Shaping Process Semantics

Silviu S. Crăciunaş Christoph M. Kirsch Harald Röck

Technical Report 2007-01 April 2007

Department of Computer Sciences

Jakob-Haringer-Straße 2
5020 Salzburg
Austria
www.cosy.sbg.ac.at

Technical Report Series

Shaping Process Semantics [⋆]

Silviu S. Crăciunaş   Christoph M. Kirsch   Harald Röck

Department of Computer Sciences
University of Salzburg, Austria
{scraciunas,ck,hroeck}@cs.uni-salzburg.at

1 Analysis

Composition of virtually all concurrent, distributed, real-time, or embedded software running on an operating system is based on some notion of software processes. Operating systems including many real-time operating systems provide widely used abstractions of which software processes are probably still the most successful. A (software) process is typically a sequential or multi-threaded program with an isolated virtual address space and an execution context, which contains state information such as the processor registers and scheduling status as well as timing and I/O descriptors. Interprocess communication (IPC) provides disciplined and well-understood means to overcome process isolation, in many cases, even across machines. In general, processes compute in isolation and invoke system calls to control timing, perform I/O and IPC, and request memory, as indicated by Figure 1. A modern operating system kernel usually handles such calls with dedicated subsystems for interrupt handling, I/O scheduling, IPC handling, and memory management. The systems community has devoted a lot of attention to their semantics and performance with an arguable preference for performance. However, the behavioral requirements of many software applications on timing, I/O, IPC, and memory are often inadequately addressed by the common focus on highest throughput and lowest latency. Since the observable behavior of processes is effectively determined by the invoked system calls, *serving processes as fast as possible*, e.g., by executing system calls using the fastest I/O schedulers available, may only provide an incomplete solution.

For example, delivering a web page in 100ms rather than 10ms does not make much of a difference to a web surfer. However, if it takes 10s just because someone else is currently downloading a large file from the same sever, expectations are clearly not met. A more-dimensional space of permissible behaviors rather than just maximum speed may therefore characterize the situation more appropriately while the range of what is permissible typically decreases with the requirements. As opposed to a web server, a streaming server usually requires access guarantees to a rather confined frequency band on the network. Similar to decreasing web surfing latency, increasing streaming throughput beyond that band has obviously little benefit. The range of permissible behaviors is even smaller for real-time and embedded applications such as digital controllers where the emphasis is typically on predictable, low latency rather than average, high throughput. Interestingly, the networking community has already addressed

[⋆] Supported by Austrian Science Fund grant P18913-N15

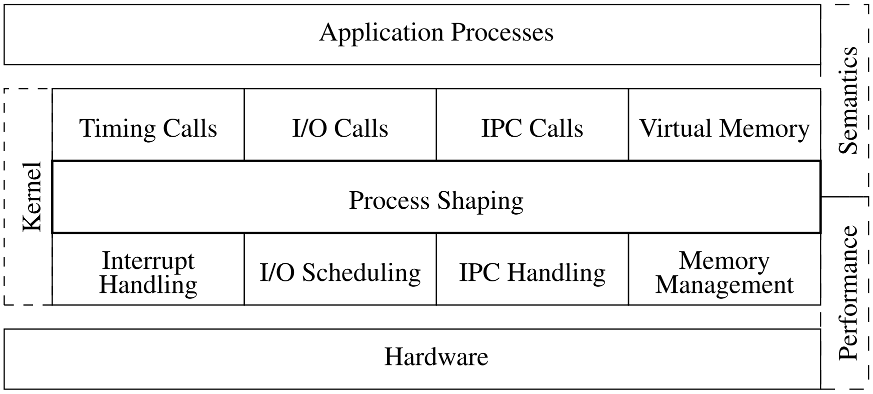


Fig. 1. A process-shaping operating system kernel

such behavioral requirements quite effectively using a variety of queueing techniques and traffic shaping. The idea is to force, i.e., shape traffic into a desired form before handling it. It turns out that such shaped traffic often not only matches application requirements better but can also be handled more efficiently. For example, the success of VoIP applications on nondeterministic but fast networks such as Gigabit Ethernet, which made deterministic networks such as ATM become unexpectedly obsolete, is rooted in the combination of unprecedented transmission speed and adequate prioritization. Higher speed means shorter packet transmission times making prioritization, i.e., traffic shaping, increasingly effective. The result is an inexpensive approximation of quality of service by probabilistic arguments rather than by deterministic but expensive guarantees.

2 Proposal

We propose the notion of *process shaping* to complement, not replace, the notion of serving processes as fast as possible, in analogy to traffic shaping in networks. Process shaping changes the order, times, and possibly the way in which potentially all side effects of processes, e.g., invoked system calls but also memory page faults, are handled before given to any performance-oriented kernel subsystems such as I/O schedulers and memory management, as indicated in Figure 1. Process shaping promotes more disciplined system composition by providing stronger behavioral process semantics than existing operating systems. Process shaping identifies an unrecognized trade off between serving and shaping, and may even result in improved system-wide performance. As a prototype system, we have designed and implemented a user-space threading library called the TAP library [10, 9], which shapes processes by queueing timing-, network-, and disk-related system calls according to a variety of policies. The implementation details and experimental results are discussed below.

With process shaping, we advocate a shift in research attention from performance- to semantics-oriented handling of software processes. Interesting new questions beyond, e.g., traditional scheduling arise such as how to identify dynamically at runtime bandwidth limits and shaping policies for optimal system-wide utilization. Queueing networks [3] may be used in modeling the dynamically changing data flow in operating systems, e.g., from disk to network devices caused by a streaming server. Queueing theory and its real-time extensions [12] may already provide analytical frameworks, in particular, for systems with strict behavioral requirements such as many real-time and embedded systems.

