Unobtrusiveness and Efficiency in Idle Cycle Stealing for PC Grids

Kyung Dong Ryu
Department of Computer Science & Engineering
Arizona State University
Tempe, AZ 85287
kdryu@asu.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland
College Park, MD 20742
hollings@cs.umd.edu

Abstract

Studies have shown that for a significant fraction of the time desktop PCs and workstations are under-utilized. To exploit these idle resources, various Desktop/Workstation Grid systems have been developed. The ultimate goal of such systems is to maximize efficiency of resource usage while maintaining low obtrusiveness to machine owners. To this end, we created a new fine-grain cycle stealing approach and conducted a performance comparison study against the traditional coarse-grain cycle stealing. We developed a prototype of fine-grain cycle stealing, the Linger-Longer system, on a Linux cluster. The experiments on a cluster of desktop Linux PCs with benchmark applications show that, overall, fine-grain cycle stealing can improve efficiency of idle cycle usage by increasing the guest job throughput by 50% to 70%, while limiting obtrusiveness with no more than 3% of host job slowdown.

Index Terms – Desktop grid, meta-computing, cluster computing, process migration, networks of workstations, idle cycle stealing.

1. Introduction

Today, collections of workstations and PC's connected through high-speed networks are available in research centers, universities and even many office environments. Studies have shown that up to three-quarters of the time workstations are idle [1]. Generally, most machines go unused during the night, lunch time, and the entire day during weekends and holidays.

Conventional cycle-stealing systems such as Condor [2], LSF [3], NOW [4] and others [5-8] have been created to use these available resources for running engineering design applications, computer hardware and software simulations, materials science simulations, optimization problems and computational biology programs. Such systems focus on coarse-grained idle periods (at the scale of minutes or hours) when users are away from their workstations. To gain access to these resources machine owners are offered a "social contract" that guest jobs are allowed to run only on idle machines. To enforce this contract, guest jobs are stopped and migrated as soon as the owner resumes use of their computer.

However, there are more available cycles that such systems don't harvest. This is due to the fact that even when the user is actively working on their machine, the resource usage is very low. This can be explained by the fact that the main use of computers at work is editing documents, web surfing, and reading email. When writing and editing a document, most time is spent thinking and typing, and it does not require significant computing resources.

The thesis of this paper is to achieve more efficient use of computing resources and hence produce higher system throughput while making such resource recycling unobtrusive to machine owners. The traditional approaches don't take advantage of extra free resources because it would slow down host processes and therefore break the "contract" with machine owners. Unobtrusive use of idle resources is essential to deploying these kinds of systems because it is otherwise difficult to convince users to participate in them.

The mechanism to harvest idle cycles from non-idle machines should include a way to protect host processes' performance. To enforce this unobtrusiveness while achieving better efficiency, we propose to run guest jobs even on non-idle machines with very low priority for CPU, memory, I/O and network bandwidth. We call this approach *fine-grain cycle stealing* and name our system *Linger-Longer*. This enables the system to utilize most idle cycles while limiting the slowdown of the owner's workload to an acceptable level.

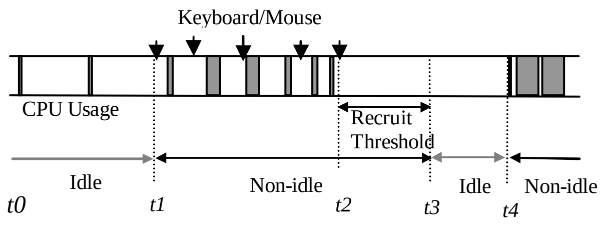

Figure 1: Idle cycles on a non-dedicated machine

Coarse-grain and fine-grain idle cycles. Idleness is determined by CPU usage, keyboard/mouse activities and recruitment threshold. A large amount of fine-grain idle cycles (empty spaces in CPU usage bar) still exist in non-idle states.

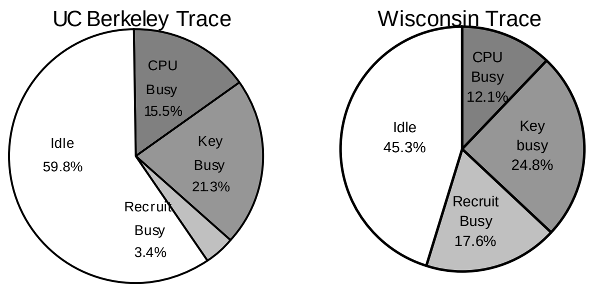

Figure 2: Non-idle time and its causes

Non-idle (busy) time percentages for different causes: CPU load (CPU Busy), keyboard/mouse activities (Key Busy) and recruitment threshold (Recruit Busy).

In [9], we demonstrated the potential of our Linger-Longer approach via simulations. In [10], our resource throttling mechanisms have been benchmarked. Finally, in this paper, we bring all of the policies, algorithms and mechanisms together and implement the Linger-Longer system by extending Condor. Using this system prototype and trace-based workload, this paper studies the performance of cycle stealing policies with respect to efficiency and unobtrusiveness.

