Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000), Taipei, Taiwan, April 10-13, 2000

Improving Distributed Workload Performance by Sharing Both CPU and Memory Resources *

Xiaodong Zhang[1] Yanxia Qu[2] Li Xiao[1]

[1]Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795
{zhang, lxiao}@cs.wm.edu

[2]Bell Laboratories, Lucent Technologies
Holmdel, NJ 07733-3030
yqu@dnrc.bell-labs.com

Abstract

We develop and examine 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 jobs.

*This work is supported in part by the National Science Foundation under grants CCR-9400719 and CCR-9812187, and EIA-9977030, by the Air Force Office of Scientific Research under grant AFOSR-95-1-0215, and by Sun Microsystems under grant EDUE-NAFO-980405.

1 Introduction

1.1 Important roles of memory considerations

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., [4], [5], [6], [9], [10] [11], [15]) mainly consider CPU load balancing by assuming 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 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 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 [1]. 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 times and the number of page faults in distributed systems.

1.2 Related work and our approach

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 [11] 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) [7] [14] 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) [2]. 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 further and significantly improved 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.

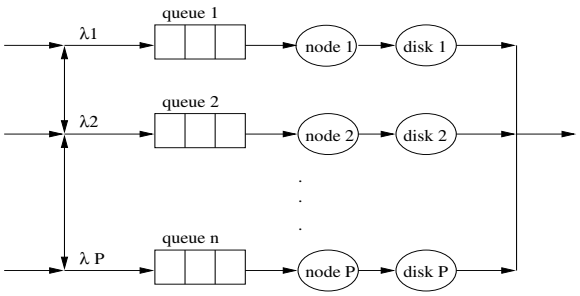

Figure 1: A queuing model for a distributed system.

2 A case for memory considerations

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 $i = 1, ..., P$, 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, we assume that all nodes have an identical computing capability, but have different memory capacities. This means 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 $\lambda_1, \lambda_2, ..., \lambda_P$ 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 loosing the generality, we apply formulas for open models [12] 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, we have two other parameters, $D_{cpu}(i)$ and $D_{disk}(i)$, which are average service demands of the CPU and the disk, respectively, at computing node i for $i = 1, ..., P$ (measured in seconds). The parameter values in Table 1 are collected from our trace-driven simulations which will be presented in section 4. The average disk service demands of node 1 ($D_{disk}(1) = 0.10$ seconds) is 10 times larger than that of node 2 ($D_{disk}(1) = 0.01$ 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

policies	node number (i)	λ_i	$D_{cpu}(i)$	$D_{disk}(i)$
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 1: Parameters of the queuing model.

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 performance is highly sensitive to

policies	$\lambda_1 : \lambda_2$	response time (seconds)
CPU	50:50	2.50
CPU & memory	45:55	1.43

Table 2: Results of the queuing model.

the average access rates λ_i, $i = 1, ..., P$, 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, we have conducted 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 Sharing both CPU and memory resources

3.1 Characterizing virtual memory activities

In a multiprogrammed environment, multiple jobs share a node for both its CPU and memory space. Using i to represent one of jobs' index in a node, we characterize its memory related operations by the following parameters:

- M_s^i: requested memory space in MBytes.
- M_a^i: allocated memory space in MBytes.
- M_t^i: memory threshold in MBytes. After the job executes in its stable stage, its working set size should also be stable. The memory space for the stable working set is called the memory threshold of the job [13]. If $M_a^i > M_t^i$, page faults would rarely occur, otherwise, paging would be frequently conducted during the job execution.
- σ_t^i: the average page fault rate caused by job i on a computing node is measured by a number of page faults per million instructions when the allocated memory space equals the memory threshold, i.e. $M_a^i = M_t^i$.

