Efficient Network and I/O Throttling for Fine-Grain Cycle Stealing

Kyung D. Ryu
Dept. of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406
kyung.ryu@asu.edu

Jeffrey K. Hollingsworth
Dept. of Computer Science
University of Maryland
College Park, MD 20740
hollings@cs.umd.edu

Peter J. Keleher
Dept. of Computer Science
University of Maryland
College Park, MD 20740
Keleher@cs.umd.edu

Abstract

This paper proposes and evaluates a new mechanism, rate windows, for I/O and network rate policing. The goal of the proposed system is to provide a simple, yet effective way to enforce resource limits on target classes of jobs in a system. This work was motivated by our Linger Longer infrastructure, which harvests idle cycles in networks of workstations. Network and I/O throttling is crucial because Linger Longer can leave guest jobs on non-idle nodes and machine owners should not be adversely affected. Our approach is quite simple. We use a sliding window of recent events to compute the average rate for a target resource. The assigned limit is enforced by the simple expedient of putting application processes to sleep when they issue requests that would bring their resource utilization out of the allowable profile. Our I/O system call intercept model makes the rate windows mechanism light-weight and highly portable. Our experimental results show that we are able to limit resource usage to within a few percent of target usages.

1. Introduction

This paper proposes and evaluates *rate windows*, a new mechanism for I/O and network rate policing. Integrated with our existing *Linger-Longer* infrastructure for policing CPU and memory consumption [17], rate windows give unprecedented control over the resource use of user applications. More specifically, they are a low-overhead facility that gives us the ability to set hard per-process bounds on I/O and network usage.

Current general-purpose UNIX systems provide no support for prioritizing access to other resources such as memory, communication and I/O. Priorities are, to some degree, implied by the corresponding CPU scheduling priorities. For example, physical pages used by a lower-priority process will often be lost to higher-priority processes. LRU-like page replacement policies are more likely to page out the lower-priority process's pages, because it runs less frequently. However, this might not be true with a higher-priority process that is not computationally intensive, and a lower priority process that is. We therefore need an additional mechanism to control the memory allocation between local and guest processes. Similarly, I/O and network access by guest jobs can interfere with host jobs that are doing I/O or accessing the network. To prevent this, I/O and network policing mechanisms are needed.

Our rate window mechanism has applications in several areas; we perform a detailed investigation of two in this paper. First, we show that network and I/O throttling is crucial in order to provide guarantees to users who allow their workstations to be used in Condor-like systems. Condor-like facilities allow *guest* processes to efficiently exploit otherwise-idle workstation resources. The opportunity for harvesting cycles in idle workstations has long been recognized [13], since the majority of workstation cycles go unused. In combination with ever-increasing needs for cycles, this presents an obvious opportunity to better exploit existing resources.

SC2001 Nov. 2001, Denver © 2001 ACM 1-58113-293-X/01/0011 $5.00

However, most such policies waste many opportunities to exploit cycles because of overly conservative estimates of resource contention. Our *Linger-Longer* approach [16] exploits these opportunities by delaying migrating guest processes off of a machine in the hope of exploiting fine-grained idle periods that exist even while users are actively using their computers. These idle periods, on the order of tens of milliseconds, occur when users are thinking, or waiting for external events such as disks or networks. Our prior work [17] consisted of new mechanisms and policies that limit the use of CPU cycles and memory by guest jobs. The work proposed in this paper complements that work in extending similar protection to network and I/O bandwidth usage.

Second, we show that rate windows can be used to efficiently provide rate policing of network connections. Rate limiting is useful for managing resource allocations of competing users (such as virtual hosting of web servers) and also can be used for rate-based clocking of network protocols as a means of improving the utilization of networks with high bandwidth-delay products [8, 14].

The rest of this paper is organized as follows. Section 2 reviews the CPU and memory policing mechanisms for the Linger-Longer infrastructure. Section 3 describes the design and implementation of rate windows. Section 4 describes the use of rate windows in policing file I/O, and Section 5 describes its use with network I/O. Finally, Section 6 reviews related work, and Section 7 concludes.

2. CPU and memory policing

Before discussing rate windows, we place this work in the context of the Linger-Longer resource-policing infrastructure [16]. The Linger-Longer infrastructure is based on the thesis that current Condor-like [12] policies waste many opportunities to exploit idle cycles because of overly conservative estimates of resource contention. We believe that overall throughput is maximized if systems implement fine-grained cycle stealing by leaving guest jobs on a machine even when a primary user is present and host jobs are running. In earlier work [16], our trace-driven simulations demonstrated that Linger-Longer can harness up to 60% more idle cycles than the immediate eviction policy adopted by most of the existing systems. However, the host job will be adversely affected unless the guest job's resource use is strictly limited. Our earlier work strictly bounded CPU and memory use by guest jobs through the use of a few, simple modifications to existing kernel policies.

These policies rely on two new mechanisms. First, a new *guest priority* prevents guest processes from running when runnable host processes are present. The change essentially establishes guest processes as a different class, such that guest processes are not chosen if any runnable host processes exist. This is true even if the host processes have lower runtime priorities than the guest process. Note that running with "nice –19" is not sufficient, as the nice'd process can still consume between 8%, 15%, and 40% of the CPU for Linux (2.0.32), Solaris (SunOS 5.5), and AIX (4.2), respectively [17].

