Proceedings of International Conference
on Computer Architecture

Reducing the Overhead of Migratory-Sharing Accesses for the Linked-Based Directory Coherence Protocols in Shared-Memory Multiprocessor systems*

Jen-Pin Su, Chao-Chin Wu, and Cheng Chen
Institute of Computer Science and Information Engineering
National Chiao Tung University, Hsinchu, Taiwan, Republic of China
Tel:(8863) 5712121 EXT:54701~54704, Fax:(8863)5724176

Abstract

Migratory-sharing objects are the data structures manipulated by only a single processor at any given time [1]. This type of accesses will incur many cache misses which can be reduced by the merge of the invalidation/update requests and the cache misses. P. Stenstrom et al. proposed a mechanism to optimize the migratory-sharing accesses for the central directory coherence protocols [2]. However, their mechanism can not support the linked-based directory protocols. Therefore, in this paper, we proposed a technique to reduce the overhead of migratory-sharing accesses for the linked-based directory protocols in some degree. Our mechanism can be implemented in various write policies, including the write-invalidate, write-update, and the write-competitive policies. Based on a program-driven simulation environment and a set of benchmarks, we have evaluated the performances for the various write policies. As a result, it can be shown that our method can effectively reduce the overhead of the migratory-sharing accesses.

1. Introduction

Gupta and Weber classified data structures based on the invalidation pattern they exhibit [1]. According to their definition, data structures manipulated by only a single processor at any given time are called migratory objects. Typically, such sharing occurs when a data structure is modified within a critical section. Because the modification of migratory objects usually executes the tight Read-Modify-Write operation, P. Stenstrom et al.,[2-3] has formally defined the access pattern of migratory blocks by the following regular expression.

....Ri(Ri)*(Wi)(Ri/Wi)*Rj(Rj)*(Wj)(Rj/Wj)*...(1)

where Ri and Wi represent a read access and a write access, respectively, by processor i, '*' denotes zero or more occurrences of the preceding string, and '|' denotes the logical OR-operation.

P. Stenstrom et al. proposed a mechanism for reducing the overhead of accessing the migratory-sharing memory block for the central directory protocols [2]. They use hardware to detect the migratory-sharing memory block by recording the access pattern of each block and finding those blocks matching the pattern in the regular expression (1). After a migratory-sharing block has been detected, the read access to the block is replaced by a Read-exclusive one to prevent from the need of invalidation or update incurred by the subsequent write access. However, when the migratory-sharing memory block is detected as non-migratory-sharing one, it will be transferred to the ordinary one.

Their detection mechanism relies on that the home directory can receive all the global read/write requests; that is, all global accesses have to interrogate the home directory. The central-directory structure meets the above requirement. However, the linked directory structure [5][6] has no such property. In the SCI coherence protocol [5], for instance, before the head node of the dirty sharing list executes a write access, it has no need to interrogate the home directory because it can invalidate or update other cache copies through pointers in the dirty sharing list. Therefore, we propose a method for the linked-based directory protocols to reduce the overhead of the migratory-sharing memory block.

We first divide the shared memory accesses into four classes which will read/write the migratory/non-migratory sharing memory block, respectively. We label the shared memory accesses in the compile time, and use the information to determine which memory blocks are migratory-sharing in the run time. Then

* This paper is supported by National Science Council under contract number: NSC85-2221-E-009-037

we handle the accesses differently according to their types to reduce the subsequently explicit invalidation/update request. The feature of our method is that it can be implemented in the protocols with different write policies, including write-invalidate, write-update, and write-competitive policies. So far, we have evaluated the advantages of our method by a program-driven simulator and the results show that the overhead of the migratory-sharing accesses can be effectively reduced.

The organization of the rest of the paper is as follows. Section 2 introduces the overview of our method and Section 3 describes the detailed implementation. In Section 4, we present our simulation environment and the evaluation results. Section 5 concludes the paper.

2. Concepts and principles of our method

Our proposed method is for the linked-based directory protocols to reduce the overhead of migratory-sharing accesses for the shared memory blocks, and then improve the system performance. Because of the special access pattern of the migratory-sharing access, we can use the Read-exclusive access to avoid the need of subsequently explicit invalidation/update request. The number of global write requests is thus reduced. To prevent from the higher complexity of detecting the migratory-sharing memory blocks and to implement the method in the linked-based directory protocols, we detect the migratory-sharing memory blocks by using the labels of memory accesses to provide the information. We describe it as follows.

First, we use compiler to label the migratory-sharing memory blocks. Then, the memory subsystem uses it to identify the migratory-sharing memory blocks. The handling of the migratory-sharing blocks is different from that of the ordinary accesses. In addition, the miss handling of the migratory-sharing accesses depends on the following types of the accesses:

(1) Migratory-sharing read accesses: This kind of accesses is typically the read accesses in the tight Read-Modify-Write operations. Therefore, when the read accesses the migratory-sharing memory block, we use the read-exclusive handling method to guarantee the local cache hit for the subsequent write access. Because the invalidation or the update request incurred by the write access is avoided, the delay time for the write is reduced.

(2) Non-migratory-sharing read accesses: Those read accesses not belonging to the type (1) are called non-migratory-sharing read accesses. This kind of read accesses will access the migratory-sharing memory block because in the block there are non-migratory-sharing memory words. To keep the migratory-sharing accesses not be interfered by the non-migratory-sharing read accesses, we adopt the Read and Self-invalidate method. That is, the memory block is read to local cache and the local cache controller will invalidate the block itself later, and the read operation will not affect the corresponding shared-memory block state and the states of other caches.

