CS 252 Graduate Computer Architecture
Midterm 2 Solutions

November 27, 2007

Part A: Snoopy Cache Coherent Shared Memory (18 points)

This problem improves the snoopy cache coherence protocol presented in the practice problems. As a **review** of that protocol:

> When multiple shared copies of a *modified* data block exist, one of the caches *owns* the current copy of the data block instead of the memory (the owner has the data block in the OS state). When another cache tries to retrieve the data block from memory, the owner uses *cache to cache intervention* (CCI) to supply the data block. CCI provides a faster response relative to memory and reduces the memory bandwidth demands. However, when multiple shared copies of a *clean* data block exist, there is no owner and CCI is *not* used when another cache tries to retrieve the data block from memory.

To enable the use of CCI when multiple shared copies of a *clean* data block exist, we introduce a new cache data block state: *Clean owned shared* (COS). This state can only be entered from the clean exclusive (CE) state. The state transition from CE to COS is summarized as follows:

initial state	other cached	ops	actions by this cache	final state
cleanExclusive (CE)	no	**CR**	**CCI**	**COS**

Summary of the cache bus transactions (**identical to practice problems**):

- Coherent Read (CR): issued by a cache on a read miss to load a cache line.
- Coherent Read and Invalidate (CRI): issued by a cache on a write-allocate after a write miss.
- Coherent Invalidate (CI): issued by a cache on a write hit to a block that is in one of the shared states.
- Block Write (WR): issued by a cache on the write-back of a cache block.
- Coherent Write and Invalidate (CWI): issued by an I/O processor (DMA) on a block write (a full block at a time).
- Cache to Cache Intervention (CCI): used by a cache to satisfy other caches' read transactions when appropriate. A CCI intervenes and overrides the answers normally supplied by memory. Data should be supplied using CCI whenever possible for faster response relative to the memory. However, only the cache that *owns* the data can respond by CCI.

Summary of the possible cache data block states (**differences from practice problems highlighted in bold**):

- Invalid (I): Block is not present in the cache.
- Clean exclusive (CE): The cached data is consistent with memory, and no other cache has it. **This cache is responsible for supplying this data instead of memory when other caches request copies of this data.**
- Owned exclusive (OE): The cached data is different from memory, and no other cache has it. This cache is responsible for supplying this data instead of memory when other caches request copies of this data.
- Clean shared (CS): The data has not been modified by the corresponding CPU since cached. Multiple CS copies and at most one OS copy of the same data could exist.

- Owned shared (OS): The data is different from memory. Other CS copies of the same data could exist. This cache is responsible for supplying this data instead of memory when other caches request copies of this data. (Note, this state can only be entered from the OE state.)
- **Clean owned shared (COS): The cached data is consistent with memory. Other CS copies of the same data could exist. This cache is responsible for supplying this data instead of memory when other caches request copies of this data. (Note, this state can only be entered from the CE state.)**

Question 1 (8 points)

Fill out the state transition table for the new COS state:

initial state	other cached	ops	actions by this cache	final state
COS	yes	none	none	**COS**
		CPU read	none	COS
		CPU write	CI	OE
		replace	none	I
		CR	CCI	COS
		CRI	CCI	I
		CI	none	I
		WR	impossible	--------
		CWI	none	I

We also gave credit if you assumed that a WR is done when (clean) data is being replaced:

Replace: WR, I
WR: none, COS

Question 2 (6 points)

The COS protocol is not ideal. Complete the following table to show an example sequence of events in which multiple shared copies of a clean data block (*block B*) exist, but CCI is *not* used when another cache (*cache 4*) tries to retrieve the data block from memory.

Cache transaction	source for data	state for data block B			
		cache 1	cache 2	cache 3	cache 4
0. *initial state*	—	**I**	**I**	**I**	**I**
1. cache 1 reads data block B	**memory**	**CE**	**I**	**I**	**I**
2. cache 2 reads data block B	**CCI**	**COS**	**CS**	**I**	**I**
3. cache 3 reads data block B	**CCI**	**COS**	**CS**	**CS**	**I**
4. cache 1 replaces data block B	---	**I**	**CS**	**CS**	**I**
5. cache 4 reads data block B	**memory**	**I**	**CS**	**CS**	**CS**

Question 3 (4 points)

As an alternative protocol, we could eliminate the CE state entirely, and transition directly from I to COS when the CPU does a read and the data block is not in any other cache. This modified protocol would provide the same CCI benefits as the original COS protocol, but its performance would be worse. **Explain the advantage of having the CE state.** You should not need more than one sentence.

