Resource Policing to Support Fine-Grain Cycle Stealing in Networks of Workstations

Kyung Dong Ryu, *Member*, *IEEE Computer Society*, and
Jeffrey K. Hollingsworth, *Senior Member*, *IEEE*

Abstract—This paper presents the design, implementation, and performance evaluation of a suite of resource policing mechanisms that allow guest processes to efficiently and unobtrusively exploit otherwise idle workstation resources. Unlike traditional policies that harvest cycles only from unused machines, we employ fine-grained cycle stealing to exploit resources even from machines that have active users. We developed a suite of kernel extensions that enable these policies to operate without significantly impacting host processes: 1) a new starvation-level CPU priority for guest jobs, 2) a new page replacement policy that imposes hard bounds on physical memory usage by guest processes, and 3) a new I/O scheduling mechanism called rate windows that throttle guest processes' usage of I/O and network bandwidth. We evaluate both the individual impacts of each mechanism, and their utility for our fine-grain cycle stealing.

Index Terms—Grid computing, cluster computing, networks of workstations, parallel computing, resource scheduling, cycle stealing.

1 Introduction

This paper investigates local mechanisms and scheduling policies that allow guest processes to efficiently exploit otherwise idle workstation resources. The opportunity for harvesting cycles in idle workstations has long been recognized [1], 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. Two long-term trends are increasing this opportunity. First, increased connectivity across the Internet allows for utilization of resources in much wider domains. Recent integration efforts in Grid computing [2], [3] further expand both the application domain and the resource domain for cycle stealing beyond institution or administration boundaries. 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 the Wisconsin Condor [4] system 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. Despite the increased availability of idle machines thanks to computational Grid technologies, increased resource demand by large-scale applications necessitates more efficient use of available resources in host machines. In addition, process migrations, which are often required in previous cycle stealing systems, are much more costly when moving processes and data across institutional or local network boundaries in a Grid environment.

The thesis of this paper is that running guest processes only when host machines are idle wastes many opportunities to exploit available resources because of overly conservative estimates of resource contention. We show that the potential negative impact of guest processes can be prevented through the use of a few simple modifications to existing kernel policies. We have developed a strict priority scheduling system that ensures that local processes receive priority in four major resources: processing cycles, memory, I/O, and network bandwidth. This paper describes these mechanisms[1] and presents both a microbenchmark study to demonstrate their efficacy, and an application-oriented workload study to show the overall impact of our policies on typical interactive workloads.

The resulting systems are suitable for use with Linger-Longer [6], [7] policies. Linger-Longer delays migrating guest processes off of machines 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 new scheduling policies are able to effectively use these idle periods in a way that does not delay much the activity of a workstation's primary user.

We presented the design of Linger-Longer in a previous paper. This simulation study showed the potential of our approach to improving the throughput of sequential compute-bound processes. In trace data collected from a variety of organizations, we showed that more than 75 percent of the time machines have CPU utilization less than 10 percent. We also showed via simulation that we could improve the throughput of a compute bound batch workload by 60 percent

- K.D. Ryu is with the Computer Science and Engineering Department, Arizona State University, Tempe, AZ 85287. E-mail: kdryu@asu.edu.
- J.K. Hollingsworth is with the Computer Science Department, University of Maryland, College Park, MD 20742. E-mail: hollings@cs.umd.edu.

Manuscript received 12 July 2002; revised 19 Aug. 2003; accepted 13 Jan. 2004.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 116939.

1. Preliminary work on some of these mechanisms has been presented in [5].

compared with the scheduling policies used by the Wisconsin Condor system [4] and the Berkeley NOW project [8].

This paper presents the design, implementation and performance evaluation of mechanisms and policies that allow the use of Linger-Longer on collections of Linux workstations. Section 2 reviews the Linger-Longer policy and summarizes our previous simulation results. Section 3 describes our new CPU scheduling policy and its implementation in the Linux operating system. Section 4 presents a novel virtual memory page replacement policy to protect a host job's memory. In Section 5, we describe rate windows, a mechanism to police the I/O and network bandwidth usage of guest jobs. To validate each of the proposed policies and mechanisms, we conducted microbenchmarks. The results are presented in each section. Section 6 evaluates the efficacy of our policies and mechanisms as a whole in a real cycle stealing environment. Section 7 reviews related work in the field and, finally, Section 8 concludes this paper.

2 Fine-Grain Cycle-Stealing

This section introduces the concept of fine-grained cycle stealing, the Linger-Longer approach to realizing it, and the requirements that this approach imposes on local schedulers. The key feature of fine-grained cycle stealing is to exploit brief periods of idle processor cycles while users are either thinking or waiting for I/O events. Once we have a mechanism that can take advantage of these short idle periods, the longer idle periods exploited by previous systems can be handled automatically. We refer to the processes run by the workstation owner as host processes, and those associated with fine-grained cycle stealing as guest processes.

In order to make fine-grained cycle-stealing work, we must limit the resources used by guest processes and ensure that host processes have priority over them. Guest processes must have close to zero impact on host processes in order for the system to be palatable to users. Achieving that goal requires a scheduling policy that gives absolute priority to host processes over guest processes, even to the point of starving guest processes. This also implies the need to manage the memory, I/O, and network bandwidth via a priority.

A key question in evaluating the overhead of priority-based preemption is the time required to switch from the guest process to the host process. There are three significant sources of delay in saving and restoring the context of a process:

1. The time required to save registers state.
2. The time required (via cache and TLB misses) to reload the process's cache and TLB state.
3. The time to reload the working set of virtual pages into physical page frames.

We defer discussion of the third overhead until Section 4. On current microprocessors, the time to restore cache and TLB state dominates the register restore time. It has been reported that the time to restore a cache entry requires 12 to 200 cycles [9]. The time to restore a single TLB entry requires 10 to 100 instructions with software managed TLBs and much less with hardware managed TLBs [10]. The simulations in our previous work [7] showed that if the effective context-switch time is 100 microseconds or less, the overhead of this extra context-switch is less than 2 percent. With host CPU loads of less than 25 percent, host process slowdown remains under 5 percent, even for effective context switch times of up to 500 microseconds.

In addition, our simulations of sequential processes showed that a linger-based policy would improve average process completion time by 47 percent compared with previous approaches. Based on job throughput, the Linger-Longer policy provides a 50 percent improvement over previous policies. Likewise, our Linger-Forever policy (i.e., disabling optional migrations) permits a 60 percent improvement in throughput. For all workloads considered in the study, the delay, measured as the average increase in completion time of a CPU request, for host (local) processes was less than 0.5 percent.

Previous systems automatically migrate guest processes off of nonidle machines in order to ensure that guest processes do not interfere with host processes. A key idea of our fine-grained cycle stealing approach is that migration of a guest process off of a node is optional. Guest processes can often coexist with host processes without significantly impacting the performance of the latter, or starving the former. This effect will be more significant when processes need to migrate beyond institution or local network boundaries, such as when using Grid computing technologies.

To support our new cycle stealing policy, we need mechanisms to police the resource usage by guest processes. In later sections, we investigate the existing priority scheme and propose a suite of new resource policing mechanisms for four major resources: CPU, memory, I/O, and network bandwidth.

