Chapter Four

An Implementation of Hierarchically Managed DSM

The memory local to a node in a DSM system maintains only a part of the entire shared address space. The local memory functions as a cache for the entire distributed address space. When a processor issues an access, the datum referenced by the access is not guaranteed to be resident locally. As described in Section 3.2.1 in Chapter 3, local memory management tables are used to determine whether the access can be completed locally. If it can, the various tables locate the datum in local memory. If the datum is not available locally or its state does not permit the access, a request must be sent, through the interconnection network, to a remote node that can provide the datum with the appropriate access permission. Requests received from remote nodes must be processed to provide the required data. The inter-node interactions involved in moving data to where it is needed are governed by the coherence protocol.

We have investigated hardware implementation of the address translation mechanisms and the coherency protocol required to support hierarchically managed (and one-level managed) DSM. The address translation functionality of the Memory Management Unit (MMU) is extended to manage distributed shared memory. The design of the MMU also integrates support to handle the inter-node interactions required for maintaining coherency. We discuss our design of the MMU in Section 4.1. Designing a distributed directory coherency protocol must address several issues such as deadlocks and race conditions. Section 4.2 discusses these issues, and our solutions to these issues. Our distributed directory coherence protocol is completely described in Section 4.3. Section 4.4 describes our implementation of hierarchically managed DSM for trace-driven simulation and execution-driven simulation.

4.1. Memory Management Unit design

The Memory Management Unit (MMU) provides most of the hardware support for the DSM implementation. Figure 4.1 shows the block diagram of the MMU. To support address translation for shared accesses, the MMU implements the mapping tables Present Pages Table (PPT), Shared Pages Table (SPT) and the Block Map per Processor (BMpP) copy directory. The MMU also implements the coherence engine which ensures that all requests are serviced according to the coherence protocol.

The MMU services requests from the local processor as well as requests from remote nodes. To satisfy local and remote requests simultaneously, the MMU maintains independent datapaths for local and remote requests. Since both local and remote requests may need to look-up the Present Pages Table (comprised of the Hash Anchor Table, HAT, and the Page Frame Table, PFT) and the Shared Pages Table, the PPT and the SPT are multi-ported. In addition, the access to the Hash Anchor Table (HAT Acc. in Figure 4.1), access to the Page Frame Table (PFT Acc. in the figure) and access to the Shared Pages Table (SPT Acc. in the figure) are performed in pipeline sequence, both for local requests and remote requests. This allows multiple local requests and multiple remote requests to be serviced concurrently in a pipelined manner.

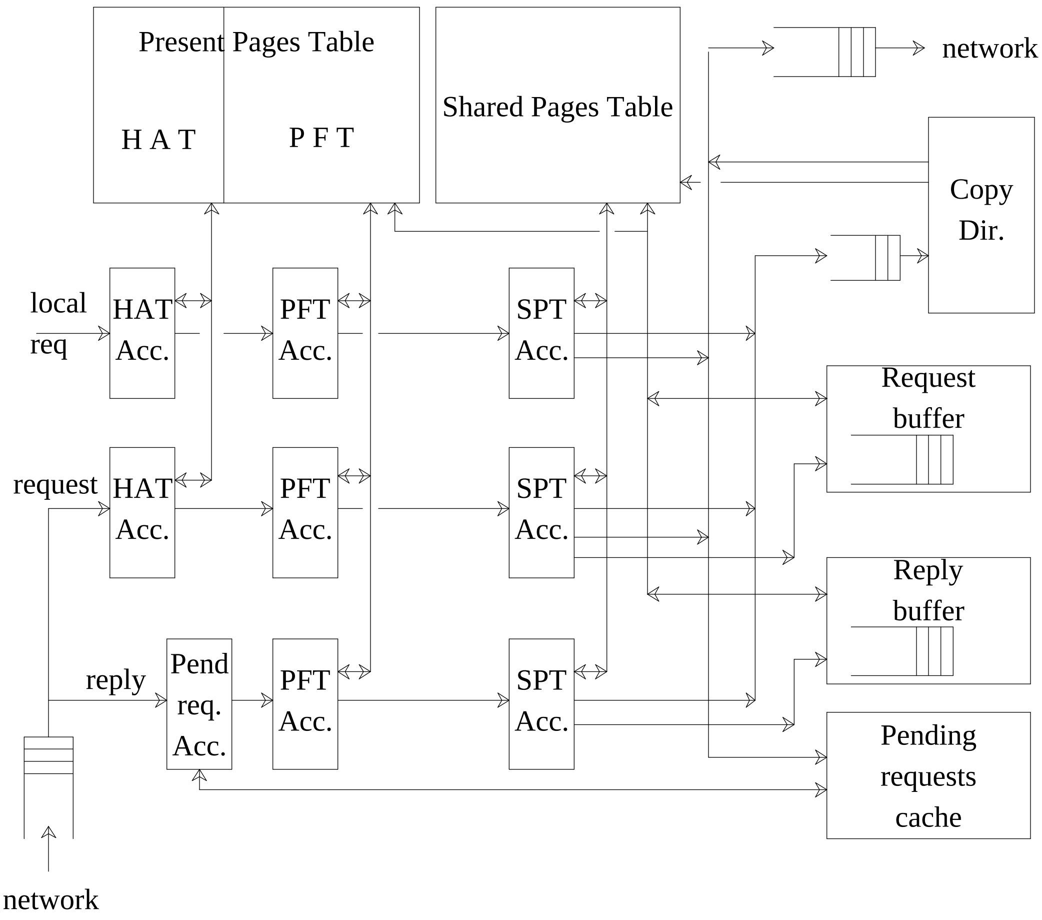


Figure 4.1: Memory Management Unit Design.

Local and remote request servicing may also require copy directory lookup. The copy directory access is internally pipelined. However, unlike the PPT and SPT, only one request can be issued to the copy directory at a time. Since requests that require copy directory access typically also incur costly remote access latencies (messaging), more sophisticated support to parallelize copy directory access may not be beneficial.

Local processor requests and remote requests can generate other network messages which require replies from other nodes in the system (e.g., invalidation on a write request). Such local and remote requests must be suspended until their replies arrive, to allow the MMU to continue servicing other local and remote requests. In addition, replacement events to free up entries in the PPT, SPT or the copy directory in the local node may also generate network requests which require replies (e.g., invalidation to free up a copy directory entry). The state of local and remote requests that are suspended waiting for remote replies as well as eviction events which are waiting for remote replies are maintained in the Pending Requests Cache (Figure 4.1).

The state maintained in the Pending Requests Cache allows replies received for the messages sent to be serviced with a minimum number of additional mapping table lookups. The state information maintained in the Pending Requests Cache identifies the suspended request. It indicates the type of the request (read, write or local replacement event), the address accessed by the request and the source of the request (local node or identity of remote node). In addition, the entry for the request maintains the page frame number of the page accessed by the request (PPT entry number), the type of message(s) that were sent out, the remote node to which the message was sent, and the number of replies (e.g., number of invalidation acknowledgments) expected. Note that all fields in the entry may not be used in all cases. For example, if many invalidations were sent (to perform a write or a replacement), the field identifying the node to which the message was sent is not used. The field identifying the page frame number may not be used when the entry corresponds to the eviction of a copy directory entry.

When a reply message is received from the network, the Pending Requests

Cache is looked up associatively (Pend req. Acc. in Figure 4.1) by matching with the address field of the reply. If the reply allows the suspended request to proceed, (e.g., all acknowledgements have been received) the MMU removes the suspended request from the Pending Requests Cache and completes servicing it. The MMU processes replies concurrent with local requests and remote requests. Reply servicing is also pipelined in the MMU. If the PPT and SPT need to be updated to service the suspended request, the updates are performed in a pipelined manner.

To service a remote request or a reply to a local request, an entry may need to be allocated in the Shared Pages Table or in the copy directory. If an entry is not available for allocation, service to the request or reply must be suspended until an entry becomes available (mechanisms to free entries will be discussed shortly in Section 4.1.1). The request or reply is queued. As shown in the block diagram (Figure 4.1), replies and requests are queued in independent buffers. When an SPT or copy directory entry becomes free, any request or reply that has been suspended and queued is serviced. The reply queue is prioritized higher than the request queue so that data can be provided to the local processor quickly.

The functionality of the MMU and MMT (Section 3.1) chips is similar to the functionality of the directory controller (DC) and the reply controller (RC) boards in the DASH prototype [Leno93], of the CMMU chip in the Alewife architecture [Agar95], and of the MAGIC chip in the Stanford FLASH multiprocessor [Kusk94].

4.1.1. Present Pages Table and Shared Pages Table

The MMU refers to mapping and coherency information when the local processor issues a memory access due to a miss in its cache and when a request is received from a remote node. As discussed in Section 3.2.3 of Chapter 3, the total size of the mapping tables (PPT and SPT) is on the order of a few hundred thousand bytes. Hence, it may be possible to store them on chip with the MMU in order to speed up accesses. The tables can also be maintained off-chip in external memory. The physical organization of the PPT and SPT is straightforward [Chan88] and similar to the logical structure shown in Section 3.2.1 (Figure 3.4). The MMU can exploit locality by caching this information in a Translation Lookaside Buffer (TLB). As in conventional systems, the TLB provides fast translation of virtual to physical page numbers. Here, it also provides the full SPT entry for shared pages. With two read ports and one write port, the TLB can service local and remote requests concurrently. We have not investigated the impact of the size and organization of the TLB, and do not show it in the block diagram in Figure 4.1.

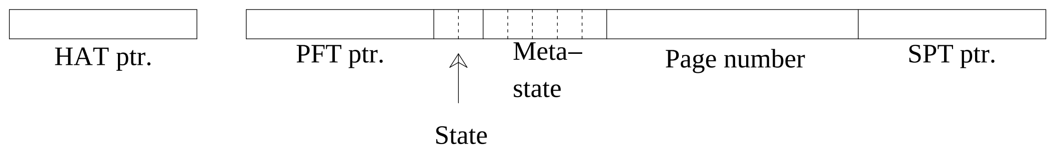


Figure 4.2: Present Pages Table (PPT) entry format

The PPT entry format is shown in Figure 4.2, which is similar to the logical structure shown in Figure 3.4. Each entry maintains two pointer fields used in the inverted table organization, one part of the Hash Anchor Table (HAT) and the other part of the Page Frame Table (PFT). In the inverted table organization, each table entry is the start (i.e., anchor) of an hash bucket, and the entry itself may dynamically be part of another hash bucket chain. The HAT pointer field identifies the first PPT entry in the hash bucket anchored at this entry. The PFT pointer field points to the next PPT entry in the hash bucket chain to which this entry belongs. When a PPT entry is not allocated in any hash bucket, the PFT pointer is re-used to maintain a list of free PPT entries. The usage of each PPT entry is indicated by the state bits maintained in the entry. Two bits of entry state identify if the entry is invalid, valid, or if the entry is allocated pending data from a read request or write request. When a page entry is valid, additional information about the page is maintained by a few meta-state bits to assist in situations such as page replacement. Two of the five meta-state bits identify if a valid page is private or shared, and if the page is clean or dirty. The other three meta-state bits identify whether the page is pinned from being replaced, whether the page is being replaced to free up its PPT entry and whether the page is being replaced to free up its SPT entry. If the entry is valid, the page number field identifies the virtual page number of the page mapped to this entry. If the page is shared, a pointer to this page's entry in the second level Shared Pages Table (SPT) is maintained.

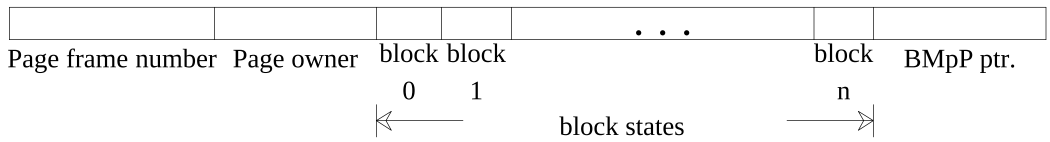


Figure 4.3: Shared Pages Table (SPT) entry format

Figure 4.3 shows the SPT entry format. Each entry identifies the owner of the shared page. It also identifies the physical page frame to which the page is mapped. When the SPT overflows and an SPT entry is evicted, the state of the page in its PPT entry must be appropriately modified. The page frame number maintained in the SPT entry serves as a back pointer to the PPT, allowing the identification of the PPT entry used by the page. The SPT entry also maintains the state of each block of the page, and potentially whether the blocks are dirty or clean. The state of each block identifies the type of accesses (read or write) that are permitted for the local processor. If a request has been issued to the network to service an access to the block, the block state also identifies the access that is pending on the block. We discuss the actual states of blocks along with the discussion of the coherence protocol in Section 4.3.4. The SPT entry also maintains a pointer to its BMpP copy directory. If the SPT entry is not currently used, this pointer is used to maintain a chain of available SPT entries.

An entry is allocated in the PPT when an access by the local processor misses in the PPT. A read or write request is sent out to the network and the PPT entry state is updated to indicate that read or write access has been requested. No entry is allocated in the SPT at the time the request is sent out. Rather, an SPT entry is allocated when the reply arrives from the network, if the reply indicates that the page is shared. At that time, the state of the PPT entry, the SPT pointer field in the PPT entry, and the state of the block in the SPT entry are all suitably modified. The PPT entry state bits are set to valid, and the page is marked as shared in the meta-state bit of the PPT entry. The SPT pointer field of the PPT entry is set to point to the SPT entry allocated for the page. The state of the block in the SPT entry is updated to identify the access permission granted by the reply message.

An SPT entry is also allocated when a request from a remote node makes a page which is currently private, shared.

If no entry is available in the PPT when a local processor access requires the allocation of a new entry, the local processor is blocked. The processor is unblocked when a entry becomes available due to invalidation or replacement. If no free SPT entry is available when required by a reply from the network (to a request previously issued from this node) or when a request from a remote node requires one, the reply or the request is buffered. The buffered request or reply is serviced when an SPT entry is freed by eviction or invalidation. To minimize blocking and queuing, we employ premature eviction, with both the PPT and the SPT, to maintain some free entries. When the usage of these tables goes above a threshold, we pick a victim and initiate its replacement.

When a PPT entry must be freed, the victim page must be migrated to another node or disk. If page ownership is not allowed to change, owned pages can only be evicted to disk. Since this incurs significant performance impact, it is preferable to evict pages that are not owned locally. However, if the PPT entry of an owned page must be freed, it is preferable to evict private pages rather than shared pages. When the victim page is a shared page, the corresponding SPT entry must also be freed which additionally requires interactions with other nodes.

It may be necessary to evict an SPT entry due to the eviction of the corresponding local PPT entry or in order to allocate the SPT entry to a different page. As described previously, if the ownership of the page cannot change, it is preferable to evict a page that is not owned locally to avoid disk access. The SPT entry of a page not owned by the local node, can be freed by ‘‘pushing’’ away the page. All blocks of the page that are owned locally can be written back to the page owner, and all blocks that are held read-only can be dropped. If the SPT entry of a page owned locally must be freed, all blocks of the page must be invalidated from all other nodes in the system. All blocks of the page can be ‘‘pulled’’ from the remote nodes, by commanding all nodes that own blocks of the page to write-back the blocks.

4.1.2. Block Map per Processor Copy Directory

A lookup in the Block Map per Processor (BMpP) copy directory may be performed in order to respond to a remote request or as part of an access initiated by the local MMU. A remote request may request a copy of a locally-owned block or send a notification that it has deleted its copy of a locally owned block. The local MMU may require a copy directory lookup in order to find the locations of all the copies of a locally-owned block or in order to find the owner of a block that is not locally-owned. With the first type of copy directory access, the directory must be searched sequentially based on the processor identifier. Once a match is found (or a new entry is allocated), the corresponding bit vector is updated. For the second type of access, the directory is accessed directly using the block offset within the page. In this case the desired result is a column bit vector from the table shown in Figure 3.4, i.e., from each directory entry, the one bit that corresponds to the specified block. Once the bit vector is read, it is used to obtain the processor identifiers corresponding to ‘‘1’’ entries.

