Exploiting Fine-Grained Idle Periods in Networks of Workstations

Kyung Dong Ryu and Jeffrey K. Hollingsworth, *Member*, *IEEE*

Abstract—Studies have shown that for a significant fraction of the time, workstations are idle. In this paper, we present a new scheduling policy called Linger-Longer that exploits the fine-grained availability of workstations to run sequential and parallel jobs. We present a two-level workload characterization study and use it to simulate a cluster of workstations running our new policy. We compare two variations of our policy to two previous policies: Immediate-Eviction and Pause-and-Migrate. Our study shows that the Linger-Longer policy can improve the throughput of foreign jobs on a cluster by 60 percent with only a 0.5 percent slowdown of local jobs. For parallel computing, we show that the Linger-Longer policy outperforms reconfiguration strategies when the processor utilization by the local process is 20 percent or less in both synthetic bulk synchronous and real data-parallel applications.

Index Terms—Meta-computing, cluster computing, process migration, networks of workstations, parallel computing.

1 Introduction

Studies have shown that up to three-quarters of the time workstations are idle [21]. Systems such as Condor [19], LSF [32], and NOW [3] have been created to use these available resources. Such systems define a "social contract" that permits foreign jobs to run only when a workstation's owner is not using the machine. To enforce this contract, foreign jobs are stopped and migrated as soon as the owner resumes use of their computer. We propose a policy, called *Linger-Longer*, that refines the social contract to permit fine-grained cycle stealing. By permitting foreign jobs to linger on a machine at low priority even when local tasks are active, we can improve the throughput of foreign jobs in shared clusters by 60 percent while holding the slowdown of local jobs to only 0.5 percent.

The motivation for the Linger-Longer approach is simple: Even when users are "actively" using workstations, the processor is idle for a substantial fraction of the time. In addition, a significant amount of memory is usually available. To minimize the effect on the owner's workload, current techniques do not use these fine-grained idle cycles.[1] Linger-Longer exploits these fine-grained idle periods to run foreign jobs with very low priority (so low that local jobs are allowed to starve the foreign task). Our approach enables the system to utilize most idle cycles while limiting the slowdown of the owner's workload to an acceptable level. To improve job response time, Linger-Longer will not let the foreign jobs linger forever on a busy machine. We employ a cost model to predict when the benefit of running on a free node outweighs the overhead of a migration.

The primary beneficiaries of the Linger-Longer scheduling policy are large compute-bound sequential jobs. Since most of these jobs are batch (no user interaction during execution), and consist of a family of related jobs that are submitted as a unit and must all be completed prior to the results being used (e.g., a collection of simulation runs with different parameters), job throughput rather than response time is the primary performance metric. We will concentrate on throughput as the metric we try to optimize.

A key question about Linger-Longer is whether a scheduling policy that can delay users' local jobs will be accepted. For several reasons, we think this problem can be overcome. First, as shown in Section 5.1, the delay that users will experience with our approach is very low. Second, existing systems that exploit free workstations also have an indirect impact on users due to the time required to reload virtual memory pages and caches after a foreign job has been evicted.

In the rest of this paper, we present an overview of the Linger-Longer policy and evaluate its performance via simulation. First, related work is surveyed in Section 2. Section 3 describes the Linger-Longer policy and explains its prediction model for migration. Section 4 characterizes the utilization of workstations, evaluates the potential for lingering, and presents a study of the available CPU time and physical memory on user workstations. Section 5 evaluates the Linger-Longer policy at a fine-grained level on a single node, and, then Section 6 measures cluster level performance by simulating a medium scale cluster of 64 nodes with sequential jobs. Running parallel jobs using Linger-Longer is also investigated in Section 7. Finally, we will conclude with a summary in Section 8.

2 Related Work

Previous work on exploiting available idle time on workstation clusters used a conservative model that would only

1. Part of the motivation for this policy is to promote user acceptance of foreign jobs running on their system. However, after 10 years of experience with environments such as Condor, user acceptance seems to have been reached.

• *The authors are with the Computer Science Department, University of Maryland, College Park, MD 20742.*
E-mail: {kdryu, hollings}@cs.umd.edu.

Manuscript received 8 Sept. 1998; accepted 1 Feb. 2000.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 107366.

run jobs when the local user was away from his/her workstation and no local processes were runnable. Condor [19], LSF [32], and NOW [4] use variations on a "social contract" to strictly limit interference with local users. However, even with these policies, there is some disruption of the local user when he/she returns since the foreign job must be evicted and the local state restored. Our Linger-Longer approach permits slightly more disruption of the user, but tries to limit the delay to an acceptable level. In opposition to these transparency-based approaches, Legion [13] adopted negotiation-based resource harvesting. Machine owners are allowed to specify how much CPU and memory can be used by foreign processes. Generous machine owners receive more credits in return for the future use of Legion resources. The Butler system [9] also gives users access to idle workstations. 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. One system that used nonidle workstations was the Stealth distributed scheduler [16]. It implemented a priority-based approach to running foreign jobs. However, none of the trade-offs in how long to run foreign jobs, or the potential of running parallel jobs were investigated.

Prior studies that investigated running parallel jobs on shared workstation clusters also employed fairly conservative eviction policies. Dusseau et al. [10] used a policy based on immediate eviction. They were able to use a cluster of 60 machines to achieve the performance of a dedicated parallel computer with 32 processors. Acharya et al. [1] adopted a different approach that reconfigured the parallel job to use fewer nodes when one became unavailable. This approach permitted running more jobs on a given cluster, although the performance of any single job would be somewhat reduced. Bhatt et al. [6] studied the trade-off between cycle stealing granularity and checkpoint frequency. Leutenegger and Sun [18] considered running parallel jobs on a cluster, but did not use actual workloads to drive their simulations.

PVM [12] and MPI [11] are widely used packages to run parallel programs on clusters, but do not include a scheduling policy. Pruyne and Livny [23] developed CARMI to use idle machines for parallel programs written in PVM. The MIST [7] project also extended PVM to support checkpointing and migration. The distinction between the two systems is that CARMI requires a 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. Hector [24] and Cocheck [26] support checkpointing and migration for parallel programs written with MPI.

Process migration and load balancing have been studied extensively. MOSIX [5] provides load-balancing and preemptive migration for traditional UNIX processes. Harchol-Balter and Downey [14] studied the lifetimes of processes, and developed a predictive model about what processes are worth migrating. Our prediction of the lifetime of nonidle episodes uses predictions similar to theirs. Chowdhury et al. [8] characterized when to reconfigure sequential workloads. DEMOS/MP [22], Accent [31], Locus [28], and V [27] all provided manual or semiautomated migration of processes.

3 Fine-Grained Cycle Stealing

We use the term "cycle" stealing to mean running jobs that don't belong to the workstation's owner (*foreign 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 nonidle. 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. Acharya et al. [1] showed that for their definition of idleness, machines are in a nonidle state for 50 percent of the time. However, even while the machine is in use by the owner, substantial resources are available to run other jobs.

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 use the term lingering to mean running foreign jobs, while the user processes are active. Since the owner has priority over foreign jobs using their personal machine, use of these idle intervals should not affect the performance of the owner's jobs (*local jobs*).

Delay of local jobs should be avoided. 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 local and foreign job classes. Local processes have the highest priority and can starve foreign processes. In addition, when a foreign process is running, an interrupt that results in a local process becoming runnable causes the local process to be scheduled onto the processor even if the foreign job's scheduling quanta has not expired.

