Proceedings of the 22nd International Conference on Distributed Computing Systems, (ICDCS'2002).

Adaptive and Virtual Reconfigurations for Effective Dynamic Job Scheduling in Cluster Systems *

Songqing Chen, Li Xiao, and Xiaodong Zhang
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795
{sqchen, lxiao, zhang}@cs.wm.edu

Abstract

In a cluster system with dynamic load sharing support, a job submission or migration to a workstation is determined by the availability of CPU and memory resources of the workstation at the time [3]. In such a system, a small number of running jobs with unexpectedly large memory allocation requirements may significantly increase the queuing delay times of the rest of jobs with normal memory requirements, slowing down executions of individual jobs and decreasing the system throughput. We call this phenomenon as the *job blocking* problem because the big jobs block the execution pace of majority jobs in the cluster. Since the memory demand of jobs may not be known in advance and may change dynamically, the possibility of unsuitable job submissions/migrations to cause the blocking problem is high, and the existing load sharing schemes are unable to effectively handle this problem. We propose a software method incorporating with dynamic load sharing, which adaptively reserves a small set of workstations through virtual cluster reconfiguration to provide special services to the jobs demanding large memory allocations. This policy implies the principle of shortest-remaining-processing-time policy. As soon as the blocking problem is resolved by the reconfiguration, the system will adaptively switch back to the normal load sharing state. We present three contributions in this study. (1) we quantitatively present the conditions to cause the job blocking problem. (2) We present the adaptive software method in a dynamic load sharing system. We show the adaptive process causes little additional overhead. (3) Conducting trace-driven simulations, we show that our method can effectively improve the cluster computing performance by quickly resolving the job blocking problem. The effectiveness and performance insights are also analytically verified.

1 Introduction

Load sharing provides a system mechanism to dynamically migrate jobs from heavily loaded workstations to lightly loaded workstations, aiming at fully utilizing system resources. Following the load sharing principle, researchers have designed different alternatives by balancing the number of jobs/tasks among the workstations (see e.g. [5], [11], [14]), by considering memory allocation sizes of jobs (see e.g. [1], [2]), and by considering both CPU and memory resources (see e.g. [12], [13]). Recently, we have developed dynamic load sharing schemes to schedule or migrate jobs without the knowledge of their memory allocation sizes before jobs start running [3]. All above cited schemes are designed for job scheduling and migrations among the workstations (inter-workstation scheduling). The job scheduling in multiprogramming environment of each conventional workstation (intra-workstation scheduling) is normally conducted in a round-robin fashion to fairly share processor cycles.

It has been proved that the optimal inter-workstation scheduling policy is to always schedule the job with the shortest remaining processing time [8]. This policy minimizes the mean response time of the submitted jobs. In practice, the optimal scheduling policy is impossible to be implemented for two reasons. First, the remaining processing time of each job is unknown to the scheduler, which is the major reason. Second, jobs with long remaining times, may be unfairly treated with unreasonably long delays. We believe that the round-robin scheduling is practically effective unless the job sizes are known in advance or accurately predictable. One of our focuses is to study its effects to inter-workstation scheduling.

In a cluster system with dynamic load sharing support, a new job can be submitted to a workstation or a running job can be migrated to the workstation under following conditions. When the workstation has idle memory space, the job can be accepted if the number of running jobs in the workstation is still less than a predetermined threshold which is the maximum number of job slots a CPU is willing to take (also called the CPU Threshold). When the workstation does not have idle memory space, or is even oversized, no jobs will be accepted without further checking the status of the CPU threshold. This strategy has been shown its effectiveness in load sharing, particularly to schedule jobs with unknown memory allocation sizes [3]. In such a system, a small number of running jobs with large memory allocation requirements can

*This work is supported in part by the National Science Foundation under grants CCR-9812187, EIA-9977030, and CCR-0098055, and by a Usenix Association Research Fellowship.

be scattered among workstations to quickly use up the memory space, causing slow job submissions to these workstations. Since these large jobs normally have long remaining processing times, eventually, all the workstations may become heavily loaded, stopping job submissions and migrations. We call this phenomenon as the *job blocking* problem, which is rooted from unsuitable placements of these large jobs. The existence of these large jobs in a few workstations may increase the queuing delay times of the rest of jobs with relatively small memory requirements, slowing down executions of individual jobs and decreasing the cluster system's throughput. Since job sizes including the memory allocations are unknown in advance, the possibility of unsuitable job placements to cause the blocking problem is high, and existing load sharing schemes are unable to effectively handle this problem.

We have developed a framework of a *dynamic load sharing system* with three major functions. First, since the scheduler in each node does not have the knowledge of the size of demanded memory and its changing range of each job in its lifetime, the scheduler dynamically monitors the amount of page faults, memory demands of jobs, and considers the available physical memory space to make scheduling decisions accordingly. Second, a memory threshold is set to ensure that memory demands of jobs are not oversized or only oversized to a certain degree. The CPU threshold is used to balance the number of jobs in the cluster, and to set a reasonable queuing delay time for jobs in each workstation. Third, whenever a certain amount of page faults due to memory shortage or exceeding a memory threshold are detected in a workstation, or whenever the number of running jobs reaches the CPU threshold, new job submissions to this workstation will be blocked and will be remotely submitted to other lightly loaded workstations with available memory space or/and with additional job slots if possible. Meanwhile, one or more jobs already executing in the workstation that becomes overloaded may also be migrated to other lightly loaded workstations if possible. For detailed descriptions of dynamic load sharing with considerations of both CPU and memory utilizations, the interested readers may refer to [3].