Our second mechanism limited guest consumption of memory resources. The cost of reclaiming page frames from a running process is negligible for clean pages, but quite large for modified pages because they need to be flushed to disk before being reclaimed. Our approach does-not impose any hard restrictions on the number of physical pages that can be used by a guest process. Instead, we implemented a policy that establishes low and high thresholds for the number of physical pages used by guest processes. We modified the Linux kernel to support this prioritized page replacement. Two new global kernel variables were added for the memory thresholds, and are configurable at run-time via system calls.

The kernel keeps track of resident memory size for guest processes and host processes. Periodically, the virtual memory system triggers the page-out mechanism. When it scans in-memory pages for replacement, it checks the resident memory size of guest processes against the memory thresholds. If they are below the lower thresholds, the host processes' pages are scanned first for page-out. Resident sizes of guest processes larger than the upper threshold cause the guest processes' pages to be scanned first. Between the two thresholds, older pages are paged out first no matter what processes own them. These thresholds are usually set very low (5-10% of the total memory) so as not to affect memory intensive host jobs.

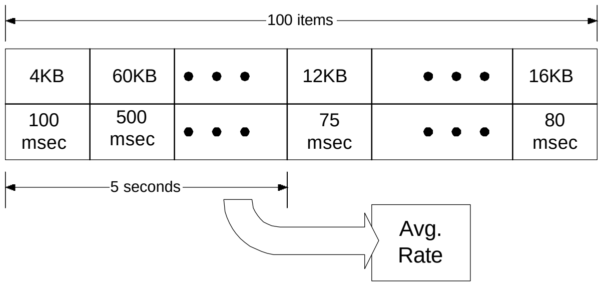


Figure 1: Maintaining a sliding window of resource utilization.

3. Rate Windows

Rate windows are proposed here as a simple, portable, and effective strategy for enforcing limits on I/O and network bandwidth, analogous to the limits on CPU and memory usage. The rest of this section describes our rate-window policies, and the mechanisms that are needed to support I/O throttling.

3.1 Policy

First, we distinguish between "unconstrained" and "constrained" job classes. The default for all processes is unconstrained; jobs must be explicitly put into constrained classes. The unconstrained class is allowed to consume all available I/O. Each distinct constrained class has a different *threshold bandwidth*, defining the maximum aggregate bandwidth that all processes in that class can consume. As an optimization, however, if there is only one class of constrained jobs, and no I/O-bound unconstrained jobs, the constrained jobs are allowed unfettered access to the available bandwidth.

We identify the presence of unconstrained I/O-bound jobs by monitoring I/O bandwidth, moving the system into the *throttled* state when unconstrained bandwidth exceeds $thresh_{high}$, and into the unthrottled state when unconstrained bandwidth drops below $thresh_{low}$. Note that $thresh_{low}$ is lower than $thresh_{high}$, providing hysteresis to the system to prevent oscillations between throttled and un-throttled mode when the I/O rate is near the threshold. The state of the system is reflected in the global variable `throttled`. Note that the current unconstrained bandwidth is not an instantaneous measure; it is measured over the life of the rate window, defined below.

3.2 Mechanism

The implementation of rate windows is straightforward. We currently have a hard-coded set of job equivalence classes, although this could be easily generalized for an arbitrary number. Each class has two kernel *window structures*, one for file I/O and one for network I/O. Each window structure contains a circular queue, implemented via a 100-element array (see Figure 1).

The window structure describes the last I/O operations performed by jobs in the class, plus a few other scalar variables. The window structure only describes I/O events that occurred during the previous 5 seconds, so there may be fewer than 100 operations in the array. We experimented with several different window sizes, finding little sensitivity to the exact value. Nonetheless, it is clearly possible that new environments or applications could be best served by using other values. We provide a means of tuning these and other parameters from a user-level tool.

We implemented our mechanism via a loadable kernel module which intercepts each of the kernel calls for I/O and network communication: `read()`, `write()`, `send()`, and `recv()`. Whenever such system functions are triggered, we first call `rate_check()` with the process ID, I/O length, and I/O type and then call the original system call. The process ID is used to map to an I/O class, and the I/O type is used to distinguish between file and network I/O. The `rate_check()` routine maintains a sliding window of operations performed for each class of service and for the overall system. However, to prevent using too old of information, we limit the sliding window to a fixed interval of time (currently 5 seconds).

At the time that a constrained process attempts to perform I/O, we define the *window*

