Report Number: WUCSE-2004-52

2004-12-01

Using Fine-Grained Cycle Stealing to Improve Throughput, Efficiency and Response Time on a Dedicated Cluster while Maintaining Quality of Service

Gary Stiehr

For various reasons, a dedicated cluster is not always 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. Keeping in mind the needs of both the cluster's system administrators and its users, we would like to increase the throughput and efficiency of the cluster while maintaining or improving the average turnaround time of the jobs and the quality of service of the "primary" jobs originally scheduled on the cluster. To increase the throughput and efficiency of... **Read complete abstract on page 2.**

Using Fine-Grained Cycle Stealing to Improve Throughput, Efficiency and Response Time on a Dedicated Cluster while Maintaining Quality of Service

Gary Stiehr

Complete Abstract:

For various reasons, a dedicated cluster is not always 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. Keeping in mind the needs of both the cluster's system administrators and its users, we would like to increase the throughput and efficiency of the cluster while maintaining or improving the average turnaround time of the jobs and the quality of service of the "primary" jobs originally scheduled on the cluster. To increase the throughput and efficiency of the cluster, we schedule background jobs to run concurrently with the primary jobs. However, to achieve our goal of maintaining or improving the average turnaround time of the jobs and the quality of service of the primary jobs, we investigate two methods of prioritizing the CPU usage of the primary and background jobs. The first method uses the existing "nice" mechanism in the 2.4 Linux kernel to give background processes a lower priority than primary processes. The second method involves modifying the 2.4 Linux kernel's CPU scheduler to create a new guest process priority that prevents guest processes from running when primary processes are runnable. Our results come from empirical investigations using real production applications. Production runs using these applications are regularly performed in the dedicated cluster environment that we used for testing. Measurements of various statistics, such as wall time and CPU time, are taken directly from test runs that use these same production applications. This was helpful for comparison to results from models and synthetic applications. 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 running times increased 5-25%). On the other hand, we can use the guest process priority to get similar improvements in throughput, efficiency and average turnaround time while not significantly impacting the quality of service of the primary jobs (primary job running times changed less than 1%).

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2004-52

Using Fine-Grained Cycle Stealing to Improve Throughput, Efficiency and Response Time on a Dedicated Cluster while Maintaining Quality of Service, Masters Thesis, December 2004

Authors: Stiehr, Gary

September 13, 2004

Abstract: For various reasons, a dedicated cluster is not always 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. Keeping in mind the needs of both the cluster™s system administrators and its users, we would like to increase the throughput and efficiency of the cluster while maintaining or improving the average turnaround time of the jobs and the quality of service of the fiprimary?jobs originally scheduled on the cluster.
To increase the throughput and efficiency of the cluster, we schedule background jobs to run concurrently with the primary jobs. However, to achieve our goal of maintaining or improving the average turnaround time of the jobs and the quality of service of the primary jobs, we investigate two methods of prioritizing the CPU usage of the primary and background jobs. The first method uses the existing finice?mechanism in the 2.4 Linux kernel to give background processes a lower priority than primary processes. The second method involves modifying the 2.4 Linux kernel™s CPU scheduler to create a new guest process priority that prevents guest processes from running when primary processes are runnable.
Our results come from empirical investigations using real production applications. Production runs using these applications are regularly performed in the dedicated cluster environment that we used for testing. Measurements of various statistics, such as wall time and CPU time, are taken directly from test runs that use these same production applications. This was helpful for comparison to results from models and synthetic

Using Fine-Grained Cycle Stealing to Improve Throughput, Efficiency and Response Time on a Dedicated Cluster while Maintaining Quality of Service

Gary Stiehr

Gary Stiehr, “Using Fine-Grained Cycle Stealing to Improve Throughput, Efficiency and Response Time on a Dedicated Cluster while Maintaining Quality of Service,” Master’s Thesis, Department of Computer Science and Engineering, Technical Report WUCSE-2004-52, 2004.

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

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

USING FINE-GRAINED CYCLE STEALING TO IMPROVE THROUGHPUT, EFFICIENCY AND RESPONSE TIME ON A DEDICATED CLUSTER WHILE MAINTAINING QUALITY OF SERVICE

by

Gary Stiehr

Prepared under the direction of Professor R. D. Chamberlain

A thesis presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment
of the requirements of the degree of

MASTER OF SCIENCE

December 2004

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

USING FINE-GRAINED CYCLE STEALING TO IMPROVE THROUGHPUT, EFFICIENCY AND RESPONSE TIME ON A DEDICATED CLUSTER WHILE MAINTAINING QUALITY OF SERVICE

by Gary Stiehr

ADVISOR: Professor Roger D. Chamberlain

December 2004
Saint Louis, Missouri

For various reasons, a dedicated cluster is not always 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. Keeping in mind the needs of both the cluster's system administrators and its users, we would like to increase the throughput and efficiency of the cluster while maintaining or improving the average turnaround time of the jobs and the quality of service of the "primary" jobs originally scheduled on the cluster.

To increase the throughput and efficiency of the cluster, we schedule background jobs to run concurrently with the primary jobs. However, to achieve our goal of maintaining or improving the average turnaround time of the jobs and the quality of service of the primary jobs, we investigate two methods of prioritizing the CPU usage of the primary and background jobs. The first method uses the existing "nice" mechanism in the 2.4 Linux kernel to give background processes a lower priority than primary processes. The second method involves modifying the 2.4 Linux kernel's CPU scheduler to create a new guest process priority that prevents guest processes from running when primary processes are runnable.

Our results come from empirical investigations using real production applications. Production runs using these applications are regularly performed in the dedicated cluster environment that we used for testing. Measurements of various statistics, such as wall time and CPU time, are taken directly from test runs that use these same production applications. This was helpful for comparison to results from models and synthetic applications.

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 running times increased 5-25%). On the other hand, we

can use the guest process priority to get similar improvements in throughput, efficiency and average turnaround time while not significantly impacting the quality of service of the primary jobs (primary job running times changed less than 1%).

Contents

Tables

Figures

Glossary

cluster-level scheduler: the scheduler responsible for allocating processors to a job. This is distinguished from the local OS scheduler that actually performs time-sharing of the processor between runnable processes.

guest process: a process that is not enabled to run if there is a higher-priority non-guest process in the kernel's run queue. This means that a guest process may experience starvation if it is run along with a CPU-bound process. The meaning is similar when the term *guest* precedes other words such as *task* or *job*.

job: the smallest unit accepted by a cluster-level scheduler. The cluster-level scheduler allocates processors that it manages to a job. A job then starts any number of processes on these processors. If multiple processes are started, they do not necessarily have to be a part of a parallel application but this is often the case.

low-priority process: a process that has been assigned the lowest priority allowable by the Linux kernel's "nice" mechanism. The meaning is similar when the term *low-priority* precedes other words such as *task* or *job*.

primary process: a process that has high-priority usage of the system resources (processor, memory, I/O and network bandwidth, etc.) of the system on which it is running. The meaning is similar when the term *primary* precedes other words such as *task* or *job*.

Acknowledgements

The author would like to thank Kyung Dong Ryu for providing the 2.0 and 2.2 Linux kernel modifications used by his Linger-Longer cycle stealing system. Additionally, the author would like to acknowledge the University of Missouri – St. Louis for allowing the use of their Linux cluster to perform the experiments described in this paper. Thanks also go to a number of researchers at the University of Missouri – St. Louis for providing feedback and support in obtaining appropriate input files for and/or analyzing the behavior of various applications. This includes, but is not limited to, Simon Malcomber and Mark Beilstein of the Biology department's Kellogg Lab (for MrBayes, PAUP); Clinton Greene of the Economics department (for GAUSS); and James Campbell of the College of Business Administration (for HAL1). Additionally, thanks go to Eric Lenning of the National Weather Service for help with WRF.

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 when a relatively high-overhead network, such as Ethernet, connects its 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 would not, however, want to interfere with the primary task being performed on a given processor. To help avoid interference, we would 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).

This work is motivated by observing the behavior above on a sixty-four-processor production cluster used by researchers at the University of Missouri—St. Louis. The cluster is dedicated to running resource-intensive programs. Some of the applications run do not fully utilize the processors assigned to them. It is desirable in this production environment to increase the throughput and efficiency of the cluster while maintaining quality of service for the primary jobs.

We evaluate two methods for controlling the CPU usage of a process to see how they might help us achieve our objectives of higher throughput and efficiency while maintaining quality of service for the higher-priority process. One of these is the 2.4

Linux kernel's "nice" mechanism that allows us to specify the CPU priority of a given process. Secondly, we examine how a set of modifications to the kernel's CPU scheduler—originally designed for the Linger-Longer system to provide fine-grained cycle stealing in a network of workstations [19]—might help us achieve our goals. In this thesis, we are on the cluster level dealing with whole applications and so methods that can be used to prioritize threads within an application are not available to us.

The Linger-Longer system provides mechanisms and policies that support running guest processes along with host processes in order to exploit otherwise idle workstation resources in a non-dedicated network of workstations. The host processes are those processes, such as a web browsers and editors, which might be running on a workstation. The guest processes are processes that will be running concurrently with the host process on the workstation in order to consume any idle cycles. It is a goal of the Linger-Longer system to make sure that these guest processes do not significantly interfere with the host processes.

Our work applies some concepts from the Linger-Longer system to a dedicated cluster as opposed to a non-dedicated network of workstations. Users submit their jobs to the cluster and expect that the nodes allocated to their jobs will be available exclusively to their jobs. We call the processes running on the cluster as a part of these jobs primary processes (contrast this with Linger-Longer's host processes). As in the Linger-Longer system, we would like to run jobs in the background in order to use any spare CPU cycles. One of our goals is to make sure that these jobs do not significantly interfere with the primary jobs. When we use the Linux kernel's existing "nice" mechanism to help us control the CPU usage of these background jobs, we call the background jobs "low-priority" jobs. When we use the kernel extensions, we call them guest jobs.

The Linger-Longer work in [18], [19] and [20] presents results of experiments using benchmarks, models and simulations. In [19], the benchmarks were run as host processes and guest processes. The effect on the running time of the host processes was measured. In [18] and [20], the efficiency with which idles cycles were used by guest jobs as well as the change in the throughput of guest jobs was measured by running jobs in a simulated environment.

In our experiments, we use a number of real production applications that are normally run on a dedicated production cluster. Our testing environment is an actual subset of this production cluster. These production applications are used as both primary jobs and guest jobs. The effect on the running time of primary jobs is measured. We measure the change in the efficiency of the use of the entire cluster as opposed to just the efficiency of the use of the idle cycles. Similarly, we measure the change in the throughput of the entire cluster instead of just the change in the throughput of the guest jobs. Additionally, we measure the change in the average response time of the entire set of jobs.

The remainder of this chapter describes our testing environment and the applications we used in our experiments as well as gives an overview of the experiments. Chapter 2 discusses related work. Chapter 3 presents results obtained from running experiments using existing mechanisms in the unmodified 2.4 Linux kernel. Chapter 4 describes how these results change when the experiments are run again using a modified 2.4 kernel. Our conclusions are presented in Chapter 5.

1.1 Testing Environment

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 100 Mb/s 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 cluster is dedicated to running resource-intensive programs. Each researcher submits a job to the cluster's resource management software, which allocates nodes based on the resources the researcher requested. Thus if the researcher requested four nodes, then his/her job will be given exclusive[1] access to four dual processor nodes. This would include exclusive access to all of the memory (4GB total), all scratch space (on the four

[1] There are, of course, a small number of administrative utilities that run in the background, such as monitoring applications. The amount of resources used by these utilities is very minimal; however, queries by the administrator can occasionally cause these utilities to run even when a researcher's job has "exclusive" access. Additionally, the kernel will always need access to some of the resources.

hard disks), all processors (eight total), all four network connections[2], and other resources (e.g., buses).

The resource management software used on the cluster is OpenPBS [14], commonly referred to as PBS. A job is submitted to PBS by specifying a shell script that contains PBS-specific directives followed by the commands to be executed once processors are assigned to the job. The PBS submit node runs a `pbs_server` process that accepts job submissions and a `pbs_sched` process that decides when and where to schedule the jobs. The PBS submit node communicates with each worker node via a `pbs_mom` process running on each node. Once the scheduler determines where the job will run, the job script submitted earlier is transferred to that node. The `pbs_mom` process is responsible for starting execution of that script. The commands in that script now have complete access to the one or more nodes requested during job submission. Parallel processes may be started via the PBS API or via any other method, such as MPICH [6].

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 and connected by a 100 Mb/s Ethernet network. This means that each processor had exclusive access to the entire 1 GB of memory in the node as well as all scratch space and the full bandwidth of the various buses.

1.2 Application Descriptions

A number of applications were used during our research. As shown in Table 1-1, 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. Measurements showing each application's CPU-usage are presented in Section 3.1. With the exception of HPL, these applications have been used by the cluster's users to do real-world research. Hereafter, we will refer to these applications by the names listed in Table 1-1. Detailed

[2] Although the job will have exclusive access to the four network connections (one per node), the availability of the full bandwidth is not guaranteed. Although it is not likely, the network switch's backplane could be saturated by traffic from other nodes and thus, network availability would be affected.

descriptions of each application are given in the subsections below and summarized in Table 1-2 through Table1-7.

Table 1-1. 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

Table 1-2. GAUSS

Name	GAUSS
Description	GAUSS is a mathematical and statistical package that provides a matrix programming language. Programs written with GAUSS are run on the Linux cluster by Economics researchers.
Properties	This program is used in an embarrassingly parallel fashion in that the same program is run many times with different input. These different simulations are independent and can be run simultaneously on the cluster. Thus, linear speedup is achieved when running the different simulations in parallel. Each simulation is CPU-intensive. Other resource usage is minimal.

Table 1-3. HAL1

Name	HAL1
Description	HAL1 is a program used by Logistics researchers. It applies an intelligent enumeration algorithm to analyze all possible locations for hub arcs in a logistics network [4]. This program is run in parallel on the Linux cluster.
Properties	This program is CPU-intensive. Other resource usage is minimal. It is written using a master/slave model. If one slave process is slowed for some reason, the program as a whole will not be halted since the resulting excess work will be given out to the other slave processes.

Table 1-4. HPL

Name	HPL
Description	HPL is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers. It can thus be regarded as a portable as well as freely available implementation of the High Performance Computing Linpack Benchmark. We use HPL as a benchmark code. HPL is an MPI program.
Properties	This program can be CPU intensive or communication intensive depending on the size of the system being solved and the number of processors being used in the computation. With larger systems, more data is communicated between processes. Similarly, when the number of processors used to solve the problem is increased, so is the amount of communication between processors. If one process is stopped for some reason, the entire program will halt waiting for that process.