One concern with some of these mechanisms is the need for kernel modifications. In general, it is much harder to gain acceptance for software that requires kernel modifications. However, we feel that modest kernel modifications are a reasonable burden for two reasons. First, we are using the Linux operating system as an initial implementation platform, and many software packages for Linux already require kernel patches to work. Second, the relatively modest kernel changes required could be implemented on stock kernels using the KernInst technology [11], which allows fairly complex customizations of a UNIX kernel at runtime via dynamic patching.

3 CPU Policing: Starvation-Level CPU Priority

We need mechanisms to make host processes always have a higher CPU priority than guest processes. In other words, the guest jobs need to be preempted as soon as a host job becomes ready to execute. Also, a guest process should not preempt or prevent host processes from running for any reason.

We first investigated the impact of simply using the UNIX nice command to provide CPU priorities. To do this, we constructed a compute bound test program that simply ran an empty loop a fixed number of iterations. We ran two copies of this process. The first simulates a host process by running with the default nice value (0), and the other simulates a guest process by running at the lowest possible priority, nice level -19. The CPU utilizations resulting from this experiment for four different versions of UNIX are shown in Table 1. The table shows the percent of the processor that each process received. Except when running under OSF-1, the guest process received a significant

TABLE 1
CPU Utilization with Single Host and Guest (Niced at Level 19) Processes

OS	Host	Guest
Solaris (SunOS 5.5)	84%	15%
Linux (2.0.32)	91%	8%
OSF1	99%	0%
AIX (4.2)	60%	40%

```
While (1) {
  If exists p such that p.state = RUNNABLE
    Foreach process p
      p.quanta = 20 + p.niceLevel + 1/2 * p.quanta;
  While exists a process p
    such that (p.state = RUNNABLE) and (p.quanta > 0)
      Select p with largest p.quanta;
      Decrement p.quanta;
      Run p;
}
```

Fig. 1. Original Linux scheduler.

amount of processing time (ranging from 8 percent to 40 percent). This simple experiment demonstrates the need for our more sophisticated priority mechanism.

First, we investigated the Linux CPU scheduler to understand why a higher priority process loses some CPU cycles. The scheduler chooses a process to run by selecting the ready process with the highest runtime priority, where the runtime priority can be thought of as the number of 10ms time slices held by the process. The runtime priority is initialized from a static priority derived from the nice level of the process. Static priorities range from -19 to +19, with +19 being the highest.[2] New processes are given $20 + p$ slices, where p is the static priority level. The process chosen to run has its store of slices decremented by one. Hence, all runnable processes tend to decrease in priority until no runnable processes have any remaining slices. At this point, all processes are reset to their initial runtime priorities. Blocked processes receive an additional credit of half of their remaining slices. For example, a blocked process having 10 time slices left will have 20 slices from an initial priority of zero, plus five slices as a credit from the previous round. This feature is designed to ensure that compute-bound processes do not receive undue processor priority compared to I/O bound processes. The algorithm is summarized in Fig. 1.

This scheduling policy implies that processes with the lowest priority (nice -19) will be assigned a single slice during each round, while normal processes consume 20 slices. When running two CPU-bound processes, where one has normal priority and the other is niced to the minimum priority, -19, the latter will still be scheduled 5 percent of the time. While this degree of processor contention might or might not be visible to a user, running the process could still cause contention for other resources, such as memory.

2. Nice priorities inside the kernel have the opposite sign of the nice values seen by user processes.

```
While (1) {
  If exists p such that p.state = RUNNABLE
    Foreach process p where is a host process
      p.quanta = 20 + p.niceLevel + 1/2 * p.quanta;
  While exists p such that p.state = RUNNABLE
    and p.quanta > 0 and p.priority = HOST
      Select p with largest p.quanta;
      Decrement p.quanta;
      Run p;
  If not (exists p such that p.state = RUNNABLE and p.priority =
    HOST) and exists q such that q.state = RUNNABLE and priority
    = GUEST
      Run q;
}
```

Fig. 2. Modified Linux scheduler to support starvation-level CPU priority.

We implemented a new guest priority in order to prevent 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. The modified scheduling algorithm is shown in Fig. 2.

Second, 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. This case will be rare and the blocking time will be very short in our application domain because guest and host processes usually do not share locks, or any other nonrevocable resources and locks for shared kernel resources are typically held for a very short time.

We first validated our scheduling modifications by comparing the CPU utilization of a CPU-intensive guest process competing with that of a host process for three different scheduling policies. Our independent variable is the percent utilization of the host process in the absence of any competing processes. The CPU-intensive guest process is representative of typical guest processes, such as scientific simulations, decision support (data mining), and graphics rendering. This process also provides us with a worst-case (in terms of contention for the CPU) test of scheduling policies.

Fig. 3 shows the resulting behavior. Ideally, the CPU utilization of the host processes would track linearly with the utilization of the job in isolation. The "equal" lines show the default case where guest processes are treated identically to host processes. The "nice-h" line shows that host process utilization is unaffected by the presence of a niced guest process up to approximately 90 percent utilization. The drop-off at this point corresponds to the 91 percent limit shown for Linux in Table 1. Note that "linger-h," included for comparison, accurately tracks expected utilization up to 99 percent. The data shows that a guest process is unable to significantly interfere with CPU utilization of a host process with our CPU

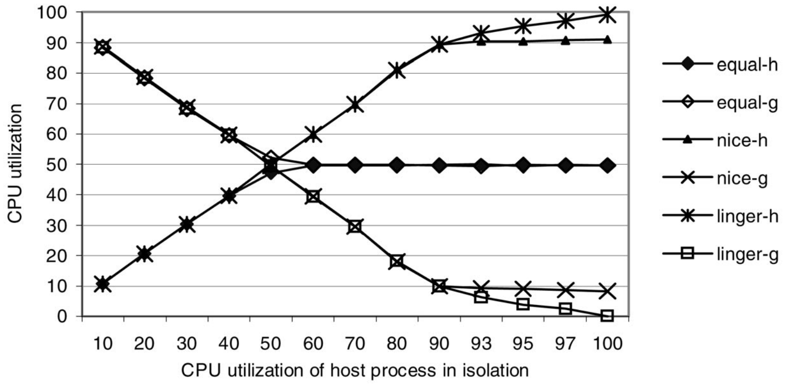


Fig. 3. CPU utilization for a single CPU-intensive host process running with a single guest process. "Equal" means the default scheduler policy, "nice" implies that guest process is niced with parameter -19, and "linger" refers to the use of the Linger-Longer guest priority. "-h" and "-g" identify the host and guest processes.

scheduler modifications. Similar modifications to the other operating systems discussed earlier in Table 1 would presumably show analogous curves, with the difference being that "nice-h" utilization would flatten out at 84 percent for Solaris and at only 60 percent for AIX.

4 Memory Policing: Prioritized Page Replacement

Another way in which guest processes could adversely affect host processes is by tying up physical memory. Having pages resident in memory can be as important to a process's performance as getting time quanta on processors. Our approach to prioritizing access to physical memory tries to ensure that the presence of a guest process on a node will not increase the page fault rate of the host processes.

