CE658 Parallel Computer Architecture
Fall 2018

Homework 3

Due Tuesday 29/01/2019

1) Snoop protocol performance (15 points)

Assume a snoopy system with the following message latencies. Determine **how many bus cycles** the trace will require to complete under the MSI cache coherence protocol using *a) Cache invalidation,* and *b) Cache update*. Cache update requires that each time a cache writes to a memory location X, it also broadcast the new value of X to the bus. Caches are initially empty.

- A bus read transaction that is satisfied by main memory takes 30 bus cycles
- A bus read transaction that is satisfied by another cache takes 20 cycles. If main memory must be updated, the supplier sets a 'writeback' line and memory is updated at no additional cost
- A bus invalidation message takes 4 bus cycles
- A bus update message takes 22 bus cycles. It updates both memory and caches

Processor 1 reads from Memory Location 1
Processor 1 writes to Memory Location 1
Processor 2 reads from Memory Location 1
Processor 3 reads from Memory Location 1
Processor 2 writes to Memory Location 1
Processor 1 reads from Memory Location 1
Processor 3 reads from Memory Location 1
Processor 3 writes to Memory Location 1
Processor 1 reads from Memory Location 1
Processor 2 reads from Memory Location 1
Processor 2 reads from Memory Location 1

2) Directory-based cache coherence (25 points, 5*5)

In this problem we consider a CCDSM (cache-coherent distributed shared memory) system. The 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.

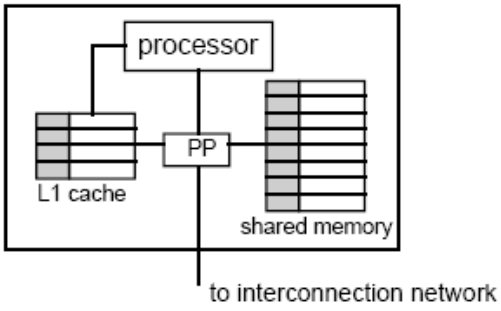

Figure 1: Site Configuration

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 $S = \{m, k\}$.

1a)

Table 1 shows the cache state transitions of the protocol (note that some tricky transitions are intentionally ignored here). The "current state" is the current cache line state. The "event received" is the event that the PP receives, which can be a load/store instruction, a replace operation, or a protocol message issued from the home site. The "next state" is the next cache line state after the PP processes the received event. The "action" is what the PP must do when processing the received event. This usually includes generating some new protocol message, placing some data in the cache, and so on.

Complete Table 1.

1b)

Table 2 shows the home directory state transitions of the protocol (note that some tricky transitions are intentionally ignored here). The "current state" is the current home directory state. The "message received" is the protocol message that the PP receives. The "next state" is the next directory state after the PP processes the received message. The "action" is what the PP must do when processing the received event. This usually includes generating some new protocol message(s), updating memory with the most up-to-date data, and so on. We use k to represent the site that issued the received message. For H-transient state, we use j to represent the site that issued the *original* protocol request (load-request/store-request).

Complete Table 2.

1c)

Consider the situation in which the home site sends an exclusive-copy-request to a site. This can only happen when the home directory shows that the modified copy resides at that site. The home site intends to obtain the most up-to-date data and exclusive ownership, and then supply them to another site that has issued a store-request. In Table 1, the last row (line 19) specifies the PP behavior when the current cache state is C-transient (not C-modified) and an exclusive-copy-request is received.

Give a simple scenario that causes this situation. You should explain your answer clearly.

1d)

FIFO message passing is a necessary assumption for the correctness of the protocol. Assume now that the network is non-FIFO. Give a simple scenario that shows how the protocol fails.

1e)

In the current scheme, when a replace operation happens, the PP simply changes the cache state to C-invalid, but does not inform the home node that the local node is no longer a sharer. Explain why this still preserves global cache coherence. Also describe the advantage(s) and disadvantage(s) of this scheme, compared with the scheme of having the PP always inform the home node that the local node is no longer a sharer after every replace operation.

