Proceedings of the 21st International Conference on Distributed Computing Systems, (ICDCS'2001).

Dynamic Load Sharing With Unknown Memory Demands in Clusters *

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

Abstract

A compute farm is a pool of clustered workstations to provide high performance computing services for CPU-intensive, memory-intensive, and I/O active jobs in a batch mode. Existing load sharing schemes with memory considerations assume jobs' memory demand sizes are known in advance or predictable based on users' hints. This assumption can greatly simplify the designs and implementations of load sharing schemes, but is not desirable in practice. In order to address this concern, we present three new results and contributions in this study. (1) Conducting Linux kernel instrumentation, we have collected different types of workload execution traces to quantitatively characterize job interactions, and modeled page fault behavior as a function of the overloaded memory sizes and the amount of jobs' I/O activities. (2) Based on experimental results and collected dynamic system information, we have built a simulation model which accurately emulates the memory system operations and job migrations with virtual memory considerations. (3) We have proposed a memory-centric load sharing scheme and its variations to effectively process dynamic memory allocation demands, aiming at minimizing execution time of each individual job by dynamically migrating and remotely submitting jobs to eliminate or reduce page faults and to reduce the queuing time for CPU services. Conducting trace-driven simulations, we have examined these load sharing policies to show their effectiveness.

1 Introduction

With rapid development of CPU chips and increasing demand of data-intensive applications, the memory related operations in computer systems become more and more expensive relative to CPU cycles. In addition, the buffer cache for jobs' I/O data in modern Unix systems competes for the same physical main memory with application programs' virtual pages, making memory accesses and allocations even more dynamic and demanding. Thus, overall performance of a distributed system and the response time of each individual job in the system are increasingly dependent on the effective utilization of the entire memory hierarchy in the system. Researchers have started to revise load sharing policies by considering effective usage of global memory resources in addition to the CPU load balancing (see e.g. [2, 3, 13]). The load sharing schemes cited above need to know the memory allocation size of each job in advance. Some users may be able to provide an estimated size along with a job submission, which may not be accurate. A load sharing scheme should not heavily rely on the estimated memory sizes provided by users. In addition, the memory accesses and allocation sizes of jobs can change dynamically during execution and are hard to estimate and emulate without a low level system monitoring. There are other limits in related studies:

- *Analysis and designs based on the traces obtained at the user level*: Many memory operations, such as virtual-physical memory mapping, buffer cache accesses, and page faults, are serviced by the operating system. The most accurate way to obtain memory system information is to monitor memory activities and to collect memory traces of workloads at the system level.
- *Simplifying the estimation of the available memory space*: The available memory space may be estimated by the difference between the size of available memory space and the accumulated memory sizes of running jobs. The estimation of available memory space needs more dynamic system information including the status of buffer caches.
- *Quantifying lifetimes of jobs without detailed breakdowns:* The lifetime of a job has been used as an important factor in load sharing designs. Which process to be migrated in existing scheduling policies (see e.g. [6]) is considered by predicting the lifetime of CPU intensive jobs. The lifetime of a job mainly consists of CPU time, memory paging time, and I/O service time. Detailed breakdowns of the lifetime into these major portions will provide a multi-dimensional set of information for load sharing decisions. The quantum distributions of different execution portions may be application dependent. However, significant portions of lifetimes of large scale application jobs will be spent on memory and I/O operations as the speed gap between processors and memory accesses continues to widen.
- *Lack of robust support for mixed types of jobs*: In a compute farm, CPU-intensive, memory-intensive, and I/O active jobs

*This work is supported in part by the National Science Foundation under grants CCR-9812187, EIA-9977030, and CCR-0098055, by the Air Force Office of Scientific Research under grant AFOSR-95-1-0215, and by Sun Microsystems under grant EDUE-NAFO-980405.

normally co-exist. A load sharing system should be beneficial to all job types instead of focusing on one or two types.

