A SIMULATION STUDY OF SNOOPY CACHE COHERENCE PROTOCOLS

By

IN-SUK CHUNG

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1993

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
May, 1996

OKLAHOMA STATE UNIVERSITY

A SIMULATION STUDY OF SNOOPY CACHE

COHERENCE PROTOCOLS

Thesis Approved:

[illegible]

Thesis Adviser

J. Chandler

[illegible]

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGEMENTS

I sincerely thank my graduate adviser Dr. K. M. George for the guidance, help and time he has given me toward the completion of my thesis work. His tenacity and hard work inspired me to venture into the advanced aspects of this work. I would like to express my sincere thanks to Dr. G. E. Hedrick 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. J. P. Chandler for serving on my committee. His suggestions have helped me to improve the quality of this work.

My respectful thanks goes to my parents Mr. Lee-June Chung and Mrs. Boon-Ok Kim for all the love and support they have given me throughout my life. And, I thank all other members of my family for the love, encouragement and confidence they have contributed to me.

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Shared-memory multiprocessors have provided a cost-effective solution to the problem of increased computing power and speed because they use relatively low-cost microprocessors interconnected with shared memory modules. But shared-memory multiprocessors are faced with three problems: Memory contention, Communication contention and Latency. These problems all contribute to increased memory access time and hence slow down the processors' execution speeds [19].

Cache memories have served as a significant way to reduce the average memory access time. The main memory traffic from each processor is determined by the success of the cache memory in satisfying memory requests without main memory operations [17]. In shared memory multiprocessors, all processors with private caches are limited in their performance by cache access time. Accordingly, cache memory performance is one of the most significant factors in achieving high machine performance.

Private caches in shared memory multiprocessors are essential to reduce the average time to access main memory and to decrease bus congestion [12]. But shared-memory multiprocessor systems introduce a *cache coherence problem* because multiple caches could have different copies of the same memory block if one of the processors has modified its copy. A system of caches is said to be coherent if all copies of a main memory location in multiple caches remain consistent when the contents of that memory location are modified [3].

For an example, consider cache coherence problem that can be caused by the sharing of writable data. Three figures are provided to describe cache coherence problem in shared memory multiprocessors. We assume that X' and Y' refer to the cached copies of X and Y in a shared memory. If P_0 and P_1 read two words in X and Y from a shared memory, then the *read* of two words in X and Y by two processors results in consistent copies of X and Y. Figure 1 shows caches and shared memory in a coherent state.

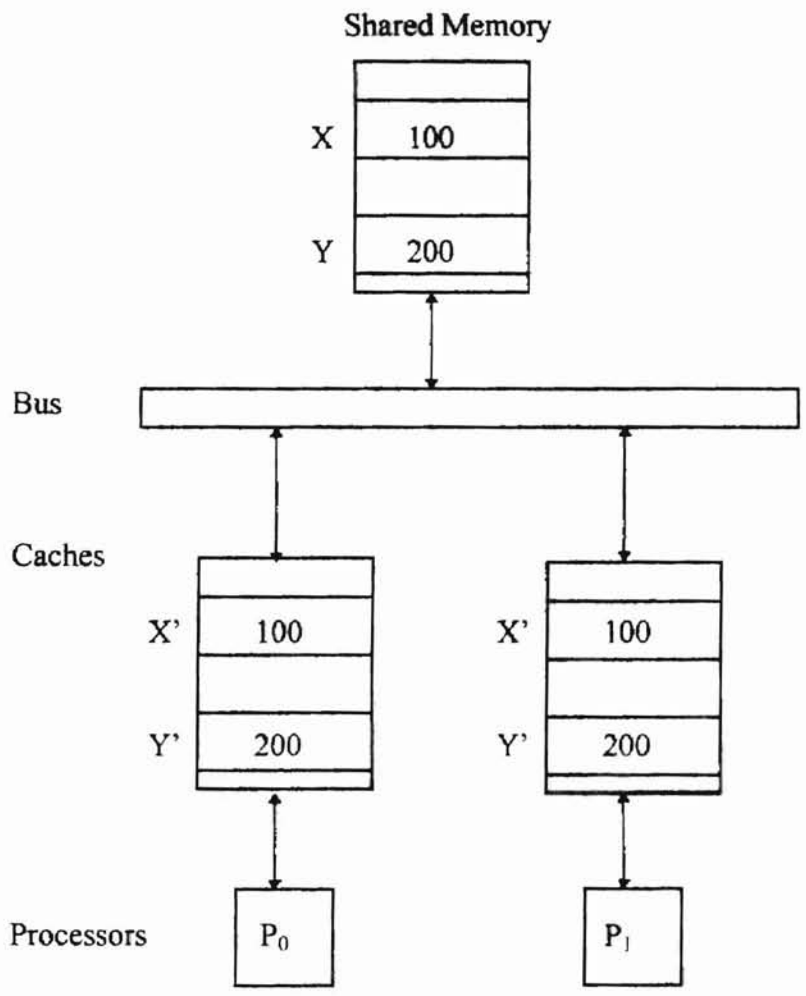

Figure 1. Cache configuration after reading two words by P_0 and P_1

Depending on the memory update policy used in the cache, the cache level may also be inconsistent with respect to main memory. A write-through policy maintains consistency between main memory and cache. If P_0 writes 300 into X' in P_0's cache , then the copies of X' in both caches become inconsistent, whereas the copies between

P_0's cache and memory are consistent. A *read* of a word in X' by P_1 will not return the latest value. Figure 2 shows an inconsistent state between P_0's cache and P_1's cache.

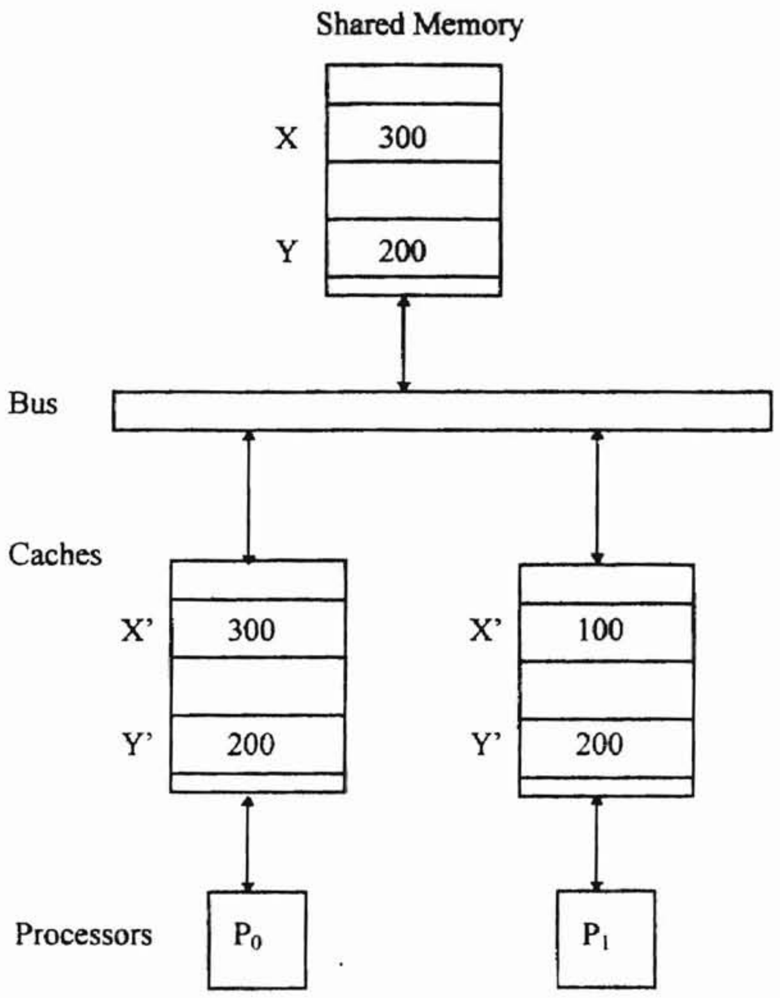

Figure 2. Cache configuration after writing a word in X' by P_0 (write-through cache).

However, a write-back policy does not maintain such consistency between main memory and cache at the time of *write*. The memory is updated eventually when the modified data in the cache are replaced or invalidated. If P_0 writes 300 into X' in P_0's cache , then the copies in both caches are inconsistent. Also, the copies between caches and memory are inconsistent. Figure 3 on the next page depicts the inconsistent state of the caches and memory for write-back policy.

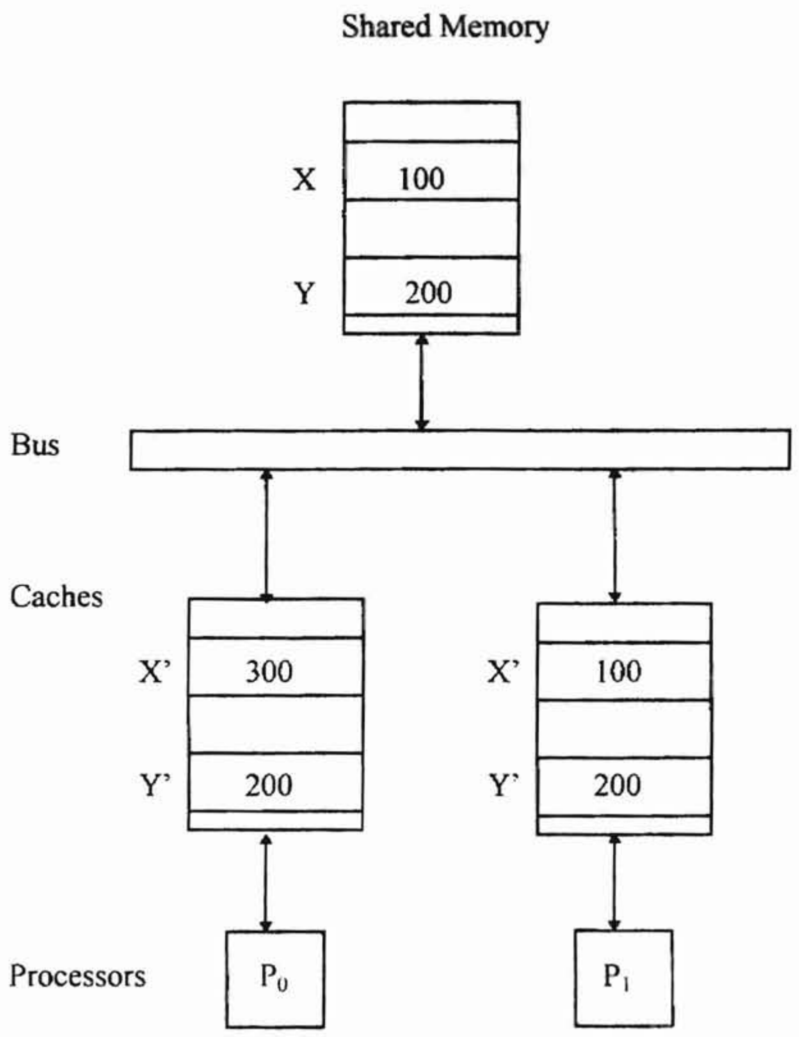

Figure 3. Cache configuration after writing a word in X' by P_0 (write-back cache).

The cache coherence problem has attracted considerable attention over the past years. A lot of research within university environments and company environments has been devoted to this problem, resulting in a number of proposed solutions.

Write-invalidate protocols, one of the many solutions to the cache coherence problem, allow multiple readers of the shared block, but only one writer at a time [21]. Write-invalidate protocols maintain coherency by requiring a writing processor to invalidate all other cached copies sharing the same data before updating its own data. It can then perform the current write, and any subsequent writes, without invalidation requests. Write-invalidate protocols have two main sources of bus-related coherency overhead. The first is the invalidation request of shared data in each cache. The second is the cache misses that occur when processors need to reference invalidated data. These misses, called *invalidation misses*, can result from an invalidation requested by another processor prior to the cache access. Invalidation request and invalidation misses are recognized as a main obstacle in achieving high performance for write-invalidate protocols [21].

In this thesis, we propose a *Hybrid Word Invalidate/Read Broadcast* protocol (HWRP) to reduce the invalidation misses which are an important performance issue for write-invalidate protocols. The hybrid word invalidate/read broadcast protocol is an extension of Word Invalidate Protocol presented by Tomašević and Milutinović [21], with one major difference: It uses a *read broadcast* mechanism [16] which can simultaneously update invalid copies while a data item is transferred on the bus as a response to a read miss request. Also, we study in this thesis the effectiveness of the new scheme using simulation. The organization of the rest of the thesis is as follows. A literature review of cache coherency protocols is presented in Section 2. Then in Section 3, we present a hybrid word invalidate/read broadcast protocol (HWRP). Section 4 presents the simulation model and results are analyzed in Section 5. We finally conclude in Section 6.

2. LITERATURE REVIEW

Basically, all solutions to the cache coherence problem can be classified in two large groups: software-based and hardware-based [19].

In the software-based approach, most solutions generally depend on the actions of the programmer, compiler, or operating system, in handling the cache coherence problem. Several software-based protocols have been proposed where memory blocks are tagged as cacheable or noncacheable depending on the access pattern to shared data. Read-only or non-shared data can always be cached, but shared read-writable data can never be cached to prevent the existence of inconsistent cached data. In software-based schemes, an advantage is that software schemes are generally less expensive than their hardware counterparts, although they may require considerable hardware support [22]. A disadvantage is that they all suffer from high cache miss ratio for shared read-writeable data structures simultaneously accessed by several processors [18]. Software-based solutions are not considered further in this thesis.

In the hardware-based approach, cache coherence protocols can be divided into two large groups: directory-based protocols and snoopy-based protocols. Directory protocols are appropriate for multiprocessors with general interconnection networks. The directory protocols are characterized by the existence of some kind of global table or directory that stores the information concerning the current location and state of shared blocks. Unlike the directory protocols, snoopy protocols are suitable for multiprocessors with a shared bus. Snoopy protocols differ substantially from directory protocols for general networks because first, they depend on each snoopy cache controller observing the bus transactions of all other processors in the system, then taking appropriate actions to maintain consistency, and second, the state of each block in the system is encoded in a distributed way among all cache controllers [3].

2.1 Directory Cache Coherency Protocols

Directory Protocol is characterized by the existence of some kind of global table or directory that stores the information concerning the current location and state of shared blocks [22]. Directories can be organized in different ways and it is the responsibility of the centralized controller to take appropriate actions to preserve the coherence by sending directed individual messages to known locations, avoiding the broadcasts [22]. The Directory Protocols are predominantly delegated to a centralized controller that implements the algorithm which moves data into and out of the cache memory and the cache directory. The centralized controller checks the directory and issues necessary commands for data transfer between memory and caches, or between caches themselves. It is also responsible for keeping status information up-to-date, so every local action that can affect the global state of the block must be reported to the centralized controller. Besides the global directory maintained by centralized controller, the private caches store some local state information about cached blocks.

Directory methods generally suffer from significant memory overhead for tag storage, so newly generated solution try to avoid this problem by introducing a limited number of pointers in the directory or employing distributed directories in the form of linked lists [22]. Since the directory is a critical system resource, frequent needs for directory accesses can seriously damage the system performance.

The directory methods can be divided into three groups: full-map directory, limited directory, and chained directory schemes. Chained directory scheme and Limited directory scheme are reviewed in this thesis.

2.1.1 Limited Directory Protocol

The Limited directory protocol is designed to solve the directory size problem which is a significant memory overhead for tag storage.

A directory protocol can be classified as Dir_i X using the notation from Agarwal [1]. The symbol *i* stands for the number of pointers, and X is either NB for a scheme with no broadcast or B for one with broadcast. A full-map scheme without broadcast is represented as Dir_N NB. A limited directory protocol that uses $i < N$ pointers is denoted by Dir_i NB. The limited directory protocol is similar to the full-map directory, except in case when more than *i* caches request read copies of a particular block of data.

Figure 4 on the next page shows the cache configuration after cache C_1 and cache C_2 request a copy of a location X copy in a memory system with a Dir_2 NB protocol. In this case, we can view the two-pointer directory as a two-way set-associative cache of pointers to shared copies. If cache C_3 requests a copy of a location X, the memory module must invalidate the copy in either cache C_1 or cache C_2. This process of pointer replacement is sometimes called *eviction*. Since the directory acts as a set associative cache, it must have a pointer replacement policy that requires no extra memory overhead [5]. In Figure 5, the pointer to cache C_3 replaces the pointer to cache C_2.

Limited Directory Protocol is very storage efficient and expandable for growing number of processors without any further modification. State information is distributed over memory or cache modules, which reduces contention. Furthermore, the presence flag vector stores the residency of copies, eliminating the need for the search associated with full-map directory scheme.