No.	Current State	Event Received	Next State	Action
1	C-invalid	load	C-transient	load-request -> home
2	C-invalid	store		
3	C-invalid	invalidate-request		
4	C-invalid	shared-copy-request		
5	C-invalid	exclusive-copy-request		
6	C-shared	load		processor reads cache
7	C-shared	store		
8	C-shared	replace		nothing
9	C-shared	invalidate-request		
10	C-modified	load		
11	C-modified	store		
12	C-modified	replace		
13	C-modified	shared-copy-request		
14	C-modified	exclusive-copy-request		
15	C-transient	load-reply		data -> cache, processor reads cache
16	C-transient	store-reply		
17	C-transient	invalidate-request		
18	C-transient	shared-copy-request		
19	C-transient	exclusive-copy-request		

Table 1: Cache State Transitions

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]	
3	H-shared[S]	load-request	H-shared[$S \cup \{k\}$]	
4	H-shared[S]	store-request		
5	H-modified[m]	load-request		
6	H-modified[m]	store-request		
7	H-modified[m]	write-back		data -> memory
8	H-transient	load-request		
9	H-transient	store-request		
10	H-transient	write-back		
11	H-transient[count > 1]	invalidate-reply	H-transient[--count]	nothing
12	H-transient[count = 1]	invalidate-reply		
13	H-transient	shared-copy-reply		
14	H-transient	exclusive-copy-reply		

Table 2: Home Directory State Transitions

3) Synchronization , Sequential Consistency, Relaxed Memory Models (30 points) (10+10+10)

Cy D. Fect wants to run the following code sequences on processors P1 and P2, which are part of a two-processor MIPS64 machine. The sequences operate on memory values located at addresses A, B, C, D, E, F, and G, which are all sequentially located in memory (e.g. B is 8 bytes after A). Initially, M[A], M[B], M[C], M[D], and M[E] are all 0. M[F] is 1, and M[G] is 2. For each processor, R1 contains the address A (all of the addresses are located in a shared region of memory). Also, remember that for a MIPS processor, R0 is hardwired to 0. In the below sequences, a semicolon is used to indicate the start of a comment.

```
P1                                  P2
ADDI R2,R0,#1   ;R2=1               ADDI R2,R0,#1   ;R2=1
SD   R2,0(R1)   ;A=R2               SD   R2,8(R1)   ;B=R2
LD   R3,40(R1) ;R3=F                LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3                SD   R3,16(R1) ;C=R3
LD   R4,8(R1)   ;R4=B               LD   R4,0(R1)   ;R4=A
SD   R4,24(R1) ;D=R4                SD   R4,32(R1) ;E=R4
```

1a)

If Cy's code runs on a system with a sequentially consistent memory model, what are the possible results of execution? List all possible results in terms of values of M[C], M[D], and M[E] (since the values in the other locations will be the same across all possible execution paths).

Assume now that Cy's code is run on a system that does not guarantee sequential consistency, but that memory dependencies are not violated for the accesses made by any individual processor. The system has a MEMBAR memory barrier instruction that guarantees the effects of all memory instructions executed before the MEMBAR will be made globally visible before any memory instruction after the MEMBAR is executed.

Add MEMBAR instructions to Cy's code sequences to give the same results as if the system were sequentially consistent. Use the minimum number of MEMBAR instructions.

