To appear: 1999 ACM International Conference on Supercomputing (ICS'99), Rhodes, Greece, June, 1999.

Mechanisms and Policies for Supporting Fine-Grained Cycle Stealing

Kyung Dong Ryu Jeffrey K. Hollingsworth Peter J. Keleher

Computer Science Department
University of Maryland
College Park, MD 20742

{kdryu,hollings,keleher}@cs.umd.edu

ABSTRACT

This paper presents an investigation into local mechanisms and scheduling policies that allow guest processes to efficiently exploit otherwise-idle workstation resources. Unlike traditional policies that harvest cycles only from unused machines, we target policies that exploit resources even from machines that have active users. We present a set of kernel extensions that allow these policies to operate without significantly impacting host processes: 1) a new *guest* process priority that prevents processes from stealing any processor time from host processes, 2) a new page replacement policy that imposes hard bounds on the number of physical pages that can be obtained by guest processes when host processes are active, and 3) a new page-out strategy that adaptively increases the pageout rate of guest processes when new host processes are started.

We evaluate both the individual impacts of each mechanism, and their utility in supporting *Linger-Longer*, an aggressive cycle-harvesting policy.

Keywords

High-performance computing, cluster computing, networks of workstations, parallel computing

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 [13], since the majority of workstation cycles go unused. In combination with ever-increasing needs for cycles, this presents an obvious opportunity to better exploit existing resources. 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.

Copyright© 1999 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies this notice and the full citation on the first page or initial screen of the document. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

The thesis of this paper is that such policies waste many opportunities to exploit cycles because of overly conservative estimates of resource contention. We show that the potential negative impact of guest processes can be severely limited 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 both processing cycles and memory. This paper describes these mechanisms and presents both a micro-benchmark 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* [16] 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 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 improve the throughput of sequential compute-bound processes. In trace data collected from a variety of organizations, we showed that over 75% of the time nodes have a CPU utilization less than 10%. We also showed via simulation that we could improve the throughput of a compute bound batch workload by 60% compared with the scheduling policies used by the Wisconsin Condor system and the Berkeley NOW project.

This paper presents the design, implementation and performance evaluation of policies that allow the use of Linger-Longer on collections of Linux workstations. Section 2 reviews the Linger-Longer policy, summarizes our previous simulation results, and describes the additions to local schedulers required to support Linger-Longer. Section 3 describes our implementation of mechanisms that meet these constraints in the Linux 2.0.32 operating system. Section 4 reports on the results of our mico-benchmarks of the kernel extensions and a case-study of distributed shared memory (DSM)-based parallel applications on a Linger-Longer cluster. Section 5 reviews related work in the field, Section 6 presents future work, and finally Section 7 summaries our conclusions.

2. FINE-GRAIN CYCLE-STEALING

This section introduces the concept of fined-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. To achieve 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 virtual memory via a priority scheme. The basic idea is to tag all pages as either guest or host pages, and to give priority on page replacement to the host pages. The complete mechanism is presented in Section 3.2.

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 caches misses) to reload the process's cache state, and 3) the time to reload the working set of virtual pages into physical page frames. We defer discussion of the latter overhead until Section 3.2. On current microprocessors, the time to restore cache state dominates the register restore time. Our simulations showed that if the *effective context-switch time* is 100 microseconds or less, the overhead of this extra context-switch is less than 2%. With host CPU loads of less than 25%, host process slowdown remains under 5% even for effective context switch times of up to 500 micro-seconds.

In addition, our simulations of sequential processes showed that a linger-based policy would improve average process completion time by 47% compared with previous approaches. Based on job throughput, the Linger-Longer policy provides a 50% improvement over previous policies. Likewise our Linger-Forever policy (i.e. disabling optional migrations) permits a 60% 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%.

Previous systems automatically migrate guest processes off of non-idle 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 co-exist with host processes without significantly impacting the performance of the latter, or starving the former.

3. LINUX KERNEL EXTENSIONS

This section introduces the modifications to the local Linux scheduler necessary to support the Linger-Longer scheduling policy. These extensions are designed to ensure that guest processes can not impede the performance of host processes. We first describe the general nature of our kernel modifications, and then describe how we modified the scheduler and virtual memory system of Linux to meet our needs.

