Our goal is to understand the cost of providing real-time services transparently to applications that are unaware of their existence. We seek to understand the complexity of providing such services in conventional operating systems and how such services are likely to perform.

Our work makes the following contributions. First we demonstrate a model for proportional share scheduling of operating system services through minimal modifications to the existing operating system. We show how a monolithic, single-threaded operating system kernel such as the FreeBSD kernel can be extended to allow proportional share execution of network packet and protocol processing. Second, we demonstrate that proportional share execution of the network packet and protocol processing solves the receive livelock problem. We show that packets are received only at the rates at which applications are able to process them and how unmodified applications process the packets that are received in real time.

The following section discusses related work in the design of real-time operating systems. Section 3 briefly reviews the main concepts of proportional share resource allocation. Section 4 describes the network protocol processing components of FreeBSD and describes their proportional share implementation. Section 5 evaluates the implementation by demonstrating both the effects of the receive livelock problem and its elimination through proportional share scheduling of the network interface. We conclude in Section 6 with some comments for future investigations.

2. Related Work

Research into the design and construction of real-time operating systems can be crudely partitioned into three categories: the development of brand new real-time operating systems, the extension of existing operating systems to support real-time processing, and the provision of real-time execution facilities by virtualizing the underlying hardware and executing a largely unmodified general purpose operating system on the resulting virtual machine. The most recent example of a new real-time operating system is the Rialto operating system developed at Microsoft [15]. Recent examples of the real-time extensions to existing operating systems are more numerous and include Real-Time Mach [21], the Processor Capacity Reserves variant of Real-Time Mach [18], the SMART Solaris system [7], and variants of Linux [16]. Of particular note here are extensions to UNIX kernels to support proportional share scheduling. These include the SFQ SVR4 UNIX system [3], the Mach- and FreeBSD-based Lottery Scheduling implementations [13], and the EEVDF version of FreeBSD [9]. Each of these systems supports only proportional share execution of user processes. Examples of the providing real-time services through virtual machine emulation include Real-Time Linux [17], the Real-Time IBM Microkernel [22], and a real-time HAL extension for Windows NT [12].

Our approach falls under the category of extending an existing operating system to support proportional share processing. From our perspective the most relevant related work is the work done within the context of the Processor Capacity Reserves (PCR) extensions to Real-Time Mach (RT Mach) [18, 20]. In this work, the RT Mach developers were also concerned with the impact of network protocol processing and explicitly scheduled this process as a real-time process using the PCR abstraction. Protocol processing was performed in a user-level process and hence scheduling its execution was straightforward as real-time scheduling of user processes is the cornerstone of RT Mach. The PCR abstraction ensures that real-time activities do not execute for longer than they are expected to. Thus, although the RT Mach developers were primarily interested in showing the utility of integrated protocol and application scheduling, it is likely that a PCR system could be made immune to the receive livelock problem.

Our work differs in two respects. First, we are dealing a differently structured host operating system. Unlike the original microkernel origins of RT Mach which enable user-level execution of system processes, we are dealing with a monolithic, single-threaded operating system. Our challenge here is to schedule kernel "processes" without rewriting the kernel so as to create physical, schedulable processes. Second, whereas RT Mach employs rate-monotonic scheduling technology, we are experimenting with proportional share resource allocation. Ultimately we hope to show that real-time execution of operating systems services is possible in a proportional share system without having to resort to explicitly restructure the kernel to make it multi-threaded.

3. Proportional Share Resource Allocation

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 process is selected to use the resource. Once the process 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, we associate a *weight* with each process that determines the relative *share* of the resource that the process should receive. Let w_i denote the weight of process i, and let $A(t)$ be the set of all processes active at time t. Define the (instantaneous) share $f_i(t)$ of process i at time t as

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

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

$$S_i(t_0, t_1) = \int_{t_1}^{t_2} f_i(t)\, dt \qquad (2)$$

time units.

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. 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^i be the time at which process i becomes active, and let $s(t_0^i, t)$ be the service time process i receives in the interval $[t_0^i, t]$. Then if process i is active in the interval $[t_0^i, t]$, its lag at time t is defined as

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

Since the lag quantifies the allocation accuracy, we use it as our primary metric for evaluating the performance of *PS* scheduling algorithms. Previously we have shown that one can schedule a set of processes in a *PS* system such that the lag is bounded by a constant over all time intervals [9]. This means that a *PS* system's deviation from a system with perfectly uniform allocation is bounded and thus, as explained below, real-time execution is possible.

3.1 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 processes and at the end of each quantum, considering only processes whose lag is positive [9]. If a process'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 process's lag is positive it is considered eligible to execute. If its lag is negative, then the process has received more processor time than it should have up to the current time and it is considered ineligible to execute

When multiple processes are eligible, they are scheduled using an *earliest deadline first* rule, where a process'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 process should complete execution if it receives exactly its share of the CPU. For example, if a process'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 process actually receives 10% of the CPU, over the next 20 *ms* it will execute for 2 *ms*.

