AN ANALYSIS OF CACHE COHERENCE PROTOCOLS FOR MULTILEVEL CACHE ARCHITECTURE

By

DO-YOUNG CHUNG

Bachelor of Science

Ohio University

Athens, Ohio

1992

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1998

AN ANALYSIS OF CACHE COHERENCE PROTOCOLS

FOR MULTILEVEL CACHE ARCHITECTURE

Thesis Approved:

[illegible]

Thesis Adviser

J Chandler

[illegible]

Wayne B. Powell

Dean of the Graduate College

ACKNOWLEDGMENTS

I sincerely thank my graduate adviser Dr. K. M. George for the guidance, help and time he has given me for the completion of my thesis work. His perseverance and hard work inspired me to venture into the advanced aspects of this work. I would like to express my sincere thanks to Dr. J. P. Chandler for his direction and leadership. Without the encouragement and help he has given me, the completion of this work would have been impossible. I also sincerely thank Dr. Mayfield for serving on my committee.

My respectful thanks goes to my parents Mr. Yeon-Ok Chung and Mrs. Soon-Ja Kim for all the love and support they have given me throughout my life. Especially, I would like to express thank my wife Yang-Eui Kang. Without her support and encouragement, the completion of this work would have been impossible. And, my love goes to my beautiful children, my son Min-Jae and my daughter Kyung-Ran.

I would also like to express my gratitude to all those people who have contributed by giving many valuable insights.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

For decades, computer designers have been looking for speedup of computers. The most important factor to achieve this is to share the information. The desire for more computing power introduces shared-memory multiprocessors. However, the shared-memory multiprocessors suffer longer latencies in accessing shared memory. CPUs are getting faster and main memories are getting larger, but slower relative to the faster CPUs. As shown in Figure 1.1, the cache memory is introduced as a solution to this problem. These attached private caches to processors help reduce the average latencies. Such caches temporarily holds the in-use contents of main memory. The usefulness of cache memory depends on the property of *locality*. The sequence of memory addresses generated by a program typically exhibits the properties of *temporal* and *spatial locality* [LIL93, FRA84]. The temporal locality means that data to be referenced in the near future is likely to have been in use recently. It is exhibited by program loops in which instructions and data are reused. Spatial locality means nearby locations will be referenced in the near future. It results from some common characteristics of programs, such as sequences of instructions and related data items that are stored. Because of that, a system retrieves the information from memory and store it in a cache as a *block* (or a line) of consecutive words. A block is the minimum unit of information that can be either present or not present in a cache [HEN90]. Private caches exploit these memory-referencing properties to reduce the average time required to access the large main memory. Processors can access the data in related caches in much less time than it would take if it were in main memory. Thus processor performance is increased, since less time is spent waiting for instructions and for data to be fetched.

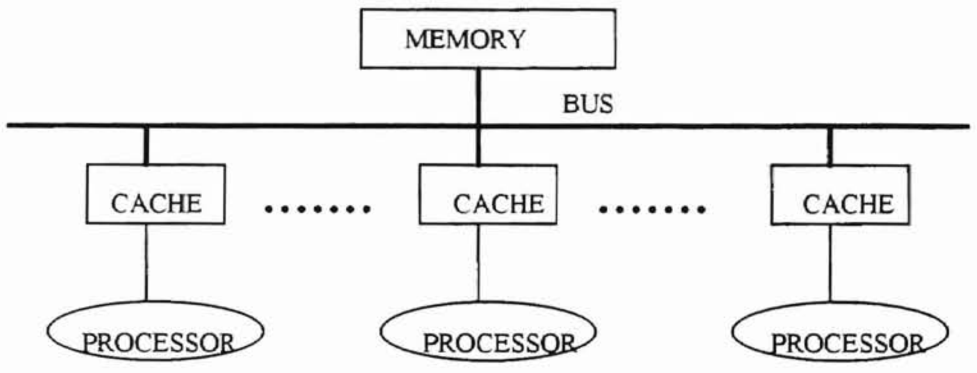


Figure 1.1 A shared-memory multiprocessor system with private caches

However, single-level caches can not satisfy the adequate speed and size. The growing disparity between fast processors and relatively slow memories leads to introduce cache memory. Two contradictory demands are required from the cache : to be fast and to be large enough. This is not possible using only single-level caches, so multiple-level cache hierarchies emerge as appropriate solutions. Multilevel cache architecture seems to be the unavoidable solution to the problem[TOM94]. It makes the cache faster to keep pace with the speed of CPUs and the cache larger to overcome the widening gap between the CPU and main memory. By adding another level of cache between the original cache and memory, the first-level cache can be small enough to match the clock cycle time of the CPU while the larger second-level cache can be enough to capture many accesses that would go to main memory. Thus, this architecture reduces the memory latency by smaller but faster lower level caches and reduces the traffic on the interconnection network by slower but much larger upper level caches. The problem of cache speed is solved on the first level and hit ratio on the second level. A large second level cache not only reduces the memory traffic but also shields the first-level cache from unnecessary coherence interference, which is achieved by following the *inclusion property*. Inclusion implies that upper level cache is the superset of all caches in the hierarchy below [BAE88]. This filters coherence actions toward lower levels and reduces the number of actions to the really necessary ones, lowering the cache interference. This inclusion property must be ensured to maintain cache coherence in multilevel caches. In [TOM94-2], multilevel caches are mainly classified into three types as shown in Figure 1.2.

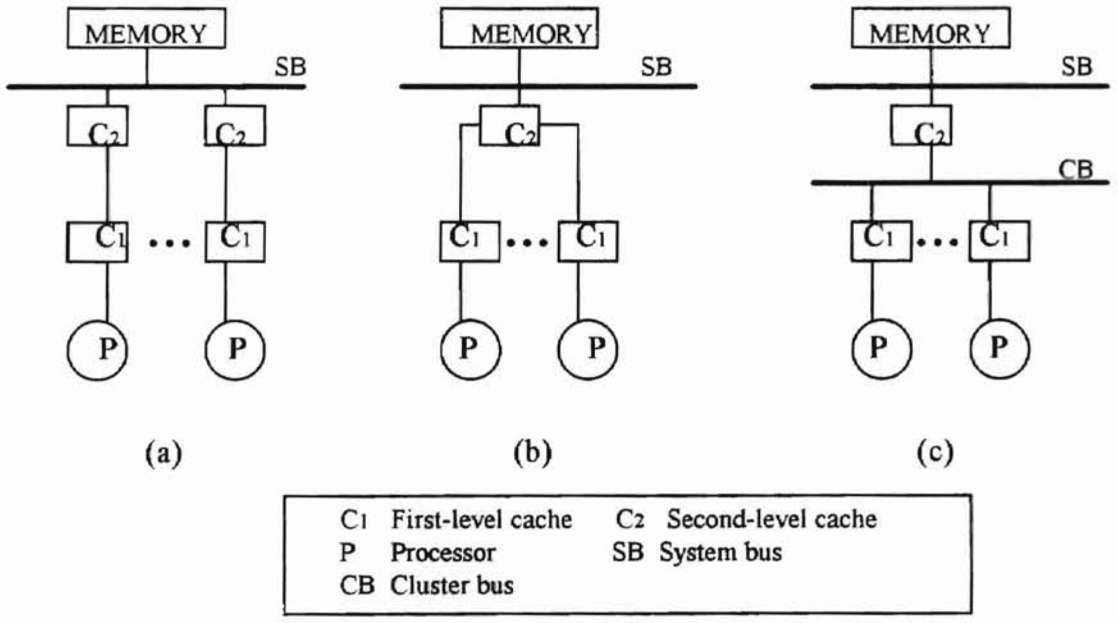


Figure 1.2 Multilevel cache organizations: (a) private, (b) multiport shared, and (c) bus-based shared

The first organization extends a single level cache to one in which every processor has its private hierarchy of caches. On-chip read-only caches belong to this paradigm. In the second organization, the upper level cache is multiported between the lower level caches. It consists of a second-level cache C_2 shared directly by a limited number (at most 4) of first level caches C_1 as shown in figure 1.2 (b). This architecture can be extended to one where several C_2 caches are used. Directory cache coherence approach between the C_1 caches is used for this architecture. The third organization consists of a large second-level cache C_2 being shared by up to two dozen first-level caches C_1. Bus-based protocols are used to maintain cache coherence. An extension of this architecture leads to a system with clusters of second-level cache, first-level caches and associated processors with the clusters being connected by a common bus.

CHAPTER 2

LITERATURE REVIEW

2.1 Cache coherence problem

Data can be found in memory or in the caches. When caches are used in multiprocessors, multiple copies of the same data block can exist in different caches at the same time. If processors are allowed to update their own copies independently, an inconsistent view of the memory is possible, leading to program malfunction. The different cached copies may have different values at the same time. This is generally referred to as the *cache coherence problem* [HEN90]. Performance of a multiprocessor program depends on the performance of the system when sharing data. The protocols to maintain coherency for multiple processors are called *cache coherence protocols*. These protocols ensure that whenever a processor reads a memory location, it receives the correct value. There are two different policies when a write operation of data in cache is to be performed. If the corresponding memory block is also updated on that occasion, the *write-through* policy is applied. In the *write-back* policy, the update is postponed until the cache block is evicted to make room for another block (*replacement*). Data coherence problems do not exist in multiprocessors if only a single copy of data is allowed. Cache coherence problems exist in multiprocessors with private caches (Figure 1.1) and are caused by three factors: sharing of writable data, process migration, and I/O activity [DUB88]. Examples of data inconsistencies are illustrated below in the following.

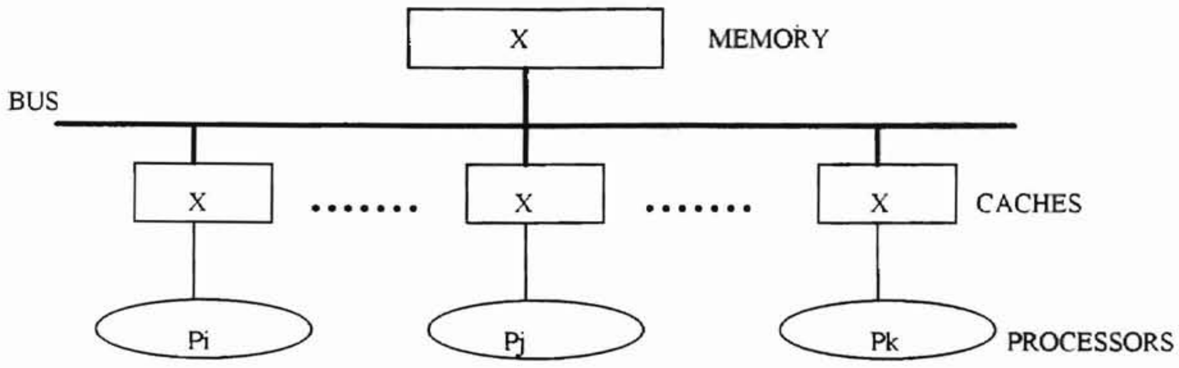


Figure 2.1 Cache configuration after a Load on X by processors.

If the caches do not contain copies of X initially, a Load (primitive operation) of X by the three processors results in consistent copies of X as shown in figure 2.1. Next, if Pi issues a Store on X, then the copies in the caches are inconsistent. But, consistency is maintained between cache and memory if write-through policy is used as shown in the following Figure 2.2.