3 Claims

Our proposal rests on semantics- and performance-related claims that need to be verified. With process shaping, the behavioral semantics of processes may have to be expressed in terms of yet to be developed languages that translate into a process-shaping infrastructure. We claim that typical networking terminology such as bandwidth, flow, burst, collision, and so on, in addition to already universal terminology such as frequency and latency, may also be used to describe many process-based application requirements, in particular, in real-time and embedded systems.

Many systems merely guarantee fair but highly non-deterministic access to shared resources such as processors and I/O devices. Even real-time operating systems usually just provide time-bounded access to shared resources, which is, however, still non-deterministic with respect to the order and time at which system calls are actually executed. For example, a given system of application processes may run perfectly fine utilizing, say, fifty percent of the available resources. However, adding just one more, possibly unrelated application process that requires the remaining fifty percent or less will in general change the I/O behavior of the original system, certainly with respect to real time, but also with respect to other, I/O-related properties such as throughput and latency. The lack of stronger notions of process composition is the main reason for the poor predictability, portability, and reusability of many concurrent applications. Even performance-related problems such as thrashing may in some cases be explained in a similar way.

Process shaping is meant to complement, not replace, operating system facilities that operate under the regime of serving processes as fast as possible. We claim that the trends to higher processor speed, also in embedded systems, and to more efficient scheduling facilities and lower kernel latency in operating systems, in analogy to shorter packet transmission times, will make process shaping increasingly effective. The result will be stronger guarantees on process behavior and higher system-wide performance especially in overload scenarios. Note that real-time patches providing lower kernel latency, which used to be rather exotic, unsupported code, increasingly make their way into general-purpose operating systems such as Linux, even with industry support.

4 Related Work

The TAP library essentially implements efficient stack management for threads combined with an event-driven state machine that uses prioritized queues of threads. In other words, the TAP library uses event-driven programming in the sense of [1] to implement the illusion of thread-based programming [11], and is therefore related to similar implementations such as Capriccio [17] or State Threads [14]. Unlike Capriccio and State Threads as well as other, hybrid solutions such as SEDA [18] or Flash [6] that focus on the efficiency and usability of the programming model, e.g., event-driven programming [13] vs. thread-based programming [16] or combinations thereof, we emphasize queueing mechanisms and policies for system call scheduling. We aim at controlling the threads' access to network and disk I/O explicitly in order to provide stronger I/O-related process isolation and thus enable disciplined system composition with low overhead.

The idea of system call scheduling for threads is inspired by the notion of threading by appointment (TAP) [8]. 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 the appointment. An earlier version of the TAP library implemented full threading by appointment but suffered from poor performance.

5 Implementation

Our prototype implementation is a cooperative, user-space threading library called the TAP library [9], which supports explicit scheduling of network- and disk-related I/O calls.

Resource	TAP wrapper call (POSIX)	System calls ...	... executed in context of
Network	`read()`	`epoll_ctl()`	calling thread
		`epoll_wait()`	reactor thread
		`sys_read()`	reactor thread
Network	`write()`	`epoll_ctl()`	calling thread
		`epoll_wait()`	reactor thread
		`sys_write()`	reactor thread

Network	`accept(), recv(), send(),`
	`recvfrom(), sendto(),`
		`epoll_ctl()`	calling thread

		`epoll_wait()`	reactor thread
		`sys_socketcall()`	reactor thread
Disk	`read()`	`sys_read()`	reactor thread
Disk	`write()`	`sys_write()`	reactor thread

Table 1. The POSIX-compliant TAP wrapper calls and their corresponding system calls.

The TAP library implements a POSIX-compliant API for threads, i.e., multi-threaded applications that use POSIX-compliant network and disk calls may readily run on TAP. The implementation is based on an efficient queueing mechanism that uses (virtual) tokens to support a wide range of queueing policies. We use queues to implement our policies because in user space they are efficient and do not require timing mechanisms usually used in kernel space. The queueing mechanism can be used to control and bound, e.g., the rate at which system calls are executed. For this purpose, we have implemented queueing policies that are known in the networking community as traffic-shaping policies. For example, we have implemented the well-known leaky-bucket policy, which shapes bursty traffic (in our case, system calls) into a steady stream below a given bandwidth limit.

Table 1 shows the POSIX calls that are implemented by and routed through the library by appropriate wrappers. The table also shows for a given TAP wrapper call which system calls are actually invoked. The TAP library currently uses nonblocking I/O for network calls and blocking I/O for disk calls. POSIX mutex calls are also routed through the TAP library for applications that invoke TAP wrapper calls in critical sections to run correctly. However, the current version of the library does not traffic-shape mutex calls yet. This is interesting but non-trivial future work.

The TAP library implements $n+1$ threads for n user threads of a single application process. The $(n+1)$-th thread is called the *reactor*, which implements CPU and system call scheduling for the n user threads. The reactor is essentially a state machine that enqueues and dequeues user threads according to a given queueing policy. In the design and implementation of the reactor, we explicitly distinguish queueing mechanisms and policies. However, since the reactor, unlike a traditional CPU scheduler, directly determines the reactive (I/O) behavior of a system of threads, we decided to use a non-traditional term [7] to emphasize the reactor's actual function rather than the mechanism it is based on.