The remainder of this paper is organized as follows. Section 2 defines idle cycles and investigates their availability. Section 3 summarizes resource policing mechanisms to support fine-grain cycle stealing. Section 4 describes integration of all the mechanisms to construct the prototype of the Linger-Longer system, and Section 5 presents a head-to-head performance comparison between our Linger-Longer system and Condor, a coarse-grain cycle stealing system. Section 6 discusses related work, and finally, Section 6 concludes with summary and future work.

2. Idle cycle stealing policies

Traditional coarse-grain cycle harvesting systems (e.g., Condor and NOW) determine machine states using three factors: CPU usage, keyboard/mouse activities and recruitment threshold. A recruitment threshold is the fixed time to be waited to ensure that the machine will not immediately return to the busy state. Thus, this interval should be included in non-idle periods. Figure 1illustrates a typical pattern of idle and non-idle intervals. Coarse-grain idle periods are the intervals (t_0, t_1) and (t_3, t_4). The non-idle interval (t_1, t_2) is mostly caused by keyboard/mouse activities whereas the second non-idle interval (t_4, ~) is due to high CPU usage. The empty spaces in the CPU usage bar are fine-grain idle cycles.

To estimate availability of idle resources, we analyzed resource usage traces of non-dedicated machines. They are comprised of two trace sets: the UC Berkeley trace collected from 132 machines for NOW project [11] and the Wisconsin trace collected by Acharya, et al [12, 13] from a Condor pool having 310 machines at the University of Wisconsin.

First, we investigated the traces as to how much of the non-idle state of the machines were contributed respectively by three different factors: high CPU load (CPU busy), keyboard and mouse activities (keyboard busy) and the recruitment threshold (recruit busy). The results are summarized in Figure 2. The machine state of the UC Berkeley trace were determined by the idleness definition of the NOW system. For the Wisconsin trace, the default Condor threshold values were used: 15 minutes for recruitment threshold and 1 minute average CPU load 0.3. As shown in Figure 2, 21.3% of the time the machines were keyboard busy in the UC Berkeley trace. The Condor trace showed similar results: 24.8% of the time the workstations were keyboard-busy. However, because the default Condor recruitment threshold of 15 minutes is much longer than that of the NOW system (1 minute), about 14% more busy time was introduced. This busy time is a good source for our fine-grain cycle stealing since its resource availability is the same as idle machines.

For memory availability, the result of the UC Berkeley trace is backed up by a study by Acharya, et al [13]. With the traces from various institutions they showed that most of the time, more than a half of main memory is unused and a larger portion of memory is available on the machines with larger main memory.

In our previous work [9], we introduced a new technique to harvest fine grain idle cycles. We term running guest jobs, while the user processes are active, lingering. Since the owner has priority over guest jobs using their personal machine, use of these idle intervals should not affect the performance of the owner's jobs (host jobs). In other words, we need mechanisms to enforce this unobtrusiveness policy.

3. Resource policing

The efficiency of resource usage can be maximized if systems implement fine-grained cycle stealing by leaving guest jobs on machine even when resource-intensive host jobs start up. However, the host job will be adversely affected unless the guest job's resource use is strictly limited. In this section, we summarize our earlier work [10, 14] on resource policing mechanisms for CPU, memory, I/O and Network bandwidth.

3.1 CPU and memory regulation

We developed the mechanisms that can allocate only unused CPU time and memory to guest jobs. First, a new *guest priority* prevents guest processes from running when runnable host processes are present. The change essentially establishes guest processes as a different class, such that guest processes are not chosen if any runnable host processes exist. This is true even if the host processes have lower runtime priorities than the guest process. Note that running with "nice –19" is not sufficient, as the nice'd process can still consume between 8%, 15%, and 40% of the CPU for Linux (2.0.32), Solaris (SunOS 5.5), and AIX (4.2), respectively [14].

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

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

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.

Both CPU and memory regulation mechanisms require simple modifications of OS kernel. In general, it is harder to gain acceptance for software that requires kernel modifications. However, we feel that modest kernel modifications are a reasonable solution 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 [15, 16], which allows fairly complex customizations of a UNIX kernel at runtime via dynamic patch.

3.2 I/O and network throttling

To enforce limits on I/O and network bandwidth, rate windows were proposed as a simple, portable, and effective strategy, analogously to the limits on CPU and memory usage. Here, we summarize our rate-window policies, and the mechanisms that are needed to support I/O throttling.

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

Rate windows were implemented using two kernel *window structures*, one for file I/O and one for network I/O. Each window structure contains a circular queue. The window structure describes the last I/O operations performed by jobs in the class (host or guest), plus a few other scalar variables.

