DESIGN AND VALIDATION OF UPDATE-BASED CACHE COHERENCE PROTOCOLS

David B. Glasco
Bruce A. Delagi
Michael J. Flynn

Technical Report No. CSL-TR-94-613

March 1994

This work was supported by NASA under contract NAG2-248 using facilities provided by Sun Microsystems, Inc.

DESIGN AND VALIDATION OF UPDATE-BASED CACHE COHERENCE PROTOCOLS

by

David B. Glasco

Bruce A. Delagi

Michael J. Flynn

Technical Report No. CSL-TR-94-613

March 1994

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305

Abstract

In this paper, we present the details of the two update-based cache coherence protocols for scalable shared-memory multiprocessors that were studied in our previous work. First, the directory structures required for the protocols are briefly reviewed. Next, the state diagrams and some examples of the two update-based protocols are presented; one of the protocols is based on a centralized directory, and the other is based on a singly-linked distributed directory. Protocol deadlock and the additional requirements placed the protocols to avoid such deadlock are also examined. Finally, protocol verification using an exhaustive verification tool known as Murφ is discussed.

Key Words and Phrases: Update-based cache coherence protocols, hardware-based cache coherence protocols, shared-memory multiprocessors

Contents

1 Introduction

In our previous work [8, 7, 9], we demonstrated the possible performance gains from update-based cache coherence protocols when compared to invalidate-based protocols for a set of fine-grain scientific applications running on a scalable shared-memory multiprocessor. In this paper, we present the details of the update-based cache coherence protocols and discuss verification of the protocols.

The paper is organized as follows. Section 2 gives a brief review of the directory structures required for the protocols, and section 3 discusses protocol level deadlock. Next, section 4 presents the details of the update-based protocols. In particular, section 4.1 describes a centralized directory protocol, and section 4.2 describes a singly-linked distributed directory protocol. Section 5 discusses verification of the protocols using an exhaustive verification tool called Murφ [5]. And finally, section 6 summarizes the paper.

2 Directory Structures

Directory-based cache coherence protocols must maintain a directory entry for each memory line in the system. This directory entry indicates which caches in the system have a copy of the respective memory line. Each directory entry can be stored in a single, central location (centralized directory protocol) or distributed among the caches holding a copy of the line (distributed directory protocol). In both cases, the directory entries are distributed throughout the system with their respective memory lines.

2.1 Centralized Directory

In a centralized directory (CD) protocol, each directory entry contains a pointer to each cache in the system that contains a copy of the respective memory line. In the CD protocols studied in this work, a fully mapped directory is used in which there is a single bit pointer for each cache in the system [18]. For example, figure 1 shows a directory entry for a memory line in a four cache system. In the example, caches 1 and 3 have a copy of the given memory line.

In this fully mapped scheme, each directory entry contains N_{Caches} bits for a total of

$$\begin{aligned} Bits &= N_{Caches} * N_{MemoryLines} \\ &= O(N_{Caches} N_{MemoryLines}) \end{aligned}$$

bits where N_{Caches} is the number of caches and $N_{MemoryLines}$ is the number of memory lines in the system.

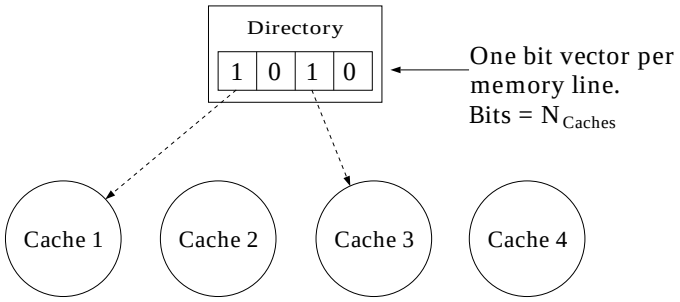


Figure 1: Centralized Directory Structure

2.2 Distributed Directory

In a distributed directory (DD) protocol, a linked list structure is used to maintain the list of caches that have a copy of a given memory line. The directory entry contains a pointer to the head of this list, and each cache line contains the necessary pointers to construct the list. The list may be singly-linked or doubly-linked. It is important to note that the order of the list is not optimized in any way. The order is determined by the order that the requests for the memory line reach the directory.

2.2.1 Singly-Linked Directory Structures

In a singly-linked distributed directory protocol [20], a singly-linked list is used to maintain the list as shown in figure 2. In this example, caches 0, 2 and 3 have a copy of the line.

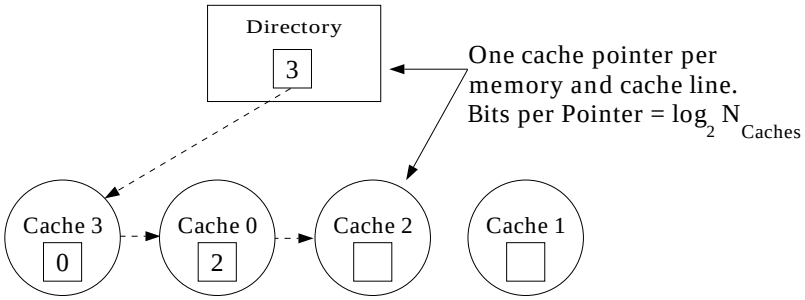


Figure 2: Singly-Linked Distributed Directory Structure

In this case, each directory entry contains $log_2(N_{Caches})$ bits, and each cache line must also include a single pointer. This requires a total of

$$\begin{aligned} Bits &= N_{MemoryLines} log_2(N_{Caches}) + N_{CacheLines} log_2(N_{Caches}) \\ &= log_2(N_{Caches})(N_{MemoryLines} + N_{CacheLines}) \end{aligned}$$

$$= O(log_2(N_{Caches})N_{MemoryLines})$$

bits, which scales better than the fully-mapped CD directory structure as the size of the system increases.

2.2.2 Doubly-Linked Directory Structures

Alternatively, a doubly-linked directory structure may be used [11], as shown in figure 3. In this example, caches 0, 2 and 3 have a copy of the line.

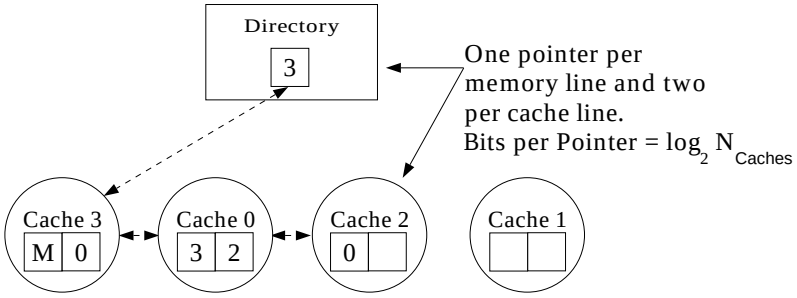


Figure 3: Doubly-Linked Distributed Directory Structure

The amount of storage required is slightly more than that of the singly-linked distributed directory structure since each cache line must now maintain two pointers. This requires

$$\begin{aligned} Bits &= N_{MemoryLines} log_2(N_{Caches}) + 2N_{CacheLines} log_2(N_{Caches}) \\ &= log_2(N_{Caches})(N_{MemoryLines} + 2N_{CacheLines}) \\ &= O(Log_2(N_{Caches})N_{MemoryLines}) \end{aligned}$$

bits of storage.

2.3 Scalability of Directory Structures

As shown above, the centralized directory structure scales as $O(N_{Caches}N_{MemoryLines})$, but the distributed directories scale better as $O(log_2(N_{Caches})N_{MemoryLines})$. However, several different approaches have been suggested to improve the scalability of the centralized directory schemes. These include limited pointer schemes and cached directories.

The limited pointer schemes limit the number of cached copies of each memory line. When this limit is exceeded, the limited pointer schemes either invalidates one of the copies to make room for the new request [1], assumes all caches now have a copy of the line [1], switches to a coarse grain mode where each bit represents several caches [10] or traps to software to extend the directory list [3]. With a limited pointer scheme, the centralized directory scales as $O(N_{Limited}N_{MemoryLines})$ where $N_{Limited}$ is the number of bits in the limited directory entry.

The other approach is to note that the maximum number of cached copies of a memory line is limited by the total size of all caches and not the size of memory. In this case, a directory cache could be used to cache this smaller set of directory entries [10]. Also, the bits for each directory entry can be dynamically allocated out of a pool of directory bits [17].

Several studies have indicated that the average number of shared copies of a memory line is small [2, 16, 21, 1, 6]. Therefore, the limited directory schemes result in minimal performance loss [1, 3, 10]. Similarly, the cached directory has also been shown to have a minimal affect on execution time [10].

Some of the limited pointer schemes require that the directory be able to invalidate cached copies of a line. The centralized directory update-based protocol presented in this work currently does not support invalidations, but the protocol could be extended to support such directory initiated invalidations.

3 Protocol Deadlock

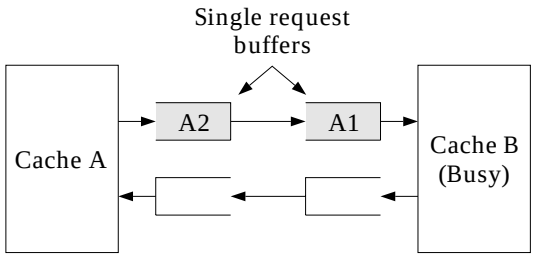
If the system has finite buffering, then protocol level deadlock is possible [19, 15, 8][1]. For example, figure 4 shows two caches that are sending requests to each other through a set of finite buffers. Each buffer can hold a single request. First, cache A sends two requests to cache B, and it begins processing a request that will generate another request to cache B. But because the buffers are already full, cache A must wait until a buffer becomes available before it is able to complete the processing of the new request. Meanwhile, cache B generates two requests to cache A, and it attempts to generate a third request. The system is now deadlocked. Neither cache A nor B can complete the processing of their current request because their output buffers are full, and they will never empty. A timeout is used to detect such deadlocked situations.

There are two techniques to handle protocol level deadlock. The first technique attempts to avoid deadlock by using local memory to expand the buffer space as needed [14]. When a buffer fills and a possible deadlock situation is detected, packets are removed from the buffer and placed in a secondary buffer created in the local memory. The cache and directory controllers must then process packets from this secondary buffer until it is empty. This technique essentially creates an almost infinitely sized buffer, but it requires a tight coupling of the cache controller, directory controller and local memory.

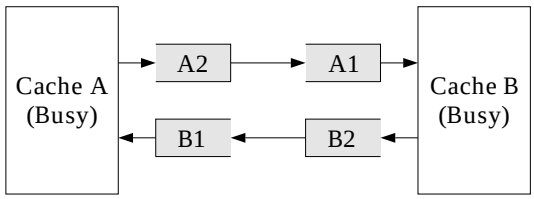
The second technique attempts to break the deadlock by removing requests from the deadlocked buffer and sending them back to their source through an exception network [15, 19, 8]. To minimize the probability of deadlock, messages are statically divided by the protocol into request and reply messages. A request message is a message that may generate another message and, therefore, lead to deadlock. A reply message is a message that never generates any new messages and, therefore, can always be consumed. This is the technique assumed throughout this work.