We first investigated the impact of simply using the UNIX nice command for memory intensive applications. Table 2 shows a simple example of memory thrashing caused by allowing a guest process to compete with host processes for physical memory. In all cases, both processes have working sets of approximately 128 MB, while the total physical memory of the machine is only 192 MB. Both processes take 82 seconds to run in isolation. When they are run serially (first row), the total running time is just 164 seconds. The second row shows that if the two are started simultaneously, and with equal priorities, the processes thrash and lose efficiency. We stopped the processes after five hours. The third row shows the expected result of a late-arriving guest process being unable to steal pages from the host process, and effectively being serialized after the host process. However, it does slow down the host process by about 8 percent. The last row, however, shows that changing the order in which the processes arrive dramatically changes the result. The host process takes a long time to steal enough pages from the guest process in order to hold its working set. We again stopped the execution after about five hours. The reason for the thrashing is that the guest process had modified its pages before the host process started requesting memory. Each initial page fault by the host process is delayed while a guest page is flushed to disk. Meanwhile, the guest process also has page faults that require host pages to be flushed to disk. Therefore, neither process makes much progress since CPU priority does little to prevent thrashing when two processes desire more memory than the system has.

This last case is quite common. For example, a user returning to his workstation and starting GNU Emacs would often see this behavior if their workstation is running a large guest simulation. Therefore, handling this case efficiently is essential to reduce the impact of guest processes on host processes.

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 and TLB 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.

TABLE 2
Completion Times for Two Competing Large Memory Jobs

Policy and Setup	Host time (secs)	Guest time (secs)
Run serially (host then guest)	82	164
Started at the same time, run w/ equal priority	> 5 hours	> 5 hours
Host starts at 0, guest at 10, guest niced to -19	89	176
Guest starts at 0, host at 10, guest niced to -19	> 5 hours	> 5 hours

```
If guest.memory < LowWater
If exists host page whose age > limit
    Replace host page;
    Return;
Else if guest.memory > highWater
If exists guest page
    Replace guest page;
    Return;
Scan for page whose age > limit and replace page
```

Fig. 4. Modified page replacement policy to support memory prioritization.

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 now consider applying our new policy to the Linux VM system. In Linux, the default replacement policy is an LRU-like policy based on the "clock" algorithm [12]. The Linux algorithm uses a one-bit flag and an age counter for each page. Each access to a page sets its flag. Periodically, the virtual memory system scans the list of pages and records which ones have the use bit set, clears the bit, and increments the age by three for the accessed pages. Pages that are not touched during the period of a single sweep have their age decremented by one. Only pages whose age value is less than a system-wide constant are candidates for replacement.

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 runtime via system calls. The kernel keeps track of resident memory size for guest 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 and the least recently used guest page to be paged out.[3] Between the two thresholds, older pages are paged out first no matter what processes they belong to. The overhead of this modification is negligible since it simply changes the order of page scanning.

3. Similar mechanisms can be applied to most UNIX systems, including Solaris, which uses unified paging system for virtual memory and file buffer cache. A modest modification, which simply tags file cache pages to indicate whether they have been accessed by guest processes or host processes, can suffice.

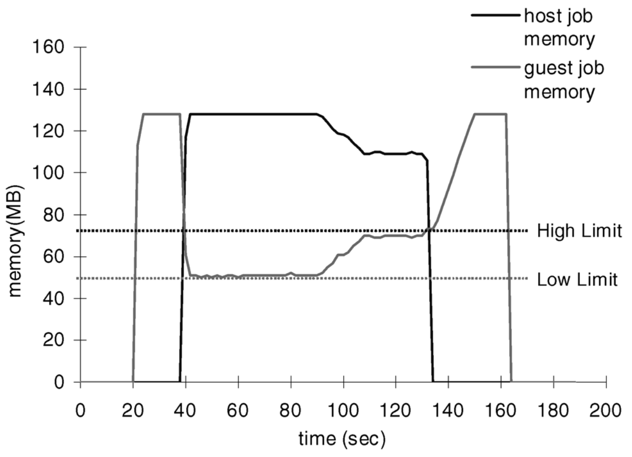


Fig. 5. Threshold validations. Low and high thresholds are set to 50MB and 70 MB. At time 90, the host job becomes I/O-bound. Host process acquires 150 MB when running without contention, and guest process acquires 128 MB without contention. Total available memory is 179 MB.

The modifications to the page replacement algorithm are shown in Fig. 4. Correct selection of the two parameters is critical to meet the goal of exploiting fine-grained idle intervals without significantly impacting the performance of host processes. Too high a value for the low threshold will cause undue delay for host processes, and too low a value will cause the guest process to constantly thrash. However, if minimum intrusiveness by the guest process is paramount, the low memory threshold can be set to zero to guarantee the use of the entire physical memory by the host process.

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 Fig. 5. 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.

We also repeated the experiment shown in Table 2 with our memory priority system enabled. The results are shown in Table 3. When the host process starts first and then the guest process (this is the behavior seen when a user is working, but not using the processor heavily and a guest process then arrives), the use of our modified virtual memory policies reduces the delay seen by the host process from 8.0 percent seen when nice is used to 0.8 percent. For

TABLE 3
Benefits of Memory Priority for Large Footprint Processes

Policy and Setup	Host time (secs)	Guest time (secs)	Host delay
Host starts then guest,			
Guest niced -19	89	176	8.0%
Linger priority	83	165	0.8%
Guest starts then host,			
Guest niced -19	> 5 hours	> 5 hours	> 20,000%
Linger priority	99	255	8.1%

Host delay is computed relatively to the host job time of 82 seconds when running in isolation. In the case, the niced guest job starts first, the experiment has been stopped after five hours due to an excessive delay.

the case when the guest process starts and then the user process, the delay with nice was larger than 200 times the original execution time (recall that we gave up after waiting five hours). In contrast, using linger priority only had a delay of about 8 percent. These two results demonstrate the ability of our kernel modifications to limit the overhead experienced by guest processes.

Host delay is computed relatively to the host job time of 82 seconds when running in isolation. In the case the niced guest job starts first, the experiment has been stopped after five hours due to an excessive delay. In this section, we demonstrated that the Unix CPU priority is not effective to promote memory bound host processes. Thus, a new prioritized page replacement was introduced and validated by a set of experiments. Our experiments reported that even in a rare worst case, our mechanisms can decrease host job slowdown to 8 percent.

5 I/O and Network Policing: Rate Windows

Priority mechanisms for I/O and network are also essential since I/O or communication intensive guest jobs can significantly slow down host jobs. To protect host jobs' I/O performance, a new I/O scheduling mechanism, called rate windows, is proposed as a simple, portable, and effective option. Many real-time systems provide rate-based scheduling which is similar to our rate windows. However, none of those mechanisms is suitable for our use since they all require new I/O job queues inserted into an operating system and, hence, imposes an extra scheduling overhead. The rest of this section describes our rate-window policies, and the mechanisms that are needed to support I/O and network throttling. Then, we validate our mechanisms with a series of experiments.

5.1 Rate Windows Mechanisms

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. In the context of cycle stealing, host processes are unconstrained and guest processes are constrained.

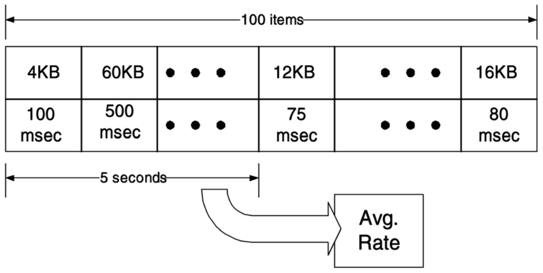


Fig. 6. Maintaining a sliding window of resource utilization.

