Rate Windows for Efficient Network and I/O Throttling

Kyung D. Ryu, Jeffrey K. Hollingsworth, and Peter J. Keleher

Dept. of Computer Science
University of Maryland
{kdryu,hollings,keleher} @cs.umd.edu

This paper proposes and evaluates a new mechanism for I/O and network rate policing. The goal of the proposed system is to provide an simple, yet effective way to enforce resource limits on target classes of jobs in a system. The basic approach is useful for several types of systems including running background jobs on idle workstations, and providing resource limits on network intensive applications such as virtual web server hosting. Our approach is quite simple, we use a sliding window average of recent events to compute the average rate for a target resource. The assigned limit is enforced by forcing application processes to sleep when they issue requests that would bring their resource utilization out of the allowable profile. Our experimental results that show that we are able to provide the target resource limitations within a few percent, and do so with no measurable slowdown of the overall system.

Contact author:

Dr. Peter Keleher
Computer Science Department
A. V. Williams Bldg.
University of Maryland
College Park, MD 20742-3255
301 405-0345
Fax: 301 405-6707
keleher@cs.umd.edu

Rate Windows for Efficient Network and I/O Throttling

Kyung D. Ryu, Jeffrey K. Hollingsworth, and Peter J. Keleher

Dept. of Computer Science
University of Maryland
{kdryu,hollings,keleher}@cs.umd.edu

Abstract

This paper proposes and evaluates a new mechanism for I/O and network rate policing. The goal of the proposed system is to provide an simple, yet effective way to enforce resource limits on target classes of jobs in a system. The basic approach is useful for several types of systems including running background jobs on idle workstations, and providing resource limits on network intensive applications such as virtual web server hosting. Our approach is quite simple, we use a sliding window average of recent events to compute the average rate for a target resource The assigned limit is enforced by forcing application processes to sleep when they issue requests that would bring their resource utilization out of the allowable profile. Our experimental results that show that we are able to provide the target resource limitations within a few percent, and do so with no measurable slowdown of the overall system.

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 [15], the rate windows give unprecedented control over the resource use of user applications. More specifically, rate windows is 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. Like CPU scheduling, this modification should not affect the memory allocation (or page replacement) between processes in the same class.

This ability 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 [12], 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.

However, most such policies waste many opportunities to exploit cycles because of overly conservative estimates of resource contention. Our *linger-longer* approach [14] 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 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. Rating limiting is useful both for managing resource allocations of competing users (such as virtual hosting of web servers) and can be used for rate-based clocking of network protocols as a means of improving the utilization of networks with high bandwidth-delay products [7, 13].

The rest of this paper is organized as follows. Section 2 describes the implementation of rate windows and evaluates its use with micro-benchmarks. Section 3 reviews the Linger-Longer infrastructure, motivates the use of rate windows for Linger-Longer. In particular, we show that a significant class of guest applications is still able to affect host processes via network and I/O contention. Further we show that there is no general way to prevent this using CPU and memory policing that still allows the guest to make progress. 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 **Error! Reference source not found.** concludes.

2. CPU and memory policing

Before discussing rate windows, we place this work in the context of the Linger-Longer resource-policing infrastructure [14]. The Linger-Longer infrastructure is based on the thesis that current Condor-like [11] 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 machine even when resource-intensive host jobs start up. 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 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 [15].

