Hindawi Publishing Corporation
Journal of Computer Systems, Networks, and Communications
Volume 2008, Article ID 294106, 9 pages
doi:10.1155/2008/294106

Research Article

Generalized Load Sharing for Homogeneous Networks of Distributed Environment

A. Satheesh,[1] K. Vimal Kumar,[1,2] and S. Krishnaveni[3]

[1] *Department of Computer Science and Engineering, Periyar Maniammai University, Vallam Thanjavur 613403, India*
[2] *Department of Computer Science and Engineering, SRM Institute of Management and Technology, New Delhi 201204, India*
[3] *Department of Computer Applications, Periyar Maniammai University, Vallam Thanjavur 613403, India*

Correspondence should be addressed to A. Satheesh, vbsatheesh@yahoo.com

Received 16 January 2008; Revised 9 March 2008; Accepted 11 April 2008

Recommended by Chadi Assi

We propose a method for job migration policies by considering effective usage of global memory in addition to CPU load sharing in distributed systems. When a node is identified for lacking sufficient memory space to serve jobs, one or more jobs of the node will be migrated to remote nodes with low memory allocations. If the memory space is sufficiently large, the jobs will be scheduled by a CPU-based load sharing policy. Following the principle of sharing both CPU and memory resources, we present several load sharing alternatives. Our objective is to reduce the number of page faults caused by unbalanced memory allocations for jobs among distributed nodes, so that overall performance of a distributed system can be significantly improved. We have conducted trace-driven simulations to compare CPU-based load sharing policies with our policies. We show that our load sharing policies not only improve performance of memory bound jobs, but also maintain the same load sharing quality as the CPU-based policies for CPU-bound jobs. Regarding remote execution and preemptive migration strategies, our experiments indicate that a strategy selection in load sharing is dependent on the amount of memory demand of jobs, remote execution is more effective for memory-bound jobs, and preemptive migration is more effective for CPU-bound jobs. Our CPU-memory-based policy using either high performance or high throughput approach and using the remote execution strategy performs the best for both CPU-bound and memory-bound job in homogeneous networks of distributed environment.

1. INTRODUCTION

A major performance objective of implementing a load sharing policy in a distributed 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., [1–5]) mainly consider CPU load balancing by assuming that each computer 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 distributed system become more and more expensive relative to CPU cycles. We believe that the overheads of data accesses and movement, such as page faults, have grown to the point where the overall performance of distributed systems would be considerably degraded without serious considerations concerning memory resources in the design of load sharing policies. We have the following reasons to support our claim. First, with the rapid development of reduced instruction set computer (RISC) and very large scale integration (VLSI) technology, the speed of processors has increased dramatically in the past decade. We have seen an increasing gap in speed between processor and memory, and this gap makes performance of application programs on uniprocessor, multiprocessor, and distributed systems rely more and more on effective usage of their entire memory hierarchies. 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 distributed systems has significantly increased accordingly with the rapid establishment of local and wide-area internet infrastructure. Third, the latency of a memory miss or a page fault is about 1000 times higher than that of

a memory hit. Therefore, minimizing page faults through memory load sharing has a great potential to significantly improve overall performance of distributed systems. Finally, it has been shown that memory utilizations among the different nodes in a distributed 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 [7]. Our objective of a new load sharing policy design is to share both CPU and memory services among the nodes in order to minimize both CPU idle times and the number of page faults in distributed systems.

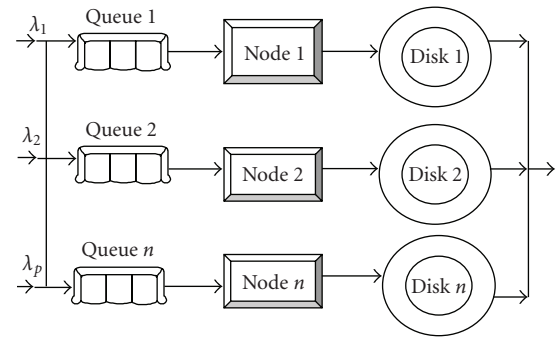

Figure 1: A Structure of queuing model.

1.1. Organization of the paper

This paper is organized as follows. Section 2 reviews CPU-based and memory-based load sharing policies as the background. Section 3 introduces the generalized load sharing model. Section 4 explains the simulator implementation. Section 5 conducts experiments with different configurations. Section 6 concludes and discusses some possible extensions to our work.

