ABSTRACT

Title of dissertation: EXPLOITING IDLE CYCLES

IN NETWORS OF WORKSTATIONS

Kyung Dong Ryu, Doctor of Philosophy, 2001

Dissertation directed by: Associate Professor Jeffrey K. Hollingsworth
Department of Computer Science

Studies have shown that workstations are idle a significant fraction of the time. Traditional idle resource harvesting systems define a social contract that permits guest jobs to run only when a workstation is idle. To enforce this contract, guest jobs are stopped and migrated as soon as the owner resumes use of their machines. However, such systems waste many opportunities to exploit idle cycles because of overly conservative estimates of resource contention.

In this thesis, we present a new policy, called Linger-Longer, that refines the social contract to permit fine-grain cycle stealing. Linger-Longer allows guest jobs to linger on a machine at low priority even when local tasks are active. Also, we developed a new adaptive job migration scheme based on runtime cost/benefit analysis. Our simulation study shows that the Linger-Longer policy can improve the throughput of guest jobs on a cluster by up to 60% with only a few percent slowdown of local jobs. The simulation also demonstrates that guest parallel jobs can perform better with our new approach than with the traditional run-time reconfiguration approach.

To limit the impact of guest jobs' resource use, new local resource scheduling policies and mechanisms are required. We present a suite of mechanisms to support prioritized use of CPU, memory, I/O and network bandwidth.

An overall prototype of the Linger-Longer system has been implemented in Linux. The prototype integrates the operating system extensions for resource throttling and a new adaptive migration policy module while leveraging general job scheduling and checkpointing mechanisms of an existing system. Using this prototype, we conduct a head-to-head performance comparison between our fine-grain cycle stealing policies and the traditional coarse-grain cycle stealing policies. The experiment on a Linux cluster with a set of benchmark applications show that, overall, Linger-Longer can improve the guest job throughput by 50% to 70%, with only a 3% host job slowdown.

EXPLOITING IDLE CYCLES

IN NETWORKS OF WORKSTATIONS

by

Kyung Dong Ryu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2001

DEDICATION

To my wife, Kyoung-Ah,

who has provided me with so much love, support and encouragement,

and

to my daughter, Jenna,

whose smile has always relieved my fatigue.

ACKNOWLEDGMENTS

I would like to thank many people who have given me their help and support throughout the long journey to my Ph.D.

First of all, I would like to express my deepest gratitude to my advisor, Dr. Jeffrey Hollingsworth. Without his enthusiastic support and constant encouragement, this dissertation would not have been possible. I would also thank Dr. Peter Keleher and Dr. Bobby Battacharjee for their excellent guidance and constructive criticism. Their help was especially appreciated while my advisor was away on sabbatical. My appreciation goes to my other committee members as well. Dr. Alan Sussman gave many helpful comments and suggestions which helped me make my dissertation more complete. Dr. Sung Won Lee, who has always been a God father for Korean graduate students here, has given me worthy advice on how to be a good professor in my first career.

My graduate years were more happy memories than painful ones because I had cheering officemates, Bryan Buck, Tikir Mustafa and Chadd Williams. I also owe a great deal to Chris League, who is now studying in Yale. I should not forget to thank Ugur Centitemel for encouraging me whenever I was down.

I'm grateful to all my Korean friends. Sungjoon Ahn and Hwansoo Han have been great friends and tennis partners to me for more than 10 years. Jung Min Kim and Kyong Il Yoon have helped me in learning how to build and maintain a happy family life. I also thank all the Korean graduate student tennis club members: Eungchul Kim, Jung-Jun Park, Youngsoo Lee, Jongwon Kim and all the others. Playing tennis, chatting and drinking with them on Friday nights has always re-energized me so that I could happily jump back into my research.

To my parents, Jae-Il Ryu and Hyeon-Ja Kim, thank you for supporting me whatever I do and wherever I go. They have been and will be my role models, forever. I also appreciate my parents-in-law, Doo-Ki Hur and Soon-Ja Choi, who have prayed for my family and me all the time.

To my wife, Kyoung-Ah, and lovely daughter, Jenna, I cannot express my thanks with any words. Getting a Ph.D. is still nothing compared to sharing my life with them.

Lastly, I thank God who has provided me with all my opportunities and blessings.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation

The need for large amount computing power over sustained intervals of time has increased in the scientific and engineering community. In contrast to interactive users, who are concerned with response time, these users are primarily concerned with the throughput of their applications over relatively long time periods. The application domains requiring substantial computational resources includes a wide spectrum of areas, such as engineering design applications, neural networks, high-energy physics, computer hardware and software simulations, materials science simulations, optimization problems and computational biology.

Such applications have been the main workload of supercomputers. However, better cost-performance ratios and high-speed networking technologies have brought workstations and PCs into this arena. Today, collections of workstations and PC's connected through high-speed networks are available in research centers, universities and even many office environments. Studies have shown that up to three-quarters of the time workstations are idle [51]. Generally, most machines go unused during the night, lunch time, and the whole day during weekends and holidays.

Conventional cycle-stealing systems such as Condor [44], LSF [77], NOW [5] and others [21, 34, 37] have been created to use these available resources. Such systems focus on coarse-grained idle periods (at the scale of minutes or hours) when users are away from their workstations. To gain access to these resources machine owners are offered a "social contract" that guest jobs are allowed to run only on idle machines. To enforce this contract, guest jobs are stopped and migrated as soon as the owner resumes use of their computer.

However, there are more available cycles that such systems don’t harvest. This is due to the fact that even when the user is actively working on their machine, the resource usage is very low. This can be explained by the fact that the main use of computers at work is editing documents, web surfing, and reading email. When writing and editing a document, most time is spent thinking and typing, and it does not require significant computing resources. Likewise, when web browsing and reading email, network resources are required, but the CPU is not generally loaded. Even among most demanding users, only rarely are CPUs fully utilized.

1.2 Overview

The thesis of this research is to achieve more efficient use of computing resources and hence produce higher system throughput. The traditional approaches don't take advantage of extra free resources because it would slow down host processes and therefore break the "contract" with machine owners. Unobtrusive use of idle resources is essential to deploying these kinds of systems because it is otherwise difficult to convince users to participate in them. The mechanism to steal idle cycles from non-idle machines should include a way to protect host processes' performance. To enforce this unobtrusiveness while achieving better efficiency, we propose to run guest jobs with very low priority (so low that host jobs are allowed to starve the guest jobs) on non-idle machines: we call this approach *fine-grain cycle stealing* and name our system *Linger-*

Longer. This enables the system to utilize most idle cycles while limiting the slowdown of the owner's workload to an acceptable level.

In Linger-Longer, job migration is no longer compulsory to satisfy the user “contract” to allow optimizing performance. When resource usage becomes high on the current node and there is a fully idle node, timely migration will speed up the guest job. We developed a new migration policy in which the migration point is adaptively determined taking migration cost and migration benefit into consideration.

Before full implementation, we conducted simulations to measure the performance gain of the new fine-grain cycle stealing approach. Since synthetic workloads often produce misleading results, we used trace data to drive the local workload representing local user activities. We compare two variations of our policy: Linger Longer and Linger Forever, to two previous policies: Immediate-Eviction and Pause-and-Migrate. The simulations demonstrated that the Linger-Longer policy can improve the throughput of guest jobs on a 64 machine cluster by up to 60% with only a 1% slowdown of host jobs. For parallel computing, we showed that our policy outperforms reconfiguration strategies when the processor utilization by the host process is 20% or less in both synthetic bulk synchronous and real data-parallel applications.

Unfortunately, the priority mechanisms of current operating systems, such as the Unix *nice* command, do not fully support unobtrusive use of the idle resources. As a result, we developed operating system support for the Linger priority for guest jobs. This includes memory and other resources as well as CPU. A kernel extension for a Linger priority of CPU scheduling and the memory management have been developed. Also, we provided additional mechanisms for efficient communication and file I/O throttling. We demonstrated that these mechanisms effectively limit the resource usage by guest jobs on non-idle nodes and minimize the slowdown of host jobs. Although all these mechanisms were developed for our Linger-Longer system, they are general enough to serve other types of use. CPU and memory priority mechanisms can provide ultra-low job priority augmenting the nice command in Unix. Our communication and I/O throttling mechanisms can support light-weight rate-based bandwidth scheduling which can bound maximum bandwidth usage.

A prototype of the overall Linger-Longer system was implemented on the Linux operating system. Rather than implementing everything from the scratch, we leveraged an existing system, Condor. Condor already provides modules for guest job scheduling, checkpointing and migration. Our prioritized local resource scheduling mechanisms at the kernel level and the cost/benefit based adaptive migration module have been incorporated into Condor. We conducted a head to head performance comparison between the traditional policies: Immediate-Eviction and Pause-and-Migrate, and our novel policies: Linger-Longer and Linger-Forever in a non-dedicated Linux cluster. The experiments with trace-based local workloads in eight machine non-dedicated cluster reported that Linger-Longer can improve the guest job throughput by 50% to 70%.

1.3 Organization of Thesis

The remainder of this thesis comprises six chapters.

Chapter 2 surveys related work in the area. We discuss the literature in idle cycle harvesting, process migration, parallel job scheduling, meta-computing, adaptive applications, and operating system support for resource aware scheduling.

Chapter 3 presents our new fine-grain cycle stealing policy that harvest more idle cycles from non-dedicated machines without an obtrusive impact on machine owners. Also, we introduce a new adaptive job migration scheme which determines the migration point based on cost/benefit analysis.

Chapter 4 presents our simulation model and performance evaluation results. The simulations demonstrate the potential performance gain of our fine-grain cycle stealing policy (named Linger-Longer) over traditional eviction-based coarse-grain cycle stealing policies.

Chapter 5 discusses operating system mechanisms to support fine-grain cycle stealing. A suite of resource policing mechanisms for CPU, memory, I/O and network is introduced. Each mechanism is implemented on Linux and validated by a series of experiments.

Chapter 6 presents integration of all the mechanisms in the prototype Linger-Longer system. Then, we conduct a head-to-head performance comparison between our Linger-Longer system and the Condor system, which is one of the most widely used coarse-grain cycle stealing systems.

Chapter 7 summarizes the contribution of this thesis and concludes with future work.

Chapter 2

Related Work

2.1 Idle Cycle Harvesting and Process Migration

Private workstations connected by a network have long been recognized as useful for computation intensive applications. Since a large fraction of the time workstations are unused, idle cycles can be harnessed to run scientific computations or simulation programs as a single process or a parallel job. The usefulness of this approach depends on 1) how much time the machines are available, 2) how well those available resources can be harnessed and 3) how little the machine owners are negatively affected.

Mutka and Livny [51] found that on the average, 75% of the time machines were idle. Similarly, Krueger and Chawla [40] demonstrated that in 199 diskless Sun Workstations idle time reached 91% on the average and that only 50% of those idle cycles are made available for background jobs. The other half of the idle cycles could not be harnessed since they belong to the time when the owners are using their machines or a waiting time to ensure that users are away. More recently, Acharya et al [1] also observed that 60% to 80% of the workstations in a pool are available by analyzing machine usage traces from three academic institutions.

Many research prototypes and practical systems have been developed to harvest those idle cycles. Condor [44] is built on the principle of distributing batch jobs around a cluster of computers. It identifies idle workstations and schedules background jobs on them. The primary rule Condor attempts to follow is that workstation owners should be able to access their machine immediately when they want them. To do this, as soon as a workstation’s owner activity is detected at an idle machine, the background job is stopped and quickly migrated to another idle machines. This low perturbation led to successful deployment of the Condor system. For fault tolerance, Condor checkpoints jobs periodically for restoration and resumption. It also provides participating machine owners with the mechanisms to individually describe in what condition their machine can be considered idle and used. IBM modified Condor to produce a commercial version, Load Leveler [37]. It supports not only private desktop machines but also IBM’s highly parallel machines.

Sprite [22] is another system that provided process migration to use idle machines. A migrated process is evicted when the owner reclaims their machine. The major difference from Condor is that job migration is implemented at the operating system level. The advantage is the migration overhead is much lower: a few hundred milliseconds while user level evictions typically occur in a few seconds. However, the fully custom OS kernel hinders the wide deployment of the system.

DQS [34] is an early non-commercial cluster computing system. Like Condor, it supports most of the existing operating systems. Different job queues are provided based on architecture and group. DQS ensures the local autonomy of a private machine by suspending currently running background jobs when keyboard or mouse activities are detected. However, their emphasis is placed on distributing the jobs to different shared machine clusters in a balanced manner. Jobs can be suspended and resumed, but migration is not supported. DQS 3.0 [23], the latest version, is widely used by companies such as Boeing and Apple Computer.

Load Sharing Facility (LSF) [77] was developed to automatically queue and distribute jobs across a heterogeneous network of Unix computers. This system was intended for much larger group of workstation clusters consisting of thousands of machines. The basic assumption on each participating machine is that a host is by default sharable, which is opposite to the policy of Condor-like systems. Local autonomy is not respected, and hence few mechanisms are provided to protect machine owners. LSF focuses on two major goals. The first is to place the jobs on the node which meets the application's resources requirements. The second is to balance the load across machine clusters to achieve better turn-around time of jobs and system utilization in the entirety. Since job migration is not supported, a process only can run on the node where it started execution. It also lacks a checkpointing mechanism, so remote jobs are vulnerable to node failure.

The Butler system [21] also gives users access to idle workstations. This system requires the Andrew System [50] for shared file access. The basic concept of this system is to provide transparent remote execution on idle nodes. Lack of support for job migration in Butler can lead to loss of work by remote jobs when the machine owner returns. The system will just warn the remote user, then kill the process so as not to disturb the machine owner.

There is one approach, which supports local autonomy of individual machines in a different way. The Stealth system [40] runs remote processes with lower priority to preserve the performance of local work. Thus, when the owner reclaims their machine, the remote job does not leave the node, rather keeps running with low priority. They prioritized several major resources including CPU, memory and the file buffer cache on the

MACH operating system so as not to interfere with local processes. This is similar to our fine-grain cycle stealing approach. However, lack of support for job migration can lead to extreme delay or even starvation of background jobs. Also, this system is not intended for parallel jobs.

There have been various studies on specific issues in using idle cycles of machines. Theimer and Lantz [65] investigated how to find idle machines more efficiently. They found that a centralized architecture can be scaled better and can be more easily monitored, while a decentralized architecture is easier to implement. Bhatt et al [15] investigated finding an optimal work size to run remotely assuming that the partial result will be lost when idle machines are recliamed. In their cycle stealing model, too small chunks of remote work will suffer from high network overhead and too large chunks can waste cycles by losing more partial work. While checkpointing can avoid this problem, a study by Basney and Livny [13], based on their Condor experience, reported that checkpointing can cause a bursty consumption of network bandwidth, and hence interfere with interactive processes and even unrelated remote job migrations. They suggested a better design that gives priority to applications with low network requirements, which tends to favor smaller processes. Also, their proposed design limits the bandwidth allocation for job placement and checkpointing for a given period of time.

Immediately migrating a job off a remote workstation does not always satisfy machine owners since it takes a noticeable time to recover the previous state of machine such as CPU cache, I/O cache buffers and memory pages. Arpaci et al [7] limited this perturbation by restricting the number of times when returning users notice disruptions on their machine to a fixed limit per day. Petrou et al [56] present a more complex solution that tries to maximize the available time of idle machines. Their system predicts a user return time from the history and actively restores the memory-resident state before a user comes back.

General load balancing and process migration mechanisms have been studied extensively. MOSIX [12] provides load-balancing and preemptive migration for traditional UNIX processes. DEMO/MP [57], Accent [75], Locus [67], and V [66] all provide manual or semi-automated migration of processes.

2.2 Parallel Job Scheduling

Many computation-intensive applications have been parallelized to try to reduce their execution time. Since a collection of idle machines connected by a high-speed network can be viewed as a virtual parallel machine, several groups have investigated parallel computing in this environment. This is much more complicated than running multiple independent sequential jobs on idle machines. Furthermore, multiple parallel jobs should be able to be served at the same time because a very large pool of workstations can be wasted if only a single parallel job can run at once. In general, scheduling of multiple parallel programs is classified to two styles: time sharing and space sharing [27].

In time shared scheduling, different parallel jobs share the nodes and take turns for execution. However, global coordination across the processors is essential since independent process switching on each node will lead to large inefficiencies. In Gang scheduling, context switches between different parallel jobs occur simultaneously at all the processors in a synchronized manner. Thus, processes(or threads) can interact at a fine granularity.

There have been extensive studies to efficiently implement Gang scheduling. Ousterhout [55] suggested and evaluated a few algorithms for coscheduling. He verified that avoiding fragmentation of slots for processes on processors is critical for system throughput. A number of variants of coscheduling were also explored. Sbalvarro et al [63] presented demand-based coscheduling. This dynamic scheduling algorithm coschedules a group of processes that exchange messages. The implicit coscheduling work by Dessau et al[24] shows that coscheduling can be achieved implicitly by independent decisions of local schedulers based on the communication pattern. This study focuses more on how to immediately schedule the corresponding process and keep the communicating processes on the processors without blocking.

Space sharing partitions the nodes and assigns each job for exclusive use of a partition. This has the advantage of reducing the operating system overhead due to context switching [69]. However, an important constraint is that activities on one partition should not interfere with the others. Therefore, processor partitioning in a classic parallel machine needs to be aware of the interconnection architecture between processing units. The simplest way of space sharing is fixed partitioning. Fixed partitions are set by the system administrators. Certain partitions can be dedicated to a certain group of users or different job classes[47, 52]. Many commercial systems split the system into two partitions: one for interactive jobs and the other for batch jobs. This is because the responsiveness of interactive jobs shouldn't be compromised by a heavy load of batch jobs. In spite of its simplicity, this system will suffer from internal fragmentation.

Variable partitioning can change the partition size based on jobs' requests. Nonetheless, external fragmentation remains an issue because the number of free processors left might not be enough for any queued jobs. Another scheduling decision is which job in the queue should be executed next. Obviously, first-come-first-service will introduce more external fragmentation. The "shortest job first" can give better average response time, but starve long jobs. In addition, it is not easy to know the lifetime of submitted jobs in advance. Other policies such as "smaller job first" and the opposite, "longer job first" were explored and turned out to be not much better than a simple FCFS scheduling [41]. Backfilling [28] is another scheduling algorithm to reduce external fragmentation while still offering fairness to queued jobs with respect to their arrival time.

The flexibility of applications can increase opportunities for the scheduler to perform global optimization. Many parallel programs can be written so that they can run on different numbers of processors. However, this does not necessarily mean that they can adapt the partition size at run time. For these parallel jobs, the scheduler can decide the number of processors to be allocated. Fairness plays a more important role here since the decision is made mostly by the scheduler with little application involvement. Various on-line algorithms have been suggested. Equipartition [61] allocates the same number of processors to all queued jobs. However, since not all "malleable" jobs can change the level of parallelism at run time, jobs will not have the same number of processors as the jobs enter and leave the system. Equipartition works as it is intended when all jobs are "moldable" (i.e., they can adapt the partition size at run time). However, too frequent reconfiguration should be avoided to limit overhead. For dynamic partitioning with moldable parallel jobs, two different schemes have been proposed:

the two-level scheduler [74] designed at the University of Washington and the “process control” scheduler [69] designed at Stanford University. Zahorjan et al [54] measured the job efficiency of different processor allocations and found the best configurations yielding maximum speedup at run time. This self-tuning is based on the fact that the speedup stops increasing after a certain size of the partition due to a heavy communication overhead. Studies have shown that memory constraints should also be considered since it can put a lower bound on the partition size [45] and below this lower bound, using virtual memory unacceptably degrades the parallel job performance due to heavy paging [60].

The scheduling policies surveyed above were originally developed for dedicated parallel systems. Thus, they cannot be directly applied to the network of workstation environment where interactive jobs require a quick response time. In most such systems, local interactive processes are not controllable and should be protected from aggressive background jobs. Parallel jobs are much more difficult to run on idle machines than sequential jobs because the suspension of one constituent process can block the whole parallel job resulting in poor system usage.

There have been many studies on running parallel jobs on non-dedicated workstation pools. Pruyne and Livny [58] interfaced the Condor system and PVM [31] through CARMI(Condor Application Resource Management Interface) to support parallel programming. Their Work Distributor helped the parallel job adapt as the resources came and went. The MIST [20] project also extended PVM to use only idle machines. The distinction between the two systems is that CARMI requires master-workers style programming and the inclusion and exclusion of machines is handled by creation and deletion of new worker processes, whereas MIST migrates running PVM processes. Piranha [19] works similarly, but it is restricted to programs using Linda [32] tuple-space based communication, whereas CARMI and MIST can serve in a general message passing environment. Cilk-NOW [16] also supports this adaptive parallelism for parallel programs written in Cilk. When a given workstation is not being used by its owner, the workstation automatically joins in and helps out with the execution of a Cilk program. When the owner returns to work, the machine automatically retreats from the Cilk program. Hector [59] and Cocheck [64] support checkpointing and migration for parallel programs written with MPI.

The NOW project [5] investigates various aspects of running parallel jobs on a network of workstations. They developed the River system [8] that supports I/O intensive parallel applications, such as external sort, running on dynamically changing resources. Their load balancing scheme, using a distributed queue and data replication, removes the drastic performance degradation of I/O intensive parallel applications due to the reduced I/O bandwidth on some nodes. Another study [7] in the NOW project investigated, through simulation, running parallel jobs and interactive sequential jobs of local users. This showed that a non-dedicated NOW cluseter of 60 machines can sustain a 32-node parallel workload. Acharya et al [2] also studied running adaptive parallel jobs on non-dedicated workstations. Their trace-driven simulations confirmed NOW's 2:1 ratio of nodes to effective parallel partition size when running parallel jobs. They also showed that the parallel job throughput depends on the flexibility of adaptive parallel jobs. Restricting the possible number of constituent processes to a certain number, like power of two, would yield a poor performance since all the available nodes are not used. To run data-parallel programs in an adaptive environment, Edjlali et al [25] developed a runtime library that redistributes data and determine new loop bounds and communication patterns when the number of processors changes.

2.3 Meta-computing and Adaptive Applications

Combining multiple autonomous computing facilities into a uniform platform through a high-speed network can serve more complex problems in shorter time. Such systems are called meta-computers [29] or a computational grid [30]. Meta-computing enables more users to exploit supercomputers, workstation clusters, archival storage services and many other computing resources distributed over the world. Globus [29] and Legion [43] are major projects to build an infrastructure for meta-computing. They are trying to address various requirements to seamlessly aggregate heterogeneous computing resources. The required services include global naming, resource location and allocation, authorization, and communications. The Prospero Resource Manager(PRM) [53] also provides uniform resource access to nodes in different administration domains so that users don't have to manage them manually.

From the application's view, it is advantageous for the application to adjust itself to changing resource status since the application knows more than the underlying resource manager how to obtain good performance from different resources. Based upon this concept, the AppLeS [14] project developed application-centric scheduling. In this system, applications are informed of resource changes and provided with a list of available resource sets. Then, each application allocates resources based upon a customized scheduling to maximize its own performance. This is different from most other systems, which strive to enhance system-wide throughput or resource usage. The Network

Weather Service [72] is used to forecast the network performance and available CPU to AppLeS agents so that applications can adapt. Dome [6] is another parallel programming model which supports application-level adaptation using load balancing and checkpointing. While the load balancing for the different CPU speeds and network performance is transparent, the programmers are responsible for writing suitable checkpointing codes using provided interfaces.

The Active Harmony system [39] also provides dynamic reconfiguration of ongoing computations to adapt to changing resources conditions. Possible runtime options include alternative algorithms, changing the level of parallelism for parallel applications and object migrations for data affinity. Applications first export their alternatives and metrics through a well-defined specification language (Harmony RSL). During the application execution, the Harmony middle-ware evaluates given metrics against current system conditions and reconfigures the application with the best tuning option. Our Linger-Longer system is a part of this project and taps unused resources in non-dedicated machine pools to serve such applications. The adaptive migration scheme in this thesis also follows the base concept of dynamic reconfiguration in the Harmony system.

2.4 Local Scheduling for Resource Limiting

Fine-grain cycle stealing requires mechanisms to allow guest jobs to use most of the unused CPU cycles, memory, I/O and network bandwidth while minimizing contention for such resources with local host jobs. Therefore, the desired mechanisms should be capable of 1) specifying different job classes, an *unconstrained* class for host jobs and a *constrained* class for guest jobs, 2) setting the limits for the major resources including CPU, memory, disk I/O and network bandwidth, 3) activating such limits only when there is resource contention between constrained and unconstrained jobs and 4) extending current operating systems without compromising existing algorithms and their efficiency.

The Stealth Distributed Scheduler [40] supports a constrained job class using a local scheduler which protects the performance of an owner's processes. They prioritized not only CPU scheduling but also memory management and file accesses. However, the mechanisms in Stealth are hard-wired into the Mach 2.5 operating system.

Verghese et al [71] investigated dynamic sharing of multi-user multiprocessor systems. Their primary scheme is to isolate the performance of the processes belonging to the same logical entity, such as a user. Logical smaller machines named Software Performance Units(SPU) were suggested to achieve two performance goals: 1) Isolation: If the allocated resources meet the requirement of an SPU, its performance should not be degraded by the load placed on the systems by others, and 2) Sharing: If the allocated resources are less than the requirement, the SPU should be able to improve its performance by utilizing idle resources. Like Stealth, only CPU idle time and unused memory can be loaned to non-host SPUs and will be revoked immediately when the host SPU needs them. Their system was implemented in the Silicon Graphics IRIX operating system.