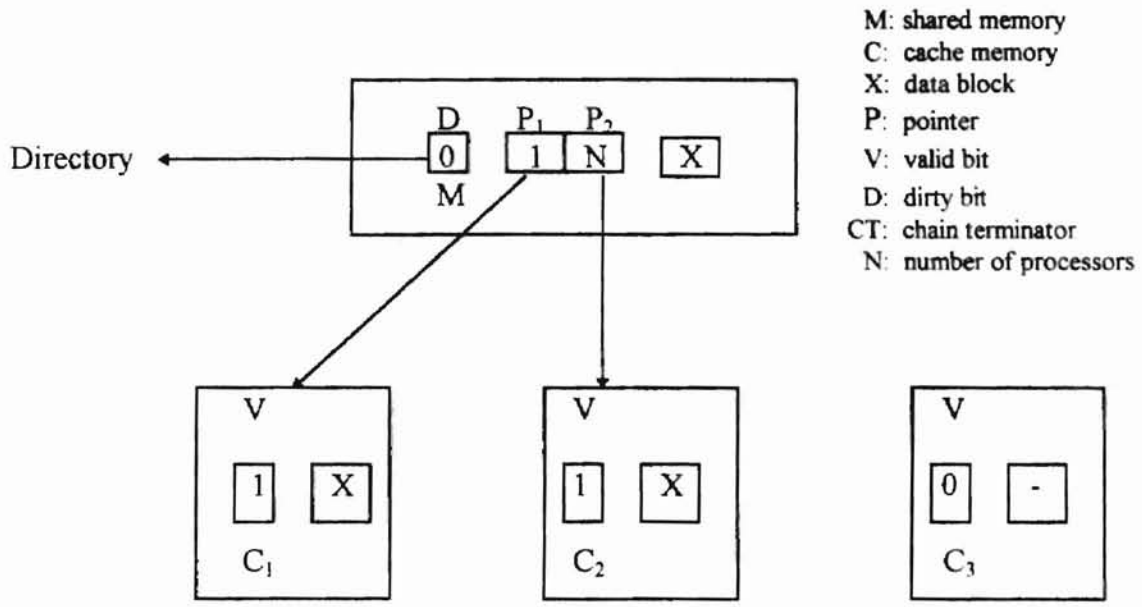

Figure 4. Cache configuration after cache C_1 and cache C_2 request a data block of a location X in memory.

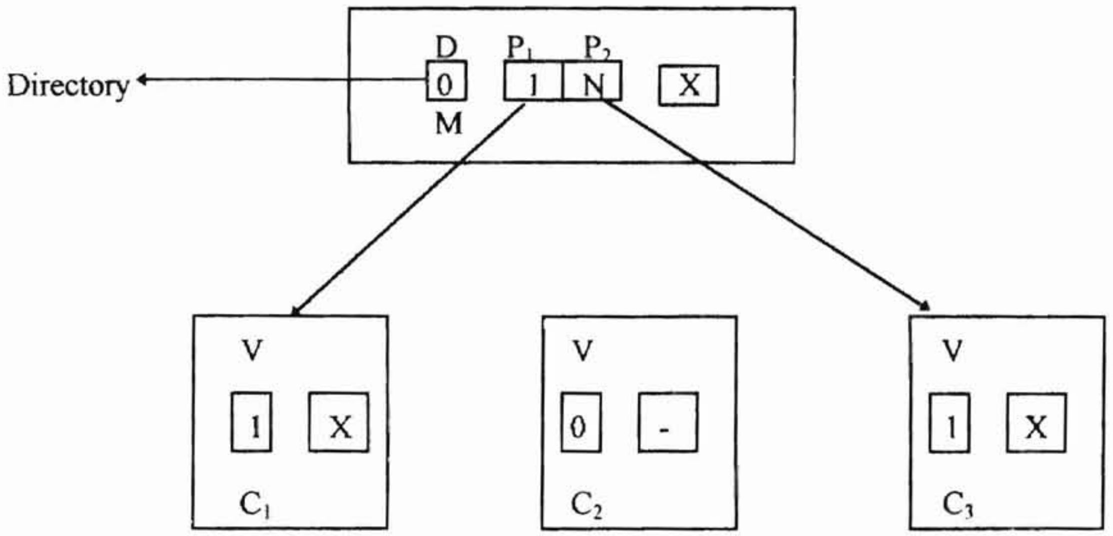

Figure 5. Cache configuration after cache C_3 requests a data block of a location X in memory.

2.1.2 Chained Directory Protocol

Another way to ensure scalability of directory schemes with respect to tag storage efficiency is the introduction of chained directory scheme. It is important that the approach does not limit the number of cached copies. Entries in such a directory are organized in the form of linked lists, where all caches sharing the same block are chained through pointers into one list. Unlike the limited directory approach, a chained directory scheme is spread across the individual caches. Entry into the main memory is used only to point to the head of the list and keep the block status.

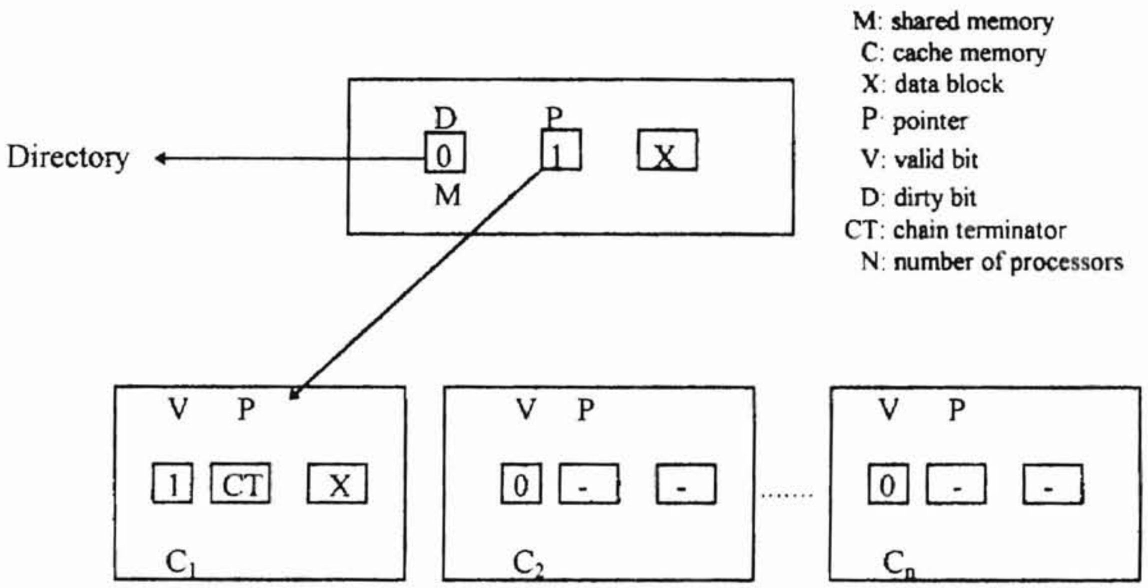

Figure 6. Cache configuration after cache C_1 requests a data block of a location X in memory.

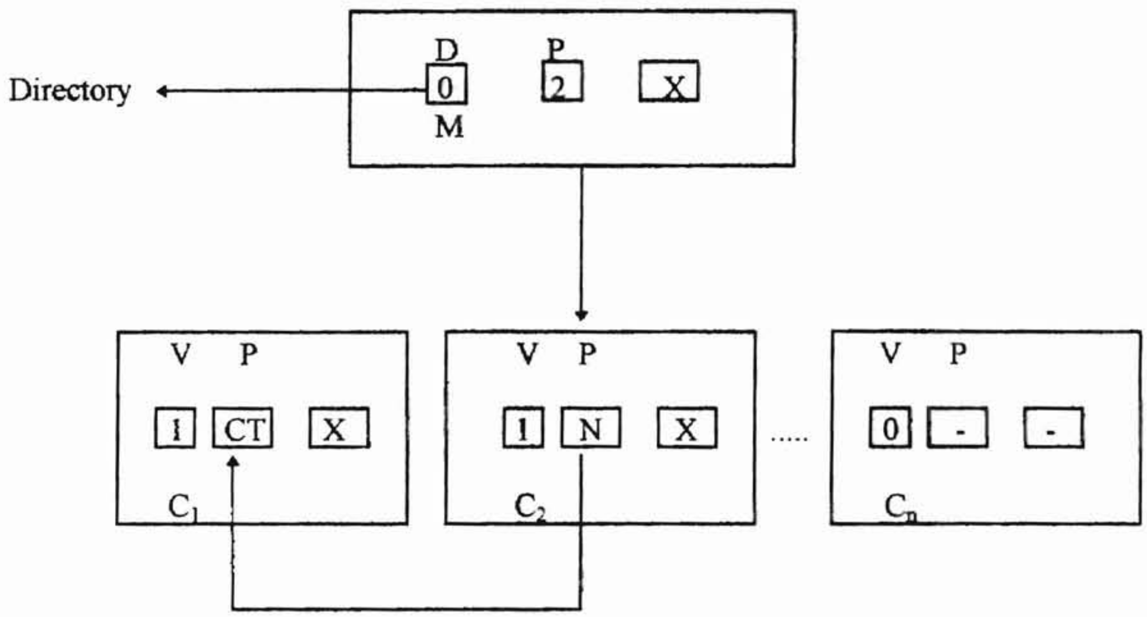

Figure 7. Cache configuration after cache C_2 requests a data block of a location X in memory.

Requests for the block are issued to the memory and subsequent commands from the memory controller are usually forwarded through the list, using the pointers. Then the chained directory can be organized in the form of either singly-linked lists or doubly-linked lists.

Suppose there are no shared copies of location X. If cache C_1 reads location X, the memory sends a copy to cache C_1, along with a chain termination (CT) pointer in Figure 6. The memory also keeps a pointer to cache C_1. Subsequently, when cache C_2 reads location X, the memory sends a copy to cache C_2, along with the pointer to cache C_1. The memory then keeps a pointer to cache C_2 in Figure 7. By repeating this step, all of the caches can cache a copy of location X.

Although the chained protocols are more complex than the limited directory protocols, they are still scaleable in terms of the amount of memory used for the directories. The main advantage of chained directory schemes is their scalability, while performance is almost as good as in full-map schemes [5].

2.2 Snoopy Cache Coherence Protocols

In snoopy cache coherence protocols [8,10,14,15,16,20,21], the approach to cache coherence is based on the actions of local cache controllers and distributed local state information by watching all coherency transactions from the bus. All the transactions for the currently shared block must broadcast to all other caches to maintain cache coherence. Local cache controllers are able to snoop on the bus and to recognize the actions and conditions for a coherence violation [22]. Actions are taken to preserve cache coherence according to the protocol used. The snoopy cache coherence protocols are divided into two groups by applying two write policies: *write-invalidate protocols* and *write-update protocols*. In write-invalidate protocols, a processor invalidates all other cached copies of shared data and can then update its own without further bus operations [6]. Unlike write-invalidate protocols, write-update protocols follow a distributed write approach that allows the existence of multiple copies with write permission. The word to be written to a shared block is broadcast to all caches, and caches containing that block can update it. Write-update protocols usually employ a special bus line for dynamic detection of the sharing status for a cache block. While invalidation misses are effectively eliminated by write-update protocols, their major disadvantage is the extraneous network traffic caused by the updates that now have to be propagated to all caches having copies of a block [9]. The following table 1 shows how the actions of various snoopy cache coherence protocols to maintain cache coherence.

Table 1. Snoopy Cache Coherence Protocols

WI : Write Invalidate Protocol
WU : Write Update Protocol

Protocols	**Read Miss**	**Write Hit**	**Write Miss**
Write - Once (WI)	**If another cache is the owner of missed block,** • The owner writes the block back to main memory and supplies the block to the requesting cache. • The requesting cache sets its local state to Valid. **If main memory is the owner of missed block,** • The block comes from memory. • All caches with a copy of the block set their state to Valid.	**If the block is in state Dirty or in state Reserved,** • Write to the block and update the local state to Dirty. **If the block is in state Valid,** • Write to the block and update main memory with the new data. • A Write-Inv consistency command is broadcast to all caches, invalidating their copies. • Updates the local state to Reserved.	Like a read miss, the block always comes from the owner. **If another cache is the owner of the missed block,** • The owner writes the block back to main memory and supplies the block to the requesting cache. • Send a Read-Inv consistency command which invalidates all cached copies. • The requesting cache sets its local state to Dirty.
Synapse N+1 (WI)	**If another cache is the owner of missed block,** • The owner writes the block back to main memory. • The owner updates the local state to Invalid. • The requesting cache must then send an additional miss request to get the block from main memory. **If main memory is the owner of missed block,** • The block comes from main memory. • The loaded block state always is set to Valid.	**If the block is in state Dirty,** • Write to the block and update the local state to Dirty. **If the block is in state Valid,** • The procedure is identical to a write miss since there is no invalidation signal.	Like a read miss, the block always comes from memory. **If another cache is the owner of missed block,** • It must first be written to memory by the owner. • All other caches with copies change their state to Invalid. • The block in the requesting cache is loaded in state Dirty.
Berkeley (WI)	**If another cache is in Dirty state or in Shared-Dirty state,** • The owner must supply the block directly to the requesting cache and set its local state to Shared-Dirty.	**If the block is in state Dirty,** • Write to the block and update the local state to Dirty. **If the block is in state Valid or in state Shared-Dirty,** • Send an invalidation signal	Like a read miss, the block comes directly from the owner. **If another cache is the owner of missed block,** • All other caches with copies change their state

	• The requesting cache sets its local state to Shared-Dirty. **If main memory has Dirty copy,** • The block comes from main memory. • The loaded block state is set to Valid.	to system bus before the write is allowed to proceed. • All other caches invalidate their copies upon matching the block address. • Update the local state to Dirty.	to Invalid. • The block in the requesting cache is loaded in state Dirty.
Illinois (WI)	**If another cache is the owner of missed block,** • The owner supplies the block directly to the requesting cache, updates main memory with dirty copy and sets its local state to Shared. • The requesting cache sets its local state to Shared. **If another cache has Shared or Valid Exclusive copy,** • The owner supplies the block directly to the requesting cache and sets its local state to Shared. • The requesting cache sets its local state to Shared. **If main memory is the owner of missed block,** • The block comes from main memory. • The loaded block state is set to Valid-Exclusive.	**If the block is in state Dirty or in state Valid-Exclusive,** • Write to the block and update the local state to Dirty. **If the block is in state Shared,** • Send an invalidation signal to system bus before the write is allowed to proceed. • All other caches invalidate their copies upon matching the block address. • Update the local state to Dirty.	Like a read miss, the block comes directly from the owner. **If another cache is the owner of missed block,** • All other caches with copies change their state to Invalid. • The block in the requesting cache is loaded in state Dirty
RB (WI)	**If another cache is the owner of missed block,** • The owner interrupts the bus read and performs its own bus write. • Updates memory to the correct value. • The bus read will be retried immediately. • All the caches update with the correct value from the bus read and change into state Read. **If main memory is the owner of missed block,** • The block comes from main memory • The loaded block state is set to Read.	**If the block is in state Local (Dirty),** • Write to the block and update the local state to Local. **If the block is in state Read,** • A write updates the block and a bus write is generated. • The cache state is set to Local. • The bus write updates the memory and at the same time causes all other caches to change into state Invalid.	Like a read miss, the block comes directly from the owner. **If another cache is the owner of missed block,** • A write updates the block and a bus write is generated. • The cache state is set to Local. • The bus write updates the memory and at the same time causes all other caches to change into state Invalid

RWB (WI)	**If another cache is the owner of missed block,** • The owner interrupts the bus read and performs its own bus write updating memory to the correct value. • The bus read will be retried immediately. • All the caches update with the correct value from the bus read and change into state Read. **If main memory is the owner of missed block,** • The block comes from main memory. • The loaded block state is set to Read.	**If the block is in state Local,** • Write to the block and update the local state to Local. **If the block is in state Read,** • The first write to a shared block updates that block and broadcasts the new value to all other caches sharing that block. • The requesting state is changed to F but all other caches'state with the copy of that block remain in state Read. • A subsequent write carries out and broadcasts an invalidate signal that all other caches invalidate their copies upon matching the block address.	**If another cache is the owner of missed block,** • A bus write is generated, the cache value is updated to this new value, and broadcasts the new value to all other caches sharing that block. • The requesting state is changed to F but all other caches'state with the copy of that block remain in state Read. • A subsequent write carries out and broadcasts an invalidate signal causing all other caches to enter state Invalid .
WIP (WI)	**If another cache is the owner of missed block,** • If a block is not in cache or invalidated, the owner will supply a whole valid block to the requesting cache. The requesting cache sets its local state to Unmod-Shared. • If a missed block is in state IW1, the owner will supply only a valid word, not a whole valid block, to the requesting cache. The requesting cache sets its local state to Unmod-Shared. • If a missed block is in state IW2, the owner will supply only a valid word, not a whole valid block to the requesting cache. The requesting cache sets its local state to IW1. **If main memory is the owner of missed block,** • The whole valid block comes from main memory. • The loaded block state is set to Unmod-Exclusive.	**If the block is in state Mod-Exc or Unmod-Exc,** • Write to the block and update the local state to Mod-Exc. **If the block is in state Mod-Shd or Unmod-Shd,** • Send an invalidation signal to system bus before the write is allowed to proceed. • All other caches invalidate their copies upon matching the block address. • Update the local state to Mod-Shd. **If the block is in state IW1 or IW2** • The whole valid block will be reloaded before the write takes place, as in the case of a write miss. • Send an invalidation signal to system bus before the write is allowed to proceed. • All other caches invalidate their copies upon matching the block address. • If the invalidated block is in state IW1, update the state to IW2.	Like a read miss, the block comes directly from the owner. **If the missed block is in state INV, not in cache or in state IW2,** • It will be loaded in the same way as when a read miss occurs and then a write is followed. • Update the local state to MOD-Exc or Mod-Shd. **If the missed block is in state IW1,** • Send an invalidation signal to system bus before the write is allowed to proceed. • All other caches invalidate their copies upon matching the block address. • Update the local state to Mod-Shd.