In [9] it was shown that this proportional share version of deadline scheduling provides optimal (*i.e.*, minimum possible) lag bounds. This algorithm forms the basis for the *PS* implementation described in Section 4.

3.2 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 *fair* allocation). A *PS* scheduler achieves uniform allocation if it can guarantee that processes' lags are always bounded.

Real-time computing is achieved in a *PS* system by (*i*) ensuring that a process's share of the CPU (and other required resources) remains constant over time, and by (*ii*) scheduling processes such that each process's lag is always bounded by a constant. If these two conditions hold over an interval of length t for a process i, then process 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 process i, and ε is the allocation error, $0 \leq \varepsilon \leq \delta$, for some constant δ [9]. 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 the implementation described below ε is a set-able parameter, but is fixed at 1 *ms*.

Our deadline-based scheduling algorithm ensures that each process's lag is bounded by a constant [9] (condition (*i*)). To ensure a process's share remains constant over time (condition (*ii*)), whenever the total weight in the system changes, a "real-time" process'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 processes are created), then a real-time process'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).)

4. Realizing Proportional Share Execution of Operating System Activities: A Case Study

The challenges in realizing proportional share execution of operating system activities are numerous. They include:

- Identifying "threads" of control within the operating system kernel that need to be scheduled and subjecting them to the purview of a *PS* scheduler.
- Assigning weights and shares to kernel activities.
- Ensuring mutually exclusive access to shared data structures in the kernel.
- Assigning buffer capacity in a proportional manner at asynchronous kernel boundaries.

We illustrate these problems using the network packet and protocol processing portions of the FreeBSD operating system as an example. For brevity, we consider only processing associated with the receipt of inbound packets. (Processing of outbound packets turns out to be an easier problem.)

4.1 Scheduling of Operating System Activities in FreeBSD

FreeBSD is a derivative of the 4.4 BSD Operating System [5]. Network processing occurs in three distinct layers in FreeBSD: the *socket layer*, the *protocol layer*, and the *device interface* layer. Figure 1 illustrates these layers for UDP packets. The layers for other transport protocols are similar. Processing within each layer is controlled by events external to the kernel such as hardware interrupts from the network interface or software interrupts from user processes making system calls to receive network messages. Interrupts from the network interface device are serviced by a device-specific interrupt handler that is executed at a high priority level (called *splimp*) that preempts all other network-related processing and is preemptable only by interrupts from the hardware clock. 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 plus auxiliary data such as queue pointers. 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 uses the type field from the Ethernet header to determine which protocol (*e.g.*, IP) should receive the packet and enqueues the packet on that protocol's input queue. It then posts a software interrupt (with an intermediate priority, *splnet*) that will cause the protocol layer to be executed when no higher priority hardware or software activities are pending. It then returns from interrupt processing at the *splimp* level.

Processing by the protocol layer occurs asynchronously with respect to the device driver processing. When the software interrupt posted by the device driver at priority *splnet* is the highest priority, the protocol-layer input routine is entered. It executes a main loop in which each iteration removes the *mbuf* chain at the head of the input queue and passes it to the appropriate processing routines for IP and UDP. To protect the input queue data structure shared by the protocol layer and the interface layer, the protocol layer dequeue function temporarily raises its priority to the *splimp* level to prevent preemption by the device driver. The *mbuf* chain is then processed completely in the protocol layer and finally 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. Software interrupt porcessing returns when no more *mbufs* remain on the protocol input queue.

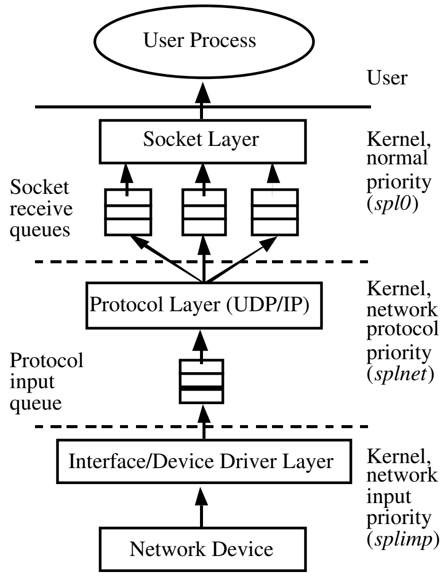

Figure 1: FreeBSD, UDP, network input processing.

The kernel socket layer code executes when a process invokes some form of receive system call on a socket descriptor and runs at the lowest-priority software interrupt level (*spl0*). This priority is used for all normal kernel processing so the socket code can execute when no higher priority interrupts are pending. When there is a receive system call active for the socket, data to be received is copied into the receiving process's local buffers from the *mbuf* chain(s) at the head of that socket's receive queue. This queue is protected by a locking mechanism and by temporarily raising the socket layer priority to *splnet* to prevent preemption by the protocol layer. 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.

