Reprinted from the

Proceedings of the
Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Review Committee

Proceedings Formatting Team

Class-based Prioritized Resource Control in Linux

Shailabh Nagar, Hubertus Franke, Jonghyuk Choi, Chandra Seetharaman
Scott Kaplan,[*] *Nivedita Singhvi, Vivek Kashyap, Mike Kravetz*
IBM Corp.
`{nagar,frankeh,jongchoi,chandra.sekharan}@us.ibm.com`
`{nivedita,vivk,kravetz}@us.ibm.com`
`sfkaplan@cs.amherst.edu`

Abstract

In Linux, control of key resources such as memory, CPU time, disk I/O bandwidth and network bandwidth is strongly tied to kernel tasks and address spaces. The kernel offers very limited support for enforcing user-specified priorities during the allocation of these resources.

In this paper, we argue that Linux kernel resource management should be based on classes rather than tasks alone and be guided by class shares rather than system utilization alone. Such class-based kernel resource management (CKRM) allows system administrators to provide differentiated service at a user or job level and prevent denial of service attacks. It also enables accurate metering of resource consumption in user and kernel mode. The paper proposes a framework for CKRM and discusses incremental modifications to kernel schedulers to implement the framework.

1 Introduction

With Linux having made rapid advances in scalability, making it the operating system of choice for enterprise servers, it is useful and timely to examine its support for resource management. Enterprise workloads typically run on two types of servers: clusters of 1-4 way SMPs and large (8-way and upward) SMPs and mainframes. In both cases, system administrators must balance the needs of the workload with the often conflicting goal of maintaining high system utilization. The balancing becomes particularly important for large SMPs as they often run multiple workloads to amortize the higher cost of the hardware.

A key aspect of multiple workloads is that they vary in the *business importance* to the server owner. To maximize the server's utility, the system administrator needs to ensure that workloads with higher business importance get a larger share of server resources. The kernel's resource schedulers need to allow some form of differentiated service to meet this goal. It is also important that the resource usage by different workloads be accounted accurately so that the customers can be billed according to their true usage rather than an average cost. Kernel support for accurate accounting of resource usage is required, especially for resource consumption in kernel mode.

Differentiated service is also useful to the desktop user. It would allow large file transfers to get reduced priority to the disk compared to disk accesses resulting from interactive com-

[*] on sabbatical from Amherst College, MA

mands. A kernel compile could be configured to run in the background with respect to the CPU, memory and disk, allowing a more important activity such as browsing to continue unimpeded.

The current Linux kernel (version 2.5.69 at the time of writing) lacks support for the abovementioned needs. There is limited and varying support for any kind of performance isolation in each of the major resource schedulers (CPU, network, disk I/O and memory). CPU and inbound network scheduling offer the greatest support by allowing specification of priorities. The deadline I/O scheduler [3] offers some isolation between disk reads and writes but not between users or applications and the VM subsystem has support for limiting address space size of a user. More importantly, the granularity of kernel supported service differentiation is a task (process), or infrequently the userid. It does not allow the user to define the granularity at which resources get apportioned. Finally, there is no common framework for a system administrator to specify priorities for usage of different physical resources.

The work described in this paper addresses these shortcomings. It proposes a class-based framework for prioritized resource management of all the major physical resources managed by the kernel. A class is a user-defined, dynamic grouping of tasks that have associated priorities for each physical resource. The proposed framework permits a better separation of user-specified policies from the enforcement mechanisms implemented by the kernel. Most importantly, it attempts to achieve these goals using incremental modifications of the existing mechanisms.

The paper is organized as follows. Section 2 introduces the proposed framework and the central concepts of classification, monitoring and control. Sections 3,4,5,6 explore the framework for the CPU, disk I/O, network and memory subsystems and propose the extensions necessary to implement it. Section 7 concludes with directions for future work.

2 Framework

Before describing the proposed framework, we define a few terms.

Tasks are the Linux kernel's common representation for both processes and threads. A *class* is a group of tasks. The grouping of tasks into classes is decided by the user using rules and policies.

A *classification rule*, henceforth simply called a rule, is a method by which a task can be classified into a class. Rules are defined by the system administrator, generally as part of a policy (defined later) but also individually, typically as modifications or increments to an existing policy. Attributes of a task, such as real uid, real gid, effective uid, effective gid, path name, command name and task or application tag (defined later) can be used to define rules. A rule consists of two parts: a set of attribute-value tuples (A,V) and a class C. If the rule's tuple values match the corresponding attributes of a task , then it is considered to belong to the class C1.

A *policy* is a collection of class definitions and classification rules. Only one policy is active in a system at any point of time. Policies are constructed by the system administrator and submitted to a CKRM kernel. The kernel optionally verifies the policy for consistency and activates it. The order of rules in a policy is important. Rules are applied in the order they are defined (one exception is the application tags as described in the notes below).

An *Application/Task Tag* is a user-defined attribute associated with a task. Such an attribute is useful when tasks need to be classified and managed based on application-specific criteria. In such scenarios, an applications tasks can specify its tag to the kernel using a system call, ioctl, /proc entry etc. and trigger its classification using a rule that uses the task tag attribute. Since the task tag is opaque to the kernel, it allows applications and system administrators additional flexibility in grouping tasks.

A *Resource Manager* is the entity which determines the proportions in which resources should be allocated to classes. This could be either a human system administrator or a resource management application middleware.

With all the new terms of the framework defined, we can now describe how the framework operates to provide class-based resource management. Figure 1 illustrates the lifecycle of tasks in the proposed framework with an emphasis on the three central aspects of classification, monitoring and control.

2.1 Initialization

Sometime after system boot up, the Resource Manager commits a policy to the kernel. The policy defines the classes and it is used to classify all tasks (pre-existing and new) created and all incoming network packets. A CKRM-enabled Linux kernel also contains a default policy with a single default class to which all tasks belong. The default policy determines classification and control behaviour until the Resource Manager's policy gets loaded. New policies can be loaded at any point and override the existing policy. Such policy loads trigger reclassification and reinitialization of monitoring data and are expected to be very infrequent.

2.2 Classification

Classification refers to the association of tasks to classes and the association of resource requests to a class. The distinction is mostly irrelevant as most resource requests are initiated by a task except for incoming network packets which need to be classified before it is known which task will consume them. Classification is a continuous operation. It happens on a large scale each time a new policy is committed and all existing tasks get reclassified. At later points, classification occurs whenever (1) a new task is created e.g. fork(), exec(); (2) the attributes of a task change e.g setuid(), setgid(), application tag change (initiated by the application) and (3) explicit reclassification of a specific task by the Resource Manager. Scenarios (1) and (2) are illustrated in Figure 1.