When both job submissions and migrations are blocked in a cluster, it implies that the resource allocation in each workstation either reaches its memory threshold due to arrivals of some jobs with large memory demands, or reaches its CPU threshold, or both. Further job submissions or migrations will cause more page faults or queuing delay in a destination workstation. One simple solution would be to temporarily suspend the large jobs so that the job submissions will not be blocked. However, this approach will not be fair to the large jobs that may starve if job submissions continue to flow, or that can be executed only when the cluster becomes lightly loaded.

We have observed that CPU and memory resources are actually not fully utilized during the period of blocking. For example, some workstations reaching their CPU thresholds may still have idle memory space, while some workstations experiencing page faults may still have additional job slots available. Our recent experiments show that when a cluster system is not able to further accept or migrate jobs, there are still large accumulated idle memory space volumes available among the workstations. This is because demanded memory allocations of a handful jobs could not fit in any single workstation with other running jobs. We have also found that jobs are not evenly distributed among workstations, which increases the total job queuing time. Unfortunately, the dynamic load sharing scheme or job suspension described above is not able to efficiently resolve this blocking problem by further utilizing the potential available resources. We target to address these problems of inefficient resource allocations. Our observations and experimental results have motivated us to consider scheduling jobs by adaptively and virtually reconfiguring a cluster system to further utilize resources and to quickly resolve the blocking problem.

To make a system more resilient against the blocking problem, it is preferable to incorporate a dynamic protection mechanism with the load sharing system, rather than waiting for slow self-recovering or using a brute-force approach, such as job suspensions. In order to make our solution effective, we need to address two potential concerns. First, we need to dynamically identify large jobs, and to find suitable workstations for them to execute on. Second, the policy should be beneficial to both large and other jobs. We propose a software method incorporated with dynamic load sharing, which adaptively reserves a small set of workstations (called reserved workstations) through a virtual cluster reconfiguration to provide special services to the jobs demanding large memory space. As soon as the blocking problem is resolved by the reconfiguration, the system will adaptively switch back to the normal load sharing state. On one hand, this method can improve the utilization of CPU and memory resources of non-reserved workstations, because jobs with normal sizes can be smoothly executed without the interference of large jobs. On the other hand, large jobs are treated fairly because they are served by reserved workstations.

We present three contributions in this study. (1) We present the conditions to cause the job blocking problem. (2) We present the adaptive software method in a dynamic load sharing system. We show the adaptive process causes little additional overhead. (3) Conducting trace-driven simulations, we show that our method can effectively improve the cluster computing performance by quickly resolving the job blocking problem. The effectiveness and performance insights are also analytically verified.

2 An Adaptively and Virtually Reconfigured Cluster System

2.1 Job reallocations

Here is the basic idea of our software method for adaptive and virtual cluster reconfigurations, which can be easily incorporated with the dynamic load sharing scheme. The blocking problem is initially detected when a workstation experiences a certain amount of page faults, but the scheduler could not find a qualified destination to migrate jobs from this workstation. If the accumulated idle memory space size in the cluster is larger than the average user memory space of workstations in the cluster, the reconfiguration routine is activated. The routine first identifies the most lightly loaded workstation with largest idle memory space, and continues to block job submissions and migrations to this workstation. The time period between identifying the workstation and completions of the running jobs in the workstation is called the *re-*

serving period. (One alternative is to end the reserving period as soon as the available memory space in the reserved workstation is sufficiently large for a job migration with large memory demand). During the reserving period, if the blocking problem disappears, the system will be back to the normal load sharing state. If the blocking problem still exists after the reserving period, the reconfiguration routine will migrate a job with the largest memory demand suffering serious page faults to the reserved workstation. When the blocking problem is detected again in a workstation, the reconfiguration routine will first try to migrate a job to a reserved workstation if it exists, that is able to provide sufficient memory space and job slots. Otherwise, the reconfiguration routine will start another reserving period to identify an appropriate workstation. As soon as the blocking problem is resolved by the virtual reconfiguration, the system will be adaptively switched back to the normal load sharing state. The transition between the cluster reconfiguration for reserved computing and normal load sharing is quite natural. As a reserved workstation completes its special service, the scheduler will view it as a regular workstation and resume normal job submissions to the workstation. Notice that the processes of starting and releasing a reconfiguration are not only adaptive and virtual, but they cause little additional overhead as well.

