Effect of Scheduling Discipline on CPU-MEM Load Sharing System

Lei Shi[1,2], Yuyan Sun[1,2], Lin Wei[1,2]
1. School of Information Engineering, Zhengzhou University, Zhengzhou 450052, China
2. Henan Provincial Key Lab on Information Network, Zhengzhou 450052, China
shilei@zzu.edu.cn Syxsyy1230@yahoo.com.cn weilin@shengda.edu.cn

Abstract

Scheduling schemes have important effects on the performance of load sharing in distributed systems. Because the load sharing policy based on CPU-memory has made the system memory an important role in effecting the system performance, it can reduce the paging fault and enhance the usage of the system resource. In light of the characteristics of the load sharing based on the CPU-memory and the variation of the jobs in executing, new CPU local scheduling schemes named more memory-request more CPU Slice based on Round Robin (RR-MMMCS) mechanism and more memory-request more CPU slice based on predicting (MMMCS-P) mechanism, are presented. Furthermore, the effects on the load sharing policy based on the CPU-memory of the variance of the inter-arrival time and service time are discussed. The trace-driven simulations show that the load sharing policy based on the CPU-memory and RR-MMMCS, MMMCS-P scheduling schemes are effective and have better performance in average response time for both CPU-memory and memory–bound jobs.

1. Introduction

Two classes of load-distribution policies exist[1]: load-sharing policies aim at keeping as many processors busy as possible, whereas load-balancing policies aim, in addition, at equalizing the queue lengths at all processors. Load sharing is a technique to enhance resources, utilizing parallelism exploiting throughput improvisation, and to cut response time through an appropriate distribution of the application[2]. The elementary issues which a high system performance should take into account are how to decrease the average response time and reduce the cost of distributed system by decreasing the times of paging fault, reducing the queuing time of tasks, etc. In order to solve the above problems, the stage of CPU resource and the condition of memory resource are considered when a load sharing policy is designed because the system performance will be deteriorated seriously and the merit of policy will exhausted if the problems from the memory (i.e., the cost of paging fault) are lack of consideration. And the delay from memory problems makes a more important role with the difference between memory speed and CPU speed greater. Based on these reasons, the important aspect of an efficient load sharing policy is how to use the memory resource sufficiently and efficiently. Load sharing policies based on CPU-memory can have a better system performance because of considering the CPU and memory resource more sufficiently.

Traditional scheduling disciplines in most distributed systems are FCFS (first come first serve), RR[3] (round robin) and MWRR[4] (multiclass weighted round robin) which lack the consideration of the data attribution of tasks memory request but the time attribute of tasks. Load sharing policy based on CPU-memory can reduce the paging fault and enhance the usage of the system resource. In light of the characteristics of the load sharing based on the CPU-memory and the variation of the jobs in executing, new CPU local scheduling schemes named more memory-request more CPU Slice based on Round Robin (RR-MMMCS) mechanism and more memory-request more CPU slice based on predicting (MMMCS-P) mechanism, are presented. Furthermore, the effects on the load sharing policy based on the CPU-memory of the variance of the inter-arrival time and service time are discussed.

The rest of this paper is organized as follows. Section 2 addresses related work. Section 3 reviews the load sharing policy based on CPU-memory. Section 4 introduces the proposed load sharing metrics and presents the analytical results. Task scheduling algorithms are then given. Section 5 presents implementation details of the simulator and system

The Sixth International Conference on Grid and Cooperative Computing(GCC 2007)
0-7695-2871-6/07 $25.00 © 2007 IEEE

parameter measurement. Section 6 presents the simulation and experimental results. Finally, in Section 7, the current work is summarized.