There are two types of page replacement policies in a multiprogrammed environment: global replacement and local replacement. A global replacement allows the paging system to select a memory page for a page replacement of a job throughout the memory space of a computing node. A local replacement requires that the paging system select pages for a job only from its allocated memory space. Most time-sharing operating systems use a global LRU replacement policy. Using the parameters defined above for an individual job, we further characterize the memory related operations in a multiprogrammed environment using a global LRU replacement policy on a single computing node:

- RAM: the available user memory space in MBytes of the computing node.
- $U = \sum_{i=1}^{N_p} M_s^i$: the memory usage is the total amount of requested memory space in MBytes accumulated from N_p jobs running on the computing node.
- $MT = \sum_{i=1}^{N_p} M_t^i$: the memory threshold is the total amount of memory thresholds accumulated from N_p jobs running on the computing node.
- σ: the average page fault rate caused by all jobs on a computing node is measured by a number of page faults per million instructions.

3.2 CPU-Memory-based load indices

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:

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

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

Our new load index considers both CPU and memory resources, which is used for our **CPU-Memory-based** load sharing policies. The basic principle of this new index is as follows. When a computing node in a distributed system 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, the load sharing decision is made by a memory-based policy which attempts to migrate jobs to nodes with sufficient memory space, or even to hold the jobs in a waiting pool if necessary.

In our new load sharing policy design, 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 the memory space of a system is sufficiently large for the jobs, while $MT \geq RAM$ means 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

$$Index_{hp}(L, MT) = \begin{cases} L, & MT < RAM, \\ CT, & MT \geq RAM. \end{cases}$$

When $MT < RAM$, CPU-based load sharing is used. When $MT \geq RAM$, 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 $MT \geq RAM$, the local scheduler immediately searches for the most lightly loaded node in the system as the job's destination.

Since the load index is set to CT when $MT \geq RAM$, 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 is a principle of *high performance computing*. That is the reason we define this option as an high performance computing load index, defined as $Index_{hp}$.

However, it may not be in favor of *high throughput computing* which emphasizes on effective management and exploitation of all available computing nodes. For example, when $MT \geq RAM$ 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 $MT \geq RAM$. The memory utilization parameter is defined as $\gamma = \frac{U}{RAM}$. When $\gamma < 1$, it means the memory space of the computing node is sufficiently large for the jobs. When $\gamma > 1$, it means the memory system is overloaded. This option is designed for high throughput computing, and its load index is defined as follows:

$$Index_{ht}(L, MT) = \begin{cases} L, & MT < RAM, \\ L \times \gamma, & MT \geq RAM. \end{cases}$$

Memory utilization parameter γ is used to proportionally adjust the load index. When $MT \geq RAM$, 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 above two load indices can be expressed as follows:

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

where $Index$ is either $Index_{hp}$ or $Index_{ht}$.

Besides comparing our policies with the CPU-based load sharing policy, we have also compared with a **memory-based** load 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:

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

4 Performance Evaluation Methodology

Our performance evaluation is simulation-based, consisting of two major components: a simulated distributed system and workloads[1]. We will discuss our simulation model, system conditions and the workloads in this section.

4.1 A simulated distributed system

Harchol-Balter and Downey [9] developed a simulator of a homogeneous distributed system with 6 nodes, where each local scheduler is CPU-based only. Using this simulator as a framework, we simulated a homogeneous distributed system with 6 nodes, where each local scheduler holds all the load-sharing policies we have discussed in the paper: CPU-based, Memory-based, CPU-Memory-based and their variations. The simulated system is currently configured with following parameters:

[1] The traces and simulation programs can be accessed at http://www.cs.wm.edu/hpcs/WWW/HTML/publications/abs00-1.html.

- *CPU speed:* 100 MIPS (million instructions per second).
- RAM size: 48 MBytes (The memory size is 64 MBytes including 16 MBytes for kernel and file cache usage [3]).
- *memory page size:* 4 KBytes.
- *Ethernet connection:* 10 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 SPARC-20 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 SPARC-20 workstations, and they are also used in [9]:

- *a remote execution cost*: 0.1 second.
- *a preemptive migration cost:* $0.1 + \frac{D}{B}$, 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).

4.2 System conditions

We have 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 we use is to find the most lightly loaded node in the distributed system.
- We assume that the memory allocation for a job is done at the arrival of the job.
- Similar to the assumptions in [8] and [14], we assume that page faults are uniformly distributed during job executions.
- We assume 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 [8] and [14].