(3) Migratory-sharing write accesses: This type of accesses are those writes in the tight Read-Modify-Write operations. Migratory-sharing write accesses usually hit the cache because the previous migratory-sharing read access will guarantee the existence of the memory block in the local cache. The exception is that the memory block is replaced and writes back to the home memory when other non-migratory-sharing accesses reference the block. We handle this kind of accesses by the Write-invalidate method.

(4) Non-migratory-sharing write accesses: The write accesses not belonging to the type (3) are called non-migratory-sharing write accesses. Similar to the type (3), we handle these writes by the Write-Invalidate method.

To prevent the non-migratory sharing accesses from interfering the handling of the migratory-sharing memory block, we use the Read and Self-Invalidate method as mentioned in the above (2) to handle the non-migratory-sharing read accesses. There are two alternatives here:

(1) Uncached policy: The migratory-sharing memory block is not cached in the local cache.

(2) Stale cached policy: Use the mechanism of delayed consistency [Dubo 91]. Cache the memory block in the local cache but set it to the stale state. When encountering an acquire synchronization access, flush all the stale blocks to prevent from reading the stale data. This policy can eliminate the read miss incurred by the uncached policy.

Based on the above concepts and principles, in the next section, we will describe the detailed design procedure of our method under the release consistency model [7] for the doubly-linked directory cache protocols with three write policies, including write-invalidate, write-update, and write-competitive [8] policies.

3. Design procedures

The whole design procedures include the following three parts: (1) Categorization and labeling, (2) Maintenance, and (3) Handling. We will describe the more details in the following subsections.

3.1 Categorization and labeling

For convenience, we use the categorization of shared-memory accesses in the release consistency model [7] to describe how the compiler labels different memory accesses. As shown in Figure 1, Gharachorllo et al. divide the shared-memory accesses into acquire accesses, release accesses, ordinary reads and ordinary writes; in addition, the italic words represent the access types we need [7]. We categorize the shared accesses (not including the synchro-nization accesses) into four classes, including Migratory-Sharing Read (MS-Read), Migratory-Sharing Write (MS-Write), Non-Migratory-Sharing Read (NMS-Read), and Non-Migratory-Sharing Write (NMS-Write). Typically, the sh-ared accesses with the MS-Read label indicate the read accesses in the tight Read-Modify-Write operations. The shared accesses with the MS-Write label indicate the write accesses in tight Read-Modify-Write operations. The shared read accesses not belonging to the MS-Read access are labeled as the NMS-Read. The shared write accesses not belonging to the MS-Write access are labeled as the NMS-Write. So far, only the accesses referencing to critical sections are labeled migratory-sharing accesses because of the tradeoff between the implementation complexity and the impact on the critical path of the program execution time.

3.2 Maintenance

To identify and maintain the migratory-sharing memory blocks, the cache controller and the memory controller for each memory block have to respectively add one and two more states.

(1) For each cache block, add the new Migratory-Dirty (MD) state. The cache block with MD state indicates that the block is migratory-sharing, and this means that the memory subsystem has identified the memory block as migratory-sharing one. The new state is to prevent the memory subsystem from unnecessary detection of migratory-sharing memory blocks afterwards.

(2) For each shared memory block, add two new states as follows:

(a) Migratory-Dirty-GONE(MD-GONE): The shared memory block with MD-GONE state indicates that the block is migratory-sharing, and this means that the memory subsystem has identified the memory block as migratory-sharing one.

(b) Migratory-Uncached (MU): The shared memory block with MD-GONE state indicates that this block has no copy in any cache. The new state is to prevent the memory subsystem from unnecessary detection of migratory-sharing memory blocks.

3.3 Handling

Our proposed method will implement directly in three different protocols, including write-invalidate, write-update, and write-competitive policies. For all the references accessing to the non-migratory-sharing memory blocks and the hit references accessing to the migratory-sharing memory blocks, the handling methods are the same as that in the original protocols. However, for the miss references accessing to the migratory-sharing memory blocks, we handle them differently according to their access types and the sharing status. The detailed handling techniques are described by the following different sharing situations.

(1) No cache copy: In this situation, the memory state must be MU, and the handling for the four types of accesses are similar. Figure 2 shows the handling for the MS-Reads or the NMS-Reads. When the memory controller receives a read miss (RM) request, it replies the memory state and the block to the requesting cache. The memory state will then be transferred from MU to MD-GONE, and the cache state be changed to MD. The handling procedure for the MS-Writes or the NMS-Writes shown in Figure 3 is similar to that for the read miss.

(2) Only one cache copy: In this situation, the memory state must be MD-GONE, and the cache state will be MD. The handling methods for the four different access types are described as follows.

(a) MS-Read: We adopt the Read-exclusive policy to handle this type of read. The advantage of this policy is that the following write access will incur a local hit, and thus reduce the unnecessary invalidation or update. Now, let us see the detailed procedure as shown in Figure 4. When the memory controller has received the request for a migratory-sharing read miss, it keeps the memory state unchanged because the state is MD-GONE. Then, it replies the state and the pointer (pointing to the head of the dirty sharing-list, Cache 1) to the requester (Cache 0). Then, it sets the pointer to point the requesting cache. When the requester has received the reply, according to the replied message, it issues a Read-exclusive request to the head node of the sharing-list. After receiving the Read-exclusive request, Cache 1 changes its cache state from MD to INVALIDATE, and replies the

MD state to the requesting cache. In addition, Cache 0 changes its state to MD after it receives the reply.