The physical organization of the BMpP copy directory can be geared to the above access requirements by organizing it as two separate banks — one

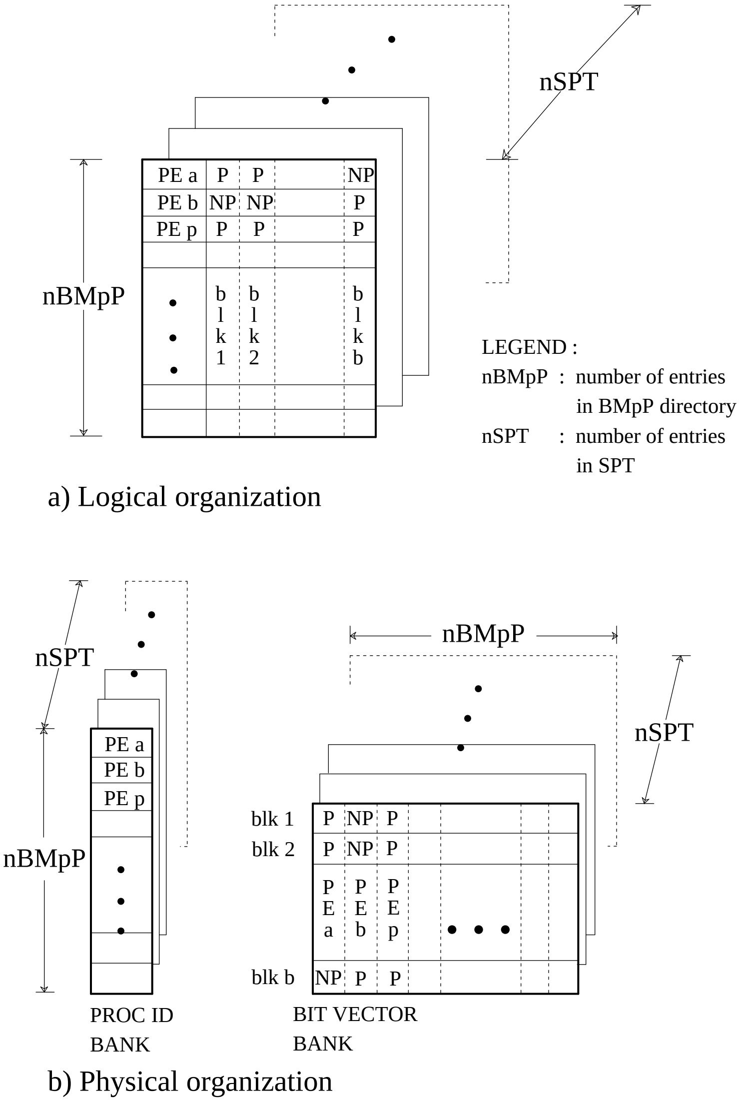


Figure 4.4: BMpP copy directory organization.

maintaining the processor identifier field of all copy directory entries and the other maintaining the bit vectors (Figure 4.4-b). Both types of copy directory accesses discussed earlier require access to the processor identifier and the bit vector banks. Since these banks are physically independent, the banks can be concurrently accessed. This also allows copy directory accesses to be pipelined.

The bit vector bank is also transposed with respect to the logical structure of the copy directory (Figure 4.4-b versus Figure 4.4-a). Each word in this bank (or multiple words, depending on the word length of the bank and the number of entries in the copy directory) maintains the set of bits associated with a *specific* block of a page (a row in Figure 4.4-b and a column in Figure 3.4). Unlike the logical organization, this organization maintains the bit map associated with any block in contiguous bit positions. Hence, copy directory accesses to locate copies of the block can be performed efficiently by reading fewer words than if the logical structure was used.

Copy directory accesses can be further speeded up by special hardware support such as associative lookup of the processor identifier bank and pipelining in the bit vector bank access. However, since a memory reference that involves a copy directory access must also involve network communication, the copy directory access latency may be overshadowed by the network communication latency. Thus, additional hardware support to speed up copy directory access may not be justified.

If a fixed number of BMpP entries are allocated per shared page, the copy directory may not be able to simultaneously accommodate all the processors that are accessing a particular block. As with other limited copy directory schemes [Agar88], contention for BMpP entries may be resolved by eviction of occupied entries. Where such a policy is inefficient (e.g., for read-only shared pages), an unlimited number of simultaneous copies may be supported by requiring system-wide broadcasts to invalidate all the copies. A *broadcast flag* in the SPT entry can be used to indicate that broadcasts are required.

Techniques such as the dynamic pointer allocation [Kusk94] scheme used in the Stanford FLASH multiprocessor can also be employed here. BMpP entries can be allocated in batches from a memory bank of BMpP entries shared by all pages. As implemented in [Kusk94], BMpP entries can be chained, and BMpP entries that are freed can be returned to the free list in the memory bank. We did not investigate the implementation of the dynamic pointer allocation scheme for BMpP copy directories.

The Alewife machine designed at MIT [Chai91, Agar95] allows the copy directory to overflow into software. When the number of copies of a block goes over the static bound of five, the copy directory is maintained by software, and all remote requests to the block are handled by software. Such software support schemes are also applicable for the BMpP copy directory scheme, but have not been investigated.

4.2. Distributed Directory Coherence Protocol Design Issues

We have investigated the implementation of a distributed directory coherence protocol for maintaining the node memories coherent. Following most current DSM implementations [Leno90, Kusk94, Agar95], we chose an invalidation-based ownership protocol. Nodes must obtain exclusive ownership of the block before modifying it, by invalidating all other copies of the block. Multiple read-only copies can be distributed throughout the system.

As we discussed earlier in Section 2.2, the design of distributed directory coherence protocols is more complicated than the design of snoopy protocols. In a shared bus multiprocessor using snoopy coherence protocol, the bus can be used by only one processor at a time, while all processors can snoop on the bus activity. As a result, the shared bus serves as a mutual exclusion lock ordering accesses by different processors.

In the multicomputer system, multiple processors can simultaneously send requests out to the network. These requests can arrive out of order at the destination if the network does not guarantee FIFO delivery of messages between any pair of nodes. Even when the network guarantees pair-wise FIFO delivery of messages, such race conditions can occur if the coherence protocol requires requests to be forwarded by intermediate nodes (such as when a request is forwarded by the page owner to the block owner).

Distributed directory coherence protocol design must also address the potential for deadlock. Since the directory information is distributed, accesses can require several request and reply messages to be exchanged. Hence, accesses are not inherently atomic and they progress by allocating distributed directory resources incrementally. Directory resource requirements of conflicting accesses can, therefore, lead to a cycle of dependencies resulting in deadlock.

In the rest of this section, we discuss the issues in implementing a distributed directory coherence protocol and our solutions to these issues. In our discussion, we refer to specific states, and specific message types used in the coherence protocol. While the semantics of the states and the message types used here will be clear from the context of the discussion, they are formally defined in Sections 4.3.2 and 4.3.3.

4.2.1. Deadlock

The potential for deadlocks exists in any system where there are multiple operations in progress concurrently and where these operations require use of shared resources. Specifically, deadlocks may occur when operations can obtain exclusive access to a subset of the resources they require for completion and maintain this exclusive access until completion. The problem occurs when the additional resources required for completion are held by another operation which, in turn, is waiting for shared resources to be freed. The design of the system must prevent the possibility of a *deadlock* where multiple operations are unable to progress due to a cycle of resource dependencies [Silb94].

In the DSM system, requests serviced by the coherence protocol compete for logical and physical resources. Requests compete for logical resources such as read access permission and exclusive access permission. For example, a write request can compete for exclusive access to a block with other read requests and write requests to the same block. To complete the write request, the node issuing the request must invalidate the access permissions of all other nodes in the system. Since these logical read access permissions are distributed through the system, they can be invalidated only incrementally. When concurrent writes to the same block compete for control of all read access permissions (to invalidate them), a cycle of dependencies can manifest leading to deadlock.

Similar cycle of dependencies can also occur when requests compete for physical resources such as table entries and buffer space. When a request is generated in a node, storage may be allocated on that node to maintain information associated with that request. When this protocol request is received at the destination node, other child protocol requests may need to be generated (e.g., a writeback request may initiate generation of invalidate messages). This requires allocation of storage in the destination node also. The parent request progresses by incrementally acquiring storage in several nodes, and its completion depends on the availability of sufficient storage in all these nodes. Competition for the distributed storage resources amongst concurrent coherence protocol requests can also lead to cyclic dependencies and deadlock.

We integrate several mechanisms into our coherence protocol to address the problem of deadlock in DSM systems. We prioritize invalidate commands and ensure that invalidate messages are always consumed. As a result, invalidate messages do not participate in any deadlock. This is similar to the approach in the DASH [Leno90] and FLASH [Kusk94] directory coherence protocols where reply messages are always consumed. Our coherence protocol also prioritizes the owner over other nodes. When requests require conflicting access to the same block, all nodes uniformly grant the request issued by the owner. In addition, our coherence protocol uses negative acknowledgements during contention or resource scarcity to avoid deadlocks. When a request cannot be serviced, a negative acknowledgement is sent to the requesting node and resources allocated for the request are freed. Our mechanisms for handling deadlock may not be sufficient to handle all deadlock situations that are possible in DSM systems. When new deadlock situations arise in a full-scale implementation, additional mechanisms may be necessary to handle them. In the rest of this subsection, we discuss two examples illustrating the occurrence of deadlock in DSM systems and our method of handling them.

Consider the example shown in Figure 4.5. Two processors A and B are

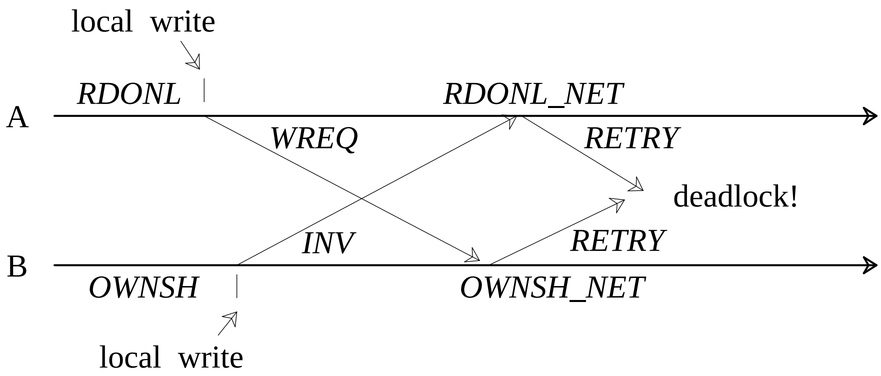


Figure 4.5: Deadlock due to competing writes. Processor *A* does not service *B*'s invalidate request (inv) due to its pending write, and processor *B* does not service *A*'s write request (wreq) due to its pending write resulting in deadlock.

attempting to write to the same block. Initially, processor A has a read access permission to the block, and processor B is the owner of the block. With the invalidation-based ownership protocol, exclusive access permission and ownership must be obtained by invalidating all other read copies before writing the block. In this case, processor A issues a write request (*WREQ*) to the owner B to obtain exclusive ownership for its write. Simultaneously, processor B sends an invalidate message (*INV*) to processor A to proceed with its own write. Suppose each processor refused to service the other processor's request (using a *RETRY* message) due to its own pending request. In this case, each processor retains its access permission and must wait to acquire control of the other's access permission. This leads to a cycles of dependencies. Hence, neither write access would progress, and the accesses would be deadlocked.

To avoid this deadlock, one of the two competing accesses should be allowed to progress. This can be accomplished by giving higher priority to the current

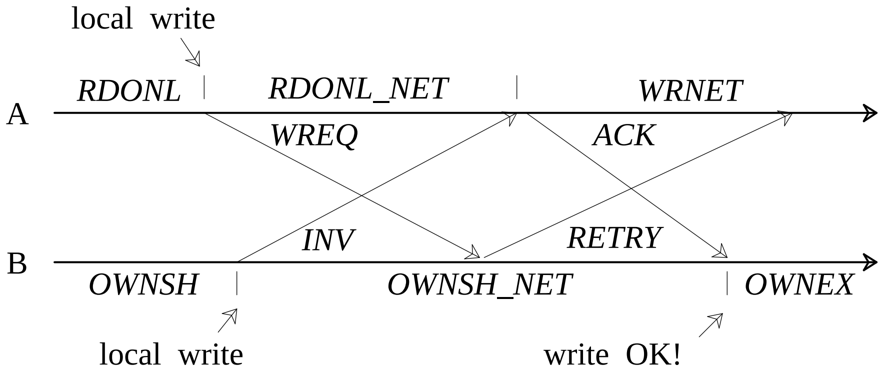


Figure 4.6: Deadlock avoided by ‘‘sink’’ing invalidate. Deadlock is broken by requiring that invalidate messages (*INV*) cannot be denied service. Processor *B*’s write access will succeed first. Subsequently, processor *A*’s write will be retried and will complete successfully.

owner of the block. Specifically, in our implementation, *invalidate* request messages from the owner are handled and ‘‘obeyed’’ without delay. The modified handling of the example is shown in Figure 4.6. Processor A invalidates its read copy and sends an acknowledgement to the owner B, while still maintaining its pending write. Processor B will, hence, complete its write access, and it will subsequently service processor A’s write request.

Dependencies may also be created between processors due to resource constraints, which may lead to deadlocks [Leno90]. For example, consider the following scenario summarized in Table 4.1. Processor B owns page *p1* containing block *b1*. No block from *p1* is present in processor A. Processor A owns page *p2* containing block *b2*. Processor B owns block *b2*. Processor A possesses a read-only copy of *b2*. Processor A executes a load from block *b1*. As a result a read only copy of block *b1* is requested from processor B. Processor A receives a read copy of block *b1*, which it buffers since no free SPT entry is available. Page *p2* is picked for eviction to free up an SPT entry. To evict this page a message is sent to processor B, which owns block *b2* of the page, requesting it to write-back block *b2*. Concurrently, processor B issues a write access for the block *b1*, and sends an invalidate message to processor A. Processor B records its pending write to block *b1* in its pending request cache in order to collect invalidation acknowledgements. To service the write-back request from A associated with the SPT eviction, processor B must invalidate copies of block *b2*. This invalidation can proceed only if there is space in the pending request cache of processor B since the pending write-back of block *b2* must be recorded in the pending request cache of processor B (again, to collect invalidation acknowledgements). Table 4.1 lists the sequence of events at each node, the action initiated by the event and the dependency created by the action.

If the invalidate for block *b1* in processor A cannot be processed until its SPT eviction is completed and the freed SPT entry is allocated to the buffered block *b1*, a cycle of dependencies can manifest as follows. The invalidate for block *b1* cannot be processed by processor A until the SPT eviction of page *p2* is completed (dependency 1 in Table 4.1). The SPT eviction cannot progress until space is available in the pending request cache of processor B for its invalidate of block *b2* (dependencies 2 and 4 in Table 4.1). Space will not become available in this pending request cache until A services the invalidate of block *b1* (dependencies 3 and 5 in Table 4.1). This cycle of dependencies (1→2→4→3→5→1 in Table 4.1) leads to a deadlock. In our implementation, we break this dependency by allowing the invalidate message to invalidate the buffered reply without waiting for the SPT eviction to complete.

Node	Event	Consequent Action	Resulting Dependency
A	*b1* read copy is received; buffered due to full SPT	eviction of page *p2*'s SPT entry is initiated	*b1* copy buffered until completion of SPT entry eviction (1)
A	eviction of page *p2*'s SPT entry	request node B to writeback block *b2* of page *p2*	*p2*'s SPT entry freed when *b2* is written back by B (2)
B	local write request for *b1* must invalidate A's copy	invalidation is allocated storage in Pending Requests Cache	Pending Requests Cache storage freed when A sends acknowledgment (3)
B	request to writeback *b2* received — must invalidate copies of *b2*	invalidation is suspended due to full Pending Requests Cache	*b2*'s writeback can progress if space freed in Pending Requests Cache (4)
A	invalidate request for *b1* received from B	invalidation is suspended until *b1* is unbuffered	A can acknowledge *b1*'s invalidation when *b1* is unbuffered (5)

Table 4.1: Deadlock due to contention for physical resources. The sequence of events, consequent action, and resulting dependency at each node are shown. Deadlock occurs due to the cycle of dependencies 1→2→4→3→5→1. The deadlock is avoided in our implementation by allowing invalidate messages to invalidate buffered read copies. This breaks the cycle by eliminating dependency 5.

4.2.2. Handling of Race Conditions

In a system that supports multiple concurrent processes, the outcome of the execution may depend on the order of accesses to shared resources. This order may vary from one execution to the next, depending on the access times to the memory system (which may vary due to caching) or variations in the communication latencies through the network. A *race condition* is present if the outcome can vary from one execution to the next due to these causes. The system must either ensure that all possible outcomes are ''correct'' or use explicit locking and synchronization to prevent incorrect or undesirable ordering of accesses to shared resources.

Race conditions can occur in a shared bus multiprocessor with snoopy caches [Katz85]. In systems based on snoopy caches, the ‘‘races’’ of concern are between accesses to the cache from the local processor and remote accesses across the bus. Many more races are possible in a system where the nodes are interconnected by a point-to-point network. Specifically, in such a system messages may be delivered in a different order than they were sent. If the network provides multiple paths between pairs of nodes and does not guarantee pairwise FIFO (First-In-First-Out) delivery of messages, it is obvious that messages can race in the system. Even when the network guarantees pairwise FIFO ordering of messages, coherence protocols can themselves introduce race conditions. Coherence protocols (such as in [Leno90], for example) may require different messages to be sent through different sequences of nodes. For example, some messages may be sent directly to the servicing node, while other messages may be processed by intermediate nodes before reaching the servicing node. The underlying network is oblivious to the coherence protocol-level routing. Hence, even when the network guarantees pairwise-FIFO, messages from the same node can reach the servicing node in different order than in which they were generated. We discuss this race problem associated with coherence protocols in more detail below.

