Dynamic Cluster Resource Allocations for Jobs with Known and Unknown Memory Demands

Li Xiao, *Student Member*, *IEEE*, Songqing Chen, and Xiaodong Zhang, *Senior Member*, *IEEE*

Abstract—The cluster system we consider for load sharing is a compute farm which is a pool of networked server nodes providing high-performance computing for CPU-intensive, memory-intensive, and I/O active jobs in a batch mode. Existing resource management systems mainly target at balancing the usage of CPU loads among server nodes. With the rapid advancement of CPU chips, memory and disk access speed improvements significantly lag behind advancement of CPU speed, increasing the penalty for data movement, such as page faults and I/O operations, relative to normal CPU operations. Aiming at reducing the memory resource contention caused by page faults and I/O activities, we have developed and examined load sharing policies by considering effective usage of global memory in addition to CPU load balancing in clusters. We study two types of application workloads: 1) Memory demands are known in advance or are predictable and 2) memory demands are unknown and dynamically changed during execution. Besides using workload traces with known memory demands, we have also made kernel instrumentation to collect different types of workload execution traces to capture dynamic memory access patterns. Conducting different groups of trace-driven simulations, we show that our proposed policies can effectively improve overall job execution performance by well utilizing both CPU and memory resources with known and unknown memory demands.

Index Terms—Cluster computing, distributed systems, load sharing, memory-intensive workloads, and trace-driven simulations.

1 Introduction

The cluster system we consider for load sharing is a compute farm which is a pool of networked server nodes providing high-performance computing for CPU-intensive, memory-intensive, and I/O active jobs in a batch mode. A major performance objective of implementing a load sharing policy in a cluster system is to minimize execution time of each individual job and to maximize the system throughput by effectively using the distributed resources, such as CPUs, memory modules, and I/Os. Most load sharing schemes (e.g., [6], [7], [8], [14], [16], and [18]) mainly consider CPU load balancing by assuming each server node in the system has a sufficient amount of memory space. These schemes have proved to be effective on overall performance improvement of distributed systems. However, with the rapid development of CPU chips and the increasing demand of data accesses in applications, the memory resources in a cluster system become more and more expensive relative to CPU cycles. We believe that the overhead of data accesses and movement, such as page faults, has grown to the point where the overall performance of a cluster system would be considerably degraded without serious considerations of memory resources in the design of load sharing policies. We have the following reasons to support our claim: First, with the rapid development of RISC and VLSI technology, the speed of processors has increased dramatically in the past decade. We have seen the increasing speed gap between processor and memory, which makes performance of application programs on uniprocessor, multiprocessors, and cluster systems rely more and more on effective usage of their entire memory resources. The buffer cache for jobs' I/O data in modern Unix systems competes for the same physical main memory with the application programs' virtual pages, making memory accesses and allocations even more dynamic and demanding. In addition, the memory and I/O components have a dominant portion in the total cost of a computer system. Second, the demand for data accesses in applications running on cluster systems has significantly increased accordingly with the rapid growth of local and wide-area Internet infrastructure. Third, the latency of a memory miss (a page fault) is more than 1,000 times longer than that of a memory hit [15], [12]. Therefore, minimizing page faults through memory load sharing has a great potential to significantly improve the overall performance of cluster systems. Finally, it has been shown that memory utilizations among different nodes in a cluster system are highly unbalanced in practice, where page faults frequently occur in some heavily loaded nodes but a few memory accesses or no memory accesses are requested on some lightly loaded nodes or idle nodes [1]. A load sharing policy by only considering memory resource without considering CPU resource is very likely to cause uneven job distributions among workstations, which is not favorable for optimizing the average job queuing time. Our objective of new load sharing policy design is to share both CPU and memory services among the nodes in order to minimize both CPU idle time and the number of page faults in cluster systems.

1.1 Related Work

Besides the cited work on CPU-based load sharing policies at the beginning of the introduction, some work has been reported on memory resource considerations of load

• *The authors are with the Department of Computer Science, College of William and Mary, Williamsburg, VA 23187-8795. E-mail: {lxiao, sqchen, zhang}@cs.wm.edu.*

Manuscript received 27 Apr. 2001; accepted 21 Sept. 2001. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 114060.

sharing [1], [3], [9], [21]. An early study in [18] considers using free memory size in each node as an index for load sharing. Compared with CPU-based policies, this study did not find the effectiveness of memory-based policies. This is because the workloads were CPU intensive and processors were much slower than what we are using today. The Global Memory System (GMS) [9], [26] attempts to reduce the page fault overhead by remote paging techniques. Although the page fault cost is reduced, remote paging may also increase network contention. DoDo [1] is designed to improve system throughput by harvesting idle memory space in a distributed system. The owner processes have the highest priority for their CPUs and memory allocations in their nodes, which divides the global memory system into different local regions. In the MOSIX load sharing system, a memory ushering algorithm is used when the free memory of a node is lower than a certain amount (e.g., 1/4 MBytes) [3]. A preemptive migration is then applied to the smallest running job in the node by moving it to a remote node with the largest free memory. There are several differences in the algorithm design and evaluation between the memory ushering and our proposed CPU-Memory-based load sharing: 1) Instead of considering only memory load in the selection of a remote node, we consider both CPU and memory loads. 2) Besides preemptive migrations, we have evaluated remote executions in our simulations. In a previous study [29], we have compared the two strategies and found that remote executions could be more beneficial to memory-intensive jobs. 3) Instead of generating job memory demands with an exponential distribution, we traced job memory demands from real-world applications. The authors in [2] convert the usage of multiple resources including CPU and memory in a node to a single "cost." Load sharing is performed based on this cost on each node. They assume that a job's memory demand is known when the job arrives and generate CPU and memory demands using Pareto distributions. The distribution of lifetimes has been observed to be Pareto for CPU intensive Unix processes [14]. However, existing studies have not shown that the distribution of memory demands is Pareto. In contrast, we study load sharing to deal with both known and unknown memory demands and use real-world application workloads.

1.2 Our Approach

Aiming at reducing the memory resource contention caused by page faults and I/O activities, we have developed and examined load sharing policies by considering effective utilization of global memory in addition to CPU load balancing in clusters. Our study consists of two parts: load sharing policies dealing with known memory demands and with unknown memory workloads.

For the first part of the study, we use the real-world application traces obtained from the public domain, which contain average requested and used CPU times and average requested and used memory space for each job. Relying on the knowledge of memory demands, we develop several load sharing policies with coordinated utilizations of both CPU and memory resources. Trace-driven simulations are conducted for performance comparison and evaluation. The practical basis of this part is that memory demands of some applications can be known or can be predicted based on users' hints.

Since memory demands of many other applications may not be known in advance or hard to predict and memory accesses and allocations can be dynamically changed, it is highly desirable to develop load sharing schemes with unknown memory demands. We have addressed this issue in the second part of this study. This investigation requires workloads with dynamic memory access and allocation traces. To our knowledge, there have not been workload traces with dynamic memory information available in the public domain. Thus, we have conducted kernel instrumentation to collect application workload execution traces to capture dynamic memory access patterns and have proposed load sharing schemes dynamically monitoring the jobs status of resource utilizations and making resource allocation decisions timely and adaptively.

Conducting trace-driven simulations, we show that our policies can improve overall performance by effectively utilizing both CPU and memory resources. The rest of the paper is organized as follows: We present load sharing study with known memory demands in Section 2 and the study with unknown memory demands in Section 3, where the performance evaluation methodologies are described and the performance results are reported. Finally, we conclude the work in Section 4.

2 Sharing Both CPU and Memory Resources with Known Memory Demands

In practice, some jobs' memory demands are known in advance or are predictable based on users' hints [4]. In this part of our study, the job's memory demand is assumed to be known and the memory allocation for this job is done at the arrival of the job. A job's working set size is assumed to be stable during its execution.

2.1 CPU-Memory-Based Load Sharing

In a multiprogramming environment, multiple jobs share a node for both its CPU and memory space. There are two types of page replacement policies in a multiprogramming environment: global replacement and local replacement. A global replacement allows the paging system to select a memory page for replacement throughout the memory space of a node. A local replacement requires that the paging system select a page for a job only from its allocated memory space. Most time-sharing operating systems use a global LRU replacement policy. We use node index j to represent one node in a cluster and variable P to represent the total number of nodes in the cluster. We give the following memory related characterizations in a multiprogramming environment using a global LRU replacement policy on a single node:

- RAM_j: the amount of user available memory space on node j for $j = 1, ..., P$.
- U_j: The memory usage is the total amount of requested memory space accumulated from jobs in node j. This requested or declared amount of space reflects the maximum amount of the working set, but not the real memory load in executions.

- ML_j: The memory load in bytes is the total amount of memory loads accumulated from running jobs on node j. (After a job is in its stable stage, its working set size should also be stable [30]. We call the memory space for the stable working set the memory load of the job. If $RAM_j > ML_j$, page faults would rarely occur, otherwise, paging would be frequently conducted during the execution of jobs in node j.)
- σ_j: The average page fault rate caused by all jobs on a node is measured by the number of page faults per million instructions when the allocated memory space equals the memory load.

When a job migration is necessary in load sharing, the migration can be either a remote execution which makes jobs be executed on remote nodes in a nonpreemptive way or a preemptive migration which may suspend the selected jobs, move them to a remote node, then restart them. We have compared the performance of the remote executions with preemptive migrations for load sharing in a homogeneous environment [29]. Our study indicates that an effective preemptive migration for a memory-intensive workload is not only affected by the workload's lifetime, but also by its data access patterns. Without a thorough understanding of workloads' execution patterns interleaving among the CPU, the memory, and the I/O, it is difficult to effectively use preemptive migrations in load sharing policies. For this reason, we have decided to only use the remote execution strategy in this part. This part focuses on the three policies using remote executions: The first one is based on CPU resource information, the second one uses information on memory usage, and the third one is based on data concerning both CPU and memory resources. Descriptions of the three policies are given as follows:

CPU-Based Load Sharing. The load index in each node is represented by the length of the CPU waiting queue, L_j. A CPU threshold on node j, denoted as CT_j, is the maximum number of jobs the CPU is willing to take, which is set based on the CPU computing capability. For a new arriving job in a node, if the waiting queue is shorter than the CPU threshold ($L_j < CT_j$), the job is executed locally. Otherwise, the load sharing system tries to find the remote node with the shortest waiting queue to remotely execute this job. This policy is denoted as CPU in performance figures.

Memory-Based Load Sharing. Instead of using L_j, we propose to use the memory load, ML_j to represent the load index. For a new arriving job, if the memory load is smaller than the user memory space ($ML_j < RAM_j$), the job is executed locally. Otherwise, the load sharing system tries to find the remote node with the lightest memory load to remotely execute this job. This policy is denoted as MEM.

