International Journal of Advanced Information Science and Technology (IJAIST) *ISSN: 2319:2682*
Vol.5, No.3, march 2016 **DOI:10.15693/ijaist.2016.v5.i3.29-38**

Directory Based Cache Coherence Modellar in Multiprocessor using Scalable Cache Coherence(SCI)

ManojJadhav
SCOPE, VIT University,Vellore

G. Gopichand
SCOPE, VIT University, Vellore

ABSTRACT

In computational system such as multiprocessors and uniprocessor need to avoid the cache coherence because cache coherence arise the problem when there is a data inconsistency among main memory and private cache. The problem of cache coherence is solved by implementing the cache coherence protocol such as snoopy based or directory based protocol. The cache coherence protocol affects the performance of distributed shared memory multiprocessor system. In general snoopy based protocol is used for small networks and directory based protocol is used for large scale distributed network. In this paper we have proposed the coherence model based upon cache based principle that is IEEE standards for Scalable coherent Interface (SCI). SCI supports the shared and distributed memory with cache coherence for tightly coupled system and message passing for loosely coupled system. It also supports efficient multiprocessor lock transactions, which is used for read and write operation of cache memory.

Keywords: Cache coherence protocol, SCI, Distributed shared memory.

1. INTRODUCTION

Cache memories are introduced into computers in order to bring data closer to the processor and hence to reduce the memory latency. In multiprocessor machines where many processors required a copy of same memory block to be cached in their local cache, the maintenance of consistency among these copies raises the problem of cache coherence. In general there are three causes sharing of writable data, Process migration, I/O activity. To handle the problem of cache coherency following techniques can be used

- Hardware based protocols
- Software based schemes

1.1 Hardware based protocols:

This protocol provides general solution to the problem of cache coherence. Hardware based protocols can be classified according to three major consideration as follows

a. **Memory Update Policy:** When the processor finds a word in cache during read operation main memory not involved in transfer but in case of write operation there are two commonly used procedures to update memory.
   - Write Through policy: Both cache and memory is updated during every write operation.
   - Write back policy: Only cache is updated and location is marked so that it can be copied latter into main memory.Hence it is faster than write through policy.

b. **Cache Coherence policy:** Similar to memory update policy for updating copies data,a greedy and lazy cache coherence policy has been introduced.
   - Write update policy(a greedy policy): Whenever processor updates a cached data it immediately updates all other cached copies.
   - Write Invalidate policy (a lazy policy): The updated cache block is not sent immediately to other caches, instead simple invalidate command is sent to all other cached copies and to original version in shared memory.

c. **Interconnection Scheme:** Hardware based protocols can be further classified based on the nature of interconnection network applied in the shared memory system.
   - Single bus Snoopy cache protocol: This scheme is typically used in single bus based shared memory system where invalid or update command are broadcast via the bus and each cache snoops the bus for the incoming consistency command.
   - Multistage Directory scheme: In this scheme a directory must be maintained for each block of shared memory to manage the actual location of blocks in the possible caches.
   - Multiple Bus Hierarchical Cache coherence protocol: in this protocol multiple bus network with the application of hierarchical cache coherence protocol that are generalized or extended version of single bus snoopy protocol. In which each parent keeps track of exactly its immediate children has a copy of block

1.2 Software Based Scheme:

Software based approach represents a good and competitive compromised since they require nearly negligible hardware support and they can lead to the same small number of invalidation misses as hardware based protocol. All software based protocol relies on the computer assistance. The compiler analyses the program and classifies the variable into four classes

a. Read only for any number of processes: Read only variables can be cached without restrictions
b. Read only for any number of processes and read write for one process: variables can be cached by read wire process alone and the main memory must be kept always consistent by using write-through policy
c. Read write for exactly one process: variable can be cached only for that process since only one process uses this variable. Variable can be cached and updated by write-back policy.
d. Read write for any number of processes: variable must be marked as a non-cacheable that is must not be cached in software based schemes.

Cache coherence is an important part of the proposed standard. Current mechanisms prove insufficient when the number of processors increases dramatically. This calls for a new approach to the cache consistency problem. The SCI working group is defining a scalable distributed directory scheme where processors sharing cache lines are linked together by pointers stored in the caches.

1.3 Implementation Schemes

The previous section describes the cache coherence problem and introduces the coherence protocols as the agents that solve the coherence problem. There are two main implementation schemes of cache coherence protocols, bus-based protocols (snoopy) and directory-based protocols. In this section, we present the two dominant hardware schemes that are used to enforce the cache coherence property.

1.3.1 Bus Based Protocol (Snooping)

Shared memory systems that are based on a shared broadcast medium follow the Snooping approach; no parts of memory are assigned to any processor. Assuming a single level of private caches, a processor that requests to access a memory block, which does not, resides in its local cache, it sends a message to all the other caches and main memory. All the caches snoop the traffic on the interconnection network to identify a new message. If no cache has a copy of the requested block then the block is loaded from main memory. If, however, one or more caches maintain a valid copy, one of them sends the requested block back to the cache that requested it. Messages are used not only to facilitate data transferring. Every message is assigned a type, which has a specific meaning for the coherency protocol. Based on this type caches that receive such messages are becoming aware of the intention of the requesting processor. Having this knowledge, caches are able to follow the steps imposed by the coherency protocol. This category of coherency protocols adds a requirement to the interconnection network properties, which constitutes the basic property of the protocol. This requirement refers to the ability that must be offered to any cache to broadcast messages and also to snoop the bus activity. Otherwise, it is impossible for the distributed protocol to synchronize the requests of processors. In snooping protocols the bus act as the serialization point for coherence transactions.