Table 1-5. MrBayes

Name	MrBayes
Description	MrBayes is a program for the Bayesian estimation of phylogeny. It is used on the cluster by Biology researchers. MrBayes is an MPI program.
Properties	This has moderate network bandwidth requirements and small memory requirements. It statically assigns independent portions of its work to different processors. If one process is stopped for some reason, the other processes will continue on their independent portions.

Table 1-6. PAUP

Name	PAUP
Description	PAUP (Phylogenetic Analysis Using Parsimony) is a package for inference of evolutionary trees.
Properties	This program is used in an embarrassingly parallel fashion in that the same program is run many times with different input. These different simulations are independent and can be run simultaneously on the cluster. Thus, linear speedup is achieved when running the different simulations in parallel. Each simulation is CPU-intensive. Other resource usage is minimal.

Table 1-7. WRF

Name	WRF
Description	WRF is designed to be a flexible, state-of-the-art atmospheric simulation system that is portable and efficient on available parallel computing platforms. WRF is suitable for use in a broad range of applications across scales ranging from meters to thousands of kilometers.
Properties	This program is very communication intensive. As a result, it does not fully utilize the CPU. As the number of processors used increases, the CPU utilization decreases. If one process is stopped for some reason, the entire program will halt waiting for that process.

1.2.1 GAUSS

GAUSS is a mathematical and statistical package that provides a matrix programming language [5]. Researchers in the Economics department at the University of Missouri – St. Louis use this package to write their analysis programs. One such analysis program, which was written with GAUSS, was used in production to perform actual research. The only change made to the input file was to reduce the runtime to an appropriate value. Many copies of this program using different datasets were run on the cluster in an embarrassingly parallel manner. The program is CPU-bound.

1.2.2 HAL1

HAL1 is a program written by researchers in the College of Business Administration at the University of Missouri – St. Louis. It applies an intelligent enumeration algorithm to analyze all possible locations for hub arcs in a logistics network [4]. This program is run in parallel on the Linux cluster using the MPICH [6] programming system.

HAL1 aims to optimize the location of hub arcs (the paths between hubs) instead of optimizing individual hub locations. Instead of enumerating through every possible combination of hub arcs between hub cities, however, HAL1 keeps track of the best set of hub arcs it has found so far and stops analyzing a given combination of hub cities once its cost exceeds the best (i.e., lowest) cost found so far.

The problem is solved in parallel using the master/slave paradigm with load balancing. One coordinator (master) process coordinates multiple worker (slave) processes. The workers are allocated a set of possible hub arc configurations through which to enumerate. At certain times, each worker process will report its best answer found so far to the coordinator process. The coordinator process will then push this information out to other workers so that they can eliminate more of the hub arc configurations that they are considering (based on this new information).

For the experiments using HAL1 in this thesis, we use input files that were used in production runs on the cluster. The only change was to scale the running time to an appropriate value. We ran the program using four processors for three worker processes and the coordinator process. This program is CPU-intensive. Other resource usage is

minimal. If one worker process is slowed for some reason, the program as a whole will not be halted since the resulting excess work will be given out to the other worker processes.

1.2.3 HPL

HPL is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers. It can thus be regarded as a portable as well as freely available implementation of the High Performance Computing Linpack Benchmark [16]. HPL was used as a benchmark code on the cluster. HPL is an MPI program.

This program has a number of tunable parameters. While all of the parameters affect performance in one way or the other, changing the matrix size and number of processors allows us to simulate a number of different conditions on the cluster. Depending on the matrix size, this program can have a small or very large memory footprint. As more processors are used, HPL communicates more and uses the CPU less. With smaller numbers of processors, it is CPU-intensive as well as communication-intensive. If one process is stopped for some reason, the entire program will halt waiting for that process. For the experiments in this thesis, we configured HPL in such a way that we would see the communication-intensive behavior.

1.2.4 MrBayes

MrBayes is a program for the Bayesian estimation of phylogeny [8]. It is used in production on the cluster by Biology researchers. Researchers have many datasets that can be processed independently. Furthermore, MrBayes allows each dataset to be processed in parallel by allocating each chain of the analysis to its own processor. Since each chain is independent and statically assigned to a processor, if one process is slowed, the others can continue. MrBayes is an MPI program.

The dataset used in the experiments in this thesis is one that was used to do production research. It used four chains and thus four processors were used by MrBayes to analyze this dataset.

1.2.5 WRF

WRF is designed to be a flexible, state-of-the-art atmospheric simulation system that is portable and efficient on available parallel computing platforms. WRF is suitable for use in a broad range of applications across scales ranging from meters to thousands of kilometers [26]. WRF is a parallel program and is very communication intensive. As a result, it does not fully utilize the CPU. As the number of processors used increases, the CPU utilization decreases. If one process is stopped for some reason, the entire program will halt waiting for that process.

1.2.6 PAUP

PAUP (Phylogenetic Analysis Using Parsimony) is a package for inference of evolutionary trees [23]. It is used in production by researchers in the Biology department at the University of Missouri – St. Louis. PAUP is a sequential program that is used with many different datasets in an embarrassingly parallel manner on the cluster. It is CPU-bound. For the experiments in this thesis, a dataset was used that, when analyzed by PAUP, exhibited the same runtime properties as those that were run in production on the cluster (i.e., its analysis was CPU-bound).

1.3 Overview of Experiments

Our results are based on seven distinct sets of experiments as summarized in Table 1-8. Each set consists of a number of individual experiments. As a convention used throughout the thesis, we will refer to experiment Y in set X as “experiment sXeY.” For example, experiment 3 of set 1 would be referred to as experiment s1e3.

Table 1-8. Summary of Experiments

Set	Purpose of Experiments
1	To measure the running time of each application when it is run by itself using the unmodified kernel. This establishes the baseline performance of each application.
2	To measure the effect of running a low-priority job concurrently with a primary job using the unmodified kernel. A set of one or more processes (a job) is assigned the kernel's lowest "nice" priority (low-priority processes) and run concurrently with a job consisting of processes with the default kernel nice priority (primary processes).
3	To measure the throughput, efficiency and turnaround time when running all applications such that there is only one application process per processor at any given time (i.e., we do not allow more than one job to be assigned to a given processor at a time) using the unmodified kernel. Additionally, the total running time of the primary jobs is measured as an indication of the primary jobs' quality of service. These are baseline measurements to be used for comparison with results from experiment sets 6 and 7.
4	The same as experiments in set 1 but this time with the modified kernel. These experiments are used to verify that the running times of the applications are not changed when run using the modified kernel.
5	The same as experiments in set 2 but this time with the modified kernel (and thus a "guest" job instead of a "low-priority" job).
6	To measure the throughput, efficiency and turnaround time when running a low-priority job concurrently with a primary job on the unmodified kernel. Additionally, the total running time of the primary jobs is measured as an indication of the primary jobs' quality of service.
7	The same as experiments in set 6 but this time with the modified kernel (and thus a "guest" job instead of a "low-priority" job). Additionally, the total running time of the primary jobs is measured as an indication of the primary jobs' quality of service.

Each experiment was allocated four nodes by the cluster-level scheduler regardless of the number of processes it was going to create. Each process, including individual parallel processes of a parallel application, were started by a wrapper script, `getstats`, that would start the process as one of its children and then collect resource usage statistics using the Linux kernel's `getrusage()` system call when the child process completed. Wall time was measured using a call to `gettimeofday()` immediately before the

process was started and immediately after it finished. All parallel applications used four processors and were run using the MPICH 1.2.5 implementation of the MPI specification [6]. Figure 1-1 shows how a parallel application, using MPICH, is started in order to gather the desired statistics. MPICH starts parallel processes on remote nodes using a specified program, such as `rsh` or `ssh`. We specified the use of a custom script, `fsh`, that modifies the command normally started via `rsh` by MPICH such that it uses the wrapper script, `getstats`, to start each parallel process as a child. Then, as mentioned above, `getstats` uses standard system calls to collect the desired statistics.

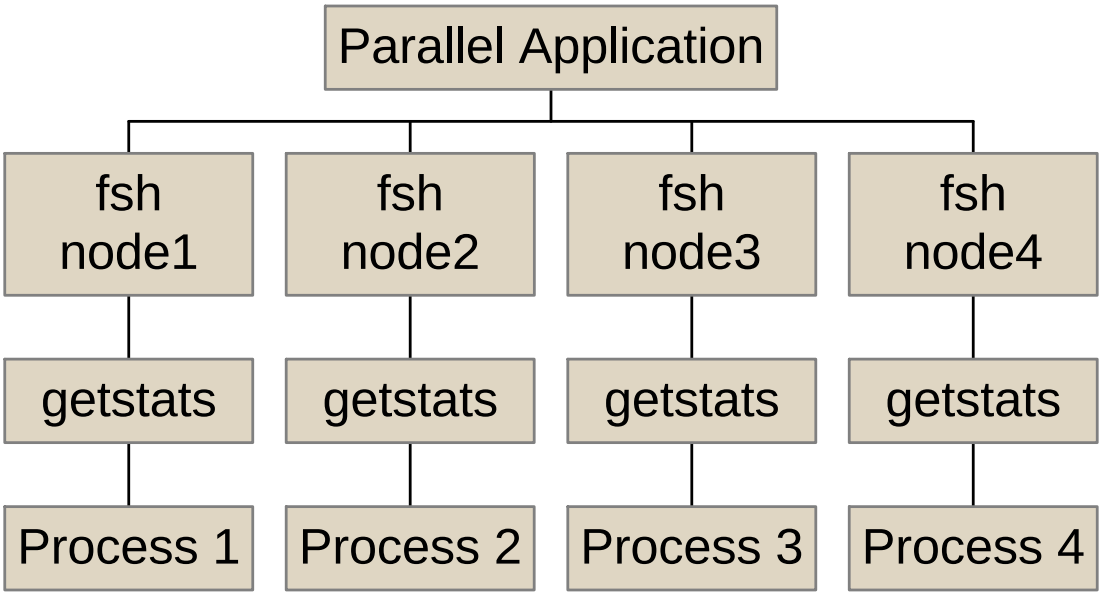

Figure 1-1. Gathering Statistics from the Applications

With the exception of HPL, all applications used in these experiments have been run on the production cluster in order to perform real-world research. HPL is benchmarking software and a dataset was designed for it that would cause it to behave as an I/O-bound application. For all other applications, datasets were chosen so that they would exhibit the properties observed by the actual researchers using them on the cluster but with smaller running times. Wherever possible, actual datasets from the researchers were used but scaled down to make the experiments run within an appropriate amount of time.

The seven sets of experiments described in Table 1-8 can be grouped into three groups as described in Table 1-9.

Table 1-9. Three Groups of Experiments

Group	Sets in Group	Purpose of Group
A	1, 4	To collect baseline statistics for primary jobs.
B	2, 5	To measure the impact on the quality of service of primary jobs when other jobs are run concurrently.
C	3, 6, 7	To measure the change in average turnaround time of a set of jobs; measure the change in throughput and efficiency of the nodes used by these jobs; and measure the impact on the quality of service of primary jobs.

1.4 Contributions of Thesis

The work done to complete this thesis has resulted in a number of contributions. Before any work could be done, it was necessary to develop scripts to gather and report throughput, efficiency, response time and quality of service measurements. It was then necessary to gather a variety of production applications to measure. The Linger-Longer kernel extensions had to be ported to the 2.4 Linux kernel. Additional scripts were written to collect, manage and average 156 experimental runs (fifty-two experiments run three times each). Over 4300 lines of shell scripts and C code was written to support the work done for this thesis. The results of these experiments are analyzed in this thesis.

The scripts written to report the performance data from the applications had to consider both sequential and parallel processes. Parallel applications were run with MPICH. A script was written to be called from MPICH in place of rsh or ssh. This script modifies the command line that MPICH uses to create a parallel process on a remote host. These modifications force the parallel process to be started by yet another script written for this thesis work. That script creates the parallel process as a child process and reports statistics for that process once it exits.

The applications chosen were mostly production applications. Input files for these applications were gathered and tuned so that run times would be appropriate and so that the applications would still exhibit the runtime characteristics that researchers saw when using them on the production cluster of which our test environment was a part. The

applications represented a variety of fields, such as biology, economics, logistics and weather modeling.

As provided, the Linger-Longer kernel extensions were for the 2.0.x and 2.2.x Linux kernels. Our test systems used the 2.4.x Linux kernel. Due to changes in the way priorities are calculated in the 2.4 kernels versus the 2.2 and 2.0 kernels, these kernel extensions had to be ported to the 2.4 Linux kernel.

With all of the above preparations, scripts were written to do the actual experiments. These scripts had to ensure that certain applications were started on certain processors during the correct time frames. Timing information from these experiments had to be reported in addition to the statistics reported by the other scripts for each application. Each of the fifty-two experiments was run three times to detect any major variation in the measurements. Scripts had to be written to calculate the averages of the many measurements recorded. Additionally, scripts were needed to extract statistics from the hundreds of output files and calculate from them statistics for the cluster as a whole.

An analysis of the results of these empirical experiments is presented in this thesis. It serves to strengthen previous results which were obtained through simulations and modeling using benchmarking applications. Additionally, the analysis shows the viability of extending the Linger-Longer work to a dedicated cluster. It provides support for new methods to improve the throughput, efficiency and response time in a dedicated cluster while maintaining quality of service.

2 Background and Related Work

A goal of our research 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 improve or maintain the average turnaround time of the jobs and the quality of service of the primary jobs. Condor [12], LSF [27], NOW [1] and Linger-Longer [17], [18], [19], [20] 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.

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). In our dedicated cluster environment, CPU activity from the primary jobs is ideally always present. Thus the mechanisms used above would not be sufficient in helping us to exploit idle time while minimizing the impact on primary jobs.

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. Although this method was designed with the workload characteristics of a network of non-dedicated workstations in mind, it is also well suited for our dedicated cluster environment because it would allow us to keep the guest jobs running concurrently with the primary jobs yet limit the impact on the primary jobs.

Linux is widely used as the operating system for commodity clusters [21]. However, there are times when extensions or modifications of the Linux kernel are considered in order to improve the efficiency of the cluster [3]. We investigate how some modifications to the Linux kernel can help us to run guest jobs concurrently with primary jobs without significantly impacting the primary jobs. Using extensions to the Linux

kernel that support real-time scheduling, He et al. [7] developed models of idle time in heterogeneous clusters that can be used to run aperiodic jobs without interfering with existing periodic jobs. This model uses specific properties of the periodic jobs that will be present in the system, such as their deadline and period. Since the primary jobs in our system are not periodic, this information is not available and the model cannot be directly applied to our scenario.

