Application Performance in the QLinux Multimedia Operating System*

Vijay Sundaram, Abhishek Chandra, Pawan Goyal†
Prashant Shenoy, Jasleen Sahni‡and **Harrick Vin§**

Department of Computer Science, University of Massachusetts Amherst.
†IBM Almaden Research Center.
‡Department of Computer Science, University of North Carolina at Chapel Hill.
§Department of Computer Sciences, University of Texas at Austin.

Abstract

In this paper, we argue that conventional operating systems need to be enhanced with predictable resource management mechanisms to meet the diverse performance requirements of emerging multimedia and web applications. We present QLinux—a multimedia operating system based on the Linux kernel that meets this requirement. QLinux employs hierarchical schedulers for fair, predictable allocation of processor, disk and network bandwidth. We experimentally evaluate the efficacy of these mechanisms using benchmarks and real-world applications. Our experimental results show that (i) emerging applications can indeed benefit from predictable allocation of resources, and (ii) the overheads imposed by the resource allocation mechanisms in QLinux are small. For instance, we show that the QLinux CPU scheduler can provide predictable performance guarantees to applications such as web servers and MPEG players, albeit at the expense of increasing the scheduling overhead. We conclude from our experiments that the benefits due to the resource management mechanisms in QLinux outweigh their increased overheads, making them a practical choice for conventional operating systems.

1 Introduction

Recent advances in computing and communication technologies have led to the emergence of a wide variety of applications with diverse performance requirements. Today's general purpose operating systems are required to support a mix of (i) conventional best-effort applications that desire low average response times but no absolute performance guarantees, (ii) throughput-intensive applications that desire high average throughput, and (iii) soft real-time applications that require performance guarantees from the operating system. To illustrate, PCs in office environments run a mix of word processors, spreadsheets, streaming media players and large compilation jobs, while large-scale servers run a mix of network file services, web services, database applications and streaming media servers.

*A preliminary version of this paper appeared in the proceedings of the ACM Multimedia conference, Los Angeles, CA, November 2000.

Whereas less demanding application mixes can be easily handled by a conventional best-effort operating system running on a fast processor, studies have shown that such operating systems are grossly inadequate for meeting the diverse requirements imposed by demanding application mixes [12, 14]. To illustrate, conventional operating systems running on even the fastest processors today are unable to provide jitter-free playback of full-motion MPEG-2 video in the presence of other applications such as long-running compile tasks. The primary reason for this inadequacy is the lack of service differentiation among applications—such operating systems provide a single class of best-effort service to all applications regardless of their actual performance requirements.[1] Moreover, special-purpose operating systems designed for a particular application class (e.g., real-time operating systems [11, 23]) are typically unable or inefficient at handling other classes of applications. This necessitates the design of an operating system that (i) multiplexes its resources among applications in a predictable manner, and (ii) uses service differentiation to meet the performance requirements of individual applications.

The QLinux operating system that we have developed meets these requirements by enhancing the standard Linux operating system with quality of service support. To do so, QLinux employs schedulers that can allocate resources to individual applications as well as application classes in a predictable manner. These schedulers are hierarchical—they support class-specific schedulers that schedule requests based on the performance requirements of that class (and thereby provide service differentiation across application classes). Specifically, QLinux employs three key components: (i) hierarchical start-time fair queueing (H-SFQ) CPU scheduler that allocates CPU bandwidth fairly among application classes [6], (ii) hierarchical start-time fair queueing (H-SFQ) packet scheduler that can fairly allocate network interface bandwidth to various applications [7], and (iii) Cello disk scheduler that can support disk requests with diverse performance requirements [16]. Figure 1 illustrates these components. We have implemented these components into QLinux and have made the source code freely available to the research community.[2]

In this paper, we make four key contributions. First, we show how to synthesize several recent innovations in OS resource management into a seamless multimedia operating system. Second, we consider several real-world applications and application scenarios and demonstrate that these resource management techniques enable QLinux to provide benefits such as predictable performance, application isolation and fair resource allocation. For instance, we show that QLinux enables a streaming media server to stream MPEG-1 files at their real-time rates regardless of the background load. Third, we show that existing/legacy applications can also benefit from these features without any modifications whatsoever to the application source code. Finally, we show that the

[1] Rather than reduce the processor shares of all applications equally, an operating system that provides service differentiation might reduce the fraction of the CPU bandwidth allocated to best-effort compile jobs and thereby reduce the jitter in soft real-time video playback.

[2] Source code and documentation for QLinux is available from http://www.cs.umass.edu/~lass/software/qlinux.

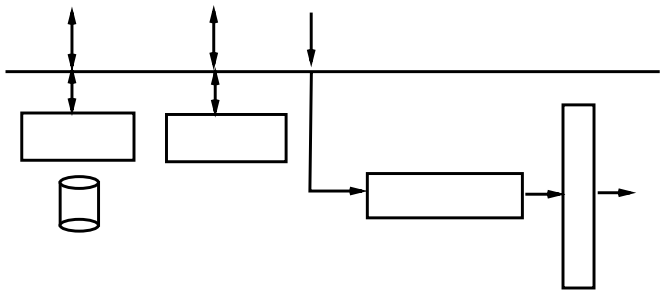


Figure 1: Key components of QLinux.

implementation overheads of these sophisticated resource management techniques are small, making them a practical choice for general-purpose operating systems. For instance, we show that the context switch overhead due to the H-SFQ CPU scheduler increases from 1 μs to 4 μs, but the increased overhead is still substantially smaller than the quantum duration. Based on these results, we argue that conventional operating systems should be enhanced with such resource management mechanisms so as to meet the needs of emerging applications as well as existing and legacy applications.

The rest of this paper is structured as follows. Section 2 discusses the principles underlying the design of QLinux and briefly describes each component employed by QLinux. Section 3 presents the results of our experimental evaluation. Section 4 discusses related work, and finally, Section 5 presents some concluding remarks.

2 QLinux Philosophy and Overview

In this section, we first present the principles underlying the design and implementation of QLinux. We then briefly describe each resource management component employed by QLinux (these mechanisms are described in detail elsewhere [6, 7, 16]).

2.1 QLinux Design Principles

The design and implementation of QLinux is based on the following principles:

- *Support for Multiple Service Classes:* Today's general purpose computing environments consist of a heterogeneous mix of applications with different performance requirements. As argued in Section 1, operating systems that provide a single class of service to all applications are inadequate for handling such diverse application mixes. To efficiently support such mixes, an operating system should *support*

multiple classes of service and align the service provided within each class with application needs. For instance, an operating system may support three classes of service—interactive, throughput-intensive and soft real-time—and treat applications within each class differently (interactive applications are provided low average response times, real-time applications are provided performance guarantees, and throughput-intensive applications are provided high aggregate throughput). Other operating systems such as Nemesis [15] have also espoused such a *multi-service* approach to operating system design.

- *Predictable resource allocation:* A multi-service operating system requires mechanisms that can multiplex its resources among applications in a predictable manner. Many operating systems (e.g., Solaris, UNIX SVR4) support multiple application classes using strict priority across classes. Studies have shown that such an approach can induce starvation in lower priority tasks even for common application mixes [12]. For instance, it has been shown that running a compute-intensive MPEG decoder in the highest priority real-time class on Solaris can cause even kernel tasks (which run at a lower priority) to starve, causing the entire system to "freeze" [12]. One approach to alleviate the starvation problem is to use dynamic priorities. Whereas the design of dynamic priority mechanisms for homogeneous workloads is easy, the design of such techniques for heterogeneous workloads is challenging. Consequently, QLinux advocates rate-based mechanisms over priority-based mechanisms for predictable resource allocation. Rate-based techniques allow a weight to be assigned to individual applications and/or application classes and allocate resources in proportion to these weights. Thus, an application with weight w_i is allocated $\frac{w_i}{\sum_j w_j}$ fraction of the resource.[3] Observe that, rate-based allocation techniques are distinct from static partitioning of resources—they can dynamically reallocate resources unused by an application to other applications, and thereby yield better resource utilization than static partitioning.

