Retrofitting Quality of Service into a Time-Sharing Operating System

John Bruno, José Brustoloni, Eran Gabber, Banu Özden and Abraham Silberschatz
Information Sciences Research Center
Lucent Technologies — Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, USA
{jbruno, jcb, eran, ozden, avi}@research.bell-labs.com

Abstract

Theoretical aspects of proportional share schedulers have received considerable attention recently. We contribute practical considerations on how to retrofit such schedulers into mainstream time-sharing systems. In particular, we propose `/reserv`, a uniform API for hierarchical proportional resource sharing. The central idea in `/reserv` is associating resource reservations with *references* to shared objects (and not with the objects themselves). We discuss in detail the implementation of `/reserv` and several proportional share schedulers on FreeBSD; the modified system is called *Eclipse/BSD*. Our experiments demonstrate that the proposed modifications allow selected applications to isolate their (or their clients') performance from CPU, disk, or network overloads caused by other applications. This capability is increasingly important for soft real-time, multimedia, Web, and distributed client-server applications.

1 Introduction

On a typical system, multiple applications may contend for the same physical resources, such as CPU, memory, and disk or network bandwidth. An important goal for an operating system is therefore to schedule requests from different applications so that each application and the system as a whole perform well.

Resource management schemes of time-sharing operating systems, such as Unix [15] and Windows NT [8], often achieve acceptably low response time and high system throughput for time-sharing workloads. However, as explained in the following paragraphs, several trends make those schemes increasingly inappropriate.

First, many workloads now include real-time applications (e.g., multimedia). Unlike time-sharing applications, real-time ones must have their requests processed within certain performance bounds (e.g., minimum throughput). To support real-time applications correctly under arbitrary system load, the operating system must perform *admission control* and offer *quality of service* (QoS) guarantees: The operating system admits a request only if the operating system has set aside enough resources to process the request within the specified performance bounds.

Second, even for purely time-sharing workloads, the trend toward distributed client-server architectures increases the importance of *fairness*, that is, of preventing certain clients from monopolizing system resources. The fairness of time-sharing systems can be quite spotty. For example, time-sharing systems typically cannot isolate the performance of a Web site from that of other Web sites hosted on the same system. If one of the sites becomes very popular, the performance of the other sites may become unacceptably (and unfairly) poor.

Finally, the same trend toward client-server architectures also makes it necessary to manage resources *hierarchically*, that is, recursively allowing each client to grant to its servers part of the client's resources. For example, new Web and other user-level servers often need mechanisms for processing client requests with specified QoS and/or fairness bounds. However, time-sharing operating systems usually do not provide such mechanisms.

The mentioned shortcomings of time-sharing operating systems have motivated considerable recent

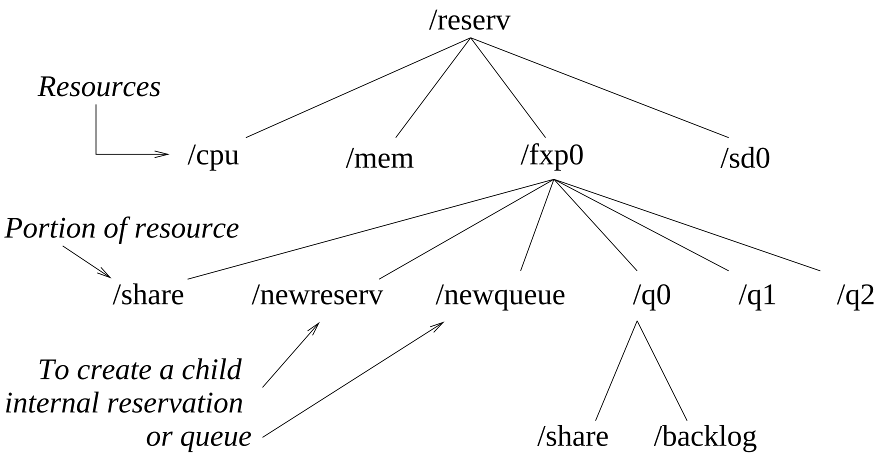

Figure 1: Eclipse/BSD's `/reserv` file system allows applications to create resource reservations.

work on new algorithms for CPU [14, 11, 13, 17, 6], disk [20, 21, 4], and network [2, 3, 12, 23] scheduling. In particular, we recently proposed MTR-LS, a new CPU scheduling algorithm with demonstrated throughput, delay, and fairness guarantees [6]. MTR-LS is an example of a *proportional share* scheduler. Proportional share schedulers stand out for their sound theoretical foundations [22, 11, 6, 2, 3, 23].

This paper considers the practical aspects of how to integrate proportional share schedulers into mainstream operating systems. We contribute a new application programming interface (API) for hierarchical proportional resource sharing: the `/reserv` file system. We discuss in detail the implementation of `/reserv` and several proportional share schedulers (MTR-LS, YFQ, H-WF^{2}Q) on FreeBSD. (FreeBSD is a freely available derivative of 4.4 BSD Unix; other Unix variants, Windows NT, and other time-sharing operating systems could be similarly modified.) We call the modified FreeBSD system *Eclipse/BSD*, as opposed to the *Eclipse/Plan 9* system used in our previous MTR-LS work [6] (where the distinction is obvious or unimportant, we will say simply *Eclipse*). Our experiments demonstrate how Eclipse/BSD's `/reserv` API and schedulers improve on FreeBSD's, providing QoS guarantees, fairness, and hierarchical resource management.