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 unthrottled mode when the I/O rate is near the threshold.[4] The state of the system is reflected in the global variable throttled. Our measurement of unconstrained bandwidth is not instantaneous; it is measured over the life of the rate window, defined below.

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 Fig. 6).

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 have experimented with several different window sizes before arriving at these constants. The window for I/O averaging is bounded by two parameters: window size and duration. When I/O is rare, the window is bounded by window duration. Contrarily, when I/O is frequent, the window is limited by window size. In the current kernel-level implementation, the window size is limited to 100 elements to avoid excessive use of kernel memory. However, it is possible that new environments or applications could be best served by using other values for these parameters. We provide a means of tuning these and other parameters from a user-level tool.

We implemented our mechanism via a loadable kernel module that intercepts each of the kernel calls for I/O and network communication: `read()`, `write()`, `send()`, and `recv()`.[5] 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.

4. For all the experiments, the low and high thresholds are set to 5 percent (500 KB/s) and 10 percent (1,000 KB/x) of maximum bandwidth, respectively.

5. This light-weight I/O function call interception mechanism has a limitation of missing memory mapped I/Os. Our mechanism can be extended to record memory mapped I/O mappings and account for I/O requests through the mapped memory. However, such extension is undesirable as it will decrease the simplicity and portability of our mechanism.

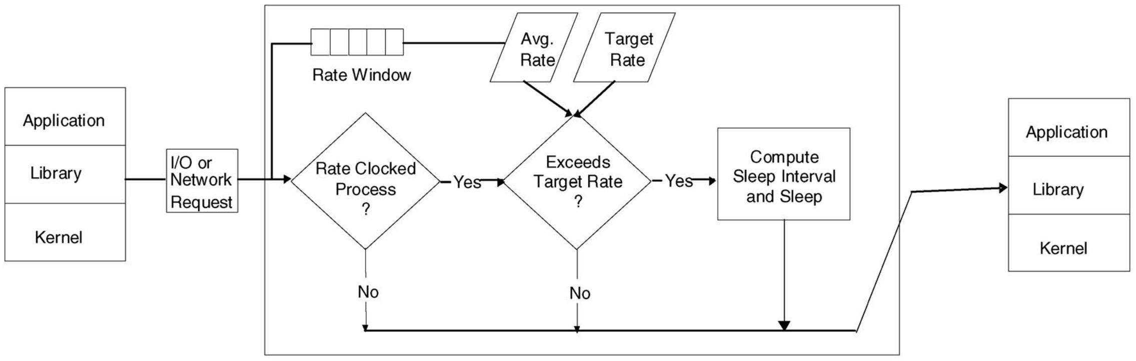


Fig. 7. Policing I/O requests.

The process ID is used to map to an I/O class, and the I/O type is used to distinguish between file I/O 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 information that is too old, we limit the sliding window to a fixed interval of time (currently 5 seconds).

We define B_w, the window bandwidth, 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:

$$\frac{B_w}{T_w} \leq R_t. \tag{1}$$

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

$$\text{delay} = \frac{B_w}{R_t} - T_w. \tag{2}$$

Then, the kernel suspends the process for delay time units before calling the original I/O system call. This process is illustrated graphically in Fig. 7. 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. For network connections, TCP provides a byte-oriented stream rather than a record oriented one, so breaking a single request into a smaller one will not affect the correctness of any protocol.

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 XP (nee Windows NT) and the stackable file system [13] provide the required calls.

5.2 File I/O Policing

In order to validate our approach, we conducted a series of microbenchmarks and application benchmarks. The purpose of these experiments is threefold. First, we want to show that our mechanism does not add any significant delay to 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' overall file I/O limit. We wanted to measure this effect.

We first measured resource usage in order to verify that the use of rate windows does not add significant overhead to the system. On an otherwise idle machine, we ran a single tar program, which created a 52 MB archive file, 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. Fig. 8a represents a run without throttling, and Fig. 8b 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 percent of the maximum I/O or network bandwidth, whereas a "host" process has unconstrained use of all bandwidth. We measured the effective maximum bandwidth by reading or sending a large file sequentially.

In both graphs in Fig. 8, the guest job starts first, followed somewhat later by the host job. At this point, the guest job throttles down to its 10 percent rate (500 KB/s). When the host job finishes, the guest job throttles back up after the rate window empties. Note that the version with I/O throttling is less thrifty with resources (the guest job finishes later). This is a design decision: our goal is to prevent undue degradation of unconstrained host job performance regardless of the effect on guest jobs.

The behavior of one of the tar processes is shown in more detail in Fig. 9. 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 sizes increase near the start as the file system read ahead becomes more aggressive. Besides the read-ahead effect, the buffer cache hit ratio is expected to be very low since tar sequentially reads each file only once.

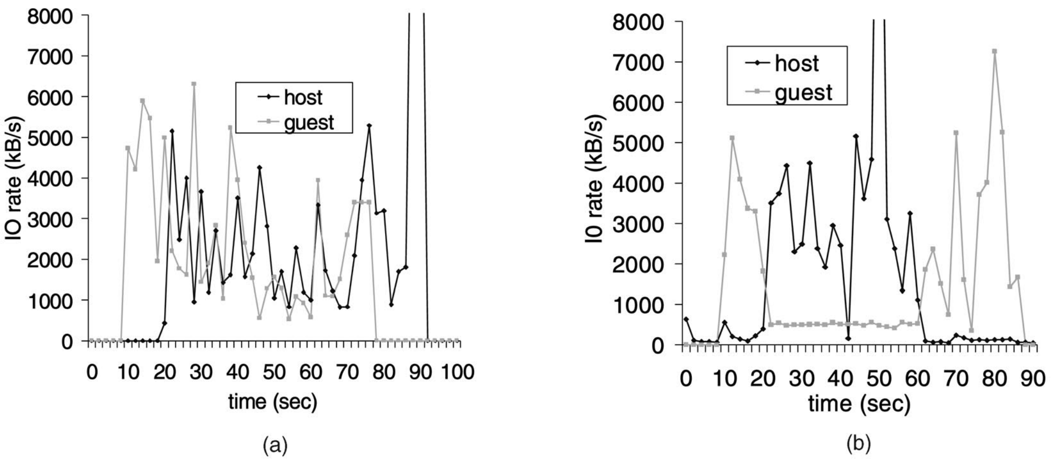


Fig. 8. File I/O of competing tar applications without (a) and with (b) file I/O policing. In (b), the host I/Os occurring before time 20 are system daemon processes' background I/O activities and can be considered as noise.

Our third set of microbenchmark experiments is designed to examine the relationship between file I/O and disk I/O and imposed sleep time for guest applications. The first application was 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.

File I/O can dilate because 1) 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 2) the buffer cache's read-ahead policy may speculatively bring in disk blocks that are never referenced. File I/O rates can also attenuate due to buffer cache hits, which is a consequence of the I/O locality of the applications. Table 4 shows the behavior of `tar`, `agrep`, and `compile` applications with various metrics.

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`,[6] 14 percent less for `agrep`, and 233 percent 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, 500 KB/sec.

