Advanced Workload Management Support for Linux

Hubertus Franke, Shailabh Nagar, Chandra Seetharaman, Vivek Kashyap Rik van Riel

{frankeh, nagar, sekharan, kashyapv}@us.ibm.com, riel@redhat.com

IBM Corp., Red Hat, Inc.

Abstract

Workload management in enterprise operating systems is being increasingly driven by two requirements. On the one hand, workloads with diverse and dynamically changing resource demands are being consolidated on larger symmetric multiprocessors requiring efficient, business goal oriented workload management. On the other hand, autonomic computing initiatives are seeking to reduce the complexity and manual involvement in systems management. We argue that the goal-oriented workload managers that can satisfy these conflicting objectives require the operating system kernel to provide class-based differentiated service for all the resources that it manages. We discuss an extensible framework for class-based kernel resource management (CKRM) that provides policy-driven classification and differentiated service of CPU, memory, I/O and network bandwidth. The paper describes the design and implementation of the framework in the Linux 2.6 kernel. It also presents preliminary performance evaluation results that demonstrate the viability of the approach.

Introduction

Workload management is an increasingly important requirement of modern enterprise computing systems. There are two trends driving the development of enterprise workload management middleware. One is the consolidation of multiple workloads onto large symmetric multiprocessors (SMPs) and mainframes. Their diverse and dynamic resource demands require workload managers (WLMs) to provide efficient differentiated service at finer time scales to maintain high utilization of expensive hardware. The second trend is the move towards specification of workload performance in terms of the business importance of the workload rather than in terms of low-level system resource usage. This has led to the increasing use of goal-oriented workload managers which are more tightly integrated into the business processes of an enterprise.

Traditional workload managers [1,2] have been built with two layers. The lower, OS specific layer deals with modifying and monitoring operating system parameters. The upper layer(s) provide a largely OS independent API, generally through a graphical user interface, allowing a multi-tier or clustered system to be managed through a unified API despite containing heterogenous operating systems. Such WLMs are able to manage heterogeneity but do little to address complexity. The burden of translating business goals into workload resource requirements and the latter into OS specific tuning parameters remains on the human system administrator. Increasing workload consolidation only adds more complexity to an already onerous problem.

The autonomic computing initiatives recently announced by several server vendors are starting to address the complexity problem through goal-oriented workload managers [3,4] which allow a system to be more self-managed. Such WLMs allow the human system administrator to specify high level performance objectives in the form of *policies*, closely aligned with the business importance of the workload. The WLM middleware then uses adaptive feedback control over OS tuning parameters to realize the given objectives.

In mainstream operating systems the control of key resources such as memory, CPU time, disk I/O bandwidth and network bandwidth is typically strongly tied to processes, tasks and address spaces and are highly tuned to maximize system utilization. This introduces additional complexity to the WLM which needs to translate the QoS requirements into these low level per task requirements, tough typically QoS is enforced at work class level. Hence, in order to isolate the autonomic goal oriented layers of the system management from the intricacies of the operating system we introduce the class concept into the operating system kernel and require the OS to provide *differentiated* service for *all* major resources at a class granularity defined *by the WLM*.

In this paper, we discuss a framework called class-based kernel resource management (CKRM) that implements this support under Linux. In CKRM, a class is defined as a dynamic grouping of OS objects of a particular type (*classtype*) and defined through policies provided by the WLM. Each class has an associated share of each of its resources. For instance, CKRM tasks classes provides resource management for four principal physical resources managed by the kernel namely CPU time, physical memory pages, disk I/O and bandwidth. Sockets classes provide inbound network bandwidth resource control. The Linux resource schedulers are modified to provide differentiated service at a class granularity based on the assigned shares. The WLM can dynamically modify the composition of a class and its share in order to meet higher level business goals. We evaluate the performance of the CKRM using simple benchmarks that demonstrate the efficacy of its approach.

This work makes several contributions that distinguish it from previous related work such as resource containers [5] and cluster reserves [6]. First, it describes the design of a flexible kernel framework for class-based management that can be used to manage both physical and virtual resources (such as number of open files). The framework allows the various resource schedulers and classification engine to be developed and deployed independent of each other. Second, it shows how incremental modifications to existing Linux resource schedulers can make them provide differentiated service effectively at a class granularity. To our knowledge, this is the first open-source resource management package that attempts to provide control over all the major physical resources i.e. CPU, memory, I/O and network. Third, it provides a policy-driven classification engine that eases the development of new higher level WLMs and enables better coordination between multiple WLMs through policy exchange. Finally, it develops a tagging mechanism that allows server applications to participate in their resource management in conjunction with the WLM.

Overview

A typical WLM defines a workload to be any system work with a distinct business goal. From a Linux operating system's viewpoint, a workload is a set of kernel tasks executing over some duration. Some of these tasks are dedicated to this workload. Other tasks, running server applications such as database or web servers, perform work for multiple workloads. Such tasks can be viewed as executing in *phases* with each phase dedicated to one workload. Server tasks can explicitly inform the WLM of its phase by setting an application tag. A WLM can also infer the phase by monitoring significant system events such as forks, execs, setuid etc. and classifying the server task as best as possible.

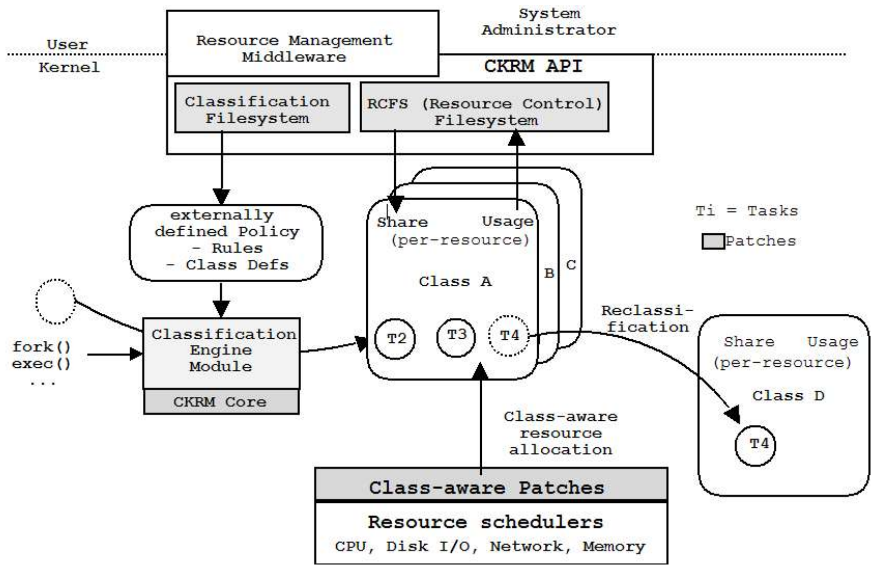
User
Kernel
Resource Management Middleware
System Administrator
CKRM API
Classification Filesystem
RCFS (Resource Control) Filesystem
externally defined Policy - Rules - Class Defs
Share
Usage
(per-resource)
Class A
B
C
Ti = Tasks
Patches
fork()
exec()
...
Classification Engine Module
CKRM Core
T2
T3
T4
Reclassi- fication
Share
Usage
(per-resource)
Class D
T4
Class-aware resource allocation
Class-aware Patches
Resource schedulers
CPU, Disk I/O, Network, Memory