2. Related work

In distributed systems, there are global and local scheduling policies. Global scheduling decides which processor serves which jobs. In contrast, local scheduling decides when and at what rate to serve jobs on a single processor. The global and local scheduling decisions policies are taken according to the global and local scheduling policies. Several load sharing schemes have been presented in the distributed systems (e.g. [3], [6], and [7]) mainly consider effectively local scheduling schemes. Although these load-sharing policies have been effective in increasing the utilization of resources in distributed systems, at most they may only consider two node scheduling policies-FCFS and RR ([3]) without any difference. It is aggressed that the efficient integration of global and local scheduling may greatly improve the system performance. L. Xiao, X. Zhang, and Y. Qu ([6], [7]) have studied specific load-sharing policies, which consider both system heterogeneous and effective usage of memory resources. Their work shows that load sharing policies considering both CPU and memory resource are robust and effective in heterogeneous systems.

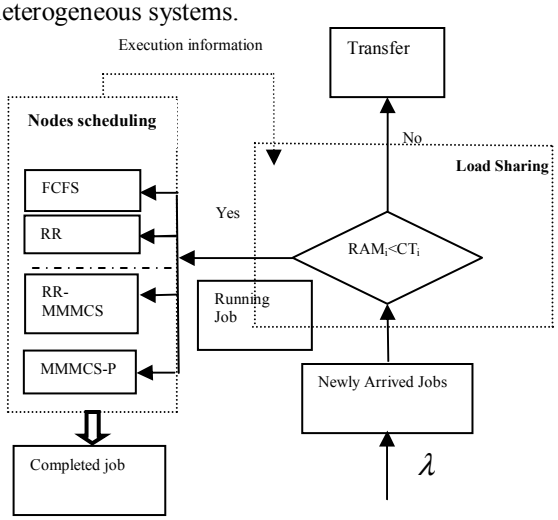

Figure 1: Load sharing model

Based on the CPU-memory policy, new local scheduling disciplines are presented in this paper. According to the expanded trace-driven simulations, better routing policies that can enhance the distributed system performance are discussed. The nodes scheduling model is shown in figure 1. When a job arrives with an arrival rate λ, load sharing would decide the job executed locally or not. Once the job is executed locally, nodes scheduling will select a node scheduling discipline to implement the task. Of course, some executive information is fed back to the load sharing.

3. Load sharing based on CPU-MEM

The load sharing policy based on CPU-memory combines the memory resource and the CPU resource during sharing workload, which can be described as follows:

```
While (taski is arrival at nodej)
     taskmemi=allocate_mem();
 if (nodememi+taskmemi <=MT) indexj=L;
 else  indexj= CT;
if  (indexj<CT)  process the taski
                 locally ;
else if (selected the most lightly
          loaded node) taski remote
          execution;
     else  block taski;
end while
```

The above description shows that computation nodes make the migration decision according to the CPU load index which usually is the length of CPU service queue (L_i) when there is enough memory space offered to the running and arrived tasks ($nodemem_j+taskmem_j<=MT_j$). But the nodes will become the overloaded nodes, if the nodes are short of the memory space, then the load sharing policy will attempt to transfer tasks to nodes with sufficient memory space, or even to hold the tasks in a waiting queue if necessary.

The above algorithm can be explained more distinctly by formula (1) and (2):

$$Index(i)(L_i, MT_i) = \langle MT_i \prec RAM_i \rangle ? L_i : CT_i \quad (1)$$

Where L_i refers to the length of CPU waiting queue (the number of tasks) of node i. CT_i means CPU threshold, which is normally set based on the CPU computing capability. MT_i means the memory threshold which is the total amount of memory thresholds accumulated from all tasks running on the computing node i. The memory threshold of a task means the memory size which the stable working set requests after the task executes in its stable stage. RAM_i is the available user memory space of the computing node i.

$$LS(Index(i)) = \langle Index(i) \prec CT_i \rangle ? local-execution : remote-execution \quad (2)$$

Formula (2) describes how the sharing system to decide the execution status of tasks. The above two formulae indicate that the sharing policy based on the

CPU-memory is similar to the policy based on the CPU when $MT_i<RAM_i$. But the load indices of node i are set to CT_i in order that this node will reject to accept any new task when $MT_i>RAM_i$.