Coherency information is distributed among several nodes with the distributed directory coherence protocol. As a result, a node that requires access to a block may not be able to identify the owner of the block from its local fragment of the coherence directory. To acquire access to the block the node must contact the node that can identify the block owner. Traditional coherence protocols employ a technique called *forwarding* to facilitate this step. With the use of forwarding, the request message is first sent to the node that maintains the identity of the owner. This message is subsequently *forwarded* by this intermediate node to the node that owns the block. However, protocols typically do not always use the forwarded route. For example, when a message must be sent replying to a message from the owner, the reply message may be sent directly to the owner without forwarding through additional nodes. These protocols selectively determine whether a protocol message uses the forwarded route or the direct unforwarded route. We refer to such protocols as protocols based on *selective forwarded-routing*. Since requests from one node to another may follow different paths, protocols with selective forwarded-routing introduce race conditions.

To illustrate the use of selective forwarded-routing, consider the handling of a read request and an acknowledgement in the DASH multiprocessor [Leno90]. In DASH, when a cluster has a read miss, the cluster sends a read request to the home cluster of the block. If the block is owned dirty by a remote cluster, the home cluster forwards the request to the dirty, owning remote cluster (use of forwarded-routing). This owning cluster sends the data directly to the requesting cluster (see [Leno90] for a definition of home and remote clusters). However, when the cluster with the read copy receives an invalidate message from the owning cluster, the acknowledgement reply is sent directly to the owning cluster. Unlike the original read request, the acknowledgement message is not forwarded through the home cluster (forwarded-routing is not used).

Whether the coherence protocol selects the forwarded route or the

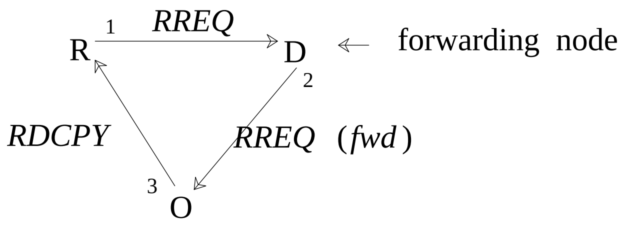


Figure 4.7: Forwarding in Protocol. Node *D* forwards node *R*'s request to node *O*.

unforwarded route can also depend on the state of the protocol rather than on the type of the message. For example, forwarding can be used only when a node initiating a request does not know the owner of a block. In Figure 4.7, the node *R* does not know the current owner of a page, and, hence, sends its read request to the default owner *D*. The default owner *D* knows the current owner *O*, and it forwards the request to *O* who sends a reply directly to *R*. For better performance, *R* can cache the identity of the owner when it receives the reply to its first request, and subsequent requests can be directly sent to the owner, without using forwarding (selective forwarded-routing).

We illustrate the occurrence of races in DSM systems due to the use of forwarding (since the use of the forwarded-route is selective in all protocols that we are aware of, we simply refer to these protocols as protocols that use forwarding) in the next sub-section.

4.2.2.1. Race Conditions in DSM Systems

We have previously observed that read requests and acknowledgements can follow different paths in the DASH multiprocessor [Leno93]. Hence, in DASH, a race condition is possible between a read request and an acknowledgement message sent from a node. Suppose a cluster *R* has a read copy of a block, which gets invalidated due to a write by cluster *O*. Subsequently, cluster *R* requests a read copy of the block. The home cluster of the block is *H*. As we discussed previously, in DASH [Leno90], *R* sends its acknowledgement for the invalidate directly to *O*. However, *R*'s subsequent read request is sent to *H*, which can be forwarded to *O* for service. Since the acknowledgement and the read request follow different paths from *R* to *O*, the read request can race the acknowledgement to *O*. When the read request is received ahead of the acknowledgement, the situation is equivalent to the case where a read request is received while a write is in progress. Hence, the race condition can be handled by queuing the read request or denying service (negative acknowledgement) as a read request received during a write would be handled.

A race condition such as the one discussed above can also occur in our DSM system if forwarding is used. We describe a situation in our DSM system that is slightly modified from the previous example. In Figure 4.8, node *R* initially requests and receives a read copy of a block from owner *O*. Subsequently, node *R* drops its read copy (replacement) and sends a copy invalidated message *CPINV* (message types are discussed in Section 4.3.3) to update the owner *O*. When node *R* subsequently needs to read the block, it does not know the current owner and, hence, sends the read request (*RREQ*) to the default owner *D* who forwards it to the

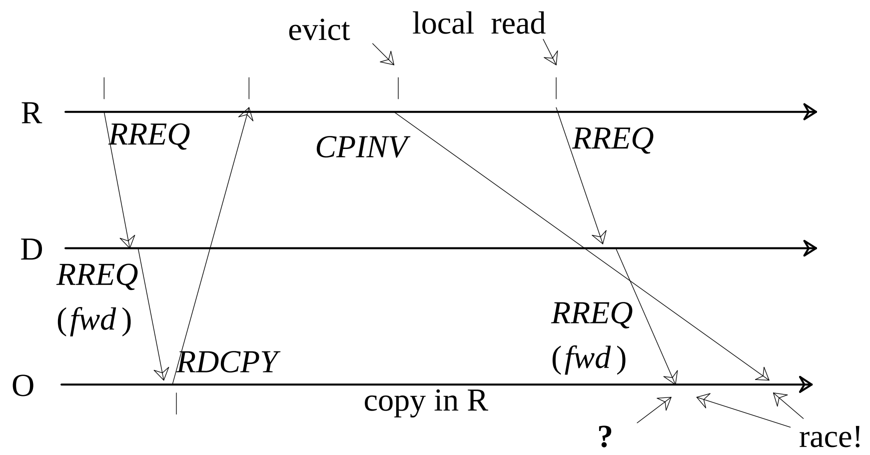


Figure 4.8: Races in protocols with forwarding. Node *R*'s read request is forwarded by node *D* to node *O*. The read request sent after the copy invalidated message *CPINV* can arrive at node *O* earlier since it followed a different path due to forwarding.

owner *O*. As with the example in DASH, the messages *CPINV* and *RREQ* follow different paths from node *R* to node *O*. Consequently, the *RREQ* can race the *CPINV* message, and arrive first at *O*. If this occurs, node *O* would receive a read request from a node that, in its view, already has a copy of the block. The arrival of a read request message from a node that already possesses a copy of the requested block can pose a problem for the coherence protocol.

In the example scenario of Figure 4.8, it may be possible for *O* to identify the race condition, and act appropriately. If *O* maintains a copy directory, it can identify the race condition by observing that the *RREQ* was sent by a node *R* that is already present in the copy directory. Node *O* can resolve the race by denying the service for the read request (force a retry using a *RETRY* message). In general, however, it may not be possible to identify a race condition purely from the local state.

As an example of a race condition which cannot be correctly resolved based on local information, consider a modification to the example shown in Figure 4.8. In the modified scenario, shown in Figure 4.9, node *O* is forced to evict node *R* from its copy directory (due to copy directory overflow). To perform this eviction, node *O* sends an invalidate message (*INV*) to node *R*, and drops node *R* from the copy directory. Since node *R* is no longer identified in the copy directory, when node *O* receives node *R*'s read request *RREQ* ahead of its copy invalidate message *CPINV*, node *O* <u>cannot</u> identify that a race has occurred. From its local state, node *O* cannot distinguish if the *RREQ* raced another message, or if it is a request from a new node. Node *O* will provide a read copy to the read request. This can cause the system to enter an incorrect state, as discussed below. The messages in transit at this point are the copy invalidated message (*CPINV*) from node *R* to node *O*, and the invalidate (*INV*) and read copy (*RDCPY*) messages from node *O* to node *R*. Upon receiving the copy invalidated message, node *O* will (erroneously) drop node *R* from its copy directory. Meanwhile, node *R* will acknowledge (*ACK*) the invalidate message, and subsequently receive the read copy. After node *O* receives the acknowledgement (*ACK*), there are no messages in transit. Since, node *R* possesses a read copy, and node *O* does not record this in the copy directory, the copy directory state is incorrect. This can allow future writes to proceed without invalidating the read copy of node *R*, violating the write exclusive protocol. This specific race condition may not be a problem in the coherence protocols used in [Leno93, Agar95, Kusk94] since they employ a full copy directory without need for eviction. Such race conditions have not been discussed in their literature. As will be discussed in Section 4.2.2.2, our coherency protocol avoids this problem by using ‘‘bilateral exchanges.’’ With this approach, the read request message REQ from node *R* cannot race its earlier copy invalidate message *CPINV*.

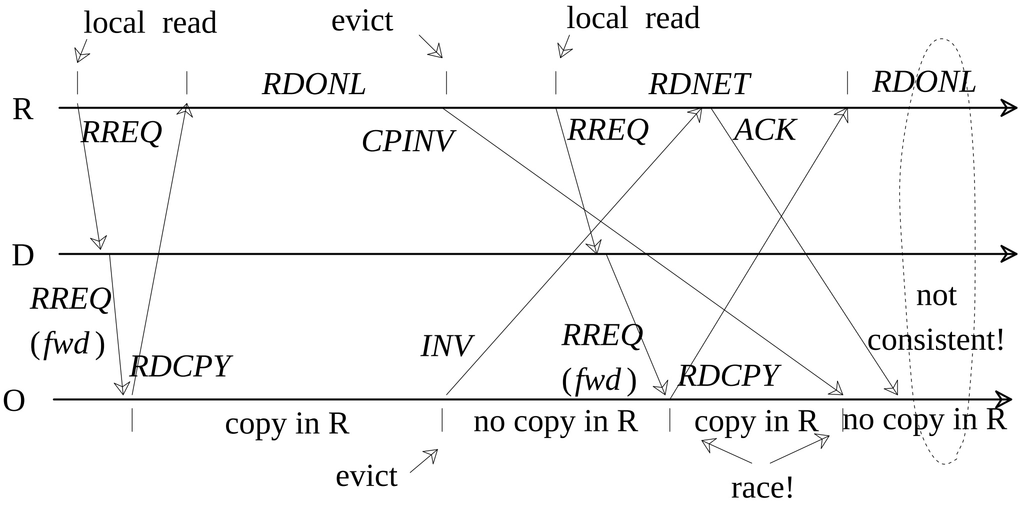


Figure 4.9: Race condition that cannot be locally identified. Node *O* evicts node *R* from its copy directory, and, hence, cannot locally determine that *R*'s read request raced its copy invalidated message. In the end, although *R* has a read copy, the copy directory in owner *O* does not reflect this. The copy directory state is incorrect.

The designers of the DASH prototype [Leno90] also discuss race conditions caused by the use of two independent networks in their system. DASH uses two networks, one for requests and one for replies. Since requests and replies use different networks, races between request and reply messages are possible. For example, when a node sends a read copy reply to another node and subsequently sends it an invalidate request, the invalidate requests can race the preceding read copy reply. In DASH, such race conditions are handled by designing the protocol engine to identify race conditions and providing special mechanisms to process them. To handle the race between an invalidate request and a read copy reply, an additional state is added to the coherence protocol. When an invalidate is received for a block that has not yet been received, the block is changed to a state which identifies that the block must be invalidated when it is received. When the read copy is received, it is discarded. The block is changed to the invalid state, and an acknowledgement reply is sent to the owner.

In summary, protocols that employ forwarding involve race conditions. This possibility of races (even in a network with FIFO service discipline) presents an obvious problem in the design of correct distributed directory coherence protocols. The protocol design needs to identify possible race conditions and develop mechanisms to avoid them, or develop mechanisms to detect and handle them. This makes the protocol design and implementation complex. Races can also mask implementation bugs and runtime error conditions, which makes design verification and error detection difficult. We discuss a new protocol technique that addresses the race problem in the next sub-section.

4.2.2.2. DSM Coherency based on Bilateral Exchanges

As discussed in the previous sub-section, many race conditions are introduced by the use of forwarding. The problems caused by these race conditions provide the motivation for developing coherency protocols that avoid forwarding. In this sub-section, we introduce a coherency protocol where all exchanges between nodes are *bilateral*, i.e., do not involve any forwarding. In a protocol based on *bilateral exchanges*, if a node cannot service a request itself, it will send an appropriate reply message back to the requester, and not forward the request to any other node. For example, consider the example where the default owner forwards a read request to the current owner node. This protocol interaction is modified to use bilateral exchanges as shown in Figure 4.10. Since the default owner *D* cannot

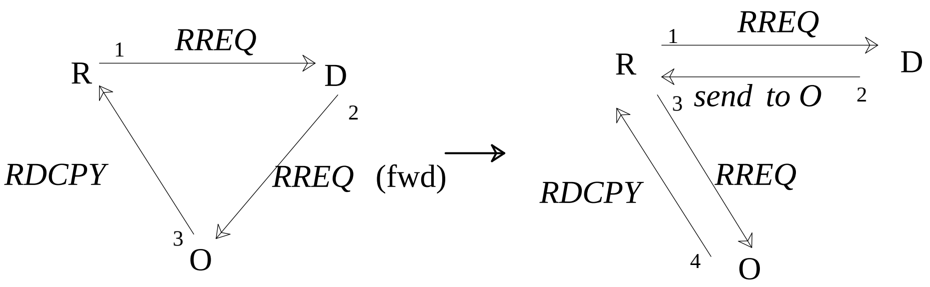


Figure 4.10: Protocol based on bilateral exchanges. The protocol exchange on the left uses forwarding. The modified protocol exchange using bilateral exchanges is shown on the right. All protocol exchanges between nodes are bilateral. Node *D* identifies the owner *O* to the requester *R*, instead of forwarding the request. *R* requests a read copy from the owner *O* bilaterally.

service node *R*'s read request itself, it sends a reply back to node *R* identifying the current owner *O*. The exchange between node *R* and node *D* is bilateral. After receiving the response from the default owner, node *R* has another bilateral exchange with the current owner of the block *O* to receive the read copy.

The bilateral exchange technique is designed to avoid races in networks that guarantee pairwise FIFO ordering of messages. Since all exchanges in the protocol are bilateral, i.e., pairwise, protocol messages exchanged between any pair of nodes will always arrive in order. Figure 4.11 shows how a race situation such as the one described earlier in Figure 4.8 is not possible in a protocol based on bilateral exchanges. Since node *O* receives the copy invalidated message *CPINV* and the read request message *RREQ* directly from node *R*, the network's FIFO service discipline guarantees that they will be received in the exact order they were sent.

The absence of message races with bilateral exchanges makes the design and implementation of the protocol simpler. Verification of the design and detection of

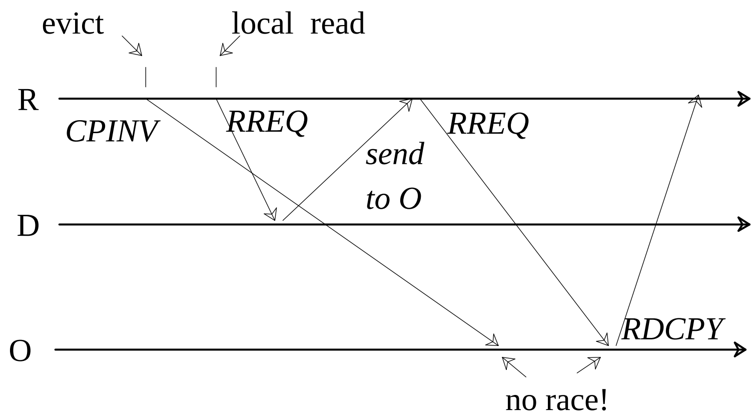


Figure 4.11: Protocol based on bilateral exchanges avoids races. The copy invalidated message and the read request from *R* follow the same path to *O* due to the bilateral exchange. With the network guaranteeing pairwise FIFO, these messages will arrive at *O* in the order they were sent.

runtime errors will also be simplified. However, while the protocol based on bilateral exchanges is simpler, it is less efficient. Comparing Figures 4.8 and 4.11, it is clear that the number of messages and the number of host processing delays will be higher with the usage of bilateral exchanges. In our protocol implementation (described later in this chapter), we use an hybrid mechanism to reduce the inefficiency due to bilateral exchanges.

4.2.3. Necessity of Transient Block States

The coherence directory in each node must maintain the state of the blocks in that node. The state of each block identifies whether the block is invalid, readable or writable. To change the state of a block, a request must be sent to a remote node requesting the new access permission. The block state can be changed only after a reply message granting the request has been received. Since this request-reply protocol exchange is not atomic, additional block states may be needed to maintain the transient state of the blocks.

Maintaining transient block states clearly provides several benefits. Since transient states identify outstanding requests, they allow the coherence protocol to verify whether a reply message that is received is a valid reply. Second, if the reply message received denies the requested access (negative acknowledgement), the transient block state allows the coherence protocol to determine the type of the request message that must be re-issued. Transient block states are also useful in systems where multiple processes can execute on a node. When a second process requests similar access to the block, the coherence protocol can identify that a request for the access has already been issued to the network. A second request need not be sent to the network. Thus, the transient state allows the coherence protocol to filter requests to the network.

