Selective Preemption Strategies for Parallel Job Scheduling

Rajkumar Kettimuthu, Vijay Subramani, Srividya Srinivasan, Thiagaraja Gopalsamy, D. K. Panda, and P. Sadayappan

***Abstract*— Although theoretical results have been established regarding the utility of preemptive scheduling in reducing average job turnaround time, job suspension/restart is not much used in practice at supercomputer centers for parallel job scheduling. A number of questions remain unanswered regarding the practical utility of preemptive scheduling. We explore this issue through a simulation-based study, using real job logs from supercomputer centers. We develop a tunable selective-suspension strategy and demonstrate its effectiveness. We also present new insights into the effect of preemptive scheduling on different job classes and deal with the impact of suspensions on worst-case response time. Further, we analyze the performance of the proposed schemes under different load conditions.**

***Index Terms*— Preemptive scheduling, Parallel job scheduling, Backfilling.**

I. Introduction

Although theoretical results have been established regarding the effectiveness of preemptive scheduling strategies in reducing average job turnaround time [1]–[5], preemptive scheduling is not currently used for scheduling parallel jobs at supercomputer centers. Compared to the large number of studies that have investigated nonpreemptive scheduling of parallel jobs [6]–[21], little research has been reported on evaluation of preemptive scheduling strategies using real job logs [22]–[25]. The basic idea behind preemptive scheduling is simple: If a long-running job is temporarily suspended and a waiting short job is allowed to run to completion first, the wait time of the short job is significantly decreased, without much fractional increase in the turnaround time of the long job. Consider a long job with run time T_l. After time t, let a short job arrive with run time T_s. If the short job were to run after completion of the long job, the average job turnaround time would be $\frac{(T_l+(T_l+T_s-t))}{2}$, or $T_l + \frac{(T_s-t)}{2}$. Instead, if the long job were suspended when the short job arrived, the turnaround times of the short and long jobs would be T_s and $(T_s + T_l)$, respectively, giving an average of $T_s + \frac{T_l}{2}$. The average turnaround time with suspension is less if $T_s < T_l - t$, that is, the remaining run time of the running job is greater than the run time of the waiting job.

The suspension criterion has to be chosen carefully to ensure freedom from starvation. Also, the suspension scheme should bring down the average turnaround times without increasing the worst-case turnaround times. Even though theoretical results [1]–[5] have established that preemption improves the average turnaround time, it is important to perform evaluations of preemptive scheduling schemes using realistic job mixes derived from actual job logs from supercomputer centers, to understand the effect of suspension on various categories of jobs.

The primary contributions of this work are as follows:

- Development of a selective-suspension strategy for preemptive scheduling of parallel jobs,
- Characterization of the significant variability in the average job turnaround time for different job categories,
- Demonstration of the impact of suspension on the worst-case turnaround times of various categories, and development of a tunable scheme to improve worst-case turnaround times.

This paper is organized as follows. Section II provides background on parallel job scheduling and discusses prior work on preemptive job scheduling. Section III characterizes the workload used for the simulations. Section IV presents the proposed selective preemption strategies and evaluates their performance under the assumption of accurate estimation of job run times. Section V studies the impact of inaccuracies in user estimates of run time on the selective preemption strategies. It also models the overhead for job suspension and restart and evaluates the proposed schemes in the presence of overhead. Section VI describes the performance of the selective preemption strategies under different load conditions. Section VII summarizes the results of this work.

II. Background and Related Work

Scheduling of parallel jobs is usually viewed in terms of a 2D chart with time along one axis and the number of processors along the other axis. Each job can be thought of as a rectangle whose width is the user-estimated run time

R. Kettimuthu is with Argonne National Laboratory.
V. Subramani and S. Srinivasan are with Microsoft Corporation.
T. Gopalsamy is with Altera Corporation.
D. K. Panda and P. Sadayappan are with the Ohio State University.

and height is the number of processors requested. Parallel job scheduling strategies have been widely studied in the past [26]–[33]. The simplest way to schedule jobs is to use the first-come-first-served (FCFS) policy. This approach suffers from low system utilization, however, because of fragmentation of the available processors. Consider a scenario where a few jobs are running in the system and many processors are idle, but the next queued job requires all the processors in the system. An FCFS scheduler would leave the free processors idle even if there were waiting queued jobs requiring only a few processors. Some solutions to this problem are to use dynamic partitioning [34] or gang scheduling [35]. An alternative approach to improve the system utilization is backfilling.

A. Backfilling

Backfilling was developed for the IBM SP1 parallel supercomputer as part of the Extensible Argonne Scheduling sYstem (EASY) [13] and has been implemented in several production schedulers [36], [37]. Backfilling works by identifying "holes" in the 2D schedule and moving forward smaller jobs that fit those holes. With backfilling, users are required to provide an estimate of the length of the jobs submitted for execution. This information is used by the scheduler to predict when the next queued job will be able to run. Thus, a scheduler can determine whether a job is sufficiently small to run without delaying any previously reserved jobs.

It is desirable that a scheduler with backfilling support two conflicting goals. On the one hand, it is important to move forward as many short jobs as possible, in order to improve utilization and responsiveness. On the other hand, it is also important to avoid starvation of large jobs and, in particular, to be able to predict when each job will run. There are two common variants to backfilling — conservative and aggressive (EASY) — that attempt to balance these goals in different ways.

1) Conservative Backfilling: With conservative backfilling, every job is given a reservation (start time guarantee) when it enters the system. A smaller job is allowed to backfill only if it does not delay any previously queued job. Thus, when a new job arrives, the following allocation procedure is executed by a conservative backfilling scheduler. Based on the current knowledge of the system state, the scheduler finds the earliest time at which a sufficient number of processors are available to run the job for a duration equal to the user-estimated run time. This is called the "anchor point." The scheduler then updates the system state to reflect the allocation of processors to this job starting from its anchor point. If the job's anchor point is the current time, the job is started immediately.

An example is given in Fig. 1. The first job in the queue does not have enough processors to run. Hence, a reservation is made for it at the anticipated termination time of the

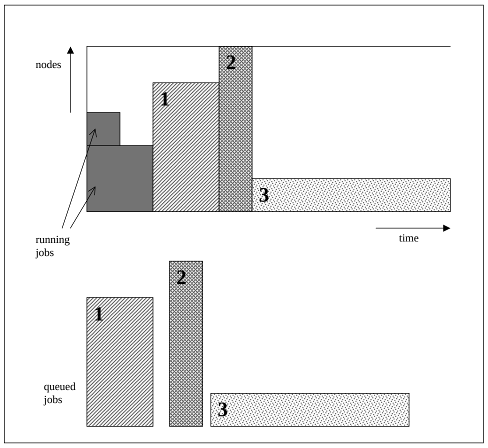

Fig. 1. Conservative backfilling.

longer-running job. Similarly, the second queued job is given a reservation at the anticipated termination time of the first queued job. Although enough processors are available for the third queued job to start immediately, it would delay the second job; therefore, the third job is given a reservation after the second queued job's anticipated termination time.

Thus, in conservative backfilling, jobs are assigned a start time when they are submitted, based on the current usage profile. But they may actually be able to run sooner if previous jobs terminate earlier than expected. In this scenario, the original schedule is compressed by releasing the existing reservations one by one, when a running job terminates, in the order of increasing reservation start time guarantees and attempting backfill for the released job. If as a result of early termination of some job, "holes" of the right size are created for a job, then it gets an earlier reservation. In the worst case, each released job is reinserted in the same position it held previously. With this scheme, there is no danger of starvation, since a reservation is made for each job when it is submitted.

2) Aggressive Backfilling: Conservative backfilling moves jobs forward only if they do not delay any previously queued job. Aggressive backfilling takes a more aggressive approach and allows jobs to skip ahead provided they do not delay the job at the head of the queue. The objective is to improve the current utilization as much as possible, subject to some consideration for the queue order. The price is that execution guarantees cannot be made, because it is impossible to predict how much each job will be delayed in the queue.

An aggressive backfilling scheduler scans the queue of waiting jobs and allocates processors as requested. The scheduler gives a reservation guarantee to the first job in the queue that does not have enough processors to start. This reservation is given at the earliest time at which the required processors are expected to become free, based on the current system state.

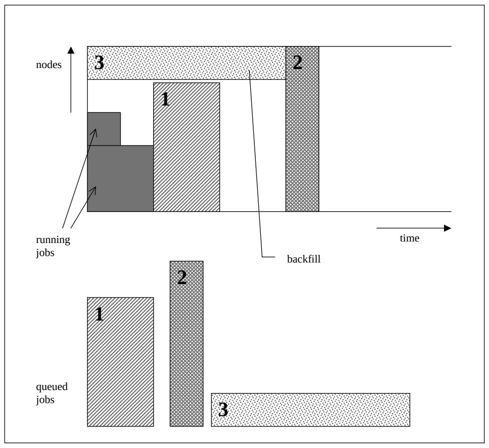

Fig. 2. Aggressive backfilling.

The scheduler then attempts to backfill the other queued jobs. To be eligible for backfilling, a job must require no more than the currently available processors and must satisfy either of two conditions that guarantee it will not delay the first job in the queue:

- It must terminate by the time the first queued job is scheduled to commence, or
- It must use no more nodes than those are free at the time the first queued job is scheduled to start.

Figure 2 shows an example.

B. Metrics

Two common metrics used to evaluate the performance of scheduling schemes are the average turnaround time and the average bounded slowdown. We use these metrics for our studies. The bounded slowdown [38] of a job is defined as follows:

$$Bounded\,slowdown = (Wait\,time + Max(Run\,time, 10))/ \\ Max(Run\,time, 10) \tag{1}$$

The threshold of 10 seconds is used to limit the influence of very short jobs on the metric.

Preemptive scheduling aims at providing lower delay to short jobs relative to long jobs. Since long jobs have greater tolerance to delays as compared to short jobs, our suspension criterion is based on the expansion factor (xfactor), which increases rapidly for short jobs and gradually for long jobs.

$$xfactor = (Wait\,time + Estimated\,run\,time)/ \\ Estimated\,run\,time \tag{2}$$

C. Related Work

Although preemptive scheduling is universally used at the operating system level to multiplex processes on single-processor systems and shared-memory multi-processors, it is rarely used in parallel job scheduling. A large number of studies have addressed the problem of parallel job scheduling (see [38] for a survey of work on this topic), but most of them address nonpreemptive scheduling strategies. Further, most of the work on preemptive scheduling of parallel jobs considers the jobs to be malleable [3], [25], [39], [40]; in other words, the number of processors used to execute the job is permitted to vary dynamically over time.

In practice, parallel jobs submitted to supercomputer centers are generally rigid; that is, the number of processors used to execute a job is fixed. Under this scenario, the various schemes proposed for a malleable job model are inapplicable. Few studies have addressed preemptive scheduling under a model of rigid jobs, where the preemption is "local," that is, the suspended job must be restarted on exactly the same set of processors on which they were suspended.

Chiang and Vernon [23] evaluate a preemptive scheduling strategy called "immediate service (IS)" for shared-memory systems. With this strategy, each arriving job is given an immediate timeslice of 10 minutes, by suspending one or more running jobs if needed. The selection of jobs for suspension is based on their instantaneous-xfactor, defined as (wait time + total accumulated run time) / (total accumulated run time). Jobs with the lowest instantaneous-xfactor are suspended. The IS strategy significantly decreases the average job slowdown for the traces simulated. A potential shortcoming of the IS strategy, however, is that its preemption decisions do not reflect the expected run time of a job. The IS strategy can be expected to significantly improve the slowdown of aborted jobs in the trace. Hence, it is unclear how much, if any, of the improvement in slowdown is experienced by the jobs that completed normally. However, no information is provided on how different job categories are affected.

Chiang et al. [22] examine the run-to-completion policy with a suspension policy that allows a job to be suspended at most once. Both this approach and the IS strategy limit the number of suspensions, whereas we use a "suspension factor" to control the rate of suspensions, without limiting the number of times a job can be suspended.

Parsons and Sevcik [25] discuss the design and implementation of a number of multiprocessor preemptive scheduling disciplines. They study the effect of preemption under the models of rigid, migratable, and malleable jobs. They conclude that their proposed preemption scheme may increase the response time for the model of rigid jobs.

So far, few simulation-based studies have been done on preemption strategies for clusters. With no process migration, the distributed-memory systems impose an additional constraint that a suspended job should get the same set of processors when it restarts. In this paper, we propose tunable suspension strategies for parallel job scheduling in environments where process migration is not feasible.