For a more complete description of these functions see [5] and [19].

4.2 *PS* Scheduling of Operating System Activities in FreeBSD

Conceptually each layer of protocol processing represents a separate logical process that must be scheduled. These layers are not processes in the traditional sense but instead are more akin to procedure calls that are called by a software interrupt dispatching mechanism that is invoked upon the completion of every system call or quantum expiration. The existing dispatching mechanism is, in essence, a simple static priority scheduler. We have modified the FreeBSD software interrupt dispatching mechanism to dispatch kernel activities only at quantum expirations. This has the positive effect of ensuring that user processes execute for a full quantum once scheduled, but also the negative effect of delaying kernel activities. This effect can be mitigated, however, by an appropriate choice of a quantum duration. In the experiments described below we used a 1 *ms* quantum (compared to the default FreeBSD quantum of 100 *ms*) without ill-effect.

In addition to dispatching software interrupts at quantum boundaries, we also assign a weight to each interrupt and perform an eligibility and deadline test for each interrupt [9]. Software interrupts are considered equally with user processes and either the eligible user process or eligible software interrupt with the earliest deadline is scheduled next. One complexity here is that kernel activities communicate and synchronize through shared memory in the kernel. Previously, the software interrupt priority structure was used to ensure shared data in the kernel is accessed in a mutually exclusive manner. We can no longer use these mechanisms because in order for the lag bounds guaranteed by the theory to hold, a process must be able to execute for its entire quantum once scheduled. Thus when a software interrupt handler executes, we must ensure that no shared data structures within the kernel are locked when a quantum expires.

In general, there are several approaches to solving this problem. For the FreeBSD network protocol processing code, each software interrupt routine consists of a loop that removes a packet from one queue, performs some processing and then inserts the packet on a second queue. These loops typically execute until the source queue is empty. We modified these loops to execute until either the source queue is empty or until a maximum number of packets has been processed. In the latter case the interrupt routine will reschedule itself (post another software interrupt for itself) and then terminate. The maximum number of packets to be processed is chosen to be the maximum number of packets that are guaranteed to be processed to completion within one quantum. Bounding the number of packets processed in this way ensures that software interrupt processing is never preempted by a quantum expiry and hence that shared data structures are always in a consistent state at the end of each quantum. Note that although this technique appears at first to reduce performance (because kernel activities may not execute for an entire quantum), the actual rate at which activities are performed depends on the share of the processor they are assigned and not on how long they actually execute once scheduled. Moreover, by adopting this approach we need not include any synchronization code in the packet processing code and hence overall this code is more efficient.

The maximum number of packets that could be processed was determined by hand timing the loops to determine the maximum number of packets that could be processed within a quantum. The time to process a packet is, for the most part, independent of the size of a packet (much of the processing consists of pointer manipulations) and in all cases is bounded. These measurements were also used to determine the cost of a particular kernel activity and used by the dispatcher to compute deadlines for kernel activities as described in Section 3.1.

4.3 *PS* Scheduling of User Processes

User processes are scheduled using the FreeBSD scheduler modified to perform the eligibility and deadline calculations. Process descriptors were changed to record a weight and share for each process as well as a measure of its current execution time cost. Processes can either assign their own weights or have them assigned by a separate manager process. If programs are written with knowledge of our kernel modifications they can alter their weights through a system call. Pre-existing programs receive a default weight that ensures they make progress as in a time-sharing system. This weight (and the weight of any user process) can be changed by the manager process. By manipulating a process's weight, its rate of progress relative to the other processes in the system can be controled. By manipulating a process's share, one can control its absolute rate of progress independent of the other processes in the system. If the manager sets a process's share, the kernel determines what weight the process should have in order to guarantee that it receives the appropriate share [10].

The kernel records a bit indicating whether a process has a fixed weight or a fixed share. Processes with a fixed share are "real-time" processes and their weights will be adjusted to maintain a constant share whenever the total weight of processes in the system changes (*i.e.*, whenever a process is created or destroyed). Beyond the fixed weight/share dichotomy, there is no distinction between real-time and non-real-time processes in the kernel. Any process can become a real-time process at any time so long as the total shares of all real-time processes remains less than 1.0.