Figure 1: CKRM Framework and Lifecycle

In this scenario, a WLM translates a high level business goal of a workload (say response time) into system goals for the set of tasks executing the work-load. The system goals are a set of delays seen by the workload in waiting for individual resources such as CPU ticks, memory pages etc. The WLM monitors the business goals, possibly using applica-tion assistance, and the system usage of its resources. If the business goal is not being met, it identifies the system resource(s) which form a performance bottleneck for the workload and adjusts the workload's share of the resource appropriately.

The CKRM framework enables a WLM to regulate workloads through a number of components, as shown in Figure 1:

Core : The core defines the basic entities used by CKRM and serves as the link between all the other components. A *class* is a group of kernel objects with an associated set of contraints for resource controllers operating on those kernel objects e.g. a class could consist of a group of tasks which have a joint share of cpu time and resident page frames. Each class has an associated *classtype* which identifies the kernel object being grouped. CKRM currently defines two classtypes called *task_class* and *socket_class* for grouping tasks and sockets. For brevity, the term taskclass and socketclass will be used to denote a class of classytpe task_class and socket_class respectively. Classtypes can be enabled selectively and independent of each other. A user not interested in network regulation could choose to disable socket_classes. Classes in CKRM are hierarchical. Children classes can be defined to subdivide the resources allocated to the parent.

Classification engine (CE): This optional components assists in the association of kernel objects to classes of its associated classtype. Each kernel object managed by CKRM is always associated with some class. If no classes are defined by the user, all objects belong to the default class for the classtype. At significant kernel events such as fork, exec, setuid, listen, when the attributes of a kernel object are changed, the Core queries the CE, if one is present, to get the class into which the object should be placed. CE's are free to use any logic to return the classification. CKRM provides a rule-based classification engine (RBCE) which allows privileged users to define rules which use attribute matching to return the class. RBCE is expected to meet the needs of most users though they can define their own CE's or choose not to have any and rely upon manual classification of each kernel object through CKRM's rcfs user interface (described later).

Resource Controllers/Managers: Each classtype has a set of associated resource controllers, typically one for each resource associated with the classtype e.g. taskclasses have cpu, memory and I/O controllers to regulate the cpu ticks, resident page frames and per-disk I/O bandwidth consumed by it while socketclasses have an accept queue controller to regulate the number of TCP connections accepted by member sockets. Resource requests by a kernel object in a class are regulated by the corresponding resource controller, if one exists and is enabled. The resource controllers are deployed independent of each other so a user interested only in controlling CPU time for taskclasses could choose to disable the memory and I/O controllers (as well as the socketclass classtype and all its resource controllers).

Resource Control File System (RCFS): It forms the main user-kernel interface for CKRM. Once RCFS is mounted, it provides a hierarchy of directories and files which can be manipulated using well-known file operations such as open, close, read, write, mkdir, rmdir and unlink. Directories of rcfs correspond to classes. User-kernel communication of commands and responses is done through reads/writes to virtual files in the directories. Writes to the virtual files trigger CKRM Core functions and responses are available through reads of the same virtual file.

The CKRM architecture outlined above achieves three major objectives:

- *Efficient, class-based differentiation of resource allocation and monitoring for dynamic workloads* : Regulate and monitor kernel resource allocation by classes which are defined by the privileged user and not only in terms of tasks. The differentiation should work in the face of relatively rapid changes in class membership and over roughly the same time intervals at which process-centric regulation currently works.
- *Low overhead for non-users*: Users disinterested in CKRM's functionality should see minimum overhead even if CKRM support is compiled into the kernel. Signs of user disinterest include omitting to mount rcfs or not defining any classes. Even for users, CKRM tries to keep overheads proportional to the features used.
- *Flexibility and extensibility through minimization of cross-component dependencies*: Classification engines should be independent of classtypes and optional, classtypes should be independent of each other and so should resource controllers, even within the same classtype. This goal is achieved through object-oriented interfaces between components. Minimizing dependencies allows kernel developers to selectively include components based on their perception of its utility, performance and stability. It also permits alternative versions of the components to be used depending on the

target environment e.g. embedded Linux distributions could have a different set of taskclass resource controllers (or even classtypes) than server-oriented distributions.

Classification Engine

The Classification Engine (*CE*) is an optional component that enables CKRM to automatically classify kernel objects within the context of its classtype. Since the CE is optional and since we want to main flexibility in its implementation, functionality and deployment, it is supplied as a dynamically loadable module. The CE interacts with CKRM core as follows. The CKRM core defines a set of ckrm events that constitute a point during execution where a kernel object could potentially change its class. A classtype can register a callback at any of these events. As an example, the task class hooks the fork, exec, exit, setuid, setgid calls where as the socket class hooks the listen and accept calls. In these callbacks the classtypes typically invoke the optional CE to obtain a new class. If no CE is registered or the CE does not determine a class, the object remains in its current class, otherwise the object is moved to the new class and the corresponding resource managers of that class's type are informed about the switch.

