University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Fall 1998 J. D. Kubiatowicz

Quiz 2
November 20, 1998
CS252 Graduate Computer Architecture

You are allowed to use a calculator and one 8.5" x 11" double-sided page of notes. Show your work on all problems. If you find a problem unclear or underspecified, please ask for clarification of the assumptions. If you make assumptions not listed in the problem, please state them clearly in your solutions. Good luck!

Your name:	**SOLUTION SET**
SID number:	
Email address:	

1	35/30
2	35/35
3	35/35
Total	**105/100**

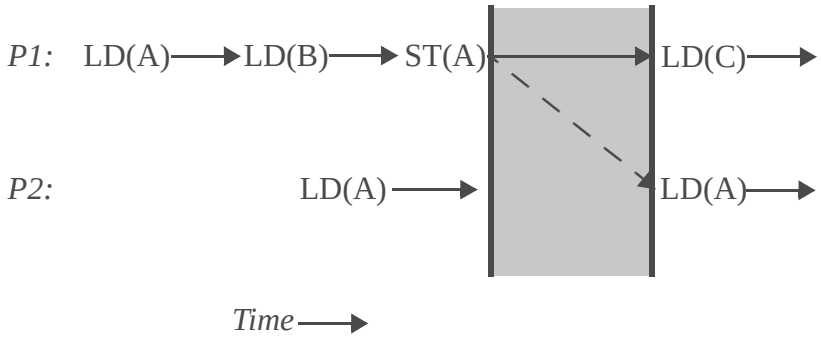
Question 1: Being Consistent

a) List the three conditions sufficient for sequential consistency that were presented in class. *Hint: one of these dictates the way in which processors can make requests; the other two deal with write atomicity)*

The following are the three conditions (taken from Culler & Singh, *Parallel Computer Architecture, a Hardware/Software Approach*, MKP, 1998, p289):

- Every processor issues memory operations in program order
- After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation
- After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation. That is, if the write whose value is being returned has been performed with respect to this processor, then the processor should wait until the write has been performed with respect to all processors

b) Draw a diagram illustrating the conditions of write atomicity for two processors (*hint: this diagram appeared several times in class*)



In the above diagram, arrows indicate the time during which an operation is in progress (the tail is the start of the operation and the head is the end of the operation).

Question 1 (continued)

c) Describe how the two conditions for write atomicity are satisfied in the following three cases:

1) a snoopy-cache protocol on an atomic bus

The bus is atomic, so only one memory transaction can be outstanding at any time. The bus is shared, so it serializes accesses of different processors. Thus it is impossible for a read (or any memory access, for that matter) on processor *j* to start until any prior write by processor *i* has completed and released the bus. Additionally, since the bus is atomic, the coherence transaction needed to read a cache line into the exclusive/modified state will not return the data until all processors have received and processed invalidate messages.

Finally, to meet the first condition of sequential consistency, we assume that the processor issues memory operations in-order, and waits for them to complete (*e.g.*, there is no write buffer).

2) a snoopy-cache protocol on a split-transaction bus

Each processor keeps track of all outstanding bus transactions (for example by tagging them and having a tag store/transaction buffer in the bus interface controller, as in the SGI Challenge). This is possible because the bus is a broadcast medium, so all processors see all transactions. Before issuing a write to address X, a processor checks for any outstanding transactions involving address X, and blocks until they have finished. Similarly, on a read to address X, if any outstanding transactions match, the processor piggybacks onto the existing transaction and does not issue its own bus request.

This hardware gives atomic bus behavior on a per-address basis, which gives sequential consistency as above, while still allowing interleaving of bus transactions across addresses.

3) a directory-based protocol in a distributed-memory multiprocessor

All memory accesses to a given address X are sent to the directory at the home node for X, which serves as a single coordination point for memory traffic to that address. If a write is in progress to X (*e.g.*, waiting for ACKs to come back from invalidates), the directory controller knows this and can NACK any read or write requests that arrive for X while the original write is still outstanding. Typically, if a read for address X is outstanding and another request comes in, it is NACK'd as well, although in this case the request could be queued for later processing.