III. Workload Characterization

We perform simulation studies using a locally developed simulator with workload logs from different supercomputer centers. Most supercomputer centers keep a trace file as a record of the scheduling events that occur in the system. This file contains information about each job submitted and its actual execution. Typically the following data is recorded for each job:

- Name of job, user name, and so forth
- Job submission time
- Job resources requested, such as memory and processors
- User-estimated run time
- Time when job started execution
- Time when job finished execution

TABLE I
Job categorization criteria

	1 Proc	2-8 Procs	9-32 Procs	>32 Procs
0 - 10 min	VS Seq	VS N	VS W	VS VW
10 min - 1 hr	S Seq	S N	S W	S VW
1 hr - 8 hr	L Seq	L N	L W	L VW
>8 hr	VL Seq	VL N	VL W	VL VW

TABLE II
Job distribution by category - CTC Trace

	1 Proc	2-8 Procs	9-32 Procs	>32 Procs
0 - 10 min	14%	8%	13%	9%
10 min - 1 hr	18%	4%	6%	2%
1 hr - 8 hr	6%	3%	9%	2%
>8 hr	2%	2%	1%	1%

TABLE III
Job distribution by category - SDSC Trace

	1 Proc	2-8 Procs	9-32 Procs	>32 Procs
0 - 10 min	8%	29%	9%	4%
10 min - 1 hr	2%	8%	5%	3%
1 hr - 8 hr	8%	5%	6%	1%
>8 hr	3%	5%	3%	1%

From the collection of workload logs available from Feitelson's archive [41], subsets of the CTC workload trace, the SDSC workload trace and the KTH workload trace were used to evaluate the various schemes. The CTC trace was logged from a 430-node IBM SP2 system at the Cornell Theory Center, the SDSC trace from a 128-node IBM SP2 system at the San Diego Supercomputer Center, and the KTH trace from a 100-node IBM SP2 system at the Swedish Royal Institute of Technology. The other traces did not contain user estimates of run time. We observed similar performance trends with all the three traces. In order to minimize the number of graphs, we report the performance results for CTC and SDSC traces alone. This selection is purely arbitrary.

Although user estimates are known to be quite inaccurate in practice, as explained above, we first studied the effect of preemptive scheduling under the idealized assumption of accurate estimation, before studying the effect of inaccuracies in user estimates of job run time. Also, we first studied the impact of preemption under the assumption that the overhead for job suspension and restart were negligible and then studied the influence of the overhead.

TABLE IV
Average slowdown for various categories with nonpreemptive scheduling - CTC Trace

	1 Proc	2-8 Procs	9-32 Procs	>32 Procs
0 - 10 min	2.6	4.76	13.01	34.07
10 min - 1 hr	1.26	1.76	3.04	7.14
1 hr - 8 hr	1.13	1.43	1.88	1.63
>8 hr	1.03	1.05	1.09	1.15

TABLE V
Average slowdown for various categories with nonpreemptive scheduling - SDSC Trace

	1 Proc	2-8 Procs	9-32 Procs	>32 Procs
0 - 10 min	2.53	14.41	37.78	113.31
10 min - 1 hr	1.15	2.43	4.83	15.56
1 hr - 8 hr	1.19	1.24	1.96	2.79
>8 hr	1.03	1.09	1.18	1.43

Any analysis that is based only on the average slowdown or turnaround time of all jobs in the system cannot provide insights into the variability within different job categories. Therefore, in our discussion, we classify the jobs into various categories based on the run time and the number of processors requested, and we analyze the slowdown and turnaround time for each category.

To analyze the performance of jobs of different sizes and lengths, we classified jobs into 16 categories: considering four partitions for run time — Very Short (VS), Short (S), Long (L) and Very Long (VL) — and four partitions for the number of processors requested — Sequential (Seq), Narrow (N), Wide (W) and Very Wide (VW). The criteria used for job classification are shown in Table I. The distribution of jobs in the trace, corresponding to the sixteen categories, is given in Tables II and III.

Tables IV and V show the average slowdowns for the different job categories under a nonpreemptive aggressive backfilling strategy. The overall slowdown for the CTC trace was 3.58, and for the SDSC trace was 14.13. Even though the overall slowdowns are low, from the tables one can observe that some of the Very Short categories have slowdowns as high as 34 (CTC trace) and 113 (SDSC trace). Preemptive strategies aim at reducing the high average slowdowns for the short categories without significant degradation to long jobs.

IV. Selective Suspension

We first propose a preemptive scheduling scheme called Selective Suspension (SS), where an idle job may preempt a running job if its "suspension priority" is sufficiently higher than the running job. An idle job attempts to suspend a collection of running jobs so as to obtain enough free processors. In order to control the rate of suspensions, a suspension factor (SF) is used. This specifies the minimum ratio of the suspension priority of a candidate idle job to the suspension priority of a running job for preemption to occur. The suspension priority used is the xfactor of the job.

A. Theoretical Analysis

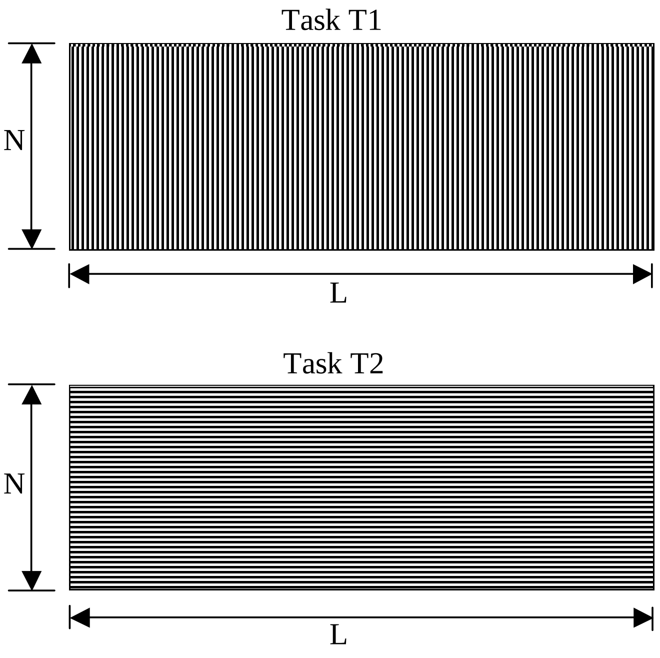

Fig. 3. Two simultaneously submitted tasks T1 and T2, each requiring 'N' processors for 'L' seconds.

Let T_1 and T_2 be two tasks submitted to the scheduler at the same time. Let both tasks be of the same length and require the entire system for execution, with the system being free when the two tasks are submitted. Let "s" be the suspension factor. Before starting, both tasks have a suspension priority of 1. The suspension priority of a task remains constant when the task executes and increases when the task waits. One of the two tasks, say T_1, will start instantly. The other task, say T_2, will wait until its suspension priority τ_2 becomes s times the priority of T_1 before it can preempt T_1. Now T_1 will have to wait until its suspension priority τ_1 becomes s times τ_2 before it can preempt T_2. Thus, execution of the two tasks will alternate, controlled by the suspension factor. Figures 4, 5, and 6 show the execution pattern of the tasks T_1 and T_2 for various values of SF. The optimal value for SF, to restrict the number of repeated suspensions by two similar tasks arriving at the same time, can be obtained as follows:

Let τ_w represent the suspension priority of the waiting job and τ_r represent the suspension priority of the running job.

The condition for the first suspension is

τ_w = s.

The preemption swaps the running job and the waiting job. Thus, after the preemption, τ_w = 1 and τ_r = s.

The condition for the second suspension is

$\tau_w = s\tau_r$

$\tau_w = s^2$.

Similarly, the condition for the n^{th} suspension is $\tau_w = s^n$.

The lowest value of s for which at most *n* suspensions occur is given by

$\tau_w = s^{n+1}$, when the running job completes.

When the running job completes,

$\tau_w = \frac{wait\ time + run\ time}{run\ time}$,

that is, τ_w = 2, since the wait time of the waiting job = the run time of the running job

s^{n+1} = 2 and s = $2^{\frac{1}{n+1}}$.

Thus, if the number of suspensions is to be 0, then s = 2. For at most one suspension, s = $\sqrt{2}$. With s = 1, the number of suspensions is very large, bounded only by the granularity of the preemption routine.

With all jobs having equal length, any suspension factor greater than 2 will not result in suspension and will be the same as a suspension factor of 2. However, with jobs of varying length, the number of suspensions reduces with higher suspension factors. Thus, to avoid thrashing and to reduce the number of suspensions, we use different suspension factors between 1.5 and 5 in evaluating our schemes.

B. Preventing Starvation without Reservation Guarantees

With priority-based suspension, an idle job can preempt a running job only if its priority is at least SF times greater than the priority of the running job. All the idle jobs that are able to find the required number of processors by suspending lower

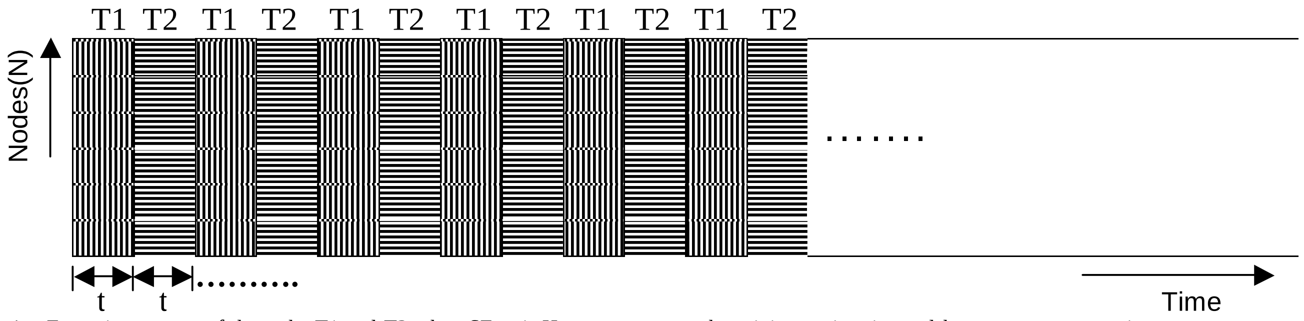

Fig. 4. Execution pattern of the tasks T1 and T2 when SF = 1. Here, t represents the minimum time interval between two suspensions.

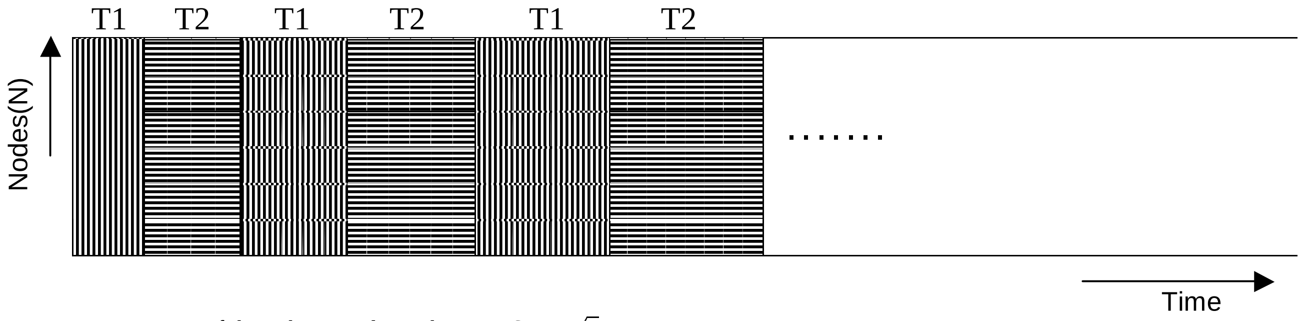

Fig. 5. Execution pattern of the tasks T1 and T2 when $1 < \mathrm{SF} < \sqrt{2}$.