2. BACKGROUND

Besides the cited work on CPU-based load sharing policies in the previous subsection, some work has been reported on memory resource considerations of load sharing. An early study in [6] considers using the free memory size in each node as an index for load sharing. Compared with CPU-based policies, this study did not find the memory-based policy particularly effective. This is because the workloads were CPU intensive, and the processors then were much slower than what we are using today. The global memory system (GMS) [8, 9] 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 [7] 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 MB) [10]. 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 algorithms design and evaluation between the memory ushering and our proposed CPU-memory-based load sharing. (1) Instead of only considering memory load in the selection of a remote node, we consider both CPU and memory loads. (2) Besides preemptive migrations, we have also evaluated remote executions in our algorithms, and found that remote executions could be more beneficial to memory-intensive jobs. (3) Instead of using an exponential distribution for job memory demands, we use a Pareto distribution. There are two major challenges in designing load sharing policies that consider both CPU and memory resources. First, a load sharing for CPU and a load sharing for memory may have conflicting interests. Second, the additional cost involved with a job migration for load sharing is non-trivial. Therefore, a migration decision must be beneficial to performance improvement. In this study, we address the following question: under what conditions is a load sharing policy considering memory resources necessary for performance improvement. These conditions are important additions in our new load sharing policy design. Using a simple queuing model, we first demonstrate the significance of considering both CPU and memory resources in load sharing policy design. We have conducted trace-driven simulations to compare load sharing policies for CPU load balancing and our policies for both CPU and memory resource sharing. We show that the overall performance of a distributed system and individual jobs on the system can be significantly improved further by considering memory resources as an additional and important factor in load sharing policies. We also show that our new load sharing policies not only improve performance of memory bound jobs, but also maintain the same load sharing quality as the CPU-based policies for CPU-bound jobs. Regarding remote execution and preemptive migration strategies, our experiments indicate that a strategy selection in load sharing is dependent on the amount of memory demand of jobs; remote execution is more effective for memory-bound jobs, and preemptive migration is more effective for CPU-bound jobs.

3. GENERALIZED LOAD SHARING MODEL

3.1. Structure of queuing model

We use a queuing network model to characterize load sharing in a distributed system. By doing so, we can compare the performance effects of a load sharing system only considering CPU resources with a load sharing system considering both CPU and memory resources. We can also show the performance benefits of sharing both CPU and memory resources.

The model shown in Figure 1 is designed to measure the total response time of a distributed system. For a job arriving in a node, the load sharing system may send the job directly to the local node or transfer it to another node depending on

the load status of nodes in the system. The transferred job will either execute on the destination node immediately or wait for its turn in the queue. Parameters λ_i, for $i = 1, \ldots, P$, represent the arrival rates of jobs to each of P nodes in the system (measured by the number of jobs per second). In order to compare the two load sharing policies, the system assumes that all nodes have an identical computing capability, but have different memory capacities. This means that the service demand of a job on each computing node is the same, and service demands on disks are different, because the same job may cause more page faults on a node with a small memory size than on a node with a large memory size. In a load sharing system only considering CPU resources, the arrival rates $\lambda_1, \lambda_2, \ldots, \lambda_P$ will be identical after a certain period of execution time because all nodes have the same computing capability. In a load sharing system considering both CPU and memory resources, the arrival rates will be different. Without loss of generality, we apply formulas for open models [11] to solve this model with $P = 2$ in order to predict the response time. Table 1 gives the parameters that characterize the workload we have used to solve the model for the two load sharing policies. Besides arrival rates, the system has two other parameters, $D_{\text{cpu}}(i)$ and $D_{\text{disk}}(i)$, which are average service demands of the CPU and the disk, respectively, at computing node i for $i = 1, \ldots, P$ (measured in seconds). The parameter values in Table 1 are collected from our modified trace-driven simulations, which will be presented in Section 4. The average disk service demands of node 1 ($D_{\text{disk}}(1) = 0.10$ seconds) is 10 times larger than that of node 2 ($D_{\text{disk}}(1) = 0.01$ seconds), which means that node 1 has a smaller memory capacity to cause more page faults. Results in Table 2 show that a load sharing policy considering both CPU and memory resources is effective in reducing the response time. For example, the response time is 2.50 seconds by the load sharing policy mainly considering CPU resources, which gives 50% of the jobs to each node. The response time could be reduced to 1.43 seconds by just slightly changing the job distribution rate in each node, distributing 45% of the jobs to node 1 with a small memory capacity, and 55% to node 2 with a large memory capacity.

This example shows that performance is highly sensitive to the average access rates λ_i, $i = 1, \ldots, P$, which are determined by a load sharing policy. The simple model also indicates that potential performance gain is high by considering both CPU and memory resources in load sharing. On the other hand, this model lacks the ability to characterize dynamic scheduling on practical workloads. In order to address this limit, the system has conduct trace-driven simulations to give a much more insightful evaluation of different load sharing policies in Sections 4 and 5, which will further show the performance benefits.

3.2. Metrics

We evaluate the effectiveness of each strategy according to the following performance metrics.

Mean slowdown. Slowdown is the ratio of wall-clock execution time to CPU time (thus, it is always greater than one). The average slowdown of all jobs is a common metric of system performance. When we compute the ratio of mean slowdowns (as from different strategies) we will use normalized slowdown, which is the ratio of inactive time (the excess slowdown caused by queuing and migration delays), to CPU time. For example, if the (unnormalized) mean slowdown drops from 2.0 to 1.5, the ratio of normalized mean slowdown is 0.5/1.0 = 0.5, and thus it is more meaningful to report a 50% reduction in delay than a 25% improvement in performance.

Mean slowdown alone, however, is not a sufficient measure of the difference in performance of the two strategies; it understates the advantages of the preemptive strategy for these two reasons.

