This paper is submitted to the Workshop on Computer system of ICS2002

Involving Memory Resource Consideration in Workload Distribution for Software DSM Systems

Tyng-Yeu Liang[+] (Assistant Professor), Yen-Tso Liu (Ph.D. Candidate)
Yi-Ching Chen (Master), Ce-Kuen Shieh (Professor)

[+]Department of Computer Science and Information Engineering, Leader University
No. 188, Sec. 5, An-Chung Road, Tainan, Taiwan
Department of Electrical Engineering, National Chung Kung University
No. 1, Ta-Hsueh Road, Tainan, Taiwan
{lty,andy,ycchen,shieh}@hpds.ee.ncku.edu.tw
Phone: 886-6-2550217 Fax:886-6-2748678

Abstract-This paper is aimed at resolving the problem of workload distribution for software distributed shared memory (DSM) systems. According to the past methods, DSM systems usually took only CPU resource into account for workload distribution. They distributed program threads based on the computational capability of processors to make load balance, expecting thereby the minimal execution time of applications. However, the cost of memory accesses is an important factor for program performance in addition to computational cost. If processors cannot afford enough physical memory space to cache all the data needed by threads, these processors must perform page replacements to make memory space for data caching while executing the threads. The execution of the threads absolutely will be delayed due to the latency of performing page replacement, and then the execution time of programs will increase. Therefore, only accomplishing load balance cannot exactly guarantee the minimal execution time of programs. In this paper, we propose a new workload distributing method, which simultaneously considers CPU resource and memory resource, for DSM systems. Our experimental results show that memory resource is indeed an important consideration for workload distribution on DSM systems, and our method is more effective for minimizing the execution time of DSM applications than the others considering only either CPU resource or memory resource.

Keywords: workload distribution, distributed shared memory, page replacement, load balance, memory resource

+: the corresponding author of this paper

1. Introduction

Workload distribution is an important issue for the performance of network-based computation. When an application is parallilized on a set of machines, it is necessary to carefully distribute the workload of this application to make the working machines complete their assigned work at the same time, thereby expecting the minimal execution time of the application. However, to accomplish this goal on network-based working environments is a big challenge. One reason is that machines on computer network are not identical in resource capability such as CPU power, physical memory space, I/O and so on. Even if all the working machines are assigned with the same amount of work, they still will not complete their assigned work at the same time due to different resource capabilities. Hence, a better solution is to distribute program workload based on the resource capabilities of processors. Another reason is that computer network is a shared working environment and any application has to compete with other applications for system resource. Any user job submission or completion can cause that the residual resource capabilities of processors change. Therefore, dynamically adapting the workload distribution of applications according to resource capability changes is required while executing these applications. Obviously, workload distribution is not a trivial job for users even system designers based on the previous reasons.

Recently, software distributed shared memory (DSM) [1][2][3][4][5][6] has successfully provides an easy user interface for network-based computation. With the support of DSM run time systems, programmers can make use of shared variables instead of message passing to develop their applications on computer network. Consequently, they can concentrate on developing application

algorithms but not handling data transfer between processes. The complexity of programming on computer network has also been simplified by DSM systems. On the other hand, modern DSM systems [7][8][9] support multithreading per node and thread migration to automatically adapt workload distribution in order to provide good performance for user applications. Therefore, many applications that need massive computation such as image processing, numeric analysis and scenic simulation have been implemented on DSM systems.

However, the past workload distributing methods [10][11][12][13] proposed for DSM systems usually only took CPU resource into account. According to these methods, DSM systems usually distributed working threads of programs based on the computational power of processors to make load balance, thereby expecting the minimal execution time of the programs. They never cared about whether processors have enough memory resource to meet thread memory demand. However, the cost of memory access is an important factor for program performance in addition to computational cost. If processors have not enough physical memory space to caching all the data needed by working threads, these processors need to consecutively execute page replacement for data caching while executing the threads. Although the processors still can complete the work of the threads, however the execution of these threads will be absolutely delayed by the latency of executing page replacements. Therefore, load balance cannot necessarily guarantee the minimal execution time of programs. Ignoring memory resource consideration, DSM systems probably make long-termed wrong decisions in workload distribution, and then degrade program performance due to the latency of executing page replacements even when load balance is achieved.