CPU-Memory-Based Load Sharing. We have proposed a load index which considers both CPU and memory resources. The basic principle is as follows: When a node has sufficient memory space for both running and requesting jobs, the load sharing decision is made by a CPU-based policy. When the node does not have sufficient memory space for the jobs, the system will experience a large number of page faults, resulting in long delays for each job in the node. In this case, a memory-based policy makes the load sharing decision to either submit jobs to suitable nodes or to hold the jobs in a waiting queue if necessary.

The load index of node j ($j = 1, ..., P$) combining the resources of CPU cycles and memory space is defined as

$$Index_{hp}(j)(L_j, ML_j) = \begin{cases} L_j, & ML_j < RAM_j, \\ CT_j, & ML_j \geq RAM_j. \end{cases}$$

When $ML_j < RAM_j$, CPU-based load sharing is used. When $ML_j \geq RAM_j$, the CPU queue length (the load index) is set to CT_j as if the CPU is overloaded so that the system refuses to accept jobs.

Since the load index of node j is set to CT_j when $ML_j \geq RAM_j$, it may not allow a node with the overloaded memory to accept additional jobs. This approach attempts to minimize the number of page faults in each node. This load index option is in favor of making each job execute as fast as possible, which is a principle of *high performance computing*. That is the reason we define this option as a high-performance computing load index, defined as $Index_{hp}$.

However, it may not be in favor of *high-throughput-computing* which emphasizes effective management and exploitation of all available nodes. For example, when $ML_j \geq RAM_j$ on one node, this condition may be true in several nodes. If the load indices in many nodes have been set to CT and consequently they may refuse to accept jobs, the amount of node resources accessible to users would be low. For this reason, we design an alternative load index for high-throughput-computing. Instead of aggressively setting the load index to CT_j, we conservatively adjust the load index by a memory utilization status parameter when $ML_j \geq RAM_j$. The memory utilization parameter is defined as $\gamma_j = \frac{U_j}{RAM_j}$. When $\gamma_j < 1$, it means the memory space of the node is sufficiently large for jobs. When $\gamma_j > 1$, it means the memory system is overloaded. This option is designed for high throughput computing and its load index is defined as follows:

$$Index_{ht}(j)(L_j, ML_j) = \begin{cases} L_j, & ML_j < RAM_j, \\ L_j \times \gamma_j, & ML_j \geq RAM_j. \end{cases}$$

Memory utilization parameter γ_j is used to proportionally adjust the load index. When $ML_j \geq RAM_j$, the CPU queue length is enlarged by a factor of γ_j as if the CPU were increasingly loaded. The increase of the load index would reduce the chance of this node being selected soon for a new job assignment.

Both load index options have their merits and limits and they are workload and system dependent. The load sharing policy based on the above two load indices can be expressed as follows:

$$LS(Index(j)) = \begin{cases} \texttt{local execution}, & Index(j) < CT_j, \\ \texttt{remote execution}, & Index(j) \geq CT_j, \end{cases}$$

where $Index$ is either $Index_{hp}$ or $Index_{ht}$. This policy is denoted as CPU_MEM_HP or CPU_MEM_HT.

TABLE 1
Trace Description

trace name	duration	# jobs	average CPU demand	average memory demands
MAY	May, 1996; June 1, 1996	4177	10166236 MIPS	1006 MB
JUNE	June, 1996; July 1-2, 1996	3738	9783912 MIPS	735 MB
JULY	July, 1996; August 1, 1996	8639	5121149 MIPS	552 MB
AUGUST	August, 1996; September 1, 1996	3209	11428627 MIPS	901 MB

2.2 Performance Evaluation Methodology

Our performance evaluation is simulation-based, consisting of two major components: a simulated cluster and workloads.[1]

2.2.1 A Simulated Cluster

Workstations in a cluster could be heterogeneous with different CPU powers and memory capacities. In a heterogeneous system, load indices of a node can be adjusted based on the node's relative computing capability and memory capacity in this system [28]. In order to simplify the description, we focus on presenting our study on a homogeneous system in this paper.

We simulated a homogeneous cluster with 32 nodes, where each local scheduler holds all the load-sharing policies we just discussed in Section 2.1: CPU-based, Memory-based, CPU-Memory-based, and their variations. The simulated system is configured with workstations of 800 MHZ CPUs and 1 GBytes Memory each. The memory page size is 4 Kbytes. The Ethernet connection is 100 Mbps. Each page fault service time is 10 ms and the context switch time is 0.1 ms. The overhead of a remote execution is 0.05 seconds.

The widening speed gap between CPU and memory makes memory accesses and page faults increasingly expensive. Using SPEC CPU 1995 and SPEC CPU 2000 benchmark programs and execution-driven simulations of modern computer architectures, researchers have quantitatively evaluated their execution time portions for CPU operations and memory accesses [19] and [31]. For example, using the SPEC CPU 2000 benchmarks on a simulated 1.6 GHz, 4-way issue, out-of-order core with 64 KB split L1 caches, a 1 MB on-chip L2 cache, and an infinitely large main memory, on average, the system spends 57 percent total execution time serving memory accesses (L2 misses), 12 percent of its time serving L1 misses and only 31 percent of its time for CPU operations. If we consider a small percentage of the memory accesses experiences page faults in a system with limited main memory space, the percentage of the execution time spent for CPU operations can be significantly low. Our cluster simulation environment is consistent with the reported results.

The CPU local scheduling uses the round-robin policy. Each job is in one of the following states: "ready," "execution," "paging," "data transferring," or "finish." When a page fault happens in the middle of a job execution, the job is suspended from the CPU during the paging service. The CPU service is switched to a different job. When page faults happen in executions of several jobs, they will be served in FIFO order. The page faults in each node are simulated as follows: When the memory load of jobs in a node is equal to or larger than the available memory space ($ML_j \geq RAM_j$), each job in the node will cause page faults at a given page fault rate, $\sigma_j \times \frac{ML_j^i}{MA_j^i}$, where ML_j^i is the memory load of job i in node j, and MA_j^i is the allocated memory space for job i in node j. According to our memory access traces for many application programs to be reported in Section 3, we obtain a proportional increased relationship between the page fault rate and the memory oversizing percentage. However, it is not a perfectly linear model. But our linear approximation here is sufficiently accurate to reflect the proportional increase.

In practice, application jobs have page fault rates from 1 to 10.

2.2.2 Workload Traces

We select a workload from the Los Alamos National Lab, which contains detailed information about resource requests and usage, including memory. This workload was collected from a 1,024-node Connection Machine CM-5 during October 1994 through September 1996. This workload can be downloaded from Feitelson's Workload Archive [10]. We extract four traces from this workload, which are summarized in Table 1. Trace "MAY," "JUNE," "JULY," and "AUGUST" include jobs submitted in May 1996, June 1996, July 1996, and August 1996, respectively. The parallel workloads have been converted to the sequential workloads by accumulating CPU and memory demands of all parallel tasks of each job to a sequential job. Each job in our trace has four items: 1) arrival time, 2) arrival node, 3) requested memory size, and 4) requested CPU time. Item 1 can be obtained from the original CM-5 workload directly. Item 3 and 4 are the total amount of requested CPU time and memory size of a job. Item 2 is assigned to a node whose number is the same as the job's submission date. For example, if a job is submitted on May 16, 1996, this job is assumed to be submitted to node 16. We specially assign jobs to node 31 and/or 32 as follows: Node 32 in trace "MAY," "JULY," and "AUGUST" contains all jobs submitted on June 1, 1996, August 1, 1996, and September 1, 1996, respectively. Node 31 and node 32 in trace "June" include all jobs submitted on June 1, 1996 and

1. The simulator can be accessed at http://www.cs.wm.edu/hpcs/WWW/HTML/publications/abs00-1.html.

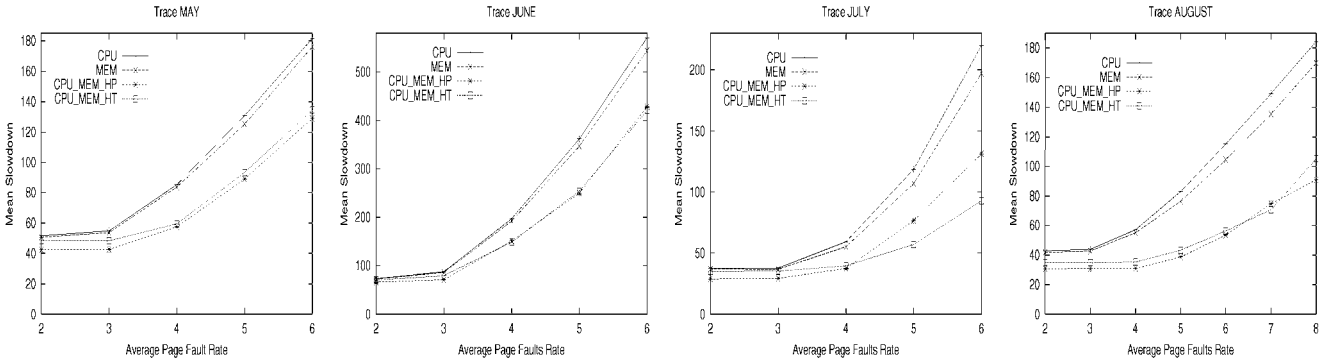

Fig. 1. Mean slowdowns of the four load sharing policies as the page fault rate increases on four traces.

June 2, 1996, respectively. We converted the job duration time into Million Instructions according to the CPU speed.

2.2.3 System Conditions

We have the following conditions and assumptions for evaluating the load sharing policies in the cluster:

- Each node maintains a global load index file which contains both CPU and memory load status information of other nodes. The load sharing system periodically collects and distributes the load information among all nodes.
- The location policy determines which node to be selected for a job execution. The policy we use is to find the most lightly loaded node in the cluster.
- Similar to assumptions in [13] and [26], we assume that page faults are uniformly distributed during job executions.
- We assume that the memory load of a job is 40 percent of its requested memory size. The practical value of this assumption has also been confirmed by the studies in [13] and [26].

2.3 Performance Results and Analysis

Slowdown is the ratio between the wall clock execution time and the CPU execution time of a job. A major timing measurement we have used is the mean slowdown which is the average of each program's slowdown in a trace. In the rest of the paper, "slowdown" means the "mean slowdown." Major contributions to the slowdown come from the delays of page faults, waiting time for CPU service, and the overhead of remote execution. The mean slowdown measurement can determine the overall performance of a load sharing policy, but may not be sufficient to provide performance insights. We have also looked into the total execution time and its breakdowns. For a given workload scheduled by a load sharing policy (or without load sharing), we have measured the total execution time. The execution time is further broken into CPU service time, queuing time, paging time, and migration time.

