Improving Cluster Utilization through Intelligent Processor Sharing

Gary Stiehr
Roger D. Chamberlain

Gary Stiehr and Roger D. Chamberlain, “Improving Cluster Utilization through Intelligent Processor Sharing,” in *Proc. of Workshop on System Management Tools for Large-Scale Parallel Systems,* April 2006.

Dept. of Computer Science and Engineering
Washington University
Campus Box 1045
One Brookings Dr.
St. Louis, MO 63130-4899

Improving Cluster Utilization through Intelligent Processor Sharing

Gary Stiehr and Roger D. Chamberlain
Dept. of Computer Science and Engineering, Washington University in St. Louis
garystiehr@wustl.edu, roger@wustl.edu

Abstract

A dedicated cluster is often not fully utilized even when all of its processors are allocated to jobs. This occurs any time that a running job does not use 100% of each of the processors allocated to it. We increase the throughput and efficiency of the cluster by scheduling background jobs to run concurrently with the "primary" jobs originally scheduled on the cluster. We do this while maintaining the quality of service provided to the primary jobs. Our results come from empirical measurements using production applications.

1. Introduction

In a cluster environment, where processors are only allocated to a single application at a time, some parallel applications do not fully utilize the processors to which they have been assigned. For example, certain classes of applications, when parallelized, require a high amount of communication between individual parallel processes running on different processors. Depending on the latency and bandwidth of the connection between the individual processors, the frequency and/or size of inter-process messages can cause these processes to underutilize the processors on which they are running. This is particularly prevalent in dedicated clusters where a relatively high-overhead network, such as Ethernet, connects the processors.

It is desirable to allow other "guest" processes to run in the background concurrently with these applications and benefit from the unused processor time. This would help to increase the throughput and efficiency of the cluster. We do not, however, want to interfere with the primary task being performed on a given processor. To help avoid interference, we do not want to schedule other processes in the background that utilize the same resource that is the bottleneck for the primary task. In addition, we must be able to control the guest processes' resource usage to keep them from interfering with the primary task (i.e., we must maintain the primary task's quality of service).

We envision a two-tier cluster scheduling system, in which jobs are partitioned into two priority classes. High-priority jobs are scheduled in a traditional manner, consuming resources as they are allocated to processors for execution. Low-priority jobs (also called guest jobs) are scheduled on the same processing nodes as the high-priority jobs; however, the underlying OS scheduler is asked to ensure that the low-priority jobs only consume otherwise underutilized resources (i.e., the priority is strict).

This paper presents empirical results that quantify the impact of allowing guest processes to co-exist with primary processes on a dedicated compute cluster. We refer to this as *intelligent processor sharing*. The goal is to minimize the impact of the guest processes on the execution of the primary processes (which we define as maintaining the primary processes' quality of service) while allowing the guest processes to use otherwise unused compute cycles on the system. We evaluate a modification to the Linux kernel's process scheduler that supports the control of guest processes' resource usage. The modification was developed as part of the Linger-Longer system [1,2,3] to provide fine-grained cycle stealing in a network of workstations. In this work, we apply the concept to a dedicated cluster.

Using a set of production applications from the Linux cluster installed at the University of Missouri – St. Louis (UMSL), we quantify the impact on overall system throughput, efficiency, and response time as well as quality of service for primary processes using two mechanisms that support varying the service priority for jobs. The first mechanism is the traditional 'nice' facilities built into the standard Linux scheduler, and the second mechanism is the kernel modifications provided by the Linger-Longer system.

We found that using the existing nice mechanism significantly improves the throughput, efficiency and average turnaround time of the cluster but only at the expense of the quality of service of the primary jobs (primary job run times increased 5-25%). On the other hand, when using intelligent processor sharing we get similar improvements in throughput, efficiency and average turnaround time while not significantly impacting the quality of service of the primary jobs (primary job run times changed less than 1%).

2. Related Work

A goal of this work is to increase the throughput and efficiency of a dedicated cluster by exploiting its available idle time. At the same time, we would like to maintain the average turnaround time and the quality of service of the primary jobs. Condor [4], LSF [5], NOW [6], and Linger-Longer [1,2,3,7] all present ways to exploit the available idle time in a network of non-dedicated workstations while taking steps to limit the impact on the owner of the workstation.

1-4244-0054-6/06/$20.00

Condor, LSF and NOW attempt to limit the impact on the user by removing guest processes whenever CPU activity is generated by the primary processes (i.e., the processes that the workstation owner has started). This makes these techniques ineffective in a dedicated cluster environment.