4. Scheduling discipline

The most important merit of load sharing policy based on the CPU-memory is that it not only uses the CPU resource but takes into account the memory resource of system when it computes the effect of load indices on the system performance. It also has some shortages: the CPU-memory load sharing may not improve the usage of memory in a single node, the paging fault is still serious, and the task migration rate increases because of considering the memory resource when the memory load of most nodes in the distributed system is heavy. In this case, the overload which is caused by the paging fault and the increasing migration exhausts the benefit from considering the memory resource in load sharing policy to some extent. Two new node scheduling disciplines: Round Robin based-More Memory-request More CPU Slice (RR-MMMCS) policy and More Memory-request More CPU Slice based on predicting (MMMCS-P) policy are proposed in order to improve the performance of load sharing system and compensate the shortage of sharing policy.

4.1 RR-MMMCS scheduling discipline

Following is the pseudo-code procedure of RR-MMMCS scheduling discipline.

```
While (! Ready-Queue-NULL)
For (get task_i from Ready-Queue)
          summem=summem+taskmem_i;
end;
avemem=summem/tasknum;
remove the first task_i in the Ready-Queue;
if     (task_i/avemem>3)     continual-times_i=3;
else if (task_i/avemem>2)
                  continual-times_i
             =2;
        else    continual-times_i =1;
if(continual-times_i>=1)
              execution=execute(task_i)
              ;
if(execution==paging-fault)
          push   task_i   into   Block-Queue;
else if (execution=finish)
          push   task_i   into   Finish-Queue;
     else   if(continual-times_i==0)
push task_i into the end of Ready-Queue;
    else push task_i into the front of Ready-Queue;
end while
```

In the above description, $taskmem_i$ refers to the memory request of task i; summem means the sum of memory request of all the tasks in ready queue: $sumem = \sum_{i=1}^{tasknum} taskmem_i$; avemem means the average memory request of all tasks in the present queue; continual-$times_i$ means the number of quantum which the task i can execute continually with ; The function of execute () is that a task is processed in a quantum Q , and the return value represents the events happened in the function; Ready-Queue, Block-Queue, Finish-Queue and mean the task ready queue (CPU waiting queue), the blocking queue and the finishing queue. So, a conclusion can be drawn by summarizing the above algorithm: RR-MMMCS Policy computes the number of CPU slice which a task can execute continually with by the size of memory request of tasks in the CPU waiting queue, then these slices are given to the corresponding task as an integer, and all the tasks are executed round.

The formula of computing for the number of slice of execution continually is shown:

$$continual-times_i = \begin{cases} 1, taskmem_i \le 2 * avemem \\ 2, taskmem_i \le 3 * avemem \\ 3, taskmem_i > 3 * avemem \end{cases}$$

One slice is given to the task i (continual-$times_i$=1) when the memory demand of task i is smaller than the average memory demand. But when the memory demand of task i is larger than the average memory demand, more than one slices are given to this task (continual-$times_i$=2 or3). The target of RR-MMMCS algorithm is that the more memory demand task should get more service time, so the larger memory demand task will be completed more quickly, the average memory demand in a node is reduced, the paging fault rate and the number of migration are decreased. But, with respect to the merit of fairness in RR, the most number slices of tasks is limited into three in RR-MMMCS, so RR-MMMCS algorithm remains of the basic merit of RR.

Here an example is given to explain how the algorithm decreases the average memory demand.

Assume that there are two tasks P_1, P_2 executing in a node, the service time is T_1, T_2 ($T_1<T_2$), and the memory demand size of working set is M_1, M_2 ($M_1>M_2$). Then in Round Robin algorithm, the average memory demand per unit time of a node is shown as formula (3):

$$\frac{2*T_1(M_1+M_2)+(T_2-T_1)M_2}{T_1+T_2} \quad (3)$$

But in RR-MMMCS algorithm (assume that p_1 can execute with two slices continually), the average memory demand is:

$$\frac{\frac{3}{2}T_1(M_1+M_2)+\left(T_2-\frac{T_1}{2}\right)M_2}{T_1+T_2} \quad (4)$$

Obviously, formula (3) is larger than formula (4).