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 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 deliberately have not taken this path because we wish to keep our system at as high a level as possible. We currently implement our entire I/O and communication policing 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 its allocated bandwidth. The dilation factor for each process is computed by counting file page cache faults,[7] which lead to disk I/Os, during file I/O requests. 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. Fig. 10 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. Fig. 10a 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 (Fig. 10b) 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 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. Fig. 10c shows the behavior of the compile application when the

6. The `tar` file size is 52 Mbytes.

7. In the Linux kernel, this count is available in the `maj_fault` field.

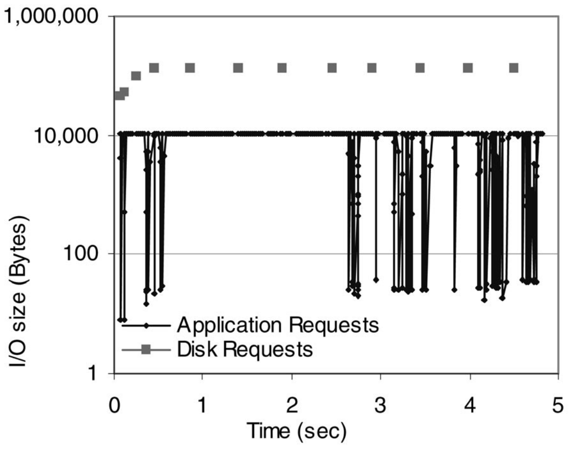


Fig. 9. I/O sizes versus time for `tar`.

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.

Although rate windows provide protection of the host I/O rate by limiting guest I/O rates, guest jobs can still slow down host file I/Os by polluting the host file buffer cache. This can be handled by our memory policing mechanism which can prevent guest jobs from replacing out host jobs' file buffer cache pages.

5.3 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 an application of network I/O throttling using our rate windows.

Most of the experiments in Section 5.2 assumed the use of rate windows in a cycle stealing 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 the rate windows are able to throttle even the checkpoint and prevent it from affecting host jobs.

Fig. 11 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 in this case) in the unthrottled case shown in Fig. 11a. Only after the checkpoint is finished does the Web server claim most of the bandwidth.

In the throttled case shown in Fig. 11b, 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.

In this section, 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 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.

TABLE 4
I/O Application Behavior

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

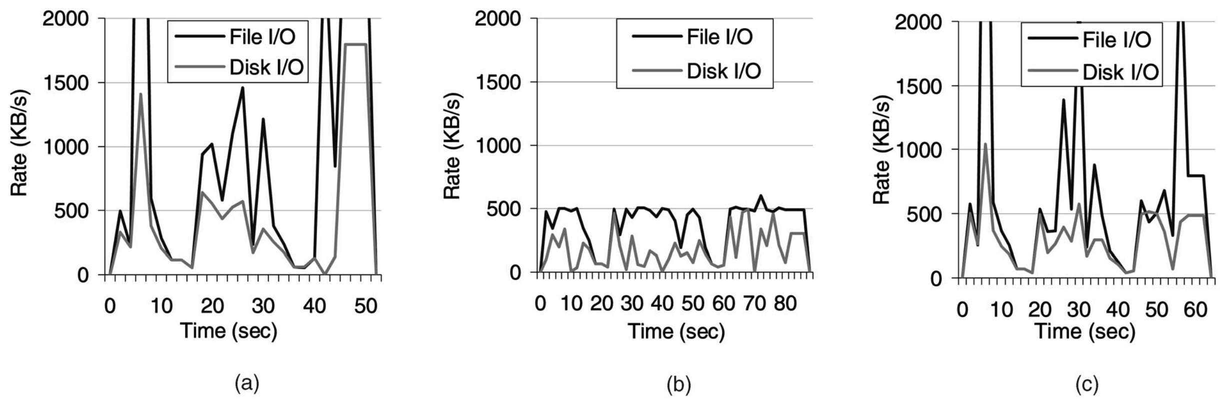


Fig. 10. File and disk I/O rates for the compile workload. (a) is without any rate limit while (b) is with the file I/O limit of 500KB/sec, and (c) is with the disk I/O limit of 500KB/sec.

6 Application Validation

In this section, we validate effectiveness of our resource policing mechanisms as a whole. The first set of experiments implements a scenario where interactive users are traditional terminal-based UNIX program developers and guest jobs are computation-intensive scientific applications. The second experiment scenario is emulating a case where machine owners are working interactively using window-based applications, while guest jobs applications are requiring various resources intensively at different stages. For both scenarios, we demonstrate how efficiently and unobtrusively idle resources can be exploited with our resource policing mechanisms.

6.1 Case 1: Parallel Guest Applications

We first present a study of the effect of our resource policing mechanisms on interactive host jobs and guest parallel applications in our test cluster. This cluster is comprised of eight Pentium workstations running Linux 2.2, connected by a 1.2 Gigabit Myrinet and a 100 Megabit switched Ethernet. For resource policing configuration, the low and high thresholds for memory policing are set to 5 percent and 10 percent of total memory (196 MB), respectively. The maximum I/O and network bandwidth (Rt) for guest jobs are constrained to 10 percent (500 KB/sec) when host jobs are active in using such resources.

We use the Musbus interactive UNIX benchmark suite [14] to simulate the behavior of actual interactive users. Musbus simulates an interactive user conducting a series of compile-edit cycles through text-based terminals. The benchmark creates processes to simulate interactive editing (including appropriate pauses between keystrokes), UNIX command line utilities, and compiler invocations. We varied the size of the program being edited and compiled by the "user" in order to change the mean CPU utilization of the simulated local user. In all cases, the file being manipulated was at least as large as the original file supplied with the benchmark.

The guest applications are `water` and `fft` from the Splash-2 benchmark suite [15], and `sor`, a simple red-black successive overrelaxation application [16]. `Water` is a molecular dynamics code, while `fft` implements a three-dimensional fast Fourier transform. All three applications are run on top of CVM [17], a user-level DSM system. These three applications are intended to be representative of three common classes of distributed applications. `Water` has relatively fine-grained communication and synchronization, and `fft` is quite communication-intensive, while `sor` is mostly compute-bound. General computation and communication ratios of these applications are presented in our previous work [7]. The input for `sor` is $2,048 \times 1,024$ array. For `water`, 512 molecules are simulated while a three-dimensional array of $2^6 \times 2^6 \times 2^6$ is used for `fft`. To support these datasets, 35MB of memory is allocated for `sor`, 3.3 MB for `water`, and 27 MB for `fft`. While memory requirements by these applications are moderate, contentions for CPU cycles are significant; their CPU usages are

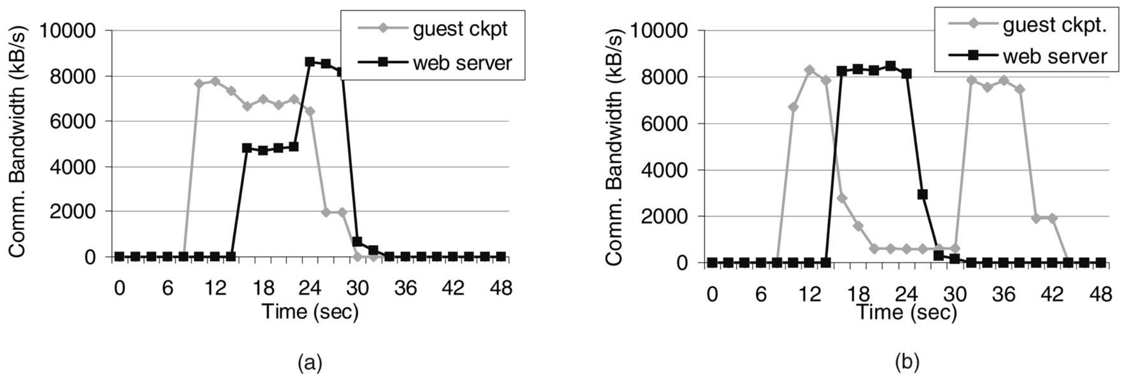


Fig. 11. Guest job checkpoint versus host Web server. The graphs show network bandwidth usages (a) without and (b) with network I/O policing.