Since the directory serializes accesses with respect to writes on a per-address basis, sequential consistency is achieved.

Question 1 (continued)

d) Describe how your answer to subpart 3 of part c (the directory-based MP) can lead to livelock in large systems.

The livelock results from the possibility that any read request made by a node for a certain address can be repeatedly NACK'd due to pending transactions for that address at the home node. This might arise in a large system where certain processors are farther (in terms of network delay) from a given directory node than others: a nearby processor might repeatedly and rapidly issue write or read requests for a given memory line to the directory, causing all requests from a distant processor for that same line to encounter contention and be NACK'd. Because the remote processor cannot make forward progress in a situation like this, the system is livelocked.

e) How might you modify the cache controller and/or the directory controller to fix this livelock problem? Give complete architectural details, if possible.

The typical hardware solution is to add priorities to requests. The first time a processor makes a request, it assigns it priority 0. Each time the request is NACK'd, the processor increases the priority before retrying the request. When the request succeeds, the processor resets is priority to 0 before starting a new request. The directory now requires a single-entry buffer that stores the current highest-priority request. If a request is received at the same or lower priority than the request in the buffer, the incoming request is NACK'd and ignored. If a request is received at a higher priority than the request in the buffer, the *buffered* request is discarded and NACK'd, and the new request is placed in the buffer. Note that it is safe to discard the buffered request because it has not been processed and has generated no messages; as long as a NACK is sent back to its originator, it be ignored.

The livelock is eliminated because any request that is repeatedly NACK'd will increase in priority until it higher priority than any pending request, at which point it will be serviced.

f) EXTRA CREDIT (5 points max): describe how to exploit the hardware features of a modern out-of-order processor *without* violating sequential consistency.

The basic idea here is to issue memory loads speculatively, and use the reorder buffer to both track the status of the loads and to clean up if they cause sequential consistency to be violated. As in a standard out-of-order processor, the reorder buffer keeps memory operations in order. Stores are sent to the memory system only when the reach the head of the reorder buffer. Loads are issued speculatively as soon as their addresses are available, and their data is bypassed to waiting computation instructions when it is available. To preserve sequential consistency, the reorder buffer must assume the role of a transaction buffer, and keep track of any invalidates received for the cache lines corresponding to loads in the buffer. When a load reaches the head of the reorder buffer, it is checked to see if it has received any invalidates for its address. If so, the computation is aborted, the reorder buffer state behind and including that load is discarded, and instruction issue begins again at the point of the misspeculated load. This preserves sequential consistency while still allowing out-of-order execution by catching violations of the protocol (reads moving before writes) and cleaning up using the existing reorder buffer mechanisms.

Question 2: Caches and Busses

For this problem, assume that we are dealing with a bus-based shared memory multiprocessor (SMP) using the MESI protocol.

Each processor in the system is identical, and the following information applies to each processor in the system individually. 50% of the instructions executed by each processor are loads or stores (instructions are 32 bits wide). Of these loads and stores, on average 70% are reads to private data, 20% are writes to private data, 8% are reads to shared data, and 2% are writes to shared data.

Each processor has a single-level split instruction/data cache. The instruction cache is 16KB, two-way associative, and has 16 byte lines. The data cache is 16KB, direct mapped, and also has 16 byte lines. The hit rates in the caches are as follows: 97% for private data, 95% for shared data, and 98.5% for instructions. Cache hit time is one cycle for both caches.