The rest of this paper is organized as follows. Section 2 describes the Eclipse/BSD resource management model and its retrofitting into FreeBSD. Section 3 discusses the scheduling algorithms used in Eclipse/BSD. Section 4 shows that Eclipse/BSD implementation requires only a modest amount of changes to FreeBSD. Experiments in Section 5 illustrate how Eclipse/BSD scheduling improves the isolation between Web sites hosted on the same system. Section 6 discusses related and future work, and Section 7 concludes.

2 Resource management model

This section describes the Eclipse/BSD hierarchical resource management model and its implementation on FreeBSD, including Eclipse/BSD's `/reserv` API.

2.1 Resource reservations

Eclipse/BSD applications obtain a desired quality of service by initially acquiring a *resource reservation* for each required physical resource. Physical resources include CPU, memory, disks, and network interfaces, each managed by a scheduler. A resource reservation specifies a fraction of the resource set aside for exclusive use by one or more processes. Applications can subdivide resource reservations hierarchically. Admission control guarantees that reservations do not exceed resources. Eclipse/BSD's schedulers share fractions of the respective resource fairly among all applications currently using the resource, as explained in the rest of this subsection.

Applications specify resource reservations as directories in a new file system mounted under `/reserv`. Each independently scheduled resource in the system corresponds to a directory under `/reserv`: `/reserv/cpu` (CPU), `/reserv/mem` (physical memory), `/reserv/fxp0` (network interface 0), `/reserv/sd0` (disk 0), and so on, as shown in Figure 1. Devices with multiple independently scheduled resources correspond to multiple directories, whereas multiple jointly scheduled resources (e.g., mirrored disks) correspond to a single directory.

A resource reservation r is called an *internal reservation* if it can have children, or a *queue* otherwise. r's parent p is always either `/reserv` or another reservation for the same resource. Each resource reservation r contains a `share` file that specifies two values: m_r, the minimum absolute value of the resources that r obtains from p, and ϕ_r, the weight with which r shares p's resources. m_r is specified in units appropriate to the respective resource (e.g., SPECint95 for CPU, bytes for physical memory, or Kbps for disk or network interfaces). If p is `/reserv`, $m_r = V$, the entirety of the resource, and ϕ_r is 100%. The amount of resources apportioned to a reservation r, v_r, depends dynamically on what reservations actually are being used. Every request arriving at a scheduler must specify a queue for processing that request; the request is said to *use* that queue. Schedulers enqueue and service in FIFO order requests that use the same queue. A reservation r is said to be *busy* while there is at least one request that uses r or a descendent of r.

If a resource reservation r is internal, then it also contains the files `newreserv` and `newqueue`. By opening either of these files, an application creates an internal reservation or queue that is r's child, respectively. The open call returns the file descriptor of the newly created `share` file, initialized with $m_r = 0$ and $\phi_r = 0$. Internal reservations thus created are consecutively numbered `r0`, `r1`, and so on, whereas queues are numbered `q0`, `q1`, and so on.

If resource reservation r is a queue, then it also contains the file `backlog`. Writing into `backlog` clears the number of requests served and amount of service provided and sets the maximum number of requests and amount of service that may concurrently be waiting in the queue. Reading from `backlog` returns the number of requests served and the amount of service provided (in units appropriate to the respective resource, e.g. CPU time or bytes).

Eclipse/BSD prevents reservations from exceeding resources as follows. Let S_p be the set of p's children and $M_{S_p} = \sum_{i \in S_p} m_i$. Then writing into the `share` file of $r \in S_p$ is subject to the following admission control rule: the call fails if p is `/reserv` (i.e., the entirety of the resource has a fixed value), $m_p < M_{S_p}$ (i.e., a parent's minimum resources must at least equal the sum of its children's minima after the attempted write), or $\phi_r < 0$ (i.e., weights must be non-negative).

Eclipse/BSD shares resources fairly according to the weights of the busy reservations. If reservation r is not busy, then its apportionment is $v_r = 0$. Otherwise, let p be the parent of r, B_p be the set of p's busy children, and $\Phi_{B_p} = \sum_{i \in B_p} \phi_i$. If p is `/reserv`, then:

$$v_r = V \tag{1}$$

where V is the entirety of the resource, else:

$$v_r = \frac{\phi_r}{\Phi_{B_p}} v_p \tag{2}$$

2.2 Reservation domains and root reservations

This subsection defines what resource reservations each process is allowed to create or use.

In Eclipse/BSD, a process P's *reservation domain* is the list of P's internal *root reservations*, one for for each resource[1]. Queue `q0` of process P's root reservation r is called P's *default queue* for the respective resource. A process P can list any directory under `/reserv` and open and read any `share` or `backlog` file, but can write on `share` or `backlog` files or open `newreserv` or `newqueue` files (i.e., create children) only in reservations that are equal to or descend from one of P's root reservations.

The reservation domain of a process `pid` is represented by a new read-only file, `/proc/pid/rdom`, added to FreeBSD's `proc` file system (where `rdom` stands for "reservation domain"). For example, `/proc/103/rdom` could contain:

```
/reserv/cpu/r2 /reserv/mem/r1
/reserv/fxp0/r0 /reserv/sd0/r3
```

meaning that process 103 has root CPU reservation r2, root memory reservation r1, root network reservation r0, and root disk reservation r3.

[1] Note that our current concept of reservation domain is somewhat different from that in our previous work [6].

If process 104 is in the same reservation domain, `/proc/104/rdom` would have the same contents. The reservation domain of the current process is also named `/proc/curproc/rdom`.

The reservation domain of processes spawned by a process `pid` is given by the new file `/proc/pid/crdom` (where `crdom` stands for "child reservation domain"). When a child is forked, its `rdom` and `crdom` files are initialized to the contents of the parent's `crdom` file. File `/proc/pid/crdom` is writable by any process with the same effective user id as that of process pid, or by the superuser. Writing into `crdom` files is checked for consistency and may fail: For each root reservation r in `/proc/pid/rdom`, `/proc/pid/crdom` must contain an internal reservation r' that is equal to or descends from r.