This second technique requires three logical networks: a request, reply and exception network. The reply network is deadlock free since replies can always be consumed. The request network

[1] The actual network is assumed to be deadlock free.



(a) Cache A sends two requests to Cache B which is currently busy. Cache A begins processing a request that will generate another request for Cache B.



(a) Cache A can not complete current request since the buffer is full. Meanwhile, Cache B generates two requests to Cache A and attempts to generate a third request. Neither cache can proceed since their output buffers are full. The system is deadlocked.

Figure 4: Protocol Level Deadlock

may experience deadlock if the request at the head of the buffer generates another request. If the request generates a reply, then the request can always be completed since the reply network will never deadlock. The request-request deadlock is broken by sending the request at the head of the deadlocked buffer back to the source of the request to be retried. The act of removing a message from the deadlocked network may break the deadlock. If not, this process would remove another request packet and send it back to the source. This is repeated until the deadlock condition is eliminated.

The frequency of deadlock is dependent on the size of the buffers. If reasonable buffer sizes are used, deadlock is extremely rare [19, 15]. In the system simulated in this work, the cache and memory buffers were 128 words deep, and deadlock never occurred for any of the cases examined [8, 7, 9].

The separate request and reply networks also require logically separate controllers in both the cache and directory. Otherwise, a cache that is attempting to process a request would not be able to consume pending replies. This would violate the condition that replies always be consumed. Also, since replies are always consumed, the reply network may also be used as the exception network.

4 Update-Based Protocol

This section presents the details of the update-based protocols. Since the scalability of the directory structures is still an open research topic, update-based protocols are presented for both a fully-mapped centralized directory and a singly-linked distributed directory. Both update-based protocols require an order preserving network. An order preserving network guarantees that messages between nodes are received in the same order that they are sent.

In the protocols examined in this work, each memory line may be in one of several states. These states usually specify if there is a single or multiple cached copies of the memory line and if memory's copy of the line is valid. Each cache line may also be in one of several states. These states specify the local processor's access privileges for the cached copy of the line. The state may also specify if the cache has the only cached copy of the memory line. In this section, the actions of the cache coherence protocols will be described using state transition diagrams for both the memory and cache line states. The state of the lines may change as a result of a request from the local processor, the directory or a remote cache.

4.1 Centralized Directory (CD-UP)

In this section, the details of the centralized directory update-based protocol (CD-UP) are presented.

4.1.1 Cache and Memory Line States

In the CD-UP protocol, a cache line can be in one of five states:

- *Invalid* - the cache's copy of the line is not valid
- *Pending* - the cache has issued a miss and is waiting for a reply
- *Shared* - the line is shared by multiple caches - writes must update other cached copies of the line and memory
- *Replacing* - the cache's copy is being replaced
- *Exclusive* - the cache's copy is the only copy of the memory line (cache is "owner" of line),

and a memory line can be in one of four states:

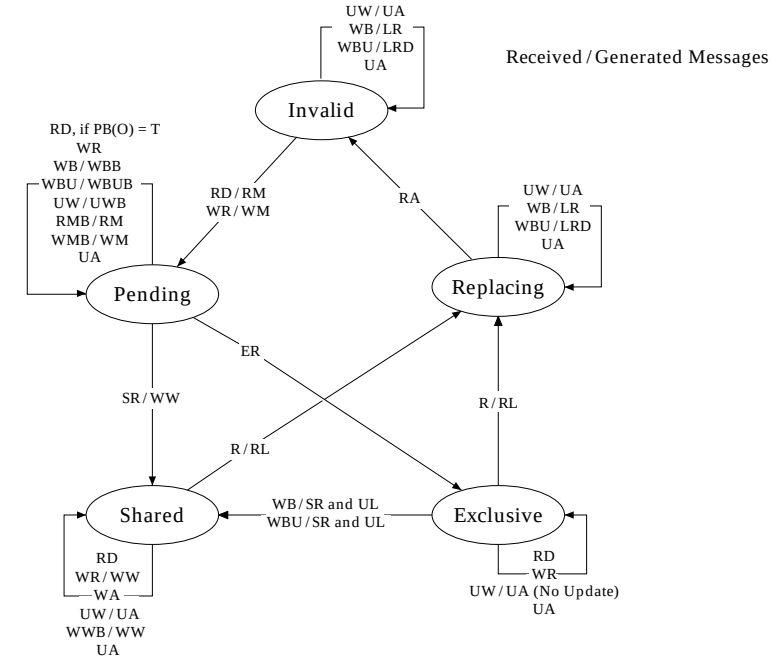
- *Absent* - no cached copies of the line exist
- *Shared* - at least one cache has a copy of the line - memory is consistent
- *Exclusive* - one cache has a copy of the line - memory is not consistent
- *Pending* - the directory is awaiting a write back for the memory line.

Table 1 gives a description of the protocol messages. The first column of the table gives the message name, and the next two columns give the source and destination type of the message. The source (Src) and destination (Dst) can be a processor (P), a cache (C) or a directory (D). The next column specifies if the message is a request (Req) or reply (Rep). This determines which network channel the message will traverse. The next column gives a brief description of the message, and the data column specifies what data type, a word or line, is sent with the message. Finally, the last column specifies what actions are taken by the destination node. These actions may include incrementing and decrementing the pending write counter (PWC) and the pending update counter (PUC). These counters are used to determined when all issued writes have been performed. Other actions include writing the data value (V) from the message into the cache line data (CD) or memory line data (MD) at the specified cache line offsets (O) and setting or clearing the directory (DIR) pointers. A Stall indicates that the processor is stalled until the proper reply is received by the cache, and Block indicates that the write is not processed, and it is not retired from the write buffer. A pending bit (PB) is used to control the writing of data into the cache line for certain conditions. The pending bits are identical to the valid bits in the invalidate-based protocols [19].

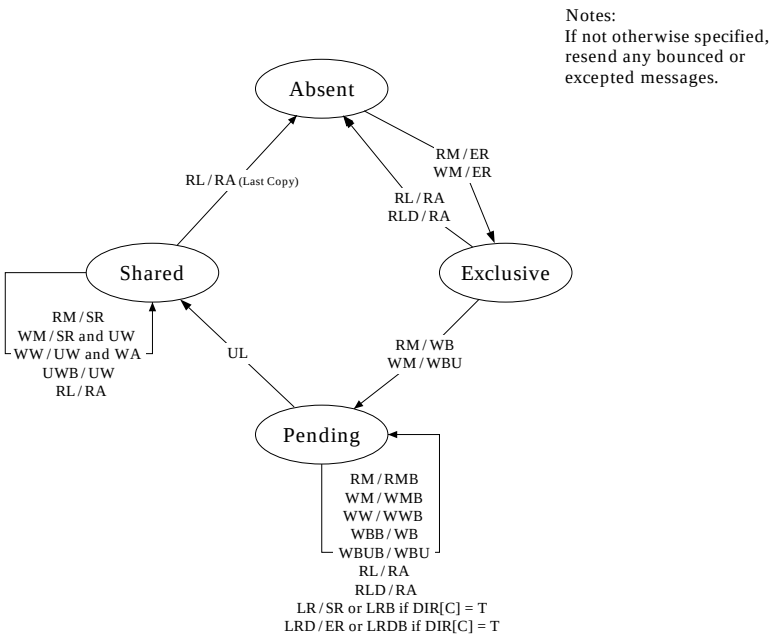
Figure 5 shows the state transition diagrams for cache and memory lines. The diagrams show the state changes for each received message and the resulting messages generated, if any.

Message	Src	Dst	Type	Description	Data	Destination Actions
RD	P	C		Processor read		If *Pending* & PB(O) = F Then Stall
WR	P	C		Processor write	Word	If *Shared* & PB(O) = T Then Stall Else CD(O) = V If *Invalid* Then PWC += 1 If *Pending* Then PB(O) = T If *Shared* Then PWC += 1, PB(O) = T
R	P	C		Line replacement		If ForAny O: PB(O) = T Then Stall
FENCE	P			Stall until writes performed		Stall until PWC = PUC = O
RM	C	D	Req	Read miss		DIR(Src) = T
WM	C	D	Req	Write miss	Word	MD(O) = V, DIR(Src) = T
WW	C	D	Req	Write word to directory	Word	MD(O) = V
RL	C	D	Req	Replace line		DIR(Src) = F
RLD	C	D	Req	Replace line with data	Line	MD = V, DIR(Src) = F
SR	M or C	C	Req	Shared miss reply with update count (U)	Line	PWC -= 1, PUC += U Forall O: If PB(O) = F Then CD(O) = V Else PB(O) = F, send WW with CD(O)
ER	D	C	Rep	Exclusive miss reply	Line	PWC -= 1 Forall O: If PB(O) = F Then D(O) = V Else PB(O) = F
WB	D	C	Req	Write back line to directory		
WBU	D	C	Req	Write back line to directory with data update		CD(O) = V
UW	D	C	Req	Update word to cache(s)	Word	If PB(O) = F Then CD(O) = V
RA	D	C	Req	Replacement ack		
WA	D	C	Req	Write ack with update count (U)		PWC -= 1, PUC += U, PB(O) = F
UA	C	C	Rep	Update ack		PUC -= 1
LR	C	D	Rep	Line already replaced		
LRD	C	D	Rep	Line already replaced	Word	
UL	C	D	Rep	Write back of line	Line	MD = V
RMB	D	C	Req	Bounce RM to cache		Resend RM
WMB	D	C	Req	Bounce WM to cache		Resend WM, V = CD(O)
WWB	D	C	Req	Bounce WW to cache		Resend WW, V = CD(O)
LRB	D	C	Req	Bounce LR to cache		Resend LR
LRDB	D	C	Req	Bounce LR to cache	Word	Resend LRD
UWB	C	D	Req	Bounce UW to directory		Resend UW, V = MD(O)
WBB	C	D	Req	Bounce WB to directory		Resend WB
WBUB	C	D	Req	Bounce WBU to directory		Resend WBU, V = MD(O)

Table 1: CD-UP: Description of Messages



(a) Cache Line State Diagram



(b) Memory Line State Diagram

Figure 5: CD-UP: Cache and Memory Line State Diagrams

The next few sections describe the actions taken by the CD-UP protocol for typical read misses, write misses, write hits and line replacements. Protocol races and exception handling are also discussed.

4.1.2 Read Miss

For a read miss, the requesting cache sends a *Read Miss* to the directory, and the state of the cache line is set to *Pending*, as shown in figure 6. When the directory receives the miss request, the directory can reply with the line's data if the memory line state is *Absent* or *Shared*. If the memory line state is *Absent*, then the directory replies with an *Exclusive Reply* and the memory line state is set to *Exclusive*. If the memory line state is *Shared*, then the reply is a *Shared Reply*. After the cache receives the reply, the cache line state is set to *Exclusive* for an *Exclusive Reply* or *Shared* for a *Shared Reply*.