When the CPU does a write, it can change a cache block from CE to OE with no bus operation, but to transition from COS to OE it must first broadcast a CI on the bus to invalidate any shared (CS) copies of the block.

Part B: Implementing Directories (28 points)

Ben Bitdiddle is implementing a directory-based cache coherence invalidate protocol for a 64-processor system. He first builds a smaller prototype with only 4 processors to test out the directory-based cache coherence protocol described in the practice problems. (A copy of the protocol is provided at the end of this test.) To implement the list of sharers, **S**, kept by the **home** site, he maintains a bit vector per cache block to keep track of all the sharers. The bit vector has one bit corresponding to each processor in the system. The bit is set to one if the processor is caching a shared copy of the block, and zero if the processor does not have a copy of the block. For example, if Processors 1 and 3 are caching a shared copy of some data, the corresponding bit vector would be `1010` to represent processors 3, 2, 1, 0 respectively .

Question 4 (2 points)

The bit vector worked well for the 4-processor prototype, but when building the actual 64-processor system, Ben discovered that he did not have enough hardware resources. **Assume each cache block is 32 bytes.** What is the overhead of maintaining the sharing bit vector for a 4-processor system, **as a ratio of bit vector (overhead) bits to data storage bits**? What is the overhead for a 64-processor system?

Overhead for a 4-processor system: _____4/(32*8)= 1/64_____

Overhead for a 64-processor system: _____64/(32*8) = 1/4**_____**

Question 5 (8 points)

Since Ben does not have the resources to keep track of all potential sharers in the 64-processor system, he decides to limit **S** to keep track of only 1 processor using its 6-bit ID as shown in Figure B.1 (**single-sharer scheme**). When there is a load request to a shared cache block, Ben invalidates the existing sharer to make room for the new sharer (home sends an invalidate-request to the existing sharer, the existing sharer sends an invalidate-reply to home, home replaces the existing sharer's ID with the new sharer's ID and sends a load-reply to the new sharer).

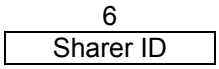

Figure B.1

Consider a 64-processor system. To compare the efficiency of the bit-vector scheme and single-sharer scheme, **fill in the number of invalidate-requests** that are generated by the protocols for each step in the following two sequences of events. Assume cache block **B** is uncached initially (H-uncached) for both sequences.

Sequence 1	**bit-vector scheme # of invalidate-requests**	**single-sharer scheme # of invalidate-requests**
Processor #0 reads **B**	0	0
Processor #1 reads **B**	0	1
Processor #0 reads **B**	0	1

Sequence 2	**bit-vector scheme # of invalidate-requests**	**single-sharer scheme # of invalidate-requests**
Processor #0 reads **B**	0	0
Processor #1 reads **B**	0	1
Processor #2 writes **B**	2	1

Name_____

Question 6 (8 points)

Ben thinks that he can improve his single-sharer scheme by adding an extra "**global bit**" to **S** as shown in Figure B.3 (**global-bit scheme**). The global bit is set when there is more than 1 processor sharing the data, and zero otherwise.

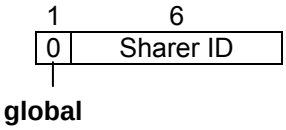

Figure B.3

When the global bit is set, home stops keeping track of a specific sharer and assumes that all processors are potential sharers.

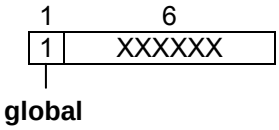

Figure B.4

Consider a 64-processor system. To determine the efficiency of the global-bit scheme, **fill in the number of invalidate-requests** that are generated for each step in the following two sequences of events. Assume cache block **B** is uncached initially (H-uncached) for both sequences.

Sequence 1	**global-bit scheme # of invalidate-requests**
Processor #0 reads **B**	0
Processor #1 reads **B**	0
Processor #0 reads **B**	0

Sequence 2	**global-bit scheme # of invalidate-requests**
Processor #0 reads **B**	0
Processor #1 reads **B**	0
Processor #2 writes **B**	64 or 63 (if we do not invalidate the writer)

Name_____

Question 7 (10 points)

Ben decides to modify the directory-based protocol from the practice problems for his global-bit scheme for a **64-processor system**. Your job is to complete the following table (Table B.1) for him.