		• If the invalidated block is in state IW2, update the state to INV(fully invalidated). • If the invalidated block is in the other states, update the state to IW1.	
Firefly (WU)	**If another cache is the owner of missed block,** • The owner will supply the block directly to the requesting cache. and update main memory. The requesting cache sets its local state to Shared. **If other caches have Shared copy,** • The other caches with shared copy supply the block to the requesting cache. The requesting cache sets its local state to Shared. **If main memory is the owner of missed block,** • The block comes from main memory • The loaded block state is set to Valid-Exclusive.	**If the block is in state Dirty or Valid Exclusive,** • Write to the block and update the local state to Dirty. **If the block is in state Shared,** • The other caches (including memory copy) with shared copy are updated. • The resulting state is Shared. • If sharing has ceased, then the next state is Valid-Exclusive.	Like a read miss, the block comes directly from the owner. **If another cache is the owner of the missed block,** • The other caches (including memory copy) with shared copy are updated. • The resulting state is Shared.
Dragon (WU)	**If another cache is the owner of missed block,** • That cache supplies the data to the requesting cache. The requesting cache sets its block state to Shared-Dirty. **If main memory is the owner of missed block,** • The block comes from main memory. • Any cache with a Valid-Exclusive or Shared-Clean copy raises the SharedLine and set their local state to Shared-Clean. • The requesting cache loads the block in state Shared-Clean if the SharedLine is high; otherwise, it is loaded in state Valid-Exclusive.	**If the block is in state Dirty or Valid Exclusive,** • Write to the block and update the local state to Dirty **If the block is in state Shared,** • The other caches (including memory copy) with shared copy are updated. • The resulting state is Shared-Clean and raises the SharedLine, indicating that the data are still shared. • By observing this line on the bus, the cache performing the write can determine whether other caches still have a copy and hence whether further write to that block must be broadcast. • If the SharedLine is not raised, the block state is changed to Dirty; else it is set to Shared-Dirty.	Like a read miss, the block comes directly from the owner. **If another cache is the owner of missed block,** • That cache supplies the data to the requesting cache. The requesting cache sets its block state to Shared-Dirty. • Other caches with copies set their local state to Shared-Clean. • Upon loading the block, the requesting cache sets the local state to Dirty if SharedLine is not raised. • If the SharedLine is high, the requesting cache sets the state to Shared-Dirty and performs a single-word bus write to broadcast the new contents.

In Section 3, we propose a hybrid word invalidate/read broadcast approach to reduce invalidation misses. The hybrid word invalidate/read broadcast is based and developed on write- invalidate protocols, specifically Word Invalidate protocol and Read broadcast. Therefore, the remainder of this chapter focuses on write-invalidate protocols to show how the hybrid word invalidate/read broadcast protocol is related to write-invalidate protocols.

2.2.1 Write-Invalidate Protocols

Figure 8 and Figure 9 illustrate how write-invalidate protocols work basically. Figure 8 demonstrates that copies in three caches are consistent. From Figure 8, if P_2 tries to write the data ($x \rightarrow \mathbf{X'}$) in the block of private cache of P_2, then P_2 sends an invalidation request to a shared bus to invalidate all other cached copies. The invalidation request is carried out via a shared bus. Caches of other three processors monitor the bus through the snoop portion of their cache controllers. When they detect an address match, they invalidate the entire cache block containing the address. Figure 9 shows that the other two caches invalidate their entire block upon matching the block address.

In the introduction, we mentioned about the bus related coherency overheads of write-invalidate protocols for maintaining cache coherency: invalidation requests and

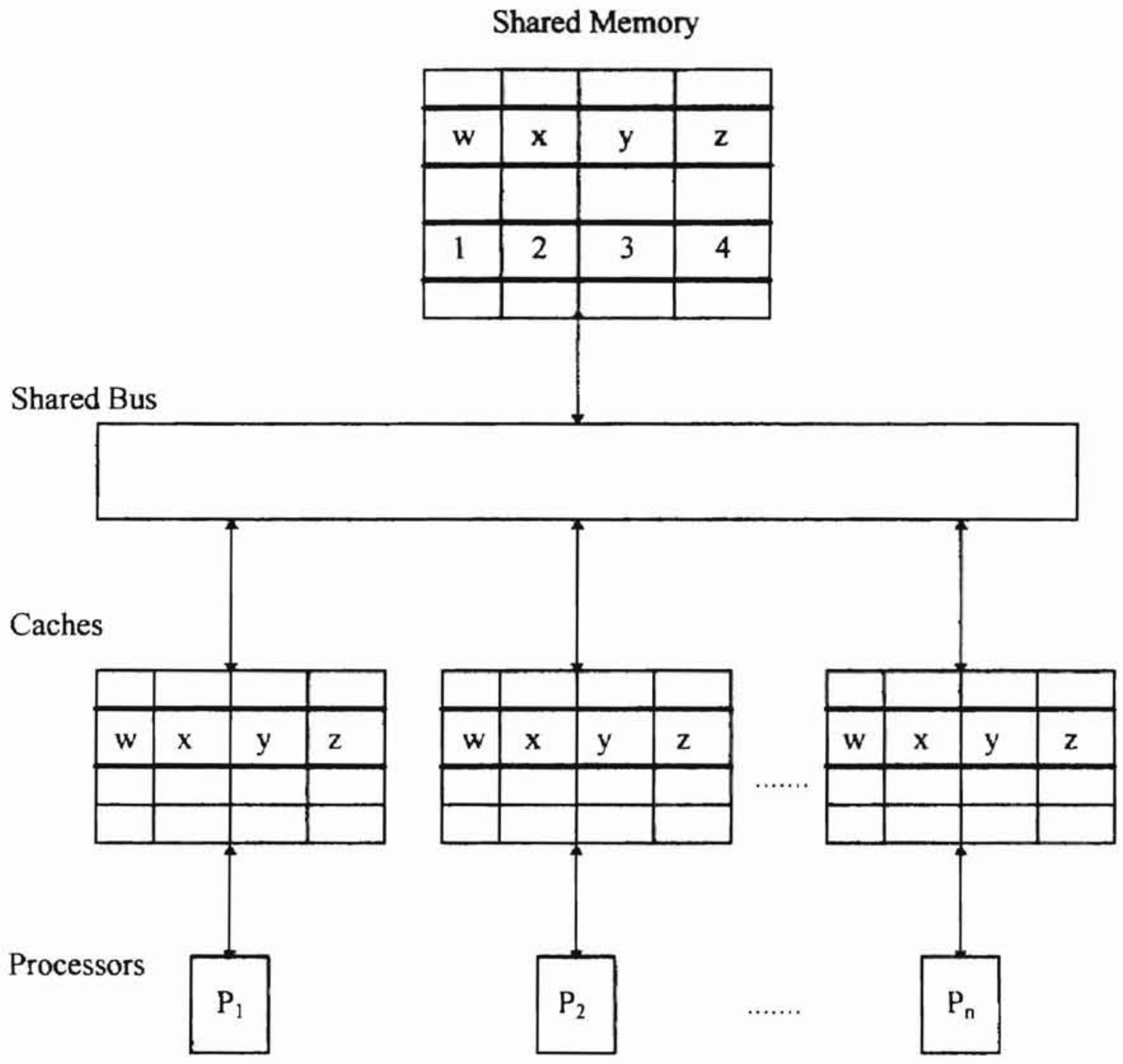

Figure 8. Cache configuration after reading four words from the memory

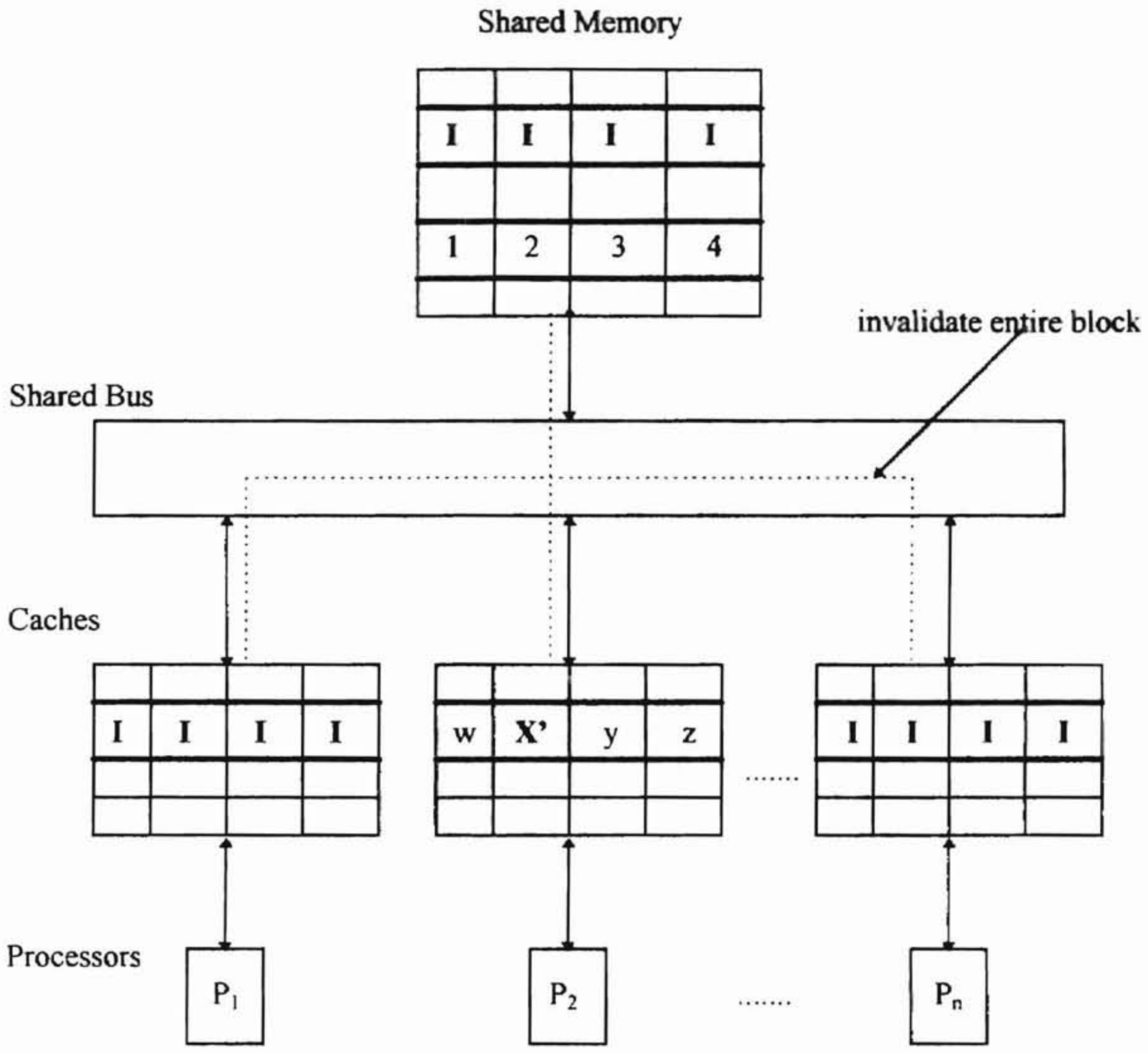

Figure 9. Cache configuration after writing ($x \rightarrow \mathbf{X'}$) on one word in block of P_2's cache.

invalidation misses. Write-invalidate protocols suffer from memory-access penalties due to invalidation misses and invalidation requests[9]. Write-invalidate protocols have two main sources that can increase invalidation requests and invalidation misses : *severe inter-processor contention* and *a large block size*. Severe inter-processor contention for an address produces more invalidations; the invalidations interrupt all processors' use of the data and increase the number of invalidation misses [6]. The overhead of maintaining cache coherency can be made worse by a larger block size, because contention can occur for any of an address in the block. Therefore, the probability that the block will be actively shared increases. So, increasing the block size cannot reduce invalidation misses [19]. Consequently, the additional invalidation requests and invalidation misses increase bus utilization. Reducing the number of invalidation requests and the number of invalidation misses is the most important performance issue for write-invalidate protocols.

2.2.2. Word Invalidate Protocol (WIP)

An invalidation of a word in a block usually causes all other words in the block to be invalidated. When other processors subsequently reread these addresses on the next reference, additional read misses are incurred because the block is invalidated fully. The overhead is paid even when a processor reads an address that was not updated. To protect useful valid data from unnecessary invalidation, Tomašević and Milutinović introduced *Word Invalidate Protocol* as an enhancement of write-invalidate protocol to minimize the overhead cost paid by an entire block invalidation in their paper [21].

The WIP (Word Invalidate Protocol) differs from the other write-invalidate protocols that usually use a whole block invalidation because WIP invalidates only one word in a block, instead of the usual block invalidation. Each time some processor updates a word in a block, it sends a request to other processors sharing that block to invalidate only the requested word , not the full block [21]. If a processor tries to read or write any invalidated word from the block in the cache, then a read miss or a write miss occurs because the word is invalidated. Only the valid word is reloaded instead of reloading the full block. After reloading only the valid word, the block is partially recovered if the block has two invalid words or fully recovered if the block has the only invalid word in the block.

WIP uses a pollution point which is a certain number of invalid words in a block. Tomaševic and Milutinovic have examined the influence of different pollution points on the WIP performance by simulating the WIP versions with 1, 2, and 3 allowed invalid words, for a block size of four words. As a result, they concluded that the version with two allowed invalid words is the most appropriate solution as the optimal pollution point for WIP. Under the pollution point, the allowed number of invalid words within the block (degree of pollution) may be just one or two. After reaching the pollution point, any subsequent invalidation request invalidates the whole block.

In [21], Tomašević and Milutinović compared WIP with Berkeley protocol which is the best representative of write invalidate protocol. As compared to Berkeley protocol, they show that WIP has lower number of invalidation requests. The WIP's selective invalidations save useful data in cache from being wasted. Consequently, WIP has a higher hit ratio for shared references than Berkeley protocol. WIP avoids some unnecessary invalidations and achieves better data utilization [22]. Tomašević and Milutinović [22] demonstrated that WIP has better data utilization and lower bus traffic than the other write-invalidate protocols using a whole block invalidation. The most important factor which brought better data utilization and lower bus traffic is reduction of invalidation misses.

2.2.3 Read Broadcast

Read Broadcast presented by Rudolph and Segal [16] and evaluated by Eggers and Katz [6] is an extension for snooping protocols that utilizes the broadcast nature of the bus. Under read broadcast, when a cache issues a bus read miss, the bus read will fetch the data stored in the memory. However, the owner of the missed cache block interrupts the bus read miss and performs its own bus write to update memory to the correct data. The original bus read will be retried immediately. Caches of the other processors monitor the bus through the snoop portion of their snoop controllers. When snoop controllers of the other caches detect the block's address on the bus with matching addresses to invalidated blocks, snoop controllers update their invalidated block with data from the bus.

3. A HYBRID WORD INVALIDATE/READ BROADCAST PROTOCOL (HWRP)

Word Invalidate Protocol may be classified as "event broadcasting", whereas in Read Broadcast, events and data values are broadcast. Through combination of the different classification between WIP and Read Broadcast, we propose HWRP (Hybrid Word Invalidate/Read Broadcast Protocol). HWRP is a modification of WIP with Read Broadcast capability for more reduction in invalidation misses.