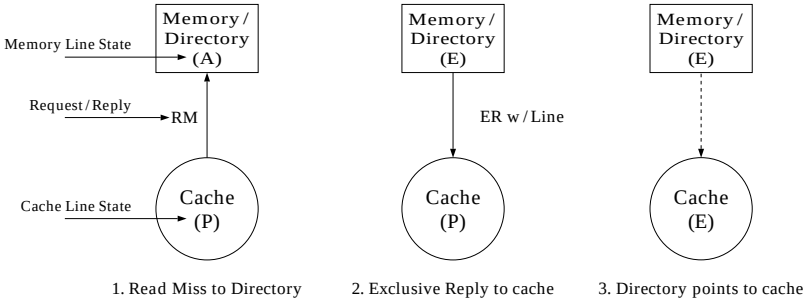


Figure 6: Read Miss to *Absent* Memory Line

If the memory line state is *Exclusive* on a *Read Miss*, the line's data must be fetched from the owning cache, as shown in figure 7. Once the requesting cache receives the line's data, the cache line state is set to *Shared*, and when the memory receives the *Update Line* with the line's data, the memory line's state is set to *Shared* and memory is updated.

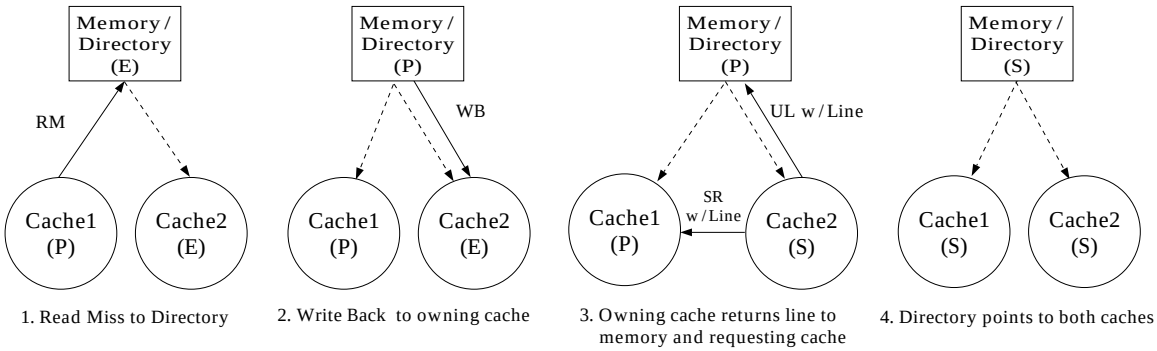


Figure 7: Read Miss to *Exclusive* Memory Line

4.1.3 Write Miss

The actions for a write miss to a memory line in the *Absent* state are identical to a read miss except that the cache sends a *Write Miss* to the directory. If the memory line is in the *Exclusive* state, the actions are similar to a *Read Miss* except that the directory updates memory's copy of the line and sends a *Write Back Update* to the owning cache. The *Write Back Update* contains the write value. This allows the owning cache to update its copy of the line before sending a *Shared Reply* to the requesting cache and an *Update Line* to the directory. After the transaction is complete, the states of both cache lines and the memory line are *Shared*.

If the memory line is in the *Shared* state on a write miss, the other copies of the line must be updated, as shown in figure 8. The writing cache sends a *Write Miss* along with the new value to the directory, increments the pending write counter (PWC) and sets the cache line state to *Pending*. Once the directory receives the miss request, it sends an *Update Word* with the new value to all caches that have a copy of the line, and the directory also sends a *Shared Reply* with a count of the updates sent and the line's data to the writing cache. When the writing cache receives the reply, it decrements the pending write counter and adds the number of updates sent to the pending update counter (PUC). The updated caches receive the *Update Word*, update their copy of the line and send an *Update Ack* to the writing cache. Upon receipt of an *Update Ack*, the writing cache decrements the pending update counter. When both the pending write and pending update counters are zero, all previously issued writes have been performed.

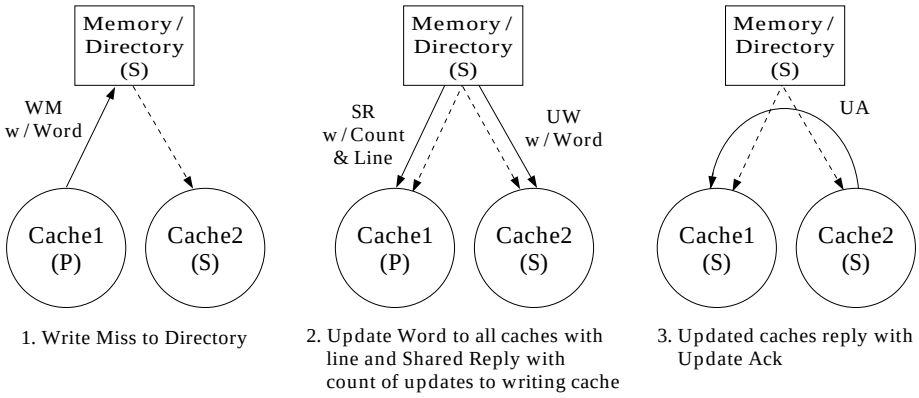


Figure 8: Write Miss to *Shared* Memory Line

4.1.4 Write Hit

Write hits to a cache line in the *Shared* state must also update all other cached copies of the line, as shown in figure 9. The resulting actions are similar to that of a *Write Miss*. The only difference is that the directory responds to the writing cache with a *Write Ack* rather than a *Shared Reply* since the writing cache already has a copy of the line.

The CD-UP protocol limits the number of outstanding updates per word to one per cache. When the processor issues a write to a cache line in the *Shared* state, the proper update-pending bit (PB)

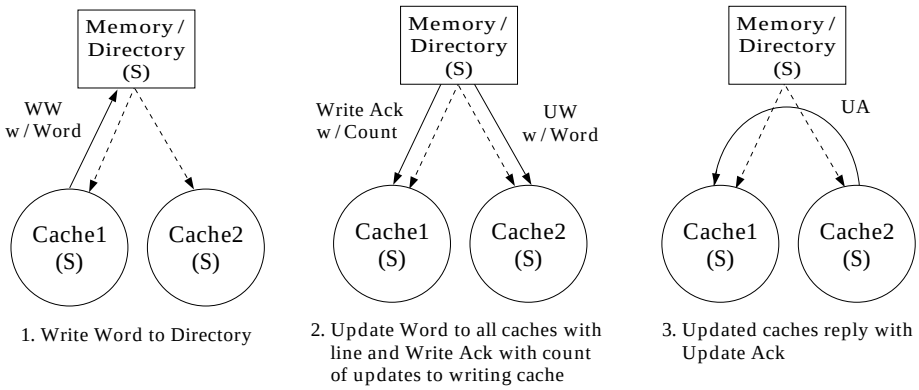


Figure 9: Write Hit to *Shared* Memory Line

is set. When the write is acknowledged by a *Write Ack*, the bit is cleared. Any write to a cache line in the *Shared* state with the update-pending bit set for the written word is blocked. If the line is in the *Pending* state, the value is written into the cache line and the word's update-pending bit is set, but unlike writes to a line in the *Shared* state, writes are not blocked if the update-pending bit is already set. They overwrite any previous value. If the pending miss reply is a *Shared Reply*, then all words with the update-pending bit set are sent to the directory in a *Write Word* message. If the miss reply is an *Exclusive Reply*, then no updates are required. In both cases, the update-pending bits are cleared after the reply is received.

4.1.5 Line Replacement

Line replacement in the CD-UP protocol is straight forward, as shown in figure 10. The cache sends a *Replace Line* to the directory. The directory clears the cache's bit pointer in the directory entry for the memory line and acknowledges the replacement with a *Replacement Ack*. If the cache line was in the *Exclusive* state and had been modified, then the cache must send the line's data along with the replacement request to the directory.

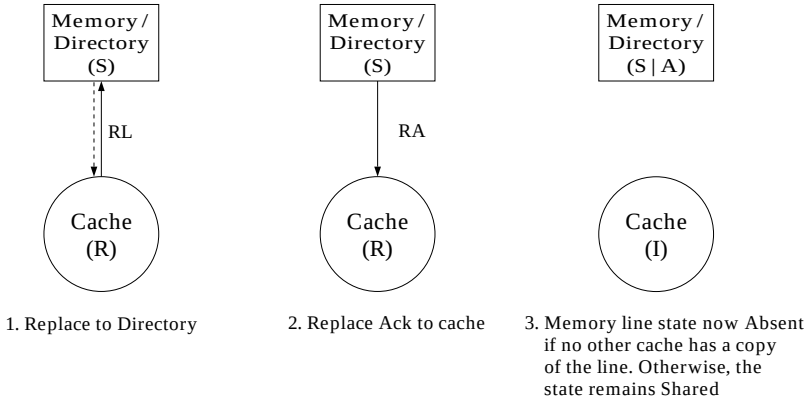


Figure 10: Replacing Cache Line in the *Shared* State

4.1.6 Protocol Races

There are several types of protocol races in the CD-UP protocol. The first type occurs when the directory receives a request to a memory line in the *Pending* state. These requests include *Read Miss*, *Write Miss* and *Write Word*. These races may occur when a cache has sent a miss request for a line that is currently owned by another cache and either the requesting cache initiates a *Write Word* or a third cache sends a miss request for the line before the *Update Line* from the owning cache is received by the directory. Figure 11 shows how a *Write Word* race might occur. For all three request types, the directory can bounce the request back to the sender to be retried.