Linger-Longer is based on Linux kernel modifications that allow the guest processes to persist on the owner's workstation despite CPU activity from the primary processes. The kernel modifications keep the guest processes from being scheduled on the CPU when primary processes are runnable. Once the primary processes' CPU activity subsides, the Linger-Longer system will allow the guest processes to proceed. The Linger-Longer system has been evaluated using benchmarks, models and simulations. In [1], benchmarks were run as host processes and guest processes. The effect on the run time of the host processes was measured. In [2] and [3], the efficiency with which idle cycles were used by guest jobs as well as the change in the throughput of guest jobs was evaluated using simulation. In [7], the impact of I/O and network resources is considered, which is beyond the scope of the present work.

We are not proposing modifications to the cluster-level scheduler. The node scheduler modifications would be made on each of the computers comprising the dedicated cluster. Any cluster-level scheduler, such as those found in Condor, LSF, PBS [8] and Sun GridEngine [9], can be used to allocate processors to the primary jobs. Additionally, any cluster-level scheduler can then be used to fill the CPU time not being used by the primary jobs by scheduling guest jobs with complimentary resource requirements to run concurrently.

3. Experimental Environment and Applications

The production Linux cluster mentioned above consists of thirty-two dual processor servers. Of the sixty-four available processors, thirty-four are 1.4 GHz Pentium IIIs and thirty are 1 GHz Pentium IIIs. All of the nodes are on the same 100 Mb/s Ethernet network. That is, each node has one full-duplex 100 Mb/s Ethernet connection to a network switch. Each node has 1 GB of 133 MHz SDRAM and an 18 GB Ultra3 SCSI (160 MB/s) 10,000 RPM hard drive. Each node runs an SMP Linux kernel (2.4.x). The resource management software used on the cluster is OpenPBS [8], commonly referred to as PBS. Parallel processes may be started via the PBS API or via other methods, such as MPICH [10].

For our experiments, we used four nodes from the production cluster. To reduce the degrees of freedom in our experiments, we chose to boot each of these nodes with a 2.4.20 uniprocessor Linux kernel. This effectively made the nodes single-processor servers. As a result, each parallel application we ran used a maximum of four processors each on a different node. This means that each processor had exclusive access to the entire 1 GB of memory in the node.

A number of applications were used for this investigation. As shown in Table 1, these applications are either CPU-bound or I/O-bound and consist of either one sequential process or four parallel processes (i.e., their degree of parallelism is either one or four). The I/O-bound applications vary in their level of CPU usage. With the exception of HPL, these applications are in use by the cluster's users to do production runs for research purposes.

Table 1. Application properties.

Name	CPU-bound	I/O-bound	Degree of Parallelism
GAUSS	X		1
HAL1	X		4
HPL		X	4
MrBayes		X	4
WRF		X	4
PAUP	X		1

GAUSS is a mathematical and statistical package that provides a matrix programming language [11]. It is used by a research group in the UMSL Economics Department. HAL1 was developed in the UMSL College of Business Administration and uses an intelligent enumeration algorithm to analyze possible locations for hub arcs in a logistics network [12]. It is parallelized using MPICH. HPL is a software package that solves a (random) dense linear system of equations. It is a freely available implementation of the High Performance Computing Linpack Benchmark [13]. HPL is used as a benchmark code on the cluster and is an MPI program. MrBayes is a program for the Bayesian estimation of phylogeny [14]. It is used by the UMSL Biology Department. WRF is a flexible, state-of-the-art atmospheric simulation system that is suitable for use in a broad range of applications across scales ranging from meters to thousands of kilometers [15]. Finally, PAUP (Phylogenetic Analysis Using Parsimony) is a package for inference of evolutionary trees [16]. It is used by researchers in the UMSL Biology Department. Additional details on the cluster, the individual applications, and the experimental setup and data gathering methodology can be found in [17].

The empirical results come from 7 sets of experiments that are grouped into 3 categories. Group A (sets 1 and 2) provide baseline data on each application when run in isolation. Group B (sets 3 and 4) measure the impact on the quality of service provided to primary jobs when other jobs are executed

concurrently. Group C (sets 5, 6, and 7) measure the overall impact of processor sharing in the cluster. The naming convention for experiments is s*X*e*Y*, where *X* identifies the set and *Y* identifies the individual experiment within the set.

4. Scheduler Modifications

The 2.4 Linux kernel was updated so that if a process is given the lowest nice value of 19, it will be considered a guest process. In the modified kernel, if a normal process is runnable, it will always be chosen to run over a guest process. Any statistics normally gathered by the kernel, such as the "goodness" value, are ignored for guest processes when being compared to a normal process. When deciding between multiple guest processes, however, the normal CPU scheduling mechanisms are used including any statistics kept for these processes.

```
next = DUMMY_PROCESS;
weight = -1000;
Foreach runnable process p do {
  process_weight = calculate_weight(p);
  If ( process_weight > weight )
    weight = process_weight;
    next = p;
}
schedule_on_CPU(next);
```

(a) original scheduler

```
next = DUMMY_PROCESS;
weight = -1000;
Foreach runnable process p do {
  process_weight = calculate_weight(p);
  if ((p == guest process) XOR
      (next == guest process)) {
    if (( p is not a guest process) ||
        (weight < 0)
      weight = process_weight;
      next = p; }
  else {
/* both are primary OR guest processes */
    if ( process_weight > weight )
       weight = process_weight;
       next = p; }
}
schedule_on_cpu(next);
```

(b) modified scheduler

Figure 1. Modified Linux 2.4 CPU scheduler.