(b) NMS-Read: When an NMS-Read accesses a migratory-sharing memory block, we adopt the handling method of Read and Self-invalidate which does not add the new cache copy to the dirty sharing-list. The advantage of this method is that the NMS-Read will not interfere the property of the migratory-sharing memory block. Read and Self-invalidate can be implemented by the following two policies.

(i) Uncached policy: The requester (Cache 0) that issues the NMS-Read miss request will not copy the memory block into its cache, as shown in Figure 5, and thus it will not change the status of the dirty sharing-list.

(ii) Stale cached policy: This policy takes advantage of the mechanism of delayed consistency [9]. We place the copy in the requesting cache but set it to stale state. Whenever reading the memory, it will be a hit until the copy is invalidated due to the cache encounters an acquire access. This policy can reduce the amount of the read miss occurred in (i). Figure 6 shows the handling procedure. The requester (Cache 0), issuing the NMS-Read miss, will bring the block into its cache and change its state to Stale. Similar to (i), the status of the dirty sharing-list will not be influenced during the handling. However, to guarantee the correctness of the program execution, when the processor executes an acquire access, all the stale blocks must be set to invalid blocks.

(c) MS-Write or NMS-Write: When these two types access the migratory-sharing memory block, we adopt the Write-invalidate handling method, as shown in Figure 7. When the memory controller has received the request for the write miss (WM), it keeps the state unchanged because the memory state is MD-GONE. In addition, it replies the memory state and the pointer (pointing to the head of the dirty sharing-list, Cache 1) to the requester Cache 0. Then, it sets the directory pointer to point to the requesting cache. After receiving the reply, the requester issues a request of Read-exclusive to Cache1. When Cache 1 has received the Read-exclusive request, it changes the state to INVALID and replies the MD state to Cache 0. After receiving the reply, Cache 0 changes its state to MD.

Moreover, our proposed method described above can be constructed by different labeling policies, and two handling policies of Read and Self-invalidate. Here, we define the following implementation options.

(1) Only the Read-Modify-Write operations in the critical sections for Barrier codes are labeled as migratory-sharing accesses. We adopt the Uncached policy to handle the Read and Self-invalidate,.

(2) The labeling is the same as that in (1). However, we adopt the Stale cached policy to handle the Read and Self-invalidate.

(3) All the Read-Modify-Write operations in critical sections are labeled as migratory-sharing accesses.

In addition, our method can be implemented in doubly-linked directory architecture with three different write policies, including write-invalidate (DD-INV), write-update (DD-UP), and write-competitive (DD-CU). As a whole, we summarize all of the implementation options of our method in Table 1. In the next section, we will give a brief description of our simulation environment and some performance evaluations of our methods for these implementation options.

4. Simulation environment and preliminary performance evaluations

In order to study and evaluate the advantage of our proposed technique, we have designed a simulation and evaluation environment. The architecture we simulate is a CC-NUMA, distributed shared-memory multiprocessors, as shown in Figure 8. The architecture consists of many nodes, and these nodes are interconnected by the K-ary, n-Cube network. Each node consists of a local shared-memory area, a processor environment, and a local inter-connection. Each processor environment includes a two-level cache hierarchy, as shown in Figure 9. In addition, in order to support release memory consistency models we implemented a lockup-free second level cache [10].

Our simulation environment is a program-driven simulator and constructed based on the MINT [11] package, as shown in Figure 10. The environment has been constructed on the SUN SPARC workstation and can work correctly. It is written in C programming language and has about 55000 statements (The memory subsystem simulator has 15000 statements). The whole environment consists of two parts: the memory reference generator and the memory subsystem simulator. The memory reference generator simulates the instruction interpretation and forwards the memory references to the memory subsystem simulator.

Our memory subsystem simulator consists of the node simulator and the global interconnection simulator. The node simulator simulates the two-level cache hierarchy, doubly-linked directory cache coherence protocols, memory consistency models and the local interconnection. The global interconnection simulator simulates the K-ary, n-Cube network [12].

Before evaluating the performance, we have several assumptions about the architecture, as shown in Table 2. In addition, memory pages are 4 Kbytes and are mapped to the local memories in a round-robin fashion. We use three benchmark programs from the SPLASH and SPLASH2 suites [13, 14] in the experimental evaluation. We summarize them in Table 3 along with the data sets used. Regarding to MP3D, we ran it with switching on the locking option. All applications are written in C using ANL macros and have been complied using cc with the optimization level 2. All statistics are collected only the parallel part of the benchmarks. In addition, in order to realize the basic characteristics of benchmark program we summarize them in Table 4.

As shown in Figures 11-13, we first analyze the impact of implementation options on the system performance. We can see that the option 2 with the *Stale-Cached* strategy has fewer coherence misses than the option 1 with the *Uncached* strategy. On the other hand, there is the non-negligible increasing of read penalty for the option 3. The main reason is that the accesses in all the critical sections are labeled as migratory-sharing ones. Because the size of migratory objects protected by the critical sections not belonging to the Barrier codes is typically larger, many non-migratory-sharing memory blocks will be detected as the migratory blocks. Due to the wrong labeling and the Read-and-Self-invalidate policy for the NME-Read, the read penalty becomes larger and degrades the system performance. The effect is obvious in MP3D and PTHOR, however, it is not so obvious in Ocean because there is lower percentage of critical sections for implementing non-Barrier codes. According to the above analysis and the consistent performance improvement, we recommend the implementation option 2.

