I/O Resource Management through System Call Scheduling[*]

Silviu S. Craciunas Christoph M. Kirsch Harald Röck

Department of Computer Sciences
University of Salzburg, Austria

firstname.lastname@cs.uni-salzburg.at

ABSTRACT

A principal challenge in operating system design is controlling system throughput and responsiveness while maximizing resource utilization. Unlike previous attempts in kernel resource management, which often involve non-trivial changes in kernel subsystems, we focus on the kernel's edge. System calls are usually the default mechanism for user processes to get access to operating system services. System calls can therefore be used to control throughput and responsiveness and thus also affect resource utilization directly. We propose a simple, non-intrusive kernel-space mechanism for explicit, per-process system call scheduling already at kernel entry in order to control the time and rate at which system calls are executed, and, as a result, the per-process utilization of the involved resources. We have developed a high-performance Linux 2.6 kernel patch with SMP support that implements system call scheduling for network- and disk-related I/O calls with policies that resemble traffic shaping in network routers. Our experiments show that already simple and easy-to-use policies provide effective I/O-related process isolation with low overhead, and reduce thrashing in certain overload scenarios. While system call scheduling may still not be able to outperform resource management systems that use specifically tuned kernel subsystems, our experiments indicate that it may sufficiently support relevant soft real-time applications yet using a vastly simpler and more generic approach.

1. INTRODUCTION

Fast mobile computers and wireless networks are key enablers of many fascinating new computer applications such as VoIP and video streaming. Even classical business applications such as online trading have evolved into ever more powerful software tools. These applications have in common that they usually not only require throughput-oriented performance but also timely execution. In other words, application software is becoming increasingly real-time. Many operating systems including Linux, however, have been designed with a focus on best-effort concepts and only in retrospect enhanced for better real-time performance. Improvements in real-time performance typically require lower kernel latency and some form of prioritization. The problem of reducing Linux kernel latency has recently received a lot of attention, which already resulted in many patches that made it into the official Linux kernel. For example, the RCU patch reduces latency by providing lockless read access to shared data structures [16]. Some patches also improve the efficiency of prioritization by implementing, for example, priority inheritance. However, managing workload for better real-time performance also requires associating resources as different as, for example, networks and disks, with the processes that generate the load. In previous, more integrated attempts such as CKRM [14] the issue has been addressed by patching multiple kernel subsystems at once, which, however, results in complex and difficult to maintain code.

In this paper, we propose system call scheduling as a simple and generic concept to control the amount of workload generated by user processes before the workload even arises thus providing administrators with a high degree of control over how resources are used. A system call scheduler determines explicitly, at kernel entry, when the system call will actually be executed according to a scheduling algorithm and user-provided limits. If necessary, the system call's execution is delayed by putting the invoking process to sleep. The main difference to traditional kernel resource management is that we control process behavior and resource utilization at the kernel's edge, which results in easy-to-maintain code and thus avoids complex kernel-wide modifications.

In principle, system call scheduling may be applied to any system call. As an example application, we have implemented system call scheduling for network and disk calls as a Linux 2.6 kernel patch [4]. Other calls may also be scheduled explicitly, which is, however, future work. As scheduling algorithm for network and disk calls, we use a technique that we call process shaping, which resembles traffic shaping in network routers but applied to system calls. Process shaping enforces bandwidth limits on system calls by evenly spreading out system call execution on the timeline. Our

[*]Supported by a 2007 IBM Faculty Award, the EU ArtistDesign Network of Excellence on Embedded Systems Design, and the Austrian Science Fund No. P18913-N15.

implementation supports per-process as well as per-thread limits on any number of network and disk devices. For system calls accessing devices with potentially large differences in gross traffic and net traffic (which actually comes from the device) such as most disk drives with caches, we have also implemented so-called resource shaping, which limits the bandwidth of system calls only when the actual device (and not the cache) is exercised. Thus a process that logically reads from disk but actually receives data from the disk cache may run at full speed when using resource shaping. In contrast, process shaping strictly enforces all limits independently of any cache effects.

Similar to traffic shaping of network data streams, process shaping considers processes (and optionally threads) completely independently of each other. Given a process with bandwidth limits on system calls, process shaping enforces those limits by merely looking at a system-wide real-time clock but not by considering any other processes. Nevertheless, when running multiple, independently shaped processes whose summed-up limits are less or equal than the capacity of the involved device, there may be fewer races for device access resulting in less thrashing than without any process shaping. In other words, process shaping just like traffic shaping rests on a probabilistic form of coordination rather than explicit coordination, with two important implications: multicore and multiprocessor scalability of shaping processes that do not collectively use a single bandwidth limit is only limited (in our implementation and not just in theory) by the scalability of the underlying kernel architecture, and process shaping may immediately benefit from improvements in kernel latency and I/O subsystem performance. Process shaping is therefore a technology that is complementary to more isolated I/O scheduling in kernel subsystems.

In our experiments, we demonstrate on a dual-core, dual-processor server machine with two Gigabit Ethernet devices and a single disk: (1) low overhead of system call scheduling in micro benchmarks, (2) effective control of network throughput with process shaping enabled in the network calls of a network-bound web server and a partially disk-bound web server, (3) limited control of separate read/write disk throughput and effective control of separate read/read disk throughput with resource shaping enabled in the disk calls of two disk-bound processes, and (4) effective, indirect control of the soft-real-time performance of a video-streaming server in the presence of either several disk-bound processes and alternatively a network-bound web server and a partially disk-bound web server with process and resource shaping enabled in their network and disk calls, respectively.

The web servers' workload in (4) is specifically designed to show that only shaping the per-process combination of network- and disk-related I/O enables flawless video streaming in a difficult overload scenario. However, we feel that even on purely network- or disk-bound workloads (for which one could also use separate network and disk traffic shapers) process and resource shaping have merit because of their easy-to-maintain implementation and simple usage. Our kernel patch [4] consists of around 2000 lines of code compared to, for example, the much larger CKRM system.

The contributions of this paper are: (1) the idea of system call scheduling, (2) the idea of using traffic shaping technology in network- and disk-related system call scheduling (process and resource shaping), (3) a Linux 2.6 kernel patch as well as a user-space monitoring tool [4], and (4) experiments with our implementation applied to unmodified state-of-the-art web and streaming servers on a dualcore, dualprocessor server machine.