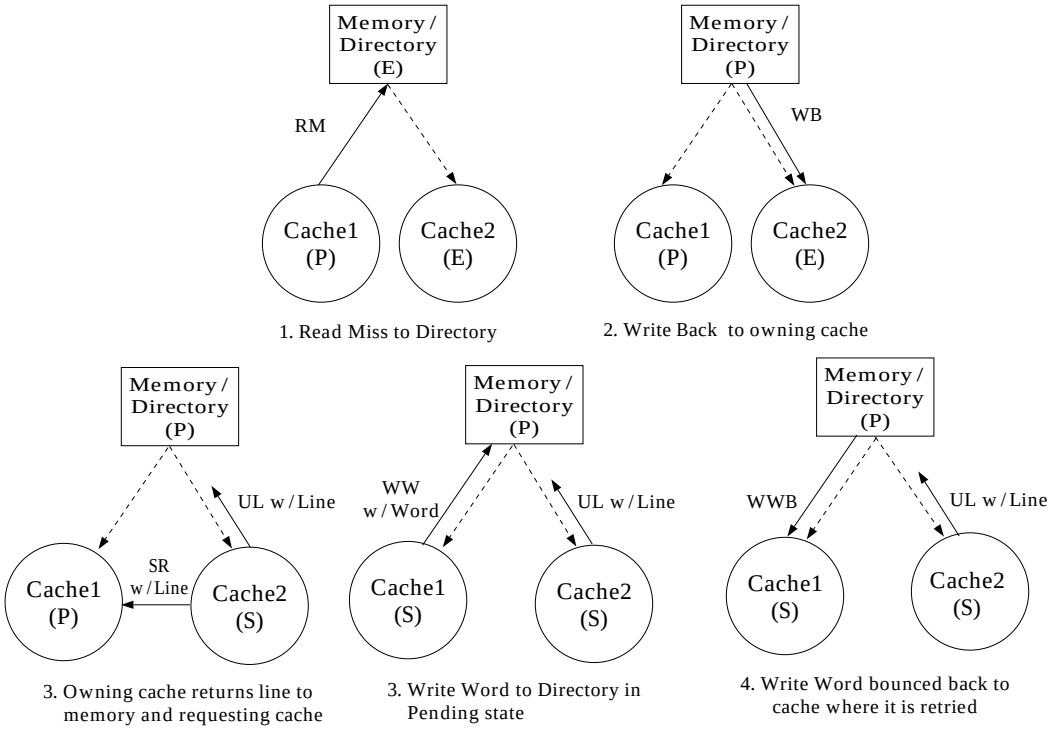


Figure 11: Race: Write Word to *Pending* Memory Line

The second type of protocol race is a request to a cache line in the *Pending* state. These requests include *Update Word* and *Write Back*. The *Update Word* race may occur if a cache sends a miss request to the directory, and if before the cache receives the miss reply, the cache receives an *Update Word* from a write by another cache, as shown in figure 12. The *Update Word* may reach the pending cache before the *Shared Reply* since the messages follow different paths: the *Update Word* from the directory and the miss reply from another cache. In this case, the cache must bounce the *Update Word* back to the directory. When the directory receives the bounced update, it resends the *Update Word* with the *latest* value of the memory word. This implies that a cache may not see all updated values if there are multiple writes to the same address without any synchronization events controlling the writes, but it will see the latest value.

The *Write Back* race may occur since an *Exclusive Reply* to a cache is a reply and the subsequent *Write Back* is a request; and therefore, they travel down different networks and may reach the cache

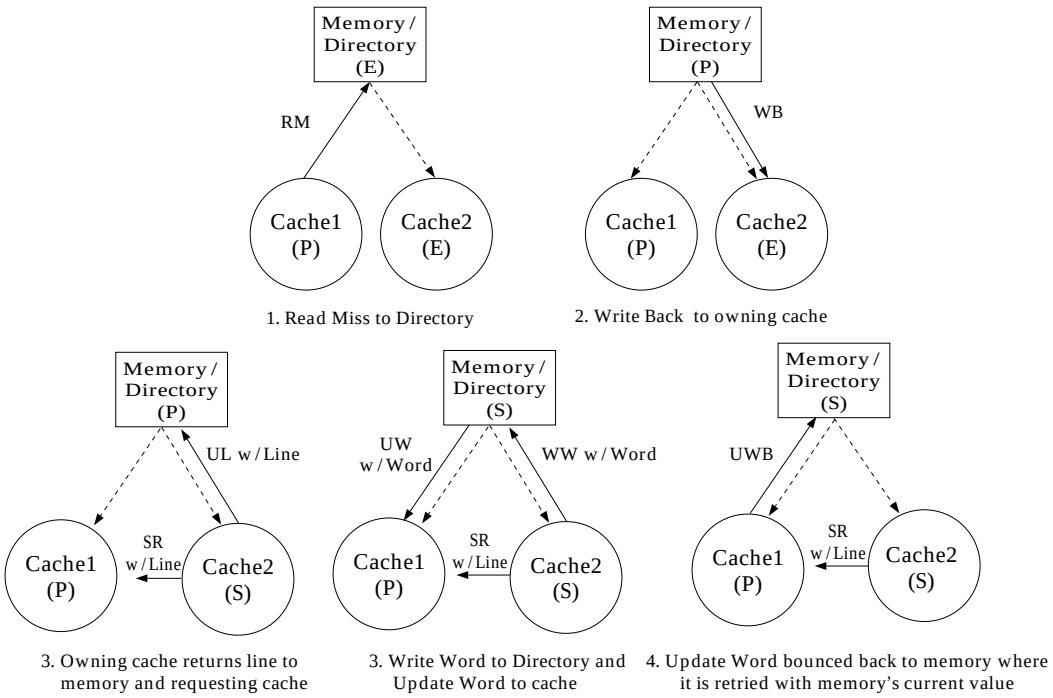


Figure 12: Race: Update Word to *Pending* Cache Line

out of order, as shown in figure 13. In this case, the *Write Back* is bounced back to the directory to be retried.

The final type of race is a request to a cache line in the *Replacing* state. These requests include *Update Word* and *Write Backs*. For the *Update Word* request, the cache can simply acknowledge the update. For the *Write Back* requests, the cache line must have been in the *Exclusive* state previously and just sent a *Replace Line* request to the directory. In this case, the cache sends a *Line Replaced* (a *Line Replaced Data* with the update data if the request was a *Write Back Update*) back to the directory. When the directory receives the *Line Replaced*, it will have already received the *Replace Line* request, and memory's data will be valid. The directory may now respond to the initial miss request with memory's current data.

4.1.7 Exceptions

As described in section 3, the protocol must be able to break request-request deadlock. Table 2 shows the request messages that may generate another request. These messages can be divided into two classes. The first class of messages are essentially order-independent (OI). They do not rely on the order preserving nature of the network and, therefore, do not introduce any additional complexity to the protocols if they are bounced back to the sender as an exception. For example, the *Read Miss* message is order-independent. Once a cache sends a *Read Miss* to the directory, the cache will not generate any other messages relating to this line, and until the directory receives the miss request, it will not send the cache any message pertaining to the line. The *Read Miss* message can take any path from the cache to the directory including being sent back to the cache

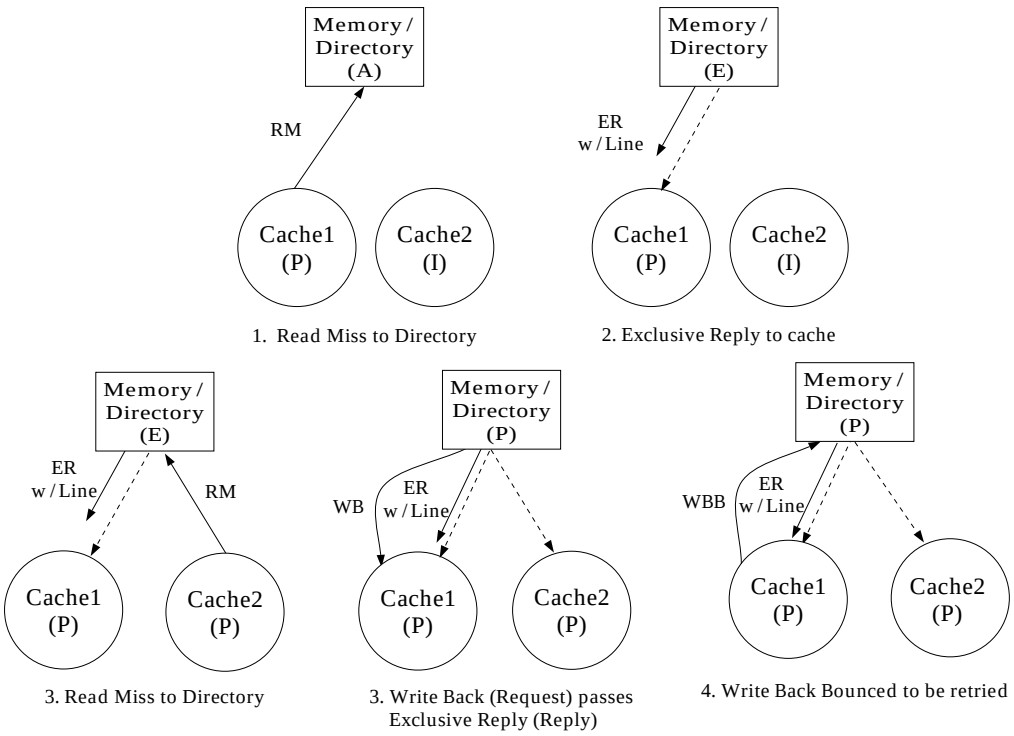


Figure 13: Race: Write Back to *Pending* Cache Line

as an exception without introducing any new complexities to the protocol.

Of the order-dependent messages, the exception of a *Replace Line* message may cause a race condition, as shown in figure 14. In this case, a *Line Replaced* may reach the directory before the *Replace Line*. The directory must bounce the *Line Replaced* back to the replacing cache. This is repeated until the directory receives the *Replace Line* message and the line's current data. Now when the *Line Replaced* is received, the directory may respond to the initial miss request.

An *Update Word Bounce* may also result in an exception, but since an *Update Word Bounce* does not carry data (it only carries a promise of an update), it can take an arbitrary path between the cache and directory without introducing any additional problems in the protocol. Once the directory is able to process the *Update Word Bounce*, it sends an *Update Word* with the memory's current data. If the destination of the update no longer has a copy of the line, the directory acknowledges the writing cache directly.

Message	Src	Dst	Description	Type
RL	C	D	Replace line	OD
UWB	C	D	Bounce UW to directory	OD
RM	C	D	Read miss	OI
WM	C	D	Write miss	OI
WW	C	D	Write word	OI
WB	D	C	Write back line	OI
WBU	D	C	Write back line w/update	OI
SR	DC	C	Shared miss reply	OI
LR	C	D	Line replaced	OI
RMB	D	C	Bounce RM to cache	OI
WMB	D	C	Bounce WM to cache	OI
WWB	D	C	Bounce WW to cache	OI
WBB	C	D	Bounce WB to directory	OI
WBUB	C	D	Bounce WB to directory	OI
LRB	D	C	Bounce LR to cache	OI