For every classtype the CE wants to provide automatic classification for, it registers a classification callback with the classtype and the set of events to which the callback is limited to. The task of CE is then to provide a target class for the kernel objects passed in the context of the classtype. For instance, task classes pass only the task, while socket classes pass the socket kernel object as well as the task object. Though the implementation of the classification engine is completely independent of CKRM, the CKRM project provides a default classification, called RBCE, that is based on classification rules. Rules consist of a set of rule terms and a target class. A rule term specifies one particular kernel object attribute, a comparision operator (=,<,>,!) and a value expression. To speed up the classification process we maintain state with tasks about which rules and rule terms have been examined for a particular task and only reexamine those terms that are indicated by the event.

RBCE provides rules based on task parameters ((pid, gid, uid, executable) and socket information (IP info). The rules in conjunction with the defined classes constitute a site policy for workload managment and is dynamically changable (See user interface section) into the RBCE. Hence, this approach ensures the separation of policy and enforcement.

To facilitate the interaction with WLMs to provide event monitoring and tracing, the CE can also register a notification callback with any classtype, that is called when a kernel object is assigned to a new class. Similar so the classification callback, the notification callback can be limited to a set of ckrm events. This facility is utilized in the Resource Monitoring section described below.

Resource Scheduling

Providing differentiated service to resources such as CPU time, page frames or resident set size (RSS), disk I/O bandwdith and number of accepted TCP connections in the Linux kernel is the primary design objective of CKRM. However, the CKRM project also has a design objective to provide class-based differentiation through small extensions/modifications to the existing task-centric schedulers in the kernel. This not only facilitates the integration of CKRM into the mainline Linux kernel, but also permits it to continue to take advantage of the advances in the underlying schedulers provided by the Linux kernel development community.

The following sections describe the CKRM's resource controllers developed for an earlier version of CKRM. In the earlier version, classes did not form a hierarchy (all classes in the system were peers of each other) and there was a single notion of resource share (compared to the upper and lower bounds in the current version). As such, all the resource controllers are now being redeveloped. However, since large parts of the design are expected to remain the same and the controllers showed promising results, it is instructive to describe the resource controllers.

CPU Controller

The CPU scheduler decides which task to run when and for how long. The Linux cpu scheduler in 2.6, a.k.a the O(1) scheduler, is a multi-queue scheduler that assigns a scheduler instance and an associated runqueue to each cpu. The 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 tasks at their respective priority level. Linux distinguishes 140 priority levels, 100 for realtime tasks and 40 for timeshared tasks to map the -20..19 task nice levels. The maximum time slice a regular task executes is a linear mapped function of its priority into [10..200] msecs. After executing its timeslice, a task moves 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. A task is defined as interactive, if its recent average sleep time exceeds a threshold. Interactive tasks remain in the active queue. Every 250msecs and on idle processing the runqueues are rebalanced based on runqueue length to ensure that a similar level of progress is made on each cpu.

In the class fair-share queueing extension (CFQ) we assign per-cpu runqueues for each class. A hierachical scheduling scheme is utilized, that selects classes for execution based on their consumed cycles and selects tasks within their classes' local runqueue based on the existing O(1) scheduling semantics. This makes performance isolation possible since tasks belonging to different classes are now maintained in different run queues. At every scheduling decision we first select the next class to run locally and within that class the best task to run using the existing task selection algorithm. We contain the code changes to the `get_next_task()` functionality. Local class objects (runqueue) maintain a local effective class priority as

```
    ecp(C) = R *  ∑ cycles(C)/share(C) + top_prio
```

where `cycles(C)` represents the amount of CPU time received by class C, `R` is a configurable proportionment value and `top_prio` (negative values present higher priorities) represents the highest priority of the tasks within class C on a particular cpu. For class selection, the class with `min(ecp(C))` is chosen. Similar to the per-class task runqueues, we maintain on each cpu a runqueue of local runnable classes based on their `ecp(C)` of classes with tasks to run on this cpu. This class runqueue is maintained as a sliding window, since `ecp(C)` is a monotonically increasing function. When a class is reactivated, i.e. a task in this class is reactivated and it is the only task locally for that class, its `ecp(C)` is forced into the sliding window to ensure that dormant classes will catch up only on its recently unused share and not on all its share since going dormant.

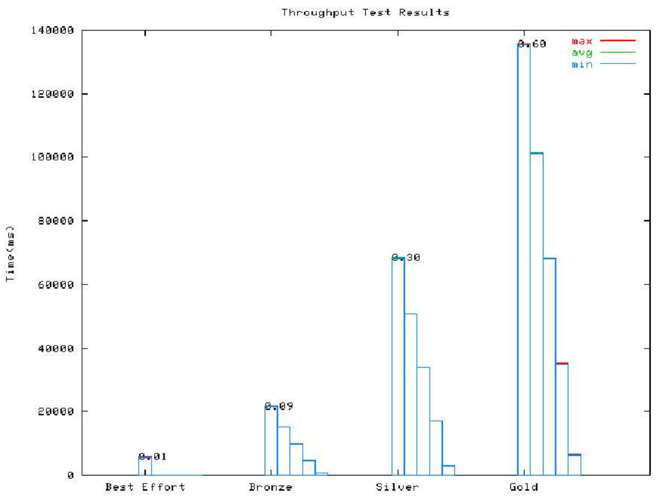

Figure 2: Uni-processor throughput QoS

By combining both the progress (`cycles(C)/share(C)`) and the urgency (`top_prio`) together, our class fair scheduler achieves accurate proportional sharing while preserving good interactive job support. This is illustrated in Figure 2 for the uni-processor case. Here 4 classes (gold,silver,bronze,best effort) with shares of 60,30,9,1 are defined and each populated with 15 cpu bound jobs (3 for for each nice value of (-20,-10,0,10,19)). First, over a 30 minute run, each class obtained its assigned share. Within each class the relative fairness is also maintained in that cycles obtained per nice level is in proportion to its priority and tasks within the same nice level obtain the same cylces.