4.2 MMMCS-P scheduling discipline

Let the tasks execution waiting interval times be exponentially random variables with mean $\frac{1}{\lambda}$. Its cumulative distribution function and probability density function are F(x) and f(x) respectively. Therefore, $\overline{F}(x)$ is a task at time x survival probability.

$$F(x)=1-e^{-\lambda x}, f(x)=\lambda e^{-\lambda x}, \overline{F}(x)=e^{-\lambda x}, \lambda \succ 0 \quad (5)$$

And t_1, as a task's elapsed time, is the time interval from the last execution time to now. Then, a task remaining life can be addressed as follows:

$$F_{t_1}(x)=p\{X-t_1\le x \mid X\succ t_1\}=\begin{cases}\frac{F(x+t_1)-F(t_1)}{1-F(t_1)} & x\ge 0,\\ 0 & x\prec 0\end{cases}$$ Or

$$\overline{F}_{t_1}(x)=\frac{\overline{F}(x+t_1)}{\overline{F}(x)}, \text{when } x\ge 0 \quad (6)$$

The mathematical expectation of remaining life is:

$$M(t)=E\{X=t_1 \mid X\succ t_1\}=\int_0^{\infty} x dF_{t_1}(x)$$

$$=\int_0^{\infty}\frac{\overline{F}(x+t_1)}{\overline{F}(x)}dx=\mu-\int_0^{t_1}\overline{F}(x)dx \quad (7)$$

Especially, $\mu = Ex$ is life expectation,

$$\mu=E(x)=\int_0^{\infty} t f(t)dt \quad (8)$$

From formula (7) and formula (8), obtains the following conclusion:

$$M(t)=\mu-\int_0^{t_1}\overline{F}(x)dx=\int_0^{\infty} t\lambda e^{-\lambda t}dt-\int_0^{t_1} e^{-\lambda t}dx$$

$$=\frac{1}{\lambda}e^{-\lambda t_1} \quad (9)$$

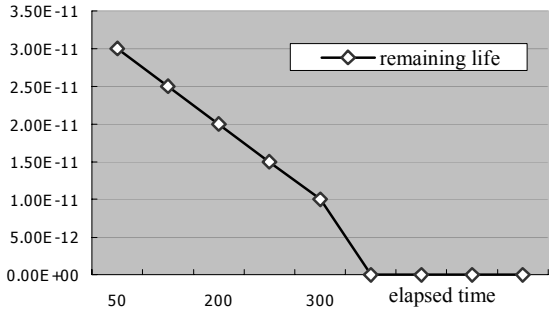

Figure 2: Remaining life with the elapsed time

While the execution time has elapsed t_1, tasks how to be executed once more are depended on formula (9). So, formula (9) can be the foundation of scheduling discipline based on prediction. In figure 2, remaining life reduces along with elapsed time increasing rapidly, and increases along with the task execution gap distributed index, changes like a straight line. Usage of elapsed time replacing remaining life to compute can reduce the computation quantities.

The MMMCS-P scheduling discipline is based on the remaining life prediction and the request of memory size, which is described as follows:

```
While (! Ready-Queue-NULL)
  For (get task_i in the Read-Queue)
      summen=summen+taskmem_i;
end;
avemen=summen/tasknum;
remove the first task_i in the Ready-
Queue;
if(summen/tasknum>3)        continual-
                      time=3;
else if (summen/tasknum>2)
           continual-times_i =2;
      else  continual-times_i =1;
for (get task_i in the Read-Queue)
  if (flag==1) j=i; break;
  else computer(time-elapsed_i);
end;
j=get-max-num(time-elapsed_i);
execution=execute(task_j);
if(execution==paging-fault)
          push   task_j   into   Block-
                      Queue;
else if(execution= =finish)
         push   task_j   into   Finish-
                       Queue;
     else if(continual-times_i==0)
            flag=0;
    {
     q_i =GetTimeofNow();
     push  q_i  into  the  end  of  Rem-
Queue;
     push task_i into the end of Ready-
     Queue;
    }
```

```
            else flag=1;
            push task_j into the front
             of Read-Queue;
end While;
```