- *Service differentiation:* Since different application classes have different performance requirements, an operating system that supports multiple service classes should provide service differentiation by treating applications within each class differently. To do so, QLinux employs hierarchical schedulers that support multiple class-specific schedulers via a flexible multi-level scheduling structure. A hierarchical scheduler in QLinux allocates a certain fraction of the resource to each class-specific scheduler using rate-based mechanisms; class-specific schedulers, in turn, use their allocations to service requests using an appropriate scheduling algorithm. The flexibility of using a different class-specific scheduler for each class allows

[3] Such a resource allocation mechanism performs relative allocations—the fraction allocated to an application depends on the weights assigned to other applications. Rate-based mechanisms that allocate resource in absolute terms have also been developed. Such mechanisms allow applications to be allocated an absolute fraction f_i ($\sum f_i < 1$), or allocate x_i units every y_i units of time. We chose a relative allocation mechanism based on weights due to its simplicity.

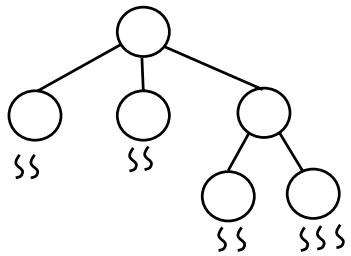


Figure 2: A sample hierarchy employed by the H-SFQ CPU scheduler. The figure shows three classes—interactive, throughput-intensive and soft real-time—with equal share of the processor bandwidth. The bandwidth within the soft real-time class is further partitioned among the audio and video classes in the proportion 1:4. Individual threads can also be assigned weights, assuming the leaf node scheduler supports rate-based allocation.

QLinux to tailor its service to the needs of individual applications. Moreover, the approach is extensible since it allows existing class-specific schedulers to be modified, or new schedulers to be added.

- *Support for legacy applications:* We believe that only those mechanisms that preserve compatibility with existing and legacy applications are likely to be adapted by mainstream operating systems in the near future. Hence, QLinux chooses an incremental approach to OS design. Each mechanism within QLinux is carefully designed to maintain full compatibility with existing applications at the binary level. We also decided that mere compatibility was not enough—we wanted existing applications to possibly benefit (but definitely not suffer) from the new resource allocation mechanisms in QLinux (although the degree to which they benefit would be less than new applications that are explicitly designed to take advantage of these features).

Next, we describe the three key components of QLinux.

2.2 Hierarchical Start-time Fair Queuing (H-SFQ) CPU Scheduler

Hierarchical start-time fair queuing (H-SFQ) is a hierarchical CPU scheduler that fairly allocates processor bandwidth to different application classes and employs class-specific schedulers to manage requests within each class [6]. The scheduler uses a tree-like structure to describe its scheduling hierarchy (see Figure 2). Each process or thread in the system belongs to exactly one leaf node. A leaf node is an aggregation of threads and represents an *application class* in the system. Each non-leaf node is an aggregation of application classes. Each node in the tree has a weight that determines the fraction of its parent's bandwidth that should be allocated to it. Thus, if $w_1, w_2, \ldots, w_n$ denote the weights on the n children of a node, and if B denotes the processor

bandwidth allocated to the node, then the bandwidth received by each child node i is given by

$$B_i = \left(\frac{w_i}{\sum_j w_j} \right) * B$$

Each node is also associated with a scheduler. Whereas the scheduler of the leaf node schedules all threads belonging to the leaf, the scheduler of an intermediate node schedules all its children. Scheduling of threads occurs hierarchically in H-SFQ: the root node schedules one of its child nodes; the child node, in turn, schedules one of its children until a leaf node schedules a thread for execution. Any class-specific scheduler may be employed to schedule a leaf node. For instance, the standard time-sharing scheduler could be employed for scheduling threads in the interactive class, whereas the earliest deadline first (EDF) scheduler could be used to schedule soft real-time tasks. H-SFQ employs start-time fair queuing (SFQ) as the scheduling algorithm for a non-leaf node. SFQ is a rate-based scheduler that allocates weighted fair shares—bandwidth allocated to each child node is in proportion to its weight. Bandwidth unused by a node is redistributed to other nodes according to their weights. In addition to rate-based allocation, SFQ has the following properties: (i) it achieves fair allocation of CPU bandwidth regardless of variation in available capacity, (ii) it does not require the length of the quantum to be known a priori (and hence, can be used in general-purpose environments where threads may block for I/O before their quantum expires), and (iii) SFQ provides provable guarantees on fairness, delay, and throughput received by each thread in the system [6, 7].

H-SFQ replaces the standard time-sharing scheduler in QLinux. The default scheduling hierarchy in H-SFQ consists of a root node with a single child that uses the standard time-sharing scheduler to schedule threads. An application, by default, is assigned to the time-sharing scheduler, thereby allowing QLinux to mimic the behavior of standard Linux. The scheduling hierarchy can be modified dynamically at run-time by creating new nodes on the fly. Creating a new node involves specifying the parent node, a weight, and a scheduling algorithm, if the node is a leaf node (non-leaf nodes are scheduled using SFQ). QLinux allows processes and threads to be assigned to a specific node at process/thread creation time; processes and threads can be moved from one leaf node to another at any time. Moreover, weights assigned to an application or a node in the scheduling hierarchy can be modified dynamically. QLinux employs a set of system calls to achieve these objectives (see Table 1). We have also implemented several utility programs to manipulate the scheduling hierarchy as well as individual applications within the hierarchy. These utilities allow existing/legacy applications to benefit from the features of H-SFQ since users can assign weights to applications without modifying the source code.

As mentioned before, the H-SFQ CPU scheduler uses a tree-like structure to describe its scheduling hierarchy. Each node in the scheduling structure has a weight, a start-tag, and a finish-tag that are maintained as per the

Table 1: System call interface supported by the H-SFQ CPU scheduler

System call	Purpose
`hsfq_mknod`	create a new node in the scheduling hierarchy
`hsfq_rmnod`	delete an existing node from the hierarchy
`hsfq_join_nod`	attach the current process to a leaf node
`hsfq_move`	move a process to a specified child node
`hsfq_parse`	parse a pathname in the scheduling hierarchy
`hsfq_admin`	administer a node (e.g., change weights)

SFQ algorithm. A non-leaf node maintains a list of child nodes, a list of runnable child nodes sorted by their start tags, and a virtual time of the node which, as per SFQ, is the minimum of the start-tags of the runnable child nodes. A leaf node has a pointer to a function that is invoked, when it is scheduled by its parent node, to select one of its thread for execution; this function implements the scheduling algorithm for this leaf node. Given a scheduling structure, the actual scheduling of threads occurs recursively. To select a thread for execution, a function `hsfq_schedule()` is invoked. This function traverses the scheduling structure by always selecting the child node with the smallest start tag until a leaf node is selected. When a leaf node is selected, a function that is dependent on the leaf node scheduler, determined through the function pointer that is stored in the leaf node, is invoked to determine the thread to be scheduled. When a thread blocks or is preempted, the finish and start tags of all the ancestors of the node to which the thread belongs have to be updated. This is done by invoking a function `hsfq_update()` with the duration for which the thread executed and the node identifier of the leaf node as parameters. A node in the scheduling structure is scheduled if and only if at least one of the leaf nodes in the sub-tree rooted at that node has a runnable thread. The eligibility of the node is determined as follows. When the first thread in the leaf node becomes eligible for scheduling, function `hsfq_setrun()` is invoked with the leaf nodes identifier. This function marks the leaf node as runnable and all the other ancestor nodes that may become eligible as a consequence. Note that this function has to traverse the path from the leaf up the tree only until a node that is already runnable is found. On the other hand, when the last thread in a leaf node makes a transition to sleep mode, function `hsfq_sleep()` is called with the leaf node's identifier. This function marks the leaf node as ineligible and all the other ancestor nodes that may become ineligible as a consequence. This function has to traverse the path from the leaf only until a node that has more than one runnable child node is found. Finally, in our implementation, any scheduling algorithm can be used at the leaf node as long as it: (1) provides an interface function that can be invoked by `hsfq_schedule()` to select the next thread for execution, and (2) invokes `hsfq_setrun()`, `hsfq_sleep` and `hsfq_update()` as per the rules defined above.