We verified that the scheduler reschedules processes any time a host process unblocks while a guest process is running. This is the default behavior on Linux, but not on many BSD derived operating systems. One potential problem of our strict priority policy is that it could cause priority inversion. Priority inversion occurs when a higher priority process is not able to run due to a lower priority process holding a shared resource. However, this is not possible in our application

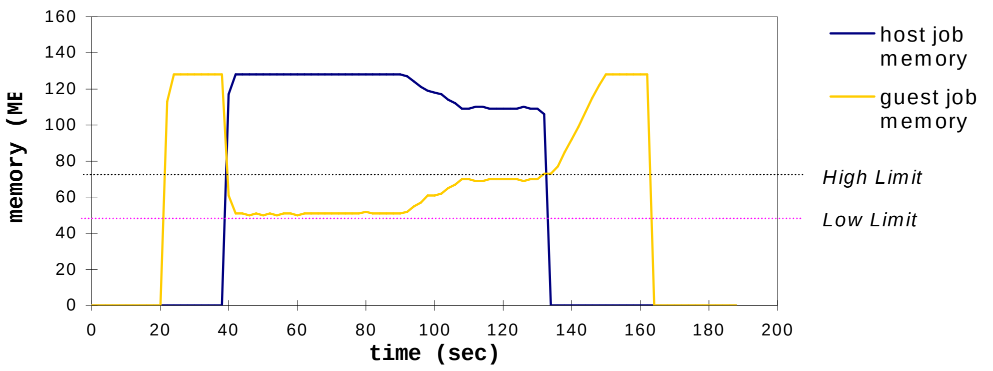


Figure 1: Threshold validations – Low and high thresholds are set to 50MB and 70 MB. At time 90, the host job becomes I/O-bound. Host process acquires 150 MB when running without contention, guest process acquires 128 MB without contention. Total available memory is 179 MB.

domain because guest and host processes do not share locks, or any other non-revocable resources.

Our second mechanism limited guest consumption of memory resources. Unfortunately, memory is more difficult to deal with than the CPU. The cost of reclaiming the processor from a running process in order to run a new process consists only of saving processor state and restoring cache state. 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. The simple solution to this problem is to permanently reserve physical memory for the host processes. The drawback is that many guest processes are quite large. Simulations and graphics rendering applications can often fill all available memory. Hence, not allowing guest processes to use the majority of physical memory would prevent a large class of applications from taking advantage of idle cycles.

We therefore decided not to 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. Essentially, the page replacement policy prefers to evict a page from a host process if the total number of physical pages held by the guest process is less than the low threshold. The replacement policy defaults to the standard clock-based pseudo-LRU policy up until the upper threshold. Above the high threshold, the policy prefers to evict a guest page. The effect of this policy is to encourage guest processes to steal pages from host processes until the lower threshold is reached, to encourage host processes to steal from guest processes above the high threshold, and to allow them to compete evenly in the region between the two thresholds. However, the host priority will lead to the number of pages held by the guest processes being closer to the lower threshold, because the host processes will run more frequently.

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.

We validated our memory threshold modifications by tracking the resident memory size of host and guest processes for two CPU-intensive applications with large memory footprints. The result is shown in Figure 1. The chart shows memory competition between a guest and a host process. The application behavior and memory thresholds shown are not meant to be representative, but were constructed to demonstrate that the memory thresholds are strictly enforced by our modifications to Linux's page replacement policy. The guest process starts at time 20 and grabs 128MB. The host process starts at time 38 and quickly grabs a total of 128 MB. Note that the host actually touches 150 MB. It is prevented from obtaining all of this memory by the low threshold. Since the guest process' total memory has dropped to the low threshold, all replacements come from host pages. Hence, no more pages can be stolen from the guest. At time 90, the host process turns into a highly I/O-bound application that uses little CPU time. When this happens, the guest process becomes a stronger competitor for physical pages, despite the lower CPU priority, and slowly steals pages from the host process. This continues until time 106, at which point the guest process reaches the high threshold and all replacements come from its own pages. For this experiment, we deliberately set the limits very high to demonstrate the mechanism. 5/10% of total memory are very acceptable guest job memory limits for most of cases. However, these values can be adapted at run time to meet the different requirements of applications.