This paper is aimed at resolving the problem of workload distribution for software DSM systems as previous described. We develop a new workload distributing method for DSM systems in this paper. First, we analyze the execution of programs running on DSM systems. Then, we derive a set of mathematical formulas for precisely estimating the execution time of programs with considering both of CPU resource and memory resource. With these derived mathematical formulas, the proposed method can find out good thread distributing ways to effectively adapt workload distribution and enhance program performance. We have implemented this new method on a test bed called Teamster that is a DSM system built on a cluster of Intelx86 PCs connected with Ethernet network. In addition, we have implemented three loop applications, i.e., SOR, Jacobi and Matrix Multiplication to evaluate the effectiveness of the proposed workload distributing method. The experimental results shows that memory resource consideration indeed is very important for workload distribution on DSM systems. In addition, our method is more effective for minimizing the execution time of the test programs than the others considering only either CPU resource or memory resource.

The rest of this paper is organized as follows. Section 2 discusses work related to the field of DSM workload distribution. Section 3 introduces the proposed method and Section 4 describes the implementation of the proposed method. Section 5 discusses the results of performance evaluation. Finally, Section 6 gives the conclusions of this paper and our future work.

2. Related Work

Currently, the DSM systems that support dynamic workload distribution are CVM, JIAJIA and Cohesion. CVM [14] focuses on making load balance and reducing data consistency communication. In order to accomplish these two goals, CVM distributes program threads onto processors based on the computational power of processors and the computational demand of threads. Basically, a processor with more computational power is assigned with more program threads. In addition, CVM locates on the same node the pairs of threads that show the highest degree of mutual data sharing, expecting thereby a maximum communication reduction. On the other hand, JIAJIA [15] assumes that processors have enough physical memory space to hold the data needed by threads. This system considers only CPU resource while distributing program workload. Same as CVM, JIAJIA also distributes program workload onto processors based on the computational power of processors. As to Cohesion [16], the job of distributing program workload is divided into two phases. One is migration phase and the other is exchange phase. In the migration phase, Cohesion estimates the workload of each processor and then migrate threads from the heavily loaded node to the lightly loaded node to reduce load imbalance cost. In the exchange phase, Cohesion locates on the same node the pairs of threads that show the highest degree of mutual data sharing to reduce communication cost via thread exchange.

In addition to the DSM studies, some researches of distributed systems have discussed the problem of workload distribution. Peris [17] analyzed the influence of physical memory size on system performance. He proposed a stochastic model to predict the execution cost of parallel

programs including memory access cost. His study shows that wrong workload distribution will increase overhead of memory accesses and then degrade system performance. However, the proposed model is difficult to make it practical. On the other hand, Zhang [18] proposed making use of CPU load index and memory load index for workload distribution. When a new user job is created on a machine, a CPU-based policy is used for workload distribution if memory load index shows that this machine has enough memory resource capability to support the user job. Otherwise, a memory-based policy is added into workload distribution. This job will be migrated to another machine that can support enough memory resource or blocked in a waiting queue until local node can provide enough memory resource for its memory demand based this memory-based policy. Although this method considers memory resource into workload distribution, its aim is to promote the utility of system resource and throughput. However, the main purpose of workload distribution on DSM systems is to minimize the execution time of user applications. In addition, the working threads of user programs usually have to cooperate with each other to complete the work of the same programs. Therefore, Zhang's method is not suitable to apply to DSM systems.

As the above description, the previous DSM methods always considered the computation time and communication time of programs while distributing program threads onto processors. They never were concerned with the problem of whether processors have enough physical memory space for caching the data needed by threads. On the other hand, the other methods proposed for distributed systems are not suitable for DSM systems. Therefore, it is necessary to study how to develop an effective method to resolve the problem of workload distribution for DSM systems.