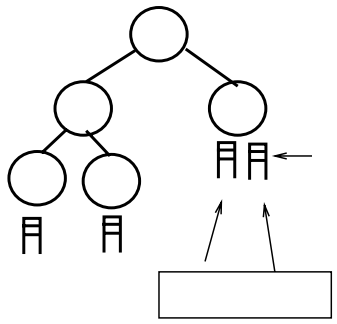


Figure 3: The H-SFQ network packet scheduler. The figure shows a sample scheduling hierarchy with two classes—http and soft real-time. The bandwidth within the http class is further partitioned among two web domains, $D1$ and $D2$, in the ratio 1:1. Note that individual sockets can either share a queue or have a queue of their own. Since each queue has its own weight, in the latter case, bandwidth allocation can be controlled on a per-socket basis.

2.3 H-SFQ Packet Scheduler

An operating system employs a packet scheduler at each of its network interfaces to determine the order in which outgoing packets are transmitted. Traditionally, most operating systems have employed the FIFO scheduler to schedule outgoing packets. To better meet the needs of applications with different requirements, QLinux employs H-SFQ to schedule outgoing packets. As described in Section 2.2, H-SFQ can fairly allocate resource bandwidth among different application classes in a hierarchical manner. As in the case of CPU, the H-SFQ packet scheduler employs a multi-level tree-like scheduling structure to hierarchically allocate network interface bandwidth (see Figure 3). Each leaf node in the tree consists of one or more queues of outgoing network packets and any class-specific scheduler can be employed to schedule the transmission of packets from these queues; the default leaf scheduler is FIFO. A non-leaf node is scheduled using SFQ. Every node in the hierarchy is assigned a weight; H-SFQ allocates bandwidth to nodes in proportion to their weights. Bandwidth unused by a node is reallocated fairly among the nodes with pending packets, thereby improving overall utilization.

The H-SFQ packet scheduler in QLinux replaces the FIFO scheduler employed by Linux. The default scheduling hierarchy in H-SFQ is a root node with a single child that employs FIFO scheduling. Packets sent by applications are, by default, queued up at this node, enabling QLinux to emulate the behavior of Linux. As in the case of the CPU scheduler, the scheduling hierarchy can be modified by adding new nodes to the tree or deleting existing nodes. QLinux allows applications to be associated to a specific queue at a leaf node (via the `setsockopt` system call); this association can be done on a per-socket basis. Packet classifiers [18] are then employed to map each transmitted packet to the corresponding queue at a leaf node. Table 2 lists the system call interface exported by the packet scheduler to achieve these objectives. We are currently implementing utility

Table 2: System call interface supported by the H-SFQ packet scheduler

System call	Purpose
`hsfq_qdisc_install`	Install HSFQ queuing discipline at a network interface
`hsfq_link_mknod`	create a node in the scheduling hierarchy
`hsfq_link_createq`	create a packet queue
`hsfq_link_attachq`	attach a queue to a leaf node
`hsfq_link_moveq`	move a queue between schedulers
`hsfq_link_rmnod`	delete the specified node
`hsfq_link_rmq`	delete the specified queue
`hsfq_link_modify`	change the weight of a node/queue
`hsfq_link_parsenode`	parse a pathname in the scheduling hierarchy
`hsfq_link_getroot`	get the ID of the root node at a particular network interface
`hsfq_link_status`	display the scheduling tree
`setsockopt`	attach a socket to a queue

programs using these system calls that will enable existing applications to benefit from these features without having to modify their source code.

Our implementaion of the H-SFQ packet scheduler is similar in nature to the implementation of the H-SFQ CPU scheduler. A similar tree-like structure is used to describe the scheduling hierearchy. The functional equivalent of a thread being scheduled by the CPU scheduler, is a queue being scheduled for service by the packet scheduler. And the functions `hsfq_dequeue()`, `hsfq_wakeup_node()`, `hsfq_sleep_node` and `hsfq_net_update()`, respectively, in the packet scheduler, perform actions similar to that of `hsfq_schedule()`, `hsfq_setrun()`, `hsfq_sleep` and `hsfq_update()`, respectively, in the CPU scheduler. Again, as in the CPU-scheduler, any scheduling algorithm can be used at the leaf-scheduler.

2.4 Cello Disk Scheduler

Unlike disk scheduling algorithms such as SCAN that provide a best-effort service to disk requests, QLinux employs the Cello disk scheduling algorithm to support multiple application classes. Cello services disk requests using a two level scheduling algorithm, consisting of a class-independent scheduler and a set of class-specific schedulers [16]. The class-independent scheduler is responsible for allocating disk bandwidth to classes based on their weights, whereas the class-specific schedulers use these allocations to schedule individual requests based on their requirements. Unlike pure rate-based schedulers that focus only on fair allocation of resources, Cello

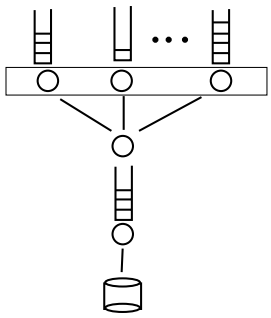


Figure 4: The Cello disk scheduling algorithm.

Table 3: System call interface supported by Cello

System call	Purpose
`cello_open`	Open a file and associate it with the specified class
`cello_close`	Close a file
`cello_read`	read data using an optional deadline
`cello_write`	write data using an optional deadline
`cello_set_class`	associate a class with a process
`cello_get_class`	get the class associated with a process
`cello_admin`	administer a class (e.g., specify weights)
`cello_start`	Enable cello scheduling for a device
`cello_stop`	Disable cello scheduling for a device
`cello_round`	Change the round duration for Cello scheduling

also takes disk seek and rotational latency overheads into account when making scheduling decisions (thereby improving disk throughput).

The implementation of Cello in QLinux supports three application classes—interactive, throughput-intensive and soft real-time. To do so, QLinux maintains three pending queues, one for each application class and a scheduled queue (see Figure 4). Newly arriving requests are queued up in the appropriate pending queue. They are eventually moved to the scheduled queue and dispatched to the disk in FIFO order. The class-independent scheduler determines *when* and *how many* requests to move from each class-specific pending queue to the scheduled queue, while the class-specific schedulers determine *where* to insert it into the scheduled queue. In QLinux, Cello disk scheduling is achieved by means of a kernel thread; one such thread is required for each device (disk) for which Cello disk scheduling is desired. These threads can either be started at boot up time or they can be started using a command line utility. These threads are responsible for moving requests from the pending queue to the scheduled queue, as well as for issuing scheduled requests to the device queue. Cello partitions disk bandwidth between application classes over periods of fixed length also called a *round*; the time share of each application class over a round is in proportion to it's weight. The kernel thread maintains round statistics, such as the time elapsed in the round, as well as the total and the remaining time share of each application class in the current round; this information is used for making scheduling decisions. In addition, our implementation of the Cello disk scheduler is work-conserving; when the scheduler detects the disk to be idle, it schedules requests from the application classes which have exhausted their share in a round-robin manner.