Under HWRP, if a processor tries to read or write an invalidated word or an invalidated block in the cache, then a read miss request or a write miss request is broadcasted on the bus and then the owner of the missed word or the missed block puts either a valid word on the bus or a valid block on the bus. Snoop Controllers of the other caches with an invalidated word or an invalidated block update either an invalidated word or an invalidated block upon matching address from bus, when snoop controllers detect a read operation or a write bus operation for the block's address. Like the WIP, HWRP uses the idea of ownership. If the cache that has the block in state MOD-SHD or

MOD-EXC is the owner of that block. If a block is not owned by any cache, memory is the owner.

Under read broadcast, on a bus read miss, the owner of the missed cache block interrupts the bus read miss and performs its own bus write updating memory to the correct value. Unlike read broadcast, HWRP does not need to interrupt the bus read miss and to perform the bus write to update memory because HWRP uses direct cache to cache transfers, if a cache is the owner of a missed cache block.

The HWRP uses the same seven states that WIP uses. The seven states for cached blocks are given in Table 2.

Table 2. Summary of Cache Block States

State	Description
INV	Block does not contain valid word
IW1	Block has only one invalid word
IW2	Block has only two invalid words
UNMOD-EXC	Unmodified-Exclusive. No other cache has this block. Word in block is consistent with main memory
UNMOD-SHD	Unmodified-Shared. Some other caches may have this block
MOD-SHD	Modified-Shared. This block is owned, but it can not be updated without informing the other caches. Its data must be given to any requesting cache and flushed back to main memory.
MOD-EXC	Modified-Exclusive. This block is owned and unique. Therefore, data can be updated locally. Its data must be given to any requesting cache and flushed back to main memory.

The operation of the HWRP protocol is specified for all possible situation as follows:

Read Hit

- Upon a read hit, no coherence action is necessary because the read hit is defined as the read access to a valid block or to a valid word within a partially valid block.

Read Miss

Case 1) The block is not in cache.

- A request is made to the owner for the block.
- The owner puts the *valid block* on the bus.
- If the block is shared by any other cache, caches update that block with the value from the bus. If the state of the owner is MOD-SHD or MOD-EXC, then set the state of the cache as MOD-SHD.
- If the state of owner is MOD-EXC, change it to MOD-SHD.
- If memory is the owner, then the state of all sharing caches are set to UNMOD-SHD. If no other cache shares the block, then set the state of the requesting cache to UNMOD-EXC.

Case 2) The block is in cache with state INV.

A. If a cache is the owner of the *missed block*,

- The owner of the *missed block* accesses its own cache memory to provide a *valid block* to any requesting cache for a bus read miss request.
- The owner of the *missed block* puts the *valid block* on the bus.
- If the block shared by the other caches has been invalidated fully, the snoop controller of each cache accesses its own cache memory to update the *invalidated block* with the *valid block* from the bus upon matching the block address.
- The updated block state of the other caches is set to MOD-SHD state, after reloading the *valid block* from the bus.

B. If main memory is the owner of the missed block,

- The block comes from main memory.
- The loaded block state is set to UNMOD-EXC.

Case 3) The block is in cache with state IW1 or IW2.

A. If a cache is the owner of the *missed word*,

- The owner of the *missed word* accesses its own cache memory to provide a *valid word* to any requesting cache for a bus read miss request.
- The owner of the *missed word* puts the *valid word* on the bus.
- If the word in the block shared by the other caches has been invalidated, the snoop controller of each cache updates the *invalidated word* with the *valid*

word from the bus upon matching the block address.

- If the updated block state of the other caches is IW1, the updated block state of the other caches is set to MOD-SHD state because the updated blocks are recovered fully.
- If the updated block state of the other caches is IW2, the updated block state of the other caches is set to IW1 state because the updated blocks are recovered partially.

B. If main memory is the owner of the *missed word*,

- The *word* comes from main memory.
- The loaded block state is set to UNMOD-SHD, if the other caches are sharing the same block.
- The loaded block state is set to UNMOD-EXC, if the other caches are not sharing the same block.

Write Hit

If the block is in state MOD-EXC or UNMOD-EXC,

- Write to the block without an *invalidation request* and update the local state to MOD-EXC.

If the block is in state MOD-SHD or UNMOD-SHD,

- A *word invalidation request* will be issued on the bus before the write is allowed to proceed.
- All other caches sharing the block invalidate the *corresponding word* in their block upon matching the block address.

  1. If the block state of the *invalidated word* is in state MOD-SHD or in state UNMOD-SHD, they will be changed to the state IW1.
  2. If the block state of the *invalidated word* is in state IW1 and a *word invalidation request* tries to invalidate one of the valid words of the block, the block in state IW1 is changed to the IW2 state.
  3. If the block state of the *invalidated word* is in state IW2 and a *word invalidation request* tries to invalidate one of the valid words of the block, the block in state IW2 will be fully invalidated (IW2 $\rightarrow$ INV).

- Update the local state to MOD-SHD.

If the block is in state IW1 or IW2,

- The *whole valid block* will be reloaded before the write takes place.
- Like read miss, if the snoop controllers of the other caches detect a bus read

request upon matching the block address, update *invalidated words* in block with data from the bus.

- Send a *word invalidation request* to system bus before the write is allowed to proceed. All other caches invalidate a word in their block upon matching the block address.
- Update the local state to MOD-SHD.

Write Miss

Case 1) The block is not in cache.

A. If a cache is the owner of the *missed block*,

- The *write missed block* will be loaded in the same way as when a read miss occurs.
- Send a *word invalidation request* to system bus before the write is allowed to proceed.
- All other caches sharing the block invalidate the *corresponding word* in their block upon matching the block address.
- Update the local state to MOD-SHD.

B. If main memory is the owner of the *missed block*,

- The block comes from main memory.
- Send a *word invalidation request* to system bus before the write is allowed to proceed.
- The loaded block state is set to MOD-SHD, if the other caches are sharing the same block.
- The loaded block state is set to MOD-EXC, if the other caches are not sharing the same block.

Case 2) The block is in cache with state INV and being shared.

A. If a cache is the owner of the *missed block*,

- The *write missed block* will be loaded in the same way as when a read miss occurs.
- Send a *word invalidation request* to system bus before the write is allowed to proceed.
- All other caches sharing the block invalidate the *corresponding word* in their block upon matching the block address.
- Update the local state to MOD-SHD.

B. If main memory is the owner of the *missed block*,

- The block comes from main memory.
- The loaded block state is set to MOD-EXC.

Case 3) The block is in cache with state IW1.

A. If a cache is the owner of the *missed word*,

- Only the *write missed word* will be loaded in the same way as when a read miss occurs.
- Send a *word invalidation request* to system bus before the write is allowed to proceed.
- All other caches sharing the block invalidate the *corresponding word* in their block upon matching the block address.
- Update the local state to MOD-SHD.

B. If main memory is the owner of the *missed word*,

- Only the *write missed word* comes from main memory.
- Send a *word invalidation request* to system bus before the write is allowed to proceed.
- The loaded block state is set to MOD-SHD, if the other caches are sharing the same block.
- The loaded block state is set to MOD-EXC, if the other caches are not sharing the same block.

Case 4) The block is in cache with state IW2.

A. If a cache is the owner of the *missed word*,

- The *whole valid block*, not the *missed word* will be reloaded in the same way as when a read miss occurs to a block in state INV.
- Send a *word invalidation request* to system bus before the write is allowed to proceed.
- All other caches sharing the block invalidate the *corresponding word* in their block upon matching the block address.
- Update the local state to MOD-SHD.

B. If main memory is the owner of the *missed word*,

- The *whole valid block*, not the *missed word* comes from main memory.
- Send a *word invalidation request* to system bus before the write is allowed to proceed.
- The loaded block state is set to MOD-SHD, if the other caches are sharing the same block.

- The loaded block state is set to MOD-EXC, if the other caches are not sharing the same block.

3.1 The Hybrid Write Invalidate/Read Broadcast Protocol Detailed Description

This section provides some figures to describe how HWRP and WIP work differently for maintaining cache coherency. Only misses due to invalidation are considered. Assume that P_1, P_2 and P_3 are sharing the same data block as shown in Figure 10. The block state in the cache of each processor is UNMOD-SHD.

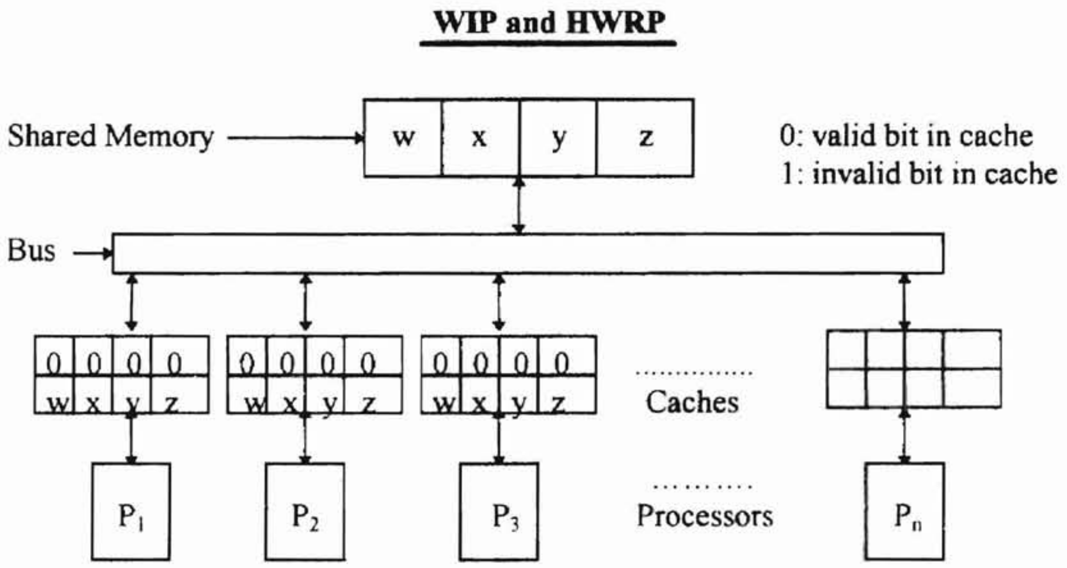

Figure 10. Cache configuration after a read on four words in a block of private caches of P_1, P_2 and P_3. Copies in all three caches are consistent.

WIP and HWRP

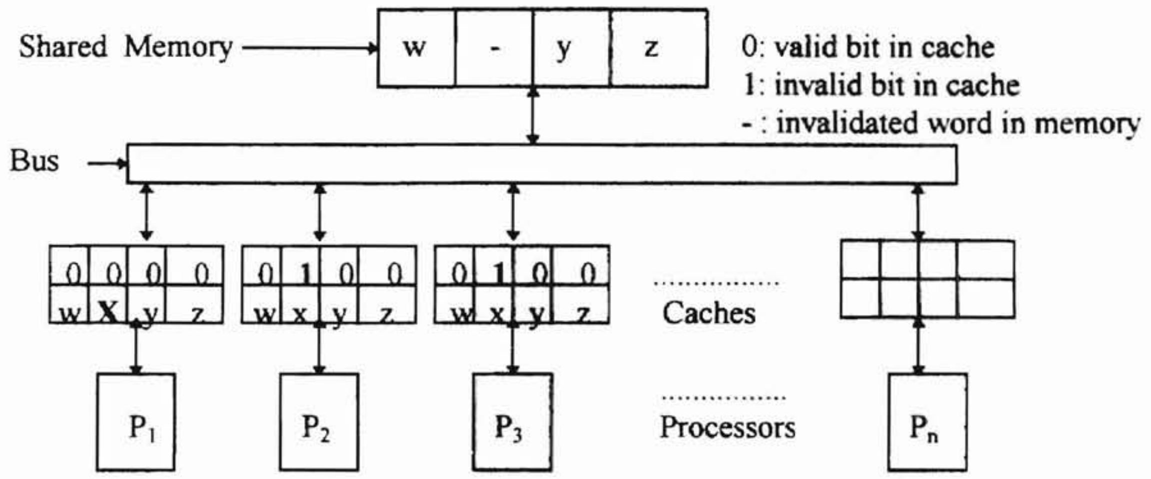

Figure 11. Cache configuration after a write on word ($x \rightarrow \mathbf{X}$) in a block by P_1 (write-back cache). The word "x" in a block of P_2's cache and P_3's cache is invalidated. The block state in P_2's cache and P_3's cache is changed to IW1. The block state in P_1's cache is MOD-SHD.

Figure 11 shows the cache configuration after P_1 modifies "x" in a block of P_1's cache. The Cache Controller of P_1 sends an invalidation request to the shared bus. The Snoop Controllers of the other processors monitor the bus and then the Snoop Controllers of P_2 and P_3 invalidate only that particular word upon matching address. The state of the block in P_2's cache and P_3's cache is changed from UNMOD-SHD to IW1. The state of the block in P_1's cache is changed from UNMOD-SHD to MOD-SHD.

Read Miss to A Block in The IW1 State

In Figure 11, if P_3 tries to read a word "x" from the block in P_3's cache, then a read miss occurs because the word is already invalidated by P_1. A read miss request is broadcasted on the bus and then P_1 (the owner of a missed block) puts the word "**X**" on the bus.

On a read miss of WIP, WIP updates only a word ($x \rightarrow \mathbf{X}$) of the block in P_3's cache. The state of the block in P_3's cache is changed from IW1 to UNMOD-SHD.

On a read miss of HWRP, the Snoop Controllers of P_2's cache and P_3's cache updates an invalidated word ($x \rightarrow \mathbf{X}$) with data from bus, when the Snoop Controllers of P_2's cache and P_3's cache detect a read bus operation for the block's address. Therefore, the state of the block in P_2's cache and P_3's cache is changed from IW1to UNMOD-SHD because there is no invalid word in the block of P_2's cache and P_3's cache.

Figure 12 and Figure 13 show the difference in cache configuration between WIP and HWRP on read miss to a block in the IW1 state.

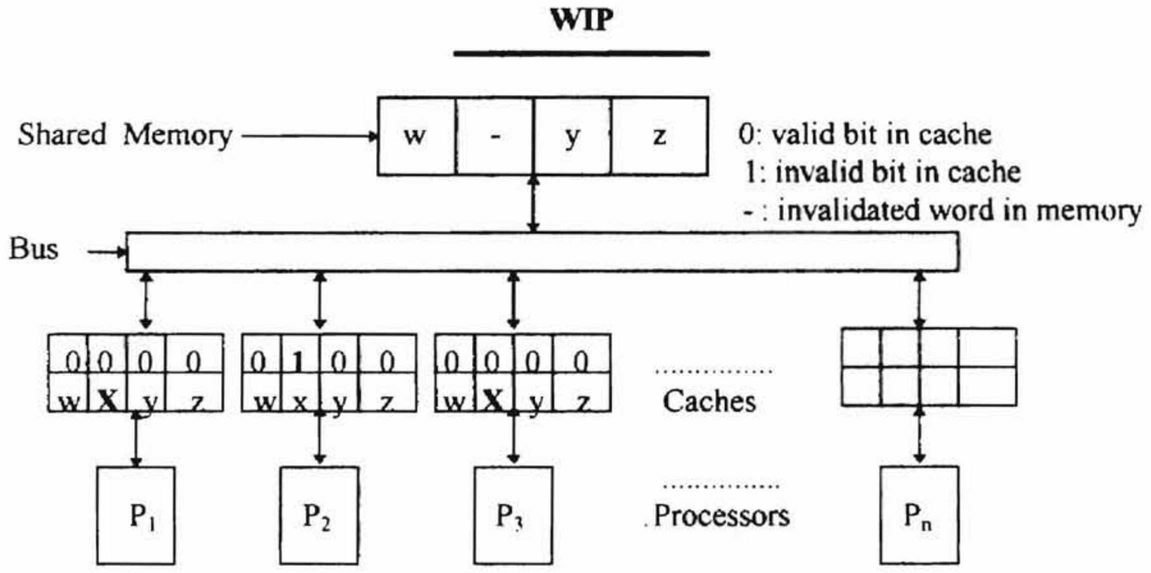

Figure 12. Cache configuration for WIP. The block state in P_2's cache is IW1. The block state in P_3's cache is UNMOD-SHD. The block state in P_1's cache is MOD-SHD.