2.3.1 Overall Performance Comparisons

We have experimentally evaluated the four load sharing policies and present performance comparisons of all the traces. Fig. 1 presents the mean slowdowns of four traces scheduled by different load sharing policies. The average memory demand of a job is known in advance, but memory access interactions of multiple running jobs are unknown. We use different page fault rates to characterize different interactions. Intensive interactions mean that memory accesses of multiple running jobs happened at the same time, which could cause more page faults than those in less intensive interactions. Before getting into details, we present two general observations based on the results in the figures. First, the slowdown are proportionally increased as the page fault rate increases. Second, when average page fault rates are low, the performance differences among the load sharing policies are insignificant. However, when average page fault rates are high, the CPU-Memory-based load sharing policies significantly outperform both CPU-based and Memory-based policies.

Policy CPU does reasonably well when the page fault rate is low, but does poorly when the rate is high. Policy MEM performs slightly better than CPU, but it is still far below the performance of CPU_MEM-based policies. Policies CPU_MEM_HP and CPU_MEM_HT perform well under all conditions and do show their effectiveness. Here is an example on trace AUGUST: When the page fault rate is 4, the slowdowns of the last three policies are about 1.04 times lower, 1.84 times lower, 1.62 times lower than that of CPU policy, respectively. When the page fault rate is increased to 8, the slowdowns of these three policies are about 1.09 times lower, 2.03 times lower, 1.75 times lower than that of CPU policy, respectively.

2.3.2 Paging and Queuing

Our simulator also records execution breakdowns. Our experiments confirm that, in different load sharing policies, the CPU service time is not changed. The migration time spending on remote execution is neglected. So, paging time and queuing time become the major parts to evaluate performance of load sharing policies. Fig. 2 presents the paging time reduction and queuing time reduction of policies MEM, CPU_MEM_HP and CPU_MEM_HT over policy CPU for different traces when the average page fault rate is 6.

Surprisingly, the paging time reduction of policy MEM is very small. This is because policy MEM does not consider CPU load balancing at all so that some nodes may hold a large number of running jobs. However, these nodes could be viewed as lightly loaded because idle memory may still be available there. The heavy CPU load tends to keep these

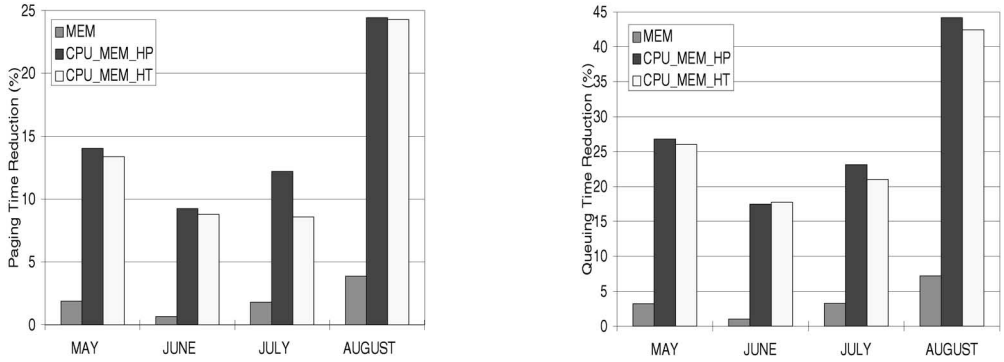

Fig. 2. Paging time reduction (left figure) and queuing time reduction (right figure) of policies MEM, CPU_MEM_HP and CPU_MEM_HT over policy CPU.

running jobs to stay longer and cause more page faults in these nodes when more jobs have to move in, which offsets the page fault reduction gained from other nodes holding less number of running jobs. In contrast, the paging time reductions of policies CPU_MEM_HP and CPU_MEM_HT are significant. For example, in trace AUGUST, the paging time reduction of policy MEM is only 3.88 percent. The reductions of CPU_MEM_HP and CPU_MEM_HT are 24.43 percent and 24.27 percent, respectively.

Queuing time reductions for different policies follow the same trend. The reduction of MEM is very small. On one hand, some nodes hold a small number of run jobs with large memory demands. The queuing time could be significantly reduced in these nodes. On the other hand, it is very likely that a large number of jobs running in the same node in a time-sharing mode because these jobs demand small memory space. The queuing time in these nodes significantly increases. The right figure of Fig. 2 clearly shows that the two parts are comparable so that the queuing time reduction of MEM is only modest. The queuing time reductions of Policies CPU_MEM_HP and CPU_MEM_HT balancing both CPU and memory loads are very effective. For example, in trace AUGUST, when page fault rate is 6, the queuing time reduction of MEM is 7.25 percent, while reductions of CPU_MEM_HP and CPU_MEM_HT are 44.16 percent and 42.43 percent, respectively.

2.3.3 High Performance and High Throughput

We have further compared the high-performance (HP) approach and the high-throughput (HT) approach in our load sharing policies (see Fig. 1). Generally, the high-performance approach is comparable with, but is slightly more effective than, the high throughput approach for all cases. This is because the high throughput approach tends to encourage more jobs to be executed in a time-sharing mode in a cluster so that it could cause slightly more page faults compared with the high-performance approach. Occasionally, the high-throughput approach outperforms the high-performance approach. A cluster managed by CPU_MEM_HP refuses to accept jobs when either CPU or memory is overloaded. This approach attempts to make each running job execute as fast as possible. But, if many jobs are refused or some jobs are delayed for a very long period of time, the overall performance could be affected. In these cases, the high throughput approach can outperform the high-performance approach. For example (see Fig. 1), the performance results in trace JULY with page fault rate of 5 and 6, in trace JUNE with page fault rate of 4, and in trace AUGUST with page fault rate of 7 give such examples.

2.4 Summary

We summarize our study on load sharing with known job memory demands as follows:

- The performance of a load sharing policy considering both CPU or memory resources is robust for all traces in this part of the study and is much better than the performance of a load sharing policy considering only CPU or only memory resource, particularly when the memory access interactions are intensive.
- The reason that CPU-MEM-based policies perform well is that these policies effectively reduce the paging time and queuing time. Meanwhile, CPU policy suffers large paging overhead and MEM policy could not reduce queuing time.
- The high-performance approach is slightly more effective than the high-throughput approach for all traces in this part.

3 Load Sharing with Unknown Memory Demands

Our study on load sharing with known job memory demands, presented in the previous section, has shown the effectiveness of polices considering both CPU and memory resources on real-world application workloads. However, job memory demands in many applications may not be known in advance or may be difficult to predict. Even in the case that a job's average memory demand could be predicted, the dynamic changes of memory accesses and allocations of the job can be hard to estimate or emulate without a low level system monitoring. In order to effectively conduct dynamic load sharing with unknown memory demands, in this section we have characterized memory performance of different types of jobs (e.g., CPU-intensive, memory-intensive, I/O active jobs) by analyzing workload traces collected from Linux kernel instrumentation. We have also developed and evaluated a dynamic load sharing

scheme and its variations by considering effective usage of global memories, networks, and CPUs. The load index on each workstation consists of the resource usage status of CPU, memory, and I/O. A job migration decision is made based on online system information to minimize the memory space competition for allocating jobs' memory space and the buffer cache of I/O data and to trade cheap CPU cycles and minor network overhead for significant memory performance gain. Some preliminary results of this part of the study have been presented in [5].

3.1 System Monitoring and Tracing

3.1.1 Why Is Kernel Instrumentation Necessary?

When memory related activities in a program execution occur, such as memory accesses and page faults, the operating system or the kernel is heavily involved. Although researchers have used several other approaches to study memory performance and to obtain memory traces, they all have some limits to characterize dynamic memory access behavior of program executions.

Memory trace collections without using kernel instrumentation can be categorized into four types. The first approach is to use workload with synthesized memory accesses. For example, the memory allocation sizes of jobs can be generated by certain distributions (such as Pareto and exponential distributions) based on experimental observations (see e.g., [29]) and there is a fixed number of working sets for each job in the workload with synthesized memory accesses. In practice, the number of working sets for a job varies from one program to another. The second approach is to make instrumentation in user programs to predict memory access statistics. For example, assuming virtual memory pages are contiguously allocated in the physical memory, one can record the assumed memory locations for each data access in the program execution (see e.g., [23]). This approach may be effective in a dedicated environment. However, in a time-sharing environment, the mapping between virtual pages and physical locations are highly dynamic. The third approach is to collect memory traces on a simulated system (see e.g., [13]). Again, most existing simulated systems do not have operating system support. Thus, dynamic behavior of memory accesses could not be accurately captured in this approach. The fourth approach is to use system utilities, such as the *top* utility in Unix, to collect instant program execution statistics, such as the memory usage, [1], [4]. The interactions between a user program and the kernel and detailed system activities may not be captured in this approach.

3.1.2 System Utilities and Instrumentation

Top is a user-level system utility which provides CPU and memory usage information in real time. Besides using *top*, we have also made careful instrumentation in the Linux kernel to accurately capture and measure system operations for a program execution. The system facility and the instrumentation tool monitor the entire life of each job. The lifetime of a job execution is divided into *CPU service time*, *paging time*, and *I/O operation time*. The CPU service time is the cycle time used for computing operations which can be partially overlapped with other non-CPU operations, such as memory accesses and the time spent for system calls to provide or initiate system services. The paging time is the CPU idle cycles waiting for long delay of memory accesses caused by page faults. The I/O operation time consists of non-page-fault disk operations, such as reads/writes of data files. In a multiprogramming environment, the lifetime of a job also includes a queuing time waiting for its own turn before other jobs finish their turns.

Our kernel instrumentation measures the three 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. The library is built with the following data structures and facilities:

- *Trace buffer*: We have built a data buffer to collect the system traces. This buffer resides in reserved kernel memory space and is nonpageable. The trace buffer is initially allocated when the system is booted. The size of the buffer is set to 4 MB.
- *Ages and the lifetime of a job*: The system creates a process control block (PCB) for each job, where the process starting time is recorded when the job is executed. The termination time is recorded by an instrumentation statement as the program exits. An age or the lifetime (the interval between the starting and a current time or the termination time) can be accurately obtained and written to the trace buffer after the execution of the job in the kernel without intrusive effect on the job execution.
- *Memory allocations*: The memory management system of the kernel provides facilities to obtain the size of the memory allocation for each process and the free memory space for user jobs. We use these facilities to periodically collect these values. Since the PCB of each job process records the memory allocation information, the operations of getting the process memory allocation size and the free memory size are quite cheap.
- *Page faults*: There are two types of page faults for each job: minor page faults and major page faults. A minor page fault will cause an operation to relink the page table to the requested page in the physical memory. The timing cost of a minor page fault is trivial. A major page fault happens when the requested page is not in the memory, which has to be fetched from a secondary storage. We collect major page fault events for each process. The major page faults are further divided into two types: the one due to memory shortage and the one for other reasons, such as cold misses.
- *Read/writes*: These I/O operations for each job process are monitored at the VFS (virtual file system) level in the kernel (the instrumentation statements are inserted inside "sys_read()" and "sys_write()" functions). The starting and termination times of each read/write and the number of bytes transferred are collected and written to the trace buffer.
- *Status of the I/O buffer cache*: The cache buffer competes with user memory allocations in the main memory. Besides the buffer size, the instrumentation records the buffer page replacement activities.