The framework of the reconfiguration routine is embedded in the dynamic load sharing system in a workstation as follows:

While the load sharing system is on
  if job submissions or/and migrations are allowed
    $general\_dynamic\_load\_sharing()$;
  else start reconfiguration by {
    if ($\exists\ reservation\_flag(reserved\_ID) == 1$)
    && (the workstation has enough available resources)
      $node\_ID = reserved\_ID$;
    else {
      $node\_ID = reserve\_a\_workstation()$;
      $reservation\_flag(node\_ID) = 1$;
      }
    $job\_ID = find\_most\_memory\_intensive\_job()$;
    $migrate\_job(job\_ID, node\_ID)$;
    }

The functions in the framework are defined as follows:

- $general\_dynamic\_load\_sharing()$: conducts regular operations of dynamic load sharing including monitoring, local/remote job submissions, and job migrations [3].
- $reserve\_a\_workstation()$: identifies the lightly loaded workstation in the cluster, continues to block job submissions to the workstation until execution completions of all running jobs in the workstation, and returns its $node\_ID$ and sets the $reservation\_flag$.
- $find\_most\_memory\_intensive\_job()$: identifies the job with the largest memory demand, and returns its job_ID.
- $migrate\_job(job\_ID, node\_ID)$: migrates the identified job to the reserved workstation.

The $reservation\_flag$ is turned off when the reserved workstation completes executions of all the migrated jobs, which resumes the normal job submissions and migrations to the workstation.

2.2 The rationale of our solution

The potential performance gain of our approach comes from four sources. First, *the idle memory space is available among workstations*. Unfortunately, the available space in each individual workstation is not large enough to serve any incoming jobs. A considerable amount of accumulated idle memory space in the cluster can be utilized by job reallocations. Reserving a workstation plays an equivalent role to moving some accumulated idle memory space to the reserved workstation, so that the workstation is able to serve large jobs that could not fit in any individual idle memory space before the reservation.

Second, *the identified large job is likely to be a large job with long lifetime*. The job is identified after the reserving period. If a job is observed to demand a large memory space, causing page faults for a period of time, this job will be likely to continue to stay and execute for a longer time than other jobs in the workstation for two reasons: (1) experiments have shown that a job with a large memory demand in process interactions is less competitive than jobs with small memory allocations in conventional operating systems, such as Unix and Linux [6]; (2) experiments have also shown that a job having stayed for a relatively long time is predicted to continue to stay for a even longer time than other jobs [5]. After a large job is migrated away, the rest of jobs will be served quickly, and submissions to the workstation will continue to flow. The principle of the shortest remaining processing time policy [8] is implicitly applied here.

Third, *the concern of unfairly treating large jobs does not exist*. We specially reserve workstations to process these large jobs that are less competitive than jobs with small memory allocations.

Finally, *in practice, the percentage of large jobs in a job pool is low*. If there are too many large jobs, the proposed method will reserve too many workstations so that normal jobs can not run. This causes unfairness to normal jobs. This concern could be easily addressed because several studies (e.g. [5, 9]) have shown that the percentage of exceptionally large jobs is very low in real-world workloads.

2.3 What is the virtual reconfiguration not able to do?

When the accumulated idle memory space in the cluster is not sufficiently large, the virtual reconfiguration will not be effective. If the accumulated idle memory space is smaller than the user space of a single workstation, it will be difficult to reserve a workstation providing its entire memory space. Under such a condition, we believe that the cluster memory resources have been sufficiently utilized. Examining the accumulated idle memory space in the cluster is one way to detect this condition. If a workstation can not be reserved within a pre-determined time interval, it implies that the cluster is truly heavily loaded.

In a heterogeneous cluster system, a reserved workstation will be the one with relatively large physical memory space. If the user space in the reserved workstation is still not sufficiently large for a migrated job, this implies that this job may not be suitable in this cluster unless the network RAM technique is applied [12]. If this job has to be executed in the cluster, the virtual reconfiguration method will provide a reserved workstation for dedicated

service, where its page faults will not affect performance of other jobs. The virtual reconfiguration may not work well for specific workloads where big jobs are dominant. Again this case can be rare in practice.

3 Experimental Environment

3.1 Tracing job execution at the kernel level

The lifetime of a job has been used as an important factor in load sharing designs. Which process to be migrated in existing scheduling policies (see e.g. [5]) is considered by predicting the lifetime of CPU intensive jobs. Detailed breakdowns of the lifetime will provide more insightful considerations for load sharing decisions.

We have developed facilities by kernel instrumentation [3], which measures different portions of the lifetime for a job execution. Particularly, the instrumentation records when a job process is interrupted for a system event, and how long this event lasts. A trace buffer is initially allocated when the system is booted to collect the system traces. The facilities also dynamically measure (1) current ages and lifetime of jobs, (2) the sizes of memory allocation for each running job, and idle memory space in each workstation, (3) events of page faults in each workstation, (4) the read/write operations of each running job, and (5) the status of I/O buffer cache in each workstation.

3.2 Application workloads

We have selected two groups of workloads. The first group (workload group 1) consists of 6 SPEC-2000 benchmark programs: *apsi*, *gcc*, *gzip*, *mcf*, *vortex*, and *bzip*, which are both CPU and memory intensive. Using the facilities described above, we first run each program in a dedicated environment to observe the memory access behavior without major page faults (except for cold misses) and page replacement (the demanded memory space is smaller than the available user space). We selected a 400 MHz Pentium II with 384 Mbyte physical memory and a swap space of 380 MBytes to run workload group 1. The operating system is Redhat Linux release 6.1 with the kernel 2.2.14. Table 1 presents the basic experimental results of the 6 SPEC-2000 programs, where the "description" gives the application nature of each program, the "input file" is the input file names from SPEC200 benchmarks, the "working set" gives the maximum size of the allocated memory space during the execution, the "lifetime" is the total execution time of each program.