2.3 Request tagging

In Eclipse/BSD, every request arriving at a scheduler must be tagged with the queue used for that request, as explained in this section.

Resource reservations often cannot simply be associated with shared objects because different clients' requests may specify the same object but different queues. For example, two processes may be in different reservation domains and each need to use a different disk queue to access a shared file, or a different network output link queue to send packets over a shared socket. It would be difficult to compound reservations used on the same object correctly if reservations were associated with the object, since then one client could benefit from another client's reservations.

Therefore, Eclipse/BSD queues are associated with *references* to shared objects, rather than the shared objects themselves (e.g., process, memory object, vnode, or socket). This is accomplished by modifying FreeBSD data structures as follows:

- The CPU scheduler manages *activations* instead of processes. An activation points to a process *and* to the CPU queue in which that process should run.
- The *memory region* structure points to the region's memory object *and* memory queue.
- The *file descriptor* structure points to the file (and thereby to the vnode or socket) *and* to the device queue used for I/O on that file descriptor.

CPU, memory, and device queue pointers are always initialized to the process's default queue for the respective resource. Queue pointers can subsequently be modified only to descendents of the process's root reservation for the respective resource. Initialization and modification of queue pointers occur as follows:

- The initial activation created when a process P is spawned has CPU queue pointer according to the `crdom` file of P's parent. P can subsequently create children of its CPU root reservation, e.g. to process each client's requests. P can switch directly from one CPU queue to another by using a new system call, `activation_switch`. Alternatively, P can spawn new processes that run on CPU queues according to P's `crdom` file.
- The memory queue pointer of a region R is initialized when R is allocated, and can subsequently be modified using a new system call, `mreserv`, with region address, length, and name of the new memory queue as arguments.
- The device queue pointer of a file descriptor fd is initialized: for vnodes, at `open` time; for connected sockets, at `connect` or `accept` time; for unconnected sockets, at `sendto` or `sendmsg` time if fd's device queue pointer has not yet been initialized. A new command to the `fcntl` system call, F_QUEUE_GET, returns the name of the queue to which fd currently points. The queue pointer can subsequently be modified using the new command F_QUEUE_SET to the `fcntl` system call, with the name of the new device queue as argument.

Additionally, I/O request data structures (including `uio` for all I/O, `mbuf` for all network output, and `buf` for disk input that misses in the buffer cache and for all disk output) gain a pointer to the queue they use. Eclipse/BSD copies a file descriptor's queue pointer to the I/O requests generated using that file descriptor.

2.4 Reservation garbage collection

The previous subsections described how resource reservations are created and used; this subsection explains how they are destroyed.

Each resource reservation has a reference count equal to the number of times the reservation appears in an `rdom` or `crdom` file or is pointed by an activation, memory region, or file descriptor. A process's `rdom` and `crdom` files are created when the process is forked and are destroyed when the process exits. The file descriptor of a `share` file in the `/reserv` file system points to the respective resource reservation; additionally, as described in the previous subsection, file descriptors for vnodes and sockets also point to the resource reservations they use. Eclipse/BSD updates reservation reference counts on process `fork` and `exit`, `activation_switch`, memory region allocation and deallocation, `mreserv`, file `open` or `close`, socket `connect` or `accept`, `sendto`, `sendmsg`, and `fcntl` F_QUEUE_SET.

A `GC` flag determines whether a resource reservation should be garbage-collected when the number of references to the reservation drops to zero. When a resource reservation is created, its `GC` flag is enabled, but a privileged process can disable it. New commands to the `fcntl` system call, F_COLLECT_SET and F_COLLECT_GET, can be used on the file descriptor of a reservation's `share` file to set or get the reservation's `GC` flag.

Garbage collection of a queue q may need to be deferred. If q is being used by at least one request, q cannot be removed immediately; instead, q's REMOVE_WHEN_EMPTY flag is set. When the last request that uses q completes and q's REMOVE_WHEN_EMPTY flag is set, if q's reference count is still zero, the scheduler garbage-collects q, else the scheduler resets the flag.

3 Schedulers

The `/reserv` API described in the previous section provides an interface to proportional share schedulers. Eclipse/BSD incorporates a proportional share scheduler for each resource, as discussed in this section.

3.1 MTR-LS

Eclipse/BSD's CPU scheduler uses the MTR-LS (Move-To-Rear List Scheduling) algorithm [6]. When a process blocks (e.g., waiting for I/O), MTR-LS keeps the unused portion of the process's quota in the same position in the scheduling list, unlike the Weighted Round Robin (WRR) algorithm, which removes the process from the runnable list and, when the process becomes runnable again, places it back at the tail of the list. Consequently, MTR-LS may delay I/O-bound processes much less than does WRR. MTR-LS may also provide greater throughput than does WRR, whose scheduling delays may prevent I/O-bound processes from from fully utilizing their CPU reservations.

MTR-LS was specifically designed for CPU scheduling, where the time necessary to process a request cannot be predicted. To the best of our knowledge, MTR-LS is the only algorithm that provides the optimal cumulative service guarantee [6] when the durations of service requests are unknown *a priori*. However, MTR-LS assumes that requests can be preempted either at any instant or at fixed intervals. This is true of CPU scheduling, but usually is not true of disk or network scheduling, where requests cannot be preempted after they start and may take varying time to complete. Therefore, Eclipse/BSD uses other algorithms for I/O scheduling.