We implemented rate windows mechanism via a loadable kernel module that intercepts each of the kernel calls for I/O and network communication: read(), write(), send(), recv(). Whenever such system functions are triggered, we first call rate_check() with the process ID, I/O length, and I/O type and then call the original system call. The process ID is used to map to an I/O class, and the I/O type is used to distinguish between file and network I/O. The rate_check() routine maintains a sliding window of operations performed for each class of service and for the overall system.

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

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

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

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

The kernel then suspends the process for delay time units before calling the original I/O system call.

With several micro benchmarks, we demonstrated that I/O and network bandwidth usage of guest jobs could be limited effectively to within a few percent of target usages [10].

Since this rate windows mechanism is implemented as a runtime loadable kernel module, it can be dynamically loaded to the regular kernel and enabled without a kernel rebuild or reboot. In addition, this system-call intercept based mechanism is lightweight and highly portable to any POSIX-compliant UNIX system.

4. System prototype

We developed a prototype of the proposed Linger-Longer system. Rather than implementing everything from scratch, we leveraged an existing system, Condor (version 6.2.0). Whereas Condor's cycle stealing policy is different from ours, it provides general mechanisms for guest job scheduling, checkpointing and migration. As a result, we could easily integrate our Linger-Longer policies and supporting modules into Condor.

The overall prototype of the Linger-Longer system is depicted in Figure 3. The leveraged Condor modules are as follows[1].

Guest Job Scheduler: this module queues the submitted guest jobs and allocates idle machines to execute them. It negotiates with local guest job starters to launch a guest job on the machine satisfying the resource requirements (OS, CPU speed, memory size and etc.).

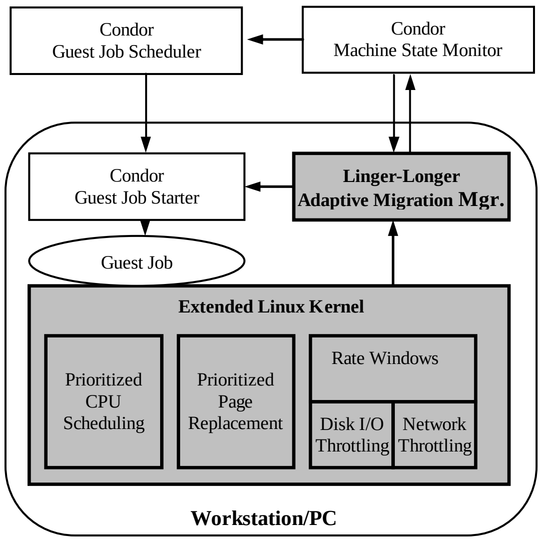

Figure 3: Linger-Longer system prototype

Machine State Monitor: this module periodically gathers resource usage (CPU load, available memory and keyboard/mouse activity) from each machine and determines which machines are idle.

Guest Job Starter: this module locally handles execution of guest jobs. It creates processes for guest jobs, starts the execution and checkpoints the current state. This module has been customized to run a guest job with the Linger priority.

Our prototype also leveraged the Condor job migration mechanism which is based on checkpointing. In Condor, a job starts migration by dumping the current process image to the checkpoint server. Then, the image is transferred to the destination machine and the execution is resumed where it checkpointed at the source machine.

Two new modules were added to enable Linger-Longer (shown in gray boxes in Figure 3). The first module contains the operating system extensions for the Linger priority. In the previous section, we already described the policies and their implementation mechanisms. To summarize, we implemented a starvation-level priority for CPU, prioritized page replacement for memory and Rate windows for efficient I/O and network throttling.

The second module is the Adaptive Migration Manager. This module replaces the existing migration policies of Condor with our cost/benefit based migration scheme. As described in [11], a guest job can linger even when the machine where the job is running becomes non-idle and migrate to another machine only when the benefit outweighs the migration cost. To measure the required parameters for Linger-Longer migration, another resource state monitor has been

[1] In the Condor system, the modules are named condor_schedd, condor_collector and condor_startd, respectively.

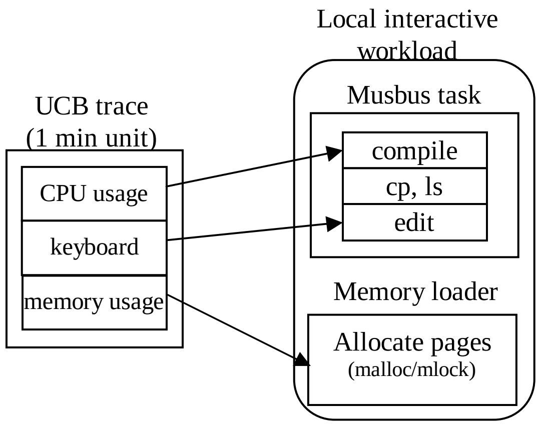

Figure 4: Interactive local workload (host jobs) generation

integrated in this module. The resource monitor can measure current CPU and physical memory usage for local jobs and guest jobs.