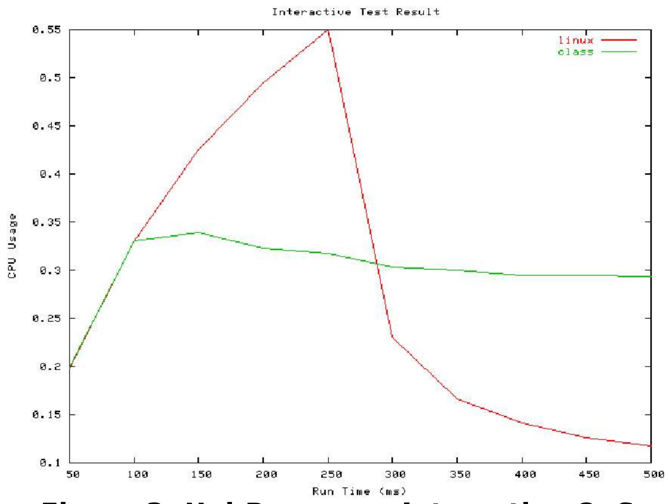

Figure 3: Uni-Processor Interactive QoS Share Maintainence

Figure 3 provides QoS for an interactive task, where the silver class is populated with a single interactive job that sleeps for 200 msec and then executes for a finite amount of time (50-500msecs). The default cycles received under the O(1) linux scheduler varies significantly and degrades when the job becomes more cpu-bound, while the CFQ extension ensures that the cycles received are close to the desired share of 30%.

Figure 4 presents the wait time of a said interactive task and shows a significantly smoother and more gracefully degrading wait functions.

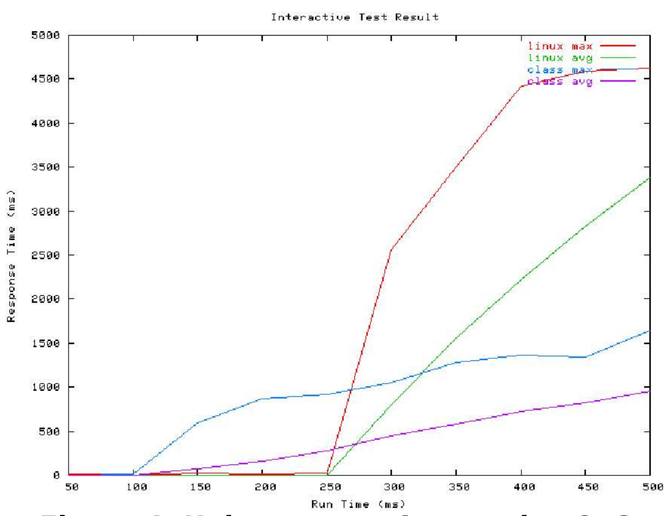

Figure 4: Uni-processor Interactive QoS Response Time

Finally, the CFQ extension scale at the same pace as the O(1) scheduler introducing a fixed overhead of aprox. .4 microseconds on average where the two task case consumes 2 microseconds per context-switch and the 256 task case consumes 5.5 microseconds.

So far we have demonstrated the efficacy of CFQ on the uni-processor case. In the multi-processor case we perform load-balancing based on a the concept of class pressure. The epoch time of a set of tasks is defined as the time required to execute each task for it designated time slice. We then approximate the epoch time of each class on each cpu by

$$EP(C,cpu) = \sum ts(t) * ia(t)$$

where `ts(t)` is the priority based time slice of a task and `ia(t)` is a measure of its interactive level (0..1). The class pressure on a particular cpu is then defined as

$$P(C,cpu) = EP(C,cpu) / cpu\_usage(C,cpu)$$

with `cpu_usage(C,cpu)` dentifying the actual recently consumed cycles by class C on said cpu. Load balancing is then performed by balancing the pressures of classes across different cpus. It is noteworthy that we do not try to attempt to give each class its share on every cpu.

For a sufficiently loaded system, we observed that the overall class shares are still maintained, and the QoS that tasks receive within a group same class, same nice level) are still reasonable close (< 5%).

Memory Controller

Differentiated usage of physical memory has traditionally received very little attention in operating systems resource management research. Waldspurger [7] describes a method of proportionally sharing of physical memory between virtual machines created by VMWare's ESX Server. While some of the ideas described there such as the taxation of unused shares can be applied in our context, it addresses a fundamentally different problem of sharing memory across multiple OS kernels. In [8], a cooperative mechanism is described, where kernel hints allow applications to regulate their memory consumption.

One of the major contributions of CKRM is providing simple and effective control over the resident set size (RSS) of a class. We first describe the existing mechanism for controlling physical memory in Linux 2.6.

A process' RSS, the number of physical page frames allocated to a process, is critical in ensuring the progress of the corresponding application. An RSS which is significantly lower than its average working set size can cause a process to spend much of its time in page faults which also affects overall system performance due to the pressure on the I/O subsystem. Most VMMs, including Linux, regulate memory usage only at system-wide granularity. Per-process RSS limits are available but are rarely used due to the difficulty in estimating working set sizes.

The default 2.6 Linux VMM controls system memory usage primarily through page reclamation. The physical memory of a system is divided into three zones - DMA (< 16 MB), Normal (16-896MB) and High (> 896 MB). The page descriptors for page frames belonging to each zone are kept in three lists - active, inactive and free. Recently accessed pages are kept in the active list while older pages which are candidates for reclamation are kept in the inactive list. The free list stores page frames ready for allocation. When system memory falls below a threshold, the kernel's page swapper scans the inactive list looking for page frames to reclaim. Clean pages containing unmodified data are directly reclaimed into the free list after unmapping them from the appropriate address spaces. Modified pages are scheduled for writeback to their backing store (either a filesystem file or the swap file) and added to the free list after the writeback completes.

When CKRM memory control is enabled, each class has an associated share denoting the fraction of available physical memory to which it is entitled. The CKRM memory controller follows two design principles to control the average physical memory consumed by a class with minimal impact on overall system performance. First, it only enforces class shares when overall system memory is low. The threshold for share enforcement is the same as that for page reclamation in a default system. This ensures the overhead of unnecessary regulation and provides some insulation of system performance from share settings that do not reflect the average working set size of a class. Second, CKRM only enforces shares by modifying the page reclamation mechanism and does not alter page frame allocation. This allows a smooth and gradual control over average page frame usage.