Table 2: CD-UP: Messages That May Deadlock

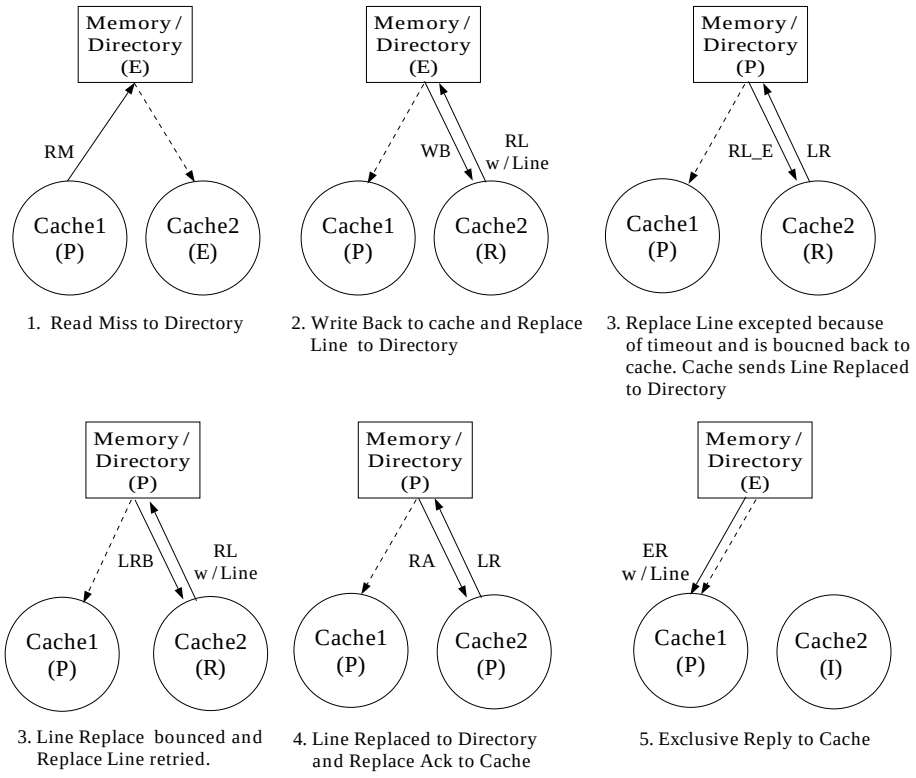


Figure 14: Race: Line Replaced to *Pending* Memory Line

4.2 Distributed Directory (DD-UP)

In this section, the details of the singly-linked distributed directory update-based protocol (DD-UP) are presented. The DD-UP is based on the directory structure and singly-linked lists of the SDD invalidate-based protocol [19, 20].

4.2.1 Cache and Memory Line States

In the DD-UP protocol, a cache line may be in one of 12 states:

- *Invalid* - the cache's copy of the line is not valid
- *Pending* - the cache has issued a miss and is waiting for a reply
- *Exclusive* - the cache's copy is the only copy of the line (cache is owner of line),
- *Shared* - the line is shared by multiple caches and the cache is not the head nor tail of the list - writes must update other cached copies of the line
- *Shared-Head* - the line is shared by multiple caches and the cache is the head of the list - writes must update other cached copies of the line
- *Shared-Tail* - the line is shared by multiple caches and the cache is the tail of the list - writes must update other cached copies of the line
- *Replacing-Head* - the cache's copy, which is the head of the list, is being replaced
- *Replacing* - the cache's copy, which is not the head nor tail of the list, is being replaced
- *Replacing-Tail* - the cache's copy, which is the tail of the list, is being replaced
- *Replacing-Exclusive* - the cache's copy, which is the only copy of the line, is being replaced
- *Exclusive-Replacing* - the cache's copy was the head of the list and the next cache in the list, which was the tail of the list, is replacing itself
- *Shared-Head-Replacing* - the cache's copy is the head of the list and a cache in the list is replacing itself.

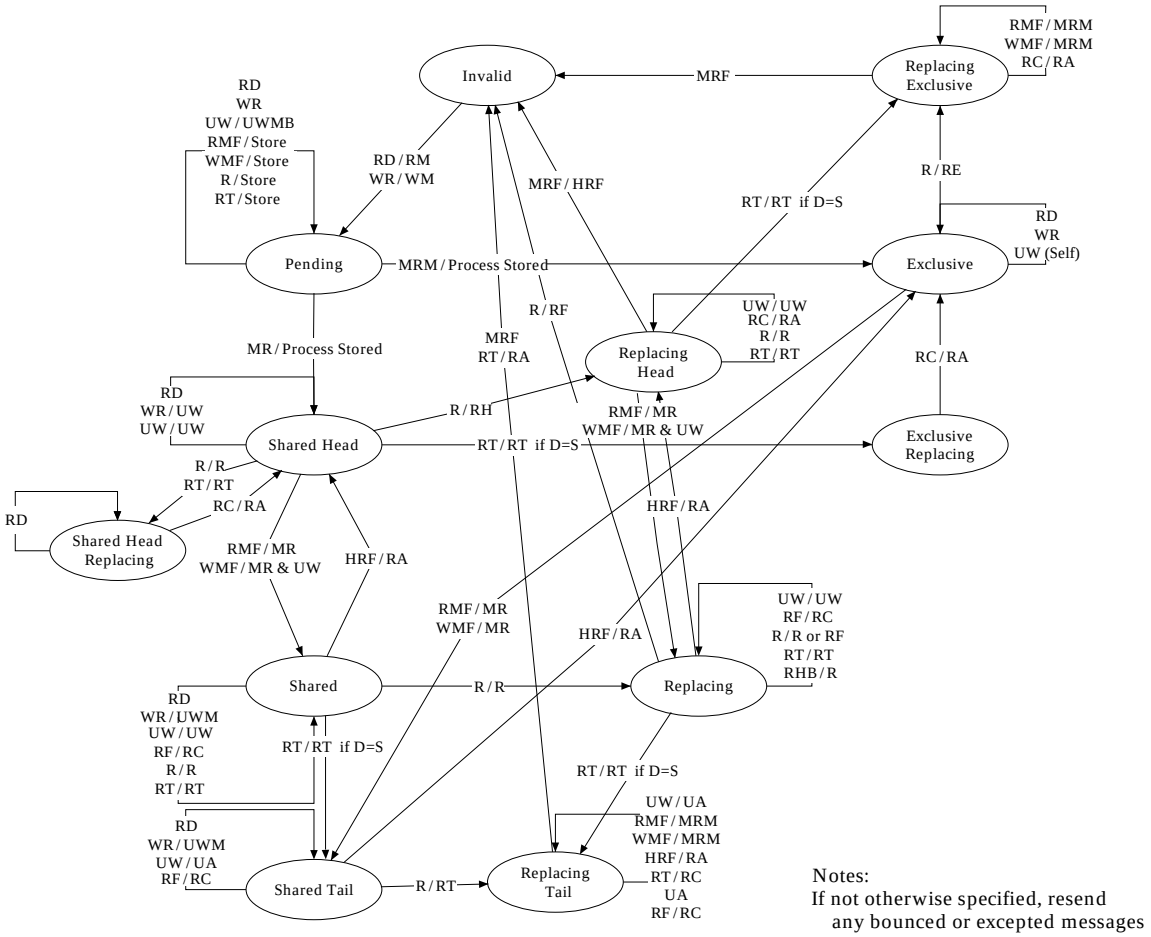
A memory line may be in one of 3 states:

- *Absent* - no cached copies of the line exist
- *Present* - at least one cache has a copy of the memory line - memory is not consistent
- *Replacing* - a cache in the list is removing itself from the list - memory is not consistent.

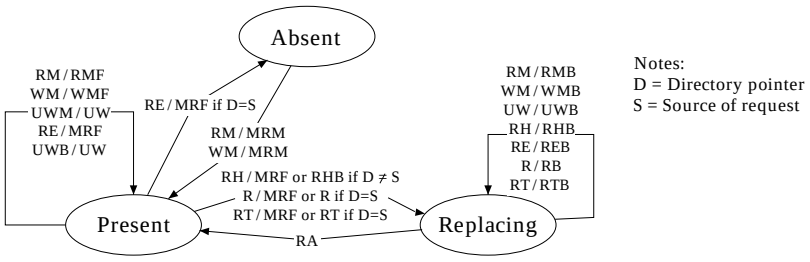
Table 3 gives a brief description of the protocol messages. The columns have the same meaning as in the CD-UP protocol. The DD-UP protocol can store (Store) requests to cache lines in the *Pending* state in the pointer field of the cache line. When the appropriate reply is received and the cache line state is changed, the pending signal is processed (Process-Stored). The state *Shared-States* implies any of the three shared states: *Shared-Head*, *Shared* or *Shared-Tail*. The Aux field is an additional pointer field used by some of the message types. Figure 15 shows the state transition diagrams for cache and memory lines.

Message	Src	Dst	Type	Description	Data	Destination Actions
RD	P	C		Processor read		If *Pending* & PB(O) = F Then Stall
WR	P	C		Processor write	Word	If *SharedStates* & PB(O) = T Then Stall Else CD(O) = V If *Invalid* Then PWC += 1 If *Pending* Then PB(O) = T If *Shared-States* Then PWC += 1, PB(O) = T
R	P	C	Req	Line replacement		If ForAny O: PB(O) = T Then Stall Else Aux = Dir
FENCE	P	C		Stall until writes performed		Stall until PWC = PUC = O
RM	C	D	Req	Read miss		Dir = Src
WM	C	D	Req	Write miss	Word	Dir = Src, Aux = Src
UWM	C	D	Req	Update word memory	Word	Aux = Src
UW	D or C	C	Req	Update word	Word	If PB(O) = F Then CD(O) = V If Src = Aux Then PB(O) = F
RE	C	D	Req	Replace exclusive line		
RH	C	D	Req	Replace shared-head line		Dir = Aux
R	C D or C	D C	Req Req	Replace shared line Replace shared line		 If Dir = Src Then Dir = Aux If Dir = Aux Then Dst is replacing cache
RT	C D or C	D C	Req Req	Replace shared-tail line Replace shared-tail line		
HRF	C	C	Req	Head replace flush		
RC	C	C	Req	Replace complete		
MRF	D	C	Req	Memory replace flush		
RF	C	C	Req	Replace flush		
MR	C	C	Req	Miss reply with update count (U) (U = 0,1)	Line	PWC -= 1, PUC += U, DIR = Src Forall O: If PB(O) = F Then CD(O) = V Else send UW with CD(O), PB(O) = F
MRM	D	C	Rep	Memory miss reply	Line	PWC -= 1 Forall O: If PB(O) = F Then CD(O) = V Else PB(O) = F
RMF	D	C	Req	Read miss forward		
WMF	D	C	Req	Write miss forward with data update		CD(O) = V
RA	D	C	Req	Replace ack		
UA	C	C	Rep	Update ack		PUC -= 1, PB(O) = F
RMB	D	C	Req	Bounce RM to cache		Resend RM
WMB	D	C	Req	Bounce WM to cache		Resend WM, V = CD(O)
UWB	D C	C D	Req Req	Bounce UW to cache Bounce UW to directory		Resend UW, V = CD(O) Resend UW
RHB	D	C	Req	Bounce RH to cache		Resend RH
REB	D	C	Req	Bounce RE to cache		Resend RE
RTB	D	C	Req	Bounce RT to cache		Resend RT
RB	D	C	Req	Bounce R to cache		Resend R

Table 3: DD-UP: Description of Messages



(a) Cache Line State Diagram



(b) Memory Line State Diagram

Figure 15: DD-UP: Cache and Memory Line State Diagrams

The next few sections describe the actions taken by the DD-UP protocol for typical read misses, write misses, write hits and line replacements. Protocol races and exception handling are also discussed.