3.2 YFQ

Eclipse/BSD's I/O schedulers use approximations to the GPS (Generalized Processor Sharing) [18] model. GPS assumes an ideal "fluid" system where each backlogged "flow" in the system instantaneously receives service in proportion to the flow's share and inversely proportionally to the sum of the shares of all backlogged flows (where a backlogged flow is analogous to a busy queue). GPS cannot be precisely implemented for I/O because typically (1) I/O servers can only service one request at a time and (2) an I/O request cannot be preempted once service on it begins. GPS approximations estimate the time necessary for servicing each request and interleave requests from different queues so that each queue receives service proportionally to its share (although not instantaneously). However, the necessary time estimates may be difficult to compute precisely because GPS's rate of service for each flow

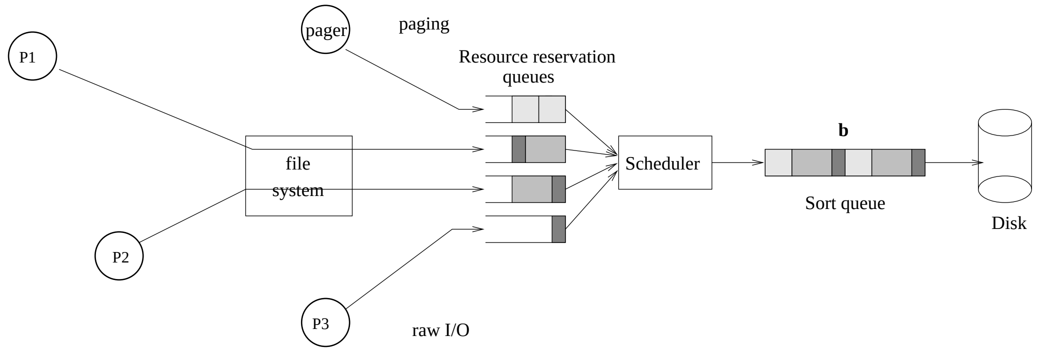

Figure 2: The sort queue allows the disk driver or disk to reorder requests and minimize disk latency and seek overheads.

depends on what flows are backlogged at each instant [3].

Eclipse/BSD's disk scheduler uses a new GPS approximation, the YFQ (Yet another Fair Queueing) algorithm [5], which can be implemented very efficiently. A resource is called *busy* if it has at least one busy queue, or *idle* otherwise. YFQ associates a *start tag*, S_i, and a *finish tag*, F_i, with each queue q_i. S_i and F_i are initially zero. YFQ defines a *virtual work* function, $v(t)$, such that: (1) $v(0) = 0$; (2) While the resource is busy, $v(t)$ is the minimum of the start tags of its busy queues at time t; and (3) When the resource becomes idle, $v(t)$ is set to the maximum of all finish tags of the resource.

When a new request r_i that uses queue q_i arrives: (1) If q_i was previously empty, YFQ makes $S_i = \max(v(t), F_i)$ followed by $F_i = S_i + \frac{l_i}{w_i}$, where l_i is the data length of request r_i; and (2) YFQ appends r_i to q_i.

YFQ selects for servicing the request r_i at the head of the busy queue q_i with the smallest finish tag F_i. r_i remains at the head of q_i while r_i is being serviced. When r_i completes, YFQ dequeues it; if queue q_i is still non-empty, YFQ makes $S_i = F_i$ followed by $F_i = S_i + \frac{l'_i}{w_i}$, where l'_i is the data length of the request r'_i now at the head of q_i.

Selecting one request at a time, as described above, allows YFQ to approximate GPS quite well, providing good cumulative service, delay, and fairness guarantees. However, such guarantees may come at the cost of excessive disk latency and seek overheads, harming aggregate disk throughput. Therefore, YFQ can be configured to select up to b requests (a *batch*) at a time and place them in a *sort queue*, as shown in Figure 2. The disk driver or the disk itself may reorder requests within a batch so as to minimize disk latency and seek overheads.

3.3 WF^{2}Q

Eclipse/BSD's network output link scheduler uses the hierarchical WF^{2}Q (Worst-case Fair Weighted Fair Queueing) algorithm [3]. This algorithm is similar to an earlier GPS approximation, WFQ (Weighted Fair Queueing) [9]. However, unlike WFQ, WF^{2}Q does not schedule a packet until it is *eligible*, i.e., its transmission would have started under GPS. Consequently, WF^{2}Q has optimal worst-case fair index bound, making it a good choice for a hierarchical scheduler [3].

Note that neither YFQ nor WF^{2}Q could be used for CPU scheduling, since they assume that the time necessary to process a request can be estimated and they never preempt a request.

3.4 SRP

Eclipse/BSD employs SRP (Signaled Receiver Processing) [7] for network input processing. SRP demultiplexes incoming packets *before* network and higher-level protocol processing. Unlike FreeBSD's single IP input queue and input protocol processing at software interrupt level, SRP uses an unprocessed input queue (UIQ) per socket and processes input protocols in the context of the respective applications. If a socket's queue is full, SRP drops new packets for that socket immediately, unlike FreeBSD, which wastefully processes packets that will need to be dropped. Because SRP processes protocols in the context of the respective receiving applications, SRP can avoid *receive livelock* [16], a network input overload condition that prevents any packets from being processed by an application.

When SRP enqueues a packet into a socket's UIQ, SRP signals SIGUIQ to the applications that own that socket. The default action for SIGUIQ is to perform input protocol processing (asynchronously to the applications). However, applications can synchronize such processing by catching SIGUIQ and deferring protocol processing until a later input call (e.g., `recv`). Synchronous protocol processing may improve cache locality. Unlike LRP (Lazy Receive Processing) [10], SRP does not use separate kernel threads for asynchronous protocol processing (kernel threads are not available in FreeBSD).