Use the same assumptions for the interconnection network, cache states, home directory states, and protocol messages as the practice problems (a copy is attached to the end of this test). However, **H-shared[*S*]** now means that there is only one processor sharing the cache data (global bit is unset), and **H-shared[*all*]** means the global bit is set.

Use *k* to represent the site that issued the received message. For H-transient state, use *j* to represent the site that issued the original protocol request (load-request/store-request).

No.	Current State	Message Received	Next State	Action
1	H-uncached	load-request	H-shared[*k*]	load-reply -> *k*
2	H-uncached	store-request	H-modified[*k*]	store-reply -> *k*
3	H-shared[*S*]	load-request	H-shared[all]	load-reply->k
4	H-shared[*all*]	load-request	H-shared[all]	Load-reply->k
5	H-shared[*S*]	store-request	H-transient <count = 1>	Invalidate-request->S
6	H-shared[*all*]	store-request	H-transient <count=64 or 63>	For(i=0; i<count; i++) Invalidate-request->i
7	H-modified[*m*]	load-request	H-transient	shared-copy-request → *m*
8	H-transient[count > 1]	invalidate-reply	H-transient[--count]	nothing
9	H-transient[count = 1]	invalidate-reply	H-modified[*j*]	store-reply → *j*
10	H-transient	shared-copy-reply	H-shared[all]	Data->memory Load-reply->j

Table B.1 Partial List of Home Directory State Transitions

Part C: Relaxed Memory Models (14 points)

Consider a system which uses Weak Ordering, meaning that a read or a write may complete before a read or a write that is earlier in program order if they are to different addresses and there are no data dependencies.

Our processor has four fine-grained memory barrier instructions, same as in the practice problems. Below is the description of these instructions.

- **MEMBAR$_{RR}$** guarantees that all read operations initiated before the **MEMBAR$_{RR}$** will be seen before any read operation initiated after it.
- **MEMBAR$_{RW}$** guarantees that all read operations initiated before the **MEMBAR$_{RW}$** will be seen before any write operation initiated after it.
- **MEMBAR$_{WR}$** guarantees that all write operations initiated before the **MEMBAR$_{WR}$** will be seen before any read operation initiated after it.
- **MEMBAR$_{WW}$** guarantees that all write operations initiated before the **MEMBAR$_{WW}$** will be seen before any write operation initiated after it.

We will study the interaction between two processes on different processors on such a system:

P1	P2
P1.1: LW R2, 0(R8)	P2.1: LW R4, 0(R9)
P1.2: SW R2, 0(R9)	P2.2: SW R5, 0(R8)
P1.3: LW R3, 0(R8)	P2.3: SW R4, 0(R8)

We begin with following values in registers and memory (same for both processes):

register/memory	Contents
R2	0
R3	0
R4	0
R5	8
R8	0x01234564
R9	0x89abcdec
M[R8]	6
M[R9]	7

After both processes have executed, is it possible to have the following machine state? Please circle the correct answer. If you circle **Yes**, please provide sequence of instructions that lead to the desired result (one sequence is sufficient if several exist). If you circle **No**, please explain which ordering constraint prevents the result.

Question 8 (3 points)

Is it possible for M[R8] to hold 7 and for M[R9] to hold 6?

Yes **No**

If yes, provide sequence of instructions. If no, explain which ordering constraint prevents the result.

P1.1 P2.1 P1.2 P1.3 P2.2 P2.3

Question 9 (3 points)

Is it possible for M[R8] to hold 6 and for M[R9] to hold 7?

Yes **No**

If yes, provide sequence of instructions. If no, explain which ordering constraint prevents the result.

This result would require that the memory contents don't change. Since each thread reads a data value from one address and writes it to the other address (and the read and write of each thread cannot be reordered due to a data dependence), this scenario is impossible.

Name_____

Now consider the same program, but with two **MEMBAR** instructions.

P1	P2
P1.1: LW R2, 0(R8)	P2.1: LW R4, 0(R9)
P1.2: SW R2, 0(R9)	MEMBAR_{RW}
MEMBAR_{WR}	P2.2: SW R5, 0(R8)
P1.3: LW R3, 0(R8)	P2.3: SW R4, 0(R8)

We want to compare execution of the two programs on our system.

Question 10 (4 points)

If both M[R8] and M[R9] contain 6, is it possible for R3 to hold 8?

Without **MEMBAR** instructions? **Yes** **No**

If yes, provide sequence of instructions. If no, explain which ordering constraint prevents the result.