Where time-elapsed$_i$ refers to the elapsed time of task i; Rem-Queue is used to record in the last execution time of tasks; flag is used to decide the task computed once more or not, the task can directly carry out (flag=1); otherwise, the last execution finish time of task should be withdrawed and join it into the queue of computing the tasks' remaining life. So, the above algorithm can be summarized as follows: MMMCS-P policy computes the number of CPU slice which a task can execute continually with by the size of memory request of tasks in the CPU waiting queue, then these slices are given to the corresponding task as an integer. When the size of memory request of tasks is smaller than the average, it will defer to elapsed time arrangement relations, where the bigger elapsed time of tasks first to carry out. When the size of memory request of tasks is far bigger than the average memory demand MMMCS-P policy make two or three continual time pieces to achieve a great time piece, then tasks j join this great time piece to computer elapsed time round , perform to carry out.

The goal of MMMCS-P algorithm is to make the task execution waiting gap distribution to be the prediction function which its next time carries out, and the more memory demand task should get more service time, so the larger memory demand and the longer waiting task will be completed more quickly, and the average memory demand in a node is reduced, the length of CPU waiting queue is quickly decreased.

5. Model of the system and the load

5. 1 System stage and simulation system model

Li Xiao ect. developed a simulator of a distributed system with 6 nodes[6], where each local scheduler is CPU-based, memory-based and CPU-memory-based, but the node scheduling discipline is RR only. Using this simulator as a framework, a distributed system with 6 nodes is simulated, where each node scheduling discipline holds four policies: FCFS, RR, RR-MMMCS and MMMCS-P which performance and characters are analyzed in the simulations.

The simulated system is configured with the parameters of table 1, in which the parameter values are similar to the ones of Sun SPARC-20, Sun Ultra 5 and Sun Ultra 10 workstations. The migration related costs match the Ethernet network service times for SPARC-20 workstation.

Table1: Parameters of network and node in the simulated distributed system

The speed of CPU	100 MIPS
The size of the memory(MS)	64 MBytes
The size of operating system memory space (U-sys)	16 MBytes
The size of available user memory space	MS-U-sys
Memory page size	4KBytes
Ethernet connection	10Mpbs
Paging fault service time	10ms
Context switch time	6ms
A remote execution cost	0.1ms
Time slice of CPU local scheduling	0.1sec

The remote execution overhead matches the 10 Mbps Ethernet network service times. We have conducted experiments on an Ethernet network of 100 mbps. However, compared with the experiments on the 10 Mbps Ethernet, we found that little performance improvement was gained for all policies because it is an insignificant factor for performance degradation in our experiments[7].

Each task is in one of the following states: "ready", "execution", "paging", "data transferring", or "finish". When a page fault happens in an execution of a task, the task is suspended from the CPU during the paging service. The CPU service is switch to a different job. When there are several page faults happening, they will be served in FIFO order. We have following conditions and assumptions in the discussion:

- Each computing node maintains a global load sharing 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 task execution. The policy is to find the most lightly loaded node in the distributed system.
- Assume that the memory allocation for a task is done at the arrival of the task.
- The memory threshold of a task is 40% of its request memory size.
- Assume that page faults are uniformly memory threshold of tasks in node is equal to or larger than the available memory space of the node, each task in the node will cause page faults at a given page fault rate. The page fault rates of tasks range from 0.06 to 4.1 per million instructions in this

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

- The sender-initiated policy is used, where a heavily loaded node attempts to transfer work to a lightly loaded node.

5.2 Model of system load

In order to explain the effect of load sharing, node scheduling discipline policy and the variance of memory demand and service time on system performance, different trace files are used in the simulations. 8 traces[6] are used in the experiments, and each of which was collected from one workstation on different daytime intervals. The tasks in each trace were distributed among 6 workstations. And the mean memory demand for each trace is 1 Mbytes and 4 Mbytes. Each task has 5 items: {Arrival-Time, Arrival-Node, Memory-Size, Service-Time, Task-Type} Arrival-Time refers to the arrival time of task, and the precision of time is 10 ms; Arrival-node means the node which the task arrive at, and there are 6 nodes in the simulation experiment which are denoted from 0 to 5; Memory-Size means the size of memory demand of task, but the practical memory size the system allocating to the task is decided by the status of memory demand; Service-Time means the CPU service time used to complete the task; Task-Type is the type of tasks, this parameter shows whether the task can be migrated in the simulation system.