As provided from its developers, the Linger-Longer kernel modifications were for the 2.0.x and 2.2.x Linux kernels. In those kernels, the nice value assigned to a process was its priority. In the 2.4.x Linux kernel, the nice value is just one of the quantities used to calculate the priority of a process at any given time. Over time, the priority of a given process in the 2.4 kernel will change. At each call to the CPU scheduler, the weight or "goodness" value for a process is calculated. The runnable process with the highest weight is selected to run. When a primary process is runnable, a guest process will never be run even if it has a higher weight according to the kernel's normal scheme. However, if there are no runnable primary processes, but one or more guest processes, the guest process with the highest weight will run (i.e., the normal kernel scheduling algorithm is applied). As provided, the Linger-Longer kernel code relied on the priority being set to the nice value. It considered a process with a priority of 19 (the lowest nice value inside the kernel) to be a guest process. This worked with the static priorities of the 2.0.x and 2.2.x kernels. However, we cannot use this method to classify guest processes in the 2.4 kernel since the priority changes over time. We instead looked explicitly at the nice value of the process to determine if it was a guest process. Pseudocode for the core of the original 2.4 Linux kernel CPU scheduler and our modified CPU scheduler is shown in Figure 1.

5. Experiments and Results

Baseline Application Performance (Group A)

The initial group of experiments is intended to establish the baseline performance for each of the applications used. Group A consists of two experiment sets. Set 1 consists of each application being executed in isolation using an unaltered kernel. Set 2 consists of each application being executed on the modified kernel. Throughout the paper, presented performance results represent the mean of three independent executions of the same experiment. The variation across these independent executions was quite small. Further details are available in [17]. Figure 2 and Figure 3 show the run (wall clock) time and processor efficiency for the group A experiment sets. Table 2 shows the applications used in experiments e1 through e6 of sets s1 and s2 as referenced in Figure 2 and Figure 3.

Table 2. Applications used in sets s1 and s2.

Experiment	Application Name
e1	GAUSS
e2	HAL1
e3	HPL
e4	MrBayes
e5	WRF
e6	PAUP

This data helps us to confirm the claims in Table 1 about the properties of the applications. The high efficiency measured for GAUSS, HAL1, and PAUP demonstrate that they are CPU-bound. Likewise, we can see that HPL, MrBayes and WRF are not CPU-bound in our experiments (the properties of the cluster interconnect network slowed these applications down).

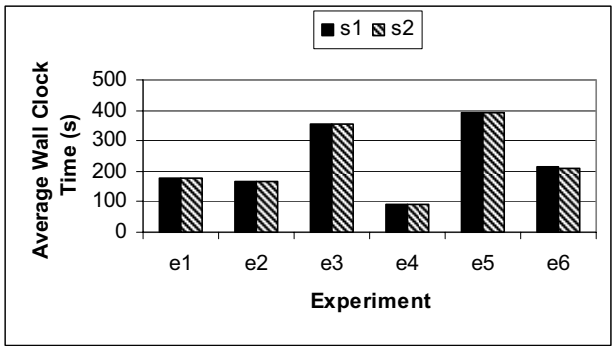

Figure 2. Application run time.

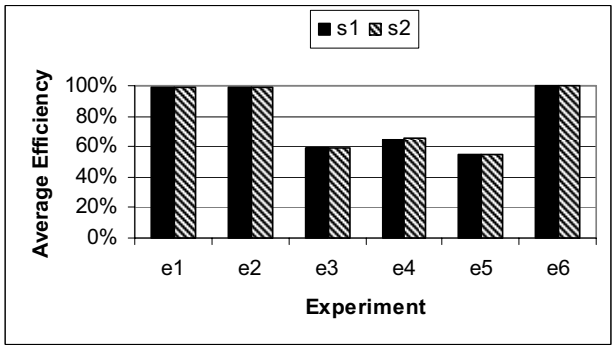

Figure 3. Application efficiency.