4 Implementation

This brief section shows that Eclipse implementation does not require too many changes to the underlying time-sharing system.

Our current Eclipse/BSD implementation adds approximately 6500 lines of code to FreeBSD version 2.2.8: 2400 lines for the `reserv` file system and modifications to the `proc` file system, and 4100 lines for the new schedulers and their integration into the kernel. The kernel size in the GENERIC configuration is 1601351 bytes for FreeBSD and 1639297 bytes for Eclipse/BSD (an increase of only 38 KB).

5 Experimental results

This section demonstrates experimentally that applications can use Eclipse/BSD's `/reserv` API and CPU, disk, and network schedulers so as to obtain minimum performance guarantees, regardless of other load on the system.

We ran experiments on the configuration shown in Figure 3, where HTTP clients on nodes A to E make requests to the HTTP server on node S. Nodes A to C are Pentium Pro PC's running FreeBSD. Nodes D and E are Sun workstations running Solaris. The operating system varies only in node S, being either

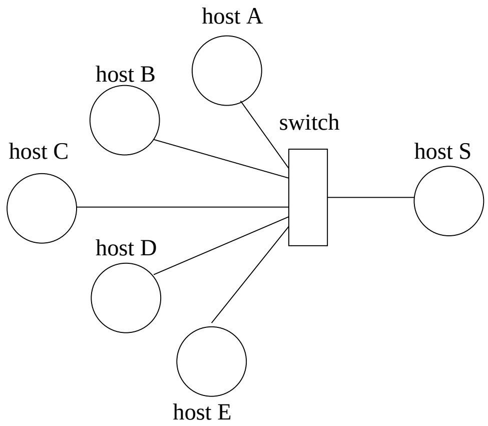

Figure 3: Node S is a Web server that hosts multiple sites on either FreeBSD or Eclipse/BSD.

FreeBSD or Eclipse/BSD. Node S is a PC with 266 MHz Pentium Pro CPU, 64 MB RAM, and 9 GB Seagate ST39173W fast wide SCSI disk. All nodes are connected by a Lucent P550 Cajun Ethernet switch (unless otherwise noted, at 10 Mbps). Node S runs the Apache 1.3.3 HTTP server and hosts multiple Web sites. Nodes A to E run client applications (some derived from the WebStone benchmark) that make requests to the server. At most ten clients run at each of the nodes A to E. Unless otherwise noted, all measurements are the averages of three runs.

Each experiment overloaded one of the server's resources, as described in the following subsections.

5.1 CPU scheduling

In the first experiment, an increasing number of clients continuously made CGI requests to either of two Web sites hosted at node S. Processing of each of these CGI requests consists of computing half a million random numbers (using `rand()`) and returning a 1 KB reply. Therefore, the bottleneck resource is the CPU. We measured the average throughput and response time (over three minutes) under the following scenarios: (1) The site of interest reserves 50% of the CPU and the competing site reserves 49% of the CPU; (2) The site of interest reserves 99% of the CPU; and (3) Both sites run in the same CPU reservation and reserve 99% of the CPU. Figure 4 shows the throughput of the site of interest when the latter has ten clients and the competing site has a varying number of clients, and Figure 5 shows the corresponding response times. Perfor-

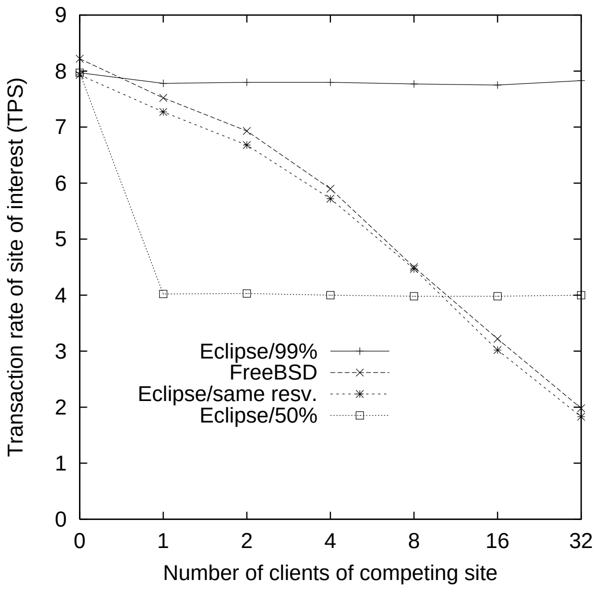

Figure 4: Appropriate CPU reservations can guarantee a minimum throughput for the site of interest.

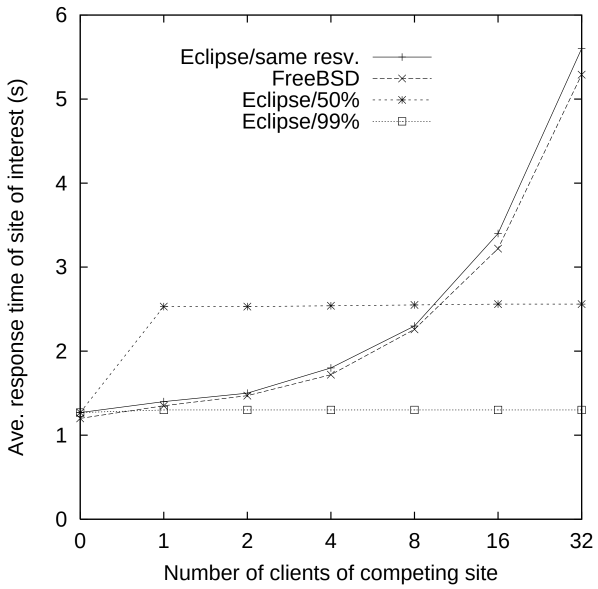

Figure 5: Appropriate CPU reservations can guarantee a maximum response time for the site of interest.