In order to address these limits, we have characterized memory performance of different types of jobs by analyzing workload traces collected from Linux kernel instrumentation. We have also developed and evaluated a 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/Os. A job migration decision is made by minimizing the memory space competition for allocating jobs' memory space and the buffer cache of I/O data, and by trading cheap CPU cycles and minor network overheads for significant memory performance gain. Our load sharing objective is to minimize execution time of each individual job by effectively reducing page faults and queuing times, and minimizing the migration cost.

2 System Monitoring and Tracing

2.1 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. [13]), 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 a 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. [9]). This approach may be effective in a dedicated environment. However, in a time-shared 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. [5]). 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], [3]. The interactions between a user program and the kernel, and detailed system activities may not be captured in this approach.

2.2 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 queueing 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 non-pageable. 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 in the memory system. 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.
- *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.

2.3 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 the 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 the seven benchmark programs (to be introduced in the next section) without using the instrumentation and the ones with the instrumentation. The comparisons show that the average lifetime increase is only 2.3% with insignificant intrusive effects; and that the number of major page faults does not increase due to the instrumentation.

3 Characterizing Job Interactions

3.1 Characterizations of workloads

We have selected seven large scientific and system programs which are representative CPU-intensive, memory-intensive, or/and 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 [14].
- *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 [11].
- *matrix multiplication,* (m-m): This is a standard matrix multiplication program, and is both CPU-intensive and memory-intensive.
- *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 [13].
- *partitioning meshes,* (metis): This program, called METIS, is developed to partition unstructured graphs, partition meshes, and compute fill-reducing orderings of sparse matrices [7]. 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 [8]. 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 [8].

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 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% of the physical memory space in this system, the operating system will immediately terminate the newly arrived process to release the memory space.

Table 1 presents the experimental results of all the seven programs, where the data size is the number of entries of the input data, the working set gives a range of the memory space demand during the execution, the number of paging and the time are the number of page swaps between the main memory and the disk due to memory shortage and the time spent for the paging, the number of I/Os/time(s) is the number of reads/writes and the time spent in seconds, the k-time is the execution time used in the system mode for kernel system calls, the u-time is the execution time used in the user mode including the CPU service time and memory access time without page faults, the lifetime is the sum of k-time, u-time, I/O time, and paging time, the paging and I/O portions (paging %, 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 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 1.

3.2 Characterizations of program interactions

We have monitored executions of many groups of programs in a time-shared environment to observe the effects of program interactions to memory performance. The executions of most groups with 4 programs experienced severe thrashing and were not able to complete. The executions of groups with 6 or 7 programs were immediately stuck due to huge amount of thrashing. Table 2 presents monitored execution performance of three representative job interactions of 2 jobs (m-m and m-sort), 3 jobs (bit-r, m-m, and r-wing), and 4 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 following observations:

- The page faults continuously happen as the memory demand from the interacted jobs oversizes the available physical memory space.

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

Table 1: Execution performance and memory related data of the seven benchmark 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

Table 2: Execution performance and memory related data of three groups of interacted programs.

- As the memory demand from the interacted jobs oversizes about 65% of the available physical memory space, some programs start to experience thrashing with little useful execution. For example, as the 4 programs of "bit-r", "m-m", "m-sort", and "t-sim" are interacted, more than 90% execution time of "m-sort" is used for paging. As the memory demand oversizes more than 70%, all the programs experience thrashing with little useful work done, where the sharply increased paging rate starts to grow slowly to reach a stable state.
- When the memory demand from the interacted jobs oversizes less or around 65% 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 large amount 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%, to 24.75%, and to 64.95% caused by the increase of the number of interacted programs, the paging time portion in the execution of this program changes from 37.40%, to 43.3%, and to 90.52%, 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 to 64.95%, the paging time portion is 13.18%, 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/Os 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:

$$R_p(Q_o) = R_{max} - \frac{\alpha}{\beta Q_o^2 + \gamma Q_o + \delta}, \tag{1}$$

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 stably threshing 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%. Increases of Q_o beyond it turn $R_p(Q_o)$ into a stable stage reaching to R_{max}. Here are the values of the variables 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:

$$R_p(N_{io}) = \eta + \mu N_{io}, \tag{2}$$

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). Figure 1 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 (1) and (2):

$$R_p(Q_o, N_{io}) = \eta + \mu N_{io} + R_{max} - \frac{\alpha}{\beta Q_o^2 + \gamma Q_o + \delta}, \quad (3)$$

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

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

This model is used in our trace-driven simulation.