Impact on Quality of Service for Primary Applications (Group B)

The next group of experiments is used to explore the impact that low-priority processes have on the quality of service of primary processes. Group B consists of two experiment sets. Set 3 uses the unaltered kernel and its built-in nice mechanism, and set 4 uses the modified kernel that supports intelligent processor sharing. Table 3 shows the group B experiments; applications in parentheses are started simultaneously and applications in the third column are only requesting a low level of service (i.e., they are scheduled as "nice" or as a "guest").

Table 3. Group B experiments.

Experiment	Primary Application	"Nice" or "Guest" Applications
e1	GAUSS	GAUSS
e2	MrBayes	(GAUSS, GAUSS, GAUSS, GAUSS)
e3	HAL1	PAUP
e4	WRF	PAUP, PAUP
e5	WRF	HAL1
e6	HAL1	PAUP
e7	HAL1	PAUP, PAUP, PAUP, PAUP
e8	HAL1	PAUP, PAUP, PAUP

We tested four of the applications: HAL1, WRF, MrBayes and GAUSS. These particular applications were chosen so that we would have half I/O-bound applications (WRF, MrBayes) and half CPU-bound applications (HAL1, GAUSS). The low-priority jobs were CPU-bound applications chosen at random except as follows. In experiment e6, the single guest PAUP process was run concurrently with one HAL1 slave process; in experiment e7, the four PAUP processes were run currently with all HAL1 processes; and in experiment e8, the three PAUP processes were run concurrently with the three HAL1 slave processes.

Figure 4 shows the wall clock times for the primary applications, and Figure 5 shows the percent change from the baseline wall clock times for the primary applications (our measure of quality of service). The wall clock time is the time elapsed between when the primary application's first process started and when its last process exited. The figures show that running a low-priority job concurrently with a primary job using the unaltered kernel consistently impacts the quality of service of the primary job. For an I/O-bound job, we would expect that the impact would be less since it does not generally use the full processor and thus is not giving up as much CPU time as a CPU-bound job. This intuition is supported by the data presented. The two I/O-bound jobs, MrBayes and WRF, have the lowest percent increase in run time.

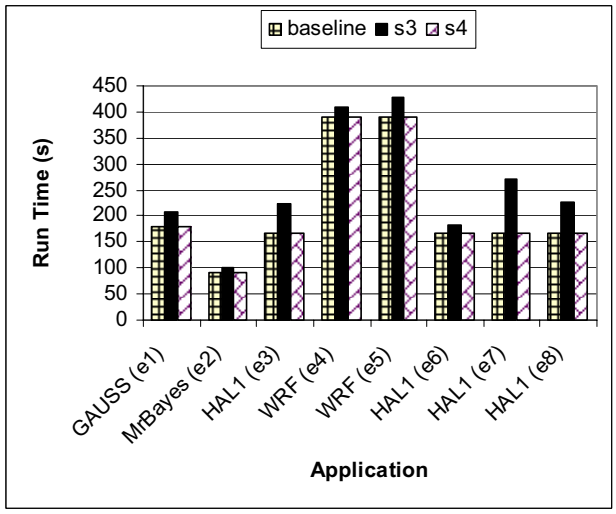

Figure 4. Primary application run time.

While experiment set 3 used the unmodified kernel, experiment set 4 shows how the run times of the primary jobs have been affected by running a guest job concurrently using the intelligent processor sharing kernel. We see here that the kernel modifications have had the desired effect. That is, the run times of the primary jobs have not changed significantly from the baseline run times found in the set 2 experiments. Note that the largest percent increase from set 4 shown

in Figure 5 is well under 1%, making it difficult to see in the figure. We can see that the kernel modifications have virtually eliminated the impact on the primary jobs that is seen when using the kernel's existing "nice" mechanism to enable low-priority processes. Since we use the run time as our measure of the quality of service received by the primary jobs, we can say that running guest jobs concurrently with the primary jobs does not impact the quality of service that the primary jobs receive when using the intelligent processor sharing kernel.

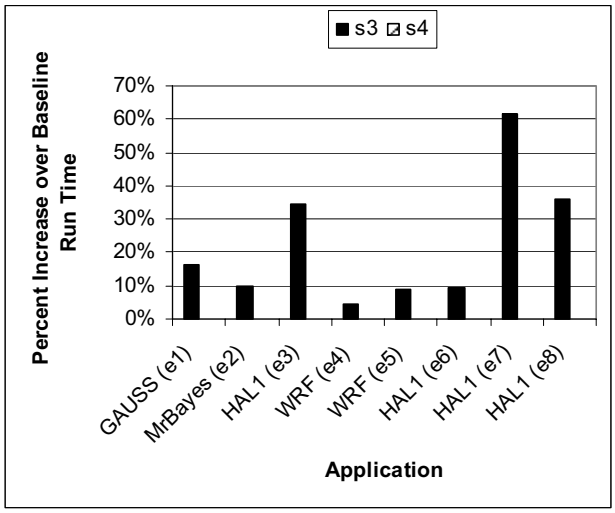

Figure 5. Impact on primary application quality of service.