3.1 Linger-Longer Migration

Two strategies have been used in the past to migrate foreign jobs: Immediate-Eviction and Pause-and-Migrate. In *Immediate-Eviction*, the foreign job is migrated as soon as the machine becomes nonidle. Because this can cause unnecessary and expensive migrations for short nonidle intervals, an alternative policy, called *Pause-and-Migrate*, that suspends the foreign processes for a fixed time prior to migration is often used. The fixed suspend time should not be long because the foreign job makes no progress in the suspend state. With Linger-Longer scheduling, foreign 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 the foreign job's available resources, there is a cost to move the process' state. Also, the advantage of running on the idle machine depends on the difference in available processor time between the idle machines and a

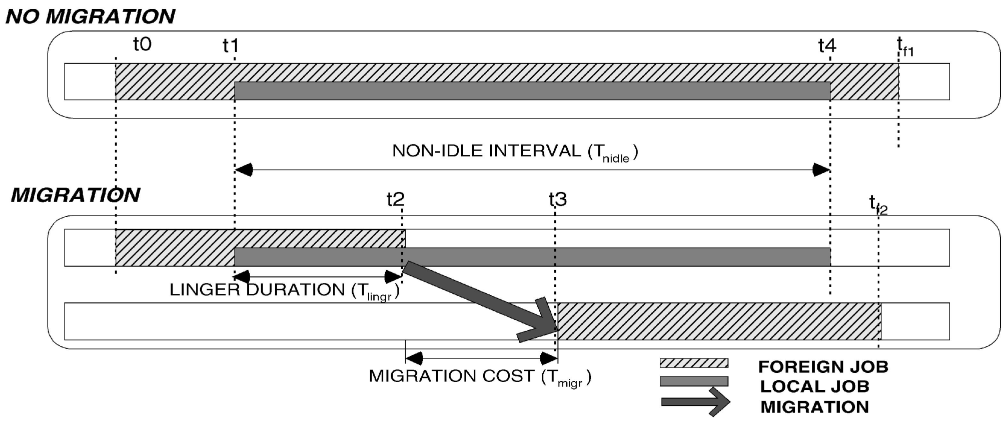

NO MIGRATION
t0
t1
t4
t_{f1}
NON-IDLE INTERVAL (T_{nidle})
MIGRATION
t2
t3
t_{f2}
LINGER DURATION (T_{lingr})
MIGRATION COST (T_{migr})
FOREIGN JOB
LOCAL JOB
MIGRATION

Fig. 1. The timeline for migration using Linger-Longer scheduling. The top case shows a foreign job that remains on a node throughout an episode of processor activity due to local jobs. The lower case shows migration after an initial linger interval (t_1 to t_2) where the foreign job remained on the nonidle node.

current nonidle one. To maximize processor time available to a foreign job, we need a policy that determines the linger duration.

When to migrate in a Linger-Longer scheduler depends on the local CPU utilization on the source and destination nodes, the duration of nonidle state, and the migration cost. The question is when will the foreign job benefit from migration? Given the local CPU utilization and migration cost, the minimum duration of nonidle 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 Fig.1.

In the nonidle state, utilization by the workstation owner starts at t_1 and ends at t_4.The average utilization of the nonidle node is h, and the average utilization on an idle node is l. We assume the execution time of the foreign job exceeds the duration of the nonidle state, so the foreign 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.

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

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.

$$t_{f2}-t_{f1} = (t_4-t_2)\cdot\frac{1-h}{1-l}-(t_4-t_3). \tag{2}$$

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

$$T_{nidle} \geq T_{lingr} + \left(\frac{1-l}{h-l}\right)\cdot T_{migr}, \tag{3}$$

where $T_{nidle} = t_4 - t_1$is the nonidle state duration, $T_{lingr} = t_2 - t_1$ is the lingering duration and the migration cost is denoted as $T_{migr} = t_3 - t_2$. If we knew the nonidle 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 nonidle state will end, we have to predict it. We use the observations of Harchol-Balter and Downey [14], and Leland and Ott [17], which states 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 nonidle 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:

$$T_{lingr} = (\frac{1-l}{h-l})\cdot T_{migr}. \tag{4}$$

So, the foreign job should linger T_{lingr} before migrating. For a nonidle 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 foreign 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:

$$\begin{aligned} T_{migr} &= Suspend\_Time(source)\\ &\quad + Process\_size/Network\_Bandwidth\\ &\quad + Resume\_Time(destination). \end{aligned} \tag{5}$$

This equation can be easily extended for different environments.

We denote the policy of lingering on a node for T_{lingr}, LL. An alternative strategy of never leaving a node (called

Linger-Forever) is denoted LF. This policy attempts to maximize the overall throughput of a cluster at the expense of the response time of those unfortunate foreign jobs that land (and are stuck) on nodes with high local utilization.

4 Workload Analysis and Characterization

To evaluate our Linger-Longer approach, we need to characterize the workload of workstations to be used in such a system. The performance of the various scheduling disciplines for shared clusters depends on the characteristics of the workstation cluster. Workstation utilization depends on many factors including the time of day, day of week, and schedule of the primary users. Developing an analytical model for such a complex pattern would be difficult, if not impossible. Instead, we choose to evaluate traces of the utilization patterns of existing workstations. Dusseau [10] and Acharya [1] have also used this approach. To evaluate scheduling policies that transfer work off a workstation as soon as the user returns (Immediate-Eviction and Pause-and-Migrate), it is sufficient to consider coarse-grained metrics of utilization (on the time scale of seconds). However, because of the fine-grained interaction between local and foreign processes when using the Linger-Longer policy, we need data about individual requests for processors at the granularity of scheduler dispatch records, 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 adopt a two-level strategy to characterize the workload. First, we measure the fine-grained run-idle bursts at various levels of processor utilization from 0 percent (idle) to 100 percent (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. We explain how to combine two workloads of different time-scales in greater detail in Section 5.

We are also interested in evaluating the amount of free memory that is available on the workstations. To do this, we considered a coarse-grained trace of available memory, and looked at the size of the free memory both when the owner's processes were running and when the workstation was idle.

4.1 Fine-Grained Workload Analysis

To analyze the fine-grained utilization of the CPU, 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 we give 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 scheduling of individual processes, OS specific factors, such as scheduling policy and time quanta, would not impact the characteristics of trace data.

We divided utilization into 21 buckets ranging from 0 percent to 100 percent 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 Fig. 2. The solid line in the figure shows three sample distributions for low (10 percent), medium (50 percent), and high (90 percent) CPU utilization for run and idle bursts, respectively. Based on the analysis of 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 [29, p. 479].

The dashed line in Fig. 2 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 foreign 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. Fig. 3 shows a comparison between the number of run bursts generated by our two-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.[2]

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 Fig. 4. 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 bottom two graphs in Fig. 4 show the idle duration mean and variance, respectively.

4.2 Coarse-Grained Workload Analysis

To generate the long-term variations in processor utilization, we use the traces collected by Arpaci et al. [10]. These traces cover data from 132 machines of the CAD group at the UC Berkeley and measured over 40 days, and contain samples every two seconds of: CPU usage, memory usage,

2. The dips at 15 percent and 60 percent in Fig. 3 came from the empirically observed workload distribution in Fig. 4c and 4d.