mance when both sites run in the same CPU reservation on Eclipse/BSD is roughly the same as performance on FreeBSD. When the site of interest reserves 99% of the CPU, its performance is essentially unaffected by other load. When the site of interest reserves 50% of the CPU, it still gets essentially all of the CPU if there is no other load, but, as would be expected, the throughput goes down by half and the response time doubles when there is other load. However, throughput and response time of the site of interest remain constant when further load is added, while on FreeBSD throughput decreases and response time increases without bound. This shows that FreeBSD and Eclipse/BSD are equally good if there is excess CPU capacity, but Eclipse/BSD can also guarantee a certain minimum CPU allocation (and consequently minimum throughput and maximum response time).

5.2 Disk scheduling

Again in the second experiment, an increasing number of clients continuously made CGI requests to either of two Web sites hosted at node S. However, these requests are I/O-intensive, consisting of reading a 100 MB file and returning a 10 KB reply. Because requests and replies are small and each request involves considerable disk I/O but little processing, the bottleneck resource is the disk. We reserved 50% of S's disk bandwidth to the Web site of interest and measured the latter's average throughput over three minutes. YFQ's sort queue was configured with a batch size of 4 requests. During the measurements, the site of interest had ten clients and the competing site had a varying number of clients. Figure 6 shows that in the absence of other load, Eclipse/BSD gives to the site of interest essentially all of the bottleneck resource, even though the site has only 50% reserved. When the load on the competing site increases, the throughput of the site of interest decreases. However, on Eclipse/BSD, the throughput bottoms out at roughly the reserved amount, whereas on FreeBSD the throughput decreases without bound. This shows that FreeBSD and Eclipse/BSD are equally good when there is excess disk bandwidth, but when bandwidth is scarce, Eclipse/BSD is also able to guarantee a minimum disk bandwidth allocation.

5.3 Output link scheduling

In the third experiment, an increasing number of clients continuously requested the same 1.5 MB document from either of two Web sites hosted at node S. Given that requests are much smaller than replies, little processing is required per request, and the re-

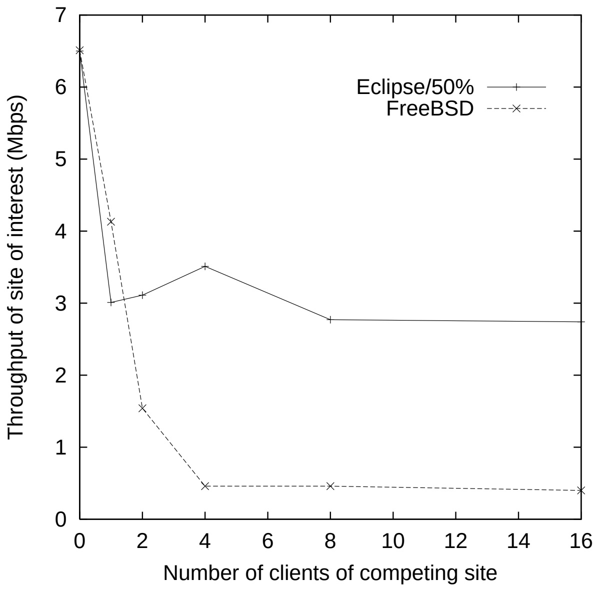

Figure 6: The site of interest gets at least its reserved fraction (50%) of the disk bandwidth.

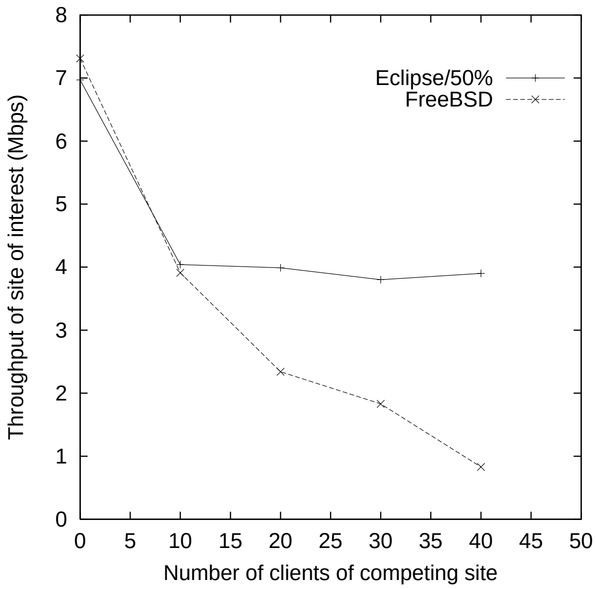

Figure 7: The site of interest gets at least its reserved fraction (50%) of the output link bandwidth.

quested document fits easily in the node S's buffer cache, the bottleneck resource is S's network output link. We reserved 50% of S's output link bandwidth to the Web site of interest and measured the latter's average throughput over three minutes. During the measurements, the site of interest had ten clients and the competing site had a varying number of clients. Figure 7 shows the results, which are very similar to those of Figure 6, where the disk is the bottleneck. FreeBSD and Eclipse/BSD are equally good when there is excess output link bandwidth, but when bandwidth is scarce, Eclipse/BSD is also able to guarantee a minimum output link bandwidth allocation.

5.4 Input link scheduling

The final set of experiments addresses network reception overload. In these experiments, the network operated at 100 Mbps full-duplex, and measurements are the averages of five runs.