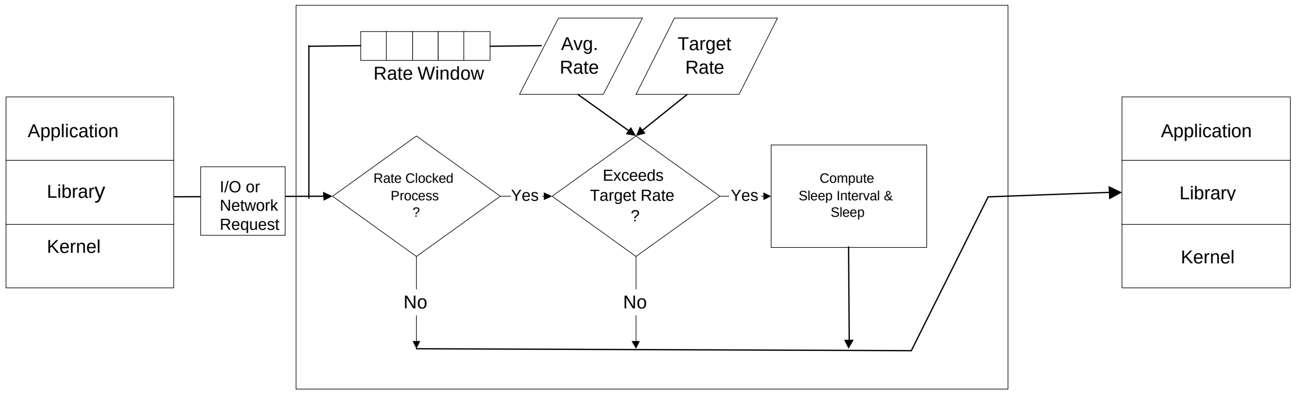


Figure 2: Policing I/O Requests.

bandwidth, B_w, as the total amount of I/O in the window's operations, including the new operation. We define T_w, the *window time*, as the interval from the beginning of the oldest operation in the window until the expected completion of the new operation, assuming it starts immediately. Let R_t be the threshold bandwidth per second for this class. We then allow the new operation to proceed immediately if the class is currently throttled and:

$$\frac{B_w}{T_w} \le R_t$$

Otherwise, we calculate the `sleep()` delay as follows:

$$\text{delay} = \frac{B_w}{R_t} - T_w$$

And then the kernel suspends the process for delay time units before calling the original I/O system call. This process is illustrated graphically in Figure 2. Note that we have upper and lower bounds on allowable sleep times.

Sleep durations that are too small degrade overall efficiency, so durations under our lower bound are set to zero. Sleep durations that are too large tend to make the stream bursty. If our computed delay is above the computed threshold we break the I/O into multiple pieces and spread the total delay over the pieces. This will not affect application execution since file I/O requests will eventually be broken into individual disk blocks and for network connections TCP provides a byte-oriented stream rather than a record oriented one.

We chose Linux as our target operating system for several reasons. First, it is one of the most widely used UNIX operating systems. Second, the source code is open and widely available. Because our throttling mechanisms are implemented as a loadable kernel module, end users can easily load and enable them at run-time. By contrast, a source code patch would require rebuilding a kernel and rebooting a machine.

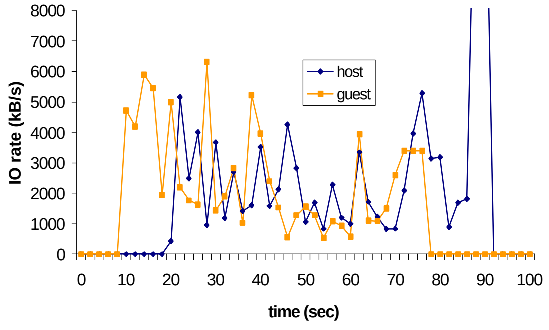
Also, since our mechanism simply requires the ability to intercept I/O calls, it would be easy to implement on other systems that defined an API to intercept I/O calls. Windows 2000 (nee Windows NT) and the stackable file system [10] provide the required calls.

In order to provide the finer granularity of sleep time to allow our policing to be implemented, we augmented the standard 2.2 Linux kernel with extensions developed by the KURT Real-time Linux project [3]. KURT's microsecond resolution timer support was enabled since Linux 2.2 can support only a 10 millisecond resolution timer for sleep[1].

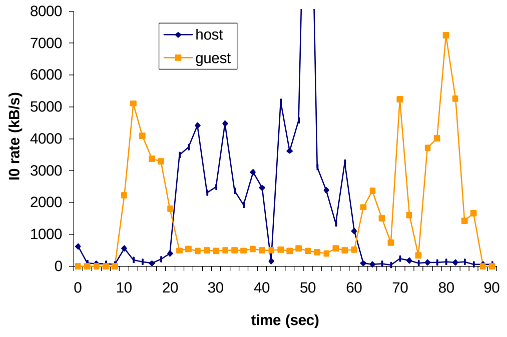
4. File I/O Policing

In order to validate our approach, we conducted a series of micro-benchmarks and application benchmarks. The purpose of these experiments is three fold. First, we want to show that our mechanism does not introduce any significant delay on normal operation of the system. Second, we want to show that we can effectively police the I/O rates. Third, since our policing mechanism sits above the file buffer cache, it will be conservative in policing the disk since hits in cache will be charged against a job classes's overall file I/O limit. We wanted to measure this affect.

[1] This is due to the default setup of the timer unit on Linux. Linux 2.4 can now support a higher resolution timer using APIC, so the KURT patch will not be needed.



(a)



(b)

Figure 3: File I/O of competing tar applications without (left) and with (right) file I/O policing.

We first measured resource usage in order to verify that the use of rate windows does not add significant overhead to the system. We ran a single tar program by itself both with and without rate windows enabled. We did not set the I/O limit since we wished to measure the overhead of maintaining rate windows and computing delays. The difference in completion time of the tar application with rate windows enabled was less than the variation between several runs of the experiment. This was expected, as there are no computationally expensive portions of the algorithm.