In Section 2, we describe the kernel-level mechanisms and implementations of system call scheduling, and process and resource shaping. Section 3 briefly describes the user interface of the kernel patch and an htop-based monitoring tool. Related work is discussed in Section 4 and experiments are presented in Section 5. We present future work in Section 6 and conclude the paper in Section 7.

2. SYSTEM CALL SCHEDULING

In this section, we describe the features of our patch and its underlying scheduler concept as well as its kernel-level data structure extensions and scheduler implementation.

2.1 Features

Our patch supports shaping and limiting network and disk traffic of groups of processes and threads in so-called classes. Each class is assigned bandwidth limits on network and disk devices, which are enforced by delaying the execution of network- and disk-related system calls. When a process invokes such a system call and the limits of its class are exceeded, the process is put to sleep for a certain amount of time depending on the bandwidth limits of its class.

All classes are disjoint, i.e., a process is a member of at most one class. If a process is not a member of any class, it is called unmanaged and thus not controlled by our mechanism. Currently, the user-space API of our implementation does not allow an unmanaged process to join an existing class. However, by design of the kernel patch, this feature may be made available in a future version. In order to become managed, a process can only generate a new class of which it will be the only member. An existing class can then be extended by a member of the class spawning a new process. By default, a new child process of a managed process joins the class of its parent. If requested by the parent, the child process may also create and join a new class. In case a new class is generated during a fork, the new class inherits the properties of its parent class. The properties of a class include bandwidth limits and its currently unused bandwidth for each network and disk device in the system. Each class also has a unique identifier, which is not inherited. An example of a class with several members is a multi-threaded web server. All threads of the server are in the same class, and limits are enforced process-wide. Examples of a class with different processes are applications that use multiple processes such as version 1 of the Apache web server. To enforce bandwidth limits on a web server with multiple processes all of its processes should be a member of the same class.

A class may also be associated with the following two modes of operation. For disk-related system calls, bandwidth limits may be enforced on the actual amount of data transferred from the disk, i.e., ignoring any data that comes from the page cache (`RESOURCE` mode). If this mode is not used, our mechanism accounts for all read and written data, regardless of how much data is actually read from the page cache. The second mode distinguishes read and write operations (`READ-WRITE` mode). In this mode, different limits for read and write operations may be assigned to the resources of a class and enforced independently.

2.2 Scheduler Concept

We adopted the well-known token bucket algorithm, which is extensively used in the context of network traffic shaping [17]. The algorithm controls the amount of data processed in a given amount of time, and thus deals with issues of enforcing policies, congestion management, quality of service (QoS), and fairness.

The token bucket algorithm allows traffic to go through only if there are virtual tokens available. Given the presence or absence of tokens in a virtual token bucket, where tokens usually represent units of bytes, the algorithm allows data to be transferred, or introduces a delay, respectively. Tokens are generated at a fixed rate and placed in the token bucket, which has a limited size. If a token arrives when the bucket is full, the token is discarded.

The effect of the token bucket algorithm is similar to the widely used leaky bucket algorithm in that it shapes bursty traffic into a steady stream, except that it also allows for infrequent bursts to go through at full speed. We choose the token bucket algorithm instead of the leaky bucket algorithm because it is simpler and also more efficient to implement. In contrast to the leaky bucket algorithm, the token bucket algorithm does not need any timers or work queues, a simple token counter is sufficient.

Our patch applies the token bucket algorithm to schedule the execution of network- and disk-related system calls. If the token bucket corresponding to the involved resource contains the required number of tokens, data is transferred, the system call is completed, and the number of consumed tokens is subtracted from the bucket. If the bucket does not contain enough tokens, the process is put to sleep for the needed amount of time, which is calculated based on the rate at which tokens are generated and the currently available number of tokens.

2.3 Process Descriptor Extensions

In this section, we describe the new data structures introduced by our patch as well as our modifications to existing kernel data structures. We have implemented the patch against the Linux 2.6.20 kernel. The patch currently supports SMP systems of two architectures, namely `i386` and

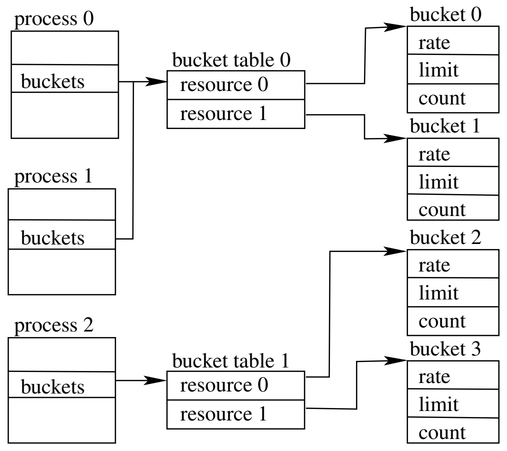


Figure 1: Processes and bucket tables

`x86_64`. Both versions are almost equivalent and differ only in the code modifications of the kernel's exception handler for system calls.

A class is represented by a table of token buckets. A process is a member of a class if its `task_struct` contains a pointer to the bucket table of the class. In addition, we added a flag to the `thread_info` structure called `SHAPED`, which indicates whether a process is managed or unmanaged.

Similar to a file descriptor table, which is shared among the threads of a process, all processes of a class use the same bucket table. For instance, in Figure 1, process 0 and process 1 are in the same class because they share a common bucket table. Process 2 does not share its bucket table with any other process and therefore belongs to another class. In our implementation, it is not necessary to maintain more information about the relationship between classes and processes than a reference to a shared bucket table.

For each resource in the system, the bucket table maintains a token bucket. In Figure 1, bucket table 0 uses buckets 0 and 1, and bucket table 1 uses buckets 2 and 3. A token bucket essentially consists of three elements:

- rate: the number of tokens to generate per second.
- limit: the maximum number of tokens (bucket size).
- count: the number of tokens currently available.

In the actual implementation, however, we need some additional fields to store a lock, two time stamps, and a reference counter. The lock is required to synchronize the access of multiple processes to the same bucket (`lock` field). We use a reader/writer lock, but in principle, may also be able to use the RCU mechanism [16]. The two time stamps keep track of when new tokens were last added to the bucket (`last_update` field), and when new tokens are expected to arrive (`next_update` field). See the next section for details of how these time stamps are used. To ensure correct deallocation of buckets (and bucket tables) we use reference counters (`refcount` field).