(1) Skewed distribution of slowdowns: even in the absence of migration, the majority of processes suffer small slowdowns (typically 80% are less than 3.0). This majority will dominate the value of the mean slowdown.

(2) User perception: from the user point of view, the important processes are the ones in the tail of the distribution, because although they are the minority, they cause the most noticeable and annoying delays. Eliminating these delays might have a small effect on themean slowdown, but a large effect on the user perception of performance. Therefore, we will also consider the following two metrics.

Variance of slowdown. This metric is often cited as a measure of the unpredictability of response time, which is a nuisance for users trying to schedule tasks. In light of the distribution of slowdowns, however, it may be more meaningful to interpret this metric as a measure of the length of the tail of the distribution; that is, the number of jobs that experience long delays.

Number of severely slowed processes. In order to quantify the number of noticeable delays explicitly, we consider the number (or percentage) of processes that are severely impacted by queuing and migration penalties.

For the sake of simplicity, we assume that processes are always ready to run (i.e., are never blocked on I/O). During

Table 1: Parameters of the queuing model.

Policies	Node number (i)	λ_i	$D_{\text{cpu}}(i)$	$D_{\text{disk}}(i)$
CPU	1	1.92	0.39	0.10
	2	1.92	0.39	0.01
CPU & memory	1	1.73	0.39	0.10
	2	2.11	0.39	0.01

Table 2: Results of the queuing model.

Policies	$\lambda_1 : \lambda_2$	Response time (seconds)
CPU	50 : 50	2.50
CPU and memory	45 : 55	1.43

a given time interval, we divide CPU time equally among the processes on the host.

3.3. CPU-memory load sharing policy

A commonly used load sharing index is CPU based, which uses the length of a CPU waiting queue (the number of jobs), denoted as L, to determine the load status of a computing node. A CPU threshold, denoted as CT, is the maximum number of jobs the CPU is willing to take, which is normally set based on the CPU computing capability. If the waiting queue is shorter than the CPU threshold, a requesting job is executed locally, otherwise, the load sharing system will find a remote node with the shortest waiting queue to either remotely execute this job, or find such a node for an eligible job to do a preemptive migration from the local node to the remote node. The load sharing policy (LS) based on the CPU-based load index is expressed in the following form:

$$\mathrm{LS}(L) = \begin{cases} \text{local execution}, & L < \mathrm{CT}, \\ \text{migration}, & L \geq \mathrm{CT}, \end{cases} \tag{1}$$

where migration is either a remote execution or a preemptive migration.

In our new load sharing policy design [9], the CPU queue length, L, is still used as the load index. However, this index is also a function of the memory threshold. The checking result of MT < RAM means that the memory space of a system is sufficiently large for the jobs, while MT ≥ RAM means that the memory space is overloaded. A new design option combines the resources of CPU cycles and memory space to a single load index, and is defined as

$$\mathrm{Index}_{\mathrm{hp}}(L, \mathrm{MT}) = \begin{cases} L, & \mathrm{MT} < \mathrm{RAM}, \\ \mathrm{CT}, & \mathrm{MT} \geq \mathrm{RAM}. \end{cases} \tag{2}$$

When MT < RAM, CPU-based load sharing is used. When MT ≥ RAM, the CPU queue length or the load index is set to CT as if CPU were overloaded so that the system refuses to accept jobs or migrates jobs to a lightly loaded computing node. But the real reason comes from the overloaded memory allocation. In our implementation, when MT ≥ RAM, the local scheduler immediately searches for the most lightly loaded node in the system as the job destination. Since the load index is set to CT when MT > RAM, it may not allow a computing 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 a principle of high performance is *computing*. That is the reason we define this option as a high performance computing load index, defined as $\mathrm{Index}_{\mathrm{hp}}$. However, it may not be in favor of *high throughput computing* which emphasizes on effective management and exploitation ofall available computing nodes. For example, when MT ≥ RAM on one node, this condition may be true in several computing nodes. If the load indices in many nodes have been set to CT and consequently they may refuse to accept jobs, the amount of computing 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, we conservatively adjust the load index by a memory utilization status parameter when MT ≥ RAM. The memory utilization parameter is defined as

$$\gamma = \frac{U}{\mathrm{RAM}}. \tag{3}$$

When $\gamma < 1$, it means that the memory space of the computing node is sufficiently large for the jobs. When $\gamma > 1$, it means that the memory system is overloaded. This option is designed for high-throughput computing, and its load index is defined as follows:

$$\mathrm{Index}_{\mathrm{ht}}(L, \mathrm{MT}) = \begin{cases} L, & \mathrm{MT} < \mathrm{RAM}, \\ L \times \gamma, & \mathrm{MT} \geq \mathrm{RAM}. \end{cases} \tag{4}$$

Memory utilization parameter is used to proportionally adjust the load index. When MT > RAM, the CPU queue length is enlarged by a factor of as if the CPU queue length is enlarged by a factor of γ as if the CPU were increasingly loaded. The increase of the load index would reduce the chance of this computing 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:

$$\mathrm{LS}(\mathrm{Index}) = \begin{cases} \text{local execution}, & \mathrm{Index} < \mathrm{CT}, \\ \text{migration}, & \mathrm{Index} \geq \mathrm{CT}, \end{cases} \tag{5}$$

where Index is either $\mathrm{Index}_{\mathrm{hp}}$or $\mathrm{Index}_{\mathrm{ht}}$.