One possible concern with our approach is the need for kernel modifications. In general, it is much harder to gain acceptance for software that requires kernel modifications. However, for the type of system we are building, such modifications are both necessary and reasonable. First, guest processes must be able to stay running, yet impose only an unnoticeable impact on foreground local processes. There is no practical way to achieve this without kernel modifications. Additionally, we feel that 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 [18]. KernInst allows fairly complex customizations of a UNIX kernel at runtime via dynamic binary re-writing. All of the changes we have made could be implemented using this technique.

Current UNIX systems support CPU priority via a per-process parameter called the *nice* value. Via nice, different priorities can be assigned to different processes. These priority levels are intended to reflect the relative importance of different tasks, but they do not necessarily implement a strict priority scheme that always schedules the highest priority process. The nice value of a process is just a single component that is used to compute the dynamic priority during execution. As a result, sometimes a lower *static priority* process gets scheduled over higher static priority processes to prevent starvation, and to ensure progress of the lower priority processes. However, we need a stricter concept of priority in CPU scheduling between our two classes of processes. Guest processes should not be scheduled (and can even starve) when any host process is ready no matter what its run time priority is. Meanwhile, the scheduling between the processes in the same class should be maintained as it is under current scheduling implementation.

While many UNIX kernels provide strict priorities in order to support real-time deadlines, these real-time priorities are *higher* than traditional UNIX processes. For Linger-Longer, we require just the opposite, a lower priority than normal.

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.

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.

3.1 Starvation Level CPU Scheduling

The Linux 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[1]. 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

[1] 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
            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;
}
```

(a) original scheduler

```
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
        If exists q such that q.state = RUNNABLE and q.priority = GUEST
            Run q;
}
```

(b) modified scheduler

Figure 1: Modified Linux Scheduler.

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 Figure 1a.

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% 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.

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 Figure 1b.

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 is not possible in our application domain because guest and host processes do not share locks, or any other non-revocable resources.

3.2 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.

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.

In more detail, the default Linux replacement policy is an LRU-like policy based on the "clock" algorithm used in BSD UNIX. 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