4 A memory-centric load sharing scheme

4.1 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 compute farm. 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. In contrast, the scheduling decision is made when a job arrives if the demanded memory size is fixed and known. (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 3.

The framework of the memory-centric load sharing scheme in machine j ($j = 1, ..., m$) 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) >= MT(j))
      if (Q_o(j) <= 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;
          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;
    if (N_job(j) > CT(j))
      while NOT find_a_suitable_machine_for_migration
          continue local executions;
      migrate the identified_job to the suitable_workstation;
      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 returns the ID of the selected machine for a job migration, and *find_a_suitable_machine_for_remote_submission* which returns the ID of the selected 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.

4.2 The variations of the load sharing scheme

Here are the four alternatives of identifying a job for a migration.

1. identifying the most memory-active job:

   identified_job()
   d = the job ID with the highest $R_p(d)$ **or** the highest $WS(d)$;
   identified_job = d;

2. identifying the most I/O active job:

   identified_job()
   d = the job ID with the highest $N_{io}(d)$;
   identified_job = d;

3. identifying either the oldest or youngest job:

   identified_job()
   d = the job ID with the highest (**or** the lowest) $Age(d)$;
   identified_job = 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) $Weight(d)$;
   identified_job = 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:

$$Weight(d) = \frac{1}{4}\left(\frac{Age(i)}{\max_{i=1}^{n} Age(i)} + \frac{R_p(i)}{\max_{i=1}^{n} R_p(i)} + \frac{WS(i)}{\max_{i=1}^{n} WS(i)} + \frac{N_{io}(i)}{\max_{i=1}^{n} N_{io}(i)}\right) \quad (5)$$