Having logical smaller machines in Verghese's study is similar to the classic virtual machine concept developed by IBM machines [48]. However, the concept of virtual machines on a physical machine was developed mainly to provide an illusion of having multiple machines and to permit running different operating systems on a single machine. The system efficiency can be limited due to internal fragmentation resulting from lack of support for dynamic resource sharing between virtual machines. Therefore, these techniques are not appropriate for implementing the low resource priority in fine-grain cycle stealing.

The idea of resource partitioning through the use of virtual machines has been popular both in the 1970s [33] as well as in recent projects such as Disco [18]. The key difference is that while virtual machines provide hard isolation of resources between VMs at considerable runtime overhead, our approach is to have a simple and portable extension to an existing operating system or runtime library.

In the current version of the IRIX operating system [62], the Miser feature provides deterministic scheduling of batch jobs. Miser manages a set of resources, including logical CPUs and physical memory, that Miser batch jobs can reserve and use in preference to interactive jobs. This strategy is almost the opposite of our approach, which promotes interactive host jobs.

Many have studied general quality of service (QoS) support for server applications. The reservation domains of Eclipse [17], and Resource Containers [11] can group a set of processes or threads as a unit for resource scheduling. They are both capable of setting a different of resources for different job classes. Again, those systems are integrated deep into the kernel and heavy-weight because they are intended for a general use for QoS. Particularly for network resources, the idea of regulating traffic rates has been extensively studied. Congestion avoidance schemes such as leaky bucket [70] and its variants [26, 76] use averages over various time intervals to determine which traffic is within its negotiated bandwidth. However, these approaches are designed for policing traffic at routers while we need rate limiting at end nodes.

Chapter 3

Fine-Grain Cycle Stealing & Migration

We use the term cycle stealing to mean running jobs that don't belong to the workstation's owner (guest jobs). The idle cycles of machines can be defined at different levels. Traditionally, studies have investigated using machines only when they are not in use by the owner. Thus, the machine state can be divided into two states: idle and non-idle. In addition to processor utilization, user interaction such as keyboard and mouse activity has been used to detect if the owner is actively using their machine. Acha [1] showed that for their definition of idleness, machines are in a non-idle state for 50% of the time. However, even while the machine is in use by the owner, substantial resources are available to run other jobs.

In this chapter, we first investigate how much resources are available in networks of workstations. By analyzing two large sets of trace data, we demonstrate that a significant amount of resources are still available in non-idle machines. Then, we introduce our new policy, fine-grain cycle stealing to exploit more idle cycles without a noticeable impact on machine owners. Also, a new adaptive migration scheme based on runtime cost/benefit analysis is described.

3.1 Available Idle Resources

The thesis of this dissertation is predicated on the idea that a significant amount of resources are still available even when the owners use their machines. The Linger-Longer approach was suggested to collect those fine-grained idle cycles without a noticeable interference. Before proceeding, it is worth investigating how much CPU time and memory is really available from the machines that are in use. We first analyzed trace data gathered from 132 machines measured over 40 days. This trace data contains samples taken every two seconds of: CPU usage, memory usage, keyboard activity, and a Boolean indicating idle/non-idle state. As in the NOW project [44], an idle interval is defined as a period of time with the CPU less than 10% used and no keyboard action for one minute.

First, we measured the overall CPU utilization and compared it to the CPU utilization during idle and non-idle intervals. The graph in Figure 1 shows the Cumulative Distribution Function (CDF) for processor utilization. The solid line shows the overall utilization from the traces, and the two dashed lines show the utilization for idle and non-idle time. Even non-idle intervals have very low usage, although it is somewhat higher than idle time. While 46% of the time a machine is in a non-idle state, 76% of the time in non-idle intervals, the processor utilization is less than 10%. The reason that

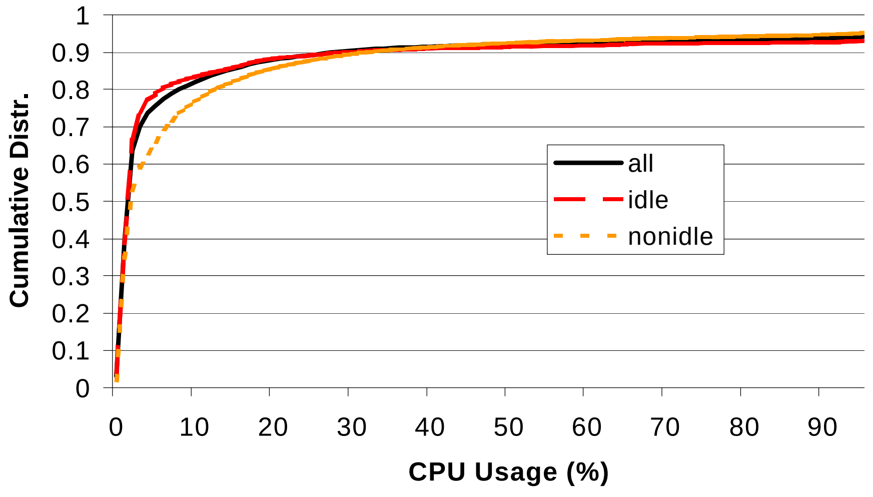


Figure 1: Distribution of overall processor utilization

The graph shows the processor utilization for all time. The solid line is for all states, and the two dashed lines show the utilization of idle and non-idle nodes.

these intervals of time are considered non-idle is due to keyboard activity and the requirement that a workstation have low utilization at least for 1 minute to be considered idle. This data hints at the potential leverage for a Linger-Longer approach to use short idle periods.

Several other interesting trends exist in the coarse-grained cluster data. On the whole, 82% of the time, processor utilization is less than 10%. In addition, moderate processor utilization is rare. Only 13% of the time is the utilization between 10% and 80%. However, high utilization is somewhat more common (utilization of 80% or more occurs 7% of the time). The basic conclusion of this data is that except for rare instances of heavy use, processors on these workstations had significant available capacity over 90% of the time.

We now consider the time of day variation in processor utilization. Conventional wisdom holds that there should be a significant increase in utilization during working hours. However, analysis of the cluster data shows that there is little variation in processor utilization by time of day. The sharp cliff in the graph shows that most of the time the processor utilization is low. Figure 2 shows the utilization CDF plotted versus the time of day. The peaks and valleys along the plateau show the time of day variation in utilization. It shows little correlation with time of day, and most of the time the processor is between zero and fifteen percent utilized.

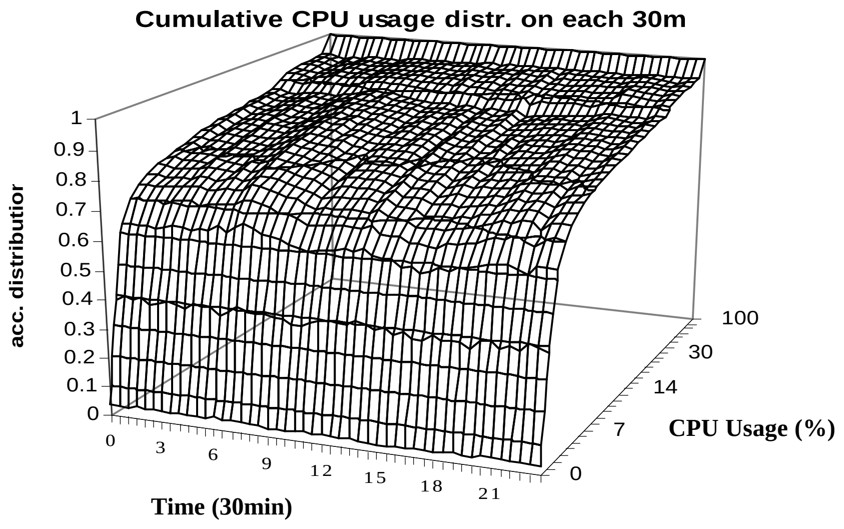


Figure 2: Processor utilization as a function of the time of day

This graph shows processor utilization as a function of the time of day. The sheer face of the surface indicates that most of the time nodes are lightly loaded.

To meet our goal of allowing guest jobs to linger on a workstation and at the same time not to interfere with host jobs, we need to ensure that enough real memory is available to accommodate the guest job. As an approximation of the available local memory, we analyzed the same workstation trace data used to evaluate processor availability to estimate available free memory. Each workstation has 64 megabytes of main memory. The probability of available memory is shown in Figure 3. The y-axis shows the fraction of time that at least x Mbytes of memory are available. This graph demonstrates that 90% of time, more than 14 megabytes of memory is available for guest jobs, and that 95% of the time at least 10 Mbytes of memory is available. Interestingly, there is no significant difference in the available memory between idle and non-idle

[1] To implement this, we have added priority to the Linux paging mechanism.

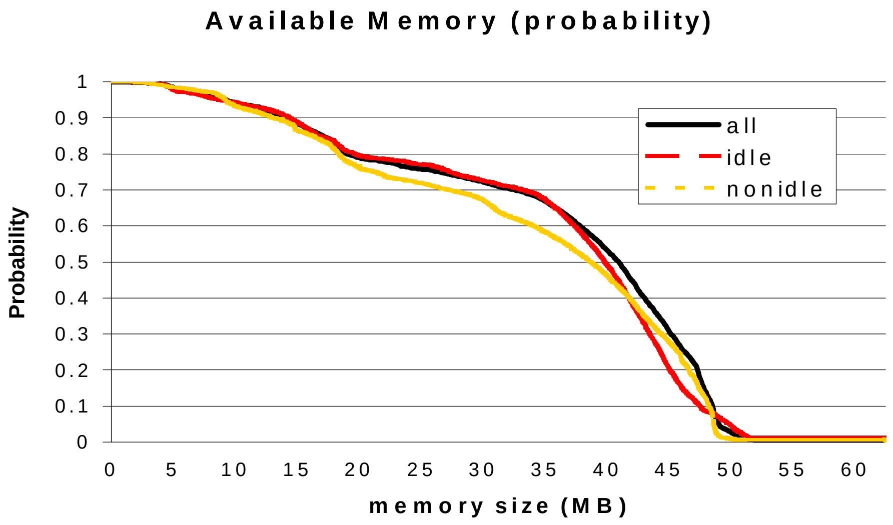


Figure 3: Distribution of available memory

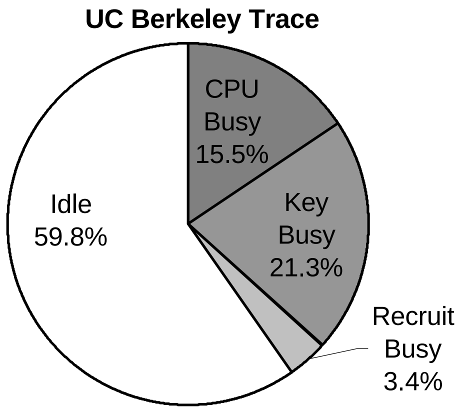
The solid line shows the overall free memory and the two dashed lines show the free memory during idle and non-idle intervals. The y-axis shows the fraction of time that at least x *MB of memory are available. Each workstation has 64 Mbyte main memory.*

states.[2] We feel that the amount of free memory generally available is sufficient to accommodate one compute-bound guest job of moderate size.

We also looked into other traces collected by Acharya, et al [2, 3]. They comprise a trace collected from a Condor pool having 310 machines at the University of Wisconsin. The traces were generated for two weeks in September 1997.

First, we investigated the traces to know how much of the non-idle state of the machines were contributed respectively by three different factors: high CPU load (CPU busy), keyboard and mouse activities (keyboard busy) and the recruitment threshold (recruit busy). The results are summarized in Figure 4. The traditional idle cycle harvesting systems don't immediately use the machines that have just become idle.

[2] One possible explanation for this is that current versions of the UNIX operating system employ an aggressive policy to maintain a large free list.



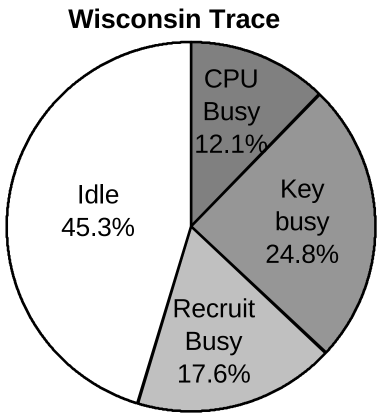


Figure 4: Non-idle time and its causes

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

Rather, they wait a fixed amount of time to make sure the machine will not immediately return to the CPU busy state. This interval of time is called the recruitment threshold. The machine state of the UC Berkeley trace were determined by the idleness definition of the NOW system. For the Wisconsin trace, the default Condor threshold values were used: 15 minutes for recruitment threshold and 1 minute average CPU load 0.3. As shown in Figure 4, 21.3% of the time the machines were keyboard busy in the UC Berkeley trace. The Condor trace showed similar results: 24.8% of the time the workstations were keyboard-busy. However, because the default condor recruitment threshold of 15 minutes is much longer than that of the NOW system, about 14% more busy time was introduced. This busy time is a good source for our fine-grain cycle stealing since its resource availability is the same as idle machines.

We also looked at the difference in CPU load when users are active and not. Unfortunately, the Condor trace has no CPU usage data. Rather it tracks CPU load average, which is the average run queue length. The CPU load during keyboard-busy time intervals was not significantly higher than that during the idle intervals: 0.25 for keyboard-busy time and 0.20 for keyboard-idle time. These results are similar to the CPU usage distribution of the UC Berkeley trace in Figure 1. In addition to keyboard/mouse activity, we use the CPU load information to define the idleness of machines. Using a load average threshold of 0.3 as busy for the Condor traces results in an average machine idle rate of 45.3%. Although the data gathered was somewhat different, the resulting idle and non-idle time was similar.

For memory availability, the result of the UC Berkeley trace is backed up by a recent study by Acharya, et al [3]. With the traces from various institutions they showed that most of the time, more than a half of main memory is unused and a larger portion of memory is available on the machines with larger main memory.

In this section, we demonstrated that a significant amount of resources including CPU cycles and memory are available even for machines declared non-idle. My thesis is that the Linger Longer system can harness most of these extra resources to run guest processes without a noticeable slowdown of the owner's local workload.

3.2 Fine-Grain Cycle Stealing

We introduce a new technique to make more idle time available. In terms of CPU utilization there are long idle intervals when the processor is waiting for user input, I/O, or the network. These intervals between run bursts by owners' jobs can be made available to others' jobs. We term running guest jobs, while the user processes are active, lingering. Since the owner has priority over guest jobs using their personal machine, use of these idle intervals should not affect the performance of the owner's jobs (host jobs).

A typical pattern of idle and non-idle periods is illustrated in Figure 5. Traditional coarse-grain cycle harvesting systems (e.g., Condor and NOW) determine machine

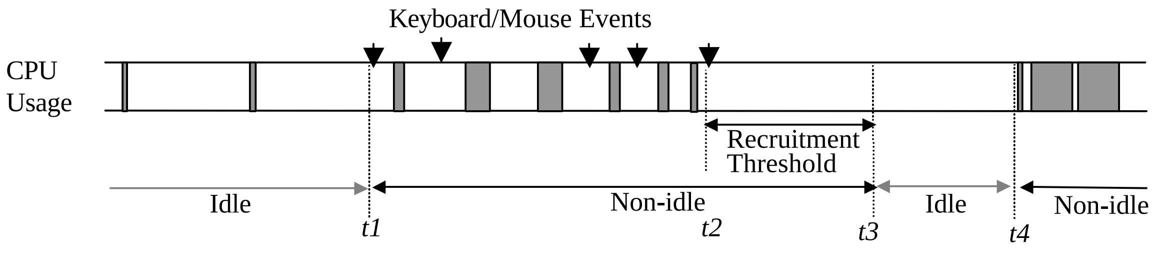


Figure 5: Idle cycles on a non-dedicated machine

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

states using three factors: CPU usage, keyboard/mouse activities and recruitment threshold. As mentioned in earlier chapters, a recruitment threshold is the time to be waited before idle machines become available. Thus, this interval can be included in non-idle periods. In Figure 5, coarse-grain non-idle periods are the intervals (t_0, t_1) and (t_3, t_4). The non-idle interval (t_1, t_2) was mostly caused by keyboard/mouse activities whereas the second non-idle interval (t_4, ~) was due to high CPU usage. The trace analysis in the prior section demonstrated that the first case is more typical and hence the CPU usage is not high. The empty spaces in the CPU usage bar are fine-grain idle cycles. Our thesis is that we can exploit much more idle time since the sum of fine-grain idle cycles is significantly larger than that of coarse-grain idle periods.

However, delay of host jobs should be avoided when guest jobs are using idle cycles from non-idle machines. If not, users will not permit their workstations to participate in the pool. Priority scheduling is a simple way to enforce this policy. Current operating systems schedule processes based on their priority, and use a complex dynamic priority allocation algorithm for efficiency and fairness. To implement lingering, we need a somewhat stronger definition of priority for host and guest job classes. Host processes have the highest priority and can starve background guest processes. In addition, when a guest process is running, an interrupt that results in a host process becoming runnable, causes the host process to be preemptively scheduled onto the processor even if the guest job's scheduling quanta has not expired.

Preemptive prioritized CPU scheduling can be sufficient for CPU intensive guest jobs using very little memory. However, host jobs can be delayed significantly by the contention for other resources such as physical memory, disk I/O and network bandwidth. These issues will be addressed in Chapter 5.

3.3 Adaptive Migration

In coarse-grain cycle stealing, guest jobs should migrate off a node when a machine owner returns. Two strategies have been used in the past to migrate guest jobs: Immediate-Eviction and Pause-and-Migrate. In *Immediate-Eviction* (NOW [5]), the guest job is migrated as soon as the machine becomes non-idle. Because this can cause unnecessary, expensive migrations for short non-idle intervals, an alternative policy, called *Pause-and-Migrate* (Condor [44]) which suspends the guest processes for a fixed time prior to migration, is often used. The fixed suspend time should not be long because the guest job makes no progress in the suspend state. With Linger-Longer scheduling, guest jobs can run even while the machine is in use by the owner; therefore migration becomes an optional move to a machine with lower utilization rather than a necessity to avoid interference with the owner's jobs. Although migration can increase a guest

e is a cost to move a process' state. Also, the advantage of running on the idle machine depends on the difference in available processor time between the idle machines and current non-idle one. To maximize processor time available to a guest job, we need a policy that determines the lingering duration.

When to migrate in a Linger-Longer scheduler depends on the local CPU utilization on the source and destination nodes, the duration of the non-idle state and the migration cost. The question is when will the guest job benefit from migration. Given the local CPU utilization and migration cost, the minimum duration of non-idle interval (called an episode) before migration is advantageous can be computed. Any idle period shorter than the minimum duration will not provoke a migration. We can compute the minimum duration by comparing the two timing diagrams in Figure 6. In the non-idle state, utilization by the workstation owner starts at t_1 and ends at t_4. The average utilization of the non-idle node is h, and the average utilization on an idle node is l. We assume the execution time of the guest job exceeds the duration of the non-idle state, so the guest job completion time t_{f1} comes after t_4. Migration happens at t_2, and the cost is T_{migr}. The following equations compute the total job CPU time $T_{C,M}$ and $T_{C,S}$ with and without migration respectively.

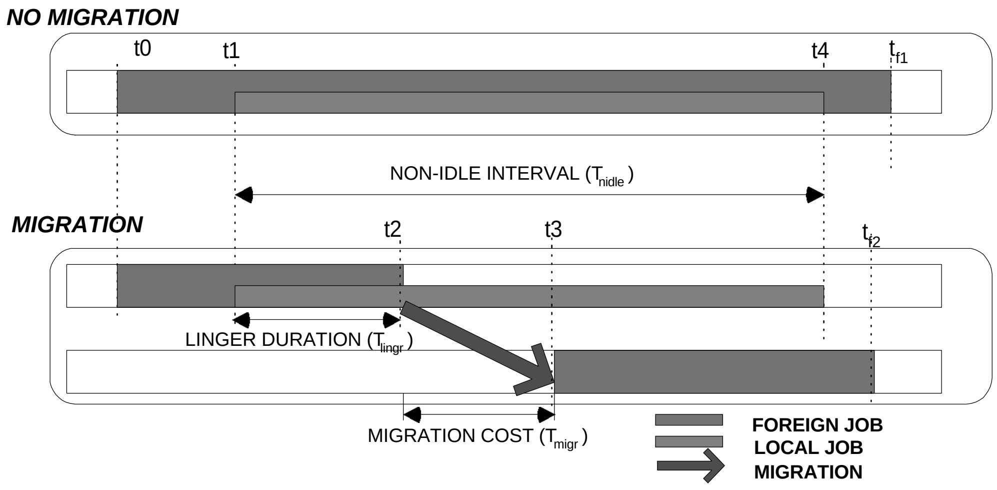


Figure 6: Migration timing in Linger-Longer

The timeline for migration using Linger-Longer scheduling. The top case shows a guest job that remains on a node throughout an episode of processor activity due to host jobs. The lower case shows migration after an initial linger interval (t_1 to t_2) where the guest job remained on the non-idle node.

$$\begin{aligned} T_{C,S} &= (1-l)\cdot(t_1-t_0)+(1-h)\cdot(t_4-t_1)+(1-l)\cdot(t_{f1}-t_4) \\ T_{C,M} &= (1-l)\cdot(t_1-t_0)+(1-h)\cdot(t_2-t_1)+(1-l)\cdot(t_{f2}-t_3) \end{aligned} \qquad (3.1)$$

Since the same amount of work should be done for both cases, $T_{C,S} = T_{C,M}$. We can solve the relationship between parameters.

$$t_{f2}-t_{f1}=(t_4-t_2)\cdot\frac{1-h}{1-l}-(t_4-t_3) \qquad (3.2)$$

And, to get benefit from the migration, $t_{f2} <= t_{f1}$. We can then express it with interval variables as:

$$T_{nidle} \geq T_{lingr} + \left(\frac{1-l}{h-l}\right)\cdot T_{migr} \qquad (3.3)$$

where $T_{nidle} = t_4 - t_1$ is the non-idle state duration, $T_{lingr} = t_2 - t_1$ is the lingering duration and the migration cost is denoted as T_{migr}. If we knew the non-idle state would last long enough to make migration advantageous, an immediate migration would be the best choice. But because we don't know when the non-idle state will end, we have to predict it. We use the observations of Harchol-Balter and Downey [35], and Leland and Ott [42], which state that the median remaining life of a process is equal to its current age. So if a process has run for T units of time, we predict its total running time will be 2T. Our use of this predictor is somewhat different since we use it to infer the duration of a non-idle episode rather than predict process lifetime. With this prediction, we can then compute the Linger duration by letting T_{nidle} be $2T_{lingr}$. If it is expected that the migration will benefit, it's better to migrate early. The lingering duration T_{lingr} will be:

$$T_{lingr} = \left(\frac{1-l}{h-l}\right)\cdot T_{migr} \qquad (3.4)$$

So, the guest job should linger T_{lingr} before migrating. For a non-idle interval shorter than T_{lingr} migration will be avoided. Compared to the Pause-and-Migrate (PM) policy, the “pause” period is determined dynamically depending on several factors including the migration cost. Furthermore, the guest job can continue to run (with lower priority) unlike PM, which suspends the job. The migration cost is the time from when the process stops at the source node to when it resumes running at the destination node. The cost consists of a fixed part and variable part. The fixed part is for handling the process-related suspension and resumption at the source and destination nodes respectively. The process transfer time depends on the network bandwidth and the process size. The following equation is used for our experiments, and it can be easily extended for different environments.

$$T_{migr} = suspend(source) + process\_size / network\_bandwidth + resume(dest.) \qquad (3.5)$$

We denote the policy of lingering on a node for T_{lingr} before migration, LL (Linger-Longer). An alternative strategy of never leaving a node (called Linger-Forever), denoted LF, is also considered. This policy attempts to maximize the overall throughput of a cluster at the expense of the response time of those unfortunate guest jobs that land (and are stuck) on nodes with high local utilization.

In this chapter, we defined coarse-grain idle periods and fine-grain idles cycles. First, with trace data analysis, we demonstrated that the aggregate amount of fine-grain idle cycles in non-idle machines are significant. Second, we suggested that lowering guest jobs’ resource priorities to starvation-level would allow fine-grain cycle stealing without noticeable impact on host processes. Third, a new migration scheme was introduced. We derived a condition where guest jobs can benefit from migrations. In this technique, when to migrate is determined based on the guest job and underlying resource information such as guest job size, current local CPU load and available network bandwidth.

Chapter 4

Simulation Study

Before full implementation, we conducted simulations since they provide a controlled environment that permits investigations of parameters and estimations of overall performance. First, we evaluate the impact of the priority-based linger mechanism on the node's host jobs, and the performance of our adaptive migration scheme. Then, by simulating 64 machines, we compare two variations of our policy: Linger Longer and Linger Forever, to two previous policies: Immediate-Eviction and Pause-and-Migrate. We also investigate the performance gain of running guest parallel applications with Linger-Longer.

4.1 Linger-Longer Scheduling and Adaptive Migration

To understand the behavior of a Linger-Longer scheduling discipline we need to evaluate the impact of the priority-based linger mechanism on the node's host jobs. Using a priority-based scheduler will cause a foreground job that moves from the blocked to the runnable state to be scheduled immediately. However, this potentially requires an additional context switch to save the state of the guest job and load the state of the host job. In this section we present a simulation study of the delay induced in a host process by a lingering guest job.

A key question to evaluating the overhead of priority-based preemption is the time required to switch from the guest job to the host one. There are two significant sources of delay in saving and restoring the context of a process. The first component is due to the time required to save the state of the registers used by one process and load the registers in use by the other process. The second component is the time spent (handling caches misses) to reload the process' cache state. On current microprocessors, the time to restore cache state dominates the register restore time. We term the combination of these two components the effective context switch time. The third component is to restore virtual memory pages into physical memory. This overhead can be several orders of magnitude larger than the other two. We assume a memory replacement policy in which guest jobs can use only free memory, and therefore will not page out the memory pages of host processes. So, virtual memory paging cost is not included in the effective context switch time. The mechanisms for this prioritized use of memory will be described in Chapter 5.