TABLE 2
Execution Performance and Memory Related Data of the Seven Application Programs

Programs	data size	working set (MB)	# paging/time(s)	# IOs/time(s)	k-time (s)	u-time (s)	lifetime (s)	paging %	IO %
bit-r	2^{23}	64.22	0/0	9/0.11	0.57	191.85	192.26	0	0.06
m-sort	2^{23}	64.27	0/0	18/0.22	1.7	80.84	82.76	0	0.27
m-m	$1,700^2$	66.37	0/0	4/0.05	1.12	4901.12	4902.29	0	0.0
t-sim	31,061	4.64	0/0	225/2.7	0.14	38.79	41.63	0	6.5
metis	1M-4M	1.37-4.30	0/0	292/3.5	8.06	112.85	124.41	0	2.8
r-sphere	150,000	36.84 — 39.66	0/0	1,624/19.49	0.97	298.18	318.64	0	6.1
r-wing	500,000	19.53 — 23.39	0/0	3,541/42.49	1.31	28.98	72.78	0	58.38

- *System clocks*: We have used two system timers in kernel instrumentation. One is the system clock (*jiffies*) which is a hardware controlled and interrupt-based timer. It ticks 100 times a second. We use this system clock to measure the lifetimes of jobs. For more precise timing measurement, such as a system event, the CPU clock cycles are used.

3.1.3 Instrumentation Validations

To ensure the accuracy of the collected traces and to evaluate the potential intrusive effects of the kernel instrumentation to job executions, we have validated the instrumentation experimentally. We have compared the timing results and the number of major page faults of workloads measured by system facilities with those of the same workloads measured by our kernel instrumentation. The results from both measurements are consistent.

We have also compared the measured lifetimes of seven application programs and six benchmark programs (to be introduced in Sections 3.2.1 and 3.6) without using the instrumentation and the ones with the instrumentation. The comparisons show that the average lifetime increase is only 2.3 percent with insignificant intrusive effects; and that the number of major page faults does not increase due to the instrumentation.

3.2 Characterizing Job Interactions

3.2.1 Characterizations of Workloads

We have selected seven large scientific and system programs which are representative CPU-intensive, memory-intensive, and/or I/O-active jobs. Here are brief descriptions for each of them:

- *Bit-reversals*, (bit-r): This program conducts data reordering operations which are required in many Fast Fourier Transform (FFT) algorithms. This program is both CPU-intensive and memory-intensive [30].
- *Merge-sort*, (m-sort): Merge-sort is sensitive to the memory hierarchy of the computer architecture, as well as sensitive to the types of data sets. This program is both CPU-intensive and memory-intensive [27].
- *Matrix multiplication*, (m-m): This is a standard matrix multiplication program and is both CPU-intensive and memory-intensive [22].
- *A trace-driven simulation*, (t-sim): This simulator is designed for evaluating load sharing schemes of distributed systems and is CPU-intensive and I/O-active [29].
- *Partitioning meshes*, (metis): This program, called METIS, is developed to partition unstructured graphs, partition meshes, and compute fill-reducing orderings of sparse matrices [17]. This program is CPU-intensive and I/O-active.
- *Cell-projection volume rendering for a sphere*, (r-sphere): This volume rendering program conducts a visualization process of creating images from aerodynamics calculations [20]. The input data of this program is a spherical volume with about 150,000 cells. The program is CPU-intensive, memory-intensive, and I/O-active.
- *Cell-projection volume rendering for flow of an aircraft wing*, (r-wing): The input data of the same volume rendering program is flow over an aircraft wing with an attached missile, with 500,000 cells.The program is CPU-intensive, memory-intensive, and I/O-intensive [20].

We first measured the execution performance of each program and monitored their memory performance related activities in a dedicated computing environment of a Pentium PC with 233 MHz CPU and 128 MByte main memory, running Linux kernel version 2.0.38. The swap space in the disk for this system is set to about the same size of the main memory (128 MB). The accumulated size of physical memory and swap space is the maximum virtual memory space application programs can practically demand. In other words, if the demanded memory space oversizes 100 percent of the physical memory space in this system, the operating system will immediately terminate the newly arrived process to release the memory space.

Table 2 presents the experimental results of all the seven programs, where "data size" is the number of entries of the input data, "working set" gives a range of the memory space demand during the execution, "# paging/time(s)" is the number of page swaps between the main memory and the disk due to memory shortage and the time spent for the paging, "# I/Os/time(s)" is the number of reads/writes and the time spent in seconds, "k-time" is the execution time used in the system mode for kernel system calls, "u-time" is the execution time used in the user mode including the CPU service time and memory access time without page faults, "lifetime" is the sum of k-time, u-time, I/O time, and paging time, "paging %" and "I/O %" are the ratios of the paging time and the I/O time, respectively, to the lifetime of a job. The I/O portion reflects the amount of

TABLE 3
Execution Performance and Memory Related Data of Three Groups of Interacted Programs

Interactions	oversizing	# paging/time(s)	paging rate (1/s)	# IOs/time (s)	k-time (s)	u-time (s)	lifetime (s)	paging %	I/O %
m-m	5.34%	4,934/258.99	12.52	4/0.05	2.32	4862.39	5,123.75	5.05	0.0
m-sort		3,922/49.06		18/0.22	2.80	78.98	131.06	37.4	0.17
bit-r	24.75	11/0.72	44.12	9/0.11	0.67	199.04	200.54	0.36	0.05
m-sort		5,152/63.45		18/0.22	2.75	80.11	146.53	43.3	0.15
r-wing		487/22.02		3,541/41.41	1.56	19.53	84.52	26.05	48.99
bit-r	64.95%	8,334/323.17	102.03	9/0.11	1.75	213.34	536.62	60.05	0.02
m-m		11,286/728.39		4/0.05	4.68	4790.49	5518.93	13.18	0.0
m-sort		37,132/769.94		18/0.22	5.12	75.34	845.50	90.52	0.03
t-sim		440/54.99		225/2.7	0.35	40.16	97.85	56.00	2.76

the I/O activities of a program. Since each program is executed in a dedicated environment and its memory demand is lower than the available memory space, the numbers of page faults, paging times, and paging portions, due to memory shortage are all 0s in Table 2.

3.2.2 Characterizations of Program Interactions

We have monitored executions of many groups of programs in a time-sharing environment to observe the effects of program interactions to memory performance. The executions of most groups with four programs experienced severe thrashing and were not able to complete. The executions of groups with six or seven programs were immediately stuck due to huge amount of thrashing. Table 3 presents monitored execution performance of three representative job interactions of two jobs (m-m and m-sort), three jobs (bit-r, m-sort, and r-wing), and four jobs (bit-r, m-m, m-sort, and t-sim). In this table, the item "oversizing" represents the overloaded percentage of memory demand from the interacted programs due to the memory space shortage and the term "paging rate" represents the number of page faults per second during the paging period for all the interacted programs.

Our experimental results characterize the effects of job interactions to memory performance with the following observations:

- The page faults continuously happen as the memory demand from the interacted jobs oversizes the available physical memory space.
- As the memory demand from the interacted jobs oversizes about 65 percent of the available physical memory space, some programs start to experience thrashing with little useful execution. For example, as the four programs of "bit-r," "m-m," "m-sort," and "t-sim" are interacted, more than 90 percent execution time of "m-sort" is used for paging. As the memory demand oversizes more than 70 percent, all the programs experience thrashing with little useful work to be done, where the sharply increasing paging rate starts to grow slowly to reach a stable state.
- When the memory demand from the interacted jobs oversizes less or around 65 percent of the available memory space, the amount of page faults are not evenly distributed among the jobs. The LRU page replacement policy targets the pages which are not frequently used. Thus, jobs experiencing a large amount of page faults are the ones which demand memory space dynamically. Program "m-sort" is such an example. As the percentage of oversized memory space changes from 5.34 percent to 24.75 percent and to 64.95 percent, caused by the increase of the number of interacted programs, the paging time portion in the execution of this program changes from 37.40 percent to 43.3 percent and to 90.52 percent, respectively. On the other hand, a program requesting a stable working set in the entire execution could soon get the working set and keep it, or a major part of it, during the execution. Program "m-m" is such an example. As the memory oversizing percentage reaches 64.95 percent, the paging time portion is 13.18 percent, which is significantly lower than that of the interacted program "m-sort."
- Our experiments also indicate that the paging rates of the interacted programs are highly correlated to the the percentage of oversized memory space demanded by the interacted programs and highly correlated to the number of read/write I/O of the interacted programs. Applying a polynomial interpolation to our measured paging rates versus the memory oversizing percentages of all the experiments of job interactions (several runs for each experiment), we obtain the following paging rate model:

$$R_p(Q_o) = R_{max} - \frac{\alpha}{\beta Q_o^2 + \gamma Q_o + \delta}, \qquad (3.1)$$

where R_p is the paging rate due to memory shortage with little I/O buffer cache involvement, R_{max} is a system dependent variable representing the maximum paging rate when the system reaches the stable thrashing state, Q_o is the percentage of oversized memory space and α, β, γ, and δ are workload dependent parameters. The model consistently characterizes our experimental observations of the paging rate changes. Paging rate R_p proportionally increases as Q_o increases to 65 percent. Further, increasing Q_o turns $R_p(Q_o)$ into a stable stage reaching to R_{max}. Here are the values of the variables

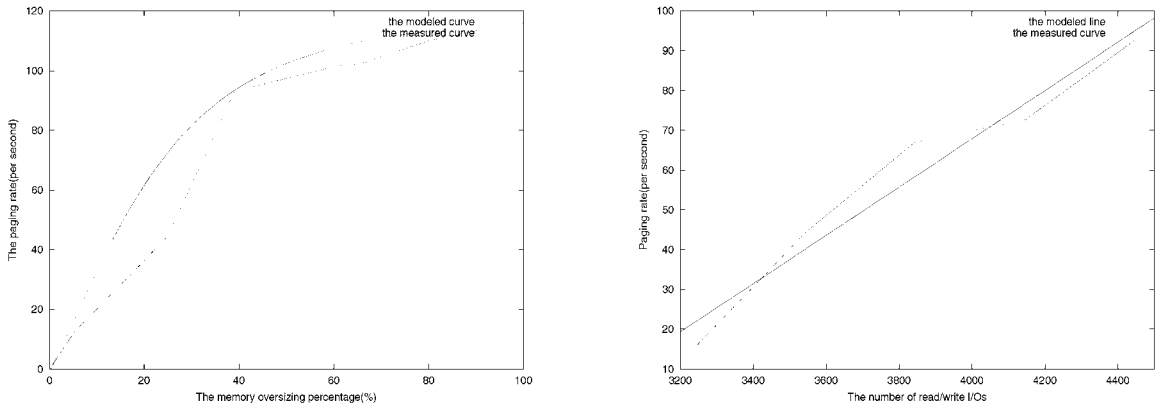

Fig. 3. Modeled and measured correlations between the paging rate and the memory oversizing percentage (left) and between the paging rate and the read/write IOs (right).

from the polynomial interpolation based on the experimental results with our workloads: $R_{max} = 120$ page faults per second, $\alpha = 4$, $\beta = 0.31$, $\gamma = 0.19$, and $\delta = 0.034$.