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))
  find_a_suitable_machine_for_migration = k;
else
  find_a_suitable_machine_for_migration = 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))
  find_a_suitable_machine_for_remote_submission = k;
else
  find_a_suitable_machine_for_remote_submission = false;
```

Overheads of page faults and job migrations are two major sources of performance degradations in a compute farm. 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, $MT(j)$, $j = 1, ..., m$) 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:

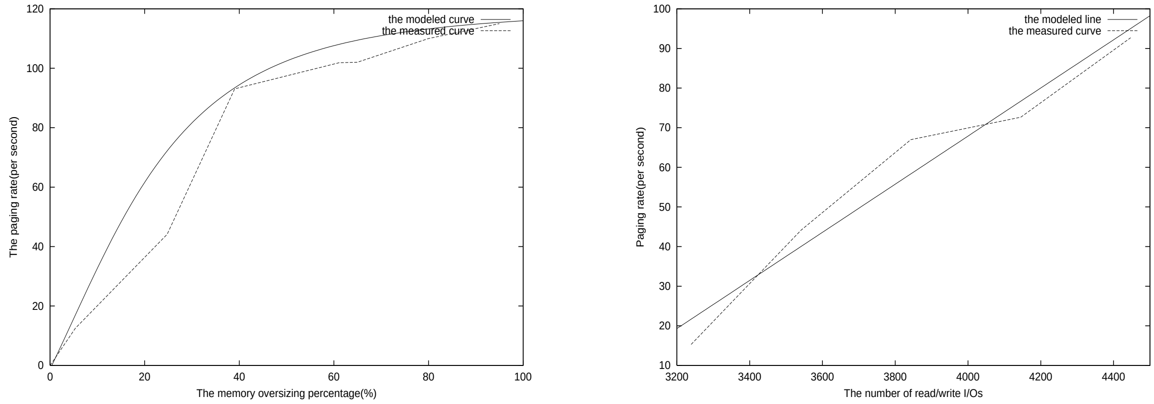

Figure 1: Modeled and measured correlations between the paging rate and the memory oversizing percentage (left), and between the paging rate and the read/write IOs (right).

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.

Table 3: Notations, variables, and parameters of the page-free-free load sharing scheme.

- *load sharing with a minimization of page faults.* The memory threshold is set below 0% ($MT(j) < 0\%$) 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% ($MT(j) = 0\%$). 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 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.* The memory threshold is set above 0% ($MT(j) > 0\%$). 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.

5 Trace-drive simulation environment

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

5.1 A simulated clustered compute farm

We have simulated a homogeneous clustered compute farm with 32 nodes, where each local scheduler making the load-sharing policies we have discussed in the paper. The simulated system is configured with workstations of 233 MHz CPUs, 128 MByte Memory each, and 128 Mbyte swap space each. The memory page size is 4 KBytes. The Ethernet connections are 10 and 100 Mbps. Each page fault service time is 10 ms. The time slice of CPU local scheduling is 10 ms, and the context switch time is 0.1 ms. The 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.

- *remote submission/execution cost*: $r = 0.1$ second for 10 Mbps network, and $r = 0.05$ second for 100 Mpbs network.
- *a preemptive migration cost:* $r + \frac{D}{B}$, where r 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 ($B = 10$ or 100 for the Ethernet).

Each computing node maintains a global load index file which contains CPU, memory, and I/O load status information of other computing nodes. The load sharing system periodically collects and distributes the load information among the computing nodes.

5.2 Workloads

The workload traces are collected by using system facility tool *top* and our instrumentation library to monitor the execution of the 7 benchmark programs at different submission rates on a Linux workstation. Job submission rates are generated by a lognormal function:

$$R_{ln}(t) = \begin{cases} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(\ln t - \mu)^2}{2\mu^2}} & t > 0 \\ 0 & t \leq 0 \end{cases} \tag{6}$$

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. [4], [10]). Three traces are collected with three different arrival rates:

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

6 Performance evaluation

The *slowdown* of a job is the ratio between its wall-clock execution time and its CPU execution time. A major performance metric we have used is the mean slowdown, which is the average slowdown of all jobs. Major contributions to slowdown come from the delays of page faults, queuing time for CPU service, and the overhead of migrations and remote submission/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.

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

6.1 Effects of load index variations