5.1 Queueing

The reactor maintains three queues of user threads: CPU, NET, and DISK. There are also auxiliary queues such as a timeout queue to support sleep calls whose description we omit here for brevity. The CPU queue is similar to a traditional ready queue, which contains all threads that are eligible to run. The NET and DISK queues contain threads that have invoked TAP wrapper calls for network and disk I/O, respectively, but whose execution may be delayed (blocked) according to the given queueing policy. The mechanism to decide when to block the running thread, i.e., enqueue the thread into the NET or DISK queue, as well as to decide when a blocked thread becomes ready again, i.e., is moved from the NET or DISK queue to the CPU queue, is based on (virtual) tokens. The execution of each TAP wrapper call consumes a certain number of tokens, which may depend on, e.g., the amount of data transferred by that call. A given queueing policy merely determines the amount and rate at which new tokens are produced as well as the number of tokens required for each TAP wrapper call. The reactor maintains separate NET and DISK tokens for network- and disk-related calls, respectively, to be able to control their rate independently. However, note that there are no CPU tokens, i.e., threads in the CPU queue are dequeued without any rate limitations. Since thread execution times are not known it is hard to quantify and therefore control automatically their resource (CPU) demand in a cooperative system as opposed to many system calls, which usually specify, e.g., the amount of data involved in their execution. The only practical way to overcome this problem may be to support preemption, which is very interesting but beyond the scope of this paper.

All queues are prioritized in order to support priority-based queueing policies, e.g., to trade off lower latency of interactive threads for lower throughput of server threads. If priorities are not used, threads are enqueued and dequeued in FIFO order. For simplicity, we first present the system's design without priorities and later describe the necessary extensions separately.

```
// precondition: calling thread is not referenced
// by any TAP library data structure

// wrapper arguments must be accessible to
// reactor's I/O subsystem
save wrapper arguments in thread context;

// block calling thread
append calling thread to DISK queue;

yield to reactor thread;
```

Listing 1.1. TAP wrapper call scheme for disk I/O

Listings 1.1 and 1.2 outline the implementation scheme of the TAP wrapper calls for disk and network I/O, respectively. Suppose that the currently running thread invokes a TAP wrapper call for disk I/O (we consider network I/O below because it involves the slightly more complicated use of nonblocking system calls). The precondition here is that the calling thread is not referenced by any TAP library data structure, e.g., is not contained in any of the queues. The first step is to save the arguments of the wrapper call in the calling thread's context. The reactor will later, when enough DISK tokens are available, invoke the necessary system calls on behalf of the calling thread, i.e., in the reactor's context. In other words, a thread that invokes a TAP wrapper call for disk I/O is always blocked until the reactor has verified that enough DISK tokens are available for that call. To minimize copying of arguments, we could also have the reactor retrieve the arguments directly from the calling thread's stack but have not done this for simplicity and portability, and because in most cases only three integers are involved anyway. Finally, the calling thread is appended to the DISK queue before transferring control to the reactor.

Now, consider the case of a TAP wrapper call for network I/O. Similar to the disk case, a thread that invokes a TAP wrapper call for network I/O is always blocked at least until the reactor has checked that enough NET tokens are available for that call. Unlike the disk case, however, the reactor also checks, before executing any of the necessary system calls, if these calls will succeed without blocking. Suppose that the currently running thread invokes a TAP wrapper call for network I/O (see Listing 1.2). The precondition here is the same as in the disk case. Again, similar to the disk case, the arguments of the wrapper call are saved in the calling thread's context. The reactor will later retrieve the arguments to invoke the necessary system calls on behalf of the calling thread. Next, an `epoll_event` that contains the calling thread and the file descriptor passed to the wrapper call is registered with the kernel using `epoll_ctl`. The reactor will

```
// precondition: calling thread is not referenced
// by any TAP library data structure

// wrapper arguments must be accessible to
// reactor’s I/O subsystem
save wrapper arguments in thread context;

// calling thread will be retrieved from epoll_event
// by reactor when file descriptor becomes ready
assemble epoll_event from calling thread
and file descriptor;

// file descriptor status will be checked
// by reactor using epoll_wait
register epoll_event with kernel
by calling epoll_ctl;

// calling thread is now considered blocked

yield to reactor thread;
```

Listing 1.2. TAP wrapper call scheme for network I/O

later retrieve the calling thread from the `epoll_event` when the file descriptor becomes ready and then append the calling thread to the NET queue. In other words, the only reference to the calling thread is now in the `epoll_event`. The calling thread is therefore considered blocked and control is transferred to the reactor. In general, we say that a thread is blocked as soon as the thread is contained in the NET or DISK queue, or in an `epoll_event`. However, note that the I/O for threads in the NET queue (and DISK queue anyway) is ready to be completed by the reactor at any time whereas the I/O for threads in any pending `epoll_events` is not ready.

Listing 1.3 shows the reactor thread, which behaves as follows. Threads in the CPU queue are always executed first in FIFO order until completion, i.e., until the threads yield cooperatively to the reactor in one of the TAP wrapper calls. Only when the CPU queue is empty, threads in the NET and DISK queue are considered, i.e., the necessary system calls are executed on behalf of these threads as long as tokens are available. Threads for which system call execution has been completed are moved from the NET and DISK queues to the end of the CPU queue. When all tokens have been consumed or the NET and DISK queues have been emptied, the reactor returns to executing the threads in the CPU queue again.

