多線頭分散式共用記憶體系統上高效率同步機制之研究
Efficient Synchronization Mechanisms for Multithreaded Distributed Shared Memory Systems

翁志昌
Ueng Jyh-Chang
Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan
ujc@eembox.ncku.edu.tw

謝錫堃
Shieh Ce-Kuen
Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan
shieh@eembox.ncku.edu.tw

梁廷宇
Liang Tyng-Yue
Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan
lty@eembox.ncku.edu.tw

張志標
Chang Jyh-Biau
Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan
andrew@eespcc.ncku.edu.tw

摘要

本論文[1]提出三種設計同步機制的方法，藉由此種設計，多線頭分散式共用記憶體系統可以減少處理機閒置。這些方法都已經在 Cohesion 上實現並測試，從實驗數據中可看出，我們的方法確實有效。
關鍵字：分散式共用記憶體、同步機制

Abstract

This paper describes three techniques for implementing synchronization mechanisms which are able to reduce processor idleness in multithreaded Distributed Shared Memory (DSM) systems. All of the techniques are implemented and tested on Cohesion. The experimental results shows that our methods indeed reduce the amount of processor idleness.
Keywords: Distributed Shared Memory, Synchronization

1. Introduction

A distributed shared memory (DSM) [7]system is a software system that emulates shared memory semantics on the hardware platform that only supports messages-passing. These systems usually provide a parallel programming environment to simplify the programming for users. Synchronization is one of the most important mechanisms in this programming environment, since it provides for programmers the control of joint activity of cooperating threads or to ensure serialization of concurrent accesses to shared data by multiple threads. Most conventional synchronization mechanisms are optimized only for the single-threaded DSM systems[1][2][6], which allows only a single thread to be executed on each node. However, multithreaded DSM systems[3][5][8][9] have become increasingly important because of the ability to hide network latency by overlapping one thread's communication with another thread's computation, and achieve load balance by thread migration. Unfortunately, the existing synchronization mechanisms optimized for the multithreaded DSM systems may induce processor idleness. In this paper we propose three techniques, which are used for implementing lock , conditional variable, and barrier respectively, to reduce the processor idleness.

The implementations of synchronization mechanisms for single-threaded DSM systems basically deal with the synchronization between nodes. Nevertheless, In multithreaded DSM systems, the synchronization not only occurs among nodes, but also among threads on a single node. Intuitively, to reduce network messages, the synchronization between threads locating at a same node is processed prior to those locating at separate nodes. However, this method will lengthen the waiting time of a synchronization object for the remote nodes. If all threads on a remote node wait for the synchronization object, the processor will become idle.

Lock, conditional variable, and barrier are usually used for synchronization of DSM programs. In case of lock, the induced amount of processor idleness may be massive and even result in starvation of processors, i.e., processors often remain idle during the execution of the program. This is because the threads on a node alternately hold a lock and the threads on other nodes starve to access the lock. The conditional variable may induce moderate amount of processor idleness. It's reason is that number of the ready threads on each node is unbalance. The amount of processor idleness in barrier depends on the amount of network communications. It may result in notable performance drop if the amount of network communication is heavy..

In this paper, we propose a conditional grant technique for implementing lock, a balancing access technique for implementing conditional variable, and a longest-job-first technique for implementing barrier.

The conditional grant technique precedentially serving the lock synchronization of threads on a single node when the number of ready threads on this node is more than a threshold value. Since this condition is not always true, a lock would not be held persistently by a

[1] This research is supported by the National Science Council of R.O.C. under contact NSC86-2745-E-006-014

node and starvation of processors would not occur.

The balancing access always resumes a thread located on a node with the most numbers of waiting threads. Because each node is kept a comparable number of ready threads, the condition that all threads on nodes waiting for a conditional variable can be almost avoided, thus reducing the processor idleness.

The longest-job-first always precedentially schedules a thread that possesses most network latency. In this way, maximum overlap of threads will be achieved and the processor idleness can be reduced.