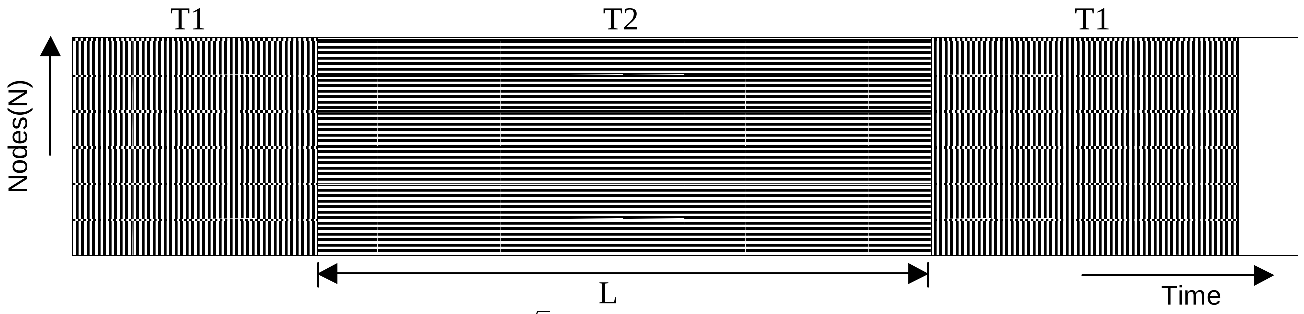

Fig. 6. Execution pattern of the tasks T1 and T2 when SF = $\sqrt{2}$.

priority running jobs are selected for execution by preempting the corresponding jobs. All backfilling scheduling schemes use job reservations for one or more jobs at the head of the idle queue as a means of guaranteeing finite progress and thereby avoiding starvation. But start time guarantees do not have much significance in a preemptive context. Even if we give start time guarantees for the jobs in the idle queue, they are not guaranteed to run to completion. Since the SS strategy uses the expected slowdown (xfactor) as the suspension priority, there is an automatic guarantee of freedom from starvation: ultimately any job's xfactor will get large enough that it will be able to preempt some running job(s) and begin execution. Thus, one can use backfilling without the usual reservation guarantees. We therefore remove guarantees for all our preemption schemes.

Jobs in some categories inherently have a higher probability of waiting longer in the queue than do jobs with comparable xfactor from other job categories. For example, consider a VW job needing 300 processors, and a Sequential job in the queue at the same time. If both jobs have the same xfactor, the probability that the Sequential job finds a running job to suspend is higher than the probability that the VW job finds enough lower-priority running jobs to suspend. Therefore, the average slowdown of the VW category will tend to be higher than the Sequential category. To redress this inequity, we impose a restriction that the number of processors requested by a suspending job should be at least half of the number of processors requested by the job that it suspends, thereby preventing the wide jobs from being suspended by the narrow jobs. The scheduler periodically (after every minute) invokes the preemption routine.

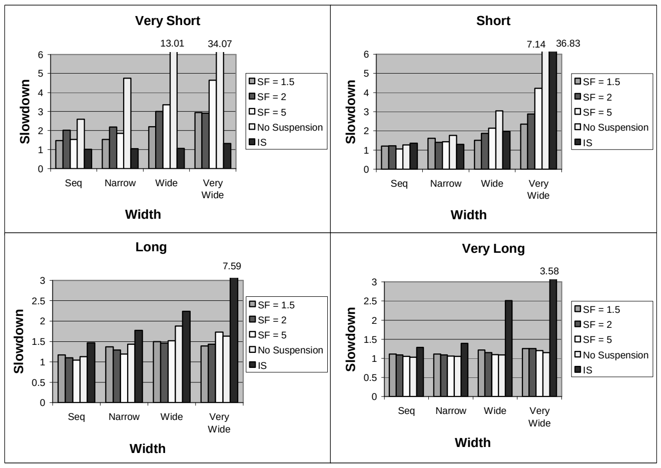

Fig. 7. Average slowdown: SS scheme, CTC trace. Compared to NS, SS provides significant benefit for the VS, S, W, and VW categories; slight improvement for most of L categories; but a slight deterioration for the VL categories. Compared to IS, SS performs better for all the categories except for the VS categories.

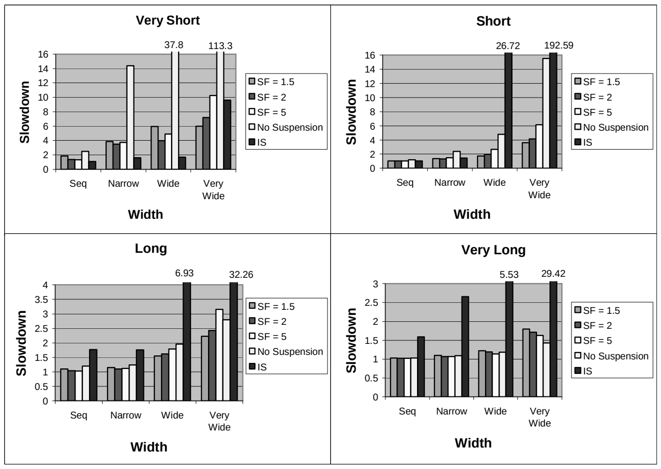

Fig. 9. Average slowdown: SS scheme, SDSC trace. Compared to NS, SS provides significant benefit for the VS, S, W, and VW categories; slight improvement for most of L categories; but a slight deterioration for the VL categories. Compared to IS, SS performs better for all the categories except for the VS categories.

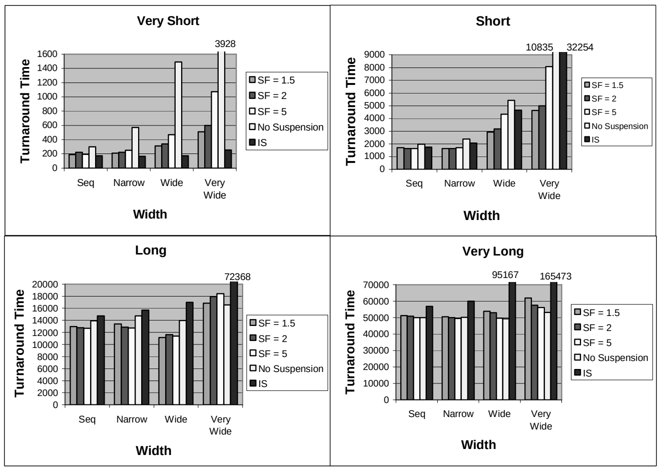

Fig. 8. Average turnaround time: SS scheme, CTC trace. The trends are similar to those with the average slowdown metric (Fig. 7).

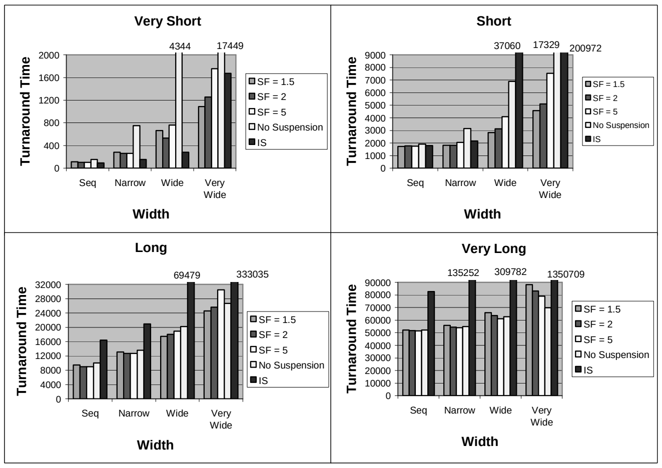

Fig. 10. Average turnaround time: SS scheme, SDSC trace. The trends are similar to those with the average slowdown metric (Fig. 9).

C. Algorithm

Let τ_i be the suspension priority for a task t_i which requests n_i processors. Let N_i represent the set of processors allocated to t_i. Let F_t represent the set of free processors and f_t represent the number of free processors at time t when the preemption is attempted.

The set of tasks that can be preempted by task t_i is given by

$$C_i = \{t_j \mid \tau_i \geq (SF)\tau_j \text{ and } \frac{n_j}{n_i} \leq 2\}.$$

Task t_i can be scheduled by preempting one or more tasks in C_i if and only if

$$n_i \leq (f_t + \sum_{j:t_j \in C_i} n_j).$$

Let $(t_1, t_2, t_3, \ldots, t_x)$ be the elements of C_i. Let ϕ be a permutation of (1,2,3,...,x) such that $n_{\phi(1)} \geq n_{\phi(2)} \geq n_{\phi(3)}, \ldots, n_{\phi(x-1)} \geq n_{\phi(x)}$. (If $n_{\phi(l)} = n_{\phi(l+1)}$, then $\tau_l \leq \tau_{l+1}$. If $\tau_l = \tau_{l+1}$, then the start time of $t_l \leq$ the start time of t_{l+1}. If the start time of t_l = the start time of t_{l+1}, then the queue time of $t_l <$ the queue time of t_{l+1}.) So,

$$\exists k_i \mid 1 \leq k_i \leq x \text{ and } (\sum_{j=1}^{k_i-1} n_{\phi(j)}) < (n_i - f_t) \text{ and}$$

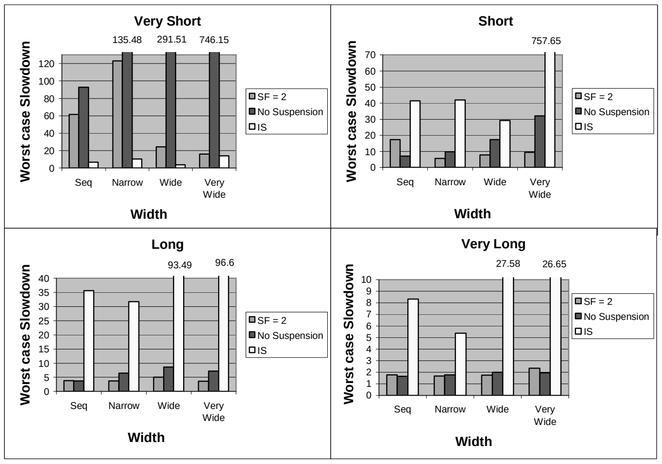

Fig. 11. Worst-case slowdown: SS scheme, CTC trace. SS is much better than NS for most of the categories and is slightly worse for some of the VL categories. Compared to IS, SS is much better for all the categories except for the VS categories.

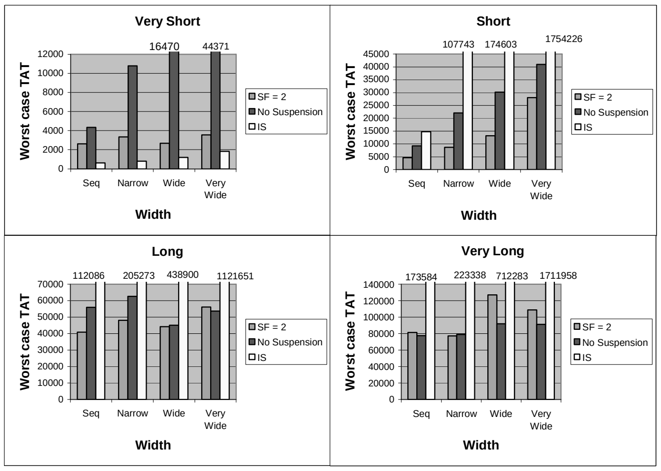

Fig. 12. Worst-case turnaround time: SS scheme, CTC trace. The trends are similar to those with the worst-case slowdown metric (Fig. 11).

$$(\sum_{j=1}^{k_i} n_{\phi(j)}) \geq (n_i - f_t).$$

The set of tasks preempted by task t_i is given by

$$P_i = \{t_{\phi(r)} \mid 1 \leq r \leq k_i\}.$$

If t_i is a previously suspended task attempting reentry, then it has to get the same set of processors that it was using before it was suspended. Here we remove the restriction that the number of processors requested by a suspending job should be at least half of the number of nodes requested by the job that it suspends. Otherwise if a VW job happens to suspend a narrow job, then in the worst-case, the narrow job has to wait till the VW job completes to get rescheduled. So the set of tasks that can be preempted by t_i in this case is given by

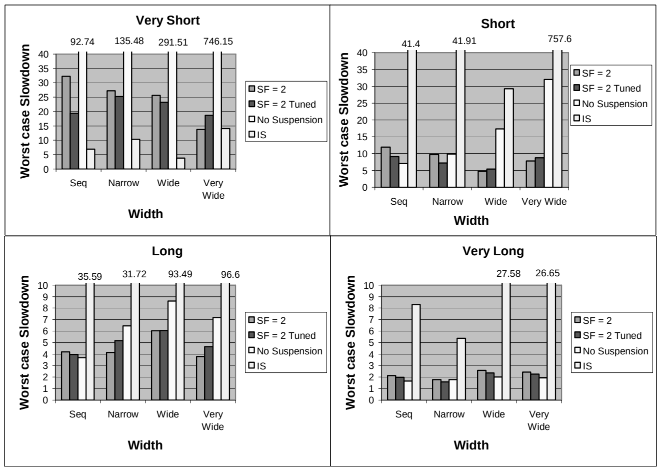

Fig. 13. Worst-case slowdown for the TSS scheme: CTC trace. TSS improves the worst-case slowdowns for many categories without affecting the worst-case slowdowns for other categories.