The SMP system bus has separate address and data busses, with 64 data lines and 32 address lines. The bus is *atomic* (not split-transaction), so the address and data busses can be treated as a single bus with respect to negotiation. The bus is clocked at one-half the speed of the processor. For reads, memory responds with data 12 bus cycles after being presented the address, and supplies one block of data per bus cycle after that. For writes, both address and data are presented to memory at the same time. Thus a single-word write consumes 1 bus cycle, while a 16-byte write consumes two cycles. Assume all requests are satisfied by the memory system, not by other caches. The processor CPI is 2.0 before considering memory penalties.

a) We want to place as many processors as possible on the bus. What is the bus utilization of a single processor if the caches are write-through with write-allocate strategy? How many of these processors can the bus support before it saturates?

For this part of the problem, assume that a write to the bus automatically invalidates any other existing copies of the data being written. Ignore bus contention and coherence messages received from other processors (*e.g.*, remote snoops), but *do* consider coherence traffic generated by the processor. Show your work.

The key to approaching this problem is to recognize that the true CPI (including memory penalties) includes three components: the 2.0 base CPI (CPI_{cache}), plus a component due to bus stalls on instruction cache misses (CPI_{busI}), plus a component due to bus stalls on data cache misses (CPI_{busD}). Once we have computed these bus CPI components, then we can compute the bus utilization as the average fraction of the per-instruction time that the bus is busy, or

$$\text{Bus utilization} = (CPI_{busI} + CPI_{busD})/(CPI_{cache} + CPI_{busI} + CPI_{busD}).$$

We will perform all calculations in *CPU cycles* in this solution.

In this part of the problem, CPI_{cache} is 2.0. It takes 26 CPU cycles to fetch a cache line from the memory system, 24 to get the first 8 bytes and 2 to get the remaining 8 bytes. Since the bus is atomic, it is busy during that entire time.

1.5% of instructions miss the cache, and each miss requires 26 cycles. So

$$CPI_{busI} = 0.015*26 = 0.39$$

Question 2 (continued)

Computing CPI_{busD} is tricker. First, there is an overall factor of 0.5, since only half of the instructions are loads or stores:

$CPI_{busD} = 0.5 * (...)$

Of these, 70% are reads to private data, of which 3% miss in the cache, each using 26 cycles of bus time to fetch a cache line:

$CPI_{busD} = 0.5 * [(0.7*0.03*26) + ...]$

20% of the memory operations are writes to private data. 97% of these hit in the cache, but because the cache is write- through, each hit still consumes 2 CPU cycles worth of bus activity to write the word through to memory. 3% of the private writes miss, causing 26 cycles of bus time to fetch the cache line accessed (since the cache is write-allocate) plus two more cycles to write-through the new value:

$CPI_{busD} = 0.5 * [(0.7*0.03*26) + 0.2*(0.97*2 + 0.03*(26+2)) + ...]$

8% of the memory operations are reads to shared data; these are identical to reads to private data, except that the miss rate is now 0.05:

$CPI_{busD} = 0.5 * [(0.7*0.03*26) + 0.2*(0.97*2 + 0.03*(26+2)) +$
$(0.08*0.05*26) + ...]$

Finally, 2% of accesses are writes to shared data, which again behave identically to writes to private data, since the cache is write-through and the bus write serves as an invalidation to other processors:

$CPI_{busD} = 0.5 * [(0.7*0.03*26) + 0.2*(0.97*2 + 0.03*(26+2)) +$
$(0.08*0.05*26) + 0.02*(0.95*2 + 0.05*28)]$

$CPI_{busD} = 0.636$

So, putting it all together,

$BusUtilization = (0.39 + 0.636)/(2 + 0.39 + 0.636) = 0.34$

So the bus is 34% saturated. This means it can support at most floor(1/0.34) = 2 processors.

Question 2 (continued)

b) How many processors can the bus support without saturating if the caches are write-back (and write-allocate)? Assume that the probability of having to replace a dirty block in the cache on a miss that fetches a new block is 0.3. Also assume a MESI protocol, and again ignore bus contention and coherence messages received from other processors (*e.g.*, remote snoops), but *do* consider coherence traffic generated by the processor. Assume that the cache protocol supports upgrades, so a write hit to a shared block causes an invalidate transaction *only*, taking one bus cycle.