Second, we ran two instances of tar, one as a guest job and one as a host job. Figure 3(a) represents a run without throttling, and Figure 3(b) shows a run with throttling enabled. There is no caching between the two because they have disjoint input. The guest job is intended to be representative of those used by cycle-stealing schedulers such as Condor. Unless specified otherwise, a "guest" job is assumed to be constrained to 10% of the maximum I/O or network bandwidth, whereas a "host" process has unconstrained use of all bandwidth.

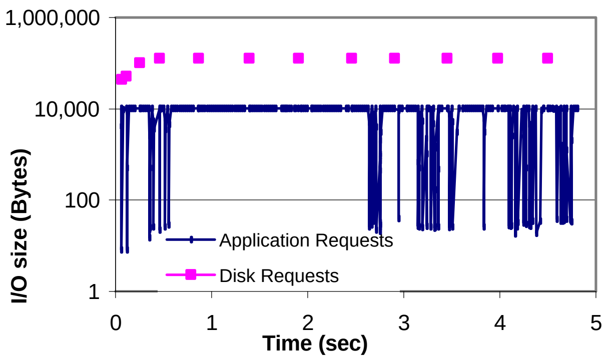


Figure 4: I/O sizes vs. time for `tar`

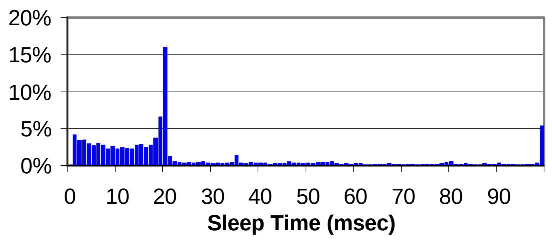
In both figures, the guest job starts first, followed somewhat later by the host job. At this point, the guest job throttles down to its 10% rate. When the host job finishes, the guest job throttles back up after the rate window empties. The sequence on the left is with throttling, on the right without. Note that the version with I/O throttling is less thrifty with resources (the guest job finish later). This is a design decision: our goal is to prevent undue degradation of unconstrained host job performance at the expense of slowing down guest jobs.

The host tar application took 35.5 seconds in isolation. It took 64.4 seconds without throttling and 42.1 seconds with throttling. This demonstrates that throttling the guest job's I/O to 500 kB/s reduces the delay of host I/O from 81% to 18%.

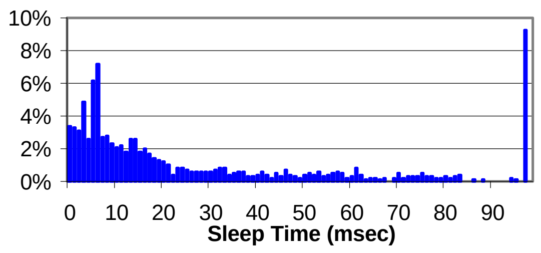
We look at the behavior of one of the tar processes in more detail in Figure 4. The graph shows that despite the frequent and varied file I/O calls and the buffer cache, disk I/O's get issued at regular intervals that precisely match the threshold value set for this experiment. Note that actual disk I/O sizes increase near the start as the file system read ahead becomes more aggressive.

Our third set of micro-benchmark experiments is designed to look at the distribution of sleep times for a guest process. For this case, we ran three different applications. The first application was again a run of the tar utility. Second, we ran the agrep utility[2] across the source directory for the Linux kernel looking for a simple pattern that did not occur in the files searched. Third, we ran a compile workload that consisted of compiling a

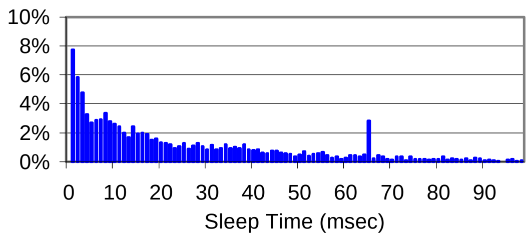
[2] A Unix command to search a file for a string or regular expression, with approximate matching capabilities.



(a) Tar



(b) Compile Workload



(c) Agrep

Figure 5: Distribution of Sleep Times for Tar program.

library of C++ methods that were divided among 34 files plus 45 header files. This third test was designed to stress the gap between monitoring at the file request level and the disk I/O level since all of the common header files would remain in the file buffer cache for the duration of the experiment.

A histogram (100 buckets) of the sleep durations is shown in Figure 5. We have omitted those events that have no delay since their frequency completely dominates the rest of the values. Figure 5(a) shows the results for the tar application. In this figure, there is a large spike in the delay time at 20msec since this is exactly the mean delay required for the I/O for the most common sized I/O request, 10K bytes, to be limited to 500 KB/sec. Figure 5(b) shows the results for the compilation workload. In this example, the most popular sleep time is the maximum sleep duration of 100msec. This is due to the fact that at several periods during the application execution, the program is highly I/O intensive and our mechanism was straining to keep the I/O rate throttled down. Figure 5(c) shows the sleep time distribution for the agrep application. The results for this application show that the most popular sleep time (other than no sleep) was 2-3 ms. This is very close to the mean sleep time of 2.5 ms for this application.