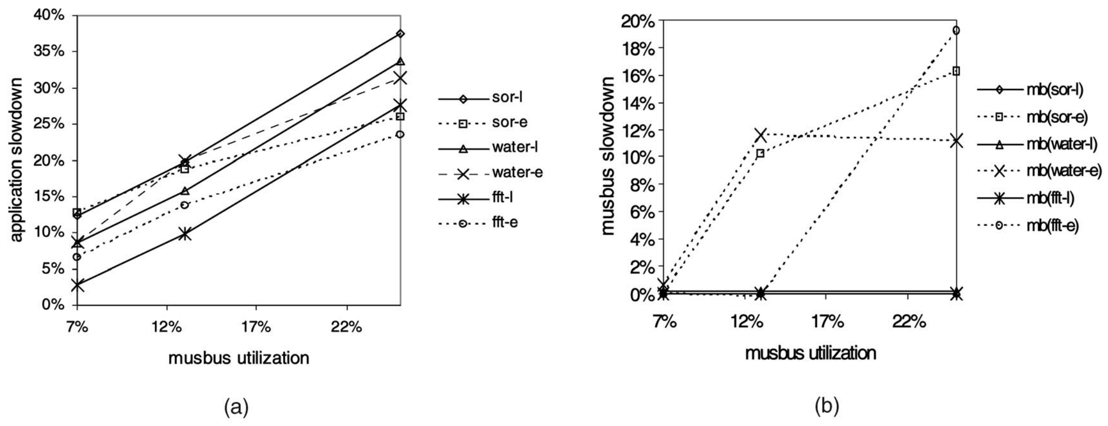


Fig. 12. Impact of running one process of four-process CVM applications as a guest process. Three applications, `sor`, `water`, and `fft`, are run with linger priority (-l) and equal priority (-e). (a) shows the slowdown of the guest applications. (b) shows the slowdown of the host Musbus processes.

about 99 percent when running on one node and no lower than 75 percent even when running on four nodes.

In the first set of experiments, we run one process of a four-process CVM application as a guest process on each of four nodes. We vary the mean CPU utilization of the host processes from 7 percent to 25 percent by changing the size of the program being compiled during the compilation phase of the benchmark. The results of these tests are shown in Fig. 12.

Fig. 12a shows the slowdown experienced by the guest applications. The solid lines show the slowdown using our Linger-Longer policy, and the dashed lines show the slowdown when the guest processes are run with equal priority. As expected, running the guest processes at starvation level priority generally slows them down more than if they are run at equal priority with the host processes. However, when the Musbus utilization is less than 15 percent the slowdown for all applications is lower with lingering than with the default priority. For comparison, running `sor`, `water`, and `fft` on three nodes instead of four slows them down by 26 percent, 25 percent, and 30 percent, respectively. Thus, for the most common levels of CPU utilization, running on one nonidle node and three idle would improve the application's performance compared to running on just three idle nodes. The simulations in our previous work [6] showed that node utilization of less than 10 percent occurs over 75 percent of the time even when users are actively using their workstations.

Fig. 12b shows the slowdown experienced by the host Musbus processes. Again, we show the behavior when the guest processes are run using our Linger-Longer policy and the default equal priority. For all three parallel guest applications, the delay seen when running with Linger-Longer is not measurable. However, when the guest processes are run with moderate CPU utilization (i.e., more than 10 percent), all three guest processes start to introduce a measurable delay in the host processes when equal priority is used. For `water` and `sor`, the delay exceeds 10 percent when the Musbus utilization reaches 13 percent. At the highest level of Musbus CPU utilization, the delay using the default priority exceeds 10 percent for all three applications and 15 percent for two of the three applications.

In this experiment, despite our increased emphasis on preserving host process performance, our resource policing mechanisms allowed parallel guest applications to perform well even when one or more of the workstations are running moderate host processes.

6.2 Case 2: Window-Based Interactive Host Applications

With windowing systems, such as X-Window on UNIX, GUI-based interactive applications exhibit much different resource usage patterns from those with terminal-based programs, as reported by Chen et al. [18]. Therefore, in the second experiment, we focus on effectiveness of our resource policing mechanisms in terms of preserving responsiveness of GUI based interactive host applications on windowing systems, in the presence of resource intensive guest jobs.

We use response time as the performance metric. Response time is defined as the time elapsed between the occurrence of an interactive event and completion of the corresponding activity. For example, when a user wants to search for a string in a text editor application, she enters the required text in the search dialog box and then clicks on a button. The application will then perform the required search and highlight the first occurrence of the search string in the text.

Precisely measuring response time to interactive user events is very difficult. Flautner et al. [19] have observed that, to respond to an event, an application has to perform a certain task, which can be usually seen by a sudden surge in utilization of resources. The types of resources to be consumed during this task depend on the type of the interactive event. In case of the previous example of text-based searching, the text editor will perform a large number of successive string comparisons causing sudden surge in CPU utilization. It will also read all pages of the document sequentially from memory, possibly resulting in disk I/O activity. In response to a user request for a webpage, a web browser will show sudden increase in CPU as well as network utilization.

From this observation, we have developed a utility that detects the occurrences of interactive events and measures their response times. This utility monitors each type of resource separately. If response to a certain event involves both CPU and I/O activity, the utility will identify it as two subevents: CPU event and I/O event.

For each run of experiments, it is essential that interactive events take place in the same sequence in each run of experiments. To this end, we use Android [20], an opens source GUI testing tool. This tool can record and playback the GUI events, such as mouse movements, mouse clicks, and

TABLE 5
Interactive Application Benchmark Overview

Phase 1	A user launches web browser application (Netscape). She maximizes the window and clicks on the “Home” icon to bring up the homepage (Redhat website). She browses various links from the homepage and then eventually returns to the home page.
Phase 2	The user performs a web search for a string and browses few of the links returned by the search query. Then she types a URL in the browser address bar and hits enter key. The website has html pages with images and text pages. She returns to the home page and exits the browser.
Phase 3	The user opens a text editor application. She goes to “File” then “Open” menu and opens a large text file. She views a few pages of the file using the scroll bar and PgUp/PgDn keys. The user then performs full text search for three different inputs. Two of the inputs result in unsuccessful searches (requiring linear access to the entire file) and one results in a successful search.
Phase 4	The user then selects entire contents of the file by highlighting them. She cuts the text and saves it in the buffer. She enters some new text in the file, edits it and then deletes all of it. She pastes the old text from the buffer back to the file. She closes the application.

Interactive workload generates about 100 typical user events using X-Window-based Web browser and text editor.