1.3.2 Directory Based Protocol

The use of an arbitrary multi-stage interconnection network poses challenges to the implementation of cache-coherent shared memory. Although connecting the processing nodes on a scalable network topology, i.e. (Mesh, Hybercube), yields to potentially more bandwidth efficient system, it also takes away the inherent broadcast capabilities of a shared bus that can be exploited to implement broadcast-based coherence. Instead, such systems are based on tracking which processor cache contains a memory line, to send the number of necessary messages, and avoid broadcasts. Sharing information is kept in an auxiliary data structure called a directory, illustrated in Figure 1.1. Furthermore, directory information can be distributed to multiple directory engines to avoid the performance bottleneck of a single, monolithic directory. Each node or group of nodes is associated with a directory corresponding to the locations in that node's group local memory.

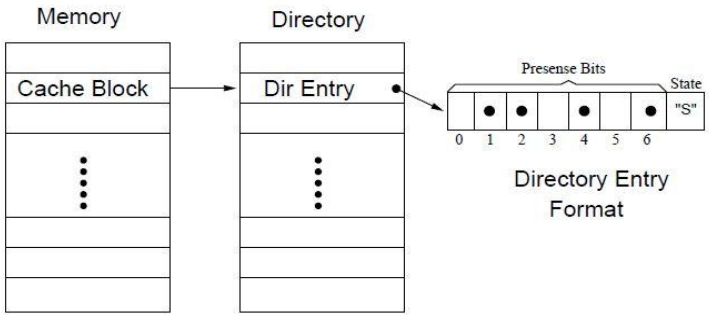

Fig.1.1 Simple Directory Structure

As shown in Figure 1.1, an example of one node's group directory contents. The directory consists of a collection of directory entries, one for each memory block in the node's local memory. Because the processor caches interface to the system at a cache line granularity that is, each processor cache miss or write back transfers a single cache line of data between memory and the cache size of the memory block tracked by each directory entry is usually one cache line. In its simplest form, a directory entry contains two fields: a state indication and a presence bit vector. In invalidation-based protocols the state indication specifies whether the memory line associated with the directory entry is held shared (i.e., read-only) in one or more caches or whether it is held exclusive (i.e., with read/write permission) in a single processor's cache. The presence bit vector indicates which processors are caching the memory line; if the memory line is held exclusive, only one presence bit may be set.

The directory entry depicted in Figure 1.1 shows a case in which the corresponding memory line is held shared, indicated symbolically by the "S" in the state field, and is present in the caches of processors 1, 2, 4, and 6, indicated by the presence bit vector. When a memory request arrives at a processing node, the controller of the node then retrieves the corresponding directory entry to determine what additional actions are required to service the request. For example, as shown in Figure 1.2, if processor 3 requested exclusive access to the memory line, the memory line first must be removed, or invalidated, from all processor caches currently holding it. In a distributed system, the controller of the node must consult the presence bit vector to determine that explicit invalidation messages need to be sent to processors 1, 2, 4, and 6. In a bus-based system, these invalidations would be performed automatically when processor 3's exclusive request was issued on the bus. This is a simplified case is just one example of the operation of a distributed cache coherence protocol. In practice, these protocols are complex, especially because so many race conditions can occur as a result of the lack of a shared bus to serialize all processors' memory requests.

1.4 Directory Organizations

Directory-based cache coherence protocols have been used for long in shared memory multiprocessors. These protocols introduce directory memory overhead due to the need of keeping the sharing status of a memory block in a directory structure. In the past, this structure would provide an entry for every block of main memory and, because of its size, was kept in DRAM. The directory information represents memory overhead as it adds state information either for each cached or also for each non-cached memory block in the system, depending on the directory organization. However, this overhead could become very high depending on both the sharing code and the number of cores that comprise the multiprocessor system, and even be in large systems prohibitive. In this section, we study a directory organization for CMPs that addresses the problem discussed above. Then it reviews the main alternatives for storing the directory information and offers a proposal to optimize look-up time for the directory organization used in this work.

Moreover, the straightforward way of tracking sharers of a block is by using a full-map sharing code where each bit represents a core in the system, which is set when that cache holds a copy of the block. The size of this directory structure scales with the number of cores (P) in the system. In particular, the order of its size is (P * M), where M is the number of memory entries and P is the number of cores in the system. For the purposes of this discussion on directory state organizations, we assume a single level of caches, since this is sufficient. Thus, the number of caches and the number of cores is assumed the same. Based on the memory is distributed among the nodes the two categories of directories schemes are the centralized and the distributed schemes, where the memory is distributed and multiple directories are responsible for a portion of the address space.

As shown in Figure 1.2 the two alternatives for finding the source of the directory information for a block are known as flat directory schemes and hierarchical schemes. The taxonomy that is showed, also divides Flat schemes into two categories based on the way they use in order to locate the copies of the memory blocks. In the following sections we analyze the distributed schemes categories.