Beside comparing our policies with the CPU-based load sharing policy, we have also compared with a memory-basedload sharing policy where the load index is represented by the memory threshold, MT. For a new job requesting service in a computing node, if the node memory threshold is smaller than the user memory space, the job is executed locally, otherwise, the load sharing system will find a remote node with the lightest memory load to either remotely execute this job, or find such a node for an eligible job to do a preemptive migration from the local node to the remote node. The load sharing policy based on the memory-based load index is expressed in the following form:

$$\mathrm{LS}(\mathrm{MT}) = \begin{cases} \text{local execution}, & \mathrm{MT} < \mathrm{RAM}, \\ \text{migration}, & \mathrm{MT} \geq \mathrm{RAM}. \end{cases} \tag{6}$$

4. IMPLEMENTATION

Our performance evaluation is simulation based, consisting of two major components: a simulated distributed system and workloads. We will discuss our simulation model, system conditions, and the workloads in this section.

Harchol-Balter and Downey [4] developed a simulator of a homogeneous distributed system with 6 nodes, where

each local scheduler is CPU-based only. In this simulator we have included the packet-based WFR traffic splitter algorithm. Using this simulator as a framework, we simulated a homogeneous distributed system with 125 nodes, where each local scheduler holds all the load-sharing policies we have discussed in this paper: CPU-based, memory-based, CPU-memory-based, and their variations. The simulated system is currently configured with the following parameters.

CPU speed. 1000 MIPS (million instructions per second). RAM size: 450 MB (The memory size is 512 MB including 62 MB for kernel and file cache usage [3]). Memory page size: 40 KBytes. Ethernet connection: 100 Mbps. page fault service time: 10 ms. time slice of CPU local scheduling: 10 ms. context switch time: 0.1 ms. The parameter values are similar to the ones of a Sun Ultra-40 M2 workstation. The CPU local scheduling uses the round-robin policy. Each program job is in one of the following states: "ready," "execution," "paging," "data transferring," or "finish." When a page fault happens in an execution of a job, 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 migration related costs match the Ethernet network service times for Sun-Ultra 40 M2 workstations, and they are also used in [4]:

(i) a remote execution cost: 0.1 second,

(ii) a preemptive migration cost: $0.1 + D/B$,

where 0.1 is a fixed remote execution cost in second, and D is the data in bits to be transferred in the job migration, and B is the network bandwidth in Mbps ($B = 10$ for the Ethernet).

4.1. System conditions

The system has following conditions and assumptions for evaluating the load sharing policies in the distributed system.

Each computing node maintains a global load index file which contains both CPU and memory load status information of other computing nodes. The load sharing system periodically collects and distributes the load information among the computing nodes. The location policy determines which computing node to be selected for a job execution. The policy of this paper is use to find the most lightly loaded node in the distributed system. The system assumes that the memory allocation for a job is done at the arrival of the job. Similar to the assumptions in [2, 3], the system assumes that page faults are uniformly distributed during job executions. The system assumes that the memory threshold of a job is 40% of its requested memory size. The practical value of this threshold assumption has also been confirmed by the studies in [2, 3].

4.2. Workloads

From the workloads, the system has used 18 traces from [4]. Each trace was collected from one workstation on different daytime intervals. The jobs in each trace were distributed among 6 homogeneous workstations. The traces were used for CPU-based scheduling, and the requested memory space allocation of each job is independent of the job service time. Each job has the following three major parameters:

$$\langle \text{arrival time, arrival node, duration time} \rangle. \quad (7)$$

The memory space allocations of jobs are generated in a Pareto distribution for preemptive migration, and the mean memory demand for each trace is 1 MB. The 8 traces have different interarrival distributions and different Pareto service time distributions.

The system has done the following modifications of the traces for our systems. This paper converted the job duration time into million instructions according the CPU speed. The memory demand of each job in the traces is generated from a Pareto distribution with the mean sizes of 11 MB, 12 MB, 13 MB, and 14 MB. An informal study in [5, 6] shows that a Pareto distribution is reasonable for simulating memory demand. Each job has the following 4 items: arrival time, arrival node, requested memory size, and duration time.

The page faults in each node are conducted in our simulation as follows. When the memory threshold of jobs in a node is equal to or larger than the available memory space (MT $\geq$ RAM), each job in the node will cause page faults at a given page fault rate. The page fault rates of jobs range from 0 to 4.1 per million instructions in our experiments. In practice, application jobs have page fault rates from 1 to 10.

5. EXPERIMENTAL RESULTS AND ANALYSIS

When a job migration is necessary, 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 make the selected jobs be suspended, be moved to a remote node, and then be restarted. This paper includes the two options in our load sharing experiments to study the merits and their limits [11, 12].