We take the same approach as in the last problem. We make two assumptions: (1) writeback is not overlapped with reading the new data, and (2) all write hits to shared data require an upgrade transaction. As in the previous part, $CPI_{cache} = 2.0$, and all computation is in CPU cycles.

Since the I&D caches are split, and instructions are never modified, instruction misses cannot cause writebacks. Thus, CPI_{busI} is the same as in the previous part:

$$CPI_{busI} = 0.015*26 = 0.39$$

To simplify the calculation of CPI_{busD}, define MissTime to be the average time spent holding the bus when service a cache miss. MissTime includes the 26 cycles needed to read in a new cache line, as before, but it also includes a term to account for the potential of writeback. We are told that on average 30% of cache misses cause a writeback, and a writeback takes four CPU cycles on the bus (two for the address and first 8 bytes of the line, and two for the remaining 8 bytes), so

$$MissTime = 26 + 0.3*4 = 27.2$$

Now, let's compute CPU_{busD}. First consider cache misses. In all cases, a cache miss behaves the same (read and write misses both need to first read a line of data, and this read encapsulates all coherence actions), taking MissTime bus cycles, so for cache misses we get:

$$CPI_{busD} = 0.5 * [(0.7*0.03*MissTime) + (0.2*0.03*MissTime) + (0.08*0.05*MissTime) + (0.02*0.05*MissTime) + ...]$$

We haven't considered cache hits yet. All read hits are absorbed entirely by the cache. Because the cache is writeback, write hits to private data behave the same as read hits. Only write hits to shared data propagate to the bus, and they only require a single bus transaction (2 CPU cycles) to send out an upgrade transaction. Thus we can complete CPI_{busD} as:

$$CPI_{busD} = 0.5 * [(0.7*0.03*MissTime) + (0.2*0.03*MissTime) + (0.08*0.05*MissTime) + (0.02*0.05*MissTime) + (0.02*0.95*2)]$$

$$CPI_{busD} = 0.45420$$

We use the same construct as before to get bus utilization:

$$BusUtilization = (0.39 + 0.45420)/(2 + 0.39 + 0.45420) = 0.297$$

So we can support 3 processors in this configuration without saturating.

Question 2 (continued)

c) Assume we add a unified write-back, write-allocate second-level cache to each processor (assuming a write-through, *no-write-allocate* first level cache with the same parameters as before). The L2 line size is 32 bytes. The local miss rate for *all traffic* to the L2 cache is 10%. The hit time to the L2 cache (which includes the time to transfer data to the L1 cache, but not the L1 hit time) is 6 cycles. The L2 cache is clocked at the same speed as the processor. Inclusion is maintained between the caches. Again, assume that the probability of having to replace a dirty block in the L2 cache on a miss that fetches a new block is 0.3.

What is the bus utilization of a single processor now? How many of these processors can the bus support without saturating?

This case differs from the previous two because we have a second level of cache, so CPI_{cache} is no longer 2.0 (it must take into account L1 misses/L2 hits). We'll get back to this later. Let's start by computing CPI_{busI} and CPI_{busD} as before. With the exception of instruction misses and write hits to shared data, this case with the L2 cache behaves exactly the same as the prior example of a writeback L1 if we redefine MissTime to include the 10% probability of missing in the L2:

MissTime = 0.1*(30 + 0.3*8) = 3.24

Notice that it now takes 30 cycles to fetch a line, since the L2 cache line is twice as long as the L1 cache line; a writeback similarly takes 8 cycles.

CPI_{busI} must now take into account a possible writeback in the L2, since the L2 cache is unified:

CPI_{busI} = 0.015*MissTime = 0.0486

For CPI_{busD} we begin with the cache miss portion, which is the same as before using the revised MissTime:

CPI_{busD} = 0.5 * [(0.7*0.03*MissTime) + (0.2*0.03*MissTime) +
(0.08*0.05*MissTime) + (0.02*0.05*MissTime) + ...]