Classification of tasks potentially allows all work initiated by the task to be associated with the class. Thus the CPU, memory and I/O requests generated by this task, or by the kernel on behalf of this task, can be monitored and regulated by the class-aware resource schedulers. Kernel-initiated resource requests which are on behalf of multiple classes e.g. a shared memory page writeout need special treatment as do application initiated requests which are processed asynchronously. Classification of incoming network connections and/or data (which are seen by the kernel before the task to which they are destined) is discussed separately in Section 5.

2.3 Monitoring

Resource usage is maintained at the class level in addition to the regular system accounting by task, user etc. The system administrator or an external control program with root privileges can obtain that information from the kernel at any time. The information can be used to assess machine utilization for billing purposes or as an input to a future decision on changing the share allotted to a class. The CKRM API provides functions to query the current usage data as shown in Figure 1.

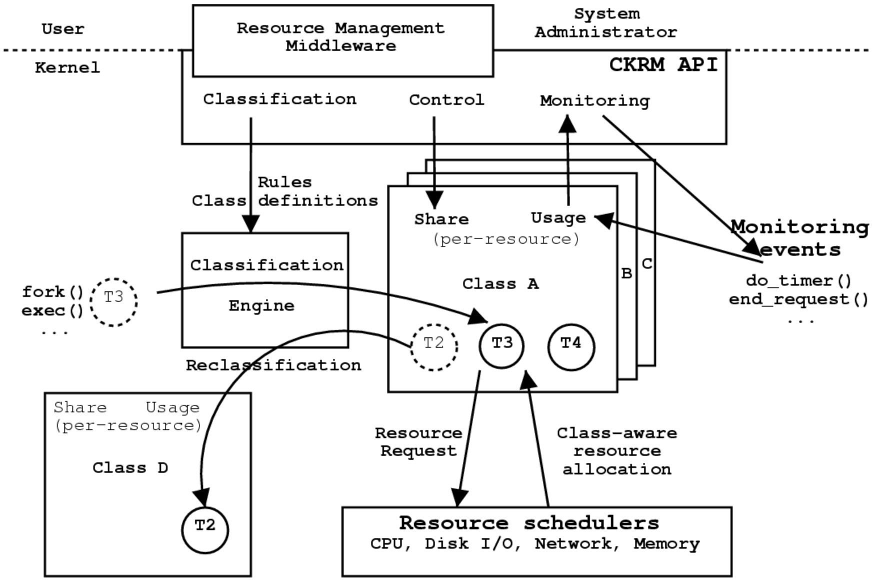

Figure 1: CKRM lifecycle

2.4 Control

The system administrator, as part of the initial policy or at a later point in time, assigns a per-resource share to each class in the system. Each class gets a separate share for CPU time, Memory pages, I/O bandwidth and incoming network I/O bandwidth. The resource schedulers try their best to respect these shares while allocating resources to a class. e.g. the CPU scheduler tries to ensure that tasks belonging to Class A with a 50% CPU share collectively get 50% of the CPU time. At the next level of the control hierarchy, the system administrator or a control program can change the shares assigned to a class based on their assessment of application progress, system utilization etc. Collectively, the setting of shares and share-based resource allocation constitute the control part of the resource management lifecycle and are shown in Figure 1. This paper concentrates on the lower level share-based resource allocation since that is done by the kernel.

The next four sections go into the details of classification, monitoring and control aspects of managing each of the four major physical resources.

3 CPU scheduling

The CPU scheduler is central to the operation of the computing platform. It decides which task to run when and how long. In general realtime and timeshared jobs are distinguished, each with different objectives. Both are realized through different scheduling disciplines implemented by the scheduler. Before addressing the share based scheduling schemes, we describe the current Linux scheduler.

3.1 Linux 2.5 Scheduler

To achieve its objectives, Linux assigns a static priority to each task that can be modified by the user through the `nice()` interface. Linux has a range of $[0 \ldots \texttt{MAX\_PRIO}]$ priority classes, of which the first `MAX_RT_PRIO` (=100) are set aside for realtime jobs and the remaining 40 priority classes are set aside for timesharing (i.e. normal) jobs, representing the $[-20 \ldots 19]$ nice value of UNIX processes. The lower the priority value, the higher the "logical" priority of a task, i.e. its general importance. In this context we always assume the logical priority when we are talking about priority increases and decreases. Realtime tasks always have a higher priority then normal tasks.

The Linux scheduler in 2.5, a.k.a the O(1) scheduler, is a multi queue scheduler that assigns to each cpu a run queue, wherein local scheduling takes place. A per-cpu runqueue consists of two arrays of task lists, the active array and the expired array. Each array index represents a list of runnable task at their respective priority level. After executing for a period task move from the active list to the expired list to guarantee that all tasks get a chance to execute. When the active array is empty, expired and active arrays are swapped. More detail is provided further on. The scheduler simply picks the first task of the highest priority queue of the active queue for execution.

Occasionally a load balancing algorithm rebalances the runqueues to ensure that a similar level of progress is made on each cpu. Realtime issues and load balancing issues are beyond this description here, hence we concentrate on the single cpu issue for now. For more details we refer to [12]. It might also be of interest to abstract this against the Linux 2.4 based scheduler, which is described in [10].

As stated earlier, the scheduler needs to decide which task to run next and for how long. Time quantums in the kernel are defined as multiples of a system *tick*. A tick is defined by the fixed delta (1/HZ) of two consecutive timer interrupts. In Linux 2.5: HZ=1000, i.e. the interrupt routine scheduler_tick() is called once every msec at which time the currently executing task is charged a tick.

Besides the *static priority* (static_prio), each task maintains an *effective priority* (prio). The distinction is made in order to account for certain temporary priority bonuses or penalties based on the recent *sleep average* sleep_avg of a given task. The sleep average, a number in the range of $[0 \ldots \texttt{MAX\_SLEEP\_AVG} * \texttt{HZ}]$, accounts for the number of ticks a task was recently descheduled. The time (in ticks) since a task went to sleep (maintained in sleep_timestamp) is added on task wakeup and for every time tick consumed running, the sleep average is decremented.