Using the trace-simulation on a 6 node distributed system, the system has evaluated the performance of the following load sharing policies:

(i) CPU RE: CPU-based load sharing policy with remote execution [4];

(ii) CPU PM: CPU-based load sharing policy with preemptive migration [4];

(iii) MEM RE: memory-based load sharing policy with remote execution;

(iv) MEM PM: memory-based load sharing policy with preemptive migration;

(v) CPU MEM HP RE: CPU-Memory-based load sharing policy using the high performance computing oriented load index with remote execution;

(vi) CPU MEM HP PM: CPU-memory-based load sharing policy using the high performance computing oriented load index with preemptive migration;

(vii) CPU MEM HT RE: CPU-memory-based load sharing policy using the high-throughput-oriented load index with remote execution;

(viii) CPU MEM HT PM: CPU-memory-based load sharing policy using the high-throughput-oriented load index with preemptive migration.

In addition, the system has also compared execution performance of the above policies with the execution performance without using load sharing, denoted as NO_LS.

A major timing measurement the system has used is the mean *slowdown*, which is the ratio between the total wall-clock executiontime of all the jobs in a trace and their total CPU execution time. Major contributions to the slowdown come from the delays of page faults, waiting time for CPU service, and the overhead of migration and remote execution.

5.1. Performance comparisons

This paper has experimentally evaluated the 8 load sharing policies plus the NO_LS policy. The upper limit vertical axis presenting the mean slowdown is set to 20. It scaled the page fault rate accordingly. The mean memory demand for each trace at different page fault rates was set to 11 MB, 12 MB, 13 MB, and 14 MB.

The performance comparisons of the load sharing policies on other traces are consistent with what the system has presented for "trace 0" in principle. Figures 2 and 3 present the mean slowdowns of all the traces ("trace 0",..., "trace 7") scheduled by different load sharing policies with the mean memory demand of 11 MB and 14 MB, respectively.

For the mean memory demand of 11 MB, the system has adjusted the page fault rate for each trace in order to obtain reasonably balanced slowdown heights of 20 among all the traces in Figure 2. We have $\sigma = 2.40$ for "trace 0," $\sigma = 0.67$ for "trace 1," $\sigma = 0.67$ for "trace 2," $\sigma = 4.10$ for "trace 3," $\sigma = 0.45$ for "trace 4," $\sigma = 2.80$ for "trace 5," $\sigma = 2.30$ for "trace 6," and $\sigma = 2.30$ for "trace 7." The NO_LS policy had the highest slowdown value for all the traces. Policies MEM_RE, CPU_MEM_HP_RE, and CPU_MEM_HT_RE perform the best for all the traces.

For the mean memory demand of 4 MB, The system also adjusted the page fault rate for each trace for the same reason as in the previous experiment. In Figure 2, the system has $\sigma = 0.6$ for "trace 0," $\sigma = 0.16$ for "trace 1," $\sigma = 0.11$ for "trace 2," $\sigma = 0.94$ for "trace 3," $\sigma = 0.11$ for "trace 4," $\sigma = 0.17$ for "trace 5," $\sigma = 0.58$ for "trace 6," and $\sigma = 0.58$ for "trace 7." Again, the NO_LS policy had the highest slowdown value for all the traces. Policies MEM_RE, CPU_MEM_HP_RE, and CPU_MEM_HT_RE still perform the best for all the traces. In addition, due to high memory demand, the remote execution strategy (RE) is more effective than the preemptive migration strategy (PM).

5.2. Preemptive migration versus remote execution

We further look into the performance comparisons among all newly designed load sharing policies NO_LS, CPU_RE, and CPU_PM. The studies in [4] indicate that preemptive strategy is far more effective than remote execution for the CPU-based policies on CPU-bound jobs. We have confirmed this result in our simulation. We compare the performance of

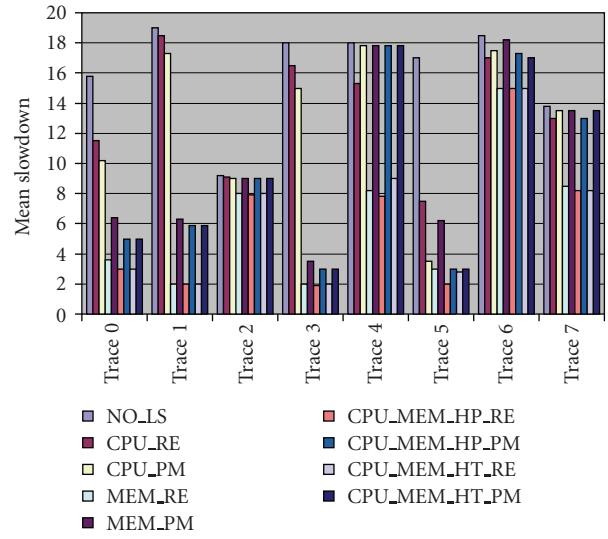

Figure 2: Mean slowdowns of all the 8 traces scheduled by different load sharing policies with the mean memory demand of 11 MB.