4.2.2 Read Miss

For a read miss, the cache sends a *Read Miss* request to the directory. If the state of the memory line is *Absent*, then the directory replies to the miss with a *Miss Reply Memory*. The state of the memory line is set to *Present*, and the cache line state is set to *Exclusive*, as shown in figure 16.

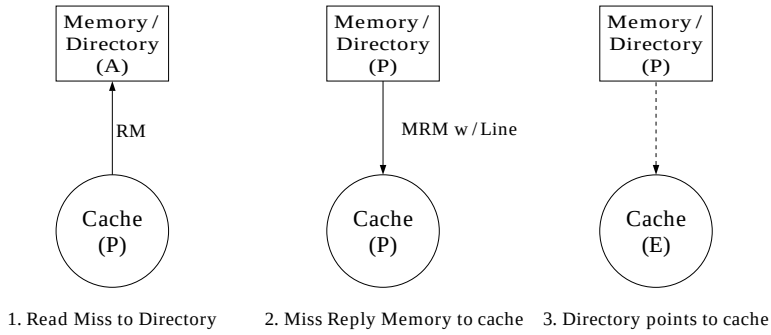


Figure 16: Read Miss to *Absent* Memory Line

If the memory line state is *Present*, then the data must be fetched from the cache at the head of the list, and the requesting cache must be added to the head of the list, as shown in figure 17. The directory responds to the miss request by sending a *Read Miss Forward* to the cache at the head of the list, and the directory pointer is changed to point to the requesting cache, the new head of the list. The old head of the list responds to the *Read Miss Forward* by sending a *Miss Reply* and the line's data to the requesting cache. After receiving the reply, the requesting cache sets its directory pointer to point to the cache that sent the reply.

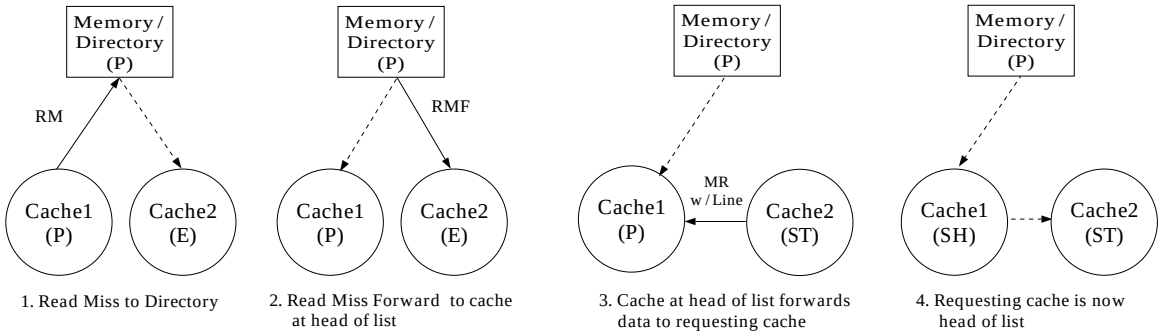


Figure 17: Read Miss to *Present* Memory Line

4.2.3 Write Miss

The actions for a write miss are identical to a read miss if the memory line state is *Absent*. If the memory line state is *Present*, then the directory sends a *Write Miss Forward* and the new data value to the old head of the list, as shown in figure 18. The old head of the list updates its copy of the line, sends a *Miss Reply* along with the line's data to the requesting cache and sends an *Update Word* down the list of caches. The cache at the end of the list acknowledges the update, and the writing cache becomes the head of the list.

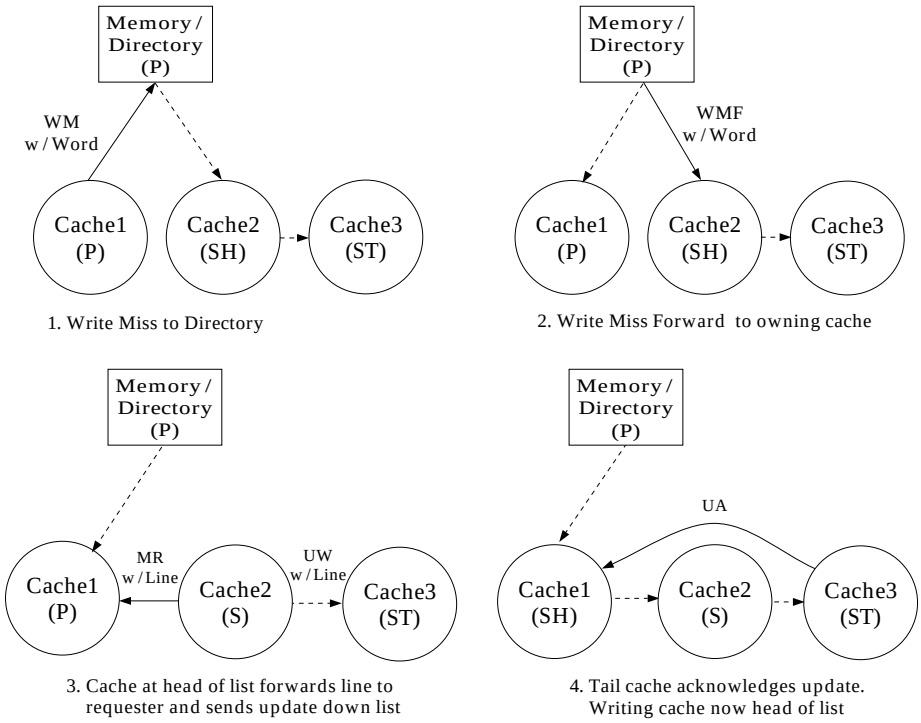


Figure 18: Write Miss to *Present* Memory Line

4.2.4 Write Hit

On a write hit to a cache line in any of the three shared states, *Shared-Head*, *Shared* or *Shared-Tail*, the other caches in the list must be updated. If the writing cache is at the head of the list, it can simply send an *Update Word* to the next cache in the list, as shown in figure 19. This cache updates its copy and forwards the *Update Word* down the list. The cache at the end of the list acknowledges the update.

If the cache is not the head of the list, the write value must be forwarded to the head of the list. To do this, the writing cache sends an *Update Word Memory* to the directory, as shown in figure 20. The directory forwards it to the head of the list, and an *Update Word* is propagated down the list. The cache at the end of the list acknowledges the update. Note that the writing cache will receive the update request, but must forward the update on to the next cache in the list.

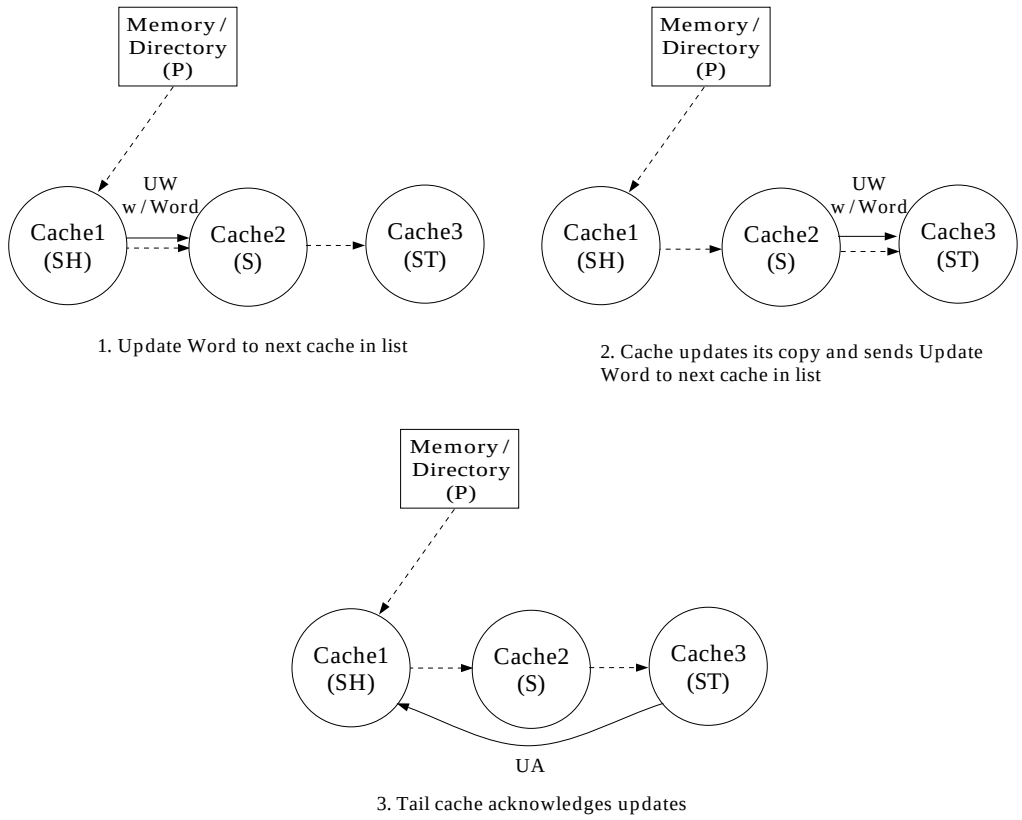


Figure 19: Write Hit from Cache Line in *Shared Head* State

As in the CD-UP protocol, the DD-UP protocol limits the number of outstanding updates per word to one per cache. When a cache modifies a cache line in the *Shared* or *Shared Tail* states, the proper pending bit (PB) is set. When the cache receives its own *Update Word*, the bit is cleared. Writes to words in shared lines with the word's pending bit already set are blocked and updates to these words do not update the cache line, but the update is still forwarded to the next cache in the list. If there are multiple writes to the same word by different processors, each processor may not see all values, but the final value will be consistent. The order that the updates reach the cache at the head of the list determines the total ordering of the writes.