In Linux all read and write requests which enter the block I/O layer are broken into fundamental operations described by buffer_heads. These buffer_heads are the basic I/O descriptors used by device drivers. Once buffer_heads have been created to describe a particular I/O operation, they are passed onto some functions to be queued for a mid-level driver (e.g., scsi or ide). The mid-level driver periodically removes requests which have been placed on its queue and sends them to lower level host bus adapter drivers. In QLinux these buffer_heads are inserted in the appropriate pending queue for scheduling by the kernel thread, instead of the block device queue as in Linux. The kernel thread, which implements Cello scheduling, then schedules these buffer heads for insertion in the block device queue. The pending queues are maintained a layer just below the *Logical Volume Manager* (LVM) and the *Software RAID* driver layer, so Cello scheduling is also transparent to the presence of any logical volume or software RAID device. The only requirement is that there be a kernel thread for each disk for which Cello scheduling is desired; this also has the added advantage that Cello scheduling can be selectively enabled only for those disks for which it is desired.

Table 3 lists the interface exported by Cello. An application can associate a request with an application class in one of two ways.

- We provide the system call `cello_open` to allow applications to associate file I/O with an application class; all subsequent read and write operations on the file are then associated with the specified class. For the soft real-time class, an application must also specify a deadline with each read or write request. The use of our enhanced open system call interface requires application source code to be modified.

- We also provide a command line utility that allows a process (or thread) to be associated with an application class—all subsequent I/O from the process is then associated with that class. Any child processes that are forked by this process inherit these attributes and their I/O requests are treated accordingly. This enables legacy applications to benefit from our techniques. For the real-time class, we tag the I/O request with a fixed deadline (from the time of arrival) inside the kernel.

In our implementation requests not assigned a class are made to bypass Cello scheduling. This is done because assigning these requests a default class would interfere with the class based scheduling of the Cello disk scheduler. Moreover, this has the added advantage that no kernel I/O requests get delayed as a result of a class having exhausted it's share for the current round.

A disk I/O request has to percolate through several levels before it is satisfied by the disk subsystem. In a typical operating system these levels may include system call interface, file system, virtual memory manager, buffer cache, operating system block device support and the device drivers. If the application uses the first approach listed above of using the system call `cello_open` to associate a request with an application class, this information has to be passed down through all the layers listed above; this involved extending several kernel data structures so that the information is available in order to insert the request in the appropriate pending queue. In the second approach, which allows a process to be associated with an application class, this information is available as part of the process context; so it is passed on to the buffer_head just before it is inserted in the pending queue.

Finally, in our implementation we disable filesystem readahead for requests in the soft real-time class. This is because in many operating systems including Linux, the file system prefetches blocks of a file when it detects that the file is being sequentially accessed. Moreover, the prefetch window is aggressively increased if prefetched pages see hits in the buffer cache. Applications in the soft real-time class typically access data in rounds sequentially, and readahead requests may retrieve data which may not be required immediately. If these requests use up the share of the soft real-time class, this would impact the throughput of other applications in the same class. Hence, we disable readahead for this class.

Note that the current implementation of Cello supports bandwidth allocation only on a per-class basis; in the future, we plan to add support for bandwidth allocation on a per-application basis. Also the current implemen-

tation provides for Cello disk scheduling only for requests issued to a local disk.

3 Experimental Evaluation

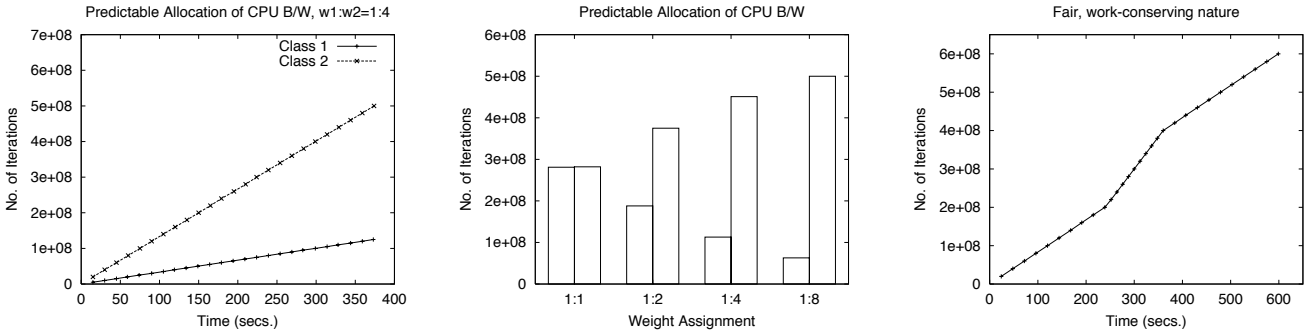
In this section, we experimentally evaluate the performance of QLinux and compare it to vanilla Linux. In particular, we examine the efficacy of the resource allocation mechanisms within QLinux to (i) allocate resource bandwidth in a predictable manner, (ii) provide application isolation, and (iii) support multiple traffic classes. We use several real applications, benchmarks and micro-benchmarks for our experimental evaluation. In what follows, we first describe the test-bed for our experiments and then present the results of our experimental evaluation.

3.1 Experimental Setup

The test-bed for our experiments consists of a cluster of PC-based workstations. Each PC is equipped with a 100 Mb/s 3-Com ethernet card (model 3c595); all machines are interconnected by a 100 Mb/s ethernet switch (model 3Com SuperStack II). The version of QLinux used in our experiments is based on the 2.4.4 Linux kernel; comparisons with vanilla Linux use the identical version of the kernel. All machines and the network are assumed to be lightly loaded during our experiments.

The workload for our experiments consists of a combination of real-world applications, benchmarks, and sample applications that we wrote to demonstrate specific features. These applications are as follows: (i) *Inf:* an application that executes an infinite loop and represents a simple compute-intensive best-effort application; (ii) *mpeg_play:* the Berkeley software MPEG-1 decoder; represents a compute-intensive soft real-time application; (iii) *Apache web server* and *webclient*: a widely-used web server and a configurable client application that generates http requests at a specified rate; represents an I/O-intensive best-effort application; (iv) *Streaming media server:* a server that transmits (*streams*) MPEG-1 files over the network using UDP; represents an I/O-intensive soft real-time application; (v) *Net_inf:* an application that sends UDP data as fast as possible on a socket; represents an I/O-intensive best-effort application; (vi) *Dhrystone:* a compute-intensive benchmark for measuring integer CPU performance; (vii) *IO_inf:* a closed-loop process with *concurrency* N which consists of N concurrent clients that issue requests continuously, i.e., each client issues a new request as soon as the previous request completes; the request sizes are assumed to be fixed and successive requests access the file at a random offset; (viii) *lmbench:* a comprehensive benchmark suite that measures various aspects of operating system performance such as context switching, memory, file I/O, networking, and cache performance.

In what follows, we present the results of our experimental evaluation using these applications and bench-



(a) (b) (c)

Figure 5: Predictable, fair allocation of processor bandwidth by the H-SFQ scheduler

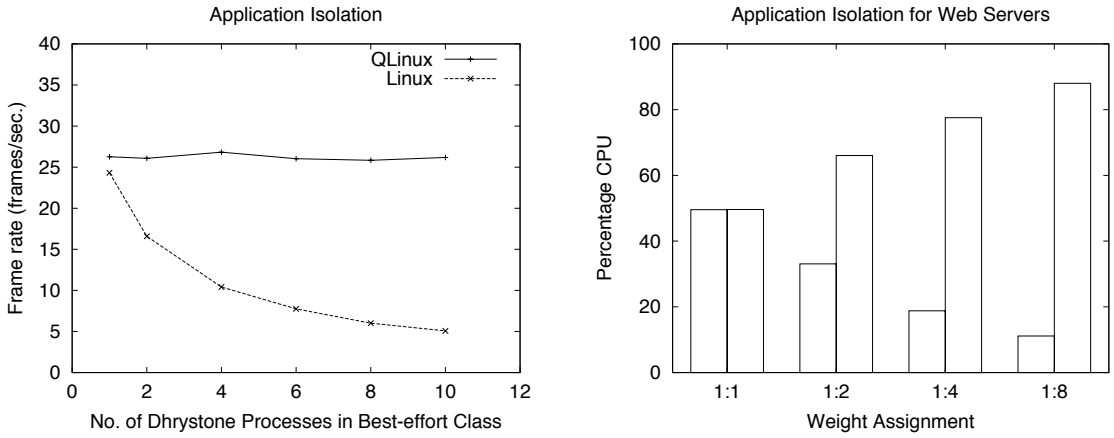
marks. Since the code for the Cello disk scheduler was unstable at the time of writing, we have not included experimental results for Cello.