All of the techniques presented in this paper have been implemented and tested on Cohesion, a DSM system which supports the eager release consistency model and provides a multithreaded programming environment. To verify the effectiveness of the proposed methods, we compare our methods with the conventional ones. All .of the experiments were done by running two real application programs, quick sort and Successive Over Relaxation(SOR). The experimental results show that our conditional grant technique can avoid the starvation of processor in quick sort program, which occurs in another version using a conventional method. Beside, using the balancing access technique, speedup of the quick sort program is improved by 5%. Finally, using the longest-job-first technique, the speedup of SOR is improved by 8%.

The rest of this paper is organized as follows. Section 2 discusses the processor idleness problem in synchronization mechanisms. Section 3 describes our techniques. Section 4 describes the implementation of these techniques on Cohesion. Section 5 shows the performance results. Finally, we conclude in section 6.

2. Processor idleness in synchronization mechanisms

In this section, we will first introduce the conventional designs of the synchronization mechanisms. Then we will describe the processor idleness problem that occurs in these synchronization mechanisms.

2.1 Conventional designs of the synchronization mechanisms

The decentralized methods[8][9] for implementing lock are adopted in current multithreaded DSM systems to reduce unnecessary competition of a lock and network messages. In this method, each node is allocated a local lock queue. When a thread acquires a used lock, it is inserted into the local lock queue. When a thread releases the lock, the system selects a thread in the local lock queue for execution. The lock is granted to the remote threads only when the local lock queue is empty. Since granting the lock to the thread in the local queue requires no message, the number of messages can be reduced.

A centralized method using a global queue is usually used for implementing conditional variable. In this method ,when a thread waits for a conditional variable, it is inserted into the global queue. When a thread executes a signal operation, the thread at the head of the global queue is selected for execution. With the global queue, a DSM system can easily finds the threads waiting for a conditional variable.

The local barrier technique[8][9] is exploited to reduce network traffic for barrier. With this technique, a local barrier manager is allocated on each node in additional to a global barrier manager. The local barrier manager groups the arrival messages from the local threads into a single message, and then sends the message to the global barrier manager. By the technique, a lot of messages can be reduced.

2.2 Processor idleness resulted from multithreading

Although the multithreading can potentially improve the performance of the applications, the improvement may be limited or even worsened if the synchronization mechanisms do not provide suitable occurrence for thread switching. In this subsection, we discuss this problem in detail.

2.2.1 Processor idleness in lock

A program sharing a set of data protected by a lock may result in processor idleness. Suppose M threads are executed on each node. Furthermore, suppose that N of the M threads are waiting for a lock on the node S, and all of M threads on some other nodes are waiting for the lock. If the N threads are consecutively resumed for accessing the lock before the lock is granted to any thread on the other nodes. The nodes with M threads waiting for the lock will be idle until the N threads on node S finish accessing the lock.

A program that heavily shares a set of data protected by a lock may result in heavy processor idleness, even starvation of processor. In some applications, all threads in the programs heavily compete a lock, resulting most of the threads simultaneously wait for the lock. If we always give the node holding the lock highest priority to access the lock, the lock may be consecutively released and acquired by the threads on the same node. This scenario may continue until all threads on this node terminate. As a result, all other processors starve to access the lock. This is a serious problem, since it cause a program to be executed sequentially.

2.2.2 Processor idleness in conditional variable

In multithreaded DSM systems, a lot of threads may be executed on these systems. If the conventional global queue method is used, the queue will become long when most of the threads wait for a conditional variable. Further, if all of the threads on a node are appended at the tail of the queue, this node will be forced idle and result in processor idleness.

2.2.3 Processor idleness in barrier

A barrier is usually used to synchronize all of the threads in DSM programs. In this circumstance, the threads resumed from a barrier will be more than one on each node. If we schedule the threads using a conventional method, such as FCFS(First Come First Serve), the thread that demands most amount of communication usually finishes execution last. Since this last thread has no other thread for overlapping to hide network latency, its waiting time due to network latency will result in much amount of processor idleness.

3. Techniques to reduce processor idleness

Since the behavior of these three synchronization mechanisms is different from each other, we have to propose different technique for each of them. In this section, we present the proposed techniques, including a conditional grant technique, a balancing access technique , and a longest-job-first technique.

3.1 Conditional grant in lock

The conditional grant technique prevents the starvation of processors by ensuring fair access of a lock for each thread. It grant a lock to a thread at remote node instead of local node by default. Besides, to avoid the granting node to become idle after granting the remote request, it grants a remote request only when the number of ready threads on the local node is more than a threshold value. In this way, a lock will never be held persistently by a node. Starvation of processor, therefore, can be avoided.