The current design provisions a range of PRIO_BONUS_RATIO=25% $[-12.5\%..12.5\%]$ of the priority range for the sleep average. For instance a “nice=0” task has a static priority of 120. With a sleep average of 0 this task is penalized by 5 resulting in an effective priority of 125. On the other hand, if the sleep average is MAX_SLEEP_AVG = 10 secs, a bonus of 5 is granted leading to an effective priority of 115. The effective priority determines the priority list in the active and expired array of the run queue. A task is declared *interactive* when its effective priority exceeds its static priority by a certain level (which can only be due to its accumulating sleep average). High priority tasks reach interactivity with a much smaller sleep average than lower priority tasks.

The timeslice, defined as the maximum time a task can run without yielding to another task, is simply a linear function of the static priority of normal tasks projected into the range of $[\texttt{MIN\_TIMESLICE} \ldots \texttt{MAX\_TIMESLICE}]$. The defaults are set to $[10 \ldots 200]$ msecs. The higher the priority of a task the longer its timeslice. A task’s initial timeslice is deducted from parents’ remaining timeslice. For every timer tick the running task’s timeslice is decremented. If decremented to “0”, the scheduler replenishes the timeslice, recomputes the effective priority and either reenqueues the task into the active array if it is interactive or into the expired array if it is non-interactive. It then picks the next best task from the active array. This guarantees that all others tasks will execute first before any expired task will run again. If a task is descheduled, its timeslice will not be replenished at wakeup time, however its effective priority might have changed due to any sleep time.

If all runnable tasks have expired their timeslices and have been moved to the expired list, the expired array and the active array are swapped. This makes the scheduler O(1) as it does not have to traverse a potentially large list of tasks as was needed in the 2.4 scheduler. Due to interactivity the situation can arise that the active queue continues to have runnable tasks. As a result tasks in the expired queue might get starved. To avoid this, if the longest expired task is older than `STARVATION_LIMIT`=10secs, the arrays are switched again.

3.2 CPU Share Extensions to the Linux Scheduler

We now examine the problem of extending the O(1) scheduler to allocate CPU time to classes in proportion of their CPU shares. Proportional share scheduling of CPUs has been studied in depth [7, 21, 13] but not in the context of making minimal modifications to an existing scheduler such as O(1).

Let C_i, i=$[1 \ldots N]$ be the set of N distinct classes with corresponding cpu shares S_i^{cpu} such that $\sum_{i=1}^{N} S_i^{cpu} = 1$. Let `SHARE_TIME_EPOCH` (STE) be the time interval, measured in ticks, over which the modified scheduler attempts to maintain the proportions of allocated CPU time. Further, we use $\Phi(a)$ and $\Phi(a, b)$ to denote a functions of parameters a and b.

Weighted Fair Share (WFS): In the first approach considered, henceforth called weighted fair share (WFS), a per class scheduling resource container is introduced that accounts for timeslices consumed by the tasks currently assigned to the class. Initially, the timeslice TS_i, i=$[1 \ldots N]$ of each class C_i is determined by $TS_i = S_i^{cpu} \times STE$. The timeslice allocated to a task $ts(p)$ remains the same as in O(1). Everytime a task consumes one of its own timeslice ticks, it also consumes one from the class' timeslice. When a class' ticks are exhausted, the task consuming the last tick is put into the expired array. When the scheduler picks other tasks from the same class to run, they immediately move to the expired array as well. Eventually the expired and active arrays are switched at which time all resource containers are refilled to $TS_i = S_i^{cpu} \times STE$. Since the array switch occurs as soon as the active list becomes empty, this approach is work conserving (the CPU is never idle if there are runnable tasks). A variant of this approach was initially implemented by [17] based on a patch from Rik van Riel for allocating *equal* shares to all *users* in the system.

However, WFS has some problems. If the tasks of a class are CPU bound and $\sum_{p \in C_i} ts(p) > TS_i$ then a class could exhaust its timeslice before all its tasks have had a chance to run atleast once. Therefore the lower priority tasks of the class could perpetually move from the active to expired lists without ever being granted execution time. Starvation occurs because neither the static priority (sp) nor the sleep average (sa) of the tasks is changed at any time. Hence each task's timeslice $ts(p) = \Phi(sp)$ and effective priority $ep(p) = \Phi(sp, sa)$ remain unchanged. Hence the relative priority of tasks of a class never changes (in a CPU bound situation) nor does the amount of CPU time consumed by the higher priority tasks.

To ensure a fair share for individual tasks within classes, we need to ensure that the rate of progress of a task depends on the share assigned to its class. Three approaches to achieve this are discussed next.

Priority modifications (WFS+P): Let a *switching interval* be defined as the time interval between consecutive array switches of the modified scheduler, Δ_j be its duration and t_j and t_{j+1} be the timestamps of the switches. In the priority modifications approach to alleviating starvation in WFS, henceforth called WFS+P, we track the number of array switches se at which a task got starved due to its class' timeslice exhaustion and increase the task's effective priority based on se, i.e. $ep(p) = \Phi(sp, sa, se)$. This ensures that starving tasks eventually get their ep high enough to get a chance to run at which point se is reset. The drawback of this approach is that the increased scheduler overhead of tasks being selected for execution and moving directly to the expired list due to class timeslice exhaustion, remains unchanged.

Timeslice modifications (WFS+T1, WFS+T2): Recognizing that starvation can occur in WFS for class C_i only if $\sum_{p \in C_i} ts(p) > TS_i$, the timeslice modification approaches attempt to change one or the other side of the inequality to convert it to an equality. In WFS+T1, the left hand side of the inequality is changed by reducing the timeslices of each task of a starved class as follows. Let $exh_i = \sum_{p \in C_i} ts(p) - TS_i$ when class timeslice exhaustion occurs. At array switch time, each $ts(p)$ is multiplied by $\lambda_i = \frac{TS_i}{TS_i + exh_i}$ which results in the desired equality. WFS+T1 is slow to respond to starvation because task timeslices are recomputed in O(1) *before* they move into the expired array and not at array switch time. Hence any task timeslice changes take effect only one switching interval later i.e. two intervals beyond the one in which starvation occurred. One way to address this problem is to treat a task as having exhausted its timeslice when $ts(p)$ gets decremented to $(1 - \lambda_i \times ts(p))$ instead of 0. A bigger problem with WFS+T1 is that smaller timeslices for tasks could lead to increased context switches with potentially negative cache effects.

To avoid reducing $ts(p)$'s, WFS+T2 increases TS_i of a starving class to make$\sum_{p \in C_i} ts(p) = TS_i$ i.e. the class does not exhaust its timeslice until each of its tasks have exhausted their individual timeslices. To preserve the relative proportions between class timeslices, all other class timeslices also need to be changed appropriately. Doing so would disturb the same equality for those classes and hence WFS+T2 is not a workable approach.