P1.1 P1.2 P2.1 P2.2 P1.3 P2.3

With **MEMBAR** instructions? **Yes** **No**

If yes, provide sequence of instructions. If no, explain which ordering constraint prevents the result.

P1.1 P1.2 P2.1 P2.2 P1.3 P2.3

Name_____

Question 11 (4 points)

Is it possible for both M[R8] and M[R9] to hold 8?

Without **MEMBAR** instructions? **Yes** **No**

If yes, provide sequence of instructions. If no, explain which ordering constraint prevents the result.

P2.2 P1.1 P1.2 P2.1 P2.3 P1.3

With **MEMBAR** instructions? **Yes** **No**

If yes, provide sequence of instructions. If no, explain which ordering constraint prevents the result.

P2's $MEMBAR_{RW}$ prevents P2.2 from executing before P2.1, which is the only way to get 8 in both memory locations.

Part D: Multithreading (20 points)

This problem evaluates the effectiveness of multithreading using a simple database benchmark. The benchmark searches for an entry in a linked list built from the following structure, which contains a key, a pointer to the next node in the linked list, and a pointer to the data entry.

```
struct node {
     int key;
     struct node *next;
     struct data *ptr;
}
```

The following MIPS code shows the core of the benchmark, which traverses the linked list and finds an entry with a particular key. Assume MIPS has no delay slots.

```
      ;
      ; R1: a pointer to the linked list
      ; R2: the key to find
      ;
loop: LW      R3, 0(R1)  ; load a key
      LW      R4, 4(R1)  ; load the next pointer
      SEQ     R3, R3, R2 ; set R3 if R3 == R2
      BNEZ    R3, End    ; found the entry
      ADD     R1, R0, R4 ; copy R4 to R1
      BNEZ    R1, Loop   ; check the next node
End:
      ; R1 contains a pointer to the matching entry or zero
      ; if not found
```

We run this benchmark on a single-issue in-order processor. The processor can fetch and issue (dispatch) one instruction per cycle. If an instruction cannot be issued due to a data dependency, the processor stalls. Integer instructions take one cycle to execute and the result can be used in the next cycle. For example, if `SEQ` is executed in cycle 1, `BNEZ` can be executed in cycle 2. We also assume that the processor has a perfect branch predictor with no penalty for both taken and not-taken branches.

Question 12 (5 points)

Assume that our system does not have a cache. Each memory operation directly accesses main memory and takes 100 CPU cycles. The load/store unit is fully pipelined, and non-blocking. After the processor issues a memory operation, it can continue executing instructions until it reaches an instruction that is dependent on an outstanding memory operation. How many cycles does it take to execute one iteration of the loop in steady state?

Since there is no penalty for conditional branches, instructions take one cycle to execute unless there is a dependency problem. The following table summarizes the execution time for each instruction. From the table, the loop takes **104 cycles** to execute.

Instruction	Start Cycle	End Cycle
LW R3, 0(R1)	1	100
LW R4, 4(R1)	2	101
SEQ R3, R3, R2	101	101
BNEZ R3, End	102	102
ADD R1, R0, R4	103	103
BNEZ R1, Loop	104	104

Question 13 (5 points)

Now we add zero-overhead multithreading to our pipeline. A processor executes multiple threads, each of which performs an independent search. Hardware mechanisms schedule a thread to execute each cycle.

In our first implementation, the processor switches to a different thread every cycle using fixed round robin scheduling (similar to CDC 6600 PPUs). Each of the N threads executes one instruction every N cycles. What is the **minimum** number of threads that we need to fully utilize the processor, i.e., execute one instruction per cycle?

If we have N threads and the first load executes at cycle 1, SEQ, which depends on this load, is considered for execution at cycle 2*N + 1. To fully utilize the processor, we need to hide the 100-cycle memory latency, so 2*N + 1 ≥ 101. The minimum number of threads needed is **50**.

Name_____

Question 14 (5 points)

How does multithreading affect throughput (number of keys the processor can find within a given time) and latency (time the processor takes to find an entry with a specific key)? Assume the processor switches to a different thread every cycle and is fully utilized. Check the correct boxes.

	Throughput	**Latency**
Better	√	
Same		
Worse		√

Question 15 (5 points)

We change the processor to only switch to a different thread when an instruction cannot execute due to data dependency. What is the minimum number of threads to fully utilize the processor now? Note that the processor issues instructions in-order in each thread.

In steady state, each thread can execute 6 instructions (SEQ, BNEZ, ADD, BNEZ, LW, LW). Therefore, to hide 98 cycles between the second LW and SEQ, a processor needs ceiling(98/6)+1 **= 18** threads.