3. Rate Windows

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

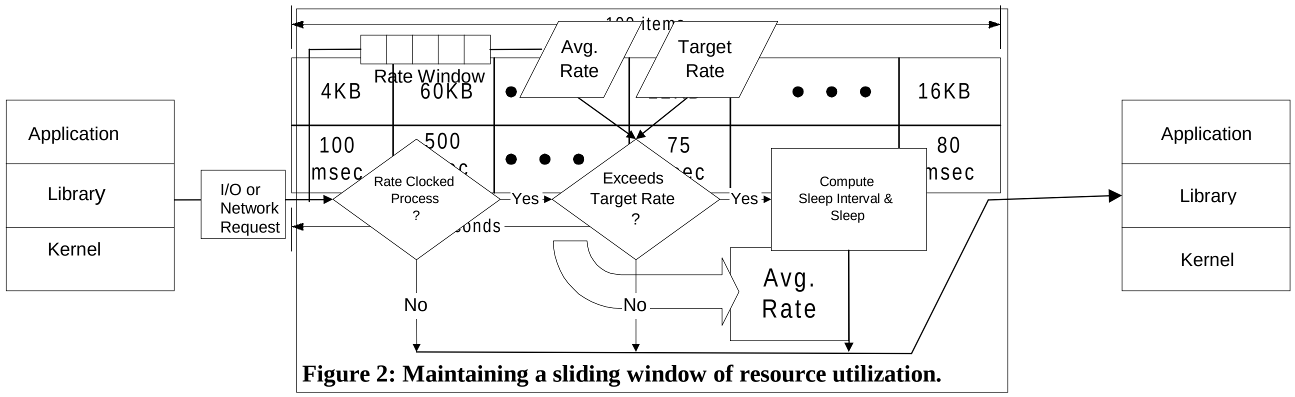


Figure 2: Maintaining a sliding window of resource utilization.

Figure 3: Policing I/O Requests.

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 2). 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 before arriving at these constants, but 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 currently trigger our mechanism via hooks placed high in the kernel, at each of the kernel calls that implement I/O and network communication: `read()`, `write()`, `send()`, etc. Each hook calls `rate_check()` with process ID, I/O length, and I/O type. 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. We maintain a window of up to 100 recent events. However, to prevent using too old of information, we limit the sliding window to a fixed interval of time (currently 5 seconds).

Define B_w, the *window bandwidth*, as the total amount of I/O in the window's operations, including the new operation. 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:

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

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

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

This process is illustrated graphically in Figure 3. 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 is will not affect application execution since for 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[2816].

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. Since many active Linux users build their own customized kernels, our mechanisms could easily be patched into existing installations by end users. This is important because most PCs are deployed on people's desks, and cycle-stealing approaches are probably more applicable to desktop environments than to server environments. Also since our mechanism simply requires the ability to intercept I/O calls, it would be easy to implement as a loadable kernel modules on systems that defined an API to intercept I/O calls. Windows 2000 (nee Window NT) and the stackable filesystem [9] 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 KURT Real-time Linux project [3].

[2816] For UDP this is not a problem since the max user level request is constrained by the network's MTU.

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 doesn't 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.

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. The completion time of the tar application with rate windows enabled was less than the variation between consecutive runs of the experiment. This was expected, as there are no computationally expensive portions of the algorithm. Note that this experiment does not account for the system cost of extra context switches caused by sleeping guest jobs.