4.3 Workloads

The workloads we have used are the 8 traces from [9]. 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:

`< arrival time, arrival node, duration time>`

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 MBytes. The 8 traces have different inter-arrival distributions and different Pareto service time distributions.

We have done the following modifications of the traces for our study. We 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 1 MBytes, 2 MBytes, 3 MBytes, and 4 MBytes. An informal study in [9] shows a Pareto distribution is reasonable for simulating memory demand. Each job has the following 4 items:

```
< arrival time, arrival node, requested
      memory size, 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 non-preemptive way, or a preemptive migration which may make the selected jobs be suspended, be moved to a remote node, and then be restarted. We include the two options in our load sharing experiments to study the merits and their limits.

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

- CPU_RE: CPU-based load sharing policy with remote execution (published in [9]).
- CPU_PM: CPU-based load sharing policy with preemptive migration (published in [9]).
- MEM_RE: Memory-based load sharing policy with remote execution.
- MEM_PM: Memory-based load sharing policy with preemptive migration.
- CPU_MEM_HP_RE: CPU-Memory-based load sharing policy using the high performance computing oriented load index with remote execution.
- CPU_MEM_HP_PM: CPU-Memory-based load sharing policy using the high performance computing oriented load index with preemptive migration.
- CPU_MEM_HT_RE: CPU-Memory-based load sharing policy using the high throughput oriented load index with remote execution.

- CPU_MEM_HT_PM: CPU-Memory-based load sharing policy using the high throughput oriented load index with preemptive migration.

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

A major timing measurement we have used is the mean *slowdown* which is the ratio between the total wall-clock execution time 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 Overall performance comparisons

We have 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. We scaled the page fault rate accordingly. The mean memory demand for each trace at different page fault rates was set to 1 MBytes, 2 MBytes, 3 MBytes, and 4 MBytes. We first concentrate on the performance comparisons of "trace 0" in different directions, and will present performance comparisons of all the traces immediately after this focused study.

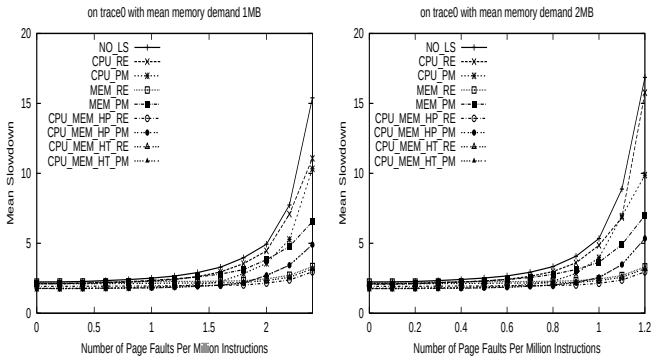

Figure 2: Mean slowdowns of the 9 load sharing policies as the page fault rate increases on "trace 0" with mean memory demands of 1 MBytes (left) and 2 MBytes (right).

Figures 2 and 3 present the mean slowdowns of "trace 0" scheduled by different load sharing policies with the mean memory demands of 1 MBytes, 2 MBytes, 3 MBytes and 4 MBytes, respectively. Before getting into details, we present several general observations based on the results in the figures. First, the slowdown is positively proportional to both the page fault rate and the mean memory demand of the trace. For example, if the maximum slowdown is no larger than 20 by any of the 9 policies, we could scale the page fault rate to 2.4 for the mean memory demand of 1 MBytes. Under the same condition of the maximum slowdown, we could only scale the page fault rate to 0.6 when we increase the mean memory demand to 4 MBytes. Second, when average memory access demand of jobs is low (low page fault rates and low mean memory demands), the performance differences among the load sharing policies are insignif-