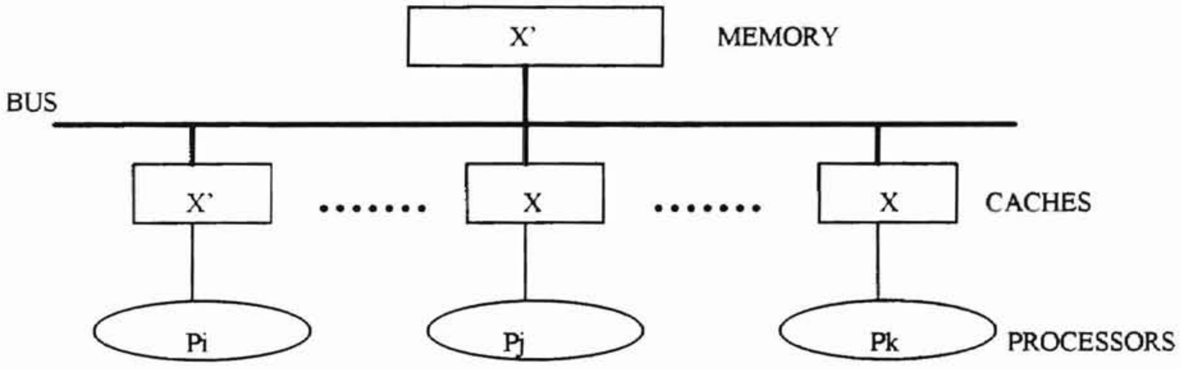


Figure 2.2 Cache configuration after a Store on X by processor i (write-through)

However, cache-memory consistency like the above is not maintained at the time of "store" by a processor if write-back policy is used. Figure 2.3 shows the state after a store operation. Eventually memory is updated when the modified data in the cache are replaced.

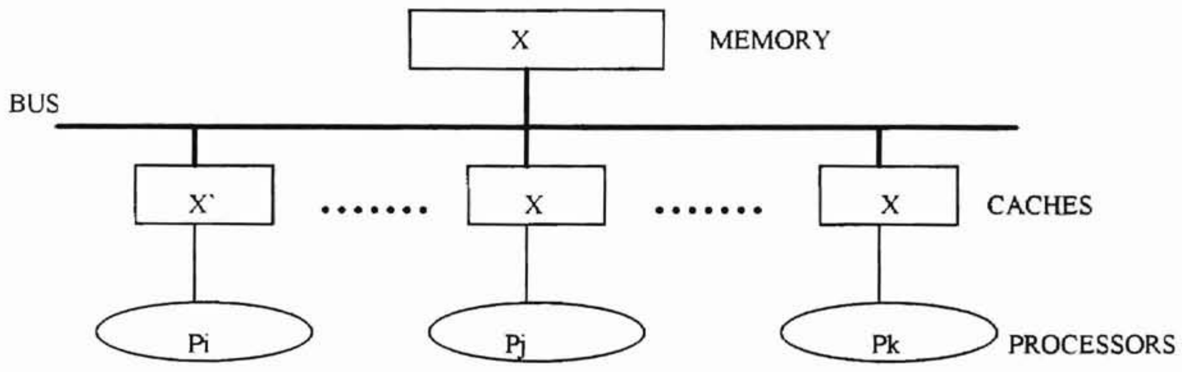


Figure 2.3 Cache configuration after a Store on X by processor i (write-back)

2.2 Cache coherence solutions

There are two large groups of cache coherence protocols: software-based and hardware-based. Usually all proposed protocols fall into one of these traditional but useful classification. Sometimes the combined solutions of software and hardware are used for cache coherence protocols.

2.2.1 Hardware solution

Hardware-based solutions are usually called cache coherence protocols. This approach has several advantages over software-based ones. First, it gives better performance because of the dynamic recognition of inconsistent conditions for shared data at runtime. Secondly programmers and compilers are free from responsibilities for coherence maintenance and there are no restrictions on any layer of software. This is a result of being totally transparent to software. Hardware based solutions are classified into snooping protocols and directory protocols.

2.2.1.1 Snooping protocols

In snooping protocols, every cache that has a copy of the data from a block of physical memory also has a copy of the information about it. These caches are usually on a shared-memory bus, and all cache controllers monitor or snoop on the bus to determine whether or not they have a copy of the shared block. And, these local controllers recognize the actions and conditions for coherence violations. As a processor snoops on the other processors' memory references, it detects when a block that it has cached has been changed by another processor. It then *invalidates* its cached copy so that its next reference to the block will force a cache miss, and thus a current value will be obtained from memory, or from another cache. Alternately, it can directly *update* its cached copy with the new value available on the bus. Lilja's study shows advantages and disadvantages of these two approaches [LIL93]. Invalidation strategy marks all cached copies as invalid within the cache to force the processor to miss the next time it references that block. This approach reduces the bus-traffic compared to the update strategy, but it increases the miss rate if the block is reused. With an update approach, the new value of the shared location is distributed to all processors with a copy of the block whenever it is written by any processor. The advantage of this approach is that it prevents an additional miss if the cache block is reused by a processor with a cached copy after it has been written by another processor. A disadvantage is the additional bus traffic produced by the potentially large number of update messages. Snooping protocols became popular with multiprocessors that use a shared bus as a global interconnection, since the shared bus provides very inexpensive and speedy broadcasts.

2.2.1.2 Directory protocols

Directory-based schemes keep the information about data blocks in just one location. There is logically a single directory that keeps the state of every block in main memory. Thus the directory stores the global, systemwide state information relevant for coherence maintenance. It is called centralized in memory. Agarwal et al. introduces one useful classification of directory schemes for broadcast and non-broadcast schemes [AGA89]. The directory maintains information about which processors have a copy of the same block cached at the same time. Before a processor writes to a block, it must request exclusive access to the block from the directory. Before the directory grants this exclusive access, it sends a message to all processors with a cached copy of the block forcing each processor to invalidate its copy. After receiving acknowledgments from all of these processors, the directory grants exclusive access to the writing processor. When a processor tries to read a block that is exclusive in a different processor, it will send a miss service request to the directory. The directory then will send a message to the processor with the exclusive copy telling it to write the new value back to memory. After receiving this new value, the directory sends a copy of the block to the requesting processor.

Directory protocols are primarily suitable for multiprocessors with general interconnection networks.

2.2.2 Software solution

According to Tomasevic, software-based solutions generally rely on the actions of the programmer, compiler, or operating system [TOM94-1]. These coherence schemes try to predict which memory addresses may become stale by analyzing the program's referencing behavior when it is compiled. The static coherence schemes determine at compile-time which particular cache blocks may be stale, and when they may be stale, and then invalidate stale cache entries before they are accessed. These schemes are software based since they rely on a compiler, they also need some hardware support to maintain the current state information about the memory locations. Therefore, it is not correct to refer to these mechanisms as software only coherence mechanisms. The advantage of software-based approaches is that they are less expensive than hardware-based ones. The disadvantage is that the compiler analysis cannot predict the flow of program execution accurately.

CHAPTER 3

OVERVIEW OF PROTOCOLS

In this chapter, three cache coherence protocols with invalidation policy in multiple cache/ bus-based architectures are introduced and studied for performance analysis. Reducing access time and bus traffic is the most important factor to improve system performance. These protocols are designed to reduce the amount of traffic by using higher level cache as filter, which will ignore the remote bus request if it dose not have a line with its cache for the request. Also, they reduce average latency by high hit ratio in lower level cache so that a request for a line dose not need to go up to memory. Consequently, this will reduce miss penalty. Wilson introduced the concept of hierarchies of shared buses and proposed a simple cache coherence scheme based on the write-once protocol [WIL87]. This protocol will be called **extended write-once protocol** from now on. Yang and Bhuyan proposed similar hierarchical bus coherence protocol relying on inclusion and block ownership [YAN92]. This protocol will be called **mastership-based protocol** in this paper. Anderson and Baer proposed a scheme that is based on clusters of processors connected via a tree hierarchy of buses. Determining proper number of processors that share a common bus in a cluster is essential to avoid bus saturation [AND92]. This will be called **cluster-based protocol**.

3.1 The architecture

The architecture is represented as:

Cache(S,L,N)   where:

S is the supercluster number,

L is the level number, and

N is the number of the cache on that level.

For example, Cache(N,0,Z) in the bottom right most in figure 3.1 specifies that this cache belongs to group N in right hand side dotted rectangle and is located at the bottom -level hierarchy and attached level of tree to Zth processor under Nth group in the system.

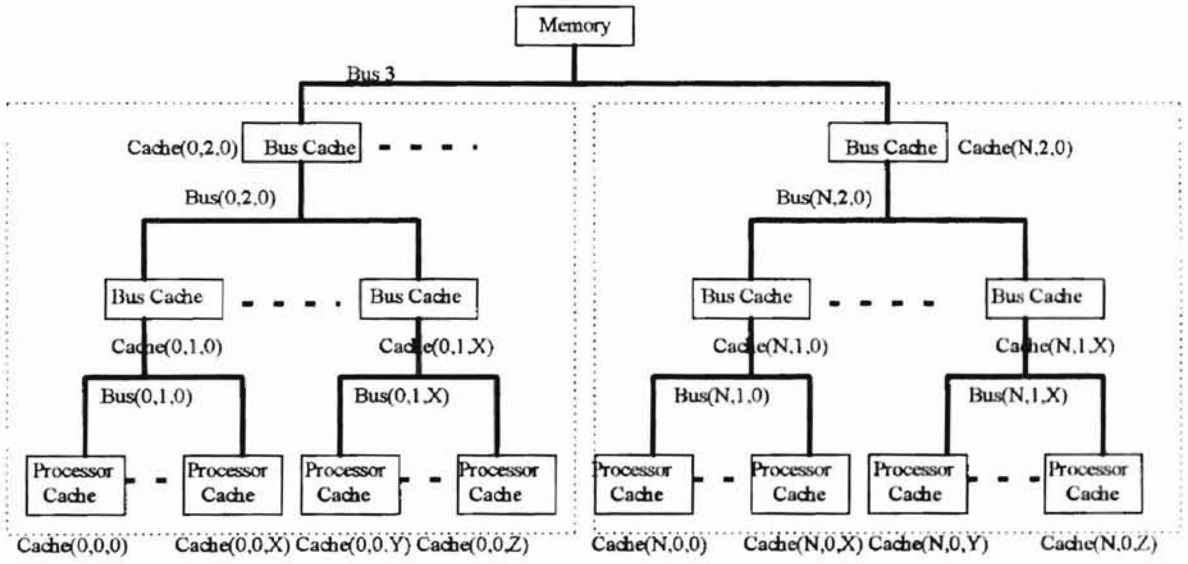


Figure 3.1 Machine Architecture

3.1.1 Extended write-once architecture

This architecture is viewed as a hierarchy of caches/buses where a cache contains a copy of all blocks cached underneath it [WIL87]. This requires large higher level cache modules. Memory modules connect to the topmost bus.

3.1.2 Mastership-based architecture

A hierarchical architecture that consists of a hierarchy of caches and buses except the dotted rectangular line in Figure 3.1. Every cache at any level of the system has an intelligent snooping controller that continuously monitors the higher level bus and receives requests from the lower level bus, or from local processor. The entire system forms a tree of buses with the main memory being at the root that is connected to the top level bus and processors being at leaves connected to the bottom level buses through private caches [YAN92]. This hierarchical approach is quite suitable for building large scale multiprocessors.

3.1.3 Cluster-based architecture

The architecture consists of a hierarchy of buses in tree-like structure [AND92]. At the bottom of hierarchy (level 0) are processors along with their caches. In the cluster-based architecture, the processors are grouped into clusters of some number of processors per cluster as shown in the above figure 3.1, typically two or eight.

3.2 Protocol descriptions

To maintain consistency among copies, Wilson proposed an extended write-once protocol. Consistency among copies stored at the same level is maintained in the same way as for traditional snoopy cache protocols. However, an invalidation must propagate vertically to invalidate copies in all caches[WIL87]. The most important elements in cluster based protocol is to keep the inclusion property, that is, each cache at a given level contains a superset of the contents of the cache below it in the hierarchy [AND92]. At the top of the hierarchy is memory that has lines with no state. The mastership based coherence protocol is similar to extended write-once protocol and it allows multiple copies of a shared block [YAN92]. The existence of multiple copies of a shared block in the large hierarchical network can result in a significant amount of traffic to enforce cache coherence and thus degrade the system performance. This cache coherence scheme for hierarchical networks is designed to effectively handle multiple shared read while reducing the network traffic. The protocols above are designed to minimize response time and bus traffic, especially at the higher levels of the hierarchy. To do this, requests are satisfied as close as possible to the requesting processor. The protocol will attempt to satisfy a request using caches attached to the same bus as the requester; the request is forwarded to higher levels only when necessary. In some cases, a request may need to travel up the hierarchy, then back down to another branch of the tree; the reply retraces the same path back to the request originator. Lines are written back only to the next higher level of the hierarchy.