3. The Proposed Workload Distributing Method

DSM applications can be categorized into three kinds. They are *fork-join*, *run-to-complete* and *iterative* [19]. These three different kinds of applications have different execution characteristics. Consequently, it is necessary to design different workload distributing methods for these three different kinds of applications. Since working threads of iterative applications usually repeat the same things, this program characteristic is helpful to predict thread behavior and adapt workload distribution to enhance program performance. Therefore, we focus on resolving the problem of workload distribution for iterative applications running on DSM systems in this paper.

We analyze the execution of iterative DSM applications and derive a set of mathematical formulas in this paper. Using these derived formulas, the proposed working distributing method can precisely predict the execution time of programs running with a candidate thread distributing way, and then decide if this thread distributing way is helpful to minimize the execution time of programs. Then, the proposed method can search good thread distributing ways for DSM systems to effectively enhance the performance of user programs. Our analysis, the derived formulas and the algorithm of adapting workload distribution are described as follows.

3.1 Analysis

In DSM, an iterative problem is usually partitioned into a number of threads. Then, these threads are distributed onto processors for parallel execution. When the threads finish their work in the current iteration, they have to join at a barrier before entering the next iteration. Let $T(x,k)$

represent the time that node x executes its local threads in the kth iteration. The execution time of the kth iteration and the total execution time of the program can be estimated as Eq(1) and Eq(2), respectively. Since iterative threads usually have regular behavior and data access pattern, $T(x,k)$ theoretically will keep the same value in each iteration if system status and the mapping of threads onto processors do not change. Consequently, $T(x,k)$ can be simplified as $T(x)$ and the total execution of program can be simply predicted as Eq(3).

$I(k)$=Max$\{T(x,k) \mid x = 1..M, M$ is the number of execution nodes$\}$ Eq(1)

$P.E.T = \sum_{k=1}^{N} I(k)$, N is the number of iterations. Eq(2)

$P.E.T = N * IT$, Eq(3)

where IT is Max$\{T(x) \mid x = 1..M, M$ is the number of execution nodes$\}$

Basically, $T(x)$ can be divided into three parts. They are computation time, communication time and memory time as shown in Eq(4). Computation time (i.e., $T_{comp}(x)$) is the time that node x executes the computational work of local threads. Communication time (i.e., $T_{comm}(x)$) is the time that node x communicates with other nodes for data sharing between threads. Memory time (i.e., $T_{mem}(x)$) is the time that node x executes page replacements to cache the data accessed by its local threads.

$T(x) = T_{comp}(x) + T_{comm}(x) + T_{mem}(x)$ Eq (4)

Let S_x be the set of threads running on node x, t_{comp}^{jx} be the computation time of thread j running on node x, t_{mem}^{jx} be the time of executing page replacements for caching the data accessed by thread j on

node x. Then, $T_{comp}(x)$ and $T_{mem}(x)$ can be further derived as Eq(5) and Eq(6).

$$T_{comp}(x) = \sum_{j \in S_{ix}} t_{comp}^{jx} \qquad \text{Eq(5)}$$

$$T_{mem}(x) = \sum_{j \in S_x} t_{mem}^{jx} \qquad \text{Eq(6)}$$

Moreover, the time of executing page replacement for caching the data needed by working threads can be further divided into two parts according to our analysis. One is the time of scanning physical memory to search least-recently used (LRU) data pages and then swapping out the data pages from physical memory to disk. Another is the time of swapping in the data pages needed by threads from disk to physical memory. Therefore, Eq(6) can be further derived as follows.

$$T_{mem}(x) = \sum_{j \in S_x} f^{jx} \times \left(t_{spo}(x) + t_{spi}(x)\right) \quad \text{Eq(7)}$$

f^{jx} ：the number of page replacements that node x executes for caching the data needed by thread j.