Fourth, we examine the relationship between file I/O and disk I/O using three applications, `tar`, `agrep` and `compile`, which have different I/O patterns. File I/O can dilate because i) file I/O's can be done in small sizes, but disk I/O is always rounded up to the next multiple of the page size, and ii) the buffer cache's read-ahead policy may speculatively bring in disk blocks that are never referenced. File I/O can also attenuate due to buffer cache hits, which is a consequence of the I/O locality of the applications. We measured 1) the total amount of file I/O requested, 2) the actual I/O requests performed by the disk, 3) the total number of I/O events 4) the total number of I/O events that were delayed by sleep calls, 5) the total amount of sleep time, 6) the total runtime of the workload, and 7) the average actual disk I/O rate (total disk I/O's divided by execution time). The results are shown in Table 1.

When comparing the difference between file I/O and disk I/O, the file I/O is equal to the disk I/O for `tar`[3], 14% less for `agrep`, and 233% larger for `compile`. Notice that for the two I/O intensive applications, the overall I/O rate for the application is very close to the target rate.

For the tar application, our mechanism worked fine with the aggressive read ahead used by the file system. For `agrep`, we observed a higher total I/O volume due to small reads being rounded to larger disk pages. The low file I/O

[3] The tar file size is 52 MB.

Metric	Tar	Agrep	Compile
Total File I/O	103.0 MB	50.0 MB	23.3 MB
Total Disk I/O	103.0 MB	58.1 MB	10.0 MB
Total I/O Events	17,430	11,526	3,859
Total Sleep Events	6,928	3,324	1,004
Total Sleep Time	178.0 sec	83.3 sec	29.1 Sec
Total Execution Time	211.2 sec	108.7 sec	70.6 Sec
Average Disk I/O Rate	487 KB/sec	534 KB/sec	141 KB/sec

Table 1: I/O Application Behavior

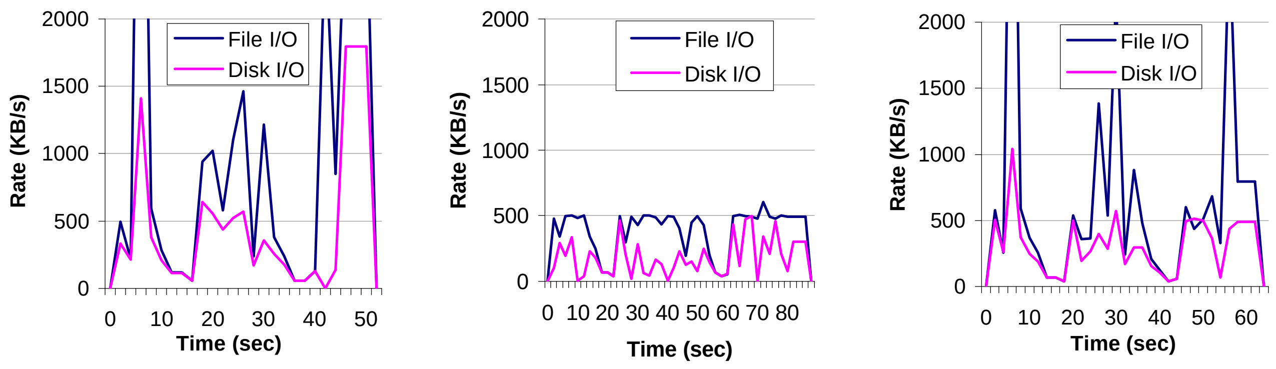
number for `compile`, of course, is due to good buffer cache locality.

There are two potential approaches to recouping this lost bandwidth. The first is to add a hook into the buffer cache to check for a cache miss before adding the I/O to our window, and deciding whether to sleep and how long to sleep. We avoided this path because we wish to avoid kernel source modifications outside of our module whenever possible. We currently keep our entire system as a loadable kernel module, which uses only externally available information such as the system call interface. This would be compromised if we put hooks deeper into the kernel.

A second approach is to use statistics from the `proc` file system to apply a "dilation factor" to our limit calculations. We define the dilation factor as the ratio of file I/O and disk I/O requests. If the ratio is 1.0, each file I/O is being transformed into the same amount of disk activity, i.e. there is no caching or reuse. If the ratio is 0.5, e.g. 100 KB of file I/O is being transformed into only 50 KB of disk I/O, then the limited job is not fully utilizing it's allocated bandwidth. Resources can be used more efficiently by multiplying the file I/O threshold by the inverse of the dilation factor. The disadvantage of this approach is that dynamic caching behavior will lead to time-varying dilation factors, and poor policing. The advantages are better bandwidth utilization, and that the approach can be implemented entirely outside of the kernel.

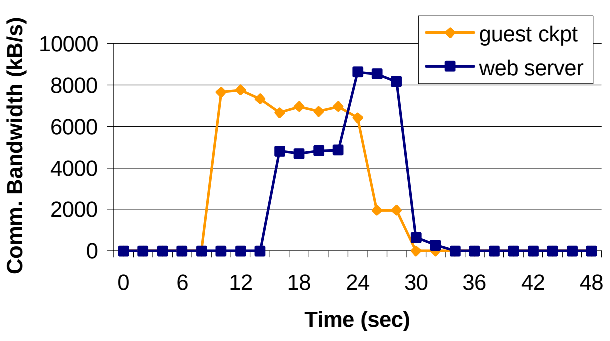
We investigated this approach by adding another field in the I/O rate window to record the resulting disk I/O size. A rolling average of the dilation factor is used to scale the file I/O threshold for future requests.