HWRP

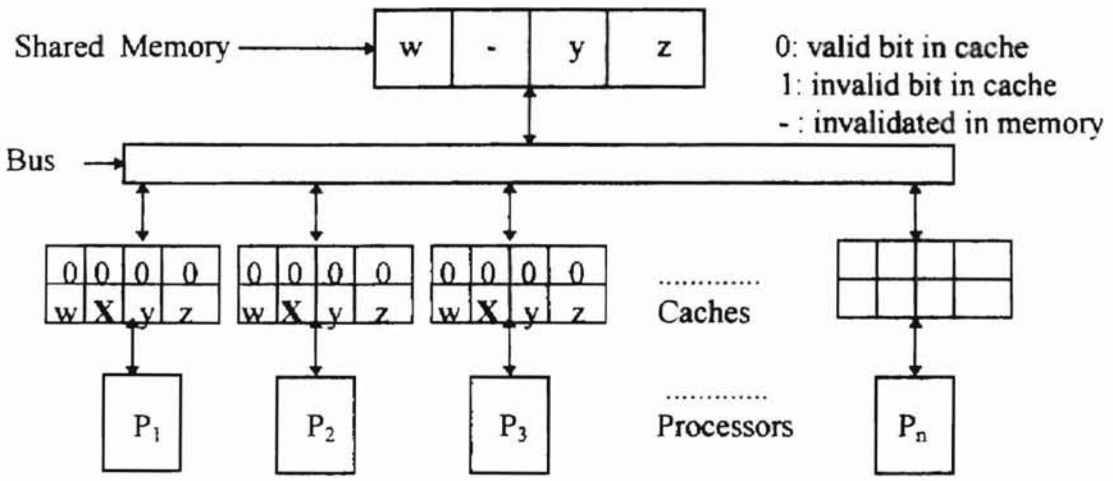

Figure 13. Cache configuration for HWRP. The block state in P_2's cache is UNMOD-SHD. The block state in P_3's cache is UNMOD-SHD. The block state in P_1's cache is MOD-SHD.

From Figure 11, if P_1 requests one more *write* to a valid word of a block in P_1's cache , then an invalidation request will be issued on the bus to invalidate one of the valid words of the block in P_2's cache and P_3's cache. If an invalidation request tries to invalidate one of the valid words of the block in the IW1 state, the block in P_2's cache and P_3's cache will get into the IW2 state. The block in P_1's cache stays in the MOD-SHD state. Figure 14 illustrates a cache configuration after a write on a valid word ($y \rightarrow \mathbf{Y}$) of the block in P_1's cache.

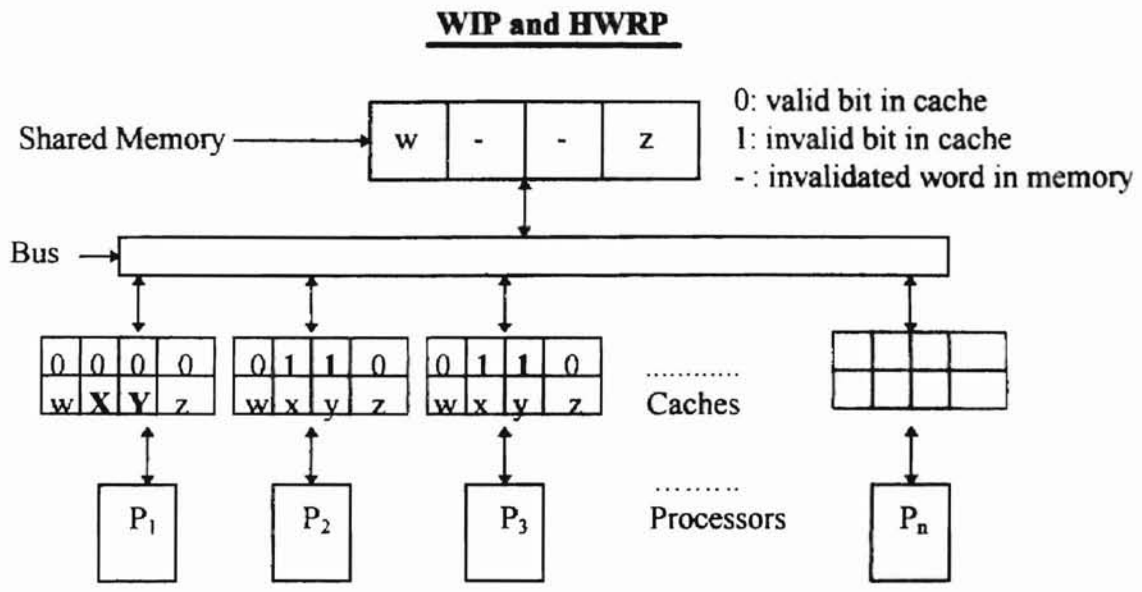

Figure 14. Cache configuration after a write on a word (y → **Y**) in block of P_1's cache by P_1. The word "y" of block in P_2's cache and P_3's cache is invalidated. The block state of P_2's cache and P_3's cache is changed to IW2. The block state of P_1's cache is MOD-SHD.

In the configuration shown from Figure 14, if P_3 tries to read either "x" or "y" from the block in P_3's cache, then a read miss occurs because the words already are invalidated by P_1. Assume that a read missed word is "**Y**". A read miss request is broadcasted on the bus and then P_1 (the owner of a missed block) puts a word "**Y**" on the bus.

Read Miss to A Block in IW2 State

On a read miss of WIP, WIP updates only a requested word (y → **Y**) of the block in P_3's cache. The state of the block in P_3's cache is changed from IW2 to IW1.

On a read miss of HWRP, P_2's cache gets the word "**Y**" from bus when the Snoop Controller of P_2's cache detects a read bus operation for the block's address. And then the invalidated word "y" is updated to "**Y**". Therefore, the state of the block in P_2's cache is changed from IW2 to IW1, since the number of invalidated words in the block are reduced from two to one. Like the WIP, HWRP updates a requested word ($y \rightarrow \mathbf{Y}$) of the block in P_3's cache. The state of the block in P_3's cache is changed from IW2 to IW1.

Figure 15 and Figure 16 show the difference in cache configuration between WIP and HWRP on read miss to a block in the IW2 state.

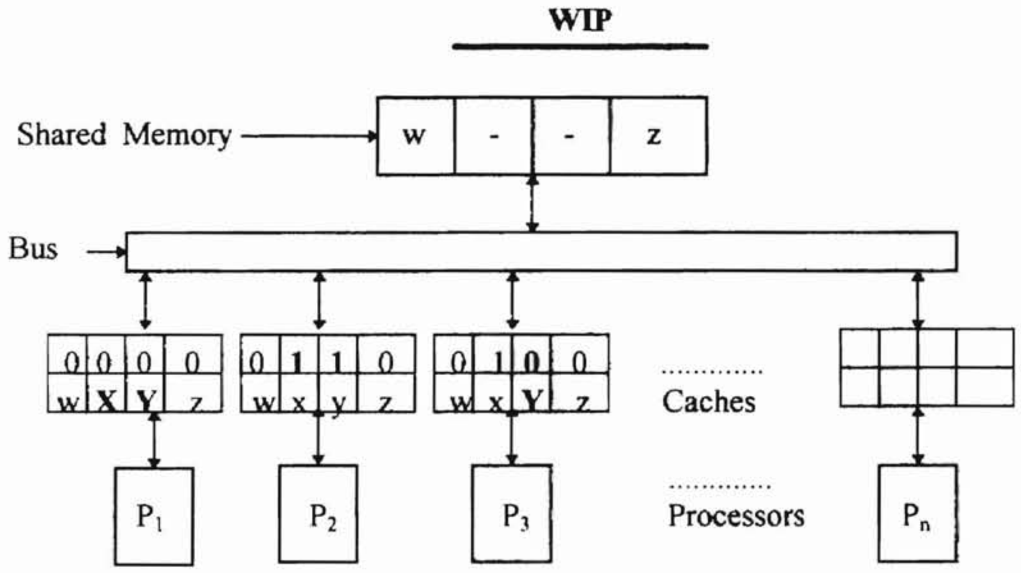

Figure 15. Cache configuration for WIP. The block state in P_2's cache is IW2. The block state in P_3's cache is IW1. The block state in P_1's cache is MOD-SHD.

HWRP

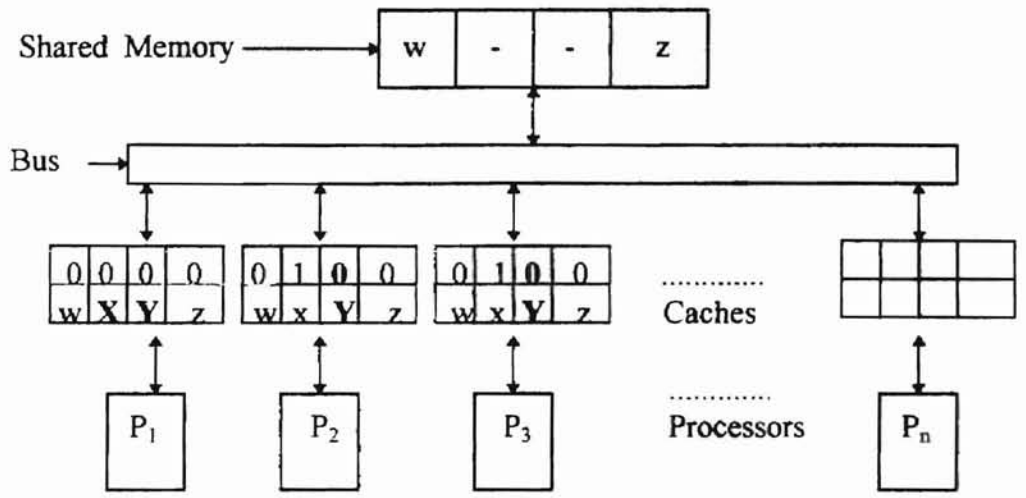

Figure 16. Cache configuration for HWRP. The block state in P_2's cache is IW1. The block state in P_3's cache is IW1. The block state in P_1's cache is MOD-SHD.

From the configuration shown in Figure 14, if P_1 tries to update a word ($z \rightarrow \mathbf{Z}$) in the block of P_1's cache, then the processor, P_1 sends an invalidation request on the bus to invalidate one of the valid words of the block in P_2's cache and in P_3's cache. The block in P_2's cache and the block in P_3's cache are fully invalidated (IW2 $\rightarrow$ INV). The processor P_1 updates a word ($z \rightarrow \mathbf{Z}$) in the block of P_1's cache. Figure 17 shows the cache configuration after a write ($z \rightarrow \mathbf{Z}$) on a word in block of P_1's cache.

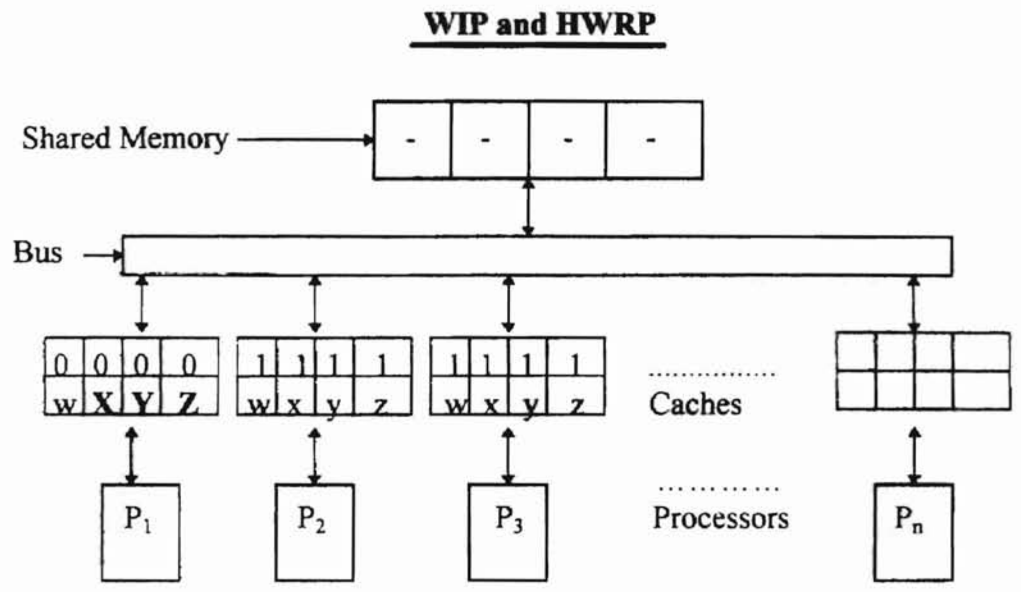

Figure 17. Cache configuration after a write on a word ($z \rightarrow \mathbf{Z}$) in block of P_1's cache by P_1. The entire block in P_2's cache and P_3's cache is fully invalidated. The block state in P_1's cache is MOD-SHD.

Read Miss or Write Miss to A Block in INV State

In Figure 17, if P_3 tries to read or write any word from the block in P_3's cache, then a read miss or a write miss occurs because the words are fully invalidated. A read miss request or a write miss request is broadcasted on the bus, then P_1 (the owner of a missed block) puts a whole valid block on the bus.

On a read miss of HWRP or on a write miss of WIP, the whole valid block in P_1's cache is reloaded into a block in P_3's cache. On the case of a write miss, the entire valid block should be reloaded before the write takes place. On a read miss, the state of the block in P_3's cache is changed from INV to UNMOD-SHD. But on a write miss, the state of the block in P_1's cache is changed from MOD-SHD to IW1 because a word of the block in P_1's cache is invalidated by an invalidation request issued by P_3 and then writes a word in the reloaded whole valid block. The owner of that block is changed from P_1's cache to P_3's cache. Therefore, the state of the block in P_3's cache will be MOD-SHD. The block in P_2's cache still stays in the INV state.

On a read miss of HWRP or on a write miss of HWRP, the Snoop Controllers of P_2's cache and P_3's cache with a fully invalidated block catch the entire valid block from the bus when the Snoop Controllers of P_2's cache and P_3's cache detect a read bus operation for the block's address. And then the invalidated block is reloaded. On a read miss, the state of the block in P_3's cache is changed from INV to UNMOD-SHD. The state of the block in the cache of P_2 is changed from INV to UNMOD-SHD. On a write miss, the state of the block in P_1's cache and P_2's cache is changed from MOD-SHD to IW1 because a word in the block of P_1's cache and P_2's cache is invalidated by an invalidation request issued by P_3. After the invalidation of a word in the block of P_1's cache and in the block of P_2's cache, P_3 updates a word in the reloaded valid block. The owner of that block is changed from P_1's cache to P_3's cache. Therefore, the state of block in P_3's cache will be MOD-SHD.

Figure 18 and Figure 19 show the difference in cache configurations between WIP and HWRP on read miss of a block in the INV state. Figure 20 and Figure 21 show the difference in cache configurations between the WIP and HWRP on write miss of a block in the INV state.

WIP on Read Miss

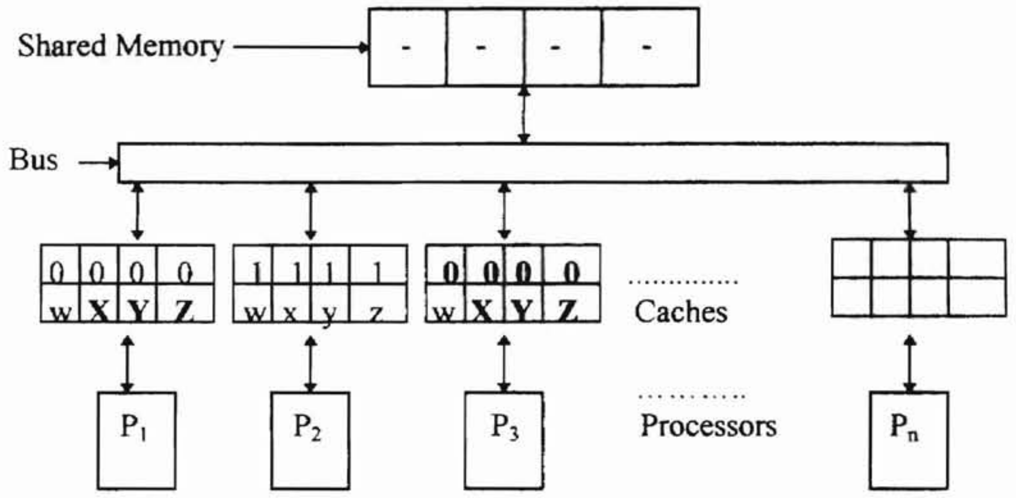

Figure 18. Cache configuration for WIP after a read request of P_3. The whole valid block is reloaded into a block in P_3's cache. The block state in P_1's cache is MOD-SHD. The block state in P_2's cache is still in INV state. The block state in P_3's cache is UNMOD-SHD.