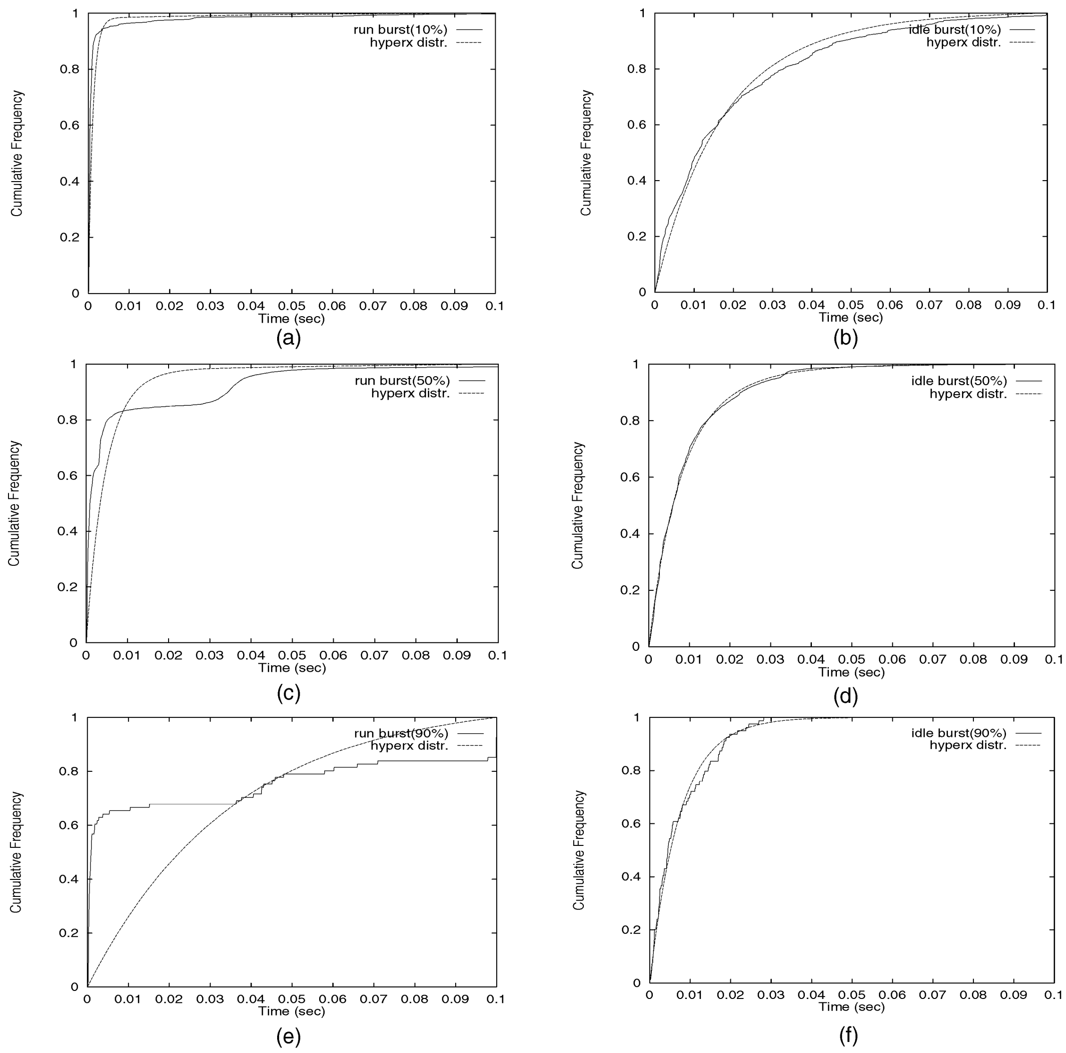


Fig. 2. Run and Idle burst histograms. The CDF (Cumulative Distribution Function) for the run and idle duration of local jobs on workstations: (a) and (b) are for 10 percent utilization, (c) and (d) 50 percent, and (e) and (f) 90 percent.

keyboard activity, and a Boolean indicating idle/nonidle state. An idle interval is a period of time with the CPU less than 10 percent used and no keyboard action for 1 minute (called the recruitment threshold).

To assess the potential for Linger-Longer, we measured the overall CPU utilization and compared it to the CPU utilization during idle and nonidle intervals. The graph in Fig. 5a 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 nonidle time. Even nonidle intervals have very low usage, although it is somewhat higher than idle time. While 46 percent of the time a machine is in a nonidle state, 76 percent of the time in nonidle intervals, the processor utilization is less than 10 percent. The reason that these intervals of time are considered nonidle is due to keyboard activity and the requirement that a workstation have low utilization at least for one 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 percent of the time, processor utilization is less than 10 percent. In addition, moderate processor utilization is rare. Only 13 percent of

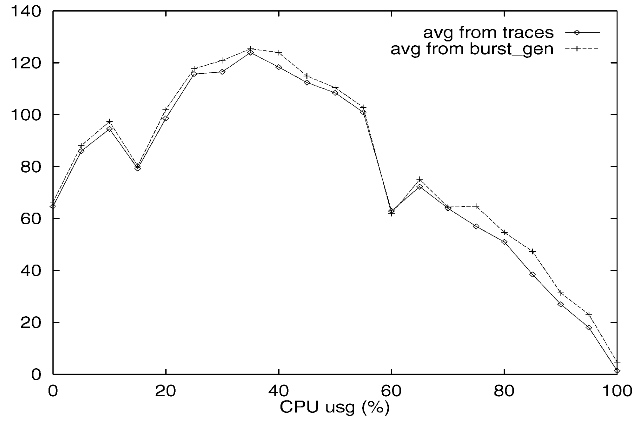


Fig. 3. Number of run bursts. The solid line shows the number of run bursts (context switches) in our measured processor load. The dashed curve shows the same data generated by our model.

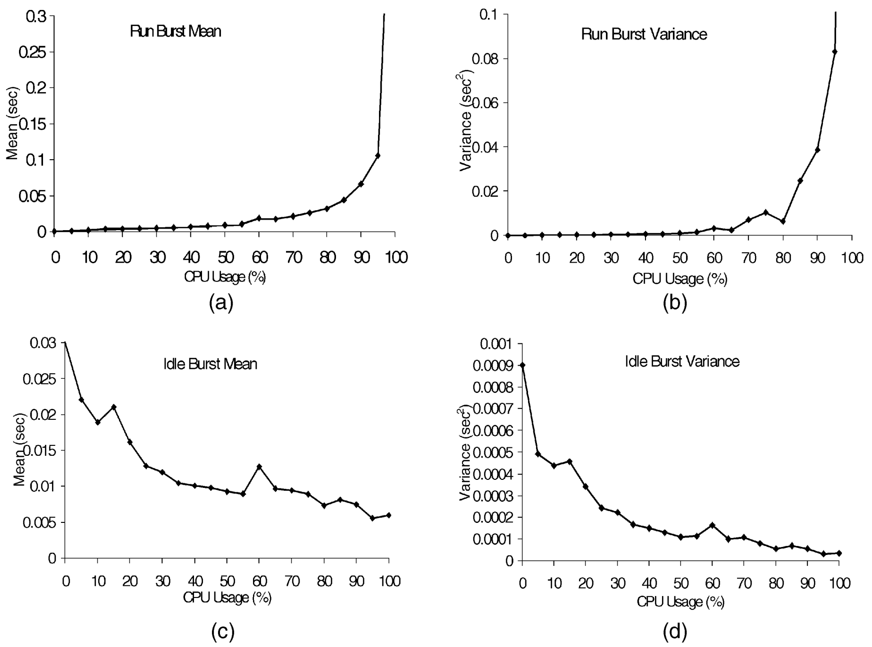


Fig. 4. Workload parameters as a function of processor utilization. The mean and variance of the run and idle bursts seen in the fine-grained workload traces as a function of the processor utilization. (a) Run burst mean, (b) run burst variance, (c) idle burst mean, and (d) idle burst variance.

the time is the utilization between 10 percent and 80 percent. However, high utilization is somewhat more common (utilization of 80 percent or more occurs 7 percent of the time). The basic conclusion of this data is that except for rare instances of heavy use, processors on these workstations had significant capacity available over 90 percent 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. The Fig. 5b 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 clearly shows little correlation with time of day, and that most of the time the processor is between zero and fifteen percent utilized.

To meet our goal of allowing foreign jobs to linger on a workstation and at the same time not to interfere with local jobs, we need to ensure that enough real memory is available to accommodate the foreign job. Like processor time, we propose to use priority as a mechanism to ensure that foreign jobs do not consume memory needed by local jobs.[3] The idea is to divide memory into two pools: one for local jobs and the other for foreign jobs. Whenever a page is placed on the free-list by a local job, the foreign job is able to use the page. Likewise, when the local job runs out of pages, it reclaims them from the foreign job prior to paging out any of its pages. A similar technique was employed in the Stealth scheduler [16].

To fully evaluate the availability of pages for foreign jobs, a complete simulation of the priority-based page replacement scheme is required. However, 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 64MB main memory. The CDF of available memory is shown in Fig. 6. This graph shows that 90 percent of time, more than 14 Mbytes of memory is available for foreign jobs, and that 75 percent of the time at least 30 MB of memory is available. Interestingly, there is no significant difference in the available memory between idle and nonidle states.[4] We feel that the amount of free memory generally available is sufficient to accommodate one compute-bound foreign job of moderate size.

To see if the above workload characteristics were typical, we looked into other traces collected by Acharya et al. [1, 2]. They comprise one big trace and three small scale traces. The big trace was collected from a Condor pool having 310 machines at the University of Wisconsin. The small scale traces cover only between 20 and 30 workstations, respectively, from two academic institutions and one software company. All traces were generated for two weeks in September 1997.