To estimate the effective context switch time, we use the results obtained by Mogul and Borg [49] concerning the effects of context switches on cache performance. They simulated the impact of context switches in inducing additional cache misses for various UNIX-style workloads. One processor they evaluated had a split 8KB L1 cache and a unified 2MB L2 cache (with cache miss penalties of 13 and 200 cycles, respectively). This configuration is similar to current microprocessors. For this configuration, the largest average cache delay due to a context switch was 26,300 cycles. For a current processor with a 300 Mhz clock, this delay corresponds to 87 microseconds. Based on this information, we believe that a reasonable, somewhat conservative, *effective context-switch time* is 100 microseconds. It is worth noting that the only time a guest job would be executed is after all host jobs had voluntarily yielded the processor (i.e., blocked for events). Mogul and Borg found that the "live" cache state after such voluntary context switches was significantly less than at preemption, and so our estimate should be conservative. However, despite this study, it is possible that some guest jobs will introduce significant additional cache misses. To evaluate the impact of this possibility, we also simulated effective context switch times of 300 and 500 microseconds.

To evaluate the behavior of Linger-Longer we simulated a single node with a single compute bound (always runnable) process and various levels of processor utilization by foreground jobs. For each simulation, we computed two metrics: the local job delay ratio (LDR) and fine-grain cycle stealing ratio (FCSR). The LDR metric records the average slowdown experienced by host jobs due to the extra context switch delay introduced by background jobs. The FCSR metric records the fraction of the available idle processor cycles that are used by the guest job.

Figure 7 and Figure 8 show the LDR and FCSR metrics respectively for three different effective context switch times at various level of processor utilization by host jobs. For the chosen effective context switch time of 100 microseconds, the delay seen

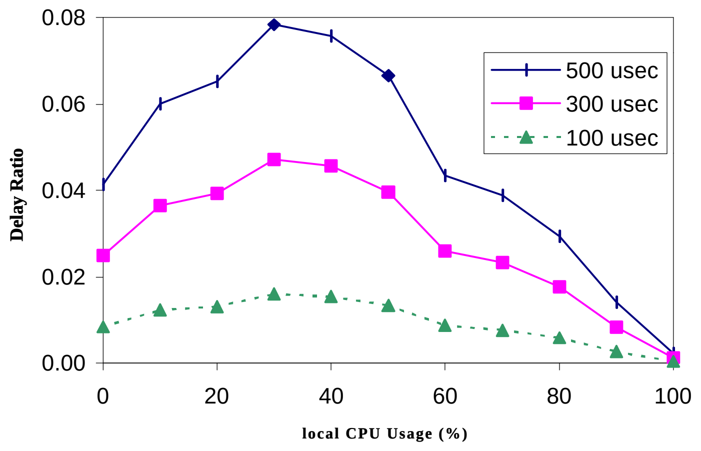


Figure 7: Local job Delay Ratio (LDR)

Each curve shows the impact of three different effective context switch times (100, 300, and 500 microseconds). The graph shows the delay experienced by foreground jobs at various levels of utilization.

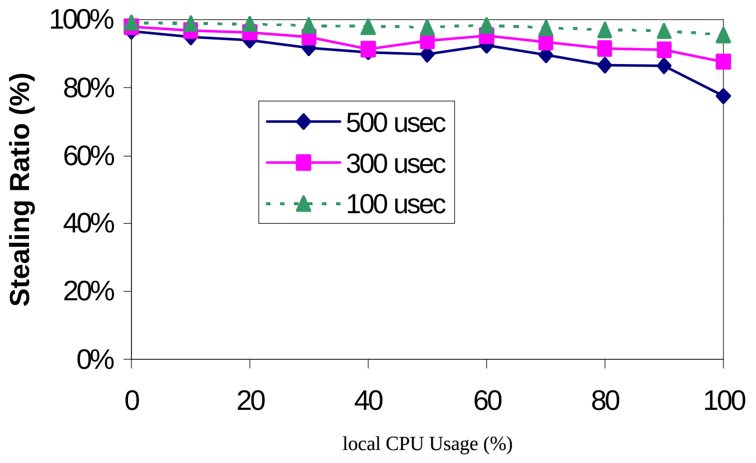


Figure 8: Fine-grain Cycle Stealing Ratio (FCSR)

The curves show the percent of the available *idle processor time made available to a compute bound guest job at different levels of host job processor utilization.*

by the application process is about 1%. For context switch times up to 300 microseconds, the delay remains under 5%. However, when the effective context switch time is 500 microseconds, the overhead is 8%. In all of these cases, a guest job was able to make productive use of over 90% of the available processor idle cycles.

We next consider the impact of making choices about when to migrate a task to a less loaded node. If there are free nodes available, it is best to try to migrate a guest job onto that node. As described before, we dynamically decide when to move a process based on four parameters: migration cost, local CPU utilization, predicted duration of the local activity, and predicted duration of the guest job. Migration cost is primarily a function of the memory size of the guest job. For this simulation, we assume that all guest jobs are 8 Megabytes, migration takes places over a 10 Mbps Ethernet at an effective rate of 3Mbps (to limit the load placed on the network by process migration), and that the guest job is suspended for the entire duration of the migration. In addition, we assume that the processor on both the source and destination nodes must be used to migrate the process.

Figure 9 shows a comparison of the completion time of a hypothetical 50-second guest job that experiences a local processor utilization of 20% when the workstation's owner returns. The x-axis is the duration of episode of the local user's return to their workstation. The diamond marked line reports the completion time of the Linger-Longer policy and the box marked line the completion time of the immediate eviction policy. The results show that if the workstation is non-idle for a period of time less than 30 seconds, the Linger-Longer policy will finish the job before the Immediate-Eviction policy. However, if the workstation is non-idle for more than 30 seconds, the immediate eviction policy will finish the job sooner since it will migrate the job to a free node immediately. These results describe the response time of a single sequential node when a single local user returns. They are intended to explain the local behavior of lingering.

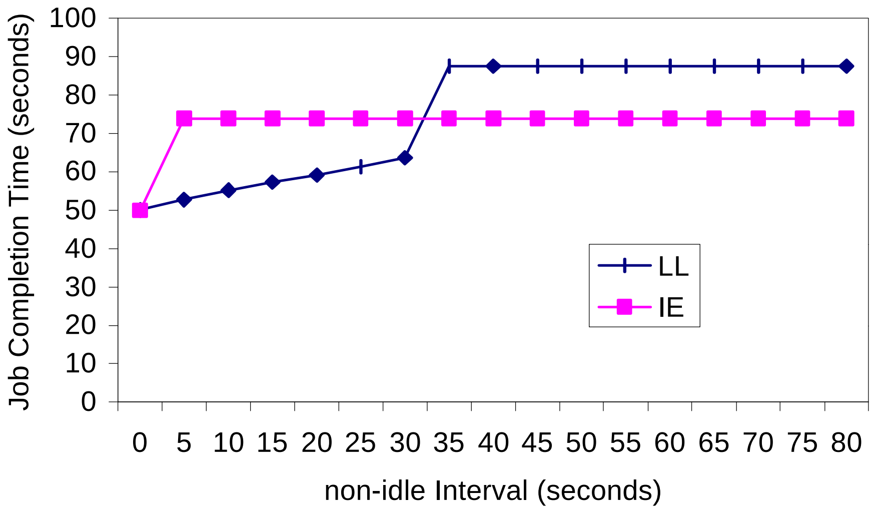


Figure 9: Migration timing

Completion time of a 50 second guest job using Linger-Longer (LL) and Immediate-Eviction (IE) as a function of the episode of a workstation owner returning.

Linger-Longer not only provides fine-grain cycle stealing, but also reduces unnecessary migrations. An overall simulation of a full cluster is presented in the following section.

4.2 Sequential Jobs in a Cluster

We now turn our attention to the cluster-level behavior of our scheduling policy. We first evaluate the behavior of a cluster running a collection of sequential jobs. We evaluated the Linger-Longer, Linger-Forever, Immediate-Eviction, and Pause-and-Migrate policies on a simulated cluster of workstations.

To evaluate traditional eviction-based policies, it is sufficient to consider coarse-grained metrics of utilization (on the time scale of seconds). However, because of the fine-grained interaction between host and guest processes when using linger-based policies, we need data about individual requests for processors, at the granularity of scheduler dispatch records (on the time scale of microseconds), to evaluate it.

It is not practical to record fine-grained requests for the long time periods required to capture the time of day and day of week changes in free workstations. As a result, we developed a two-level workload generator which combines trace data and a stochastic model. First, we measure the fine-grained run-idle bursts at various levels of processor utilization from 0% (idle) to 100% (full) utilization. We model a fine-grained workload as a random variable that is parameterized by the average utilization over a two-second window. This permits using a coarse-grained trace of workstation utilization to generate fine-grained requests for the processor. The detailed modeling of fine-grain CPU usage is described in Appendix A.

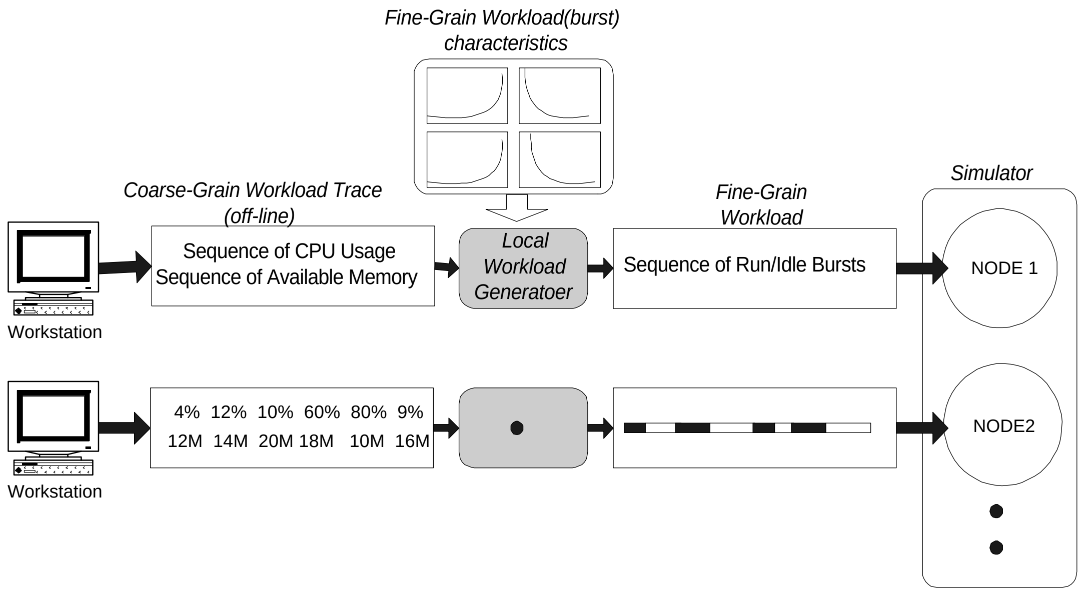

Fine-Grain Workload(burst) characteristics
Coarse-Grain Workload Trace (off-line)
Sequence of CPU Usage
Sequence of Available Memory
Local Workload Generatoer
Fine-Grain Workload
Sequence of Run/Idle Bursts
Simulator
NODE 1
NODE2
Workstation
Workstation
4% 12% 10% 60% 80% 9%
12M 14M 20M 18M 10M 16M

Figure 10: Local workload generation

The process of generating long-term processor utilization requests. By combining coarse-grained traces of workstation use with a short-term stochastic model of processor requests, long duration run-idle intervals can be generated.

We used this two-level workload generator to produce a local user workload for a 64-node cluster. Figure 10 shows the process that we use to generate fine-grained processor requests from long-term trace data. We randomly select a trace of a single node and map it to a logical node in our simulation. To draw a representative sample of jobs from different times of the day, each node in the simulation was started at a randomly selected offset into a different machine trace. The fine-grain resource usage is generated by a stochastic model using appropriate parameters, mean and variance, based on the current coarse-grain resource data from the trace files.

We then ran two different types of sequential guest jobs on the cluster. Workload-1 contains 128 guest jobs each requiring 600 processor seconds. This workload was designed to represent a cluster with a significant demand being placed on the guest job scheduler, since on average each node had two guest jobs to execute. Workload-2 contains 16 jobs each requiring 1,800 CPU seconds each. This workload was designed to simulate a somewhat lighter workload on the cluster since only ¼ of the nodes are required to run the guest jobs. All guest jobs are 8 Megabytes and migration takes places over a 10 Mbps Ethernet at an effective rate of 3Mbps. We also assume that the guest job is suspended for the entire duration of the migration. For each configuration, we computed four metrics:

Average completion time: The average time to completion of a guest job. This includes waiting time before initially being executed, paused time, and migration time.

Variation: the standard deviation of job execution time (time from first starting execution to completion).

Family Time: The completion time of the last job in the family of processes submitted as a group.

Throughput: The average amount of processor time used by guest jobs per second when the number of jobs in the system was held constant.

The results of the simulation are summarized in Table 1. For the first workload, the average job completion time and throughput are much better for the Linger-Longer and Linger-Forever policies. Average job completion time is 47% faster with Linger-

	Metric	**LL**	**LF**	**IE**	**PM**
Workload-1 (many jobs)	Avg. Job Time	1044	1026	1531	1531
	Variation	13.7%	20.5%	27.7%	22.5%
	Family Time	1847	1844	2616	2521
	Throughput	52.2	55.5	34.6	34.6
Workload-2 (few jobs)	Avg. Job Time	1859	1861	1860	1862
	Variation	0.9%	1.3%	1.3%	1.6%
	Family Time	1896	1925	1925	1956
	Throughput	15.0	14.7	14.5	14.5

Table 1: Performance of cluster scheduling policies

For each of the four scheduling policies (LL, LF, IE, and PM), four performance metrics are shown for two different workloads.

Longer than Immediate-Eviction or Pause-and-Migrate, and Linger-Forever's jobs completion time is 49% faster than either of the non-lingering policies. There is virtually no difference between IE and PM in terms of average completion time. For the second workload, the average job completion time of all four policies is almost identical. Notice that the average job completion time ranges from 1,859 to 1,860 seconds; this implies that on average they were running 97% of the time. Since there is sufficient idle capacity in the cluster to run these jobs, all four policies perform about the same.

In terms of the variation in response time for workload-1, the LL policy is much better than either IE or PM. This improvement results from LL's ability to run jobs on any semi-available node, and thus expedite their departure from the system; so the benefit of lingering on a non-idle node exceeds the advantage of waiting for a fully free node. The LF policy has a somewhat higher variation due to the fact that some jobs may end up on nodes that had temporarily low utilization when the job was placed there, but which subsequently had higher load. For workload-2, the availability of resources means that each policy has relatively little variation in its job completion time.

The third metric is "Family Time". This metric is designed to show the completion time of a family of sequential jobs that are submitted at once. This is a metric designed to characterize the responsiveness of a cluster to a collection of jobs that represent a family of jobs. For workload-1, the LL and LF metrics provide a 36% improvement over the PM policy and 42% improvement over the IE policy. For workload-2, the LL policy provides slight (1-3%) improvement over the IE and PM policies.

The fourth metric we computed for the cluster-level simulations was throughput. The throughput metric is designed to measure the ability of each scheduling policy to make processing time available to guest jobs. This metric is computed using a slightly different simulation configuration. In this case, we hold the number of jobs in the system (running or queued to run) constant for a simulated one-hour execution of the cluster. The number of jobs in the system is 128 for workload-1 and 16 for workload-2. The throughput metric is designed to show the steady-state behavior of each policy at delivering cycles to guest jobs. Using the throughput metric, the LL policy provides a 50% improvement over the PM policy. Likewise the LF policy permits a 60% improvement over the PM policy. For workload-2, the throughput was very similar for all policies. Again, this is to be expected since the cluster is lightly loaded. For both workloads the delay, measured as the average increase in completion time of a CPU request, for host (foreground) processes was less than 0.5%. This average is somewhat less than the 1% delay reported in the previous section since not all non-idle nodes have guest processes lingering.

To better understand the ability of Linger-Longer to improve average job completion time, we profiled the amount of time jobs spent in each possible state: queued, running, lingering (running on a non-idle node), paused, suspended and migrating. The results in Figure 11 show the behavior of workload-1. The major difference between the linger and non-linger policies is due to the reduced queue time. The time spent running (run time plus linger time) is somewhat larger for the linger policies, but the reduction in queuing delays more than offsets this increase. Figure 12 shows the break-

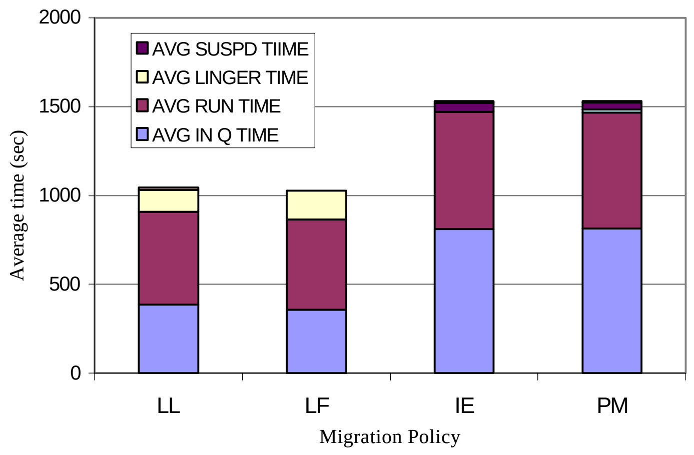


Figure 11: Average completion time for Workload-1

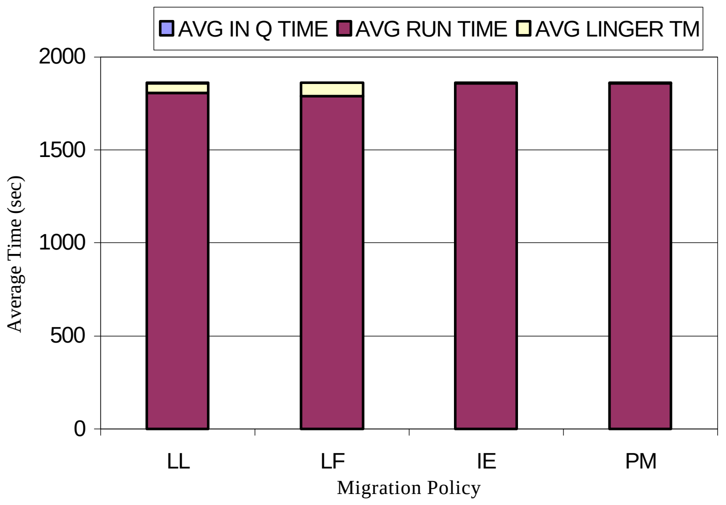


Figure 12: Average completion time for Workload-2

down for workload-2. With the exception that LL and LF spent a small fraction of the time lingering, there is no noticeable difference between any of these cases.

The overall trends in the cluster level simulation show that Linger-Longer and Linger-Forever provide significant increased performance to a cluster when there are more jobs than available nodes, and that there is no difference in performance at low levels of cluster utilization.

4.3 Synthetic Parallel Jobs

The trade-offs in using Linger-Longer scheduling for parallel programs are more complex. When a single process of a job is slowed down due to a host job running on the node, this can result in all of the nodes being slowed down due to synchronization between processes. On the other hand, when a migration is taking place, any attempt to communicate with the migrating process will be delayed until the migration has been completed. However, we feel the strongest argument for using Linger-Longer is the potential gain in the throughput of a cluster due to the ability to run more parallel jobs at once. Improved throughput likely will come at the expense of response time, but we feel that throughput is the most important performance metric for shared clusters. To evaluate these different options, we simulated various configurations to determine the impact of lingering on parallel jobs.

To evaluate the impact of lingering on a single parallel job, we first simulated a bulk-synchronous style of communication where each process computes serially for some period of time, and then an opening barrier is performed to start a communication phase. During the communication phase, each process can exchange messages with other processes. The communication phase ends with an optional barrier. This synthetic parallel job model has been successfully used in [24] to explore various performance factors. We simulated an eight-process application with 100 milliseconds between each synchronization phase, and a NEWS[3] style of message passing within a communication phase. The graph in Figure 13 shows the slowdown (compared to running on 8 idle nodes) experienced in the application's execution time when one node is non-idle

[3] A process exchanges messages only with its four neighbors.

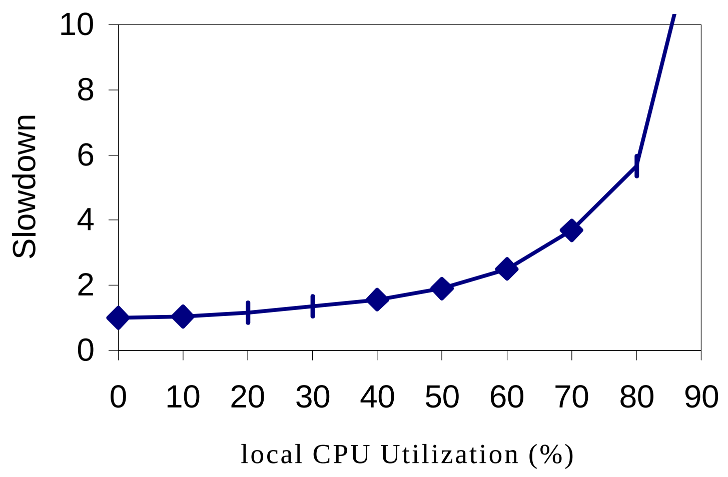


Figure 13: Parallel job slowdown vs. local utilizations

and the CPU utilization by the foreground processes are varied from 0% to 90%. At utilization above 50%, the slowdown is so large that lingering slows down the jobs dramatically. A useful comparison of this slowdown is to consider alternatives to running on the non-idle node.

The NOW [24] project has proposed migrating to an idle node when the user returns, however if there is a substantial load on the cluster we would have to keep idle nodes in reserve (i.e. not running other parallel jobs) to have one available. Alternatively, Acha et al. [2] proposed re-configuring the application to use fewer nodes. However, many applications are restricted to running on a power of two number of nodes (or a square number of nodes). Thus the unavailability of a single node could preclude using many otherwise available nodes. Within this context, our slowdown of only 1.1 to 1.5 when the load is less than 40% is an attractive alternative.

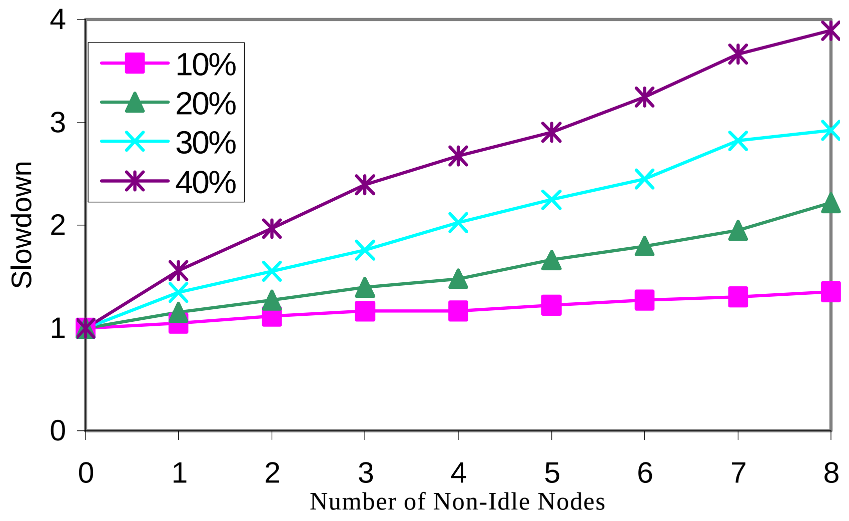
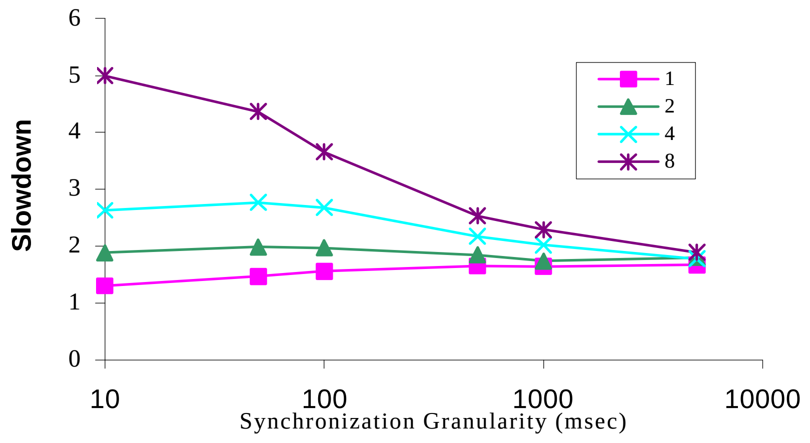


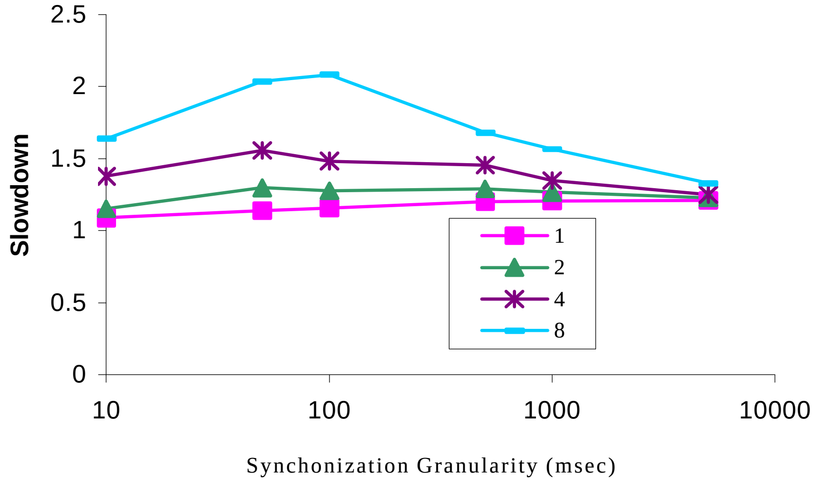
Figure 14: Parallel job slowdown vs. the numbers of non-idle nodes

The graph shows the slowdown of an eight-process parallel application at four levels of processor utilization by host jobs when the number of non-idle nodes is varied from one to eight.