HWRP on Read Miss

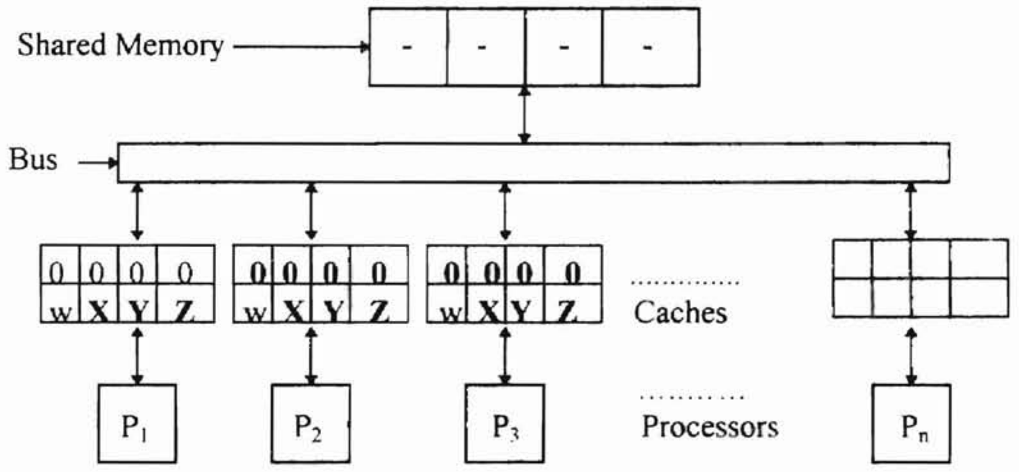

Figure 19. Cache configuration for HWRP after a read request of P_3. The whole valid block is reloaded into a block in P_2's cache and P_3's cache. The block state in P_1's cache is MOD-SHD. The block state in P_2's cache and P_3's cache is changed to UNMOD-SHD from INV.

WIP on Write Miss

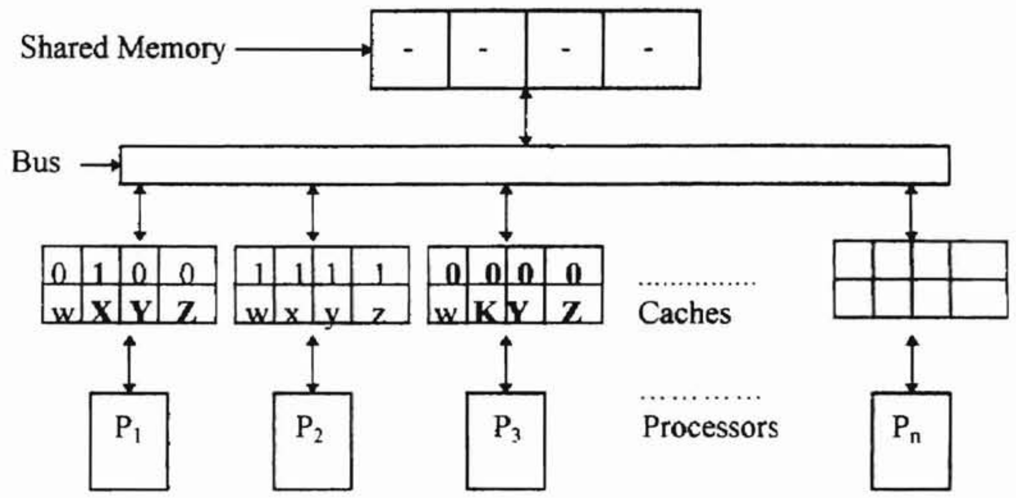

Figure 20. Cache configuration after the whole valid block is reloaded into a block in P_3's cache and then P_3 writes from a word "**X**" to a word " **K**" in the block of P_3's cache. The block state in P_1's cache is changed from MOD-SHD to IW1. The block state in P_2's cache is still in INV state. The block state in P_3's cache is changed from INV to MOD-SHD.

HWRP on Write Miss

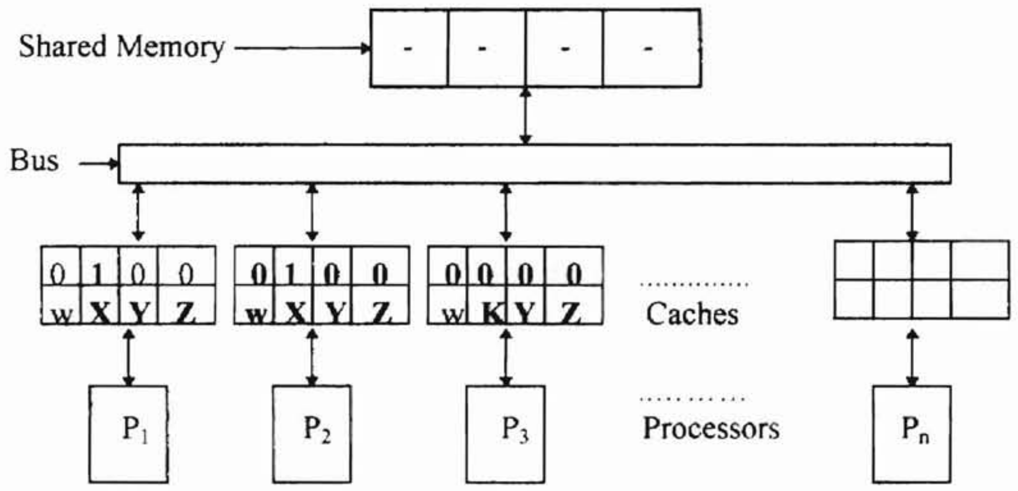

Figure 21. Cache configuration after the whole valid block is reloaded into a block in P_2's cache and P_3's cache and then P_3 updates from "**X**" to "**K**" in the block of P_3's cache. The block state in P_1's cache is changed from MOD-SHD to IW1. The block state in P_2's cache is changed from INV to IW1. The block state in P_3's cache is changed from INV to MOD-SHD.

Write Hit on IW1 State

There is a difference in cache configurations between WIP and HWRP on a write hit of a block in the IW1 state. Assume that P_1 tries to update a valid word (**Y** $\rightarrow$ **H**) in a block of P_1's cache from Figure 21.

In WIP, the whole valid block is reloaded into a block of P_1's cache from P_3's cache (the owner of the block) before the write takes place. The write is delayed until an invalidation signal can be sent on the bus to invalidate the word in block of all other caches with the same word. Therefore, the block state in P_3's cache is changed from MOD-SHD to IW1. The block state in P_2's cache is changed from IW1 to IW2. The block state in P_1's cache is changed from IW1 to MOD-SHD after the word is changed from "**Y**" to "**H**". P_1's cache is the new owner for that block.

In HWRP, the Snoop Controllers of P_1's cache and P_2's cache catch the entire valid block from the bus when the Snoop Controllers of P_1's cache and P_2's cache detect a read bus operation for the block's address. And then the entire valid block is reloaded into the block of P_1's cache and the block of P_2's cache from P_3's cache (the owner of the block) before the write takes place. The write is delayed until an invalidation signal can be sent on the bus to invalidate the word in block of all other caches with the same word. Therefore, the block state in P_3's cache is changed from MOD-SHD to IW1. The block state in P_2's cache is changed from UNMOD-SHD to IW1. The block state in P_1's cache is changed from IW1 to MOD-SHD after the word is updated from "**Y**" to "**H**". The P_1's cache is the new owner for that block.

Figure 22 and Figure 23 show the difference in cache configurations between WIP and HWRP on a write hit of a block in the state IW1.

WIP

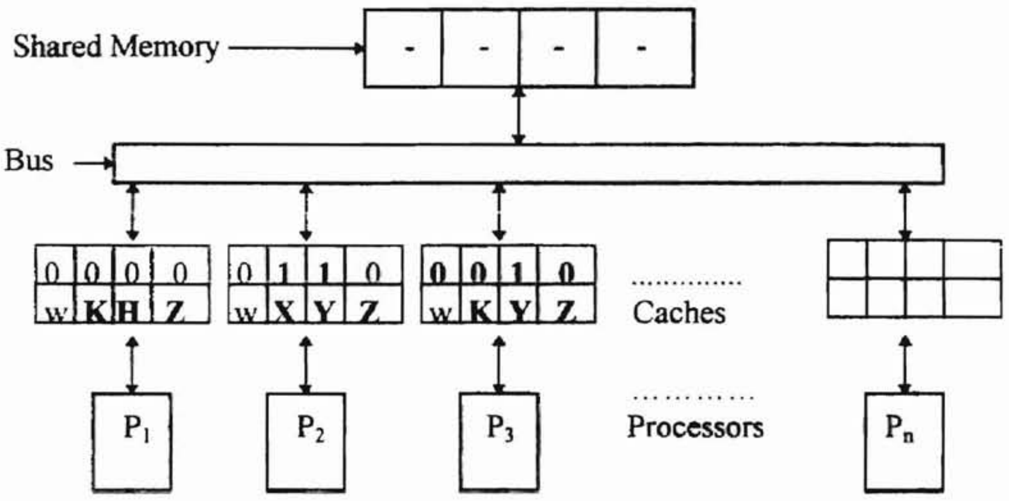

Figure 22. Cache configuration after the write hit by P_1 on the block in the IW1 state and then P_1 updates from a word "**Y**" to a word "**H**" in the block of P_1's cache. The block state P_1's cache is changed from IW1 to MOD-SHD. The block state in P_2's cache is changed from IW1 to IW2. The block state in P_3's cache is changed from MOD-SHD to IW1 .

HWRP

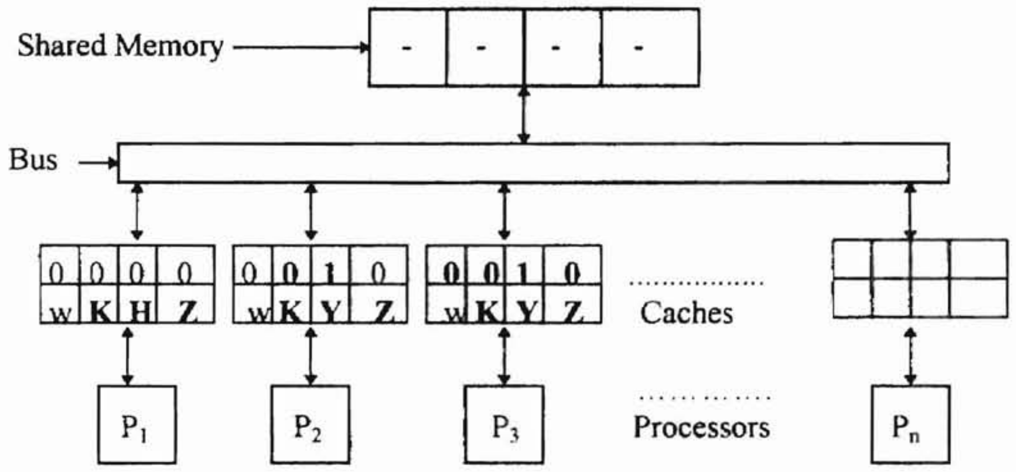

Figure 23. Cache configuration after the write hit by P_1 on the block in the IW1 state and then P_1 updates from a word "**Y**" to a word "**H**" in the block of P_1's cache. The block state in P_1's cache is changed from IW1to MOD-SHD. The block state in P_2's cache is changed from UNMOD-SHD to IW1. The block state in P_3's cache is changed from MOD-SHD to IW1.

Write Hit on IW2 State

There is cache configuration difference between WIP and HWRP on a write hit of a block in the IW2 state. Assume that P_3 tries to update a valid word ($z \rightarrow \mathbf{M}$) in a block of P_3's cache from Figure 14.

In WIP, the entire valid block is reloaded into only a block of P_3's cache from P_1's cache (the owner of the block) before the write takes place. The write is delayed until an invalidation signal can be sent on the bus to invalidate the word in block of all other caches with the same word. Therefore, the block state in P_1's cache is changed from MOD-SHD to IW1. The block state in P_2's cache is changed from IW2 to INV. The block state in P_3's cache is changed from IW2 to MOD-SHD after the word is changed from "z" to "**M**". P_3's cache is the new owner for that block.

In HWRP, the Snoop Controllers of P_2's cache and P_3's cache catch the entire valid block from the bus when the Snoop Controllers of P_2's cache and P_3's cache detect a read bus operation for the block's address. And then the whole valid block is reloaded into a block of P_2's cache and P_3's cache from P_1's cache (the owner of the block) before the write takes place. On a miss, a block must be chosen for replacement. If the chosen block is owned, then it is written to memory. The requested block is then read in UNMOD-EXC state and is updated. The final state of the entry becomes MOD-EXC. Therefore, the block state in P_1's cache is changed from MOD-SHD to IW1. The block state in P_2's cache is changed from UNMOD-SHD to IW1. The block state in P_3's cache is changed from IW2 to MOD-SHD after the word is changed from "z" to "**M**". P_3's cache is the new owner for that block. Figure 24 and Figure 25 show different cache configurations in the WIP and in HWRP on write hit of a block in the IW2 state.

WIP on Write Hit

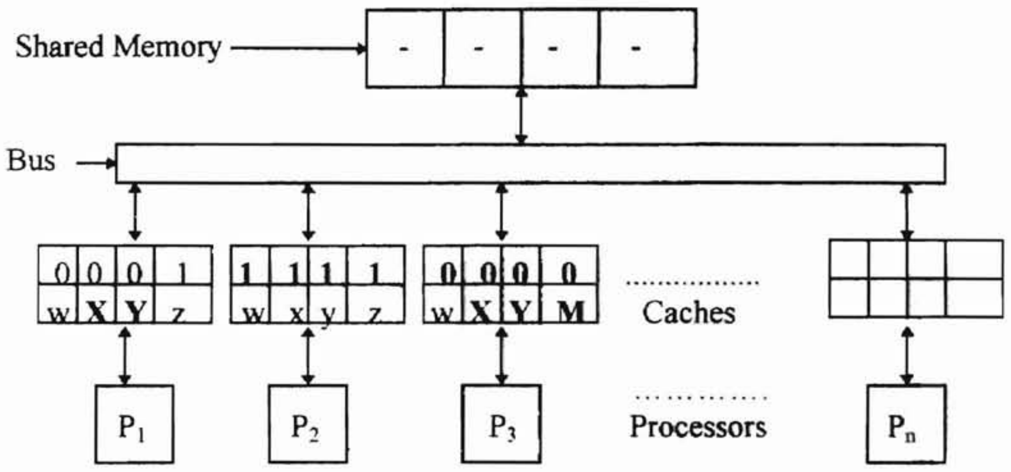

Figure 24. Cache configuration after the write hit by P_3 on the block in the IW2 state and then P_3 updates from "z" to "**M**" in the block of P_3's cache The block state in P_1's cache is changed from MOD-SHD to IW1. The block state in P_2's cache is changed from IW2 to INV. The block state in P_3's cache is changed from IW2 to MOD-SHD.

HWRP on Write Hit

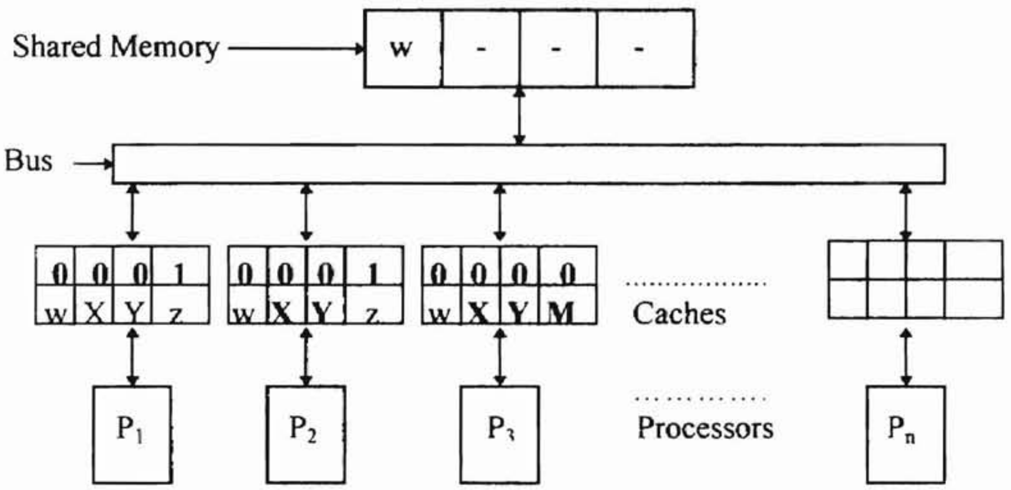

Figure 25. Cache configuration after the write hit by P_3 on the block in the IW2 state and then P_3 updates from "z" to "**M**" in the block of P_3's cache The block state in P_1's cache is changed from MOD-SHD to IW1. The block state in P_2's cache is changed from UNMOD-SHD to IW1. The block state in P_3's cache is changed from IW2 to MOD-SHD.