3.2.1 Protocol states

All the caches that are directly connected to the processors are called level 0 caches or processor level caches. In the figure 3.1, the second parameter L indicates levels in (S,L,N). If L is greater than one, then the caches in that level are called non-processor caches.

3.2.1.1 Extended write-once protocol

Processor / non-processor level cache states

INVALID	There is no data in the block.
VALID	There is data in the block which has been read from backing store and has not been modified.
RESERVED	The data in the block has been locally modified exactly once since it was brought into the cache and the change has been transmitted to backing store.
DIRTY	The data in the block has been locally modified more than once since it was brought into the cache and the latest change has not been transmitted to backing store.

3.2.1.2 Mastership-based protocol

Processor level

VALID -MASTER	The block is owned exclusively and can be read and written locally; for a remote write request, the block is invalidated and given to the requesting cache; for a remote read, the block is supplied and mastership moves up; write back is necessary in case of replacement.

VALID-SLAVE	Valid block, multiple copies may exist; a read can be performed locally; write is not allowed before acquiring mastership; write back is not necessary when purged.
INVALID	The block is not present or does not contain useful data.

Non-processor cache states

VALID -MASTER	The cache having this block has mastership; multiple copies may exist in the subtree; for a remote read, block is supplied and mastership moved up; invalidated and supplied for a remote write; If purged, should be written back and other copies invalidated; a local write request moves mastership down to the requesting cache.
VALID-SLAVE	Block is valid; a V-M copy exists in an ancestor cache or the main memory; a local write request causes invalidation of other copies and change of state to D-O; a remote write request causes invalidation.
DESCENDANT-OWNED	A descendant cache has the mastership for the block; D-O serves as a pointer to set up read paths to the latest copy for remote requests. The block need not be physically present and can be purged.
INVALID	Block does not contain valid data or is not present.

3.2.1.3 Cluster-based protocol

Processor level

INVALID	The line is not present in the cache
VALID EXCLUSIVE	The line is present in no other processor cache except this one, and the data has not been written by the processor.
READ SHARED	The data in the block is up-to-date, and is possibly shared by other processor caches. The data may be dirty with respect to main memory, but it is not the responsibility of this cache to write it back.
DIRTY	The line is present in no other processor cache except this one, and this cache has modified the data with respect to memory or other non-processor cache. This cache is responsible for writing the line back.

Non-processor cache states

INVALID	The line is not present in this cache or any other in the sub-hierarchy rooted at this cache.
READ SHARED	The line is present in the cache in a clean state. The line may exist in a clean state in caches below this one in the hierarchy. The line may exist outside the hierarchy in a clean state.
VALID EXCLUSIVE	The line is present in the cache, but is not guaranteed to be up-to-date. The line may exist below this cache in a dirty or clean state.

DIRTY SHARED	The line is present in the cache and is dirty with respect to memory. This cache is responsible for writing the value back to the next higher level in the hierarchy. In addition, the line may exist lower in the hierarchy in READ SHARED state only.
DIRTY OWNED	The line is present in the cache and is dirty with respect to memory. The line does not exist lower in the hierarchy.

The transitional states

TRANS INVALID	The line is in transition to the INVALID state.
TRANS SHARED	The line is in transition to the READ SHARED state.
TRANS EXCLUSIVE	The line is in transition to DIRTY (for processor caches) or in transition to the VALID EXCLUSIVE state (for bus caches).

This transition states inhibit additional requests for a line when the state is in flux. If a cache that has a line in transitional state detects a request for that line on the bus, it signals that the request is canceled. The request is then sent back to the originator, which will try the request again after waiting a period of time.

3.2.2 Bus request types

3.2.2.1 Extended write-once protocol

CLUSTER BUS READ	A request to load a cache line.

GLOBAL BUS READ	A request to load a cache line that is traveling down one part of the hierarchy after traveling up another part of the hierarchy.
CLUSTER BUS WRITE	A request to load a cache line so that the originating processor may write the line.
GLOBAL BUS WRITE	Similar to a CLUSTER BUS WRITE request except that it is traveling down the hierarchy.
CLUSTER BUS INVAL	A request to invalidate other copies of a given line.
GLOBAL BUS INVAL	A request to invalidate other copies of a given line. The request is traveling down the hierarchy
PURGE	A request to invalidate the line in question in all caches at this level of the hierarchy and below. No response is needed by the originating cache.

3.2.2.2 Mastership-based protocol

LOCAL READ	A request to load a cache line.
REMOTE READ	A request to load a cache line that is traveling down one part of the hierarchy after traveling up another part of the hierarchy.
LOCAL WRITE	A request to load a cache line so that the originating processor may write the line.
REMOTE WRITE	Similar to a CLUSTER BUS WRITE request except that it is traveling down the hierarchy.

READ-MEMORY	A request to load a cache line from memory.

3.2.2.3 Cluster-based protocol

READ	A request to load a cache line.
READ DOWN	A request to load a cache line that is traveling down one part of the hierarchy after traveling up another part of the hierarchy.
READ EXCLUSIVE	A request to load a cache line so that the originating processor may write the line.
READ EXCLUSIVE DOWN	Similar to a READ EXCLUSIVE request except that it is traveling down the hierarchy.
WRITE-BACK PURGE	The line is being written back to the next higher level in the hierarchy due to replacement in the lower level cache.
INVALIDATE	A request to invalidate other copies of a given line.
INVALIDATE DOWN	A request to invalidate other copies of a given line. The request is traveling down the hierarchy, unlike an INVALIDATE request which travels up the hierarchy.
SWAP INVALIDATE	Similar to an INVALIDATE request except that (part of) the line is atomically swapped with a register value rather than written in the originating cache. SWAP INVALIDATE requests are propagated as INVALIDATE requests above the bottom layer in the hierarchy.

SWAP READ EXCLUSIVE	Similar to a READ EXCLUSIVE request except that the line is swapped rather than written in the originating cache.
PURGE	A request to invalidate the line in question in all caches at this level of the hierarchy and below. No response is needed by the originating cache.
PURGE (and) REPLY	Like a PURGE request, except that the originating cache waits for some sort of reply from a cache below it.

3.2.3 Processor / bus cache actions

3.2.3.1 Extended write-once protocol

This scheme extends the algorithm proposed by Goodman [GOO84]. Write-once utilize initial write-through mode for recently acquired copies to invalidate other caches in the local modification of data. Data write following first modification will be handled in local so that no write to memory is neccessary. The operation of the protocol can be specified by making clear the actions taken on processor reads and writes.

Read hits : can always be performed locally in the cache and do not result in state transitions.

Read miss : If no dirty copy exists, then memory has a consistent copy and supplies a copy to the cache. This copy will be in the valid state.

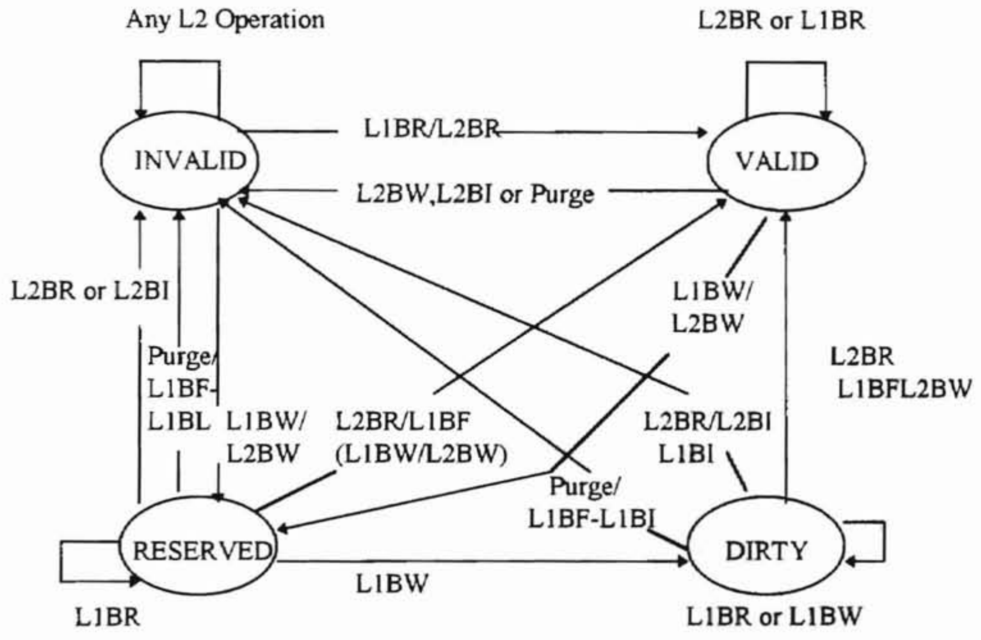
If dirty copy exists, then the corresponding cache inhibits memory and sends a copy to the requesting cache. Both copies will change to valid and the memory is updated.

Write hit : If the copy is in the dirty or reserved states, then the write can be carried out locally and the new state is dirty. If the state is valid, then a Write-Inv consistency command is broad-cast to all caches, invalidating their copies. The memory copy is updated and the resulting state is Reserved.

Write miss : The copy either comes from a cache with a dirty copy, which then updates memory, or from memory. This is accomplished by sending a Read-Inv consistency command, which invalidates all cached copies. The copy is updated locally and the resulting state is dirty.

Replacement : If the copy is dirty, then it has to be written back to main memory. Otherwise, no actions are taken.

The state transitions are shown in figure 3.2.



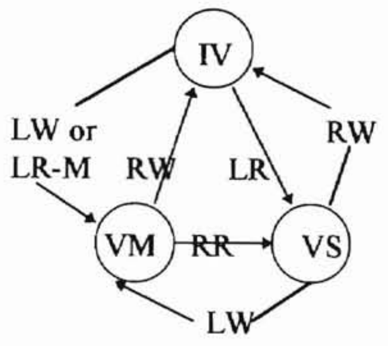
L1BR:	Custer Bus Read	L2BR:	Global Bus Read
L1BW:	Cluster Bus Write	L2BW:	Global Bus Write
L1BI:	Cluster Bus Inval	L2BI:	Global Bus Inval
L1BF:	Cluster Bus Flush	Purge:	Replacement of cache memory

Figure 3.2 State Diagram for a Second Level Cache Using the Extended Goodman's Algorithm

3.2.3.2 Mastership-based protocol

Processor level

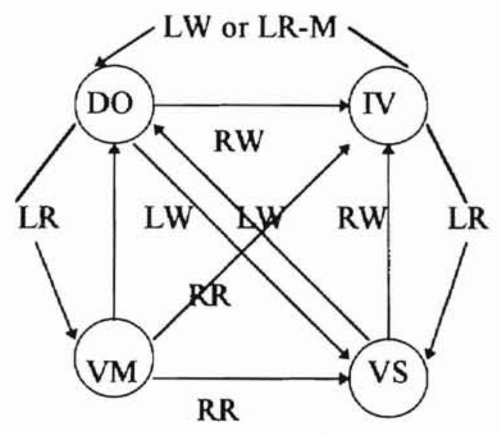
A read hit is said to occur if a cache at level L, second parameter in (S,L,N) of figure 3.1, receives a read request from level L-1, and finds the block in V-M or V-S state. The block is moved to the requesting processor and the state is unchanged. All the caches along the path including the local cache of the requesting processor change their states to V-S. A read miss occurs if a cache at level L finds on a read request from level L-1 that the block is in I-V. In this case the request is propagated on the level L bus. There are four different ways in which the request gets data. 1) The cache at level L+1 has the block in V-M or V-S. 2) One of the peer caches has the block indicated as D-O. 3) One of the peer caches has the block in V-M state. 4) The cache at level L+1 also has the block in I-V. If a processor issues a write request, and the level 0 cache has the block in V-M state then the write is performed locally since it is the only copy in the system. However, if the block is in V-S state, the mastership has to be obtained and an invalidation signal has to be broadcast before the write operation is performed. Also all the ancestor caches up to the previous master cache have to be informed about the write operation to change their block state from V-S to D-O. Upon a write miss, a requested block is loaded in the same way as a read miss except that the block is loaded with V-M state. All the caches that have a copy of the block invalidate their copies and the ancestor caches change the block state to D-O state. State transitions for processor and non-processor level are shown in figure 3.3 and 3.4, respectively.