3.2 Supporting Multiple Application Classes using the H-SFQ CPU Scheduler

To demonstrate that the H-SFQ CPU scheduler can allocate CPU bandwidth to applications in proportion to their weights, we created two classes in the scheduling hierarchy and ran the *Inf* application in each class. We assigned different combination of weights to the two classes (e.g., 1:1, 1:2, 1:4) and measured the number of loops executed by *Inf* in each case. Figures 5(a) and (b) depict our results. Figure 5(a) shows the progress made by the two *Inf* applications for a specific weight assignment of 1:4. Figure 5(b) shows the number of iterations executed by the two processes at t=337 seconds for different weight assignments. Together, the two figures show that each application gets processor bandwidth in proportion to its weight.

Next, we conducted an experiment to demonstrate the fair work-conserving nature of H-SFQ. Again, we created two application classes and gave them equal weights (1:1). The *Inf* application was run in each class and as expected each received 50% of the CPU bandwidth. At t=250 seconds, we suspended one of the *Inf* processes. Since H-SFQ is work-conserving in nature, the scheduler reallocated bandwidth unused by the suspended processes to the running *Inf* process (causing it's rate of progress to double). The suspended process was restarted at t=350 seconds, causing the two processes to again receive bandwidth in the proportion 1:1. Figure 5(c) depicts this scenario by plotting the progress made by the continuously running *Inf* process. As shown, the process makes progress at twice the rate between $250 \leq t < 350$ and receives its normal share in other time intervals.

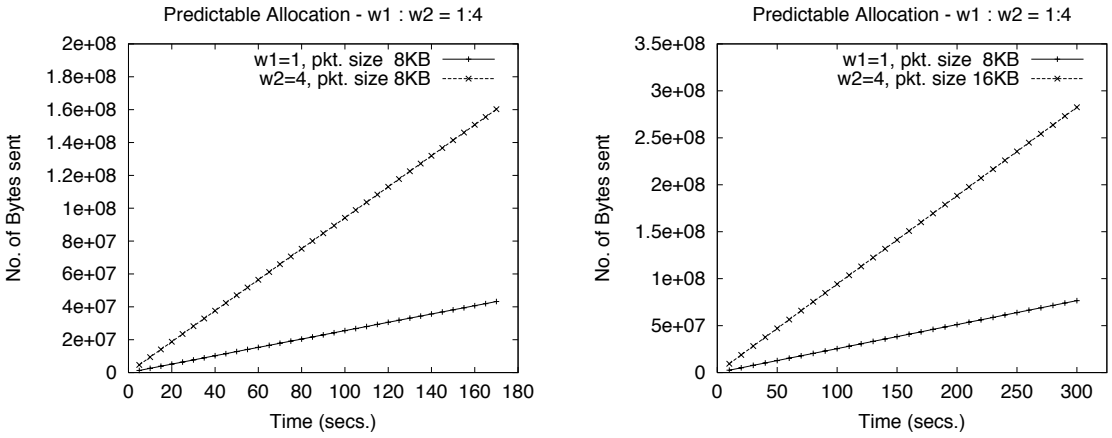
We then conducted experiments to show that real-world applications also benefit from H-SFQ. To show that the CPU scheduler can effectively isolate applications from one another, we created two classes—soft



(a) Isolation between soft real-time and text (b) Isolation between two web domains

Figure 6: Application isolation and flexibility in the H-SFQ CPU scheduler.

real-time and best-effort—and assigned them equal weights. The best-effort leaf class was scheduled using the standard time sharing scheduler, while the soft real-time leaf class was scheduled using SFQ. We ran the Berkeley software MPEG decoder (mpeg_play) in the soft real-time class and used it to decode a five minute long MPEG-1 clip with an average bit rate of 1.49 Mb/s. The Dhrystone benchmark constituted the load in the best-effort class. We increased the load in the best-effort class (by increasing the number of independent Dhrystone processes) and measured the CPU bandwidth received by the MPEG decoder in each case. We then repeated this experiment using vanilla Linux. Figure 6(a) plots our results. As shown in the figure, in case of QLinux, the CPU bandwidth received by the MPEG decoder was independent of the load in the best-effort classes. Since vanilla Linux employs a best-effort scheduler, all applications, including the MPEG decoder, are degraded equally as the load increases. This demonstrates that H-SFQ, in addition to proportionate allocation, can also isolate application classes from one another. To further demonstrate this behavior, we ran two Apache web servers in two different classes and gave them different weights. The *webclient* application was used to send a large number of http requests to each web server and we measured the processor bandwidth received by each class. As shown in Figure 6(b), the H-SFQ scheduler allocates processor bandwidth to the two classes in proportion to their weights. These experiments demonstrate that QLinux can be employed for web hosting scenarios where multiple web domains are hosted from the same physical server. Each web domain can be allocated a certain fraction of the resources and can be effectively isolated from the load in other domains.



(a) Same packet size (b) Different packet sizes

Figure 7: Predictable allocation in the H-SFQ Packet Scheduler.

3.3 Supporting Multiple Traffic Classes Using the H-SFQ Packet Scheduler

To demonstrate that the H-SFQ packet scheduler can allocate network interface bandwidth to applications in proportion to their weights, we created two classes in the scheduling hierarchy and ran the *Net_inf* application in each class. The UDP packets sent by *Net_inf* were received as fast as possible by a receiver process running on a lightly loaded PC. We varied the weights assigned to the two classes and measured the number of packets sent by the two processes for different weight assignments. Figure 7(a) depicts the number of bytes received from each *Net_inf* for one particular weight assignment (1:4). As expected, both classes receive bandwidth in proportion to their weights. To demonstrate that bandwidth received by a class is independent of the packet size, we repeated the experiment using different packet sizes for the two classes. Figure 7(b) shows that, despite using different packet sizes, the two classes again receive bandwidth in proportion to their weights.

To demonstrate that real-world applications also benefit from these features, we conducted an experiment with two classes—soft real-time and best-effort. The streaming media server was run in the soft real-time class and was used to stream a five minute long variable bit-rate MPEG-1 clip (average bit rate of the clip was 1.49 Mb/s). We ran an increasing number of *Net_inf* applications in the best-effort class and measured their impact on the bandwidth received by the streaming media server. We then repeated this experiment on vanilla Linux. As shown in Figure 8, QLinux is able to effectively isolate the streaming media server from the best-effort class—the server is able to stream data at its real-time rate regardless of the best-effort load. Linux, on the other hand, is unable to provide this isolation—increasing the best-effort load reduces the bandwidth received by the streaming media server and also increases the amount of packet loss incurred by all applications.