Two-level scheduler: Another way to regulate CPU shares in WFS is to take tasks out of the runqueue upon timeslice exhaustion and return them to the runqueue at a rate commensurate with the share of the class. A prototype implementation of this approach was described in [17] in the context of user-based fair sharing. This approach effectively implements a two-level scheduler and is illustrated in Figure 2. A modified O(1) scheduler forms the lower level and a coarse-grain scheduler operates at the upper level, replenishing class timeslice ticks. In the modified O(1), when a task expires, it is moved into a FIFO list associated with its class instead of moving to the expired array. At a coarse-granularity determined by the upper level scheduler, the class receives new time ticks and reinserts tasks from the FIFO back into O(1)'s runqueue. Class time tick replenishment can be done for all classes at every array switch point but that violates the O(1) behaviour of the scheduler as a whole. To address this problem, [17] uses a kernel thread to replenish 8 ms worth of ticks to one class (user) every 8 ms and round robin through the classes (users). A variant of this idea is currently being explored.

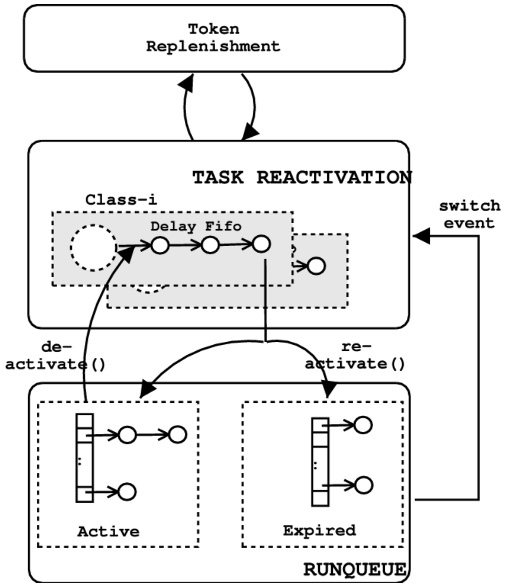

Figure 2: Proposed two-level CPU scheduler

4 Disk scheduling

The I/O scheduler in Linux forms the interface between the generic block layer and the low level device drivers. The block layer provides functions which are used by filesystems and the virtual memory manager to submit I/O requests to block devices. These requests are transformed by the I/O scheduler and made available to the low-level device drivers (henceforth only called device drivers). Device drivers consume the transformed requests and forward them, using device specific protocols, to the device controllers which perform the I/O. Since prioritized resource management seeks to regulate the use of a disk by an application, the I/O scheduler is an important kernel component that is sought to be changed. It is also possible to regulate disk usage in the kernel layers above and below the I/O scheduler. Changing the pattern of I/O load generated by filesytems or the virtual memory manager (VMM) is an important option. A less explored option is to change the way specific device drivers or even device controllers consume the I/O requests submitted to them. The latter approach is outside the scope of general kernel development and this paper.

Class-based resource management requires two fundamental changes to the traditional approach to I/O scheduling. First, I/O requests should be managed based on the priority or weight of the request submitter with disk utilization being a secondary, albeit important objective. Second, I/O requests should be associated with the class of the request submitter and not a process or task. Hence the weight associated with an I/O request should be derived from the weight of the class generating the request.

The first change is already occurring in the development of the 2.5 Linux kernel with the development of different I/O schedulers such as deadline, anticipatory, stochastic fair queueing and complete fair queueing. Consequently, the *additional* requirements imposed by the second change (scheduling by class) are relatively minor. This fits in well with our project goal of minimal changes to existing resource schedulers.

We now describe the existing Linux I/O schedulers followed by an overview of the changes being proposed.

4.1 Existing Linux I/O schedulers

The various Linux I/O schedulers can be abstracted into a generic model shown in Figure 3. I/O requests are generated by the block layer on behalf of processes accessing filesystems, processes performing raw I/O and from the virtual memory management (VMM) components of the kernel such as kswapd, pdflush etc. These producers of I/O requests call __make_request() which invokes various I/O scheduler functions such as elevator_merge_fn. The enqueuing functions' generally try to merge the newly submitted block I/O unit (bio in 2.5 kernels, buffer_head in

2.4 kernels) with previously submitted requests and sort it into one or more internal queues. Together, the internal queues form a single logical queue that is associated with each block device. At a later point, the low-level device driver calls the generic kernel function elv_next_request() to get the next request from the logical queue. elv_next_request interacts with the I/O scheduler's dequeue function elevator_next_req_fn and the latter has an opportunity to pick the appropriate request from one of the internal queues. The device driver then processes the request, converting it to scatter-gather lists and protocol-specific commands that are then sent to the device controller. As far as the I/O scheduler is concerned, the block layer is the producer of I/O requests and the device drivers are the consumers. Strictly speaking, the block layer includes the I/O scheduler but we distinguish the two for the purposes of our discussion.

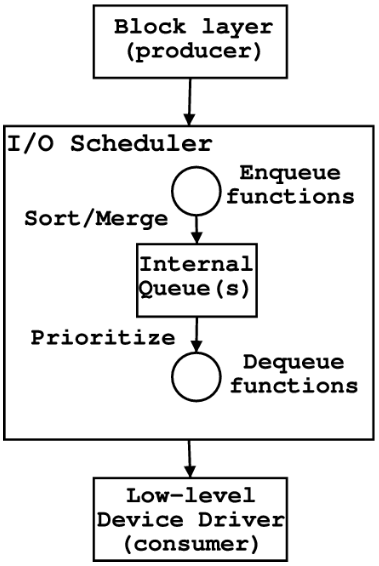

Figure 3: Abstraction of Linux I/O scheduler

Default 2.4 Linux I/O scheduler: The 2.4 Linux kernel's default I/O scheduler (elevator_linus) primarily manages disk utilization. It has a single internal queue. For each new bio, the I/O scheduler checks to see if it can be merged with an existing request. If not, a new request is placed in the internal queue sorted by the starting device block number of the request. This minimizes disk seek times if the disk processes requests in FIFO order from the queue. An aging mechanism limits the number of times an existing request can get bypassed by a newer request, preventing starvation. The dequeue function is simply a removal of requests from the head of the internal queue. Elevator_linus also has the welcome property of improving request response times *averaged* over all processes.