For the evaluation of the effect of variance of system parameter and workload on performance, the above traces are modified in the experiment. Assume that the inter-arrival time of tasks is exponential distributed in the new traces, and the mean interval is $1/\lambda$ .The service time and the size of memory demand of tasks is also exponential distributed, the mean time is $1/\mu$, and the mean size is $1/\alpha$. Their coefficients of variance are C_s and C_m respectively.

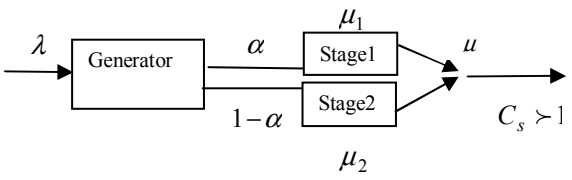

Figure 3: Generate the variance coefficient

Furthermore, a two-stage hyper-exponential model (shown in figure 2) is used to generate service time and the size of memory demand coefficient of variance greater than 1. μ is the mean service time (or the mean size of memory demand); μ_1 is the mean service time (or the mean size of memory demand) of stage 1, α is the selection probability of this stage; μ_2 is the mean service time (or the mean size of memory demand) of stage 2, and the selection probability of this stage is $1-\alpha$; C_s is the desired coefficient of variance which value is set from 0 to 4. μ_1, μ_2 can be computed by the following formulae:

$$\mu_1 = \mu - \mu(1-\alpha)\left((C_s^2-2)/(2\alpha(1-\alpha))\right)^{1/2}$$

$$\mu_2 = \mu + \mu\alpha\left((C_s^2-2)/(2\alpha(1-\alpha))\right)^{1/2}$$

The related study[7] has shown that a specify value of α only affects the absolute values of the results but not the relative performance, and the value should satisfy the following relationship: $(1+\alpha)/(1-\alpha) > C_s^2$ In this paper, α =0.90.

6. Performance analyses

Based on the model of system and workload, the impact of seven different policies is shown in Table 2:

Table 2: Load sharing policies and scheduling discipline policies

1	CPU-RR	CPU-based load sharing and Round Robin scheduling discipline.
2	CPU-RR-MMMCS	CPU-based load sharing and more memory-request more CPU Slice based on Round Robin
3	CPU-MMMCS-P	CPU-based load sharing and more memory-request more CPU slice based on predicting
4	CPU-MEM-FCFS	CPU-MEM-based load sharing and first come first service
5	CPU-MEM-RR	CPU-MEM-based load sharing and Round Robin scheduling discipline.
6	CPU-MEM-RR-MMMCS	CPU-MEM-based load sharing and more memory-request more CPU Slice based on Round Robin
7	CPU-MEM-MMMCS-P	CPU-MEM-based load sharing and more memory-request more CPU slice based on predicting

The experiment is simulated with the above 8 traces. Figure 4 and figure 5 show that the load sharing policy based on CPU-memory is beneficial to both CPU - bound and memory- bound jobs, RR-MMMCS and MMMC-P policies are also more beneficial to system based on CPU, CPU-memory than FCFS and RR.

As shown in figure 4, when the memory demand of tasks isn't large enough, load sharing policy based on

CPU-memory can balance the memory workload well and make the memory workload of every task to be light, and the page fault rate is low. So, RR-MMMCS will get little improvement, CPU-MEM–RR and CPU-MME-RR-MMMCS have little difference in performance. But, for load sharing system based on the CPU, the load sharing policy is lack of consideration to memory resource, which takes full advantage of RR-MMMCS policy. Thus, much performance improvement is gained by RR-MMMCS policy, and figure 4 shows that the performance is improved 22.7% mostly. Furthermore, MMMC-P policy can deal with tasks scheduling in a node very well, and reduces tasks' waiting time quickly, it cuts mean response time in a single node largely. So, much performance improvement is gained by the MMMC-P policy. The experimental results show that compared with RR-MMMCS, MMMCS-P policy is more suitable for the CPU-bound jobs.