To be more precise, consider the case when the CPU queue is empty and the currently running thread yields to the reactor thread. Then, the reactor enters its main while loop and first distinguishes three subcases: (1) If the NET and DISK queues are empty, there might still be blocked threads in `epoll_events` whose status will be checked by a blocking `epoll_wait` (the case when there are no more pending `epoll_events` means that all threads have terminated; for simplicity, we have omitted the code that handles

```
// can only be reached from TAP wrapper calls
// except when bootstrapping

// invariant: each thread is referenced exactly once
// either by one of the queues, or else
// by one of the epoll_events

while (CPU queue is empty) {
 if (NET and DISK queues are empty) {
  // all queues are empty but there can be pending
  // epoll_events for blocked threads still waiting
  // for network I/O

   // results in blocking call to epoll_wait
   timeout = -1;
 } else if (any tokens are available) {
  // more tokens than demand:
  // for fairness check if network I/O is ready

  // results in nonblocking call to epoll_wait
  timeout = 0;
 } else {
  // more demand than tokens:
  // there are blocked threads but no more tokens

  // wait up to 1ms
  timeout = 1;
 }

 // determine threads with ready network I/O
 get list of ready epoll_events by
 calling epoll_wait with timeout;
 append threads retrieved from
 ready epoll_events to NET queue;

 set current_time to current time;
 elapsed_time = current_time - last_time;
 last_time = current_time;

 // policy:
 generate NET and DISK tokens
 for amount of elapsed_time;

 // do network I/O for threads in NET queue
 // as long as NET tokens are available
 call I/O subsystem on NET queue;

 // do disk I/O for threads in DISK queue
 // as long as DISK tokens are available
 call I/O subsystem on DISK queue;
}

resume execution of first thread in CPU
queue after removing it from the queue;
```

Listing 1.3. The reactor thread

this case, i.e., system termination). (2) If the NET or DISK queues contain threads, and there are tokens available, a nonblocking `epoll_wait` will check the status of any pending `epoll_events` not to delay network calls unfairly. Conversely, the `epoll_wait` must be invoked nonblocking here not to delay any disk calls unfairly. (3) If the NET or DISK queues contain threads but there are no more tokens available, the reactor will `epoll_wait` for any pending `epoll_events` to become ready but not for more than 1ms. The timeout of 1ms has been experimentally determined. This case is the most interesting because it handles the situation where there is more demand than tokens, i.e., where limiting system call bandwidth takes effect. To optimize I/O throughput, i.e., have the actual I/O throughput approach the given limit as closely as possible, we decided to use the timeout parameter of `epoll_wait` because it allows us to monitor network I/O and idle at the same time. Thus the optimization problem here is to distribute I/O as evenly as possible across the timeline with the least number of `epoll_wait` calls. There is, however, the following quantization problem. If the rate at which tokens are generated is high, i.e., the bandwidth limit is high, it is better to have `epoll_wait` return as soon as there is ready network I/O and not wait for the timeout because the probability that new tokens are available even in a short period of time is high. This is what we have implemented as default. However, if the rate at which tokens are generated is low, it is obviously better to `epoll_wait` for the full timeout even if network I/O becomes ready earlier because only few new tokens if any might have been generated in the meantime. Unfortunately, this semantics is not supported by `epoll_wait`. We presume that implementing this approach as an alternative and a mechanism that chooses automatically the supposedly better option could lead to interesting results.

The I/O throughput in case (2), where there is more tokens than demand, may also be optimized by reducing the number of nonblocking `epoll_wait` calls as follows. As long as there are threads in the NET queue we can skip nonblocking calls to `epoll_wait` without loosing fairness since we may first do the I/O for these threads anyway. Recall that the I/O for threads in the NET queue (and DISK queue anyway) is ready to be completed by the reactor at any time. However, in our experiments, it turned out that the additional complexity (especially with prioritization) is not justified by the rather low benefit that we observed.

After `epoll_waiting` for network I/O, the reactor computes the real time that has elapsed since the last time the reactor reached this point. Then, new NET and DISK tokens are generated with respect to that amount of real time and according to the given queueing policy. Finally, the reactor invokes the TAP library's I/O subsystem on the NET and DISK queues to perform network and disk I/O, respectively.

Listing 1.4 shows, for brevity, the I/O subsystem for a generic queue called X. Threads are removed from X in FIFO order and appended to the CPU queue as long as tokens for X are available. The I/O subsystem executes the necessary system calls on behalf of each thread in the reactor's context. Recall that the required arguments to the TAP wrapper calls are stored in the thread contexts. In order to account for the actual I/O workload, the number of tokens consumed by each wrapper call is determined, for network calls, with respect to the actual amount of transferred data and, for disk calls, with respect to the actual amount of transferred data scaled logically by the real time required to transfer the data. Network I/O workload typically correlates closely with

```
// called by reactor on queue X

while (queue X is not empty and
       tokens for queue X are available) {
  remove first thread from queue X and
  append it to CPU queue;

  // wrapper arguments are in thread context
  do I/O on behalf of this thread
  but in reactor context;

  // policy:
  consume tokens for queue X
  for actual I/O workload;
}

return to reactor;
```

Listing 1.4. The I/O subsystem

the amount of transferred data whereas disk I/O workload also depends on cache and disk effects. To compute the number of disk tokens, we therefore scale the amount of transferred disk data heuristically by a factor w, which can be 0 (effectively disabling process shaping assuming a memory cache hit occurred), 0.1 (disk cache hit), 1 (sequential disk access), and 5 (seek). If the number of CPU cycles c per transferred byte, which we measure with the Intel time-stamp counter, is below or above experimentally determined low and high thresholds, w is set to 0 or 2, respectively. Otherwise, if c is below or above another experimentally determined medium threshold, w is set to 0.1 or 1, respectively.