Transient block states are also a compulsory requirement in some cases. Certain accesses to a block, such as a write access, require that concurrent requests received from other nodes for competing or conflicting accesses be denied service. A transient block state that identifies the specific access in progress for the block is necessary to allow the coherence protocol to refuse service to competing accesses. Transient block states are also required in the coherence protocol if messages that cannot be denied service can be received at a time when they cannot be immediately serviced (see the second example later in this subsection). In such cases, a transient state must be maintained indicating that the message has been received. This allows the coherence protocol to subsequently service the message when service can be granted. Transient block states are also required to handle some race conditions, as previously discussed in Section 4.2.2.

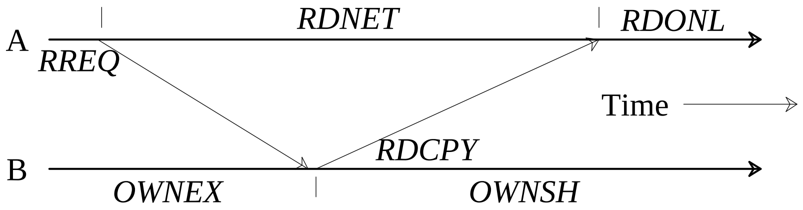


Figure 4.12: Protocol actions for a read access. Node *A* requests and receives a read copy to service local processor's read access. The block is maintained in the read network state *RDNET* while the request is outstanding.

In the rest of this sub-section, we discuss some examples showing the use of transient block states. Our first example illustrates the use of block states to identify outstanding requests. Consider the flow of a read access in the coherence protocol. Figure 4.12 illustrates the sequence of actions. Node *A* requires read access to a block. To service the read access, node *A* sends the read request (*RREQ*) to node *B*, which is the owner of the block. To indicate that a read request is outstanding for the block, the local state of block in *A* is marked to be in the transient read network state *RDNET*. This transient read network state *RDNET* is necessary in our design for at least three reasons. First, the state allows a memory frame to be reserved for the block at the time the request is issued. The alternative approach of allocating a memory frame for the block only when the reply arrives is more complex. For example, the scenario when no memory frame is free when the reply is received must be handled without the possibility of deadlock. Second, if another thread running on the same processor accesses the block again while the request is in progress, the transient state prevents a second request from being issued. Allowing multiple requests for the same block to be issued from a node adds complexity to the design for no benefit. Such a design must address additional correctness issues associated with the copy directory. For example, suppose the block received in reply to the first request is discarded when the reply to the second request is in transit. In this case, the copy directory for the block at the owner must continue to indicate the presence of a read copy at the requester even after the owner receives notification that a copy was discarded. Third, if a retry reply (*RETRY*) is received for the request, the transient state identifies that a read request must be re-issued (in most cases the necessary information can be obtained from the Pending Requests Cache (Section 4.1) without accessing the directory). In general, the coherence protocol and the system must be carefully studied to determine if the transient read network state can be eliminated from the design.

When node *B* receives the read request, node *A* is added to the copy directory, and a read copy (*RDCPY*) is sent to *A*. The block which was previously owned exclusively (*OWNEX*) in node *B* is moved to the *OWNSH* state to indicate the presence of remote copies. When node *A* receives the read copy, it verifies that this reply corresponds to the read network state *RDNET* of the block. Hence, the block is updated to the read-only state *RDONL*, and the processor is allowed to complete the read access. A similar transient block state *WRNET* is used to identify outstanding write requests. Read or write requests for the block from other nodes are denied service when the block is in the *WRNET* state.

To illustrate another situation where transient block states are necessary, consider the addition of a replacement event to the read access scenario. In our coherence protocol, a node that replaces a block must send a message updating the owner with this information. The node evicting the block will reclaim the space freed by the eviction without waiting for any confirmation from the owner. When the owner receives the replacement update, it does not have the option to deny service to the message.

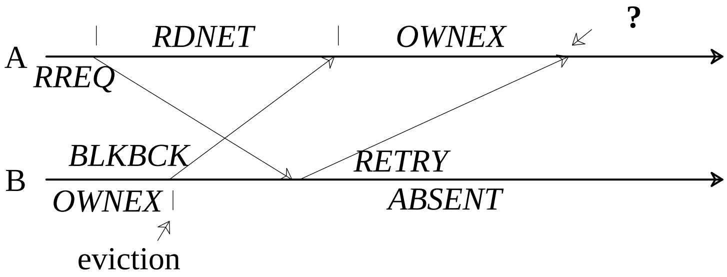


Figure 4.13: Replacement during read access. Node *B* which is the owner of the block replaces the block, and writes the block back to node *A*. Node *A* must process the writeback, and does so by changing the block to the exclusively owned state *OWNEX*. The subsequent retry message *RETRY* is unexpected in the *OWNEX* state.

The example including a replacement event during a read access is shown in Figure 4.13. Node *B* evicts the block due to local memory contention. Node *B* writes the block data back (*BLKBCK*) to node *A* while a read request to the block is in progress in node *A*. (The block will be written back to node *A* when node *A* is either the the default owner of the block or the owner of the page to which the block belongs.) After the write-back, node *B* does not possess the block. Hence, when it receives node *A*'s read request for that block, a retry reply message (*RETRY*) is sent to *A*. When node *A* receives the block data back (*BLKBCK*) it must service it. *A* recognizes that there are no copies of the block in the system from the *BLKBCK* message and the local state of the block is changed to the exclusively owned state *OWNEX*. The read access of the processor on node *A* can be completed at this time. However, when node *A* receives the retry message *RETRY* subsequently, the local state of the block indicates no outstanding request. The only options that are available are to flag an exception or error in the protocol, or to ignore the retry message. The first option is unsuitable since it would halt the system for a perfectly normal (possibly common) event. While the second option may be safe in this specific case, it has the potential to mask other exception conditions, implementation or runtime errors, thus compromising the robustness of the protocol.

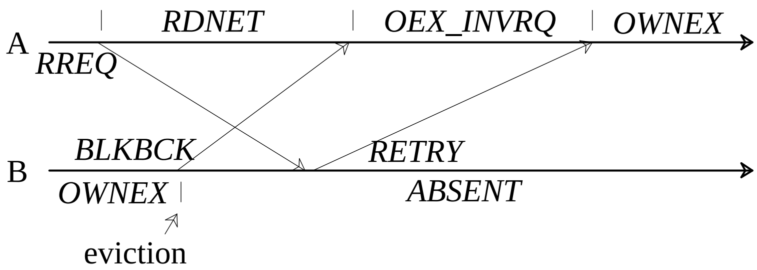


Figure 4.14: Addition of a transient protocol state to handle replacement. On receipt of the writeback, node *A* changes the block to a new *OEX_INVRQ* state which identifies the outstanding request. When the retry message *RETRY* is received, the block moves to the exclusively owned state *OWNEX*, and the message can be safely dropped.

The solution to this problem is to modify the coherence protocol design to preserve the information about the outstanding request even after the block back (*BLKBCK*) update message is received. Specifically, a new state can be added to the coherence protocol for this purpose. Upon receiving the writeback message in the read network state *RDNET*, the block is moved to a new state *OEX_INVRQ* which indicates that the block is owned exclusively with a request outstanding (*OEX_INVRQ* is a mnemonic for ‘‘owned exclusive with a pending invalid request’’). As before, the processor’s read access on node *A* can be completed at this point. When the retry message is received, the block can be transferred to the owned exclusive state *OWNEX*, and the retry message safely dropped. The modified coherence protocol flow is depicted in Figure 4.14.

Similar situations can occur during a write access, and similar solutions can be used. The handling of remote read and write requests in the *OEX_INVRQ* state (when a reply is pending) is an interesting issue. Although it is possible to service these remote requests while continuing to maintain the information about the outstanding request, it can lead to an increase in the number of protocol states and complicate design and verification. A viable alternative is to deny or postpone service to these remote requests until the reply for the outstanding request is received. This is the solution that we use in our protocol implementation.

We have previously discussed (Section 4.2.2) the use of transient block states to handle race conditions. As we discussed earlier, in the DASH protocol [Leno90], an invalidate request message can race a previously issued read copy reply since they use different networks. The invalidate cannot be denied service, but cannot be serviced until the read copy is received. A transient block state is maintained to track the outstanding invalidate. When the read copy is received subsequently, the transient block state indicates that the block must be invalidated and an acknowledgement must be sent.

4.2.4. Handling of Write Requests

A write miss occurs if the block to be modified is either not present locally or is present without write permission. To resolve the write miss, the block data and/or write permission must be acquired from the current block owner. This is accomplished by sending a write request to the block owner. When handling write requests, an important issue is whether or not the block data has to be sent to the requesting node over the network. If the block data is not present in the node and only part of the block is to be modified, then, generally, a block data transfer is necessary. However, it may not be necessary to transfer block data for a write request in other cases. For example, if the requesting node already possesses a read copy of the data, it is not necessary to re-send the data to the requester. It is sufficient to send a message granting write access (i.e., granting ownership). This approach can also be used if the requesting node will overwrite the entire data without reading the previous contents. While these cases can also be handled correctly by sending the data, the elimination of data messages (which are typically larger than control messages) can reduce the latency of the access and the load on the network.

It may not be possible to accommodate this optimization, however, for reasons discussed below. First, it is typically not possible to identify the case where a processor intends to write the entire data without reading it prior to the write. The size of the data being written may be smaller or larger than the size of data to be transferred (block or page) and cannot be determined by the coherence protocol. Also, the coherence protocol cannot typically distinguish between pure writes and read-modify-writes of the processor. Second, the node servicing the write request may not be able to determine if the requester has a read copy of the data. For example, this will be the case if the coherence protocol allows data to be shared in the broadcast state. In this case, the owner of the data may not maintain a copy directory and cannot determine if the requester already has a read copy.

It is reasonable to consider requiring the requester to indicate whether it possesses a read copy, to solve this problem. But, this is not sufficient when the copy directory is not maintained by the node servicing the write request. In this case, a concurrent write by the owner of the block can invalidate the read copy of the requester, after the requester indicated that it possessed a read copy. Note that, from the discussion in Section 4.2.1 on deadlocks, invalidate messages must be immediately serviced. The requesters write would be granted after the write of the block owner, and if the data is not sent to the requester, the last copy of the data would have been discarded from the system.

With both the one-level and the hierarchical schemes, the copy directory may not be maintained by the node servicing write requests. In the one-level scheme, the copy directory is maintained by the block owner, but write requests are serviced by the default owner. In the hierarchically managed system, the write access to the block will be granted by the node owning the page (to which the block belongs), but the copy directory for the block is maintained by the owner of the block (who may be different from the owner of the page). In the hierarchical scheme, additional coordination between the block owner and the page owner would be needed for the page owner to identify if the requester possesses a read copy. Such coordination could be achieved, for instance, by requiring the block owner to send the copy directory of the block to the page owner, when the block owner relinquishes ownership of the block. The page owner can determine whether the node requesting the write currently has a copy of the block by looking at this copy directory.

In our DSM implementation, we maintain the flexibility to allow blocks to be shared in the broadcast state to investigate support for heavily shared data. For this reason and the additional complexity associated with determining if data has to be transferred to the requester, we transfer the data on all write requests in our coherence protocol implementation. Protocol implementations for DASH [Leno90] and FLASH [Kusk94] also transfer data on all write requests.

4.3. A Coherence Protocol for Hierarchically Managed DSM

We present our distributed directory coherence protocol in detail in this section. While our coherence protocol is an invalidation based ownership protocol similar to the protocols implemented in DASH [Leno90], FLASH [Kusk94], and Alewife [Agar95], it differs in its support for multiple levels of ownership. We sketched our approach to resolving issues in designing distributed directory coherence protocols in Section 4.2. Deadlocks are avoided by ensuring that invalidate and reply messages are always sunk. Special block states are used to indicate pending operations. Data is always transferred for write requests. Many potential race conditions are avoided by using bilateral exchanges rather than forwarding. Specifically, we employ a hybrid approach, which uses bilateral exchanges only for write requests (which are fewer than read requests) and forwarding for read requests. We describe our protocol in detail in the following sub-sections.

4.3.1. Protocol Ownerships

The hierarchical scheme supports ownership of data at both the page and block levels. Each page that is in the physical memory of some node in the system is owned at both the page and block levels. Initially, a page is owned by the node (*page owner*) that reads the page from disk. Subsequently, the page ownership can be transferred by migrating the page to another node or by swapping the page to disk. Each block of the page can also be independently owned at the block level. When a page is read from disk, the page owner is the *block owner* of all blocks of the page. When a node other than the current block owner writes to the block, the block ownership is transferred to the writing node. The page owner coordinates these block ownership transfers, and maintains the identity of the block owner of each block of the page.

Each page in the virtual address space is also statically assigned a *default owner*. The default owner maintains the identity of the current page owner in the Migrated Page Table (MPT). If a page is not indicated in the MPT, the page resides only on disk. The default owner and its MPT are updated whenever the page is migrated or written back to disk. It is important to note that while both the page owner and the block owner maintain storage for the page in their physical memory, the default owner does not maintain storage for the page in its physical memory (unless it is also accessing the page).

Directory information about a shared page is distributed among the nodes in the system as follows. The default owner of the page identifies the current page owner. The page owner maintains the identity of the block owner of each block of the page, as well as the location of read only copies of any blocks that it owns itself. The block owners maintain the location of read only copies of the block. Any node in the system that has at least one block of the page in its physical memory maintains the identity of the page owner, but it does not maintain the identity of the block owners. All nodes in the system know the identity of the default page owner of every virtual page.

The responsibility for handling access requests to a shared page is distributed as follows. The block owner services read requests for the block by providing read only copies of the block. The block owner is also responsible for invalidating these read only copies during a write. The page owner services write requests by coordinating the transfer of block ownership from the previous block owner to the new writer. The default owner services read and write requests by identifying the page owner if it is resident in some physical memory in the system, and by issuing the request to disk, otherwise. Any node can send read and write requests to the page owner, if it knows the page owner, and to the default owner, otherwise.

4.3.2. Page and Block States

A page can be in one of the following protocol states — invalid (*INVALID*), valid private (*PRIVATE*), valid shared (*SHARED*), read requested (*RDNET*) or write requested (*WRNET*). As previously mentioned in Section 4.1.1, this state of the page is maintained by the PPT entry in 2 PPT state bits and 5 PPT meta-state bits. The two PPT state bits identify whether the page is invalid, valid, read requested or write requested. When the state bits indicate that the page is valid, one of the meta-state bits identifies if the page is private or shared. The other meta-state bits maintain additional information about the page when the page is valid. One meta-state bit indicates if the page is clean or dirty. The other three meta-state bits maintain information relevant in page replacement. They indicate if the page is pinned from being replaced, if the page is being evicted to free the PPT entry of the page and if the page is being evicted to free the SPT entry of the page respectively.

access_permit	Implication for local processor
ABS	Absent; no access permitted
RDO	Only reads permitted
OEX	Owned exclusive; reads and writes permitted
OSH	Owned shared; read permitted; invalidate for write

Table 4.2: *access_permits* of a block.

When a page is shared, the protocol also maintains a state for each block of the page. The block state has two components: *access_permit* and *access_waitlist*. The *access_permit* component specifies what accesses to this block are permitted for the local processor. The *access_waitlist* component identifies what access is pending on the block. There are four possible values for access_permit (Table 4.2). The *ABS* state indicates the block is absent and the local processor has neither read nor write access permission. The local processor is allowed read access but no write access if the access_permit is either *RDO* (read only) or *OSH* (owned shared). The *OSH* state differs from the *RDO* state in two respects. First, the *OSH* state indicates the local processor owns the block. Second, it indicates the processor can secure write access by invalidating all read copies. The local processor is permitted both read and write access when the access_permit is *OEX* (owned exclusive).

If the access_permit state does not allow the local processor's access, access

access_waitlist	Access in progress
NONE	None
RDNET	Local read request
WRNET	Local write request
LOCK	Remote write request

Table 4.3: *access_waitlists* of a block.

permission needs to be obtained from the block owner or the page owner. Since this is not an atomic operation, there is a need to keep track of pending operations on the blocks using the access_waitlist state (Table 4.3). The access_waitlist can indicate that no request is outstanding (*NONE*), a local processor's read request is outstanding (*RDNET*), a local processor's write request is outstanding (*WRNET*), or that a remote processor's write request is outstanding (*LOCK*). At most one outstanding request is allowed per block. An outstanding request for a block blocks all other requests for the block. The semantics of access_waitlists *NONE*, *RDNET*, and *WRNET* are obvious. The *LOCK* access_waitlist indicates that, in response to a remote request, exclusive block ownership is being acquired, and the block ownership should be transferred as soon as it is acquired. As we will see in the discussion of protocol messages (Section 4.3.3), exclusive block ownership is acquired by invalidating read copies, if the local node is the block owner, or by requesting a write back from the block owner, if the local node is the page owner (but not the block owner).