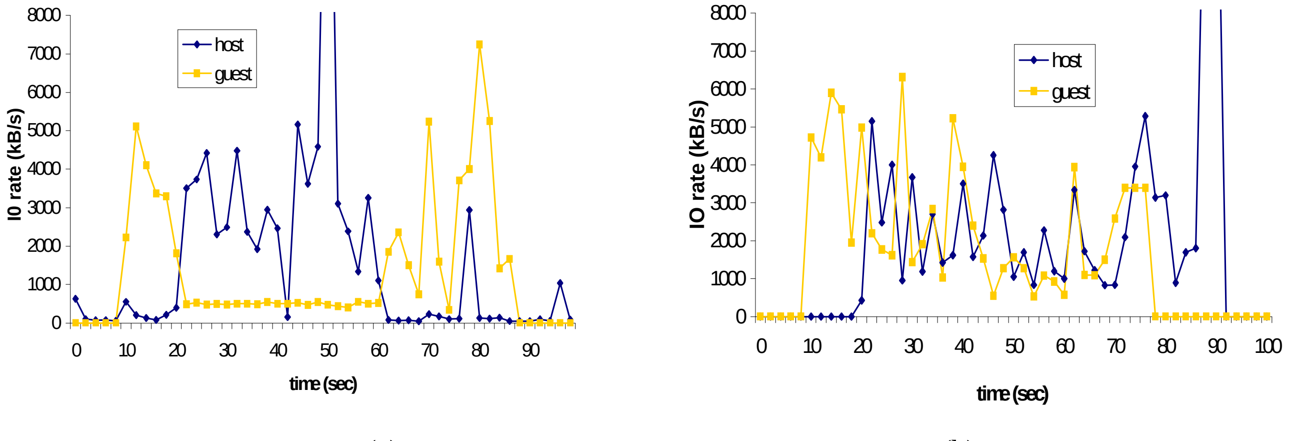


Figure 4: File I/O of competing tar applications with (left) and without (right) file I/O policing.

Second, we ran two instances of tar, one as a guest job and one as a host job. Figure 4a represents a run with throttling enabled, and Figure 4 shows a run without throttling. 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.

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 jobs finish later). This is a design decision: our goal is to prevent undue degradation of unconstrained host job performance regardless of the effect on any guest jobs.

We look at the behavior of one of the tar processes in more detail in Figure 5. The point of this figure is that despite the frequent and varied file I/O calls, and despite 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.

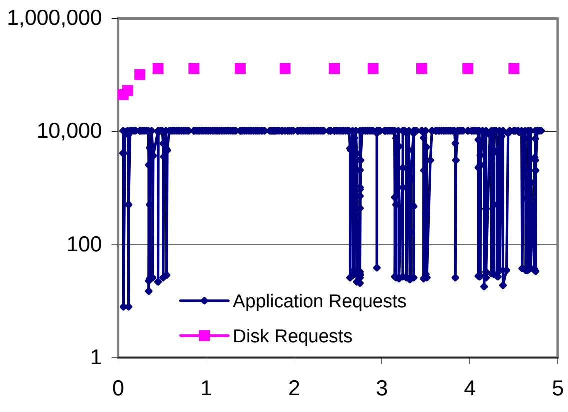
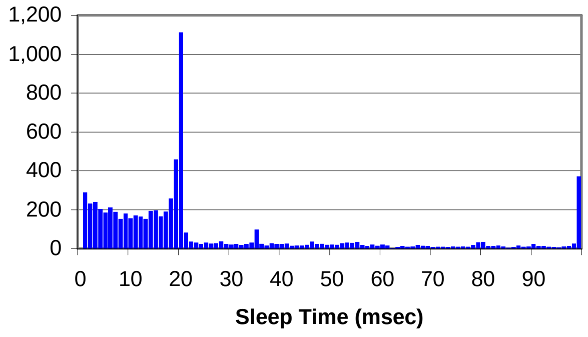
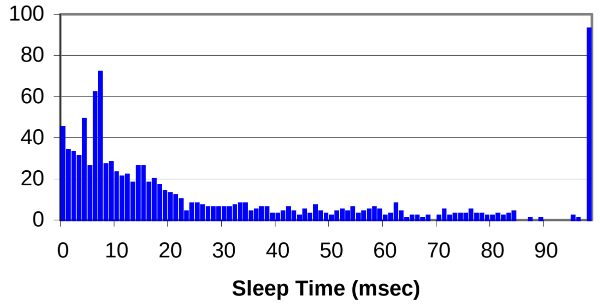