LR-Processor read
RR-Remote read
LR-M-Processor read satisfied in memory
RW-Remote write
LW-Processor write

Figure 3.3 State transition diagram of a block in a processor cache.

Non-processor cache states

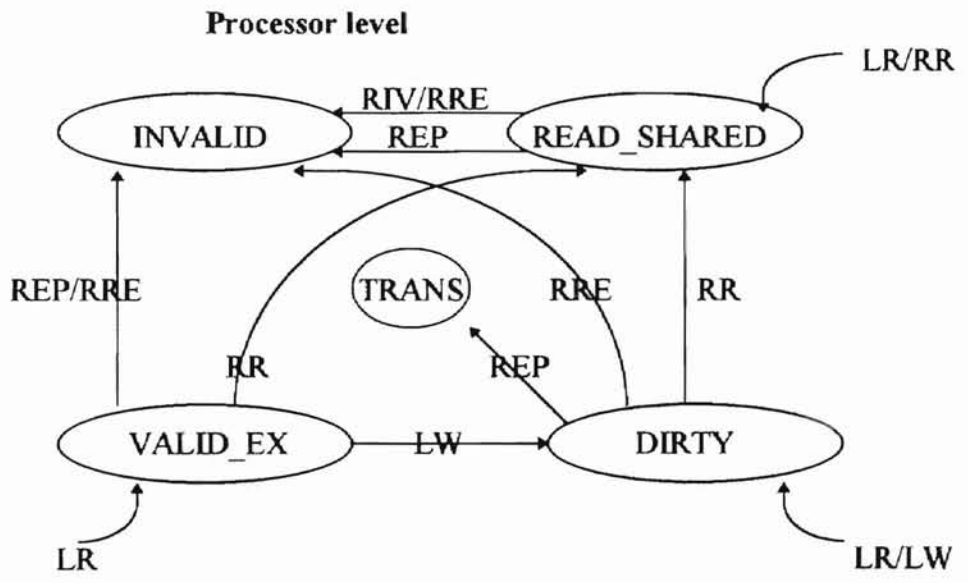


LR-Local read
RR-Remote read
LW-Local write
RW-Remote write
LR-Read from a local cache
that does not have the block
LR-M-Local read satisfied in memory

Figure 3.4 State transition diagram of a block in a higher level cache

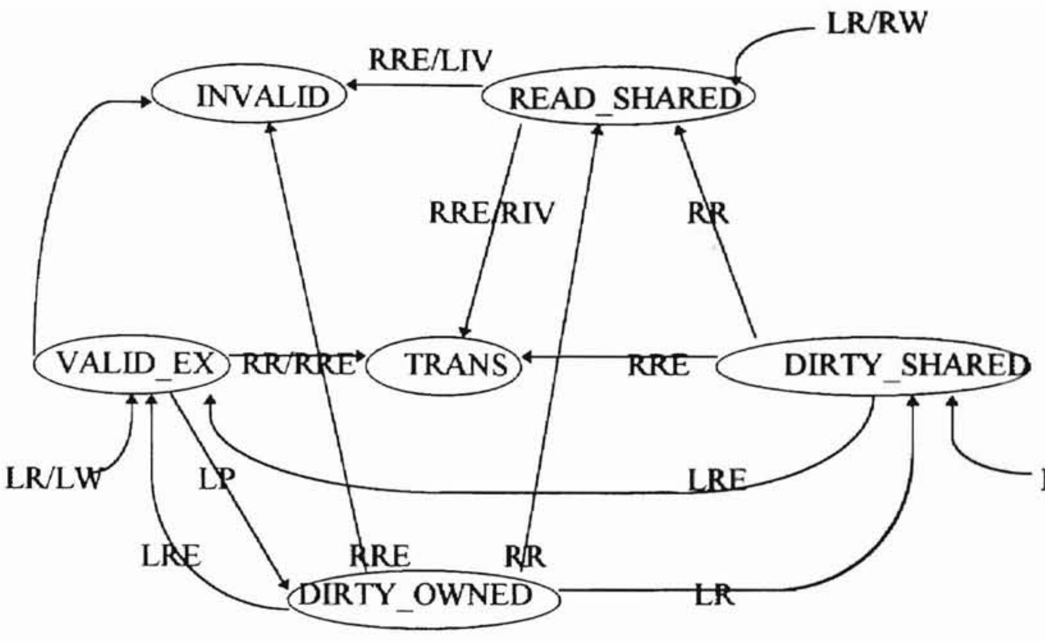
3.2.3.3 Cluster-based protocol

All memory accesses hit if the line is DIRTY or VALID EXCLUSIVE. If the line is VALID EXCLUSIVE and the request is not a LOAD, the line status is changed to DIRTY. A LOAD request to READ SHARED line hits. All other cases are considered processor cache misses. When a request misses in a processor cache, the processor waits until the request is satisfied. The processor's cache allocates a line for the data and enqueues the appropriate bus request. If the request succeeds, the processor is restarted; otherwise the processor will wait some fixed amount of time after the failed request has been returned to it, then retry the request. A hit in a bus cache means that the request will not need to proceed further up or down the hierarchy. A bus cache in a READ SHARED state can satisfy a READ request, while a bus cache in either the DIRTY OWNED or DIRTY SHARED state can satisfy *READ* or *READ EXCLUSIVE* requests.



LR:	Processor read	RR:	Remote read
LW:	Processor write	RRE:	Remote read exclusive
REP:	Line replace	RIV:	Remote invalidate

Figure 3.5 State transition diagram of a block in a processor cache. Non-processor cache states



LR:	Local read	RR:	Remote read
LRE:	Local read exclusive	RRE:	Remote read exclusive
LIV:	Local invalidate	RIV:	Remote invalidate
LP:	Local write back purge		

Figure 3.6 State transition diagram of a block in a higher level cache

3.2.4 Block replacement

3.2.4.1 Extended write-once protocol

If a processor cache needs to replace a line not in the DIRTY state, it simply overwrites the current line with no further action needed. If the line is in the DIRTY state, the processor schedules a write-back PURGE request. When the request is serviced, the line in the second level cache changes its status to DIRTY state. Replacing a line in non-processor cache must maintain inclusion property. If the state of the line in the replacing cache is DIRTY, the cache simply writes the line to the next higher level in the hierarchy.

3.2.4.2 Mastership-based protocol

If the block selected for replacement is in V-S state in a cache, then all the copies of the block in the cache's descendant must be invalidated before it is purged. This invalidation is necessary because the removal of a V-S block in a cache breaks the path from its descendants to the master cache. The invalidation counter of the parent cache needs to be decremented. However, write back is not necessary. Any request for the same block from the cache's descendants will be satisfied in a higher level ancestor cache or the main memory. In case the block selected for replacement is in V-M state, the cache sends invalidation messages to its descendant caches so that copies of the block are invalidated. The V-S states in the intermediate caches and the D-O states in the ancestors are changed to I-V states. Then the block is written back into the main memory. This write back operation is necessary since the block may have been modified during the time when the cache has the mastership. It is also possible that the block is written into an ancestor cache when purged instead of the main memory, provied there is enough space in the ancestor cache. In this case the D-O states of the caches between the ancestor cache and the main memory remain unchanged. The ancestor cache where the block is wrtten assumes the new mastership of the block.

3.2.4.3 Cluster-based protocol

If a processor cache needs to replace a line not in the DIRTY state, it simply overwrites the current line with no further action needed. If the line is in the DIRTY state, the processor schedules a WRITE-BACK PURGE request. When the request is serviced, the line in the second level cache changes its status from VALID EXCLUSIVE to DIRTY OWNED, since no other cache in the cluster has the line. Replacing a line in non-processor cache must maintain inclusion property.If the state of the line in the replacing cache is DIRTY OWNED, the cache simply writes the line to the next higher level in the hierarchy. If the state is READ SHARED, a PURGE request is enqueued on the bus below this cache. The PURGE request propagates down the hierarchy

4. ILLUSTRATIONS OF PROTOCOL IN ACTIONS

4.1 Extended write-once protocol

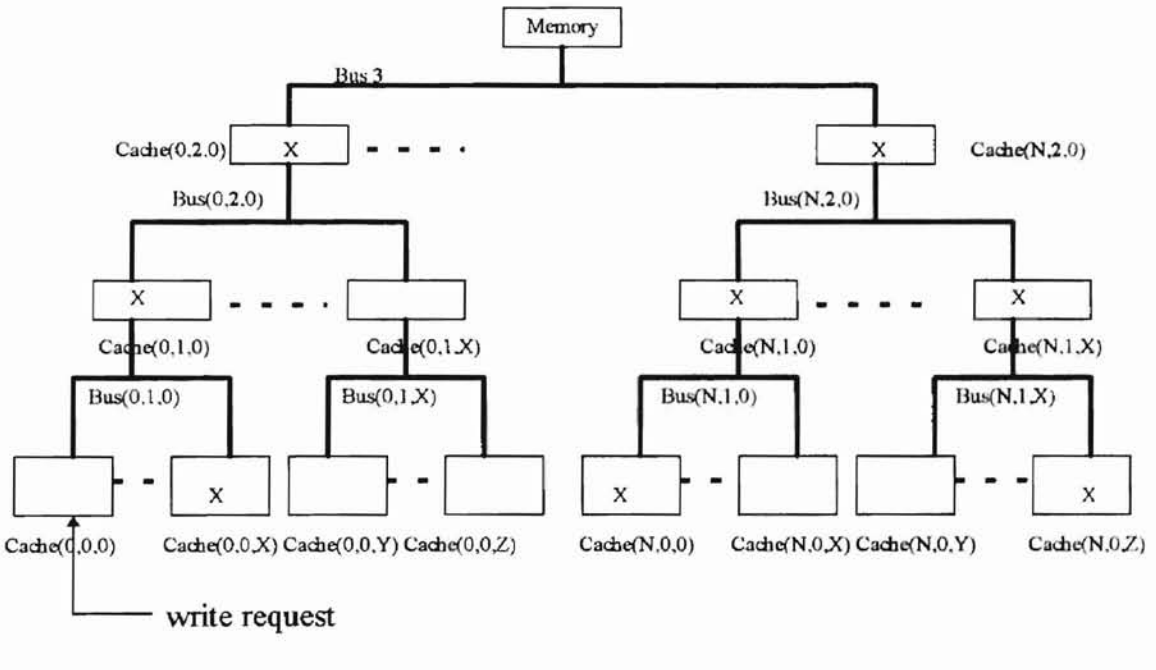


Figure 4.1 : Initial configuration before write request

Processor attached to Cache(0,0,0) issues an initial write in figure 4.1 and does not find a block on its private cache. Then it puts a write request on Bus(0,1,0).

Cache(0,0,X) detects the block requested by Cache(0,0,0) and supplies the block X. Then it invalidate its own copy.