Applying a least squares fit to our measured page fault rates due to I/O activities versus the number of read/write I/Os of all the experiments of job interactions, we obtain another related paging rate model:

$$R_p(N_{io}) = \eta + \mu N_{io}, \tag{3.2}$$

where N_{io} is the number of read/write I/Os from the interacted programs, and η and μ are workload dependent parameters ($\eta = -174.98$ and $\mu = 0.061$ determined by experiments with our workloads). Fig. 3 plots the two correction curves and their modeled curves.

- However, we found that the paging rates have little correlations with the number of jobs interacted in the system for a given amount of the oversized memory demand.

Since the buffer cache in modern Unix systems competes for the same physical main memory with application programs' virtual pages, the page fault rate is linearly proportional to the changes of Q_o and N_{io}. Thus, the page fault rate model can be expressed as the sum of (3.1) and (3.2):

$$R_p(Q_o, N_{io}) = \eta + \mu N_{io} + R_{max} - \frac{\alpha}{\beta Q_o^2 + \gamma Q_o + \delta}, \tag{3.3}$$

Particularly, the paging rate is determined based on experiments with our workloads as follows:

$$R_p(Q_o, N_{io}) = -54.98 + 0.061 N_{io} - \frac{4}{0.31 Q_o^2 + 0.19 Q_o + 0.034}. \tag{3.4}$$

This model is used in our trace-driven simulation.

3.3 A Dynamic Memory-Centric Load Sharing Scheme

3.3.1 The Framework and Organizations

Our load sharing scheme aims at either eliminating or reducing page faults with low migration costs in each machine in a cluster. There are three unique features of our scheme compared with existing ones: 1) Since the scheduler in each machine does not have the knowledge of demanded memory's size and its range of each job in its lifetime, it dynamically monitors the demanded memory allocations of jobs and compares them with the available physical memory space to make scheduling decisions accordingly. 2) A memory threshold is set to ensure that demanded memory allocations of jobs are not oversized or only oversized to a certain degree. 3) Whenever page faults due to memory shortage or exceeding a memory threshold are detected in a machine, new job submissions to the machine will be blocked and remotely submitted to other lightly loaded machines with available memory space and additional CPU cycles. Meanwhile, one or more jobs which are already executed in the machine may also be migrated to other lightly loaded machines. The load sharing scheme is described by the variables and parameters in Table 4.

The framework of the memory-centric load sharing scheme in machine j ($j = 1, \ldots, m$) is described by the following loop:

While the load sharing system is on
  while ($Q_o(j) < MT(j)$ and $N_{job}(j) < CT(j)$)
    enjoy page-fault-free job executions and
    allow new job submissions;
  block job submissions;
  if ($Q_o(j) \geq MT(j)$)
    if ($Q_o(j) \leq 0$)
      continue enjoying page-fault-free executions of existing jobs;
      if there are arrival jobs to the machine
        while NOT *find_a_suitable_machine_for_remote_submission*
          continue to block job submissions;

TABLE 4
Notations, Variables, and Parameters of the Page-Free Load Sharing Scheme

Variables for job $i = 1, ..., n$	Variable descriptions
$R_p(i)$	The number of page faults per second due to memory shortage.
$N_{io}(i)$	The number of reads and writes.
$Age(i)$	Executed time of the job.
$WS(i)$	Currently allocated memory space (working set size).
Variables for machine $j = 1, ..., m$	Variable descriptions
$N_{job}(j)$	The number of running jobs in the machine.
$Q_o(j)$	The oversized memory demand percentage.
$FM(j)$	The amount of free memory in MBytes.
$FB(j)$	The buffer cache size.
Parameters set in machine $j = 1, ..., m$	Parameter descriptions
$CT(j)$	CPU threshold: the maximum number of jobs allowed in this machine.
$MT(j)$	Memory threshold: the maximum oversized memory demand percentage.

submit the first arrival job to the *suitable_machine*;
else
while NOT *find_a_suitable_machine_for_migration*
continue local executions;
migrate the *identified_job* to the *suitable_machine*;
$N_{job}(j) = N_{job}(j) - 1$;

For a given $MT(j)$, three load sharing decisions are made in procedures *identified_job*, which returns the ID of the to-be-migrated job, *find_a_suitable_machine_for_migration*, which seeks a *suitable_machine* for a job migration and *find_a_suitable_machine_for_remote_submission*, which seeks a *suitable_machine* for a remote job submission. The variations of the memory-centric load sharing scheme can be designed by giving different alternatives of the three procedures, and/or by changing the memory threshold parameter.

3.3.2 The Variations of the Load Sharing Scheme

Here are the four alternatives of identifying a job for migrations:

1. Identifying the most memory-active job:
   identified_job()
   d = the job ID with the highest $R_p(d)$ or the highest $WS(d)$;
   return d;
2. Identifying the most I/O active job:
   identified_job()
   d = the job ID with the highest $N_{io}(d)$;
   return d;
3. Identifying either the oldest or youngest job:
   identified_job()
   d = the job ID with the highest (**or** the lowest) $Age(d)$;
   return d;
4. Identifying a job with the highest (or the lowest) average weight of all the activities.
   identified_job()
   d = the job ID with the highest (**or** the lowest) $Weight(d)$;
   return d;
   where $Weight(d)$ is the weight of a job with highest $(\max)$ or lowest $(\min)$ average activities of CPU, memory, and I/O among all the jobs:

$$Weight(d) = \frac{1}{4}\left(\frac{Age(i)}{\max_{i=1}^{n} Age(i)} + \frac{R_p(i)}{\max_{i=1}^{n} R_p(i)} + \frac{WS(i)}{\max_{i=1}^{n} WS(i)} + \frac{N_{io}(i)}{\max_{i=1}^{n} N_{io}(i)}\right). \tag{3.5}$$

The procedures of finding a suitable machine for migrations and for remote submissions with the ID of the identified job, d, are defined as follows:

find_a_suitable_machine_for_migration $(WS(d))$
k = the machine ID with the largest free memory and $N_{job}(k) < CT(k)$;
if $(Q_o(k) < MT(k))$ or $(FM(k) > WS(d)$ or $(FM(k) + FB(k)) > WS(d))$
suitable_machine= k;
return TRUE;
else
return FALSE;

find_a_suitable_machine_for_remote_submission()
k = the machine ID with the largest free memory and $N_{job}(k) < CT(k)$;
if $(Q_o(k) < MT(k))$
suitable_machine= k;
return TRUE;
else
return FALSE;

Overheads of page faults and job migrations are two major sources of performance degradations in a cluster. Reductions of page faults and of migration costs conflict because page faults can be reduced by frequent migrations.

The memory threshold in each machine (oversized memory demand percentage, $MT(j)$, $j = 1, \ldots, m$) is a key parameter to trade-off the two overhead costs. By setting different values to MT, we are able to design load sharing schemes for different objectives and under different system and workload conditions:

- *Load sharing with a minimization of page faults.* The memory threshold is set below 0 percent ($MT(j) < 0\%$) so that a warning signal could be given before the page faults really happen. The warning signal will immediately block submissions to the machine. In addition, whenever page faults are detected, one or more identified jobs will be migrated away to stop page faults in the machine. Although this alternative is to eliminate the page faults in each machine, a small amount of page faults is not avoidable because the memory allocations is not known until the job is executed and page faults are detected during the execution. The number of page faults can be minimized by this approach. Obviously, this approach would likely conduct frequent migrations.
- *Load sharing allowing a short period of page faults.* Our experiments in the previous sections have shown that page faults often happen continuously during job interactions. However, if one or more interacted jobs could complete executions soon, the page faults would stop. In this approach, the memory threshold can be set to 0 percent ($MT(j) = 0\%$). Instead of immediately identifying a job to migrate immediately after page faults happen, in this approach we delay the migration for a short period of time, hoping one of the jobs will finish execution during this period to release the memory space and stop the page faults in the machine. This approach aims at reducing the number of migrations without causing significant increase of page faults.
- *Load sharing with a minimization of migrations.* This approach is a combination of the previous two. The memory threshold is set above 0 percent ($MT(j) > 0\%$). As the memory threshold is reached, job submissions to the machine will be blocked. After that, an internal memory threshold is used, which will allow page faults happen to a certain degree. This tolerance serves two purposes: 1) to hope page faults go away due to completions of some jobs and 2) since the page faults are under control, we hope to trade the low penalty from a short period of page faults with high migration costs.

3.4 Trace-Drive Simulation Environment

3.4.1 A Simulated Cluster

We have simulated a homogeneous cluster with 32 nodes, where each local scheduler making the load-sharing policies we have discussed in this part of the paper.

The simulated system is configured with workstations of 233 MHz CPUs, 128 MByte Memory each, and 128 Mbyte swap space each. Other parameters are the same as those in Section 2.2.1. The remote submission/execution cost, r, is 0.1 seconds for 10 Mbps network and is 0.05 seconds for 100 Mpbs 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 D is the amount of data in bits to be transferred in the job migration and B is the network bandwidth in Mbps ($B = 10$ or 100 for the Ethernet). Each node maintains a global load index file which contains CPU, memory, and I/O load status information of other nodes. The load sharing system periodically collects and distributes the load information among all nodes.

3.4.2 Workloads

The workload traces are collected by using system facility tool *top* and our instrumentation library to monitor the execution of the seven application programs at different submission rates on a Linux workstation. Job submission rates are generated by a lognormal function:

$$R_{ln}(t) = \begin{cases} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(\ln t - \mu)^2}{2\mu^2}} & t > 0 \\ 0 & t \leq 0, \end{cases} \tag{3.6}$$

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., [11], [25]). Three traces are collected with three different arrival rates:

1. *Trace 1* (highly intensive job submissions): $\sigma = 0.5$, $\mu = 0.5$, and 503 jobs submitted in 1,365 seconds.
2. *Trace 2* (moderately intensive job submissions): $\sigma = 3.5$, $\mu = 3.5$, and 420 jobs submitted in 2,503 seconds.
3. *Trace 3* (normal job submissions): $\sigma = 5.0$, $\mu = 5.0$, and 359 jobs submitted in 3,634 seconds.