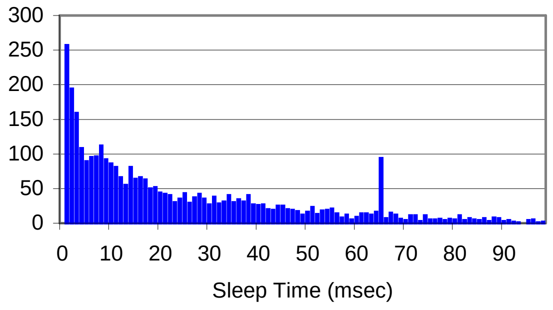
Figure 5: I/O sizes vs. time for `tar`



(a) Tar



(b) Compile Workload



(c) Agrep

Figure 6: Distribution of Sleep Times for Tar program.

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 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 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 6. We have omitted those events that have no delay since their frequency completely dominates the rest of the values. Figure 6(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 the must common sized I/O request of 10K bytes to be limited to 500 KB/sec. Figure 6(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 6(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.

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 I/O Rate	487 KB/sec	534 KB/sec	141 KB/sec

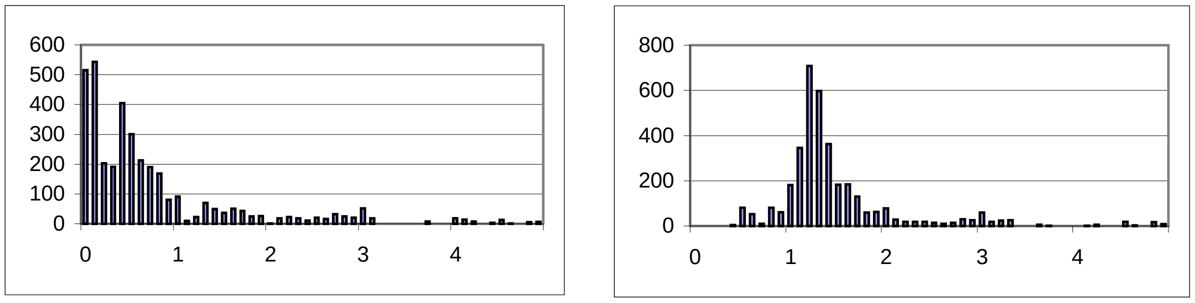
Table 1: I/O Application Behavior

Fourth, we examine the relationship between file I/O and disk I/O. 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 bring speculatively bring in disk blocks that are never referenced. File I/O can also attenuate due to buffer cache hits, which is a consequence on 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.

Looking first at the difference between file I/O and disk I/O, note that file I/O is equal to the disk I/O for `tar`, 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.

We did not observe poor read-ahead behavior in our experiments; `agrep`'s behavior is due to small reads being rounded to larger disk pages. The low file I/O number for `compile`, of course, is due to good buffer cache locality.

Since the temporal extent of our window automatically adapts based on the effective I/O rate (due to the limit of 100 items), we wanted to look at how the size of this window changed during the execution of a program. Figure 8 shows the distribution of the effective window size for the compilation workload. The bar chart on the left shows the effective window size (in seconds) for the workload when it is run without any I/O rate control. The curve on the right shows the same information when I/O rate control is enabled. In both cases the effective window size is much less than the upper limited of 5 seconds. The average size for the limited case is 0.98 seconds, and 1.71 seconds for the limited case. <what is the conclusion for this ??>



(a) (b)

Figure 8: Comparison of Effective Window Size (Compilation workload).

The full story of the I/O dilation is seen when we look at the time varying behavior of the I/O. Figure 7 shows the one-second average I/O rates for the compile workload. Notice that although this workload has considerable hits in the file buffer cache, our mechanism ensured that 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 we average I/O requests over a 5 second window and we are showing data over a 1 second window in this figure.