Deadline I/O scheduler: The 2.5 kernel's default I/O scheduler (deadline_iosched) introduces the notion of a per-request deadline which is currently used to give a higher preference to read requests. Internally, it maintains five queues. During enqueing, each request is assigned a deadline and inserted into queues sorted by starting sector (sort_list) *and* by deadline (fifo_list). Separate sort and fifo lists are maintained for read and write requests. The fifth internal queue contains requests to be handed off to the driver. During a dequeue operation, if the dispatch queue is empty, requests are moved from one of the four sort or fifo lists in batches. Thereafter, or if the dispatch queue was not empty, the head request on the dispatch queue is passed on to the driver. The logic for moving requests from the sort or fifo lists ensures that each read request is processed by its deadline without starving write requests. Disk seek times are amortized by moving a large batch of requests from the sort_list (which are likely to have few seeks as they are already sector sorted) and balancing it with a controlled number of requests from the fifo list (each of which could cause a seek since they are ordered by deadline and not sector). Thus, deadline_iosched effectively emphasizes average read request response times over disk utilization and total average request response time.

Anticipatory I/O scheduler: The anticipatory I/O scheduler [9, 4] attempts to reduce *per-process* read response times. It introduces a controlled delay in dispatching *any* new requests to the device driver, thereby allowing a process whose request just got serviced to submit a new request, potentially requiring a smaller seek. The tradeoff between reduced seeks and decreased disk utilization (due to the additional delays in dispatch) are managed using a cost-benefit calculation. anticipatory I/O scheduling method is an additional optimization that can potentially be added to any of the I/O scheduler mentioned in this paper

Complete Fair Queueing I/O scheduler: Two new I/O schedulers recently proposed in the Linux kernel community, introduce the concept of fair allocation of I/O bandwidth amongst producers of I/O requests. The Stochastic Fair Queueing (SFQ) scheduler [5] is based on an algorithm originally proposed for network scheduling [11]. It tries to apportion I/O bandwidth equally amongst all *processes* in a system using 64 internal queues and one output (dispatch) queue. During an enqueue, the process ID of the currently running process (very likely to be the I/O request producer) is used to select one of the internal queues and the request inserted in FIFO order within it. During dequeue, SFQ round-robins through the non-empty internal queues, picking requests from the head. To avoid too many seeks, one full round of requests are collected, sorted and merged into the dispatch queue. The head request of the dispatch queue is then passed to the device driver. Complete Fair Queuing is an extension of the same approach where no hash function is used. Hence each process in the system has a corresponding internal queue and can get an fair share of the I/O bandwidth (equal share if all processes generate I/O requests at the same rate). Both CFQ and SFQ manage per-process I/O bandwidth and can provide fairness at a process granularity.

Cello disk scheduler: Cello is a two-level I/O scheduler [16] that distinguishes between classes of I/O requests and allows each class to be serviced by a different policy. A coarse grain class-independent scheduler decides how many requests to service from each class. The second level class-dependent scheduler then decides which of the requests from its class should be serviced next. Each class has its own internal queue which is manipulated by the class-specific scheduler. There is one output queue common to all classes. Enqueuing into the output queue is done by the class-specific schedulers in a way that ensures individual request deadlines are met as far as possible while reducing overall seek time. Dequeuing from the output queue occurs in FIFO order as in most of the previous I/O schedulers. Cello has been shown to provide good isolation between classes as well as the ability to meet the needs of streaming media applications that have soft realtime requirements for I/O requests.

4.2 Costa: Proposed I/O scheduler

This paper proposes that a modified version of the class-independent scheduler of the Cello I/O scheduling framework can provide a low-overhead class-based I/O scheduler suitable for CKRM's goals.

The key difference between the proposed scheduler called Costa and Cello is the elimination of the class-specific I/O schedulers which may be add unnecessary overhead for CKRM's goal of I/O bandwidth regulation. Fig 4 illustrates the Costa design. When the kernel is configured for CKRM support, a new internal queue is created for each class that gets added to the kernel. Since each process is always associated with a class, I/O requests that they generate can also be tagged with the class id and used to enqueue the request in the class-

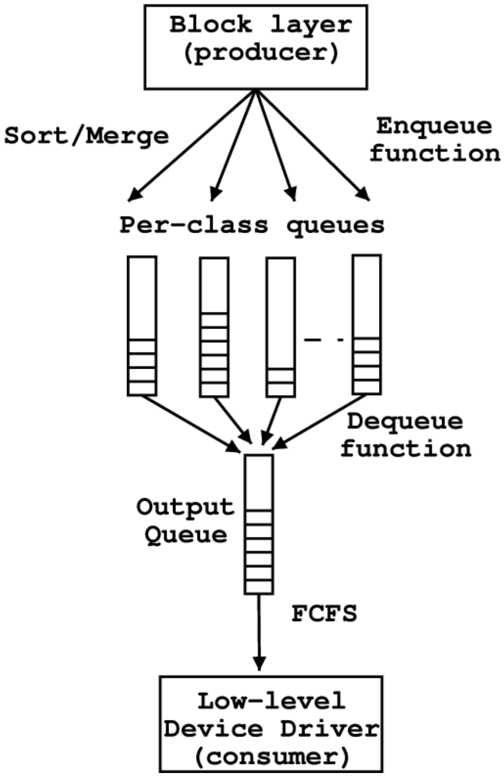

Figure 4: Proposed Costa I/O scheduler

specific internal queue. The request->class association cannot be done through a lookup of the *current* process' class alone. During dequeue, the Costa I/O scheduler picks up requests from several internal queues and sorts them into the common dispatch queue. The head request of the dispatch queue is then handed to the device driver.

The mechanism also allows internal queues to be associated with *system* classes that group I/O requests coming from important producers such as the VMM. By separating these out, Costa can give them preferential treatment for urgent VM writeout or swaps.

In addition to a weight value (which determines the fraction of I/O bandwidth that a class will receive), the internal queues could also have an associated *priority* value which determines their relative importance. At a given priority level, all queues could receive I/O bandwidth in proportion of their weights with the set of queues at a higher level always getting serviced first. Some check for preventing starvation of lower priority queues could be used similar to the ones used in deadline_iosched.

5 QoS Networking in Linux

Many research efforts have been made in networking QoS (Quality of Service) to provide quality assurance of latency, bandwidth, jitter, and loss rate. With the proliferation of multimedia and quality-sensitive business traffic, it becomes essential to provide reserved quality services (IntServ [23]) or differentiated services (DiffServ [2]) for important client traffic.