Since the actual amount of data and real time involved in a system call is not known prior to the call, we take an optimistic approach to maximize throughput. In the network case, if there are tokens available but less than required to complete a wrapper call, the I/O subsystem decreases the amount of data involved in the execution of the according system call to the amount for which sufficiently many tokens are available. This approach is optimistic because the system call may actually transfer even less data. After completing the system call, the wrapper call returns to the calling thread with the actual amount of transferred data, effectively creating back-pressure on the application code, which is conform to the semantics of POSIX I/O calls such as read and write. In the disk case, the system call is invoked on the full amount of data as soon as there is at least one token available. Otherwise, the system call is delayed and other work is done first. This approach is again optimistic because it assumes that the system call hits the memory cache. If not, even more tokens than available may be consumed. In this case, token credit is granted but subsequent disk calls will be delayed until there is again a positive token balance.

Our approach approximates a given bandwidth limit as closely as possible from below and above at, however, an increased risk for thrashing. An alternative is to not execute a system call at all if not enough tokens are available but have the wrapper call immediately return with an error code at, however, the risk of loosing fairness. We have therefore not implemented the alternative. Down-sizing at least network wrapper calls because of lack of tokens is also important in order to enable prioritization effectively in overload scenarios since, e.g., a low-priority thread that invokes wrapper calls involving large amounts of data may otherwise be prevented from progressing even by just a few higher-priority threads.

To complete this section let us get back to prioritization in the TAP library. All queues in the library are prioritized in order to support priority-based queueing policies. The current version of the library supports ten different priorities with constant-time insertion and deletion operations using separate doubly-linked lists for each priority. Entries with the same priority are handled in FIFO order. The reactor honors priorities globally across all queues: at any time, the threads with the highest priority are dequeued first independently of whether they are enqueued in the CPU, NET, or DISK queue. This affects the implementation of line 8 in Listing 1.3 and line 3 in Listing 1.4. The reactor actually enters the while loop only if there are no threads in the CPU queue that have priorities equal or higher than the priorities of any threads in the NET and DISK queues. The I/O subsystem enters the while loop only if, besides enough tokens for X, there are threads in queue X that have priorities equal or higher than the priorities of any threads in the other queues. This means that once the system has started dequeueing threads from any of the queues, threads are being dequeued even when threads with the same priority are contained or turn up in the other queues, which is essential for fairness among threads with the same priority.

5.2 Policies

The token-based queueing mechanism in the TAP library supports a wide range of traffic-shaping policies of which we have implemented the classical leaky- and token-bucket policies, see, e.g., [15]. With the leaky-bucket policy, traffic is enqueued into a queue of finite size (the bucket size) and dequeued at a fixed rate (the leak size). The leaky-bucket policy shapes possibly bursty traffic into a steady stream that flows at or below some bandwidth limit (the leak size). In networking, packets are discarded if the queue is full. In our implementation, all queues are unbounded although the queues cannot grow beyond the number of active threads in the system because each thread can obviously invoke only a single TAP wrapper call at a time. Nevertheless, as future work it could be interesting to provide a way to down-size queues and then return wrapper calls with an error code if queues are full. This would allow the system to create back-pressure on the application code before even queueing anything.

With the token-bucket policy, traffic is dequeued only if tokens are available. Tokens are generated at a fixed rate. Moreover, unused tokens are collected up to a given limit (the bucket size), i.e., here the bucket contains tokens rather than traffic. The token-bucket policy is similar to the leaky-bucket policy in that it shapes bursty traffic into a steady stream but also allows infrequent bursts to go through at full speed. The maximum duration of a burst that will go through at full speed is determined by the bucket size. The advantage of the token-bucket policy is that unused resources may be utilized more effectively during times of low traffic than with the leaky-bucket policy. However, the disadvantage is that the QoS guarantees on resource demand and therefore the system's isolation properties are weaker than the guarantees by the leaky-bucket policy since bursty traffic may utilize resources more than the bandwidth limit allows. Note that both policies may be combined, e.g., a token bucket followed by a leaky bucket, in order to reestablish stronger guarantees.

We have implemented both policies in the TAP library using tokens. The leaky-bucket policy can be seen as a special case of the token-bucket policy by limiting the size of the token bucket to one token (modulo quantization). In the implementation, a token bucket is represented by a single integer that we use to count the number of tokens in the bucket.

Besides the leaky- and token-bucket policies, we have also implemented and experimented with two prioritizing queueing policies: (1) a latency-oriented policy that gives interactive threads highest priority but decreases their priority as they invoke more and more I/O system calls, and (2) a throughput-oriented policy that gives server threads increasing priority as they invoke more and more I/O system calls. We intuitively characterize threads as follows: a thread whose system calls have consumed just a few tokens since the thread has last invoked an accept call on a socket is an interactive thread; however, if the thread has consumed a lot of tokens since then it is a server thread. Since priority-based policies can easily be combined with token-based policies, we have also experimented with combinations such as the latency-oriented policy combined with the leaky-bucket policy, which enforces a given bandwidth limit while decreasing the latency of interactive threads.