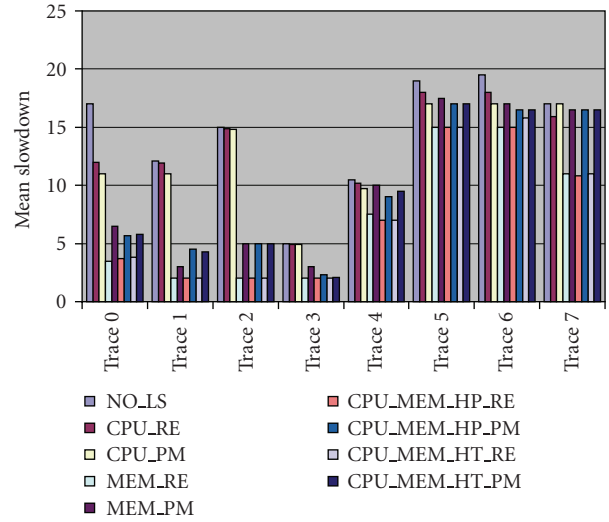

Figure 3: Mean slowdowns of all the 8 traces scheduled by different load sharing policies with the mean memory demand of 14 MB.

all the newly designed policies with CPU_PM by calculating the speedups of the new policies:

$$\text{Speedup} = \frac{\text{slowdown of CPU\_PM}}{\text{slowdown of a new policy}}. \quad (8)$$

Figures 4 and 5 show the speedups. When the memory demand is low, not only are the speedups of memory-based policies slightly lower than 1, but also are the other policies using remote execution strategies. All our experiments show that the memory-based policies are not beneficial to CPU-bound jobs. However, when the memory demand is high, the memory consideration is effective, and the remote execution strategies perform better than preemptive migrations. For example, CPU_MEM_HP_RE achieves the highest speedup of

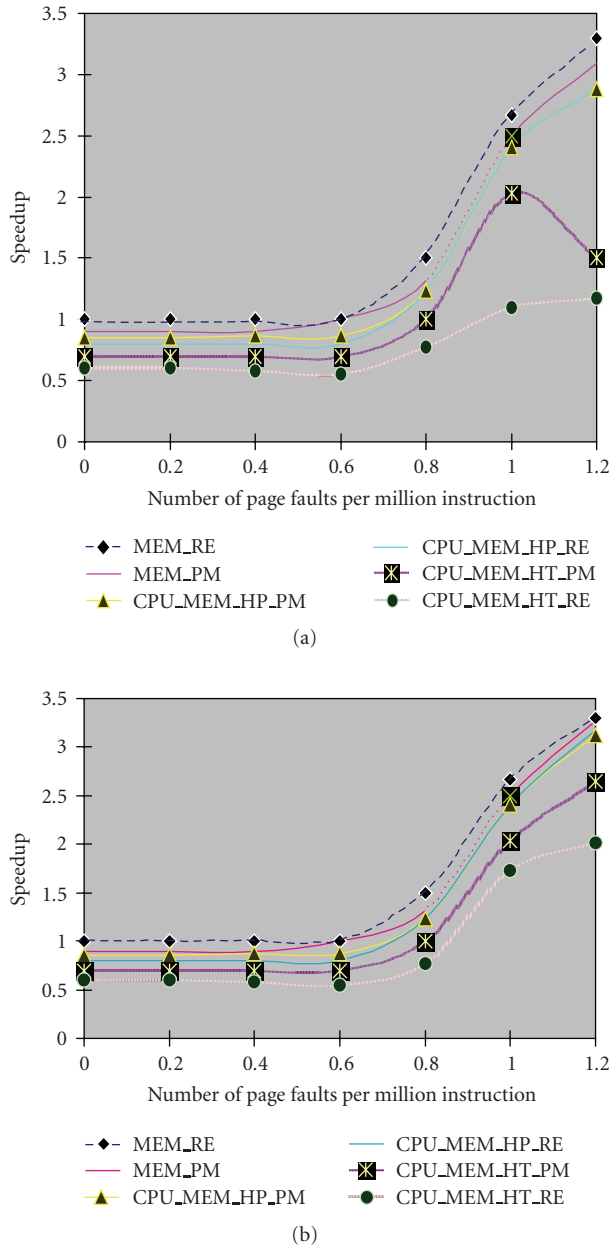

Figure 4: Speedups of the new load sharing policies over CPU_PM as the page fault rate increases on "trace 0" with mean memory demand of 11 MB (a) and 12 MB (b).