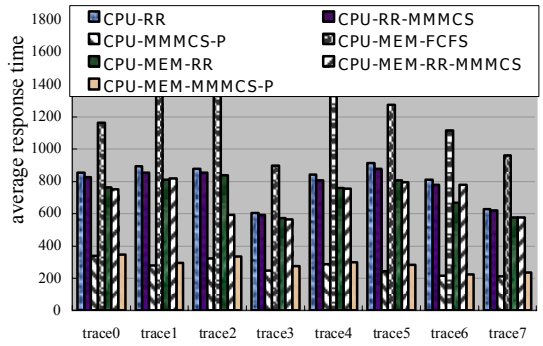

Figure 4: Average response time of 8 traces with the mean memory demand 1 Mbytes

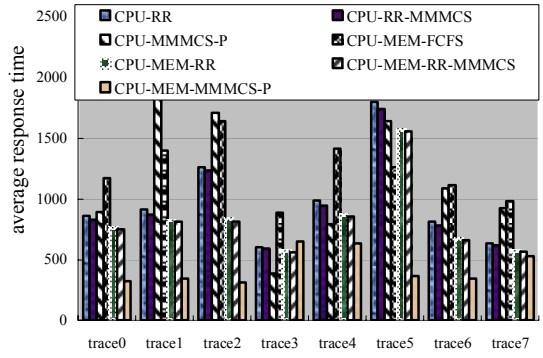

Figure 5: Average response time of 8 traces with the memory demand 4 Mbytes

In figure 5, for memory-bound jobs, MMMCS-P and RR-MMMCS policies have much effect on the performance improvement of distributed system. Because of increased prediction mechanism, MMMCS-P policy gains more. For load sharing system based on CPU, although RR- MMMCS can improve the usage of memory resource in a single node, it may not improve the performance of the whole distributed system observably because the load sharing policy might not balance the memory workload in the distributed system efficiently. In addition, the figures also show that the efficiency of FCFS is even worse.

Then, the traces are modifies to simulate and analyze all the policies in other aspects. Figure 6 and figure 7 show the effect of variance coefficient of memory demand and service time on variant policies.

Figure 6 shows the effect of variance coefficient of memory demand on the mean response time of tasks. a conclusion from figure 6 is that: the effect of variant coefficient of memory demand on load sharing based on CPU-memory is more than that on load sharing based on CPU, and the effect on RR-MMMCS is also more than that on RR, furthermore, the effect on MMMCS-P is more than that on RR-MMMCS. According to the figure, RR-MMMCS scheduling discipline has higher performance when the variance coefficient is between 2 and 3, and CPU-MEM-MMMCS-P along with the memory demand coefficients of variation change, the change tendency is steady, which indicates that the RR-MMMCS is adaptive to the instability on memory demand from the point of performance although it is sensitive to the variance.

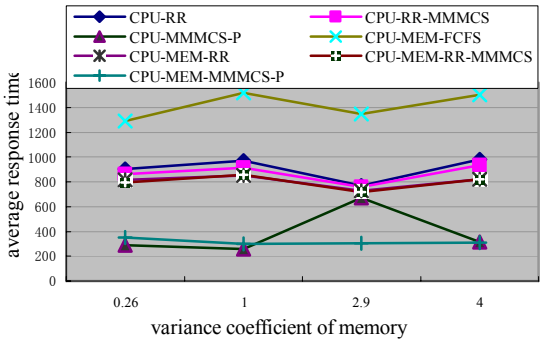

Figure 6: Performance sensitivity to the memory demand coefficient of variance.