Although we do not claim that our set of I/O-intensive applications is representative, our experiments support our intuition that file I/O dilation is not a problem. Rather, the main concern is that of lost opportunity. Consider an example where we would like to share all available bandwidth equally between two applications. We can set thresholds for each application at half of the maximum achievable disk bandwidth. However, good buffer cache locality would mean that file I/O at this rate would generate less, possibly much less, disk I/O. Such attenuation represents unused bandwidth.

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. We deliberately have not taken this path because we wish to keep our system at as high a level as possible. We could currently move our entire system into `libc` without loss of functionality or accuracy. This would be compromised if we put hooks deeper into the kernel.

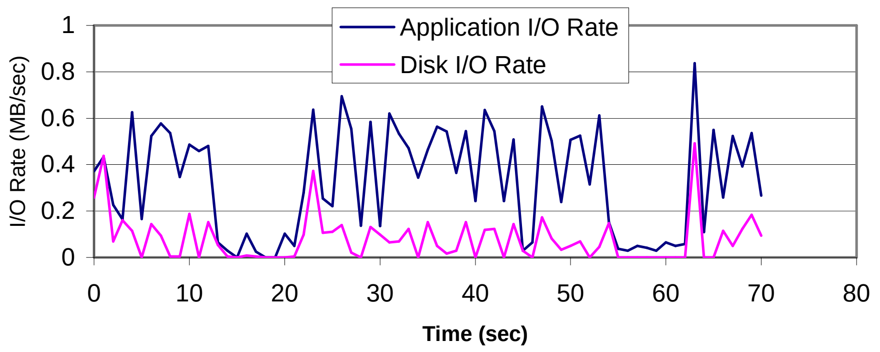
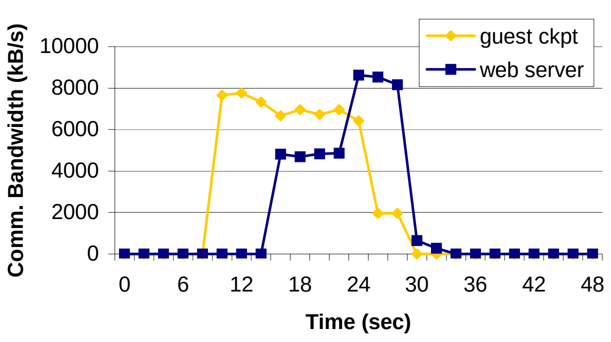
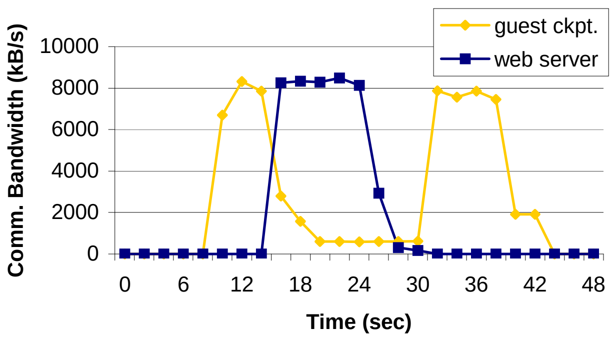


Figure 7: I/O Rates for the Compile Workload.



(a)



(b)

Figure 9: Guest job checkpoint vs. host web server

A second approach is to use statistics from `proc` file system to apply a "normalization factor" to our limit calculations. Of necessity, this would be inexact. The advantage is that it can be implemented entirely outside of the kernel. We are currently pursuing this approach, but the mechanism is not yet in place.

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. Hence, network bandwidth is a somewhat better target for our current implementation of rate windows than file I/O. Since contention for network resources is probably more common than disk bandwidth contention, this preference is fortuitous.

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 more 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 the checkpoint is throttled and can be prevented from affect host jobs.

Figure 9 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[2817] in this case) in the unthrottled case shown in Figure 9a. Only after the checkpoint is finished does the web server claim most of the bandwidth.