We also investigated how the slowdown of a parallel program was affected by having more than one non-idle node. To evaluate this case, we again used the same eight-process bulk-synchronous application that we used in the previous simulation and varied the number of non-idle nodes. The results of this simulation are presented in Figure 14, and show that at low levels of foreground processor utilization, the parallel application slowdown is relatively insensitive to the number of non-idle nodes. For example, at local utilization of up 20%, the largest slowdown was a factor of two even when all 8 processes are lingering on non-idle nodes. However, when local utilization was above 30%, and the number of non-idle nodes is more than one or two, the slowdown to the parallel application is significant.



(a) 20% local utilization



(b) 40% local utilization

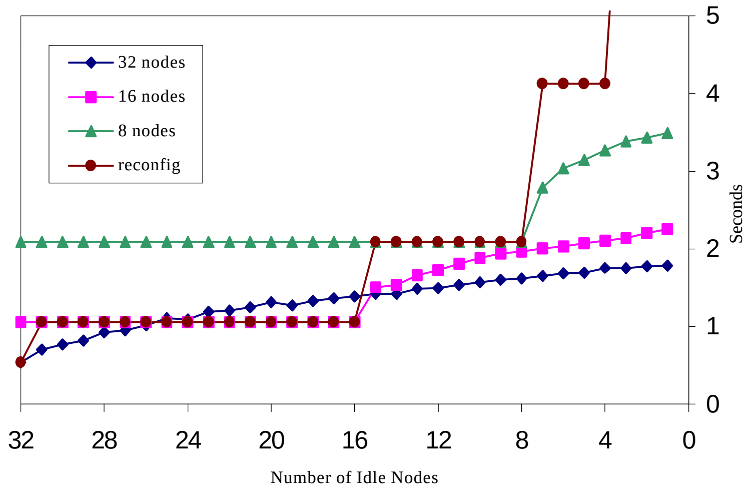
Figure 15: Synchronization granularity vs. slowdown

The graph (a) shows the slowdown of running a parallel program with 1, 2, 4, or 8 non-idle nodes compared with running on all 8 idle nodes as a function of the synchronization granularity when the non-idle nodes have 20% utilization by host jobs. The graph (b) shows the same data when the non-idle utilization is 40%.

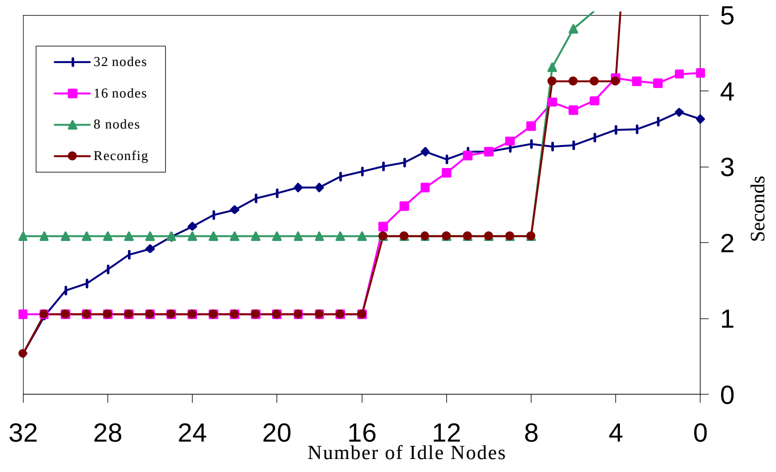
One of the key parameters in understanding the performance of parallel jobs using Linger-Longer is the frequency of synchronization. Figure 15 shows the relationship between the granularity of communication and the slowdown experienced by an eight process bulk synchronous program. The x-axis is the computation time between communications in milli-seconds. Each of the four curves shows the slowdown when 1, 2, 4, and 8 nodes have 20% (a) and 40% (b) processor utilization by host jobs. The results show that larger synchronization granularity produces less slowdown. Also, for 20% local processor utilization, lingering provides an attractive alternative to reconfiguration since, even when four nodes are non-idle, the slowdown remains under a factor 1.5. (Note that reconfiguration with four nodes unavailable would have a slowdown of at least 2.)

We wanted to provide a head-to-head comparison of the Linger-Longer policy with the reconfiguration strategy. To do this, we simulated a 32 node parallel cluster. For each scheduling policy we considered the effect if the average processor utilization by the host jobs on non-idle workstations was 20% or 40%. We defined the average synchronization granularity to be 500 msec. In this simulation, we didn't consider the time required to reconfigure the application to use fewer nodes, and assumed that the application was constrained to run on a power of two number of nodes. The results of running this simulation are shown in Figure 16. In each graph, the curves show the Linger-Longer policy using 8, 16, and 32 nodes, and the reconfiguration policy using the maximum idle nodes available. The Linger-Longer with *k* nodes means if *k* or more idle nodes are available in the cluster, the parallel job runs k processes on k idle nodes, otherwise it runs on all idle nodes available and some non-idle nodes by lingering.

The left graph shows the results for 20% utilization and the right 40%. For the case of 20% utilization, the Linger-Longer policy outperforms reconfiguration, when either 8 or 16 nodes are used. For 40% utilization, the reconfiguration strategy generally provides faster completion of the application. However, using 32 nodes and a Lin-



(a) 20% local utlization



(b) 40% local utilization

Figure 16: Linger-Longer vs. reconfiguration

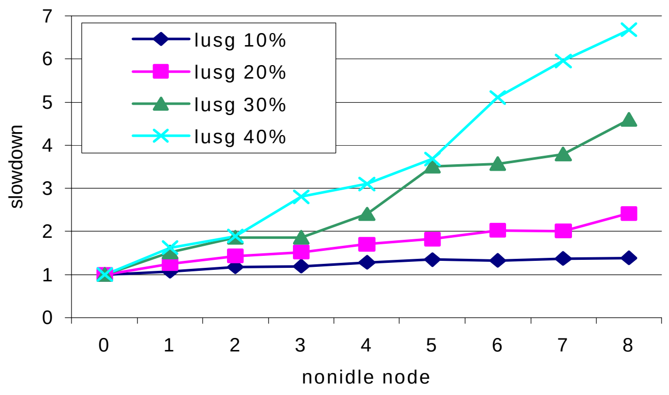
The left graph shows the completion time of a parallel job running on a cluster using several different scheduling policies. The x-axis shows the number of idle nodes. The first three curves show the Linger-Longer policy running using 8, 16, or 32 nodes.

ger-Longer policy outperforms reconfiguration when 5 or fewer non-idle nodes are used for the 20% processor utilization case.

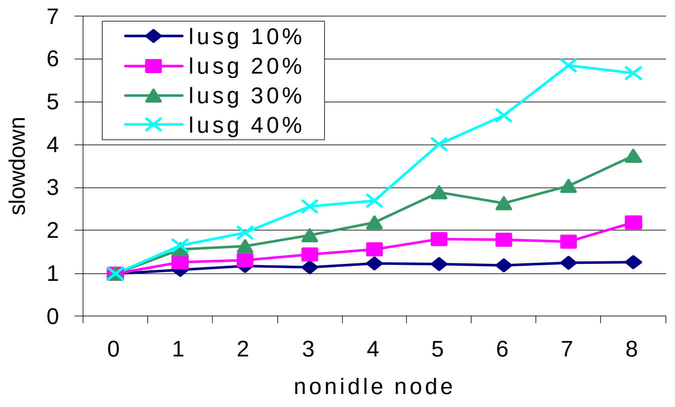
4.4 Real Parallel Jobs

To validate the results from the synthetic bulk synchronous application case, we ran several real parallel applications with Linger-Longer. To do so, we combined two different types of simulators: our Linger-Longer simulator generating local workloads and a CVM [38] simulator which can run shared memory parallel applications. We chose three common shared-memory parallel benchmarks: `sor`(Jacobi relaxation), `water`(a molecular dynamics; from SPLASH-2 benchmark suite [73]) and `fft`(fast Fourier transformation) which have different computation and communication patterns. Throughout all the experiments, the network bandwidth was set to 155 Mbps. The input for `sor` was a 2048×1024 array. For `water`, 512 molecules were simulated. A three dimensional input array of $2^6 \times 2^6 \times 2^4$ was used for `fft`.

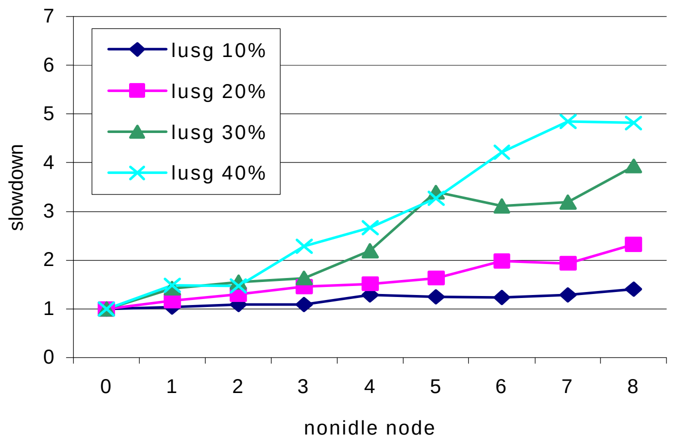
First of all, we looked at how Linger-Longer on non-idle nodes slows down the parallel jobs. An eight node cluster is used to run each application. The number of non-idle nodes and their local utilization were controlled. The results are summarized in Figure 17. For all three cases, when only one non-idle node is involved even with 40% local utilization the slowdown (compared to running on eight idle nodes) reaches only 1.7. When more than half the nodes are non-idle, 0 to 20% local utilization looks endurable. Linger-Longer with four non-idle nodes and 20% local utilization causes only a 1.5 to 1.6 slowdown. Even when all eight nodes are non-idle, the job is slowed down by just above a factor of two for all three applications. While a factor of two may seem like a large slowdown, it's substantially better than not using any of the non-idle nodes, which previous policies required.



(a) sor



(b) water

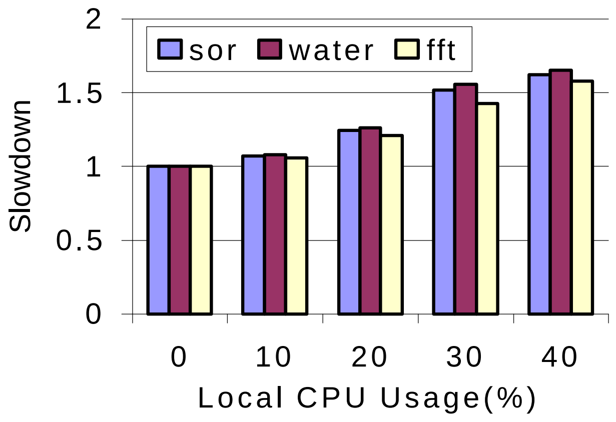


(c) fft

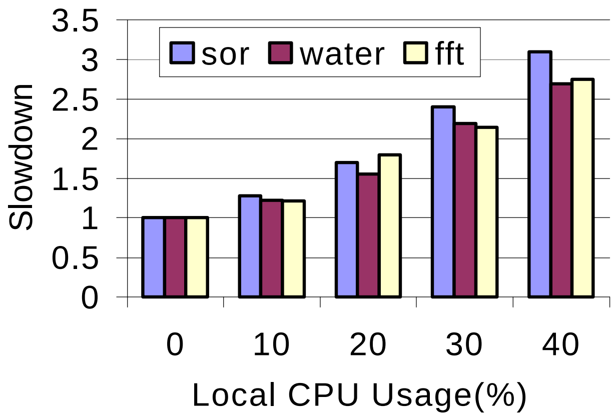
Figure 17: Slowdown by non-idle nodes and their local CPU usage

The graphs show the slowdown of parallel jobs: `sor` *(a),* `water` *(b) and,* `fft`*(c) as the number of non-idle nodes varies from 0 to all 8 with Linger-Longer. The cluster size is eight. The curves in each graph represent the different local utilization of non-idle nodes.*

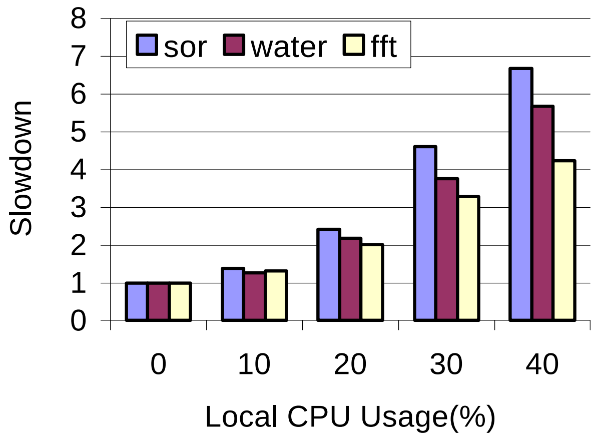
The Linger-Longer effect on the performance also depends on the applications. With the same data used above, we compared three applications for various local utilizations and varied the number of non-idle nodes. The results are shown in Figure 18. The difference in the slowdown between the three applications becomes more noticeable as local utilization increases. In general `sor` is the most sensitive to local utilization. `water` is less sensitive to local activity and `fft` is the least.



(a) only 1 non-idle node



(b) 4 non-idle node (half)



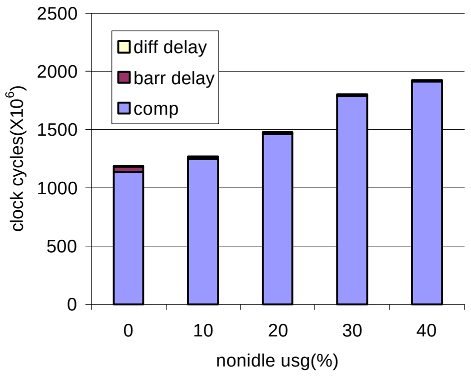
(c) 8 non-idle node (all)

Figure 18: Performance sensitivity to local CPU usage

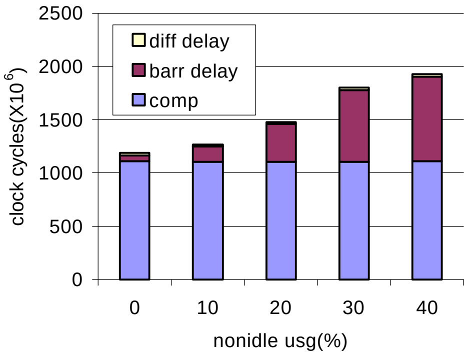
The graphs compare 3 parallel jobs (`sor, water` *and* `fft`*) in how sensitive to local utilization of non-idle nodes. The left-most graph (a) is for the case only 1 of total 8 nodes is non-idle, (b) and (c) is for 4 and 8 non-idle nodes, respectively.*

To see the cause of this difference, we divided the completion time into computation time, barrier waiting time, diff delay and lock waiting time. The computation time includes any CPU time the process uses. Barrier waiting time is the time that a process spends waiting until all the processes reach a barrier. Diff delay is the time to access a shared page which can involve communication to obtain coherent data. Diffs are required to reintegrate disjoint updates of a shared page by multiple processes as part of the DSM coherency protocol (Diffs and the lazy release consistency protocol in CVM are explained in [38]). Lock waiting time is the time to obtain the lock on a shared object. In our applications, only `water` uses locks.

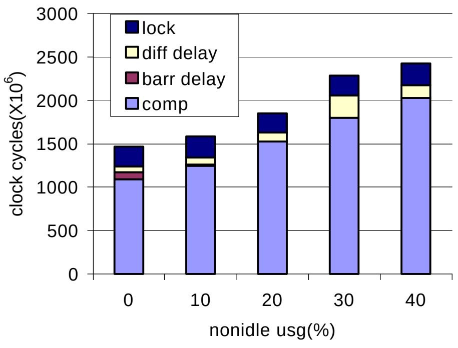
The graphs in Figure 19 and Figure 20 summarize the results. Figure 19 shows the case when only one node is non-idle out of eight nodes. Each graph contains five bars for different local CPU utilizations of the non-idle node. In Figure 19, the left hand side graphs (a, c, e) show the profiles for the non-idle node for the three applications and the second row shows the behavior of an idle node. Since only one non-idle node is used, barrier delay is limited and does not increase due to the local CPU utilization. Not surprisingly, the right hand side graphs (b, d, f) demonstrate that the barrier waiting time on idle nodes increases as local CPU utilization on the non-idle grows. There is no noticeable change in diff delay and lock waiting time for this case.



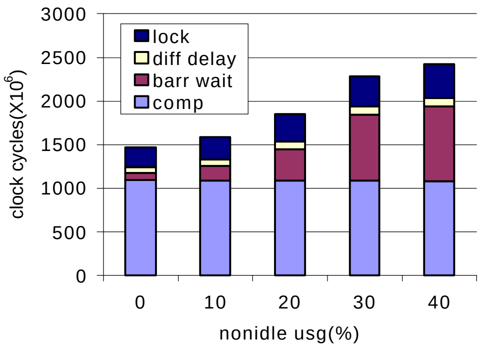
(a) sor (non-idle node)



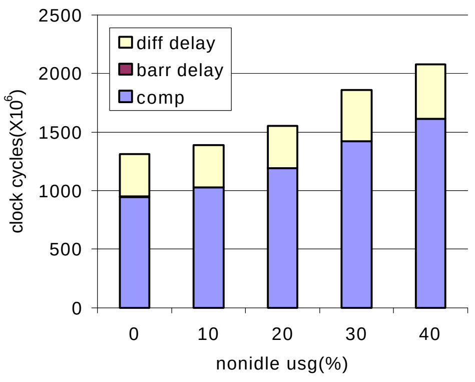
(b) sor (idle node)



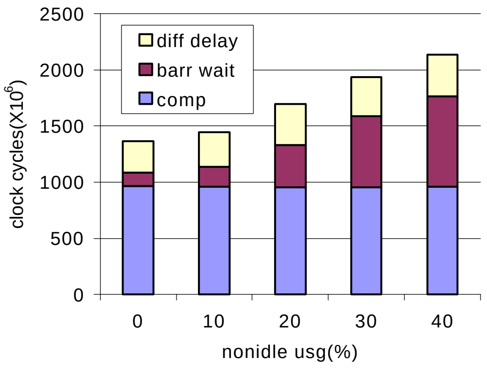
(c) water (non-idle node)



(d) water (idle node)



(e) fft (non-idle node)



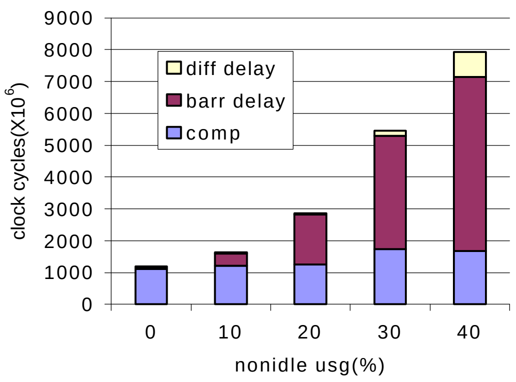
(f) fft (idle node)

Figure 19: Application sensitivity to local activity - 1 non-idle case

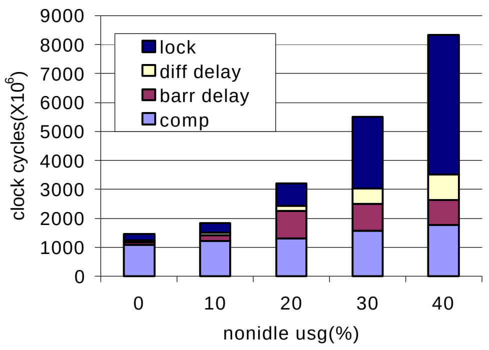
The graphs show the time spent in different activities for three parallel jobs running on a cluster of eight nodes. Only one node is non-idle. The left graph of each row shows the results for the non-idle node. The right graph of each row is for an idle-nod. For each graph, the x-axis shows local CPU utilization on a non-idle node. Each bar comprises computation time, barrier waiting time, diff delay and lock waiting time.

Figure 20 illustrates the case when all eight nodes are non-idle. Since interference due to CPU contention occurs on every node, synchronization overhead increases. The key parameter to understanding Linger-Longer delay is synchronization frequency. The barrier synchronization granularity is 10.1 million cycles for `sor`, 50.7 million cycles for `water`, and 38.4 million cycles for `fft` for the given input size. In other words, barrier synchronization in `sor` is five times as frequent as in `water`. The first graph in Figure 20 shows that barrier delay is a dominant factor for the slowdown of `sor`. For `water`, lock waiting time affects the slowdown the most. `fft` has the least slowdown and it is mostly due to an increase in the diff delay. These results are consistent with the synchronization granularity impact seen for the synthetic parallel application in Section 4.3.

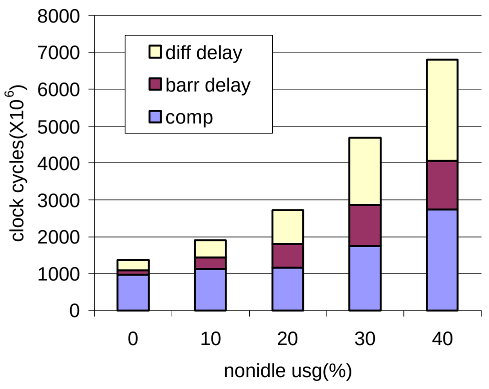
To summarize, frequent synchronization makes parallel jobs more sensitive to local CPU activity and the effect of global synchronization, such as barriers, is more than local synchronization, such as locks. However, for a practical lingering local utilization, 0 to 20%, the slowdown is almost the same for all three applications. In general, when local utilization exceeds 20%, we would prefer to migrate the job to free nodes (if any exists) or to reconfigure the application to run on fewer nodes.



(a) sor (non-idle node)



(b) water (non-idle node)

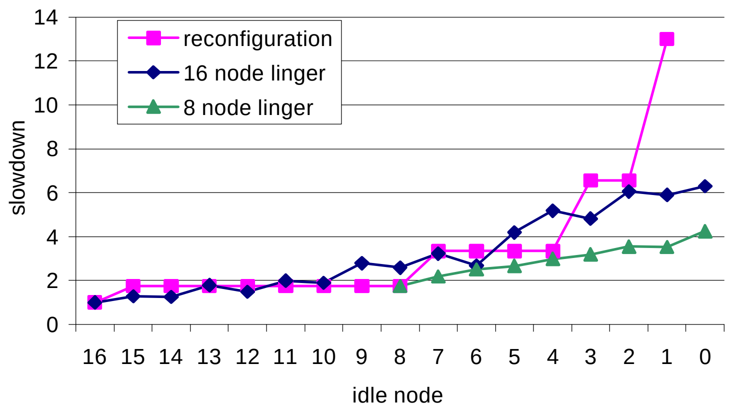


(c) fft (non-idle node)

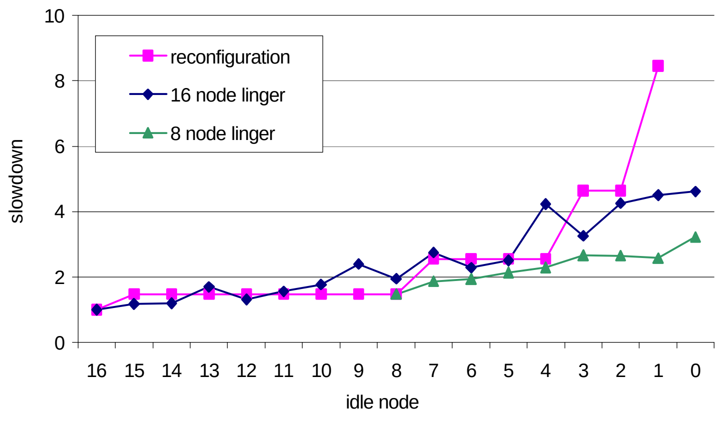
Figure 20: Application sensitivity to local activity - 8 non-idle case

The graphs show the time spent in different stages for three parallel jobs running on a cluster of 8 nodes. All eight nodes are non-idle. For each application, the x-axis shows local CPU utilization on an non-idle node. Each bar comprises computation time, barrier waiting time, diff delay and lock waiting time (from bottom to top).