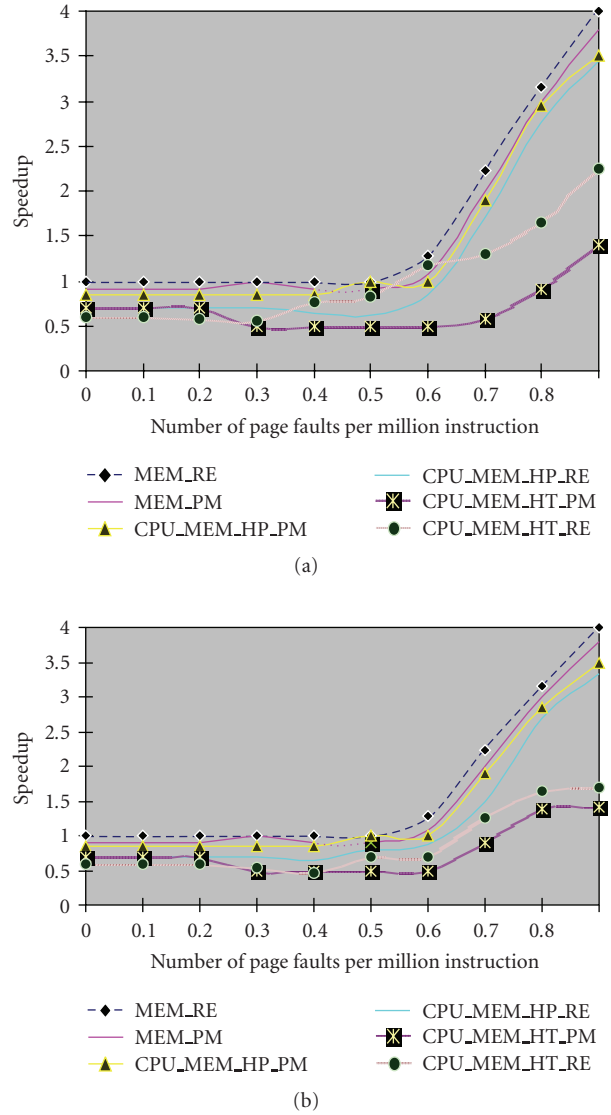

Figure 5: Speedups of the new load sharing policies over CPU PM as the page fault rate increases on "trace 0" with mean memory demand of 13 MB (a) and 14 MB (b).

3.6 for the mean memory demand of 4 MB at the page rate of 0.6, while CPU_MEM_HP_PM only achieves the speedup of 1.9 under the same condition. This is because remote execution has significantly lower data movement cost than that of preemptive execution for jobs with high memory demand.

5.3. High performance versus high throughput

Consider that remote execution is a more effective strategy than preemptive migration for scheduling jobs with high memory demands, we chose remote execution to further compare the high performance (HP) approach and the high throughput (HT) approach in our load sharing policies. We compared the speedups of CPU_MEM_HP_RE and CPU_MEM_HT_RE over CPU_PM. Figure 6 shows the speedups of the two load sharing policies on "trace 0." Our experiments show that when the memory demand is low (low page fault rate and low mean memory demand), the performance of CPU_MEM_HP_RE and the performance of CPU_PM are comparable (the speedup is close to 1.0). However, when the memory demand is high, the speedup of CPU_MEM_HP_RE over CPU_PM is high. For example, the speedup of CPU_MEM_HP_RE over CPU_PM is about 3.6 when the mean memory demand is 4 MB and the page rate is 0.6, which is the highest among the speedups achieved by other policies. We have shown that the high-

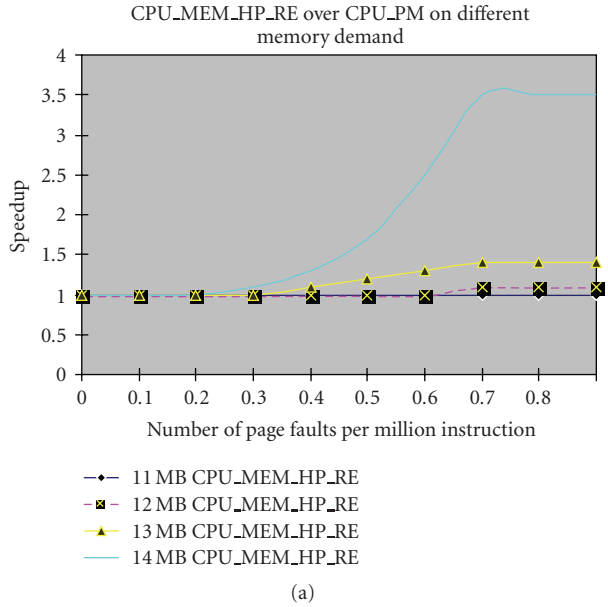

Figure 6: Speedups of our load sharing policies of remote execution over CPU PM as both the page fault rate and mean memory demand increase on "trace 0" for the high performance approach (CPU MEM HP RE, (a)) and for the high-throughput approach (CPU MEM HT RE, (b)).

performance approach is beneficial to jobs with both low and high memory demands.

We have two observations of the high throughput approach in load sharing from Figure 6. First, when the memory demand is low (low page fault rate and low mean memory demand), the performance of CPU_MEM_HT_RE is over 10% worse than that of CPU_PM (the speedup is less than 0.9). There are two reasons for this: (1) the high-throughput approach could cause slightly more page faults for jobs with low memory demand, because the approach encourages more jobs to be executed concurrently in the distributed system, and (2) a preemptive migration strategy