3.4 Supporting Multiple Application Classes Using the Cello Disk Scheduler

To demonstrate that the Cello disk scheduler can allocate disk bandwidth to applications in proportion to their weights, we configured the Cello disk scheduler with two application classes and ran one *IO_inf* application in each class. The concurrency of each *IO_inf* application was set to 6. Recall that the implementation of Cello in QLinux supports three application classes—interactive, throughput-intensive and soft real-time; in this experiment we set the weight of the soft real-time class to 0 and assigned different combinations of weights to the two remaining classes. We then repeated the experiment for vanilla Linux. Figures 9 (a) and (b) depict our results. Figure 9 (a) plots the fraction of requests serviced for class 2 to its fractional share of the disk bandwidth; the figure shows that the class gets disk bandwidth in proportion to it's weight. Figure 9 (b) plots the total number of requests serviced for Linux, as well as for different weight combinations in QLinux. As can be seen the total number of requests serviced is similar in all cases; this shows that the Cello disk scheduler has low overheads and has throughput comparable to that of vanilla Linux.

Next, we conducted an experiment to demonstrate application isolation. For this experiment we configured Cello with two application classes, the interactive class and the soft real-time class. The weights of the two application classes were chosen to be in the ratio 1:3. We ran the *streaming media server* application in the soft real-time class and the *IO_inf* application constituted the load in the interactive class. The streaming media server accessed four different files each at an average bit rate of 1.5 Mb/s. With the load in the soft real-time class fixed, we increased the load in the interactive class by increasing the number of *IO_inf* applications; the concurrency of each application was fixed at 2. We then repeated this experiment with vanilla Linux. Figure 10 plots the results. As can be seen in QLinux the streaming media server is able to retrieve data at the real-time rate regardless of the load in the interactive class. In Linux, on the other hand, the bandwidth received by the streaming media server decreases as one increases the load in the interactive class.

3.5 Combined Impact of H-SFQ Packet Scheduler and the Cello Disk Scheduler

In this experiment, we demonstrate the combined benefits of the HSFQ packet scheduler and the Cello disk scheduler. For this experiment we created two classes in the packet scheduler hierarchy; the Cello disk scheduler was also configured with two application classes, the interactive class and the soft real-time class. We ran the *streaming media server* application in the soft real-time class and the *IO_inf* application in the interactive class, respectively, of the Cello disk scheduler. We assigned the same *streaming media server* application to one of the classes in the packet scheduler and ran the *Net_inf* application in the other class (called best-effort). The use of different best-effort applications, *IO_inf* in the Cello disk scheduler and *Net_inf* in the packet scheduler, isolates

the performance of one from other; so these compete independently with the same *streaming media server* application. We set the weight of the best-effort class to the soft real-time class to be in the ratio 1:2 for both the Cello disk scheduler as well as the H-SFQ Packet scheduler. The streaming media server accessed a video file at 30 frames/s, which corresponded to an average bit rate of 1.5 Mb/s. Keeping the load of the *streaming media server* application constant we simultaneously increased the number of *IO_inf* and *Net_inf* applications. The concurrency of the *IO_inf* application was fixed at two. Note that the *streaming media server* application streamed data to the client in rounds. This involved reading 30 frames worth of data in each round, and then streaming it to the client, which was on another machine. The data was sent using 8 KB UDP packets. The round duration was one second. At the client we record the sequence number, the arrival time and the size of each packet received. We then computed the mean inter-arrival time of packets in each round, and computed the average over all the rounds for a given run. We also measured the percentage of packets lost, and the mean bit rate observed by the client. We then repeated the experiment with vanilla Linux. Figure 11 plots the results. The mean inter-arrival time of packets in a round and the percentage packet loss are impacted solely by the packet scheduler, as the data has already been retrieved before it is to be sent over the network. The mean bit rate however, is a result of the combined impact of the disk scheduler and the packet scheduler, as it is takes accounts for the data retrieval rate from disk, packet scheduling as well as packet loss. Figure 11 (a) shows that the mean inter-arrival time of packets increases for both vanilla Linux and QLinux as one increases the load in the best-effort class, however due to isolation provided by the packet scheduler the increase is less gradual in QLinux. Figures 11 (b) and (c) show that an increase in the number of best-effort clients results in increase of packet loss and a drop in the average bit rate in Linux; effective application isolation in QLinux isolates the performance of the *streaming media server* application, and it observes no packet loss and is able to stream at the real-time rate.

3.6 Combined Impact of H-SFQ CPU and Packet Schedulers

To demonstrate the combined benefits of the CPU and packet schedulers, we considered a scenario consisting of a loaded web server and several I/O intensive applications. We created two classes in the CPU and packet scheduler hierarchies. We ran a simulated web server in one CPU/packet scheduler class and ran all the I/O-intensive *Net_inf* applications in the other CPU/packet scheduler class. Our simulated web server consisted of a sender application that reads an actual web server trace and sends data using TCP (each send corresponds to an http request in the trace file; the timing and size of each request was taken directly from the information specified in the traces). The publicly-available ClarkNet server traces were employed to simulate the web server workload [4]. We increased the number of *Net_inf* applications in the best-effort class and measured their impact

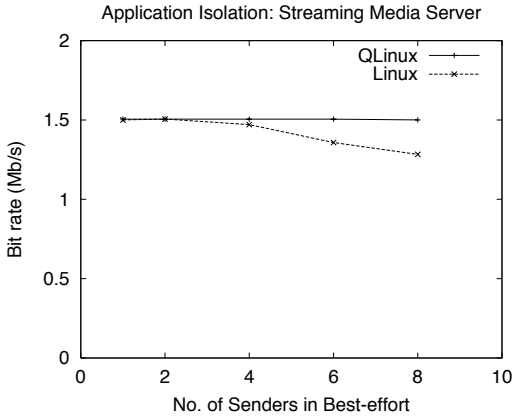
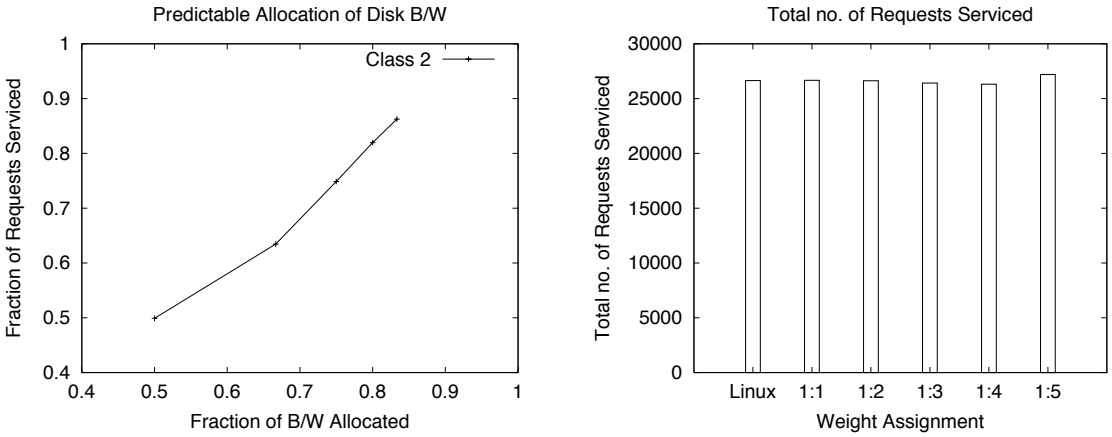


Figure 8: Application isolation in the H-SFQ Packet Scheduler.



(a) (b)

Figure 9: Predictable allocation in the Cello Disk Scheduler.