$t_{spo}(x)$ ：the average time of searching LRU data pages and swapping out the data pages on node x.

$t_{spi}(x)$ ：the average time of swapping in pages on node x.

In this paper, we assume there is no data sharing between threads to simplify our work and put our attention to the influence of memory resource on workload distribution. Therefore, we omit $T_{comm}(x)$ while implementing the proposed workload distributing method on a test bed.

3.2 The Algorithm of Adapting Workload Distribution

First, the workload-distributing algorithm of the proposed method searches the node with the most value of $T(x)$ and the node with the least value of $T(x)$ to respectively be source node and

destination node by using the previous formulas. Then, it sets the length of program critical path as the most value of $T(x)$. Second, it simulates to migrate one thread from source node to destination node and estimate the length of program critical path again. If the length of program critical path is not reduced, the algorithm will stop adapting workload distribution. Otherwise, it continues migrating threads from source node to destination node until the length of program critical path cannot be reduced further. Then, it searches a new pair of source node and destination node and repeats the previous operations. After finishing this workload-distributing algorithm, a new thread distributing way, that is a set of numbers of threads assigned onto each processor, can be gained and used to adapt program workload distribution via thread migration.

4. Implementation

We have implemented the proposed workload distributing method on a test bed, called Teamster. We took advantage of the information mechanism used by Teamster to collect the information necessary for workload distribution such as the computational power of processors, the computational demand of threads, the amount of available physical memory space on each processor, thread access pattern and so on. Using the collected information and the proposed method, Teamster dynamically adapts program workload distribution to minimize the execution time of programs. The details about system characteristics, information connection and the architecture of workload distribution are described as follows.

4.1 System Overview

Teamster [20] is a user-level DSM system built on a cluster of Intel 80x86 PCs connected with Ethernet network. This DSM system provides a single and global address space for user applications to prevent data address translation between nodes. In addition, Teamster supports multiple memory consistency protocols, i.e., *sequential* and *eager released* to manage data consistency in order to reduce data-consistency communication. On the other hand, Teamster supports multiple processors per node and multithreading per node to execute user programs. Moreover, it makes use of a diffusion reorganization mechanism to dynamically add nodes into or delete nodes from program execution, and migrate user threads from one node to another. The main purpose of this support is to increase the utility of system and enhance program performance.

4.2 Information Collection

The information necessary for the proposed workload distributing method can be classified into *static information* and *dynamic information*. Static information includes CPU power, available physical memory space and the average time of executing page replacements. The static information of working machines is collected before the setup of DSM systems. On the other hand, dynamic information consists of the computation time of each thread, the data access pattern of threads and the residual memory capability of each processor. The dynamic information is collected during program execution. Since Teamster is built on user level, it is easy to modify this run time system to collect the dynamic information. We make use of a mechanism called *active correlation track* [21]

to collect the data access pattern of threads. We can know the working sets of threads and derive the amount of thread memory demand with the access pattern of threads. In addition, we make use of system primitive such as *se* to fetch the system information of residual physical memory space. Then, we can predict the memory time of thread *j* on node *x* according to the memory demand of thread *j*, the residual memory resource capability of node *x* and the average time of executing page replacements on node *x*.

On the other hand, the computation time of a thread is measured as the time that this thread arrives at the end of iteration. When a thread is fetched form the local queue to execute, the start time is recorded. After this thread reaches the end of iteration, the arrival time also is recorded. Then, the computation time of this thread is estimated as the interval between the start time and the arrival time. Since the computational capability of processors is non-identical, the computation time of a thread is different from node to node. When a thread *j* is migrated from node *x* to node *y*, the computation time of this thread on node *y*, i.e., t_{comp}^{jy} can be estimated as $t_{comp}^{jx} * power_x / power_y$ where $power_x$ and $power_y$ are the computational power factors of node *x* and *y*.