Again, all read hits and write hits to private data are absorbed by the L2 cache. Write hits in the L2 cache to shared data are a bit tricky. We know that 95% of them hit data already in the L1 cache, requiring 2 CPU cycles worth of bus activity for the upgrade, as before. However, there is the possibility that the write may miss in the L1 but hit in the L2; this case also requires 2 cycles for an upgrade transaction. Thus we add in the term .02*(0.95+0.05*0.9)*2, to get:

CPI_{busD} = 0.5 * [(0.7*0.03*MissTime) + (0.2*0.03*MissTime) +
(0.08*0.05*MissTime) + (0.02*0.05*MissTime) + 0.02*(0.95 + 0.05*0.9)*2]

CPI_{busD} =0.07174

Question 2 (continued)

To compute bus utilization, we need to get CPI_{cache}. It's actually easier to calculate the overall CPI (CPI_{cache} + CPI_{busI} + CPI_{busD}) than to get the CPI_{cache} component. The overall CPI is the base CPI, 2.0, plus the time spent in the memory hierarchy. Notice that if we merely redefine MissTime to include the time spent checking the L2 on an L1 cache miss (6 cycles), the above formulae for CPI_{busI} and CPI_{busD} give us CPU cycles spent servicing memory operations. So:

MissTime = 6 + 0.1*(30 + 0.3*8) = 9.24

Now, plugging back in to the previous formula:

CPI_{memory} = 0.015*MissTime +
0.5 * [(0.7*0.03*MissTime) + (0.2*0.03*MissTime) +
(0.08*0.05*MissTime) + (0.02*0.05*MissTime) + 0.02*(0.95 + 0.05*0.9)*2]

CPI_{memory} = 0.1386 + 0.16774

Adding back in the 2.0 base CPI gives us an overall CPI of

CPI = 2.0 + CPI_{memory} = 2.30634

Bus utilization is now just (CPI_{busI} + CPI_{busD})/CPI, or

BusUtilization = (0.0486 + 0.07174)/2.30634 = 0.052

So the bus is 5.2% utilized, allowing up to 19 processors before saturation.

Question 2 (continued)

d) Finally, compute the average memory access time (AMAT) in *CPU cycles* for the processor described in part (c). Be sure to include both instruction and data references. First show the equation, then the numerical results.

For a two-level cache of the type in this problem, the general formula for AMAT is

$$AMAT = HitTime_{L1} + MissRate_{L1} * (HitTime_{L2} + MissRate_{L2} * MissPenalty_{L2})$$

To simplify the calculation, we will calculate the instruction AMAT ($AMAT_i$) and data AMAT ($AMAT_d$) separately. The total AMAT is simply the weighted average taking into account the fact that only half the instructions are data accesses:

$$AMAT = (AMAT_i + 0.5*AMAT_d)/1.5$$

We'll use the same concept of MissTime as in the last part of the previous problem. MissTime here is the number of CPU cycles beyond the base CPI spent on average for a load or store instruction:

$$MissTime = HitTime_{L2} + MissRate_{L2} * (MissPenalty_{L2})$$
$$= 6 + 0.1*(30 + 0.3*8)$$

Then the general AMAT formula becomes

$$AMAT = HitTime_{L1} + MissRate_{L1} * MissTime$$

Now for $AMAT_i$, $HitTime_{L1}$ is 1, $MissRate_{L1}$ is 0.015, so

$$AMAT_i = 1 + 0.015 * MissTime$$

We can split up $AMAT_d$ into four components corresponding to the different types of memory access:

$$AMAT_{d\text{-}read\text{-}private} = 1 + 0.03*MissTime$$
$$AMAT_{d\text{-}read\text{-}shared} = 1 + 0.05*MissTime$$
$$AMAT_{d\text{-}write\text{-}private} = 1 + 0.03*MissTime$$
$$AMAT_{d\text{-}write\text{-}shared} = 1 + 0.05*MissTime + (0.95 + 0.05 * 0.9)(2))$$