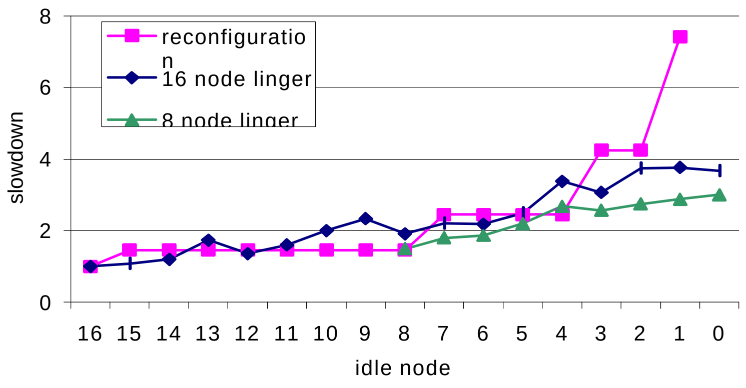
Also, we compared the effect of Linger-Longer and reconfiguration policies on the performance of three real parallel application performance. The same assumptions as in the synthetic application case were maintained except that only a 16 node cluster was simulated. The results of running this simulation are shown in Figure 21. In each graph, the curves show the Linger-Longer policy using 16 and 8 nodes, and the reconfiguration policy using the maximum power of 2 number of idle nodes available. The graphs show the results for `sor`, `water` and `fft` when the local utilization for non-idle nodes is 20%. For all cases, the Linger-Longer policy using 16 nodes outperforms the reconfiguration when the number of idle nodes is at least 12. Considering the cost in running time to reconfigure the parallel job, the gain would be even bigger. However, when fewer than eight idle nodes are left, lingering with 8 nodes looks much better than both lingering using 16 nodes and the reconfiguration policy. This indicates that a hybrid strategy of lingering and reconfiguration may be the best approach.



(a) sor



(b) water



(c) fft

Figure 21: Linger-Longer vs. reconfiguration for shared-memory parallel applications

The graphs show the slowdown of 3 parallel jobs running on a cluster of 16 nodes using several different scheduling policies. The x-axis shows the number of idle nodes. The first (box marked) curve shows the reconfigure policy. The other two curves show Linger-Longer policy running using 16 or 8 nodes. For all curves, the local utilization of non-idle nodes is 20%.

In this section, we investigated running parallel applications with Linger-Longer and varied not only local CPU utilization but also the number of non-idle nodes. In general, larger synchronization granularity produces less slowdown. These facts were verified through both synthetic bulk synchronous and real shared-memory parallel programs. To make effective use of Linger-Longer, frequent global synchronization should be avoided or replaced with local synchronization if possible. For both types of parallel applications, Linger-Longer can be useful since it can use lightly loaded non-idle nodes rather than suspending the whole program until the node returns to idle or reconfiguring to run on fewer nodes.

In this chapter, our simulations demonstrated that guest jobs with ultra-low CPU priority can exploit more than 90% of fine-grain idle cycles with only a few percent slowdown of host jobs. We also showed that the Linger-Longer policy can improve the throughput of guest jobs on a 64 machine cluster by up to 60% with only a 1% slowdown of local jobs. Finally, for parallel guest jobs, our policy outperforms reconfiguration strategies when the processor utilization by the local process is 20% or less in both synthetic bulk synchronous and real data-parallel applications.

Chapter 5

Operating System Support

This chapter presents the design, implementation and performance evaluation of a suite of kernel mechanisms that allow the use of Linger-Longer on non-dedicated workstations. We developed strict priority scheduling for CPU policing, prioritized memory page replacement for memory policing and rate windows for I/O and network bandwidth throttling. These mechanisms are essential since the priority mechanisms of current operating systems are not sufficient to protect the performance of higher priority jobs.

We chose Linux as our target operating system for several reasons. First, it is one of the most widely used UNIX operating systems. Second, the source code is open and widely available. Since many active Linux users build their own customized kernels, our mechanisms could easily be patched into existing installations by end users. This is important because most PCs are deployed on people's desks, and cycle-stealing approaches are probably more applicable to desktop environments than to server environments.

5.1 Starvation-level CPU Priority

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

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

Table 2: CPU utilization with single host and guest (*niced* at level 19) processes

We first investigated the impact of simply using the UNIX nice command to provide CPU priorities. To do this, we constructed a compute bound test program that simply ran an empty loop a fixed number of iterations. We ran two copies of this process. The first simulates a host process by running with the default nice value (0), and the other simulates a guest process by running at the lowest possible priority, nice level -19. The CPU utilizations resulting from this experiment for four different versions of UNIX are shown in Table 2. The table shows the percent of the processor that each process received. Except when running under OSF-1, the guest process received a significant amount of processing time (ranging from 8% to 40%). This simple experiment demonstrates the need for our more sophisticated priority mechanism.

First, we investigated the Linux CPU scheduler to understand why a higher priority process loses some CPU cycles. The scheduler chooses a process to run by selecting the ready process with the highest runtime priority, where the runtime priority can be thought of as the number of 10ms time slices held by the process. The runtime priority is initialized from a static priority derived from the *nice* level of the process. Static priorities range from -19 to +19, with +19 being the highest[4]. New processes are given

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

20+*p* slices, where *p* is the static priority level. The process chosen to run has its store of slices decremented by one. Hence, all runnable processes tend to decrease in priority until no runnable processes have any remaining slices. At this point, all processes are reset to their initial runtime priorities. Blocked processes receive an additional credit of half of their remaining slices. For example, a blocked process having 10 time slices left will have 20 slices from an initial priority of zero, plus five slices as a credit from the previous round. This feature is designed to ensure that compute-bound processes do not receive undue processor priority compared to I/O bound processes. The algorithm is summarized in Figure 22.

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

We implemented a new *guest priority* in order to prevent guest processes from

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

Figure 22 Original Linux scheduler

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

Figure 23: Modified Linux scheduler.

running when runnable host processes are present. The change essentially establishes guest processes as a different class, such that guest processes are not chosen if any runnable host processes exist. This is true even if the host processes have lower runtime priorities than the guest process. The modified scheduling algorithm is shown in Figure 23.

Second, we verified that the scheduler reschedules processes any time a host process unblocks while a guest process is running. This is the default behavior on Linux, but not on many BSD derived operating systems. One potential problem of our strict priority policy is that it could cause priority inversion. Priority inversion occurs when a higher priority process is not able to run due to a lower priority process holding a shared resource. This is not possible in our application domain because guest and host processes do not share locks, or any other non-revocable resources.

We first validated our scheduling modifications by comparing the CPU utilization of a CPU-intensive guest process competing with that of a host process for three different scheduling policies. Our independent variable is the percent utilization of the

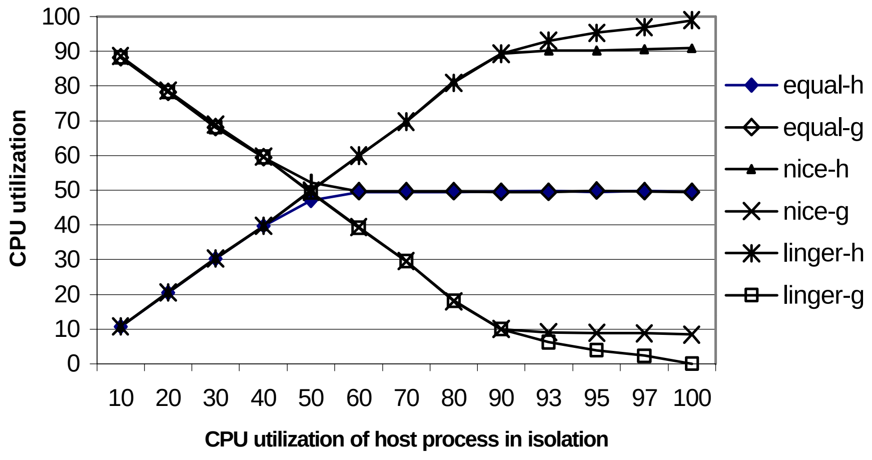


Figure 24: CPU utilization

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

host process in the absence of any competing processes. The CPU-intensive guest process is representative of typical guest processes, such as scientific simulations, decision support(data mining), and graphics rendering. This process also provides us with a worst-case (in terms of contention for the CPU) test of scheduling policies.

Figure 24 shows the resulting behavior. Ideally, the CPU utilization of the host processes would track linearly with the utilization of the job in isolation. The 'equ

lines show the default case where guest processes are treated identically to host processes. The 'nice-h' line shows that host process utilization is unaffected by the presence of a niced guest process up to approximately 90% utilization. The drop-off at this point corresponds to the 91% limit shown for Linux in Figure 24. Note that 'linger-h', included for comparison, accurately tracks expected utilization up to 99%. The data shows that a guest process is unable to significantly interfere with CPU utilization of a host process with the Linger-Longer modifications. Similar modifications to the other operating systems discussed earlier in Table 2 would presumably show analogous curves, with the difference being that 'nice-h' utilization would flatten out at 84% for Solaris and at only 60% for AIX.

5.2 Prioritized Page Replacement

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

We first investigated the impact of simply using the UNIX nice command for memory intensive applications. Table 3 shows a simple example of memory thrashing caused by allowing a guest process to compete with host processes for physical memory. In all cases, both processes have working sets of approximately 128 MB, while the total physical memory of the machine is only 192 MB. Both processes take 82 seconds to run in isolation. When they are run serially (first row), the total running time is just 164 seconds. The second row shows that if the two are started simultaneously, and with

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

Table 3: Completion times for two competing large memory jobs

equal priorities, the processes thrash and lose efficiency. We stopped the processes after five hours. The third row shows the expected result of a late-arriving guest process being unable to steal pages from the host process, and effectively being serialized after the host process. However, it does slow down the host process by about 8%. The last row, however, shows that changing the order in which the processes arrive dramatically changes the result. The host process takes a long time to steal enough pages from the guest process in order to hold its working set. We again stopped the execution after about five hours. The reason for the thrashing is that the guest process had modified its pages before the host process starting requesting memory. Each initial page fault by the host process is delayed while a guest page is flushed to disk. Meanwhile, the guest process also has page faults that require host pages to be flushed to disk. Therefore, neither process makes much progress since CPU priority does little to prevent thrashing when two processes desire more memory that the system has.

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

Unfortunately, memory is more difficult to deal with than the CPU. The cost of reclaiming the processor from a running process in order to run a new process consists only of saving processor state and restoring cache state. The cost of reclaiming page frames from a running process is negligible for clean pages, but quite large for modified pages because they need to be flushed to disk before being reclaimed. The simple solution to this problem is to permanently reserve physical memory for the host processes. The drawback is that many guest processes are quite large. Simulations and graphics rendering applications can often fill all available memory. Hence, not allowing guest processes to use the majority of physical memory would prevent a large class of applications from taking advantage of idle cycles.

We therefore decided not to impose any hard restrictions on the number of physical pages that can be used by a guest process. Instead, we implemented a policy that establishes low and high thresholds for the number of physical pages used by guest processes. Essentially, the page replacement policy prefers to evict a page from a host process if the total number of physical pages held by the guest process is less than the low threshold. The replacement policy defaults to the standard clock-based pseudo-LRU policy up until the upper threshold. Above the high threshold, the policy prefers to evict a guest page. The effect of this policy is to encourage guest processes to steal pages from host processes until the lower threshold is reached, to encourage host processes to steal from guest processes above the high threshold, and to allow them to compete evenly in the region between the two thresholds. However, the host priority will lead to the number of pages held by the guest processes being closer to the lower threshold, because the host processes will run more frequently.

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

We modified the Linux kernel to support this prioritized page replacement. Two new global kernel variables were added for the memory thresholds, and are configurable at run-time via system calls. The kernel keeps track of resident memory size for guest processes and host processes. Periodically, the virtual memory system triggers the page-out mechanism. When it scans in-memory pages for replacement, it checks the resident memory size of guest processes against the memory thresholds. If they are below the lower thresholds, the host processes' pages are scanned first for page-out. Resident sizes of guest processes larger than the upper threshold cause the guest processes' pages to be scanned first. Between the two thresholds, older pages are paged out first no matter what processes they belong to. The modifications to the page replacement algorithm are shown in Figure 25. Correct selection of the two parameters is critical to meeting the goal of exploiting fine-grained idle intervals without significantly impacting the performance of host processes. Too high a value for the low threshold will cause undue delay for host processes, and too low a value will cause the guest process to constantly thrash. However, if minimum intrusiveness by the guest process

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

Figure 25: Modified page replacement policy.

is paramount, the low memory threshold can be set to zero to guarantee the use of the entire physical memory by the foreground process.

We validated our memory threshold modifications by tracking the resident memory size of host and guest processes for two CPU-intensive applications with large memory footprints. The result is shown in Figure 26. The chart shows memory competition between a guest and a host process. The application behavior and memory thresholds shown are not meant to be representative, but were constructed to demonstrate that the memory thresholds are strictly enforced by our modifications to Linux's page replacement policy. The guest process starts at time 20 and grabs 128MB. The host process starts at time 38 and quickly grabs a total of 128 MB. Note that the host actually touches 150 MB. It is prevented from obtaining all of this memory by the low threshold. Since the guest process' total memory has dropped to the low threshold, all replacements come from host pages. Hence, no more pages can be stolen from the

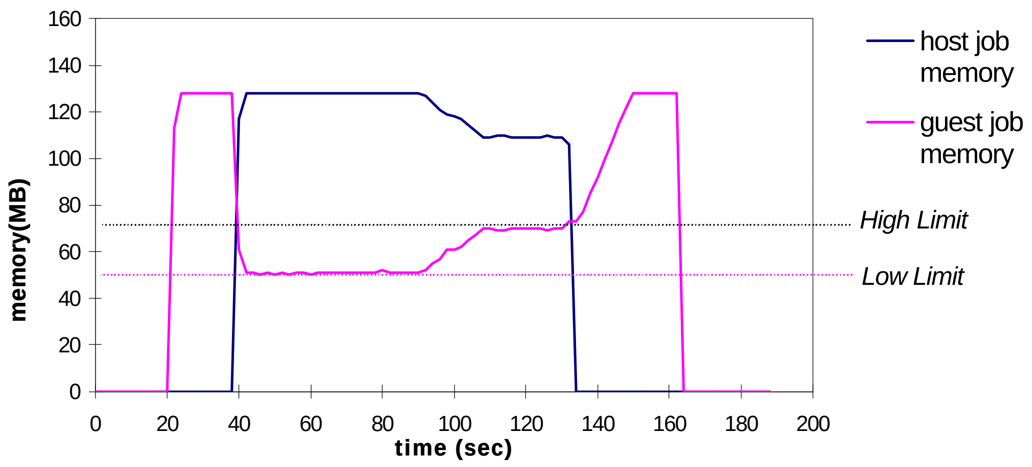


Figure 26: Threshold validations

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

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

Table 4: Benefits of memory priority for large footprint processes.

guest. At time 90, the host process turns into a highly I/O-bound application that uses little CPU time. When this happens, the guest process becomes a stronger competitor for physical pages, despite the lower CPU priority, and slowly steals pages from the host process. This continues until time 106, at which point the guest process reaches the high threshold and all replacements come from its own pages.

We also repeated the experiment shown in Table 3 with our memory priority system enabled. The results are shown in Table 4. When the host process starts first and then the guest process (this is the behavior seen when a user is working, but not using the processor heavily and a guest process then arrives), the use of our modified virtual memory policies reduces the delay seen by the host process from 8.0% seen when nice is used to 0.8%. For the case when the guest process starts and then the user process, the delay with nice was larger than a factor of 200 (recall that we gave up after waiting five hours). In contrast, using linger priority only had a delay of about 8%. These two results demonstrate the ability of our kernel modifications to limit the overhead experienced by guest processes.

Another aspect of application behavior that needs to be addressed is the slow reclamation of dirty pages from a guest process by a later-starting host process. This is the problem illustrated by the last row in Table 4. The host process is delayed by the one-at-a-time flushing of dirty guest pages to disk. While this is a special case, it is probably the most visible problem of cycle-stealing distributed schedulers. We addressed this problem by adding a trigger mechanism to the pageout process that notices when a new host process starts causing pageouts of guest pages. The trigger mechanism artificially increases the rate at which pages are flushed to disk, analogously to block prefetches. We implement this policy by temporarily increasing the number of pages that the VM system tries to page out at a time (called maxpages) on a fault that is triggered by a host process reclaiming pages from guests.

Figure 27 shows the impact of using different multiplicative factors for the desired free list length. When the initial value is used, the host process is delayed by a factor of 3.5 compared to time it would take without the guest process being present. However, once we increase this value to about 50 times its original value, the system pages out most of the pages used by the guest process quickly, and the delay is only about 20%. While this delay seems large, the test program ran only two iterations, and so most of the time was spent getting its working set into memory. A real application would run

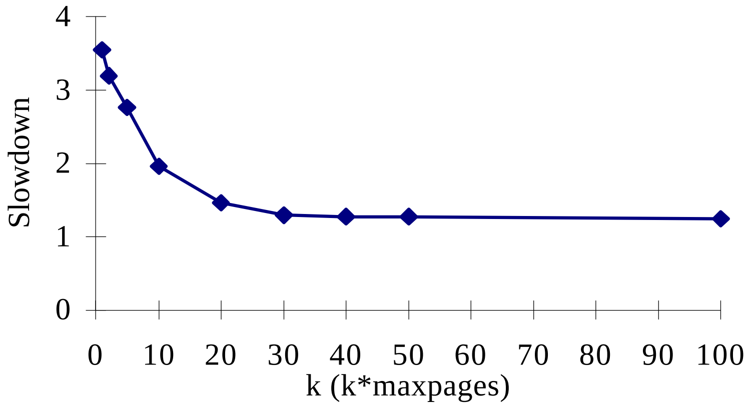


Figure 27: Impact of varying the number of pages paged out at a time

longer and the overhead would be amortized.

In this section, we demonstrated that the Unix CPU priority is not effective to promote memory bound host processes. Thus, a new prioritized page replacement was introduced and validated by a set of experiments. To reduce the slowdown of large memory host processes, we developed another mechanism that increases the page-out unit to expedite supplying physical memory pages to host jobs. Our experiments reported that even a rare worst case, our mechanisms can decrease host job slowdown from a factor of 200 % to 8%.

5.3 Rate Windows for I/O and Network Throttling

We need priority mechanisms for I/O and network since I/O or communication intensive guest jobs can significantly slow down host jobs. Also, network intensive jobs can slow others on the network. To protect host jobs' I/O performance, rate windows are proposed here as a simple, portable, and effective option. Many real-time systems provide rate based scheduling which is similar to our rate windows. However, all those mechanisms are not suitable for our use since a new scheduler for I/O causes high overhead. The rest of this section describes our rate-window policies, and the mechanisms that are needed to support I/O throttling. Then we validate our mechanisms with a series of experiments.

5.3.1 Rate Windows Mechanisms

First, we distinguish between "unconstrained" and "constrained" job classes. The default for all processes is unconstrained; jobs must be explicitly put into constrained classes. The unconstrained class is allowed to consume all available I/O. Each distinct constrained class has a different *threshold bandwidth*, defining the maximum aggregate

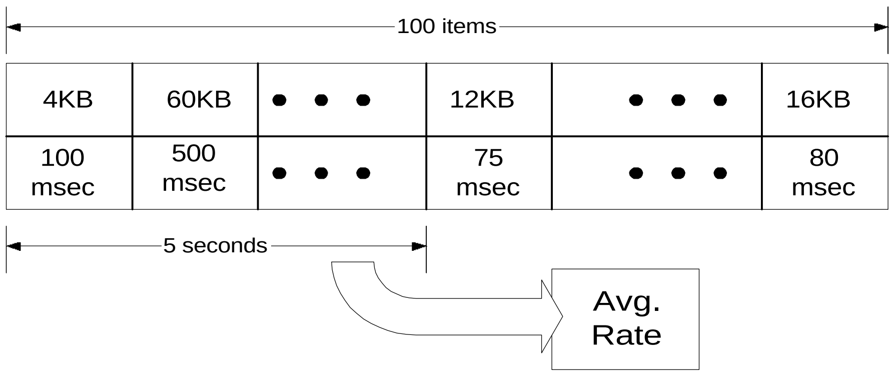


Figure 28: Maintaining a sliding window of resource utilization

bandwidth that all processes in that class can consume. As an optimization, however, if there is only one class of constrained jobs, and no I/O-bound unconstrained jobs, the constrained jobs are allowed unfettered access to the available bandwidth.

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

The implementation of rate windows is straightforward. We currently have a hard-coded set of job equivalence classes, although this could be easily generalized for an arbitrary number. Each class has two kernel *window structures*, one for file I/O and one for network I/O. Each window structure contains a circular queue, implemented via a 100-element array (see Figure 28).

The window structure describes the last I/O operations performed by jobs in the class, plus a few other scalar variables. The window structure only describes I/O events that occurred during the previous 5 seconds, so there may be fewer than 100 operations in the array. We experimented with several different window sizes before arriving at these constants, but it is clearly possible that new environments or applications could be best served by using other values. We provide a means of tuning these and other parameters from a user-level tool.

We implemented our mechanism via a loadable kernel module which intercepts each of the kernel calls for I/O and network communication: `read()`, `write()`, `send()`, and `recv()`. Whenever such system functions are triggered, we first call `rate_check()` with the process ID, I/O length, and I/O type and then call the original system call. The process ID is used to map to an I/O class, and the I/O type is used to distinguish between file and network I/O. The `rate_check()` routine maintains a sliding window of operations performed for each class of service and for the overall system. However, to prevent using information that is too old, we limit the sliding window to a fixed interval of time (currently 5 seconds).

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

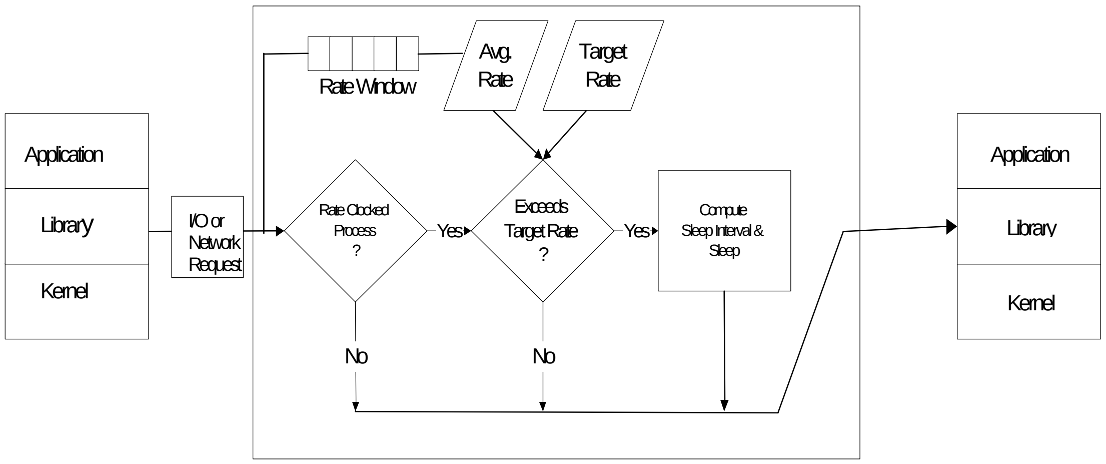


Figure 29: Policing I/O Requests

$$\frac{B_w}{T_w} \leq R_t \tag{5.1}$$

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

$$\text{delay} = \frac{B_w}{R_t} - T_w \tag{5.2}$$

And then the kernel suspends the process for delay time units before calling the original I/O system call. This process is illustrated graphically in

Figure 29. Note that we have upper and lower bounds on allowable sleep times.

Sleep durations that are too small degrade overall efficiency, so durations under our lower bound are set to zero. Sleep durations that are too large tend to make the stream bursty. If our computed delay is above the computed threshold we break the I/O into multiple pieces and spread the total delay over the pieces. This will not affect application execution since file I/O requests will eventually be broken into individual disk blocks. For network connections TCP provides a byte-oriented stream rather than a record oriented one, so breaking a single request into a smaller one will not affect the correctness of any protocol.

Again, we chose Linux as our target operating system for several reasons. First, it is one of the most widely used UNIX operating systems. Second, the source code is open and widely available. Because our throttling mechanisms are implemented as a loadable kernel module, end users can easily load and enable them at run-time. By contrast, a source code patch would require rebuilding a kernel and rebooting a machine.

Also, since our mechanism simply requires the ability to intercept I/O calls, it would be easy to implement on other systems that defined an API to intercept I/O calls. Windows 2000 (nee Windows NT) and the stackable file system [36] provide the required calls.

In order to provide the finer granularity of sleep time to allow our policing to be implemented, we augmented the standard 2.2 Linux kernel with extensions developed by the KURT Real-time Linux project [9]. KURT's microsecond resolution timer support was enabled since Linux 2.2 can support only a 10 millisecond resolution timer for sleep[5].

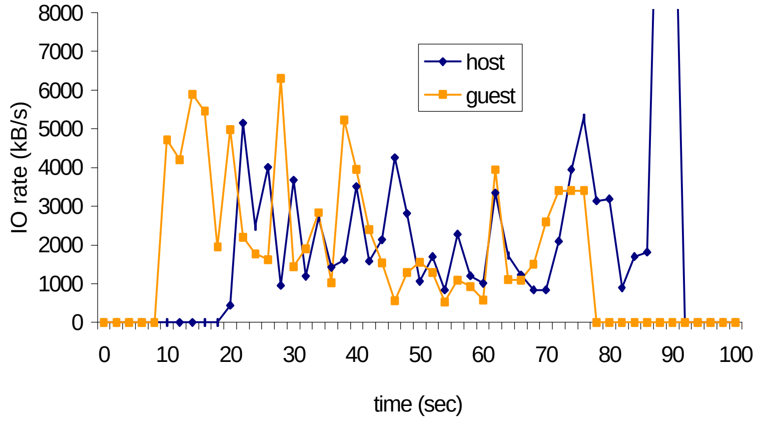
5.3.2 File I/O Policing

In order to validate our approach, we conducted a series of micro-benchmarks and application benchmarks. The purpose of these experiments is three fold. First, we want to show that our mechanism does not introduce any significant delay on normal operation of the system. Second, we want to show that we can effectively police the I/O rates.