```
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
```

Figure 2: Modified Page Replacement Policy.

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 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 they belong to. The modifications to the page replacement algorithm are shown in Figure 2.

Correct selection of the two parameters is critical to meeting the goal of exploiting fine-grained idle intervals without significantly impacting the performance of host processes. Too high of value for the low threshold will cause undue delay for host processes, and too low of 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 foreground process.

4. EVALUATION

We conducted a series of micro and macro benchmark studies in order to evaluate the performance of kernel modifications. Unless otherwise specified, all experiments were run on a cluster of eight 266-MHz Pentium II workstations running Linux 2.0.32, connected by a 1.2 Gbit/sec Myrinet. We start with a series of tests that demonstrate the need for our mechanisms by comparing the performance of synthetic programs both with and without our policies enabled. We conclude with a couple of brief case studies that demonstrate that we can run parallel DSM applications as guest processes while interactive applications are running on workstations.

4.1 Motivation

We first investigated the impact of simply using the UNIX nice command to provide local processes with higher priority than guest processes. 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, 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. With the exception of OSF-1, the guest process received a significant amount of processing time (ranging from 8% to 40%). This simple experiment demonstrates the need for our more sophisticated priority mechanism.

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

Table 1: CPU utilization with single host and guest (*niced* at level 19) processes.

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 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%. 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 is that the guest process had modified its pages before the host process starting 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 trashing when two processes desire more memory that 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 her workstation is running a large guest simulation. Therefore, handling this case efficiently is essential to reduce the impact of guest processes on host processes.

4.2 Micro-benchmarks

Before moving to application studies, we validated our kernel extensions by testing each modification in isolation. We first validated our scheduling modifications by comparing 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

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

Table 2: Completion times for two 128 MB processes running on a machine with 192 MB of physical memory.

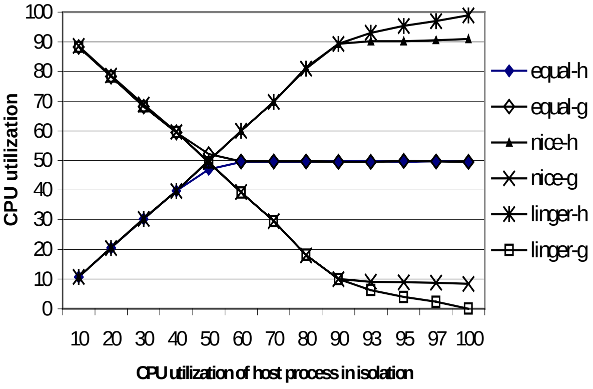


Figure 3: CPU utilization – We show utilization for a single CPU-intensive host process running with a single guest process, for each of three different scheduling policies. 'equal' means no policy at all, 'nice' implies that that guest process is niced with parameter -19, and 'linger' refers to use of the Linger-Longer guest priority. '-h' and '-g' identify the host and guest processes.

contention for the CPU) test of scheduling policies.

Figure 3 shows the resulting behavior. Ideally, 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% utilization. The drop-off at this point corresponds to the 91% limit shown for Linux in Figure 3. Note that 'linger-h', included for comparison, accurately tracks expected utilization up to 99%. The data shows that a guest process is unable to significantly interfere with CPU utilization of a host process with the Linger-Longer modifications. Similar modifications to the other systems discussed in Table 1 would presumably show analogous curves, with the difference being that 'nice-h' utilization would flatten out at 84% for Solaris and at only 60% for AIX.

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 4. 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.

Policy and Setup	Guest time (secs)	Host time (secs)	Host Delay
Host starts then guest,			
Guest niced -19	176	89	8.0%
Linger priority	165	83	0.8%
Guest starts then host			
Guest niced -19	> 5 hours	> 5 hours	> 200
Linger priority	255	99	8.1%

Table 3: Benefits of memory priority for large footprint processes.