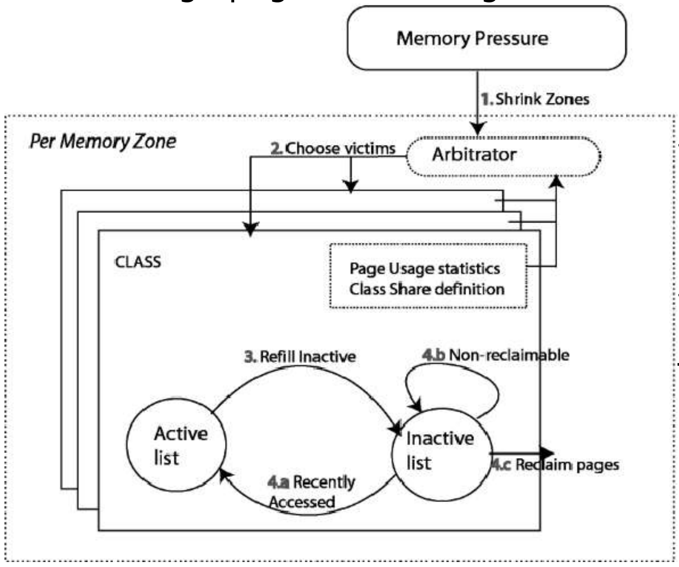

Figure 5: Class Aware Page Reclaimation

Figure 5 shows the logical representation of page reclamation in CKRM. Each page frame descriptor in the system is now associated with a class. Statistics are maintained on the number of page frames allocated to a class. Consequently, at any given time, each class is either below, at or above its share of physical memory. The page frames of a zone are logically divided into per-class active, inactive and free lists. When free memory is low, the VMM chooses a zone (as before) and then uses an arbitrator function to choose a victim class within the zone from amongst the over-share classes. The inactive pages of the victim class are evaluated to select victim page frames using the same criteria as used in the default VMM. Victim pages are reclaimed either directly or following a writeback as before. If system memory is still below the threshold, the next victim class is chosen until sufficient page frames have been reclaimed. The statistics on per-class usage are updated for use by CKRM's monitoring facilities and the arbitrator.

The implementation of the above design does not use physically separate per-class lists of pages so that pages can continue to be arranged in order of their age. Instead, the same per-zone active, inactive and free lists are used with modifications to the scanning functions to make them consider the class of a page before its age, state etc. first. Pages belonging to under-share classes are skipped during a scan. Doing a logical rather than physical separation of per-class page lists preserves the benefits of selecting victim pages in system-wide LRU order even amongst the over-share classes.

The CKRM memory controller was evaluated on a 2.4GHz Pentium 4 desktop running Redhat Linux 9 and the 2.5.69 Linux kernel running using a simple microbenchmark which allocates a variable number of pages and proceeds to access them using a user-specified access pattern. Two classes, A and B, were created, each running one instance of the microbenchmark. Each microbenchmark was configured to consume 200MB of memory but Class A's access frequency was set at twice that of Class B to cause some variance in application progress. Against a total demand of 400 MB, the available system memory was limited to 352 MB to ensure that regulation would be performed.

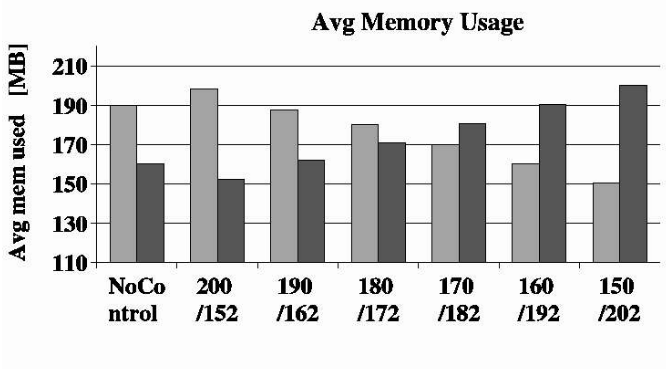

Figure 6: Differentiated memory usage by two instances of memory microbenchmark

Figure 6 shows the actual memory usage of classes (A,B) as their shares were modified. The first bar shows the memory usage under the default VMM (without CKRM). Class A is seen to consume more memory than B due to its higher access frequency which keeps more of its pages in the active list (and hence unreclaimable). With CKRM enabled, as the class shares are varied from (200M,152M) through (150M, 202M), the usage of Class A decreases while that of B increases. As can be seen, the usage of the classes closely tracks the share settings. During the experiments, it took only a few seconds for the usage values to stabilize to the values shown after a share setting had been changed.

I/O Controller

The CKRM I/O controller aims at providing class-based control over I/O bandwidth of block devices, most commonly a disk. In Linux, I/O requests to a block device are typically serviced through a single logical queue which is managed by an I/O scheduler. CKRM modifies the I/O scheduler to enforce class I/O bandwidth shares. We briefly describe the existing Linux I/O schedulers followed by the specifics of CKRM I/O control.

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, most commonly by merging and sorting, before being made available to the low-level device drivers (henceforth only called device drivers). Device drivers consume the transformed requests and forward them, using device specificprotocols, to the device controllers which perform the I/O. The mainline Linux 2.6 kernel provides multiple I/O schedulers such as *anticipatory, deadline, linus* and *noop* with *anticipatory* [9] being the default. The recently proposed Complete Fair Queuing I/O scheduler [10] provides share-based control over per-disk bandwidth for each process. CKRM's I/O controller is a variant of CFQ which provides per-class control.

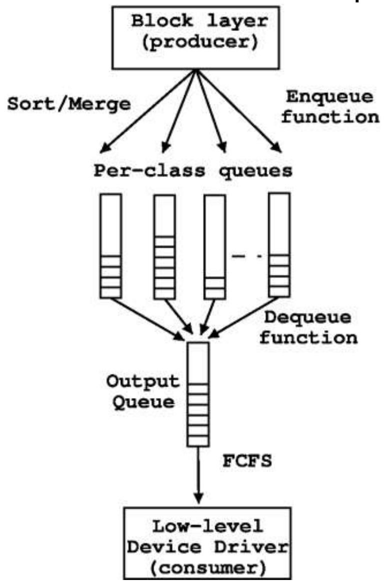

Figure 7: CKRM I/O Scheduler

Figure 7 shows the structure of CKRM I/O control. The single logical I/O request queue of a block device is physically represented by several queues: one input queue per class and one common output or dispatch queue. Processes submit I/O requests through the Virtual File System (VFS) using either synchronous or asynchronous I/O system calls. Each of these requests is associated with the class of the submitting process and gets queued into the class-specific queue of the block device. Each queue has an associated weight which is proportional to its assigned share. When the device driver is ready to service the next request, the I/O scheduler moves requests from the class queues to the common dispatch queue in proportion of their weighted size. The device driver picks the next request off the dispatch queue in FIFO order and submits it to the device.