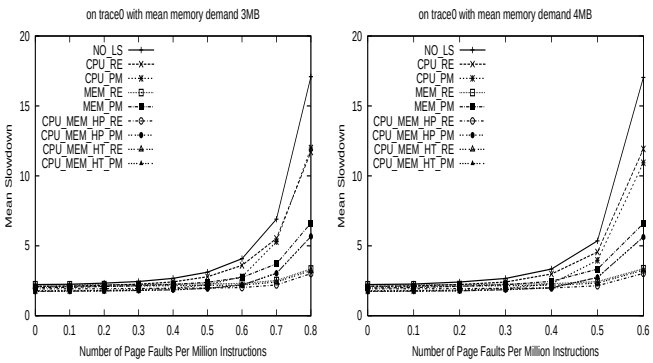

Figure 3: Mean slowdowns of the 9 load sharing policies as the page fault rate increases on "trace 0" with mean memory demands of 3 MBytes (left) and 4 MBytes (right).

icant. However, when the memory demand is high by increasing the mean memory demand, the memory related load sharing policies significantly outperform non-memory related policies, such as CPU-based ones. Finally, the policy of NO_LS is always the worst choice in all our experiments.

The CPU-based policies do reasonably well when the memory demand is low, but do poorly when the demand is high. Policies CPU_MEM_HP_RE, CPU_MEM_HT_RE, and MEM_RE perform well under all the conditions, and do show their effectiveness. For example, when the mean memory demand is 1 MBytes and the page fault rate is 2.4, the slowdown of these three policies is about 4.6 times lower, 3.3 times lower, 3.1 times lower, 2.0 times lower, 1.5 times lower, and 1.5 times lower than that of policies NO_LS, CPU_RE, CPU_PM, MEM_PM, CPU_MEM_HT_PM and CPU_MEM_HP_PM, respectively. When the mean memory demand is 4 MBytes and the page fault rate is 0.6, the slowdown of these three policies is about 5.1 times lower, 3.5 times lower, 3.2 times lower, 2.0 times lower, 1.7 times lower, and 1.7 times lower than that of policies NO_LS, CPU_RE, CPU_PM, MEM_PM, CPU_MEM_HT_PM, and CPU_MEM_HP_PM, respectively.

5.2 Remote execution vs. preemptive migration

We further look into the performance comparisons among all newly designed load sharing policies excluding policies NO_LS, CPU_RE, and CPU_PM. The studies in [9] 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 all the newly designed policies with CPU_PM by calculating the speedups of the new policies:

$$Speedup = \frac{slowdown\ of\ CPU_PM}{slowdown\ of\ a\ new\ policy}$$

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 so 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 3.6 for the mean memory demand of 4 MBytes 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.

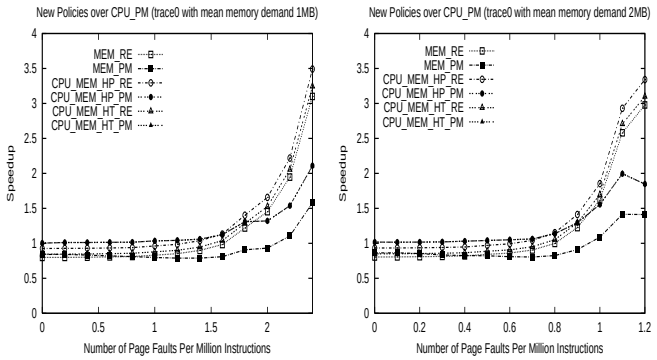

Figure 4: Speedups of the new load sharing policies over CPU_PM as the page fault rate increases on "trace 0" with mean memory demand of 1 MBytes (left) and 2 MBytes (right).

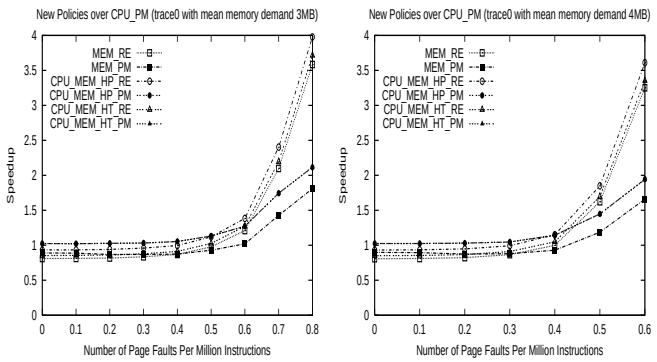

Figure 5: Speedups of the new load sharing policies over CPU_PM as the page fault rate increases on "trace 0" with mean memory demand of 3 MBytes (left) and 4 MBytes (right).

5.3 High performance vs. high throughput

Considering 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 MBytes and the page rate is 0.6, which is the highest among the speedups achieved by other policies. We have shown that the high 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 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 (for example, the speedup is 3.3 at the page rate of 0.6 with the mean memory demand of 4 MBytes), 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.