In the throttled case shown in Figure 9b, 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

[2817] The host process could be any network intensive process such as an FTP or a Web browser.

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) 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.

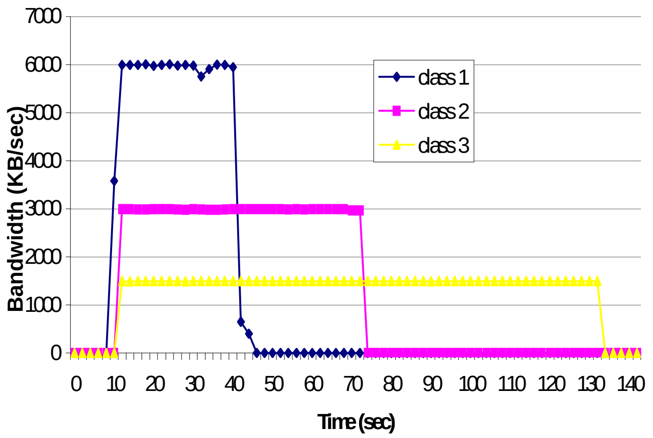
Figure 10 shows achieved bandwidth for three Apache web servers run on a single host. Each server is driven by clients that repeatedly request the same file. Hence, all requests but the first are satisfied in the server's cache and the performance of the servers is completely limited by available bandwidth. The total available bandwidth is ~11MB/sec with the smaller problem size, the three servers are limited to 1.5MB, 3MB, and 6MB, respectively. The maximum bandwidth achieved with the large files is 11.5MB, and 8.6MB 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 is 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 are strictly, and implicitly, bounded by a combination of target bandwidth and window size.

6. Related work



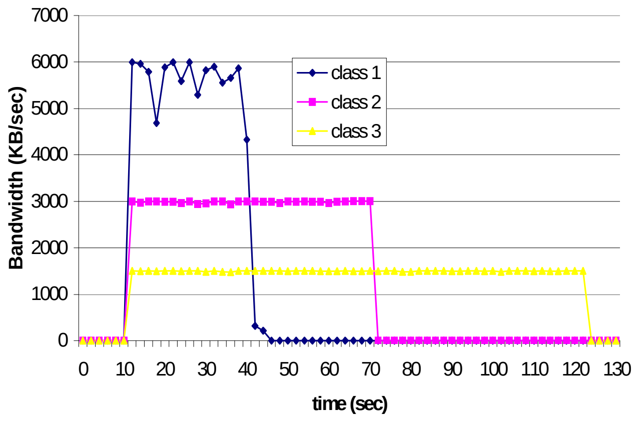


Figure 10: Three web servers: The numbers on the left are for 1.7MB files, for 71KB files on the right.

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 [11], LSF [19], 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 [10]. 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 the IRIX [16], 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. For example, soft timers could be used to limit the rate at which data is sent from a server by setting the soft timer longer than the optimal rate for the link. Likewise, our rate throttling mechanism could be used to provide rate-based clocking of packets by limiting a connection to a packet rate that matches the capacity of the bottleneck link between the two communicating parties.

The idea of regulating traffic rates in the network has been extensively studied. Congestion avoidance schemes such as leaky bucket [17] and its variants [6, 18] 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 [8] as well as in recent projects such as Disco [5]. 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 on runtime library.

7. Discussion

As mentioned in Section 2, researchers have long recognized that idle machines represent a vast untapped resource. Two long-term trends are increasing this opportunity. First, increased connectivity across the Internet allows for utilization of resources in much wider domains. Second, new software technologies are making it possible to better exploit heterogeneous sets of workstations. For example, new Java compilers promise to allow write-once/run-anywhere applications to perform within a small factor of the best host-code compilers for traditional languages. These two trends vastly increase the set of candidates for wide-area computing.