2.4 Scheduler Implementation

In this section, we present the implementation details of our system call scheduling mechanism. We first describe the hook code at kernel entry for scheduling system calls and then discuss the entry and exit routines of the scheduling mechanism, which are invoked before and after the execution of each relevant system call handler. The entry routine determines if and how long a process is to be delayed and the number of tokens that need to be generated. The exit routine consumes tokens for the actual amount of data transferred.

Our patch introduces two hooks into the kernel's exception handler for system calls. The first hook is installed right before the exception handler invokes the adequate system call handler. The hook code checks the new `SHAPED` flag in the `thread_info` structure. If the flag is not set, the process continues regularly. If the flag is set, the process enters a filter routine, which first saves the system call arguments on the stack for later examination and state regeneration. Then, the filter determines whether this system call is marked for scheduling by looking at the system call number. If the system call is not marked, the filter returns immediately. Otherwise, the filter proceeds to the entry routine of the system call scheduler. When the entry routine returns to the hook code, the system call arguments are restored from the stack and the system call handler is invoked.

The second hook is installed right after the invocation of the system call handler, and works similarly to the first hook. In contrast to the first hook, however, the second hook stores the return value of the system call handler on the stack and calls the exit routine of the system call scheduler.

The entry routine first searches for the resource that the system call uses (network or disk) by examining the mode field of the provided file descriptor's inode using `S_ISSOCK`. If `S_ISSOCK` returns true, the file descriptor is a socket and the correct network device is found by calling `ip_dev_find` using the local address of the socket. If `S_ISSOCK` returns false, the file descriptor refers to a regular file on disk. In both cases, the identifier of the device is cached in a new field of `struct file` to speed up the lookup operation for future requests. Assuming mode fields and local addresses do not change, no extra locking during the lookup operation is required (we increment the reference counter of the `struct file` in order to prevent its deallocation).

After finding the correct device the associated token bucket is examined to see if tokens are available. We define a token to be the equivalent of one page in size, i.e., for one token a process can read or write one page. Moreover, to determine whether the process is allowed to continue, we have to check not only the token count but also if any of the other processes in the same class are already waiting for tokens. For this purpose, the `next_update` field in the token bucket structure holds the time instant when the last process waiting for tokens continues to execute. If `next_update` is in the past, no processes are waiting. If it is in the future, at least one process of this class is already waiting for tokens.

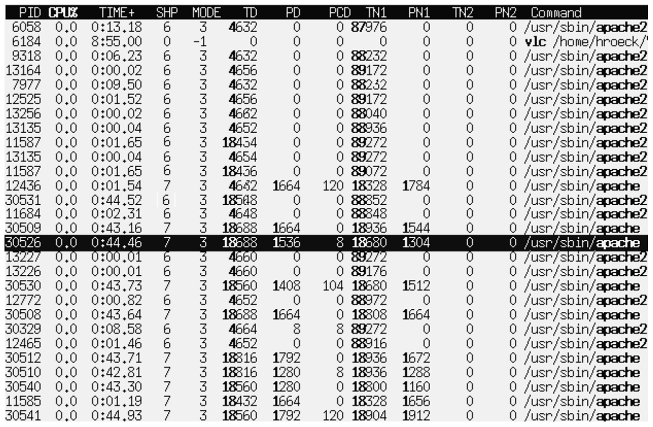
In case no processes are already waiting, the token count is checked and if positive, the current process returns from the entry function to the exception handler. Note that the process continues as long as some tokens are available, which may result in a negative token count if the process' I/O operation turns out to consume more tokens than the current token count. In order to compensate for a negative token count, the next process of this class accessing the same resource is delayed for the time required to generate the negative number of tokens and the tokens needed for its current I/O operation. Avoiding negative token counts is possible but introduces additional overhead and usually prevents the actual throughput from reaching the bandwidth limits.

In case processes are already waiting for tokens or the token count is zero or negative, the current process is delayed. If no processes are already waiting and the token count is zero or negative, the current process is the first process of this class that runs out of tokens. From now on all processes of this class requesting tokens from this bucket are delayed until tokens are available again. The first process sets `next_update` to the current time plus the time required to generate enough tokens for its I/O operation to complete. Later processes in the same class that access the same resource before the time instant in `next_update`, increment `next_update` by the time required to generate enough tokens for their respective I/O operations to complete (unless there are tokens available again). The required time is calculated using the system call argument that specifies the amount of data that the system call attempts to transfer divided by the token rate of the bucket. Processes are then delayed until the time instant in `next_update`. In other words, a FIFO-style waiting queue is implicitly built by using cumulative waiting times but without explicitly using queues.

To avoid excessive sleep times, we enforce an upper limit on the sleep time by scaling down system calls involving large amounts of data, whenever possible. Note that at the time instant when a system call is invoked, we cannot determine the actual amount of data it transfers, only the desired amount, since it is possible that a process transfers less data than it originally asked for in the system call arguments. Therefore, our cumulative-waiting-times method may be slightly inaccurate. However, our experiments show that this effect is negligible.

When a process wakes up, tokens are added to the bucket according to the given rate and the elapsed time since the time specified in `last_update`, which is then set to the current time before giving back control to the exception handler.