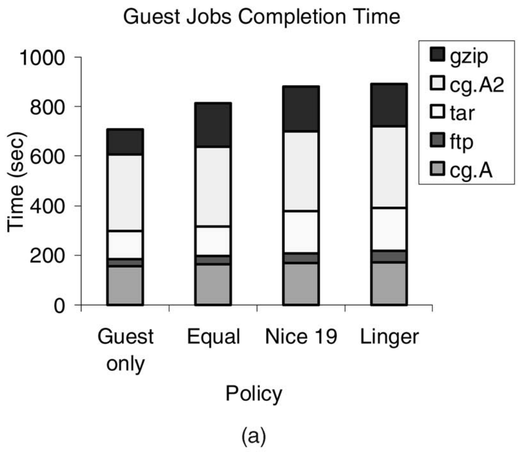
keyboard typing, on X-Window systems. It can also record the delay between two user events to reflect think time.

To the best of our knowledge, there are no benchmarks that measure interactive response time for X-Window based applications. Therefore, we have designed and generated an interactive user event script while a human operator is naturally using some of GUI-based applications. This interactive host workload contains two popular X-Window applications: a Web browser (Netscape) and KDE’s text editor (KWrite). Netscape is primarily network and CPU intensive while KWrite is more I/O and CPU intensive. This benchmark consists of about 100 user activities including browsing sites, clicking links, Web searches, opening and closing files, text searching, cutting and pasting of text. Table 5 shows overview of these user activities divided into four phases. It has been observed that KWrite consumes more than 800MB of memory in Phase 5.

On the other hand, the guest workload is composed of a sequence of tasks that intensively use different resources: `cg.A` from NAS Parallel Benchmark [21] as a CPU intensive application, `tar` and `gzip` as I/O and CPU intensive applications, and `ftp` as a network intensive application. To synchronize completions of host workload and guest workload, `cg.A` is run twice and, in the second time, two instances run at the same time (denoted as `cg.A2`).

Now, we conduct responsiveness experiments by playing back the interactive event script while running batch guest jobs with three different priorities: equal, nice 19, and Linger-Longer. We conduct this experiment on machines with an Intel Pentium 4 processor running at 2.2 GHz, equipped with 1 GB RAM and 7,200 RPM IDE hard disk, and connected to 100Mbps Ethernet. The Linux kernel has been patched with our Linger-Longer resource policing module and running an X server for Window-based applications.

Fig. 13a shows the guest completion time when the guest jobs are run at different priorities. The first bar provides the base case when only guest jobs run. To describe characteristics of each task briefly, `cg.A` requires 60 MB of memory and completes in 154 seconds on an idle machine; `Tar` reads 27 files, of which size is 277 MB in total, and writes an output file taking 115 seconds on an idle machine; `gzip` takes 101 seconds to compresses this large output file; `cg.A2` allocates 120 MB of memory and finishes in 307 seconds on an idle machine. Not surprisingly, the guest completion time increases when there are host applications running. With equal priority, guest workload



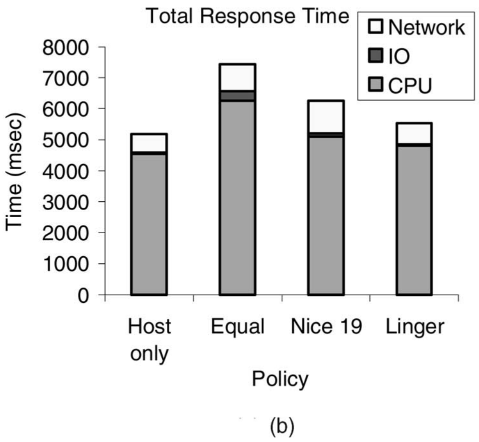


Fig. 13. Guest job performance and host job responsiveness. Guest job consists of several resource intensive applications such as `gzip`, `cg.A`, `tar`, and `ftp`. Host event response time is measured for three resources: CPU, disk I/O, and network. (a) shows the completion time of the batch guest jobs and (b) shows the response time of the Window-based host jobs.

is slowed down by 15 percent. With guest jobs niced, the slowdown further increases to 25 percent, due to its lower priority. However, with Linger-Longer, the delay increase from the nice 19 case is unnoticeable.

Fig. 13b shows the total response time in milliseconds for user events with host applications, when the guest jobs are run at different priorities. In the absence of guest jobs, total response time for host workload is 5,170 milliseconds where 4,543 milliseconds are spent for 22 CPU related events, 39 milliseconds for 12 I/O related events, and 588 milliseconds for 54 network related events. When guest jobs are run with equal priority, the overall host response time increases by 44 percent. With guest jobs niced, the delay that interactive users experience is somewhat decreased, yet a significant delay of 21 percent remains. Notice that this reduction via nice is mainly due to reduced response time to CPU events. The last bar demonstrates that our Linger-Longer resource policing significantly reduces this delay to only 6 percent by recovering the original responsiveness to I/O and network related events of interactive host workload.

In this experiment, we demonstrate that our Linger-Longer resource policing mechanisms effectively protect system responsiveness to interactive host applications running on a windowing system from various resource-intensive guest applications.

7 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 [4], LSF [22], and NOW [23] 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 nonidle workstations was the Stealth distributed scheduler [24]. It implemented a priority-based approach to running guest processes.

In the area of operating system support for providing resource management, research and commercial operating systems have provided similar functionality. In IRIX [25], 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.

Verghese et al. [26] proposed a way to isolate the performance of applications running on an SMP system. While their approach requires changes to similar parts of the operating system, their primary goal was to increase fairness to all applications, while our goal is to create an inherently unfair priority level for guest processes.

Aron and Druschel's soft timers [27] 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 [28] and Resource Containers [29] can group a set of processes or threads as a unit for resource scheduling. This is similar to our job classes. The Nemesis kernel [30] 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 rate windows mechanism resides between the kernel and the user-level I/O library and can be loaded and unloaded at runtime. 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.

The idea of regulating traffic rates in the network has been extensively studied. Congestion avoidance schemes such as leaky bucket [31] and its variants [32], [33] 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 nonconforming traffic. In contrast, since our approach is at the source, we can delay traffic to enforce bandwidth limits.

Resource partitioning using virtual machines has been popular both in the 1970s [34] as well as in recent projects such as Disco [35]. The key difference is that while virtual machines provide hard isolation of resources between VMs at considerable runtime overhead, our approach is a simple extension to an existing operating system or runtime library.

8 Conclusions

In this paper, we have shown that it is possible to achieve fine-grained cycle stealing on workstations without significantly impacting host processes. We presented the design, implementation, and performance of a suite of resource policing mechanisms that provide this vital safety net even in the presence of guest processes that aggressively demand resources.

We first have addressed resource contention for CPU and memory. A new guest class of processes prevents guest processes from stealing any processor time from host processes. This change alone can have an effect of 8 percent CPU consumption on Linux systems and up to 40 percent on other operating systems. We implemented a new page replacement policy that imposes hard upper and lower limits on the number of physical pages that can be obtained by guest processes when host processes are active.

To police network and I/O contention between guest and host processes, we have presented the rate windows 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 guest jobs with little overhead.

Using two sets of experiments with all the resource policing mechanisms put together, we demonstrated that resource intensive guest jobs would not noticeably degrade system responsiveness to both terminal-based and Window-based interactive applications.

Although all these mechanisms were developed for our Linger-Longer system, they are general enough to serve other types of use. CPU and memory priority mechanisms can provide an ultralow job priority augmenting the nice command in UNIX. Our communication and I/O throttling mechanisms can support lightweight rate-based bandwidth scheduling which can bound maximum bandwidth usage.