Information for Part B (from practice problems):

Directory-based Cache Coherence Protocol

This CCDSM (cache-coherent distributed shared memory) system consists of a number of sites connected by an interconnection network. As shown in Figure 1, each *site* has a processor, an L1 cache, a shared memory, and a protocol processing component (PP). The PP implements global cache coherence using a directory-based cache coherence protocol. For each cache line, we maintain a cache state to specify the current coherence state of the cache line. For each memory block, we maintain a directory entry to record the sites that are currently caching that block. For every global address, there is a *home* site where the physical memory and directory entry is maintained. Assume that the home site can be determined by the global address using its most significant bits.

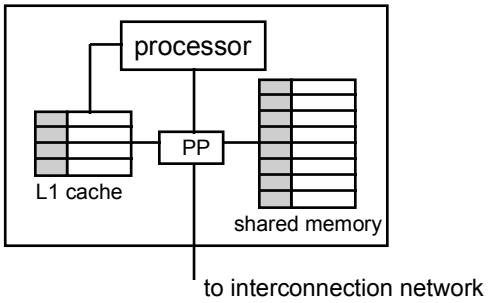

Figure 1: Site Configuration

A simple full-map directory structure is used. Each directory entry keeps a complete record of the sites that are sharing the memory block. The most common implementation keeps a bit-vector in each directory entry. The bit-vector has one bit for each site, indicating if a valid copy of the memory block is cached at that site. A dirty bit is also needed to indicate if the block has been modified. Unlike bus-based snoopy protocols, the directory-based protocol does not rely on broadcast to invalidate stale copies. Instead, because the locations of shared copies are known, cache coherence can be achieved by sending point-to-point protocol messages to only the sites that have cached the accessed memory block. The elimination of broadcast overcomes the major limitation on scaling cache coherent systems to parallel machines with a large number of processors.

The PPs are responsible for servicing memory access instructions, processing protocol messages, and maintaining cache line states and home directory states. When the processor issues a memory access instruction, the PP checks the addressed cache line's state. If the cache state shows that the instruction cannot be completed locally, the PP suspends the instruction, and sends a protocol request message to the corresponding home site. When this request arrives at the home site, the PP at the home site checks the home directory state, and sends a protocol reply message back to the requesting site to supply the requested data and/or exclusive ownership (in order to do this, the home site may need to obtain, from a remote site, the most up-to-date data and/or exclusive ownership if the memory block has been modified; or invalidate all the shared copies if the memory block is shared and the protocol message received is a store-request). At the requesting site, when the PP receives the protocol reply message, it resumes the suspended memory access instruction.

We make the following assumptions about the interconnection network:

- Message passing is reliable, and free from deadlock, livelock and starvation. In other words, the transfer latency of any protocol message is finite.
- Message passing is FIFO. That is, protocol messages with the same source and destination sites are always received in the same order as that in which they were issued.

Memory instructions: The basic memory access instructions are load and store. A load instruction reads the most up-to-date value of a given location, while a store instruction writes a specific value to a given location. We also need to consider cache replacement operations. A replace operation invalidates a cache line and, if the cache line has been modified, writes the modified data back to memory.

Both load and store are processor-issued instructions. Replace, on the other hand, is normally caused by a load/store instruction when a read/write miss leads to an associative conflict in the cache. In this situation, the load/store instruction cannot be processed before the replace (which is a "side-effect" of the load/store instruction) operation is completed. A cache line in a transient state cannot be replaced.

Cache states: For each cache line, there are 4 possible states:

- C-invalid: The line has no valid data.
- C-shared: The accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid.
- C-modified: The accessed data is exclusively resident in this cache, and has been modified. Memory does not have the most up-to-date data.
- C-transient: The accessed data is in a *transient* state (for example, the site has just issued a protocol request, but has not received the corresponding protocol reply).

Home directory states: For each memory block, there are 4 possible states:

- H-uncached: The memory block is not cached by any site. Memory has the most up-to-date data.
- H-shared[*S*]: The memory block is shared by the sites specified in *S* (*S* is a set of sites). The data in memory is also valid.
- H-modified[*m*]: The memory block is exclusively cached at site *m*, and has been modified at that site. Memory does not have the most up-to-date data.
- H-transient: The memory block is in a transient state (for example, the home site has just sent a protocol request to the modified site in order to obtain the most up-to-date data, but has not received the corresponding protocol reply). A counter, *count*, is needed when H-transient represents a transient state in which the home site is waiting for the acknowledgments to the invalidation requests it has issued.