The above method may include the time of requiring remote pages and executing page replacements into the computation time of threads. Fortunately, the thread scheduling of Teamster is non-preemptive. Consequently, the time of requiring remote page and executing page replacements can be measured. Then, we can get rid of the time from the computation time of threads.

4.3 The Architecture of Workload Distribution

The implementation of the proposed workload distributing method adopts a central architecture. That is, there is a root node is responsible for coordinating with the other nodes to adapt workload distribution of user programs. After initiating an iterative application, the root node distributes working threads to processors for execution. Then, each node begins to execute local threads and collect the information necessary for workload distribution. When a node finishes the work of its local threads, this node sends a barrier arrival message tailed with the local connected information to the root node. After receiving the arrival messages and the collected information from all the execution nodes, the root node searches a new thread distributing ways according to the algorithm of adapting workload distribution and then broadcast the searching result. Based on this new thread distributing way, the processors exchange threads by using the mechanism of thread migration to adapt workload distribution of the executed program. Finally, all the working threads are resumed to execute for the next iteration.

Basically, it is ideal to perform the adaptation of workload distribution at the end of each iteration. However, the cost of adapting workload distribution is too big. In order to reduce this cost, our current implementation is to perform the mechanism of adapting workload distribution only once during program execution. A compromise method is to develop a stochastic policy to trigger the adaptation of workload distribution.

5. Performance Evaluation

We have implemented three iterative applications, i.e., SOR, Jacobi and Matrix Multiplication to evaluate the effectiveness of considering memory resource into workload distribution. These three applications were executed on 4 Xeon-500Mhz PCs connected with 100Mbps fast Ethernet network. The parameters of these three applications are listed in Table 1. Table 2 is the amount of physical memory space provided by each node for executing these three applications, respectively. In addition, we ran each application with three different workload distributing methods, i.e., cpu-based, memory-based and cpu&memory-based. The cpu-based method considers only CPU resource but ignores memory resource. In contrast, the memory-based method distributes threads according to the amount of physical memory space that each processor can provide for executing DSM applications. Basically, a processor having more free physical memory space is assigned with more threads based on the memory-based method. Finally, the cpu&memory-based method is our workload distributing method. Table 3 is the number of threads assigned onto each processor while respectively applying these three methods for the test applications. The experimental results are shown from Figure 1 to Figure 3.

	Problem size (double)	Thread number	Memory demand per thread (Mbytes)	Loop count
SOR	6144x6144	32	9	5
Jocobi	6144x6144	32	9	5
MatrixMultiplication	3072x3072	32	6.75	5

Table 1. The parameters of test applications

Unit: Mbyes	Node 0	Node 1	Node 2	Node 3
SOR	400	36	45	72
Jocobi	400	36	45	72
MatrixMultiplication	400	27	33.75	54

Table 2. The amount of physical memory space provided by each processor for test applications\

(node0, node1, node2, node3)	Cpu-based	Memory-based	CPU&memory based
SOR	(8,8,8,8)	(15,4,5,8)	(14,5,5,8)
Jocobi	(8,8,8,8)	(15,4,5,8)	(11,6,6,9)
MatrixMultiplication	(8,8,8,8)	(15,4,5,8)	(10,7,7,8)