To schedule a process the kernel needs an estimate of the process's execution time. The estimated execution time is used to determine the process's deadline as outlined in Section 3.1. We use a simple heuristic of monitoring the execution time consumed by processes between system calls and assume that the execution time used in the recent past is a good indicator of the time a process will require in the near future. We use the standard FreeBSD execution time monitoring infrastructure to record the elapsed execution time of processes.[1] When a process performs network I/O a new deadline is computed for the process based on the amount of execution time consumed since the last I/O operation. For processes with a regular structure (such as most cyclic real-time processes) this heuristic should work quite well. For the multimedia processing load considered in our experiments (see Section 5), this simple heuristic was sufficient. Moreover, if execution time estimates are inaccurate the kernel will be able to detect this fact. If the execution time estimate for a process is too low then the process will not have made another system call by its deadline. If the estimate is too high, the process will complete its current execution before its deadline. Thus one can employ software phased-locked loops to further refine the estimates of execution time[4].

Note, however, that if the estimate of a process's execution time is inaccurate, it effects only the performance of that process. Independent of the execution time estimate, a process can never consume more than its share of the processor (and shares are determined by weights not costs). Thus if an estimate of a process's execution time is overly optimistic or if a programmer willfully specifies an execution time that is too low, the performance of other processes is unaffected.

4.4 Assigning Weights to Kernel Activities

Kernel processing is scheduled together with user processing; each according to its weight. Weights for user processes are either set by the processes directly or by the manager process. Weights for kernel activities are derived from user weights. For the purpose of defining weights, we distinguish between two types of kernel processing: per user process activity and demultiplexing activity. Per user process activities consist of the kernel processing associated with system calls made by user processes. When a system call is made it is considered a logical extension of the invoking process and execution of the appropriate kernel activity is performed (scheduled) with the same weight as the invoking process. For example, when a process attempts to receive a message from a socket, the socket layer processing is performed with the same weight as the receiving process. In this manner kernel processing directly related to an individual process occurs at the same rate at which the process executes.

[1] Note that because software interrupts are now scheduled, timings of user activities are more accurate than in unmodified FreeBSD (as was the case in [23]). Previously, software interrupt processing was charged to the user process executing at the time of the interrupt.

Other kernel activities such as IP processing are performed on behalf of a collection of processes. For example, when the IP processing software interrupt is posted after a packet arrives, the ultimate destination of the packet is not known and hence the rate at which it should be processed cannot be determined (without actually processing the packet!). In this case, IP processing needs to make progress at the sum of the rates of all processes there are currently receiving packets from the network. To ensure this is the case, whenever a user process binds to a socket, the kernel records its weight and adds a corresponding amount of weight to the weight of the IP processing kernel activity.

One subtlety here is that whereas user processes may measure rates in arbitrary units (*e.g.*, execution time received per second), IP processing makes progress in units of packets processed per time unit. Thus a user process's weight must be mapped into a IP weight by estimating how many packets a user process is likely to receive per unit time. To do this, we use the deadline of a process as an estimate of its period and assume that the process will receive one packet per period. For example, if a process's weight is such that its share is 10% of the CPU, and the process's measured execution time is 2 *ms*, its period (the product of dividing its execution time by its share) will be 20 *ms*. Thus the weight of the IP activity needs to be set so as to ensure that IP processing is performed at least once every 20 *ms*. Therefore, whenever a process with deadline d binds to a socket, the weight of the IP activity is increased so that the share of the activity increases by c/d, where c is the cost of IP processing for a single packet (a constant). In addition, the weight of the IP processing activity is further inflated to increase its share by a configurable amount that is sufficient to ensure that non-requested IP packets (*e.g.*, ARPs and other broadcast packets) can be processed without effecting the performance of user packet processing.

4.5 Proportional Share Allocation of Kernel Buffers

A final issue to consider is the allocation of buffer space within the kernel. Just as processes require a share of the CPU in order to make progress, they also require a share of the buffers available in each of the interface and protocol processing layers within the kernel. In FreeBSD, at most 50 packets can be queued pending processing by the IP layer. If these queue entries are allocated to arriving packets in a FCFS manner, it is possible that applications expecting to receive packets at a slow rate may be adversely effected by applications that either are not processing packets fast enough or whose senders "misbehave" by sending packets at a higher rate than the application is prepared to receive. For example, consider a scenario wherein an audio phone application expects to receive one packet every 20 *ms*, and a file transfer program expects to receive 200 packets/second (one packet every 5 *ms* on average). In this case the IP processing activity will be assigned a weight so as to ensure it is able to process at least 50 + 200 = 250 packets/second. If the file transfer sender does not pace its transmission or if it sends at a higher than expected average rate, the device interface queue may become full with unprocessed file transfer packets and when an audio phone packet arrives, it is dropped for lack of space. In this case a "misbehaving" non-real-time application is negatively impacting a "well behaved" real-time application.

The solution, a variant of that proposed by Druschel and Banga [23], is to allocate queue capacity (a number of queue entries) for packets destined for user processes in proportion to the rate at which the process is expected to receive packets. For example, if a user process currently is expecting to receive 1 packet every 20 *ms*, and the period of the IP processing activity is currently 10 *ms*, then at least one queue entry should be reserved for this process. (In practice, one would reserve more entries to deal with a less than periodic arrival process.) If a user process is executing fast enough to receive 1 packet every 5 *ms*, then at least 2 queue entries should be reserved for this process.