Then

$$AMAT_d = 0.7 * AMAT_{d\text{-}read\text{-}private} + 0.08 * AMAT_{d\text{-}read\text{-}shared} + 0.2 * AMAT_{d\text{-}write\text{-}private} + 0.02 * AMAT_{d\text{-}write\text{-}shared}$$

Plugging in all the numbers gives

$AMAT_i = 1.1386$

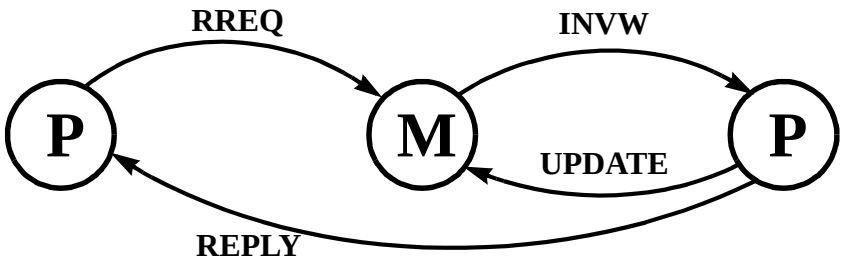
$AMAT_d = 1.3355$

and

$AMAT = 1.204$

Question 3: Deadlock

This question examines some of the issues of deadlock that can arise in directory-based distributed memory multiprocessors. For this question, we'll be primarily concerned with the following coherence transaction taken from a DASH-like protocol:

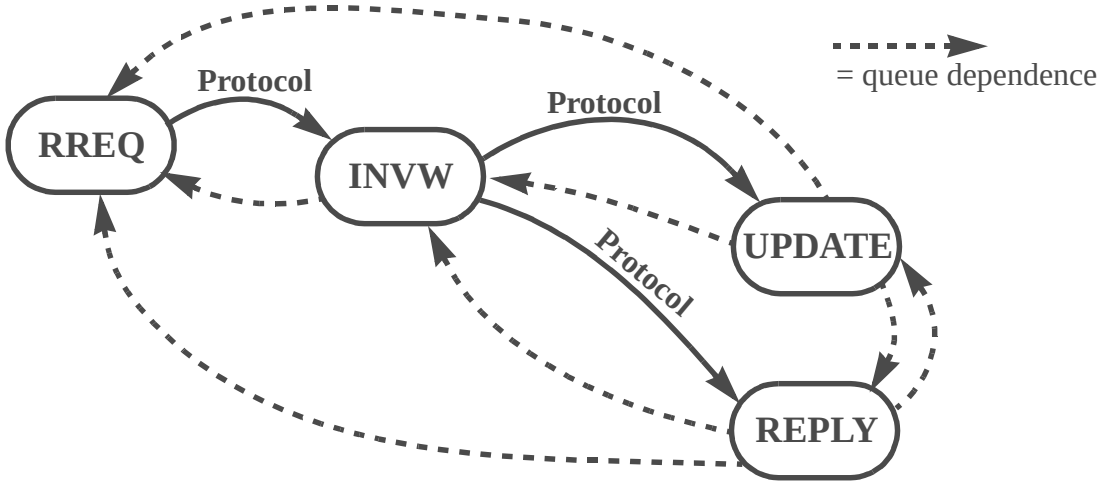


The **P** nodes correspond to processor nodes, and the **M** node corresponds to the home (memory) node for the data in question. **RREQ** is a *read request* message; **INVW** is a *invalidate-writer* message; **UPDATE** is a data reply that updates memory at the home node, and **REPLY** is a data reply to the original requestor.

a) Describe a situation in which the above transaction might occur (that is, what action is the left processor taking, and what is the original state of the data in the directory and the rightmost processor).

This might occur when the rightmost processor's cache holds dirty data for a given memory line (that is, the line is in the modified state in the rightmost processor), and the leftmost processor attempts to read that line.

b) Draw a message dependency diagram (where nodes are messages and arcs correspond to protocol dependencies or queue dependencies). Assume that only one virtual network exists.