It is possible to further enhance the virtual memory system to increase the speed at which pages are reclaimed from the guest processes by the host processes. In particular, dirty guest pages require writing back to the swap device before they can be allocated to the host process. One extension that we are planning would trigger the VM system to aggressively write dirty pages to disk for guest processes when this can be done without causing resource contention with the host process. This can be thought of as a background cleaning process, analogous to the cleaner in log-structured file systems.

One area of future work for our rate windows mechanism 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.

Acknowledgments

The authors would like to thank Mr. Nimish Pachapurkar for developing the interactive application benchmark. This work was supported in part by the Consortium for Embedded Internetworking Technologies (CEINT) Research Awards.

References

[1] M.W. Mutka and M. Livny, "The Available Capacity of a Privately Owned Workstation Environment," *Performance Evaluation*, vol. 12, no. 4, pp. 269-284, 1991.

[2] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure.* San Francisco: Morgan-Kaufmann, 1998.

[3] F. Berman, G. Fox, and A. Hey, *Grid Computing: Making the Global Infrastructure a Reality.* Wiley, 2003.

[4] M. Litzkow, M. Livny, and M. Mutka, "Condor—A Hunter of Idle Workstations," *Proc. Int'l Conf. Distributed Computing Systems*, pp. 104-111, 1988.

[5] K.D. Ryu, J.K. Hollingsworth, and P. Keleher, "Efficient Network and I/O Throttling for Fine-Grain Cycle Stealing," *Proc. Supercomputing Conf.*, 2001.

[6] K.D. Ryu and J.K. Hollingsworth, "Linger Longer: Fine-Grain Cycle Stealing for Networks of Workstations," *Proc. Supercomputing Conf.*, 1998.

[7] K.D. Ryu and J. Hollingsworth, "Exploiting Fine Grained Idle Periods in Networks of Workstations," *IEEE Trans. Parallel and Distributed Systems*, vol. 11, no. 7, pp. 683-698, July 2000.

[8] A. Barak, O. Laden, and Y. Yarom, "The NOW Mosix and its Preemptive Process Migration Scheme," *Bull. IEEE Technical Committee on Operating Systems and Application Environments*, vol. 7, no. 2, pp. 5-11, 1995.

[9] J.C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 75-84, 1991.

[10] B.L. Jacob and T.N. Mudge, "A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 295-306, 1998.

[11] A. Tamches and B.P. Miller, "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels," *Proc. Third Symp. Operating Systems Design and Implementation (OSDI)*, pp. 117-130, 1999.

[12] W.R. Carr and J.L. Hennessy, "WSClock—A Simple and Effective Algorithm for Virtual Memory Management," *Proc. ACM Symp. Operating System Principles*, pp. 87-95, 1981.

[13] J.S. Heidemann and G.J. Popek, "File-System Development with Stackable Layers," *ACM Trans. Computer Systems*, vol. 12, no. 1, pp. 58-89, 1994.

[14] K.J. McDonell, "Taking Performance Evaluation Out of the 'Stone Age'," *Proc. Summer USENIX Conf.*, pp. 8-12, 1987.

[15] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, pp. 24-37, 1995.

[16] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, vol. 29, no. 2, pp. 18-28, Feb. 1996

[17] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," *Proc. Int'l Conf. Distributed Computing Systems*, pp. 91-98, 1996.

[18] J.B. Chen, "Memory Behavior for an X11 Window System," *Proc. USENIX Winter Conf.*, pp. 189-200, 1994.

[19] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge, "Thread-Level Parallelism and Interactive Performance of Desktop Applications," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 129-138, 2000.

[20] L. Smith and C. Laird, "Android, Open Source Scripting for Testing & Automation," *Dr. Dobbs J.*, vol. 326, pp. 58-61, 2001.

[21] D.H. Bailey, E. Barszcz, J.T. Barton, and D.S. Browning, "The NAS Parallel Benchmarks," *Int'l J. Supercomputer Applications*, vol. 5, no. 3, pp. 63-73, 1991.

[22] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," *SPE*, vol. 23, no. 12, pp. 1305-1336, 1993.

[23] 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," *Proc. SIGMETRICS*, pp. 267-278, 1995.

[24] P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS)*, pp. 336-343, 1991,

[25] SiliconGraphics, IRIX 6.4 Technical Brief, http://www.sgi.com/software/irix6.5/techbrief.pdf, 1998.

[26] B. Verghese, A. Gupta, and M. Rosenblum, "Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 181-192, 1998.

[27] M. Aron and P. Durschel, "Soft Timers: Efficient Microsecond Software Timer Support for Network Processing," *Proc. ACM Symp. Operating Systems Principles (SOSP)*, pp. 232-246, 1999.

[28] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz, "The Eclipse Operating System: Providing Quality of Service via Reservation Domains," *Proc. USENIX Ann. Technical Conf.*, pp. 235-246, 1998.

[29] G. Banga, P. Druschel, and J. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," *Proc. USENIX Third Symp. Operating System Design and Implementation*, pp. 45-58, 1999.

[30] D. Reed and R. Fairbairns, "The Nemesis Kernel," *United Feature Syndicate, Inc.*, 1997.

[31] J.S. Turner, "New Directions in Communications (or which Way to the Information Age?)," *IEEE Comm. Magazine*, vol. 24, no. 10, pp. 8-15, 1986.

[32] J. Csirik, J.B.G. Frenk, M. Labbe, and S. Zhang, "On the Multidimensional Vector Bin Packing," *Acta Cybernetica*, vol. 9, no. 4, pp. 361-369, 1990.

[33] T. Faber, L.H. Landweber, and A. Mukherjee, "Dynamic Time Windows: Packet Admission Control with Feedback," *Proc. SIGCOMM*, pp. 124-135, 1992.

[34] R.P. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer Magazine*, vol. 7, no. 6, pp. 34-45, 1974.

[35] E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *Proc. ACM Symp. Operating Systems Principles (SOSP)*, pp. 143-156, 1997.

Kyung Dong Ryu received the BS and MS degree in computer engineering from Seoul National University in Korea in 1993 and 1995, respectively. He received the PhD degree in computer science from University of Maryland at College Park in 2001. He is an assistant professor in the Department of Computer Science and Engineering at the Arizona State University. His current research interests include grid-computing, peer-to-peer computing, and embedded system performance tuning. Dr. Ryu's current projects include scalable peer-to-peer computing infrastructure and μ-Watch: a performance monitoring and tuning tool for networked embedded systems. Dr. Ryu is a member of the IEEE Computer Society and ACM.

Jeffrey K. Hollingsworth received the PhD and MS degrees in computer science from the University of Wisconsin in 1994 and 1990, respectively. He received the BS degree in electrical engineering from the University of California at Berkeley in 1988. He is an associate professor in the Computer Science Department at the University of Maryland, College Park, and affiliated with the Department of Electrical Engineering and the University of Maryland Institute for Advanced Computer Studies. His research interests include instrumentation and measurement tools, resource aware computing, high performance distributed computing, and computer networks. Dr. Hollingsworth's current projects include the dyninst runtime binary editing tool and harmony—a system for building adaptable, resource-aware programs. Dr. Hollingsworth is a senior member of the IEEE and a member of the ACM.