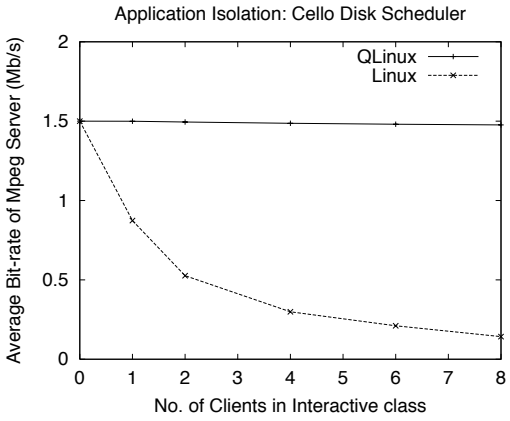
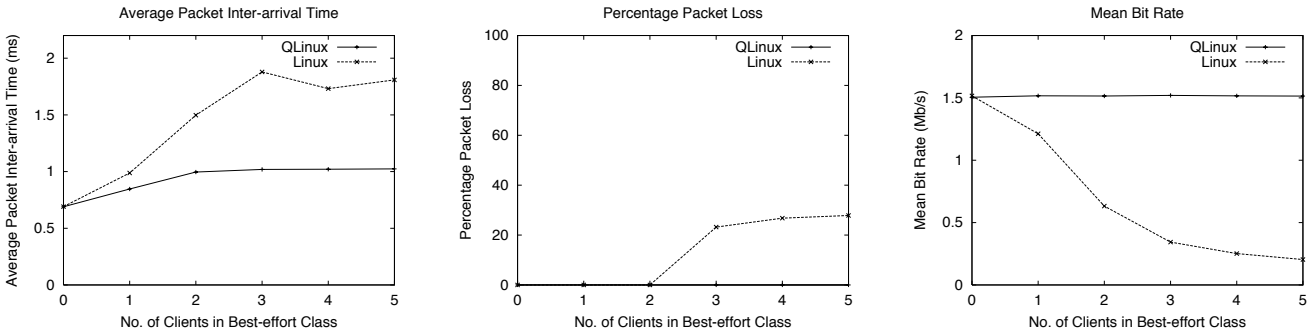


Figure 10: Application isolation in the Cello Disk Scheduler.



(a) (b) (c)

Figure 11: Impact of the H-SFQ packet scheduler and the Cello disk scheduler on video workloads.

on the throughput of the web server. The experiment was then repeated for vanilla Linux. Figure 12 depicts our results. Observe that, the web server simulates the http protocol which runs on TCP. TCP employs congestion control mechanisms that back off in the presence of congestion. Consequently, as the load due to *Net_inf* applications increases, congestion builds up in the ethernet switch interconnecting the senders and receivers (due to the presence of limited buffers at switches), causing TCP to reduce its sending rate. Both QLinux and Linux experience this phenomenon, resulting in a degradation in throughput for the web workload. However, since the QLinux CPU and packet schedulers reserve bandwidth for the web server, they can effectively isolate the web workload from the *Net_inf* applications. Hence, the degradation in throughput in QLinux is significantly smaller than that in Linux. This demonstrates that use of fair, predictable schedulers for each resource in an OS can yield significant performance benefits to applications.

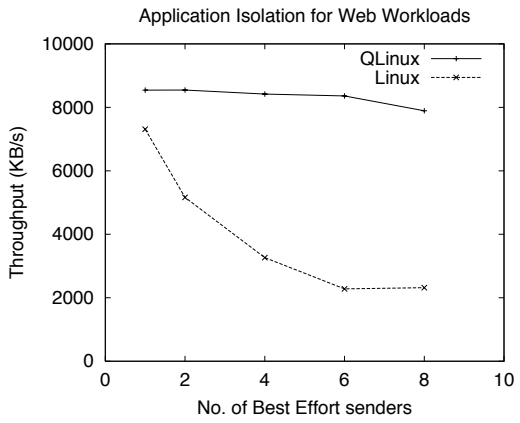


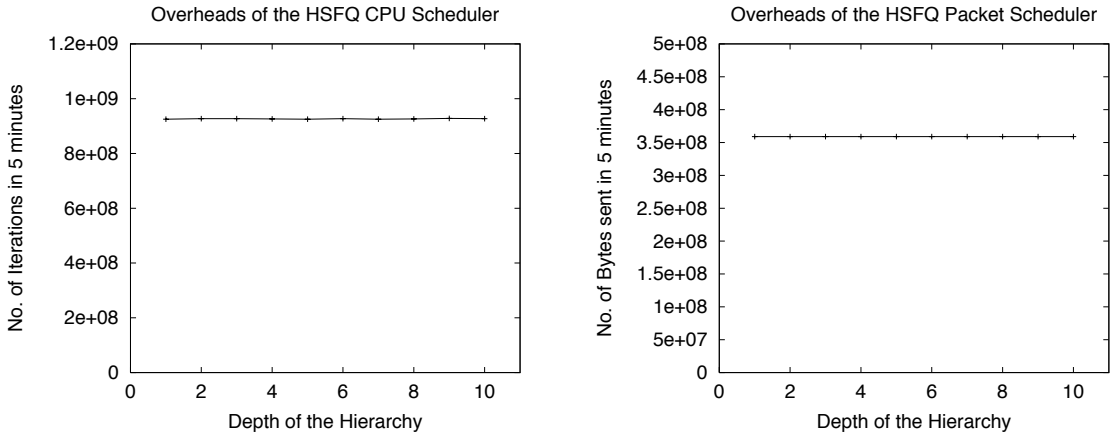
Figure 12: Impact of the H-SFQ CPU and packet schedulers on web workloads.

3.7 Microbenchmarking QLinux: Scheduling Overheads

In the previous sections, we demonstrated that applications can benefit from the sophisticated resource management techniques employed by QLinux. In what follows, we measure the overheads imposed by these mechanisms using microbenchmarks.

To measure the overhead imposed by the CPU scheduler, we created a leaf node and ran a solitary *Inf* process in that class. We then progressively increased the depth of the scheduling hierarchy (by introducing intermediate nodes between this leaf and the root) and measured the bandwidth received by *Inf* in each case. Observe that, increasing the depth of the scheduling hierarchy may increase the scheduling overhead (since H-SFQ recursively calls the scheduler at each intermediate node until a thread in the leaf class is selected). A larger scheduling overhead will correspondingly reduce the bandwidth received by applications (since a larger fraction of the CPU time would be spent in making scheduling decisions). Figure 13(a) plots the number of iterations executed by *Inf* in 300 seconds as we increase the depth of the scheduling hierarchy. As shown in the figure, the bandwidth received by *Inf* is relatively unaffected by the increasing scheduling overhead, thereby demonstrating that the overheads imposed by H-SFQ are small in practice.

We then performed a similar experiment for the H-SFQ packet scheduler. The experiment consisted of running the *Net_inf* process in a scheduling hierarchy with increasing depth and measuring the bandwidth received by *Net_inf* in each case. As in the case of the CPU scheduler, the bandwidth received by *Net_inf* was relatively unaffected by the scheduling overhead (see 13(b)). Together, these experiments show that hierarchical schedulers such as H-SFQ are feasible in practice.



(a) Overhead of CPU scheduler (b) Overhead of packet scheduler

Figure 13: Microbenchmarking QLinux: overheads imposed by the CPU and Packet Schedulers

Table 4: Lmbench Results

Test	QLinux	Linux
syscall overhead	1 μs	1 μs
`fork()`	400 μs	400 μs
`exec()`	2 ms	2 ms
Context switch (2 proc/ 0KB)	4 μs	1 μs
Context switch (16 proc/ 64KB)	286 μs	283 μs
Local UDP latency	47 μs	53 μs
File create (0 KB file)	21 μs	21 μs
File delete (0 KB file)	2 μs	2 μs

3.8 Benchmarking QLinux

In our final experiment, we employed the widely used *Lmbench* benchmark to compare QLinux and Linux. Lmbench is a sophisticated benchmark that measures several aspects of system performance, such as system call overheads, context switch times, network I/O, file I/O and memory performance [9]. We employed Lmbench version 1.9 for our experiments. We first ran Lmbench in the default best-effort class on QLinux and then repeated the experiment on Linux. In each case, we averaged the statistics reported by Lmbench over several runs to eliminate experimental error. Table 4 summarizes our results (Lmbench produces a large number of statistics; we only list those statistics that are relevant to QLinux).