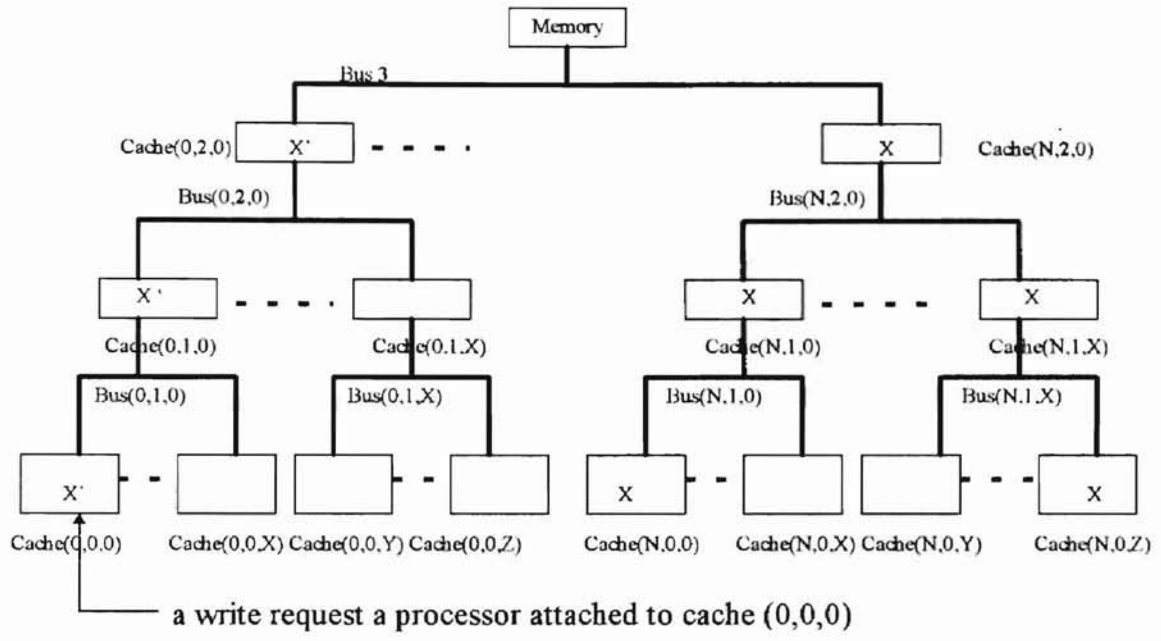


Figure 4.2 Operation of Initial Write-through in Write-Once Algorithm

Processor attached to Cache(0,0,0) receives the block from Cache(0,0,X) and modifies its own copy Cache(0,0,0) in figure 4.2 above. Then it propagate write-access up to the highest bus. On the way to the Bus 3, the new modified value is rewritten on each cache that has an associated block.

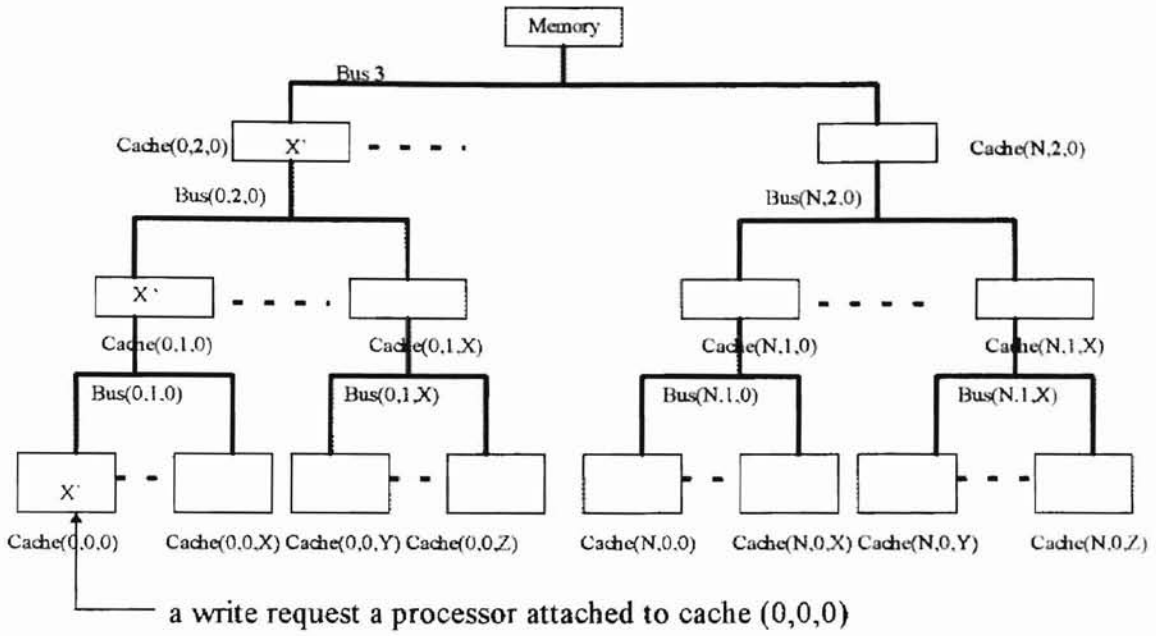


Figure 4.3 Final configuration of initial write-through

When Cache(N,2,0) detects the write operation, it invalidates its own copy and send a invalidation signal on Bus(N,2,0). Cache(N,1,0) and Cache(N,1,X) do the same operation as Cache(N,2,0). Finally all operations are done at this point as shown in figure 4.3.

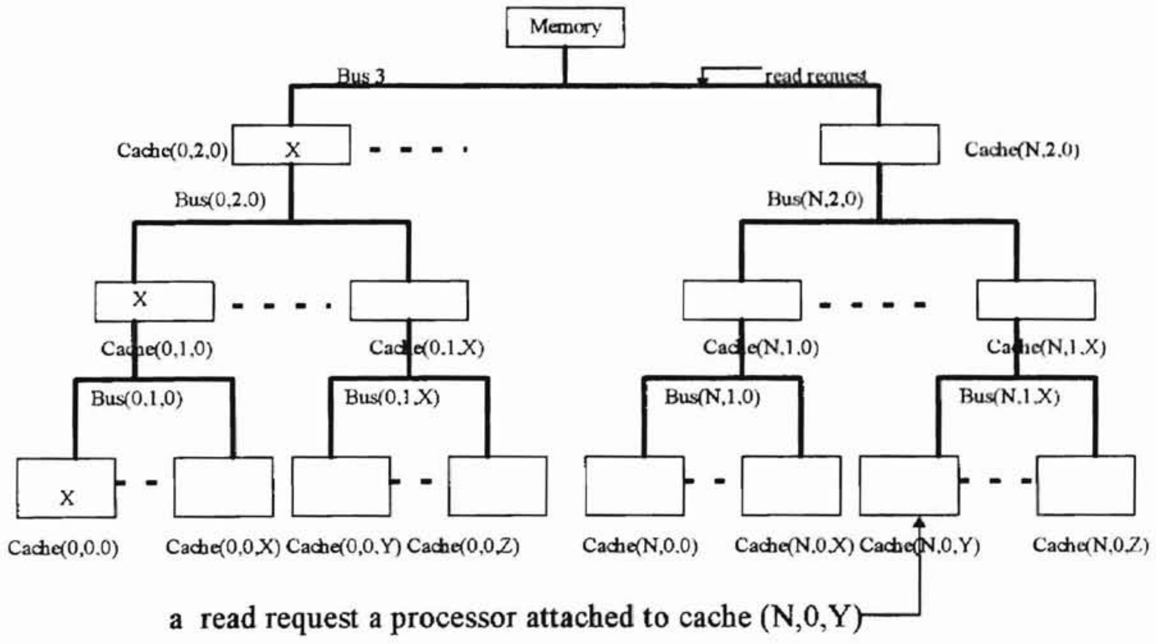


Figure 4.4 Initial configuration

A processor attached to Cache(N,0,Y) issues a read request on a block X. It does miss on Cache(N,0,Y) and puts a read request on Bus(N,1,X). No caches on level 0 has the block requested by Cache(N,0,Y) and propagates upper level. Then Cache (N,1,X) does the same process as lower level caches did.

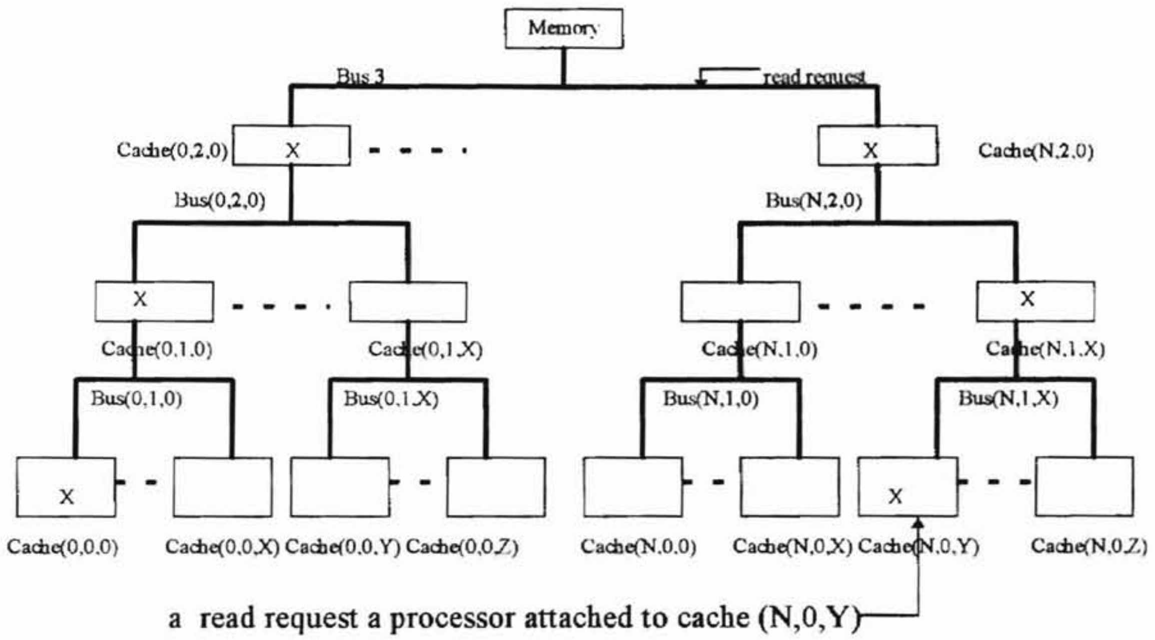


Figure 4.5 Final configuration

When Cache(0,2,0) detects read request on Bus 3, it supplies its own copy. And, process is repeated in the reverse direction. Finally, the processor that issued the read request gets the requested data as shown figure 4.5.

4.2 Mastership-based protocol

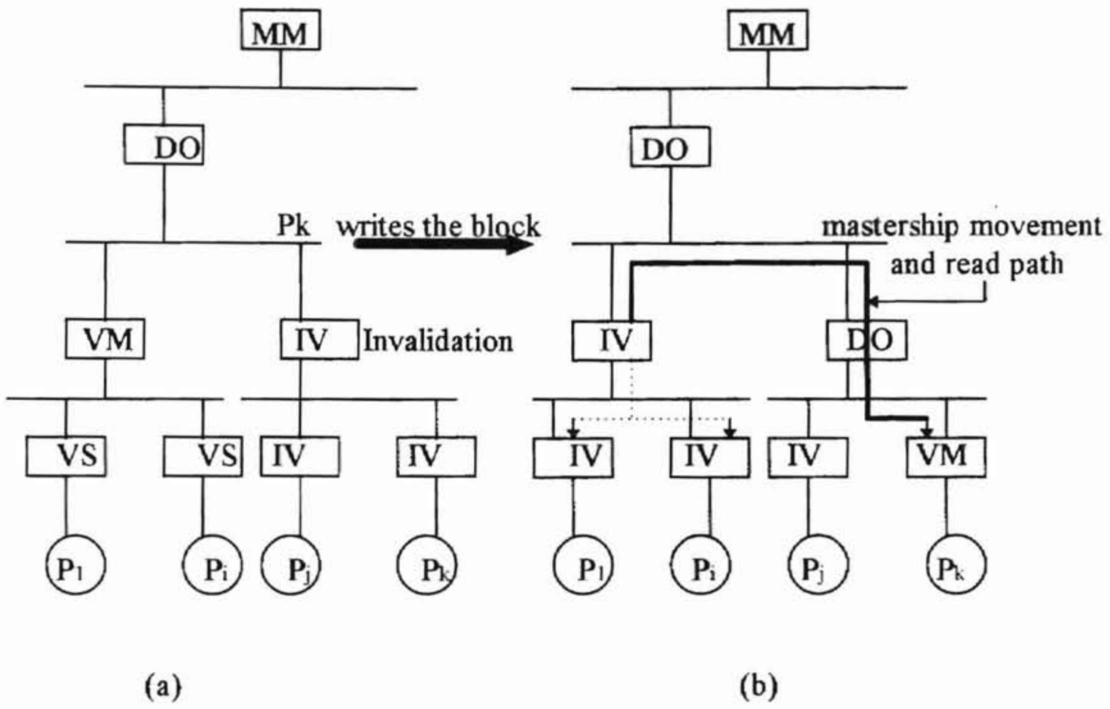


Figure 4.6 Mastership movement for a write operation
(a) States of the block in the caches before write operation
(b) States of the block in the caches after write operation

P_k issues a write request to the block that is not present in its local cache. The request is routed to the parent cache of P_l and P_i which invalidates copies of the block in its descendant caches. After the invalidation, the parent cache of P_l and P_i gives in the mastership to P_k's local cache. The state changes as result of the write operation are shown in Fig. 4.6(b).