The second group (workload group 2) consists of seven large scientific and system programs that are representative CPU-intensive, memory-intensive, and/or I/O-active jobs: *bit-reversals* (bit-r), *merge-sort* (m-sort), *matrix multiplication* (m-m), *a trace-driven simulation* (t-sim), *partitioning meshes* (metis), *cell-projection volume rendering for a sphere* (r-sphere), and *cell-projection volume rendering for flow of an aircraft wing* (r-wing). The descriptions and related citations of these applications can be found in [3].

We first measured the execution performance of each program and monitored their memory performance related activities in a dedicated computing environment of a 233 MHz Pentium PC with 128 MByte main memory and a swap space of 128MB, running Linux version 2.0.38. The program memory demands in this group are smaller than the ones in workload group 1. So, the workstations we selected here are less powerful than the workstations used to run programs of workload group 1. Table 2 presents the experimental results of all the seven programs, where the "data size" is the number of entries of the input data, the "working set" gives a range of the memory space demand during the execution, the "lifetime" is the total execution time of each program.

Programs	data size	working set (MB)	lifetime (s)
bit-r	2^{23}	64.22	192.26
m-sort	2^{23}	64.27	82.76
m-m	$1,700^2$	66.37	4902.29
t-sim	31,061	4.64	41.63
metis	1M-4M	1.37-4.30	124.41
r-sphere	150,000	36.84 — 39.66	318.64
r-wing	500,000	19.53 — 23.39	72.78

Table 2: Execution performance and memory related data of the seven application programs.

3.3 Trace-driven simulations

Performance evaluation of the virtual reconfiguration is simulation-based, consisting of two major components: a simulated cluster and application workload traces. We will discuss our simulation model, system conditions and the workload traces in this section.

3.3.1 Two simulated clusters

We have simulated two homogeneous clusters, each of which has 32 workstations whose types are consistent with the two groups of experiments presented in the previous section. Application programs in workload group 1 are run in cluster 1, where the CPU speed is 400 MHz, memory size is 384 MBytes, and the swap space is 380 MBytes. While application programs in workload group 2 are run in cluster 2, where the CPU speed is 233 MHz, memory size is 128 MBytes, and the swap space is 128 MBytes. In simulations of both clusters, the memory page size is 4 KBytes, page fault service time is 10 ms, and the context switch time is 0.1 ms. Ethernet connection speed, B, is 10 Mbps. The remote submission/execution cost, r, is 0.1 second for 10 Mbps network. The preemptive migration cost is estimated by assuming the entire memory image of the working set will be transferred from a source to a destination node for a job migration, which is $r + \frac{D}{B}$, where r is a fixed remote execution cost in second, and D is the amount of data in bits to be transferred in the job migration. Each workstation maintains a global load index file which contains CPU, memory, and I/O load status information of other computing nodes. The load sharing system periodically collects and distributes the load information among the workstations.

Programs	description	input file	working set (MB)	lifetime (s)
apsi	climate modeling	apsi.in	196.0	2,619.0
gcc	optimized C compiler	166.i	145.0	228.0
gzip	data compression	input.graphic	195.0	249.0
mcf	combinatorial optimization	inp.in	80.0	969.0
vortex	database	lendian1.raw	115.0	345.0
bzip	data compression	input.graphic	200.0	403.0

Table 1: Execution performance and memory related data of the 6 SPEC 2000 benchmark programs.

3.3.2 Workload traces

The two groups of workload traces are collected by using our facilities to monitor the execution of the 6 SPEC 2000 benchmark programs (workload group 1) and the 7 application programs (workload group 2) at different submission rates on a Linux workstation. Job submission rates are generated by a lognormal function:

$$R_{ln}(t) = \begin{cases} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(\ln t-\mu)^2}{2\mu^2}} & t > 0 \\ 0 & t \leq 0 \end{cases} \quad (1)$$

where $R_{ln}(t)$ is the lognormal arrival rate function, t is the time duration for job submissions in a unit of seconds, the values of μ and σ adjust the degrees of the submission rate. The lognormal job submission rate has been observed in several practical studies (see e.g. [4], [10]). Five traces for each group are collected in each workload group (1 and 2) with five different arrival rates:

- *Trace-1* (light job submissions): $\sigma = 4.0$, $\mu = 4.0$, and 359 jobs submitted in 3,586 seconds.
- *Trace-2* (moderate job submissions): $\sigma = 3.7$, $\mu = 3.7$, and 448 jobs submitted in 3,589 seconds.
- *Trace-3* (normal job submissions): $\sigma = 3.0$, $\mu = 3.0$, and 578 jobs submitted in 3,581 seconds.
- *Trace-4* (moderately intensive job submissions): $\sigma = 2.0$, $\mu = 2.0$, and 684 jobs submitted in 3,585 seconds.
- *Trace-5* (highly intensive job submissions): $\sigma = 1.5$, $\mu = 1.5$, and 777 jobs submitted in 3,582 seconds.

The jobs in each trace were randomly submitted to 32 workstations. Each job has a header item recording the submission time, the job ID, and its lifetime measured in the dedicated environment. Following the header item, the execution activities of the job are recorded in a time interval of every 10 ms including CPU cycles, the memory demand/allocation, buffer cache allocation, number of I/Os, and others. Thus, dynamic memory and I/O activities can be closely monitored. During job interactions, page faults are generated accordingly by an experiment-based model presented in [3].