In the fourth experiment, a client application sent 10-byte UDP packets at a fixed rate to a server application running at node S. Both on FreeBSD and on Eclipse/BSD, the server application received essentially all of the packets when the transmission rate was up to about 5600 packets per second (pkts/s). Above that transmission rate, as shown on Figure 8, the reception rate on Eclipse/BSD reached a plateau at around 5700 pkts/s. On FreeBSD, on the contrary, the reception rate dropped precipitously. This experiment shows that on Eclipse/BSD applications can make forward progress even when there is network reception overload, while on FreeBSD applications can enter livelock [16] in such situations. Eclipse/BSD prevents receive livelock because of SRP.

However, SRP cannot by itself guarantee that *important* applications will make forward progress. Eclipse/BSD can guarantee that by combining SRP and CPU reservations. In the fifth and final experiment, four different client applications sent 10-byte UDP packets at the same fixed rate to a different server application running on node S. We measured reception rates in two scenarios: (1) All four server applications reserved each 25% of the CPU; and (2) One server application reserved 97% of the CPU and the remaining server applications reserved 1% each. While the transmission rate was below 5600 pkts/s, essentially all packets were received. Reception rates increased slightly to 5900 pkts/s for a transmission rate of 28.5 Kpkts/s. Above that rate, results differ for the two scenarios, as shown in Figure 9. In the first scenario, reception rate goes down to about 1200 pkts/s. In the second scenario, the reception rate of the application with 97% of the CPU goes

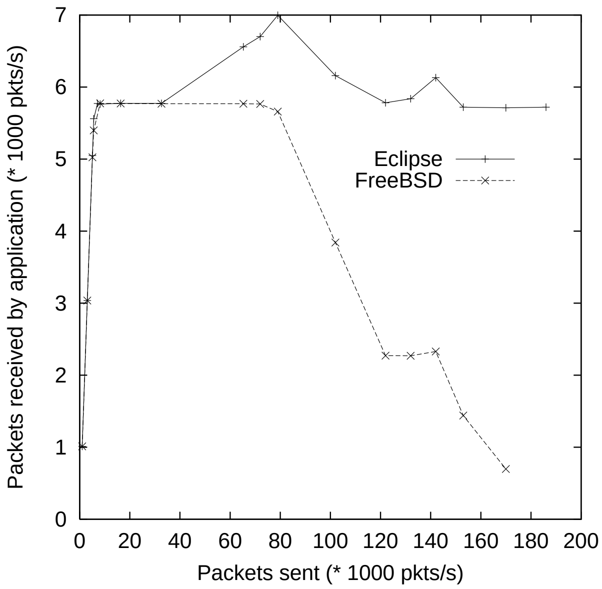

Figure 8: Eclipse/BSD avoids receive livelock.

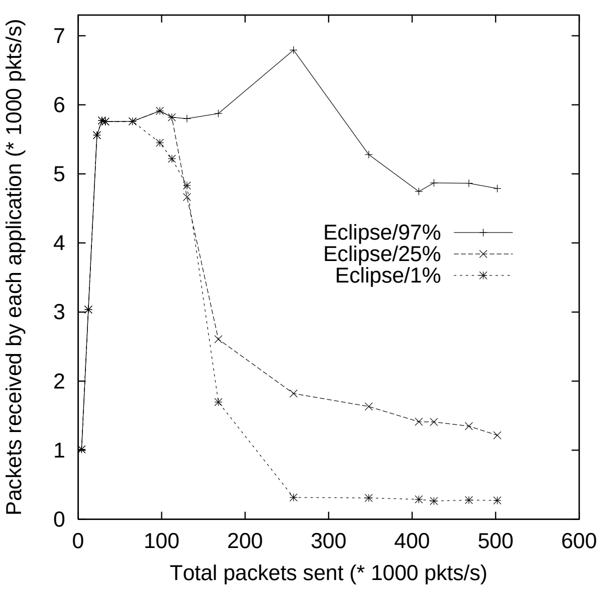

Figure 9: Eclipse/BSD guarantees forward progress according to CPU reservation.

down to about 4800 pkts/s, while the reception rate of the applications with 1% of the CPU goes down to about 160 pkts/s.

6 Related and future work

There are numerous recent works on proportional share scheduling [11, 6, 2, 3, 12, 23]. This paper complements those works by providing a uniform API for their schedulers and considering practical aspects of retrofitting them into mainstream operating systems. The API proposed here is also set apart by promoting uniformity not only across scheduling algorithms, but also across different resources.

The `/reserv` file system resembles many Plan 9 [19] APIs, which also use special file systems. We used Plan 9 in our previous MTR-LS work [6] but decided to replace it by FreeBSD because of FreeBSD's greater popularity and support for more current hardware.

Stride scheduling and the associated *currency* abstraction [24] can be used to group and isolate users, processes, or threads, much like the resource reservations discussed here. Another alternative is *resource containers* [1], which can isolate resources used by each client, whether within a single process or across multiple processes. Resource containers have been demonstrated primarily for priority-based CPU scheduling, not for hierarchical proportional sharing of different resources, as we advocate here. Our solutions for retrofitting reservations into a time-sharing system (e.g., how to associate reservations with references to shared objects, tag requests, and garbage collect reservations) may be useful also in conjunction with those frameworks.

Nemesis [14] is an operating system that uses a radical new architecture in order to eliminate QoS *crosstalk*, i.e., the degradation of one application's performance due to the load on another application. The Nemesis kernel provides only scheduling, and most other operating system services are implemented as libraries that are linked with applications and run in each application's address space. Eclipse/BSD attempts to provide similar isolation in a conventional monolithic architecture, requiring comparatively much less implementation effort.