In addition, we also observe that the performance of the DD-CU protocols with our proposed technique (implementation option 2) will be obviously improved. Moreover, the performance of the DD-CU protocol will close to the DD-INV protocol, and sometimes is better than the DD-INV protocol, such as OAD2-CU1 for PTHOR and OAD2-CU4 for MP3D. According to the results, we also see the importance of the threshold for the DD-CU protocol. The optimal threshold for an application is difficult, or even impossible, to predict statically because it varies for different applications and depends on (1) the ratio of communication time and computation time and (2) the access patterns to shared data structures. Therefore, we hope to find a method that can dynamically assign a better threshold to each cache line in runtime using hardware or compiler in the future.

5. Concluding Remarks

In this paper, we have proposed a mechanism to reduce the overhead of the migratory-sharing accesses for the linked-based directory coherence protocols in shared-memory multiprocessor systems. We first divide the shared accesses into four classes, including Migratory-Sharing Reads, Migratory-Sharing Writes, Non-Migratory-Sharing Reads, and Non-Migratory-Sharing Writes. Then we use the compiler to label them. In the runtime, we use the label to detect which memory blocks are migratory-sharing. Finally, we handle the accesses differently according to their types. The advantage of our mechanism is that it can be implemented in the protocols with different write policies, including write-invalidate, write-update, and write-competitive policies. According to our simulation results, our method can effectively reduce the overhead of the migratory-sharing accesses. In the future, we hope to find a method that can dynamically assign the optimal threshold for the DD-CU protocol to each cache line in the runtime by using hardware or compiler.

References

[1]A. Gupta and W-D. Weber, "*Cache Invalidation Patterns in Shared-Memory Multiprocessors*", IEEE Trans. on Computers, 41(7), pp.794-810, July 1992.

[2]Stenstrom, P, Brorsson, M., and Sandberg, L. "*An adaptive cache coherence protocol optimized for migratory sharing*", Proc. 20th Annual International Symposium on Computer Architecture. IEEE Computer Society Press, Los Alamitos, CA. 1993. pp.109-118.

[3]H. Nilsson and P.Stenstrom,"*An adaptive update-based cache coherence protocol for reduction of miss rate and traffic*", Proc. PARLE Conf., Athens, Greece (Lecture Notes in Computer Science,817, Springer-Verlag, Berlin, Jul. 1994) pp.336-374.

[4]D.Lenowshi, and J.London, **Stanford DASH Multiprocessor, Technical Report**,CS-TR-89-403.

[5]IEEE SCI draft 2.00: **SCI Scalable Coherence Interface**, Draft Document for the IEEE SCI standard,1992.

[6]David Brain Glasco, **Design and Analysis of Updated-Based Cache Coherence Protocols for Scalable Shared-Memory Multipro-cessors**, Technical Report No. CSL-TR-95-670, Computer Systems Laboratory Depart-ment of Electrical Engineering and Computer Science Stanford University, Stanford, California 94305, June 1995.

[7]Gharachorllo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, " *Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors*", In Proc. of the 17th Annu. Inter. Symp. on Comp. Arch. ,pp.15-26,May 1990.

[8]H. Nilsson,P. Stenstrom, and M. Dubois, **Implementation and Evaluation of Update-Based Cache Coherence Protocols Under Relaxed Memory Consistency Models**, Technical Report, Dept. of Computer Engineering, Lund University, Sweden, July 1993.

[9]Dubois, M.,et al. "*Delayed consistency and its effects on the miss rate of parallel programs*". proc. of Supercomputing '91.ACM Press, New York,1991,pp. 197-206.

[10]Fredrik Dahlgren and Per Stenstrom. "*Using Write Caches to Improve Performance of Cache Cohrence Protocols in Shared-Memory Multiprocessors*" ,Journal Parallel and Distributed Computing 26. pp.193-210,1995.

[11]Jack E. Veenstra and Robert J. Fowler. **MINT Tutorial and User Manual.** Technical Report 452,The University of Rochester, New York,1994.

[12]W. J. Dally, "*Performance Analysis of k-ary n-Cube Interconnection Networks*", IEEE Trans. Computers,39(6):775-785,1990.

[13]J-P. Singh, W-D. Weber, and A. Gupta, "*SPLASH: Stanford parallel applications for shared-memory*", ACM SIGARCH Computer Architecture News20(1) (Mar. 1992) pp.5-44.

[14]Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations",1995 ACM 0-89791-698-0/95/0006,pp24-36.