The 5 traces for workload group 1 are represented by *SPEC-Trace-1*, *SPEC-Trace-2*, *SPEC-Trace-3*, *SPEC-Trace-4*, and *SPEC-Trace-5*; and the 5 traces for workload group 2 are represented by *App-Trace-1*, *App-Trace-2*, *App-Trace-3*, *App-Trace-4*, and *App-Trace-5*.

4 Performance Evaluation

The *slowdown* of a job is the ratio between its wall-clock execution time and its CPU execution time. A major performance metric we have used is the average slowdown of all jobs in a trace. Major contributions to slowdown come from queuing time waiting for CPU service, the delays of page faults, and the overhead of migrations and remote submission/execution. The average slowdown measurement can determine the overall performance of a load sharing policy, but may not be sufficient to provide performance insights. For a given workload trace, we have also measured the total execution time and its breakdowns.

Conducting the trace-driven-simulations on a 32 node cluster, we have evaluated the performance of dynamic load sharing supported by the adaptive and virtual reconfiguration method by comparing its slowdowns and execution times of several application workloads with dynamic load sharing without such a support.

4.1 Improving memory utilization

Figure 1 presents the total execution times (left figure) and the queuing times (right figure) during the executions on a 32-node cluster, where the jobs in workload group 1 are scheduled by either the dynamic load sharing scheme (G-Loadsharing) [3] or the dynamic load sharing supported by the virtual reconfiguration method (V-Reconfiguration). The trace-driven simulation results show the virtual reconfiguration method significantly reduced the total execution times and the queuing times, and is particularly effective to the workloads with higher job arrival rates. For example, applying virtual reconfigurations, we were able to reduce the execution times and the queuing times by 29.3% and 24.8%, respectively for workload SPEC-Trace-1, by 32.4% and 35.8%, respectively for workload SPEC-Trace-2, by 32.4% and 36.7%, respectively for workload SPEC-Trace-3, by 30.3% and 34.0%, respectively for workload SPEC-Trace-4, and 27.4% and 38.2%, respectively for workload SPEC-Trace-5.

The reduction of the total execution times caused mainly by the reduction of the queuing times effectively reduces the average slowdowns for each trace in workload group 1. The left figure in Figure 2 presents the comparative average slowdowns of job executions using G-Loadsharing and V-Reconfiguration. The virtual reconfiguration method reduced the average slowdowns by 23.4%, 27.7%, 22.6%, 24.6% and 28.46% for workloads SPEC-Trace-1, SPEC-Trace-2, SPEC-Trace-3, SPEC-Trace-4, and SPEC-Trace-5, respectively.

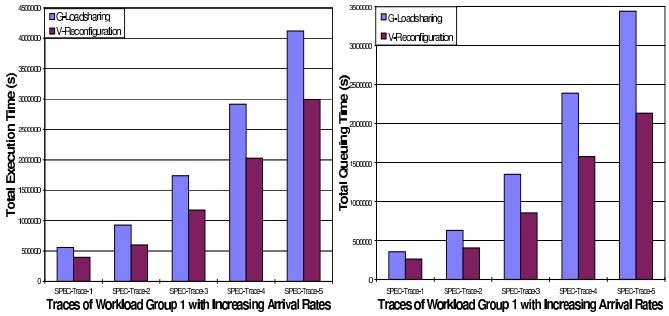

Figure 1: The total execution times (left figure) and queuing times (right figure) of the 5 traces of workload group 1 on a 32 workstation cluster scheduled by the dynamic load sharing scheme (G-Loadsharing) and virtual reconfiguration based dynamic load sharing (V-Reconfiguration).

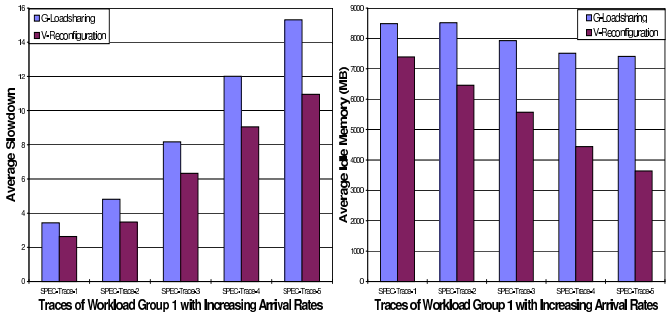

Figure 2: The average slowdowns (left figure) and the average idle memory volumes (right figure) for executing the 5 traces of workload group 1 on a 32 workstation cluster scheduled by the dynamic load sharing scheme (G-Loadsharing) and virtual reconfiguration based dynamic load sharing (V-Reconfiguration).