The jobs in each trace were randomly distributed among 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 allocation demand, and the number of I/Os. Thus, dynamic memory and I/O activities, such as memory allocations and buffer cache allocations, can be closely monitored. During job interactions, page faults are generated accordingly by the simulation model presented in Section 3.2.

3.5 Performance Evaluation

Conducting the trace-driven-simulations on a 32 node cluster, we have evaluated the performance of the memory-centric load sharing scheme and its variations by quantitatively answering several questions in the following sections.

3.5.1 Effects of Load Index Variations

Since our workloads are not highly I/O intensive, we do not consider the alternative of I/O activity only in our performance evaluation. We have examined the following load indices to identify a job for migration when the amount

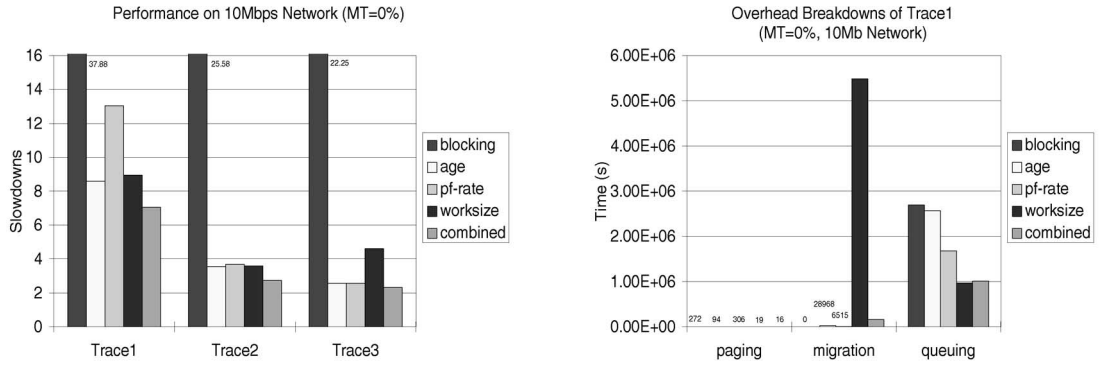

Fig. 4. Slowdowns of the three traces scheduled by the five policies (the left figure) and overhead breakdowns of *Trace 1* by the five policies (the right figure). The experiments were conducted on a 10 Mbps network with a memory threshold of 0 percent.

of workloads exceeds the CPU or/and memory thresholds in a machine.

- *page-fault-rate-based*: to migrate the job with the highest page fault rate (simplified as *pf-rate* in figures),
- *working-set-size-based*: to migrate the job with the largest working set size (simplified as *worksize* in figures),
- *age-based*: to migrate the job with the oldest age with a migration cost estimation [16] (simplified as *age* in figures),
- *CPU-memory-I/O-based*: to migrate the job with the highest activities of CPU, memory, and I/O calculated in (3.5) (simplified as *combined* in figures).

In addition, we have also compared the performance of the above schemes with a system without using load sharing, denoted as *no-LS*. Our experiments show that many jobs were terminated by the system due to a severe memory shortage by *no-LS* policy. For example, only 363 jobs completed for *Trace 1* out of 503 jobs (at a highly intensive submission rate), 390 jobs completed for *Trace 2* out of 420 jobs (at a moderately intensive submission rate), and 341 jobs completed for *Trace 3* out of 359 jobs (at a normal submission rate).

In order to make all the jobs in a trace complete, we include another policy for comparisons, in which no load sharing is used, however, the job submissions are blocked when system thrashing occurs. Therefore, all the jobs will complete. This policy is denoted as *no-LS-blocking* (simplified as *blocking* in figures).

The left figure in Fig. 4 shows the slowdowns of the three workload traces scheduled by the four load sharing policies and policy *no-LS-blocking*. The memory threshold is set to $MT = 0\%$ and the network speed is 10 Mbps. Although all the jobs completed by policy no-LS-blocking, the executions suffered significantly from huge slowdowns: 37.88 for *Trace 1*, 25.58 for *Trace 2*, and 22.25 for *Trace 3*.

Although all the four load sharing policies outperformed *No-LS-blocking* 3 to 10 times measured by the average slowdowns for all the three traces, they performed differently for different traces. The *CPU-Memory-I/O-based* policy was most effective with the lowest slowdown for all the three traces by comprehensively considering CPU, memory, and I/O buffer cache information.

The *page-fault-rate-based* policy performed the worst with the highest slowdown for *Trace 1*, (about 100 percent higher than that of *CPU-Memory-I/O-based*). For *Trace 2* and *Trace 3*, the *page-fault-rate-based* policy was quite effective, (about 30 percent higher for *Trace 2* and about 10 percent higher for *Trace 3* compared with the slowdowns of *CPU-Memory-I/O-based*). The page fault rate information reflects the page fault intensity of a job at that moment and does not tell us the working set size and how long the job has run. Thus, the *page-fault-rate-based* policy may identify a job which has a small working set and completes soon. Migrating such a job may not release sufficient space and justify the migration costs. The *working-set-size-based* policy performed reasonably well for *Trace 1* and *Trace 2* (about 30 percent higher for both traces compared with the slowdowns of *CPU-Memory-I/O-based*). However, its slowdown for *Trace 3* was more than 60 percent higher than those of three other policies. The working set size information reflects the current allocated data size of a job, which also gives a memory space size to be released if the job is identified. When the job submission rate is high, such as *Trace 1* and *Trace 2*, migrating a job with the largest working set at a high migration cost to release a large space for new jobs seems to be effective. When the job submission rate is normal as *Trace 3*, the high migration cost can degrade the performance. Although the *age-based* policy caused a slowdown in the three workload traces of 10 percent to 30 percent more than the *CPU-Memory-I/O-based* policy, it is considered to be a stable and effective choice. The age information reflects the history of a job. Previous studies (see e.g., [16]) observed that the older a job is, the longer the job will stay in a system. The *age-based* policy follows this principle to identify the oldest job to migrate to justify the migration overhead.

The right figure in Fig. 4 shows the overhead breakdowns for *Trace 1*, including the paging time, migration time, and queuing time for *Trace 1*. Except for the *No-LS-blocking* policy, the *page-fault-rate-based* policy had the highest paging time while the *working-set-size-based* and *CPU-Memory-I/O-based* policies had the lowest ones. These two policies are effective in reducing the page faults by migrations. However, the *working-set-size-based* policy had the lowest queuing time at the highest cost of migration time, while the *age-based* and *page-fault-rate-based* policies

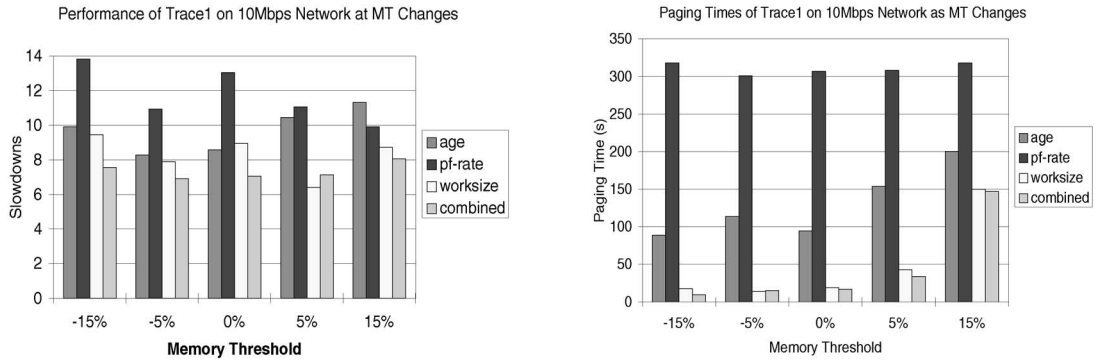

Fig. 5. Slowdowns of *Trace 1* scheduled by the four load sharing policies as the memory threshold changes (the left figure) and their paging time changes as the memory threshold changes (the right figure). The experiments were conducted on a 10 Mbps network.

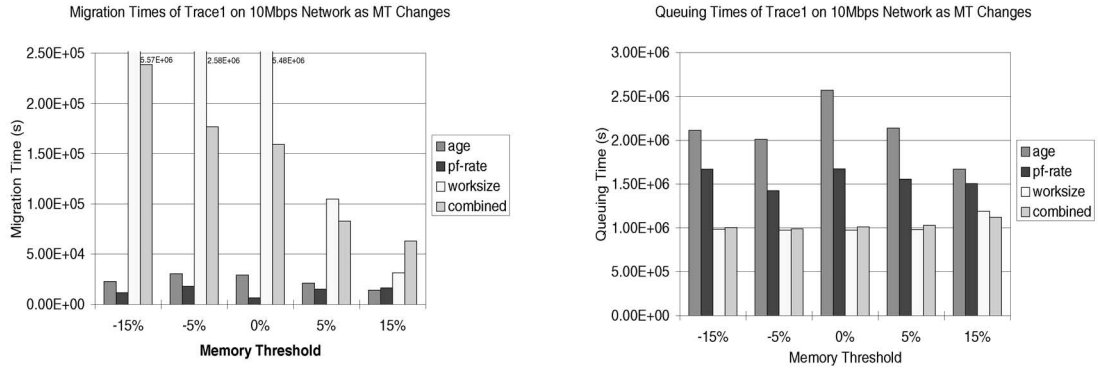

Fig. 6. Migration times of *Trace 1* scheduled by the four load sharing policies as the memory threshold changes (the left figure) and their queuing time changes as the memory threshold changes (the right figure). The experiments were conducted on a 10 Mbps network.

had lowest migration costs but high queuing times. (Notice: Since we use the mean slowdown as the metric, the value of the slowdown for a trace may not be proportional to its total execution time, the migration time, or the queuing time.) Only the *CPU-Memory-I/O-based* policy minimizes the queuing time with a slightly higher migration time than those of *age-based* and *page-fault-rate-based* policies. This balanced distribution between the migration and queuing times makes the *CPU-Memory-I/O-based* policy perform the best for all the traces.

3.5.2 Memory-Threshold-Based Optimizations

We have also examined the effects of memory threshold variations to the load sharing performance. The maximum oversized memory demand percentage was conservatively and aggressively set to -15 percent, -5 percent, 0 percent, 5 percent, and 15 percent. The left figure in Fig. 5 shows the slowdown changes of the four load sharing policies as the memory threshold changes, where the *Trace 1* is used and the network speed is 10 Mbps. The slowdowns of the *CPU-Memory-I/O* policy decreases as MT increases from -15 percent to -5 percent, stays unchanged in the range of $MT = -5\%, 0\%, 5\%$, and increases as MT increases to 15 percent. The slowdowns of the *age-based* and *working-set-size-based* policies follow a similar pattern, but reaches the minimum point at $MT = -5\%$ and at $MT = 5\%$, respectively, and increase as MT further increases. The *working-set-size-based* policy always migrates a job with the largest working set and, thus, is more generous to allow more page faults occur in a system. The slowdowns of the *page-fault-rate-based* policy seem not to follow a regular pattern as the MT changes, which again confirms its indeterministic nature.