The two components of the block state, access_permit and access_waitlist, can each be represented in 2 bits (for a 4-bit block state). The block state is shown as a composition of the access_permit and the access_waitlist in Table 4.4. The semantics of most of the block states and the state transitions follow easily from the access_waitlist (2 bits)

		NONE	RDNET	WRNET	LOCK
access _permit (2 bits)	ABS	ABSENT	ABS_RDNET	ABS_WRNET	ABS_LOCK
	RDO	RDONL	X	RDONL_NET	RDO_LOCK
	OEX	OWNEX	OEX_INVRQ	OEX_INVRQ	X
	OSH	OWNSH	X	OWNSH_NET	OWNSH_XFR

Table 4.4: block states — composition of access_permit and access_waitlist

above discussion of access_permit and access_waitlist. We elaborate on the less intuitive block states. *OWNSH_NET* and *OWNSH_XFR* both signify that invalidation is in progress, for a local write access in the former case, and for a remote write request in the latter case. When invalidation is complete and exclusive ownership is acquired, the local processor can complete its write (*OWNSH_NET*), or the ownership can be transferred to the remote requester (*OWNSH_XFR*). The block states *ABS_LOCK* and *RDO_LOCK* are valid only in the page owner, when it is coordinating the transfer of block ownership from the current block owner to a new requester. The block state *OEX_INVRQ* is also valid only in the page owner. This state indicates that the block is owned exclusive and there is a pending invalid request involving this block which has been sent from this node. As discussed in Section 4.2.3, this state occurs when the block is written back to the page owner by the previous block owner (due to a replacement in the block owner) when a local request for the blocks was outstanding at the page owner. This protocol state is necessary to handle the retry response that will be received for the pending invalid request. No remote requests are serviced in this state until a retry message is received for the outstanding request (see

Section 4.2.3). When the retry message is received, the block state is changed to *OWNEX*.

4.3.3. Protocol Messages

Message	Function	Class
RREQ	Read request	Request
WREQ	Write request	Request
WRBCK	Request write-back of block	Request
INV	Invalidate block read copy	Command
PGINV	Invalidate page	Command
PAGE	Page data, that is currently not shared	Reply
RDCPY	Block data for read access	Reply
WDATA	Block data for exclusive ownership	Reply
ACK	Acknowledge invalidation	Reply
PGOWN	Identification of page owner	Reply
RETRY	Require retry	Reply
BLKBCK	Write back of block data	Reply/Update
CPINV	Copy invalidated notice	Update
PGBCK	Write back of page data	Update

Table 4.5: Protocol messages and their function.

Table 4.5 lists the messages used in our coherence protocol. We classify the messages into four categories — requests, commands, replies, and updates. Request messages solicit a reply message indicating either that the service was provided or that it was not provided. Command messages also solicit a reply, but cannot be denied service. As we discussed in Section 4.2.1, this ensures that protocol deadlocks are broken. Reply messages are sent in response to a prior request. If the reply does not satisfy the requested service, the request will be re-issued. Update messages are caused by replacements, do not expect a reply, and cannot be denied service. We describe the functionality of some non-trivial messages below.

The request message *WRBCK* is used by the owner of the page to request the current block owner to write back the block to the page owner, when the block ownership needs to be transferred. If a local write access is not in progress, the block owner writes back the block with the *BLKBCK* message. If a request cannot be immediately satisfied, a *RETRY* response can be sent in response to any of the requests (negative acknowledgement). The requester can subsequently re-issue the request. The *PGOWN* reply is used by the default page owner to implement the bilateral exchange on a read or write request. The default owner does not forward the request to the page or block owner. Rather, the requester is informed of the identity of the page owner using the *PGOWN* message.

When a node replaces a page (and the blocks belonging to that page) from its physical memory, the default page owner, the page owner, and/or the block owners must be informed. The nodes which must be informed, and the message types that must be used to inform these nodes depend on the state of the evicted page and the state of the blocks of the page. When a node replaces a page that it owns, the contents of the page (all blocks) are written back to the default owner with a *PGBCK* update message. When the page is not owned by this node, any blocks that are owned locally are each written back to the page owner using *BLKBCK* messages. If the read copy of any blocks of the page was discarded, the owner of each block is notified using a *CPINV* message.

The *PGINV* message is a performance optimization for the hierarchical scheme. When a node replaces a page that it owns, the node ‘‘pulls’’ all blocks of the page to itself (see Section 4.1.1) so that the page data may be written back. This can be accomplished by individually requesting the writeback of each block of the page that is owned by a remote node and by individually invalidating each read only copy of blocks of the page which are owned locally. The *PGINV* message accomplishes this task with lower message count and lower node processing by implicitly commanding the writeback of any blocks of the page that are owned by the recipient, and the invalidation of any read copies held by the recipient. A single *ACK* response along with the appropriate *BLKBCK*s signals the completion of the page invalidation in the recipient.

Messages *RDCPY*, *WDATA*, and *BLKBCK* carry one block of data. Messages *PAGE* and *PGBCK* carry one page of data. All other messages are control messages. In our current implementation, the *PAGE* message is used only when the page is read from disk. *PAGE*, *PGBCK* and other newer messages can be used to support page migration, but this has not been addressed in the current implementation.

4.3.4. The Flow of Protocol Actions During Reads, Writes and Replacements

The discussion of our coherence protocol for the hierarchically managed DSM system thus far has described the assignment of ownerships in the protocol, the various states of pages and blocks in the protocol and the various messages that are used in the protocol. The ownership and state information associated with the blocks and pages in the system are distributed amongst the nodes in the system. At each node, they are maintained in the tables used by the hierarchical scheme: the Present Pages Table (PPT), the Shared Pages Table (SPT), the Migrated Pages

Table (MPT) and the copy directory (described in Chapter3). When a processor access misses in a node or when a page is replaced from a node, different protocol messages are exchanged between several nodes and necessary tables are accessed at these nodes to look-up and update protocol ownership and protocol state information. The messages that are exchanged, the tables that are looked up and the transitions in protocol state and ownership all naturally depend on the transaction (read, write or replacement) and on the state and ownership distribution at that time. In the following three sub-sections, we describe in detail the flow of protocol actions used to service read requests (Section 4.3.4.1), to service write requests (Section 4.3.4.2), and to service replacement events (Section 4.3.4.3).

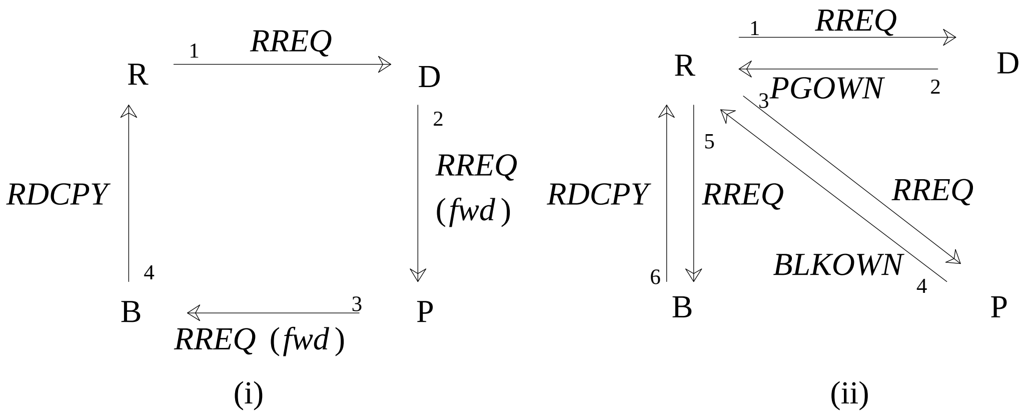


Figure 4.15: Comparison of bilateral exchange and forwarding for the hierarchical scheme. When a node making a request does not know the page owner, the default owner, the page owner and the block owner will all participate in servicing the request. The default owner identifies the owner of the page to the requester (using *PGOWN*), and the page owner identifies the owner of the block to the requester (using *BLKOWN*). With bilateral exchanges (case ii), six messages are sent and processed, as compared to four when forwarding (case i) is used.

Before we describe these flows of protocol actions, we first describe a modification to our protocol to address the inefficiency of using bilateral exchanges. As we mentioned in our discussion in Section 4.2.2.2, protocols based on bilateral exchanges incur higher message count than protocols based on forwarding. To highlight this inefficiency, consider a specific example on the hierarchically managed system. In this example, a node that does not know the owner of a page requests access to a block. Also, the default page owner, the owner of the page and the owner of the required block are three different nodes. In this case the request must first be sent to the default page owner, then to the page owner, and finally to the block owner. The comparison of messages exchanged in the protocol based on forwarding and in the protocol based on bilateral exchanges is shown in Figure 4.15. With the use of bilateral exchanges, such a request would require six messages to be exchanged and processed. Using forwarding, the same request can be satisfied with four messages.

The higher number of messages used with bilateral exchanges will increase the latency of accesses and the load on the network. However, if forwarding is used, several race conditions must be handled. Hence, we use a hybrid protocol that minimizes the inefficiency due to bilateral exchanges. The hybrid protocol uses forwarding for efficiency where race conditions are easy to resolve and uses bilateral exchanges elsewhere to avoid the race condition problems discussed in Section 4.2.2.

In our read access protocol we use bilateral exchange at the default owner but use forwarding at the page owner. Since the page owner must be contacted to request a read copy of <u>any</u> block of the page, this approach is likely to reduce message count significantly. Since the default owner is contacted much less frequently (only when a node has no blocks of the page), the reduction in message count by also forwarding at the default owner is likely to be marginal. The small improvement may not justify the additional complexity required to handle race conditions caused by forwarding at the default owner. With this approach, when the default owner receives a read request, it replies to the requester with the *PGOWN* message identifying the current page owner. The requester subsequently directly requests the page owner for a read copy of the block. When the page owner receives a read request, it is forwarded to the block owner, who directly responds to the original requester. This hybrid approach reduces the number of messages for the example shown in Figure 4.15 to five (from six messages with the use of pure bilateral exchanges).

Write accesses, which are less frequent than read accesses, are serviced using the bilateral exchange approach. The penalty for using bilateral exchanges instead of forwarding is smaller for write accesses since, even with forwarding, a message must be sent from the block owner back to the page owner to record the change in ownership.

The read and write protocols are described in detail in the following sub-sections. The use of forwarding in read accesses makes races possible. The races created by the use of forwarding are discussed in conjunction with the description of the read protocol.

4.3.4.1. Anatomy of a Read Request

A read request is initiated when a local processor's read access misses in the node's (*R*) cache and memory. We describe the flow of the protocol actions used to service the read request under different situations. The cases that we discuss include: 1) *R* has other blocks of the page, 2) *R* has no other blocks of the page, and the page is only on disk, 3) *R* has no other blocks of the page and the page is owned by the default page owner itself and 4) *R* has no other blocks of the page and the page is owned by a node other than the default page owner.

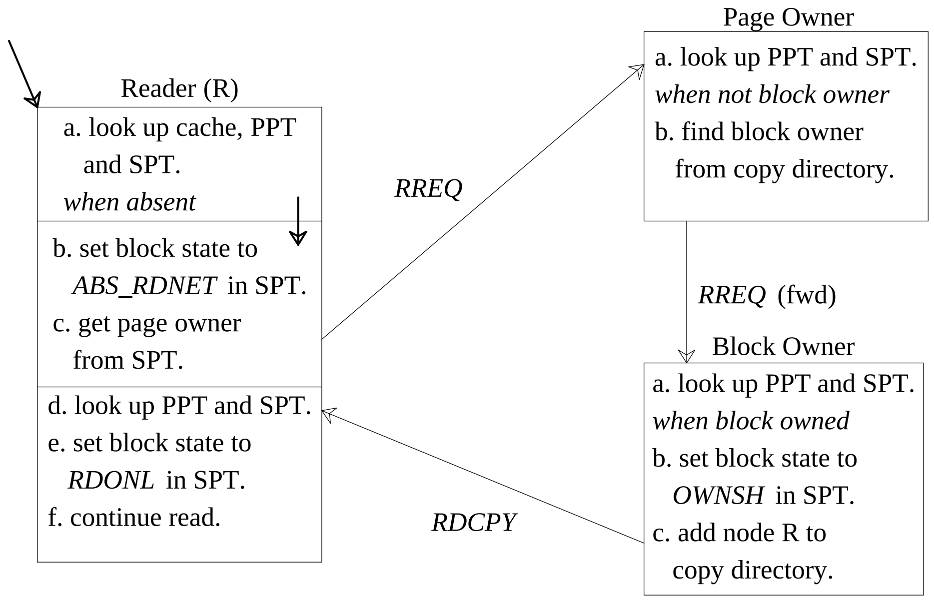


Figure 4.16: Anatomy of a read request. This is one possible flow of the protocol to satisfy a read miss by the local processor on node *R*. *R* does not have the block required by the read access, but has other blocks of the page. Hence, *R* knows the identity of the page owner. The block is owned by some node other than the page owner.

First consider the case where other blocks of the same page are present in node *R*. Since *R* has other blocks of the page, the Shared Pages Table (SPT) entry for the page on node *R* identifies the owner of the page. *R* sends a read request *RREQ* to the page owner (*P*) identified in the SPT entry. *R* changes its local state of the block to *ABS_RDNET* to mark the pending read request. On receipt of the *RREQ*, the page owner checks its SPT to determine if it owns the requested block. A block is owned locally if the local state of the block has access_permit *OEX* or *OSH*. If the block is owned by another node, the page owner looks up the identity of the block owner in its copy directory and <u>forwards</u> the request to the block owner (*B*). When the block owner receives the request, it checks its SPT to verify that it owns the block, updates the local state of the block to *OWNSH*, adds *R* to its copy directory, and sends the block data in a *RDCPY* message to *R*. When *R* receives the *RDCPY*, it updates the local block state to *RDONL* and allows the local processor to complete the read access. This flow of the read request is shown in Figure 4.16.

If the page owner *P* owns the block itself when it receives the read request from *R*, it can service the request itself. In this case, *P* does not forward the read request to any other node. *P* adds *R* to its copy directory and sends a read copy of the block (*RDCPY*) to *R*.

If *R* does not have any block of the page in its node memory, it cannot identify the owner of the page. In this case, an initial phase is needed to query the default owner for the page owner identity. *R* allocates a PPT entry for the page and marks its state to indicate a read request is in progress. *R* sends the *RREQ* to the default owner *D* of the page (the default page owner is a static function of the page number). When the default owner receives the *RREQ*, it looks up the page in its PPT. If the page is not present in the default owner (no PPT entry) it looks up its

Migrated Page Table (MPT) to see if the page is owned by any node. If the page is present in the default owner (PPT entry present) the SPT entry for the page is accessed to identify the page owner. Three outcomes are possible from the access of the MPT or the SPT — the MPT may indicate that the page is not owned by any node, the SPT may indicate that the page is owned by the default owner itself, or either the MPT or the SPT may indicate that the page is owned by some node other than the default owner. When the page is not owned by any node, the page is available only on disk. In this case, the entire page is read from disk by the default owner, and sent in a *PAGE* message to the requester *R*. The default owner also updates its MPT to indicate that *R* owns the page. On receipt of the *PAGE* data message, *R* changes its PPT entry state to private. When the default owner is also the page owner, the protocol continues as previously described (Figure 4.16).