Protocol messages: There are 12 different protocol messages, which are summarized in the following table (their meaning will become clear later). A protocol message includes the message type, the accessed memory address and, if necessary, the requested or written-back data. A protocol message usually comes in a *request* and *reply* pair. However, there are two exceptions: write-back and retry. Write-back writes a modified cache line back to the main memory. This is a one-way message that does not need a reply (compared with protocol requests, this saves one reply message). Retry is a NAK (negative acknowledgment) message which indicates that

something abnormal has happened, and some request cannot be processed and should be retried later. This is possible since the parallel system runs in a distributed way so that operations (e.g. sending a protocol message from one site to another) cannot be treated as atomic operations.

No.	Message Type	Includes data?
1	load-request	no
2	store-request	no
3	shared-copy-request	no
4	exclusive-copy-request	no
5	invalidate-request	no
6	load-reply	yes
7	store-reply	yes
8	shared-copy-reply	yes
9	exclusive-copy-reply	yes
10	invalidate-reply	no
11	write-back	yes
12	retry	no

The behavior of the PP can be defined by two finite state machines: one for cache line states, the other for home directory states. In this problem, we consider a very simple invalidation-based cache coherence protocol that implements the *sequential consistency* memory model. A brief (but neither formal nor complete) description is given below to help you understand the protocol.

Cache state transitions:

When the processor issues a load instruction,

- If the cache state is C-shared or C-modified, the PP supplies the processor the data from the cache. The cache state is not changed.
- If the cache state is C-invalid, the PP suspends the load instruction, and sends a load-request to the accessed memory's home site to request the data. The cache state is changed to C-transient. Later when the corresponding load-reply arrives, the PP places the data in the cache, changes the cache state to C-shared, and resumes the suspended load instruction.

When the processor issues a store instruction,

- If the cache state is C-modified, the PP allows the processor to write to the cache. The cache state is not changed.
- If the cache state is C-invalid or C-shared, the PP suspends the store instruction, and sends a store-request to the accessed memory's home site to request the data and exclusive ownership. The cache state is changed to C-transient. Later when the corresponding store-reply arrives, the PP places the data in the cache, changes the cache state to C-modified, and resumes the suspended store instruction.

When a replace operation happens,

- If the cache state is C-shared, the PP simply changes the cache state to C-invalid.

- If the cache state is C-modified, the PP sends a write-back message to the home site to write the modified data to memory, and changes the cache state to C-invalid.

Home directory state transitions:

When a load-request from site *k* arrives at the home site,

- If the home directory state is H-uncached, the PP sends a load-reply to site *k* to supply the requested data. The directory state is changed to H-shared[*S*], where $S = \{k\}$.
- If the home directory state is H-shared[*S*], the PP sends a load-reply to site *k* to supply the requested data. The directory state is changed to H-shared[*S'*], where $S' = S \cup \{k\}$.
- If the home directory state is H-modified[*m*], the PP sends a shared-copy-request to site *m* in order to obtain the most up-to-date data. The directory state is changed to H-transient. Later when the corresponding shared-copy-reply arrives at the home site, the PP updates memory, sends a load-reply to site *k* to supply the requested data, and then changes the directory state to H-shared[*S*], where $S = \{m, k\}$.

When a store-request from site *k* arrives at the home site,

- If the home directory state is H-uncached, the PP sends a store-reply to site *k* to supply the requested data and exclusive ownership. The directory state is changed to H-modified[*k*].
- If the home directory state is H-shared[*S*], the PP sends an invalidate-request to each of the sites specified in *S*. The directory state is then changed to H-transient, with a dedicated counter initialized to the number of invalidate-requests that have been issued. Later when all the invalidations have been acknowledged, the PP sends a store-reply to site *k*, and changes the directory state to H-modified[*k*].
- If the home directory state is H-modified[*m*], the PP sends a exclusive-copy-request to site *m* in order to obtain the most up-to-date data and exclusive ownership. The directory state is then changed to H-transient. Later when the corresponding exclusive-copy-reply is received, the PP sends a store-reply to site *k*, and changes the directory state to H-modified[*k*].

When a write-back from site *m* arrives at the home site,

- If the home directory state is H-modified[*m*] or H-transient, the PP updates the memory with the write-back data, and changes the directory state to H-uncached.

*Note: The cache state transitions described above are for each physical address at the granularity of the cache line size.