First, we look at the machine busy time based only on keyboard and mouse activity (keyboard-busy). In the University of California at Berkeley trace, 21.3 percent of the time the machines were keyboard-busy on the average. The Condor trace showed similar results: 24.8 percent of the time the workstations were keyboard-busy. 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

3. To implement this, we have added priority to the Linux paging mechanism [25].

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

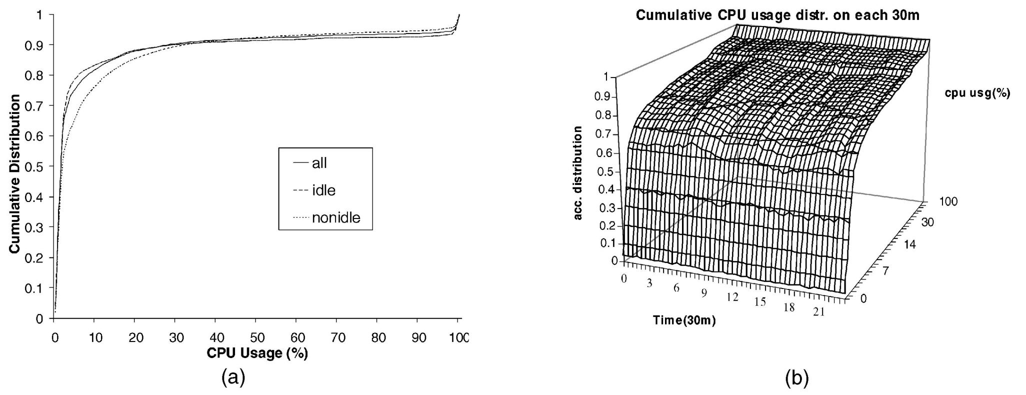


Fig. 5. Distribution of processor utilization. (a) shows the processor utilization for all time. The solid line is for all states, and the two-dashed lines show the utilization of idle and nonidle nodes. (b) shows the processor utilization as a function of the time of day. The sheer face of the surface indicates that most of the time nodes are lightly loaded.

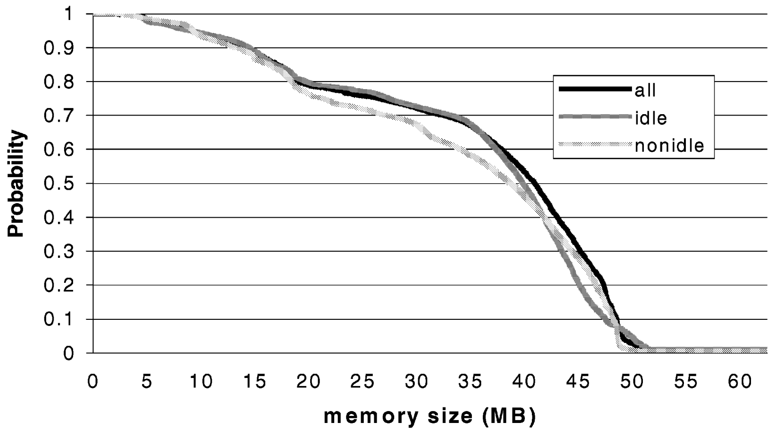


Fig. 6. Distribution of available memory. The solid line shows the overall free memory and the two dashed lines show the free memory during idle and nonidle intervals. The y-axis shows the fraction of time that at least x KB of memory are available. Each workstation has 64 Mbyte main memory.

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 Fig. 5. In addition to keyboard/mouse activity, we use the CPU load information to define idleness of machines. Using a load average threshold 0.3 as busy, for the Condor traces, 36.8 percent of the time a machine was idle on the average. Although the data gathered was somewhat different, the resulting idle and nonidle time was similar. Also, from the small traces from other institutions, we found that machines are idle for more time and the resources are less utilized than the UC Berkeley or Condor data sets.

For memory availability, the result of the UC Berkeley trace is backed up by a recent study by Acharya et al. [2]. With the recent 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. We believe these observations indicate that the Berkeley data is a representative trace.

5 Node-Level Simulation Results

Before evaluating the ability of Linger-Longer to improve aggregate cluster performance, we first present a series of simulation studies to demonstrate that using a linger approach will not cause significant slowdown to the local jobs. In this section, we also evaluate the performance of migration using a Linger-Longer scheduling policy.

5.1 Linger-Longer Scheduling

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 local jobs. Using a priority-based scheduler will cause a local 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 foreign job and load the state of the local job. In this section, we present a simulation study of the delay induced in a local process by a lingering foreign job.

A key question to evaluating the overhead of priority-based preemption is the time required to switch from the foreign job to the local one. There are three 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

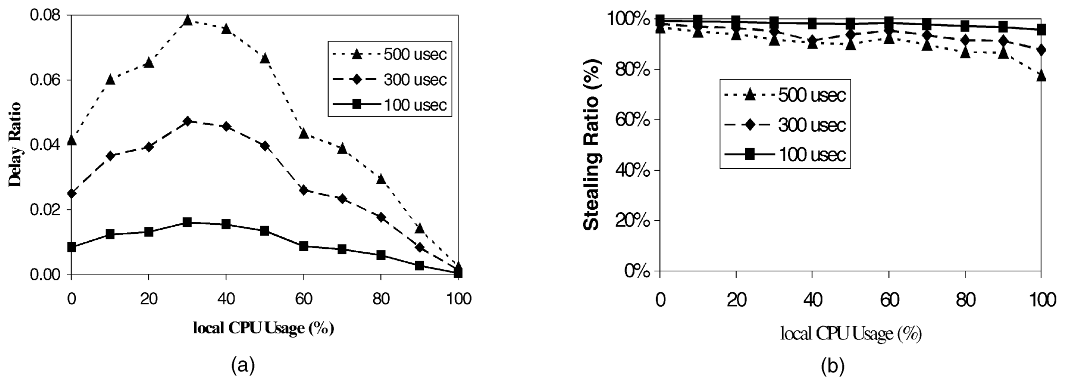


Fig. 7. Local job delay ratio (LDR) and fine-grained cycle stealing ratio (FCSR). Each curve shows the impact of three different effective context switch times (100, 300, and 500 microseconds). (a) shows the delay experienced by local jobs at various level of utilization. (b) shows the percent of the available idle processor time made available to a compute bound foreign job at different levels of local job processor utilization.

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 foreign jobs can use only free memory and, therefore, will not page out the memory pages of local processes. So, virtual memory paging cost is not included in the effective context switch time. The mechanisms for this prioritized use of memory have been investigated and developed in [25].

To estimate the effective context switch time, we use the results obtained by Mogul and Borg [20] 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 foreign job would be executed is after all local 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 foreign 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 local jobs. For each simulation, we computed two metrics: the local job delay ratio (LDR) and the fine-grained cycle stealing ratio (FCSR). The LDR metric records the average slowdown experienced by local jobs due to the extra context switch delay introduced by foreign jobs. The FCSR metric records the fraction of the available idle processor cycles that are used by the foreign job.

Fig. 7 shows the LDR and FCSR metrics for three different effective context switch times at various level of processor utilization by local jobs. For the chosen effective context switch time of 100 microseconds, the delay seen by the application process is about 1 percent. For context switch times up to 300 microseconds, the delay remains under 5 percent. However, when the effective context switch time is 500 microseconds, the overhead is 8 percent. In all of these cases, Lingering was able to make productive use of over 90 percent of the available processor idle cycles.

5.2 Linger-Longer Migration

We next consider the impact of making choices about when to migrate a task to a less loaded node. If there is a free node available, it is best to try to migrate a foreign job onto that node. As described in Section 3.1, 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 foreign job. Migration cost is primarily a function of the memory size of the foreign job. For our simulations, we assume that all foreign jobs are 8 Megabytes, migration takes place over a 10 Mbps Ethernet at an effective rate of 3 Mbps (to limit the load placed on the network by process migration), and that the foreign 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.

Fig. 8 shows a comparison of the completion time of a hypothetical 50-second foreign job that experiences local processor utilization of 20 percent 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 nonidle for a period of time less than 30 seconds[5], the Linger-Longer policy will finish the job before the Immediate-Eviction policy because Linger-Longer avoids migration and uses fine-grained idle cycles. However, if the workstation is

5. This linger time was derived using equations (4) and (5).

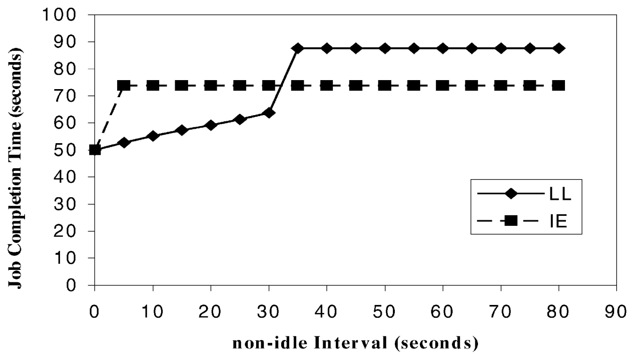


Fig. 8. Migration timing. Completion time of a 50 second foreign job using Linger-Longer (LL) and Immediate-Eviction (IE) as a function of the episode of a workstation owner returning.

nonidle 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. Linger-Longer not only provides fine-grained idle cycles, but also reduces migration. An overall simulation of a full cluster is presented in the following section.