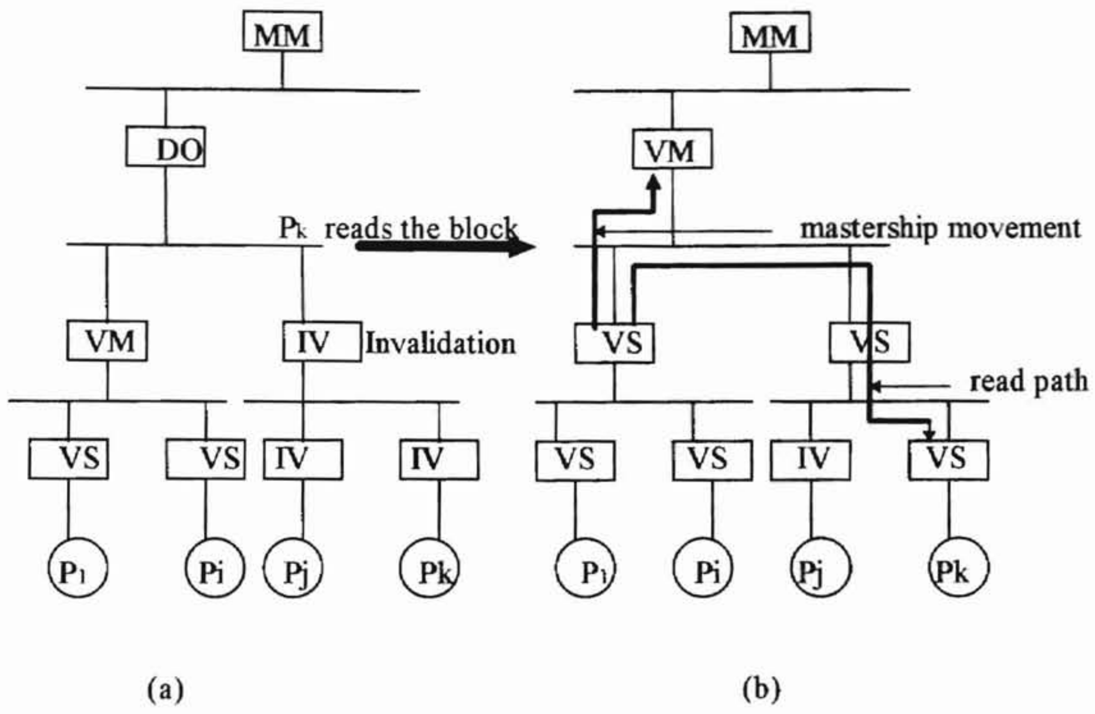


Figure 4.7 Mastership movement for a read operation
(a) States of the block in the caches before read operation
(b) States of the block in the caches after read operation

P_k issues a read request and has not found a block in its local cache. The request propagates up and father of P1 and Pi has a block with VALID-MASTER state. So the father of P1 and Pi grabs the request, supplies the block and changes its state to VALID-SLAVE. Ancestor cache at level 2 obtains the block and becomes new master. Along the path to requesting cache, all states are changed to VALID-SLAVE as shown above Fig. 4.7(b).

4.3 Cluster-based protocol

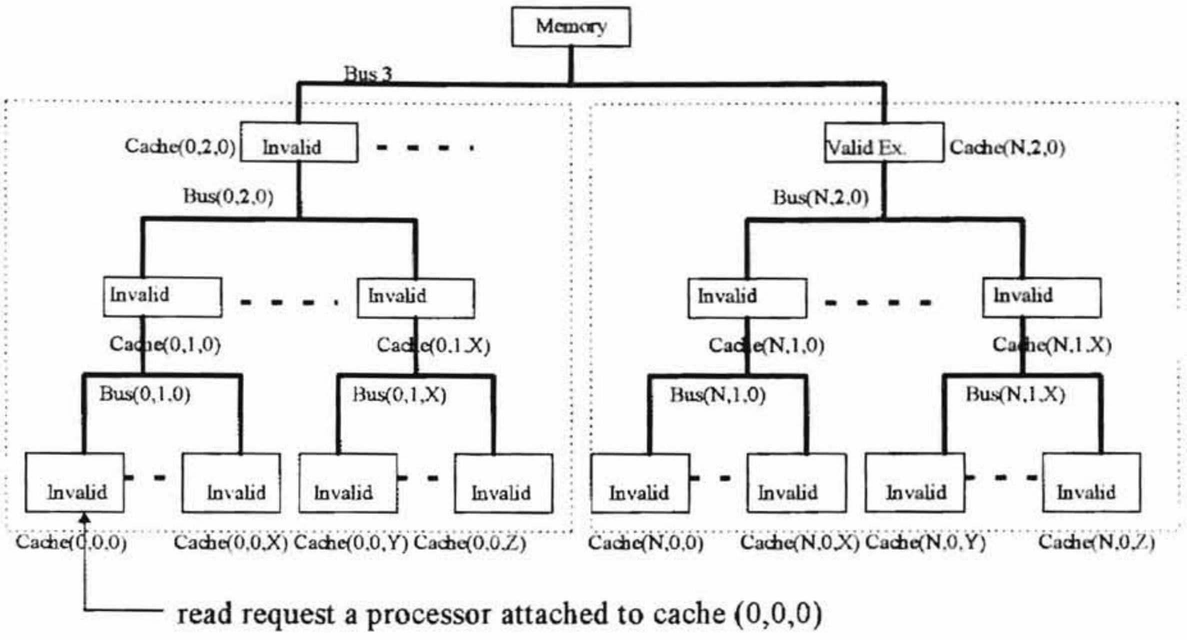


Figure 4.8 Initial configuration before read/write request

Cache(0,0,0) issues to read a line in figure 4.8. Since it is not in its cache, it pre-allocates and pins a line. The cache then enqueues a read request on bus(0,1,0). Because none of the other caches in the cluster nor the level 1 cache(0,1,0) has the line, processor cache(0,0,0) change its state to TRANS SHARED. Meanwhile, cache(0,1,0) pre-allocates and pins a block and enqueues a read request on bus(0,2,0). When the read request gets transmitted, the actions are similar to what took place on the level 1 bus. Current system state is shown below in Fig. 4.9.

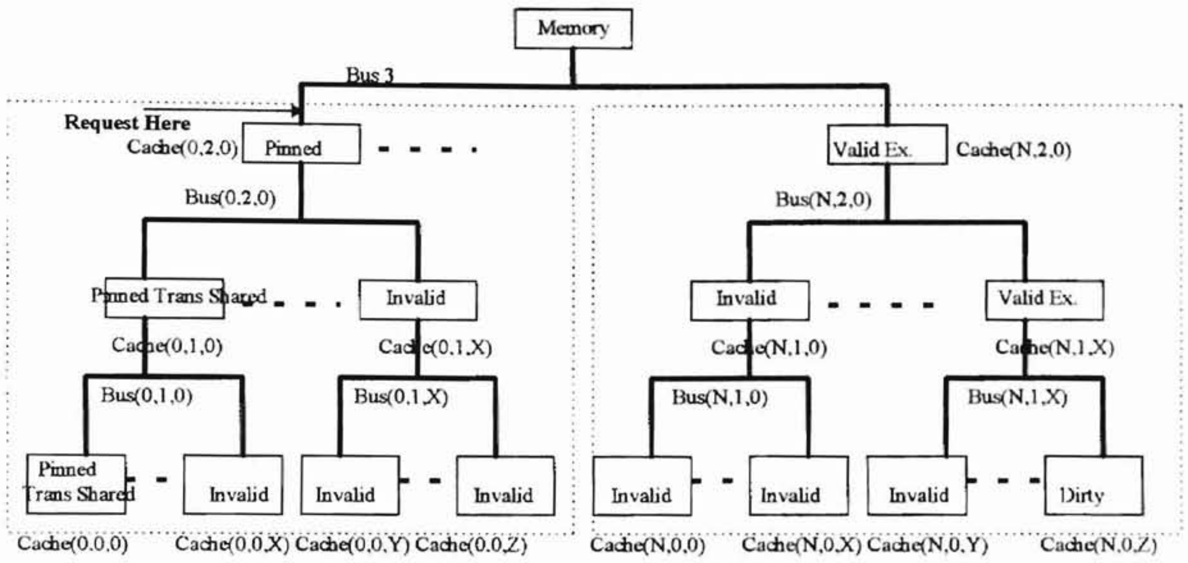


Figure 4.9 System with read request at top of hierarchy

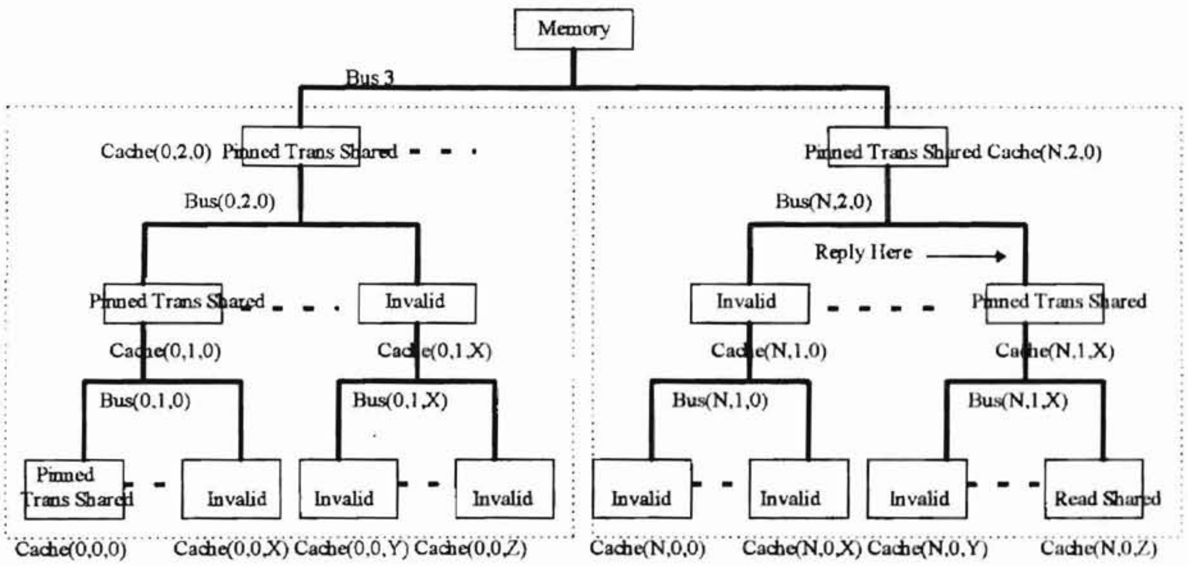


Figure 4.10 System after read reply has been generated

Once the READ request goes out over the level 3 bus, cache(N,2,0) answers with an acknowledgment. It issues READ DOWN request on bus(N,2,0), and changes its state to TRANS SHARED. The READ DOWN request proceeds down the hierarchy until it reaches bus(N,1,X).When the READ DOWN request gains access to bus(N,1,X), cache(N,0,Z) responds and transfers the line back to the cache(N,1,X), along a flag indicating that the line must be written back. It sets its state to READ SHARED. The system is now in the state depicted below in fig. 4.11.