We have also observed the average total idle memory volumes during the lifetime of job executions in each workload trace. We collect the total idle memory volume in the cluster every second to calculate the average amount of idle memory space during the entire lifetime. We have repeated the measurements by using several other time intervals, such as 10 seconds, 30 seconds, and 1 minute, and obtained almost identical average values. This implies that the average total idle memory volume is not sensitive to different measurement time intervals. The right figure in Figure 2 presents the comparative average idle memory volumes during lifetimes of 5 workload traces using G-Loadsharing and V-Reconfiguration. The virtual reconfiguration method reduced the average idle memory volumes by 12.9%, 24.2%, 29.7%, 40.9%, and 50.8% for workloads SPEC-Trace-1, SPEC-Trace-2, SPEC-Trace-3, SPEC-Trace-4, and SPEC-Trace-5, respectively. This group of results confirms that the virtual reconfiguration can further utilize idle memory space so that the cluster is able to accept (migrate and submit) more jobs and to speed up the job flow in clusters. This is the main reason to achieve significant reductions of average slowdowns in this workload.

4.2 Improving job balancing for high CPU utilizations

Figure 3 presents the total execution times (left figure) and the queuing times (right figure) during the executions on a 32-node cluster, where the jobs in workload group 2 are scheduled by either the dynamic load sharing scheme or the dynamic load sharing supported by the virtual reconfiguration method. The trace-driven simulation results show that the virtual reconfiguration method reduced the total execution times and the queuing times, and is particularly effective to the workloads of App-Trace-2 and App-Trace-3. For example, applying virtual reconfigurations, we were able to reduce the the execution time and the queuing time by 13.4% and 16.3%, respectively, for workload App-Trace-2, and by 14.0% and 16.8%, respectively for workload App-Trace-3. The reductions to other three traces are modest.

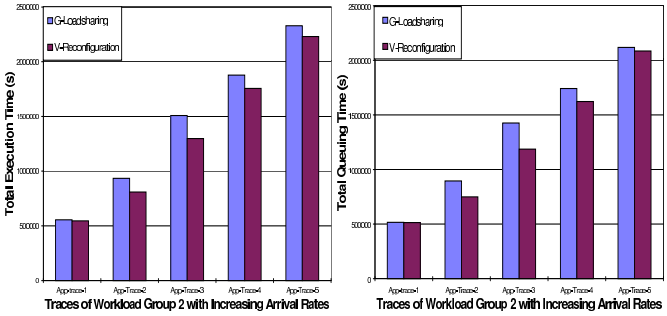

Figure 3: The total execution times (left figure) and queuing times (right figure) of the 5 traces of workload group 2 on a 32 workstation cluster scheduled by the dynamic load sharing scheme (G-Loadsharing) and virtual reconfiguration based dynamic load sharing (V-Reconfiguration).

The reduction of the total execution times caused mainly by the reduction of the queuing times reduces the average slowdowns for each trace in workload group 2. The left figure in Figure 4 presents the comparative average slowdowns of job executions using G-Loadsharing and V-Reconfiguration. The virtual reconfiguration method effectively reduced the average slowdowns by 16.3%, 16.8%, and 6.8% for workloads App-Trace-2, App-Trace-3, and App-Trace-4, respectively. The average slowdown reductions for workloads App-Trace-1 and App-Trace-5 are modest. Our experiments show that the performance gains mainly come from job balancing from the virtual reconfiguration. In fact, the total idle memory volumes are almost the same as those before virtual reconfigurations. We collect the number of active jobs in each workstation every second to calculate the standard deviation of the number of active jobs among all non-reserved workstations at this moment. This standard deviation gives the job balance skew in each workstation. We further calculate the average job balance skew during the entire lifetime among all non-reserved workstations by first summing all the individual skews and then dividing the sum by the total number of time units. We

have repeated the measurements by using several other time intervals, such as 10 seconds, 30 seconds, and 1 minute, and obtained almost identical average values. This implies that the average job balance skew is not sensitive to different measurement time intervals. The right figure in Figure 4 presents the comparative average job balance skew during lifetimes of 5 workload executions (workload group 2) using G-Loadsharing and V-Reconfiguration. The virtual reconfiguration method reduced the average job balance skew by 10.3%, 16.5%, and 6.3%, for workloads App-Trace-2, App-Trace-3, and App-Trace-4, respectively. The two job balance skew differences for traces App-Trace-1 and App-Trace-5 are small. This is a reason why little performance gains are achieved by virtual reconfiguration for these two traces. This group of results indicates that the virtual reconfiguration is able to evenly distributed jobs among non-reserved workstations to reduce total queuing time. We have showed that overall performance gains (reductions of total execution times and slowdowns) from improving job balancing is not as significant as that from improving memory utilizations by virtual reconfiguration.

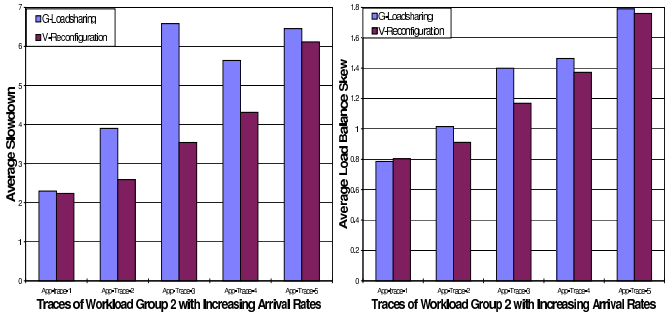

Figure 4: The average slowdowns (left figure) and average job balance skews (right figure) for executing the 5 traces of workload group 1 on a 32 workstation cluster scheduled by the dynamic load sharing scheme (G-Loadsharing) and virtual reconfiguration based dynamic load sharing (V-Reconfiguration).