In order to assess the effect of HWRP on invalidation misses, we have performed a simulation study. HWRP is compared against WIP.

4. SIMULATION MODEL

In shared-memory multiprocessor systems, the write-invalidate protocols should pay a high cache miss penalty due to invalidation misses necessitated by maintaining cache coherence. Therefore, the reduction in invalidation misses is a significant factor to get higher performance because the reduction in invalidation misses produces a corresponding decline in the cache miss ratio. The simulation presented here is designed with these factors taken into consideration.

In order to simulate HWRP and WIP, we use a simulation model driven by synthetic workload model to obtain quantitative measures rather than by actual traces. The actual traces could be created, but they would be as artificial as the method that we have employed [2]. Although synthetic traces are artificial in nature, sometimes they can be more useful than real traces [21]. Carefully varying appropriate parameters in a flexible synthetic model is a more convenient way to evaluate the performance of simulated solutions than real traces witch are influenced by the particular conditions under which they are collected. The first step in the simulation model is the definition of a basic multiprocessor model.

4.1. Multiprocessor Model

The simulated multiprocessor model is organized into two main modules: processor module and bus module. The bus module is unique in the system and contains only one process, while the processor module is replicated according to the size of the multiprocessor system. The processor module has three main processes: a process for each processor, a cache controller process, and a snoop controller process. Figure 26 shows a diagram of the simulated multiprocessor model.

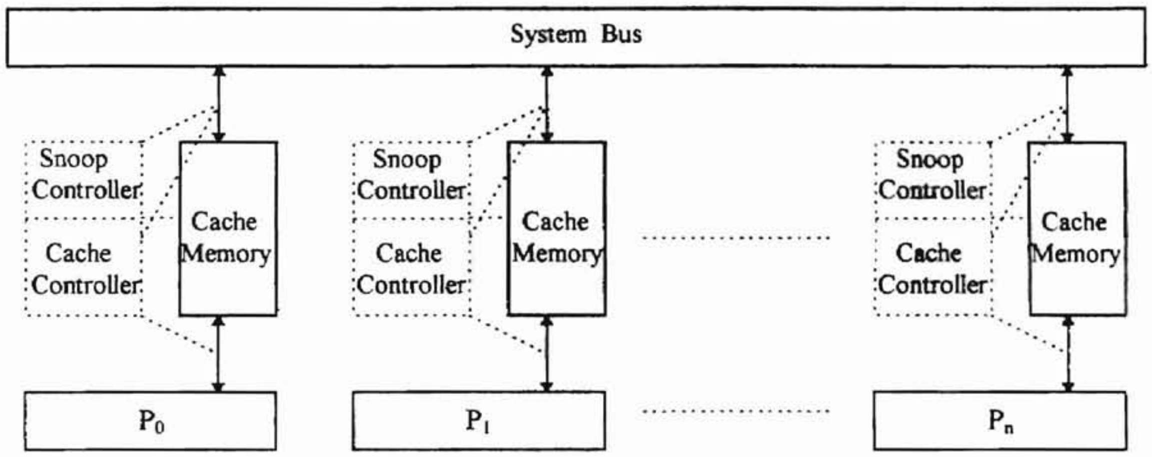

Figure 26. A diagram of multiprocessor model
Data lines are solid and control lines are broken.

When a processor generates a memory request, it sends the memory request to the cache controller process. The cache services a memory request from its processor by determining whether the requested block is present or absent. If the requested block is present in the cache, the request can be serviced without a bus transaction. If so, the cache sends the processor a command to continue. If a bus transaction is required, a bus request is generated and inserted into the service queue of the bus. The cache sends the processor a command to continue only upon completion of the bus transaction.

The cache can also receive commands from the bus process relating to actions that must be performed on blocks of which it has copies. Such commands are detected through the snoop controller process and have higher priority for service by the cache than processor memory requests. In a multiprocessor system, this is equivalent to matching a block address on a bus transaction and halting the service of processor requests to take action. After that action is completed, the cache is free to respond to processor requests.

A more detailed description of the cache controller process, the snoop controller process, and the bus process is provided in following paragraphs.

4.1.1 The Cache Controller Process

The cache controller's behavior depends on its processor's request, whether the data is in the cache, and the state of the cache entry on a hit. When a *processor read* results in a cache hit, the appropriate word is provided to the processor. Upon a miss, a miss request is broadcast through a bus to all caches and to main memory. If a missed block is in the INV state, it will be obtained from the cache-owner, if it exists, or from memory. The block will be loaded in one of the unmodified states (UNMOD-EXC[1] or UNMOD-SHD[2]), depending on the owner of the missed block.

The procedure for *a processor write* to a block in the cache is as follows. The write can be performed locally without access to the bus, if the block is in one of the exclusive states (UNMOD-EXC or MOD-EXC). For both cases, the final state is MOD-EXC. If the state of the hit block is IW1, IW2, UNMOD-SHD or MOD-SHD, then the cache controller must issue an invalidation request on the bus to invalidate the word in block of all other caches with the same word. IW1, IW2 or INV state indicates that the snoop controller invalidates the block in response to detecting an invalidation request from another processor, after the cache controller had initially detected a hit.

[1]If the missed block is supplied by main memory, then the block will be loaded in UNMOD-EXC state.
[2]If the missed block is supplied by cache-owner, then the block will be loaded in UNMOD-SHD state.

4.1.2 The Snoop Controller Process

The snoop controller process monitors the bus for *a bus read request*, *a bus write request* and *an invalidation request*. If the snoop controller processors of all other caches except a requesting cache detect an invalidation request from the bus, then they access their own cache memory to invalidate the word in block of their own caches. If the snoop controller processor of an owner cache observes a read request from the bus, then it accesses its own cache memory to provide an owned block for a bus read request. Moreover, when the snoop controller processors of all other caches with the same block detect a read bus operation for the block's address, they accesses their own cache memory to update invalidated data or an entire invalid block with data from the bus,

After updating an invalidated word in a block with data from the bus, if the block state is IW1, the snoop controller processors of the updated word or the updated block access their own cache memory to change the block's state to UNMOD-SHD. If the block state is IW2, the snoop controller processor of all other caches accesses its cache memory to change the block's state to IW1. If the block state is INV, the snoop controller process of all other caches accesses its cache memory to change the block's state to UNMOD-SHD or UNMOD-EXC after reloading an entire valid block from the bus. The Snoop's actions are a function of the system bus request, whether it hits or misses in its cache, and the state of the block.

4.1.3 The Bus Process

The bus process receives service requests of five types (*read miss, write hit, write miss, invalidation miss and invalidation signal*) from all caches. The cache controller process generates one of the five types of requests to the bus process, which serves the incoming requests in the order of arrival. Information about ongoing bus transaction is sent to all snoop controller processes.

We use the communication cost per memory reference as our basic metric. This cost is the number of cycles that the bus is busy during serving one of the five types of requests. We refer to this metric as bus-cycles-per-memory reference. The bus cycle costs per reference depend on the five different types of requests. The bus cycle costs used in the simulation model are adopted from the examples considered in [2]. The costs related to bus transaction are summarized in Table 3 and Table 4.

Table 3. Timing for Fundamental Bus Operations

Bus Operation	**Bus Cycles**
Send address	1
Transfer 1 data word	1
Invalidate	1
Wait for Memory	2
Wait for Cache	1

Table 4. Summary of Bus Cycle Costs

Access Type	**Total Bus Cycle Costs**
Memory access	7
Cache access	6
Write back	4
Invalidate	1
Write update to another cache	2

In the simulated multiprocessor model, a memory access costs 7 cycles, 1 cycle to send the address, 2 cycles to wait for the memory access, and 4 cycles to get four words. An access from another cache is 6 cycles, and takes a cycle less than the memory access, because the cache access wait is only one cycle. Write-back costs 4 cycles. While the write into memory is taking place, the bus need not be held. A write update to other caches requires 2 bus cycles, 1 cycle to send an address and 1 cycle to update an invalidated word. Invalidations cost one cycle. The data transfer width of bus is assumed to be one word (32 bits).

4.2 Workload Model

The choice of workload model is a critical point because the performance of cache coherence protocols heavily depend on the characteristics of the workload. The workload model selected is similar to one developed in [3,21]. The simulation parameters and ranges used are summarized in Table 5 on the next page.

The memory reference stream of each processor is divided into two distinct classes: reference stream to private blocks and reference stream to shared block. Each time a memory reference is called for, the processor generates a reference to a shared block with probability *shared* and a reference to a private block is generated with probability 1 - *shared.* Similarly, the probability that the reference is a read is *read* and the probability that it is a write is 1 - *read.* If the request is to a private block, it is a hit with probability *hit* and a miss with probability 1 - *hit.*

With probability (1-*shared*), references to private blocks do not affect cache coherence. The most important parameter is hit ratio for private blocks. Also, they do not create invalidation traffic, nor do they degrade the hit ratio of the other caches.

With probability *shared*, the reference is for an shared block. A reference to a shared block i is made according to a probability distribution p_i [3]for $i = 1, \ldots, N_s$

[3]p_i = 1 / the number of shared blocks(N_s).

. Therefore, the probability that a reference is a write on shared block *i is shared* * p_i * (1 - *read*).

Table 5. Summary of Parameters and Ranges

Parameters	**Ranges**
Probability of shared references (*shared*)	2% - 5 %
Read probability(*read*)	70% - 85 %
Hit ratio for private blocks(*hit*)	95% - 98%
Word size	Four bytes
Block size	Four words
Cache size	2 - 10 Kbytes
Memory Mapping Method	Fully Associative
Number of private blocks(N_p)	1024
Number of shared blocks(N_s)	16- 64
Number of processors	2-32
Number of references per processor	10000

The parameters and ranges shown in table 5 are adopted from the examples mentioned in [3,21]. All references to shared blocks in our model include a block number generated by a pseudorandom number generator. To service a shared block request, the cache determines from a directory whether the requested block is present or absent, and whether a bus request must be generated.

If a cache miss occurs, either for a shared block or for a private block, a block must be ejected to make room for the new block. The probability that a shared block is selected is equal to the percentage of blocks in the cache that are shared blocks at that point in time. If the selected block is private and is modified, it needs to be written back to main memory. If a shared block is chosen for replacement, one of those present in the cache is chosen at random. The state of that particular block determines whether or not it is to be written back.

5. DISCUSSION OF SIMULATION RESULTS

In this section, we analyze the behavior of each protocol to demonstrate the effect of various parameters on the cache miss ratio and bus traffic. Output from the simulation includes some figures as the results of the simulation we have run. Each figure shows the result obtained with the indicated parameter values for both schemes (WIP and HWRP) from two to thirty two processors. The ratio of invalidation misses[4] as the result of the simulation is a significant factor for comparison between WIP and HWRP because the objective of this research is to reduce additional invalidation misses caused by invalidation in write-invalidate protocols.

5.1 Impact on Miss Ratios with Varying Parameters

Figure 27 and Figure 28 demonstrate that the ratio of invalidation misses and total miss ratio for both protocols increase as the write ratio increases. The increase is due to the increased number of invalidations because of more writers to shared blocks. The increased number of invalidation is responsible for a subsequent rise in invalidation misses[7]. The total miss ratio of each protocol increases as much as the increased proportion of invalidation misses within total misses. From Figure 27, we see that HWRP has a lower invalidation miss ratio than the invalidation miss ratio of WIP. Consequently, HWRP has a lower total miss ratio.

[4]The invalidation miss ratio is the invalidation misses divided by total number of cache misses.

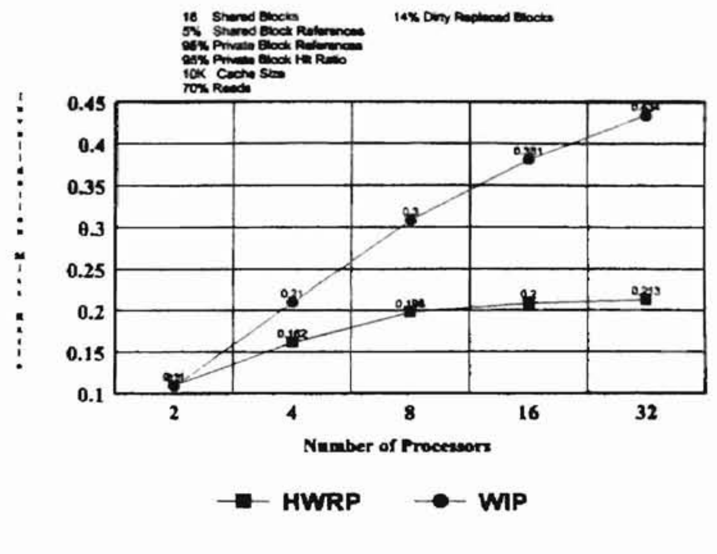

Figure 27. Ratio of Invalidation Misses for Both Protocols

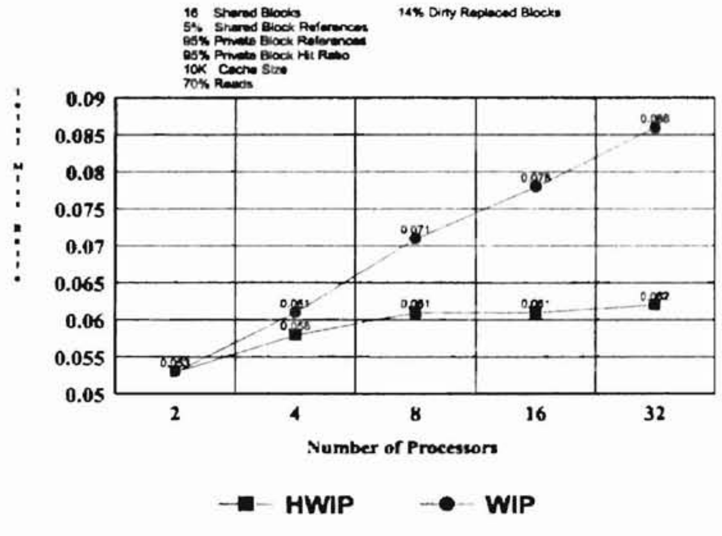

Figure 28. Ratio of Total Misses for Both Protocols

Figures 29 through 32 illustrate different invalidation miss ratios and shared miss ratios[5]for both protocols to test the impact of handling shared blocks efficiently by changing only a number of shared blocks. On the invalidation miss ratio, Figure 29 and Figure 31 show that both protocols have a higher invalidation miss ratio at a tighter sharing (32 shared blocks) than invalidation miss ratio at a looser sharing (64 shared blocks). At tighter sharing, the number of processors contending for a shared block address is relatively high. Therefore, the shared data has a higher probability to be referenced or to be invalidated, and consequently, is referenced via invalidation misses.

On the shared miss ratio, Figure 30 and Figure 32 show that both protocols have a lower shared miss ratio at a tighter sharing than shared miss ratio at a looser sharing. At looser sharing (64 shared blocks), the cache has a lower probability that a shared block is referenced by its own cache or by the other caches. Therefore, there are fewer cache hits on the shared blocks because each shared block is not accessed very often.

From these four figures, we see that the read broadcast approach in HWRP yields a lower miss ratio in handing of shared data, since the read broadcast approach in HWRP leads to preserving the valid, frequently shared data.

[5]The shared miss ratio is the number of misses to shared data divided by the total references to shared data.