6 Experiments

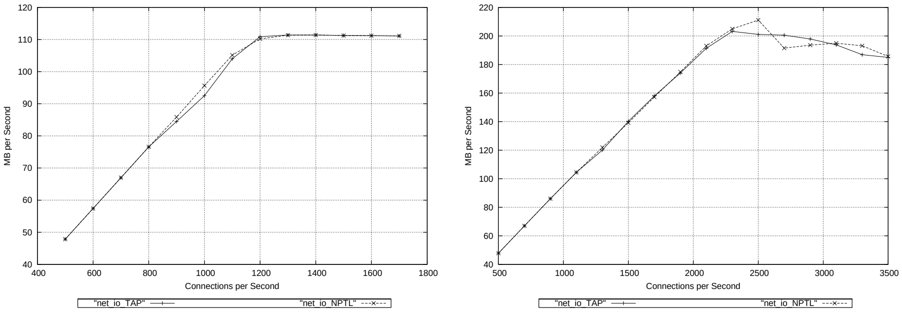
We discuss several experiments to substantiate our proposal and claims. All experiments have been conducted using our previously mentioned TAP library on an unmodified Linux 2.6.15 kernel. Multi-threaded applications such as many web and streaming servers can usually be linked against the POSIX-compliant TAP library without modifications. The applications' I/O calls, e.g., network and disk reads and writes, are then routed through the library and shaped according to a given bandwidth limit and queueing policy such as the well-known leaky- and token-bucket policies.

6.1 TAP Overhead

In this section we present the overhead of network and disk I/O induced by our user-space library.

Figure 2 shows the throughput of our web server linked against the NPTL [5] and against TAP. Process shaping is disabled by setting the token rate higher than the available network bandwidth. The results are measured on the client side and only successful connections are taken into account, i.e., when the complete requested file is successfully received on the client.

Figure 2(a) shows that we reach the maximum throughput of our connection with both TAP and NPTL. The CPU usage is at around 60% in both cases. Figure 2(b) shows



(a) One Ethernet device used (b) Two Ethernet devices used

Fig. 2. A comparison of the throughput of our web server linked against TAP or NPTL when requesting a 94KB file. The X-axis denotes the request rate and the Y-axis the transferred KB per second.

the throughput when two network cards instead of just one are used. With this setting, the CPU usage is at 100% and the throughput drops slightly as the server start thrashing. However, both configurations deliver similar throughput.

	no-TAP	TAP	no-TAP	TAP
1. run	32.20 s	32.39 s	34.24 s	34.59 s
2. run	18.90 s	22.14 s	25.97 s	26.24 s
3. run	1.26 s	2.38 s	0.55 s	0.62 s
4. run	0.67 s	1.26 s	0.52 s	0.52 s
chunk size	4KB		32KB	

Table 2. The real time in seconds for reading a 1GB file four times with "cat file > /dev/null" using different chunk sizes.

For disk I/O we show the overhead of process shaping when reading a 1GB file using `cat` (Table 2). We use `cat` to measure the overhead of the TAP disk I/O subsystem. With the preload feature of the GNU linker, we can link unmodified applications like `cat` with the TAP library without recompiling. Table 2 shows how long `cat` takes to read a 1GB file with and without TAP. We read the same file four times and measure the elapsed real time for each run (averaged over three repetitions). The last run, i.e., when the data is cached, is a benchmark of the overhead introduced by the TAP library. In the first run, `cat` with TAP is only slightly slower than `cat` without TAP. In the second and third run, `cat` with TAP takes approximately 0.6 seconds longer than `cat` without TAP. This is probably due to the additional context switches and token calculations in the TAP library. The standard `cat` uses a chunk size of 4KB for the read and write calls, and therefore generates a relative high number of system calls that must be routed through the TAP library. We also experimented with a modified version of `cat` that uses 32KB chunks of data to read and write reducing the overhead of TAP for the cached load. The overhead is negligible for non-cached data but not for cached data unless the standard I/O chunk size in `cat` is increased.

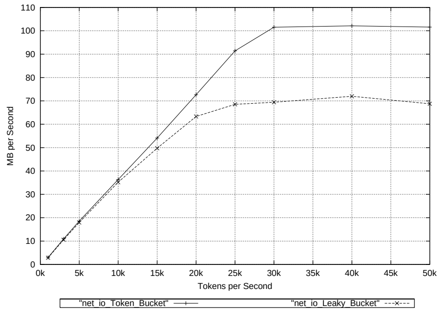


Fig. 3. The effect of network token rates on the network throughput.

6.2 Token Rate and Network Bandwidth

Using a cache-bound workload that consists of a single 94KB file, we show that (1) there is a near-linear relation between the network token rate and the achieved network throughput when using the leaky- and token-bucket policies, and (2) the leaky-bucket policy enforces a given bandwidth limit but with lower maximum throughput than the token-bucket policy (Figure 3).

The token rate is used to control the maximum throughput of an application. In Figure 3, the relation between network token rate and throughput is near-linear until reaching the limit of the network device. For all token rates, we use a connection rate of 1100 connections per second resulting in a maximum throughput that is slightly higher than 100MB per second. The token-bucket policy reaches that throughput at 30000 tokens per second similar to the maximum throughput shown in Figure 2(a). The maximum throughput of the leaky-bucket policy is about 70MB. The leaky-bucket policy provides less throughput but is more deterministic than the token-bucket policy in the sense that even bursty traffic is guaranteed not to exceed the given bandwidth limit. In the subsequent experiments, we use the token-bucket policy.

We were not able to demonstrate a similar relationship between disk token rates and throughput because of the large variations in disk throughput due to cache effects and other optimizations such as prefetching. This means that disk call shaping is harder to control than network call shaping but we show below that disk call shaping can still be effective.