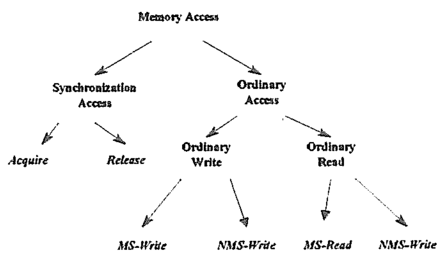

Figure 1 Categorization of shared accesses for our method

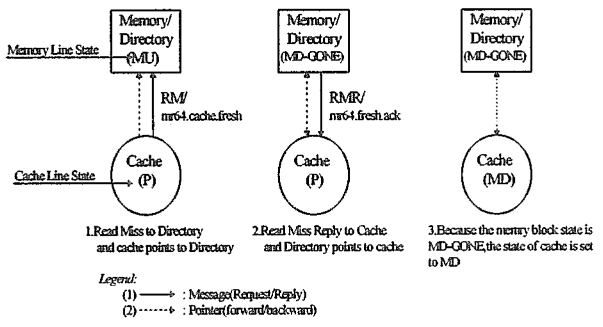
Memory Line State
Memory/ Directory (MU)
RM/ mr64.cache.fresh
Cache Line State
Cache (P)
1.Read Miss to Directory and cache points to Directory
Memory/ Directory (MD-GONE)
RMR/ mr64.fresh.ack
Cache (P)
2.Read Miss Reply to Cache and Directory points to cache
Memory/ Directory (MD-GONE)
Cache (MD)
3.Because the memry block state is MD-GONE,the state of cache is set to MD
Legend:
(1) → : Message(Request/Reply)
(2) ⇢ : Pointer(forward/backward)

Figure 2 The handling procedure for the MS-Read under the situation of no cache copy

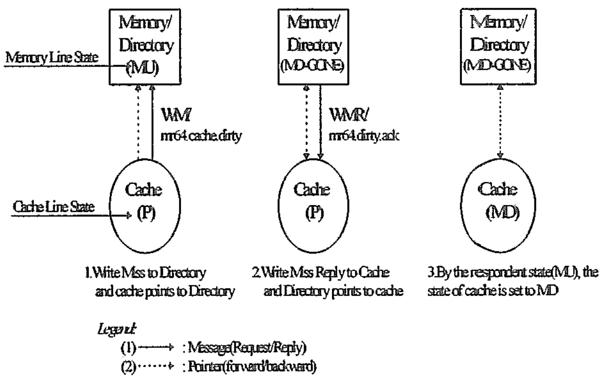
Memory Line State
Memory/ Directory (MU)
WM/ mr64.cache.dirty
Cache Line State
Cache (P)
1.Write Miss to Directory and cache points to Directory
Memory/ Directory (MD-GONE)
WMR/ mr64.dirty.ack
Cache (P)
2.Write Miss Reply to Cache and Directory points to cache
Memory/ Directory (MD-GONE)
Cache (MD)
3.By the respondent state(MU), the state of cache is set to MD
Legend
(1) → : Message(Request/Reply)
(2) ⇢ : Pointer(forward/backward)

Figure 3 The handling procedure for the MS-Write under the situation of no cache copy

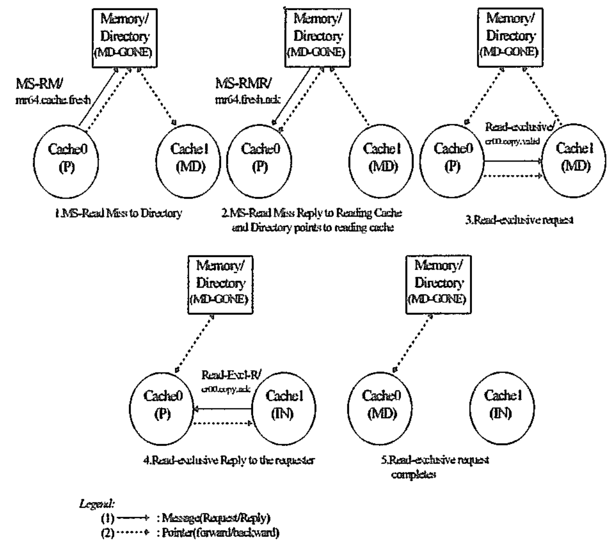
Memory/ Directory (MD-GONE)
MS-RM/ mr64.cache.fresh
Cache0 (P)
Cache1 (MD)
1.MS-Read Miss to Directory
MS-RMR/ mr64.fresh.ack
2.MS-Read Miss Reply to Reading Cache and Directory points to reading cache
Read-exclusive/ cr00.copy.valid
3.Read-exclusive request
Read-Excl-R/ cr00.copy.ack
Cache1 (IN)
4.Read-exclusive Reply to the requester
Cache0 (MD)
5.Read-exclusive request completes
Legend:
(1) ——→ : Message(Request/Reply)
(2) ······→ : Pointer(forward/backward)

Figure 4 The handling procedure for the MS-Read under the situation of only one cache copy