Note that the QLinux code is untuned, while Linux code is carefully tuned by the Linux kernel developers. Table 4 shows that the performance of QLinux is comparable to Linux; however, the increased complexity of

the QLinux schedulers do result in a larger overhead. For instance, the context switch overhead increases from 1 μs to 4 μs for two active processes; however this overhead is still several orders of magnitude smaller than the quantum duration of 100 ms. The network latency for TCP and UDP, as well as file I/O overheads and system call overheads are comparable in both cases.

4 Related Work

The growing popularity of the multimedia applications has resulted in several research efforts that have focused on the design of predictable resource allocation mechanisms. Consequently, in the recent past, several techniques have been proposed for the predictable allocation of processor [6, 8, 13, 20, 21], network interface [3, 5, 7, 17] and disk [1, 10, 22] bandwidth. While each effort differs in the exact mechanism employed to provide predictable performance (e.g., admission control, rate-based allocation, fair queuing), the broad goals are similar—add quality of service support to an operating system. The key contribution of QLinux is to synthesize/integrate many of these mechanisms into a single system and demonstrate the benefits of this integration on application performance. Whereas the mechanisms instantiated in QLinux are based on our past work in this area, we believe that it would have been relatively easy to implement some other predictable resource allocation mechanisms and demonstrate similar benefits.

Some other recent operating system efforts have also focused on the design of predictable resource allocation mechanisms. The Nemesis operating system, for instance, employs mechanisms that provide quality of service guarantees when allocating processor, network and disk bandwidth [1, 15]. Unlike QLinux, which employs weights to express resource requirements, Nemesis requires applications to specify their resource requirements in terms of tuples (s, p, x), where s units of the resource are requested every p units of time, and x is the additional bandwidth requested, if available. Nemesis is a multi-service multimedia operating system that was designed from the grounds up; QLinux, on the other hand, builds upon the Linux kernel and benefits from the continuing enhancement made to the kernel by the Linux developers. The Eclipse operating system, based on FreeBSD, is in many respects similar to QLinux [2]. Like QLinux, Eclipse employs hierarchical schedulers to allocate OS resources (the actual scheduling algorithms that are employed are, however, different). Eclipse employs a special file system called `/reserv` that is used by applications to specify their resource requirements [2]. QLinux and Eclipse are independent and parallel research efforts, both of which attempt to improve upon conventional best effort operating systems. Finally, many commercial operating systems are beginning to employ some of these features. High end versions of Solaris 2.7, for instance, include a resource manager that enables fine-grain allocation of various resources to processes and process groups [19].

5 Concluding Remarks

Emerging multimedia and web applications require conventional operating systems to be enhanced along several dimensions. In this paper, we presented the QLinux multimedia operating system that enhances the resource management mechanisms in vanilla Linux. QLinux employs three key components: the H-SFQ CPU scheduler, the H-SFQ packet scheduler, and the Cello disk scheduler. Together, these mechanisms ensure fair, predictable allocation of processor, network and disk bandwidth. We experimentally demonstrated the efficacy of these mechanisms using benchmarks as well as common multimedia and web applications. Our experimental results showed that multimedia and web applications can indeed benefit from predictable resource allocation and application isolation offered by QLinux. Furthermore, the overheads imposed by these mechanisms were shown to be small. Based on these results, we argue that all conventional operating systems should be enhanced with such mechanisms to meet the needs of emerging applications.

As part of future work, we plan to enhance QLinux along several dimensions. In particular, we are designing resource allocation mechanisms that will enable QLinux to scale to large symmetric multiprocessors and clusters of servers.

Acknowledgments

T R. Vishwanath helped us develop the initial version of QLinux. Raghav Srinivasan helped with implementation of the Cello disk scheduler in QLinux. Gisli Hjalmtysson provided useful inputs during the inception of QLinux in the summer of 1998. Finally, we thank the many users of QLinux and the research community for providing valuable feedback (and bug reports) for enhancing QLinux.

References

[1] P. Barham. A Fresh Approach to File System Quality of Service. In *Proceedings of NOSSDAV'97, St. Louis, Missouri*, pages 119–128, May 1997.

[2] J. Blanquer, J. Bruno, M. McShea, B. Ozden, A. Silberschatz, and A. Singh. Resource Management for QoS in Eclipse/BSD. In *Proceedings of the FreeBSD'99 Conference, Berkeley, CA*, October 1999.

[3] S. Chen and K. Nahrstedt. Hierarchical Scheduling for Multiple Classes of Applications in Connection-Oriented Integrated-Services Networks. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems, Florence, Italy*, June 1999.

[4] ClarkNet Web Server Traces. Available from the Internet Traffic Archive http://ita.ee.lbl.gov, 1995.

[5] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pages 1–12, September 1989.

[6] P. Goyal, X. Guo, and H.M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of Operating System Design and Implementation (OSDI'96), Seattle*, pages 107–122, October 1996.

[7] P. Goyal, H. M. Vin, and H. Cheng. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM'96*, pages 157–168, August 1996.

[8] M B. Jones, D Rosu, and M Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the sixteenth ACM symposium on Operating Systems Principles (SOSP'97), Saint-Malo, France*, pages 198–211, December 1997.

[9] L. McVoy and C. Staelin. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of USENIX'96 Technical Conference, Available from http://www.bitmover.com/lmbench*, January 1996.

[10] A. Molano, K. Juvva, and R. Rajkumar. Real-time File Systems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. In *Proceedings of IEEE Real-time Systems Symposium*, December 1997.

[11] L. Molesky, K. Ramamritham, C. Shen, J. Stankovic, and G. Zlokapa. Implementing a Predictable Real-Time Multiprocessor Kernel–The Spring Kernel. In *Proceedings of the IEEE Workshop on Real-time Operating Systems and Software*, May 1990.

[12] J. Nieh, J. Hanko, J. Northcutt, and G. Wall. SVR4UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 41–53, November 1993.

[13] J. Nieh and M S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP'97), Saint-Malo, France*, pages 184–197, December 1997.

[14] J. Nieh and M. S. Lam. Multimedia on Multiprocessors: Where's the OS When You Really Need It? In *Proceedings of the Eighth International Workshop on Network and Operating System Support for Digital Audio and Video, Cambridge,U.K.*, July 1998.

[15] Timothy Roscoe. *The Structure of a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, April 1995. Available as Technical Report No. 376.

[16] P Shenoy and H M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *Proceedings of ACM SIGMETRICS Conference, Madison, WI*, pages 44–55, June 1998.

[17] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round Robin. In *Proceedings of ACM SIGCOMM'95*, pages 231–242, 1995.

[18] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast Scalable Level Four Switching. In *Proceedings of the ACM SIGCOMM'98, Vancouver, BC*, pages 191–202, September 1998.

[19] Solaris Resource Manager 1.0: Controlling System Resources Effectively. Sun Microsystems, Inc., http://www.sun.com/software/white-papers/wp-srm/, 1998.

[20] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proceedings of ASPLOS-VIII, San Jose, CA*, pages 181–192, October 1998.

[21] C. Waldspurger and W. Weihl. Stride Scheduling: Deterministic Proportional-share Resource Management. Technical Report TM-528, MIT, Laboratory for Computer Science, June 1995.

[22] R. Wijayaratne and A. L. N. Reddy. Providing QoS Guarantees for Disk I/O. Technical Report TAMU-ECE97-02, Department of Electrical Engineering, Texas A&M University, 1997.

[23] K. Zuberi, P. Pillai, and K G. Shin. EMERALDS: A Small-Memory Real-Time Microkernel. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 277–291, December 1999.