The full story of the I/O dilation is seen when we look at the time varying behavior of the I/O. Figure 6 shows the average I/O rates for the compile workload. The dark curve of each graph is for the file I/O rate and the light curve for the disk I/O rate. We first ran it without any I/O rate limit. Figure 6(a) shows that file I/O requests resulted in much less disk I/O because many header files were reused from the file buffer cache. The second graph (b) presents the case when we limited the file I/O rate to 500 KB/sec. Notice that although this workload still has considerable hits in the file buffer cache, our mechanism ensured that

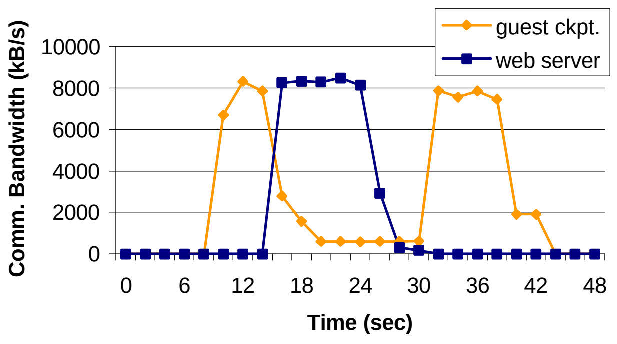


(a) No rate limit (b) File I/O limit (500 KB/s) (c) Disk I/O limit (500 KB/s)

Figure 6: File and Disk I/O Rates for the Compile Workload.



(a)



(b)

Figure 7: Guest job checkpoint vs. host web server

the actual disk I/O rate was less than the target rate of 500KB/sec. The requested I/O rate peaks are higher than our target limit, due to the fact that we average I/O requests over an effective 1.7 second window (as noted above) and we are showing data over a 1 second window in this figure. Figure 6(c) shows the behavior of the compile application when the dilation factor is used to control the disk I/O rate. The curves demonstrate that the application can take advantage of buffer hits while limiting the disk I/O rate to a certain level. The compile application was able to finish in 64 seconds, which is 27 seconds earlier than using file I/O rate policing. Note that the disk I/O rate occasionally peaks over the limit. This is because the dilation factor is derived from past I/O behavior. Any change in the dilation factor over time can cause inaccurate predictions. Overall, however, the actual disk I/O followed the limit quite well.

5. Network I/O policing

Policing network I/O is easier than file I/O because there is no analogue to the file buffer cache or read ahead, which dilate and attenuate the effective disk I/O rate. In this section, we present two applications of network I/O throttling using our rate windows.

5.1 Linger -Longer: Throttling guest processes

Most of the experiments in Section 4 assumed the use of rate windows in a Linger-Longer context. We ran one additional Linger-Longer experiment, this time with network I/O as the target. One of the main complaints about Condor and similar systems is that the act of moving a guest job from a newly loaded host often induces significant overhead to retrieve the application's checkpoint. Further, periodic checkpointing for fault tolerance produces bursty network traffic. This experiment shows that even checkpointing operations are throttled and can be prevented from affecting host jobs.

Figure 7 shows two instances of a guest process moving off of a node because a host process suddenly becomes active. Moving off the node entails writing a 90MB checkpoint file over the network. This severely reduces available bandwidth for the host workload (a web server[4] in this case) in the unthrottled case shown in Figure 7(a). Only after the checkpoint is finished does the web server claim most of the bandwidth.

In the throttled case shown in Figure 7(b), the condor daemon's network write of the checkpoint consumes a majority of the bandwidth only until the host web server starts up. At this point, the system enters throttling mode and the bandwidth available to the checkpoint is reduced to the guest class's threshold. Once the web server becomes idle again, the checkpoint resumes writing at the higher rate.

5.2 Rate-based network clocking

Finally, we look at the use of rate windows to perform an approximation of rate-based clocking of network traffic. Such clocking has been proposed as a method of preventing network contention and improving utilization in transport protocols. Specifically, modifying the TCP protocol stack to send out packets at a preset interval has advantages in 1) avoiding TCP slow-start, 2) preventing burstiness as a result of ACK compression, and 3)

[4] The host process could be any network intensive process such as an FTP or a Web browser.