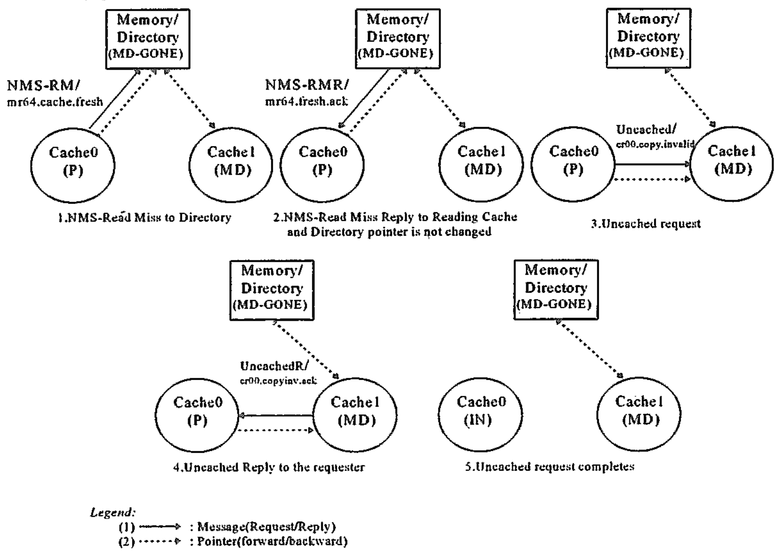
Memory/ Directory (MD-GONE)
NMS-RM/ mr64.cache.fresh
Cache0 (P)
Cache1 (MD)
1.NMS-Read Miss to Directory
NMS-RMR/ mr64.fresh.ack
2.NMS-Read Miss Reply to Reading Cache and Directory pointer is not changed
Uncached/ cr00.copy.invalid
3.Uncached request
UncachedR/ cr00.copyinv.ack
4.Uncached Reply to the requester
Cache0 (IN)
5.Uncached request completes
Legend:
(1) ——→ : Message(Request/Reply)
(2) ······→ : Pointer(forward/backward)

Figure 5 The uncached policy for the NMS-Read under the situation of only one cache copy

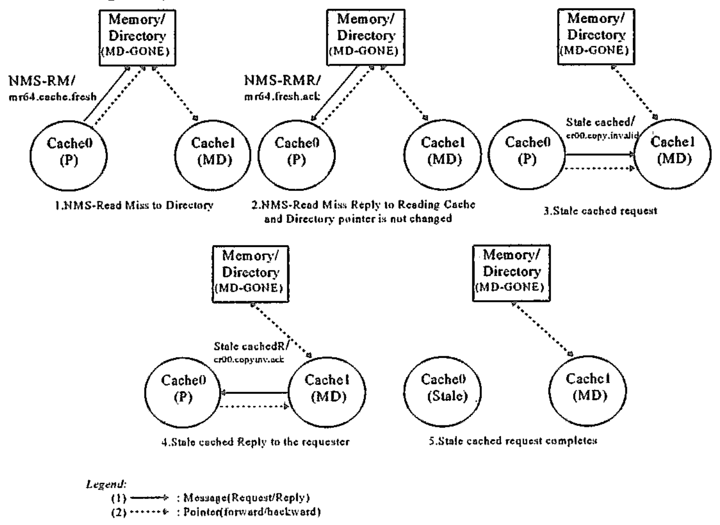
Memory/ Directory (MD-GONE)
NMS-RM/ mr64.cache.fresh
Cache0 (P)
Cache1 (MD)
1.NMS-Read Miss to Directory
NMS-RMR/ mr64.fresh.ack
2.NMS-Read Miss Reply to Reading Cache and Directory pointer is not changed
Stale cached/ cr00.copy.invalid
3.Stale cached request
Stale cachedR/ cr00.copyinv.ack
4.Stale cached Reply to the requester
Cache0 (Stale)
5.Stale cached request completes
Legend:
(1) ——→ : Message(Request/Reply)
(2) ······→ : Pointer(forward/backward)

Figure 6 The Stale cached policy for the NMS-Read under the situation of only one cache copy

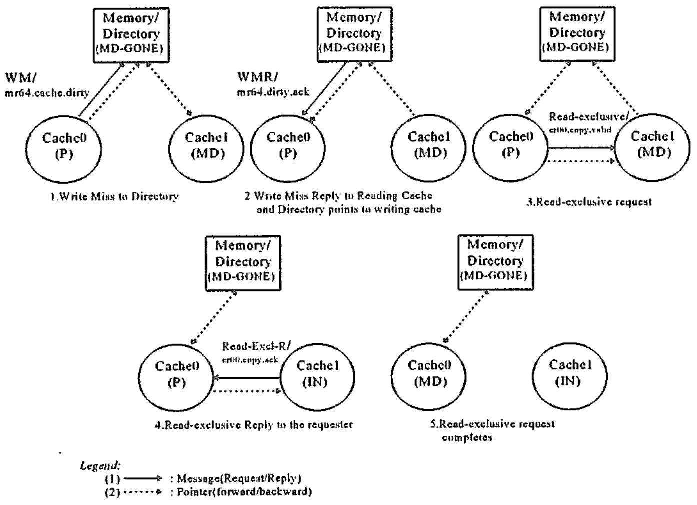
Memory/ Directory (MD-GONE)
WM/ mr64.cache.dirty
Cache0 (P)
Cache1 (MD)
1.Write Miss to Directory
WMR/ mr64.dirty.ack
2 Write Miss Reply to Reading Cache and Directory points to writing cache
Read-exclusive/ cr00.copy.valid
3.Read-exclusive request
Read-Excl-R/ cr00.copy.ack
Cache1 (IN)
4.Read-exclusive Reply to the requester
5.Read-exclusive request completes
Legend:
(1) : Message(Request/Reply)
(2) : Pointer(forward/backward)

Figure 7 The handling procedure for a write accesses the migratory-sharing memory block under the situation of only one cache copy

Protocol	Description
OAD1-INV	The DD-INV protocol with the implementation option 1.
OAD2-INV	The DD-INV protocol with the implementation option 2.
OAD3-INV	The DD-INV protocol with the implementation option 3.
OAD1-CUn	The DD-CU protocol with the implementation option 1; n is the threshold.
OAD2-CUn	The DD-CU protocol with the implementation option 1; n is the threshold.
OAD3-CUn	The DD-CU protocol with the implementation option 1; n is the threshold.
OAD1-UP	The DD-UP protocol with the implementation option 1.
OAD2-UP	The DD-UP protocol with the implementation option 2.
OAD3-UP	The DD-UP protocol with the implementation option 3.