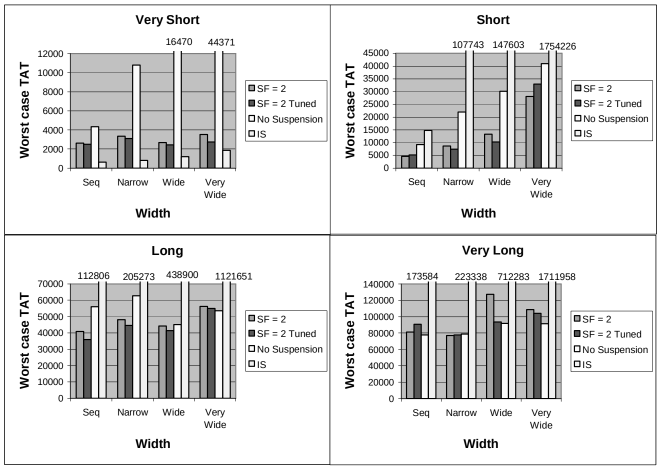

Fig. 14. Worst-case turnaround times for the TSS scheme: CTC trace. TSS improves the worst-case turnaround times for many categories without affecting the worst-case tunraround times for other categories.

$$C_i = \{t_j \mid \tau_i \geq (SF)\tau_j \text{ and } N_i \cap N_j \neq \emptyset\}.$$

Task t_i can be scheduled by preempting one or more tasks in C_i if and only if

$$N_i \subseteq (F_t \bigcup_{j: t_j \in C_j} N_j).$$

D. Results

We compare the SS scheme run under various suspension factors with the No-Suspension (NS) scheme with aggressive backfilling and the IS scheme. From Figs. 7 – 10, we can see that the SS scheme provides significant improvement for the

Pseudocode for the selective suspension scheme

```
Sort the list of running jobs in ascending order of suspension priority
Sort the list of idle jobs in descending order of suspension priority
for each idle job
do
    set the candidate_job_set to be the null set
    if  (idle job is a suspended job)
    then
        goto already_suspended
    else
        available_processors = number of free processors
        for each running job
        do
            if  (number of processors requested by the idle job > available_processors)
            then
                if  ((suspension priority of the idle job >= SF * suspension priority of the running job)  &&
                    (number of processors used by the running job <= 2 * number of processors requested by the idle job))
                then
                    available_processors = available_processors + number of processors used by the running job
                    candidate_job_set = {candidate_job_set} u {running job}
                else
                    goto next_idle_job
                end if
            else
                goto suspend_jobs_1
            end if
        done
        end for
    end if
    goto next_idle_job
    already_suspended:
        set available_processor_set to the set of free processors
        for each running job
        do
            if  (set of processors requested by idle job is not a subset of available_processor_set)
            then
                if  (suspension priority of the idle job >= SF * suspension priority of the running job)
                then
                    if  ({set of processors used by the running job}  n {set of processors requested by the idle job} is not empty)
                    then
                        available_processor_set = {available_processor_set} u {set of processors used by running job}
                        candidate_job_set = {candidate_job_set} u {running job}
                    end  if
                else
                    goto next_idle_ job
                end if
            else
                goto suspend_job_2
            end if
        done
        end for
    goto next_idle_job
    suspend_jobs_1:
        sort job(s) in candidate_job_set in descending order of number of processors used
        available_processors = number of free processors
        for each job in candidate_job_set
        do
            if  (number of processors requested by the idle job > available_processors)
            then
                suspend the job
                available_processors = available_processors + number of processors used by the suspended job
            else
                schedule the idle job
                goto next_idle_job
            end if
        done
        end for
    goto next_idle_job
    suspend_jobs_2:
        suspend all jobs in the candidate_job_set
        schedule the idle job
     next_idle_job:
        do nothing
done
end for
```

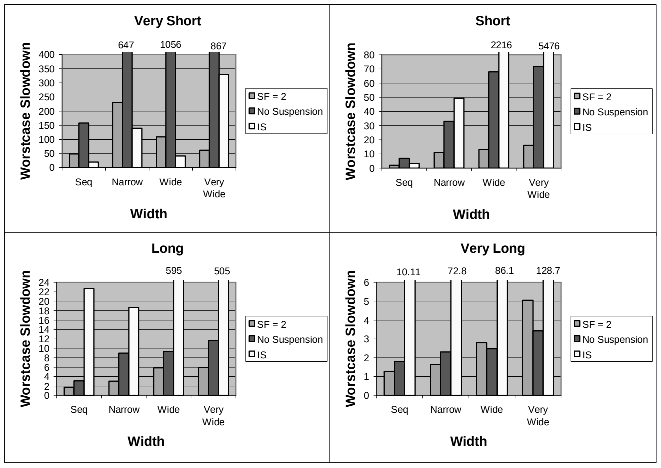

Fig. 15. Worst-case slowdown: SS scheme, SDSC trace. SS is much better than NS for most of the categories and is slightly worse for some of the VL categories. Compared to IS, SS is much better for all the categories except for the VS categories.

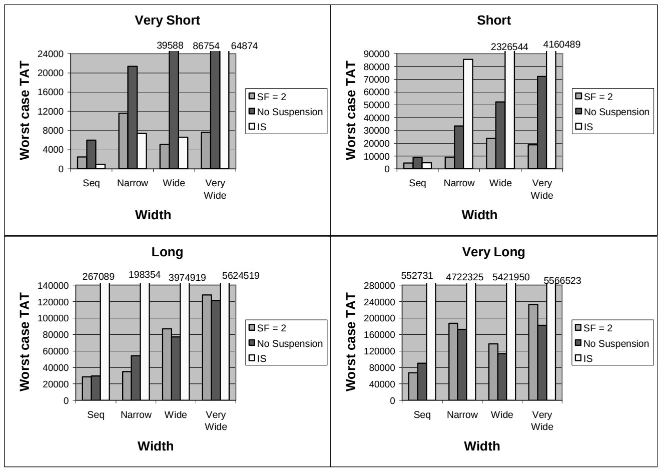

Fig. 16. Worst-case turnaround time: SS scheme, SDSC trace. The trends are similar to those with the worst-case slowdown metric (Fig. 15).

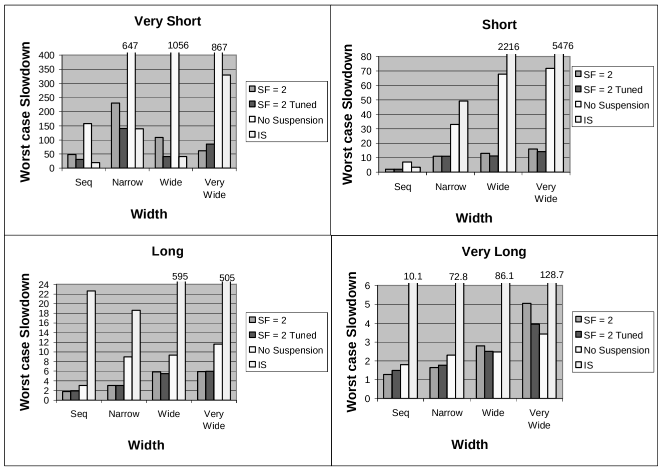

Fig. 17. Worst-case slowdown for the TSS scheme: SDSC trace. TSS improves the worst-case slowdowns for many categories without affecting the worst-case slowdowns for other categories.

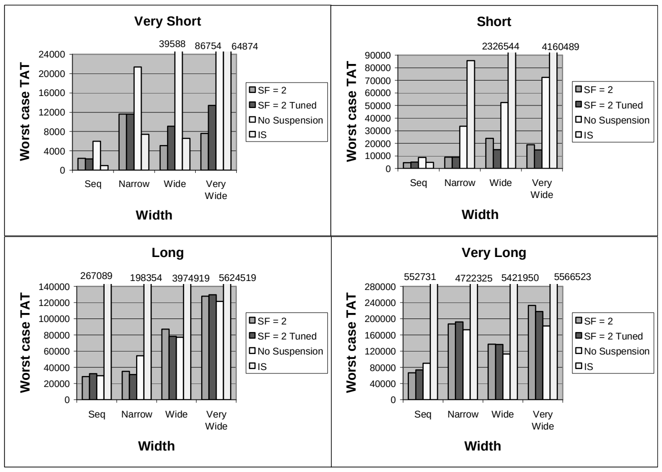

Fig. 18. Worst-case turnaround times for the TSS scheme: SDSC trace. TSS improves the worst-case turnaround times for many categories without affecting the worst-case tunraround times for other categories.

Very-Short (VS) and Short (S) length categories and Wide (W) and Very-Wide (VW) width categories. For example, for the VS-VW category, slowdown is reduced from 113 for the NS scheme to 7 for SS with SF = 2 for the SDSC trace (reduced from 34 for the NS scheme to under 3 for SS with SF = 2 for the CTC trace).

For the VS and S length categories, a lower SF results in lowered slowdown and turnaround time. This is because a lower SF increases the probability that a job in these categories will suspend a job in the Long (L) or Very-Long (VL) category. The same is also true for the L length category, but the effect of change in SF is less pronounced. For the VL length category, there is an opposite trend with decreasing SF: the slowdown and turnaround times worsen. This is due to the increasing probability that a Long job will be suspended by a job in a shorter category as SF decreases. In comparison to the base No-Suspension (NS) scheme, the SS scheme provides significant benefits for the VS and S categories and a slight improvement for most of the Long categories but is slightly worse for the VL categories. The performance of the IS scheme is very good for the VS categories. It is better than the SS scheme for the VS categories and worse for the other categories. Although the overall slowdown for IS is considerably less than for the No-Suspension scheme, it is not better than SS. Moreover, with IS the VW and VL categories get significantly worse.

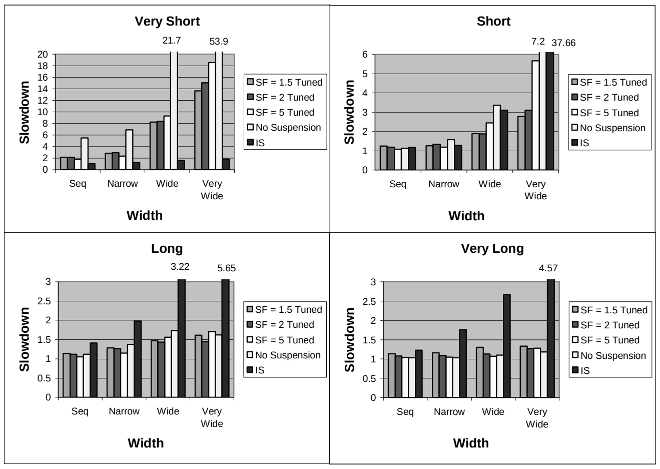

Fig. 19. Average slowdown: Inaccurate estimates of run time; CTC trace. Compared to NS, SS improves the slowdowns for most of the categories with little deterioration to other categories. The performance of IS is bad for the long jobs.

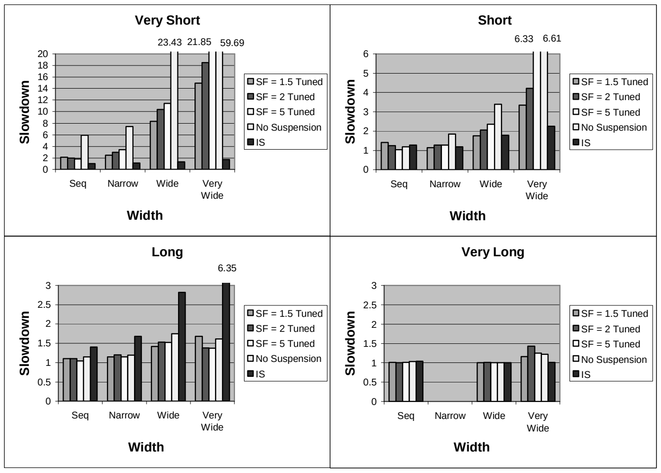

Fig. 21. Average slowdown of badly estimated jobs: CTC trace. Compared to NS, SS provides a slight improvement in slowdowns for many categories. SS tends to penalize the badly estimated jobs in VS categories. IS gives better performance for VS, S and VL categories.