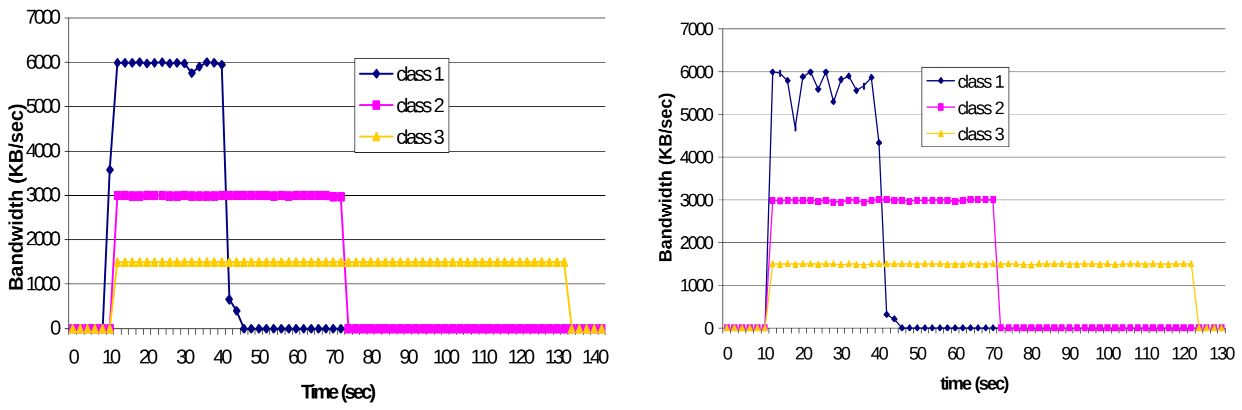


Figure 8: Three web servers: The numbers on the left are for 1.7MB files, for 71KB files on the right.

preventing downstream congestion. With our current placement of hooks high in the kernel, rate windows can only address the third motivation. Note, however, that our implementation is protocol independent, i.e. it works just as well for UDP as for TCP.

Figure 8 shows achieved bandwidth for three Apache web servers running on a single host. Each server is driven by clients that repeatedly request the same file. Hence, all requests but the first are satisfied in the server's cache and the performance of the servers is completely limited by available bandwidth. The total available bandwidth is ~12MB/sec and the three servers are limited to 1.5MB, 3MB, and 6MB, respectively. The maximum bandwidth achieved with the large files is 11.5MB/sec, and 8.6MB/sec with the small files. Hence, the thresholds do not permit the servers to use all of the available bandwidth in the first case, but do in the second.

Note that the deviation from the threshold by the small-file streams (especially the largest stream) is not a failing. In fact, this is a problematic use of rate windows since our guarantees are *not-to-exceed* guarantees, not *at-least* guarantees. Rate windows are actually ideal for this use because congestion problems only arise when bandwidth *exceeds* specific bounds, so the guarantee is of the correct polarity.

A second consequence of this characteristic is that rate windows implicitly smooth bursty traffic. Consider a rate-based stream that, despite the rate-base clocking, encounters temporary congestion and backoff. When the transmissions continue, a straightforward implementation would attempt to "make up" the lost time by transmitting at above the desired rate for some amount of time. This, in turn, could cause more congestion.

With rate windows, the decisions about how long or whether to sleep are based solely on the history contained in the window, which currently contains 100 or five seconds worth of I/O requests, whichever is less. A rate-window-based stream will attempt to make up losses within the window, but "forgets" losses that occur before the window's events. As a result, extra use of bandwidth in order to make up delayed transmission is strictly, and implicitly, bounded by a combination of target bandwidth and window size.

6. Related work

Previous work on exploiting available idle time on workstation clusters used a conservative model that would only run processes when the local user was away from their workstation, and no local processes were runnable. Condor [12], LSF [22], and NOW [2] use variations on a "social contract" to strictly limit interference with local users. However, even with these policies, there is some disruption of the local user when they return since the guest process must be evicted and the local state restored. The Linger-Longer approach permits slightly more disruption of the user, but tries to limit the delay to an acceptable level. One system that used non-idle workstations was the Stealth distributed scheduler [11]. It implemented a priority-based approach to running guest processes. However none of the tradeoffs in how long to run guest processes, or the potential of running parallel programs were investigated.

In the area of operating system support for providing resource management, research and commercial operating systems have provided similar functionality. In IRIX [18], the *Miser* feature provides deterministic scheduling of batch jobs. Miser manages a set of resources, including logical CPUs and physical memory, that Miser batch jobs can reserve and use in preference to interactive jobs. This strategy is almost the opposite of our approach, which promotes interactive jobs.

Aron and Druschel's soft timers [1] provide a way to implement rate-based clocking of network protocols. Although their motivation, avoiding the penalty of TCP slow-start for small file transfers over high delay-bandwidth networks, is different than ours, limiting the fraction of the server's network bandwidth that a single http client or virtual host server gets, both techniques can be used to achieve similar ends.

Also, many have studied general quality of service (QoS) support for server applications. The reservation domains of Eclipse [5], the Software Performance Units of Verghese et al. [20], and Resource Containers [4] can group a set of processes or threads as a unit for resource scheduling. This is similar to our job classes. The Nemesis kernel [15] also provides QoS with rate-based real-time scheduling for I/O as well as CPU. However, those systems are integrated deep into the kernel, while our mechanism resides between the kernel and the user-level I/O library and can be loaded and unloaded at run-time. Our mechanism is light-weight since we do not add any extra queues for resource scheduling. Our mechanism just intercepts resource requests, keeps track of the rate, and puts them into sleep for an appropriate time if the requests seem to exceed the limit. However, our rate windows mechanisms can be used as a light-weight and portable scheduling mechanism to support those concepts.

The idea of regulating traffic rates in the network has been extensively studied. Congestion avoidance schemes such as leaky bucket [19] and its variants [7, 21] use averages over various time intervals to determine which traffic is within its negotiated bandwidth. However, since these approaches are designed for policing traffic at routers, they must drop non-conforming traffic. In contrast, since our approach is at the source, we can delay traffic to enforce bandwidth limits.