Table 3. The number of threads distributed onto each processor

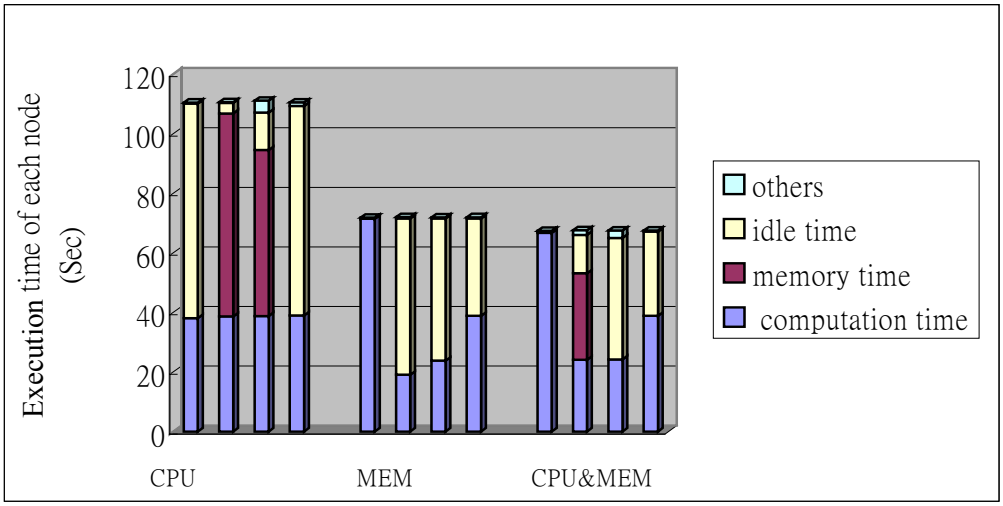

Figure 1. The experimental result of SOR

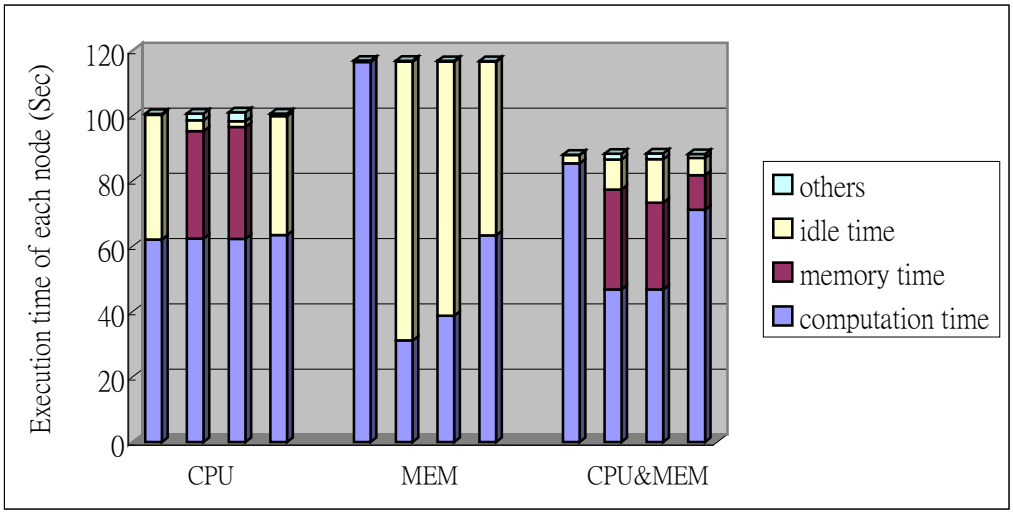

Figure 2. The experimental result of Jacobi

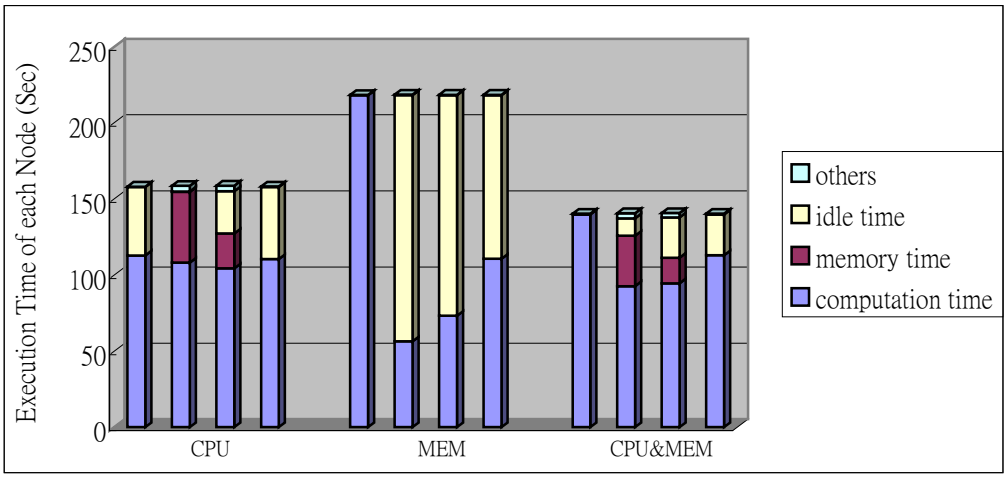

Figure 3. The experimental result of Matrix Multiplication

The experimental results of the three applications have a common situation, which is the computation time of four execution nodes is the same while using the cpu-based method. Although load balance is achieved, however the execution time of the three applications is not necessarily reduced by load balance. Since node 2 and node 3 have not enough physical memory space for thread memory demand, the memory time of these two nodes delay the completion of the test applications although node 1 and node 4 have finished their work more early than node 2 and node 3. The main reason for this result is the cpu-based method cares about only load balance but ignore whether processors are able to provide enough physical memory space for meeting thread memory demand. By contrast, the memory-based method results in that four execution nodes have no memory time in the three test programs since the number of threads assigned to a processor is dependent on the amount of physical memory space that this processor can provide for DSM applications. However, this result has different effect on the performance of the test programs.

Compared to the cpu-based method, the memory-based method minimizes the execution time of SOR but increases the execution time of Jacobi and Matrix Multiplication. The main reason is that node 0 is assigned with much more workload than the other three nodes. Consequently, if the execution time of the test programs can be minimized is dependent on whether the loss from load imbalance can be compensated by the gain from no memory time. Obviously, the consideration of CPU resource is more important than the consideration of memory resource for the Jacobi and Matrix Multiplication programs. Compared with the previous two methods, the cpu&memory based method minimizes the execution time of the three test programs. That is because this method considers not only CPU resource but also memory resource while estimating the execution time of programs. Therefore, Teamster can make better decisions in workload distribution for executing the test programs by using this method compared to the other two methods. Making a short summary, our experimental results shows that simultaneously considering both of CPU resource and memory resource is more effective for minimizing the execution time of programs than considering either one of these two factors.