Our research does not propose modifications to the cluster-level scheduler. The kernel modifications should be made on each of the worker nodes comprising the dedicated cluster. Any cluster-level scheduler, such as those found in Condor, LSF, PBS [14] and Sun GridEngine [22], 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. Also, it is still applicable to utilize techniques to tune the performance of cluster-level schedulers when appropriate, such as choosing suitable job sizes and job runtimes to establish an effective prime/non-prime time scheduling policy as studied in [13]. Another method called “interstitial computing” [9] schedules a large number of small “interstitial jobs” to fill unused CPU cycles arising from scenarios where each queued job requires more processors than are available (and as a result they leave the available processors idle). This method does not schedule the interstitial jobs on the same processor as a native job; each interstitial job receives a dedicated processor. As a result, the cluster-level scheduler for the primary and/or guest jobs could certainly utilize interstitial computing if desired.

Our work does not rely upon any special characteristics of the cluster-level scheduler. It is assumed that a scheduler would be chosen to allocate processors as efficiently as possible to the primary jobs. We are not proposing a new cluster-level scheduler but rather we are proposing that a second scheduler might be used to schedule guest jobs to use available idle time on already allocated processors. Indeed, the same scheduler could be used as long as it recognizes which guest jobs would be complimentary to the primary jobs already running on a set of nodes. The idle time on a processor could simply be viewed as a slower processor that is available. Additionally, once the processors have been allocated to primary and/or guest jobs, the proposed kernel changes do not preclude the use of other methods sometimes used to synchronize the scheduling of individual

processes within the jobs (if needed), such as coscheduling [15], methods used by the Stealth distributed scheduler [10] or the Periodic Boost scheme used by Banerjee et al. [2].

Once primary jobs are allocated to CPUs in the cluster, the distribution of idle resources available to guest jobs can resemble a non-dedicated distributed environment. As such, one could use metrics such as the "task ratio," used by Leutenegger et al. [11] to determine the feasibility of scheduling parallel jobs on non-dedicated workstations, in order to determine the feasibility of scheduling guest jobs on the available resources. It may also be beneficial for parallel guest jobs to make use of parallel programming environments that consider the potentially variable availability of distributed resources. Dyn-MPI [25] is one such programming environment. It automatically redistributes the location of data used by parallel processes as the availability of resources changes.

The Linger-Longer work also recognized that in addition to a guest job's CPU usage, the memory [19], network and I/O [17] usage of a guest job could have a significant impact on the primary jobs running concurrently. Our research does not address these other issues, such as memory, I/O and network usage by the guest jobs, which may impact the primary job. Our experiments were designed so that the only resource contention would be for the CPU. We verified the impact on primary jobs when running guest jobs concurrently using the Linux kernel's existing "nice" mechanism versus the Linger-Longer approach.

The Sharc system [24] provides resource management mechanisms to control the CPU and network bandwidth usage of applications that are run concurrently across a shared cluster. These mechanisms improve support for application isolation and performance guarantees for applications running across a shared cluster. The Sharc system relies on single node resource management mechanisms, such as reservations or shares. It then extends the benefits of these mechanisms to clustered environments. Testing and evaluation of the Sharc system was done using extensions to the Linux kernel that provide QoS schedulers that satisfy Sharc's dependencies; although it should be noted that Sharc itself does not include any kernel modifications. The scope of the work on the Sharc system is wider than that of this thesis. In fact, the Linux kernel extensions presented in this thesis and in the Linger-Longer work [17],[19],[20] may provide

alternative resource management mechanisms that could satisfy the kernel requirements of the Sharc system.

We used a mixture of parallel and sequential applications, normally used by researchers in a production cluster environment, to verify the results of the Linger-Longer work in our dedicated cluster environment instead of using synthetic applications. Our focus was on increasing the throughput and efficiency of the cluster while maintaining or improving the turnaround time of the jobs and the quality of service of the primary jobs. We did not pay particular attention to the possibility of starvation of a guest job. In fact, starvation of a guest job is a real possibility. In order to ensure the forward progress of the guest jobs as well, it may be necessary to implement process migration. Some approaches to handling process migration in a cycle stealing environment can be found in [19].

3 Existing Priority Resource Management in the Kernel

3.1 Establishing Baseline Measurements

Baseline measurements show us how each application behaves when there are no other applications running concurrently to interfere with them. Our first set of experiments, set 1, consists of six experiments—one for each application. This set of experiments comes from group A in Table 1-9. The objective of these experiments is to determine the normal running time (i.e., wall time) of these applications on our 1 GHz dual Pentium III test systems as well as their efficiency and total CPU usage. For each sequential application, only one process was run at a time ensuring that each process had unimpeded access to its critical resource. Each parallel application was run alone using a total of four processors across four nodes. The results for each application are shown in Table 3-1.

Table 3-1. Baseline Application Running Times

Experiment	Application	Run 1			Run 2			Run 3		
		CPU (s)	Wall (s)	Eff (%)	CPU (s)	Wall (s)	Eff (%)	CPU (s)	Wall (s)	Eff (%)
s1e1	GAUSS	179	180	99	179	180	99	179	180	99
s1e2	HAL1	662	167	99	661	167	100	662	167	99
s1e3	HPL	844	357	59	844	357	59	845	357	59
s1e4	MrBayes	234	92	64	239	92	65	235	90	66
s1e5	WRF	863	393	55	863	390	55	866	392	55
s1e6	PAUP	212	214	99	211	211	100	212	213	99

Each experiment was run three times. During each run, three statistics were reported: CPU time, wall time and efficiency. The CPU time reported in Table 3-1 is the sum of the CPU time used by all processes in the application. The wall time is the time elapsed between when the first process in the application started and when the last process exited.

The efficiency is the sum of the CPU times of all processes in the application divided by the sum of the wall times of all processes in the application. Efficiency was measured in this way so that we could see how efficiently the application used the processors it was allocated. This is neither a measure of the application's parallel efficiency nor a measure of how efficiently the cluster was used as a whole.

The average CPU time, wall time and efficiency for the three runs shown in Table 3-1 are presented in Figure 3-1, Figure 3-2 and Figure 3-3, respectively. Remember that the CPU time shown in these graphs is the total CPU usage across all parallel processes for each individual application and it will generally be higher than the wall time for applications with degrees of parallelism greater than 1 (see Table 1-1). Figure 3-4, Figure 3-5 and Figure 3-6 show the standard deviation of the CPU time, wall time and efficiency, respectively[3]. We can see that the standard deviation is small, indicating low variability in the execution performance of these applications. We take advantage of this result by reporting only the mean value of the three runs of each experiment throughout the remainder of the thesis.

This data, summarized in Table 3-2, helps us to confirm the claims in Table 1-1 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 (in the case of these experiments, the properties of the cluster interconnect network slowed these applications down).

Table 3-2. Averages (with Standard Deviations) for Three Baseline Runs

		CPU Time (s)		Wall Time (s)		Efficiency (%)	
Experiment	Application	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.
s1e1	GAUSS	179	0	180	0	99	0
s1e2	HAL1	661	0	167	0	99	0
s1e3	HPL	844	0	357	0	59	0
s1e4	MrBayes	236	3	91	1	65	1
s1e5	WRF	864	2	392	1	55	0
s1e6	PAUP	212	1	212	1	100	0

[3] Figure 3-4, Figure 3-5 and Figure 3-6 are drawn on separate graphs at 1/20th the scale of the graphs in Figure 3-1, Figure 3-2 and Figure 3-3 because the standard deviations were too small to be seen as error bars with the averages.