SMART [17] is a hierarchical CPU scheduling algorithm that supports both hard real-time and conventional time-sharing applications, adjusts well to overload, and can notify applications when their deadlines cannot be met. Rialto [13] combines CPU reservations and time constraints into a scheduling graph that is used by a run-time scheduler to provide strong CPU guarantees. While SMART and Rialto target especially hard real-time CPU scheduling, the work presented here addresses mostly soft real-time scheduling of different resources and the integration of such scheduling into conventional systems.

7 Conclusions

We described how Eclipse/BSD applications can obtain resource reservations and thereby guarantee a desired quality of service for themselves or for their clients. Eclipse/BSD's API, `/reserv`, provides a simple, uniform interface to hierarchical proportional sharing of system resources. We discussed the different schedulers used in Eclipse/BSD and demonstrated experimentally that they can isolate the performance of selected applications from CPU, disk, or network overloads caused by other applications. Eclipse/BSD was implemented by making straightforward modifications to FreeBSD and greatly improves the system's ability to provide QoS guarantees, fairness, and hierarchical resource management. We believe that other common time-sharing systems would benefit from similar modifications.

Acknowledgments

We thank Mary McShea, Amit Singh, and Josep Blanquer for their help in the implementation and experiments, and the anonymous referees for their valuable comments.

References

[1] G. Banga, P. Druschel and J. Mogul. "Resource Containers: A New Facility for Resource Management in Server Systems", in *Proc. OSDI'99*, USENIX, Feb. 1999.

[2] J. Bennet and H. Zhang. "WF^{2}Q: Worst-Case Fair Weighted Fair Queueing", in *Proc. INFOCOM'96*, IEEE, Mar. 1996, pp. 120-128.

[3] J. Bennet and H. Zhang. "Hierarchical Packet Fair Queueing Algorithms", in *Proc. SIGCOMM'96*, ACM, Aug. 1996.

[4] P. Barham. "A Fresh Approach to File System Quality of Service", in *Proc. NOSSDAV'97*, IEEE, May 1997, pp. 119-128.

[5] J. Bruno, J. Brustoloni, E. Gabber, B. Özden and A. Silberschatz. "Disk Scheduling with Quality of Service Guarantees", to appear in *Proc. ICMCS'99*, IEEE, June 1999.

[6] J. Bruno, E. Gabber, B. Özden and A. Silberschatz. "The Eclipse Operating System: Providing Quality of Service via Reservation Domains", in *Proc. Annual Tech. Conf.*, USENIX, June 1998, pp. 235-246.

[7] J. Brustoloni, E. Gabber and A. Silberschatz. "Signaled Receiver Processing", submitted for publication.

[8] H. Custer. "Inside Windows NT", Microsoft Press, 1993.

[9] A. Demers, S. Keshav and S. Shenker. "Design and Analysis of a Fair Queueing Algorithm", in *Proc. SIGCOMM'89*, ACM, Sept. 1989, pp. 1-12.

[10] P. Druschel and G. Banga. "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems", in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 261-275.

[11] P. Goyal, X. Guo and H. Vin. "A Hierarchical CPU Scheduler for Multimedia Operating Systems", in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 107-121.

[12] P. Goyal, H. Vin and H. Chen. "Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks", in *Proc. SIGCOMM'96*, ACM, Aug. 1996.

[13] M. Jones, D. Ro su and M. Ro su. "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities", in *Proc. SOSP'97*, ACM, Oct. 1997, pp. 198-211.

[14] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns and E. Hyden. "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", in *JSAC*, 14(7), IEEE, Sept. 1996, pp. 1280-1297.

[15] M. McKusick, K. Bostic, M. Karels and J. Quarterman. "The Design and Implementation of the 4.4 BSD Operating System", Addison-Wesley Pub. Co., Reading, MA, 1996.

[16] J. Mogul and K. K. Ramakrishnan. "Eliminating Receive Livelock in an Interrupt-Driven Kernel", in *Proc. Annual Tech. Conf.*, USENIX, 1996, pp. 99-111.

[17] J. Nieh and M. Lam. "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications", in *Proc. SOSP'97*, ACM, Oct. 1997, pp. 184-197.

[18] A. Parekh and R. Gallager. "A Generalized Processor Sharing Approach to Flow Control — The Single Node Case", in *Trans. Networking*, ACM/IEEE, 1(3):344-357, June 1993.

[19] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey and P. Winterbottom. "Plan 9 from Bell Labs", in *Computing Systems*, USENIX, 8(3):221-254, Summer 1995.

[20] P. Shenoy and H. Vin. "Cello: A Disk Scheduling Framework for Next Generation Operating Systems", in *Proc. SIGMETRICS'98*, ACM, June 1998.

[21] P. Shenoy. P. Goyal, S. Rao and H. Vin. "Design and Implementation of Symphony: An Integrated Multimedia File System", in *Proc. Multimedia Computing and Networking*, SPIE, Jan. 1998.

[22] D. Stiliadis and A. Varma. "Frame-Based Fair Queueing: A New Traffic Scheduling Algorithm for Packet-Switched Networks", Tech. Rep. UCSC-CRL-95-39, Univ. Calif. Santa Cruz, July 1995.

[23] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke and C. G. Plaxton. "A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems", in *Proc. Real Time Systems Symp.*, IEEE, Dec. 1996.

[24] C. Waldspurger and W. Weihl. "An Object-Oriented Framework for Modular Resource Management', in *Proc. IWOOOS '96*, IEEE, Oct. 1996, pp. 138-143.