6. Conclusions and Future Work

In this paper, we point out the importance of considering memory resource into workload distribution to the performance of DSM programs. On the other hand, we analyze the execution of DSM programs and develop a new workload distributing method for DSM systems according to our analysis. In addition, we have implemented the proposed workload distributing method on a test bed,

i.e., Teamster, and developed a set of applications to evaluate the effectiveness of considering memory resource into workload distribution. Our experimental results shows that memory resource into workload distribution indeed is significant for the performance of DSM programs. The proposed workload distributing method simultaneously taking CPU resource and memory resource into account is more effective for reducing the execution time of programs running on DSM systems than the other methods considering only either CPU resource or memory resource.

The workload distributing method proposed in this paper does not take communication time of processors into account. However, this factor is also very important to the performance of DSM systems. We will combine this factor with computation time and memory time to develop an advanced workload-distributing method for DSM systems in the future.

Reference:

[1] K. Li. IVY: A shared virtual memory system for parallel computing. *In Proceedings of the 1988 International Conference on Parallel Processing (ICPP'88)*, p. 94-101, 1988.

[2] J.B. Carter, J.K. Bennett and W. Zwaenepoel. Implementation and Performance of Munin. *In Proceedings of 13th ACM Symposium on Operating System Principles*, p. 152-164, 1991.

[3] B.N. Bershad, M.J. Zekauskas. The Midway Distributed Shared Memory System. *In: Proceedings of IEEE COMPCON Conference*, p. 528-537, 1993.

[4] Weiwu Hu, Weisong Shi, Zhimin Tang. JIAJIA: An SVM System Based on A New Cache Coherence, Protocol. In: *Proceedings of the High Performance Computing and Networking (HPCN'99)*, p.463-472, 1999.