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% seen when nice is used to 0.8%. For the case when the guest process start and then the user process, the delay with nice was larger than a factor of 200 (we gave up after waiting five hours). In contrast, using linger priority only had a delay of about 8%. These two results demonstrate the ability of our kernel modifications to limit the overhead experienced by guest processes.

One final aspect of application behavior that needs to be addressed is the slow reclamation of dirty pages from a guest process by a later-starting host process. This is the problem illustrated by the last row in Table 3. The host process is delayed by the one-at-a-time flushing of dirty guest pages to disk. While this is a special case, we argued in Section 4.1 that it can be quite com-

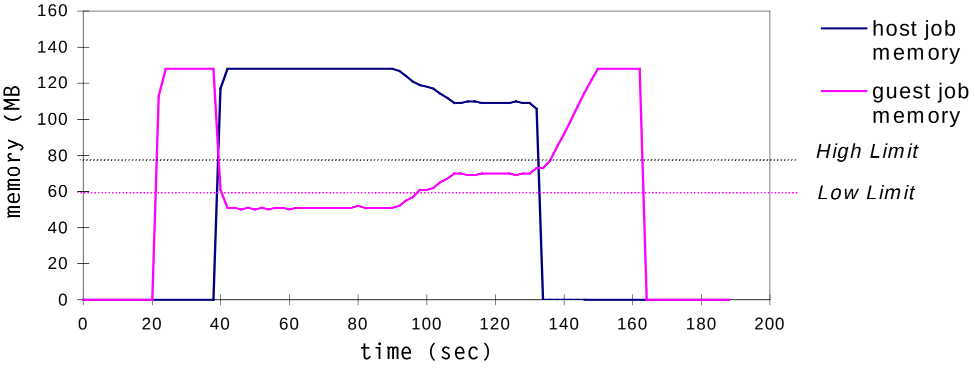


Figure 4: Threshold validations – Low and high thresholds are set to 50MB and 70 MB. At time 90, the host job becomes I/O-bound. Host process acquires 150 MB when running without contention, guest process acquires 128 MB without contention. Total available memory is 179 MB.

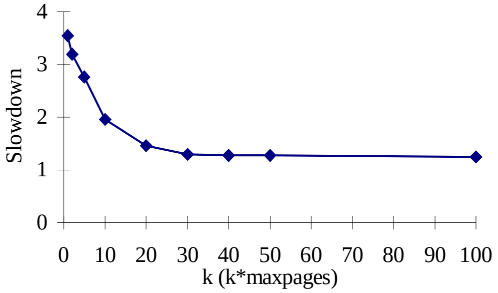


Figure 5: Impact of varying the number of pages paged out at a time.

mon, and is probably the most visible problem of cycle-stealing distributed schedulers. We addressed this problem by adding a trigger mechanism to the pageout process that notices when a new host process starts causing pageouts of guest pages. The trigger mechanism artificially increases the rate at which pages are flushed to disk, analogously to block prefetches. We implement this policy by temporarily increasing the number of pages that the VM system tries to page out at a time (called maxpages) on a fault that is triggered by a host process reclaiming pages from guests. Figure 5 shows the impact of using different multiplicative factors for the desired free list length. When the initial value is used, the host process is delayed by a factor of 3.5 compared to time it would take without the guest process being present. However, once we increase this value to about 50 times its original value, the system pages out most of the pages used by the guest process quickly, and the delay is only about 20%. While this delay seems large, the test program ran only two iterations, and so most of the time was spent getting its working set into memory. A real application would run longer.

4.3 Application experiments

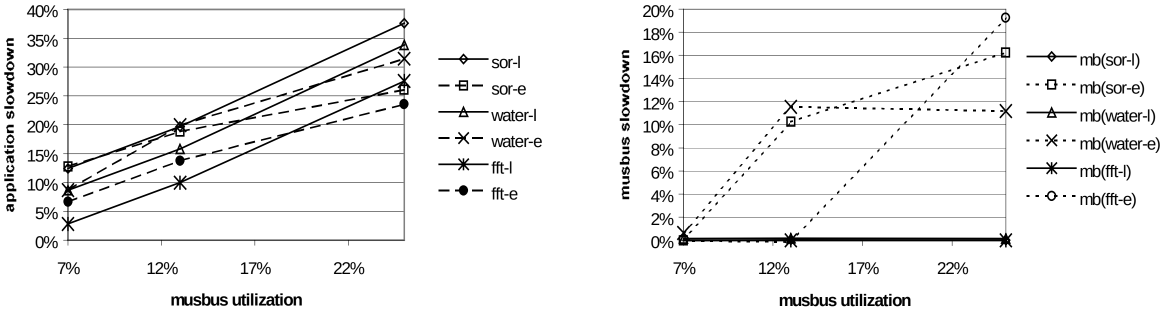
This section presents a study of Linger-Longer's effect on parallel applications on our test cluster. We use the Musbus interactive UNIX benchmark suite [12] to simulate the behavior of actual interactive users. Musbus simulates an interactive user conducting a series of compile-edit cycles. The benchmark creates processes to simulate both 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 [22], and SOR, a simple red-black successive over-relaxation application [2]. Water is a molecular dynamics code, while FFT implements a three-dimensional Fast Fourier transform. All three applications are run on top of CVM [9], 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, FFT is quite communication-intensive, while SOR is mostly compute-bound.

In the first set of experiments, we ran one process of a four-process CVM application as a guest process on each of four nodes. We varied the mean CPU utilization of the host processes from 7% to 22% by changing the size of the program being compiled during the compilation phase of the benchmark. The results of these tests are shown in Figure 6. The left graph shows the slowdown experienced by the parallel 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 the default (i.e., equal priority). As expected, running the guest processes at starvation level priority generally slows them down more than if they were run at equal priority with the host processes. However, when the Musbus utilization is less than 15% 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%, 25%, and 30%, respectively. Thus for the most common levels of CPU utilization, running on one non-idle node and three idle would improve the application's performance compared to running on just three idle nodes. Our previous study[16] showed that node utilization of less than 10% occurs over 75% of the time even when users are actively using their workstations.