Figure 4.17 describes the case when *R* does not have other blocks of the page, and the page is owned by some node other than the default owner. The default owner determines the identity of the page owner from its MPT or its SPT and sends the identity of the page owner node *P* in a *PGOWN* message to the requester *R* (bilateral exchange). When *R* receives the *PGOWN* message, it checks its PPT entry which indicates that a read request is in progress. *R* then sends a *RREQ* request message to the node *P* identified by the *PGOWN* message. Starting from this point, the transaction progresses as shown in Figure 4.16 up to the point where the block owner sends the block in a *RDCPY* message to *R*. When the *RDCPY* reply arrives at *R*, an SPT entry is allocated (since the page is shared) for the page in *R*. *R* sets the state of this block to *RDONL* and sets the state of all other blocks to *ABSENT* in the SPT entry. *R* also records the identity of the page owner in the

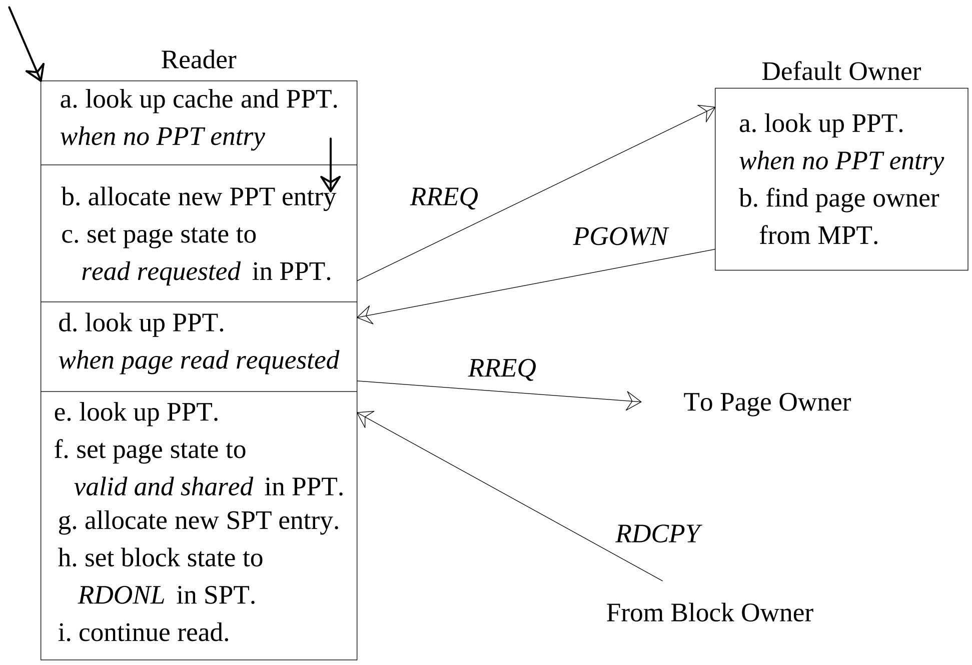
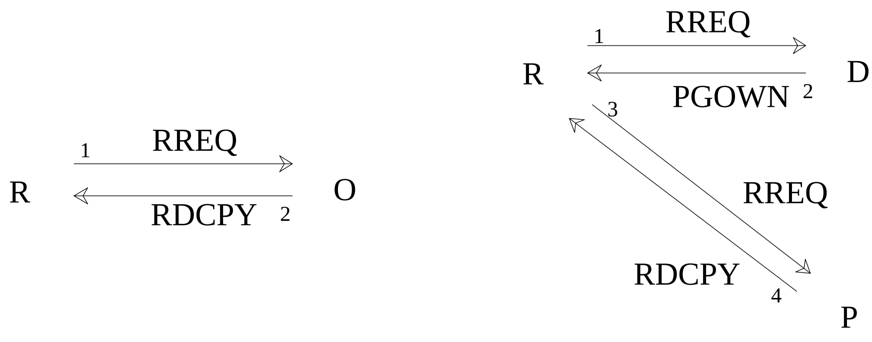


Figure 4.17: Read access when page owner is not known locally. This shows the flow of the protocol to satisfy a read miss by a local processor on node *R*, when *R* has no block of the page. *R* does not know the page owner, and requests the default owner who send the page owner identity in a *PGOWN* message. Subsequently, the protocol can flow as shown in Figure 4.16. When the *RDCPY* is received, an SPT entry is allocated in *R*. The page owner is marked in the SPT entry, the state of the block set to *RDONL*, and the states of all other blocks of the page set to *ABSENT*.

SPT entry.

When the page owner itself owns the block, clearly the request need not be forwarded anywhere. The page owner services the request and sends a *RDCPY* with the requested block to *R*. We illustrate this in Figure 4.18. It is possible for the page or block ownership to move while the read request is in transit. In this case, a read request can arrive at a node that was previously the owner after the



(i) (ii)

Figure 4.18: Additional protocol flow sequences for read requests. *R* is the read requester, *P* is the page owner, *D* is the default owner, and *O* represents either a page owner or a block owner. Protocol can flow as in case (i) in two situations — when *O* is the page owner and the block owner; or when *R* is the page owner and *O* is the block owner. Case (ii) is possible when *R* has no block of the page, and the page owner is also the block owner. Other flows are also possible, including those when the ownerships move.

node relinquished ownership. In such situations, the node receiving the request forwards the request to the default owner and the protocol flow is repeated. A pathological case where a read request is endlessly re-issued trying to locate the current owner of the block is possible. This can occur if the ownership of the block or page moves repeatedly in such a manner that the read request always reaches a node after it relinquishes ownership. However, this pathological case is unlikely (a similar scenario is also possible in DASH [Leno90] and FLASH [Kusk94]).

In our discussion thus far, we have discussed read requests which are serviced by the page owner and block owner as soon as the request is received. The page owner or block owner may not be able to service the read request immediately if a write access is in progress for the block, or if the page is being replaced. A write access in progress is indicated by the block states *OWNSH_NET*, *OWNSH_XFR* and *ABS_LOCK*. A page that is being replaced is indicated by the PPT meta-state bits associated with the page replacement mechanism (Section 4.3.2). The read request cannot be satisfied in both these cases since it conflicts with the write access and the page replacement. The write access will complete only after exclusive ownership of the block is acquired. The page replacement will complete only after exclusive ownership of all blocks of the page (including the block being requested) is acquired. If a read copy is sent, these transactions will not complete since all read copies would not have been invalidated. Under these situations, the owner sends a *RETRY* message to *R* requiring it to retry the request.

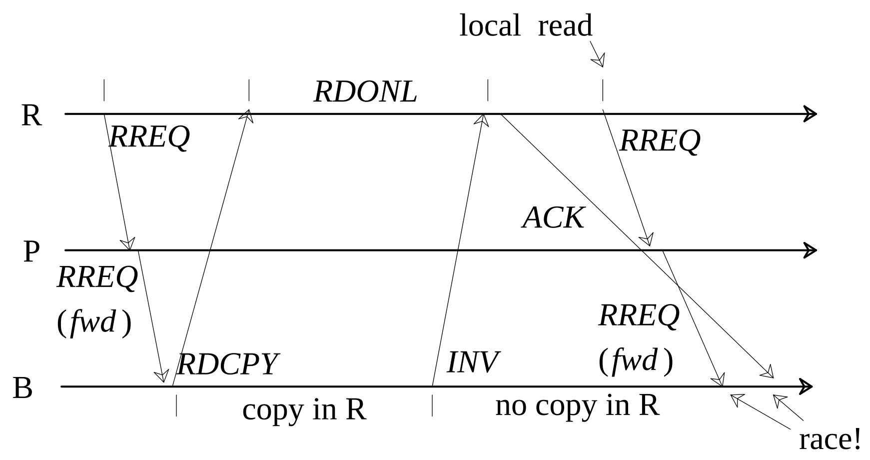


Figure 4.19: Race between *ACK* and *RREQ*. With the use of forwarding in servicing *RREQ*, a *RREQ* can race an *ACK* sent previously since they follow different paths.

The forwarding of the read request by the page owner introduces race conditions in the protocol. Consider the case where node *R* has its read copy of a block invalidated by the block owner *B* and subsequently requests a read copy of the block. Figure 4.19 shows the paths taken by the acknowledgement message

ACK and the read request *RREQ*. The read request *RREQ* is forwarded by the page owner *P* while the acknowledgement message *ACK* is sent directly to the block owner *B*. While the *ACK* message is sent from *R* before the *RREQ*, the *RREQ* can arrive at *B* before the *ACK*. (With bilateral exchanges, *P* would identify *B* to *R*, and *R* would directly send the *RREQ* to *B*, which cannot race the previous *ACK*, since they follow the same path).

In our implementation, the block owner handles the race situation between the *ACK* reply and the subsequent *RREQ* request by refusing to service any *RREQ* requests (by sending a *RETRY* reply) received when it is waiting for acknowledgements. As we described in Section 4.1, the information that the block owner is waiting for acknowledgements is maintained in its Pending Requests Cache. Since the Pending Requests Cache does not maintain the identity of nodes that must send acknowledgements, this mechanism cannot identify whether a *RREQ* request that is received is sent from a node that must send an acknowledgement (which indicates a race). When the *RREQ* is indeed sent by a node that must send the acknowledgement, this mechanism handles the race condition. This solution to the race problem may appear to be an unnecessary overhead whenever the *RREQ* request is received from a node other than one of the nodes that must send acknowledgement. However, even in such situations, it is not an overhead when the block owner had issued the invalidations to satisfy a previous write request. Read requests received while a write is in progress would anyway be denied service to allow the write access to complete. The mechanism incurs unnecessary overhead only when the block owner had invalidated read copies to evict them from the copy directory and a request is received from a node other than the nodes whose copies were evicted. This is likely to be an infrequent event.

A race condition is also possible between an *ACK* message and a *CPINV* message. Suppose a node *R* evicts its read only copy of a block when the owner of the block, *B*, is concurrently invalidating all read copies of the block to satisfy a write access. A *CPINV* message notifying the block owner of the replacement will be sent by *R* to the page owner since *R* does not know the identity of the block owner. As with read requests, when the page owner receives the *CPINV* message, it forwards it to the block owner *B*. When *R* receives the *INV* message from *B* after it had sent the *CPINV* message, *R* sends an *ACK* message acknowledging the invalidate directly to *B* (in our implementation, an *ACK* message is sent to an *INV* message even when the block is not already present in the node — to support invalidations that are broadcasted). Since the *CPINV* and the *ACK* follow different paths to *B*, the *ACK* message can race the *CPINV* message and arrive at *B* first. However, this race is not a problem for our protocol. The *CPINV* messages requires that *R* be dropped from the copy directory for the block in *B*, which was already accomplished by the the invalidation. Hence, the *CPINV* is ignored. In general, when a block owner receives a *CPINV* message from a node that is not indicated in its copy directory, the *CPINV* message is ignored.

4.3.4.2. Anatomy of a Write Request

A processor's write access is a hit in the local node if the access_permit component of the local state of the block (Section 4.3.2) is *OEX*. The processor's write access will miss in the local node if the local state of the block being accessed does not permit write access, or if the block being accessed is not available locally. Write access must be requested from remote nodes in these cases to service the write miss.

The complete flow of protocol actions needed to satisfy a write request naturally depends (as with read requests) on several factors such as whether the requesting node *W* knows the identity of the page owner and whether the page owner is also the owner of the required block. In the rest of this sub-section, we discuss the flow of protocol actions in several cases including: 1) when *W* is also the owner of the block, 2) when *W* is also the page owner but not the owner of the block, 3) when *W* has other blocks of the page, 4) when *W* has no other blocks of the page and the page is on disk and 5) when *W* has no other blocks of the page and the page is owned by some node.

The write-miss scenario is simplest when the node that issues the write access is also the owner of the block. A node that owns a block will miss on a write access to the block if the block is owned shared (access_permit component of the local state of the block is *OSH*). In this case, the node must invalidate all read copies of the block in the system to service the write. The block owner first changes the local state of the block in its Shared Page Table (SPT) to *OWNSH_NET* indicating that a local write is in progress. The block owner then looks up the copy directory and an invalidate message *INV* is sent to each node that has a read copy of the block. The write request is marked in the Pending Requests Cache on *W* (see Section 4.1). The Pending Requests Cache entry identifies the number of acknowledgements that must be received before the write access can be granted. When a node holding a read-only copy of a block receives an invalidate (*INV*) message, the local read copy is invalidated by setting the access_permit of the block state to *ABS*. An *ACK* reply is sent by the node receiving the invalidate (*INV*) message to the block owner, which sent the invalidate. Once all *ACK* messages (as indicated by the entry in the Pending Requests Cache) have been received by the block owner *W*, *W* sets the local state of the block to *OWNEX* and allows the write access to proceed.

When the node issuing the write *W* is not the owner of the block, the write request must be serviced by the page owner. When *W* is itself the owner of the page (but not the owner of the block), *W* can service the write request itself. First, in the SPT entry of the page in *W*, the block is marked to be in state *ABS_WRNET* if it was previously absent, and to be in state *RDONL_NET* if a read copy was present previously. The page owner *W* then uses its copy directory to determine the identity of the block owner. *W* then sends a write-back request *WRBCK* to the block owner (*B*). When the block owner *B* receives the write-back request, if the block is owned exclusively (*OWNEX*), *B* can immediately write the block back to the page owner *W* using a *BLKBCK* message. *B* then changes its local state of the block to *ABSENT*. When *W* receives the *BLKBCK* message, the access_waitlist component of the local block state indicates that a local write is in progress. Hence, the local block state is set to *OWNEX* and the local write access is allowed to continue.

In the scenario discussed above, where the node *W* issuing the write access is the page owner but not the block owner, the state of the block in the block owner *B* is *OWNEX*. The state of the block in the block owner *B* can also be owned shared (*OWNSH*). In this case, unlike the previous scenario, the *BLKBCK* message cannot be sent by the block owner until all read copies of the block have been invalidated. When the block owner *B* receives the *WRBCK* message, it looks up the copy directory and sends an invalidate message *INV* to each node that has a read copy of the block. The local state of the block in *B* is modified to *OWNSH_XFR* to indicate that a remote write is in progress and that the ownership of the block will be transferred away once all acknowledgements are received. Once all *ACK* messages have been received (identified using the Pending Requests Cache as previously discussed) by *B*, *B* writes the data of the block back to the page owner *W* using the *BLKBCK* message. At this point, *W* sets the state of the block to *OWNEX* (as before) and allows the write access to continue.

When node *W* is neither the block owner nor the page owner, the write request must be sent to the page owner for service. If some other blocks of the page being written to by *W* are present locally on *W*, the identity of the page owner is available in *W*'s Shared Pages Table (SPT). In this case, a write request is sent to this page owner *P*. In the SPT entry of the page in *W*, the block is marked to be in state *ABS_WRNET* if it was previously absent, and to be in state *RDONL_NET* if a read copy was present previously. When the page owner receives the write request, it looks up the local state of the block in its SPT. If the page owner possesses the block in the exclusive owned state (*OWNEX*), the block and ownership are immediately sent by a *WDATA* message to *W*. The page owner then sets its copy directory to indicate that *W* is the block owner and changes the local block state to *ABSENT*.

If the page owner *P* possesses the block in the owned shared state (*OWNSH*) when the write request is received from *W*, read copies of the block must be invalidated before ownership is transferred. At the page owner, the access_waitlist of the block state is set to *LOCK*, indicating that a remote write is in progress. The page owner looks up its copy directory and invalidate messages *INV* are sent to all nodes with read copies. An entry is also added to the Pending Requests Cache in the page owner to indicate the number of acknowledgements expected for the invalidation. The information about the original remote write request including the identity of the requester (*W*) is recorded in this entry. Once all acknowledgements are received, the page owner sends the block and its ownership to the requester *W* with a *WDATA* message (the page owner determines that node *W* issued the write request from the information previously recorded in the Pending Requests Cache). The page owner then sets its copy directory to indicate that *W* is the block owner and changes the local block state to *ABSENT*. As before, when *W* receives the *WDATA* message, the local state of the block in *W* is set to *OWNEX* and the write access is allowed to complete.