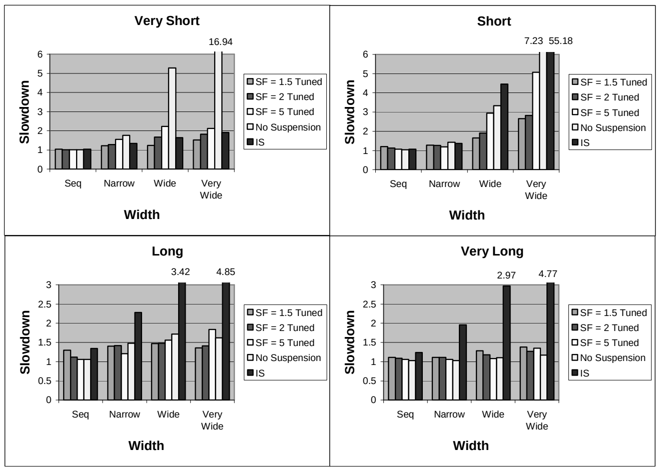

Fig. 20. Average slowdown of well estimated jobs: CTC trace. Compared to NS, SS significantly improves the slowdowns for most of the categories with little deterioration to other categories. The performance of SS is better than or comparable to IS for VS categories.

E. Tunable Selective Suspension (TSS)

From the graphs of the previous section, one can observe that the SS scheme significantly improves the average slowdown and turnaround time of various job categories. But from a practical point of view, the worst-case slowdowns and turnaround times are very important. A scheme that improves the average slowdowns and turnaround times for most of the categories but makes the worst-case slowdown and turnaround time for the long categories worse, is not an acceptable scheme. For example, a delay of 1 hour for a 10-minute job (slowdown = 7) is tolerable, whereas a slowdown of 7 for a 24-hour job is unacceptable. Figure 11 compares the worst-case slowdowns for SF = 2 with the worst-case slowdowns of the NS scheme and the IS scheme for the CTC trace. One can observe that the worst-case slowdowns with the SS scheme are much better than with the NS scheme for most of the cases. But the worst-case slowdowns for some of the long categories are worse than for the NS scheme.

Although the worst-case slowdown with SS is generally less than that with NS, the absolute worst-case slowdowns are much higher than the average slowdowns for some of the short categories. For the IS scheme, the worst-case slowdowns for the very short categories are lower, but they are very high for the long jobs, an unacceptable situation. Figure 12 compares the worst-case turnaround times for the SS scheme with worst-case turnaround times for the NS scheme and the IS scheme, for the CTC trace. Even though the trends observed here are similar to those with the worst-case slowdowns, the categories where SS is the best with respect to worst-case turnaround time are not same as the categories for which SS is the best with respect to worst-case slowdowns. This is because a job with the worst-case turnaround time need not be the one with worst-case slowdown. Similar trends can be observed for the SDSC trace from Figs. 15 and 16.

We next propose a tunable scheme to improve the worst-case slowdown and turnaround time without significant deterioration of the average slowdown and turnaround time. This scheme involves controlling the variance in the slowdowns and turnaround times by associating a limit with each job. Preemption of a job is disabled when its priority exceeds this limit. This limit is set to 1.5 times the average slowdown of the category that the job belongs to.

The candidate set of tasks that can be preempted by a task t_i is given by

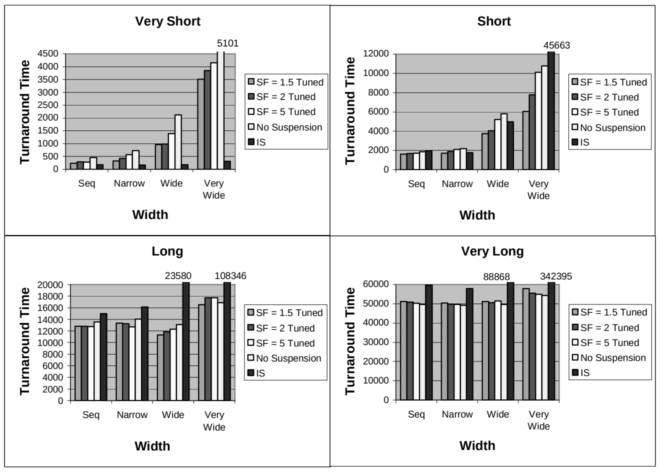

Fig. 22. Average turnaround time: Inaccurate estimates of run time; CTC trace. Compared to NS, SS improves the turnaround times for most of the categories with little deterioration to other categories. The performance of IS is bad for the long jobs.

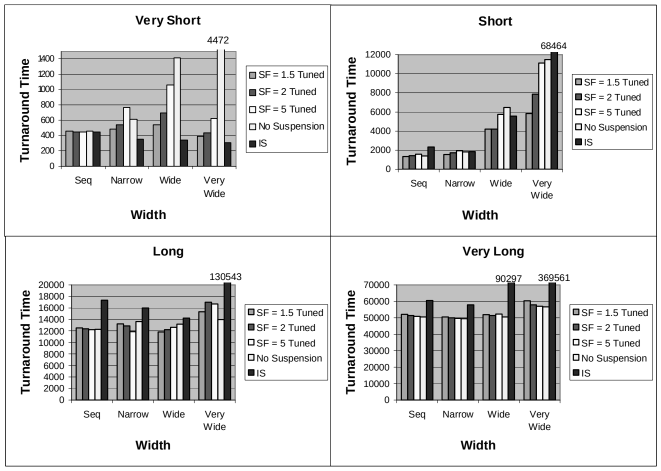

Fig. 23. Average turnaround time of well estimated jobs: CTC trace. Compared to NS, SS significantly improves the turnaround times for most of the categories with little deterioration to other categories. The performance of SS is comparable to IS for VS categories.

$$C_i = \{t_j \mid \tau_i \geq (SF)\tau_j \text{ and } \frac{n_j}{n_i} \leq 2 \text{ and}$$
$$\tau_j \leq 1.5 * SD_{avg}(category(t_j))\},$$

where SD_{avg}(category(t_j)) represents the average slowdown for the job category to which t_j belongs.

If t_i is a previously suspended task attempting reentry, then

$$C_i = \{t_j \mid \tau_i \geq (SF)\tau_j \text{ and } N_i \cap N_j \neq \emptyset \text{ and}$$
$$\tau_j \leq 1.5 * SD_{avg}(category(t_j))\}.$$

All the other conditions remain the same as mentioned in Section IV-C.

Figures 13 and 14 show the result of this tunable scheme for the CTC trace. It improves the worst-case slowdowns for some long categories (VL W, VL VW, L N) and some short categories (VS Seq, VS N, S Seq) without affecting the worst-case slowdowns of the other categories. It improves the worst-case turnaround times for most of the categories without affecting the worst-case turnaround times of the other categories. Figures 17 and 18 show similar trends for the SDSC trace. This scheme can also be applied to selectively tune the slowdowns or turnaround times for particular categories. The TSS scheme is used for all the subsequent experiments, and the term "Selective Suspension" or "SS" in the following sections refers to "Tunable Selective Suspension."

V. Impact of User Estimate Inaccuracies

We have so far assumed that the user estimates of job run time are perfect. Now, we consider the effect of user estimate inaccuracies on the proposed schemes. This analysis is needed for modeling an actual system workload. In this context, we believe that a problem has been ignored by previous studies when analyzing the effect of over estimation on scheduling strategies. Abnormally aborted jobs tend to excessively skew the average slowdown of jobs in a workload. Consider a job requesting a wall-clock limit of 24 hours, which is queued for 1 hour and then aborts within one minute due to some fatal exception. The slowdown of this job would be computed to be 60, whereas the average slowdown of normally completing long jobs is typically under 2. If even 5% of the jobs have a high slowdown of 60, while 95% of the normally completing jobs have a slowdown of 2, the average slowdown over all jobs would be around 5. Now consider a scheme such as the

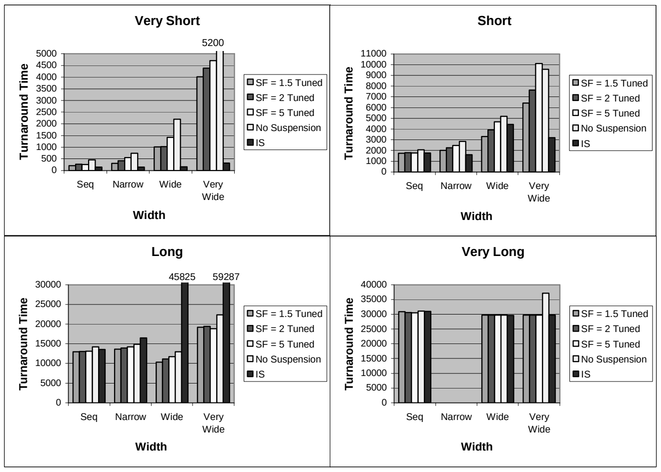

Fig. 24. Average turnaround time of badly estimated jobs: CTC trace. Compared to NS, SS provides a slight improvement in turnaround times for many categories. SS tends to penalize the badly estimated jobs in VS categories. IS gives better performance for VS, S, and VL categories.

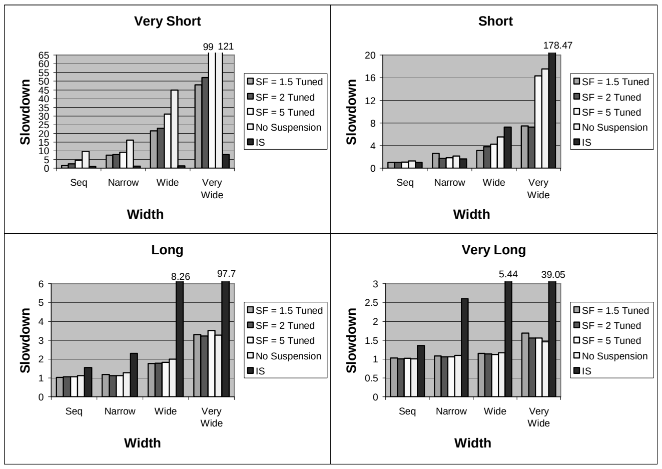

Fig. 25. Average slowdown: Inaccurate estimates of run time; SDSC trace. Compared to NS, SS improves the slowdowns for most of the categories with little deterioration to other categories. The performance of IS is bad for the long jobs.

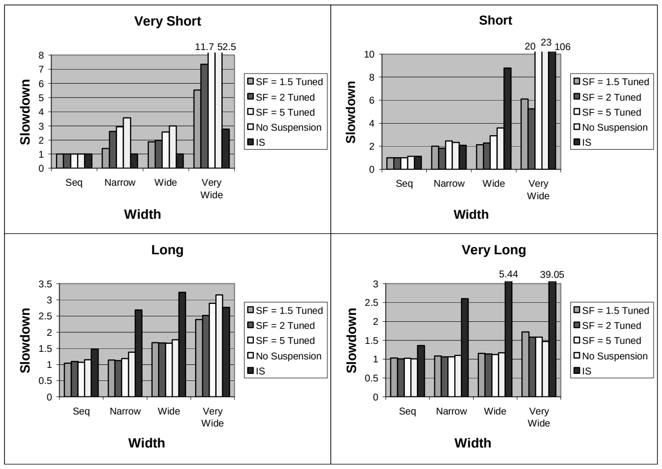

Fig. 26. Average slowdown of well estimated jobs: SDSC trace. Compared to NS, SS significantly improves the slowdowns for most of the categories with little deterioration to other categories. The performance of IS is bad except for VS categories.

speculative backfilling strategy evaluated in [29]. With this scheme, a job is given a free timeslot to execute in, even if that slot is considerably smaller than the requested wall-clock limit. Aborting jobs will quickly terminate, and since they did not have to be queued till an adequately long window was available, their slowdown would decrease dramatically with the speculative backfilling scheme. As a result, the average slowdown of the entire trace would now be close to 2, assuming that the slowdown of the normally completing jobs does not change significantly. A comparison of the average slowdowns would seem to indicate that the speculative backfilling scheme results in a significant improvement in job slowdown from 5 to 2. However, under the above scenario, the change is due only to the small fraction of aborted jobs, and not due to any benefits to the normal jobs. In order to avoid this problem, we group the jobs into two different estimation categories:

- Jobs that are well estimated (the estimated time is not more than twice the actual run time of that job) and
- Jobs that are poorly estimated (the estimated run time is more than twice the actual run time).

Within each group, the jobs are further classified into 16 categories based on their actual run time and the number of processors requested.