Table 1 Summary of the implementation options

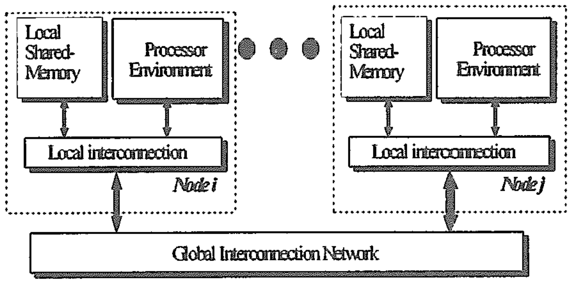

Figure 8 Our simulated architecture

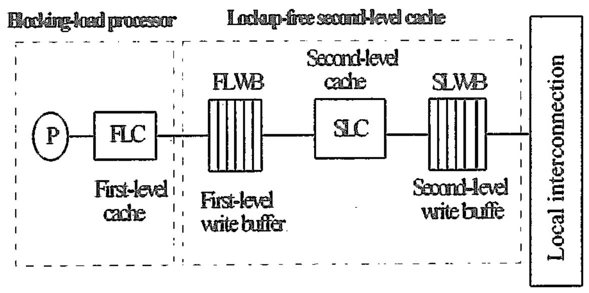

Figure 9 The processor environment

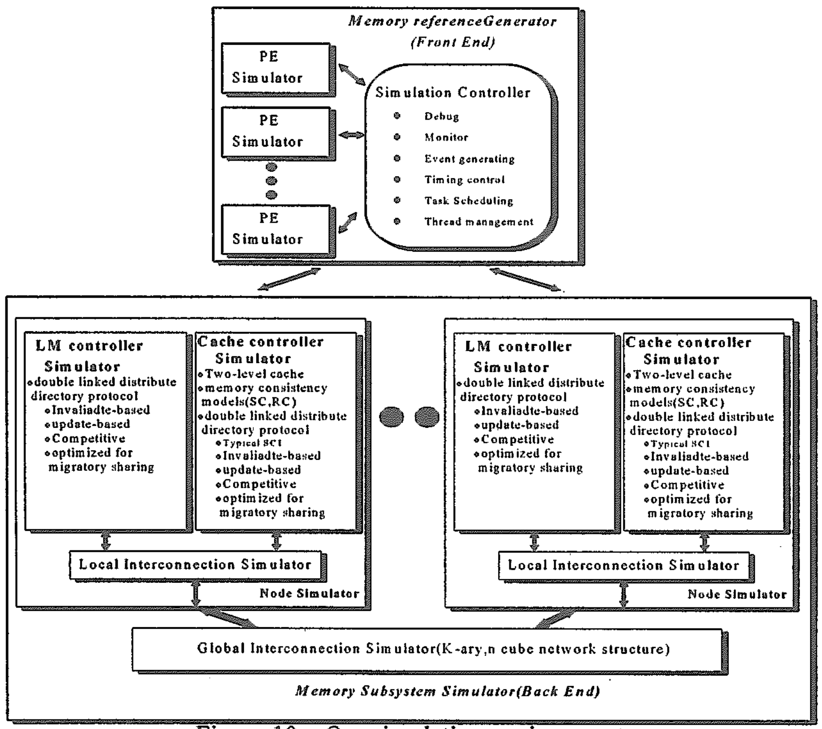
Memory referenceGenerator (Front End)
PE Simulator
PE Simulator
PE Simulator
Simulation Controller
Debug
Monitor
Event generating
Timing control
Task Scheduling
Thread management
LM controller Simulator
double linked distribute directory protocol
Invaliadte-based
update-based
Competitive
optimized for migratory sharing
Cache controller Simulator
Two-level cache
memory consistency models(SC,RC)
double linked distribute directory protocol
Typical SCI
Invaliadte-based
update-based
Competitive
optimized for migratory sharing
Local Interconnection Simulator
Node Simulator
LM controller Simulator
double linked distribute directory protocol
Invaliadte-based
update-based
Competitive
optimized for migratory sharing
Cache controller Simulator
Two-level cache
memory consistency models(SC,RC)
double linked distribute directory protocol
Typical SCI
Invaliadte-based
update-based
Competitive
optimized for migratory sharing
Local Interconnection Simulator
Node Simulator
Global Interconnection Simulator(K-ary,n cube network structure)
Memory Subsystem Simulator(Back End)

Figure 10 Our simulation environment

Architecture Parameters

Parameter	Value
Number of Processing nodes	16
Size of FLC	32Kbytes
Size of SLC	256Kbytes
Block size of FLC and SLC	64bytes
Number of entries in FLWB	8
Number of entries in SLWB	16

Table 2 Architecture Parameters

Benchmark Programs

Benchmark	Description	Data sets
MP3D	3-D particle-based wind-tunnel	5K particles,10 time steps
Ocean	Ocean basin simulator	66x66 grid,tolerance 10^{-7}
PTHOR	Distributed time digital circuit simulator	RISC circuit,1000 time step