Now, consider the scenario when the node issuing the write *W* sends the write request to the page owner *P* and the block accessed by the write is not owned by the page owner. In this case, when the page owner receives the write request, it uses its copy directory to determine the identity of the block owner. The page owner then sends a write-back request *WRBCK* to the block owner (*B*). At the page owner, the access_waitlist of the block state is set to *LOCK*, indicating that a

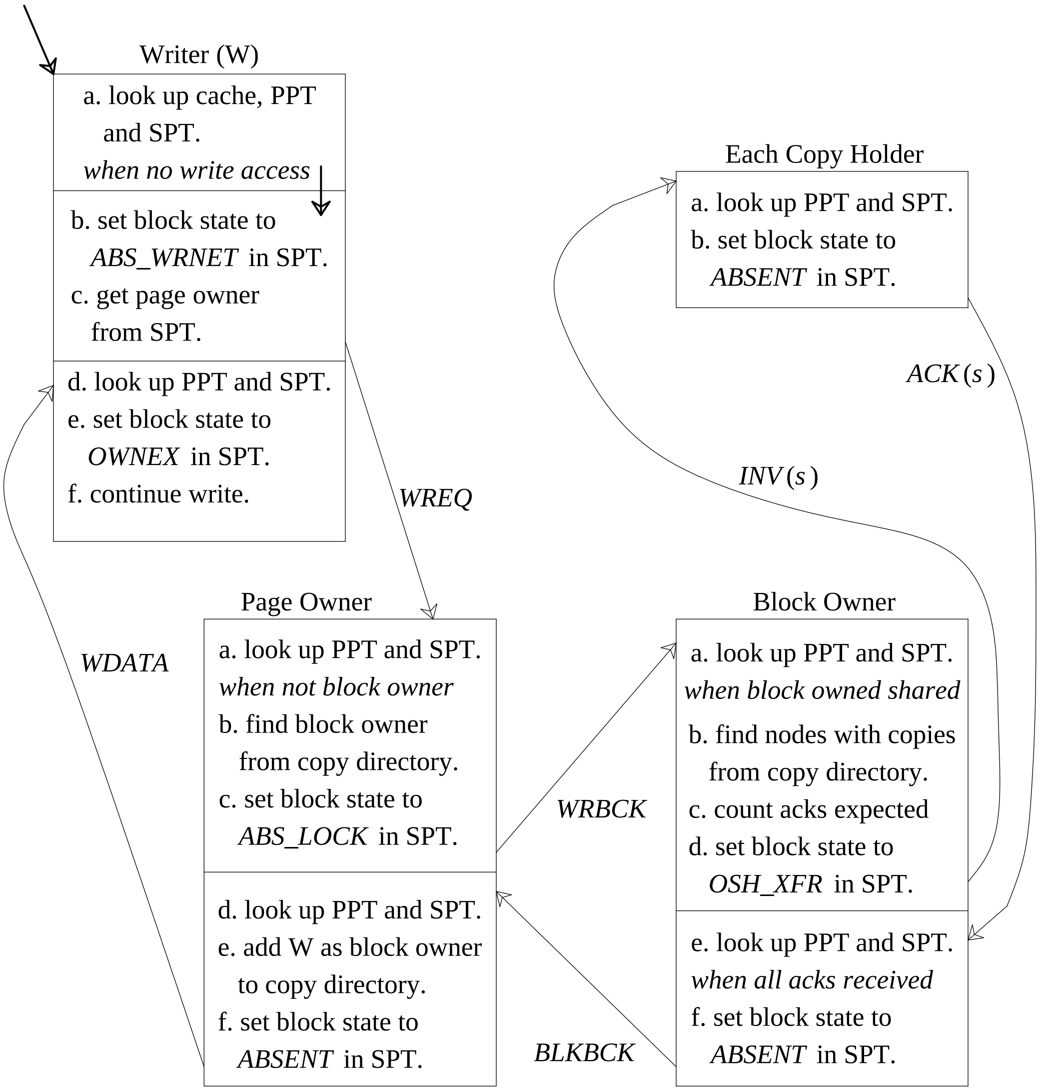


Figure 4.20: Anatomy of a write request. This is one possible flow of the protocol to satisfy a write miss by a local processor on node *W*. *W* does not have the block required by the write access, but has other blocks of the page. Hence, the SPT entry in *W* identifies the page owner. The block is owned by a node other than the page owner. Read copies of the block are present on other nodes.

remote write is in progress. An entry is added to the Pending Requests Cache in the page owner indicating that a response is expected for the write-back request. As before, the information about the original write request including the identity of the requester (*W*) is recorded in this entry. When the block owner receives the *WRBCK* request, the block must be written back to the page owner. As discussed in the previous cases, if the block is owned exclusively (*OWNEX*), the block owner can write the block back to the page owner immediately. If the block is owned shared (*OWNSH*), the block owner writes the block back to the page owner after all read copies have been invalidated. When the page owner receives the *BLKBCK* message from the block owner, since the access_waitlist component of the local block state is *LOCK*, the block and its ownership are sent to the requester *W* with a *WDATA* message. The page owner determines that node *W* issued the write request from the information previously recorded in the Pending Requests Cache. The page owner then sets its copy directory to indicate that *W* is the block owner and changes the access_waitlist component of the local block state from *LOCK* to *NONE*. When *W* receives the *WDATA* response, it changes the block state in its SPT to exclusively owned (*OWNEX*), allowing the local processor to complete its write access. This flow of a write access is shown in Figure 4.20. It should be noted that all the inter-node exchanges in this case are bilateral.

When the requester *W* does not have any block of the page, *W* sends the write request to the default owner *D*. *W* allocates a PPT entry for the page. *W* sets the state of the PPT entry to *WRNET* to indicate that a write request has been issued (Section 4.3.2). When the default owner receives the *WREQ*, it looks up the page in its PPT. If the page is not present in the default owner (no PPT entry) it looks up its Migrated Page Table (MPT) to see if the page is owned by any node. If the page is not owned by any node, the default owner reads the page from disk and sends the entire page in a *PAGE* data message to node *W*. The default owner updates its MPT to indicate that *W* owns the page. When *W* receives the *PAGE* message, it marks the page private in its PPT.

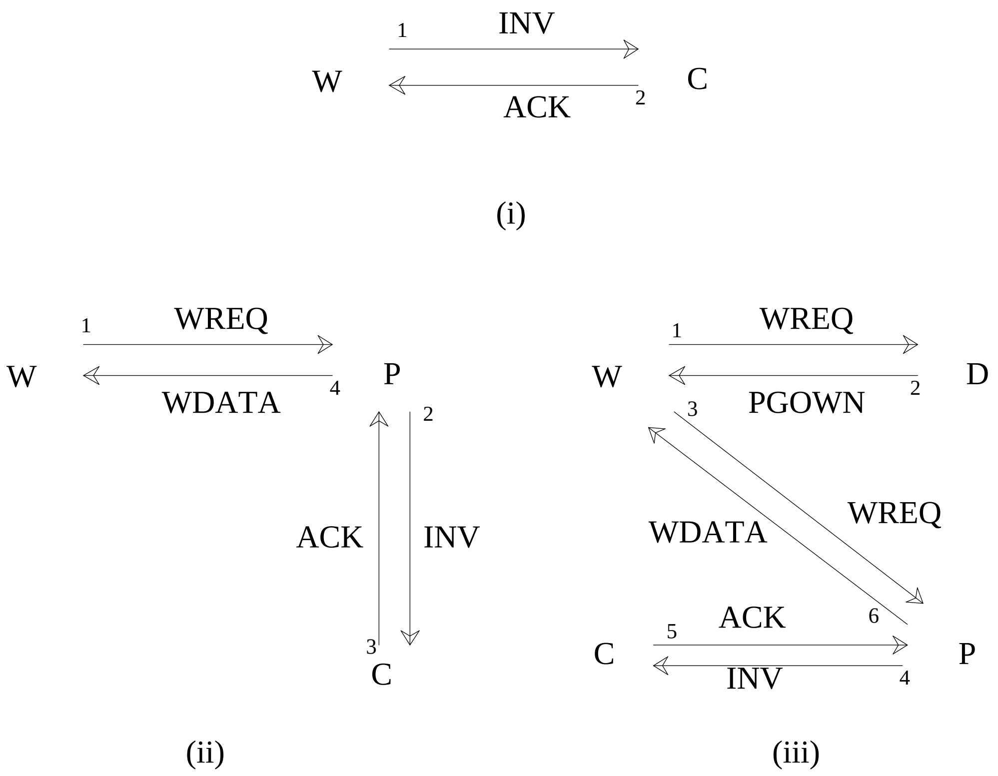


Figure 4.21: Additional protocol flow sequences for write requests. *W* is the write requester, *P* is the page owner, *D* is the default owner, and *C* is some node that has a read copy of the block. Protocol can flow as in case (i) when the *W* is the block owner. Case (ii) will occur when the page owner is also the block owner. Case (iii) also occurs when the page owner is also the block owner, but when *W* has no block of the page. Other flows are also possible, including those when the ownerships move.

Another possible write-miss scenario is where the requester *W* does not have any block of the page and the page is owned by some node other than the default page owner. In this case, the default page owner sends a *PGOWN* reply to *W* identifying the page owner *P*. When *W* receives the *PGOWN*, it checks its PPT entry. Since the PPT entry indicates a pending write request, *W* sends a *WREQ* to the page owner *P* identified in the *PGOWN* message. After this point, the protocol flows exactly as described in Figure 4.20 (where the node requesting the write knows the identity of the page owner) up to the point where *W* receives the *WDATA* message. When *W* receives the *WDATA* message, it allocates an an SPT entry for the page. *W* sets the state of the block in the SPT to *OWNEX*, and sets the states of all other blocks of the page to *ABSENT*. *W* also marks the identity of the page owner in the SPT entry. This flow of the write access and a few other flows are shown in Figure 4.21.

In the above discussion, we have covered cases where the write request is serviced once it is received. Under some situations, it may not be possible to service the write request when it is received. A write request can be denied service if the page owner has an outstanding read request for the block. When a read request for the block is in progress in the page owner, the access_waitlist component of the local state of the block in the page owner is set to *RDNET*. Since we support only one outstanding access on a block at a time, the page owner will send a *RETRY* message to the node requesting write access. A write request can also be denied service when another write request is already in progress for the block (access_waitlist of *WRNET* or *LOCK* in the local block state), or if the page is being replaced in the page owner (identified by the PPT meta-state bits associated with the page replacement mechanism). As we discussed in the context of read requests that are denied service (Section 4.3.4.1), the write request will be denied service so that the on-going write access or on-going replacement can complete. A write request is denied service by the current block owner if the current block owner is writing to the block. In this case, the current block owner sends a *RETRY* message to the page owner. The page owner must retry the *WRBCK* request later. All other cases when a write request is denied occur only at the page owner. The page owner sends a *RETRY* message to the node issuing the write request *W* in these cases. The write request must be retried by *W*.

4.3.4.3. Anatomy of a Replacement

A replacement occurs when there is a need to release resources allocated to a previous operation in order to allow a new operation to proceed. For the coherency protocol, the relevant resources are entries in the various tables and the local memory on each node. The tables where entries are dynamically allocated include the Present Pages Table (PPT), the Shared Pages Table (SPT) and the copy directory. An entry is allocated in the PPT when an operation requires a new page frame to be allocated (e.g., an access to a new page). An operation allocates an entry in the SPT when the operation causes the sharing of a new page. When a new node is allowed to share a page, a new entry must be allocated for the node in the copy directory. These operations can be blocked when entries needed by the operations are not available for allocation in these tables. To allow these operations to continue, space must be freed from these tables by replacing existing entries. We discuss the mechanism of PPT entry replacement, SPT entry replacement, and copy directory entry replacement in this sub-section.

When a node *N* requires a page frame to allocate a new page, a page must be replaced from its local memory. This frees up the PPT entry for use by the new page. The page to be replaced is chosen randomly. If the page randomly chosen is pinned from being replaced (e.g., due to outstanding requests), another page is chosen. The page that is picked for replacement can be private or shared (see Section 4.1). When a page is picked for replacement, its PPT entry is marked by setting the appropriate meta-state bit to indicate that the page is in replacement (see Section 4.1.1). From this point, the mechanism of replacement must clearly be different for replacing private and shared pages. In our implementation, the mechanism of replacing a shared page also depends on whether the page is owned by a remote node or by the local node *N*. We discuss the mechanism of replacement for each of these three cases: replacing a private page, replacing a shared page that is not owned by the node replacing the page and replacing a shared page that is owned by the node replacing the page.

Page replacement is simple when the page being replaced is currently private to the node *N*. The private page is written back to the default owner using a *PGBCK* message and the PPT entry is immediately reclaimed. When the default owner receives the *PGBCK* message, it writes the page to disk. The default owner also updates its Migrated Page Table (MPT) to indicate that the page is not owned by any node.

When a node *N* replaces a page that is owned by a remote node, *N* will relinquish the ownership of any blocks that it owns as well as any local read copies of blocks of the page. We discuss the mechanism of relinquishing ownership and then discuss the relinquishing of read copies. To relinquish block ownership, the node must write the block back to the page owner. *N* writes back every block that it owns exclusively (*OWNEX*) to the page owner using a *BLKBCK* message immediately. Node *N* can write back any block that is locally in the owned shared state only after it acquires exclusive ownership of the block. The local state of each block that is currently in the owned shared state (*OWNSH*) is changed to *OWNSH_XFR* to indicate that exclusive ownership is being acquired to return the block to the page owner. For each of these blocks, node *N* sends invalidate messages (*INV*) to nodes that have read copies indicated in the copy directory. Once all acknowledgements for the block are received by *N*, *N* writes the block back (*BLKBCK*) to the page owner and changes the local state of the block to *ABSENT*.

Node *N* relinquishes its read copies of blocks of the page being replaced by changing the local state of such blocks to *ABSENT*. For each of these blocks, *N* sends a *CPINV* message to the page owner indicating that the local copy of the block was invalidated. When the page owner receives the *CPINV* message from *N*, the page owner determines the owner of the block (from the local state of the block or by accessing the copy directory). When the block is owned by the page owner itself, the page owner removes *N* from the copy directory for the block. If the block is not owned by the page owner, the page owner forwards the *CPINV* message to the owner of the block. When the block owner receives the *CPINV* message, node *N* is removed from the copy directory.

All blocks of the page are relinquished by *N* when the local state of each block of the page is *ABSENT*. This completes the replacement of the page (which was shared but not owned by the node replacing the page) At this point, the PPT (and the SPT entry) entry associated with the replaced page is free to be reused.

Now, consider the scenario when the page being replaced to free up a PPT entry is owned by the node replacing the page (*N*) itself. In this case, the replacing node *N* first makes the page private to itself by evicting the page from all nodes that share the page. Subsequently, *N* can free the PPT entry by writing the entire page back to the default owner. *N* evicts the shared page from all nodes sharing the page by acquiring exclusive ownership of all blocks of the page. We describe the mechanism below.

When replacement of a shared page owned locally is initiated in node *N*, it modifies the local state of each block of the page to indicate that exclusive ownership is being acquired for the block. The local state of blocks in the *ABSENT*, *RDONL* and *OWNSH* states are changed to *ABS_WRNET*, *RDO_WRNET* and *OWNSH_NET* respectively. *N* can acquire exclusive ownership of these blocks by sending individual *WRBCK* messages to the owner of each block of the page that is not owned locally, and by individually invalidating (*INV*) each read copy of every block that is owned shared locally. As we mentioned in Section 4.3.3, to optimize performance, our scheme employs an alternative mechanism using the page invalidate (*PGINV*) message.

When *N* replaces a shared page that it owns (i.e., *N* is also the page owner), a page invalidate (*PGINV*) message is sent to each node that is indicated in the copy directory for the page in *N*. The nodes indicated in *N*'s (page owner's) copy directory are nodes that own any block of the page and nodes that possess a read copy of any block owned by the page owner *N*. Node *N* maintains the count of acknowledgements expected in its Pending Requests Cache. When a node receives a *PGINV* message, it performs three actions. First, it drops its read copies of any block of the page — the local state of each block of the page that is in the *RDONL* state is changed to *ABSENT*. Second, the node receiving the *PGINV* message acquires exclusive ownership of any blocks of the page that it owns so that the block may be written back to the page owner *N*. The node writes back all blocks which are locally in the *OWNEX* state to the page owner using *BLKBCK* messages and changes the local state of these blocks to *ABSENT*. The node changes the local state of blocks in the *OWNSH* state to *OWNSH_XFR*, and sends invalidate messages (*INV*) to all nodes with read copies of the block. When the node has received all acknowledgements (*ACK*) for the invalidation, it writes back the block to the page owner (*BLKBCK*) and changes the local state of the block to *ABSENT*. Lastly, when the node receiving the *PGINV* message has dropped all its read copies and written back all blocks that were owned locally, it sends an acknowledgement message (*ACK*) to the page owner *N* confirming the invalidation of the page.

The shared owned page being replaced by *N* is private to *N* when it has received all acknowledgements for its page invalidation and all blocks of the page have been written back (in a FIFO network, the *BLKBCK* messages would all be received before the acknowledgements are all received). *N* can free the PPT entry associated with the page by writing the page back to the default page owner using the *PGBCK*. As discussed previously (in the context of replacing private pages), the default owner will write the page to disk and update its MPT to indicate that the page is not owned by any node.

An SPT entry must be evicted from a node when no free SPT entries are available on that node, and an operation requires a new SPT entry to be allocated in that node. To free an SPT entry in node *N*, a page that is currently shared by *N* must be replaced. A shared page is chosen for replacement at random (by choosing an SPT entry to evict at random). When a candidate SPT entry has been chodes, the corresponding PPT entry of the page is identified from the page frame number field in the SPT (see Section 4.1.1). The appropriate meta-state bit in the PPT entry of the page is set to mark that the page is being replaced to free the SPT entry. As we described in our discussion of PPT replacements, the shared page chosen for replacement may be owned by the local node that is replacing the page (*N*) or may be owned by other remote node. When the page being replaced is not owned locally, *N* relinquishes ownership and read copies of all blocks of the page. The replacement mechanism in this case is exactly as we described for PPT replacements.

When the page being replaced to free up an SPT entry is owned by *N*, *N* will evict the page from all other nodes by acquiring exclusive ownership of all blocks of the page. The page is evicted from other nodes using the same mechanism described for PPT replacements. However, once *N* acquires exclusive ownership of all blocks of the page, the page is not discarded. The state of the page is changed to private instead of writing the page back to the default page owner (the PPT meta-state bits identify that the page is being replaced to free an SPT entry and not to free the PPT entry). Since an SPT entry need not be maintained for a private page, the SPT entry previously used by the page is free for reuse.

As we mentioned earlier, entries of the copy directory may also need to be evicted. When a new node shares blocks of a page from a node *N*, it must be indicated in the copy directory for the page. When no entry is available to add the node in the copy directory of the page on *N*, an entry must be evicted. In our Block Map per Processor (BMpP) copy directory, each entry of the copy directory identifies a specific node and the set of blocks of the page shared by that node. When a specific entry is chosen for eviction (randomly), the victim node identified by the entry (*V*) is forced to relinquish the blocks indicated in the entry. This frees the copy directory entry.