One can observe from Figs. 19 and 25 that the Selective Suspension scheme improves the slowdowns for most of the categories without adversely affecting the other categories. The slowdowns for the short and wide categories are quite high compared to the other categories, mainly because of the overestimation. Since the suspension priority used by the SS scheme is xfactor, it favors the short jobs. But if a short job

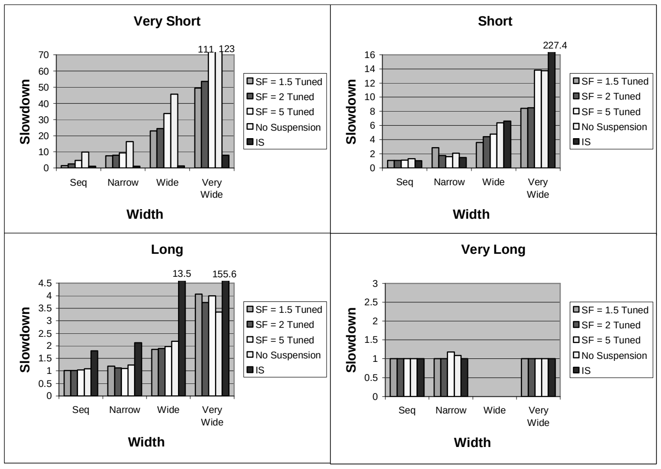

Fig. 27. Average slowdown of badly estimated jobs: SDSC trace. Compared to NS, SS provides a slight improvement in slowdowns for many categories. SS tends to penalize the badly estimated jobs in VS categories.

was badly estimated, it would be treated as a long job and its priority would increase only gradually. So, it will not be able to suspend running jobs easily and will end up with a high slowdown. This situation does not happen with IS because of the 10-minute time quantum for each arriving job irrespective of the estimated run time, and therefore the slowdowns for the very short category (whose length is less than or equal to 10 minutes) is better in IS than other schemes. For the other categories, however, SS performs much better than IS.

Figures 22 and 28 compare the average turnaround times for the SS scheme with that of the NS and IS schemes for the CTC and SDSC traces, respectively. The improvement in performance for the short and wide categories is much less when compared to the improvement achieved with the accurate

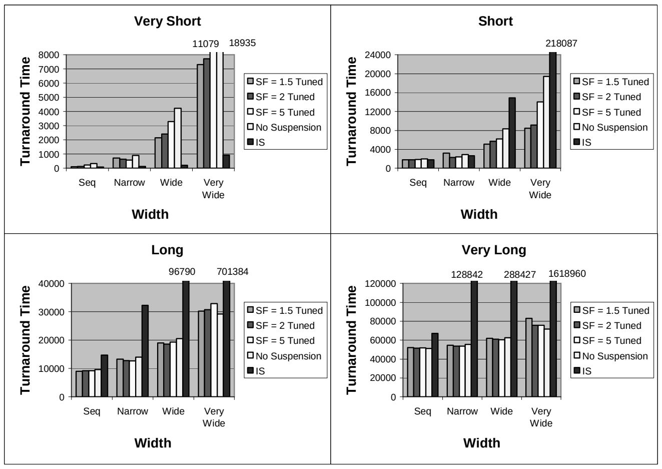

Fig. 28. Average turnaround time: Inaccurate estimates of run time; SDSC trace. Compared to NS, SS improves the turnaround times for most of the categories with little deterioration to other categories. The performance of IS is bad except for VS categories.

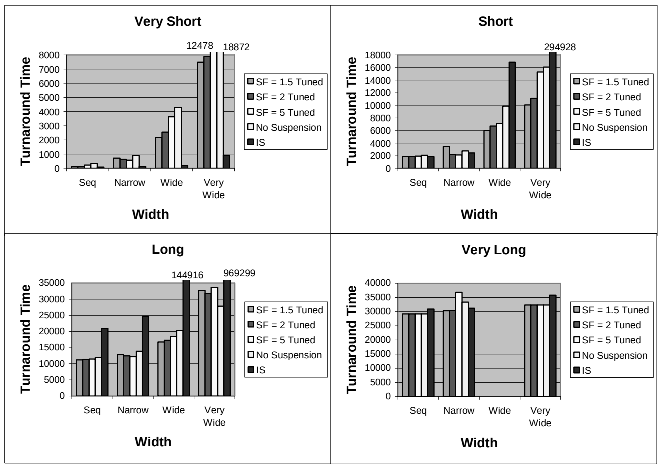

Fig. 30. Average turnaround time of badly estimated jobs: SDSC trace. Compared to NS, SS provides a slight improvement in turnaround times for many categories. SS tends to penalize the badly estimated jobs in VS categories. The performance of IS is bad except for VS categories.

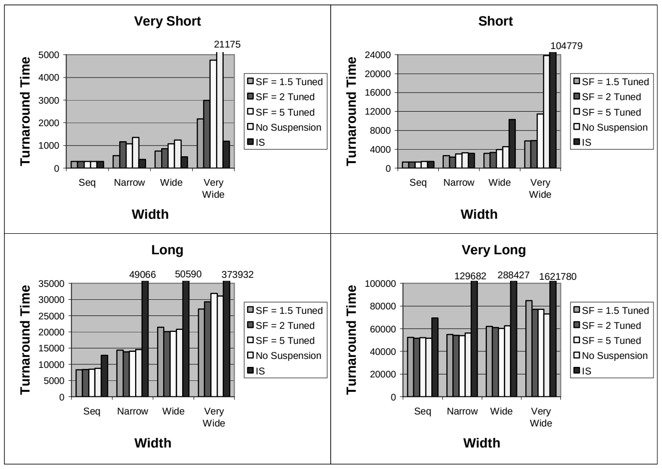

Fig. 29. Average turnaround time of well estimated jobs: SDSC trace. Compared to NS, SS significantly improves the turnaround times for most of the categories with little deterioration to other categories. The performance of IS is very bad for long jobs.

user estimate case. The reasoning provided above for the increase in slowdowns for the short and wide categories holds for this case also. The seemingly long jobs (badly estimated short jobs) are unable to suspend running jobs easily and have to wait in the queue for a longer time, thus ending up with a high turnaround time. From Figs. 20 - 21 and Figs. 26 - 27, the higher slowdowns for the VS categories with SS clearly are due to the badly estimated jobs. Figures. 23 - 24 and Figures 29 - 30 show that the reduction in the percentage improvement of the average turnaround times for the short and wide categories in SS is due to the badly estimated jobs. One can also observe that, for the well estimated jobs, SS is better than or comparable to IS for the VS categories and SS outperforms IS in all other categories.

A. Modeling of Job Suspension Overhead

We have so far assumed no overhead for preemption of jobs. In this section, we present simulation results that incorporate overheads for job suspension. Since the job traces did not have information about job memory requirements, we considered the memory requirement of jobs to be random and uniformly distributed between 100 MB and 1 GB. The overhead for suspension is calculated as the time taken to write the main memory used by the job to the disk. The memory transfer rate that we considered is based on the following scenario: with a commodity local disk for every node, with each node being a quad, the transfer rate per processor was assumed to be 2 MB/s (corresponding to a disk bandwidth of 8 MB/s).

Figures 31 and 32 compare respectively the slowdowns and turnaround times of the proposed tunable scheme with NS and IS in the presence of overhead for job suspension/restart for the CTC trace. Figures 33 and 34 compare respectively the slowdowns and turnaround times of the proposed tunable scheme with NS and IS in the presence of overhead for job suspension/restart for the SDSC trace. One can observe that overhead does not significantly affect the performance of the SS scheme.

VI. LOAD VARIATION

We have so far seen the performance of the Selective Suspension scheme under normal load. In this section, we present the performance of the SS scheme under different load conditions starting from the normal load (original trace) and increasing load until the system reaches saturation. The

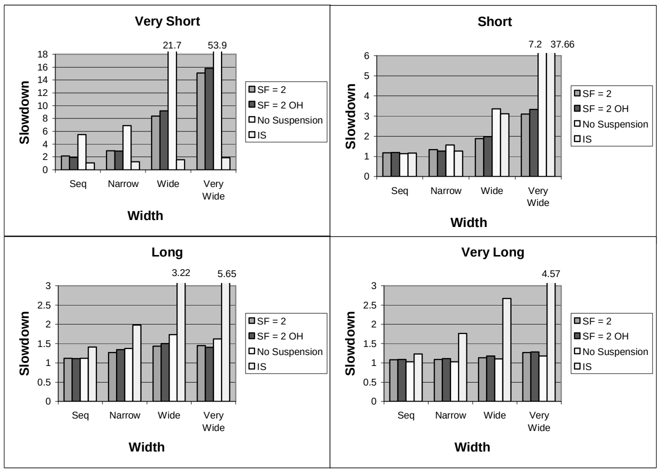
Fig. 31. Average slowdown with modeling of overhead for suspension/restart: CTC trace. The impact of overhead on the performance of SS scheme is minimal.

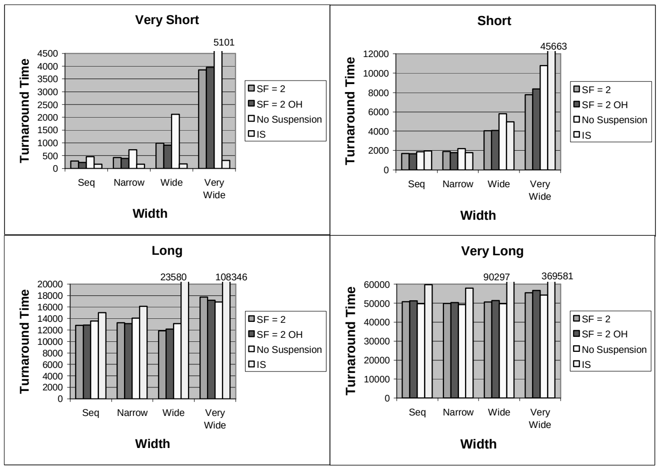
Fig. 32. Average turnaround time with modeling of overhead for suspension/restart: CTC trace. The impact of overhead on the performance of SS scheme is minimal.

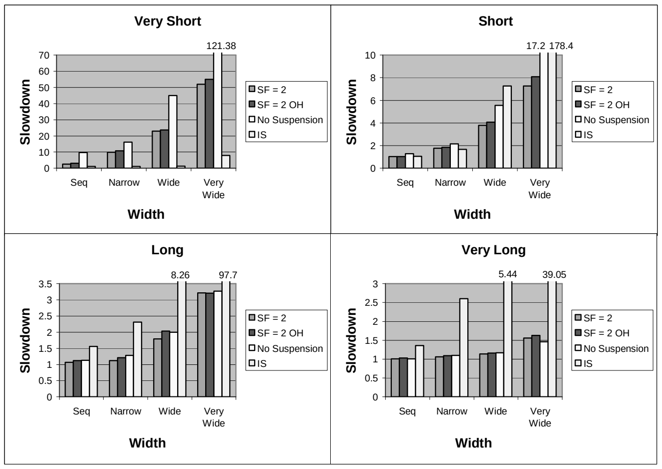
Fig. 33. Average slowdown with modeling of overhead for suspension/restart: SDSC trace. The impact of overhead on the performance of SS scheme is minimal.

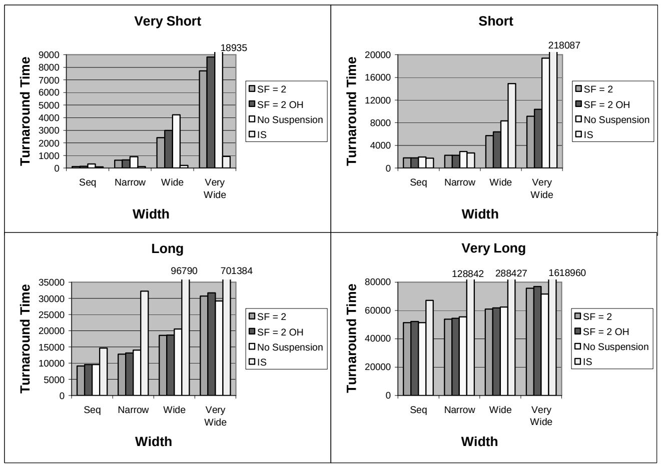
Fig. 34. Average turnaround time with modeling of overhead for suspension/restart: SDSC trace. The impact of overhead on the performance of SS scheme is minimal.

different loads correspond to modification of the traces by dividing the arrival times of the jobs by suitable constants, keeping their run time the same as in the original trace. For example, the job trace for a load factor of 1.1 is obtained by dividing the arrival times of the jobs in the original trace by 1.1.