The Linux kernel has been offering a well established QoS network infrastructure for outbound bandwidth management, policy-based routing, and DiffServ. Hence, Linux is being widely used for routers, gateways, and edge servers, where network bandwidth is the primary resource to differentiate among classes.

When it comes to Linux as an end server OS, on the other hand, networking QoS has not been given as much attention because QoS is primarily governed by the system resources such as CPU, memory, and I/O and less by the network bandwidth. However, when we consider the end-to-end service quality, we also should require networking QoS in the end servers as exemplified by the fair share admission control mechanism proposed in this section.

In the rest of the section, we first briefly introduce the existing network QoS infrastructure of Linux. Then, we describe the design of the fair share admission control in Linux and preliminary performance results.

5.1 Linux Traffic Control, Netfilter, DiffServ

The Linux traffic control [8] consists of queueing disciplines (qdisc) and filters. A qdisc consists of one or more queues and a packet scheduler. It makes traffic conform to a certain profile by shaping or policing. A hierarchy of qdiscs can be constructed jointly with a class hierarchy to make different traffic classes governed by proper traffic profiles. Traffic can be attributed to different classes by the filters that match the packet header fields. The filter matching can be stopped to police traffic above a certain rate limit. A wide range of qdiscs ranging from a simple FIFO to classful CBQ or HTB are provided for outbound bandwidth management, while only one ingress qdisc is provided for inbound traffic filtering and policing [8]. The traffic control mechanism can be used in various places where bandwidth is the primary resource to control. For instance in service providers, it manages bandwidth allocation shared among different traffic flows belonging to different customers and services based on service level agreements. It also can be used in client sites to reduce the interference between upstream and downstream traffic and to enhance the response time of the interactive and urgent traffic.

Netfilter provides sophisticated filtering rules and targets. Matched packets can be accepted, denied, marked, or mangled to carry out various edge services such as firewall, dispatcher, proxy, NAT etc. Routing decisions can be made based on the netfilter markings so packets may take different routes according to their classes. The qdiscs would enable various QoS features in such edge services when used with Netfilter. Netfilter classification can be transferred for use in later qdiscs by markings or mangled packet headers.

The Differentiated Service (DiffServ) [2] provides a scalable QoS by applying per-hop behavior (PHB) collectively to aggregate traffic classes that are identified by a 6-bit code point in the IP header. Classification and conditioning are typically done at the edge of a DiffServ domain. The domain is a contiguous set of nodes compliant to a common PHB. The DiffServ PHB is supported in Linux [22]. Classes, drop precedence, code point marking, and conditioning can be implemented by qdiscs and filters. At the end servers, the code point can be marked by setting the `IP_TOS` socket option.

In the policy based networking [18], a policy agent can configure the traffic classification of edge and end servers according to a predefined filtering rules that match layer 3/4 or layer 7 information. Netfilter, qdisc, and application layer protocol engines can classify traffic for differentiated packet processing at later stages. Classifications at prior stages can be overridden by the transaction information such as URI, cookies, and user identities as they are known. It has been shown that a coordination of server and network QoS can reduce end-to-end response time of important client requests significantly by virtual isolation from the low priority traffic [15].

5.2 Prioritized Accept Queues with Proportional Share Scheduling

We present here a simple change to the existing Linux TCP accept mechanism to provide differentiated service across priority classes. Recent work in this area has introduced the concept of prioritized accept queues [19] and accept queue schedulers using adaptive proportional shares to self-managed web servers [14].

Under certain load conditions [14], the TCP accept queue of each socket becomes the bottleneck in network input processing. Normally, listening sockets fork off a child process to handle an incoming connection request. Some optimized applications such as the Apache web

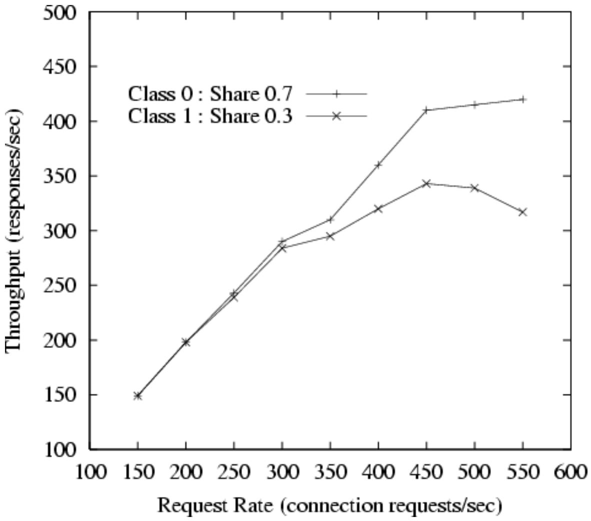

Figure 5: Proportional Accept Queue Results.

server maintain a pool of server processes to perform this task. When the number of incoming requests exceeds the number of static pool servers, additional processes are forked up to a configurable maximum. When the incoming connection request load is higher than the level that can be handled by the available server processes, requests have to wait in the accept queue until one is available.

In the typical TCP connection, the client initiates a request to connect to a server. This connection request is queued in a single global accept queue belonging to the socket associated with that server's port. Processes that perform an `accept()` call on that socket pick up the next queued connection request and process it. Thus all incoming connections to a particular TCP socket are serialized and handled in FIFO order.

We replace the existing single accept queue per socket with multiple accept queues, one for each priority class. Incoming traffic is mapped into one of the priority classes and queued on the accept queue for that priority. There are eight priority classes in the current implementation.

The accepting process schedules connection acceptance according to a simple weighted deficit round robin to pick up connection requests from each queue according to its assigned share. The share, or weight can be assigned by the sysctl interface.

In the basic priority accept queue design proposed earlier in [6], starvation of certain priority classes was a possibility as the accepting process picked up connection requests in a descending priority order. With a proportional share scheme in this paper, it is easier to avoid starvation of particular classes to give share guarantees to low priority classes.

The efficacy of the proportional accept queue mechanism is demonstrated by an experiment. In the experiment, we used Netfilter with mangle tables and `MARK` options to characterize traffic into priority classes based on source IP address. Httperfs from two client machines send requests to an Apache web server running on a single server over independent gigabit Ethernet connections. The only Apache parameter changed from the default was the maximum number of httpd threads. This was set to 50 in the experiment.

Figure 5 shows throughput of Apache for two priority classes, sharing inbound connection bandwidth by 7:3. We can see that the throughput of the priority class 0 requests is slightly higher than that of the priority class 1 requests when the load is low. As load increases, the acceptance rates to the priority classes 0 and 1 will be constrained in proportion to their relative share, which in turn determines the processing rate of the Apache web server and connection request queueing delay. Under a severe load, the priority class 0 requests are processed at a considerably higher throughput.