The right figure in Fig. 5 shows the paging time changes of the four load sharing policies as the memory threshold changes under the same system condition. As we expected, the paging times of all the policies except the *page-fault-rate-based* policy proportionally increase as MT increases (sharply increase as MT changes from 5 percent to 15 percent). The paging times of the *page-fault-rate-based* policy are significantly higher than those of other policies, but are quite independent of the changes of MT.

The left figure in Fig. 6 shows the migration time changes of the four load sharing policies as the memory threshold changes under the same system condition. The migration times of the *age-based* and *page-fault-rate-based* policies are quite independent of the changes of MT. This is because these two policies do not consider data size for a migration. Thus, increasing the memory threshold will not change the amount of the data to be migrated. In contrast, the migration times of the other two data-size-related policies are sensitive to the changes of MT, which proportionally decrease as MT increases. When MT is larger than 0 percent, the migration time of the *working-set-size-based* policy begins to decrease. More jobs will interact in a machine as MT increases. Since the memory system will be overloaded, for example, oversizing 5 percent and 15 percent, some jobs may not be able to establish their entire working sets. The maximum working set size among the jobs will also decrease under such a condition and the migration times of the *working-set-size-based* policy decrease accordingly.

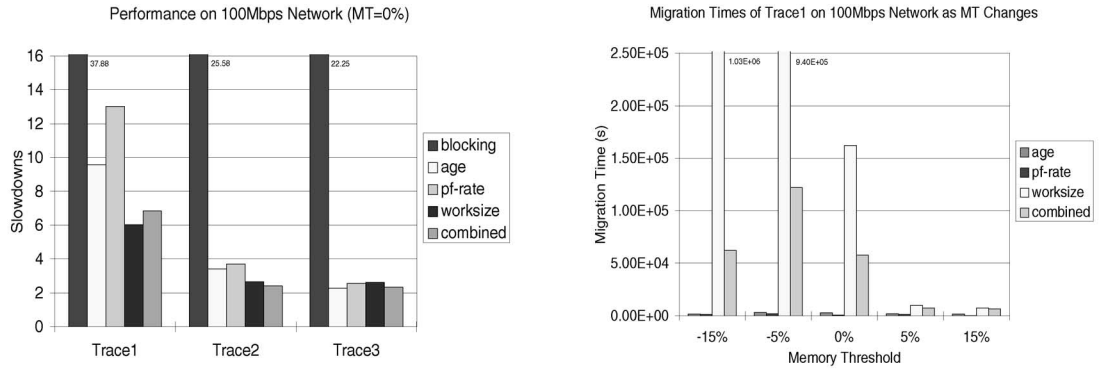

Fig. 7. Slowdowns of the three traces scheduled by the four load sharing policies (the left figure) and the migration times of *Trace 1* as the memory threshold changes (the right figure). The experiments were conducted on a 100 Mbps network.

The right figure in Fig. 6 shows the queuing time changes of the four load sharing policies as the memory threshold changes under the same system condition. We show that the queuing times of all the policies are quite independent of MT changes. As we have shown, the queuing time is mainly determined by a policy used. However, we have also noticed that the queuing times of the *working-set-size-based* policy slightly increases as MT reaches to 15 percent. This is because more jobs are allowed to execute in a machine as MT increases, so the queuing time increases also.

3.5.3 Effects of the Network Speed

Job migrations rely on the cluster network for data transfers. However, the performance of different schemes is affected differently by the changes of network speed. The cluster network we have used in this study is an Ethernet bus of 10 Mbps and 100 Mbps. The left figure in Fig. 7 presents the slowdowns of the three workload traces scheduled by the four load sharing policies and policy *no-LS-blocking* as the network is upgraded to 100 Mbps. The memory threshold is set to $MT = 0\%$. The network speed upgrading only affects the policies sensitive to migration times. The *no-LS-blocking* policy is not affected at all and the *age-based* and *page-fault-rate-based* policies are not sensitive to migration times. The slowdowns of these three policies have few changes compared with the ones running at the 10 Mbps network (see the left figure in Fig. 4). The network upgrading from 10 Mbps to 100 Mbps does reduce the slowdowns of the *working-set-size-based* policy for Trace 1, Trace 2, and Trace 3, by 32 percent, 25 percent, and 50 percent, respectively, but only slightly reduces the slowdowns of the *CPU-Memory-I/O-based* policy.

The right figure in Fig. 7 presents the migration time of the three workload traces scheduled by the four load sharing policies as the network is upgraded to 100 Mbps. Again, the migration times of the *age-based* and *page-fault-rate-based* policies are only slightly affected by the network upgrading. In contrast, the migration times of the *working-set-size-based* policy is significantly reduced by the network upgrading and the migration times of the *CPU-Memory-I/O-based* policy is moderately reduced (see the left figure in Fig. 6 for comparisons).

3.6 Performance on Workload Generated from SPEC 2000 Benchmark

Using the same method described in Section 3.4.2, we generate another group of three traces from six SPEC 2000 benchmark programs. We call them SPEC traces in order to distinguish from the three traces generated from seven application programs. The six selected programs are *apsi, gcc, gzip, mcf, vortex*, and *bzip*, which are both CPU and memory intensive. Using the facilities described in Section 3.1.2, 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 Mbytes of physical memory and a swap space of 380 MBytes. The operating system is Redhat Linux release 6.1 with kernel version 2.2.14. Table 5 presents the basic experimental results of the six SPEC 2000 programs, where "description" gives the application nature of each program, "input file" is

TABLE 5
Execution Performance and Memory Related Data of the Six SPEC 2000 Benchmark Programs

Programs	description	input file/size	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

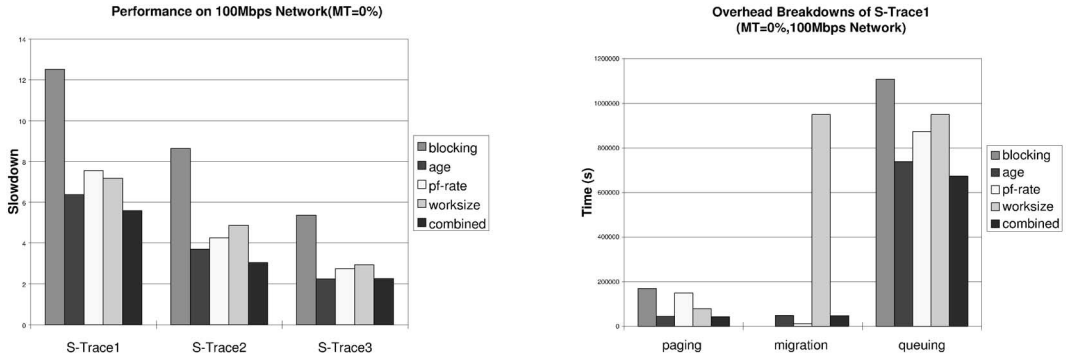

Fig. 8. Slowdowns of the three SPEC traces scheduled by the five policies (the left figure) and overhead breakdowns of *Trace 1* by the five policies (the right figure). The experiments were conducted on a 100 Mbps network with a memory threshold of 0 percent.

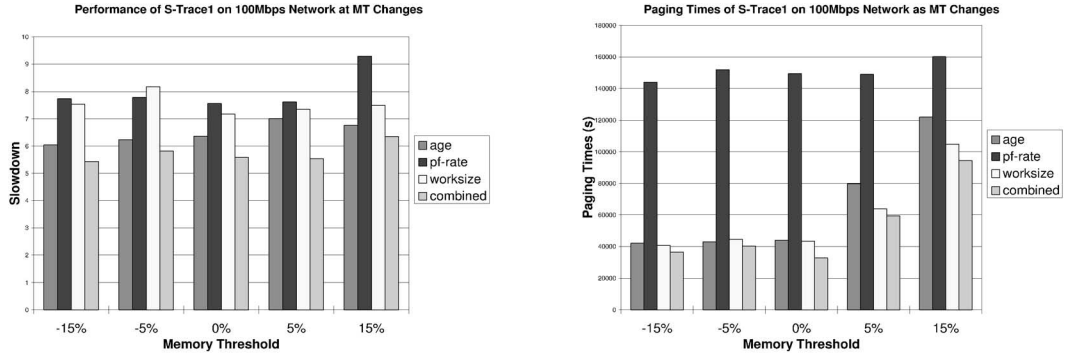

Fig. 9. Slowdowns of SPEC *Trace 1* scheduled by the four load sharing policies as the memory threshold changes (the left figure) and their paging time changes as the memory threshold changes (the right figure). The experiments were conducted on a 100 Mbps network.

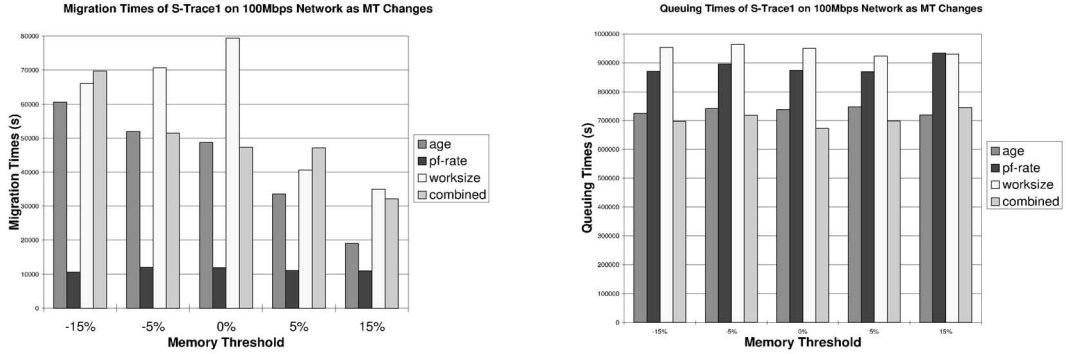

Fig. 10. Migration times of SPEC *Trace 1* scheduled by the four load sharing policies as the memory threshold changes (the left figure) and their queuing time changes as the memory threshold changes (the right figure). The experiments were conducted on a 100 Mbps network.

the input file names from SPEC 2000 benchmarks, "working set" gives the maximum size of the allocated memory space during the execution, "lifetime" is the total execution time of each program.

Conducting the same type of trace-driven simulations with the dynamic memory-centric load sharing scheme and its variations as the ones described in Section 3.5, we have evaluated the three workload traces generated from the six SPEC 2000 benchmark programs. Fig. 8, Fig. 9, Fig. 10, and Fig. 11 present the performance results including slowdowns, queuing times, and paging times, subject to the changes of the memory threshold and network speed. As we expected, the performance results are consistent with the results for the workload traces generated from the seven application programs presented in Section 3.5.