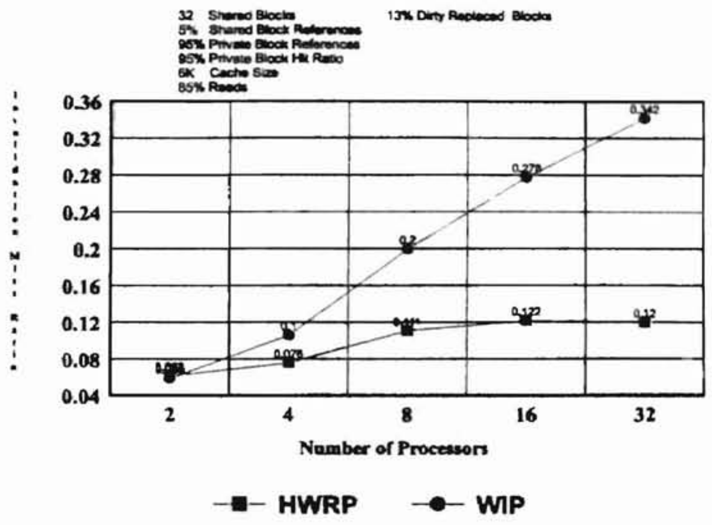

Figure 29. Ratio of Invalidation Misses for Both Protocols

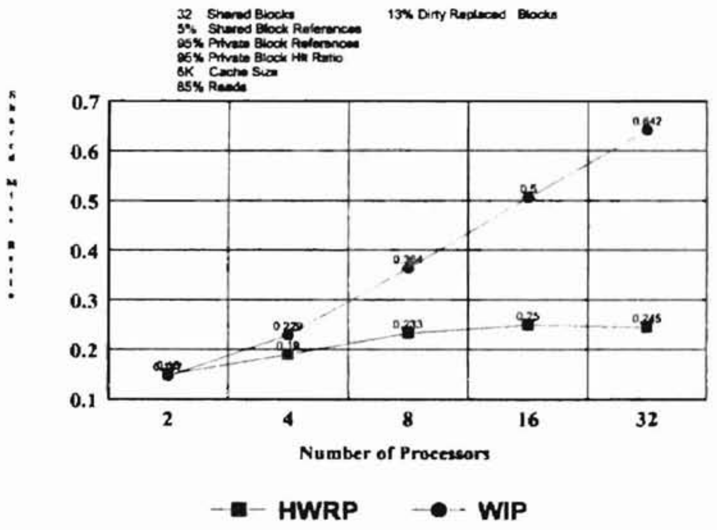

Figure 30. Ratio of Shared Misses for Both Protocols

invalidation miss ratio of WIP, even though the invalidation miss ratio of each protocol increases slightly as cache size increases.

The number of invalidation misses for both protocols is inversely proportional to the number of block replacements. At small cache sizes, the number of block replacements is relatively high. As cache size increases, the percentage of block replacements drops. Shared data tends to remain in the cache for a longer period of time, has more opportunity to be invalidated, and, consequently, is referenced via invalidation misses. The number of invalidation misses should be higher with each successively larger cache, approximately by the percentage decrease in block replacements. Note that the greater the number of processors contending for an address, the greater the number of invalidation misses [7]. Consequently, the lower invalidation miss ratio in HWRP is due to improved cache hits because of the approach updating invalidated blocks, which leads to preserving the valid, frequently shared data.

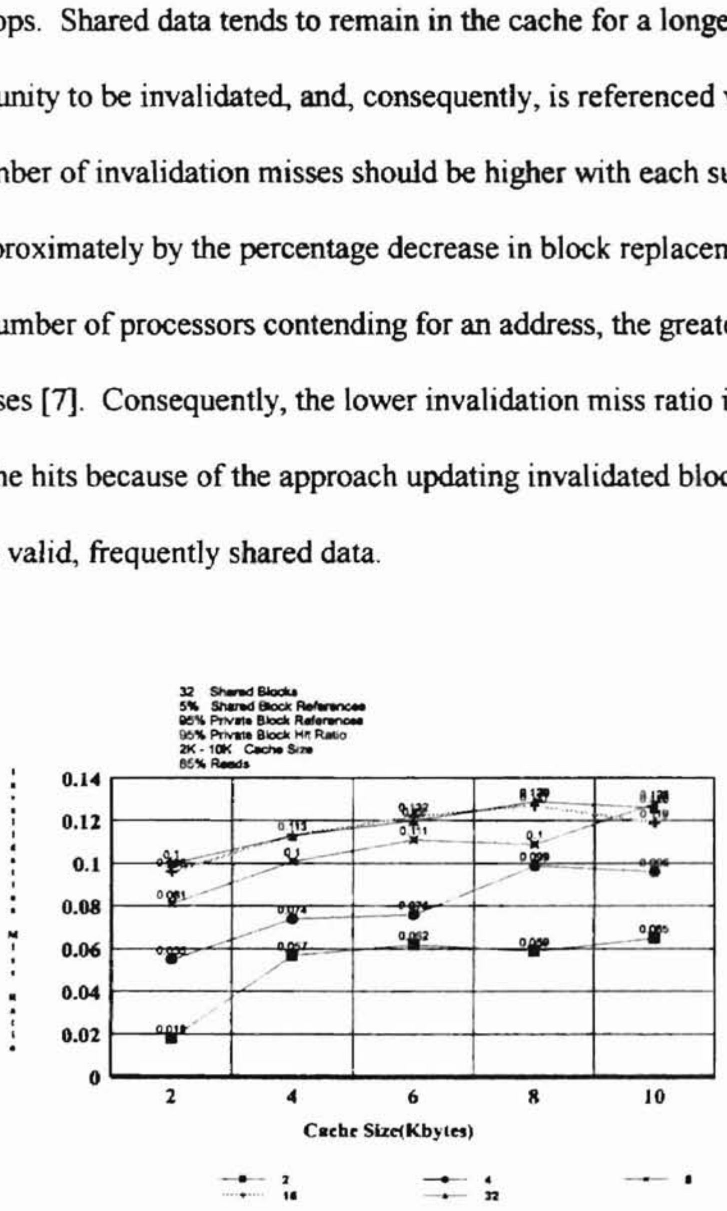

Figure 33. Ratio of Invalidation Misses for HWRP

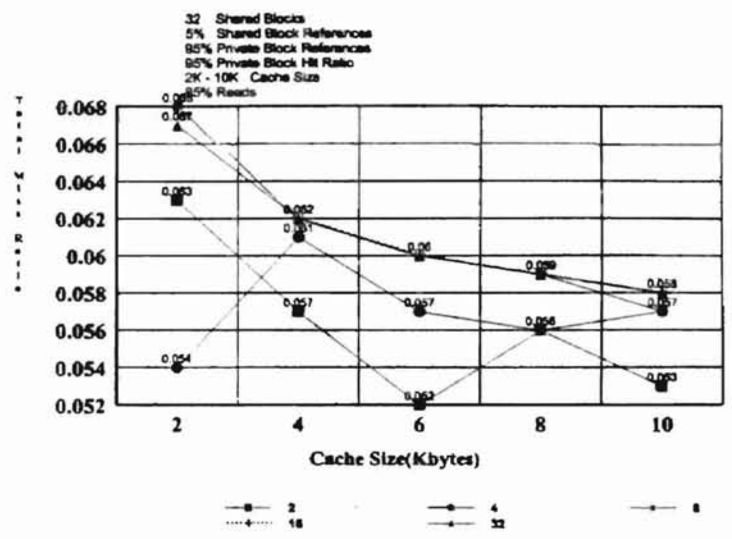

Figure 34. Total Miss Ratio for HWRP

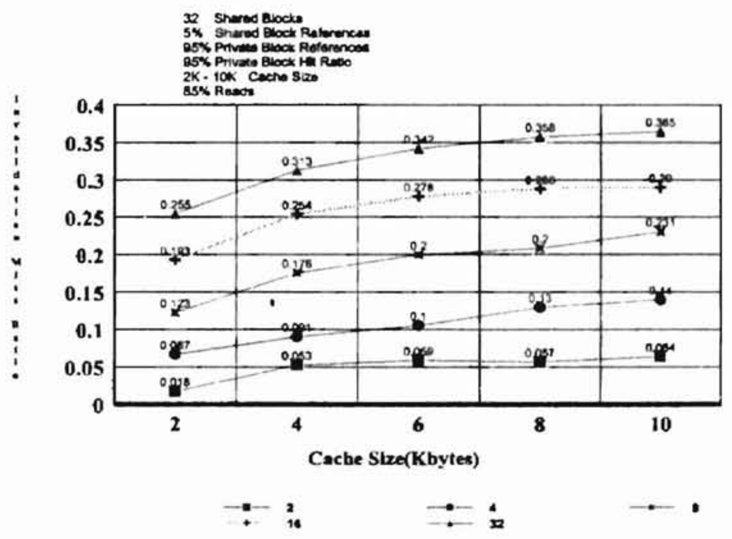

Figure 35. Ratio of Invalidation Misses for WIP

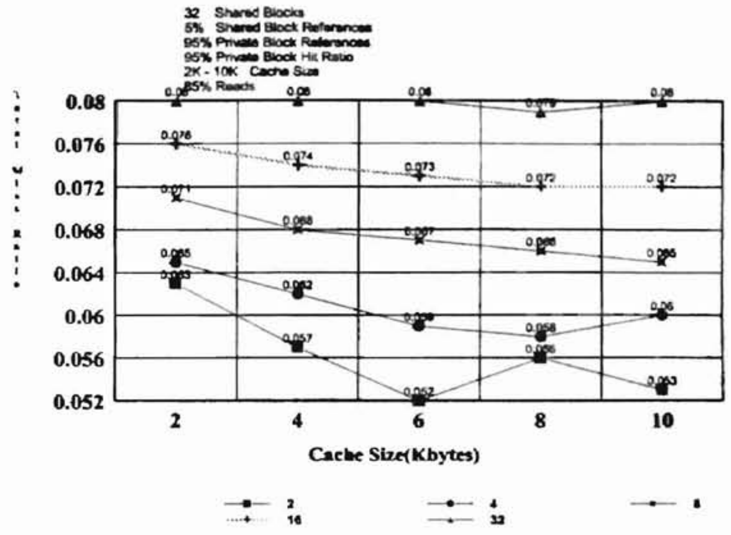

Figure 36. Total Miss Ratio for WIP

Figure 34 and Figure 36 show the total cache miss ratio for each protocol. We see that HWRP has a lower total cache miss ratio as much as the proportion of reduction in invalidation misses than WIP has. In HWRP, this reduction in invalidation misses is contributed by the read broadcast [16] mechanism which updates an invalidated block or an invalidated word, when snoop controllers detect a read or write bus operation for the block's address.

5.3 Bus Utilization

The critical system bottleneck in a single-bus, shared memory multiprocessor is the bandwidth of the system bus [7]. Write-invalidate protocols have two main sources of bus-related coherency overhead. The first is the invalidation request of shared data in each cache. The second is the invalidation misses caused by invalidation request. Consequently, reducing the number of invalidation misses produces a corresponding decline in the bus traffic for write-invalidate protocols. As discussed in the previous section, we know that HWRP always has a lower total cache-miss ratio resulting from reduction in invalidation misses than WIP has. Figure 37 and Figure 38 show that HWRP has a lower number of bus cycles for bus operation since read misses and invalidation misses for shared blocks are relatively infrequent.

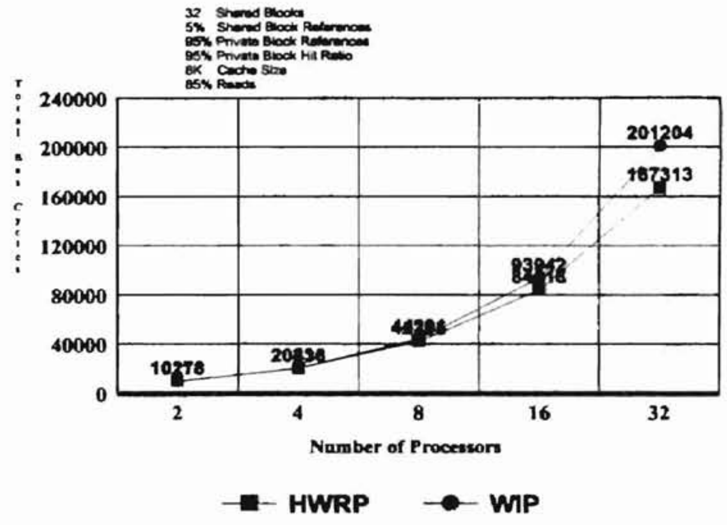

Figure 37. Total Bus Cycles

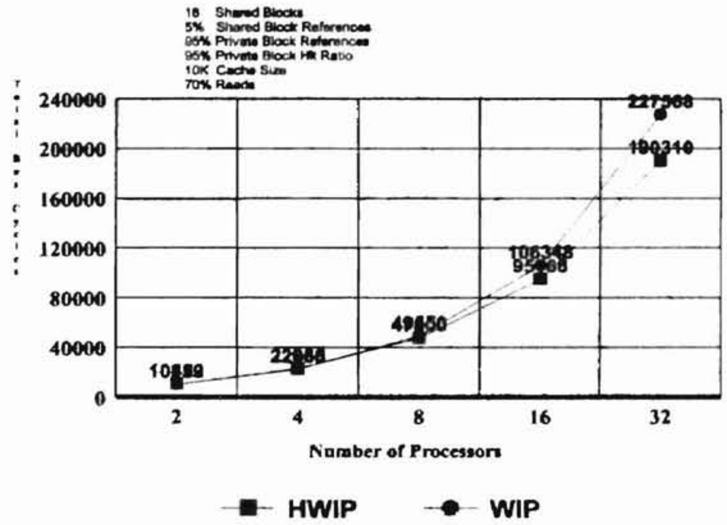

Figure 38. Total Bus Cycles

On the contrary, the number of bus cycles in WIP is higher because of a larger percentage of invalidation misses. As mentioned in [2], the cost of a write update is assumed to be much lower than the cost of an invalidation and a subsequent miss. On the bus utilization, the most important consequence of HWRP is the effect of its lower miss ratio.

6. CONCLUSION

Since the purpose of a cache is to speed up access to data, cache misses are the main hindrance for obtaining better performance in cache memory system. In shared-memory multiprocessor system, the write-invalidate protocols should pay a high cache miss penalty due to invalidation misses necessitated by maintaining cache coherence. Since invalidation misses play such a large role in caches and bus performance, coherency protocols that can reduce them are desirable. In this thesis, we presented the Hybrid Word Invalidate/Read Broadcast Protocol for more reduction in invalidation misses. We have studied the effects of the cache coherency on the miss ratios of both protocols (HWRP and WIP) and on the bus traffic between the caches. Through some experiments, we demonstrated that HWRP has a lower invalidation miss ratio than WIP. In HWRP, the reduction in invalidation misses produces a corresponding decline in total miss ratio and bus utilization. Consequently, eliminating invalidation misses leads to a potentially better utilization of data already fetched in the cache and achieves a higher hit ratio. Therefore, the solution proposed in this thesis can be expected to improve performance as compared with other write-invalidate protocols.

REFERENCES

[1] A. Agarwal. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280-289, 1988.

[2] A. Agarwal, R. Simoni, J. Hennessy and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280-289, 1988.

[3] J. Archibald and J. L. Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273-298, November 1986.

[4] M. L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112-1118, December 1978.

[5] D. Chaiken, C. Fields, K. Kurihara and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, 23(6):49-58, June 1990.

[6] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 2-15, 1989.

[7] S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs, In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257-270, 1989.

[8] S. J. Eggers and R. H. Katz. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276-283, 1985.

[9] D. Fredrik and P. Stenström. Using write caches to improve performance of cache coherence protocols in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 26(2):193-210, April 1995.

[10] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124-131, 1983.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach*. San Mateo, California: Morgan Kaufmann Publishers Inc., 1993.

[12] D. J. Lilja and C. P. Yew. Improving memory utilization in cache coherence directories. *IEEE Transactions on Parallel and Distributed Systems*, 4(10):1130-1146, October 1993.

[13] D. J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons. *ACM Computing Surveys*, 25(3): 303-338, September 1993.

[14] M. E. McCreight. The dragon computer system, an early overview, In *Proceedings of the NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, Urbino, Italy, July 1984.

[15] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the* 11th *Annual International Symposium on Computer Architecture*, pages 348-354, 1984.

[16] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340-347, 1984.

[17] A. J. Smith. Design of CPU cache memories. In *Proceedings of IEEE TENCON'87*, pages 30.2.1-30.2.10, 1987.

[18] P. Stenström. A cache consistency protocol for multiprocessors with multistage networks. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 407-415, 1989.

[19] P. Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12-24, June 1990.

[20] C. P. Thacker and L. C. Stewart. Firefly : a multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909-920, August 1988.

[21] M. Tomašević and V. Milutinović. A simulation study of snoopy cache coherence protocols. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, pages 427-436, 1992.

[22] M. Tomašević and V. Milutinović. *The Cache Coherence Problem in Shared Memory Multiprocessors : Hardware Solutions*. Los Alamitos, California: IEEE Computer Society Press. 1993.

VITA

In-Suk Chung

Candidate for the Degree of

Master of Science

Thesis: A SIMULATION STUDY OF SNOOPY CACHE COHERENCE PROTOCOLS

Major Field: Computer Science

Biographical Data:

Personal Data: Born in Seoul, Korea on September 23, 1964, the son of Lee-June Chung and Boon-Ok Kim

Education: Graduated from Hwan Il High School, Seoul, Korea, 1983; received Bachelor of Science in Computer Science from Oklahoma State University, Stillwater, Oklahoma in 1993. Completed the requirements for the Master of Science degree in Computer Science at Oklahoma State University in May 1996.