Systems like Condor [11] exploit this opportunity by allowing guest processes to run on idle participating machines. Existing systems focus on coarse-grained idle periods when users are away from their workstations. Returning users, or the start of any significant local processes, cause guest processes to be migrated off the local machine in order to avoid impacting the local user. By providing better resource management, we are able to be more aggressive in our use of found resource and thus expand the opportunities to run guest jobs. In addition, by providing a rate-limited way to migrate jobs off nodes we can reduce one of the most annoying types of disruption provided by cycle stealing systems.

Our placement of hooks high in the kernel has two advantages. First, we are above the protocol stack and so transparently catch all protocols. Second, the implementation could easily be adapted to be part of the C-runtime library, or even of a specific application. For example, consider the problem of running a collection of web servers running in a virtual hosting environment. The ability of our system to provide fixed bandwidth allocations to each virtual host could be implemented by patching the web servers. Recall from Section 3 that all of the necessary information can be collected by observing the execution of network system calls. A user-level implementation would intercept these calls and use a shared memory region to store the history information. Techniques such as online binary editing [4] could be used to implement rate windows in user-space without access to the application source code.

8. Conclusions and Future Work

We have presented a simple strategy to allow an operating system to throttle the rate at which disk and network communication is performed. Our technique is simple and general purpose. By changing parameters such as the window size, it is possible to adapt the granularity of the approach. 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, or as loadable kernel modules.

Our experiments demonstrated that we are able to enforce resource limits on applications. For I/O bound applications, we showed that despite the fact our mechanism is located close to the application level I/O requests we are able to enforce limits at the physical device level despite the imposition of the buffer cache and disk read ahead between our mechanism and the physical device. An area of future work is to implement a feedback mechanism to normalize the policing of I/O requests based on buffer cache hit rates.

For the network case, we demonstrated that rate windows allow effective bandwidth sharing and protection of network resources to allow guest jobs to run on (and migrate off) workstations without causing interference to host processes.

Finally, we plan to evaluate the overall effectiveness of our resource isolation techniques for a full scale cycle-stealing system. <what else to mention here??>

9. Bibliography

1. M. Aron and P. Durschel, "Soft Timers: efficient microsecond software timer support for network processing," *SOSP*. Dec. 1999, Kiawah Island, SC, ACM, 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. B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *Journal of Supercomputing Applications (to appear)*, 2000.

5. E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalabe Multiprocessors," *SOSP*. Oct 1997, pp. 143-156.
6. T. Faber, L. H. Landweber, and A. Mukherjee, "Dynamic Time Windows: packet admission control with feedback," *SIGCOMM*. Sept 1992, pp. 124 - 135.
7. 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.
8. R. P. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer Magazine*, **7**(6), 1974, pp. 34-45.
9. J. S. Heidemann and G. J. Popek, "File-system development with stackable layers," *ACM Trans. Computer Systems*, **12**(1), 1994, pp. 58-89.
10. P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," *International Conference on Distributed Computing Systems (ICDCS)*. May 1991, Arlington, TX, pp. 336-343.
11. M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," *International Conference on Distributed Computing Systems*. June 1988, pp. 104-111.
12. M. W. Mutka and M. Livny, "The available capacity of a privately owned workstation environment," *Performance Evaluation*, **12**, 1991, pp. 269-284.
13. 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.
14. K. D. Ryu and J. K. Hollingsworth, "Exploiting Fine Grained Idle Periods in Networks of Workstations," *IEEE Transactions on Parallel and Distributed Computing (to appear)*, 2000.
15. 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.
16. SiliconGraphics, *IRIX 6.4 Technical Brief*, http://www.sgi.com/software/irix6.5/techbrief.pdf, , .
17. J. S. Turner, "New Directions in Communications (or Which Way to the Information Age?)," *IEEE Communications Magazine*, **24**(10), 1986, pp. 8-15.
18. L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks," *SIGCOMM*. Sept. 1990, pp. 19-29.
19. 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.