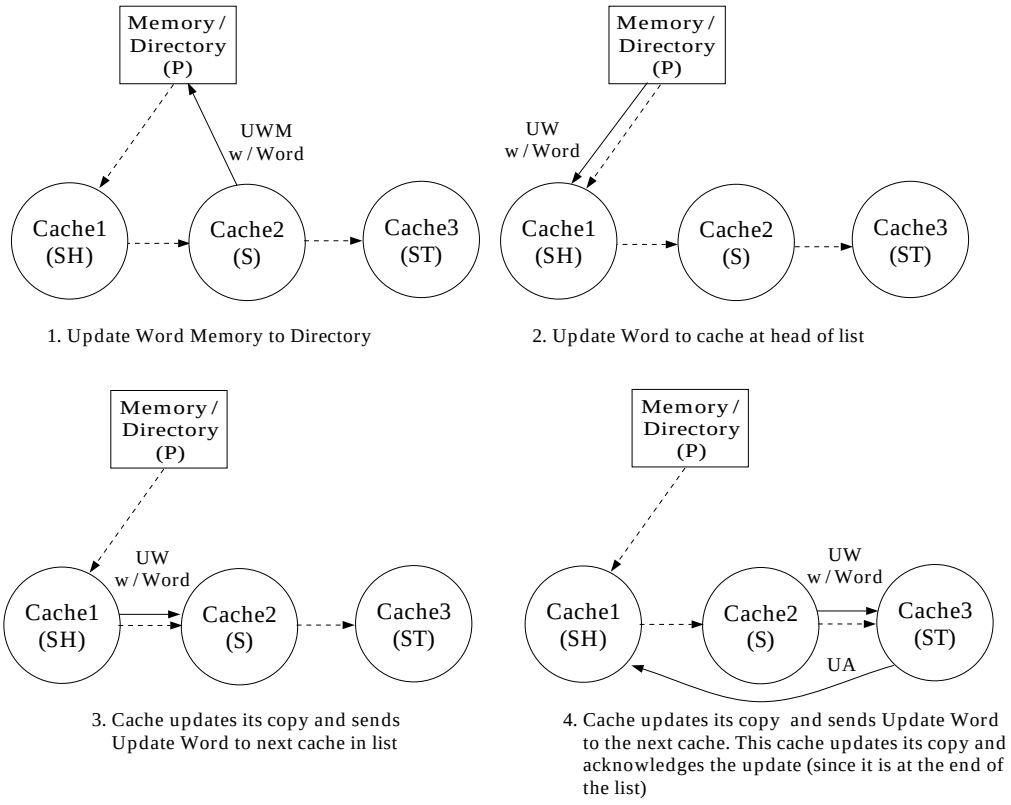


Figure 20: Write Hit from Cache Line in *Shared* State

4.2.5 Line Replacement

Line replacement is much more difficult in the DD-UP protocol than in the CD-UP protocol. In the CD-UP protocol, the message path from the directory to each cache is fixed. But in the DD-UP protocol, the path from the directory to a given cache is dependent on the current structure of the directory list for the line. As caches are added and deleted from this list, the structure of the list is altered. This changes the path of messages sent between the directory and caches and between caches.

To make line replacement possible without a significantly more complex protocol, a new state, *Replacing*, is added to the possible states of a memory line. If a memory line is in this state, all requests to the line are bounced back to the requester. This prevents new messages from attempting to traverse the list while it is being altered. Messages currently traversing the list are flushed out with special flushing messages, which will be described in the next few sections.

For a cache line in the *Exclusive* state, line replacement is identical to the CD-UP protocol. The cache sends a *Replace Line* message to the directory and the directory responds with a *Replace Ack*.

If the cache line is in the *Shared Head* state, the cache sends a *Replace Head* message to the directory, as shown in figure 21. The directory changes the memory line state to *Replacing* and replies to the replacing cache with a *Memory Replace Flush*. The replacing cache then sends a *Head Replace Flush* to the next cache in the list. This cache now becomes the head of the list, or if it was the tail of the list, the state of the cache line state is set to *Exclusive*. This cache sends a *Replace Ack* back to the directory. This indicates that the replacement is complete, and the memory line state is set back to *Present*. The *Memory Replace Flush* and the *Head Replace Flush* are used to flush any pending messages that might be traversing the list.

To replace a cache line in the *Shared* state, the cache sends a *Replace* message to the directory, as shown in figure 22. The directory sets the memory line state to *Replacing* and forwards the *Replace* to the head of the list. The cache at the head of the list changes its cache line state to *Shared Head Replacing*. This prevents this cache from generating any new updates while the list is being altered. Each cache in turn forwards the *Replace* request down the list. The request specifies the replacing cache and the next cache in the list after the replacing cache. When a cache receives the request, it checks if its directory pointer points to the replacing cache. If so, it sets its directory pointer to point to the cache following the replacing cache. When the replacing cache receives its own request, it sends a *Replace Flush* to the next cache in the list, and sets its cache line state to *Invalid*. The *Replace* and *Replace Flush* messages have flushed out any requests that were flowing down the list. Once the next cache in the list receives the *Replace Flush*, it sends a *Replace Complete* back to the head of the list. The cache at the head of the list changes its cache line state back to *Shared Head* and sends a *Replace Ack* back to the directory. The memory line state is then changed back to *Present*.

The actions to replace a cache line in the *Shared Tail* state are almost identical to the replacement of a *Shared* line, but since the replacing cache is at the end of the list, it does not need to flush the next section of the list. It can send the *Replace Complete* to the cache at the head of the list once it receives its own replace request.

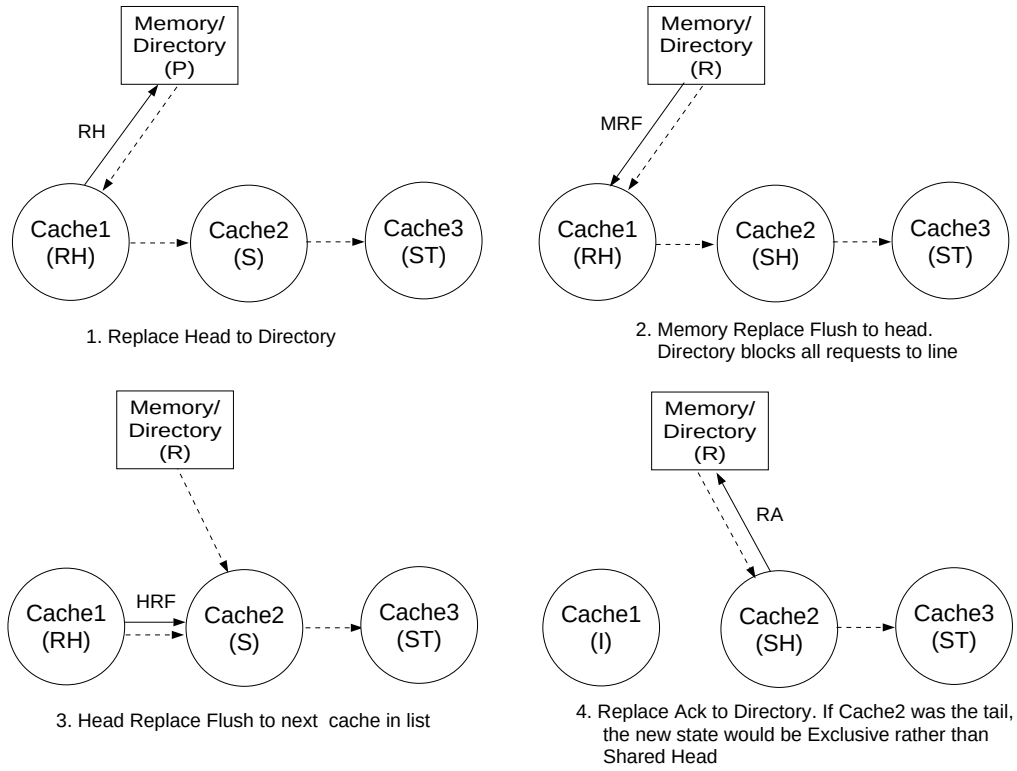


Figure 21: Replacing Cache Line in *Shared Head* State

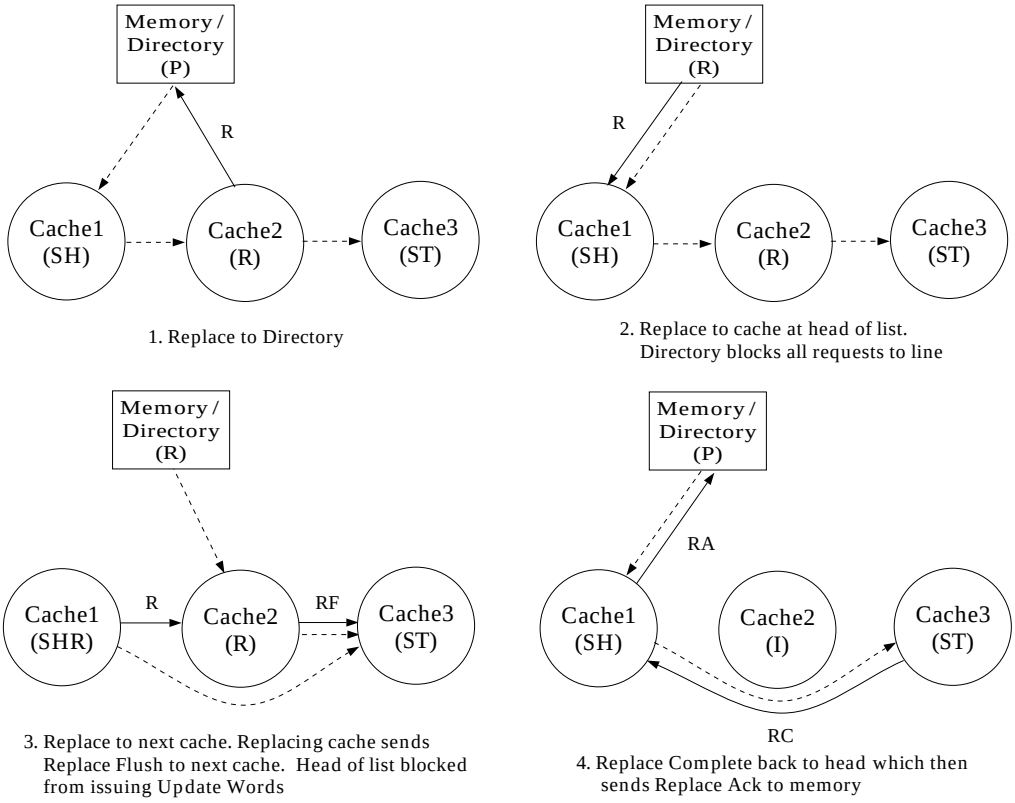


Figure 22: Replacing Cache Line in *Shared* State

4.2.6 Protocol Races

There are two types of protocol races in the DD-UP protocol. The first type is a request to a memory line in the *Replacing* state. In this case, the directory can bounce the request back to the sender to be retried. The second race condition is an *Update Word* to a cache line in the *Pending* state, as shown in figure 23. In this case, the *Update Word* can be bounced back to the directory to be retried. The *Update Word Bounce* still carries the data value, but this does not create an update ordering problem since the order is determined by the order that the *Update Words* reach the cache at the head of the list. The bounced update has yet to reach the head of the list and, therefore, is not yet an ordered update.