The idea of resource partitioning through the use of virtual machines has been popular both in the 1970s [9] as well as in recent projects such as Disco [6]. The key difference is that while virtual machines provide hard isolation of resource between VMs at considerable runtime overhead, our approach is a simple extension to an existing operating system or runtime library.

7. Conclusions and Future Work

We have presented a simple and portable mechanism that allows an operating system to throttle the rate at which disk and network communication is performed. Our experiments demonstrated that we are able to enforce these resource limits on applications with little overhead.

For I/O bound applications, we are able to enforce limits at the physical device level despite the imposition of the buffer cache and disk read-ahead mechanisms. Further, for many applications, we can enforce our limits on the actual disk I/O instead of the file I/O by compensating for the file-to-disk dilation factor. The result is more efficient use of the guest job's allocated bandwidth.

For the network case, we demonstrated that rate windows allow effective bandwidth sharing among communication-bound processes. We also used them to implement policies that protect a host process's access to network resources. This protection is applied to all network accesses by all guest jobs that are running on the local machine, and also to the large network I/O's that occur when such processes try to migrate their address spaces off of the local machine.

Our technique is simple, general purpose, and flexible. One obvious area of future work is to provide a complete study of the ability of the system to handle finer granularity policing of resources by dynamically adjusting the window size. Since our mechanism requires only the ability to monitor and delay user level I/O requests, we could implement our approach in user space libraries.

Finally, we plan to evaluate the overall effectiveness of our resource isolation techniques for a full-scale cycle-stealing system.

References

1. M. Aron and P. Durschel, "Soft Timers: efficient microsecond software timer support for network processing," *SOSP*. Dec. 1999, Kiawah Island, SC, pp. 232-246.

2. R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, "The Interaction of Parallel and Sequential Workloads on a Network of Workstations," *SIGMETRICS*. May 1995, Ottawa, pp. 267-278.
3. A. Atlas and A. Bestavros, "Design and implementation of statistical rate monotonic scheduling in KURT Linux," *Proceedings 20th IEEE Real-Time Systems Symposium*. Dec. 1999, Phoenix, AZ, pp. 272-6.
4. G. Banga, P. Druschel, and J. Mogul, "Resource containers: A new facility for resource management in server systems," *USENIX 3rd Symposium on Operating System Design and Implementation*. October 1999, New Orleans, LA.
5. J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz, "The Eclipse operating system: Providing Quality of Service via Reservation Domains," *USENIX 1998 Annual Technical Conference*. June 1998, New Orleans, Louisiana.
6. E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalabe Multiprocessors," *SOSP*. Oct 1997, pp. 143-156.
7. T. Faber, L. H. Landweber, and A. Mukherjee, "Dynamic Time Windows: packet admission control with feedback," *SIGCOMM*. Sept 1992, pp. 124 - 135.
8. W. C. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "Understanding and improving TCP performance over networks with minimum rate guarntees," *IEEE/ACM Transactions on Networking*, **7**(2), 1999, pp. 173-187.
9. R. P. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer Magazine*, **7**(6), 1974, pp. 34-45.
10. J. S. Heidemann and G. J. Popek, "File-system development with stackable layers," *ACM Trans. Computer Systems*, **12**(1), 1994, pp. 58-89.
11. P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," *International Conference on Distributed Computing Systems (ICDCS)*. May 1991, Arlington, TX, pp. 336-343.
12. M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," *International Conference on Distributed Computing Systems*. June 1988, pp. 104-111.
13. M. W. Mutka and M. Livny, "The available capacity of a privately owned workstation environment," *Performance Evaluation*, **12**, 1991, pp. 269-284.
14. V. N. Padmanabhan and R. H. Katz, "TCP Fast Start: A Techniques for Speeding Up Web Transfers," *IEEE GLOBECOMM*. Nov. 1998, Sydney, Australia, pp. 41-46.
15. D. Reed and R. Fairbairns, *The Nemesis KernelOverview*, http://citeseer.nj.nec.com/reed97nemesis.html, May 20, 1997.
16. K. D. Ryu and J. K. Hollingsworth, "Exploiting Fine Grained Idle Periods in Networks of Workstations," *IEEE Transactions on Parallel and Distributed Computing*, **11**(7), 2000.
17. K. D. Ryu, J. K. Hollingsworth, and P. J. Keleher, "Mechanisms and Policies for Supporting Fine-Grained Cycle Stealing," *ICS*. June 1999, Rhodes, Greece, pp. 93-100.
18. SiliconGraphics, *IRIX 6.4 Technical Brief*, http://www.sgi.com/software/irix6.5/techbrief.pdf , 1998.
19. J. S. Turner, "New Directions in Communications (or Which Way to the Information Age?)," *IEEE Communications Magazine*, **24**(10), 1986, pp. 8-15.
20. B. Verghese, A. Gupta, and M. Rosenblum, "Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors," *ASPLOS*. Oct. 1998, San Jose, CA, pp. 181-192.
21. L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks," *SIGCOMM*. Sept. 1990, pp. 19-29.
22. S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," *SPE*, **23**(12), 1993, pp. 1305-1336.