By giving primacy to class weight while transferring requests to the driver, CKRM runs the risk of decreasing disk utilization due to potentially higher seek overheads. To amortize this cost, requests are transferred in batches from each class queue. As each batch is already sorted in order of expected seek time during the input phase, seek overhead is somewhat reduced. The batch size is a parameter that can be varied with 4 being the default.

Inbound Network Controller

Various OS implementations offer well established QoS infrastructure for outbound bandwidth management, policy-based routing and Diffserv [11]. Linux in particular, has an elaborate infrastructure for traffic control [12] that consists of queuing 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. The traffic control mechanims can be used invarious places where bandwidth is the primary resource to control.

Due to the above features, Linux is widely used for routers, gateways, edge servers; in other words, in situtations where network bandwidth is the primary resource to differentiate among classes.

When it comes to endservers networking, QoS has not received as much attention since QoS is primarily governed by the systems resources such as memory, CPU and I/O and less by network bandwidth. When we consider end-to-end service quality, we should require networking QoS in the end servers as exemplified in the fair share admission control mechanism proposed in this section.

We present a simple change to the existing TCP accept mechanism to provide differentiated service across priority classes. Recent work in this area has introduced the concept of prioritized accept queues [13] and accept queue schedulers using adaptive proportional shares to self-managed web [14].

In a typical TCP connection, the client initiates a request to connect to a server. This connection request is queued in a global accept queue belonging to the socket associated with the server's port. The server process picks up the next queued connection request and services it. In effect, the incoming connections to a particular TCP socket are serialized and handled in FIFO order. When the incoming connection request load is higher than the level that can be handled by the server requests have to wait in the accept queue until the next can be picked up.

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.

The accept queue implements a weighted fair scheduler such that the rate of acceptance from a particular accept queue is proportional to the weight of the queue. In the basic priority accept queue design proposed earlier in [15], starvation of certain priority classes was a possibility as the accepting process picked up connection requests in the order of descending priority.

The efficacy of the proportional accept queue mechanism is demonstrated by an experiment. We used Netfilter[16] to `MARK` options to characterize traffic into two priority classes with respective weights of 3:1. The server process utilises a configurable number of threads to service the requests. The results are shown in Figure 8. When the load is low and there are service threads available no differentiation takes place and all requests are processed as they arrive. Under higher load, requests are queued in the accept queue with class 1 receiving a proportionally higher service rate than class 2. The expriment was repeated, maintaining a constant inbound connection request rate. The proportions of the two classes were then switched to see the service rate for the two classes reverse as seen in Figure 9.

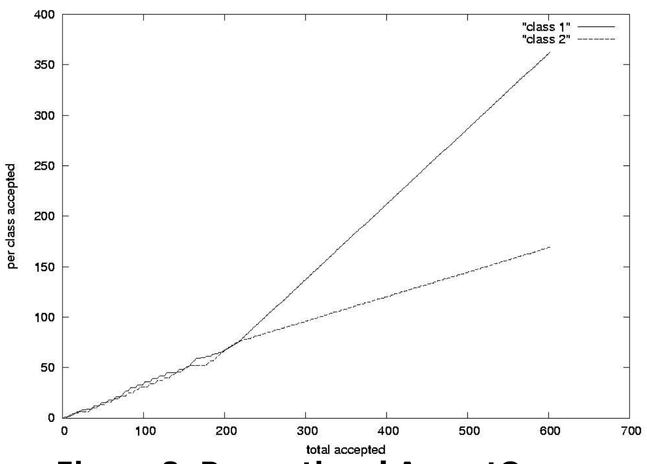

Figure 8: Proportional AcceptQueue Results

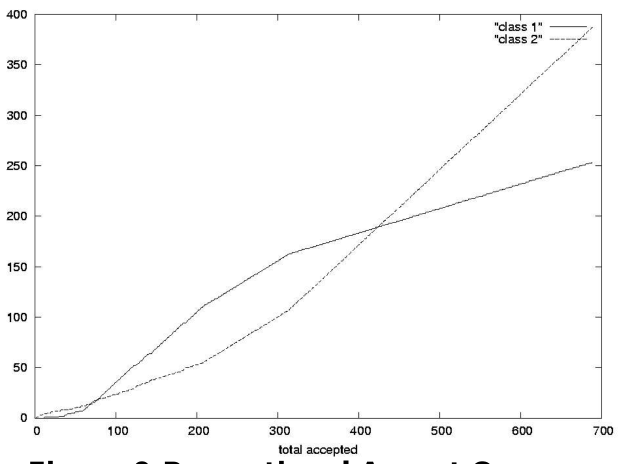

Figure 9:Proportional Accept Queue Results under change

Resource Monitoring

We now describe the monitoring infrastructure. Strictly speaking, the per-class monitoring components are part of CKRM while the per-process components are not. However, we shall describe them together as they both can be utilized by goal-based WLMs. Furthermore, they are bundled with the classification engine and utilize the CE's notification callback to obtain classification events.

The monitoring infrastructure illustrated in Figure 10 is based on the following design principles:

1. **Event-driven**: Every significant event in the kernel that affectsthe state of a task is recorded and reported back to the *state-agent*. The events of importance are aperiodic such as process fork, exit and reclassification as well as periodic events such as sampling. Commands sent by the *state-agent* are also treated as events by the kernel module.
2. **Communication Channel**: A single logical communication channel is maintained between the *state-agent* and the kernel module and is used for transferring all commands and data. Most of the data flow is from the kernel to user space in the form of records resulting from events.
3. **Minimal Kernel State**: The design minimizes the additional per-process state that needs to be maintained within the kernel. Most of the state needed for high level control purposes is kept within the state agent and updated through the records sent by the kernel.