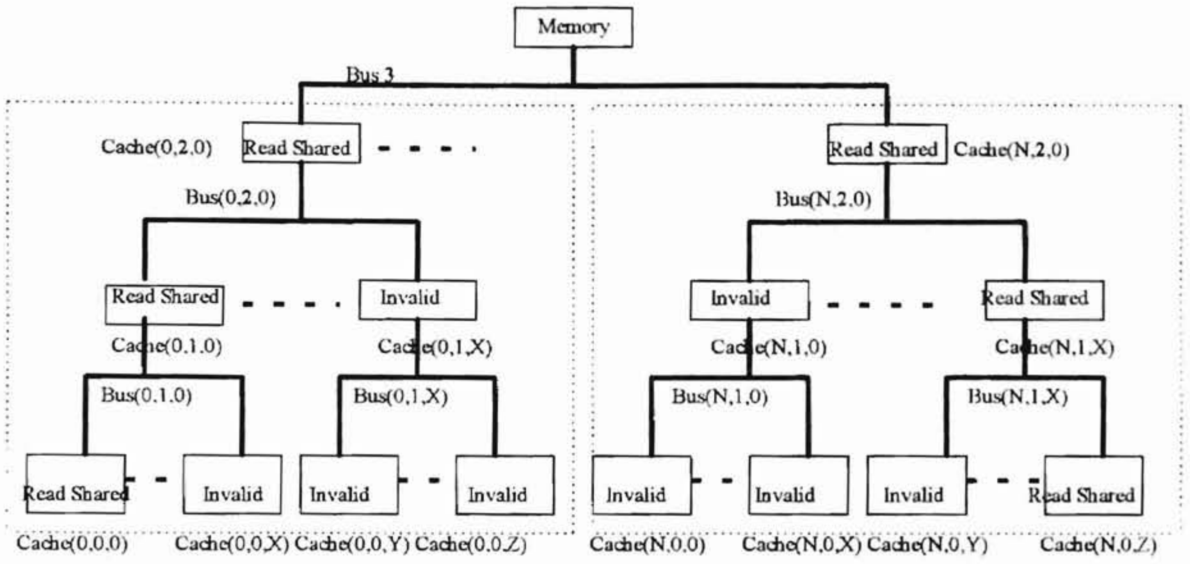


Figure 4.11 System after read has been completed

Now, assume fig. 4.8 initial state for write operation. Cache(0,0,0) issues a write request. The processor cache(0,0,0) sends a READ EXCLUSIVE request. This request proceeds up the hierarchy to the level 3 bus, then back down to bus(N,1,X). Along the way to level 3 bus, all the caches change their states to TRANS EXCLUSIVEAlong the way to bus(N,1,X), all the caches change their states to TRANS INVALID. The reply heads up the hierarchy while invalidating all affected caches, on the way back down to the originator of the request while changing states cache(0,2,0) and (0,1,0) to VALID EXCLUSIVE. Finally, this write operation completes by changing the state of cache(0,0,0) to DIRTY. The system is depicted below in fig. 4.12.

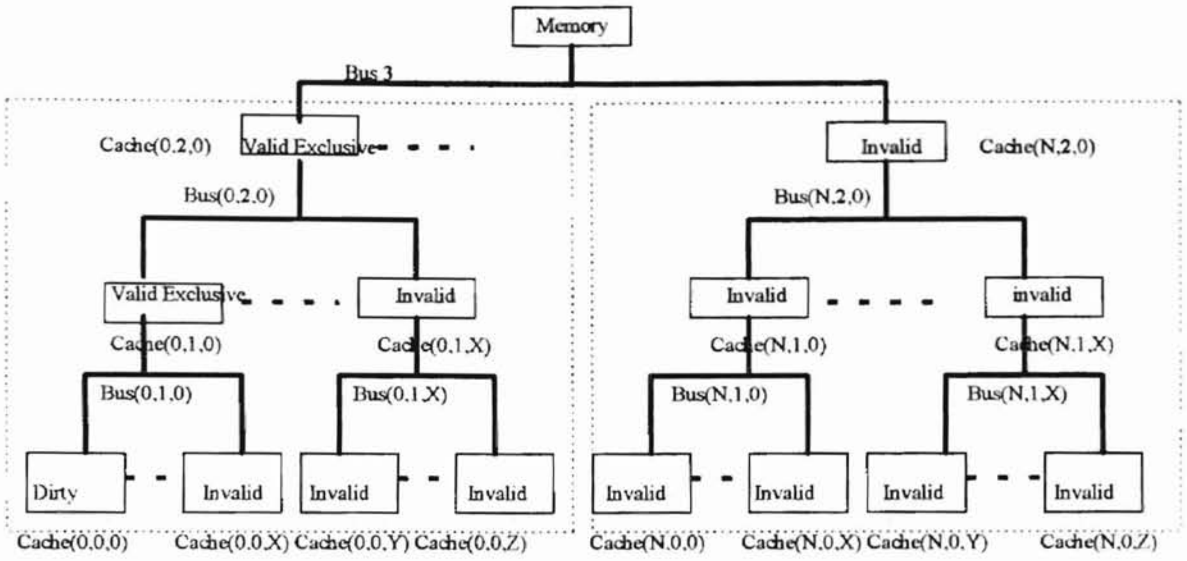


Figure 4.12 System after write has been completed

CHAPTER 5

PERFORMANCE ANALYSIS AND RESULTS

In this chapter, we analyze the behavior of each protocols under various conditions, and discuss which protocols behave better or worse using different metrics. Our simulation consists of random reference generators for each processor in parallel. Architectures for performance analysis of above protocols are multilevel cache/bus hierarchies, snoopy cache coherence protocols, and hardware based solutions. According to Frank, there are four parameters for maximum performance in cache memories [FRA84]. Three parameters must be minimized: the time needed to access data in cache, the delay in getting data when it is not in the cache (cache miss), and the overhead of updating main memory and maintaining cache coherence. One parameter, the hit rate, must be maximized.

5.1 Parameters for performance evaluation

There are typical parameters evaluating performance of multilevel caches as follows : cache size, block size, set associativity, block replacement algorithm, and write policy. Since hit time and miss penalty effect the memory access time, they should be counted too. The terms used in this performance study are defined in glossary at the end of this thesis.

Cache size

It is known that reducing the size of the cache increase the miss ratio and the bus traffic. In general, the two correlate well with respect to this parameter. Therefore, it is important to determine the size of caches (in the first and higher level) in order to achieve desired system performance in a certain architecture.

Block size

The block size plays an important role in cache coherency. If a small block size is used in first and second level caches, resulting miss rate would be higher than the miss rate for a large block size. However, a large block size contribute false sharing. It is indicated that two different shared variables are located in the same cache block, causing the block to be exchanged between processors even though the processors are accessing different variables. On the other hand, larger block size leads to lower miss rate and lesser transfer time.

Set associative

It is a matter of where to place a block in a cache. If a block is placed in a restricted set of places in the cache, the cache is said to be set associative. A set is a group of two or more blocks in the cache. A block is first mapped onto a set, and then the block may be placed anywhere within the set. If there are 2 blocks in a set, the cache placement is called 2-way set associative as used in this study.

Block placement algorithm

On a miss, a block in a cache must be replaced in favor of a newly referenced block. There are several ways to replace a block : FIFO, random, LFU and so on. In this study, LRU is chosen in case of a miss in both first and second level caches. This strategy selects that block for replacement that has not been used for the longest time. In this simulation study, LRU has been implemented with a list structure. Each time a block is referenced, that block is placed at the head of the list. Therefore, in the situation of a miss, the block in the tail is chosen to be replaced.

Write policy

While the choice between write-through and write-back has no impact on the read ratio, but it has a major impact on bus traffic. It has been known that using write-back instead of write-through for a hypothetical processor would reduce the bus traffic by more than half and if the processor ran to completion, the bus traffic would be decreased by a factor of 8 [NOR92].

5.2 Analysis

This section reports the results of executing our simulation model on the data stream. Our simulation model permits many parameters to be varied. We have chosen to fix a number of the parameters and to concentrate primarily on those which we consider most relevant, the sizes of the L1 and L2 caches, and block size. A number of assumptions are made. First, the line sizes of the L1 and L2 caches are identical (4 words). Second, it is assumed to be 2-way set associative. Third, write-back policy is applied. For the convenience of drawing graph, extended write-once, master-based and cluster-based protocols are represented in short p1, p2 and p3 respectively.

Hit Ratios

Figure 5.1 shows hit ratios for a single-level cache of various sizes with p1. These hit ratios are consistent with those reported in a study of CLA83. However, p2 and p3 performs slightly better. Because p2 and p3 are designed for multi-level cache architectures. It is worthwhile to study the effect on hit ratio by using different parameters. The hit ratio for a single level cache will be the same as the hit ratio for the L1 cache in our two-level model. The figure also shows the effect of using different block sizes.

The total variation in cache hit ratio when increasing cache size from 8K bytes to 512K bytes is less than 10 percent. For a memory access time of 15 cycles, this difference in hit ratio makes a 50 percent better performance achieved. The increased line sizes in small cache sizes produce about 10 percent performance increase for an 8K byte cache.

Figure 5.2 shows the hit ratio of the L2 cache in our two-level cache system. As the table shows, varying the size of the L2 cache has a significant effect on hit ratio, and shows L2 hit ratios ranging from 20 to 90 percent. Block size also affect the hit ratio of the L2 cache. For example, using an 8K L1 cache, increasing the L2 block size from 2 to 4 lines increases its hit ratio by 7 to 20 percent, depending on its size.

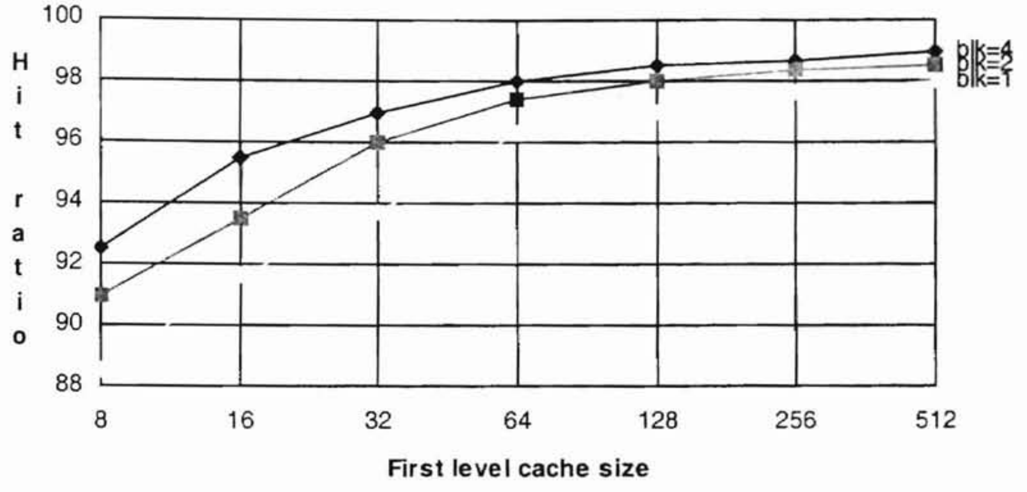


Figure 5.1 Hit ratio of first level cache

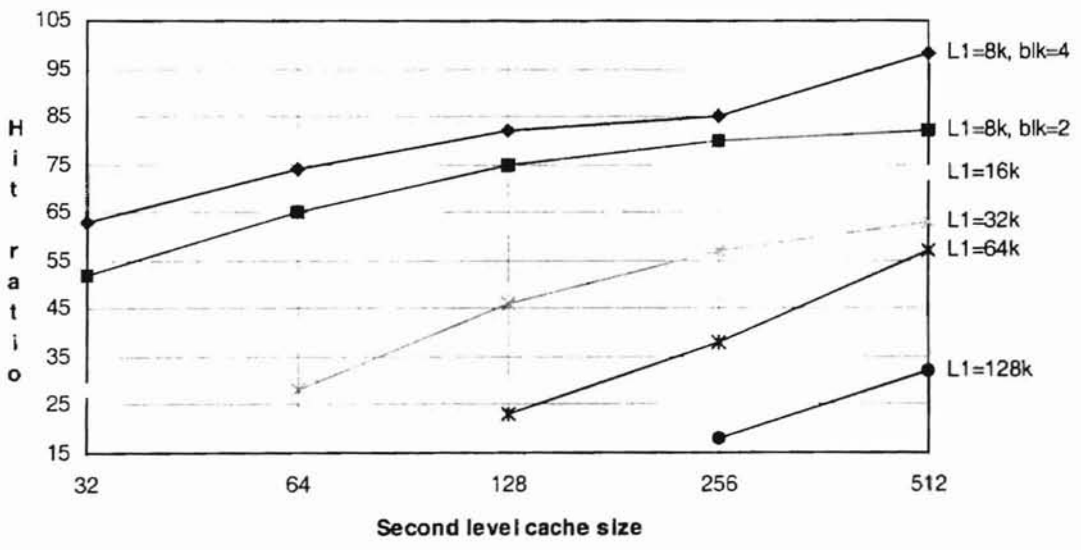


Figure 5.2 L2 cache hit ratios

Now it is time to investigate how each protocols perform on a miss. To do this, we need to fix some parameters. We assume L1 cache be 32K and L2 be 256 with a line size 16 bytes in 2way set associative. In a cache architecture, a line replacement and write-back to upper level cache on a miss is costly. Following Figure 5.3 shows clearly that the write-once protocol produces the highest number of replacement, and results in many write-back to upper level caches. Mastership-based protocol evidently produces the least number of replacements. Because of the dynamic movement of a line ownership, it is possible to avoid replacement and have a second chance to stay in a cache.