The right side of Figure 6 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 was not measurable. However, when the guest processes were run with moderate CPU utilization (i.e., over 10%), all three guest processes started to introduce a measurable delay in the host processes when equal priority was used. For Water and SOR, the delay exceeds 10% when the Musbus utilization reaches 13%. At the highest level of Musbus CPU utilization, the delay using the default priority exceeds 10% for all three applications and 15% for two of the three applications.

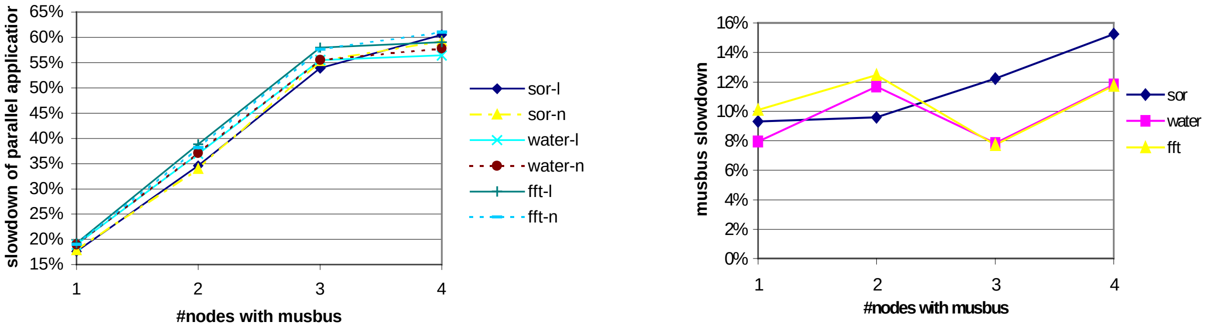
Figure 7a shows the impact on the CVM applications when competing with a host Musbus application running on more than one



(a) slowdown of parallel application (guest process)

(b) slowdown of host Musbus process

Figure 6: Impact of running one process of four-process CVM applications as a guest process.



(a) slowdown of parallel application (guest process) **(b)** slowdown of host processes for *equal* priorities

Figure 7: Impact of differing number of CVM nodes having host process (13% musbus).

of the four nodes. The applications slow remarkably uniformly until four nodes, at which point the slowdown levels off. The Musbus application was set to generate a 13% load in all cases. For comparison, we show the slowdown with both lingering and using nice. In both cases, the slowdown for each application was almost identical. Even though the slowdown compared to all idle nodes reaches 60% when 3-4 non-idle nodes are used, this is still an improvement over what other policies would have allowed (running on one or zero nodes respectively).

Figure 7b shows the slowdown of the Musbus processes with guests at equal priorities as the number of nodes with Musbus processes increases. We do not show the linger and nice cases because there is no slowdown. The fact that there is any change in the slowdown of purely sequential applications as the number of non-idle nodes changes is rather unintuitive. This is explained by the fact that the behavior of the parallel application (which does interact with all nodes) changes as the number of non-idle nodes changes, and the guest parallel processes compete for the processors with the sequential processes. In particular, increasing the number of host processes decreases the efficiency of the parallel DSM applications. They use more cycles, and therefore impose more of a load on their host processors.

5. 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 [11], LSF [24], and NOW [3] 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 IRIX operating system[17], 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 opposite of our approach, which promotes interactive jobs.

Prior studies that investigated running parallel programs on shared workstation clusters also employed fairly conservative eviction policies. Dusseau, et al. [7] used a policy based on immediate eviction. They were able to use a cluster of 60 machines to achieve the performance of a dedicated parallel computer with 32 processors. Acha et al. [1] used a different approach that reconfigured the parallel job to use fewer nodes when one became unavailable. This approach permitted running more processes on a given cluster, although the performance of any single job would be somewhat reduced. PVM [8] is the most widely used package to run programs on clusters, but does not include a scheduling policy, although Pruyne and Livny [15] have investigated adding one.

Process migration and load balancing have been studied extensively. MOSIX [4] provides load-balancing and preemptive migration for traditional UNIX processes. Chowdhury et al. [5] characterized when to reconfigure sequential workloads. DEMOS/MP [14], Accent [23], Locus [20], and V [19] all provided manual or semi-automated migration of processes.

Verghese et. al [21] 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.

6. FUTURE WORK

Access to two additional resource classes needs to be prioritized: I/O and network. Neither is likely to be as critical as managing the processors and virtual memory. However, we encountered one case that demonstrates the need for this feature. When we attempted to run the CVM application using a single shared 100 Mbps (non-switched Ethernet) and the Musbus benchmark with all of its files located on an NFS mounted filesystem, we noticed an 18% slowdown in the Musbus benchmark despite the fact that we were using the linger priority. This was primary due to contention for the shared Ethernet segment (we confirmed this by running the same test but replacing the Ethernet with Myrinet for the CVM application). As a result, we plan to implement this feature in the near future. In particular, we are likely to employ the network priority system proposed by Druschel and Peterson [6] and the I/O priority policy proposed in [21].