Throughput, Efficiency, and Response Time (Group C)

The final group of experiments is used to assess the effect of intelligent processor sharing on throughput, efficiency, and response time in the cluster. Group C consists of three experiment sets. Set 5 is shown below. Applications in parentheses are started simultaneously and applications in brackets are only requesting a low level of service (i.e., they could be scheduled as "nice" or as a "guest"). Experiments s5e1 and s5e2 specifically start only CPU-bound primary jobs. Experiments s5e3 and s5e4 specifically start only I/O-bound primary jobs. The remaining experiments start a mixture of both CPU- and I/O-bound primary jobs. As mentioned previously, only CPU-bound jobs were run as low-priority jobs since one of our goals is to increase the CPU utilization of the cluster. Except where noted above, the sequences of applications were chosen somewhat randomly. In some cases we attempted to schedule the low-priority jobs so that they would, when run concurrently with the primary jobs later (in set 6), fill up as much of the unused CPU time as possible. Additionally, the sequences we chose were guided in at least a small way by the sequences of job submissions seen on the production cluster of which our test nodes were a subset. The experiments in set 5 are:

s5e1: HAL1, (PAUP, PAUP, GAUSS, GAUSS), [HAL1], [HAL1]
s5e2: (PAUP, PAUP, PAUP, PAUP), [HAL1], ([GAUSS], [GAUSS], [GAUSS], [GAUSS])
s5e3: MrBayes, WRF, HPL, [HAL1], ([PAUP], [PAUP], [GAUSS], [GAUSS]), ([PAUP], [PAUP], [GAUSS], [GAUSS])
s5e4: HPL, [HAL1], WRF, ([GAUSS], [PAUP], [PAUP], [GAUSS]), [HAL1]
s5e5: WRF, ([GAUSS], [GAUSS], [GAUSS], [GAUSS]), ([PAUP], [PAUP], [PAUP], [PAUP]), HAL1
s5e6: [HAL1], ([PAUP] , [PAUP] , [PAUP] , [PAUP]), MrBayes, HPL
s5e7: WRF, [HAL1], ([GAUSS], [GAUSS], [GAUSS], [GAUSS])
s5e8: MrBayes, MrBayes, (PAUP, PAUP, PAUP, PAUP), [HAL1], [HAL1]

The remaining experiment sets in group C (sets 6 and 7) use the same applications as those in set 5; the difference is in how they are run. For these two sets, low-priority jobs are allowed to run concurrently with the primary jobs. For these experiments, if only low-priority jobs are running and a primary job is submitted, the primary job will immediately be allocated the processors it needs. If other primary jobs are running and there are not enough free processors, then the newly submitted primary job will be queued.

Table 4. Experiment sets 6 and 7.

e1	HAL1, (PAUP, PAUP, GAUSS, GAUSS)
	[HAL1], [HAL1]
e2	(PAUP, PAUP, PAUP, PAUP)
	[HAL1], ([GAUSS], [GAUSS], [GAUSS], [GAUSS])
e3	MrBayes, WRF, HPL
	[HAL1], ([PAUP], [PAUP], [GAUSS], [GAUSS]), ([PAUP], [PAUP], [GAUSS], [GAUSS])
e4	HPL, WRF
	[HAL1], ([GAUSS], [PAUP], [PAUP], [GAUSS]), [HAL1]
e5	WRF, HAL1
	([GAUSS], [GAUSS], [GAUSS], [GAUSS]), ([PAUP], [PAUP], [PAUP], [PAUP])
e6	MrBayes, HPL
	[HAL1], ([PAUP], [PAUP], [PAUP], [PAUP])
e7	WRF
	[HAL1], ([GAUSS], [GAUSS], [GAUSS], [GAUSS])
e8	MrBayes, MrBayes, (PAUP, PAUP, PAUP, PAUP)
	[HAL1], [HAL1]

Table 4 shows the sequence of applications run for experiment sets 6 and 7; applications in parentheses are started simultaneously and applications in brackets are only requesting a low level of service (i.e., they are scheduled as "nice" in set 6 or as a "guest" in set 7). Experiments e1 and e2 specifically start only CPU-bound primary jobs. Experiments e3 and e4 specifically start only I/O-bound primary jobs. The remaining experiments start a mixture of both CPU- and I/O-bound primary jobs. The experiments are the same as in set 5 but the order of execution was done as if there were two queues that could use the same set of processors simultaneously. The experiments are:

We measured the throughput of the cluster by taking the number of jobs run (including both primary and low-priority) and dividing it by the number of hours (i.e., seconds/3600) required for all jobs to complete (including both primary and low-priority). The efficiency measurement in group C is a calculation of how efficiently the set of jobs in a given experiment used the set of processors assigned to it. For each experiment, we considered its set of jobs to be the only jobs queued. We summed the CPU usage of each process (including all parallel processes); call this T_c. We then noted the amount of wall clock time, T_w, needed until the last job in the set completed. We considered that each of the four processors was available for use this entire time. Thus we calculated the available processor time as $4T_w$. We then calculated the efficiency as $T_c/(4T_w)$.