When the system call handler returns, the second hook in the exception handler invokes the exit routine, which first calculates the number of tokens to consume based on the actual amount of transferred data. It makes sure that at least one token is consumed, even if the transferred data was less than one page in size. This guarantees that a process cannot accumulate transferred data without consuming tokens. Thereafter, the calculated amount of tokens is subtracted from the token count of the involved resource.



```
  PID CPU%   TIME+  SHP MODE    TD     PD    PCD   TN1    PN1   TN2   PN2 Command
 6058  0.0 0:13.18    6    3  4632      0      0 87976      0     0     0 /usr/sbin/apache2
 6184  0.0 8:55.00    0   -1     0      0      0     0      0     0     0 vlc /home/hroeck/'
 9318  0.0 0:06.23    6    3  4632      0      0 88232      0     0     0 /usr/sbin/apache2
13164  0.0 0:00.02    6    3  4656      0      0 89172      0     0     0 /usr/sbin/apache2
 7977  0.0 0:09.50    6    3  4632      0      0 88232      0     0     0 /usr/sbin/apache2
12525  0.0 0:01.52    6    3  4656      0      0 89172      0     0     0 /usr/sbin/apache2
13256  0.0 0:00.02    6    3  4662      0      0 88040      0     0     0 /usr/sbin/apache2
13135  0.0 0:00.04    6    3  4652      0      0 88936      0     0     0 /usr/sbin/apache2
11587  0.0 0:01.65    6    3 18434      0      0 89272      0     0     0 /usr/sbin/apache2
13135  0.0 0:00.04    6    3  4654      0      0 89272      0     0     0 /usr/sbin/apache2
11587  0.0 0:01.65    6    3 18436      0      0 89072      0     0     0 /usr/sbin/apache2
12436  0.0 0:01.54    7    3  4642   1664    120 18328   1784     0     0 /usr/sbin/apache
30531  0.0 0:44.52    6    3 18548      0      0 88852      0     0     0 /usr/sbin/apache2
11684  0.0 0:02.31    6    3  4648      0      0 88848      0     0     0 /usr/sbin/apache2
30509  0.0 0:43.16    7    3 18688   1664      0 18936   1544     0     0 /usr/sbin/apache
30526  0.0 0:44.46    7    3 18688   1536      8 18680   1304     0     0 /usr/sbin/apache
13227  0.0 0:00.01    6    3  4660      0      0 89272      0     0     0 /usr/sbin/apache2
13226  0.0 0:00.01    6    3  4660      0      0 89176      0     0     0 /usr/sbin/apache2
30530  0.0 0:43.73    7    3 18560   1408    104 18680   1512     0     0 /usr/sbin/apache
12772  0.0 0:00.82    6    3  4652      0      0 88972      0     0     0 /usr/sbin/apache2
30508  0.0 0:43.64    7    3 18688   1664      0 18808   1664     0     0 /usr/sbin/apache
30329  0.0 0:08.58    6    3  4664      8      8 89272      0     0     0 /usr/sbin/apache2
12465  0.0 0:01.46    6    3  4652      0      0 88916      0     0     0 /usr/sbin/apache2
30512  0.0 0:43.71    7    3 18816   1792      0 18936   1672     0     0 /usr/sbin/apache
30510  0.0 0:42.81    7    3 18816   1280      8 18936   1288     0     0 /usr/sbin/apache
30540  0.0 0:43.30    7    3 18560   1280      0 18800   1160     0     0 /usr/sbin/apache
11585  0.0 0:01.19    7    3 18432   1664      0 18328   1656     0     0 /usr/sbin/apache
30541  0.0 0:44.93    7    3 18560   1792    120 18904   1912     0     0 /usr/sbin/apache
```

Figure 2: The user-space monitor htap

3. USER INTERFACE

In order to control and monitor system call scheduling from user space, we have implemented a new system call to activate and deactivate shaping of a process. The new system call `shape` accepts as input parameters a process identifier `pid`, an enable/disable flag `on_off`, and a `mode` bitmap, which can be zero for default behavior, or a combination of the `RESOURCE` and `READ-WRITE` modes of operation. An additional flag in the bitmap indicates if child processes are put into a new class, or else join the parent's class. Scheduling parameters and statistics are available to user-space applications via new entries in the `proc` file system.

Based on htop [13], we have also implemented a monitor application called htap, which controls our scheduling mechanism from user space through the `proc` file system. Essentially, htap provides additional functionality that, besides the standard process-specific data and statistics, displays also shaping-related information such as network- and disk-related throughput as well as disk cache throughput. Figure 2 shows a screen shot of htap as our mechanism is managing two different Apache web servers. Both standard and super users can use htap to monitor the system. A super user may also use htap to activate shaping of a process, set modes, and change the token rate and bucket size, and even the current token count for each process and resource.

4. RELATED WORK

In this section, we relate our general idea of system call scheduling and, in particular, the idea of process shaping to previous work. System call scheduling is inspired by the notion of threading by appointment (TAP) [7]. Instead of invoking a system call at any time, threading by appointment requires a thread to make an appointment with a scheduler for each system call the thread would like to invoke. The execution of a system call is then delayed until the time of that appointment. The original TAP implementation suffered from poor performance [15] but has since then evolved into the kernel patch [4] described here.

We relate the idea of process shaping to previous work on three levels of abstraction. Besides using system call scheduling, any I/O traffic caused by system calls can be scheduled, in principle at least, (1) directly by an I/O subsystem on the level of the traffic itself, or (2) indirectly by a process scheduler on the level of the processes that generate the traffic, or (3) explicitly by an application-specific scheduler on the level of the application that generates the traffic.

The Linux netfilter system (similar to any traffic shaping network router) is an example of the first category restricted to network I/O. It has been part of the Linux kernel since version 2.3, and replaced its predecessors ipchains of Linux 2.2 and ipfwadm of Linux 2.0 [18]. The netfilter system is very powerful and can be used to implement firewalls, network translation, transparent proxies, and to limit network traffic.

There also exist I/O schedulers for block devices that fall into the same category. In [20], a hierarchy of token bucket filters is used on top of a standard disk I/O scheduler to reserve some disk bandwidth for soft-real-time applications. However, the approach only applies to disk I/O and has only been evaluated using synthetic workloads.

A more general example of the first category is the "Class-based Linux Kernel Resource Management" (CKRM) [14]. CKRM is more general than our approach as it also provides control over other resources, besides I/O resources, such as CPU time, memory pages, and other virtual resources. However, our approach performs I/O resource management at kernel entry through system call scheduling whereas CKRM directly manages I/O resources in their respective kernel subsystems. CKRM has not been integrated in the official Linux kernel because it is considered too intrusive and hard to maintain. The problem is that CKRM touches several different subsystems and introduces callback hooks, which have to be maintained by different developers. In contrast, our solution introduces only one hook into the general system call handler. The kernel subsystems, e.g., disk scheduler and network layer, are not affected.

In general, the advantage of approaches in the first category is that I/O traffic is scheduled at a finer granularity, e.g., network packets rather than system calls. With system call scheduling this level of granularity is not achieved. Additionally, enforcing bandwidth limits in the respective subsystems may be done event-based using work queues, and hence, may reduce the number of context switches and cache flushes. The disadvantage of approaches in the first category, however, is code complexity and the missing relationship among I/O requests across different I/O subsystems, and thus the lack of a global view of the flow of traffic, e.g., from disk to network devices and back.