We currently choose the value of one as the threshold value, since the lock mechanism with this threshold value can grant a remote request as soon as possible. In fact, this value is not optimal for all applications, and the optimal value for all applications may be different. We are trying to adjust the value at runtime to approach the best value for each application.

The lock algorithm including this technique is described as follows. A waiting queue and a requesting queue is maintained for each lock on each node. The waiting queue is used to record the threads waiting for a lock on a local node, while the request queue is used to record the nodes that have threads acquiring the lock. Each lock has an owner, which is defined as the node that most currently holds the lock. Only the owner of a lock has the right to grant a thread to access the lock.

The complete algorithm consists of four procedures: lock acquiring, lock releasing, lock request receiving, and lock grant receiving.

(1) The lock acquiring procedure is invoked when a thread calls an acquire operation. It grants the thread to access the shared data when the local node is the owner of the lock and the lock is not in used. Otherwise, it inserts the thread to the waiting queue and sends a lock acquire message to the owner.

(2) The lock releasing procedure is invoked when a thread calls a release operation. This procedure releases the pertaining lock and resumes a thread waiting for the lock. To ensure fair access of a lock, the lock releasing procedure is partitioned into four parts according to the states of the waiting queue and the requesting queue.

First, if there are threads on a remote node that are acquiring the lock, i.e., the requesting queue is not empty, and no local thread is waiting for the lock, the procedure grants the thread on the remote node to access the lock.

Second, if both of the remote node and the local node have threads acquiring the lock, the procedure will further checks the number of the ready threads on the local node. If the number of the ready threads is more than a threshold value, the procedure will grants the remote request; otherwise, the procedure grants the local request.

Third, if there are threads only on the local node acquiring the lock, the procedure grants the local request.

Fourth, if no thread is acquiring the lock, the procedure returns immediately.

(3) The lock request receiving procedure is invoked when a node receives a request for a lock. The procedure grants a remote request when the local node is the owner of the lock and the lock is not in use. It inserts the node issuing the request to the requesting queue if the lock is being held. If the local node is not the owner, the request is forwarded to the owner.

(4) The lock grant receiving procedure is invoked when a lock grant message is received. It resumes a local thread waiting for the lock.

Our conditional grant technique may generate more network messages than the conventional one described in previous section. In the latter case, no message is generated while the systems grants the local acquiring threads to access the lock. However, in our method, a node may grant a lock to remote threads while it has acquiring threads. Therefore, additional messages have to be sent to the new owner to acquire the lock again, and an additional grant message need be sent back by the new owner to allow the local acquiring threads to access the lock. However, this technique induces only a little additional overhead, since the size of the acquire or grant message is small.

3.2 Balancing access in conditional variable

The balancing access technique avoids processor idleness by always resuming a thread located on a node with the maximum number of threads waiting for a condition variable. In a multithreaded DSM system, the number of threads allocated on each node is usually kept almost the same to achieve preliminary load balance. In this circumstance, a node that has the maximum number of waiting threads will most probably has the minimum number of ready threads. If the system preferentially resume the waiting threads on this node, the possibility that a node with all threads waiting for a conditional variable can be lowered. The processor idleness thereby can be reduced.

The conditional variable algorithm including the balancing access is depicted as follows. A fixed manager for each conditional variable is allocated to coordinate the task of synchronization. Whenever a thread executes a wait or signal operation, a message is sent to the manager. The manager queues the thread when it receives a wait message and resumes a thread when it receives a signal message.

A data structure consisting of multiple queues, as shown in fig. 1, is used. The data structure, which is managed by the conditional variable's manager, consists of a doubly linked queue(DLQ) and several singly linked queues(SLQ). The singly linked queues are used to record the threads waiting for a conditional variable. Each of them is allocated for a node that has threads waiting for a conditional variable. All of the singly linked queues are doubly linked to form the data structure we need. The doubly linked queue has a head node which serves as the entry point of the data structure.