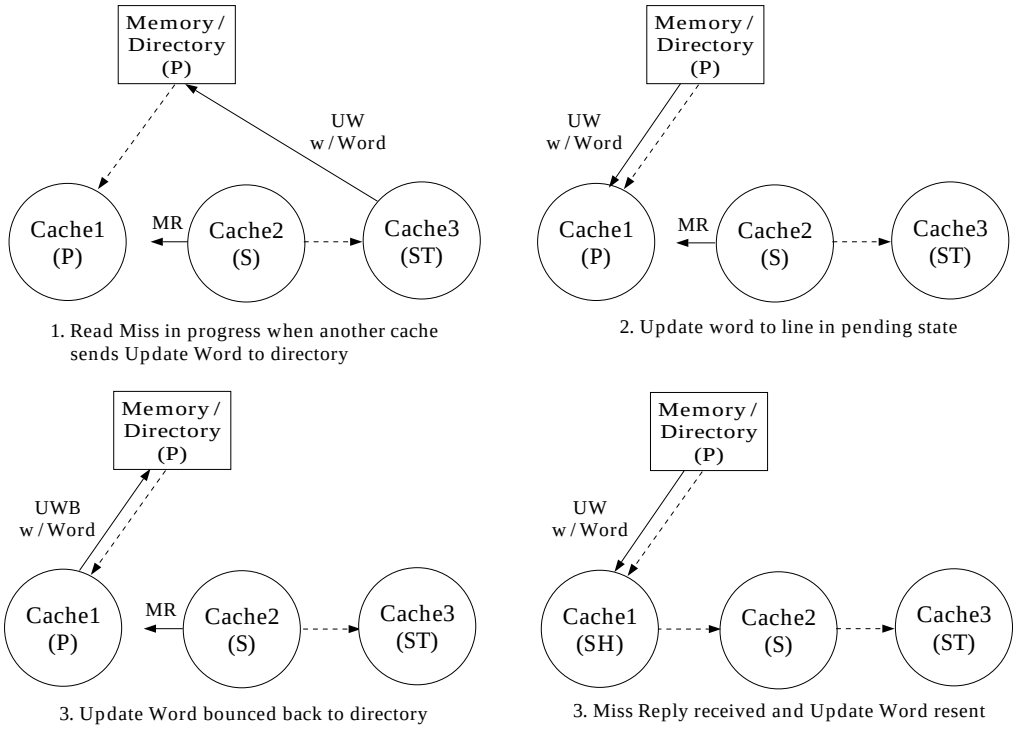


Figure 23: Race: Update Word to *Pending* Cache Line

4.2.7 Exceptions

As described in section 3, the protocol must be able to avoid protocol level deadlock. Table 4 shows the request messages that may generate another request in the DD-UP protocol. As with the CD-UP protocol, the order independent (OI) requests can be sent back to the source as an exception without adding any complexity to the protocol.

The only order-dependent message, *Update Word*, may change the order of updates flowing down the list. For example, figure 24 shows two updates from different processors flowing between two caches. The first update results in an exception and is sent back to the sender. The second update reaches the destination cache and updates the cache line. Once the sending cache receives

Message	Src	Dst	Description	Type
UW	C	C	Update Word	OD
RM	C	D	Read Miss	OI
WM	C	D	Write Miss	OI
UWM	C	D	Update word to directory	OI
RE	C	D	Replace Exclusive	OI
RH	C	D	Replace Shared-Head	OI
R	C	D	Replace Shared	OI
RT	C	D	Replace Shared-Tail	OI
UW	D	C	Update Word	OI
R	D	C	Replace Shared	OI
RT	D	C	Replace Shared-Tail	OI
MRF	D	C	Memory Replace Flush	OI
BOUNCE	D	C	All Bounced messages	OI
RF	C	C	Replace Flush	OI

Table 4: DD-UP: Messages That May Deadlock

the excepted update, it must resend it with the *latest* value, which is the value from the second update. The destination cache receives this update and updates its cache line. Now both caches are consistent with the value from the second update, although the second cache never saw the update of the first value.

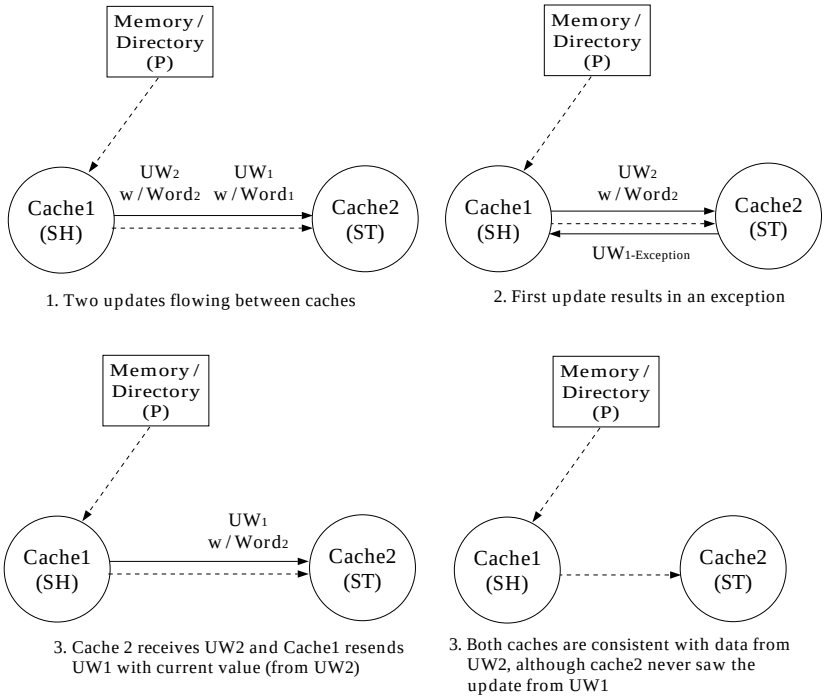


Figure 24: *Update Word* Exception

5 Protocol Verification

An exhaustive verification tool called Murφ [4, 5] was used to verify the update-based protocols. To verify a protocol using Murφ, a description of the system and a behavioral description of the protocol is required. From this, Murφ builds a system state and attempts to traverse it by applying rules from the behavioral description of the protocol. Error statements and invariants are used to detect errors.

For example, figure 25 shows the architectural model on which the update-based protocols were verified. The system consists of three caches and one directory/memory. The caches each have one request and one reply buffer, and the directory has four request and one reply buffer. The memory consists of a single line with a two-bit data word. The single line is sufficient since protocols actions do not interact between lines. The single, two-bit data word is also sufficient since if there is a case in which a "wrong" value overwrites a "correct" value then the exhaustive nature of the tool will find the same case for the two values used for the data word.

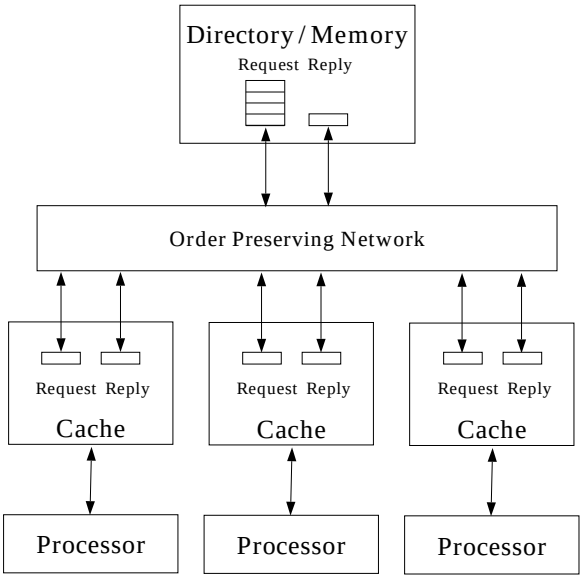


Figure 25: Verification Model

The system state created by Murφ, consists of a concatenation of all the state bits in the system. This includes the bits in the cache and memory line data and state information and the message data in the network buffers. In this simple case, this would result in several hundred bits of state, or $2^{state-bits}$ states, a significant number of states.

The actual number of states traversed is dependent on the behavioral rules of the protocol. For the update-based protocols examined, this number quickly consumes all the memory available for the verification since Murφ must remember which states have been visited. There are two techniques to reduce the number of states traversed and, therefore, Murφ's memory requirements.

The first technique is to use symmetry to eliminate redundant states [12, 13]. Symmetry in a system allows Murφ to find states that are equivalent in their current and future behavior with

respect to error checking. During verification, only one member of each equivalence class needs to be examined. This technique is able to significantly reduce the total number of states examined. Using symmetry does not affect the correctness or coverage of the protocol verification.

The second technique is to limit the number of concurrent actions. Since Murφ is an exhaustive verification tool, every possible combination of events must be verified. Therefore, the more active events, the larger number of traversed states. In the verification of the update-based protocols, the number of outstanding updates was limited to one per cache.

Overall, the verification tool was useful in verifying the correctness of the protocols. Errors were detected very quickly, but the state explosion problem limited the size and scope of the verification. As discussed above, the only limitation of the verification that might affect correctness was the limited number of outstanding updates, but the combination of Murφ verification, running simulation with the update-based protocols and hand verification have produced a correct protocol with a high confidence level.

6 Summary

In this paper, the details of two update-based cache coherence protocols were presented. The centralized-directory (CD-UP) protocol was much simpler than the singly-linked distributed directory protocol (DD-UP). The main source of the complexity in the DD-UP resulted from the extra states needed for cache line replacements. In the CD-UP protocol, the path from the directory to a cache was fixed, but in the DD-UP protocol, the path changed as the structure of the list was altered by caches adding and removing themselves from the list. This changing path required the special flushing messages and extra states of the DD-UP protocol. The alternative doubly-linked directory structure might result in a simpler protocol since replacements would be simpler.

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, 1988.

[2] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. *IEEE Computer*, June 1990.

[3] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, 1991.

[4] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[5] Andreas J. Drexler and C. Norris Ip. *Murφ Annotated Reference Manual*. Stanford University, 1992.

[6] Susan J. Eggers and Randy H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 373–382, May 1988.

[7] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. The Impact of Cache Coherence Protocols on Systems Using Fine-Grain Data Synchronization. Technical Report CSL-TR-94-611, Computer Systems Laboratory, Stanford University, March 1994.

[8] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 534–545, January 1994.

[9] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. Write Grouping for Update-Based Cache Coherence Protocols. Technical Report CSL-TR-94-612, Computer Systems Laboratory, Stanford University, March 1994.

[10] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. Technical Report No. CSL-TR-90-417, Computer Systems Laboratory, Stanford University, 1990.

[11] IEEE Standards Department, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331. *Scalable Coherent Interface: Logical, Physical and Cache Coherence Specifications*, P1596/D2.00 edition, November 1991.

[12] C. Norris Ip and David L. Dill. Better Verification Through Symmetry. In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and Their Applications*, April 1993.

[13] C. Norris Ip and David L. Dill. Efficient Verification of Symmetric Concurrent Systems. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, 1993.

[14] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *7th ACM International Conference of Supercomputing*, 1993.

[15] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[16] Brian W. O'Krafka and A. Richard Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proceedings of the 17th International Symposium on Computer Architecture*, June 1990.

[17] Richard Simoni and Mark Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991.

[18] Per Stenstrom. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[19] Manu Thapar. Cache Coherence for Scalable Shared Memory Multiprocessors. Technical Report CSL-TR-92-522, Computer Systems Laboratory, Stanford University, May 1992.

[20] Manu Thapar, Bruce A. Delagi, and Michael J. Flynn. Linked List Cache Coherence for Scalable Shared Memory Multiprocessors. In *7th International Parallel Processing Symposium*, April 1993.

[21] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Systems (ASPLOS III)*, pages 243–256, April 1989.