A traditional CPU scheduler that uses process priorities, a fixed time quantum, or real-time criteria such as deadlines is an example of the second category. Such schedulers can, however, only indirectly control access to shared resources other than the CPU, e.g., by changing process priorities. Nevertheless, system call scheduling is closest related to the second category but uses, in addition to a CPU scheduler, an explicit mechanism, i.e., the system call scheduler, to control the access to I/O resources such as disk and network devices. While the mechanisms of our system call scheduler and of traditional I/O schedulers, e.g., in the Linux kernel, are similar, their scheduling policies and thus goals are different. A traditional I/O scheduler is resource- and performance-oriented, i.e., optimizes individual resource utilization, whereas the system call scheduler is application-oriented, i.e., optimizes higher-level criteria such as system composability. For example, since a web server application may perform sufficiently fast without permanent access to shared resources such as a network device, the system call scheduler may even have the network device regularly idle although the web server has requested access to the device. The network device may then be used by other applications in a more controlled fashion. However, when the system call scheduler decides to grant access to the network device it should of course be utilized in the most efficient way. The concept of our system call scheduler is therefore complementary to the concept of a traditional resource- and performance-oriented I/O scheduler.

Application software such as the Apache web server, which implements throttling [2], is an example of the third category. Another example of the third category is connection scheduling [5], which prioritizes network traffic per connection but does not implement a mechanism to limit bandwidth usage. In principle, application software may also implement whatever policy our mechanism supports. The difference is that our mechanism makes these features available to all applications. Moreover, the super user can enforce the limits on single processes or groups of processes without any support by the applications.

Resource containers [1] are an approach that uses mechanisms from all three categories and are in principle a more general version of our token buckets. They allow accounting process activities for their resource usage on the level of I/O traffic (first category) but also incorporate CPU cycles making it necessary to assign and implement scheduling algorithms according to given container policies (second category). Furthermore, they require cooperation from the applications since processes have to determine which resource containers to use (third category). Our approach does not require cooperation but, so far, manual configuration of bandwidth limits. However, resource containers like CKRM entail intrusive modifications of the CPU scheduler implementation and the I/O subsystems.

Resource management and QoS support for new rather than existing operating systems has been a wide area with extensive research. The Scout project [11] provides QoS through I/O-driven scheduling by introducing path abstractions that represent the flow of data from an I/O source to an I/O sink. This project is centered more on achieving high and predictable performance of network connections whereas we focus on system-wide I/O traffic. The Nemesis project [8] is based on the idea of multiplexing shared resources between applications while keeping the policy at the user level to allow for more control. The Eclipse operating system [3] introduces a new abstraction called reservation domains, which provide resource QoS in overload scenarios. Nemesis and Eclipse are similar, both providing predictable performance via allocation of CPU and disk I/O to domains. However, Nemesis is based upon a radically different OS structure, for which reimplementation of most applications and device drivers is needed. Eclipse and Nemesis are comparable to our approach in goal but are both operating systems written from scratch whereas we offer a non-intrusive patch to an existing and widely used operating system, which does not require modifications of application programs and device drivers.

5. EXPERIMENTS

We demonstrate on a dual-core, dual-processor server machine (Section 5.1) that system call scheduling incurs low overhead (Section 5.2), process shaping enables effective control of network throughput (Section 5.3), and resource shaping enables effective control of disk throughput (Section 5.4) and the soft-real-time performance of a video-streaming server in difficult overload scenarios (Section 5.5).

5.1 Setup

All experiments were run on a server machine with two 2GHz dualcore AMD64 CPUs, 4GB of memory, two Gigabit Ethernet cards, and an 160GB SATA disk, running a Linux 2.6.20 kernel with our patch applied. Only one of the two Ethernet cards was used in the experiments.

In order to generate network load, we connected two client machines via Gigabit Ethernet to the server machine and emulated web server clients using httperf [12] and autobench [10]. The client hardware was the same as the server hardware. The client software, however, ran on a standard Linux 2.6.17 kernel, which was the default stable kernel version already installed on these machines. In total, we had two httperf processes per client machine generate load to avoid any bottlenecks on the clients.

In the experiments that involved network call shaping, we ran two unmodified Apache servers version 1.3.34 (Apache1) and 2.0.55 (Apache2). Apache1 uses one process per connection whereas Apache2 uses one thread per connection within a single, multi-threaded process. Our kernel patch supports both models.

In the soft real-time experiments, we used the streaming software VLC [19], which supports various audio and video formats. VLC implements both a multimedia-streaming server and a client player. On the server machine, the VLC server was configured to stream a DVD image stored on the local disk to a remote laptop, which ran a VLC client to play the stream. The VLC server needed between 800KB/s and 1MB/s disk throughput to read the DVD image in real time.

	shaping disabled		shaping enabled	
best	214 ticks	107ns	577 ticks	288ns
average	286 ticks	143ns	689 ticks	344ns
worst	4526 ticks	2263ns	5520 ticks	2760ns

Table 1: Overhead of system call scheduling

5.2 Overhead Experiment

Summary: The overhead of system call scheduling in a single read call is about 400 clock ticks on the 2GHz server machine (Table 1). The results were measured with the `rdtsc` (read time stamp counter) instruction.

Details: Table 1 depicts the best, average, and worst duration of a single read call during runs with shaping enabled and disabled. This benchmark reads 100,000 times the first page of the same file. The read call goes through the complete scheduling mechanism, i.e., determining the device, checking for available tokens, and consuming tokens. The bandwidth limit in this benchmark was set higher than the maximum bandwidth of the device to ensure that the process does not have to wait for tokens. The benchmark shows that the overhead of the scheduling mechanism is about 150 to 200 nanoseconds on the 2GHz server machine.

5.3 Network Experiment

Summary: We demonstrate effective control of network throughput with process shaping enabled in the network calls of a network-bound web server and a partially disk-bound web server.

Details: In this experiment, Apache1 and Apache2 ran on the server machine and were hit by two different request patterns. Apache1 served clients that requested twelve different 1GB files. Half of these files were serviced from the cache and the other half from the disk. At the same time, Apache2 served 1400 requests per second for a single cached 94KB file. Figure 3 depicts the throughput of the two web servers running concurrently on the server machine.