When a thread executes a wait operation, it is appended to the SLQ that is associated with the node on which it is executing. After the thread is appended, the number of threads in this SLQ is compared to that in the preceding SLQ. The sequence of these two SLQ in the DLQ will be exchanged if the number of threads in the newly modified SLQ is greater than that in the preceding SLQ. The comparison and exchange are proceeded until the number of threads in the SLQ is less than or equal to that in a preceding SLQ. Therefore, when a signal operation is executed, the manager can easily select a thread from the first SLQ in DLQ for execution. To keep the order of precedence after resuming, the number of threads in the first SLQ is compared to that in the second SLQ. If the former is greater than or equal to the latter, the process completes. Otherwise, the sequence of these two SLQs in the DLQ is exchanged. The comparison and exchange are proceeded until the number of threads in the former SLQ is greater than or equal to the latter SLQ.

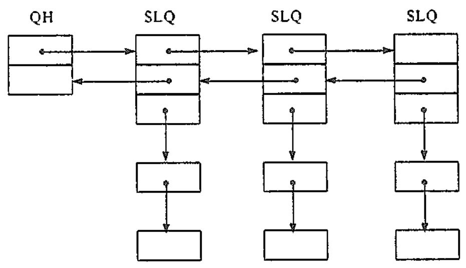

Fig. 1 Data structure for implementng balancing access

3.3 Longest-job-first in barrier

The longest-job-first technique ensures that the threads with the most amount of communication in the previous iteration are always scheduled to execute prior to the others for iterative applications. Since we cannot exactly predict the amount of communication for each thread before they are executed, we use the amount of communication in previous iteration instead. This is reasonable because barrier is usually used for synchronization at the end of each iteration, and the data accesses pattern of a thread in each iteration is similar..

In our longest-job-first technique, the amount of data sent at each barrier synchronization is recorded for each thread. When all the threads participating in a barrier synchronization have arrived at the barrier and ready for resumption, the barrier chooses a thread that demands the maximum amount of communication in the previous iteration for execution, and leaves the threads that have less amount of communication in the previous iteration to be executed later. In this way, the thread with the minimum amount of communication will be executed lastly. Thus reduce the amount of processor idleness.

4. Implementation

All of the proposed techniques are implemented based on Cohesion's parallel programming environment. In this section, we will give a overview of Cohesion. Then we will describe the notable issues in implementing these techniques.

4.1 Cohesion overview

Cohesion is a multithreaded DSM system that supports release consistency and sequential consistency model. It provides a multithreaded environment that supports a global view of threads, allowing each thread to be seen by each node. The multithreaded environment is also incorporated with a overlapping mechanism that is capable of overlapping the execution of threads to hide network latency. Cohesion supports thread migration, with which load balance is achieved by migrating thread from a node with heavy load to a node with slight load. Cohesion also provides a single shared address space spanning over all nodes. This make it differ from the shared-variable DSM systems in which the shared data must be explicitly declared.

4.2 Lock

Our lock use a decentralized local queue method. To negotiate the local lock manager on each node, a probable owner scheme is used. We allocate a probable owner field for each lock on each node and it is initially set to the root node. When an acquire operation is invoked, the probable owner field is checked to find out the probable owner of a lock. If the owner is not the local node, a lock request message is sent to the probable owner. After receiving the message, the probable owner will check the probable owner field on it. If the node is again not the true owner, the message is forwarded to the probable owner indicated in the field. Through the forwarding, a request message can finally reach the true owner.

4.3 Conditional variable

Conditional variable is usually used to synchronize threads with the producer-consumer type of accessing pattern to a shared buffer. In this kind of applications, a thread acting as a consumer executes a wait operation when the buffer is empty, whereas a thread acting as a producer executes a signal operation after it fills data to the buffer. In a multithreaded environment, several threads may act as producers and several threads may act as consumers. In this condition, the number of wait operations executed may be not equal to that of signal operations. If the number of wait operations is more than that of signal operations, the total number of threads suspended is more than that of thread resumed. As a result, some threads may still be pending after all producers terminate.

To overcome this problem, our conditional variable is allowed to set a number indicating the maximum number of threads be waiting. When a wait operation is executed, the manager of the conditional variable will first check if the number of threads waiting for this conditional variable has reached the maximum value. If the result is no, a normal operation that suspending the thread in the manager will be done. Otherwise, all threads waiting for this conditional variable will be resumed. With this support, all threads in a program can be resumed and terminated normally even with unbalanced numbers of wait and signal operations..