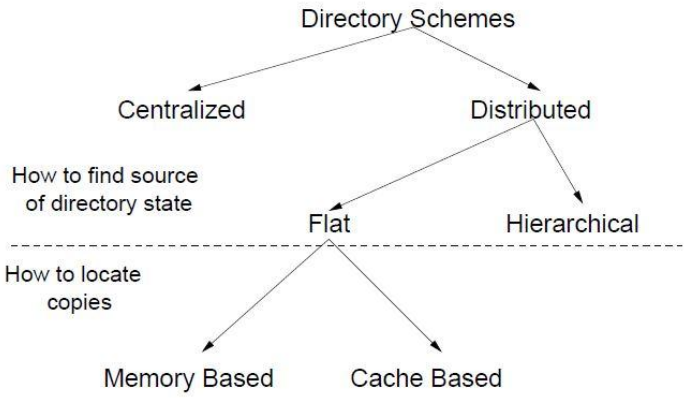

Fig 1.2. Directory Schemes

1.4.1 Flat Schemes

Flat schemes are more popular than hierarchical, and they can be classified into two categories: memory-based schemes and cache-based schemes. Memory based schemes store the

directory information about all main memory blocks, or only cached copies, at the home node of each block. The conventional architecture of Figure 1.2 which uses the full-map sharing code, is memory based.

Examples of memory based system are the Stanford FLASH/DASH and SGI Origin systems. In cache-based schemes (also known as chained directory schemes), such as the IEEE Standard Scalable Coherent Interface (SCI), the information about cached copies is not all contained at the home but is distributed among the copies themselves. The home node contains only a pointer to the first sharer in a distributed double linked-list organization with forward and backward pointers. The locations of the copies are therefore determined by traversing the list via network transactions. The most important advantage of cache-based directory schemes is their ability to significantly reduce directory memory overhead, since the number of forward and backward pointers is proportional to the number of cache entries, which is much smaller than the number of memory entries. Several improvements have been proposed for chained directory protocols and commercial multiprocessors have been designed according to these schemes, such as Sequent NUMA-Q, which has been designed for commercial workloads, and Convex Exemplar multiprocessors, destined to scientific computing. Nevertheless, these schemes increase the latency of coherence transactions as well as overload the coherence controllers and lead to complex protocols implementations. In addition, they need more cache states and extra bits for forward and backward pointers, which imply changing processor caches. These factors make more popular memory-based schemes than cache-based ones. The problem of the directory memory overhead in memory-based schemes is usually managed from two separate points of view: reducing directory width and reducing directory height. The width of the directory structure is given by the directory entries and it mainly depends on the number of bits used by the sharing code. The height of the directory structure is given by the number of entries that comprise the directory. In the following subsection we discuss the two alternatives that try to reduce the directory memory overhead.

1.4.2 Hierarchical Schemes

Hierarchical memory schemes treat the processing cores as the leaves of a logical tree, with main memory distributed along with the processing nodes. Every block is assigned to a home node (leaf) in which it is allocated, but this does not mean that the directory information is maintained or rooted there. The internal nodes of the tree are not processing cores and only hold directory information. Each such directory node keeps track of all memory blocks that are being cached or recorded by its sub-trees and it uses a presence vector per block to tell which of its sub-trees have copies of the block and a bit to tell whether one of them has it dirty. It also records information about local memory blocks that are being cached by processing nodes outside its sub-tree. This information is used then to decide when requests originating within the sub-tree should be propagated further up the hierarchy. In general, the advantages of hierarchical schemes are tightly related to the amount of locality shown by memory accesses, as the delay is high if all the buses/levels that need to be traversed to serve a high percentage of the memory accesses. The main drawback of such schemes is the latency problem, because the number of network transactions sent up and down the hierarchy to satisfy a request tends to be larger than in a flat memory-based scheme. Even though these transactions may be more localized in the network, each one is a network transaction that also requires either looking up or modifying the directory at its (intermediate) destination node. This increased endpoint overhead at the nodes along the critical path tends to prevail any reduction in the total number of network hops traversed and hence network delay, especially given characteristics of modern networks.

2. SCALABLE COHERENT INTERFACE (SCI)

The SCI standard contains two levels of interface, a physical level and a logical level. The physical level specifies electrical, mechanical and thermal characteristics of connectors and cards that meet the standard. The logical level describes the address space, data transfer protocols, cache coherence mechanisms, synchronization primitives and error recovery. SCI support multiprocessing with cache coherence for the general distributed shared memory. In this paper we are dealing with the cache coherence.

High-performance processors use local caches to reduce effective memory-access times. In a multiprocessor environment this leads to potential conflicts; several processors could be simultaneously observing and modifying local copies of shared data. Cache-coherence protocols define mechanisms that guarantee consistent data are locally cached and modified by multiple processors. The SCI cache-coherence protocol can be hardware based, thus reducing both the operating system complexity and the software effort to ensure consistency. Many cache-coherence protocols rely on the broadcasting of all transactions. This broadcasting allows use of eavesdropping and intervention techniques to achieve data consistency.

2.1 Distributed Directories

SCI uses a distributed directory-based cache-coherence protocol. Each shared line of memory is associated with a distributed list of processors sharing that line. All nodes with cached copies participate in the update of this list. Every memory line that supports coherent caching has an associated directory entry that includes a pointer to the processor at the head of the list. Each processor cache-line tag includes pointers to the next and previous nodes in the sharing list for that cache line. Thus, all nodes with cached copies of the same memory line are linked together by these pointers. The resulting doubly linked list structure is shown in figure 1.3.