Table 3 Benchmark Programs

Benchmark	Shared Reads (M)	Shared Writes (M)	R/W Ratio	Locks	Barrier	Avg. Invals Per Write
MP3D	1.29	0.71	1.8	104908	116	0.92
Ocean	17.59	4.04	4.3	4000	2084	1.15
PTHOR	8.25	0.91	9.0	117735	482	1.44

Table 4 The statistics and characteristic of access for benchmark programs

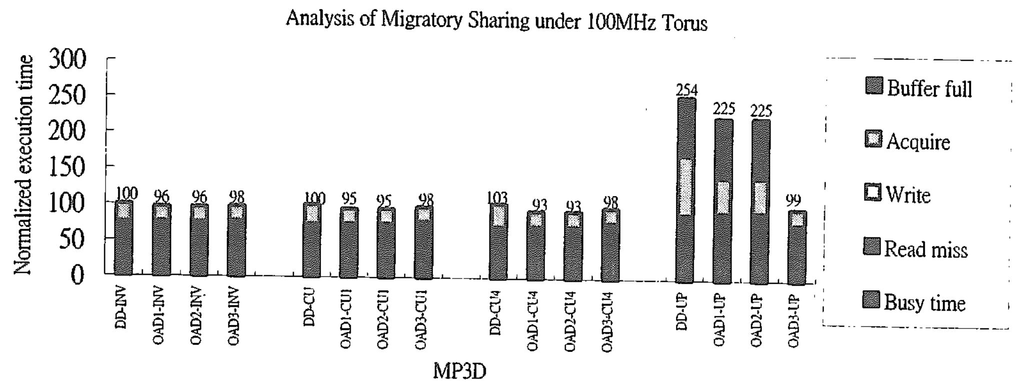

Figure 11 Evaluation statistics for migratory sharing under MP3D

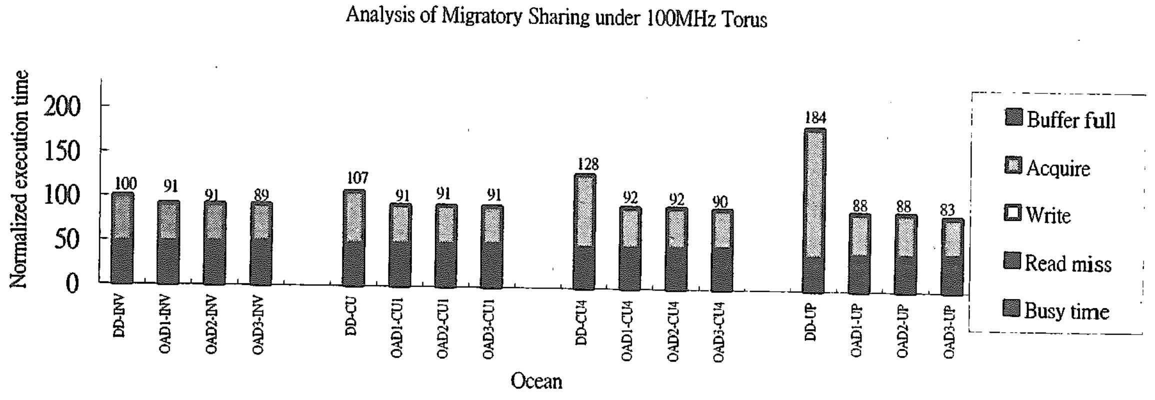

Figure 12 Evaluation statistics for migratory sharing under Ocean

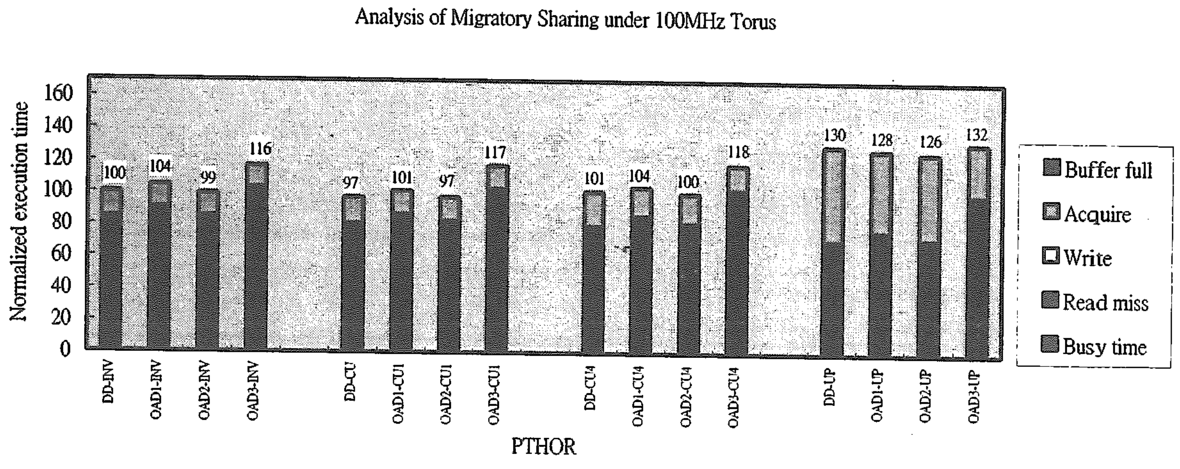

Figure 13 Evaluation statistics for migratory sharing under PTHOR