6 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. We used a two-level workload generator to produce a local user workload for a 64-node cluster. Fig. 9 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-grained resource usage is generated by looking up appropriate parameters, mean and variance, based on the current coarse-grained resource data from the trace files.

We then ran two different types of sequential foreign jobs on the cluster. Workload-1 contains 128 foreign jobs each requiring 600 processor seconds. This workload was designed to represent a cluster with a significant demand being placed on the foreign job scheduler, since on average each node had two foreign 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 $\frac{1}{4}$ of the nodes are required to run the foreign jobs. All foreign jobs are 8 Megabytes and migration takes places over a 10 Mbps Ethernet at an effective rate of 3 Mbps. We also assume that the foreign 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 foreign 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 foreign jobs per second when the number of jobs in the system was held constant.

The results of the simulation are summarized in the 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 percent faster with Linger-Longer than Immediate-Eviction or Pause-and-Migrate, and Linger-Forever's job completion time is 49 percent faster than either of the nonlingering policies. There is virtually no difference between the 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

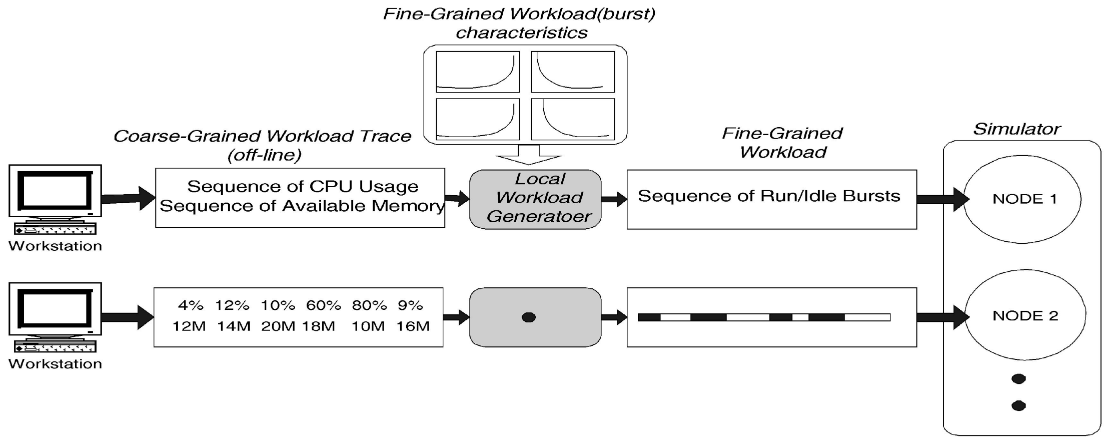

Fine-Grained Workload(burst) characteristics
Coarse-Grained Workload Trace (off-line)
Fine-Grained Workload
Simulator
Sequence of CPU Usage
Sequence of Available Memory
Local Workload Generatoer
Sequence of Run/Idle Bursts
NODE 1
Workstation
4% 12% 10% 60% 80% 9%
12M 14M 20M 18M 10M 16M
NODE 2
Workstation

Fig. 9. 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.

TABLE 1
Performance of Cluster Scheduling Policies

	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

average job completion time ranges from 1,859 to 1,860 seconds; this implies that on average they were running 97 percent 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 semiavailable node and, thus, expedite their departure from the system; so the benefit of lingering on a nonidle 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 36 percent improvement over the PM policy and 42 percent improvement over the IE policy. For workload-2, the LL and LF policies provide slight (1 to 3 percent) 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 foreign 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 foreign jobs. Using the throughput metric, the LL policy provides a 50 percent improvement over the PM policy. Likewise, the LF policy permits a 60 percent 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 local processes was less than 0.5 percent. This average is somewhat less than the one percent delay reported in the previous section since not all nonidle nodes have foreign 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 nonidle node), paused, suspended, and migrating. The results in Fig.10a show the behavior of workload-1. The major difference between the linger and nonlinger 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. Fig.10b shows the breakdown 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.

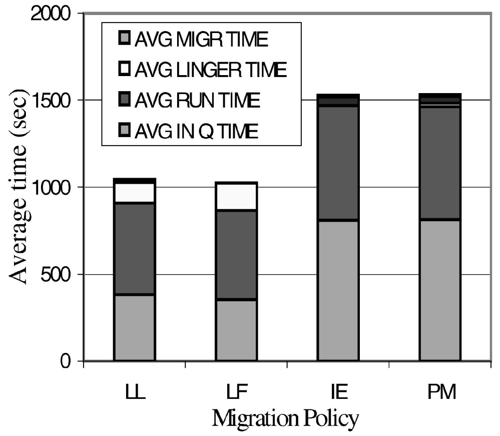
The overall trends in the cluster level simulation show that Linger-Longer and Linger-Forever provide significant increased performance to a cluster when there are more jobs than available nodes, and that there is no difference in performance at low levels of cluster utilization.

7 Parallel Jobs

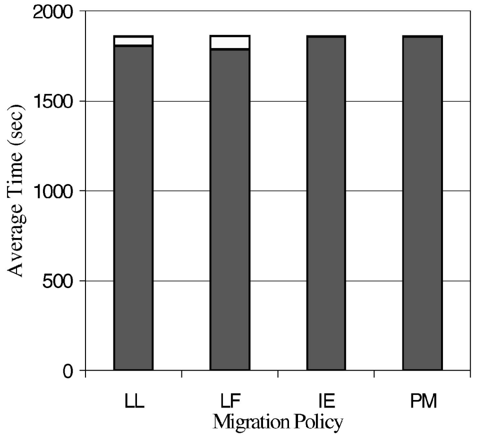
The trade-offs in using Linger-Longer scheduling for parallel programs are more complex. When a single process of a job is slowed down due to a local job running on the node, this can result in all of the nodes being slowed down due to synchronization between processes. On the other hand, when a migration 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.

7.1 Synthetic 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.

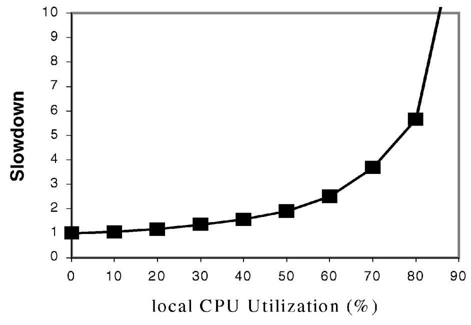


(a)

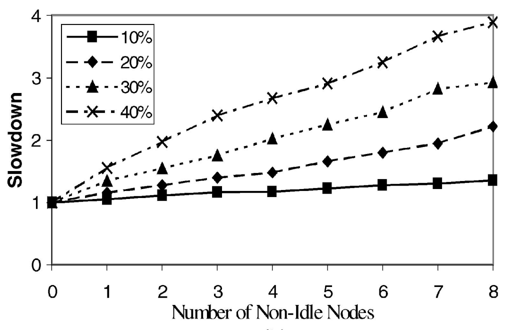


(b)

Fig. 10. Average completion time. The chart (a) shows the breakdown of the average time spent in each state (queued, running, lingering, or migrating) for workload-1 (many foreign jobs). The chart (b) shows the same information for workload-2 (few foreign jobs).



(a)



(b)

Fig. 11. Parallel job slowdown. (a) shows the slowdown of an eight-node parallel job vs. processor utilization by the local processes. (b) shows the same eight-process parallel application at four levels of processor utilization by local jobs when the number of nonidle nodes is varied from one to eight.

The communication phase ends with an optional barrier. This synthetic parallel job model has been successfully used in [10] to explore various performance factors. We simulated an eight-process application with a 100 milliseconds between each synchronization phase, and a NEWS[6] style of message passing within a communication phase.

The graph in Fig. 11a shows the slowdown (compared to running on eight idle nodes) experienced in the application's execution time when one node is nonidle and the CPU utilization by the local processes are varied from 0 percent to 90 percent. At utilization above 50 percent, 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 nonidle node. The NOW [10] 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, Acharya et al. [1] proposed reconfiguring 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 percent is an attractive alternative.

6. A process exchanges messages only with its four neighbors.

We also investigated how the slowdown of a parallel program was affected by having more than one nonidle 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 nonidle nodes. The results of this simulation are presented in Fig. 11b and show that at low levels of local processor utilization, the parallel application slowdown is relatively insensitive to the number of nonidle nodes. For example, at local utilization of up 20 percent, the largest slowdown was a factor of two even when all eight processes are lingering on nonidle nodes. However, when local utilization was above 30 percent, and the number of nonidle nodes is more than one or two, the slowdown to the parallel application is significant.