In addition to reserving queue entries for user processes, the IP processing activity also has to internally schedule the processing of individual packets. That is, it cannot simply service packets in FCFS order as this would hurt well-behaved applications when other applications are having packets delivered at inappropriate rates. Although queue entries are reserved for user processes, it is likely that at any given time there will exist more packets than the IP activity can process in one quantum, and hence the IP activity must explicitly determine which packets to service first in order to ensure that well-behaved applications do not lose packets. To do this, we simply recursively implement another instance of a proportional share scheduler inside the IP activity to select the packet to process next. Whenever the IP activity is scheduled, it internally sub-allocates its quantum to packet processing by assigning eligible times and deadline to packets based on the weights of the user process that will receive the packet. Combined, the hierarchical scheduling mechanism and queue entry reservation system ensure that when a packet for a well-behaved application arrives at the network interface, it is guaranteed to be processed at the IP layer and delivered to the user process, independent of how other applications are receiving packets. Said another way, packets for misbehaving applications are dropped as early as possible after only minimal processing.

Note that all we are doing here is managing buffers in precisely the same way routers manage buffers under fair queueing-based service disciplines [1, 2, 8, 14].

5. Experimental Results

We modified the FreeBSD 2.2.2-RELEASE system to support proportional share scheduling and ran a suite of experiments to assess the impact of proportional share execution of packet and network protocol processing. Our experiments were conducted on a 200Mhz Pentium Pro with 64 MB of memory. The network interface was a 3Com 3C595 (*vx0*) 10/100 Ethernet adapter running at 10Mbps. We used three simple applications that we believe are indicative of the types of real-time and non-realtime processing that is likely to be performed on a general purpose workstation. The applications were:

- an audio player application that handles incoming 100 byte 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 receiver that handles incoming 1470 byte messages at a rate of 90/second and computes for 5 milliseconds on each message (requiring 45% of the CPU on average), and
- file transfer program that handles incoming 1470 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 compute intensive program.

Each of these programs was run as a separate process on the modified FreeBSD system and assigned a processor share according to its CPU utilization and execution rate. (The Dhrystone was not explicitly assigned a weight. Instead FreeBSD assigned it a weight that resulted in it receiving whatever share of the CPU remained allocated.) We wrote three programs to act as sending processes and send messages with the desired size and rate to the corresponding receiver. We ran one of these programs on each of three additional machines (all 200 Mhz or greater Pentiums) running FreeBSD v2.2.2, all connected to an unloaded 10Mbps Ethernet along with the machine running the modified FreeBSD kernel.

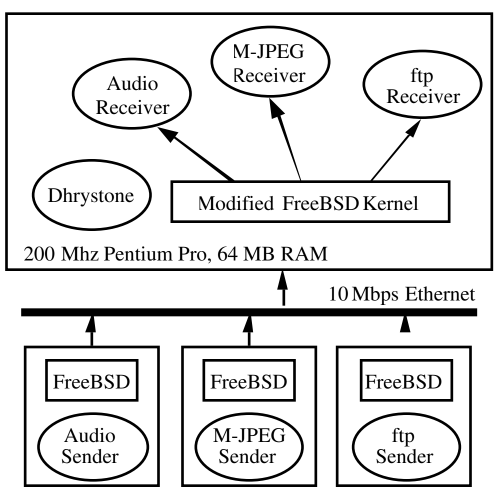

Figure 2: Experimental configuration

The experimental setup is illustrated in Figure 2.

With this experimental setup we conducted a number of experiments where we investigated the effects of different possibilities for the scheduling within the modified FreeBSD kernel. 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 exponentially distributed 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. Instrumentation was added to the modified kernel and the user processes to collect performance data. The primary data of interest are (*a*) the number of messages received by each process during a fixed length interval (1 minute in our case), (*b*) the number of packets dropped at the queue between the interface/device driver layer and the IP/UDP protocol layer, and (*c*) the number of packets dropped at the socket receive queue (see Figure 1).) For the Dhrystone benchmark we recorded only the number of iterations completed in our measurement interval. Over our measurements intervals we would nominally expect that the audio player would receive 3,000 packets (50×60), the video player would receive 5,400 packets, and the file transfer would receive 12,000 packets. In addition, we would never like to observe any loss at the socket layer. As we explain below, loss here would be an indication that too much processing time is being spent processing packets in the kernel and that because of this user processes are not able to run.