5 Performance Modeling

The effectiveness and performance insights of the proposed method are analytically verified in this section. The execution time of job i in a workload for $i = 1, ..., n$, $t_{exe}(i)$, is expressed as

$$t_{exe}(i) = t_{cpu}(i) + t_{page}(i) + t_{que}(i) + t_{mig}(i),$$

where $t_{cpu}(i)$, $t_{page}(i)$, $t_{que}(i)$, and $t_{mig}(i)$ are the CPU service time, the paging time for page faults, the queuing time waiting in a job queue, and the migration time if the job is migrated during its execution.

The total execution time of a workload with n jobs, T_{exe}, is expressed as the sum of the total CPU service time (T_{cpu}), total paging time (T_{page}), the total queuing time (T_{que}), and the total migration time (T_{mig}):

$$\begin{aligned} T_{exe} &= \sum_{i=1}^{n} t_{exe}(i) \\ &= \sum_{i=1}^{n} t_{cpu}(i) + \sum_{i=1}^{n} t_{page}(i) + \sum_{i=1}^{n} t_{que}(i) + \sum_{i=1}^{n} t_{mig}(i) \\ &= T_{cpu} + T_{page} + T_{que} + T_{mig}. \end{aligned}$$

For a given workload with n jobs running on a cluster, we compare the total execution time of the workload without virtual reconfiguration, T_{exe}, and the same quantity with virtual reconfiguration to resolve the blocking problem, denoted as $\hat{T}_{exe} = \hat{T}_{cpu} + \hat{T}_{page} + \hat{T}_{que} + \hat{T}_{mig}$. The comparison consists of the following 4 separate models.

1. *CPU service time*. The jobs demand identical CPU services on both cluster environment, so that $T_{cpu} = \hat{T}_{cpu}$.
2. *Paging time*. There will be three possible results: $T_{page} > \hat{T}_{page}$, $T_{page} = \hat{T}_{page}$, and $T_{page} < \hat{T}_{page}$. Paging time reduction ($T_{page} > \hat{T}_{page}$) is the objective of the virtual reconfiguration, which can be achieved by migrating jobs with large memory demands to reserved workstations.
3. *Queuing time*. In a cluster with virtual reconfiguration, the queuing time consists of two parts:

$$\hat{T}_{que} = \hat{T}_{n-que} + \sum_{k=1}^{m} g(Q_r(k)),$$

where $\hat{T}_{n-que}$ is the queuing time in non-reserved workstations, g is a FIFO queuing function mainly determined by the CPU service time of jobs in the queue of a reserved workstation, and m is the number of reserved workstations, and $Q_r(k)$ is the number of jobs in reserved workstation k. The queuing time in reserved workstation k satisfies

$$g(Q_r(k)) \leq \sum_{j=1}^{Q_r(k)} (Q_r(k) - j) w_{kj},$$

where the arrival order of jobs to workstation k is in an increasing order, i.e. job 1 is the first job arrived in workstation k, and job $Q_r(k)$ is the last arrived job. Variable w_{kj} is the waiting time of job $j+1$ for job j to complete in workstation k. In other words, w_{kj} is the time interval between the arrival time of job $j+1$ and the completion time of job j.
4. *Migration time*. There will be again three possible results: $T_{mig} > \hat{T}_{mig}$, $T_{mig} = \hat{T}_{mig}$, and $T_{mig} < \hat{T}_{mig}$. The migration time is workload and network speed dependent. As high speed networks become widely used in clusters, the migration time in load sharing is only a small portion in the execution time, becoming less crucial for load sharing performance. When $T_{mig} < \hat{T}_{mig}$, the virtual reconfiguration needs to sufficiently reduce queuing time to trade off the increase in migration time. Our experiments show that $T_{mig} \approx \hat{T}_{mig}$ because the number of large jobs is very small in job pools.

Using the four portions in the total execution time, considering the paging time reduction ($T_{page} > \hat{T}_{page}$), and assuming the

difference between T_{mig} and $\hat{T}_{mig}$ is insignificant in load sharing performance, we examine the potential execution time reduction from the virtual reconfiguration:

$$\begin{aligned} T_e - \hat{T}_e &= (T_{cpu} + T_{page} + T_{que} + T_{mig}) \\ &\quad -(\hat{T}_{cpu} + \hat{T}_{page} + \hat{T}_{que} + \hat{T}_{mig}) \\ &\approx (T_{page} - \hat{T}_{page}) + (T_{que} - \hat{T}_{que}) \\ &\geq (T_{page} - \hat{T}_{page}) + (T_{que} - \hat{T}_{n-que} \\ &\quad - \sum_{k=1}^{m} \sum_{j=1}^{Q_r(k)} (Q_r(k) - j) w_{kj}) \\ &> T_{que} - \hat{T}_{n-que} - \sum_{k=1}^{m} \sum_{j=1}^{Q_r(k)} (Q_r(k) - j) w_{kj} \end{aligned}$$

The above model gives conditions for the virtual reconfiguration to effectively reduce the total execution time by resolving the blocking problem. A key condition for performance gains (i.e. the above difference is larger than 0) is that the queuing time in non-reserved workstations ($\hat{T}_{n-que}$) is significantly smaller than the queuing time in all workstations without virtual reconfiguration because jobs are more evenly distributed with a virtual reconfiguration. Since time quantum $\sum_{k=1}^{m} \sum_{j=1}^{Q_r(k)} (Q_r(k) - j) w_{kj}$ in reserved workstations include CPU service times, and no page faults due to memory shortage will be conducted, the queuing time in the reserved workstations are minimized if $w_{k1} < w_{k2} ... < w_{kQ_r(k)}$. This is easy to nearly achieve because only a small portion of jobs are large ones.