To enable the Linger-Longer policy, we first configured the local guest job starter to execute guest jobs at the Linger priority. All the resource requests by guest jobs will be handled by our operating system extensions to protect the performance of local host processes. In addition, existing migration (or preemption) was disabled. Rather, the Adaptive Migration Manager directly forces Condor to migrate a guest job by invoking the *condor_vacate* command.

The Linger-Longer system also can emulate coarse-grain cycle stealing policies such as Immediate-Eviction and Pause-and-Migrate as well as run new fine-grain cycle stealing policies such as Linger-Longer and Linger-Forever.

5. Performance evaluation

Now, using the Linger-Longer prototype that integrates our proposed policies and mechanisms, we compare overall performance between the fine-grain cycle stealing policies and the coarse-grain cycle stealing policies.

5.1 Workload

This section first describes the configuration of a networked machines, host job workload and guest job sets used in the experiment.

The Linger-Longer system was run in an eight machine Linux cluster. Each machine in the cluster has a 233 MHz Pentium II processor, 192 MB of memory and a 6 GB IDE hard disk. All the machines are connected by two networks, a 100 Mbps switched Ethernet and a 1 Gbps Myrinet switch. The machines are running the Linux 2.2.5 kernel with our Linger priority kernel extensions. Each guest limit parameter was set to approximately 10% of the total resource: 20 Mbytes for high memory limit, 10 Mbytes for low memory limit, 500 Kbytes/sec for both disk I/O and network bandwidth.

Modeling an interactive user workload on a personal machine is very difficult if not impossible. Also, having real computer users use the test machines is not feasible since the workload cannot be accurately reproduced to allow comparisons of different policies. Therefore, we generated the local workload based on the trace data which was used for the simulations (UC Berkeley trace). However, in this experiment, a resource usage from the trace invokes a corresponding task script. We used two scripts to simulate interactive users and to consume memory resources. First, Musbus scripts were used to emulate an computer programmer. This script is a sequence of subtasks such as editing (ed), compiling (make and gcc), copying (cp) and file listing (ls). The mapping between resource usage and script-based interactive tasks are shown in Figure 4. Every minute, a new local task is generated. For CPU usage and keyboard activity, a corresponding Musbus based script is selected and executed. The number of files to be compiled is adjusted to generate 1 minute of CPU usage. An editing subtask is invoked for the given keyboard activity duration. Second, to emulate local memory usage, a simple memory loader program (Memload) runs separately from the Musbus task. It allocates the corresponding size of memory, and loads and stores to the various memory locations.

As guest jobs, we use a set of scientific applications from the NAS NPB benchmark [17]. A serial version of the benchmark was selected since, in this experiment, we focus on sequential guest jobs. We chose three applications with three different data sizes: mg.W, sp.A and lu.B, which require 8, 65 and 165 MB memory respectively (A job name was denoted as *applicaton_name.data_size*). We also varied job duration by changing the number of iterations. Various combinations of job size and duration are shown in Table 1.

Among 9 combinations, we selected 5 representative types of guest jobs. Also, we group a number of identical jobs into a job set. The job set size was set such that

	Application.data_size (memory size)		
	mg.W (8MB)	sp.A (65MB)	lu.B (165M)
1 min (1.5 min)	***1***		***(4)***
10 min		***3***	
30 min	***2***		***5***

Table 1: Guest job sets with various job size and duration

		Idle	LL	LF	PM	IE
mg.W.1m.480	Avg Job Time	2148	3804	3732	6694	6377
	Variation	2.3%	1.1%	2.2%	52.9%	4.6%
	Family Time	4206	7528	7376	13315	12370
	Throughput	8.0	4.5	4.6	2.5	2.7
	Migration	0	0	0	7	13
mg.W.30m.16	Avg Job Time	2873	4398	4083	6789	6124
	Variation	3.8%	4.2%	8.0%	65.8%	1.9%
	Family Time	3887	6870	6230	11674	10340
	Throughput	8.0	4.5	5.0	2.7	3.0
	Migration	0	9	0	10	16
sp.A.10m.48	Avg Job Time	2145	3239	3232	5070	4995
	Variation	1.6%	2.8%	15.1%	7.8%	7.8%
	Family Time	3646	5842	5828	9496	9374
	Throughput	8.0	5.0	5.0	3.1	3.1
	Migration	0	4	0	7	14
lu.B.1.5m.320	Avg Job Time	1986	3468	3478	6287	5919
	Variation	0.0%	0.0%	0.7%	3.0%	1.5%
	Family Time	3861	6814	6836	12469	11728
	Throughput	8.0	4.5	4.5	2.5	2.6
	Migration	0	2	0	7	14
lu.B.30m.16	Avg Job Time	2622	4223	3899	5950	5446
	Variation	0.3%	23.6%	47.5%	61.0%	36.6%
	Family Time	3501	6733	6080	10190	9160
	Throughput	8.0	4.2	4.6	2.7	3.1
	Migration	0	9	0	8	10

Table 2: Guest performance for different job sets

all guest job sets could finish in a similar time although individual jobs require different CPU time. This was intended to minimize the performance changes due to changes in local resource usage over time. We chose 1 cluster hour for the job set duration. Hence, the job set size is 480 for 1 minute jobs, 320 for 1.5 minute jobs, 48 for 10 minute jobs and 16 for 30 minute jobs.