The state-agent, which can also be integrated within a WLM, maintains state on each existing and exited task in the system and provides it to the WLM. Since the operating system does not retain the state of exited processes, the state-agent must maintain it for future consumption by the WLM. The state-agent communicates with a kernel module through a single bidirectional communication chan-nel, receiving updates to the process state in the form of records and occasionally sending com-mands. Events in the kernel such as process fork, exit, reclassify (resulting from change in any process attribute such as gid, pid) cause records to be generated through functions provided by

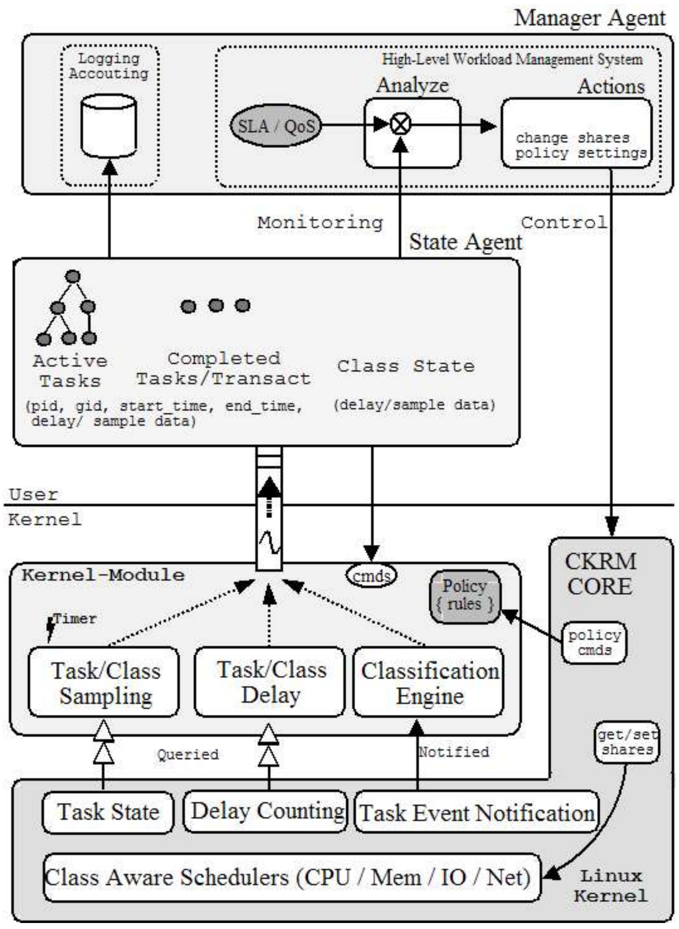

Figure 10: Monitoring and Control Approach

the kernel module.

Server tasks can assist the WLM by informing it about the phase in which they are operating (each phase corresponds to a workload). Such tasks invoke CKRM to set a tag associated with their `task_struct` in the kernel. CKRM uses this event to reclassify the task and also records the event (to be transmitted to the WLM through the state-agent). Other kernel events that might cause a task to be reclassified (such as the exec and setuid system calls etc.) are also noted by CKRM and passed to the WLM through the state-agent.

In addition, CKRM performs periodic sampling of each task's state in the kernel to determine the resource it is waiting on (if any), its resource consumption so far and the class to which it belongs. The sample information is transmitted to the state-agent. The WLM can correlate the information with the tag setting to statistically determine the resource consumption and delays of both server and dedicated processes executing a workload.

Sampling is done through a kernel module function that is invoked by a self-restarting kernel timer. Commands sent by the state-agent cause appropriate functions in the kernel module to execute and also return data in the form of records. The kernel components are kept simple and only minimal additional state has to be maintained in the kernel. In particular, the kernel does not have to maintain extra state about exited processes which introduces problems with PID reusage, memory management to name a few. Instead, relevant task information is replicated in user space, is by definition received in the correct time order (see below) and can be kept around until the WLM has consumed the information. Furthermore, the semantics of a reclassification in the kernel, which identifies a new phase in a server process, does not have to be introduced into the kernel space.

The following small changes are required to the linux kernel to track system delays. The `struct delay_info` is added to the `task_struct`. Delay_info contains 32-bit variables to store cpu delay, cpu using, io delay and memory io delay. The counters provide micro second accuracy. The current cpu scheduler records timestamps whenever a task i) becomes runnable and is entered into a runqueue and ii) when a context switch occurs from one task to another. We use these same timestamps to get per-task cpu wait and cpu using times recorded respectively. I/O delays are measured by the difference of timestamps taken when a task blocks waiting for I/O to complete and when it returns. All I/O is normally attributed to the blocking task. Pagefault delays, however, are treated as special I/O delays. On entrance to and exit from the page fault handler the task is marked or unmarked as being in a memory path using flags in `task_struct`. If during the I/O delay, this flag is set, the I/O delay is counted as a memory delay instead of as a pure I/O delay. The per-task delay information is accessible through the file `/proc/<pid>/delay`. Similarly, each class contains a `delay_info` structure.

In contrast to the precise accounting of delays, sampling examines the state of tasks at fixed interval. In particular, we sample at fixed intervals (≈1sec) the entire set of tasks in the system and increment per task counters that are integrated into the task private structure attached by the classification engine that builds the core of the kernel module. We increment counters if a task is running, waiting to run, performing I/O or handles a pagefault I/O.

Task data (sampled and/or precise) is requested by and sent to the state-agent in coarser intervals. We can send data in continuous aggregate mode or in delta mode, i.e. only if task data has changed do we send a new data record and then reset the local counters.

The task transition events are sent at the time they occur. We distinguish the fork, exit and reclassification events as records. At each reclassification (which could potentially be the end of a phase) we transmit the sample and delay data and reset them locally.

As a *communication channel* we utilize the linux relayfs pseudo filesystem, a highly efficient mechanism to share data between kernel and user space. The user accesses the shared buffers, called channels, as files, while the kernel writes to them using buffer reservations and memory read/write operations. The content and structure of the buffer is determined by the kernel and user client. Currently the communication channel is self pacing. The underlying relayfs channel buffer will dynamically resize upto a maximum size. If for any reason the relayfs buffer overflows, record sending will automatically stop, an indication is sent and the state-agent will have to drain the channel and request a full state dump from the kernel.

We have measured the data rate during a standard kernel build, which creates a significant amount of task events (fork,exec,exits). For a 2-CPU system with 2 seconds sample collection we observed a data rate of 8KB/second and a total of 190 records/sec, well within a limit that can be processed without creating significant overhead in the system.