One of the key parameters in understanding the performance of parallel jobs using Linger-Longer is the frequency of synchronization. Fig. 12 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

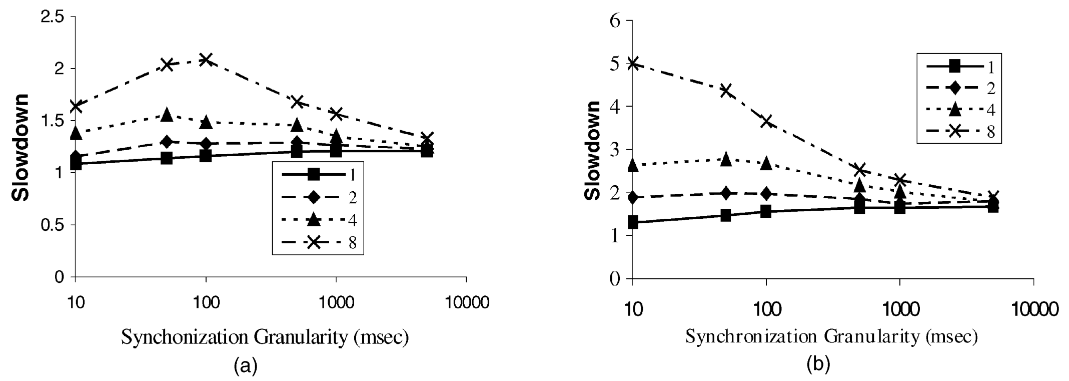


Fig. 12. Synchronization granularity vs. slowdown. (a) shows the slowdown of running a parallel program with one, two, four, or eight nonidle nodes compared with running on all eight idle nodes as a function of the synchronization granularity when the nonidle nodes have 20 percent utilization by local jobs. (b) shows the same data when the nonidle utilization is 40 percent.

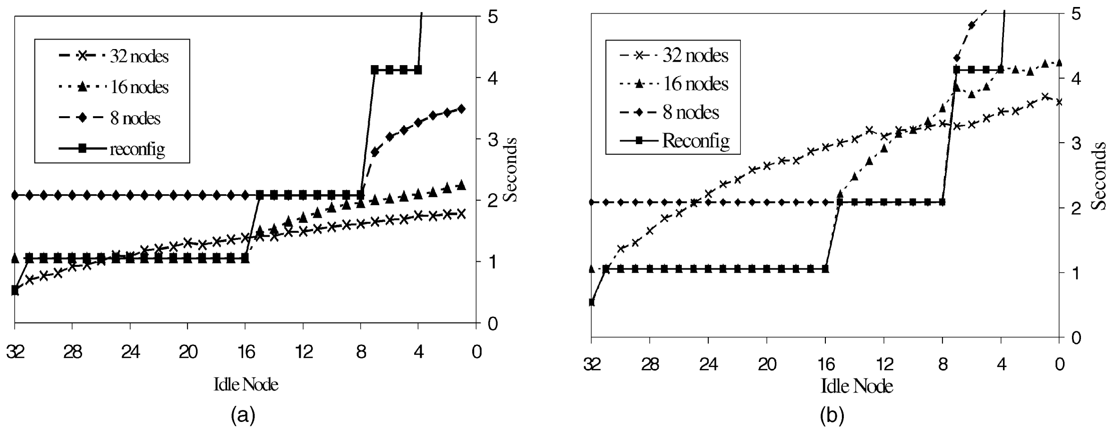


Fig. 13. Linger-Longer vs. reconfiguration. (a) shows the completion time of a parallel job running on a cluster using several different scheduling policies. The x-axis shows the number of idle nodes. The first three curves show the Linger-Longer policy running using 8, 16, or 32 nodes. The fourth curve shows the reconfigure policy. For all curves, the local utilization of nonidle nodes is 20 percent. (b) shows the same data when the local utilization is 40 percent.

communications in milliseconds. Each of the four curves shows the slowdown when one, two, four, and eight nodes have 20 percent (a) and 40 percent (b) processor utilization by local jobs. The results show that larger synchronization granularity produces less slowdown with the exception of the 10 ms case. With 10 ms of synchronization granularity, the portion of computation is much smaller than communication time. Although synchronization occurs more frequently, the parallel job slowdown is less due to the low CPU demand. Also, for 20 percent local processor utilization, lingering provides an attractive alternative to reconfiguration since, even when four nodes are nonidle, 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 local jobs on a nonidle workstations was 20 percent or 40 percent. 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 Fig. 13. 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 nonidle nodes by lingering.

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

7.2 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 [15] simulator which can run shared memory parallel applications. We chose three

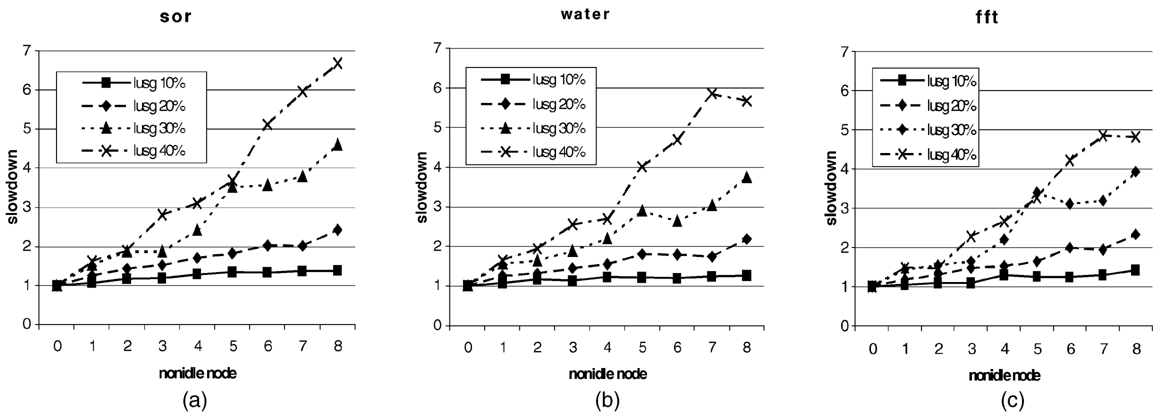


Fig. 14. Slowdown by nonidle nodes and their local CPU usage The graphs show the slowdown of parallel jobs: (a) sor, (b) water, and (c) fft as the number of nonidle nodes varies from 0 to all 8 with Linger-Longer. The cluster size is eight. The curves in each graph represent the different local utilization of nonidle nodes.

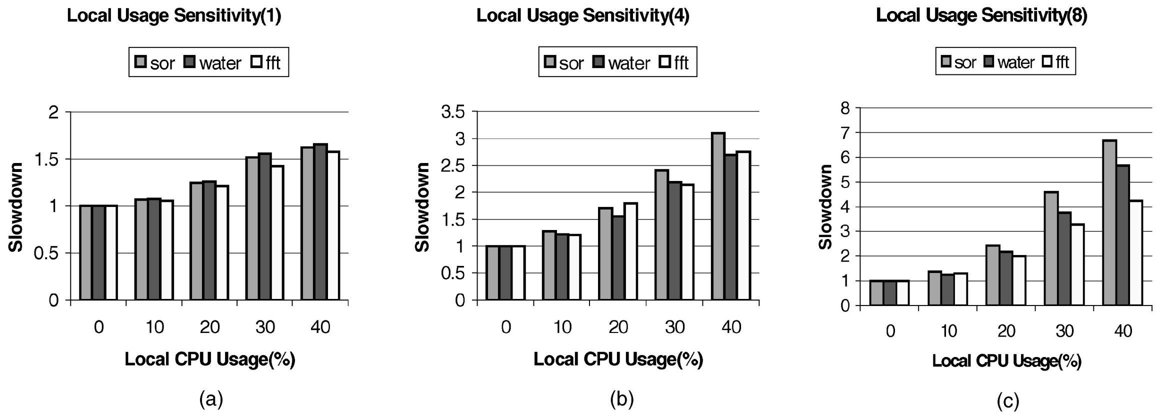


Fig. 15. Performance sensitivity to local CPU usage. The graphs compare three parallel jobs (sor, water, and fft) in how sensitive to local utilization of nonidle nodes. The left-most graph (a) is for the case only one of total eight nodes is nonidle, (b) and (c) is for four and eight nonidle nodes, respectively.

common shared-memory parallel applications: sor (Jacobi relaxation), water (a molecular dynamics; from SPLASH-2 benchmark suite [30]), 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 $2,048 \times 1,024$ 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 slows down the parallel jobs on nonidle nodes. An eight-node cluster is used to run each application. The number of nonidle nodes and its local utilization were controlled. The results are summarized in Fig. 14. For all three cases, when only one nonidle node is involved even with 40 percent local utilization the slowdown (compared to running on eight idle nodes) reaches only 1.7. When more than half the nodes are nonidle, 0 to 20 percent local utilization looks endurable. Linger-Longer with four nonidle nodes and 20 percent local utilization causes only 1.5 to 1.6 slowdown. Even when all eight nodes are nonidle, 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 is substantially better than not using any of the nonidle nodes, which previous policies required.