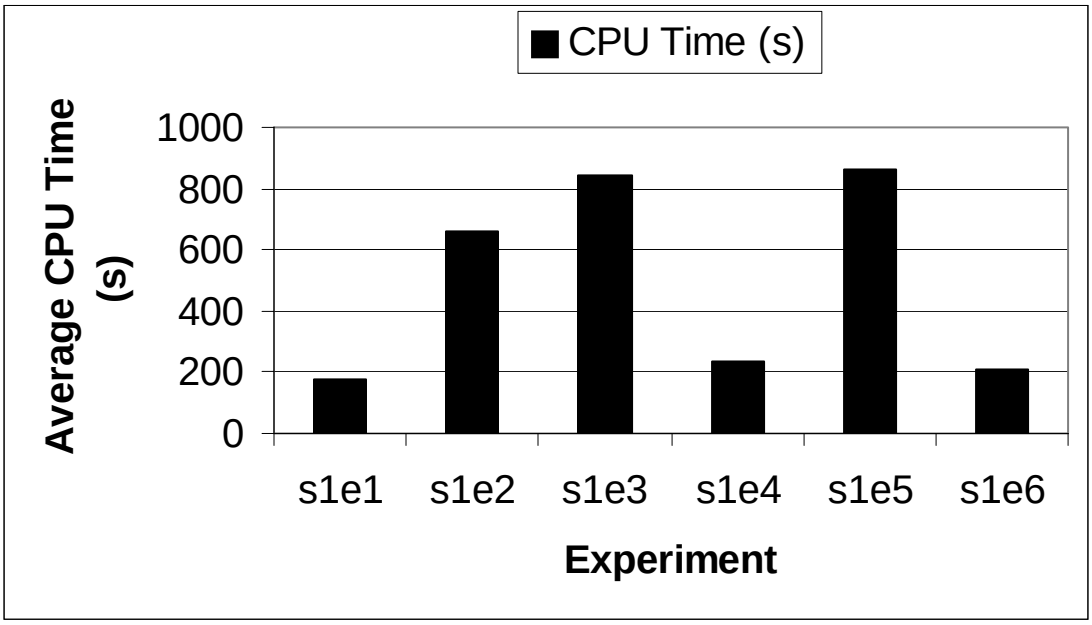

Figure 3-1. Average CPU Time for Three Baseline Runs

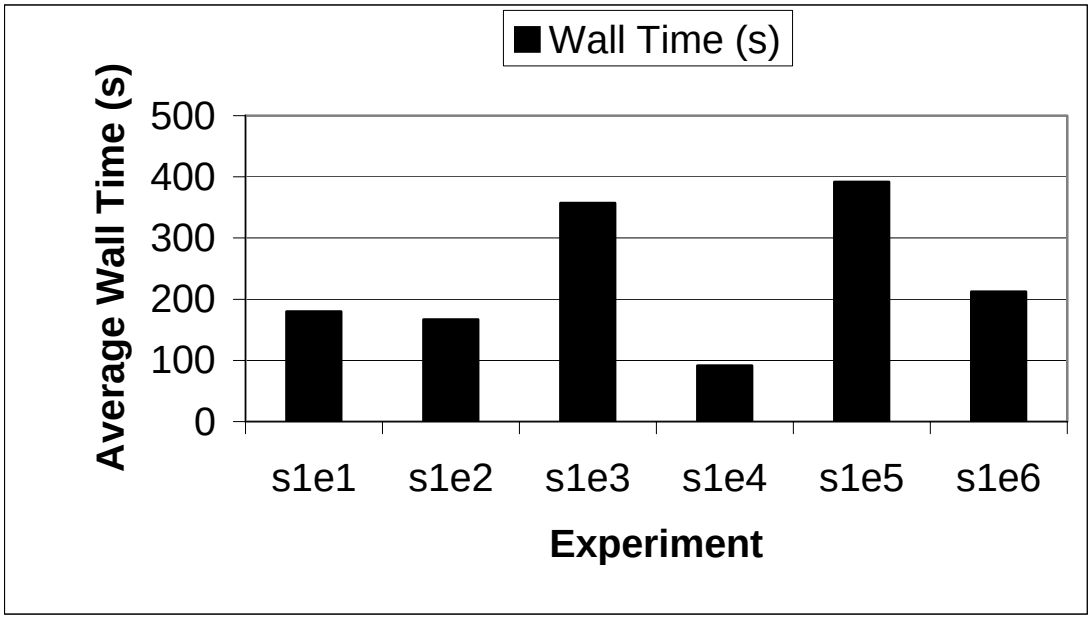

Figure 3-2. Average Wall Time for Three Baseline Runs

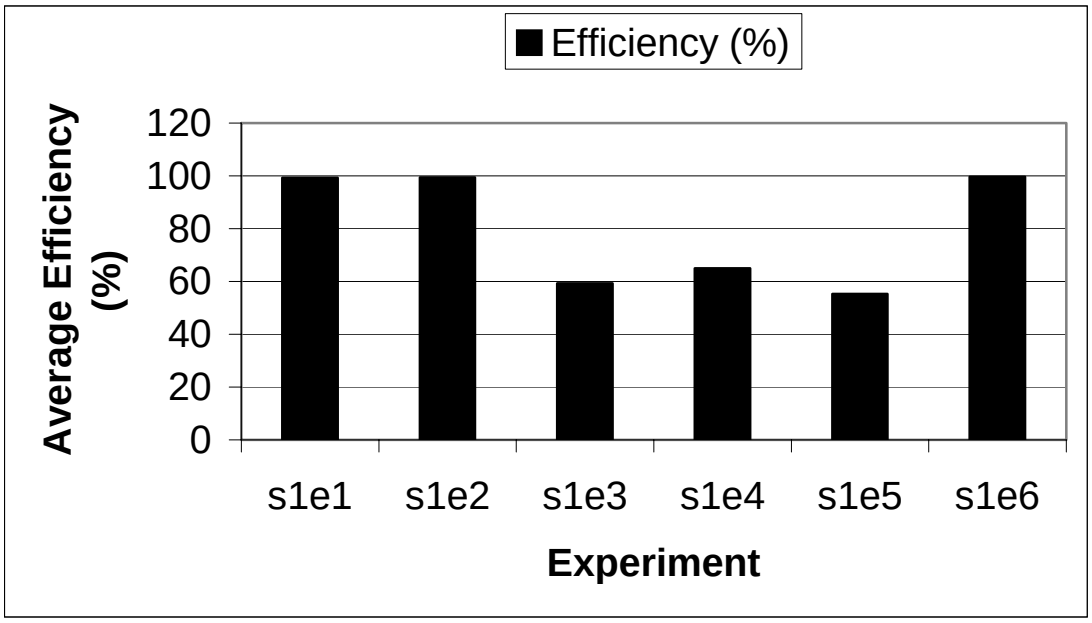

Figure 3-3. Average Efficiency for Three Baseline Runs

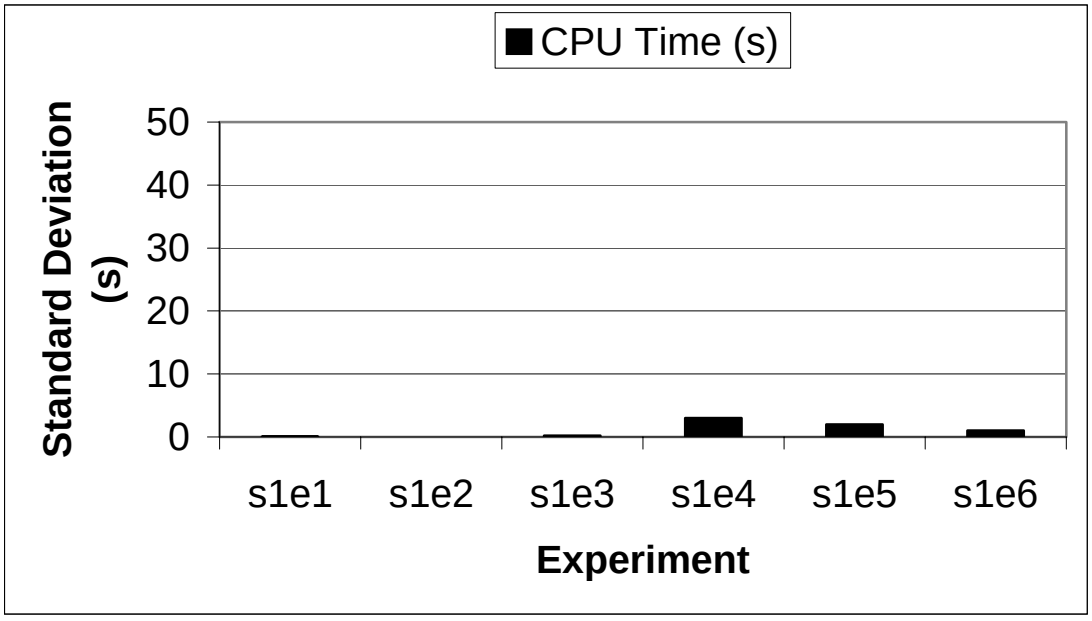

Figure 3-4. Standard Deviation of CPU Time

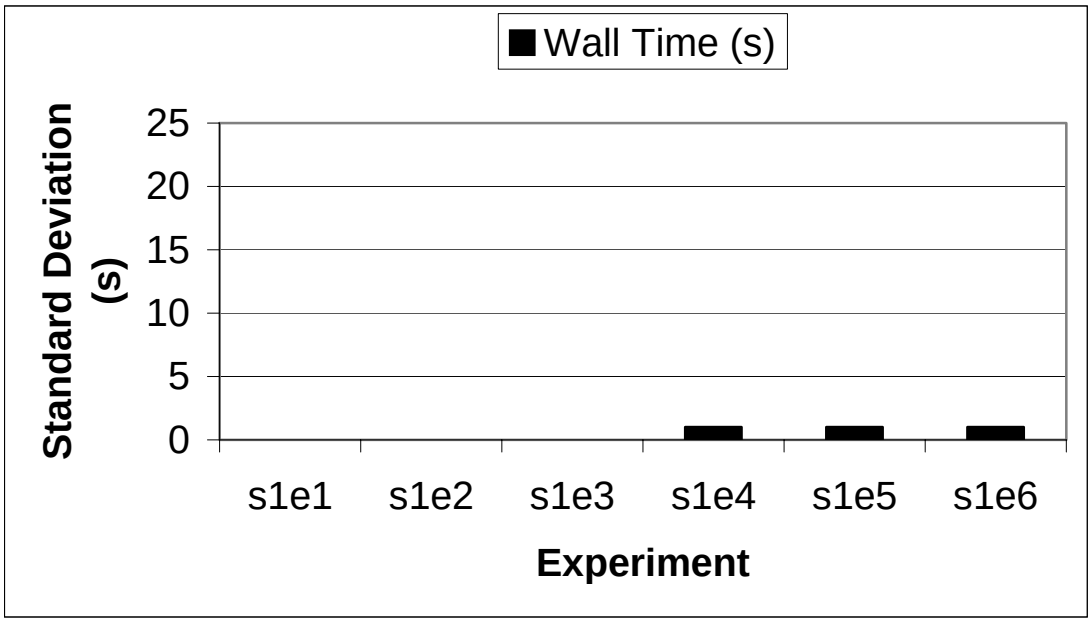

Figure 3-5. Standard Deviation of Wall Time

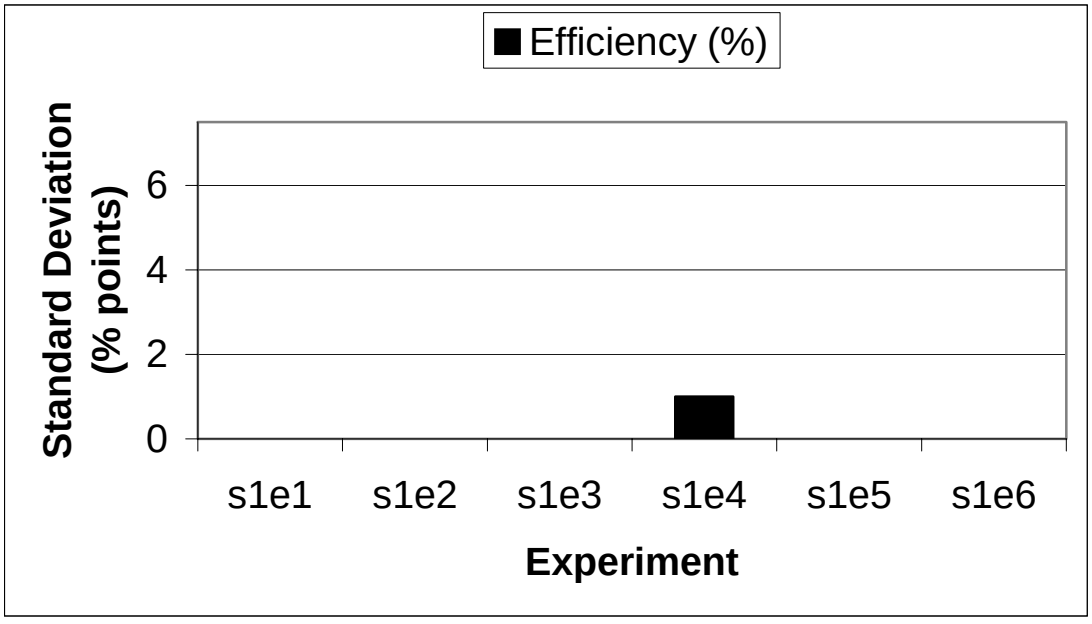

Figure 3-6. Standard Deviation of Efficiency

3.2 Evaluating Existing Priority Resource Management Facilities

The 2.4 SMP Linux kernel provides users with the ability to adjust the priority with which the kernel schedules processes for CPU time. Our next set of experiments focuses on the effect that low-priority jobs have on primary jobs that are running concurrently on the same node.

3.2.1 Effect of Low-Priority Jobs on Running Times of Primary Jobs

The experiments in set 2 were intended to determine how some of the applications, run as primary processes, are affected when low-priority processes are run concurrently. This set of experiments comes from group B in Table 1-9. The list below shows the sequence of applications run for different experiments; 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"). 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 where noted. The experiments in set 2 are listed below.

- s2e1:
  - GAUSS
  - [GAUSS]
- s2e2:
  - MrBayes
  - ([GAUSS], [GAUSS], [GAUSS], [GAUSS])
- s2e3:
  - HAL1
  - [PAUP]
- s2e4:
  - WRF
  - [PAUP], [PAUP]

- s2e5:
  - WRF
  - [HAL1]
- s2e6:
  - HAL1
  - [PAUP] (specifically run concurrently with one HAL1 slave process)
- s2e7:
  - HAL1
  - [PAUP], [PAUP], [PAUP], [PAUP] (specifically run concurrently with all HAL1 processes)
- s2e8:
  - HAL1
  - [PAUP], [PAUP], [PAUP] (specifically run concurrently with all three HAL1 slave processes)

Table 3-3 shows the mean CPU time, wall time and efficiency of each of these applications when running a low-priority process concurrently with them.

Table 3-3. CPU Time, Wall Time and Efficiency of Primary Jobs.

Experiment	Primary Application	CPU (secs)	Wall (secs)	Efficiency (%)
s2e1	GAUSS	178.72	208.91	85.55
s2e2	MrBayes	241.32	100.30	60.34
s2e3	HAL1	676.56	224.22	75.70
s2e4	WRF	885.04	409.09	54.14
s2e5	WRF	765.31	426.92	44.86
s2e6	HAL1	642.14	182.31	88.27
s2e7	HAL1	412.76	269.42	38.36
s2e8	HAL1	587.96	226.87	64.92

The CPU time reported in Table 3-3 is the sum of the CPU time used by all processes in the primary application. The wall time is the time elapsed between when the primary application's first process started and when its last process exited. The efficiency is the

sum of the CPU times of all processes in the primary application divided by the sum of the wall times of all processes in the primary application. Efficiency was measured in this way so that we could see how efficiently the primary application used the processors it was allocated. This is neither a measure of the application's parallel efficiency nor a measure of how efficiently the cluster was used as a whole.

Figure 3-7 shows how the running time of the primary applications have been affected by running a low-priority job concurrently. Figure 3-8 shows the percent change from the baseline running times of the primary applications.

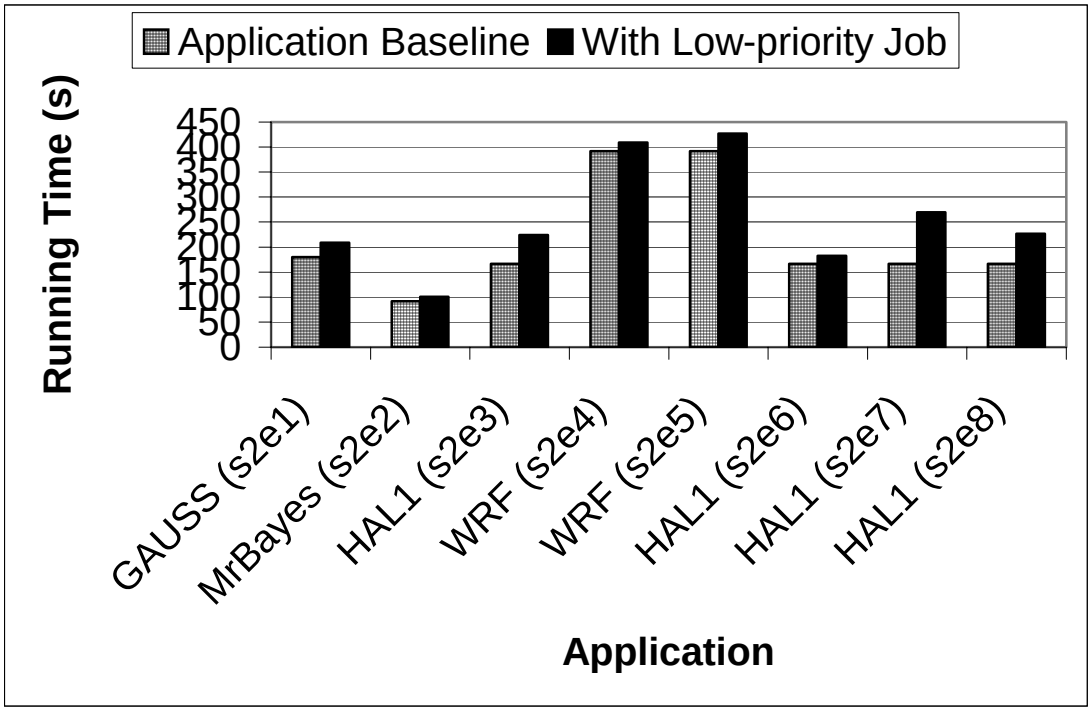

Figure 3-7. Baseline Running Times vs. Running Times with a Low-Priority Job

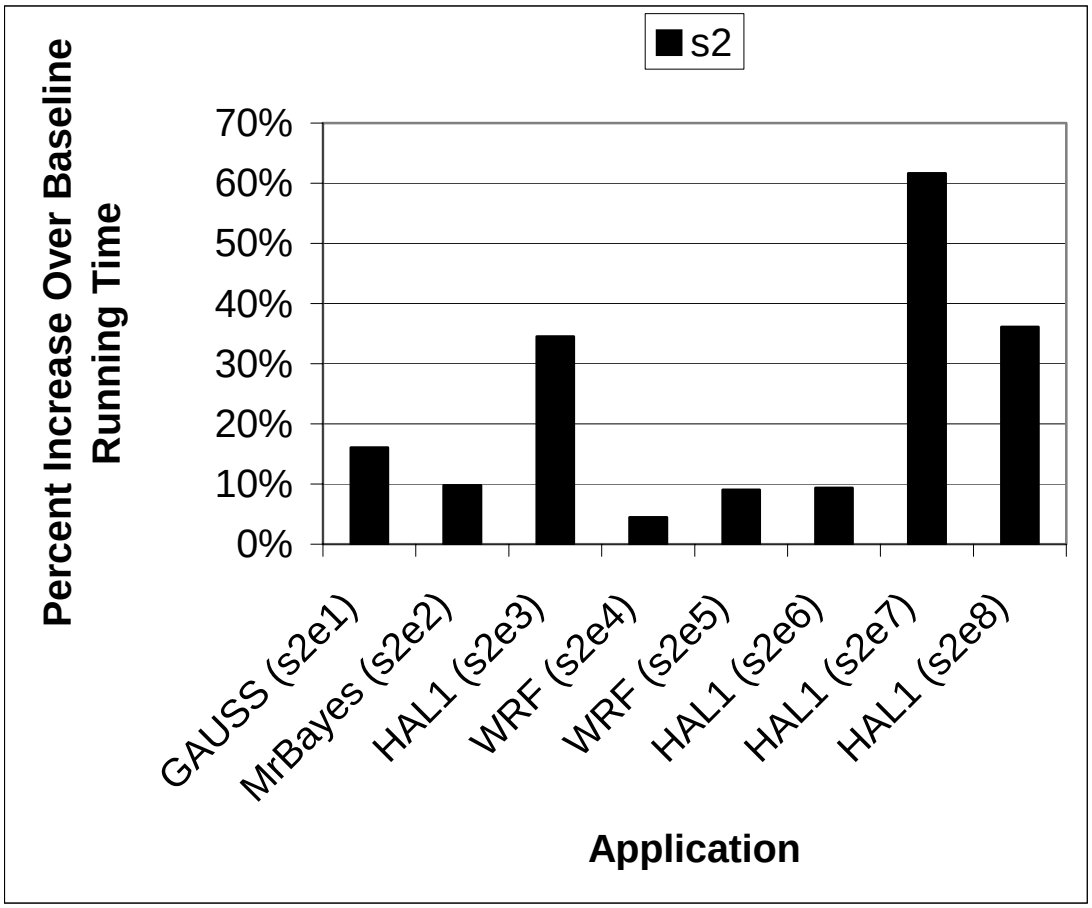

Figure 3-8. Percent Increase Over Baseline Running Times

The data presented above in Table 3-3, Figure 3-7 and Figure 3-8 show that running a low-priority job concurrently with a primary job 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 just presented. The two I/O-bound jobs, MrBayes and WRF, have the lowest percent increase in running time.