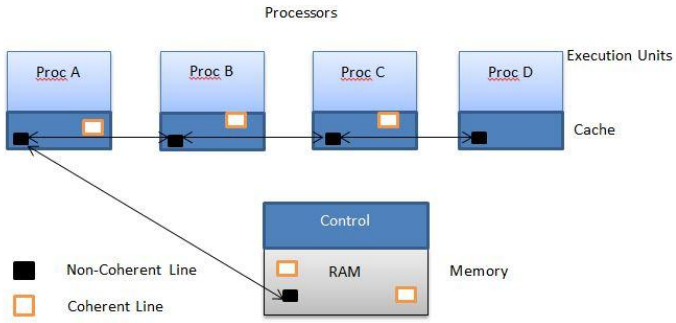

Fig.1.3 Distributed sharing list Directory.

Note that this illustrates the logical organization of the directory's sharing-list structure for one line, which may be different for each line that is cached. The processors are always shown on the top and the shared memory location is shown on the bottom. These logical illustrations should not be confused with the physical topology of a system; SCI expects that processors and memory will often be found on the same node. Coherence protocols can be selectively enabled based on bits in thee processor's virtual-address-translation tables. Depending on processor architecture and application requirements, pages could be coherently cached, non-coherently cached, or not cached at all.

This distributed-list concept scales well. Even when the number of nodes in a list grows dramatically, the memorydirectoryand processor-cache-tag sizes remain unchanged.The list pointer values are the node addresses of the processors (caches). When a node accesses memory to get a copyof coherently shared data, memory saves the requesting node's address. If there are currently no cached copies, therequesting node becomes the head of a new list. If othernodes have cached copies of the data, the pointer to the head of the sharing list is returned from memory. Therequesting node inserts itself at the head of the list and gets its data from the previous head.SCI supports both weak and strong sequential consistency, as determined by the processorarchitecture. A weakly ordered write instruction can be executed before the sharing-list purge completes, while astrongly ordered write must wait for purge completion.

2.2 SCI Node Model

An SCI node needs to be able to transmit packets while concurrently accepting other packets addressed to itself and passing packets addressed to other nodes. Because an input packet might arrive while the node is transmitting an internally generated packet, FIFO storage is provided to hold the symbols received while the packet is being sent. Since a node transmits only when its bypass FIFO is empty, the minimum bypass FIFO size is determined by the longest packet that the node originates. Idle symbols received between packets provide an opportunity to empty the bypass FIFO in preparation for the next transmission. Input and output FIFOs are needed in order to match node processing rates to the higher link-transfer rate. Since there is no facility for delaying the transmissions of symbols within a packet, each node ensures that all symbols within one packet are available for transmission at full link speed. Similarly the node is able to receive a packet at full speed. Since node application logic is not expected to match the SCI link speeds, FIFO storage is needed for both transmit and receive functions, as illustrated in figure 1.4.

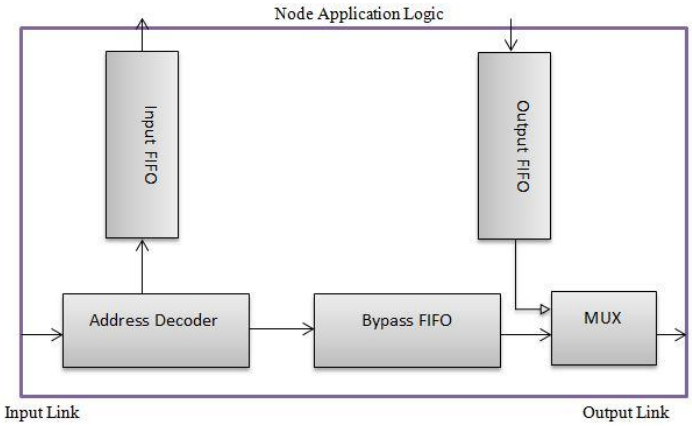

Fig.1.4 SCI node Model

3. LITERATURE SURVEY

There are different types of directory based cache coherence protocols are available to minimize the cache coherence problem, but each protocol has its drawbacks in the form of memory overhead. Thus proposed four directory-based protocols, but for memory overhead calculations they just used

three of them (full map, chained, limited directory) and after this calculation they compare it on the basis of memory overhead to find out the most suitable protocol according to the size of shared memory system. Finally came to the conclusion that as the no of processors are varies the overhead also changes, and for the same no of processors different protocols has different memory overhead requirements. Thus by comparing their overhead we can easily find out which protocol is suitable for a shared memory system. They have made the following comparison.

Number of Processors				
Scheme	P=32	P=64	P=128	P=156
Full Map Directory	1.027	2.024	4.016	8.0002
Limited Directory	0.68556	0.80958	0.93409	1.0586
Chained Directory	0.2188	0.25512	0.28137	0.31262

Table 1: The memory overhead for various schemes for given number of processors

These protocols have their advantages and disadvantages in the form of memory overhead and cache miss (read/write). Thus on the basis of these advantages and drawbacks we must chose the appropriate protocol for a particular network. After analysis and comparing the memory overhead for various schemes show how well these schemes scale, as the no of processors grows full memory directory scheme has a sharp increase in a memory overhead hence it is poorly scalable. Limited and chained protocols have similar values. Limited directory has the disadvantages of, only small set of processor can share a data block so it is not preferable. Chained directory have lower memory overhead and it scales gracefully with additional processors. Chained directory can be candidate choice in term of lower memory utilization.