The model also indicates that virtual reconfiguration can be potentially unsuccessful with the following conditions:

1. The cluster is lightly loaded, and moderate page faults in each node can be effectively reduced by dynamic load sharing.
2. Majority jobs in the workload are equally sized in their memory demands. Virtual reconfiguration will not show its effectiveness because the chance of unsuitable resource allocations is very small.
3. If the memory allocation size of a migrated job to a reserved workstation is larger than the available user space in the reserved workstation, page faults may increase the job queuing time in a reserved node.

The concern in the first condition has been addressed by adaptively reserving workstations, where the virtual reconfiguration is not initiated until the blocking problem is detected. In practice, our experiments have shown that the memory demands of jobs in a workload are rarely equally sized. Thus, the second concern should not be a base for designing a general purpose load sharing system. The third concern can be addressed by selecting the workstations with large user memory space as the reserved workstations.

6 Conclusion

Accommodating expected and unexpected workload fluctuation of service demands is highly desirable in cluster computing. Existing studies indicate that even load sharing schemes that dynamically assign and schedule resources are not able to fully utilize the available resources. We propose an adaptive and virtual reconfiguration method to address this limit. Our trace-driven simulation experiments and analysis show that the proposed method effectively improves cluster resource utilization, resulting in significant performance gain. Two technical issues should be addressed to implement the proposed method in clusters. First, the globally shared load information in each node is dynamically changed, and needs to be delivered timely and consistently [7]. Second, the system is likely to be heterogeneous from CPU speed, memory capacity, to network interfaces.

Acknowledgment: The work is a part of an independent research project sponsored by the National Science Foundation for its program directors and visiting scientists. We thank Phil Kearns for providing us a kernel programming environment. We appreciate Bill Bynum for reading the paper and for his suggestions. The comments from the anonymous referees are helpful and constructive.

References

[1] A. Barak and A. Braverman, "Memory ushering in a scalable computing cluster", *Journal of Microprocessors and Microsystems*, Vol. 22, No. 3-4, August 1998, pp. 175-182.

[2] A. Batat and D. G. Feitelson, "Gang scheduling with memory considerations", *Proceedings of 14th International Parallel & Distributed Processing Symposium* (IPDPS'2000), May 2000, pp. 109-114.

[3] S. Chen, L. Xiao, and X. Zhang, "Dynamic load sharing with unknown memory demands in clusters", *Proceedings of the 21st International Conference on Distributed Computing Systems*, (ICDCS'2001), April 2001, pp. 109-118.

[4] D. G. Feitelson and B. Nitzberg, "Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860", *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, Vol. 949, Springer, Berlin, 1995, pp. 337-360.

[5] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing", *ACM Transactions on Computer Systems*, Vol. 15, No. 3, 1997, pp. 253-285.

[6] S. Jiang and X. Zhang, "TPF: a system thrashing protection facility in Linux", *Software: Practice and Experience*, Vol. 32, Issue 3, 2002, pp. 295-318.

[7] V. Krishnaswamy, M. Ahmad, M. Raynal, and D. Bakken, "Shared state consistency for time-sensitive distributed applications", *Proceedings of the 21st International Conference on Distributed Computing Systems*, (ICDCS'2001), April 2001, pp. 602-614.

[8] L. E. Schrage, "A proof of the optimality of the shortest processing remaining time discipline", *Operational Research*, Vol. 16, 1968, pp. 678-690.

[9] S. Setia, M. S. Squillante, and V. K. Naik, "The Impact of Job Memory Requirements on Gang-Scheduling Performance", *Performance Evaluation Review*, March 1999.

[10] M. S. Squillante, D. D. Yao, and L. Zhang, "Analysis of job arrival patterns and parallel scheduling performance", *Performance Evaluation*, Vol. 36-37, 1999, pp. 137-163.

[11] V. Karamcheti and A. Chien, "A hierarchical load-balancing framework for dynamic multithreaded computations", *Proceedings of Supercomputing'98*, November 1998.

[12] L. Xiao, X. Zhang, and S. A. Kubricht, "Incorporating job migration and network RAM to share cluster memory resources", *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing* (HPDC-9), August 2000, pp. 71-78.

[13] X. Zhang, Y. Qu, and L. Xiao, "Improving distributed workload performance by sharing both CPU and memory resources", *Proceedings of 20th International Conference on Distributed Computing Systems*, (ICDCS'2000), April, 2000, pp. 233-241.

[14] S. Zhou, J. Wang, X. Zheng, and P. Delisle, "Utopia: a load-sharing facility for large heterogeneous distributed computing systems", *Software: Practice and Experience*, Vol. 23, No. 2, 1993, pp. 1305-1336.