The selected five guest job sets are (a job set name is denoted in the form of *application.data_type.duration.set_size*):

Guest job set 1 (small, short): mg.W.1m.480
Guest job set 2 (small, long): mg.W.30m.16
Guest job set 3 (medium, medium): sp.A.10m.48
Guest job set 4 (large, short): lu.B.1.5m.320
Guest job set 5 (large, long): lu.B.30m.16

With these five guest job sets, we compare the cluster performance between the fine-grain cycle stealing policies: Linger-Longer(LL) and Linger-Forever(LF)[2], and the traditional coarse-grain cycle stealing policies: Immediate Eviction(IE)[3] and Pause-and-Migrate(PM). For IE and PM, the default thresholds of the Condor system were used to define idle machines; 1 minute average CPU load is below 0.3 and no keyboard/mouse activity has been detected for the past 15 minutes. This large recruit time is typically required for coarse-grain cycle stealing since its obtrusive migrations should be minimized. In contrast, for Linger-Longer and Linger-Forever, we lower the keyboard idle time to 1 minute because job migration is unobtrusive thanks to our resource regulation mechanisms. For Pause-and-Migrate, a guest job is suspended for 10 minutes before migration.

[2] Linger-Forever is the same as Linger-Longer with no migrations.
[3] Immediate Eviction sets the pause time to zero.

5.2 Experiment results

Finally, we present a head-to-head performance comparison between our fine-grain cycle stealing and traditional coarse-grain cycle stealing. The experiments ran 5 different guest job sets with 4 different policies (LL, LF, IE and PM) in an eight machine Linux cluster.

We first analyze the cluster performance for the guest job sets using five metrics. *Average completion time* is the average time to completion of a guest job. This includes waiting time before initially being executed, paused time, and migration time. *Variation* is the standard deviation of job execution time (time from first starting execution to completion). *Family Time* is the completion time of the last job in the family of processes submitted as a group[4]. *Throughput* is the average amount of processor time used by guest jobs

[4] This metric is intended to capture response time of a parameter-sweep style parallel application as its results can be used only after all its parallel tasks finish.

per second when the number of jobs in the system was held constant. *Migration* is the total number of process migrations observed. The results are summarized in Table 2. Notice that the column with the policy Idle is used as the base case where guest jobs were run on a fully idle 8 machine cluster.

For the average job time, LF is 60% to 70% better than IE for most guest job sets. However, in the case of lu.B.1.5m.320, the gain decreases to around

50%. This is due to the fact that large size guest jobs occasionally struggled to get enough memory on some non-idle machines (recall that lu.B requires 165 MB of memory where the total memory of each machine is 192 MB). LF performs slightly better than LL since it can consume almost all the available cycles from both idle and non-idle machines and hence reduce the waiting time in queue.

For the family time and the throughput, the performance gain of LL and LF is similar to that for the average job time, a 60% to 70% gain for LF and 50% to 70% for LL for most cases. Again, the improvement is smaller for lu.B.30m.16, only 50% for LF and 36% for LL. However, this difference is not surprising since, for large-memory guest jobs, fine-grain cycle stealing will be limited by the available memory. The smaller variation of LL demonstrates that guest jobs were serviced more fairly than LF.

The number of migrations is also measured for the different policies and guest job sets. For the short duration guest jobs (mg.W.1m and lu.B.1.5m), LL reduces migrations significantly since guest jobs linger on non-idle nodes and complete before getting to the break-even point between cost and benefit. For the large duration jobs (mg.W.30m and lu.B.30m), LL migrates almost the same number of guest jobs as PM. PM migrates fewer guest jobs than IE since PM avoids unnecessary migrations when a non-idle period of a machine is shorter than a pause time.

We now turn our attention to host job delay. Throughout this paper, we have strived to limit host job delay

	Delay (%)	LL	LF	PM	IE
No Guest	musbus	0.68	0.68	0.68	0.68
	memload	0.20	0.20	0.20	0.20
mg.W.1m.480	musbus	0.82	0.88	1.44	2.05
	memload	0.25	0.24	0.32	0.42
mg.W.30.16	musbus	1.03	1.10	1.85	1.85
	memload	0.24	0.24	0.41	0.43
sp.A.10m.48	musbus	1.16	1.11	2.20	2.34
	memload	0.26	0.25	0.46	0.45
lu.B.1.5m.320	musbus	2.19	1.91	1.77	1.92
	memload	0.14	0.27	0.20	0.10
lu.B.30m.16	musbus	2.81	2.98	2.66	2.20
	memload	0.24	0.44	0.22	0.17

Table 3: Host job slowdown for different guest job sets

caused by guest jobs. So, every 1 minute, we measured the delay of two host tasks, Musbus and Memload, which have been described early in this chapter. For Musbus, we computed delay by subtracting the base Musbus time from the measured time. The base time for all possible configurations (CPU usage for every 10% between 0 to 100%, both with and without edit script) was measured by running Musbus on a fully idle node.