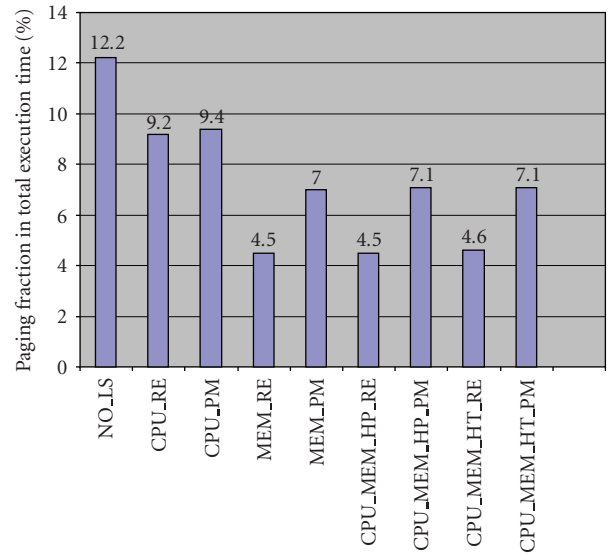

Figure 7: Paging time fractions in the total execution times for all the policies on "trace 0" with the mean memory demand of 14 MB at page fault of 0.6.

is more effective than remote execution strategy for jobs with low memory demand. Second, although CPU_MEM_HT_RE significantly overperforms CPU_PM when the memory demand is high (e.g., the speedup is 3.3 at the page rate of 0.6 with the mean memory demand of 4 MB), the performance gain is not as high as of CPU_MEM_HP_RE. For example, the highest speedup is 3.6 for CPU_MEM_HP_RE, while the highest speedup is 3.3 for CPU_MEM_HT_RE. The only reason for this is that the high-throughput approach would cause more page faults than that of high-performance approach when the memory demand is high.

5.4. *Effectiveness of memory load sharing*

In order to show a correlation between paging time reduction by effective memory load sharing and its consequent reduction of the total execution time, in Figure 7, we plot paging time fractions in the total execution times of the 9 policies for "trace 0" with the mean memory demand of 14 MB at page fault rate of 0.6. Compared with Figure 5, we can clearly see that a small paging time reduction by a load sharing policy could result in a significant reduction of the total execution time. For example, the paging fraction is 12.2% without load sharing (NO LS), while the paging fraction is reduced to 4.5 to 4.6% by policies MEM_RE, CPU_MEM_HP_RE, and CPU_MEM_HT_RE, which further causes the total execution time to be reduced by about 5.1 times. As the paging time is reduced, other memory-related stall times, such as CPU waiting time, I/O service time, and network service time, are also reduced.

6. CONCLUSION AND FUTURE WORK

We have experimentally examined and compared a CPU-based, a memory-based, and a CPU-memory-based load

sharing policies on homogeneous networks of distributed systems. Based on our experiments and analysis, this paper has the following observations and conclusions. A load sharing policy considering only CPU or only memory resource would be beneficial either to CPU-bound or to memory-bound jobs.

Only a load sharing policy considering both resources will be beneficial to jobs of both types. Our trace-driven simulations show that CPU-memory-based policies with a remote execution strategy are more effective than the policies with a preemptive migration for memory-bound jobs, but the opposite is true for CPU-bound jobs. This result may not be applied to general cases because this paper did not consider other data access patterns so that the system is not able to apply any optimizations to preemptive migrations. High-performance approach is slightly more effective than high-throughput approach for both memory-bound and CPU-bound jobs. The future work of this paper can go in different directions. Using both load sharing policies of CPU_MEM_HT_PM, and CPU_MEM_HP_PM in Open MOSIX or KERRIGHED distributed operating system will improve the workload performance.

REFERENCES

[1] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 63–72, Santa Fe, NM, USA, May 1988.

[2] G. Glass and P. Cao, "Adaptive page replacement based on memory reference behavior," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 115–126, Seattle, Wash, USA, June 1997.

[3] G. M. Voelker, H. A. Jamrozik, M. K. Vernon, H. M. Levy, and E. D. Lazowska, "Managing server load in global memory systems," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 127–138, Seattle, Wash, USA, June 1997.

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

[5] S. Zhou, "A trace-driven simulation study of dynamic load balancing," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1327–1341, 1988.

[6] S. Zhou, J. Zheng, X. Wang, and P. Delisle, "Utopia: a load sharing facility for large, heterogeneous distributed computer systems," *Software: Practice and Experience*, vol. 23, no. 12, pp. 1305–1336, 1993.

[7] A. Acharya and S. Setia, "Availability and utility of idle memory in workstation clusters," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 35–46, Atlanta, Ga, USA, May 1999.

[8] K.-C. Leung and V. O. K. Li, "Generalized load sharing for packet-switching networks I: theory and packet-based algorithm," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 694–702, 2006.

[9] X. Zhang, Y. Qu, and L. Xiao, "Improving distributed workload performance by sharing both CPU and memory resources," in *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS '00)*, pp. 233–241, Taipei, Taiwan, April 2000.

[10] A. Barak and A. Braverman, "Memory ushering in a scalable computing cluster," *Microprocessors and Microsystems*, vol. 22, no. 3-4, pp. 175–182, 1998.

[11] T. Kunz, "The influence of different workload descriptions on a heuristic load balancing scheme," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 725–730, 1991.

[12] X. Hesselbach, R. Fabregat, B. Baran, Y. Donoso, F. Solano, and M. Huerta, "Hashing based traffic partitioning in a multicast-multipath MPLS network model," in *Proceedings of the 3rd International IFIP/ACM Latin American Conference on Networking (ANC '05)*, pp. 65–71, Cali, Columbia, October 2005.