It is important to point out that the impact on the application is not always just an increase in wall clock time. Examples of this are experiments s2e5 (WRF) and s2e7/s2e8 (HAL1). Notice that the CPU time used by the applications during these particular experiments has actually decreased from baseline when the low-priority job was run

concurrently with them. These values were not anomalies; they were consistent across the three runs performed for each experiment. It is reasonable for this to happen if the application uses an algorithm that varies the work it does based on certain factors. For example, a probabilistic application may find different answers depending on which branches it chooses to examine. Changing the characteristics of the processor on which such algorithms execute can be just the cause needed to set the algorithm in a new direction.

The authors have particular knowledge of the HAL1 application. HAL1 uses the master/slave paradigm with load balancing along with an adaptive enumeration algorithm to generate its results [4]. A slave process' workload can change depending on the results found by the other slave processes so far. Therefore, if a low-priority process is taking CPU time away from one or more of the slave processes, this can affect the workload of the other slave processes. Additionally, the slave processes communicate their results to each other through the master process. Thus if there are fewer communications for the master process to manage as a result of less work being done by some of the slave processes, the master process may use less CPU time also.

3.2.2 Throughput, Efficiency and Response Time

The next experiments are intended to measure some important properties of the cluster's resource management environment. The throughput of our system is measured by looking at the number of jobs completed in a certain amount of time. The efficiency of the use of the cluster's CPU resources is measured by noting the percentage of the CPU utilization over a particular period of time. The turnaround time of a particular job is measured by noting the difference between when the job is submitted to the resource manager and when it completes execution.

A first set of experiments (set 3 in Table 1-8, group C in Table 1-9) was run to obtain measurements when jobs had to follow each other sequentially. That is, only one job at a time had access to a given set of processors. These are baseline measurements that we can use to compare with the cases where we use the kernel's existing "nice" mechanism to run more than one job per set of processors. Additionally, we can compare the

baselines to the case where we use the “guest process” kernel modifications to run more than one job per set of processors.

Running One Job per Set of Processors (Set 3, Group C)

The list below shows the sequence of applications run for different experiments; 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”). Figure 3-9 illustrates the timeline for the application sequence for s3e1 shown in the list below. Experiments s3e1 and s3e2 specifically start only CPU-bound primary jobs. Experiments s3e3 and s3e4 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 in Section 1.3, 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. We used the baseline measurements from Table 3-1 to guide us to these rough approximations. 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 are:

- s3e1: HAL1, (PAUP, PAUP, GAUSS, GAUSS), [HAL1], [HAL1]
- s3e2: (PAUP, PAUP, PAUP, PAUP), [HAL1], ([GAUSS], [GAUSS], [GAUSS], [GAUSS])
- s3e3: MrBayes, WRF, HPL, [HAL1], ([PAUP], [PAUP], [GAUSS], [GAUSS]), ([PAUP], [PAUP], [GAUSS], [GAUSS])
- s3e4: HPL, [HAL1], WRF, ([GAUSS], [PAUP], [PAUP], [GAUSS]), [HAL1]
- s3e5: WRF, ([GAUSS], [GAUSS], [GAUSS], [GAUSS]), ([PAUP], [PAUP], [PAUP], [PAUP]), HAL1
- s3e6: [HAL1], ([PAUP] , [PAUP] , [PAUP] , [PAUP]), MrBayes, HPL
- s3e7: WRF, [HAL1], ([GAUSS],[GAUSS],[GAUSS],[GAUSS])
- s3e8: MrBayes, MrBayes, (PAUP, PAUP,PAUP,PAUP), [HAL1],[HAL1]

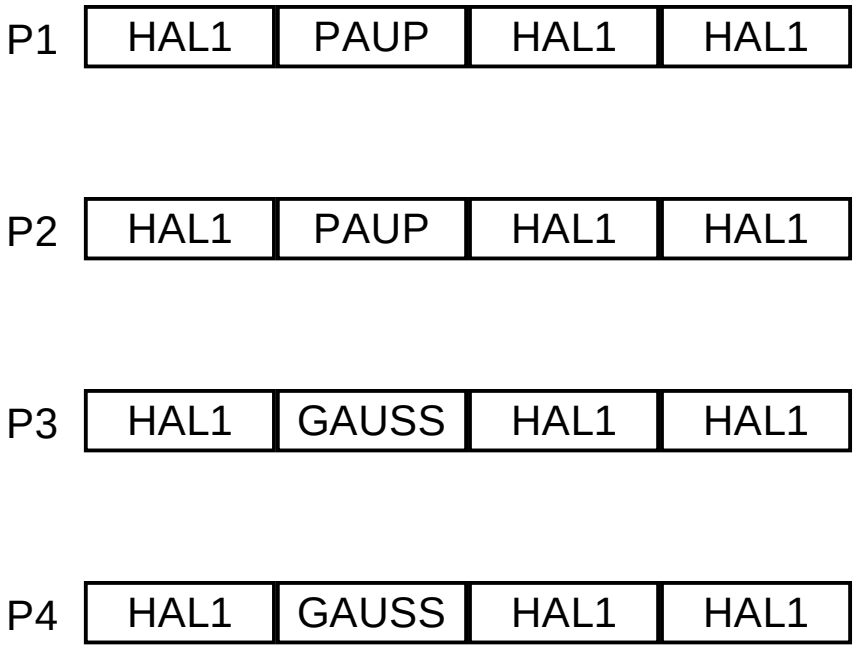

Figure 3-9. Application Timeline for s3e1

We measured the throughput of the cluster resource management system 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 here is a calculation of how efficiently the set of jobs in a given experiment used the set of processors assigned to them. 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 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 amount of 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 time needed for all primary jobs to complete. The throughput, efficiency, turnaround time and primary job running time measurements for the experiments in set 3 are shown in Table 3-4.

Table 3-4. Throughput, Efficiency, Response Time and Primary Job Running Time for One Job Case

Experiment	Number of Jobs Run	Throughput (jobs/hr)	Efficiency (%)	Avg. Turnaround Time (s)	Primary Job Running Time (s)
s3e1	7	35.45	97.21	420.61	1447.69
s3e2	9	57.71	99.05	395.62	852.19
s3e3	11	30.18	72.86	1084.11	3350.53
s3e4	8	22.24	73.61	952.79	2987.98
s3e5	10	37.74	80.93	685.83	2225.35
s3e6	7	30.36	78.09	437.99	1789.21
s3e7	6	29.36	76.06	659.27	1561.72
s3e8	8	39.56	90.69	403.33	1573.92

Running Multiple Jobs per Set of Processors

The experiments in set 6 (group C) are identical to those in set 3; the difference is in how they were run. For set 6, low-priority jobs were 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.

The list below shows the sequence of applications run for different experiments; 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”). Figure 3-10 illustrates the timeline for the application sequence for s6e1 shown in the list below. Experiments s6e1 and s6e2 specifically start only CPU-bound primary jobs. Experiments s6e3 and s6e4 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 3 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:

- s6e1:
  - HAL1, (PAUP, PAUP, GAUSS, GAUSS)
  - [HAL1], [HAL1]
- s6e2:
  - (PAUP, PAUP, PAUP, PAUP)
  - [HAL1], ([GAUSS], [GAUSS], [GAUSS], [GAUSS])
- s6e3:
  - MrBayes, WRF, HPL
  - [HAL1], ([PAUP], [PAUP], [GAUSS], [GAUSS]), ([PAUP], [PAUP], [GAUSS], [GAUSS])
- s6e4:
  - HPL, WRF
  - [HAL1], ([GAUSS], [PAUP], [PAUP], [GAUSS]), [HAL1]
- s6e5:
  - WRF, HAL1
  - ([GAUSS], [GAUSS], [GAUSS], [GAUSS]), ([PAUP], [PAUP], [PAUP], [PAUP])
- s6e6:
  - MrBayes, HPL
  - [HAL1], ([PAUP] , [PAUP] , [PAUP] , [PAUP])
- s6e7:
  - WRF
  - [HAL1], ([GAUSS], [GAUSS], [GAUSS], [GAUSS])
- s6e8:
  - MrBayes, MrBayes, (PAUP, PAUP, PAUP, PAUP)
  - [HAL1], [HAL1]

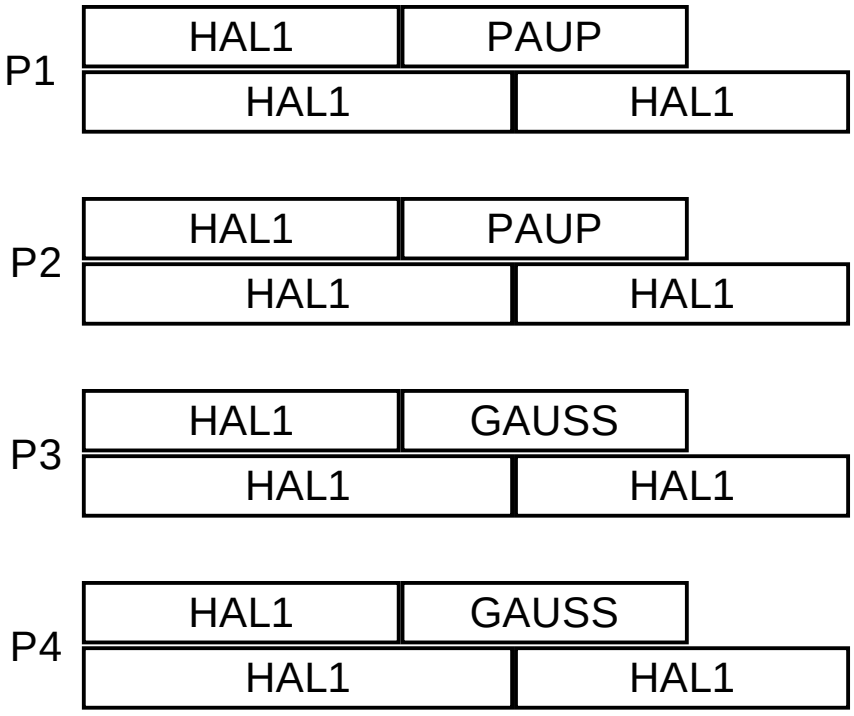

Figure 3-10. Application Timeline for s6e1

Now that we are allowing low-priority jobs to run along with the primary jobs, we would like to see what effect this has on throughput, efficiency, average turnaround times and quality of service. We measured the throughput of the cluster resource management system 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 here is a calculation of how efficiently the set of jobs in a given experiment used the set of processors assigned to them. 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 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 amount of 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 time needed for all primary jobs to complete. Our measurements for set 6 are shown in Table 3-5.

Table 3-5. Throughput, Efficiency, Response Time and Primary Job Running Time for Multiple Job Case

Experiment	Number of Jobs Run	Throughput (jobs/hr)	Efficiency (%)	Avg. Turnaround Time (s)	Primary Job Running Time (s)
s6e1	7	36.01	99.70	465.09	1728.16
s6e2	9	57.92	99.81	403.62	931.96
s6e3	11	38.79	96.42	826.41	3606.06
s6e4	8	27.60	95.16	763.2	3262.36
s6e5	10	44.35	91.06	576.13	2759.82
s6e6	7	35.51	99.39	569.04	1935.5
s6e7	6	37.15	99.25	530.61	1649.12
s6e8	8	41.63	99.63	411.83	1695.48

Figure 3-11 compares the baseline measurements from the set 3 experiments (primary jobs only) to the measurements obtained for the set 6 experiments (primary jobs with low-priority jobs run concurrently). These results show that the throughput of the cluster-level scheduler can be increased by running low-priority jobs concurrently with primary jobs. As we would expect, the increase in throughput is related to the efficiency with which the set of primary jobs (in the set 3 experiments) utilized the processor. Experiments in set 3 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 6;

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 3) and thus resulting in bigger gains in throughput when run with low-priority jobs in set 6. Figure 3-12 shows the associated increase in the efficiency of the cluster-level scheduler.