The parallelism can be improved if more than one operation is executed by different processor at the same time. So in this case replication of data is required and more than one write request is to be done. But multiple write-requests create some inconsistent result when the location is same for all write. The framework finds the address of updates and propagates it to all nodes. The coherency protocol is required to maintain memory consistency for ensuring the serialization of write operation and that any subsequent reads or writes access the update data. The updates are propagated by the remote site. Now it is upto the coherency protocol to take these update and incorporate the update into the local copies.

4. PROPOSED WORK

This proposed coherence model supports the sharing of fresh or dirty data and provides special read and write optimizations. This is a useful model that efficiently supports the sharing of read-only instructions/data as well as read/write data. This model better illustrates the complexity of a typical implementation. Each of the sharing-list states is defined by the state of the memory, *MState*, and the states of the entries in the sharing list, *CState*. In normal operation, the memory state is either HOME, FRESH, GONE or WASH. The sharing-list state names have two components.

The first component specifies the location of the entry in amultiple-entry sharing list (HEAD, MID, or TAIL), or identifies the only entry in the sharing list (ONLY). The secondcomponent specifies the entry's caching properties (FRESH, CLEAN, DIRTY, VALIDetc.). Since the head normally administers the return of dirty data to memory, it differentiates between FRESH(must be the same as memory) and the other(can modify without informing memory) states.

SrNo.	Name	Description
1	HOME	No sharing list
2	FRESH	Sharing-list copy is the same as memory
3	GONE	Sharing-list copy may be different from memory
4	WASH	Transitional state (GONE to FRESH)

Table 2: Stable sharing list

Fresh copies. The *fresh* memory state indicates that all shared copies are read-only; the data can be returned from memory when a processor is attaching to the head of the previous sharing list.

Step 1: Read-only fetch

1. Memory is in the HOME state and all caches are INVALID.
2. When fetching a read-only copy, the sharing-listcreation begins at the cache, where an entry is

changed from the INVALID to the PENDING state, and anmread64.

3. CACHE_FRESH transaction is generated to obtain a coherently cached copy.
4. The read updates (1) the memory-directory state (from HOME to FRESH), and the new entry state is changed accordingly (from PENDING to ONLY_FRESH), as illustrated in figure 2-1.

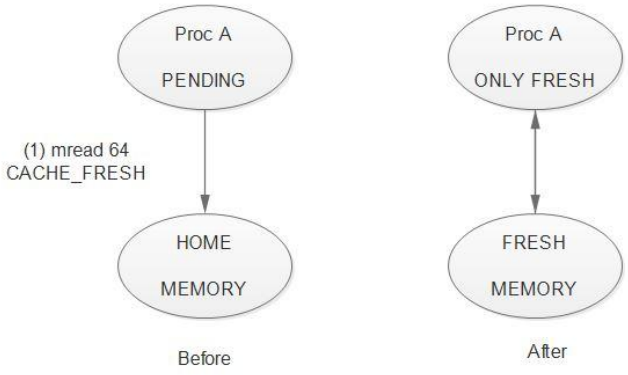

Fig. 2-1 FRESH list Creation

Leaving the memory in a FRESH state minimizes the memory-access latencies for subsequent reads, since FRESH data can be provided by memory before the new sharing-list head attaches to the existing sharing list.For subsequent accesses, the memory state is FRESH and the head of the sharing list has the unmodified data. Whenread-only data are accessed (1), fresh data are returned from memory and the new requester then attaches (2) to the oldsharing-list head. These steps are illustrated in figure 2-2 for an mread64.CACHE_FRESH request when memory isin the FRESH state.

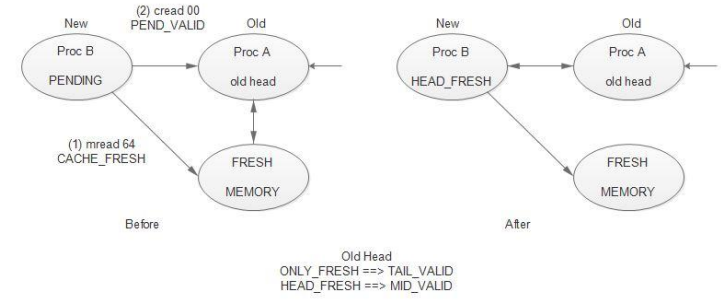

Fig. 2-2 FRESH addition to FRESH List

When the memory state is GONE, the head of the sharing list has the (possibly modified) data. The fresh data that isrequested (1) cannot be returned from memory, but the dirty sharing-list copy is returned (2) when the new requesteris attached to the old sharing-list head. These steps are illustrated in figure 2-3, for an mread64.CACHE_FRESH request when memory is in the GONE state.

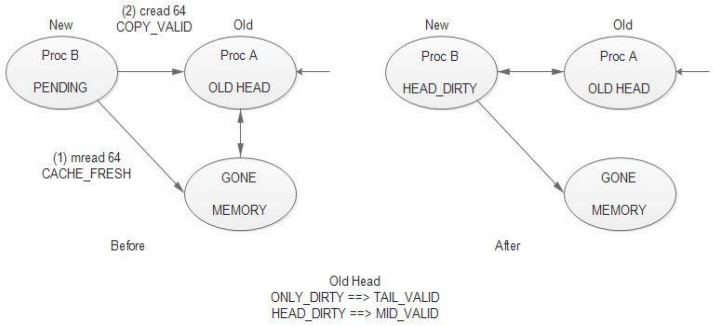

Fig. 2-3 FRESH addition to DIRTY List

The final state of the old sharing-list head is a function of the old head's initial state. The state of the new sharing-list head is HEAD_DIRTY. The states of the other mid and tail entries are unaffected by sharing-list additions.