The average delay for each guest job set is shown in Table 3. Musbus without guest jobs produced some delay (0.68%) as shown in the first column of the table. It means that the 0.68% delay is beyond the measurement precision. Interestingly, for small and medium size guest jobs, LL and LF exhibits less delay than PM and IE.

A histogram of Musbus delay with mg.W.30m.16 is shown in Figure 5. For PM and IE, there exist more delays, between 6% - 12%, than for LL and LF. These delays were caused by job migration in PM and IE due to lack of resource prioritizing mechanisms. In LL and LF, migration itself uses only idle resources at the Linger priority.

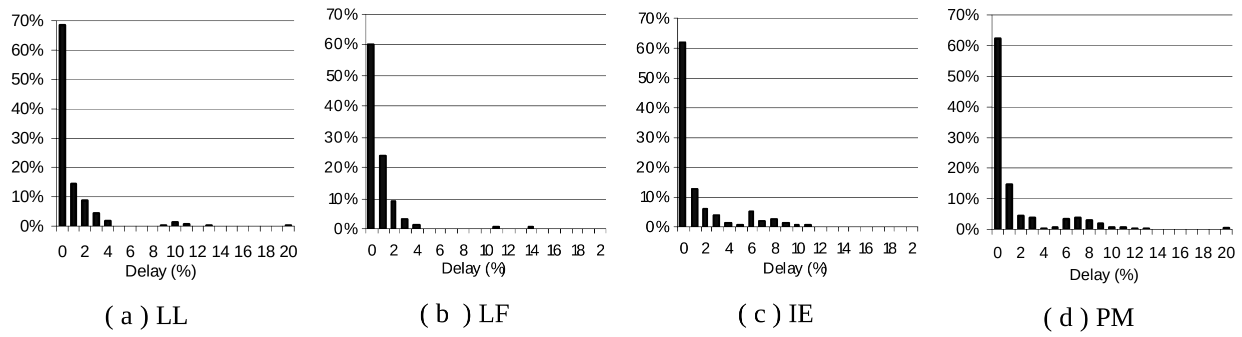

Figure 5: Musbus delay for mg.W.30m.16

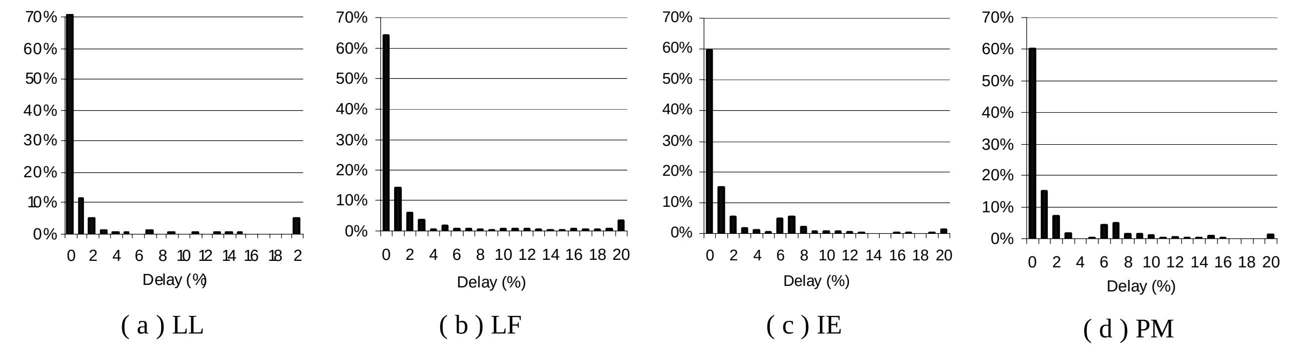

Figure 6: Musbus delay for lu.B.30m.16

For large size guest jobs (lu.B), LL and LF shows more Musbus delay than PM and IE. The delay histogram for the guest job set lu.B.30m.16 is shown in Figure 6. In the histograms for LL and LF, there are some instants whose delay exceeded 20%. These occasional delays were caused since Musbus and the large memory guest jobs were running at the same time. However, no such noticeable delays were detected for Memload as shown in Figure 7. This demonstrates that our current prioritized memory replacement effectively protected the memory pages of Memload. In contrast, Musbus was occasionally delayed since our memory replacement mechanism could not prevent a large memory guest job from sweeping the file buffer cache (recall that Musbus contains a compile workload). However, despite the fact that lu.B was very aggressive in using memory, these noticeable delays occurred less than 5% of the time.