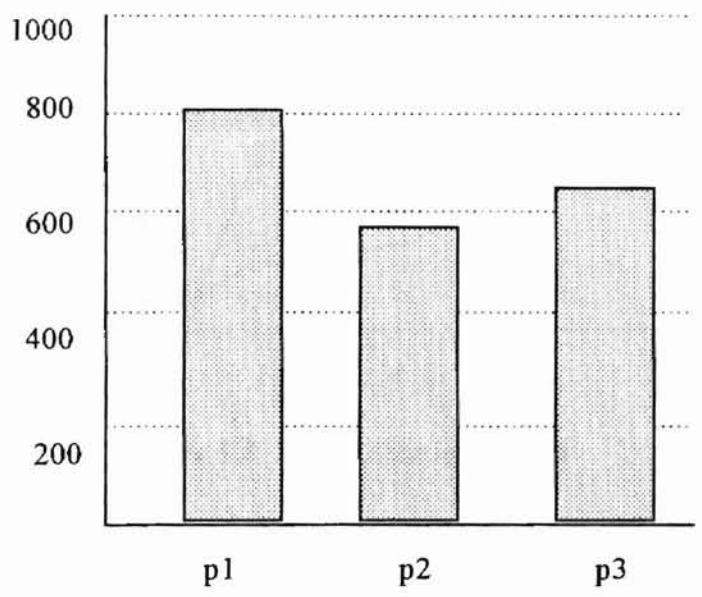


Figure 5.3 Number of replacements and write backs

Figure 5.4 shows the bus utilization ratio when the number of processors is increased. Extended write-once protocol produces the highest bus utilization ratio. Since extended write-once protocol needs a write back of the line to the shared memory when a dirty line is transferred between caches, its bus utilization ratio is large. Ownership-based protocol shows the lowest bus utilization ratio. In ownership-based protocol, there are not many bus requests since only ownership of a block moves up and down to the requested cache.

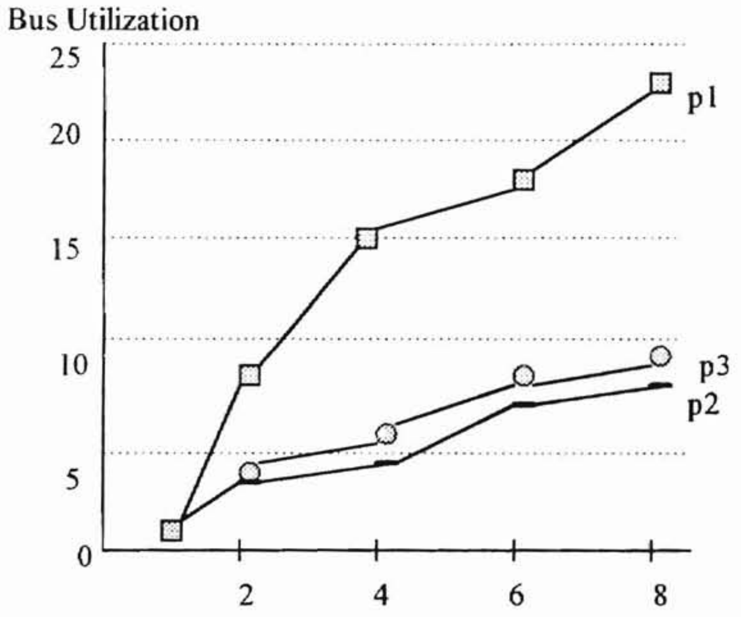


Figure 5.4 Bus utilization ration

Figure 5.5 shows average access time for each protocol. Goodman's algorithm indicates clearly the slowest one. Partly, a block that is in a dirty state being replaced would be referenced by other processors. That block is no longer in a cache, hence lead to read/write miss. Eventually the block must bring back to a requested processor from upper level caches. This contributes large waiting time for processors.

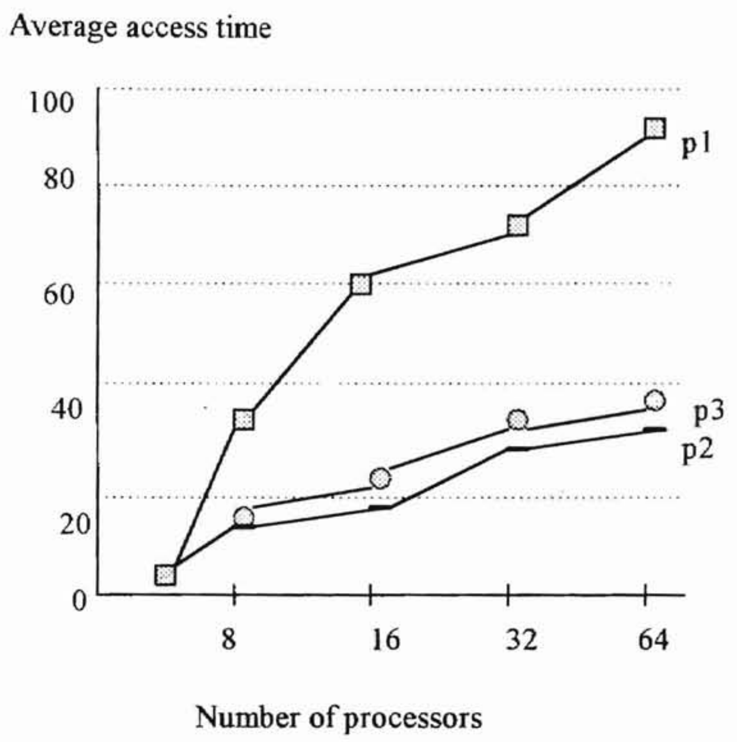


Figure 5.5 Average access time vs number of processors

CHAPER 6

CONCLUSIONS

6.1 Summary

In the design of a system where more than one processor shares a common memory, a major limiting factor on system performance is the number of processors that can effectively share memory. The limiting factor on the number of processors is the bus bandwidth and, in turn, bus and memory contention among the processors. As memory contention increases, the average memory access time increases and the performance of each processor decreases. There are several factors contributing to overall performance : block size, cache size (first and upper level caches), block replacement algorithm, and write policy. Best combination of these parameters lead to optimal system performance. From our study, each cache with large block size performs better than small block size. It is due to less transfer time required for a new request. In a multi-level cache, a large upper level cache size reduces further memory access time. Since most requests are satisfied locally, it is not necessary travel up to memory that delay memory access significantly. Goodman's algorithm results in heavy bus traffic with the case of block replacements. Since it has to do a write through operation up to the main memory. On the other hand, master-based and cluster-based protocols restricts bus operations on the bottom levels of the hierarchy so that less bus traffic results. This reduced bus traffic contributes to system performance significantly.

6.2 Future work

In this thesis, we are concerned with the performance of the three recognizable protocols in same architecture. Future work in protocol performance would find the optimal point of system performance by examining the combination of parameters. More work needs to be done simulating the whole architecture using comprehensive large and varied data.

BIBLIOGRAPHY

[AGA89] A.Agarwal et al. An evaluation of directory schemes for cache coherence. *Proc.16th ISCA*, 1989, pp. 280-289.

[AND92] C. Anderson and J.-L. Baer. A multi-level hierarchical cache coherence protocol for multiprocessors. Technical Report 92-10-04, University of Washington, 1992.

[BAE88] J.-L.Baer and W.-H.Wang. On the inclusion properties for multi-level cache hierarchies. *Proc. 15th ISCA*, 1988, pp. 73-80.

[CLA83] D. W. Clark. Cache performance in the VAX-11/780, ACM Transactions on Computer Systems, Vol.1, No.1, Feb. 1983, pp. 24-37.

[DUB88] M.Dubois and C.Scheurich. Synchronization. Coherence, and Event Ordering in multiprocessors. Computer, Vol.21, No.2, Feb. 1988.

[FRA84] S. J. Frank. Tightly coupled multiprocessor system speeds memory-access times. *Electronics*, Vol.57, Jan. 1984, pp.164-169.

[GOO83] J. Goodman. Using cache memory to reduce processor-memory traffic. *10th Annual Symposium on Computer Architecture*, 1983.

[HEN90] J.Hennessy and D.Patterson. *Computer Architecture: a Quantitative Approach*, San Mateo, Cal. : Morgan Kaufmann Publishers, 1990.

[LIL93] D. J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons. *ACM Computing Surveys*, Vol.25, No.3, Sep. 1993.

[NOR92] R. L. Norton and J. L. Abraham, Using write back cache to improve performance of multiuser multiprocessors, *Int. Conf. on Par. Proc.*, IEEE Cat. No. 82CH1794-7, 1982.

[TOM94-1] M.Tomasevic and V.Milutinovic. Hardware approaches to cache coherence in shared-memory multiprocessors, Part 1. *IEEE Micro*, Dec.1994, pp. 52-59.

[TOM94-2] M.Tomasevic and V.Milutinovic. Hardware approaches to cache coherence in shared-memory multiprocessors, Part 2. *IEEE Micro*, Dec.1994, pp. 61-66.

[WIL87] A.W.Wilson Jr. Hierarchical Cache/Bus Architecture for Shared Memory Mutiprocessors. *In Proc. 14th Symposium on Computer Architecture* 1987, pp. 244-252.

[YAN92] Q.Yang, G. Thangadurai, and L. Bhuyan. Design of an adaptive cache coherence protocol for a large scale multiprocessors. *IEEE Trans. Parallel and Distributed Systems*, Vol.3, No.3, May 1992, pp. 281-293.

GLOSSARY

hit - a memory access found in a level

miss - not found in a level

hit time - the time to access the upper level of the memory hierarchy, which includes the time to determine whether the access is a hit or miss

miss penalty - the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the requesting device (normally the CPU)

access time - the time to access the first word of a block on a miss

average memory access time= hit time + miss rate x miss penalty

transfer time - the additional time to transfer the remaining words in the block

VITA

Do-Young Chung

Candidate for the Degree of

Master of Science

Thesis: AN ANALYSIS OF CACHE COHERENCE PROTOCOLS FOR MULTILEVEL CACHE ARCHITECTURE

Major Field: Computer Science

Biographical Data:

Personal Data: Born in Seoul, Korea on December 23, 1960, the son of Yeon-Ok Chung and Soon-Ja Kim.

Education: Received Bachelor of Science in Computer Science from Ohio University, Athens, Ohio in 1992. Completed the requirements for the Master of Science degree in Computer Science at Oklahoma State University in December 1998.