To establish an unmodified FreeBSD baseline, we first ran our applications on a FreeBSD with a 1 *ms* clock tick and a 1 *ms* scheduling quantum. These results are given in Table 1. The audio and file transfer applications executed at their sender's rate because they require little compute time and are mostly I/O bound, blocked on a socket receive. The video application has a high CPU usage (45%) and is subjected to the FreeBSD aging mechanism which reduces its priority. Because of this, it is unable to receive all of its packets and some are dropped at the socket receive queue. The effect under bursty senders is similar.

When the file transfer sender misbehaves, we see the effects of fixed priority scheduling on in-

Table 1: Unmodified FreeBSD, 1 *ms* clock, 1 *ms* quantum. [2]

	Constant Rate Senders			Bursty Senders			Misbehaved File Sender		
	Packets (Iterations)	Drops at socket	Drops at IP	Packets (Iterations)	Drops at socket	Drops at IP	Packets (Iterations)	Drops at socket	Drops at IP
Audio Application	3,000 (0.5)	0	0	2,938 (15.6)	0	0	2,999 (0.5)	0	0
M-JPEG Application	3,313 (19.1)	2,110 (19.4)	0	3,466 (25.2)	1,703 (10.4)	0	2,456 (15.6)	2,967 (16.4)	0
File Transfer	11,996 (0.5)	0	0	10,897 (58.1)	0	0	11,862 (40.8)	48,043 (39.2)	0
Dhrystone	7,333,439 (49,227)	N/A	N/A	7,660,042 (37,347)	N/A	N/A	5,479,480 (48,454)	N/A	N/A

Table 2: Modified FreeBSD, proportional share for user processes only.

	Constant Rate Senders			Bursty Senders			Misbehaved File Sender		
	Packets (Iterations)	Drops at socket	Drops at IP	Packets (Iterations)	Drops at socket	Drops at IP	Packets (Iterations)	Drops at socket	Drops at IP
Audio Application	2,999 (0.9)	0	0	2,927 (18.1)	0	0	2,999 (0.9)	0	0
M-JPEG Application	5,454 (0.0)	0	0	5,126 (92.6)	0	0	5,454 (0.0)	0	0
File Transfer	11,996 (0.5)	0	0	10,483 (12.0)	0	0	12,000 (0.0)	47,952 (3.8)	0
Dhrystone	4,593,536 (46,257)	N/A	N/A	6,115,263 (235,175)	N/A	N/A	915,343 (26,109)	N/A	N/A

terrupt handling and protocol processing. CPU cycles are used for packets that are eventually dropped at the socket layer taking cycles away from the video and Dhrystone processes (since these processes consume the most CPU time they are aged quickly and soon execute at the lowest priority). Even though more file transfer packets are handled, many more are dropped as are video packets. However, because all packets received are processed up to the socket layer (where there are separate queues for each port), the audio application is still able to receive all its packets.

With this baseline established, we modified the FreeBSD kernel for proportional share scheduling of the user processes. In this case, the interface/device driver layer processing and the network protocol layer processing was executed according to the normal kernel software interrupt level mechanism and priorities. (For all proportional share experiments, we used a clock tick of 1 *ms* and a quantum of 1 *ms*.) The results of this experiment are given in Table 2.

[2] Each entry in each table reports an average and standard deviation (in parenthesis) over a set of runs.

With constant rate senders these results show the benefits of proportional share allocation. Compared with the results in Table 1, the video player now receives all its packets at the expense of the Dhrystone process. Moreover, no packets are dropped at any queue. With bursty senders no packets are dropped but some reduction in the rate of packet reception occurs due to the bursty nature of the senders and our relatively short observation interval. For the case with the misbehaving file transfer sender we are able to maintain the desired rate of progress for all real-time processes. In addition we see the effects of interrupt and protocol processing at a fixed priority in the kernel in the form of a further slowdown of Dhrystone (compared to the constant rate case) and in the loss of file transfer packets at the socket layer. This shows how CPU cycles are still being allocated with fixed priority to processing packets that will never be handled by the application process.

The next design choice we considered was to also explicitly schedule the packet and network protocol processing along with the user processes in a proportional share manner. Given the cost of processing a single packet and the rates at which user processes were estimated to receive, FreeBSD computes a scheduling period for the protocol layer of 10 *ms*. using the procedure outlined in Section 4.4. With this period and its computed share, the protocol processing layer will process 4 packets every 10 *ms*. (note that on average 3.4 packets are expected to arrive in a 10 *ms* interval). The protocol-layer input queue had the same limit on the maximum number of packets that could be enqueued as in normal FreeBSD (50 packets). The results for this experiment are given in Table 3.