[5] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel, TreadMarks: Shared Memory Computing on Networks of Workstations. *In IEEE Computer*, 29

(2), p. 18-28, 1996.

[6] E. Speight and J.K. Bennett. Brazos: A third generation DSM system. *In Proceedings of the 1997 USENIX Windows/NT Workshop*, p. 95-106, August 1997.

[7] Roy Friedman, Maxim Goldin, Ayal Itzkovitz, Assaf Schuster. Millipede: Easy Parallel Programming in Available Distributed Environments. *In Software: Practice and Experience 27 (8)*, p. 929-965, 1997.

[8] Jyh-Chang Ueng, Ce-Kuen Shieh, Su-Cheong Mac, An-Chow Lai, Tyng-Yeu Laing. Multi-threaded Design for a Software Distributed Shared Memory System. *IEICE Transaction on Information and Systems* E82-D (12), p. 1512-1523, 2000.

[9] P. Keleher. The Coherent Virtual Machine. *Technique Report Maryland TR93-215*, Department of Computer Science, University of Maryland, 1995. http://www.cs.umd.edu/~keleher/papers.html.

[10] K. Thitikamol and P. Keleher. Thread migration and load balancing in non-dedicated environments. *In Proceeding of the 14th International Parallel and Distributed Processing Symposium,* p. 583-588, May 2000.

[11] Alex Dubrovski, Roy Friedman and Assaf Schuster, Load Balancing in Distributed Shared Memory Systems. *In International Journal of Applied Software Technology*, vol 3, p. 167-202, March 1998.

[12] C. Lai, C. K. Shieh, J. C. Ueng, Y. T. Kok, and L. Y. Kung, Load Balancing in Distributed Shared Memory System. *In IEEE International Performance, Computing, and Communications Conference*, Arizona, U.S.A., p. 152-158, February 1997.

[13] Jeffrey K. Hollingsworth and Peter J. Keleher, Prediction and Adaptation in Active Harmony. *In The 7th International Symposium on High Performance Distributed Computing*, April 1998. http://www.cs.umd.edu/~keleher/papers.html.

[14] Kritchalach Thitikamol and Pete Keleher Thread Migration and Communication Minimization in DSM Systems. *IEEE Proceedings*, p. 487-497, 1999.

[15] Weisong Shi and Zhimin Tang. Dynamic Computation Scheduling for Load Balancing in Home-based Software DSMs. *In Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'99)}*, IEEE Computer Press, Perth, Australia, June, 1999.

[16] Tyng-Yeu Liang, Ce-Kuen Shieh, Deh-Cheng Liu. Scheduling Loop Applications in Software Distributed Shared Memory Systems, *IEICE Transaction on Information and Systems*, vol. E83-D, no.9, p. 1721-1730, September 2000.

[17] Vinod G.J. Peris, Mark S. Squillante, and Vijay K. Naik. Analysis of the Impact of Memory in

Distributed Parallel Processing Systems". *In Proceedings of the 1994 ACM SIGMETRICS Conference*, p. 5-18, February 1994.

[18] Li Xiao, Songqing Chen, and Xiaodong Zhang. Dynamic Cluster Resource Allocations for Jobs with Known and Unknown Memory Demands. *IEEE Transactions on Parallel and Distributed Systems, Vol.13, No.3*, p. 223-240, March 2002.

[19] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on A Cluster of Workstations. *In Proceedings of First Symposium on Operating Systems Design and Implementation*, p. 201-212, 1994.

[20] J. B. Chang and C. K. Shieh. Teamster: A Transparent Distributed Shared Memory for Cluster Symmetric Multiprocessors. *In Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*. p. 508-513, 2001

[21] Kritchal Thitikamol, Peter J. Keleher. Active Tracking Correlation. *In Proceedings of the 19th International Conference on Distributed Computing Systems*, 1999.

http://www.cs.umd.edu/~keleher/papers.html.