For simplicity, we have reduced the number of job categories from sixteen to four for the load variation studies: two categories based on their run time — Short (S) and Long (L) — and two categories based on the number of processors requested — Narrow (N) and Wide (W). The criteria used for job classifications are shown in Table VI. The distribution of jobs in the CTC and SDSC traces, corresponding to the four categories, is given in Table VII and Table VIII, respectively.

Figures 35 and 38 show the overall system utilization for different schemes under different load conditions for the CTC and SDSC traces. One can observe that the SS scheme is able to achieve a better utilization than the NS scheme at higher loads, whereas the overall system utilization is very low under the IS scheme. Also, there is no significant increase in the overall system utilization (for both the SS and NS schemes) when the load factor is increased beyond 1.6 (for CTC) and 1.3 (for SDSC). This result indicates that the system reaches saturation at a load factor of 1.6 (for CTC) and 1.3 (for SDSC). We report the performance of the SS scheme for various load factors between 1.0 (normal) and 1.6 for the CTC trace and between 1.0 and 1.3 for the SDSC trace.

Figures 36 and 37 and Figures 39 and 40 compare the performance of the SS scheme with the NS and IS schemes for different job categories under different load conditions

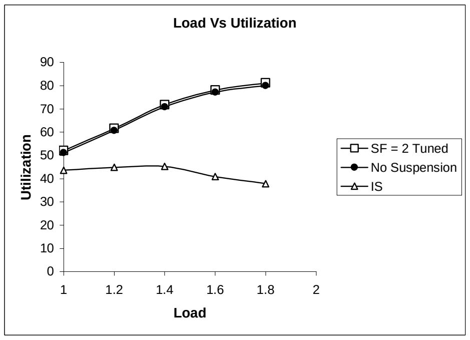

Fig. 35. Overall system utilization under different load conditions: CTC trace. The overall system utilization with the SS scheme is better than or comparable to the NS scheme. The performance of IS is much worse.

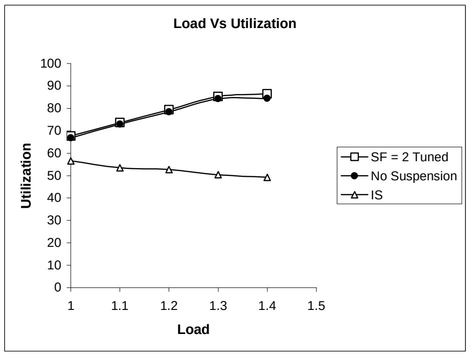

Fig. 38. Overall system utilization under different load conditions: SDSC trace. The overall system utilization with the SS scheme is better than or comparable to the NS scheme. The performance of IS is much worse.

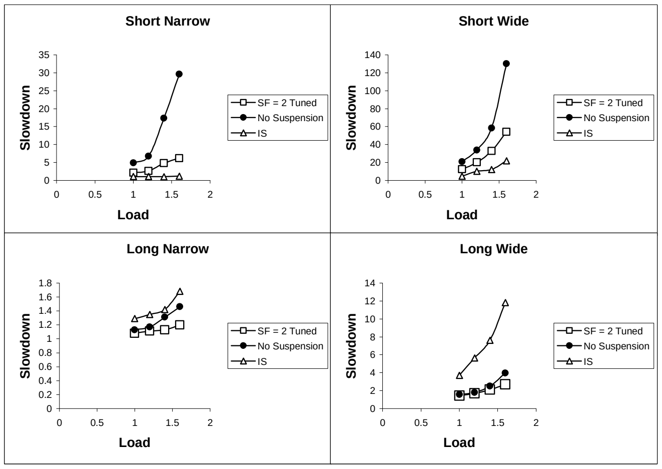

Fig. 36. Average slowdown: varying load; CTC trace. The improvements achieved by the SS scheme are more pronounced under high load.

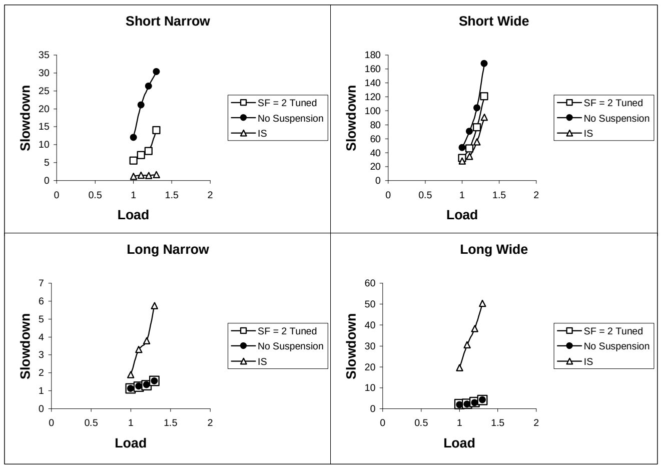

Fig. 39. Average slowdown: varying load; SDSC trace. The improvements achieved by the SS scheme are more pronounced under high load.

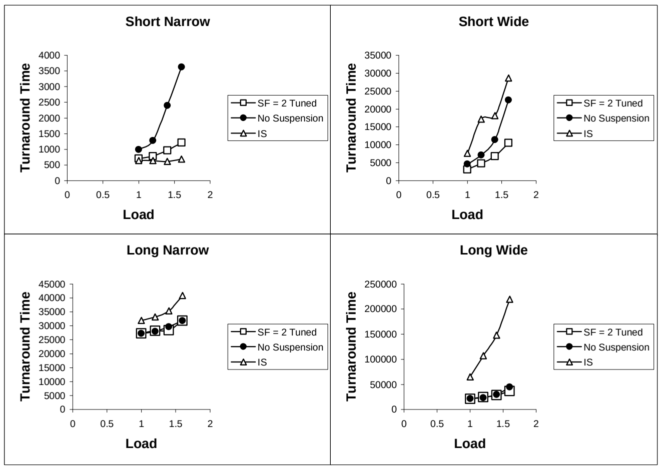

Fig. 37. Average turnaround time: varying load; CTC trace. The improvements achieved by the SS scheme are more pronounced under high load.

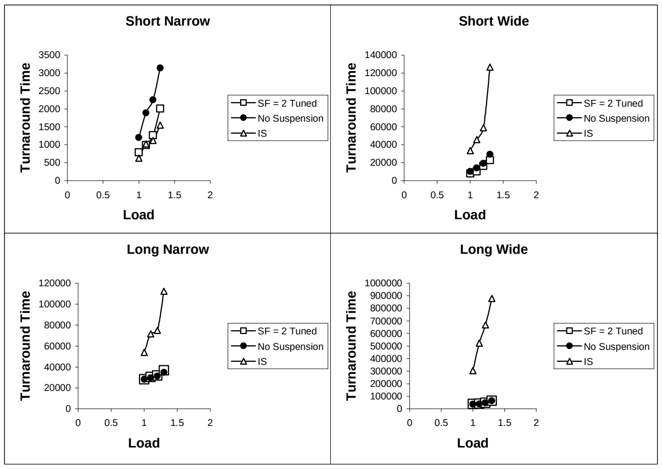

Fig. 40. Average turnaround time: varying load; SDSC trace. The improvements achieved by the SS scheme are more pronounced under high load.

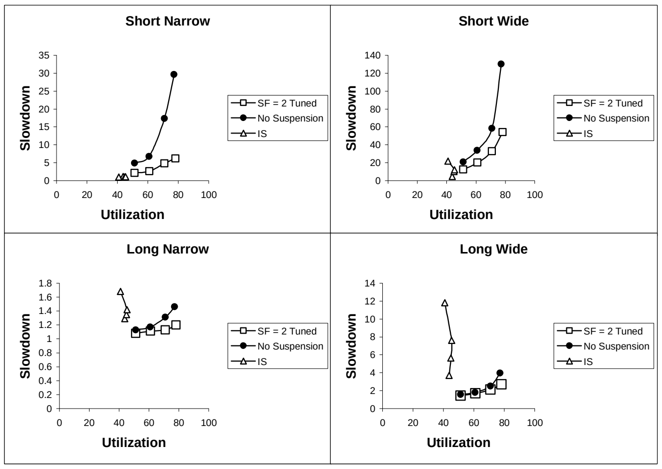

Fig. 41. Average slowdown versus system utilization: CTC trace. SS provides better performance even if the system is heavily utilized.

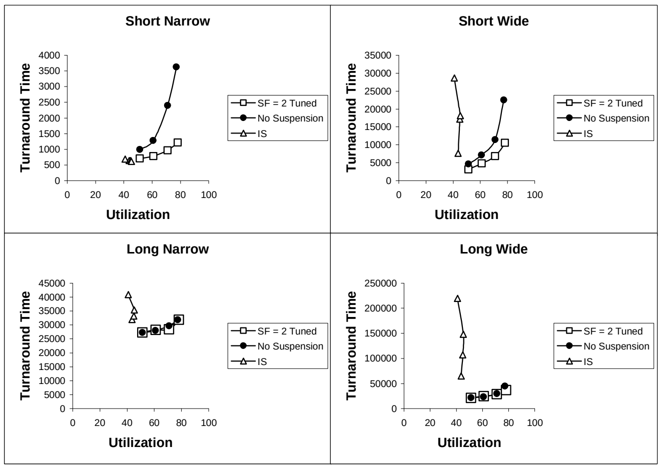

Fig. 42. Average turnaround time versus system utilization: CTC trace. SS provides better performance even if the system is heavily utilized.

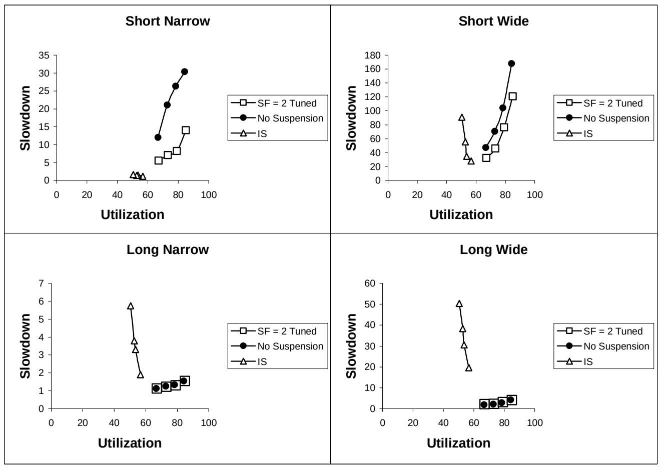

Fig. 43. Average slowdown versus system utilization: SDSC trace. SS provides better performance even if the system is heavily utilized.

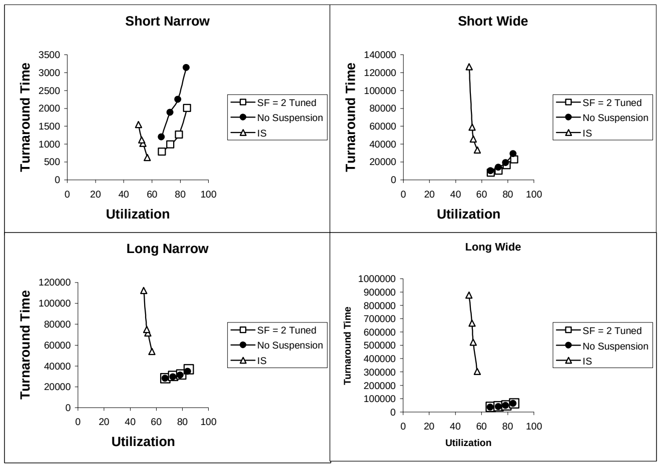

Fig. 44. Average turnaround time versus system utilization: SDSC trace. SS provides better performance even if the system is heavily utilized.

TABLE VI

JOB CATEGORIZATION CRITERIA FOR LOAD VARIATION STUDIES

	≤8 Processors	**>8 Processors**
≤1 Hr	SN	SW
>1 Hr	LN	LW

for CTC and SDSC traces. It can be observed that the improvements obtained by the SS scheme are more pronounced under high load. The trends with respect to different categories under higher loads is similar to that observed under the normal load. It provides significant benefit to the short jobs without affecting the performance of long jobs. The IS scheme is better than the SS scheme only for the SN jobs in terms of average turnaround time, whereas it is better than SS for both SN and SW jobs in terms of average slowdown. It implies that the IS scheme improves the performance of only the relatively shorter jobs in SW category by adversely affecting the performance of the relatively longer jobs. Also, the performance of the IS scheme is much worse for the long jobs, a very undesirable

TABLE VII

JOB DISTRIBUTION BY CATEGORY FOR LOAD VARIATION STUDIES - CTC TRACE

	≤8 Processors	**>8 Processors**
≤1 Hr	44%	13%
>1 Hr	30%	13%

TABLE VIII

JOB DISTRIBUTION BY CATEGORY FOR LOAD VARIATION STUDIES - SDSC TRACE

	≤8 Processors	**>8 Processors**
≤1 Hr	47%	22%
>1 Hr	21%	10%

situation.