Step 2: Read-write fetch

If a later write is expected, a data-cache miss may be designed to fetch a modifiable (but not yet modified) copy. In this case, the read64.CACHE_CLEAN transaction is used (1) to fetch modifiable (but not immediately modified) data from memory. A FRESH memory state returns its data before the memory-tag state is thanked to the GONE state. After prepending (2) to the old sharing list, the sharing list is left in the HEAD_DIRTY state, as illustrated in figure 2-5.

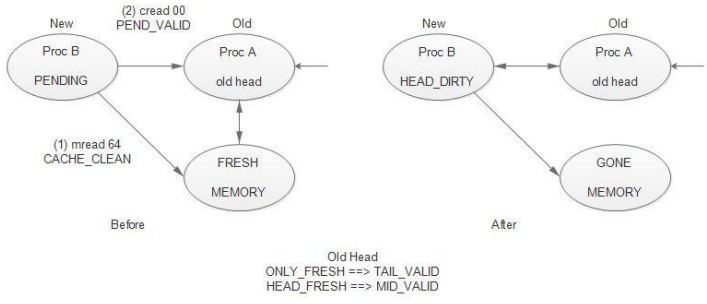

Fig. 2-4 DIRTY addition to FRESH List

The read64.CACHE_CLEAN transaction could access (1) a GONE memory state. In this case, the memory state is unchanged and no data are returned. The dirty data are eventually returned (2) when attaching to the old sharing list, as illustrated in figure 2-5.

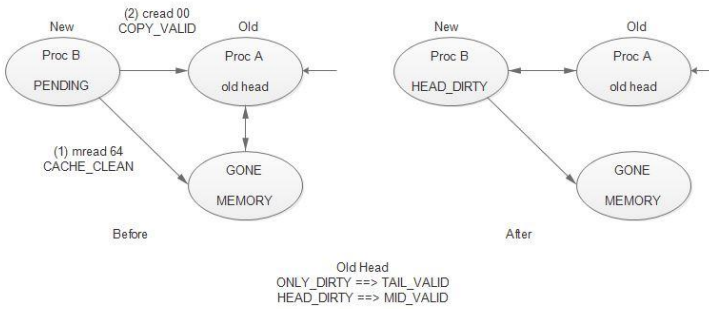

Fig. 2-5 DIRTY addition to DIRTY List

Step 3: Data modifications

Data in the HEAD_DIRTY state may be modified immediately, before the remaining sharing-list entries are invalidated. After data are modified, the head of a modifiable sharing list (HEAD_DIRTY) purges the remaining sharing-list entries. For the typical set of options, the initial transaction to the second sharing-list entry purges (1) that entry from the sharing list and returns its forward pointer. The forward pointer is used to purge (2) the next (formerly the third) sharing-list entry. The process continues until the tail entry is reached, as illustrated in figure 2-6.

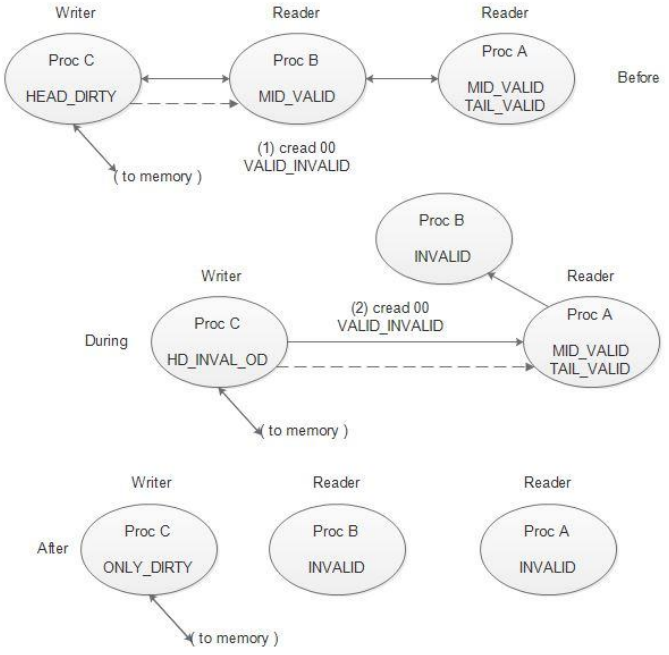

Fig 2-6. - Head Purging Others

Concurrent deletions may temporarily corrupt the *backId* pointers in one or more of the sharing-list entries. Since the head-initiated purge uses only the *forwId* pointers, the purges and deletions can safely be performed at the same time.The purging state (HD_INVAL_OD) is similar to the PENDING state, in that new sharing-list additions are delayedwhile the purges are being performed. Note that purge latencies increase linearly with the number of sharing readers.Since purge lists are often short, the linear latencies may be acceptable in many systems. An ONLY_FRESH entry is changed to the ONLY_DIRTY state before the data are modified. This requires an additional memory-access transaction (1) mread00.LIST_TO_GONE, which changes the memory-directory state from FRESH to GONE, as illustrated in figure 2-7.