We also measured the average turnaround time of the jobs in each experiment. The turnaround time for a given job is the time that elapsed from when the job was submitted to the queue (we assume that all jobs were submitted at the same time (i.e., at time 0)) and the time that it finished executing. The average turnaround time for a given experiment is found by summing the turnaround time of every job in the experiment and dividing this sum by the number of jobs in the experiment. Finally, as a measure of the quality of service received by the primary jobs in each experiment, we report the wall clock time needed for all primary jobs to complete.

Figure 6 and Figure 7 compare the throughput measurements from the set 5 experiments (primary jobs only) to the measurements obtained for the set 6 experiments (primary jobs with low-priority jobs run concurrently) and the set 7 experiments (primary jobs with guest jobs run concurrently). These results show that the throughput of the cluster-level scheduler can be increased by running guest jobs concurrently with primary jobs. The throughput improvements with intelligent processor sharing (set 7) are very similar to those achieved by running low-priority jobs concurrently with the primary jobs (in set 6). As we would expect, the increase in throughput is related to the efficiency with which the set of primary jobs (in the set 5 experiments) utilized the processor. Experiments in set 5 with all CPU-bound jobs (such as e1 and e2) obtained higher efficiency and thus there was little or no room for improvement in throughput in set 7; thus the gains are minimal or non-existent. On the other hand, experiments e3 and e4 were I/O-bound, resulting in lower efficiency when run alone (in set 5) and thus resulting in bigger gains in throughput when run with guest jobs in set 7. Obviously as throughput increases, so does the efficiency of the cluster. Figure 8 compares the initial efficiency (set 5) with the efficiency when running concurrently with low-priority jobs (set 6) and guest jobs (set 7).

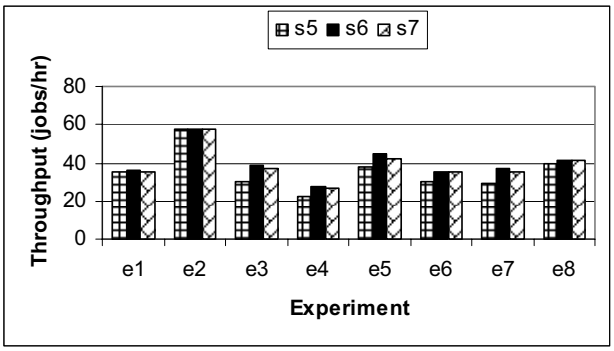

Figure 6. Throughput (s5) vs. throughput with low-priority jobs (s6) and throughput with guest jobs (s7).

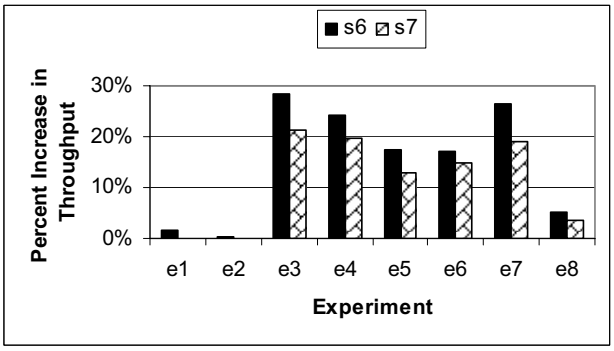

Figure 7. Throughput improvement for low-priority jobs (s6) and guest jobs (s7).

Figure 9 plots the average turnaround time (for all applications) in group C. It demonstrates that we achieve higher throughput while also lowering the average turnaround time of the jobs for most experiments. Clearly the amount that we can decrease average turnaround time depends on how many of the jobs can finish sooner than before. If we have CPU-bound primary jobs (e.g., e1 and e2) there will not be any CPU time available to run guest jobs. Since the guest processes are run only when there are no runnable primary processes (set 7), the guest jobs will

not take any CPU time away from the primary jobs. Thus the guest jobs' average turnaround time should not change nor should the primary jobs' average turnaround time.