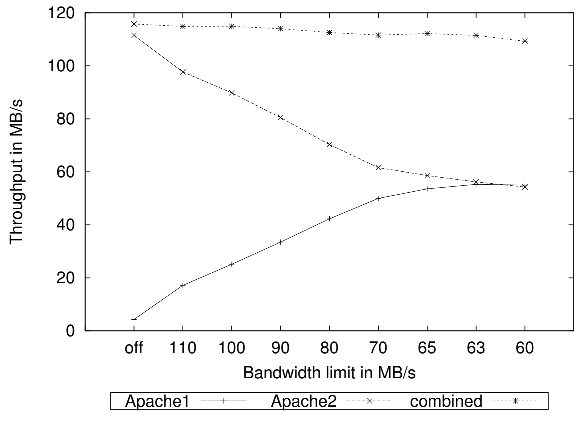


Figure 3: Network call shaping

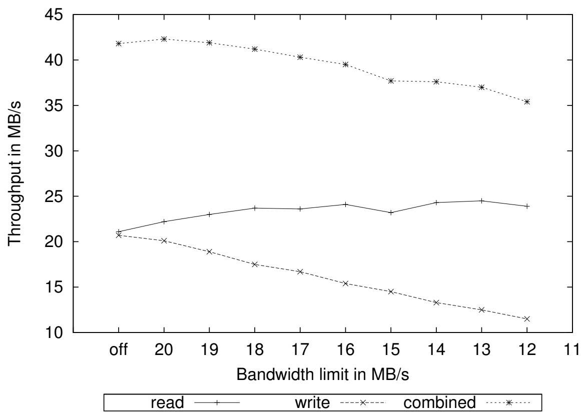


Figure 4: read/write disk call shaping

When shaping is not activated, Apache2 dominates the network device and starves the Apache1 server. Gradually reducing the bandwidth limit of Apache2 gives Apache1 a bigger fraction of the network device's bandwidth. Its throughput increases inversely to the throughput of the Apache2 server. The combined throughput, however, stays at the same level of around 115MB/s while the individual throughput of both servers converges to around 57MB/s each, i.e., each server gets about 50% of the available bandwidth. Similar to this experiment, bandwidth limits may be enforced on larger numbers of processes giving each a different percentage of the network device's bandwidth.

5.4 Disk Experiments

Summary: We demonstrate in two experiments limited control of separate read/write disk throughput and effective control of read/read disk throughput with resource shaping enabled in the disk calls of two disk-bound processes.

Details: In both experiments, we ran two processes on the server machine accessing the local disk. In the read/write experiment, one of the processes wrote to the disk while the other process read from the disk. In the read/read experiment both processes read a different file from disk. Both experiments were repeated three times using different files. The resulting average throughput is depicted in Figures 4 and 5.

When one process writes to the disk and its bandwidth limit is decreased, the reading process' throughput increases only slightly and, as a result, the combined throughput decreases slowly (Figure 4). If, however, both processes read from the disk and the first process' bandwidth limit is decreased, the combined throughput is maintained since the second process' throughput increases accordingly (Figure 5). These experiments indicate that read/read disk call scheduling is more effective across a larger range of bandwidth limits than read/write disk call scheduling. The loss of bandwidth in the read/write case may be an artifact of the underlying disk scheduler in Linux, which trades-off throughput for lower latency and better fairness when read and write operations run concurrently.

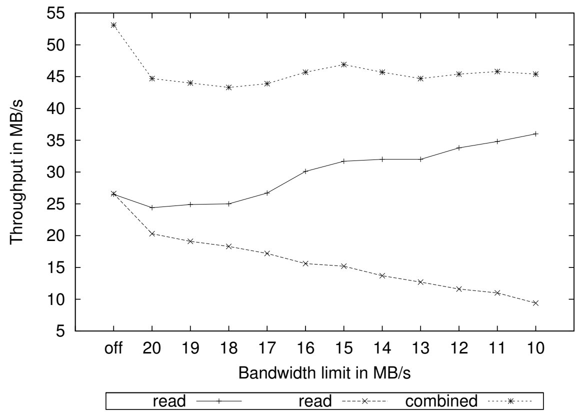


Figure 5: read/read disk call shaping

5.5 Streaming Experiments

We demonstrate in two experiments effective, indirect control of the soft-real-time performance of a video-streaming server in the presence of either several disk-bound processes and alternatively a network-bound web server and a partially disk-bound web server with process and resource shaping enabled in their network and disk calls, respectively. The video-streaming server was a VLC server streaming the content of a DVD image stored on a local disk over a network connection to a VLC client.

In the first experiment, we ran, in addition to the VLC server, several `cat` processes on the server machine to generate additional disk load with resource shaping enabled in the disk calls of the `cat` processes. Note that we control VLC's streaming performance indirectly by limiting the I/O bandwidth of the competing processes. In other words, to reserve bandwidth for the VLC server, we set bandwidth limits on the concurrently running applications that generate load on the resources shared with the VLC server.

The purpose of the first experiment is to compare VLC's streaming performance when using either the anticipatory disk I/O scheduler [6, 9] or the complete fair queueing (CFQ) disk I/O scheduler [9]. Operating systems use disk I/O schedulers to improve the throughput of block devices by minimizing the number of seeks. In the Linux kernel, every block device maintains a queue of requests that represent pending I/O operations. The request queues are managed by I/O schedulers that can be changed at runtime. The simplest I/O scheduler is the deadline scheduler, which assigns each request an expiration time in addition to merging and sorting the requests by physical location. Requests are serviced according to their occurrences in the request queue, unless a request expires in which case the request is serviced as soon as possible [9]. The anticipatory scheduler, which is based on the deadline scheduler, introduces a short delay to wait for additional requests of the process that issued the last request before servicing the next queued request. The assumption is that requests from the same process are sequential most of the time. The benefit of sequential access plus the waiting time outweigh the seek time of the disk. The CFQ scheduler is another disk I/O scheduler in Linux, which uses one queue per process in which requests are inserted on a per-process basis. Within such a queue, requests are merged and sorted. The queues are then serviced in a round-robin manner to guarantee fairness among all processes [9]. Additionally, the latest CFQ scheduler applies similar waiting times as the anticipatory scheduler. Therefore, CFQ's performance is comparable to the anticipatory scheduler's performance.

In the second experiment, we ran, in addition to the VLC server, Apache1 and Apache2 on the server machine to generate additional network and disk load with process and resource shaping enabled in their network and disk calls, respectively. The purpose of this experiment is to demonstrate that process and resource shaping can be used to share network and disk bandwidth effectively among response-oriented streaming and throughput-oriented web applications. The key challenge is to control network and disk traffic simultaneously.

In addition to the usual throughput measurements, we quantify VLC's streaming performance using three subjective categories: (1) 'poor quality' corresponds to poor streaming performance on the client such as the video freezing up or no audio transmission for several seconds, (2) 'good quality' refers to better but not perfect performance such as infrequent error fragments in the picture or noise in the audio signal, and (3) 'perfect quality' is achieved when no errors during video playback are observed.