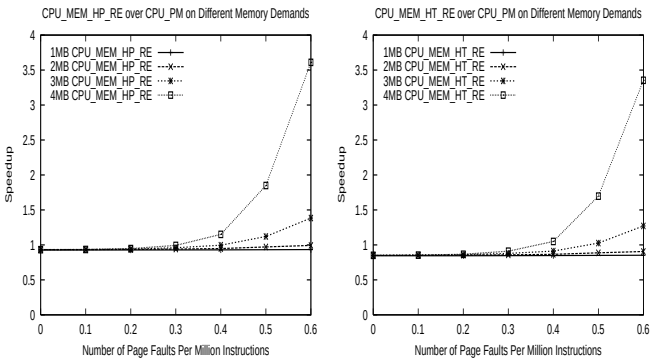

Figure 6: Speedups of our load sharing policies of remote execution over CPU_PM as both the page fault rate and mean memory demand increase on "trace 0" for the high performance approach (CPU_MEM_HP_RE, left) and for the high throughput approach (CPU_MEM_HT_RE, right).

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

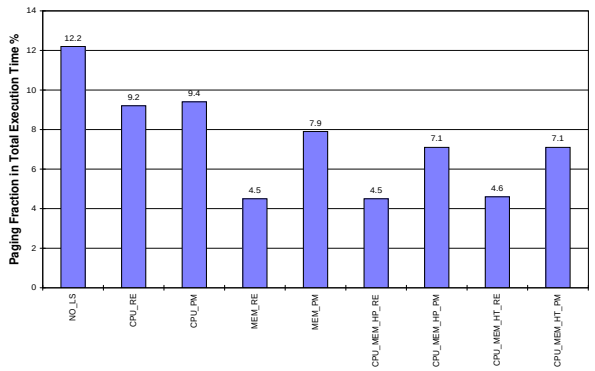

Figure 7: Paging time fractions in the total execution times for all the policies on "trace 0" with the mean memory demand of 4 MBytes at page fault of 0.6.

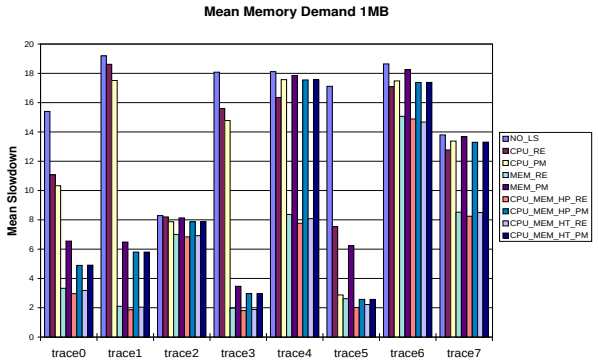

Figure 8: Mean slowdowns of all the 8 traces scheduled by different load sharing policies with the mean memory demand of 1 MBytes.

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 4 MBytes at page fault rate of 0.6. Compared with Figure 3, 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.

5.5 Performance comparisons of all the traces

The performance comparisons of the load sharing policies on other traces are consistent with what we have presented for "trace 0" in principle. Figures 8 and 9 present the mean slowdowns of all the traces ("trace 0", ..., "trace 7") scheduled by different load sharing policies with the mean memory demand of 1 MBytes and 4 MBytes, respectively.

For the mean memory demand of 1 MBytes, we adjusted the page fault rate for each trace in order to obtain reasonably balanced slowdown heights of 20 among all the traces in Figure 8. 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 MBytes, we also adjusted the page fault rate for each trace for the same reason as in the previous experiment. In Figure 9, we have $\sigma = 0.6$ 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",

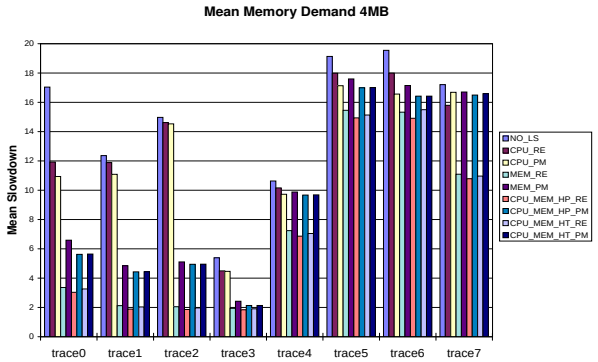

Figure 9: Mean slowdowns of all the 8 traces scheduled by different load sharing policies with the mean memory demand of 4 MBytes.

$\sigma = 0.58$ 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).

6 Conclusion

We have experimentally examined and compared 8 load sharing policies, where two policies are CPU-based, two are memory-based and four are CPU-Memory-based. Based on our experiments and analysis we have following 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 is 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 we did not consider other data access patterns so that we were 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.

We summarize the 8 policies and their merits and limits in Table 3, which gives a guideline to choose a load sharing policy based on the types of jobs in distributed systems. In the table, the relationship between a load sharing policy and each type of jobs is represented by symbol "+" (beneficial), "++" (highly beneficial) and "-" (non-beneficial).

Policies	CPU-bound	memory-bound
CPU_RE	+	-
CPU_PM	++	-
MEM_RE	-	++
MEM_PM	-	+
CPU_MEM_HP_RE	++	++
CPU_MEM_HP_PM	++	+
CPU_MEM_HT_RE	++	++
CPU_MEM_HT_PM	++	+

Table 3: Summary of the 8 load sharing policies and their impact on the job types (CPU-bound and memory-bound).