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 nonidle nodes. The results are shown in Fig. 15. The difference in the slowdown between 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.

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 [15]). Lock waiting time is the time to obtain the lock on the shared object. In our applications, only water uses locks.

The graphs in Fig. 16 and Fig. 17 summarize the results. Fig. 16 shows the case when only one node is nonidle out of eight nodes. Each graph contains five bars for different local

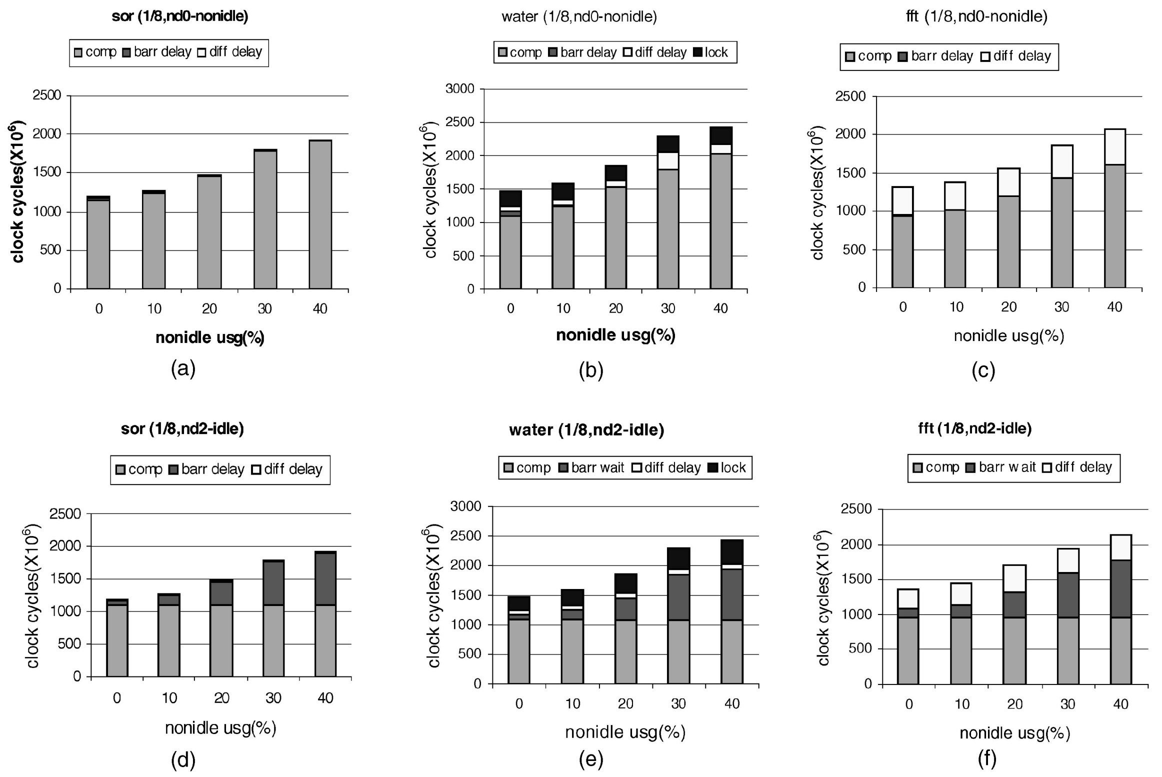


Fig. 16. Application sensitivity to local activity—one nonIdle case. The graphs show the time spent in different activities for three parallel jobs running on a cluster of eight nodes. Only one node (node 0) is nonidle. (a), (b), and (c) show the results for nonidle node, node 0. (d), (e), and (f) show the graphs for an idle node, node 2. For each graph, the x-axis shows local CPU utilization on a nonidle node. Each bar comprises computation time, barrier waiting time, diff delay, and lock waiting time (from bottom to top).

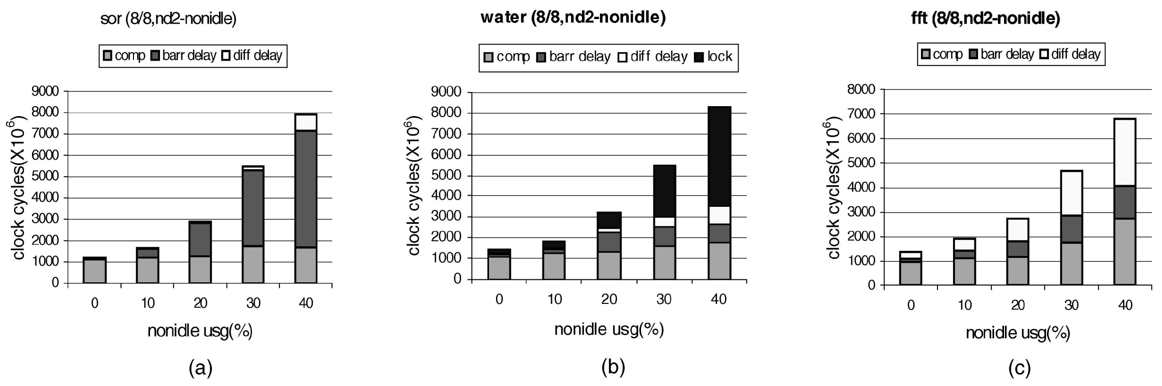
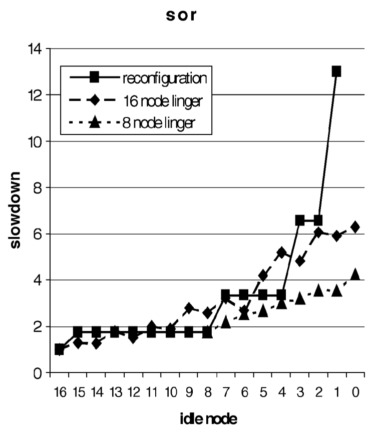


Fig. 17. Application sensitivity to local activity—eight nonIdle cases. (a), (b), and (c) show the time spent in different stages for three parallel jobs running on a cluster of eight nodes. All eight nodes are nonidle. For each application, the x-axis shows local CPU utilization on a nonidle node. Each bar comprises computation time, barrier waiting time, diff delay, and lock waiting time (from bottom to top).

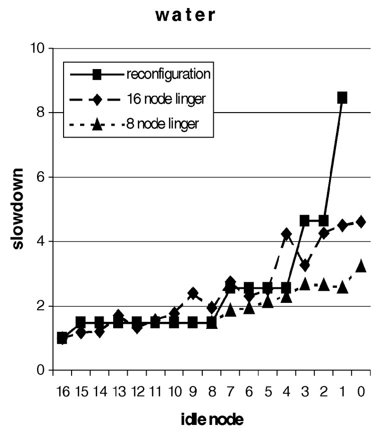
CPU utilizations of the nonidle node. The first row of graphs shows the profiles for the nonidle node for three applications and the second row shows the behavior of an idle node. Since node 0 is the only nonidle node, barrier delay is limited and does not increase due to the local CPU utilization. Not surprisingly, the second row demonstrates that the barrier waiting time on idle nodes increases as local CPU utilization on nonidle grows. There is no noticeable change in diff delay and lock waiting time for this case.

Fig. 17 illustrates the case when all eight nodes are nonidle. Since the 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`. Fig.17a 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 7.1.