3.7 Summary

We summarize our study on dynamic load sharing with unknown job memory demands as follows: Since different types of workloads demand computing resources differently, several key parameters and variations of load sharing should be adjusted and considered adaptively for performance optimization. Here are some suggestions:

- Memory threshold should be set around 0 percent. A conservative MT does not fully utilize the memory system, while an aggressive MT may generate more page faults, negatively affecting the overall performance.
- The network speed upgrading can significantly improve the performance of the *working-set-size-based* policy and moderately improve the performance of

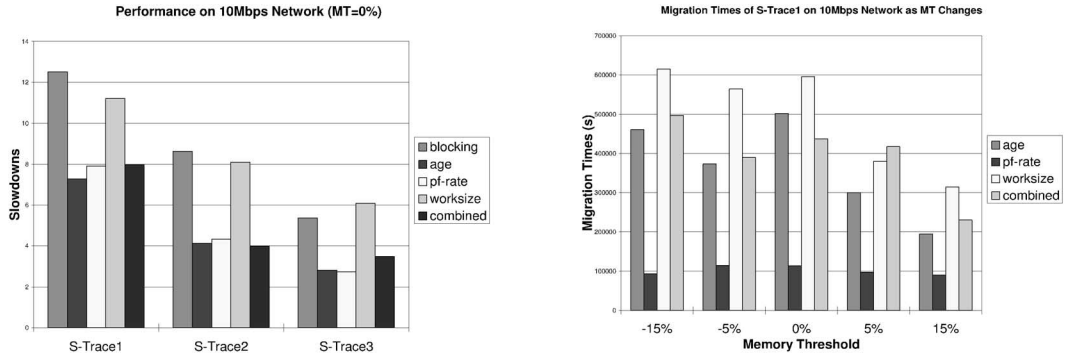

Fig. 11. Slowdowns of the three SPEC traces scheduled by the four load sharing policies (the left figure) and the migration times of SPEC *Trace 1* as the memory threshold changes (the right figure). The experiments were conducted on a 10 Mbps network.

the *CPU-Memory-I/O-based* policy. However, it affects little the performance of the other load sharing policies.

- The *page-fault-rate-based* policy is not recommended, particularly for workloads with high submission rates. The *age-based* policy can be effective for workloads with low or normal submission rates. The *working-set-size-based* policy is recommended only for high speed network clusters. The *CPU-Memory-I/O-based* policy is effective for all three types of traces on both low and high speed networks.

4 Conclusion

This paper aims at providing effective load sharing strategies to improve overall performance of cluster systems by coordinating the usage of CPU and memory resources. We first evaluate our strategies on a real workload with an assumption that jobs' memory demands are known in advance. We show that the proposed load sharing policies considering both CPU or memory resources are robust and significantly outperform the policies considering only CPU or only memory resources, particularly when the memory access interactions are intensive.

In order to address the issues of unknown memory demands and unpredictable dynamic memory access patterns, in the second part of the study, we collect dynamic memory access information from the execution of 13 large scientific programs by Linux Kernel instrumentation and system utilities and further propose load sharing policies to deal with unknown memory demands. Our trace-driven simulations consistently show the effectiveness of the policies by comprehensively considering the online information of CPU, memory, and I/O resources.

Acknowledgments

This work is supported in part by the US National Science Foundation (NSF) under grants CCR-9812187, EIA-9977030, and CCR-0098055, by a Research Scholarship of the Usenix Advanced Computing Systems Association, and by Sun Microsystems under grant EDUE-NAFO-980405. The authors would like to thank Phil Kearns' help for the kernel instrumentation on Linux systems and his suggestions. Raphael Finkel further explained his implementations of memory image transferring in job migrations. The authors thank Bill Bynum for reading this paper and his suggestions. Yanxia Qu participated in the work of the load sharing project in the early stage. This work is also part of an independent research project sponsored by the NSF for program directors and visiting scientists. Finally, they are grateful to the five anonymous reviewers for their insightful and constructive comments/critiques.

References

[1] A. Acharya and S. Setia, "Availability and Utility of Idle Memory in Workstation Clusters," *Proc. ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 35-46, May 1999.

[2] Y. Amir, B. Awerbuch, A. Barak, R.S. Borgstrom, and A. Keren, "An Opportunity Cost Approach for Job Assignment and Reassignment in a Scalable Computing Cluster," *IEEE Trans. Parallel and Distributed Systems,* vol. 11, no. 7, pp. 760-768, 2000.

[3] A. Barak and A. Braverman, "Memory Ushering in a Scalable Computing Cluster," *J. Microprocessors and Microsystems,* vol. 22, no. 3-4, pp. 175-182, Aug. 1998.

[4] A. Batat and D.G. Feitelson, "Gang Scheduling with Memory Considerations," *Proc. 14th Int'l Parallel and Distributed Processing Symp. (IPDPS '2000),* pp. 109-114, May 2000.

[5] S. Chen, L. Xiao, and X. Zhang, "Dynamic Load Sharing with Unknown Memory Demands of Jobs in Clusters," *Proc. 21st Int'l Conf. Distributed Computing Systems, (ICDCS '2001),* pp. 109-118, Apr. 2001.

[6] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software Practice and Experience,* vol. 21, no. 8, pp. 757-785, 1991.

[7] X. Du and X. Zhang, "Coordinating Parallel Processes on Networks of Workstations," *J. Parallel and Distributed Computing,* vol. 46, no. 2, pp. 125-135, 1997.

[8] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "The Limited Performance Benefits of Migrating Active Processes for Load Sharing," *Proc. ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 63-72, May 1988.

[9] M.J. Feeley et al., "Implementing Global Memory Management Systems," *Proc. 15th ACM Symp. Operating System Principles,* pp. 201-212, Dec. 1995.

[10] D. Feitelson, "The Parallel Workload Archive," http://www. cs. huji.ac.il/labs/parallel/workload/logs. html\#lanlcm5, 1998.

[11] 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,* pp. 337-360, 1995.

[12] M.D. Flouris and E.P. Markatos, "Network RAM," *High Performance Cluster Computing,* Chapter 16, R. Buyya, ed., vol. 1, pp. 383-508, New Jersey: Prentice Hall 1999.

[13] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," *Proc. ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 115-126, May 1997.
[14] M. Harchol-Balter and A.B. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *ACM Trans. Computer Systems,* vol. 15, no. 3, pp. 253-285, 1997.
[15] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* second ed., Morgan Kaufmann, 1996.
[16] C.-C. Hui and S.T. Chanson, "Improved Strategies for Dynamic Load Sharing," *IEEE Concurrency,* vol. 7, no. 3, pp. 58-67, 1999.
[17] URL: http://www.cs.umn.edu/~metis, 1998.
[18] T. Kunz, "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme," *IEEE Trans. Software Eng.,* vol. 17, no. 7, pp. 725-730, 1991.
[19] W. Lin, S.K. Reinhardt, and D. Burger, "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture, (HPCA-7),* pp. 301-312, Jan. 2001.
[20] K.-L. Ma and T.W. Crockett, "A Scalable, Cell-Projection Volume Rendering Algorithm for 3D Unstructured Data," *Proc. Parallel Rendering '97 Symp.,* pp. 95-104, Oct. 1997.
[21] E.P. Markatos and G. Dramitinos, "Implementation of a Reliable Remote Memory Pager," *Proc. 1996 Usenix Technical Conf.,* pp. 177-190, Jan. 1996.
[22] URL: http://www.netlib.org, 1985.
[23] V.G. Peris, M.S. Squillante, and V.K. Naik, "Analysis of the Impact of Memory in Distributed Parallel Processing Systems," *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems,* pp. 5-18, May 1994.
[24] A. Silberschatz and P.B. Galvin, *Operating Systems Concepts,* Fourth ed. Addison-Wesley, 1994.
[25] M.S. Squillante, D.D. Yao, and L. Zhang, "Analysis of Job Arrival Patterns and Parallel Scheduling Performance," *Performance Evaluation,* vol. 36-37, pp. 137-163, 1999.
[26] G.M. Voelker, H.A. Jamrozik, M.K. Vernon, H.M. Levy, and E.D. Lazowska, "Managing Server Load in Global Memory Systems," *Proc. ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 127-138, May 1997.
[27] L. Xiao, X. Zhang, and S.A. Kubricht, "Improving Memory Performance of Sorting Algorithms," *ACM J. Experimental Algorithmics,* vol. 5, pp. 1-23, 2000.
[28] L. Xiao, X. Zhang, and Y. Qu, "Effective Load Sharing on Heterogeneous Networks of Workstations," *Proc. 2000 Int'l Parallel and Distributed Processing Symp., (IPDPS'2000),* pp. 431-438, May 2000.
[29] X. Zhang, Y. Qu, and L. Xiao, "Improving Distributed Workload Performance by Sharing Both CPU and Memory Resources," *Proc. 20th Int'l Conf. Distributed Computing Systems, (ICDCS '2000),* pp. 233-241, Apr. 2000.
[30] Z. Zhang and X. Zhang, "Fast Bit-Reversals on Uniprocessors and SMP Multiprocessors," *SIAM J. Scientific Computing,* vol. 22, no. 6, pp. 2113-2134, 2001.
[31] Z. Zhang, Z. Zhu, and X. Zhang, "Cached DRAM for ILP Processor Memory Access Latency Reduction," *IEEE Micro,* vol. 21, no. 4, pp. 22-32, July/Aug. 2001.

Li Xiao received the BS and MS degrees in computer science from Northwestern Polytechnical University, China. She is a PhD candidate of computer science at the College of William and Mary. She is a recipient of USENIX Fellowship for PhD dissertation research from 2001 to 2002. She was a research intern at the Hewlett Packard Labs in the summer of 2001, working on a peer-to-peer Internet computing project. Her research interests are in the areas of distributed and Internet systems, system resource management, and design and implementations of experimental algorithms. She is a student member of the IEEE.

Songqing Chen received the BS and MS degrees in computer science from Huazhong University of Science and Technology, China, in 1997 and 1999, respectively. He is a PhD candidate of computer science at the College of William and Mary. His research interests are distributed and Internet systems, and kernel system programming.

Xiaodong Zhang received the BS in electrical engineering from Beijing Polytechnic University, China, in 1982, and the MS and PhD degrees in computer science from University of Colorado at Boulder, in 1985 and 1989, respectively. He is a professor of computer science at the College of William and Mary. He is also the program director of the Advanced Computational Research Program (ACR) at the US National Science Foundation. His research interests include parallel and distributed systems, computer architecture, and scientific computing. He is associate editor of *IEEE Micro* and *IEEE Transactions on Parallel and Distributed Systems*. He is a senior member of the IEEE.