There are two limits in this study. First, the model in section 2 is only used for a case study, and not accurate enough to characterize overall load sharing performance. Second, memory demands of jobs are generated by Pareto distributions. There is only one working set in each job. Although we have clearly shown the impact of the memory demand to load sharing performance by the simulated memory access patterns in our workloads, the patterns may not be the same as those of some practical jobs.

The future work of this research can go in different directions. We are expanding the simulated distributed system to be heterogeneous and scalable. We are also collecting more real-world traces for experiments. We will investigate the effects of more memory related activities and network contention to load sharing performance in distributed systems.

Acknowledgement: We wish to thank Phil Kearns to read a preliminarly report of this work, Stefan Kubricht to read the latest version of the paper, and their comments and suggestions. The comments from the anonymous referees are constructive. The trace-driven simulator and the trace data in the public domain from M. Harchol-Balter and A. B. Downey are helpful to our program development and testing of trace-driven simulations.

References

[1] A. Acharya and S. Setia, "Availability and utility of idle memory in workstation clusters", *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1999, pp. 35-46.

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

[3] A. Cockcroft, *Sun Performance and Tuning*, Second Edition, Sun Microsystems Press, 1997.

[4] F. Douglis and J. Ousterhout, "Transparent process migration: design alternatives and the sprite implementation", *Software — Practice and Experience*, Vol. 21, No. 8, 1991, pp. 757-785.

[5] X. Du and X. Zhang, "Coordinating parallel processes on networks of workstations", *Journal of Parallel and Distributed Computing*, Vol. 46, No. 2, 1997, pp. 125-135.

[6] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing", *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1988, pp. 63-72.

[7] M. J. Feeley, et. al., "Implementing global memory management systems", *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995, pp. 201-212.

[8] G. Glass and P. Cao, "Adaptive page replacement based on memory reference behavior", *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1997, pp. 115-126.

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

[10] C.-C. Hui and S. T. Chanson, "Improved strategies for dynamic load sharing", *IEEE Concurrency*, Vol. 7, No. 3, 1999, pp. 58-67.

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

[12] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy, *Capacity Planning and Performance Modeling*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[13] A. Silberschatz and P. B. Galvin, *Operating Systems Concepts*, 4th Edition, Addison-Wesley, 1994.

[14] G. M. Voelker, et. al., "Managing server load in global memory systems", *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1997, pp. 127-138.

[15] S. Zhou, "A trace-driven simulation study of load balancing", *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, 1988, pp. 1327-1341.