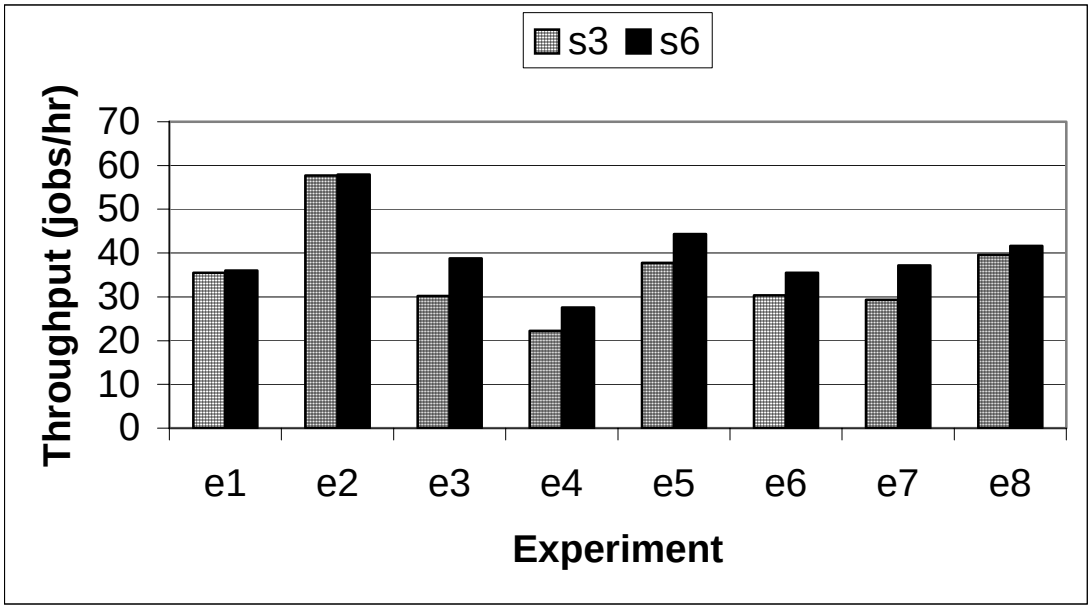

Figure 3-11. Baseline Throughput vs. Throughput with Low-priority Jobs

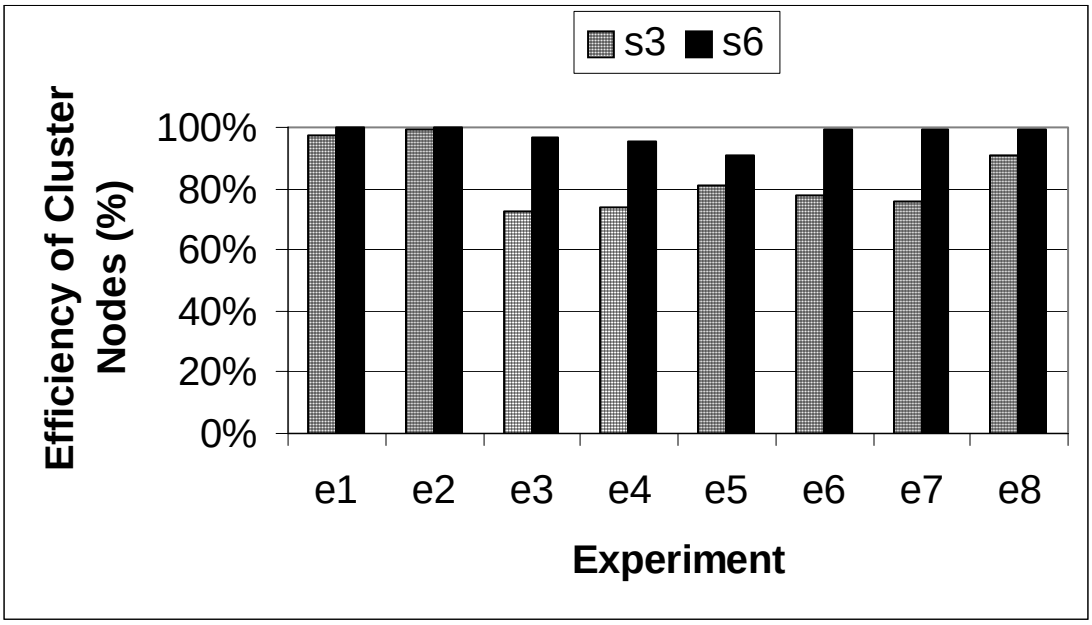

Figure 3-12. Baseline Efficiency vs. Efficiency with Low-priority Jobs

Additionally, Figure 3-13 demonstrates that we can achieve this 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 low-priority jobs. Since the low-priority jobs are run with the kernel's nice mechanism, they will take some CPU-time away from the primary job (as discussed in Section 3.2.1). Thus the low-priority jobs' average turnaround time may decrease but the primary jobs' average turnaround time may increase. On the other hand, when the primary jobs are I/O-bound (e.g., e3 and e4), the low-priority jobs are able to run sooner while having less of an impact on the primary jobs (versus when running with CPU-bound primary jobs). Thus, as the impact of the low-priority jobs on the primary jobs gets smaller, so does the increase in the primary jobs' average turnaround time. Additionally, as the primary jobs' CPU utilization efficiency decreases, so does the low-priority jobs' average turnaround time. Obviously, when the low-priority jobs' average turnaround time decreases more than the primary jobs' average turnaround time increases for a given experiment, we see an overall decrease in the average turnaround time of the set of jobs run in that experiment. Taking

note of the efficiency measurements presented in Table 3-4, we can see this behavior in Figure 3-13.

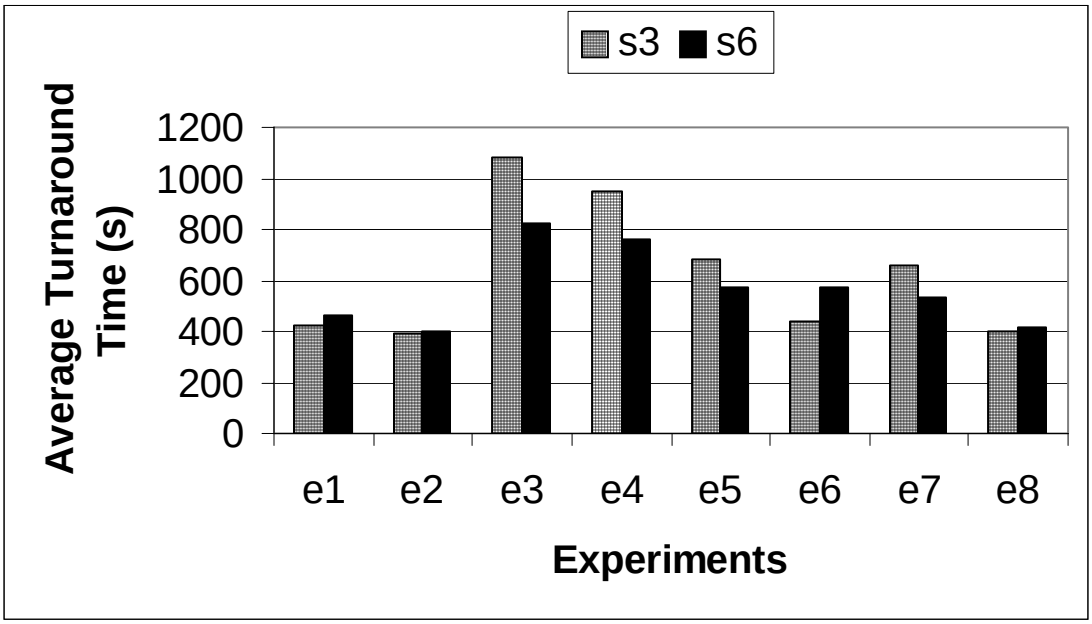

Figure 3-13. Baseline Average Turnaround Time vs. Average Turnaround Time with Low-Priority Jobs

The exception is experiment e6. Even though Table 3-4 lists an efficiency of 78% for s3e6 we do not see the decrease in average turnaround time that we expect in s6e6. Experiment e6 is different from the other experiments in that for set 3 (i.e., experiment s3e6), 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 set 6 (i.e., experiment s6e6), the primary jobs will be started immediately (i.e., at time 0). Since the primary jobs have priority over the low-priority jobs, the low-priority 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 (i.e., the jobs in e6).

Thus far we have seen that running low-priority jobs concurrently with primary jobs when there is unused CPU time on the cluster can not only increase the throughput (and

thus the efficiency) of the cluster but it can also lower the average turnaround time of the jobs. However, as we can see from Table 3-5 and Figure 3-14, this comes at the expense of the quality of service of the primary jobs. That is, in all of the experiments, the time needed to complete the primary jobs increased when low-priority jobs were run concurrently.

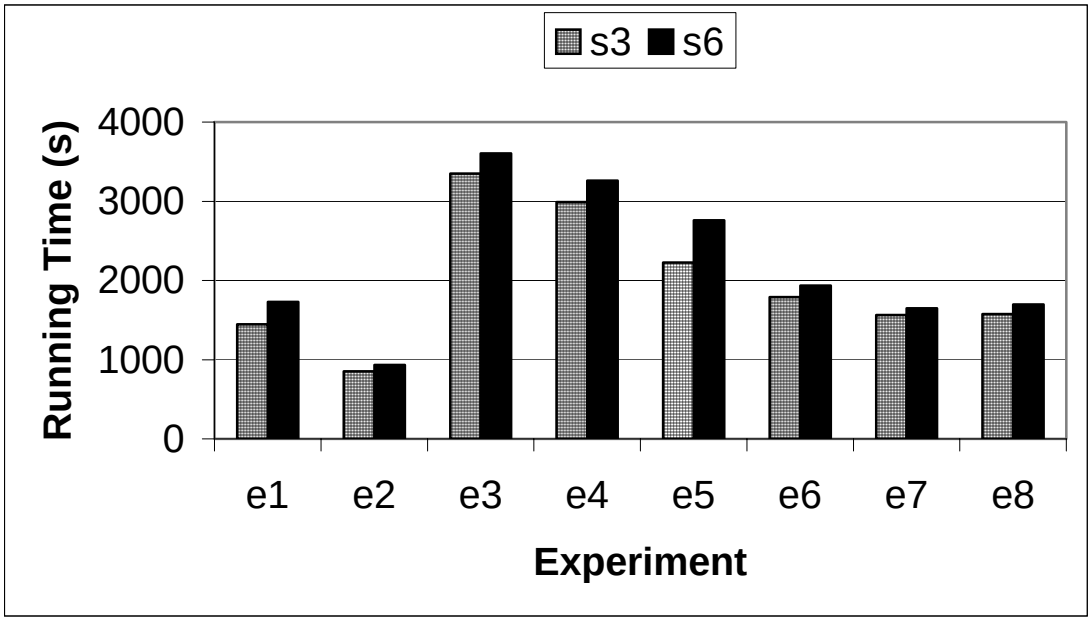

Figure 3-14. Baseline Running Time vs. Running Time with Low-priority Jobs

Figure 3-15 shows this data as the percent of increase over the baseline running times when low-priority jobs are run concurrently. We see an increase of at least 5% in the running times of the primary jobs with some sets of primary jobs being affected by 20-25%.

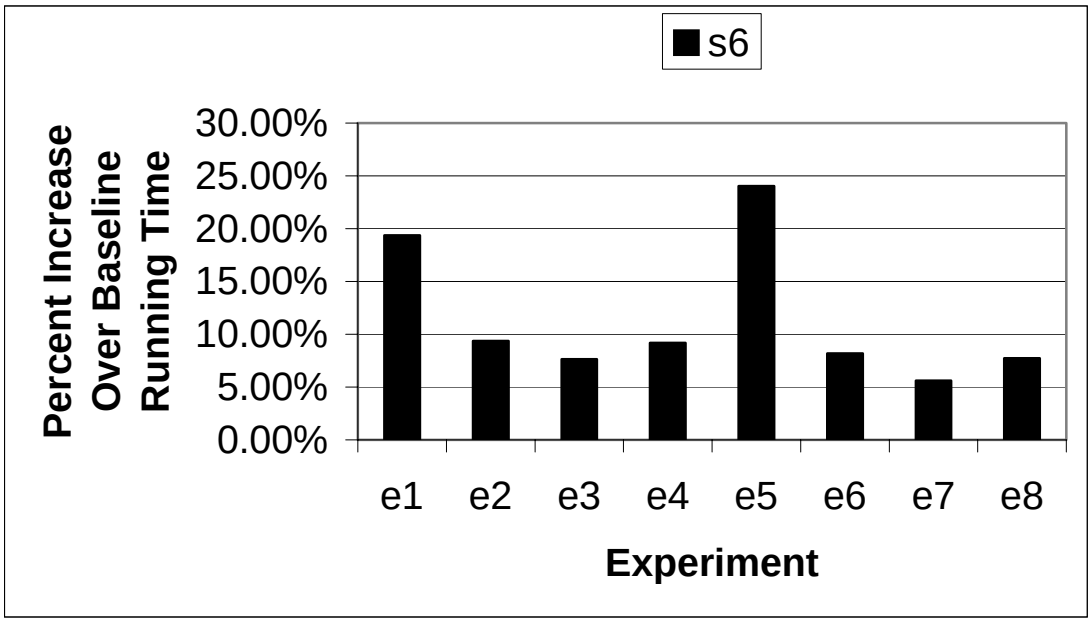

Figure 3-15. Percent Increase Over Baseline Running Time

In this chapter we have seen that by running low-priority jobs concurrently with primary jobs, we can increase the throughput and efficiency of the cluster-level scheduler as well as lower the average turnaround time of the scheduled jobs. However, this was at the expense of the quality of service of the primary jobs. In Chapter 4, we present results from experiments that use a new "guest" priority class, created by modifying the Linux kernel's CPU scheduler, to run guest jobs concurrently with primary jobs.

4 Evaluating Modifications to the Kernel's Priority Resource Management Facilities

We applied the kernel source code modifications used in [19] to our test kernel (2.4 SMP Linux kernel). As supplied by Ryu, they supported a 2.2 Linux kernel. These modifications allow the kernel to differentiate between normal processes (i.e., processes as defined by the unmodified kernel) and guest processes. A guest process is a process that we want to run alongside normal processes but in such a way that the running times of the normal processes will not be significantly affected. The kernel modifications focus on adding stricter resource management policies to the kernel. These new policies will prohibit any guest processes from getting scheduled for the processor when there are normal processes that are eligible to run. In our experiments, primary jobs consist of normal processes and guest jobs consist of guest processes. Therefore, we want the kernel modifications to allow us to run guest jobs concurrently with primary jobs without affecting the quality of service that the primary jobs receive.

4.1 Description of Kernel Modifications

The kernel was updated so that if a process were given the lowest nice value of 19, it would 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.