We have examined the following load indices to identify a job for migration when the amount 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 [6] (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 (5) (simplified as *combined* in figures).

In addition, we have also compared the performance of the above schemes with that 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 threshing occurs. Therefore all the jobs will complete. This policy is denoted as *no-LS-blocking* (simplified as *blocking* in figures).

The left figure in Figure 2 shows the slowdowns of the three workload traces scheduled by the 4 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 4 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 slowdowns 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 slowdowns for *Trace 1*, (about 100% 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% higher for *Trace 2*, and about 10% 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 soon completes. 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% higher for the both traces compared with the slowdowns of *CPU-Memory-I/O-based*). However, its slowdown for *Trace 3* was more than 60% higher than those of other three policies. The working set size information reflects the currently 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 seemed 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% to 30% 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. [6]) 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 Figure 2 shows the overhead breakdowns for *Trace 1*, including the paging time, migration time and the queuing time for *Trace 1*. Except 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 had lowest migration costs but high queueing times. Only the *CPU-Memory-I/O-based* policy minimized 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 queueing times makes the *CPU-Memory-I/O-based* policy perform the best for all the traces.

6.2 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%, -5%, 0%, 5%, and 15%. The left figure in Figure 3 shows the slowdown changes of the 4 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% to -5%, stays unchanged in the range of $MT = -5\%, 0\%, 5\%$, and increases as MT increases to 15%. 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, thus is more generous to allow more page faults occur in a system.

The slowdowns of the *page-fault-rate-based* policy do not seem to follow a regular pattern as the MT changes, which again confirms its indeterministic nature.

The right figure in Figure 3 shows the paging time changes of the 4 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% to 15%). The paging times of the *page-fault-rate-based* policy are significantly higher than other those of other policies, but are quite independent of the changes of MT.

The left figure in Figure 4 shows the migration time changes of the 4 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 an 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. The migration time reductions of the *working-set-size-based* policy are the most significant as MT increases. More jobs will be interacted in a machine as MT increases. Since the memory system will be overloaded, for example, oversizing 5% and 15%, 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.

The right figure in Figure 4 shows the queuing time changes of the 4 load sharing policies as the memory threshold changes under the same system condition. We show that the queueing times of all the policies are quite independent of MT changes. As we have shown, the queueing times 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%. This is because more jobs are allowed to execute in a machine as MT increases, so the queueing time increases also.