For constant rate and bursty senders there is essentially no difference between the proportional share scheduling of user processes only and the combined proportional share scheduling of kernel and user activities. There is, however, a dramatic effect on the results when the file sender misbehaves. As expected, the protocol layer processes at most 24,000 packets (4 packets/10 *ms* for 60 seconds) but because the aggregate number of packets received is over 68,000, the IP protocol layer input queue (with its maximum of 50 entries) is constantly overflowed. More importantly, since the audio sender is sending at the lowest rate (50 packets/second), it is more likely have its packets dropped at the protocol layer input queue. This illustrates why it is important to allocate buffer resources as well as CPU resources to achieve the desired scheduling goals. Note that in this case the performance of the Dhrystone is much improved. Since the real-time processes execute at reduced rates (for lack of data), there are more cycles to be consumed by the Dhrystone.

Following on the architecture of Druschel and Barga [23], we next established an input queue for each socket (destination process) at the asynchronous boundary between the interface/device driver layer and the protocol layer. The queue for each destination process had a limit on the maximum number of packets that could be enqueued based on the scheduling period for the protocol processing and the expected receiving rate for a destination (plus 1 or 2 additional packets to buffer short bursts). The input queue limits were: audio player = 2, video player = 2, and file transfer = 3 packets. The protocol layer processed each of the three queues to exhaustion each time it was run (*i.e.*, every 10 *ms*). These results are given in Table 4.

Again for constant rate and bursty senders there are few differences between this case and the previous one except when the file sender rate increases. With per-destination input queues allocated according to the expected rate of receiving packets, we in effect reserve buffers for the audio and video receivers and thus enable them to achieve the desired rate of packet processing. The particular allocations we used were sufficient for some of the file packets to be processed by the protocol layer only to be discarded at the socket receive queue (because in order to absorb short-lived bursts, strictly speaking the number of buffers reserved was larger than necessary), however, the majority of packets were discarded at the network interface.

The final design variation we considered was to add a form of proportional share scheduling to the IP/UDP layer processing. In this case, the input queue for each destination was serviced only if the eligible time for receiving the packet at the head of the queue had passed. These results are given in Table 5.

These results show the effect of allocating both CPU and buffer space with the desired results achieved for the all cases of senders. In each case, the processing rates for all applications were as required and all packet drops were pushed down to the point were the minimum resources were expended before the drop occurred.

6. Summary & Contributions

As commodity computers become powerful enough to execute next generation networked multimedia applications, there will be a strong demand for real-time computing and communication support in desktop operating systems. We are advocating the use of proportional share resource allocation technology as the foundation for these services. In this paper investigated the problem of proportional share execution of operating system services. We argued, and demonstrated empirically, that without real-time management of the network interface and protocol processing, the positive effects of real-time scheduling of user processes can easily be nullified. We also demonstrated that it is possible to modify a single threaded monolithic FreeBSD UNIX kernel such that packet and network protocol processing is performed in a proportional share manner. In particular, the parameters needed to schedule kernel activities, namely the weights and costs of each activity, can be either derived from user processes' scheduling parameters or estimated by simple measurement of the code. Moreover, the proportional share framework makes it easy to develop hierarchical resource allocators such as a fair queuing-based buffer manager we employed at the network device interface to further improve throughput for real-time applications.

The result of our research is a proportional share version of FreeBSD that supports integrated application and kernel scheduling and solves the receive livelock problem. Packets are processed only if the destination process is capable of receiving them and all packets received are processed by the application.

Our work contributes to the state of the art in the engineering of proportional share real-time operating systems. While the present work is largely a proof of concept and a preliminary examination of the design space for realizing proportional share services, in the future we hope to per-

Table 3: Modified FreeBSD, *PS* for user processes and protocol processing (one protocol-layer input queue).

	Constant Rate Senders			Bursty Senders			Misbehaved File Sender		
	Packets (Iterations)	Drops at socket	Drops at IP	Packets (Iterations)	Drops at socket	Drops at IP	Packets (Iterations)	Drops at socket	Drops at IP
Audio Application	3,001 (1.4)	0	0	2,992 (5.0)	0	0	757 (341.8)	0	2,244 (341.6)
M-JPEG Application	5,457 (4.2)	0	0	5,225 (8.2)	0	0	2,004 193.0)	0	3,448 (194.9)
File Transfer	12,005 (8.0)	0	0	10,532 (171.3)	0	0	11,999 (0.5)	15,211.0 (297.9)	32,745 (299.5)
Dhrystone	4,970,544 (32,032)	N/A	N/A	6,034,405 (71,884)	N/A	N/A	8,794,017 (340,194)	N/A	N/A

Table 4: *PS* scheduling of user processes and protocol processing with destination queues (no packet scheduling).