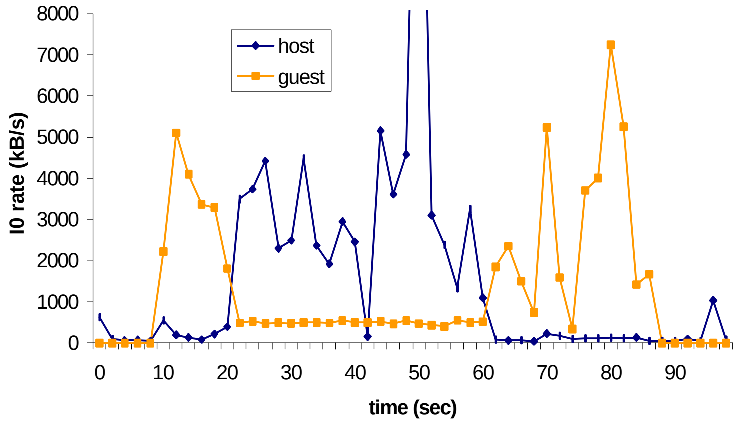
[5] This is due to the default setup of the timer unit on Linux. Linux 2.4 can now support a higher resolution timer using APIC, so the KURT patch will not be needed.

Third, since our policing mechanism sits above the file buffer cache, it will be conservative in policing the disk since hits in cache will be charged against a job classes's overall file I/O limit. We wanted to measure this effect.

We first measured resource usage in order to verify that the use of rate windows does not add significant overhead to the system. We ran a single tar program by itself both with and without rate windows enabled. We did not set the I/O limit since we wished to measure the overhead of maintaining rate windows and computing delays. The difference in completion time of the tar application with rate windows enabled was less than the variation between several runs of the experiment. This was expected, as there are no computationally expensive portions of the algorithm.



(a) Without I/O policing



(b) With I/O plicing

Figure 30: File I/O of competing tar applications

Second, we ran two instances of tar, one as a guest job and one as a host job. Figure 30(a) represents a run without throttling, and Figure 30(b) shows a run with throttling enabled. There is no caching between the two because they have disjoint input. The guest job is intended to be representative of those used by cycle-stealing schedulers such as Condor. Unless specified otherwise, a “guest” job is assumed to be constrained to 10% of the maximum I/O or network bandwidth, whereas a “host” process has unconstrained use of all bandwidth.

In both figures, the guest job starts first, followed somewhat later by the host job. At this point, the guest job throttles down to its 10% rate. When the host job finishes, the guest job throttles back up after the rate window empties. The sequence on the left is with throttling, on the right without. Note that the version with I/O throttling is less thrifty with resources (the jobs finish later). This is a design decision: our goal is to prevent undue degradation of unconstrained host job performance regardless of the effect on guest jobs.

The behavior of one of the tar processes is shown in more detail in Figure 31. The point of this figure is that despite the frequent and varied file I/O calls, and despite the buffer cache, disk I/O's get issued at regular intervals that precisely match the threshold value set for this experiment. Note that actual disk I/O sizes increase near the start as the file system read ahead becomes more aggressive.

Our third set of micro-benchmark experiments is designed to look at the distribution of sleep times for a guest process. For this case, we ran three different applications. The first application was again a run of the tar utility. Second, we ran the agrep utility across the source directory for the Linux kernel looking for a simple pattern that

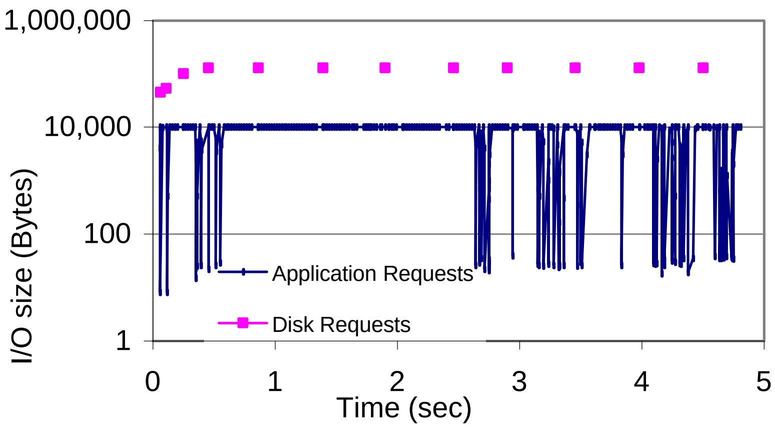
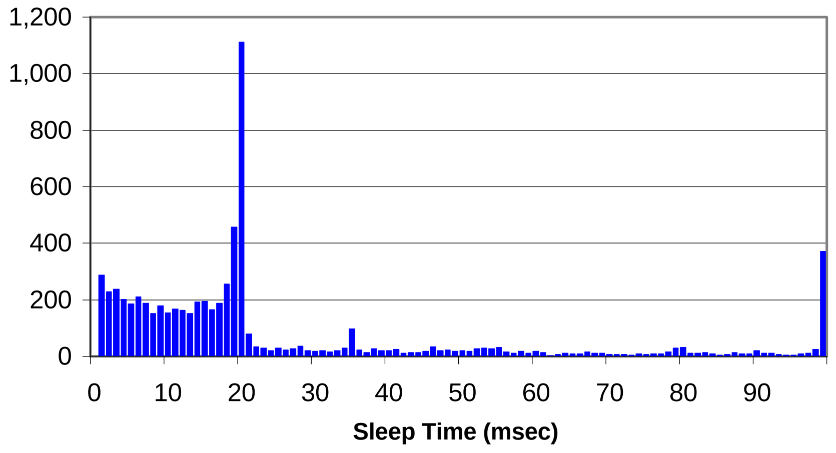


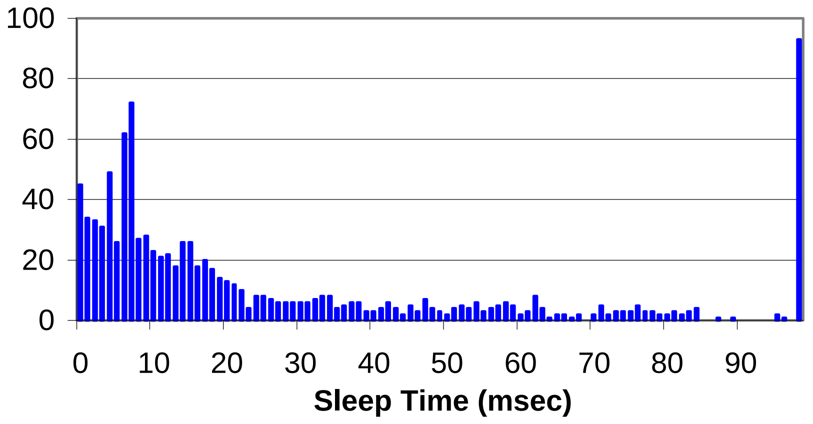
Figure 31: I/O sizes vs. time for `tar`

did not occur in the files searched. Third, we ran a compile workload that consisted of compiling a library of C++ methods that were divided among 34 files plus 45 header files. This third test was designed to stress the gap between monitoring at the file request level and the disk I/O level since all of the common header files would remain in the file buffer cache for the duration of the experiment.

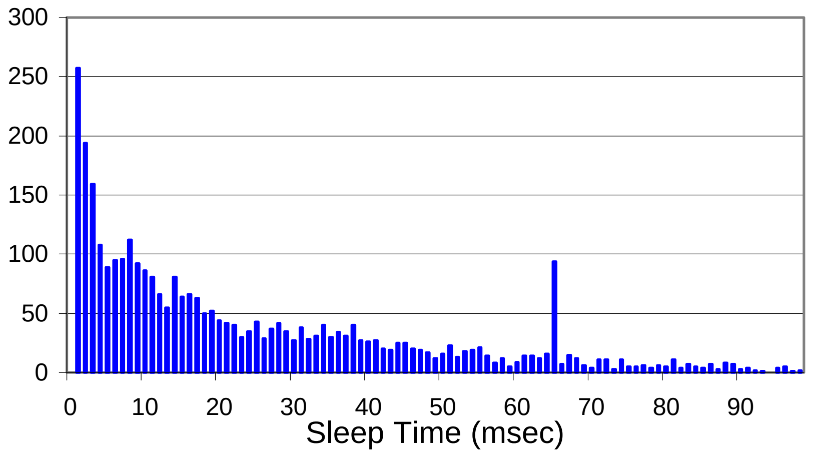
A histogram (100 buckets) of the sleep durations is shown in Figure 32. We have omitted those events that have no delay since their frequency completely dominates the rest of the values. Figure 32(a) shows the results for the tar application. In this figure, there is a large spike in the delay time at 20msec since this is exactly the mean delay required for the I/O for the most common sized I/O request, 10K bytes, to be limited to 500 KB/sec. Figure 32(b) shows the results for the compilation workload. In this example, the most popular sleep time is the maximum sleep duration of 100msec. This is due to the fact that at several periods during the application execution, the program is highly I/O intensive and our mechanism was straining to keep the I/O rate throttled down. Figure 32(c) shows the sleep time distribution for the agrep application. The results for this application show that the most popular sleep time (other than no sleep) was 2-3 ms. This is very close to the mean sleep time of 2.5 ms for this application.



(a) Tar



(b) Compile workload



(c) Agrep

Figure 32: Distribution of sleep times for tar, compile and agrep programs.

Third, we examine the relationship between file I/O and disk I/O using three applications, `tar`, `agrep` and `compile`, which have different I/O patterns. File I/O can dilate because i) file I/O's can be done in small sizes, but disk I/O is always rounded up to the next multiple of the page size, and ii) the buffer cache's read-ahead policy may speculatively bring in disk blocks that are never referenced. File I/O can also attenuate due to buffer cache hits, which is a consequence of the I/O locality of the applications. We measured 1) the total amount of file I/O requested, 2) the actual I/O requests performed by the disk, 3) the total number of I/O events 4) the total number of I/O events that were delayed by sleep calls, 5) the total amount of sleep time, 6) the total runtime of the workload, and 7) the average actual disk I/O rate (total disk I/O's divided by execution time). The results are shown in Table 5.

Looking first at the difference between file I/O and disk I/O, note that file I/O is equal to the disk I/O for `tar`[6], 14% less for `agrep`, and 233% larger for `compile`. Notice that for the two I/O intensive applications, the overall I/O rate for the application is very close to the target rate.

Metric	Tar	Agrep	Compile
Total File I/O	103.0 MB	50.0 MB	23.3 MB
Total Disk I/O	103.0 MB	58.1 MB	10.0 MB
Total I/O Events	17,430	11,526	3,859
Total Sleep Events	6,928	3,324	1,004
Total Sleep Time	178.0 sec	83.3 sec	29.1 Sec
Total Execution Time	211.2 sec	108.7 sec	70.6 Sec
Average Disk I/O Rate	487 KB/sec	534 KB/sec	141 KB/sec

Table 5: I/O application behavior

[6] The tar file size is 52 Mbytes.

For the tar application, our mechanism worked fine with the aggressive read ahead used by the file system. For `agrep`, we observed a higher total I/O volume due to small reads being rounded to larger disk pages. The low file I/O number for `compile`, of course, is due to good buffer cache locality.

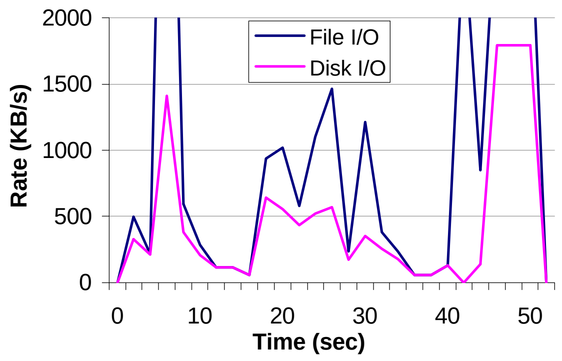
There are two potential approaches to recouping this lost bandwidth. The first is to add a hook into the buffer cache to check for a cache miss before adding the I/O to our window, and deciding whether to sleep and how long to sleep. We deliberately have not taken this path because we wish to keep our system at as high a level as possible. We currently implement our entire system as a loadable kernel module, which uses only externally available information such as the system call interface. This would be compromised if we put hooks deeper into the kernel.

A second approach is to use statistics from the `proc` file system to apply a "dilation factor" to our limit calculations. We define the dilation factor as the ratio of file I/O and disk I/O requests. If the ratio is 1.0, each file I/O is being transformed into the same amount of disk activity, i.e. there is no caching or reuse. If the ratio is 0.5, e.g. 100 KB of file I/O is being transformed into only 50 KB of disk I/O, then the limited job is not fully utilizing it's allocated bandwidth. Resources can be used more efficiently by multiplying the file I/O threshold by the inverse of the dilation factor. The disadvantage of this approach is that dynamic caching behavior will lead to time-varying dilation factors, and poor policing. The advantages are better bandwidth utilization, and that the approach can be implemented entirely outside of the kernel.

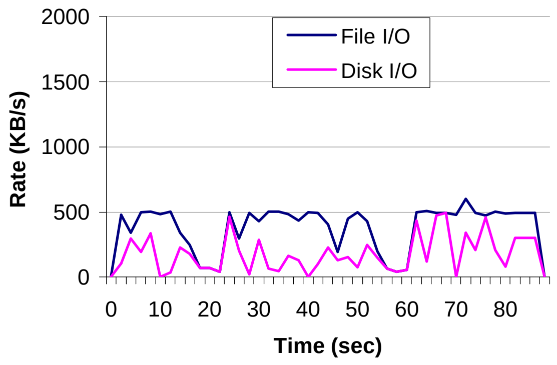
We investigated this approach by adding another field in the I/O rate window to record the resulting disk I/O size. A rolling average of the dilation factor is used to scale the file I/O threshold for future requests.

The full story of the I/O dilation is seen when we look at the time varying behavior of the I/O. Figure 33 shows the average I/O rates for the compile workload. The dark curve of each graph is for the file I/O rate and the light curve for the disk I/O rate. We first ran it without any I/O rate limit. Figure 33(a) shows that file I/O requests resulted in much less disk I/O because many header files were reused from the file buffer cache. The second graph (b) presents the case when we limited the file I/O rate to 500 KB/sec. Notice that although this workload still has considerable hits in the file buffer cache, our mechanism ensured that the actual disk I/O rate was less than the target rate of 500KB/sec. The requested I/O rate peaks are higher than our target limit, due to the fact that we average I/O requests over an effective 1.7 second window (as noted above) and we are showing data over a 1 second window in this figure.

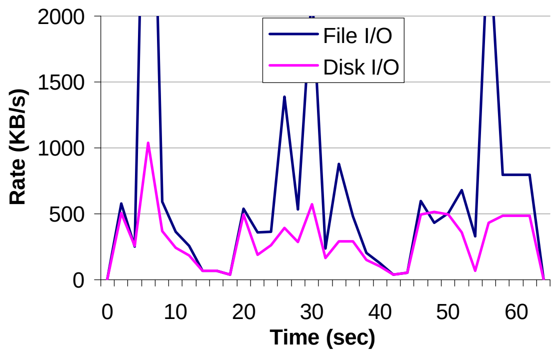
Figure 33(c) shows the behavior of the compile application when the dilation factor is used to control the disk I/O rate. The curves demonstrate that the application can take advantage of buffer hits while limiting the disk I/O rate to a certain level. The compile application was able to finish in 64 seconds, which is 27 seconds earlier than using file I/O rate policing. Note that the disk I/O rate occasionally peaks over the limit. This is because the dilation factor is derived from past I/O behavior. Any change in the dilation factor over time can cause inaccurate predictions. Overall, however, the actual disk I/O followed the limit quite well.



(a) No rate limit



(b) File I/O limit (500 KB/s)



(c) Disk I/O limit (500 KB/s)

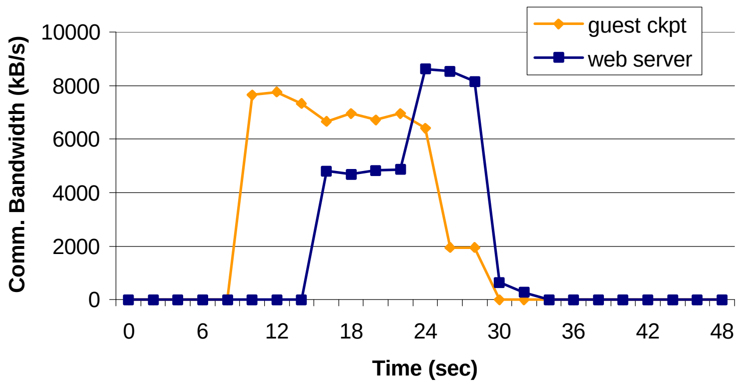
Figure 33: File and disk I/O rates for the compile workload.

5.3.3 Network I/O policing

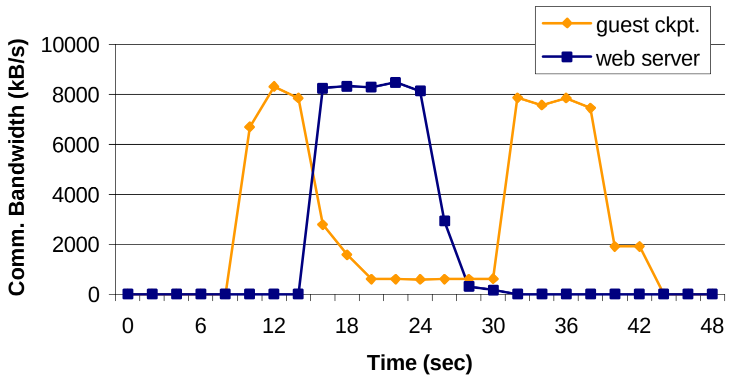
Policing network I/O is easier than file I/O because there is no analogue to the file buffer cache or read ahead, which dilate and attenuate the effective disk I/O rate. In this section, we present an application of network I/O throttling using our rate windows.

Most of the experiments in Section 5.3.2 assumed the use of rate windows in a Linger-Longer context. We ran one additional Linger-Longer experiment, this time with network I/O as the target. One of the main complaints about Condor and similar systems is that the act of moving a guest job from a newly loaded host often induces significant overhead to retrieve the application's checkpoint. Further, periodic check-pointing for fault tolerance produces bursty network traffic. This experiment shows that the rate windows are able to throttle even the checkpoint and prevented it from affecting host jobs.

Figure 34 shows two instances of a guest process moving off of a node because a



(a) Without network policing



(b) With network policing

Figure 34: Guest job checkpoint vs. host web server

host process suddenly becomes active. Moving off the node entails writing a 90MB checkpoint file over the network. This severely reduces available bandwidth for the host workload (a web server[7] in this case) in the unthrottled case shown in Figure 34(a). Only after the checkpoint is finished does the web server claim most of the bandwidth.

In the throttled case shown in Figure 34(b), the condor daemon's network write of the checkpoint consumes a majority of the bandwidth only until the host web server starts up. At this point, the system enters throttling mode and the bandwidth available to the checkpoint is reduced to the guest class's threshold. Once the web server becomes idle again, the checkpoint resumes writing at the higher rate.

In this section, we have presented a simple and portable mechanism that allows an operating system to throttle the rate at which disk and network communication is performed. Our experiments demonstrated that we are able to enforce these resource limits on applications with little overhead. For I/O bound applications, we are able to enforce limits at the physical device level despite the imposition of the buffer cache and disk-read ahead mechanisms. Further, for many applications, we can enforce our limits on the actual disk I/O instead of the file I/O by compensating for the file-to-disk dilation factor. The result is more efficient use of the guest job's allocated bandwidth. For the network case, we demonstrated that rate windows allow effective bandwidth sharing among communication-bound processes. We used them to implement policies that protect a host process's access to network resources. This protection is applied to all network accesses by all guest jobs that are running on the local machine, and also to the

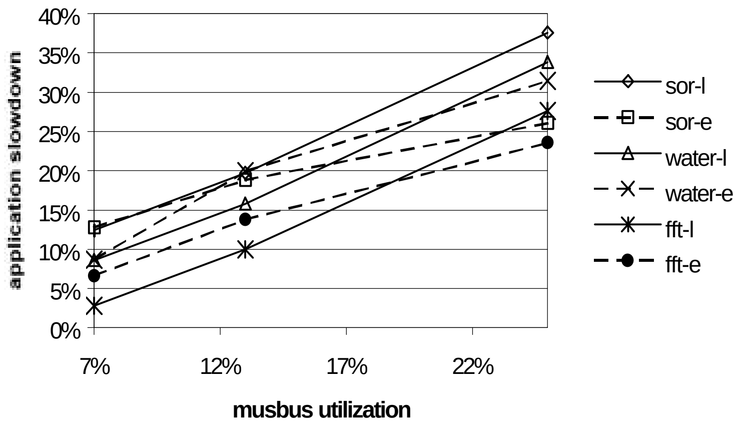
[7] The host process could be any network intensive process such as an FTP or a Web browser.

large network I/O's that occur when such processes try to migrate their address spaces off of the local machine.

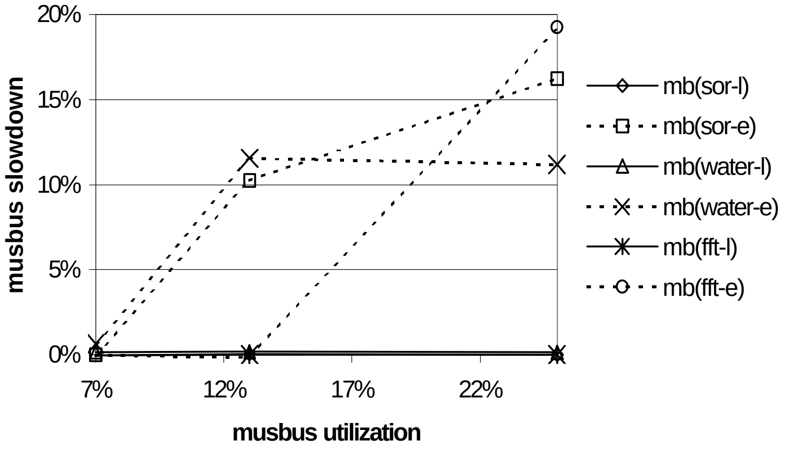
5.4 Application Validation

This section presents a study of Linger-Longer's effect on parallel applications on our test cluster. This cluster is comprised of eight 266-MHz Pentium II workstations running Linux 2.2.5-15, connected by a 1.2 Gbit/sec Myrinet and a 100 Mbit/sec switched Ethernet. We use the Musbus interactive UNIX benchmark suite [46] to simulate the behavior of actual interactive users. Musbus simulates an interactive user conducting a series of compile-edit cycles. The benchmark creates processes to simulate both interactive editing (including appropriate pauses between keystrokes), UNIX command line utilities, and compiler invocations. We varied the size of the program being edited and compiled by the "user" in order to change the mean CPU utilization of the simulated local user. In all cases, the file being manipulated was at least as large as the original file supplied with the benchmark.

The guest applications are Water and FFT from the Splash-2 benchmark suite [73], and SOR, a simple red-black successive over-relaxation application [4]. As mentioned in Chapter 4, Water is a molecular dynamics code, while FFT implements a three-dimensional Fast Fourier transform. All three applications are run on top of CVM [38], a user-level DSM system. These three applications are intended to be representative of three common classes of distributed applications. Water has relatively fine-grained communication and synchronization, FFT is quite communication-intensive, while SOR is mostly compute-bound.



(a) slowdown of parallel application (guest process)



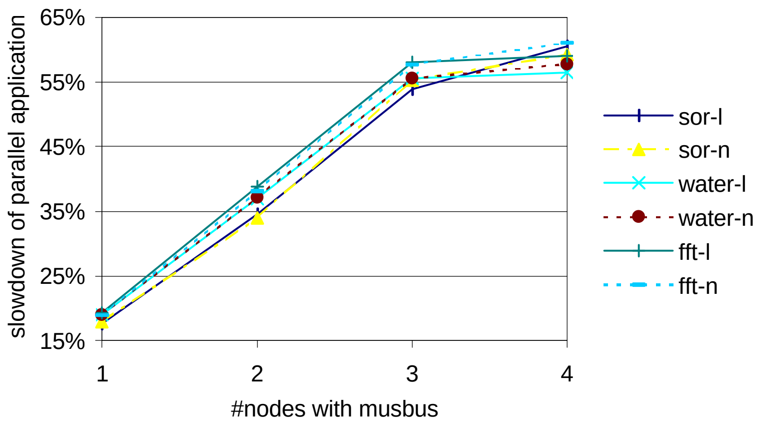
(b) slowdown of host Musbus process

Figure 35: Impact of running one process of four-process CVM applications as a guest process.

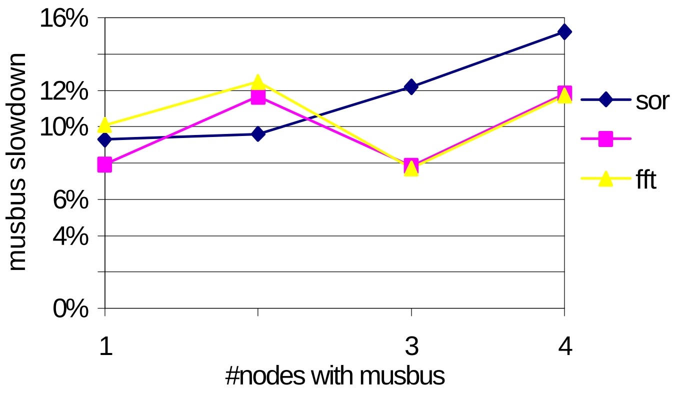
In the first set of experiments, we ran one process of a four-process CVM application as a guest process on each of four nodes. We varied the mean CPU utilization of the host processes from 7% to 25% by changing the size of the program being compiled during the compilation phase of the benchmark. The results of these tests are shown in Figure 35. The left graph shows the slowdown experienced by the parallel applications. The solid lines show the slowdown using our Linger-Longer policy, and the dashed lines show the slowdown when the guest processes are run with the default (i.e., equal priority). As expected, running the guest processes at starvation level priority generally slows them down more than if they were run at equal priority with the host processes. However, when the Musbus utilization is less than 15% the slowdown for all applications is lower with lingering than with the default priority. For comparison, running sor, water, and fft on three nodes instead of four slows them down by 26%, 25%, and 30%, respectively. Thus for the most common levels of CPU utilization, running on one non-idle node and three idle would improve the application's performance compared to running on just three idle nodes. Our simulations in Chapter 4 showed that node utilization of less than 10% occurs over 75% of the time even when users are actively using their workstations.