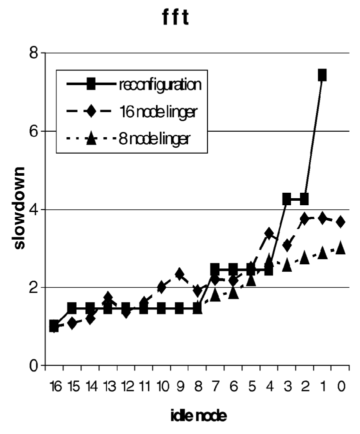
To summarize, frequent synchronization makes parallel jobs more sensitive to local CPU activity and the effect of global synchronization, such as barriers, is more than local synchronization, such as locks. However, for a practical



(a)



(b)



(c)

Fig. 18. Linger-Longer vs. reconfiguration for shared-memory parallel applications. The graphs show the slowdown of three parallel jobs running on a cluster of 16 nodes using several different scheduling policies. The x-axis shows the number of idle nodes. (a) `sor` the first (box marked) curve shows the reconfigure policy (b) `water`, and (c) `fft` show Linger-Longer policy running using 16 or 8 nodes. For all curves, the local utilization of nonidle nodes is 20 percent.

lingering local utilization, 0 to 20 percent, the slowdown is almost the same for all three applications. In general, when local utilization exceeds 20 percent, we would prefer to migrate the job to free nodes (if one exists) or to reconfigure the application to run on fewer nodes.

Also, we compared the Linger-Longer and reconfiguration policies using three real parallel applications. 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 Fig. 18. In each graph, the curves show the Linger-Longer policy using 16 and eight nodes, and the reconfiguration policy using the maximum power of two number of idle nodes available. The graphs show the results for `sor`, `water` and `fft` when the local utilization for nonidle nodes is 20 percent. 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 to reconfigure the parallel job compared to its run time, the gain would be even bigger. However, when less than eight idle nodes are left, lingering with eight 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.

In this section, we investigated running parallel applications with Linger-Longer and varied not only local CPU utilization but also the number of nonidle 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 nonidle nodes rather than suspending the whole program until the node returns to idle or reconfiguring to run on fewer nodes.

8 Conclusions

In this paper, we provided a workload characterization study that described the fine-grained requests for processor time at various levels of utilization and evaluated traces of workstation load at the two-second level for long time durations. We then presented a technique to compose the workload and to generate fine-grained workloads for long intervals of time.

We have devised a new approach, called Linger-Longer to using available workstations to perform sequential and parallel computation. We presented a cost model that determines how long a process should linger on a nonidle node. The results for our proposed approach are encouraging, we showed that for typical clusters lingering can increase throughput by 60 percent, and on average cause only a 0.5 percent slowdown of local user processes.

For parallel computing, we showed that the Linger-Longer policy outperforms reconfiguration strategies when the processor utilization by the local process is 20 percent or less in both bulk synchronous and real data-parallel applications. However, reconfiguration outperforms Lingering for higher levels of local process utilization. The throughput improvement that would be possible by making more nodes available to run parallel jobs would likely offset some of this slowdown. An end-to-end evaluation of cluster throughput for parallel jobs is being investigated.

Currently, we are implementing the prototype based on the Linux operating system running on Pentium PCs. The strict priority-based scheduler and page allocation module have been developed and are being evaluated.

Acknowledgments

The authors would like to thank Pete Keleher and Dejan Perkovic for helping us combine their CVM simulator with the Linger-Longer simulator. Anurag Acharya and

Remzi H. Arpaci have been very kind to provide the workload trace data. We also appreciate the useful comments of Bryan Buck. This work was supported in part by the U.S. National Science Foundation awards ASC-9703212 and ASC-9711364, and DOE Grant-DE-FG02-93ER25176.

REFERENCES

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

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

[3] T.E. Anderson, D.E. Culler, and D.A. Patterson, "A Case for NOW (Networks of Workstations)," *IEEE Micro,* vol. 15, no. 1, pp. 54–64, 1995.

[4] 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," *Proc. SIGMETRICS,* pp. 267–278, May 1995.

[5] A. Barak, O. Laden, and Y. Yarom, "The NOW Mosix and Its Preemptive Process Migration Scheme," *Bull. IEEE Technical Committee on Operating Systems and Application Environments,* vol. 7, no. 2, pp. 5–11, 1995.

[6] 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 Trans. Computers,* vol. 46, no. 5, pp. 545–557, 1997.

[7] 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," *Proc. Ann. PVM Users' Group Meeting,* May 1995.

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

[9] R.B. Dannenberg and P.G. Hibbard, "A Butler Process for Resource Sharing on Spice Machines," *ACM Trans. Office Information Systems,* vol. 3, no. 3, pp. 234–252.

[10] A.C. Dusseau, R.H. Arpaci, and D.E. Culler, "Effective Distributed Scheduling of Parallel Workloads," *Proc. SIGMETIRCS,* pp. 25–36, May 1996.

[11] M. Forum, "MPI: A Message Passing Interface Standard," *Int'l J. Supercomputing Applications,* vol. 8, no. 3/4, 1994.

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

[13] A.S. Grimshaw, A. Nguyen-Tuong, and W.A. Wulf, "Campus-Wide Computing: Early Results Using Legion at the University of Virginia," *J. Supercomputing Applications and High Performance Computing,* vol. 11, no. 2, pp. 129–43, 1997.

[14] M. Harchol-Balter and A.B. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *Proc. SIGMETRICS,* pp. 13–24, May 1996.

[15] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," *Proc. ICDCS,* pp. 91–98, May 1996.

[16] P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS),* pp. 336–343, May 1991.

[17] W.E. Leland and T.J. Ott, "Loadbalancing Heuristics and Process Behavior," *Proc. SIGMETRICS,* pp. 54–69, May 1986.

[18] S. Leutenegger and X.H. Sun, "Distributed Computing Feasibility in a Non-dedicated Homogenous Distributed System," *Supercomputing,* pp. 143–152, Nov. 1993.

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

[20] J.C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance," *Proc. ASPLOS,* pp. 75–84, Apr. 1991.

[21] M.W. Mutka and M. Livny, "The Available Capacity of a Privately Owned Workstation Environment," *Performance Evaluation,* vol. 12, pp. 269–284, 1991.

[22] M.L. Powell and B.P. Miller, "Process Migration in DEMOS/MP," *Proc. SOSP,* pp. 110–119, 1983.

[23] J. Pruyne and M. Livny, "Providing Resource Management Services to Parallel Applications," *Proc. Second Workshop Environments and Tools for Parallel Scientic Computing,* SIAM Proc. Series, J. Dongarra and B. Tourancheau, eds., pp. 152–161, 1994.

[24] S.H. Russ, J. Robinson, B.K. Flachs, and B. Heckel, "The Hector Distributed Run-Time Environment," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 11, pp. 1,102–1,114, 1999.

[25] K.D. Ryu, J.K. Hollingsworth, and P. Keleher, "Mechanisms and Policies for Supporting Fine-Grained Cycle Stealing," *Int'l Conf. Supercomputing,* pp. 93–100, June 1999.

[26] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," *Proc. Int'l Parallel Processing Symp.,* pp. 526–531, Apr. 1996.

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

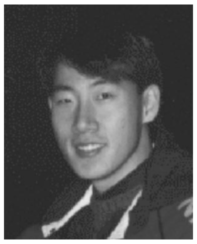
[28] G. Thiel, "Locus Operating System, A Transparent System," *Computer Comm.,* vol. 14, no. 6, pp. 336–346, 1991.

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

[30] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 24–37, 1995.

[31] E.R. Zayas, "Attacking the Process Migration Bottleneck," *Proc. SOSP,* pp. 13–24, 1987.

[32] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," *Proc. SPE,* vol. 23, no. 12, pp. 1,305–1,336, 1993.



Kyung Dong Ryu received the BS and MS degrees in computer engineering from Seoul National University, Korea, in 1993 and 1995, respectively. He is currently a PhD candidate in the Computer Science Department at the University of Maryland, College Park. His current research interests include high performance distributed and parallel systems, resource-aware computing, and wide-area information systems.

Jeffrey K. Hollingsworth received the BS degree in electrical engineering from the University of California at Berkeley in 1988. He received the MS and PhD degrees in computer science from the University of Wisconsin in 1990 and 1994, respectively. He is an assistant professor in the Computer Science Department at the University of Maryland, College Park, and affiliated with the Department of Electrical Engineering and the University of Maryland Institute for Advanced Computer Studies. His research interests include instrumentation and measurement tools, resource aware computing, high performance distributed computing, and computer networks. Dr. Hollingsworth's current projects include the dyninst runtime binary editing tool, and harmony—a system for building adaptable, resource-aware programs. Dr. Hollingsworth is a member of the IEEE Computer Society and the ACM.