To force *V* to relinquish the blocks, node *N* must first determine which of the blocks marked in the copy directory entry of *V* are owned by *V* and which blocks are owned by *N* itself. This is determined by accessing the local state of these blocks in *N*. For each of these blocks that is not owned by *N* (i.e., owned by *V*), *N* forces *V* to relinquish the ownership of the block by requesting it to write the block back. The access_permit component of the block state in *N* is set to *WRNET* to indicate that ownership is being acquired, and a *WRBCK* message is sent to *V*. For each of the blocks that is owned by *N* (i.e., not owned by *V*), *N* forces *V* to relinquish the read copy by sending it an invalidate message (*INV*). In this case, the local state of the block in *N* is not changed. The state associated with the *WRBCK* and *INV* messages that are issued is indicated in the Pending Requests Cache on *N*. The copy directory entry is free to be reused at this point. However, eviction is complete when *N* receives all blocks that must be written back and all acknowledgements that must be received.

4.4. Simulation of Our DSM System Design

Distributed shared memory systems and other large scale systems are composed of several complex sub-systems including processors, caches, coherence engines and interconnection network. The performance of these complex systems cannot be adequately evaluated using analytical models. Analytical models of these large systems are unwieldy and they cannot accurately capture the intricate interactions between the component sub-systems. Actual implementation or detailed simulation is necessary to realistically evaluate and characterize the performance of distributed shared memory systems.

Hardware-supported distributed shared memory systems have been implemented both in industry [Burk92] and in academia [Leno93, Agar95, Kusk94]. These implementation efforts have provided valuable experience in the design of DSM systems and serve as good platforms for studying the use of DSM systems. Such actual system implementations allow architectural ideas to be rigorously tested. However, they take multiple man-years to complete and are expensive, especially when the system being designed is a large scale multiprocessor.

These hardware implementations also have limited flexibility in accommodating multiple designs. Most of the functionality in these hardware implementations is built into hardware components that cannot be easily changed. For example, Alewife [Agar95] implements its base protocol in a hardware finite state machine. The base coherence protocol can be modified only by an expensive re-design of this chip. The MAGIC chip used in the FLASH architecture [Kusk94] supports some degree of flexibility by implementing the protocol code in userhandlers which can be modified to change the protocol.

System-level simulations can be implemented more rapidly than hardware prototyping and offer greater flexibility in accommodating multiple designs. These simulations allow the system to be characterized over a range of system parameters (such as table sizes and queue sizes). Unlike prototype implementations, system simulations are not constrained to design for technology that is available at the time of evaluation. Simulations can be used to evaluate system designs for future technology, modeling components which are beyond the capability of current technology (e.g., faster memory or network, sophisticated processors etc). Simulations also ease the process of verifying and refining the design. Even hardware prototyping efforts use simulations in their initial phase to refine the design (e.g., the FLASH [Kusk94] implementation).

Simulation models can be based on trace-driven simulation or execution-driven simulation. With trace-driven simulation, the system is evaluated using address traces from a prior execution of the workload. The trace of addresses issued by the workload are collected prior to simulation when the workload executes on a different architecture. Execution-driven simulation is based on the actual execution of the workload on a simulated virtual machine. The virtual machine that is simulated is a close representation of the architecture being evaluated.

Trace-driven simulation is generally more efficient than execution-driven simulation since the workload is not executed. Since the workload is not executed during a trace-driven simulations, building these simulators is also easier than building execution-driven simulators. However, trace-driven simulations are likely to be less accurate than execution-driven simulations for several reasons. Firstly, the address traces are influenced by the architecture of the machine on which the traces were generated. Second, the ordering of the accesses in the workload may be perturbed by the process of collecting the traces. Third, if the trace is simply a sequence of addresses, there is no information about time between accesses. In the absence of such timing information, trace-driven simulation cannot measure the real indicators of system performance such as application execution time, processor utilization or network throughput. Trace-driven simulation also has the disadvantage that the storage needed for maintaining large address traces is quite high.

We first implemented a simple trace-driven simulator of our DSM design, allowing rapid evaluation of some of our preliminary designs. Subsequently, we have implemented an accurate execution-driven simulator of our design that served both for performance evaluation and for experimental validation of our protocols. We describe these implementations in this section.

4.4.1. Implementation of a Simple Trace-Driven Simulator

We have implemented a simple trace-driven simulator of our hierarchical scheme to obtain initial validation of our claims about the effectiveness of the scheme. The simulator performs all accesses atomically. This is accomplished by atomically performing all the state changes throughout the entire system required for a particular block access. This simple design allowed the simulator to be built quickly without the complexities associated with the implementation of a complete distributed directory coherence protocol, or of an interconnection network. The simulator implements sufficient detail to measure the hit ratio, number of coherence messages generated, and the usage of various mapping tables.

The simulator accurately implements all the mapping tables required by the hierarchical scheme: Present Pages Table (PPT), Shared Pages Table (SPT), Migrated Pages Table (MPT) and the copy directory. PPT and SPT entries have the same format shown in Section 4.1.1. However, since all accesses are atomic, we do not maintain access_waitlist as part of the block state in the SPT (Section 4.3.2). The simulator supports both Block Map per Processor (BMpP) and the Processor List per Block (PLpB) copy directory organizations. When there are no available entries in the PPT, SPT or the copy directory, a victim is picked randomly and evicted.

For each address in the input trace, the local mapping tables in the processor issuing the address are looked up. If the access can be satisfied locally, then the access is a ‘‘hit.’’ If the access cannot be satisfied locally, the mapping tables in all relevant remote nodes (page owner, block owner, read copy holders) are looked up and modified appropriately. The simulator does not implement the coherence protocol, but accurately counts the number of coherence messages that would have been sent for servicing remote accesses and evictions.

4.4.2. Implementation of an Execution-Driven Simulator

The execution-driven simulator of the hierarchically managed DSM was implemented on top of the Proteus [Brew92a] simulation environment. Proteus is a parallel architecture simulator that can simulate any generic MIMD multiprocessor. Different parallel architectures can be modeled and studied using the Proteus simulation framework. We describe the Proteus simulation environment in Section 4.4.2.1. We also discuss the writing of applications for this environment as well as the compilation and execution of the applications. Subsequently, in Section 4.4.2.2, we discuss the implementation of hierarchically managed DSM on Proteus.

4.4.2.1. The Proteus Simulation Environment

Proteus models the system as a set of nodes interconnected by some network. Each node in the system is viewed as a collection of threads. The threads correspond to the different components of the node: processor, memory and network interface. The processor is ''simulated'' by a thread of application code executing directly on the simulation host hardware. The memory and network threads are simulated in software. The behavior of the memory and network threads are described in memory and network engines which can be modified to simulate systems with different memory architectures and systems with different interconnection networks.

The Proteus simulation kernel is essentially a threads package that manages the threads associated with each node of the system. The threads corresponding to the processor, memory and network on each node in the system are created at the start of simulation. The run-time simulation engine first schedules the simulation of the processor of node 0. To ''simulate'' the processor, the application is executed on the simulation host. During execution, the application generates memory-access and interprocessor-communication events which must be serviced by the memory, network and remote processor threads. The run-time simulation engine schedules the different threads on the simulation host machine based on the time-ordered sequence of events (we will discuss the timestamping mechanism shortly). Each thread that is scheduled executes according to the behavior specified in its model and can generate events for other threads in the system. The executing thread returns control to the simulation engine when its task is completed or when it requires the service of another thread. The kernel includes a context-switching routine to allow control to be switched between the simulation engine and a thread.

The Proteus simulation environment includes a programming environment for application development as well as an operating system environment for run-time support. To support a shared memory programming environment, Proteus includes a preprocessor that extends the C language with new storage classes and operators that allow easy definition and use of shared memory variables. Proteus provides the application with primitives for reading from and writing to shared memory, test-test-and-set synchronization, interprocessor interrupts, and message passing. Proteus also supports barrier synchronization, which is implemented through interprocessor messages [Kopp95].

The Proteus kernel provides basic operating system functions required to run applications in a parallel environment. Proteus provides thread management routines that allow the application to dynamically create and destroy threads. Proteus also provides system calls to allocate and deallocate memory. Application threads executing on Proteus can allocate and deallocate regions of shared memory (local to the node or non-local) using these system calls.

In order to be executable in the Proteus simulation environment, an application must be written to conform to Proteus' programming paradigm and must be compiled using the Proteus compilation preprocessor. The application must use the programming constructs provided in Proteus's extended C language to identify all shared variables and all accesses to the shared address space. All other variables and accesses in the application are viewed as private to each process. The application must allocate memory and perform application thread management using the primitives provided in Proteus. Processes of the application must synchronize and communicate with each other using the synchronization and communication primitives provided by Proteus.

The Proteus compilation phase preprocesses the application code to modify it in two ways. First, all shared address space accesses and all interprocess communication and synchronization events in the application source code are translated into calls to corresponding routines in the memory and network engines. Second, the assembly code generated from the preprocessed application source is instrumented with additional assembly instructions. The assembly code augmentation adds code to count the number of instructions executed, the number of private load accesses and the number of private store accesses during application execution. Each basic block in the application assembly is augmented with instructions that accumulate the number of instructions, number of private load accesses and number of private store accesses in that basic block to global variables that maintain their overall counts. These global variables are maintained independently for each node in the model. At any stage of the simulation, the value of these global variables for a specific node is the total count of instructions executed, private load accesses and private store accesses on that node until that stage of application simulation.

Applications execute in Proteus under the control of the Proteus run-time simulation engine. The simulation engine initiates application execution by scheduling the processor thread of node 0 with a timestamp of 0. When a thread that is scheduled to run requires service from another thread (e.g., processor must access memory, memory must send a message on the network etc.), it generates an output event requesting the service and transfers control to the simulation engine. The thread generating the output event must timestamp the event appropriately. Processor threads timestamp their output events based on the number of instructions of application code that was executed since it was scheduled (Proteus assumes one processor cycle per instruction). Memory and network threads timestamp their output events based on the latency in generating the output events as modeled in their respective engines. The memory engine models the latency of various tables that must be accessed to service input memory events. The network engine models the latency in the network switches as well as the latency of network links. The simulation engine handles the task of scheduling the appropriate thread (processor, memory or network threads of any node) that can service the simulation event with the earliest timestamp. Application execution is complete when no thread is scheduled to run and no simulation events are pending.

Proteus advances the execution time of the application on each node through the latency of executing instructions which access only private data and through the latency of accessing shared data. The latency of executing instructions which access only private date is measured by counting the number of such instructions (each instruction is assumed to execute in one processor cycle). The latency of accessing shared data is measured by the latencies computed in the memory and network threads that participate in satisfying the access. The process of servicing the shared access request is initiated by the memory thread on the node that issued the request. Subsequently, a sequence of network and memory events may be generated to send requests to remote nodes and to access memory tables in the memory threads of remote nodes. As described earlier, each memory and network thread timestamps their output event based on the latency of servicing their input event. Hence, the timestamp of the event that eventually grants the requested access incorporates the cumulative latency in the network and remote memories. The difference between the timestamps of the event requesting access and the event granting access measures the latency of accessing shared data. The overall execution time of the application is the time of the node with the highest time.

4.4.2.2. Implementation of Hierarchically Managed DSM on Proteus

When an application executing under Proteus accesses shared memory, the processor thread executing the application code issues a memory event which must be serviced by memory threads. The behavior of these memory threads must be described in a software memory engine module. We have developed a memory engine for Proteus that accurately models our hierarchically managed DSM.

The memory engine implements all the hierarchical management mapping tables — Present Pages Table (PPT), Shared Pages Table (SPT), Migrated Pages Table (MPT) and the copy directory — for each node to support address translation. The implementation faithfully models the inverted table implementation of the PPT. PPT entries have the same format presented earlier in

Section 4.1.1. However, the entries also maintain a pointer to the corresponding page frame of memory. The SPT is accurately implemented and SPT entries for shared pages are dynamically allocated. The SPT entries support the full set of access_permits and access_waitlists associated with block states (Section 4.3.2). They also maintain a pointer to a copy directory as required by the hierarchical management scheme. Additional fields are maintained in the SPT entry to accumulate statistics about the page during simulation. The MPT is also implemented for each node to track the current owner of all pages default owned by that node. Our copy directory implementation in the memory engine supports both Block Map per Processor (BMpP) and Processor List per Block (PLpB) organizations. In both cases, the copy directory is dynamically allocated. A copy directory is allocated for a shared page on a node only when information about the sharing of blocks need to be maintained on that node.

Since the PPT, SPT and the copy directory are bounded, as they would be in a real system, entries may need to be replaced from these tables. All replacements from the PPT, SPT and copy directory are random in our memory engine. While they are performed exactly as described in Section 4.3.4.3, the selection of a replacement victim does not consume any latency in our model. When a PPT entry is not available to service a local processor's request, the network request is not issued until a PPT entry is available. Request and Reply buffers (as shown in Figure 4.1 in Section 4.1) are used to buffer remote requests and replies when SPT entries or copy directory entries are not available to service them. However, in our implementation, the size of these buffers is unbounded. Also, we do not model the latency of accessing these buffers. Our implementation does not model the presence of the Pending Requests Cache. The information contained in the Pending Requests Cache is maintained in the PPT, SPT and copy directory.

The pipelined implementation of the MMU control is not accurately modeled in the memory engine. Specifically, the model does not implement the pipeline stages in the MMU design shown earlier in Figure 4.1 of Section 4.1. However, the implementation models some degree of pipelining by allowing an access to be issued to the MMU tables when another access issued in a previous cycle may be in progress. The model does not implement a cache or a Translation Lookaside Buffer (TLB). However, as we will discuss in Chapter 5, the presence of a cache and a TLB can be approximately modeled by assigning appropriate latencies to accessing local data, the PPT and the SPT.

The memory engine also models the presence and access of a disk on each node. At the start of simulation, all pages reside on the disk of their respective default page owners. We use a first-touch policy to assign page ownership in our implementation. When a node receives a page from disk (read from disk and forwarded by the default owner as dictated by our coherency protocol) in reply to a read or write request, it is assigned ownership of the page.

The memory engine services all application accesses to the shared address space. The memory engine implements our distributed directory coherence protocol (described in Section 4.3) completely to service these accesses. All coherence protocol states and message types discussed in Section 4.3 are supported. If the block requested by an access on a node is available locally with sufficient access permission, the access is performed by the memory thread of that node without accessing the network. If the processor access cannot be satisfied locally, messages are sent to remote nodes to acquire access. The processor is blocked until the access is satisfied (i.e., the processor thread of this node will not be scheduled until the access can be satisfied).

To service shared accesses that cannot be satisfied locally, memory threads exchange coherence messages with memory threads on remote nodes. These coherence messages are sent from a memory thread on one node to a memory thread on another node through several network threads which model the latency of sending the message through the network. A memory thread sends coherence messages to the network by generating a event that must be serviced by the network thread of the local node. A memory thread receives a coherence message from the network when the network thread local to that node generates an event for the memory thread delivering the coherence message.

The simulation events generated by the memory and network threads provide all the information associated with the coherence message. This information includes the identity of the node sending the message, the type of coherence message (Section 4.3.3), the address being accessed, the node that is the destination of the message, and the identity of the client node that is performing the memory access. When a request is forwarded in the coherence protocol (Section 4.3.4), the client field is used to identify the node that originally issued the shared memory access. When a coherence message includes data, a data field is also present.

The network engine implements the network as an interconnection of switching elements which communicate by exchanging simulation events. Messages are not partitioned into multiple packets (each packet is a complete message). The network thread corresponding to each switching element (node) implements the routing logic and maintains, for each output port at that switching element, the earliest simulation time at which the output port is available. Network threads use the source and destination node identities in any incoming packet to determine the node next in the packet route and the output port to be used. The switching elements handle packets using the store-and-forward discipline. If the required output port is marked as available at a time earlier than the simulation time at which the incoming packet is received, the packet does not wait in the switching element buffer and is immediately transmitted through the output port. Otherwise, the incoming packet must be buffered and can be transmitted only at the time the output port becomes available. In either case, the time of transmission is determined and a simulation event forwarding the packet is generated for the network thread at the successor node. The timestamp of this simulation event is calculated as the sum of the time of transmission and the switching and link latencies of the packet (switching latency is a parameter and link latency is computed from the link bandwidth parameter and the size of the packet). The time at which the output port becomes available after transmitting this packet is updated likewise. The packet is routed to the destination node through a sequence of switching elements in the network. When the packet arrives at its destination node, the network thread at that node generates a simulation event that delivers the packet to the local memory thread.

Network packets support data structures that allow statistics about the link latency and queueing delay incurred by each packet to be accumulated. Network threads at each switching element in the path of a packet add the link latency and queueing delay incurred by the packet at that switching element to these data structures. The value of these data structures at the destination node is the overall link latency and overall queueing delay incurred by the packet.