The right side of Figure 35 shows the slowdown experienced by the host Musbus processes. Again, we show the behavior when the guest processes are run using our Linger-Longer policy and the default equal priority. For all three parallel guest applications, the delay seen when running with Linger-Longer was not measurable. However, when the guest processes were run with moderate CPU utilization (i.e., over 10%), all three guest processes started to introduce a measurable delay in the host processes when equal priority was used. For Water and SOR, the delay exceeds 10% when the Musbus utilization reaches 13%. At the highest level of Musbus CPU utilization, the delay using the default priority exceeds 10% for all three applications and 15% for two of the three applications.

Figure 36(a) shows the impact on the CVM applications when competing with a host Musbus application running on more than one of the four nodes. The applications slow remarkably uniformly until four nodes, at which point the slowdown levels off.



(a) Slowdown of parallel application (guest process)



(b) Slowdown of host processes for *equal* priorities

Figure 36: Impact of differing number of CVM nodes having host process

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

Figure 36(b) shows the slowdown of the Musbus processes with guests at equal priorities as the number of nodes with Musbus processes increases. We do not show the linger and nice cases because there is no slowdown. The fact that there is any change in the slowdown of purely sequential applications as the number of non-idle nodes changes is rather unintuitive. This is explained by the fact that the behavior of the parallel application (which does interact with all nodes) changes as the number of non-idle nodes changes, and the guest parallel processes compete for the processors with the sequential processes. In particular, increasing the number of host processes decreases the efficiency of the parallel DSM applications. They use more cycles, and therefore impose more of a load on their host processors.

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

In this chapter, we provided the operating system support for the Linger priority for guest jobs. We developed the starvation-level low CPU priority and memory usage limits for guest jobs. Also, we developed the Rate windows mechanism for efficient communication and file I/O throttling. We demonstrated that these mechanisms effectively limit the resource usage by guest jobs on non-idle nodes and minimize the slowdown of host jobs. Although all these mechanisms were developed for our Linger-Longer system, they are general enough to serve other types of use. CPU and memory priority mechanisms can provide an ultra-low job priority augmenting the nice command in Unix. Our communication and I/O throttling mechanisms can support lightweight rate-based bandwidth scheduling which can bound maximum bandwidth usage.

Chapter 6

System Prototype and Performance

In this chapter, we first describe a prototype of the Linger-Longer system. Operating system kernel extensions for the Linger priority, described in the previous chapter, have been integrated into this prototype. We also implemented a new adaptive migration manager as a user-level daemon process. Then, using this prototype, we compare overall performance between the fine-grain cycle stealing policies and the coarse-grain cycle stealing policies in an eight machine Linux cluster.

6.1 Prototype Implementation

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

The overall prototype of the Linger-Longer system is depicted in Figure 37. The leveraged Condor modules are as follows[8].

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

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

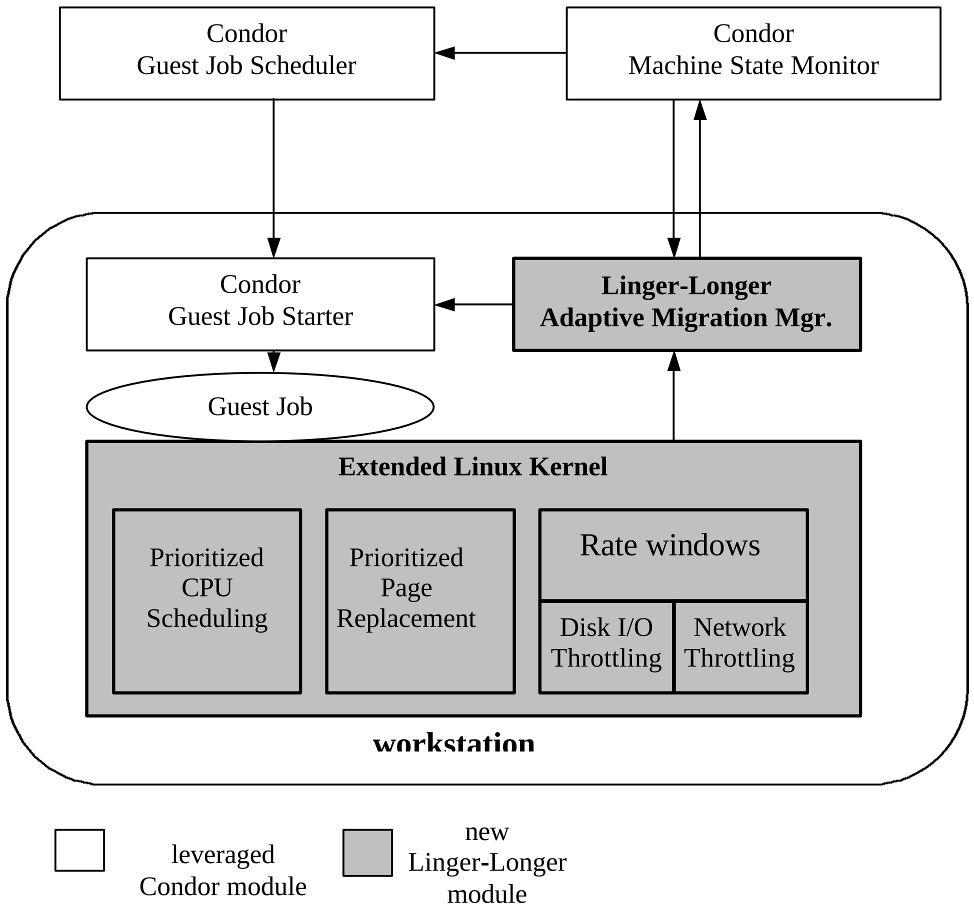


Figure 37: Linger-Longer system prototype

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

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

We also leveraged the Condor job migration mechanism which is based on checkpointing. In Condor, a job starts migration by dumping the current process image to the checkpoint server. Then, the image is transferred to the destination machine and the execution is resumed where it checkpointed at the source machine. Although direct transfer between the source and the destination can be more efficient, centralized checkpointing has an advantage of allowing a uniform mechanism for checkpointing and job migration.

Two modules were added for Linger-Longer (shown in gray boxes in Figure 37). The first module contains the operating system extensions for the Linger priority. In the previous chapter, we already described the policies and detailed implementation mechanisms for the Linger priority for guest jobs. To summarize, we implemented a starvation-level priority for CPU, prioritized page replacement for memory and Rate windows for efficient I/O and network throttling.

The second module is the Adaptive Migration Manager. This module replaces the existing migration policies of Condor with our cost/benefit based migration scheme. As described in Chapter 3, a guest job can linger even when the machine where the job is running becomes non-idle and migrate to another machine only when the benefit outweighs the migration cost. To measure the required parameters for Linger-Longer migration, another resource state monitor has been integrated in this module. The resource monitor can measure current CPU usage in percentage and physical memory usage for local jobs and guest jobs.

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

Adaptive Migration Manager directly forces Condor to migrate a guest job by invoking the *condor_vacate* command.

The Linger-Longer system, which was extended from Condor, with new modules and Linux kernel extensions, is currently operational. The system also can support the traditional Immediate-Eviction and Pause-and-Migrate policies as well as our new Linger-Longer and Linger-Forever policies.

6.2 Experiment Setup

In Chapter 4, using cluster simulation, we estimated the potential performance gain of fine-grain cycle stealing. Now we conduct the similar experiments using the Linger-Longer prototype in a real Linux cluster. We first describe the setup for a network of workstations, host job workload and guest job sets used in this experiment.

We run the Linger-Longer system in an eight machine Linux cluster. Each machine in the cluster has a 233 MHz Pentium II processor, 192 MB of memory and a 6 GB IDE hard disk. All the machines are connected by two networks, a 100 Mbps switched Ethernet and a 1 Gbps Myrinet switch. While a Gigabit Myrinet was used for the parallel job experiments in Chapter 5, we use only the Ethernet switch for the experiments with sequential guest jobs in this chapter. The machines are running the Linux 2.2.5 kernel with our Linger priority kernel extensions. We set each guest limit parameter to approximately 10% of the total resource: 20 Mbytes for high memory limit, 10 Mbytes for low memory limit, 500 Kbytes/sec for both disk I/O and network bandwidth.

Modeling an interactive user workload on a personal machine is very difficult if not impossible. Also, having real computer users use the test machines is not feasible

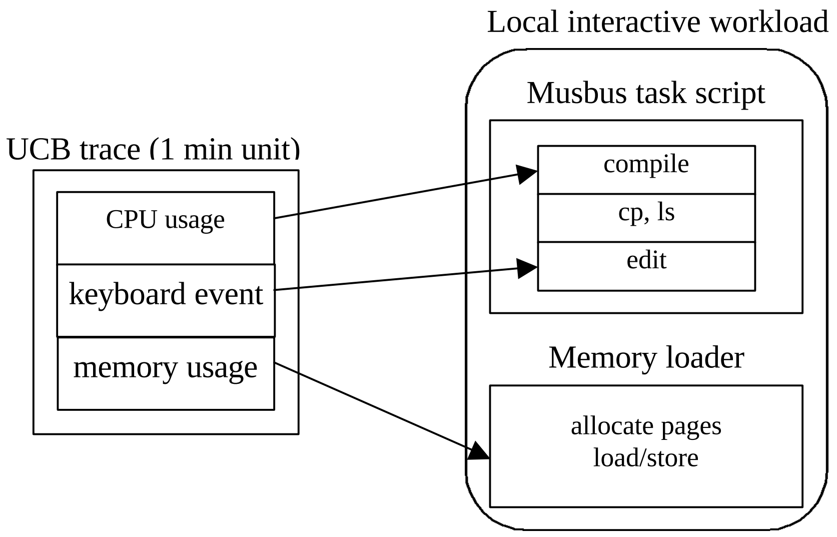


Figure 38: Interactive local workload (host jobs) generation

since the workload cannot be accurately reproduced to allow comparisons of different policies. Therefore, we generated the local workload based on the trace data which was used for the simulations (UC Berkeley trace). However, in this experiment, a resource usage from the trace invokes a corresponding task script. We used two scripts to simulate interactive users and to consume memory resources. First, Musbus scripts were used to emulate an computer programmer. This script is a sequence of subtasks such as editing (`ed`), compiling (`make` and `gcc`), copying (`cp`) and file listing (`ls`). The mapping between resource usage and script based interactive tasks are shown in Figure 38. Every minute, a new local task is generated. For CPU usage and keyboard activity, a corresponding Musbus based script is selected and executed. The number of files to be compiled is adjusted to generate 1 minute of CPU usage. An editing subtask is invoked for the given keyboard activity duration. Second, to emulate local memory usage, a simple memory loader program runs separately from the Musbus task. It allocates the corresponding size of memory, and loads and stores to the various memory locations.

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

Table 6: Guest job sets with various job size and duration

As guest jobs, we use a set of scientific applications from the NAS NPB benchmark [10]. A serial version of the benchmark was selected since, in this experiment, we focus on sequential guest jobs. We chose three applications with three different data sizes: mg.W, sp.A and lu.B, which require 8, 65 and 165 MB memory respectively (A job name was denoted as *applicaton_name.data_size*). We also varied job duration by changing the number of iterations. Various combinations of job size and duration are shown in Table 6. Among 9 combinations, we selected 5 representative types of guest jobs. Also, we group a number of identical jobs into a job set. The job set size was set such that all guest job sets could finish in a similar time although individual jobs require different CPU time. This was intended to minimize the performance changes due to changes in local resource usage over time. We chose 1 cluster hour for the job set duration. Hence, the job set size is 480 for 1 minute jobs, 320 for 1.5 minute jobs, 48 for 10 minute jobs and 16 for 30 minute jobs.

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

- Guest job set 1 (small size, short time): mg.W.1m.480
- Guest job set 2 (small size, long time): mg.W.30m.16

- Guest job set 3 (medium size, medium time): sp.A.10m.48
- Guest job set 4 (large size, short time): lu.B.1.5m.320
- Guest job set 5 (large size, large time): lu.B.30m.16

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

6.3 Performance Comparison

Finally, we present a head-to-head performance comparison between our fine-grain cycle stealing and traditional coarse-grain cycle stealing. The experiments ran 5 different guest job sets with 4 different policies (LL, LF, IE and PM) in an eight machine Linux cluster. For each configuration, we computed the four metrics: average job time, variation, family time and throughput. These metrics were defined for simulation study in Chapter 4. Also, host workload delay for each experiment will be presented later in this section.

We first analyze the cluster performance for the guest job sets. The results are summarized in Table 7. Notice that the column with the policy Idle is used as the base case where guest jobs were run on a fully idle 8 machine cluster.

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

Table 7: Guest performance for different job sets

For the average job time, LF is 60% to 70% better than IE for most guest job sets. However, in the case of lu.B.1.5m.320, the gain decreases to around 50%. This is due to the fact that large size guest jobs occasionally struggled to get enough memory on some non-idle machines (recall that lu.B requires 165 MB of memory where the total memory of each machine is 192 MB). LF performs slightly better than LL since it can consume almost all the available cycles from both idle and non-idle machines and hence reduce the waiting time in queue.

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

The number of migrations is also measured for the different policies and guest job sets. For the short duration guest jobs (mg.W.1m and lu.B.1.5m), LL reduces migrations significantly since migrations are often non-beneficial for those jobs. For the large duration jobs (mg.W.30m and lu.B.30m), LL migrates almost the same number of guest jobs as PM. PM migrates less guest jobs than IE since PM can avoid unnecessary migrations when the current machines returns to the idle state quickly.

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

The average delay for each guest job set is shown in Table 8. Musbus without guest jobs produced some delay (0.68%) as shown in the first column of the table. It means that the 0.68% delay can be just a noise. Interestingly, for small and medium size guest jobs, LL and LF exhibits less delay than PM and IE. A histogram of Musbus delay with mg.W.30m.16 is shown in Figure 39. We can observe that, for PM and IE, there exist more delays between 6% and 12% than LL and LF. These delays were caused by job migration in PM and IE due to lack of resource prioritizing mechanisms. In LL and LF, migration itself uses only idle resources at the Linger priority. For large size guest jobs (lu.B), LL and LF shows more Musbus delay than PM and IE. The delay histogram for the guest job set lu.B.30m.16 is shown in

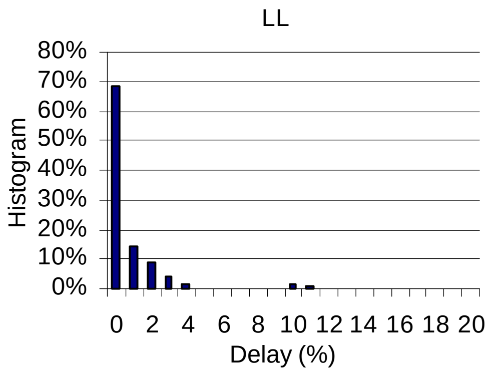
Figure 40. In the histograms for LL and LF, there are some instants whose delay exceeded 20%. These occasional delays were caused since Musbus and the large memory guest jobs were running at the same time. However, no such noticeable delays were detected for Memload (Figure 41). This demonstrates that current prioritized memory

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

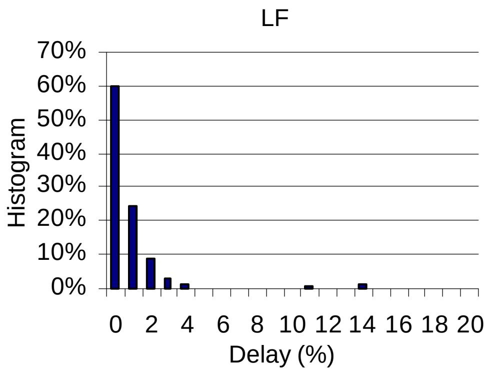
Table 8: Host job slowdown for different guest job sets

replacement effectively protected the memory pages of Memload. On the contrary, Musbus was occasionally delayed since our memory replacement mechanism could not prevent a large memory guest job from sweeping the file buffer cache (recall that Musbus contains compile workload). However, despite that lu.B is very aggressive in using memory, these noticeable delays occurred only for less than 5% of the time.

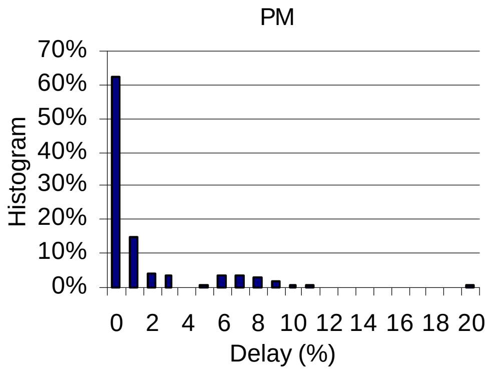
Although all the metrics for this experiment almost match the estimated performance gain in simulation study in Chapter 4, the throughput (the equivalent number of idle machines) does not. In the prototype experiment, the equivalent idle machines are 4 to 5 on an 8 machine cluster (50% to 62%) whereas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



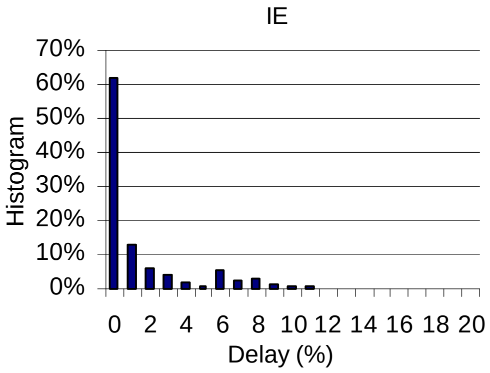
(a)



(b)

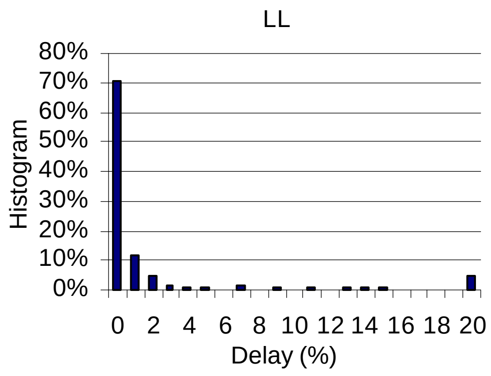


(c)

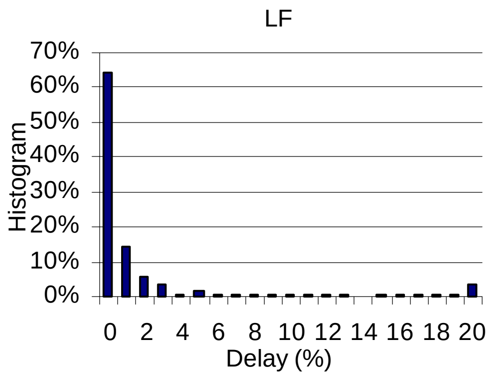


(d)

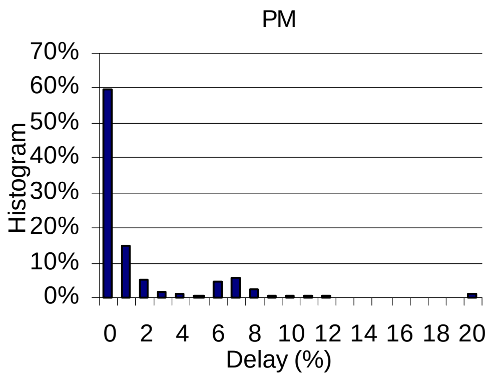
Figure 39: Musbus delay for mg.W.30m.16



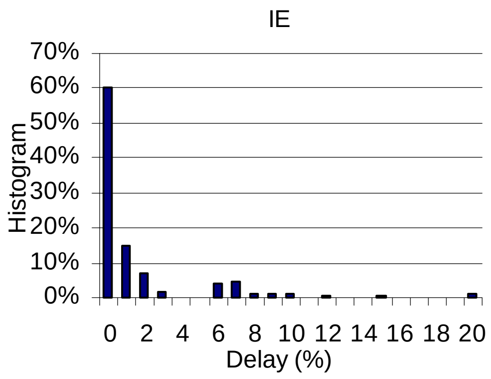
(a)



(b)

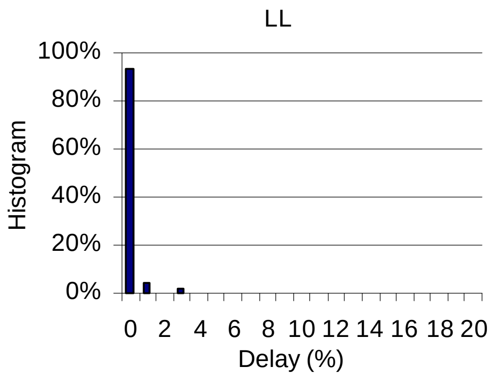


(c)

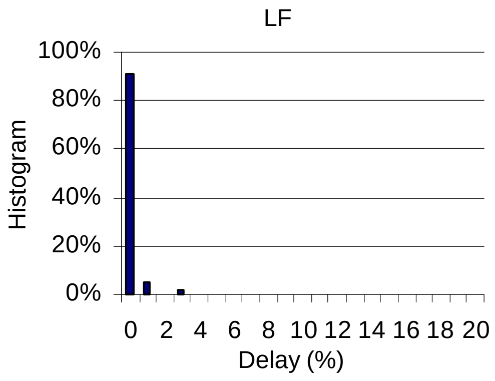


(d)

Figure 40: Musbus delay for lu.B.30m.16



(a)



(b)

Figure 41: Memload delay for lu.B.30m.16

In this chapter, we described the implementation of the Linger-Longer system. The operating system extension for starvation-level CPU priority, prioritized memory replacement and Rate widows for I/O and network throttling have been integrated to the prototype. Also, a new adaptation migration module was implemented as a daemon process. For the guest job scheduling, migration and checkpointing mechanisms, we leveraged the Condor system. Using the prototype, we conducted the experiments on an eight machine Linux cluster. We ran five different guest job sets with various sizes and durations. The local workload was generated using Musbus, a script based interactive workload. The configuration of each instance of Musbus was driven by the resource usages of the UC Berkeley trace. The results demonstrated that LL and LF can improve the cluster throughput by 50% to 70% for most cases. However, this gain can be reduced by 10 to 20% if we run guest jobs that require more than 50% of total memory of the machine. For all the cases, the average host job slowdown didn’t exceed 3%.

Chapter 7

Conclusions and Future Work

7.1 Summary

The thesis of this research is to maximize the efficiency of computing in networks of workstations while maintaining unobtrusive use of idle resources. So, we run resource-intensive batch jobs to use networks of private workstations more efficiently. Previous systems such as Condor, LSF and NOW takes a conservative approach. They harvest only coarse-grain idle periods when users are away from their workstations. Their overly conservative estimates of resource contention is to avoid disturbing machine owners.

We take a different approach. Our Linger-Longer system exploits all available idle cycles even when owner tasks are active. By analyzing trace data from different sources, we demonstrated that even while machines are in use, resource usage is very low most of the time (CPU usage is below 10% for 76% of the time and a half of memory is free for 70% of the time). However, we need to unobtrusively use those idle resources. Upon the return of a machine user, instead of moving a guest job out, our system lowers the guest job's resource priority and keeps harvesting idle cycles at fine granularity. Thus, guest jobs run in background without causing noticeable delays for machine owners.

However, current operating systems do not fully support this type of strict resource differentiation between host jobs and guest jobs. Our experiments showed that the CPU priority such as Unix nice is not sufficient for our purpose. Therefore, we developed prioritization mechanisms for all the major resources, CPU, memory, disk I/O and network bandwidth. For each mechanism, we validated its efficacy using a series of experiments.

We evaluated our new policy using both simulations and the experiments with a prototype. Two important questions for evaluation are

1. How many idle cycles can our policy harvest in most networks of workstations? (efficiency)
2. How much can we limit the slowdown that our policy introduce to machine owners' task? (unobtrusiveness)

We first conducted trace-driven cluster simulations and compare the performance of traditional coarse-grain cycle stealing (Immediate-Eviction and Pause-and-Migrate) and our fine-grain cycle stealing (Linger-Longer and Longer-Forever). The results showed that Linger-Longer can harvest 60% more idle cycles than traditional approaches while limiting host workload slowdown to a few percents. We also showed that guest parallel jobs can perform better with fine-grain cycle stealing than with run-time reconfiguration.

Finally, we implemented a prototype of the Linger-Longer system in Linux. Several resource prioritization mechanisms were implemented as kernel extensions. Using this prototype, we conducted a head-to-head performance comparison between the Linger-Longer policy and traditional coarse-grain cycle stealing policies. The experiments with several benchmark applications proved that Linger-Longer can improve guest job throughput by 50% to 70%. Furthermore, the resource prioritization mechanisms successfully limited the average host job slowdown to 3%. Both trace driven simulations and the experiments with the prototype supported our claim; fine-grain cycle stealing significantly improves cluster throughput without a noticeable delay for machine owners.