Figures 41 and 42 compare respectively the average slowdowns and the average turnaround times of the jobs in the CTC trace against the overall system utilization for various schemes. Figures 43 and 44 compare respectively the average slowdowns and the average turnaround times of the jobs in the SDSC trace against the overall system utilization for various schemes. The SS scheme clearly is much better than both the IS and NS schemes. Even when the system is highly utilized, the SS scheme is able to provide much better response times for all categories of jobs. The IS scheme is not able to achieve high system utilization.

VII. Conclusions

In this paper, we have explored the issue of preemptive scheduling of parallel jobs, using job traces from different supercomputer centers. We have proposed a tunable, selective suspension scheme and demonstrated that it provides significant improvement in the average and the worst-case slowdown of most job categories. It was also shown to provide better slowdown for most job categories over a previously proposed Immediate Service scheme. We also modeled the effect of overheads for job suspension, showing that even under stringent assumptions about available bandwidth to disk, the proposed scheme provides significant benefits over nonpreemptive scheduling and the Immediate Service strategy. We also evaluated the proposed schemes in the presence of inaccurate estimate of job run times and showed that the proposed scheme produced good results. Further, we showed that the Selective Suspension strategy provides greater benefits under high system loads compared to the other schemes.

Acknowledgments

We thank the anonymous referees for their helpful suggestions on improving the presentation of the paper.

This work was supported in part by Sandia National Laboratories, the University of Chicago under NSF Grant #SCI-0414407, and the U.S. Department of Energy under Contract W-31-109-ENG-38.

References

[1] B. DasGupta and M. A. Palis, “Online real-time preemptive scheduling of jobs with deadlines,” in *APPROX*, 2000, pp. 96–107. [Online]. Available: citeseer.nj.nec.com/dasgupta00online.html

[2] X. Deng and P. Dymond, “On multiprocessor system scheduling,” in *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*. ACM Press, 1996, pp. 82–88.

[3] X. Deng, N. Gu, T. Brecht, and K. Lu, “Preemptive scheduling of parallel jobs on multiprocessors,” in *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1996. [Online]. Available: citeseer.nj.nec.com/deng00preemptive.html

[4] L. Epstein, “Optimal preemptive scheduling on uniform processors with non-decreasing speed ratios,” *Lecture Notes in Computer Science*, vol. 2010, pp. 230–248, 2001. [Online]. Available: citeseer.nj.nec.com/epstein00optimal.html

[5] U. Schwiegelshohn and R. Yahyapour, “Fairness in parallel job scheduling,” *Journal of Scheduling*, vol. 3, no. 5, pp. 297–320, 2000. [Online]. Available: citeseer.ist.psu.edu/schwiegelshohn00fairness.html

[6] S. V. Anastasiadis and K. C. Sevcik, “Parallel application scheduling on networks of workstations,” *Journal of Parallel and Distributed Computing*, vol. 43, no. 2, pp. 109–124, 1997. [Online]. Available: citeseer.nj.nec.com/article/anastasiadis96parallel.html

[7] W. Cirne and F. Berman, “Adaptive selection of partition size for supercomputer requests,” in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2000, pp. 187–208. [Online]. Available: citeseer.nj.nec.com/479768.html

[8] D. G. Feitelson, “Analyzing the root causes of performance evaluation results,” Leibniz Center, Hebrew University, Tech. Rep., 2002.

[9] J. P. Jones and B. Nitzberg, “Scheduling for parallel supercomputing: A historical perspective of achievable utilization,” in *Workshop on Job Scheduling Strategies for Parallel Processing*, 1999, pp. 1–16. [Online]. Available: citeseer.nj.nec.com/patton99scheduling.html

[10] W. A. W. Jr., C. L. Mahood, and J. E. West, “Scheduling jobs on parallel systems using a relaxed backfill strategy,” in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2002.

[11] B. G. Lawson and E. Smirni, “Multiple-queue backfilling scheduling with priorities and reservations for parallel systems,” in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2002.

[12] B. G. Lawson, E. Smirni, and D. Puiu, “Self-adapting backfilling scheduling for parallel systems,” in *Proceedings of the International Conference on Parallel Processing*, 2002.

[13] D. Lifka, “The ANL/IBM SP scheduling system,” in *Workshop on Job Scheduling Strategies for Parallel Processing*, 1995, pp. 295–303.

[14] A. W. Mu’alem and D. G. Feitelson, “Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529–543, 2001.

[15] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan, “Scheduling of parallel jobs in a heterogeneous multi-site environment,” in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2003.

[16] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, “Selective reservation strategies for backfill job scheduling,” in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2002.

[17] S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, and P. Sadayappan, “Effective selection of partition sizes for moldable scheduling of parallel jobs,” in *Proceedings of the 9th International Conference on High Performance Computing*, 2002.

[18] V. Subramani, R. Kettimuthu, S. Srinivasan, J. Johnston, and P. Sadayappan, “Selective buddy allocation for scheduling parallel jobs on clusters,” in *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[19] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan, “Distributed job scheduling on computational grids using multiple simultaneous requests,” in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, 2002, pp. 359–366.

[20] D. Talby and D. G. Feitelson, “Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling,” in *Proceedings of the 13th International Parallel Processing Symposium*, 1999, pp. 513–517. [Online]. Available: citeseer.nj.nec.com/talby99supporting.html

[21] D. Zotkin and P. Keleher, “Job-length estimation and performance in backfilling schedulers,” in *Proceedings of the 8th High Performance Distributed Computing Conference*, 1999, pp. 236–243. [Online]. Available: citeseer.nj.nec.com/196999.html

[22] S. H. Chiang, R. K. Mansharamani, and M. K. Vernon, “Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies,” in *ACM SIGMETRICS*

Conference on Measurement and Modeling of Computer Systems, 1994, pp. 33–44. [Online]. Available: citeseer.nj.nec.com/chiang94use.html

[23] S. H. Chiang and M. K. Vernon, "Production job scheduling for parallel shared memory systems," in *Proceedings of International Parallel and Distributed Processing Symposium*, 2002. [Online]. Available: citeseer.nj.nec.com/196999.html

[24] L. T. Leutenneger and M. K. Vernon, "The performance of multiprogrammed multiprocessor scheduling policies," in *ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, May 1990, pp. 226–236. [Online]. Available: citeseer.nj.nec.com/196999.html

[25] E. W. Parsons and K. C. Sevcik, "Implementing multiprocessor scheduling disciplines," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Springer Verlag, 1997, pp. 166–192, lecture Notes in Computer Science vol. 1291.

[26] K. Aida, "Effect of job size characteristics on job scheduling performance," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2000, pp. 1–17. [Online]. Available: citeseer.nj.nec.com/319169.html

[27] O. Arndt, B. Freisleben, T. Kielmann, and F. Thilo, "A comparative study of online scheduling algorithms for networks of workstations," *Cluster Computing*, vol. 3, no. 2, pp. 95–112, 2000. [Online]. Available: citeseer.nj.nec.com/article/arndt98comparative.html

[28] W. Cirne, "When the herd is smart: The emergent behavior of sa," in *IEEE Transactions on Parallel and Distributed Systems*, 2002. [Online]. Available: citeseer.nj.nec.com/457615.html

[29] D. Perkovic and P. J. Keleher, "Randomization, speculation, and adaptation in batch schedulers," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 2000, p. 7.

[30] P. J. Keleher, D. Zotkin, and D. Perkovic, "Attacking the bottlenecks of backfilling schedulers," *Cluster Computing*, vol. 3, no. 4, pp. 245–254, 2000. [Online]. Available: citeseer.nj.nec.com/467800.html

[31] J. Krallmann, U. Schwiegelshohn, and R. Yahyapour, "On the design and evaluation of job scheduling algorithms," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 1999, pp. 17–42. [Online]. Available: citeseer.nj.nec.com/krallmann99design.html

[32] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Characterization of backfilling strategies for parallel job scheduling," in *Proceedings of the ICPP-2002 Workshops*, 2002, pp. 514–519.

[33] A. Streit, "On job scheduling for HPC-clusters and the dynP scheduler," in *Proceedings of the 8th International Conference on High Performance Computing*. Springer-Verlag, 2001, pp. 58–67.

[34] C. McCann, R. Vaswani, and J. Zahorjan, "A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 11, no. 2, pp. 146–178, 1993.

[35] D. G. Feitelson and M. A. Jette, "Improved utilization and responsiveness with gang scheduling," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1997, pp. 238–261.

[36] D. Jackson, Q. Snell, and M. J. Clement, "Core algorithms of the maui scheduler," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 2001, pp. 87–102. [Online]. Available: citeseer.nj.nec.com/479768.html

[37] J. Skovira, W. Chan, H. Zhou, and D. Lifka, "The easy - loadleveler api project," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 1996, pp. 41–47. [Online]. Available: citeseer.nj.nec.com/479768.html

[38] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1997, pp. 1–34.

[39] K. C. Sevcik, "Application scheduling and processor allocation in multiprogrammed parallel processing systems," *Performance Evaluation*, vol. 19, no. 2-3, pp. 107–140, 1994. [Online]. Available: citeseer.nj.nec.com/sevcik93application.html

[40] J. Zahorjan and C. McCann, "Processor scheduling in shared memory multiprocessors," in *ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, May 1990, pp. 214–225. [Online]. Available: citeseer.nj.nec.com/196999.html

[41] D. G. Feitelson, "Logs of real parallel workloads from production systems," http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

Rajkumar Kettimuthu is a researcher at Argonne National Laboratory's Mathematics and Computer Science Division. His research interests include data transport in high-bandwidth and high-delay networks and scheduling and resource management for cluster computing and the Grid. He has a bachelor of engineering degree in computer science and engineering from Anna Univerisy, Madras, India, and a master of science in computer and information science from the Ohio State University.

Vijay Subramani received his B.E in computer science and engineering from Anna University, India, in 2000 and his M.S in computer and information science from the Ohio State University in 2002. His research at Ohio State included scheduling and resource management for parallel and distributed systems. He currently works at Microsoft Corporation in Redmond, WA. His past work experience includes an internship at Los Alamos National Laboratory, where he worked on buffered coscheduling.

Srividya Srinivasan currently works as a software engineer at Microsoft Corporation in Redmond, WA. She worked as a software developer at Bloomberg L.P in New York earlier. She received a B.E degree in computer science and engineering from Anna University, Chennai, India, in 2000 and her M.S in computer and information science from the Ohio State University in 2002. Her research at Ohio State focused on parallel and distributed systems with an emphasis on parallel job scheduling.

Thiagaraja Gopalsamy is a senior software engineer with Altera Corporation, San Jose. He received his bachelor's degree in computer science and engineering in 1999 from Anna University, India and his master's degree in computer and information science in 2001 from the Ohio State University. His past research interests include mobile ad hoc networks and parallel computing. He is currently working on field programmable gate arrays and reconfigurable computing.

D. K. Panda is a professor of computer science at the Ohio State University. His research interests include parallel computer architecture, high performance networking, and network-based computing. He has published over 150 papers in these areas. His research group is currently collaborating with national laboratories and leading companies on designing various communication and I/O subsystems of next-generation HPC systems and datacenters with modern interconnects. The MVAPICH (MPI over VAPI for InfiniBand) package developed by his research group (http://nowlab.cis.ohio-state.edu/projects/mpi-iba/) is being used by more than 160 organizations worldwide to extract the potential of InfiniBand-based clusters for HPC applications. Dr. Panda is a recipient of the NSF CAREER Award, OSU Lumley Research Award (1997 and 2001), and an Ameritech Faculty Fellow Award. He is a senior member of IEEE Computer Society and a member of ACM.

P. Sadayappan received the B. Tech. degree from the Indian Institute of Technology, Madras, India, and an M.S. and Ph.D. from the State University of New York at Stony Brook, all in electrical engineering. He is currently a professor in the Department of Computer Science and Engineering at the Ohio State University. His research interests include scheduling and resource management for parallel/distributed systems and compiler/runtime support for high-performance computing.