6.3 Network Process Shaping

In this experiment, we ran two multi-threaded web servers on a single machine accepting and serving incoming connections on a gigabit network. Two client machines repeatedly connect to the web servers and request the same and thus cached 380KB file.

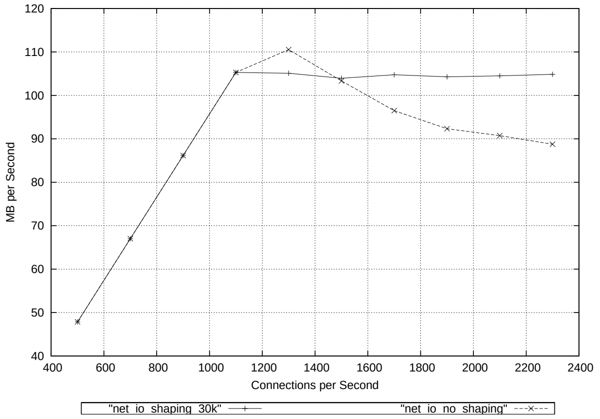


Fig. 4. The combined throughput of two web servers running concurrently on a single machine is more deterministic and scales to higher connection rates with process shaping than without it.

Figure 4 depicts the net throughput measured at the clients with shaping disabled, and when each web server is shaped to use at most 50% of the available network bandwidth. Note that each web server runs on its own instance of the TAP library, i.e., there is no explicit coordination between the two web servers. The combined throughput of the web servers without shaping shows a distinct peak of 110MB/s at 1300 connections per second, and decreases for higher connection rates to less than 90MB/s. The web servers with shaping do not achieve the same peak throughput but continue to stay at the same high rate of 105MB/s, and even achieve a higher throughput than without shaping for connection rates higher than 1500 connections per second.

6.4 Disk Process Shaping

We use a synthetic file set of 31 different files in the following configuration: 10 files are between 1KB and 10KB, 14 files are between 12KB and 64KB in steps of 4KB, and 7 files are between 128KB and 512KB in steps of 64KB as a workload. With caching disabled, we show that two independent web server processes (1) achieve lower peak disk throughput on TAP when set to no bandwidth limits than on the NPTL but (2) scale better in terms of disk throughput, and (3) show more balanced disk throughput across the workload, in particular, when set to a 1 : 1 bandwidth ratio with TAP (Figure 5).

Similar to the previous experiment, we start two web server processes that listen on different ports for incoming connections. However, the web servers are modified to open all files with `O_DIRECT`, which instructs all calls on the file descriptors to bypass the kernel page cache. In this experiment one token is worth one page of 4KB. On the client side, we also use a lower connection rate because the limiting factor is now the disk and not the network device. Figure 5 depicts the results of these experiments.

We used three different web server configurations, one with the NPTL, then with TAP but no bandwidth limits, and finally with TAP and a 1 : 1 ratio on disk band-

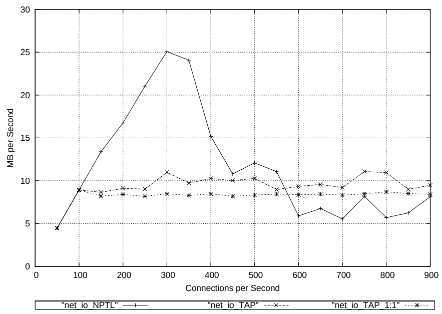
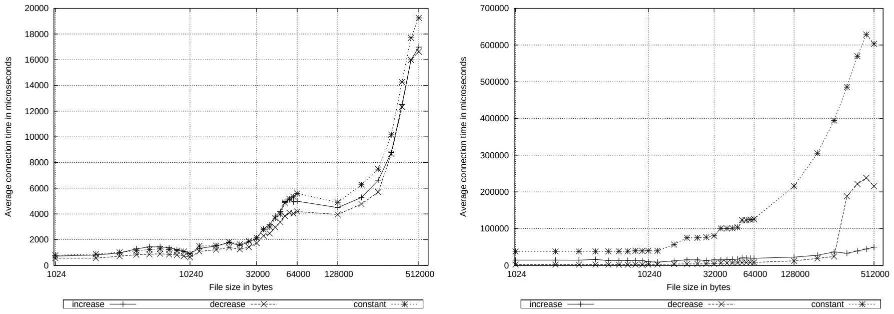


Fig. 5. Combined throughput of two web servers for a disk-bound workload when files are opened with `O_DIRECT`.

width. For the first configuration with a connection rate greater than 400 connections per second, we observed that the server machine was not responsive. Simple shell commands like `top` did not respond for several seconds. We believe this is related to the large number of pending I/O requests. We did not experience this phenomenon with the other configurations. The second configuration results in less throughput but scales better than the first. For the third configuration, a token rate of 1300 tokens per second for each server process is used resulting in less peak performance than the other configurations but a more deterministic behavior.

6.5 Prioritizing Policies



(a) 40k Tokens per Second

(b) 30k Tokens per Second

Fig. 6. Time to process a successful connection when using different prioritizing policies. The X-axis denotes the file size and the Y-axis the time the server needs to process a connection successfully. Different token rates are used to demonstrate the effect of prioritization.