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. Due to the potential increase in I/O requirements for this modification, we have deferred its implementation until the I/O priority has been implemented.

7. CONCLUSIONS

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 set of kernel extensions that provide this vital safety net even in the presence of guest processes that aggressively demand resources. Despite our increased emphasis on host process performance, however, our modified kernel allows parallel guest applications to perform well even when one or more of the workstations are running host processes.

We have addressed three specific points of resource contention. 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% on our Linux systems, and up to 40% on other operating systems. We implemented a new 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. Finally, we implemented a new pageout strategy that adaptively increases the pageout rate of guest processes when new host processes are started, and showed that this policy can reduce the delay of host processes by a factor of 2.8 for large memory jobs.

8. ACKNOWLEDGEMENTS

This work was supported in part by NSF awards ASC-9703212, CCR-9624803 and ACI-9711364, and DOE Grant DE-FG02-93ER25176.

9. REFERENCES

[1] A. Acharya, G. Edjlali, and J. Saltz, "The Utility of Exploiting Idle Workstations for Parallel Computation," *SIGMETRICS'97*. May 1997, Seattle, WA, pp. 225-236.

[2] 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*, February 1996.

[3] 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.

[4] A. Barak, O. Laden, and Y. Yarom, "The NOW Mosix and its Preemptive Process Migration Scheme," *Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments*, **7**(2), 1995, pp. 5-11.

[5] A. Chowdhury, L. D. Nicklas, S. K. Setia, and E. L. White, "Workload Characteristics for Process Migration and Load Balancing," *ICDCS*. June 1997, Baltimore, MD, pp. 1-7.

[6] P. Druschel and L. L. Peterson, *Operating Systems and Network Interfaces*, in *The Grid: Blueprint for a New Computing Infrastructure*, I. Foster and C. Kesselman, Editors. 1998, Morgan-Kaufmann: San Francisco. p. 505-532.

[7] A. C. Dusseau, R. H. Arpaci, and D. E. Culler, "Effective distributed scheduling of parallel workloads," *SIGMETIRCS*. May 1996, Philadelphia, PA, pp. 25-36.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine*. 1994, Cambridge, Mass: The MIT Press.

[9] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," *ICDCS*. May 1996, Hong Kong, pp. 91-98.

[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] K. J. McDonell, "Taking Performance Evaluation Out of the 'Stone Age'," *Summer USENIX Conference*. June 1987, Phoenix, AZ, pp. 8-12.

[13] M. W. Mutka and M. Livny, "The available capacity of a privately owned workstation environment," *Performance Evaluation*, **12**, 1991, pp. 269-284.

[14] M. L. Powell and B. P. Miller, "Process migration in DEMOS/MP," *SOSP*. 1983, pp. 110-119.

[15] J. Pruyne and M. Livny, *Providing Resource Management Services to Parallel Applications*, in *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientic Computing*, J. Dongarra and B. Tourancheau, Editors. 1994, SIAM Proceedings Series. p. 152-161.

[16] K. D. Ryu and J. K. Hollingsworth, "Linger Longer: Fine-Grain Cycle Stealing for Networks of Workstations," *SC'98*. Nov. 1998, Orlando.

[17] SiliconGraphics, *IRIX 6.5 Technical Brief*, http://www.sgi.com/software/irix6.5/techbrief.pdf, 1998.

[18] A. Tamches and B. P. Miller, "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels," *Third Symposium on Operating Systems Design and Implementation (OSDI)*. (to appear) February 1999, New Orleans.

[19] M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Premptable Remote Execution Facilities for the V-System," *SOSP*. Dec. 1985, pp. 2-12.

[20] G. Thiel, "Locus Operating System, A Transparent System," *Computer Communications*, **14**(6), 1991, pp. 336-346.

[21] B. Verghese, A. Gupta, and M. Rosenblum, "Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors," *ASPLOS*. Oct. 1998, San Jose, CA, pp. 181-192.

[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 1995, pp. 24-37.

[23] E. R. Zayas, "Attacking the Process Migration Bottleneck," *SOSP*. 1987, pp. 13-24.

[24] 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.