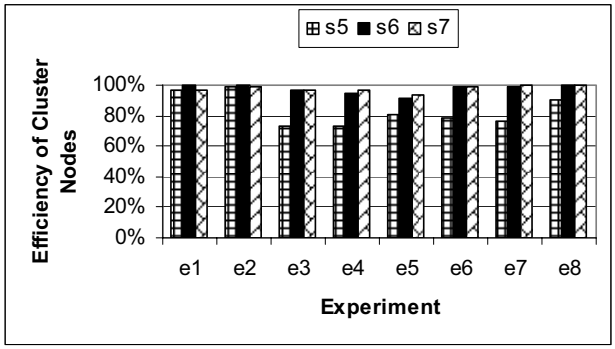

Figure 8. Efficiency (initial (s5) vs. low-priority jobs (s6) and guest jobs (s7)).

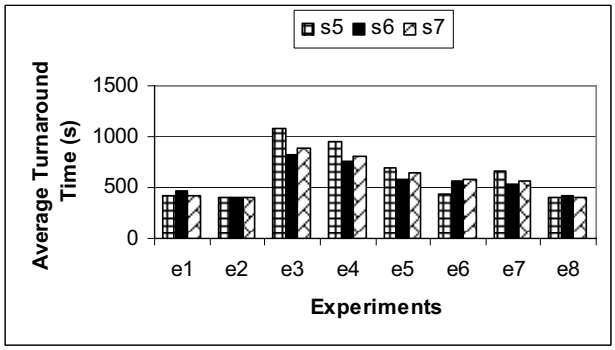

Figure 9. Turnaround time.

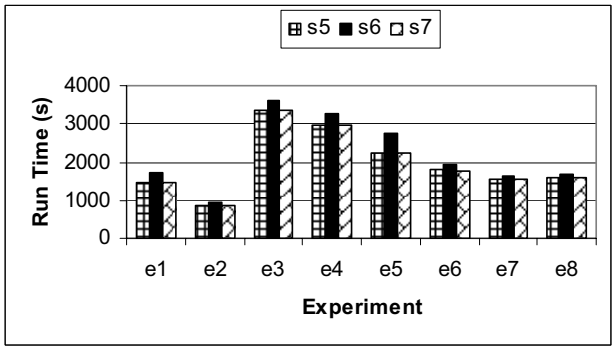

Figure 10. Primary application run time.

On the other hand, when the primary jobs are I/O-bound (e.g., e3 and e4), the guest jobs are able to run sooner while still having no impact on the primary jobs. Thus, the guest jobs should realize a decrease in their average turnaround time. Additionally, as the primary jobs' CPU utilization efficiency decreases, so does the guest jobs' average turnaround time. Since the guest jobs should not impact the average turnaround time of the primary jobs and since there is the possibility that the guest jobs' average turnaround time will decrease, we expect to see the average turnaround time of the jobs in each experiment in set 7 to either decrease or stay the same relative to the baseline average turnaround times (in set 5). Taking note of the efficiency measurements presented in Figure 8, we can see this behavior in Figure 9.

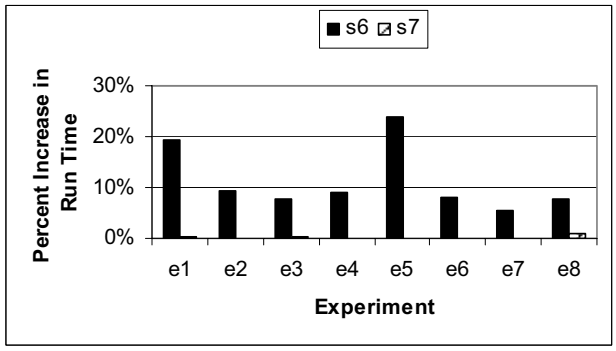

Figure 11. Impact on primary application quality of service.

The exception to the above observation is experiment e6. Even though s5e6 has an efficiency of 78%, we do not see the decrease in average turnaround time that we expect in either set 6 or set 7. Experiment e6 is different from the other experiments in that for set 5 (i.e., experiment s5e6), the jobs are executed so that the low-priority jobs are run first followed by the primary jobs. When the same jobs are run in sets 6 and 7 (i.e., experiments s6e6 and s7e6), the primary jobs will be started immediately (i.e., at time 0). Since the primary jobs have priority, the remaining jobs will all see an increase in their average turnaround times. In this case, the increase in the average turnaround time for the five low-priority jobs was greater than the decrease in the average turnaround time for the two primary jobs thus resulting in an overall increase in the average turnaround time of this set of jobs.

Figure 10 shows the run times for the primary applications in group C, while Figure 11 shows these data as a percent increase over the run times in set 5 (our quality of service measure). Notice that for the intelligent processor sharing system we do not see any significant increase in the run times of any of the primary jobs (less than 1% increase). Compare this to the low-priority job case, in which we see an increase of at least 5% in the run times of the primary jobs with some sets of primary jobs being affected by 20-25%.