Although most of the metrics for this experiment match the estimated performance gain in the simulation study in [18], the throughput (the equivalent number of idle machines) does not. In the prototype experiment, the equivalent idle machines are 4 to 5 on an 8 machine cluster (50% to 62%) whereas they were 52 to 55 on the 64 machine cluster simulations (80% to 85%). This is due to the fact that the mean CPU utilization in the traces used had a higher load than the traces used in the simulation study. We believe that increasing the number of test machines by using more trace data will produce results closer to the simulations.

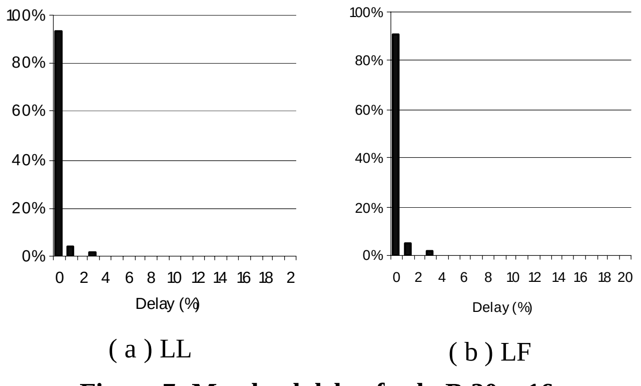

Figure 7: Memload delay for lu.B.30m.16

6. Related work

Previous work on exploiting available idle time on workstation clusters used a conservative model that would only run processes when the local user was away from their workstation, and no local processes were runnable. Condor [2], LSF [3], and NOW [11] 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.

A system that used non-idle workstations was the Stealth distributed scheduler [19]. 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. Commercial software from Entropia [20] also supports fine-grain cycle stealing on desktop PCs running MS Windows. However, it is not clear how unobtrusiveness is provided.

In the area of operating system support for providing resource management, research and commercial operating systems have provided similar functionality. In IRIX [21], 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 the opposite of our approach, which promotes interactive jobs.

Verghese et al [22] 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 [23] 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 [24] and Resource Containers [25] can group a set of processes or threads as a unit for resource scheduling. This is similar to our job classes. The Nemesis kernel [26] also provides QoS with rate-based real-time scheduling for I/O as well as CPU. However, those systems are integrated deep into the kernel, while our mechanism resides between the kernel and the user-level I/O library and can be loaded and unloaded at run-time. Our mechanism is lightweight since we do not add any extra queues for resource scheduling. Our mechanism just intercepts resource requests, keeps track of the rate, and puts them into sleep for an appropriate time if the requests seem to exceed the limit. However, our rate windows mechanisms can be used as a lightweight and portable scheduling mechanism to support those concepts.

The idea of regulating network traffic rates has been extensively studied. Congestion avoidance schemes such as leaky bucket [27] and its variants [28, 29] use averages over various time intervals to determine which traffic is within its negotiated bandwidth. However, since these approaches are designed for policing traffic at routers, they must drop non-conforming traffic. Contrarily, since our approach is at the source, we can delay traffic to enforce bandwidth limits.

The idea of resource partitioning using virtual machines has been popular both in the 1970s [30] as well as in recent projects such as Disco [31]. 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.

7. Conclusions and future work

In this paper, we presented the design, implementation, and performance of fine-grain cycle stealing along with a suite of resource policing mechanisms that provide the vital safety net for unobtrusiveness.

We implemented a prototype fine-grain cycle stealing system, Linger-Longer. The operating system extension for starvation-level CPU priority, prioritized memory replacement and Rate windows for I/O and network throttling have been integrated to the prototype. Also, a new adaptation migration module was added. For the guest job scheduling, migration and checkpointing mechanisms, we leveraged the Condor system.

Using the prototype, we conducted the experiments on a desktop Linux cluster. We ran a group of guest job sets from NPB benchmark, with various sizes and durations. The local workload was generated using Musbus benchmark, a script based interactive workload. The configuration of each instance of Musbus was driven by the resource usages of the UC Berkeley trace.

The results demonstrated that fine-grain cycle stealing can significantly increase efficiency in using idle cycles while limiting obtrusiveness to machine owners. Fine-grain cycle stealing policies, LL and LF, improved the cluster throughput by 50% to 70% for most cases. However, this gain can be reduced to 50% if we run guest jobs that require more than 50% of total memory of the machine. For all the cases, acceptable unobtrusiveness was achieved; on the average, the host job slowdown was limited within 3%.

To measure the benefits of Linger-Longer in a more prevalent PC Grid environment, we are currently conducting experiments using Window-based interactive workload that includes web browsers and GUI-based text editors.

References

1. Mutka, M.W. and M. Livny, *The available capacity of a privately owned workstation environment.* Performance Evaluation, 1991. **12**: p. 269-284.