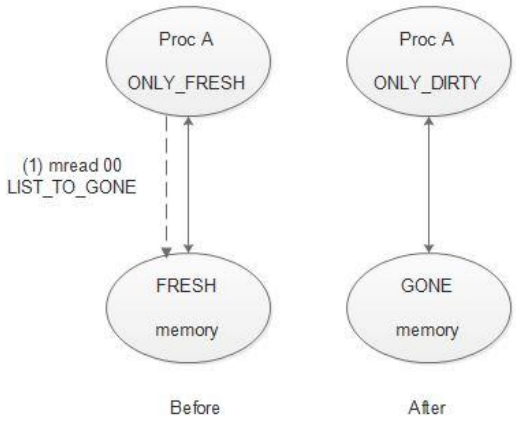

Fig 2-7. ONLY_FRESH list conversion

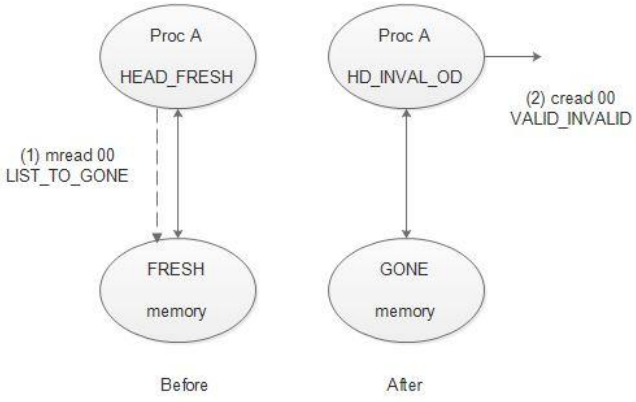

Fig 2-8. HEAD_FRESH list conversion

Similarly, a HEAD_FRESH entry is changed to an intermediate modifiable (HEAD_DIRTY) state before the data are modified and the other sharing-list entries are invalidated. The memory-access transaction(1)mread00.LIST_TO_GONE is used to change from the HEAD_FRESH to HEAD_DIRTY state, the data modifications are performed, and the cache-line state is changed to an intermediate HV_INVAL_OD state. The other copies are then invalidated (2), as illustrated in figure 2-8.

The mread00.LIST_TO_GONE transaction's update of memory state is conditional; if the memory directory points to a newly queued cache entry the update is nullified. This nullification is detected by the sharing-list head, which then deletes itself from the sharing list and re-attaches in a modifiable (ONLY_DIRTY or HEAD_DIRTY) state.

Step 4: Mid and head deletions

Entries can also be deleted from the list by their own controller when they are needed to cache data at other addresses (cache-line rollout). The sharing-list deletions involve the update of the *backId*in the next (closer to the tail) entry, and the *forwId* pointer in the previous (closer to memory) entry. Before the deletion begins the entry is converted into a locked state. A MID_VALID entry is converted into the locked MV_FORW_MV state and transactions (1 and 2) to the adjacent sharing-list entries are generated, as illustrated in figure 2-9.

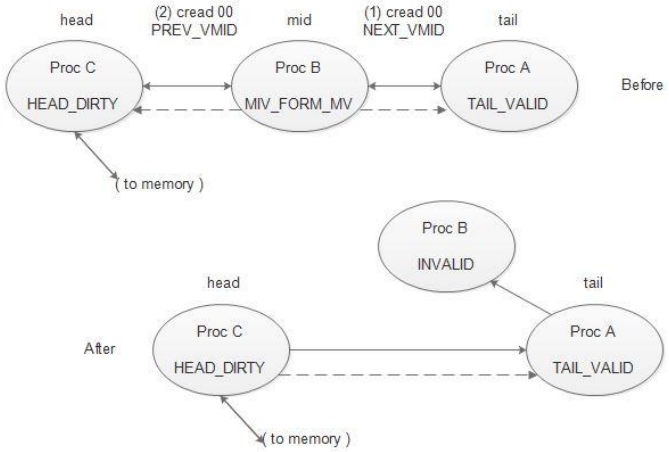

Fig 2-9. - Mid-Entry Deletion

Head entries can also delete themselves from the list, e.g., when they are needed to cache data at other addresses(cache-line rollout). The sharing-list deletions involve (1) the update of the backId in the next (closer to the tail) entry, and (2) the forwId pointer in the memory directory, as illustrated in figure 2-10.

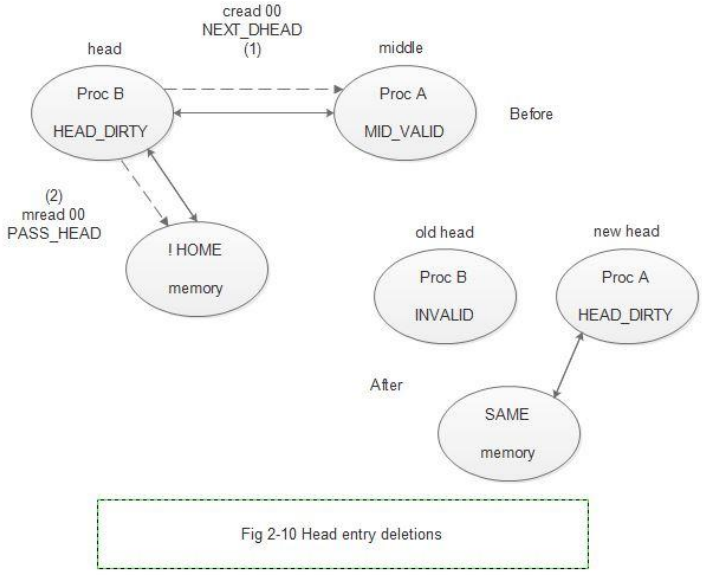

Fig 2-10 Head entry deletions

Recovery from detected transmission errors is usually possible when a single write transaction is used to collapse an ONLY_DIRTY sharing list, but cannot be guaranteed. Multiple transmission errors during a particular set of sharinglisttransitions can leave the sharing-list in an uncorrupted (the data won't be incorrectly recovered) but unrecoverable (the correct data can't be recovered) state.Therefore the fault-tolerance of the SCI system may optionally be improved by using two transactions: the firsttransaction returns (1) the dirty data and the second transaction collapses (2) the sharing list. These two steps areillustrated in figure 2-11.