1b)

Hint: A correct but not-optimized sequence of code is the following:

```
P1                                  P2
ADDI R2, R0, #1   ; R2=1            ADDI R2, R0, #1   ; R2=1
SD   R2, 0(R1)    ; A=R2            SD   R2, 8(R1)    ; B=R2
MEMBAR                              MEMBAR
LD   R3, 40(R1);  R3=F              LD   R3, 48(R1);  R3=G
SD   R3, 16(R1);  C=R3              SD   R3, 16(R1);  C=R3
MEMBAR                              MEMBAR
LD   R4,  8(R1);  R4=B              LD   R4,  0(R1);  R4=A
SD   R4, 24(R1);  D=R4              SD   R4, 32(R1);  E=R4
```

Note that Loads to one value and Stores of the same value do not need to be separated by MEMBARs because the system obeys memory dependences. Try to determine if you can remove one or more MEMBAR instructions

without violating the sequential memory consistency.

1c)

Now consider a machine that uses finer-grain memory barrier instructions. The following instructions are available:

- $\texttt{MEMBAR}_{rr}$ guarantees that all read operations initiated before the $\texttt{MEMBAR}_{rr}$ will be seen before any read operation initiated after it.
- $\texttt{MEMBAR}_{rw}$ guarantees that all read operations initiated before the $\texttt{MEMBAR}_{rw}$ will be seen before any write operation initiated after it.
- $\texttt{MEMBAR}_{wr}$ guarantees that all write operations initiated before the $\texttt{MEMBAR}_{wr}$ will be seen before any read operation initiated after it.
- $\texttt{MEMBAR}_{ww}$ guarantees that all write operations initiated before the $\texttt{MEMBAR}_{ww}$ will be seen before any write operation initiated after it.

There is no generalized MEMBAR instruction as in Part B of this problem.

In total store ordering (TSO), a read may complete before a write that is earlier in program order if the read and write are to different addresses and there are no data dependencies. For a machine using TSO, insert the minimum number of memory barrier instructions into the code sequences for P1 and P2 so that sequential consistency is preserved.

4) Performance estimation for Sequential Consistent Multiprocessors (15 points)

The following code is executed on an aggressive dynamically single-issue scheduled processor. The processor can have multiple outstanding operations, the cache allows for multiple outstanding misses, and the write buffer can hide store latencies. These features are only used if allowed by the memory consistency model.

P1

```
sendSpecial(int value) {
  A = 1;
  LOCK(L);
  C = D*3;
  E = F*10;
  G = value;
  READY = 1;
  UNLOCK(L);
}
```

P2

```
receiveSpecial() {
  LOCK(L);
  if (READY)  {
       D = C+1;
       F = E*G;
  }
 UNLOCK(L);
}
```

Assume that the locks are non-cacheable and are acquired either 50 cycles from an issued request or 20 cycles from the time a release completes, whichever is later (in our code the release operation is UNLOCK(L)). A release takes 50 cycles to complete. Read hits take 1 cycle to complete and writes take 1 cycle to put into the write buffer. Read misses to shared variables take 50 cycles to complete. Writes misses take 50 cycles to complete. The write buffer on the processors is sufficiently large that it never fills completely. Only count the latencies of reads and writes to shared variables (those listed in capitals) and the locks. All shared variables are initially uncached with a value of 0. Assume that the processor P1 obtains the lock first.

a) Under SC, how many cycles will it take from the time P1 enters the sendSpecial() routine to the time that P2 leaves receiveSpecial()? Make sure that you justify your answer. Also make sure to note the issue and completion time of each sunchronization event. Remember that the SC model requires that each instruction completes before the next starts, i.e. it does not allow for any instruction reordering within a processor.

b) How many cycles will it take to return from receiveSpecial() under the release consistency model? A Release Consistency (RC) model differentiates between regular instructions (such as C = D*3) and synchronization instructions (acquire or LOCK, release or UNLOCK). The RC model allows the reordering of Reads and Writes to different locations within a processor for the regular instructions. However, it maintains sequential consistency among *sync* operations. For the RC model the constraints are as follows:

   LOCK → all; all → UNLOCK;

   in which the notation A→B means that if operation type A precedes operation type B in program order, then program order is enforced between the two operations. The notation "all" means all instructions (sync+regular).

You will need to study the paper by Adve et. al to solve this problem.