6 Controlling Memory

While many other system resources can be managed according to priorities or proportions, *virtual memory managers (VMM)* currently do not allow such control. The *resident set size (RSS)* of each process—the number of physical page frames allocated to that process–will determine how often that process incurs a page fault. If the RSS of each process is not somehow proportially shared or prioritized, then paging behavior can overwhelm and undermine the efforts of other resource management policies.

Consider two processes, A and B, where A is assigned a larger share than B with the CPU scheduler. If A is given too small an RSS and begins to page fault frequently, then it will not often be eligible for scheduling on the CPU. Consequently, B will be scheduled more often than A, and the VMM will have become the *de facto* CPU scheduler, thus violating of the requested scheduling proportions.

Furthermore, it is possible for most existing VMM policies to exhibit a specific kind of degenerative behavior. Once process A from the example above begins to page fault, its infrequent CPU scheduling prevents it from referencing its pages at the same rate as other, frequently scheduled processes. Therefore, its pages will become more likely to evicted, thereby reducing its RSS. The smaller RSS will *increase* the probability of further page faults. This degenerative feedback loop will cease only when some other process either exits or changes its reference behavior in a manner that reduces the competition for main memory space.

Main memory use must be controlled just as any other resource is. The RSS of each *address space*—the logical space defined either by a file or by the anonymous pages within the virtual address space of a process—must be calculated as a function of the proportions (or priorities) that are used for CPU scheduling. Because this type of memory management has received little applied or academic attention, our work in this area is still nascent. We present here the structural changes to the Linux VMM necessary for proportional/prioritized memory management; we also present previous, applicable research, as well as future research directions that will address this complex problem. While the proportional/prioritized management of memory is currently less well developed than the other resources presented in this paper, it is necessary that it be comparably developed.

6.1 Basic VMM changes

Consider a function that explicitly calculates the desired RSS for each address space—the *target RSS*—when the footprints of the active address spaces exceeds the capacity of main memory. After this function sets these targets, a system could *immediately* bring the actual RSS into alignment with these targets. However, doing so may require a substantial number of page swapping operations. Since disk operations are so slow, it is inadvisable to use aggressive, pre-emptive page swapping. Instead, a system should seek to move the the actual RSS values toward their targets in a *lazy* fashion, one page fault at a time. Until the actual RSS of address space matches its target, it can be labeled as being either in *excess* or in *deficit* of its target.

As the VMM reclaims pages, it will do so from address spaces with excess RSS values. This approach to page reclamation suggests a change in the structure of the *page lists*—specifically, the *active* and *inactive* lists that impose an order on pages[1]. The current Linux

[1] In the Linux community, these are known as the

VMM uses *global* page lists. If this approach to ordering pages in unchanged, then the VMM would have to search the page lists for pages that belong to the desired address spaces that have excess RSS values. Alternatively, one pair of page lists—*active* and *inactive*—could exist for each address space. The reclamation of pages from a specific address space would therefore require no searching.

By ordering pages separately with each address space, we also enable the VMM to more easily track reference behavior for each address space. While the information to be gathered would depend on the underlying policy that selects target RSS values, we believe that such tracking may play an important role in the development of such policies.

Note that the target RSS values would need to be recalculated periodically. While the period should be directly proportional to the *memory pressure*—some measure of the current workload's demand for main memory space—it is a topic of future research to determine what that period should be. By coupling the period for target RSS calculations to the memory pressure, we can ensure that this strategy only incurs significant computational overhead when heavy paging is occuring.

6.2 Proportionally sharing space

Waldspurger [20] describes a method of proportionally sharing the space of a system running VMWare's ESX Server between a number of virtual machines. Specifically, as with a proportional share CPU scheduler, memory shares can be assigned to each virtual machine, and Waldspurger's policy will calculate target RSS values for each virtual machine.

Proportionally sharing main memory space can result in superfluous allocations to some virtual machines. If the target RSS for some virtual machine is larger than necessary, and some of the main memory reserved for that virtual machine is rarely used[2], then its target RSS should be reduced. Waldspurger addresses this problem with a *taxation policy*. In short, this policy penalizes each virtual machine for unused portions of its main memory share by reducing its target RSS.

Application to the Linux VMM. This approach to proportionally sharing main memory could easily be generalized so that it can apply to address spaces within Linux instead of virtual machines on an ESX Server. Specifically, we must describe how shares of main memory space can be assigned to each address space. Given those shares, target RSS values can be calculated in the same manner as for virtual machines in the original research.

The taxation scheme requires that the system be able to measure the active use of pages in each address space. Waldspurger used a sampling strategy where some number of randomly selected pages for each virtual machine were access protected, forcing *minor page faults* to occur upon the first subsequent reference to those pages, and therefore giving the ESX Server an opportunity to observe those page uses. The same approach could be used within the Linux VMM, where a random sampling of pages in each address space would be access protected. Alternatively, a sampling of the pages' reference bits could be used to monitor idle memory.

Waldspurger observes that, within a normal OS, the use of access protection or reference bits, taken from virtual memory mappings, will not detect references that are a result of DMA transfers. However, since such DMA transfers

LRU lists. However, they only approximate an LRU ordering, and so we refer to them only as *page lists*.

[2] Waldspurger refers to such space as being *idle*.

are scheduled within the kernel itself, those references could also be explicitly counted with help from the DMA scheduling routines.

6.3 Future directions

The description above does not address a particular problem: *shared memory*. Space can be shared between threads or processes in a number of ways, and such space presents important problems that we must solved to achieve a complete solution.

The problem of shared spaces. The assignment of shares to an address space can be complicated when that address space is shared by processes in different process groups or service classes. One simple approach is for each address space to belong to a specific service class. In this situation, its share would be defined only by that service class, and not by the processes that share the space. Another approach would be for each shared address space to adopt the highest share value of its shared tasks or processes. In this way, an important process will not be penalized because it is sharing its space with a less important process.

Note that this problem does not apply only to memory mapped files and IPC shared memory segments, but to any shared space. For example, the threads of a multi-threaded process share a virtual address space. Similarly, when a process calls `fork()`, it creates a new virtual address space whose pages are shared with the original virtual address space using the *copy-on-write (COW)* mechanism. These shared spaces must be assigned proportional shares even though the tasks and processes using them may themselves have differing shares.