4.4 Barrier

The support of longest-job-first for the barrier is achieved by slightly modifying a traditional ready queue scheduling. We allocate a variable in each thread's TCB (Thread Control Block) to record the amount of communication while executing a barrier synchronization. After a thread flushes all of the updated data to all caching nodes, the amount of the flushed data is saved in the variable. When a barrier has completed, the scheduling mechanism will sort the threads according to the value in the variable, and then insert them to the ready queue. Since the threads are executed in order, the threads with large amount of communication can always be executed before the threads with less amount of communication. In this way, we can achieve maximum overlapping and minimum processor idleness.

5. Performance

This section presents the results of two applications that use our techniques as well as the conventional methods. We first describe the experimental environment and the applications, and then proceed to discuss the effectiveness of our techniques.

5.1 Experimental environment and applications

All experiments were carried out on Cohesion, a test-bed built on a network of 8 PCs connected by 10 Mbps Ethernet. Each PC has a 90 MHz Intel Pentium processor and 32 Mbytes of memory. Cohesion is isolated from campus networks during the experiment.

Two application programs are used to evaluate the effectiveness of the proposed methods. One is a quick sort program, and the other is a Successive Over Relaxation(SOR) program. The quick sort program employs a divide-and-conquer algorithm. It sorts 2 M integrals in our experiment. Both lock and conditional variable are used in this program. The SOR is a iteration-based algorithm to model natural phenomena or calculate approximate solution of a partial differential equation. In our experiment, a grid of points in a pending area of the problem is represented by a matrix. We parallelize the program by dividing the rows in the matrix into banks, and each bank is computed by a thread. Only barrier is used in this program.

To compare our methods with the conventional methods, three versions of quick sort programs are written. The QSORT program uses both the conditional grant technique and the balancing access technique. The threshold value for lock is set to one in this program. The QSORT_CONV_C uses the conditional grant technique with threshold value of one in lock, but the conventional method in conditional variable. Finally, the QSORT_CONV_L uses the balancing access technique in conditional variable, but the conventional method in lock. The conventional method in lock exploits the local queue method to manage a lock synchronization similar to our method, but it always preferentially resumes a local thread even through other nodes have threads waiting for the lock. The conventional method in conditional variable uses a first-in-first-out method to resume threads waiting for a conditional variable.

Besides, two versions of SOR programs is written for comparison. The SOR_LJF uses the longest-job-first technique, whereas the SOR_FCFS uses the FCFS(first come first serve) method to schedule threads resumed from a barrier.

5.2 Effectiveness of conditional grant

Table 1 shows the experimental results of the quick sort programs running on eight processors with eight threads per processor. The execution time column shows the execution time of each program measured from the time the program is loaded into cohesion to the time it is terminated. The number of messages column reflects the total number of messages issued by the eight processors during the execution of the programs. The idle time shows the amount of times the processors spent waiting for locks or conditional variables.

To evaluate the effectiveness of conditional grant, we compare the result of QSORT_CONV_L with QSORT. Since these programs are the same except using different lock mechanisms, it is reasonable to use them to test effectiveness of conditional grant for lock.

As shown in table 1, the QSORT_CONV_L has much longer execution time than that of QSORT, and the QSORT_CONV_L has less number of messages as well. This implies that starvation of processors occurs when QSORT_CONV_L is executed. This conclusion is supported by the idle time column. The idle time in QSORT_CONV_L is 2173 sec, whereas the idle time in QSORT is only 505 sec. The former reflects that each processor spent 270 second on average to wait for a lock. This means that 62 % of the execution is spent to wait for a lock. Since only 38 % of the execution time is used for computation, starvation of processors must occur. Besides, we had carefully observed the behavior of execution for these two programs during the experiment. In fact, the execution of QSORT_CONV_L becomes serial after it has been executed for a while. Consequently, the performance result shows that our conditional grant is useful in implementing efficient lock mechanism for multithreaded DSM systems.