6.3 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 Figure 5 presents the slowdowns of the three workload traces scheduled by the 4 load sharing policies and policy *no-LS-blocking* as the network is upgraded to 100

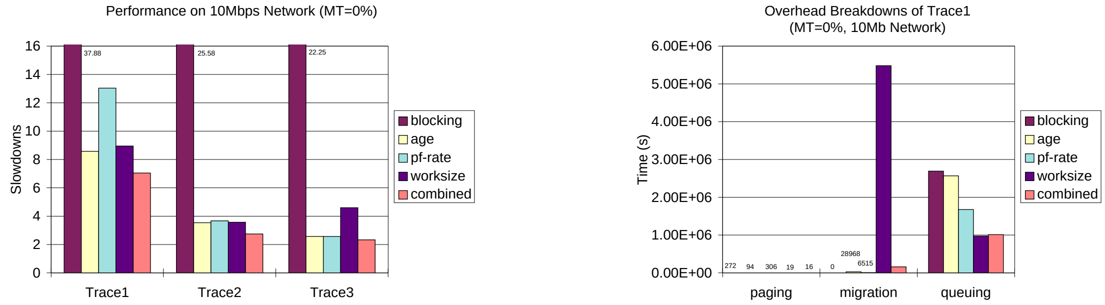

Figure 2: Slowdowns of the three traces scheduled by the 5 policies (the left figure), and overhead breakdowns of *Trace 1* by the 5 policies (the right figure). The experiments were conducted on a 10 Mbps network with a memory threshold of 0%.

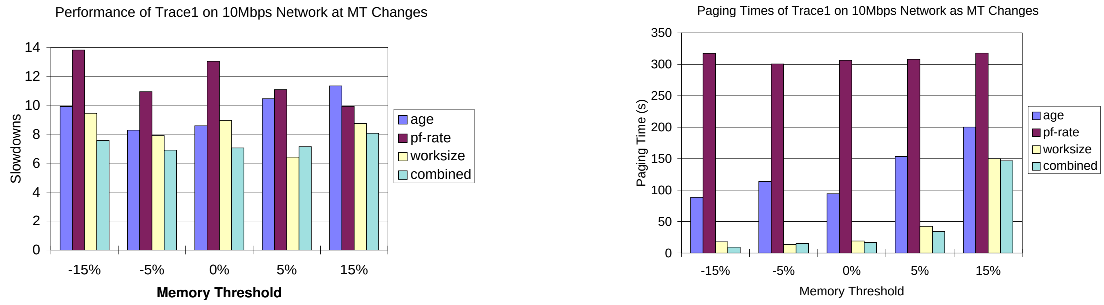

Figure 3: Slowdowns of *Trace 1* scheduled by the 4 load sharing policies as the memory threshold changes (the left figure), and their paging time changes as the memory threshold changes (the right figure). The experiments were conducted on a 10 Mbps network.

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 Figure 2). The network upgrading does reduce the slowdowns of the *working-set-size-based* policy for all the three traces, but only slightly reduces the slowdowns of the *CPU-Memory-I/O-based* policy.

The right figure in Figure 5 presents the migration time of the three workload traces scheduled by the 4 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 Figure 4 for comparisons).

7 Conclusion

We conclude the paper by summarizing what we have learned in this experimental study. 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%. A conservative MT does not fully utilize the memory system, while a 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 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 compute farms. The *CPU-Memory-I/O-based* policy is effective to all three types of traces on both low and high speed networks.

Acknowledgement: We thank Phil Kearns' help for the kernel instrumentation on Linux systems and his suggestions. Raphael Finkel further explained to us his implementations of memory image transferring in job migrations. We thank Bill Bynum for

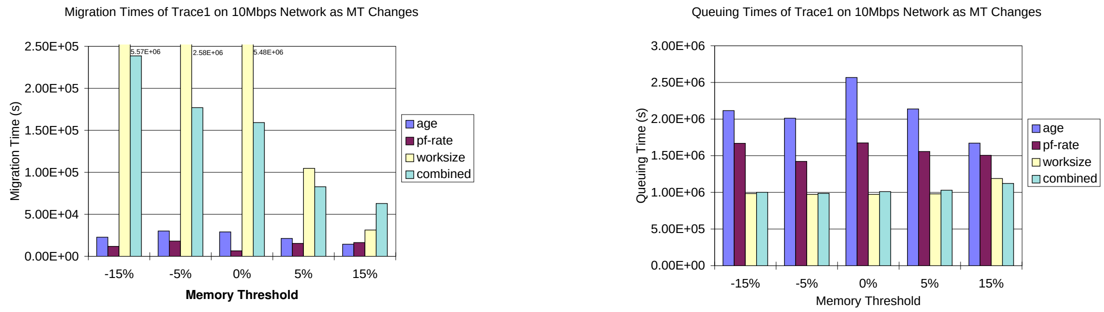

Figure 4: Migration times of *Trace 1* scheduled by the 4 load sharing policies as the memory threshold changes (the left figure), and their queueing time changes as the memory threshold changes (the right figure). The experiments were conducted on a 10 Mbps network.

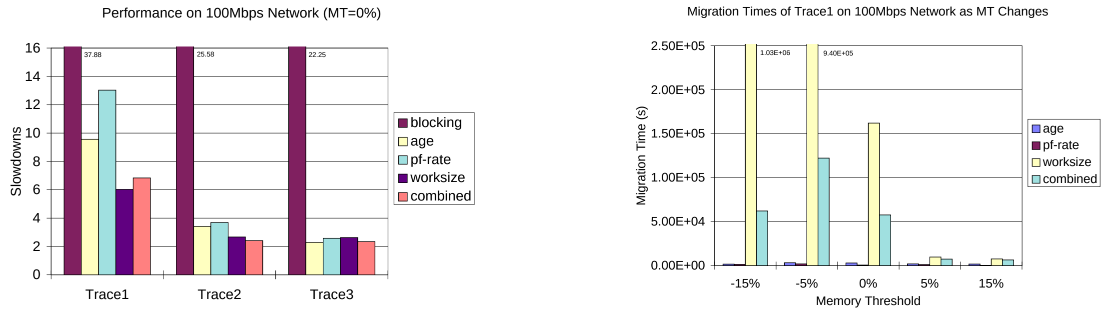

Figure 5: Slowdowns of the three traces scheduled by the 4 load sharing policies (the left figure), and the migration times of *Trace 1* as the memory threshold changes (the right figure). The experiments were conducted on a 100 Mbps network.

reading this paper and his suggestions. Finally, we are grateful to the anonymous referees' helpful comments.

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

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

[5] 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.

[6] 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.

[7] URL: http://www.cs.umn.edu/˜metis.

[8] K-L. Ma and T. W. Crockett, "A scalable, cell-projection volume rendering algorithm for 3D unstructured data", *Proceedings of the Parallel Rendering'97 Symposium*, October 1997, pp. 95-104.

[9] V. G. Peris, M. S. Squillante, and V. K. Naik, "Analysis of the impact of memory in distributed parallel processing systems", *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994, pp. 5-18.

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

[11] L. Xiao, X. Zhang, and S. A. Kubricht, "Improving memory performance of sorting algorithms", *ACM Journal on Experimental Algorithmics*, Vol. 5, 2000.

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

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

[14] Z. Zhang and X. Zhang, "Cache-optimal methods for bit-reversals", *Proceedings of Supercomputing'99*, November 1999.