6. Conclusions

The existing nice mechanisms in the 2.4 Linux kernel can be used to increase the throughput and efficiency of a cluster while also lowering the average response time of the queued jobs. It does so, however, at the expense of the quality of service of primary (high priority) jobs. The guest process mechanism in the intelligent processor sharing kernel (derived from the

Linger-Longer kernel) can maintain the quality of service of primary jobs while also increasing the throughput and efficiency of the cluster and lowering the average response time of queued jobs. When running low-priority or guest processes concurrently with the primary jobs, we saw that the gains for the throughput and efficiency of the cluster increased when the efficiency of the primary jobs decreased. Under the same conditions, the average turnaround time tended to decrease as the efficiency of the primary jobs decreased.

In our empirical results using production codes, we found that using the kernel's existing nice mechanism to start low-priority jobs concurrently with primary jobs enabled us to increase throughput by up to 29%, increase efficiency by up to 32% and decrease the average turnaround time by up to 20%. Unfortunately, this came at the expense of impacting the primary jobs' quality of service by increasing their run times anywhere from 5%-25%. Similarly, we found that by using the intelligent processor sharing kernel to run concurrent guest processes, we could increase throughput by up to 21%, increase efficiency by up to 33% and decrease the average turnaround time by up to 18%. Additionally, the quality of service of the primary jobs is maintained as run times are within 1% of their baselines.

While not yet providing a complete toolset for system managers, this work points explicitly to opportunities for improved utility of dedicated cluster systems by sharing underutilized resources in the cluster. The results presented here concentrate on the CPU as the performance critical resource. The original Linger-Longer system has since been expanded to consider both memory and I/O capability as potential performance critical resources. We anticipate that the expanded Linger-Longer system can be used to support generalized intelligent *resource* sharing in clusters as well.

Acknowledgements

The authors would like to thank the University of Missouri – St. Louis for access to its production Linux cluster; the providers of the applications: Simon Malcomber and Mark Beilstein of the UMSL Biology Department's Kellogg Lab (for MrBayes, PAUP), Clinton Greene of the UMSL Economics Department (for GAUSS), James Campbell of the UMSL College of Business Administration (for HAL1), and Eric Lenning of the National Weather Service (for help with WRF); and the original developer of the Linger-Longer system, Kyung Dong Ryu, for his kernel modifications. This work was partially supported by NSF grants CNS-0313203 and CCF-0427794.

References

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

[2] Ryu, K. D. and Hollingsworth, J. K., "Exploiting Fine-Grained Idle Periods in Networks of Workstations," *IEEE Trans. on Parallel and Distributed Systems*, 11(7). July 2000.

[3] Ryu, K. D. and Hollingsworth, J. K., "Unobtrusiveness and Efficiency in Idle Cycle Stealing for PC Grids," in *Proc. 18th Int'l Parallel and Distributed Processing Symposium*. April 2004.

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

[5] Zhou S., Zheng, X., Wang, J., Delisle, P., "Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," SPE, 23(12). 1993, pp. 1305-1336.

[6] Arpaci, R. H., et al., "The Interaction of Parallel and Sequential Workloads on a Network of Workstations," *SIGMETRICS*. May 1995, pp. 267-278.

[7] Ryu, K. D., et al., "Efficient Network and I/O Throttling for Fine-Grained Cycle Stealing," SC'01. Nov. 2001.

[8] "OpenPBS Administration Guide." http://www.openpbs.org/docs.html.

[9] "Sun Grid Engine Reference Guide." http://gridengine.sunsource.net/project/gridengine/documentation.html.

[10] Gropp, W. et al., MPICH Implementation. http://www-unix.mcs.anl.gov/mpi/mpich/

[11] GAUSS Mathematical and Statistical System. Aptech Systems, Inc. http://www.aptech.com/.

[12] Campbell, J. F., et al., "Solving Hub Arc Location Problems on a Cluster of Workstations," *Parallel Computing*, 29(5):555-574, May 2003.

[13] Petitet, A. et al., "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computer," Innovative Computing Laboratory, University of Tennessee. http://www.netlib.org/benchmark/hpl/.

[14] Huelsenbeck, J. and Ronquist, F., "MrBayes: Bayesian Inference of Phylogeny," http://morphbank.ebc.uu.se/mrbayes/info.php.

[15] "Weather Research and Forecast (WRF) Modeling System," National Center for Atmospheric Research, University Corporation for Atmospheric Research. http://www.wrf-model.org/ Welcome.html.

[16] Swafford, D., "PAUP*: Phylogenetic Analysis Using Parsimony (and Other Methods)," http://paup.csit.fsu.edu/index.html.

[17] Stiehr, G. "Using Fine-Grained Cycle Stealing to Improve Throughput, Efficiency and Response Time on a Dedicated Cluster while Maintaining Quality of Service," MS Thesis, CSE Dept., Washington Univ., St. Louis, Missouri, Dec. 2004.