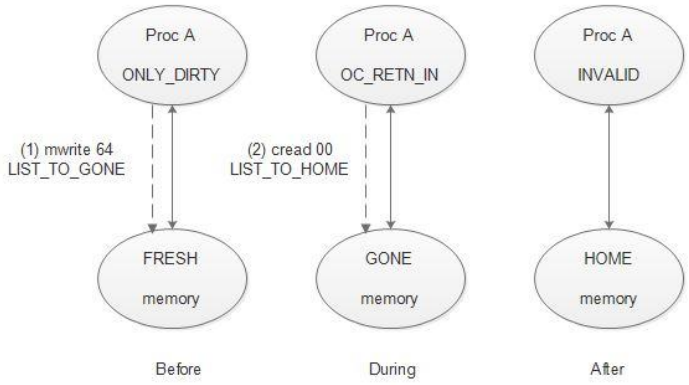

Fig 2-11. Robust ONLY_DIRTY deletion

5. CONCLUSION AND FUTURE SCOPE

Protocols for cache coherence are critical to multiprocessor systems. In general, the directory based protocol is more used

for larger systems to increase their performance; while snooping protocol is used for smaller systems. In this paper we proposed the coherency modeler based upon IEEE standard for Scalable Cache Coherence (SCI) and it provides design direction for multiprocessor scenario. Also we have analyzed the various design challenges of the directory based protocol and proposed a state machine based architecture for coherent modeler and using this design we are planning to verify it experimentally for its performance and validity.

The SCI standard intends to support several compatible future extensions. This allows implementations to quickly use the existing specification, while providing opportunities to expand the SCI capabilities when more experience is available. Although the future extensions are beyond the scope of the SCI standard, a short overview is intended to provide the reader with insights on how this standard may evolve in the future. The SCI standard supports the concept of delaying distribution of shared data, by queuing additional requesters until a cache line has been released by its current owner. A future extension to the SCI coherence protocols could implement a more-transparent lock bit, by providing an out of band lock bit for every 64-byte cache line.

6. REFERENCES

[1]. Juan Gomez-Luna Herruzo and Jose Ignacio Benavides, MESI Cache CoherenceSimulator for Teaching Purposes. CLEI ELECTRONIC journal pp1-7, 2009.
[2]. John L. Hennessy, David A. Patterson, David Goldberg, Computer Architecture: Aquantitative Approach, fourth edition, P579.
[3]. Yong J. Jang and Won W.R. Evaluation of Cache Coherence Protocols onMulti-CoreSystems with Linear Workloads, ISECS International Colloquium on Computing,Communication, Control, and Management, pp 1-4, 2009.
[4]. Aanjhan Ranganathan, Experimental Analysis ofSnoop Filters for MPSoC EmbeddedSystems, Ecole Polytechnique Federale de Lausanne, pp10.
[5]. Richard Simoni, Implementing a Directory-Based Cache Consistency Protocol, DARPA,pp 1-2, 1990 .
[6]. M. Heinrich, J. Hennessy, and A. Gupta, The Performance and Scalability of DistributedShared Memory Cache Coherence Protocols, IEEE Transactions on Computers, pp 1-7, 1999.
[7]. Wong Pak Shing, An effective model of cache coherence protocol with VHDL simulation. Object oriented computing pp16-18, 2008.
[8]. Blas Cuesta, Alberto Ros, Marı´a E. Go´mez, AntonioRobles, and Jose´ Duato, "Increasing the Effectivenessof Directory Caches by Avoiding the Tracking ofNoncoherent Memory Blocks," IEEE Transaction onComputers, vol.62, no.3, March 2013.
[9]. Huang Yongqin Yuan Aidong ; Li Jun ; HuXiangdong, "A Novel Directory-Based Non-busy,Non-blocking cache coherence," IFCSTA '09.International Forum on Computer Science-Technology and Applications, 2009, vol.1, 25-27Dec2009, pp.374-379
[10]. G. Keramidas and S. Kaxiras, "SARC Coherence: ScalingDirectory Cache Coherence in Performance and Power(preprint)," *IEEE Micro*, 2010.

BIOGRAPHIES

ManojJadhav received the B.E. degree in Computer Science and Engineering from Walchand Institute of Technology, Solapur, Solapur University, Solapur India in 2012. He is pursuing the M.Tech. in Computer Science and Engineering from VIT University, Vellore, India. His research interest includes Operating Systems and design, Big Data Analysis and cloud Computing.

Mr G.GOPICHAND is currently working as an Assistant Professor in the School of Computer Science and Engineering at VIT University, Vellore,Tamilnadu,India.He received his B.Tech degree in Computer Science and Engineering from JNTU Hyderabad in the year 2004 and also received his MTech. degree in Computer Science and Engineering from JNTU Anantapur, He is having more than 10 years of teaching experience and his area of interest include Computer Networks,Network Security,Wireless Networks, Adhoc Networks,Distributed Computing.