User Interface

In the Linux kernel development community, filesystems have become very popular as user interfaces to kernel functionality, going well beyond the traditional use for disk-based persistent storage. The Linux kernel's object-oriented Virtual File System (VFS) makes it easy to implement a custom filesystem. Common file operations like open, close, read and write map naturally to initalization, shutdown, kernel-to-user and user-to-kernel communication. For CKRM, the tree structured namespace of a filesystem offers the additional benefit of an intuitive representation of the class hierarchy. Hence CKRM uses the Resource Control Filesystem (RCFS) as its user interface.

The first-level directories in RCFS contain the roots of subtrees associated with classtypes build or loaded into the kernel (socket_class and task_class currently) and the classification engine (ce). Within the classtype subtrees, directories represent classes. Users can create new classes by creating a directory as long as they have the proper access rights.

Within the task_class directory, each directory represents a taskclass. /rcfs/task_class, the root of the task_class classtype, represents the default taskclass which is always present when CKRM is enabled in the kernel. Each task_class directory contains a set of virtual files that are created automatically when the directory is created. Each virtual file has a specific function as follows:

1. `members`: Reading it gives the names of the tasks in the taskclass.
2. `config`: To get/set any configuration parameters specific to the taskclass.
3. `target`: Writing a task's pid to this file causes the task to be moved to the taskclass, overriding any automatic classification that may have been done by a classification engine.
4. `shares`: Writing to this file sets new lower and upper bounds of the resource shares for the taskclass for each resource controller. Reading the file returns the current shares. The controller name is specified on a write which makes it possible to set the values for controllers independent of each other.
5. `stats`: Reading the file returns the statistics maintained for the taskclass by each resource controller in the system. Writing to the file (specifying the controller) resets the stats for that controller.

The socket_class directory is somewhat similar. Directories under /rcfs/socket_class/ represent listen classes and have the same magic files as task_classes. Whereas task_classes use the pid to identify the class member, socket_classes, which group listening sockets, use ip address + port name to identify their members. Within each listen class, there are automatically created directories, one for each accept queue class. The accept queue directories, numbered 1 through 7, have their own shares and stats virtual files similar to those for task_classes.

The /rcfs/ce directory is the user interface to the optional classification engine. It contains the following virtual files and directory:

1. reclassify: writing a pid or ipadress+port to the file causes the corresponding task or listen socket to be put back under the control of the classification engine. On subsequent significant kernel events, the ce will attempt to reclassify the task/socket to a new taskclass/socketclass if the task/sockets attributes have changed.
2. state: to set/get the state (active or inactive) of the classification engine. To allow a new policy to be loaded atomically, CE's can be set to inactive before loading a set of rules and activated thereafter.
3. Rules: The directory allows privileged users to create files with each file representing one rule. Reading the files, permitted for all, gives the classiication policy which is currently active. The ordering of rules in a policy is determined either by creation time of the corresponding file or by an explicitly specified order number within the file. The rule files contain

rule terms consisting of attribute-value pairs and a target class. e.g. The rule

```
gid=10, cmd = bash, target = /rcfs/task_class/A
```

indicates that tasks with gid=10 and running the bash program (shell) should get reclassified to task_class A.

Future work

The consolidation of increasingly dynamic workloads on large server platforms has considerably increased the complexity of systems management. To address this, goal-oriented workload managers are being proposed which seek to automate low-level system administration requiring human intervention only for defining high level policies that reflect business goals.

In this paper, we argue that goal-oriented WLMs require support from the operating system kernel for class-based differentiated service where a class is a dynamic policy-driven grouping of OS processes. We introduce a framework, called class-based kernel resource management, 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 provide the design of a proportional share scheduler using incremental modifications to the corresponding existing schedulers. The framework is implemented in the Linux 2.6 kernel and is publicly available at http://ckrm.sf.net. The performance evaluation of the schedulers and the flexibility of the framework demonstrate that CKRM is a viable approach for building autonomic operating systems.

With the basic framework in place, there are several directions for future work. We intend to explore the use of CKRM to manage virtual resources such as limits on open files, processes, logins, locks etc. These are also traditionally managed at a per-process or per-user granularity but are candidates for class-based management. The individual schedulers need to be refined. The CPU scheduler's load balancing design will be revisited. The memory controllers treatment of shared memory pages can be significantly improved. We are also examining alternate designs for memory controller that use explicit per-class page lists.

Perhaps the most important direction for future work is the interactions of the resource schedulers and the impact of these interactions on the shares specified. CPU, memory, I/O and network share settings are interdependent in OS specific ways. Exporting these interdependencies in the form of constraints on share specifications might considerably increase the efficacy of feedback control loops in the workload manager.

References

[1] *AIX 5L Workload Manager*
IBM Corp., http://www.redbooks.ibm.com/redbooks/SG245977.html

[2] *Solaris Resource Manager*
Sun Microsystems Inc, http://www.sun.com/software/resourcemgr/wp-srm

[3] *HP-UX Workload Manager*
Hewlett Packard Inc. http://h30081.www3.hp.com/products/wlm

[4] *Adaptive algorithms for Managing a Distributed Data Processing Workload.* J. Aman and C.K. Eilert and D. Emmes and P. Yocom and D. Dillenberger, IBM Systems Journal, volume 36(2), 1997.

[5] *Resource Containers: A new facility for resource management in server systems,* G. Banga, P.Druschel, J.C. Mogul. Operating Systems Design and Implementation, pages 45-58, 1999.

[6] *Cluster reserves: a mechanism for resource management in cluster-based network servers.* M. Aron, P. Druschel, W.Zwenepoel. Measurement and Modeling of Computer Systems, pg. 90-101,2000.

[7] *Memory resource management in VMware ESX Server*.
Carl A. Waldspurger. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI), December 2002*.

[8] *Application-assisted physical memory management.*
P.Druschel S.S.Iyer, J.Navarro. *http://www.cs.rice.edu/~ssiyer/r/mem/*

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

[10] The Continuing Development of I/O Scheduling
Jonathan Corbet. *http://lwn.net/Articles/22526*

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

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

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

[14] An Observation-based Approach Towards Self-Managing Web Servers.
P. Pradhan, R. Tewari, S. Sahu, A. Chandra, and P. Shenoy.
In Intl. Workshop on Quality of Service (IWQoS}, 2002.

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

[16] *Netfilter: Firewalling, NAT, and packet mangling for Linux 2.4.*
J. Kadlecsik, H. Welte, J. Morris, M. Boucher, and R. Russel.
http://www.netfilter.org