Using the same workload as in the previous experiment of 31 different files and additionally prioritizing policies, we (1) confirm that shortest-connection-first scheduling [4] reduces the latency of connections serving small files but also (2) add that the opposite strategy, i.e., “longest-connection-first” scheduling results in a balanced latency of connections serving files of different size (and higher throughput, not shown), and (3) demonstrate that limiting bandwidth rather than just overloading as in [4] can be used to make prioritization effective (Figure 6). As described in the implementation all queues are prioritized. We can use this fact to emulate connection scheduling [4]. In [4] the authors propose to process short connections first, i.e., shortest-connection-first scheduling, in servers that use static content. We can confirm their results, and additionally, present the results of the opposite strategy, i.e., longest-connection-first scheduling. In contrast to [4], we do not assign priorities to connections but rather to threads. All web server threads set their priority according to the size of the current file being served. We use two different strategies to demonstrate the effect of priorities: (i) increase the priority for big files, (ii) decrease the priority for big files (as in [4]). As reference, we also run the benchmark with priorities disabled by setting the priority of all threads to the same constant value. At the beginning of the thread loop, i.e., before accepting a connection, the priority is set to the maximum. The desired effect of (ii) would be better latency for shorter connections, whereas with (i) all connection times should be roughly equal. For this we modified the web server such that it is logging the connection time for each successful connection. Note, however, that this data is collected directly on the server machine and not on the client machine as opposed to the previous tests and in contrast to the experiments in [4].

Figure 6 shows the results of these tests. The connection rate is 2000 connections per second and we use both Ethernet cards. Each benchmark runs for 30 seconds. The throughput without process shaping is around 2 times 90MB per second. Similar as in [4], prioritization of connections takes effect only if there is enough load on the server. In [4] the authors increase the load on the web server by increasing the connections from the clients. In our setting we have the choice between increasing the connections from the clients or limiting the bandwidth of the server processes using process shaping. We choose to keep the connection rate, but use process shaping to limit the maximum throughput of the server process. We enable process shaping with two different limits. In Figure 6(a) we use a token rate of 40000 tokens per second on network I/O. All three strategies have a similar curve. Because the web server is only slightly overloaded nearly all connections are processed regardless of their priority. The stepwise increase in connection time is due to the used buffer size of 16KB. In Figure 6(b) we use a token rate of 30000 tokens per second. With this setting the web server is overloaded, i.e., the client processes request more connections and bandwidth than the server process is allowed to use, and the priorities take full effect. As expected, with (ii) we get the best latency for small files at the expense of the bigger files.

7 Conclusion and Future Work

We feel that our experiments show the potential of process shaping. We have demonstrated in a number of experiments that there are cache- and disk-bound workloads for which the TAP library offers similar network throughput than our baseline, the native POSIX thread library for Linux, can effectively shape, limit, and prioritize network and disk I/O. Process shaping increases the degree of I/O-related determinism while taking advantage of performance-oriented, kernel-level CPU and I/O schedulers. In general, we feel that increasing I/O-related determinism while maintaining or even improving performance is promising, not only for real-time and quality-of-service systems, but also in the context of purely performance-oriented systems.

As a next step beyond the current implementation we have finished a kernel-space mechanism to remove some of the inadequacies of the presented user-space library. The kernel-space implementation produces less overhead and applications that are shaped do not need to be compiled against any library. We have also taken the concept of process shaping one step further in the kernel-space version and defined a new approach to controlling system behavior through system call scheduling, leaving process shaping as an example of controlling behavior. Our results encourage our next, short-term steps, like studying ways to identify bandwidth limits and shaping policies for optimal system-wide utilization dynamically at runtime, and long-term goals like supporting even hard real-time applications such as the flight control system of our unmanned quadrotor helicopter called the JAviator [2].

References

1. A. Adya, J. Howell, M. Theimer, W.J. Bolosky, and J.R. Douceur. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proc. USENIX*, 2002.
2. J. Auerbach, D.F. Bacon, D. Iercan, C.M. Kirsch, H. Röck, and R. Trummer. The JAviator Project. http://javiator.cs.uni-salzburg.at/.
3. R.O. Baldwin, N.J. Davis, S.F. Midkiff, and J.E. Kobza. Queueing network analysis: concepts, terminology, and methods. *Systems and Software*, 66(2):99–117, 2003.
4. M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *Proc. USITS*, 1999.
5. U. Drepper and I. Molnar. The native posix thread library for linux. http://people.redhat.com/drepper/nptl-design.pdf, 2005.
6. P. Druschel, V.S. Pai, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proc. USENIX*, 1999.
7. C.M. Kirsch. Principles of real-time programming. In *Proc. EMSOFT*, volume 2491 of *LNCS*, pages 61–75. Springer, 2002.
8. C.M. Kirsch. Threading by appointment. In *Proc. Monterey Workshop*. CRC Press, 2004.
9. C.M. Kirsch and H. Röck. The TAP Project. http://tap.cs.uni-salzburg.at/.
10. C.M. Kirsch and H. Röck. Traffic shaping system calls using threading by appointment. Technical Report T009, Department of Computer Sciences, University of Salzburg, August 2005.
11. H.C. Lauer and R.M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, 1979.
12. J.P. Lehoczky. Real-time queueing theory. In *Proc. IEEE RTSS*, 1996.
13. J. Ousterhout. Why threads are a bad idea (for most purposes). In *Proc. USENIX (Invited Talk)*, 1996.
14. G. Shekhtman and M. Abbott. State threads for Internet applications. http://state-threads.sourceforge.net/docs/st.html.

15. A. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 2002.
16. R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proc. HotOS IX*, 2003.
17. R. von Behren, J. Condit, F. Zhou, G.C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *Proc. SOSP*, 2003.
18. M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. SOSP*, 2001.