	Constant Rate Senders			Bursty Senders			Misbehaved File Sender		
	Packets (Iterations)	Drops at socket	Drops at IP	Packets (Iterations)	Drops at socket	Drops at IP	Packets (Iterations)	Drops at socket	Drops at IP
Audio Application	2,999 (0.9)	0	0	2,958 (17.2)	0	0	2,999 (0.5)	0	0
M-JPEG Application	5,454 (0.5)	0	0	5,215 (11.9)	0	1.3 (0.5)	5,454 (0.5)	0	0
File Transfer	12,005 (3.3)	0	0	10,887 (17.8)	0	7.7 (0.9)	12,003 (5.2)	31,044 (10.1)	16,906 (0.5)
Dhrystone	4,805,747 (63,697)	N/A	N/A	5,817,508 (7,932)	N/A	N/A	1,079,076 (39,830)	N/A	N/A

Table 5: *PS* scheduling of user processes and protocol processing with destination queues and packet scheduling.

	Constant Rate Senders			Bursty Senders			Misbehaved File Sender		
	Packets (Iterations)	Drops at socket	Drops at IP	Packets (Iterations)	Drops at socket	Drops at IP	Packets (Iterations)	Drops at socket	Drops at IP
Audio Application	3,000 (0.0)	0	0	2,966 (8.2)	0	4.0 (5.7)	3,000 (0.0)	0	0
M-JPEG Application	5,454 (0.0)	0	0	5,221 (22.8)	0	6.3 (4.1)	5,454 (0.5)	0	0
File Transfer	11,999 (0.8)	0	0	10,893 (45.7)	0	0.7 (0.5)	12,000 (0.5)	0	47,983 (1.2)
Dhrystone	4,652,845 (15,045)	N/A	N/A	5,788,270 (66,791)	N/A	N/A	1,261,473 (37,910)	N/A	N/A

form a more exhaustive examination of these design issues. In particular, we are working on proportional share allocation of non-preemptible resources in the kernel such as disk bandwidth.

7. References

[1] A. Demers, S. Keshav, S. Shenkar, *Analysis and Simulation of a Fair Queueing Algorithm*, Jour. of Internetworking Research & Experience, Oct. 1990, pp. 3-12.

[2] S. J. Golestani, *A Self-Clocked Fair Queueing Scheme for Broadband Applications*, Proc., IEEE INFOCOM '94, April 1994, pp. 636-646.

[3] P. Goyal, X. Guo H. M. Vin, *A Hierarchical CPU Scheduler for Multimedia Operating Systems*, Proc., USENIX Symp. on Operating Systems Design and Implementation, Seattle, WA, Oct. 1996, pp. 107-121.

[4] H. Massalin, C. Pu, *Fine-Grain Adaptive Scheduling Using Feedback, Computing Systems*, Vol. 3, No. 1, 1990, pp. 139-173.

[5] 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.

[6] 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 .

[7] J. Nieh, M. S. Lam. *The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications*, Proc., Sixteenth ACM Symp. on Operating Systems Principles, Saint-Malo, France, Oct. 1997, pp. 184-197.

[8] A. K. Parekh, 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.

[9] 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.

[10] I. Stoica, H. Abdel-Wahab, K. Jeffay, *On the Duality Between Resource Reservation and Proportional Share Resource Allocation*, Proc. Multimedia Computing and Networking 1997, SPIE Proceedings Series, Vol. 3020, Feb. 1997, pp. 207-214.

[11] H. Tokuda, T. Kitayama, *Dynamic QOS Control Based on Real-Time Threads*, Proc., Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster, UK, Nov. 1993, Lecture Notes in Computer Science, Vol. 846, pp. 124-137.

[12] VenturCom Inc., Real-time Extension 4.1 for Windows NT, *http://www.venturcom.com/prod_serv/nt/rtx/index.html*, 1997.

[13] C. Waldspurger, W. Weihl. *Lottery Scheduling: Flexible Proportional Share Resource Management*, Proc. USENIX Symp. on Operating System Design and Implementation, Nov. 1994, pp. 1-12.

[14] L. Zhang, *VirtualClock: A New Traffic Control Algorithm for Packet-Switched Networks*, ACM Transactions on Computer Systems, vol. 9, no. 2, May 1991, pp. 101-124.

[15] 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.

[16] B. Srinivasan, S. Pather, F. Ansari, D. Niehaus. *A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software*, Proc., IEEE Real-Time Technology and Applications Symp., Denver, CO, June 1998, pp. 112-120.

[17] M. Barbarnov, V. Yodaiken, *Real-Time Linux*, Technical Report, Department of Computer Science, New Mexico Institute of Mining and Technology, undated.

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

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

[20] C. Lee, K. Yoshida, C. Mercer, R. Rajkumar, *Predictable Communication Protocol Processing in Real-Time Mach*, Proc., IEEE Real-time Technology and Applications Symposium, Boston, MA, June 1996, pp. 220-229.

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

[22] G. Bollella, K. Jeffay, *Support For Real-Time Computing Within General Purpose Operating Systems: Supporting co-resident operating systems*, Proc., IEEE Real-Time Technology and Applications Symposium, Chicago, IL, May 1995, pp. 4-14.

[23] 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.