As provided, 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 4-1.

```
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 4-1. Modified Linux 2.4 CPU Scheduler

4.2 Confirming Baseline Measurements

Our next set of experiments (set 4, group A) was used to validate the new modified kernel. Since none of the kernel modifications should affect the resource usage of normal processes (i.e., not a guest process) or the resource usage of guest processes that are not contending for resources, we should obtain matching resource usage statistics when the baseline experiments used with our unmodified kernel are run with our modified kernel. The results are shown in Table 4-1.

Table 4-1. Baseline Running Times with Modified Kernel

Experiment	Application	Run 1			Run 2			Run 3		
		CPU (s)	Wall (s)	Eff (%)	CPU (s)	Wall (s)	Eff (%)	CPU (s)	Wall (s)	Eff (%)
s4e1	GAUSS	179	180	99	179	180	99	179	180	99
s4e2	HAL1	661	167	99	661	167	99	662	167	99
s4e3	HPL	845	357	59	845	357	59	845	357	59
s4e4	MrBayes	236	90	66	230	89	65	255	97	66
s4e5	WRF	864	391	55	863	391	55	866	392	55
s4e6	PAUP	207	208	100	211	211	100	209	210	100

Comparing Table 3-1 with Table 4-1 we see that there has been no appreciable impact on the baseline performance of our set of applications.

4.3 Investigating Effect of Kernel Modifications

The primary change to our experiments is that a low-priority process will now be a guest process. That is, we will not use the kernel's original CPU priority scheduling scheme (e.g., by using the nice command). Instead, we will set the priority of the process to "guest." The modified kernel will recognize this new class of processes and act appropriately.

4.3.1 Effect on Running Times of Primary Jobs

The experiments in set 5 (group B) are the same as those in set 2 except that guest jobs are run concurrently with the primary jobs instead of running low-priority jobs concurrently with the primary jobs. The experiments in set 5 were intended to determine

how some of the applications, run as primary processes, are affected when guest processes are run concurrently. As in set 2, 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 where noted. The experiments in set 5 are:

- s5e1:
  - GAUSS
  - [GAUSS]
- s5e2:
  - MrBayes
  - ([GAUSS], [GAUSS], [GAUSS], [GAUSS])
- s5e3:
  - HAL1
  - [PAUP]
- s5e4:
  - WRF
  - [PAUP], [PAUP]
- s5e5:
  - WRF
  - [HAL1]
- s5e6:
  - HAL1
  - [PAUP] (specifically run concurrently with one HAL1 slave process)
- s5e7:
  - HAL1
  - [PAUP], [PAUP], [PAUP], [PAUP] (specifically run concurrently with all HAL1 processes)
- s5e8:
  - HAL1
  - [PAUP], [PAUP], [PAUP] (specifically run concurrently with all three HAL1 slave processes)

Table 4-2 shows the CPU time, wall time and efficiency of each of these applications when running a guest process concurrently with them.

Table 4-2. Average CPU Time, Wall Time and Efficiency of Primary Job with a Guest Job

Experiment	Primary Application	CPU (secs)	Wall (secs)	Efficiency (%)
s5e1	GAUSS	178.63	179.54	99.49
s5e2	MrBayes	239.34	90.71	66.23
s5e3	HAL1	661.35	166.30	99.60
s5e4	WRF	866.36	391.19	55.42
s5e5	WRF	864.18	391.44	55.25
s5e6	HAL1	661.09	166.61	99.39
s5e7	HAL1	661.22	166.52	99.58
s5e8	HAL1	661.62	166.72	99.52

Figure 4-2 shows how the running time of the primary jobs have been affected by running a guest job concurrently. We see here that the kernel modifications have had the desired effect. That is, the running time of the primary jobs has not changed significantly from the baseline running times found in the set 2 runs. We can see that the kernel modifications (i.e., guest processes) 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 running 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.

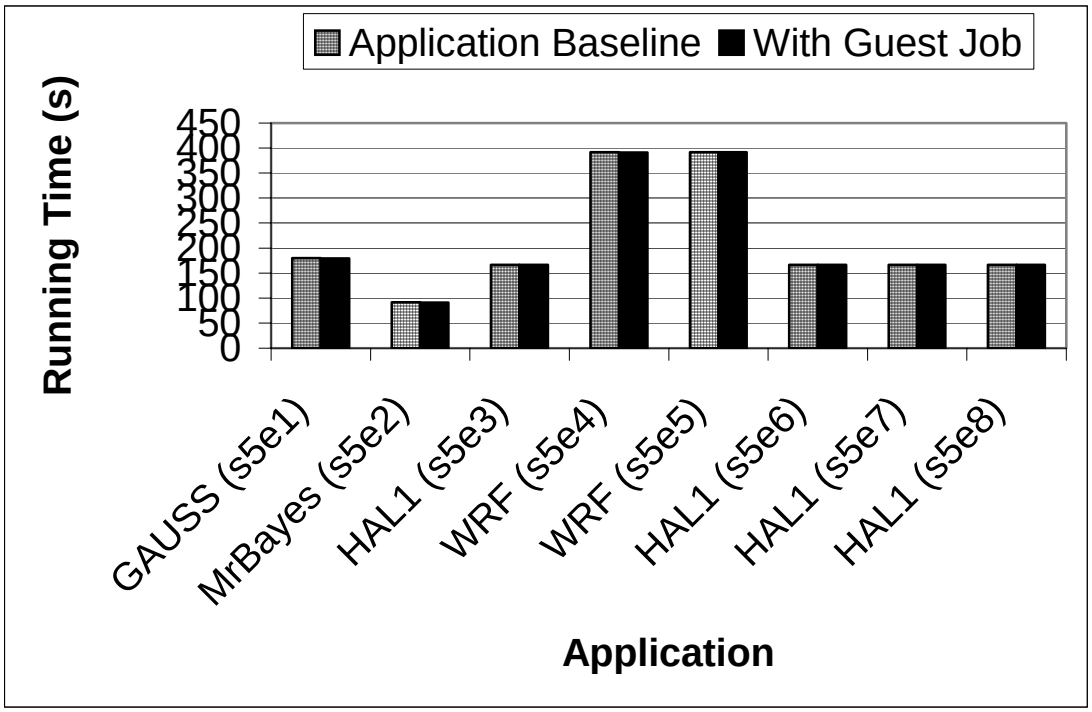

Figure 4-2. Baseline Running Times vs. Running Times with Guest Job

4.3.2 Effect on Throughput, Efficiency and Response Time

The experiments in set 7 are the same as those in presented in 3.2.2 (for set 6, group C) except that the kernel has been modified to allow guest jobs to run along with the primary jobs instead of low-priority jobs. Figure 4-3 illustrates the timeline for the application sequence for s7e1 shown in the list below. We measured the throughput of the cluster resource management system 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).

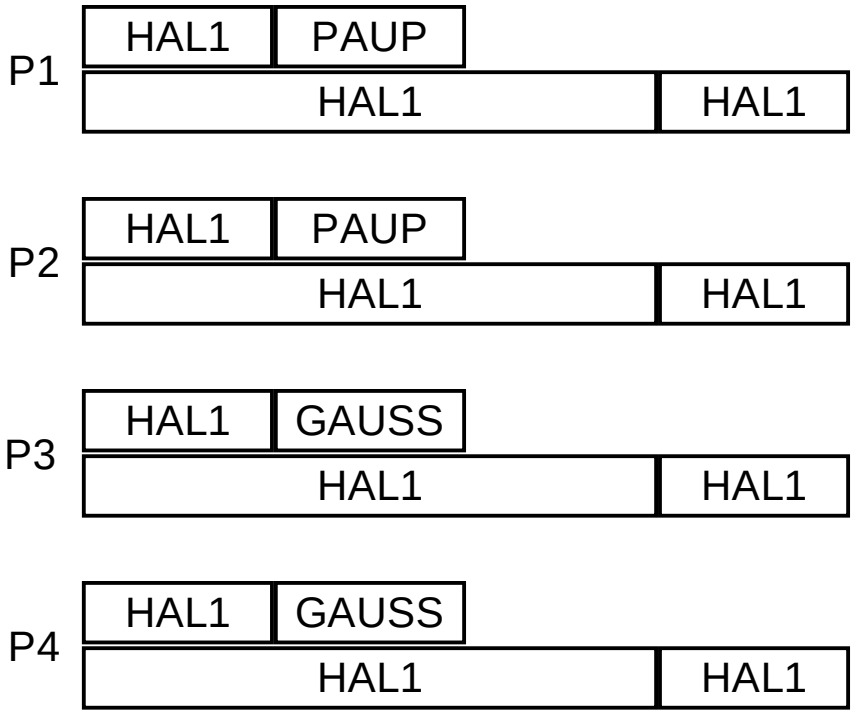

Figure 4-3. Application Timeline for s7e1

The efficiency measurement here is a calculation of how efficiently the set of jobs in a given experiment used the set of processors assigned to them. 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 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 amount of 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 time needed for all primary jobs to complete. The results from set 7 are presented in Table 4-3.

Table 4-3. Throughput, Efficiency, Response Time and Primary Job Running Time for Multiple Job Case using a Modified Kernel

Experiment	Number of Jobs Run	Throughput (jobs/hr)	Efficiency (%)	Avg. Turnaround Time (s)	Primary Job Running Time (s)
s7e1	7	35.32	97.03	422.95	1451.19
s7e2	9	57.78	99.22	396.59	852.84
s7e3	11	36.65	97.09	886.99	3357.76
s7e4	8	26.60	97.29	808.66	2988.14
s7e5	10	42.55	94.02	642.06	2222.1
s7e6	7	34.85	99.29	572.85	1784.02
s7e7	6	34.97	99.60	561.33	1559.07
s7e8	8	40.96	99.60	402.87	1590.22

Figure 4-4 and Figure 4-5 compare the baseline measurements from the set 3 experiments (primary jobs only) to the measurements obtained for the set 7 experiments (primary jobs with guest jobs run concurrently) as well as to the measurements from set 6 (primary jobs with low-priority 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 measurements 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 3 experiments) utilized the processor. Experiments in set 3 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 3) 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 4-6 compares the baseline efficiency (set 3) with the efficiency when running concurrently with low-priority jobs (set 6) and guest jobs (set 7).

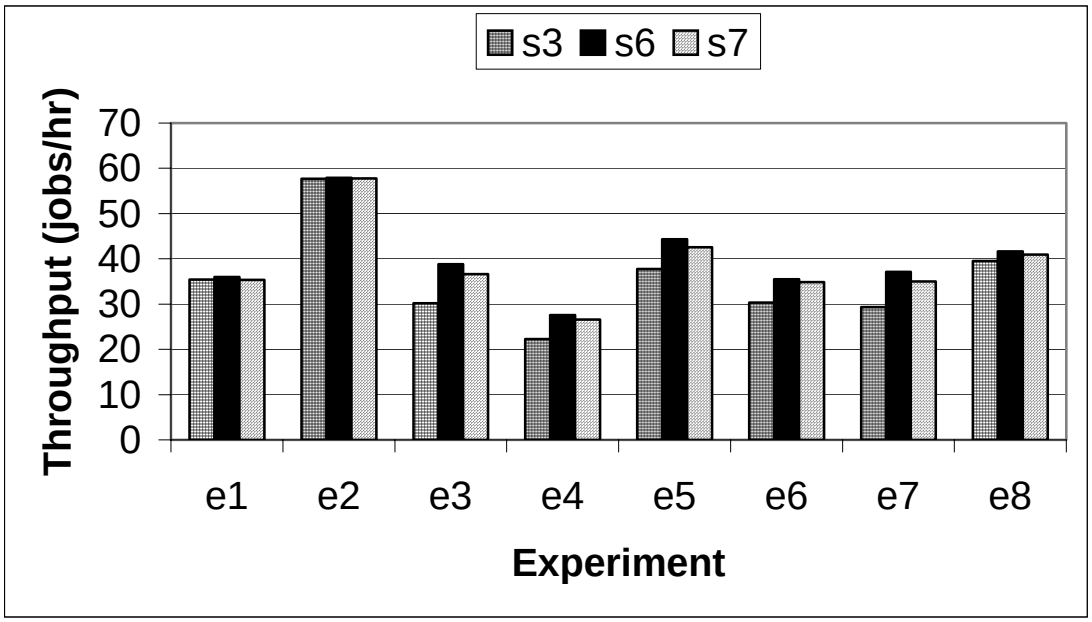

Figure 4-4. Baseline Throughput (s3) vs. Throughput with Low-priority Jobs (s6) vs. Throughput with Guest Jobs (s7)

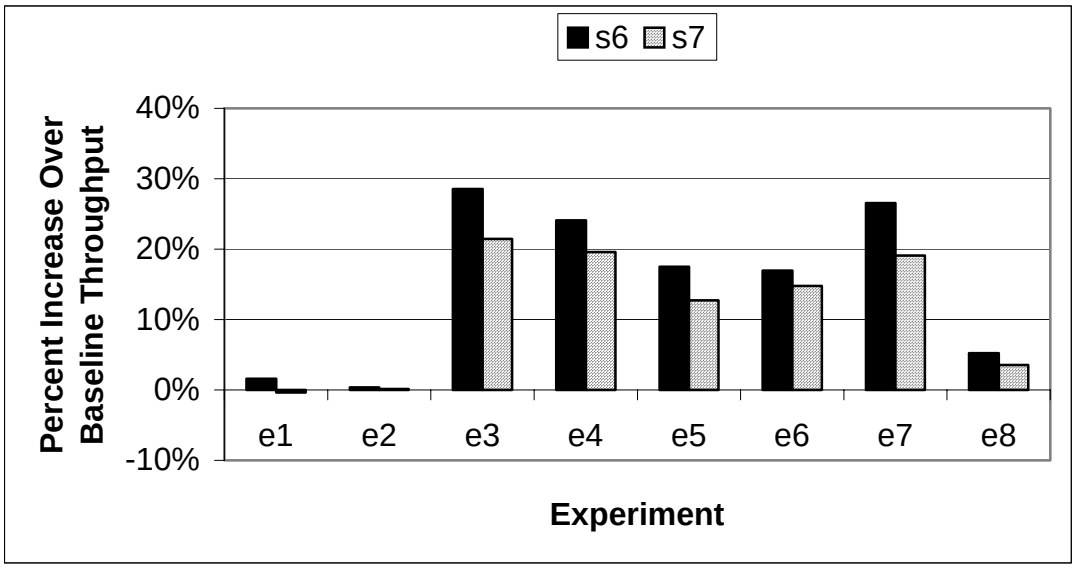

Figure 4-5. Percent Increase Over Baseline Throughput (Low-priority Jobs (s6) vs. Guest Jobs (s7))

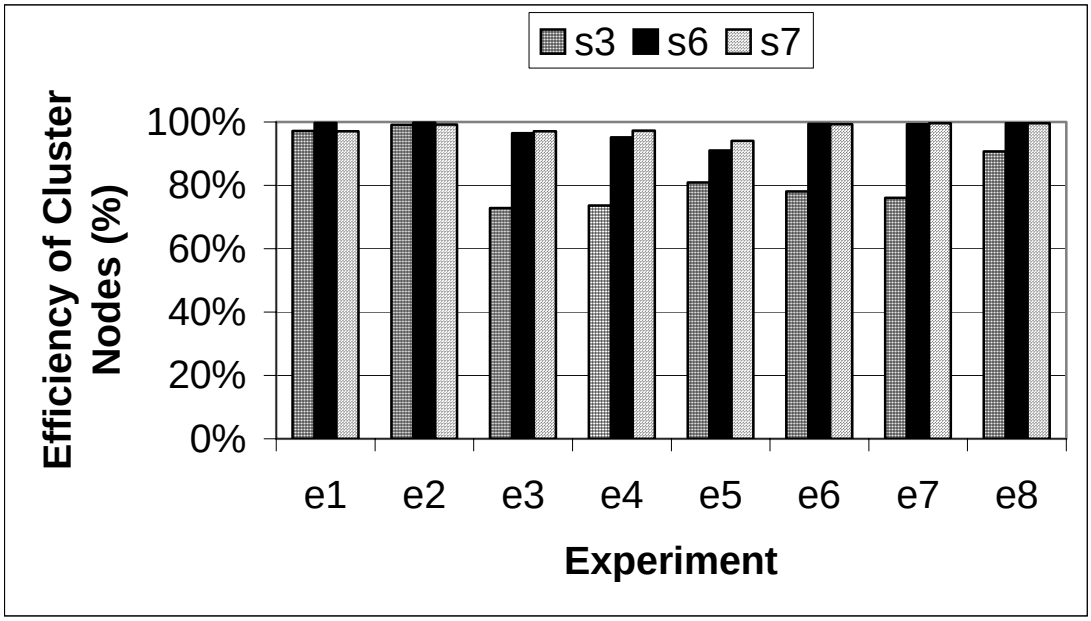

Figure 4-6. Efficiency of Cluster Nodes (Baseline (s3) vs. Low-priority Jobs (s6) vs. Guest Jobs (s7))

Additionally, Figure 4-7 demonstrates that we can achieve this 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 gust processes are run only when there are no runnable primary processes, the guest jobs should not take any CPU time away from the primary jobs (as discussed in Section 0). Thus the guest jobs' average turnaround time should not change nor should the primary jobs' average turnaround 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 3). Taking note of the efficiency measurements presented in Table 3-4, we can see this behavior in Figure 4-7.

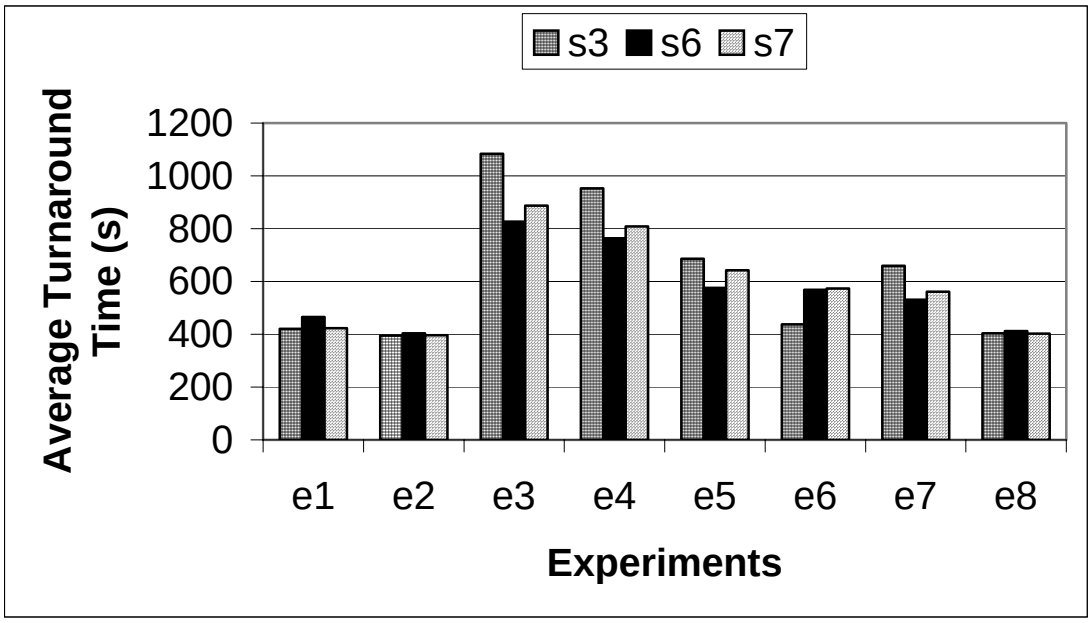

Figure 4-7. Baseline Average Turnaround Time (s3) vs. Average Turnaround Time with Low-priority Jobs (s6) vs. Average Turnaround Time with Guest Jobs (s7)

The exception is experiment e6. Even though Table 3-4 lists an efficiency of 78% for s3e6 we do not see the decrease in average turnaround time that we expect in s7e6. Experiment e6 is different from the other experiments in that for set 3 (i.e., experiment s3e6), the jobs are executed so that the guest jobs are run first followed by the primary jobs. When the same jobs are run in set 7 (i.e., experiment s7e6), the primary jobs will be started immediately (i.e., at time 0). Since the primary jobs have priority over the guest jobs, the guest jobs will all see an increase in their average turnaround times. In this case, the increase in the average turnaround time for the five guest 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 (i.e., the jobs in e6).