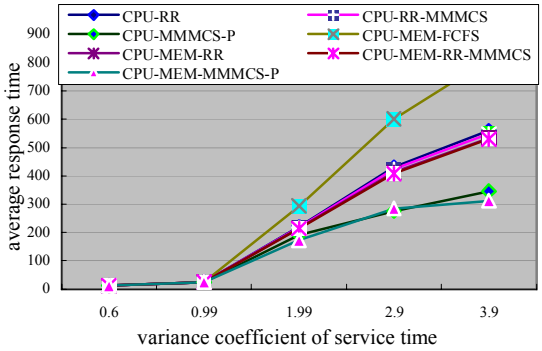

Figure 7: Performance sensitivity to the service time coefficient of variance.

In figure 7, all scheduling disciplines performance becomes lower with the variance coefficient of service time increasing. And the MMMCS-P still has the highest performance during the variance coefficient changing. So, a conclusion from this figure: the load sharing based on CPU-memory and RR-MMMCS, MMMCS-P disciplines are also more adaptive to the instability of service time from the point of performance.

The differences among the scheduling disciplines with 8 traces show that the load sharing based on CPU-memory, and RR-MMMCS and MMMCS-P scheduling disciplines have high performance. Obviously, FCFS is always the worst choice in our experiments.

7. Conclusions

In light of the characteristics of the load sharing based on CPU-memory, new CPU local scheduling policies RR-MMMCS and MMMCS-P are developed. According to the simulations and analysis of the 7 policies with the variant traces in the experiment, several important conclusions are gained as follows:

- The load sharing policy based on CPU-memory and RR-MMMMCS and MMMCS-P scheduling disciplines have higher performance in both the CPU-bound and the memory-bound jobs.
- The performance of RR-MMMCS policy is higher than that of FCFS and RR in load sharing system based on both CPU and CPU-memory.
- MMMCS-P policy regardless of CPU based or CPU-memory based, is superior to RR-MMMCS policy, and clusters performance obtains a bigger enhancement.
- The load sharing based on CPU-memory and RR-MMMCS are more sensitive to the variance of service time and memory demand of tasks, but the policies are more adaptive to the instability of system from the point of performance.

References

[1] Goscinski. A, *Distributed Operating Systems*, The Logical Design. Addison-Wesley Publishing Company, 1991.

[2] M. Bozyigita, "History-driven dynamic load balancing for recurring applications on network of workstations", *Systems and Software*, Vol. 2, 2002, pp. 61-72.

[3] S. P. Dandamudi, "The Effect of Scheduling Discipline on Dynamic Load Sharing in Heterogeneous Distributed Systems", *Fifth IEEE International Workshop on Modeling, Analysis, and Simulation of computer and Telecommunications Systems,* (MASCOTS'97), 1997, pp. 17-24.

[4] Subash. T and IndiraGandhi. S, "Performance analysis of scheduling disciplines in optical networks", *Wireless and Optical Communications Networks, 2006 IFIP International Conference*, 2006.

[5] M. H. Balter and A. B. Downey, "Exploiting process lifetime distribution for dynamic load balancing", *ACM Transaction on Computer Systems*, Vol. 15, 1999, pp. 253-285.

[6] L. Xiao, X. Zhang, and Y. Qu, "Effective Load Sharing on Heterogeneous Networks of Workstations", *Proceedings of 2000 International Parallel and Distributed Processing Symposium*, Cancun, 2000, 1-5.

[7] 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,* (ICDS'2000), Tai Pei, Tai Wan, April 10-13, 2000.

[8] S. P. Dandamudi, "The Effect of Scheduling Discipline on Sender-Initiated and Receiver-Initiated Adaptive Load Sharing in Homogeneous Distributed Systems", *Technical Report, School of Computer Science*, Carleton University, TR-95-25 1995.

[9] L. Xiao, S. Chen, and X. Zhang, "Adaptive Memory Allocations in Clusters to Handle Unexpectedly Large Data-intensive Jobs", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No.7, pp. 577-592.

[10] Sanguandikul. N, and Nuparios. N, "Implicit Information Load Sharing Policy for Grid Computing Environment", *Advanced Communication Technology*, (ICACT 2006), 2006.