2. Litzkow, M., M. Livny, and M. Mutka. *Condor - A Hunter of Idle Workstations*. in *International Conference on Distributed Computing Systems*. 1988.

3. Zhou, S., et al., *Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems.* SPE, 1993. **23**(12): p. 1305-1336.

4. Anderson, T.E., D.E. Culler, and D.A. Patterson, *A case for NOW (Networks of Workstations).* IEEE Micro, 1995. **15**(1): p. 54-64.

5. IBM, *IBM LoadLeveler: General Information.* 1993, Kingston, NY.

6. Green, T. and J. Snyder, *DQS, A Distributed Queueing System.* 1993, Florida State University.

7. Dannenberg, R.B. and P.G. Hibbard, *A Butler Process for Resource Sharing on Spice Machines.* ACM Transactions on Office Information Systems, 1985. **3**(3): p. 234-52.

8. Sullivan, W.T., et al. *A new major SETI project based on Project Serendip data and 100,000 personal comuputers*. in *Intl. Conf. on Bioastronomy*. 1997: Edutruce Compositori.

9. Ryu, K.D. and J.K. Hollingsworth, *Exploiting Fine Grained Idle Periods in Networks of Workstations.* IEEE Transactions on Parallel and Distributed Computing, 2000. **11**(7).

10. Ryu, K.D., J. Hollingsworth, and P. Keleher. *Efficient Network and I/O Throttling for Fine-Grain Cycle Stealing*. in *SC'2001*. 2001. Denver, CO.

11. Arpaci, R.H., et al. *The Interaction of Parallel and Sequential Workloads on a Network of Workstations*. in *SIGMETRICS*. 1995. Ottawa.

12. Acharya, A., G. Edjlali, and J. Saltz. *The Utility of Exploiting Idle Workstations for Parallel Computation*. in *SIGMETRICS'97*. 1997. Seattle, WA.

13. Acharya, A. and S. Setia. *Availability and Utility of Idle Memory in Workstation Clusters*. in *ACM SIGMETRICS*. 1999. Atlanta, GA.

14. Ryu, K.D., J.K. Hollingsworth, and P.J. Keleher. *Mechanisms and Policies for Supporting Fine-Grained Cycle Stealing*. in *ICS*. 1999. Rhodes, Greece.

15. Tamches, A. and B.P. Miller. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels*. in *Third Symposium on Operating Systems Design and Implementation (OSDI)*. 1999. New Orleans.

16. Pearce, D., et al. *GILK: A Dynamic Instrumentation Tool for the Linux Kernel*. in *International Conference on Computer Performance Evaluation (TOOL)*. 2002. London England.

17. Bailey, D.H., et al., *The NAS Parallel Benchmarks*. International Journal of Supercomputer Applications, 1991. **5**(3): p. 63-73.

18. Ryu, K.D. and J.K. Hollingsworth. *Linger Longer: Fine-Grain Cycle Stealing for Networks of Workstations*. in *SC'98*. 1998. Orlando, FL.

19. Krueger, P. and R. Chawla. *The Stealth Distributed Scheduler*. in *International Conference on Distributed Computing Systems (ICDCS)*. 1991. Arlington, TX.

20. Entropia, *http://www.entropica.com*.

21. SiliconGraphics, *IRIX 6.4 Technical Brief*. 1998.

22. Verghese, B., A. Gupta, and M. Rosenblum. *Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors*. in *ASPLOS*. 1998. San Jose, CA.

23. Aron, M. and P. Durschel. *Soft Timers: efficient microsecond software timer support for network processing*. in *SOSP*. 1999. Kiawah Island, SC: ACM.

24. Bruno, J., et al. *The Eclipse operating system: Providing Quality of Service via Reservation Domains*. in *USENIX 1998 Annual Technical Conference*. 1998. New Orleans, Louisiana.

25. Banga, G., P. Druschel, and J. Mogul. *Resource containers: A new facility for resource management in server systems*. in *USENIX 3rd Symposium on Operating System Design and Implementation*. 1999. New Orleans, LA.

26. Reed, D. and R. Fairbairns, *The Nemesis Kernel*. 1997, United Feature Syndicate, Inc.

27. Turner, J.S., *New Directions in Communications (or Which Way to the Information Age?)*. IEEE Communications Magazine, 1986. **24**(10): p. 8-15.

28. Zhang, L. *Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks*. in *SIGCOMM*. 1990.

29. Faber, T., L.H. Landweber, and A. Mukherjee. *Dynamic Time Windows: packet admission control with feedback*. in *SIGCOMM*. 1992.

30. Goldberg, R.P., *Survey of Virtual Machine Research*. IEEE Computer Magazine, 1974. **7**(6): p. 34-45.

31. Bugnion, E., S. Devine, and M. Rosenblum. *Disco: Running Commodity Operating Systems on Scalabe Multiprocessors*. in *SOSP*. 1997.