7.2 Contributions

The contributions of this research is as follows

- We have devised a new fine-grain cycle stealing policy, Linger-Longer.
- We have developed an adaptive migration scheme that migrate a job only when the benefit outweighs the cost.
- We have developed an accurate cluster simulator that can generate local load in a microsecond resolution.
- With simulations, we have demonstrated that Linger-Longer can improve the cluster throughput by up to 60% with only a few percent of host job delay.
- We have shown that parallel jobs also can take advantage of fine-grain cycle stealing by avoiding costly reconfigurations.
- We provided a suite of operating system extensions to limit guest jobs' resource usage: starvation-level priority support for CPU, limit based prioritized page replacement for memory, and Rate Windows to bound I/O and network bandwidth usage. Although these mechanisms were designed to support fine-grain cycle stealing. They are individually useful to support run-time resource limiting in general.
- We have implemented a prototype of the Linger-Longer system in Linux. The system is currently operational on an experimental Linux cluster.

- Using the prototype, we have shown that Linger-Longer can exploit 50% to 70% more idle cycles with a only 3% average host job slowdown.

7.3 Future Work

There are many interesting ways to extend our research. Several issues are listed below.

General performance-aware migration:
In this thesis, we have developed an adaptive migration scheme. Guest jobs are migrated only when the benefit of running faster at the destination outweighs the cost of being suspended during the job image transfer. To estimate the performance benefit we considered only CPU availability at the source and the destination. As long as enough memory is available at both sides, the performance of computation-bound is proportional to CPU speed. However, many resource intensive jobs, such as data mining, are not always CPU intensive. So, the performance can be determined by availability of other resources such as memory or disk I/O bandwidth.

If we can parameterize the performance of such applications for all different resources, more accurate prediction of migration benefit will be possible. In other words, we want to have a performance model which can answer the question, “50% of CPU cycles, 100MB of memory, 500KBps I/O bandwidth and 1Mbps of network bandwidth to the node X is available. How would my application perform using them?” The performance model to answer to this question will not only refine our migration scheme but also improve all other run-time adaptation applications and systems.

Adaptive parallel applications:

In Chapter 4, we showed the guest parallel job performance in non-dedicated networks of workstations. Individual process migration can be disastrous in parallel computing since a single suspended process can stop the whole parallel application. This will result in wasting of idle cycles on the machines where the processes of the suspended parallel job are residing. Fine-grain cycle stealing partially solves this problem because parallel application processes do not need to be stopped or migrated. However, in some cases, evicting a process from a highly loaded non-idle machine can work better. A non-idle machine can be excluded in two ways. First, a process running on the non-idle node can be migrated to another idle machine. Second, the whole parallel application can be reconfigured to use fewer machines. We need to study the performance of these different options for various parameters such as local load and reconfiguration cost. The problem becomes more complicated when considering multiple parallel jobs running in the same cluster. In this case, we think that cluster throughput is more important than speed-up of individual parallel jobs. It is interesting and also challenging to study adaptation of multiple parallel jobs in a non-dedicated cluster.

Other uses of resource prioritization mechanisms:

We have developed several resource prioritization mechanisms to support Linger-Longer. However, the mechanisms are general enough to serve other types of use. CPU and memory priority mechanisms can provide ultra-low job priority augmenting the nice command in Unix. Our communication and I/O throttling mechanisms can support light-weight rate-based bandwidth scheduling which can bound maximum bandwidth usage. We plan to investigate the applications that can take advantage of our mechanisms.

Security issues in global resource sharing:

Security issues are critical in this area because untrusted jobs may be submitted to use idle. Linger-Longer is exposed to this problem more than traditional approaches since guest jobs can coexist with host jobs. Our resource prioritization mechanisms can protect the host jobs' performance, but not their security. Thus, it is possible that guest jobs can obtain useful information by monitoring the behavior of host jobs. Another security issue arises for guest jobs. Machine owners also can take secret information from guest jobs running on his/her machine. In the worse case, a guest process can be compromised and send back false results to the submitter. These issues become more serious today since many researches, generally called grid computing, try to extend resource sharing across multiple autonomous sites and even further to the Internet.

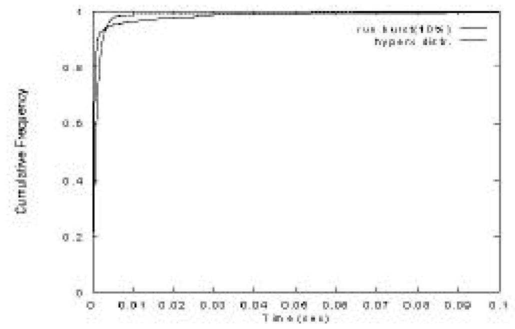
Appendix A:

Modeling of Fine-Grain CPU Run/Idle Intervals

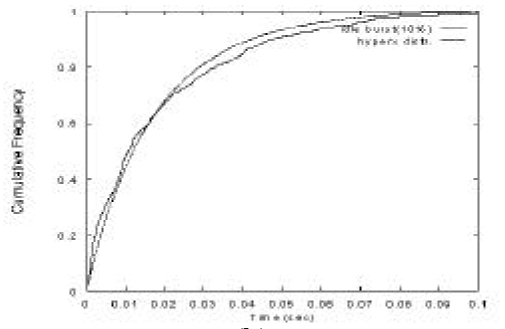
To simulate CPU utilization of a large number of machines at the dispatch level (process context switch), it is very difficult, if not impossible, to record all the scheduling events on each CPU for a long period of time. Therefore, we developed a stochastic model to generate fine-grain CPU requests (on the time scale of microseconds) from coarse-grain average CPU usage (on the time scale of seconds). We model processor activity as a sequence of run and idle periods that represent the intervals of time when the workstation owner's processes are either running or blocked. Since scheduling policy we are simulating gives priority to *any* request by one of the local processes, a single run burst may represent the dispatching and execution of several local processes. Also, there is no upper bound on the length of a processor request since we aggregate multiple consecutive dispatches due to time quanta into a single request.

To gather the fine-grained workload data, we used the tracing facility available on IBM's AIX operating system to record scheduler dispatch events. We gathered this data for several twenty-minute intervals on a collection of workstations in the University of Maryland, Computer Science Department. We then processed the data to extract different levels of utilization, and characterized the run-idle intervals for each level of utilization. Because we treated the CPU as an on/off source and didn't consider the scheduling of individual processes, OS specific factors such as scheduling policy and time quanta would not impact the characteristic of the trace data.

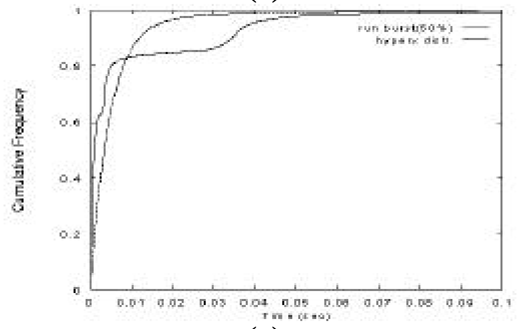
We divided utilization into 21 buckets ranging from 0% to 100% processor utilization. For each of the 21 utilization levels, we created a histogram of the duration of run and idle intervals for all two-second intervals whose average utilization was closest to that bucket. A selection of these histograms is shown in Figure 42. The solid line in the figure shows three sample distributions for low (10%), medium (50%), and high (90%) CPU utilization for run and idle bursts, respectively. Based on the analysis of



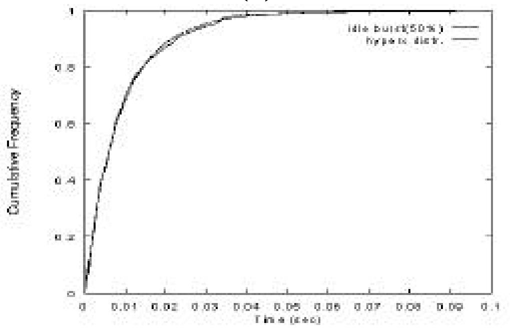
(a)



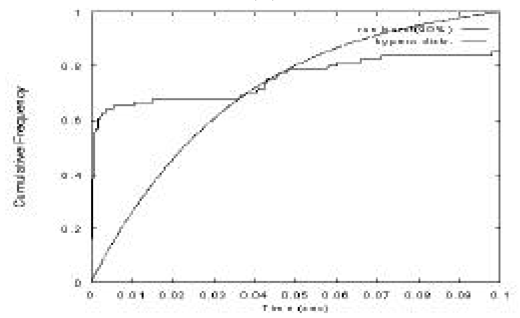
(b)



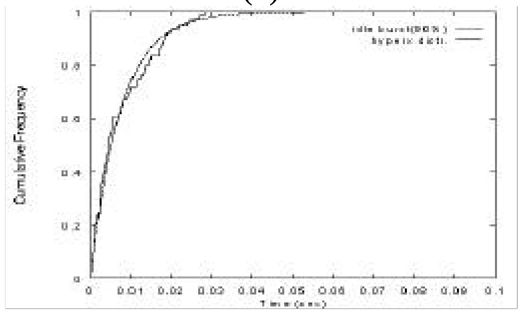
(c)



(d)



(e)



(f)

Figure 42: Run and idle burst histograms

The CDF(Cumulative Distribution Function) for the run and idle duration of local jobs on workstations. The first row is for 10% utilization, the second 50%, and the third 90%.

the data, we model the fine-grained processor utilization by two random variables: run burst duration and idle duration. For each of these variables, we generate a two-stage hyper-exponential distribution from the mean and variance using a method-of-moment estimate [68 pg. 479].

The dashed line in Figure 42 shows the CDF of the random variable selected to model the run and idle intervals at the three levels of utilization. The curves almost exactly match in idle burst distributions, but the model and observations diverge for run burst distributions with medium to high CPU utilization. However, since our scheduling policy uses average utilization to make migration decisions, and the overhead associated with guest jobs is a function of the number of context switch operations, the most important parameters for our simulation are the average CPU utilization and the *number* of run bursts generated per unit of time.

Fortunately, the model we have selected accurately tracks both of these metrics. Figure 43 shows a comparison between the number of run bursts generated by our 2-

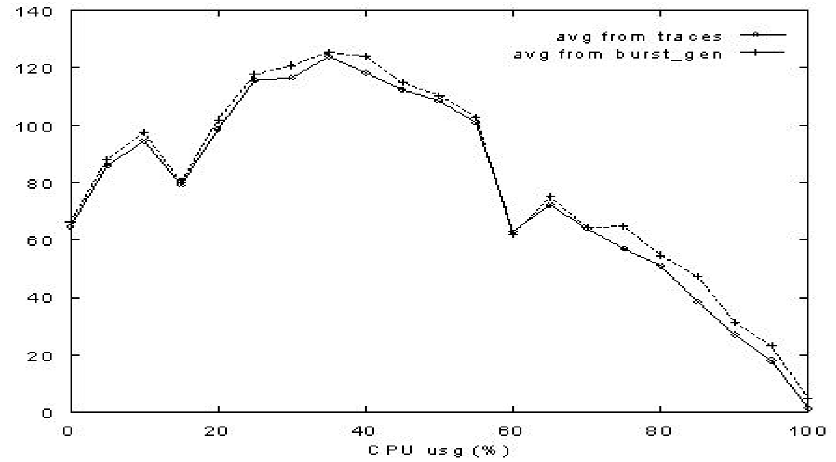


Figure 43: Number of run bursts

The solid line shows the number of run bursts (context switches) in our experimental vs. the local processor load. The dashed curve shows the same data generated by our model.

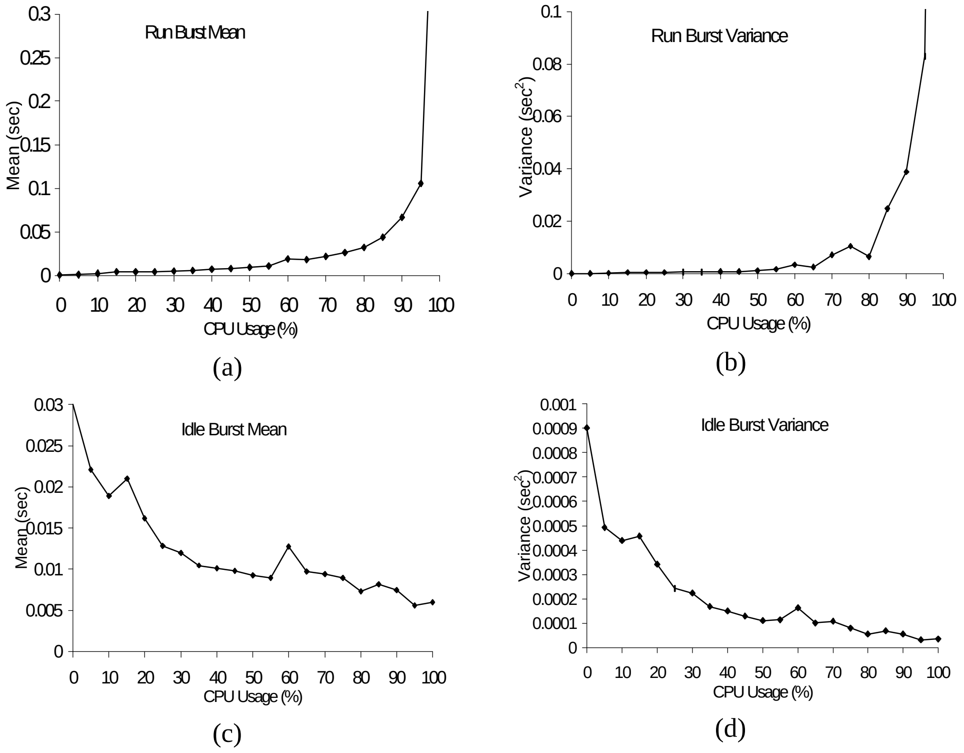


Figure 44: Workload parameters as a function of processor utilization

The mean and variance of the run and idle bursts seen in the fine-grained workload traces as a function of the processor utilization.

stage hyper-exponential distributions and those measured from traces. The data show that at all levels of processor utilization, the number of context switches generated by our model tracks those seen in actual machine execution.[9]

To generate fine-grained workloads, we use linear interpolation between the two closest of the 21 levels of utilization. The values we derived from our analysis of the dispatch records are shown in Figure 44. The top-left curve shows the mean value of the run burst duration as a function of processor utilization. The upper-right graph shows the variance in the run burst duration as a function of processor utilization. The

[9] The dips at 15% and 60% in Figure 43 came from the empirically observed workload distribution in Figure 4 (c), (d).

bottom two graphs in Figure 44 show the idle duration mean and variance, respectively. The exact mean and variance values of CPU run and idle intervals are shown in Table 9.

Combined with coarse-grain CPU usage traces, this stochastic model for run/idle intervals was successfully used to drive our long-term workstation cluster simulations. We believe our two-level load generation technique is useful for other researches that require long-term simulation of detailed resource utilization.

2 sec usage (%)	run mean (sec)	run var. (sec^2)	Idle mean (sec)	Idle var. (sec^2)
0	0.000427	0.000001	0.030033	0.000902
5	0.000974	0.000019	0.022065	0.000493
10	0.002147	0.000107	0.018916	0.000439
15	0.004016	0.000318	0.021012	0.000458
20	0.004009	0.000348	0.016168	0.000342
25	0.004402	0.000301	0.012858	0.000243
30	0.005140	0.000512	0.011972	0.000223
35	0.005624	0.000438	0.010481	0.000168
40	0.006774	0.000674	0.010091	0.000150
45	0.007988	0.000725	0.009778	0.000130
50	0.009149	0.000996	0.009279	0.000110
55	0.010833	0.001521	0.008963	0.000114
60	0.019013	0.003264	0.012757	0.000163
65	0.017979	0.002477	0.009690	0.000101
70	0.021870	0.007030	0.009414	0.000108
75	0.026148	0.010449	0.008940	0.000080
80	0.031944	0.006470	0.007320	0.000055
85	0.044013	0.024730	0.008147	0.000069
90	0.066808	0.038747	0.007450	0.000056
95	0.105832	0.083109	0.005590	0.000032
100	0.652647	0.556488	0.005968	0.000036

Table 9: Mean and variance for CPU run/idle intervals

BIBLIOGRAPHY

[1] A. Acharya, R. E. Bennett, M. D. Beynon, B. Moon, M. Uysal, A. Sussman, J. K. Hollingsworth, and J. Saltz, "Tuning the Performance of I/O Intensive Applications," *ACM /IEEE Workshop on I/O in Parallel and Distributed System*. May 1996, Philadelphia, PA, pp. 15-27.

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

[3] A. Acharya and S. Setia, "Availability and Utility of Idle Memory in Workstation Clusters," *ACM SIGMETRICS*. June 1999, Atlanta, GA, vol.27, pp. 35-46.

[4] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, February 1996.

[5] T. E. Anderson, D. E. Culler, and D. A. Patterson, "A case for NOW (Networks of Workstations)," *IEEE Micro*, **15**(1), 1995, pp. 54-64.

[6] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, *Dome: Parallel programming in a heterogeneous multi-user environment*, CMU-CS-95-137, Carnegie Mellon University, March 1995.

[7] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, "The Interaction of Parallel and Sequential Workloads on a Network of Workstations," *SIGMETRICS*. May 1995, Ottawa, pp. 267-278.

[8] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick., "Cluster I/O with River: Making the Fast Case Common.," *IOPADS '99*. May 1999, Atlanta, Georgia.

[9] A. Atlas and A. Bestavros, "Design and implementation of statistical rate monotonic scheduling in KURT Linux," *Proceedings 20th IEEE Real-Time Systems Symposium*. Dec. 1999, Phoenix, AZ, pp. 272-6.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, and D. S. Browning, "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications*, **5**(3), 1991, pp. 63-73.

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

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

[13] J. Basney and M. Livny, "Improving Goodput by Co-scheduling CPU and Network Capacity," *International Journal of High Performance Computing Applications*, **13**(3), 1999.

[14] F. Berman and R. Wolski, "Scheduling from the Perspective of the Application," *the 5th Symposium on High Performance Distributed Computing*. Aug. 1996, Syracuse, NY.

[15] S. N. Bhatt, F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg, "On Optimal Strategies for Cycle-Stealing in Networks of Workstations," *IEEE Transactions on Computers*, **46**(5), 1997, pp. 545-557.

[16] R. D. Blumofe and P. A. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," *USENIX Annual Technical Conference*. 6-10 Jan. 1997, Anaheim, CA, USA, pp. 133-47.

[17] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz, "The Eclipse operating system: Providing Quality of Service via Reservation Domains," *USENIX Annual Technical Conference*. June 1998, New Orleans, Louisiana.

[18] E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalabe Multiprocessors," *SOSP*. Oct 1997, pp. 143-156.

[19] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky, "Adaptive parallelism and Piranha," *IEEE Computer*, **28**(1), 1995, pp. 40-59.

[20] J. Casas, D. L. Clark, P. S. Galbiati, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole, "MIST: PVM with Transparent Migration and Checkpointing," *Annual PVM Users' Group Meeting*. May 7-9, 1995, Pittsburgh, PA.

[21] R. B. Dannenberg and P. G. Hibbard, "A Butler Process for Resource Sharing on Spice Machines," *ACM Transactions on Office Information Systems*, **3**(3), 1985, pp. 234-52.

[22] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software-Practice and Experience*, **21**(8), 1991, pp. 757-85.

[23] D. Duke, T. Green, and J. Pasko, *Research Toward a Heterogeneous Networked Computing Cluster: The Distributed Queueing System Version 3.0*, Florida State University, May 1994.

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

[25] G. Edjlali, G. Agrawal, A. Sussman, and J. Saltz, "Data Parallel Programming in an Adaptive Environment," *Proceedings of the Ninth International Parallel Processing Symposium*. April 1995, pp. 827--832.

[26] T. Faber, L. H. Landweber, and A. Mukherjee, "Dynamic Time Windows: packet admission control with feedback," *SIGCOMM*. Sept 1992, pp. 124 - 135.

[27] D. G. Feitelson and L. Rudolph, eds. *Parallel Job Scheduling: Issues and Approaches*. Lecture Notes in Computer Science. Vol. 949. 1995, Springer-Verlag LNCS.

[28] D. G. Feitelson and A. M. a. Weil, "Utilization and Predictability in Scheduling the IBM SP2 with Backfilling," *2th Intl. Parallel Processing Symposium*. April 1998, Orlando, Florida, pp. 542-546.

[29] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Intl J. Supercomputer Applications*, **11**(2), 1997, pp. 115-128.

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

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

[32] D. Gelernter, N. Carriero, S. Chandran, and S. Chang, "Parallel programming in Linda," *International Conference on Parallel Processing*. March 1985, pp. 255-263.

[33] R. P. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer Magazine*, **7**(6), 1974, pp. 34-45.

[34] T. Green and J. Snyder, *DQS, A Distributed Queueing System*, Florida State University, March 1993.

[35] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *SIGMETRICS*. May 23-26, 1996, Philadelphia, PA, pp. 13-24.

[36] J. S. Heidemann and G. J. Popek, "File-system development with stackable layers," *ACM Trans. Computer Systems*, **12**(1), 1994, pp. 58-89.

[37] IBM, *IBM LoadLeveler: General Information.*, Kingston, NY, September 1993.

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

[39] P. J. Keleher, J. K. Hollingsworth, and D. Perkovic, "Exposing Application Alternatives," *19th International Conference on Distributed Computing Systems*. June 1999, pp. 384-392.

[40] P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," *International Conference on Distributed Computing Systems (ICDCS)*. May 1991, Arlington, TX, pp. 336-343.

[41] P. Krueger, T.-H. Lai, and V. A. Radiya, "Processor Allocation vs. job scheduling on hypercube computers," *11th International Conference on Distributed Computing Systems*. May 1997, Arlington, TX, pp. 394-401.

[42] W. E. Leland and T. J. Ott, "Loadbalancing heuristics and process behavior," *SIGMETRICS*. May 1986, North Carolina, pp. 54-69.

[43] M. J. Lewis and A. Grimshaw, "The Core Legion Object Model," *Proceedings of the 5th International Symposium on High Performance Distributed Computing*. 1996, Los Alamitos, CA, pp. 551-561.

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

[45] C. McCann and J. Zahorjan, "Scheduling Memory Constrained Jobs on Distributed Memory Parallel Computers," *ACM SIGMETRICS*. 1995, Ottawa, Ontario, Canada, pp. 208-219.

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

[47] P. Messina, "The Concurrent Supercomputing Consortium: year 1," *IEEE Parallel and Distributed Technology*, **1**(1), 1993, pp. 9-16.

[48] R. A. Meyer and L. H. Seawright, "A Virtual Machine Time-Sharing System," *IBM Systems Journal*, **9**(3), 1970, pp. 199-218.

[49] J. C. Mogul and A. Borg, "The effect of context switches on cache performance," *ASPLOS*. Apr. 1991, Santa Clara, CA, pp. 75-84.

[50] J. H. Morris, M. Satyanarayanan, M. H. Cnner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment," *Communications on ACM*, **29**(3), 1986, pp. 184-201.

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

[52] V. K. Naik, S. K. Setia, and M. S. Squillante, "Schedulint of large scientific applications on the distributed shared memory multiprocessor systems," *the 6th SIAM conf. Parallel Processing for Scientific Computing*. 1993, vol.2, pp. 174-178.

[53] B. C. Neuman and S. Rao, "Resource Management for Distributed Parallel Systems," *the 2nd Symposium on High Performance Distributed Computing*. July 1993, pp. 316-323.

[54] T. D. Nguyen, R. Vaswani, and J. Zahorjan, "Maximizing Speedup through Self-Tuning of Processor Allocation," *Proceedings of IPPS'96*. 1996.

[55] J. K. Ousterhaut, "Scheduling Techniques for Concurrent Systems," *the 3rd International Conf. on Distributed Computing Systems*. October 1982, pp. 22-30.

[56] D. Petrou, D. P. Ghormley, and T. E. Anderson, *Predictive State Restoration in Desktop Workstation Clusters*, CSD-96-921, University of California, November 1996.

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

[58] J. Pruyne and M. Livny, "Interfacing Condor and PVM to harness the cycles of workstation clusters," *Future Generation Computer Systems*, **12**(1), 1996, pp. 67-85.

[59] S. H. Russ, J. Robinson, B. K. Flachs, and B. Heckel, "The Hector distributed run-time environment," *IEEE Transactions on Parallel and Distributed Systems*, **9**(11), 1999, pp. 1102-14.

[60] S. Setia, "The interaction between memory allocation and adaptive partitioning in message-passing multicomputers," *Job Scheduling Strategies for Parallel Processing*. 1995, vol.949.

[61] S. Setia and S. Tripathi, *An Analysis of Several Processor Partitioning Policies for Parallel Computers*, CS-TR-2684, University of Maryland Dept. of Computer Science, May.

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

[63] P. Sobalvarro and W. Weihl, "Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors," *Job Scheduling Strategies for Parallel Processing*. 1995, vol.949.

[64] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," *International Parallel Processing Symposium*. 1996, Honolulu, Hawai, pp. 526-531.

[65] M. M. Theimer and K. A. Lantz, "Finding idle machines in a workstation-based distributed system," *8th International Conference on Distributed Computing Systems*. June 1988, San Jose, CA, pp. 13-17.

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

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

[68] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. 1982: Prentice-Hall.

[69] A. Tucker and A. Gupta, "Process control and scheduling issues in multiprogrammed shared-memory multiprocessors," *the 12th Symposium on Operating System Principles*. December 1989.

[70] J. S. Turner, "New Directions in Communications (or Which Way to the Information Age?)," *IEEE Communications Magazine*, **24**(10), 1986, pp. 8-15.

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

[72] R. Wolski, "Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service," *High Performance Distributed Computing (HPDC)*. August 1997, Portland, Oregon, pp. 316-325.

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

[74] J. Zahorjan and C. McCann, "Processor scheduling in shared memory multiprocessors," *ACM SIGMETRICS*. May 1990, pp. 214-255.

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

[76] L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks," *SIGCOMM*. Sept. 1990, pp. 19-29.

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