Proportionally sharing page faults. The goal of proportional share scheduling is to fairly divide the utility of a system among competing clients (e.g., processes, users, service classes). It is relatively simple to divide the utility of the CPU because that utility is *linear* and *indepedent*. One second of scheduled CPU time yields a nearly fixed number of executed instructions[3]. Therefore, each additional second of scheduled CPU time nearly yields a constant increase in the number of executed instructions. Furthermore, the utility of CPU does not depend on the computation being performed: Every process derives equal utility from each second of scheduled CPU time.

Memory, however, is a more complex resource because its utility is neither independent nor linear. Identical RSS values for two processes may yield vastly different numbers of page faults for each process. The number of page faults is dependent on the reference patterns of each process.

To see the non-linearity in the utility of memory, consider two processes, A and B. Assume that for A, an RSS of p pages will yield m misses, where $m > 0$. If that RSS were increased to $p + q$ pages, the number of misses incurred by A may take any value m' where $0 \leq m' \leq m$[4]. Changes in RSS do not imply a constant change in the number of misses suffered by an address space.

The proportial sharing of memory *space*, therefore, does not necessarily achieve the stated goal of fairly dividing the utility of a system. Consider that A should receive 75% of the system, while B should receive the remaining 25%. Dividing the main memory space by these proportions could yeild heavy page fault-

[3] We assume no delays to access memory, as memory is a separate resource from the CPU.

[4] We ignore the possibility of *Belady's anomaly*[1], in which an increase in RSS could imply an increase in page faults. While this anomaly is likely possible for any real, in-kernel page replacement policy, it is uncommon and inconsequential for real workloads.

ing for A but not for B. Note also that none of the 25% assigned to B may be idle, and so Waldspurger's taxation scheme will not reduce its RSS. Nonetheless, it may be the case that a reduction in RSS by 5% for B may increase its misses only modestly, and that an increase in RSS by 5% for A may reduce its misses drastically.

Ultimately, a system should proportionally share the utility of main memory. We consider this topic a matter of significant future work. It is not obvious how to measure online the utility of main memory for each address space, nor how to calculate target RSS values based on these measurements. Balancing the contention between fairness and throughput for virtual memory must be considered carefully, as it will be unacceptable to achieve fairness simply by forcing some address spaces to page fault more frequently. We do, however, believe that this problem can be solved, and that the utility of memory can be proportionally shared just as with other resources.

7 Conclusion and Future Work

In this paper we make a case for providing kernel support for class-based resource management that goes beyond the traditional per process or per group resource management. We introduce a framework for classifying tasks and incoming network packets into classes, monitoring their usage of physical resources and controlling the allocation of these resources by the kernel schedulers based on the shares assigned to each class. For each of four major physical resources (CPU, disk, network and memory), we discuss ways in which proportional sharing could be achieved using incremental modifications to the corresponding existing schedulers.

Much of this work is in its infancy and the ideas proposed here serve only as a starting point for future work and for discussion in the kernel community. Prototypes of some of the schedulers discussed in this paper are under development and will be made available soon.

8 Acknowledgments

We would like to thank team members from the Linux Technology Center, particularly Theodore T'so, for their valuable comments on the paper and the work on individual resource schedulers. Thanks are also in order for numerous suggestions from the members of the kernel open-source community.

References

[1] L. A. Belady. A study of replacement algorithms for virtual storage. *IBM Systems Journal*, pages 5:78–101, 1966.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, Dec 1998.

[3] Jonathan Corbet. A new deadline I/O scheduler. `http://lwn.net/Articles/10874`.

[4] Jonathan Corbet. Anticipatory I/O scheduling. `http://lwn.net/Articles/21274`.

[5] Jonathan Corbet. The Continuing Development of I/O Scheduling. `http://lwn.net/Articles/21274`.

[6] IBM DeveloperWorks. Inbound connection control home page. `http://www-124.ibm.com/pub/qos/paq_index.html`.

[7] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996.

[8] Bert Hubert. Linux Advanced Routing & Traffic Control. `http://www.lartc.org`.

[9] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, October 2001.

[10] M. Kravetz, H. Franke, S. Nagar, and R. Ravindran. Enhancing Linux Scheduler Scalability. In *Proc. 2002 Ottawa Linux Symposium, Ottawa*, July 2001. `http://lse.sourceforge.net/scheduling/ols2001/elss.ps`.

[11] Paul E. McKenney. Stochastic Fairness Queueing. In *INFOCOM*, pages 733–740, 1990.

[12] Ingo Molnar. Goals, Design and Implementation of the new ultra-scalable O(1) scheduler. In 2.5 kernel source tree documentation (Documentation/sched-design.txt).

[13] Jason Nieh, Chris Vaill, and Hua Zhong. Virtual-time round-robin: An o(1) proportional share scheduler. In *2001 USENIX Annual Technical Conference*, June 2001.

[14] P. Pradhan, R. Tewari, S. Sahu, A. Chandra, and P. Shenoy. An Observation-based Approach Towards Self-Managing Web Servers. In *IWQoS 2002*, 2002.

[15] Quality of Service White Paper. Integrated QoS: IBM WebSphere and Cisco Can Deliver End-to-End Value. `http://www-3.ibm.com/software/webservers/edgeserver/doc/v20/QoSwhitepap%er.pdf`.

[16] Prashant J. Shenoy and Harrick M. Vin. Cello: A disk scheduling framework for bext generation operating systems. In *ACM SIGMETRICS 1998*, pages 44–55, Madison, WI, June 1998. ACM.

[17] Antonio Vargas. fairsched+O(1) process scheduler. `http://www.ussg.iu.edu/hypermail/linux/kernel/0304.0/0060.html`.

[18] Dinesh Verma, Mandis Beigi, and Raymond Jennings. Policy based SLA Management in Enterprise Networks. In *Policy Workshop*, 2001.

[19] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *2001 USENIX Annual Technical Conference*, Jun 2001.

[20] Carl A. Waldspurger. Memory resource management in {VM}ware {ESX} {S}erver. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[21] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional- share resource management. Technical Report MIT/LCS/TM-528, 1995.

[22] Werner Almesberger Werner and Jamal Hadi Salim amd Alexye Kuznetsov. Differentiated Services on Linux. In *Globecom*, volume 1, pages 831–836, 1999.

[23] J. Wroclawski. The Use of RSVP with IETF Integrated Services. RFC 2210, Sep 1997.

Trademarks and Disclaimer

This work represents the view of the authors and does not necessarily represent the view of IBM.

IBM is a trademark or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other trademarks are the property of their respective owners.