5.5.1 Disk Call Scheduling

Summary: We compare VLC's streaming performance when using the anticipatory and the CFQ disk I/O schedulers with resource shaping enabled in the disk calls of competing `cat` processes. With the anticipatory scheduler the performance is slightly better than with CFQ.

Details: In this experiment, we ran, in addition to a VLC server, twelve `cat` processes on the server machine reading different 1GB files concurrently with resource shaping enabled in the disk calls of the `cat` processes. Table 2 summarizes the results.

	Anticipatory		CFQ	
Bandwidth limit in MB/s	Network thrghput in MB/s	VLC quality	Network thrghput in MB/s	VLC quality
off	45.2	poor	45.5	poor
25	23.3	poor	23.3	poor
24	22.3	poor	22.3	poor
23	21.5	poor	21.5	poor
22	20.4	good	20.3	good
21	19.5	good	19.2	good
20	18.6	perfect	18.6	good

Table 2: Disk call shaping

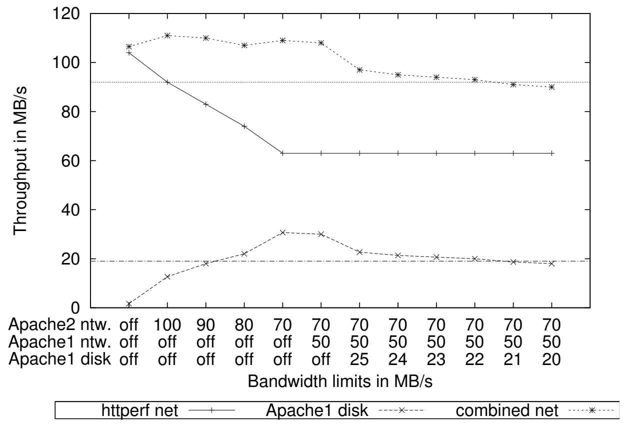


Figure 6: Disk and network call shaping

With shaping disabled, the `cat` processes achieve a throughput of 45.2MB/s independently of which I/O scheduler is used. The quality of VLC's streaming performance, however, is poor and even results in crashes in the client. With shaping enabled, gradually decreasing the total disk bandwidth limit of the `cat` processes from 25MB/s down to 23MB/s effectively reduces their disk throughput but does not improve VLC's streaming performance. Only at 22MB/s the performance improves subjectively to good quality, still with some minor imperfections such as noise in the audio stream or small delays in the video stream. The anticipatory scheduler improves VLC's streaming performance to perfect quality with a total disk bandwidth limit set to 20MB/s. With the CFQ scheduler, the performance does not reach perfect quality because of the nature of the CFQ scheduling strategy, which maintains fairness across all processes at the expense of the VLC server's timeliness.

5.5.2 Disk and Network Call Scheduling

Summary: We demonstrate that process and resource shaping can be used to share network and disk bandwidth effectively among a VLC server, a network-bound web server, and a partially disk-bound web server in a difficult overload scenario.

Details: In this experiment, we ran, in addition to a VLC server, Apache1 and Apache2 on the server machine to generate additional network and disk load with process and resource shaping enabled in their network and disk calls, respectively. Four `httperf` clients of the Apache2 server generated additional network load by creating a total of 700 connections per second and two requests per connection for a single cached 94KB file. Additional disk load was generated through eighteen `wget` clients of the Apache1 server creating two-third actual disk load and one-third disk cache load on the server. The traffic coming from the disk (and not the cache) in the Apache1 server was controlled using resource rather than process shaping. The results are shown in Figure 6 with the exact numbers listed in Table 3.

Bandwidth limits in MB/s			Results		
Apache2	Apache1		httperf	wget	
network	network	disk	thrghput in MB/s	thrghput in MB/s	VLC quality
off	off	off	104	2.5	poor
100	off	off	92	19	poor
90	off	off	83	27	poor
80	off	off	74	33	poor
70	off	off	63	46	poor
70	50	off	63	45	poor
70	50	25	63	34	poor
70	50	24	63	32	poor
70	50	23	63	31	good
70	50	22	63	30	good
70	50	21	63	28	good
70	50	20	63	27	perfect

Table 3: Disk and network call shaping

Figure 6 shows the `httperf` net throughput, the Apache1 disk throughput, and the combined `httperf` and `wget` net throughput. The two horizontal lines are subjectively observed disk and network I/O thresholds of 20MB/s and 90MB/s, respectively, at which the VLC server still functions properly. When both the network and the disk throughput approach the respective thresholds VLC's streaming performance improves to good quality. Whenever the bandwidth limits bring the total disk and network throughput strictly below these thresholds the quality is perfect. Note that the total network throughput increases when our shaping mechanism is activated. In Table 3, the first three columns list the bandwidth limits of Apache1 and Apache2. The last three columns show the resulting `httperf` and `wget` net throughput and the quality of VLC's streaming performance. The Apache1 disk throughput is not shown in the table.

With shaping disabled, the quality of VLC's streaming performance is poor. Furthermore, the `wget` clients are starved by the `httperf` clients to an average throughput of 2.5MB/s resulting in a suboptimal combined net throughput of 106.5MB/s measured on the clients. With shaping enabled, gradually decreasing the network bandwidth limit of the Apache2 server, which serves the `httperf` clients, results in a decreasing `httperf` net throughput and an increasing `wget` net throughput with a slightly better combined net throughput of up to 111MB/s. However, VLC's streaming performance does not improve. Note that the increasing `wget` net throughput results in an increasing disk load on the server caused by Apache1. Setting the network bandwidth limit of Apache1 to 50MB/s and gradually decreasing its disk bandwidth limit results in decreasing `wget` net throughput and, as a consequence, decreasing disk load on the server. With a disk bandwidth limit of 23MB/s, VLC's streaming performance improves to good quality. Further reducing the limit to 20MB/s results in perfect quality and a combined `httperf` and `wget` net throughput of 90MB/s.