Table 1.Performance of quick sort programs

	Execution Time(sec.)	No. of Messages	Idle Time(sec.)
QSORT_CONV_L	433	16509	2173
QSORT_CONV_C	220	33057	717
QSORT	208	34251	505

5.3 Effectiveness of balancing access

The effectiveness of balancing access can be evaluated by comparing the experimental result of QSORT_CONV_C and QSORT. As shown in table 1, the execution time of QSORT_CONV_C is 12 seconds more than that of QSORT. Similarly, the idle time of QSORT__CONV_C is 212 sec more than that of QSORT. This result implies that the technique of balancing access can indeed reduce the processor idleness, and improve the performance of the application.

5.4 Effectiveness of longest-job-first

Table 2 shows the execution time of SOR_LJF(uses longest-job-first) and SOR_FCFS(uses FCFS), in which a 512x4096 matrix is calculated with 100 iterations. We chose 512x4096 as the matrix size because enough amount of computation and communication can be generated. We ran these programs with various number of threads on each node for investigating the influence of degree of multithreading. In the case of eight threads per node, SOR_LJF shorten the execution time by 8 %. This implies that using longest-job-first technique can indeed improve the performance of applications. Furthermore, table 2 shows that the improvement of performance increases when the number of threads on each node increases. Consequently, we can conclude that the longest-job-first works well in a multithreaded environment, and it can improve performance more when the degree of multithreading increases.

Table 2 Execution time of SOR with various no. of threads per node

No. of threads per node / Applications	1	2	4	8
SOR_FCFS	112.3	106.9	105.1	106.2
SOR_LJF	111.6	107.5	97.9	97.8

6. Conclusions

We have proposed a conditional grant technique, a balancing access technique, and a longest-job-first technique to reduce the processor idleness occurred in lock, conditional variable, and barrier respectively. The conditional grant technique can efficiently avoid processor idleness and starvation of processor. It makes local queue implementation of lock more suitable for multithreaded DSM systems. The balancing access technique inherits the global queue implementation of conditional variable. Although it is a centralized method, it can precisely and efficiently balance the number of ready threads on each node, and thus reduce processor idleness. The longest-job-first technique applies a inverse idea from conventional scheduling algorithms. However, it works well in multithreaded DSM systems because it hides most of the network latency for each thread. As these techniques indeed reduce processor idleness and improve the performance of the applications, we can conclude that providing a set of efficient synchronization mechanisms are importance for improving the performance of distributed shared memory systems. Our future work is to design synchronization mechanisms that support thread migration, since thread migration is necessary for achieving load balance.

References

[1] Bennett, J. K., Carter, J. B., and Zwaenepoel, W. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming(PPOPP'90), pp. 168-177, March 1990.

[2] Bershad, B. N. Zekauskas, M. J., Sawdon, W. A. The Midway Distributed Shared Memory Systems. *In Proceedings of the '93 CompCon Conference*, pp. 528-537, February 1993.

[3] Freeh, V. W., Lowenthal, D. K., and Andrews, G. R. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. *In proceedings of First Symposium on Operating Systems Design and Implementation*, pp. 201-212, Monterey, CA, November, 1994.

[4] Gharachorloo, K., et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *In proceedings of the 17th Annual International Symposium on Computer Architectures*, pp. 15-26, Seattle, Washington, May 1990.

[5] Itzkovitz, A., Schuster, A. Wolfovich, L. *Thread Migration and its Applications in Distributed Shared Memory Systems*. To appear in Journal of System and Software, 1997.

[6] Keleher, P., Cox, A. L., Amza, C., Dwarkadas, S., and Zwaenepoel, W. *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems*. In USENIX Winter 1994 Conference Proceedings, pp. 115-132, San Francisco, California, January 1994.

[7] Li, K. and Hudak, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Transaction on Computer Systems*, 7(4):321-359, November 1989.

[8] Shieh, C. K., Lai, A. C., Ueng, J. Y., Laing, T. Y., Chang, T. C., and Mac, S. C., Cohesion: An efficient Distributed Shared Memory System Supporting Multiple Memory Consistency Models. *In Proceedings of the First Aizu International Symposium on parallel Argorithms/Architecture Synthesis*, Aizu-Wakamatsu, Fushima, Japan, March 1995.

[9] Thitikamol, K., Keleher, P. Multithreading and Remote Latency in Software DSMs. *In the 17th International Conference on Distributed Computing Systems*, May 1997.