Thus far we have seen that running guest jobs concurrently with primary jobs when there is unused CPU time on the cluster can not only increase the throughput (and thus the efficiency) of the cluster but it can also lower the average turnaround time of the jobs. Additionally, as we can see from Table 4-3 and Figure 4-8, running guest jobs concurrently with primary jobs will not impact the quality of service received by the primary jobs. That is, in all of the experiments in set 7, the time needed to complete the

primary jobs did not change significantly when guest jobs were run concurrently. This is what we expected after seeing the result from set 5 that were presented in Figure 4-2.

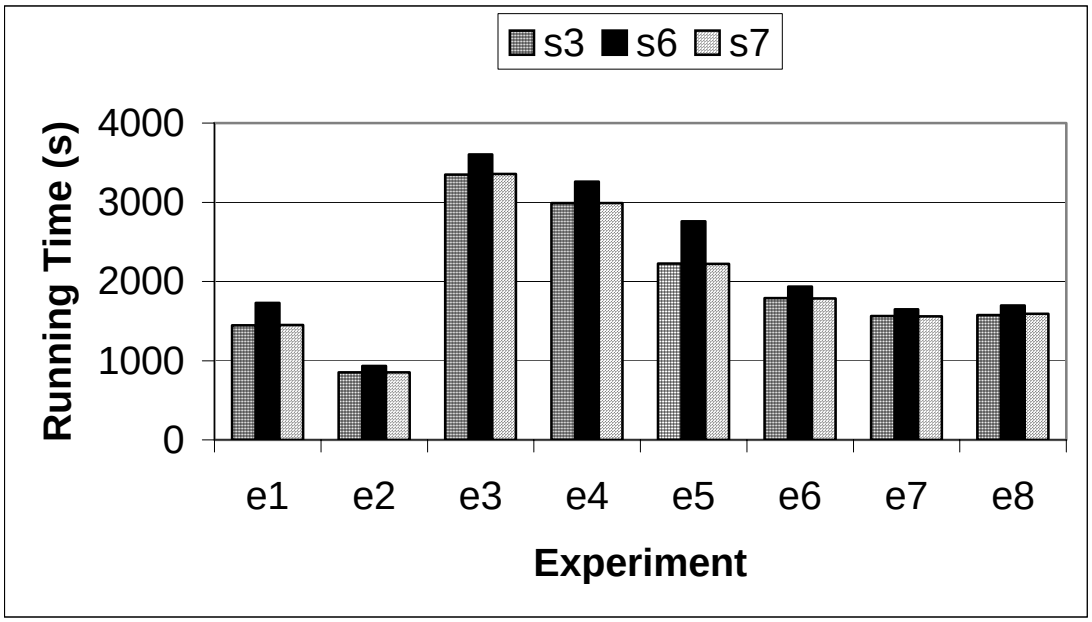

Figure 4-8. Baseline Running Time (s3) vs. Running Time with Low-priority Jobs (s6) vs. Running Time with Guest Jobs (s7)

Figure 4-9 shows the running time data as the percent of increase over the baseline running times when guest jobs are run concurrently while also comparing these percentages to those from Figure 3-15 (the set 6 runs). Notice that we do not see any significant increase in the running times of any of the primary jobs (less than 1% increase). Compare this to the low-priority job case that saw an increase of at least 5% in the running times of the primary jobs with some sets of primary jobs being affected by 20-25%.

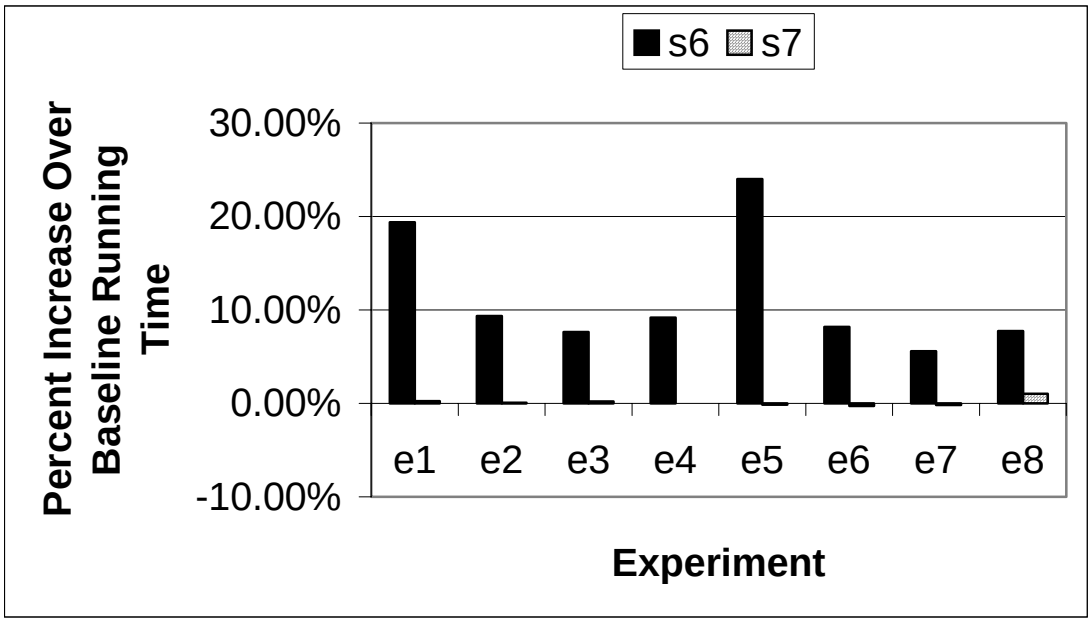

Figure 4-9. Percent Increase Over Baseline Running Time (Low-priority Jobs (s6) vs. Guest Jobs (s7))

5 Conclusions

The 2.4 Linux kernel provides users with the ability to specify a “nice” value for a given process. Using the lowest possible nice value for all processes in a low-priority job still causes low-priority jobs to impact the running time of the primary jobs. On the other hand, using the kernel modifications enabled fine-grained control over CPU usage and allowed us to successfully keep guest processes from significantly interfering with the CPU usage of primary processes.

As one would expect, the running time of CPU-bound primary jobs is impacted more than that of I/O-bound primary jobs. However, we saw that applications can be affected in ways other than increasing wall clock time. This was demonstrated with the HAL1 application where, although the wall clock time was increased, the CPU time was decreased. We saw this behavior when running low-priority jobs along with the HAL1 primary job. However, when run concurrently with a guest process, neither the CPU nor the wall time of the HAL1 primary job was significantly impacted.

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 the primary job. The guest process mechanism in the modified kernel can help to maintain the quality of service of the primary job 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. There can be exceptions, however, if there are a large number of jobs that finish later and a smaller number of jobs that finish sooner.

For our experiments, 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 kernel modifications 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 and running times are within 1% of their baselines.

References

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

[2] Banerjee, et al., “A Closer Look at Coscheduling Approaches for a Network of Workstations,” SPAA’99. Saint Malo, France.

[3] Brightwell, R. et al., “A Performance Comparison of Linux and a Lightweight Kernel,” Proc. 2003 Int’l Conf. on Cluster Computing (CLUSTER’03). December 2003.

[4] Campbell, J. F., et al., “Solving Hub Arc Location Problems on a Cluster of Workstations,” Parallel Computing, 29(5), May 2003, Pages 555-574.

[5] GAUSS Mathematical and Statistical System. Aptech Systems, Inc. http://www.aptech.com/. September 2004.

[6] Gropp, W. et al., MPICH Implementation. http://www.mcs.anl.gov/mpi/mpich/papers/mpicharticle/paper.html. September 2004.

[7] He, L., et al, “Dynamic Scheduling of Parallel Real-time Jobs by Modeling Spare Capabilities in Heterogeneous Clusters,” Proc. 2003 Int’l Conf. on Cluster Computing (CLUSTER’03). December 2003.

[8] Huelsenbeck, J. and Ronquist, F., “MrBayes: Bayesian Inference of Phylogeny,” http://morphbank.ebc.uu.se/mrbayes/info.php. September 2004.

[9] Kleban, S. et al., “Interstitial Computing: Utilizing Spare Cycles on Supercomputers.” Proc. 2003 Int’l Conf. on Cluster Computing (CLUSTER’03). December 2003.

[10] Krueger, P. and Chawla, R., “The Stealth Distributed Scheduler,” Proc. 11th Int'l Conf. Distributed Computing Systems, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 336-343.

[11] Leutenegger, S. T. and Sun, X., “Distributed Computing Feasibility in a Non-Dedicated Homogeneous Distributed System.” Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1993.

[12] Litzkow, M., Livny, M., Mutka, M., "Condor – A Hunter of Idle Workstations," International Conference on Distributed Computing Systems. June 1988, pp. 104-111.

[13] Lo, V., Mache, J., “Job Scheduling for Prime Time vs. Non-prime Time, “Job Scheduling for Prime Time vs. Non-prime Time.” Proc. 2002 Int’l Conf. on Cluster Computing (CLUSTER’02). September 2002.

[14] “OpenPBS Administration Guide.” http://www.openpbs.org/docs.html. September 2004.

[15] Ousterhout, J. K., “Scheduling Techniques for Concurrent Systems,” In Proceedings of the 3rd International Conference on Distributed Computing Systems, pp. 22-30, May 1982.

[16] 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/. September 2004.

[17] Ryu, K. D., et al., “Efficient Network and I/O Throttling for Fine-Grained Cycle Stealing,” SC2001. Nov. 2001, Denver.

[18] Ryu, K. D. and Hollingsworth, J. K., “Exploiting Fine-Grained Idle Periods in Networks of Workstations,” IEEE Transactions on Parallel and Distributed Systems, 11(7). July 2000.

[19] Ryu, K. D. and Hollingsworth, J. K., “Linger-Longer: Fine-Grain Cycle Stealing for Networks of Workstations,” SC’98. Nov. 1998, Orlando.

[20] Ryu, K. D. and Hollingsworth, J. K., “Unobtrusiveness and Efficiency in Idle Cycle Stealing for PC Grids,” Proc. 18th Int’l Parallel and Distributed Processing Symposium (IPDPS’04).

[21] Sterling, T. et al., “BEOWULF: A Parallel Workstation for Scientific Computations,” Proc. 24th Int’l Conf. on Parallel Processing, August 1995.

[22] “Sun Grid Engine Reference Guide.” http://gridengine.sunsource.net/project/gridengine/documentation.html. September 2004.

[23] Swafford, D., “PAUP*: Phylogenetic Analysis Using Parsimony (and Other Methods),” http://paup.csit.fsu.edu/index.html. September 2004.

[24] Urgaonkar, B. and Shenoy, P., " Sharc: Managing CPU and Network Bandwidth in Shared Clusters," IEEE Transactions on Parallel and Distributed Systems, 15(1), pp. 2-17, January 2004.

[25] Weatherly, D. B., "Dyn-MPI: Supporting MPI on Non Dedicated Clusters," SC'03. Nov. 2003. Phoenix.

[26] "Weather Research and Forecast (WRF) Modeling System," National Center for Atmospheric Research, University Corporation for Atmospheric Research. http://www.wrf-model.org/Welcome.html. September 2004.

[27] 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.

Vita

Name: Gary Stiehr

Undergraduate Study: University of Missouri – St. Louis:
Bachelor of Science in Computer Science (2000)
Bachelor of Arts in Mathematics (2000)

Graduate Study: Washington University in St. Louis:
Master of Science in Computer Science (2004)

Publications

- James F. Campbell, Gary Stiehr, Andreas T. Ernst and Mohan Krishnamoorthy, "Solving Hub Arc Location Problems on a Cluster of Workstations," Parallel Computing, Volume 29, Issue 5, May 2003, Pages 555-574

December 2004