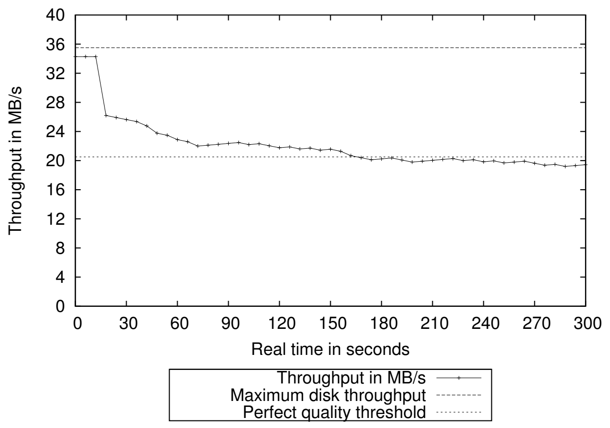


Figure 7: Automatic disk call shaping

6. FUTURE WORK

Our current implementation requires manual configuration of key parameters such as the maximum disk and network bandwidth allocated to a given process. We plan to extend our monitor tool to determine such parameters dynamically and automatically for soft real-time applications such as video streaming. We have already implemented an experimental auto-shaping extension, which detects and handles certain overload scenarios. So far, we have only applied such auto-shaping in disk-bound scenarios, similar to the experiments in 5.5.1, but in principle network-bound scenarios can also be handled.

Preliminary results with VLC show that the auto-shaping extension can automatically determine disk bandwidth limits for optimal streaming quality of a disk-bound VLC client running on the same machine as twelve other `cat` processes generating additional disk load (Figure 7). After 160 seconds into the experiment, the automatically determined bandwidth limit oscillates slightly below the value where both the video quality is optimal and the disk throughput of the `cat` processes is maximal. In our previous experiments (Table 2), the optimal shared disk throughput for optimal streaming was found to be around 20MB/s. In this experiment, the auto-shaping tool finds a disk token rate for the `cat` processes that corresponds to roughly the same value of 20MB/s. Note that, without shaping, the `cat` processes would generate disk traffic close to the devices' maximum throughput and therefore prohibit running other disk-bound soft real-time applications such as VLC. The auto-shaping mechanism is still experimental and requires more work to make it faster and apply it to other types of soft real-time applications.

This paper addresses I/O resource management through system call scheduling. However, processes may not generate all I/O traffic through system calls. For example, a process may also `mmap` a region of a file into its address space. When addresses in the mapped region are touched, the required content of the file is fetched on demand through the page fault handler. System call scheduling obviously cannot shape the resulting I/O traffic. However, we feel that in principle it may be possible to perform page fault scheduling, similarly to system call scheduling, by introducing a hook into the page fault handler. The hook code could delay handling page faults and thus provide a mechanism for some form of page fault shaping.

7. CONCLUSION

We have presented the idea of system call scheduling for the purpose of resource and workload management, and the idea of using traffic-shaping technology in network- and disk-related system call scheduling (process and resource shaping, rather than, for example, traffic shaping in I/O subsystems). We have implemented a Linux 2.6 kernel patch with SMP support that enables process and resource shaping in the kernel, and a user-space monitoring tool to control the shaping parameters. Finally, we have performed a number of experiments with our implementation applied to unmodified state-of-the-art web and streaming servers on a dualcore, dualprocessor server machine. Note that, by principle and current design, multicore and multiprocessor scalability of shaping processes and threads that use independent bandwidth limits is only limited by the scalability of the underlying kernel architecture. Our implementation only requires shaping-related synchronization of processes and threads that collectively use a single bandwidth limit (using reader/writer locks).

Our experiments show that process and resource shaping can effectively control, with low overhead, I/O-related, throughput-oriented process behavior (and, as a consequence, resource utilization) as well as soft real-time performance of disk-bound processes such as video-streaming servers even in difficult overload scenarios. Network and disk traffic may, at least separately, be also controlled by existing extensions of I/O subsystems in the kernel [20]. Nevertheless, we feel that process and resource shaping, and system call scheduling in general, has more potential because, at kernel entry, (1) process behavior such as real-time performance and not only I/O traffic (resource utilization) can be controlled, (2) there is process-related and not only traffic-related information available, (3) other aspects of process behavior such as locking and memory management may be considered in the same system, (4) the implementation is simpler and easier-to-maintain than other approaches that require kernel-wide modifications, and (5) usage is, in our experience, also simpler since it is process- rather than traffic-oriented.

8. REFERENCES

[1] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *Proc. OSDI* (1999).

[2] BARRERA, I. bw_mod: Apache2 module for bandwidth and connection control. http://modules.apache.org/search?id=786.

[3] Bruno, J., Gabber, E., Özden, B., and Silberschatz, A. The Eclipse operating system: providing quality of service via reservation domains. In *Proc. USENIX* (1998).

[4] Craciunas, S., Kirsch, C., and Röck, H. The TAP Project. http://tap.cs.uni-salzburg.at/.

[5] Crovella, M., Frangioso, R., and Harchol-Balter, M. Connection scheduling in web servers. In *Proc. USITS* (1999).

[6] Iyer, S., and Druschel, P. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proc. SOSP* (2001).

[7] Kirsch, C. Threading by appointment. In *Proc. Monterey Workshop* (2004), CRC Press.

[8] Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P. T., Evers, D., Fairbairns, R., and Hyden, E. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications 14*, 7 (1996), 1280–1297.

[9] Love, R. *Linux Kernel Development*, 2nd ed. Novell Press, 2005.

[10] Midgley, J. T. J. autobench - automates the benchmarking of web servers using httperf. http://www.xenoclast.org/autobench/.

[11] Montz, A. B., Mosberger, D., O'Malley, S. W., Peterson, L. L., Proebsting, T. A., and Hartman, J. H. Scout: A communications-oriented operating system (abstract). In *Proc. OSDI* (1994).

[12] Mosberger, D., and Jin, T. httperf - a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev. 26*, 3 (1998), 31–37.

[13] Muhammad, H. htop - an interactive process viewer for Linux. http://htop.sourceforge.net/.

[14] Nagar, S., Franke, H., Kashyap, V., van Riel, R., Seetharaman, C., and Zheng, H. Improving Linux resource control using CKRM. In *Proc. OLS* (2004).

[15] Röck, H. The TAP System: Concurrent programming with threading by appointment. Master's thesis, University of Salzburg, Salzburg, Austria, 2006.

[16] Sarma, D., and McKenney, P. E. Making RCU safe for deep sub-millisecond response real-time applications. In *Proc. USENIX* (2004).

[17] Tanenbaum, A. *Computer Networks*, 3rd ed. Prentice Hall, 2002.

[18] The netfilter.org Project. netfilter: firewalling, NAT, and packet mangling for Linux. http://www.netfilter.org.

[19] The VideoLAN Project. VLC - the cross-platform media player and streaming server. http://www.videolan.org/vlc/.

[20] Wu, J. C., Banachowski, S., and Brandt, S. A. Hierarchical disk sharing for multimedia systems. In *Proc. NOSSDAV* (2005).