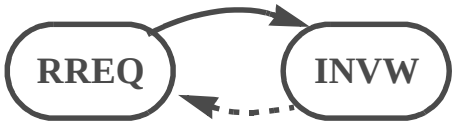


Question 3 (continued)

c) This protocol can deadlock. Explain why.

The protocol can deadlock because there are cycles in the graph that consist of at least one protocol arc. Note that only protocol arcs can block messages at their destination, so any cycle must have at least one protocol arc.

To understand this deadlock, consider a simpler example:

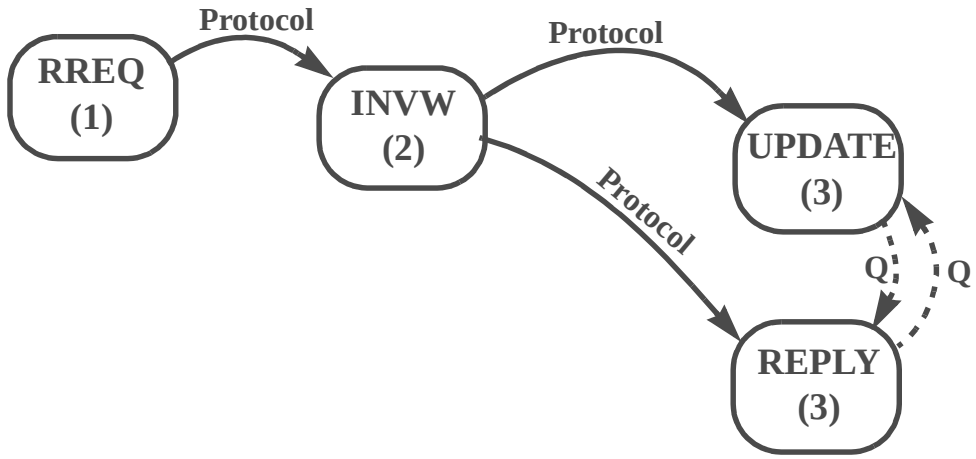


In this case, two nodes can deadlock each other if each has a RREQ at the head of its input queue, and full output queues.

d) One way of removing the deadlocks is by adding a sufficient number of virtual networks. How many virtual networks do you need to remove the possibility for deadlock in this transaction? Why does this remove the deadlock?

Three virtual networks are needed. In this case, there are no remaining cycles that have at least one protocol arc.

e) Given the number of virtual networks above, draw a new version of the message dependence diagram of part (b), labelling the nodes by the virtual network ID that the corresponding messages are sent on.



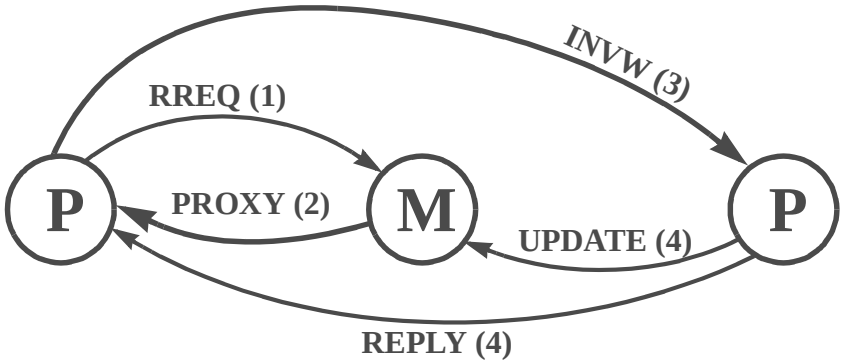
Question 3 (continued)

f) Assume that, for reasons of cost, you can only have two virtual networks. By introducing a new message type, *proxy*, that can include any information that you wish, describe how to transform this protocol into a purely request-response protocol over the two networks.

Send back a proxy message from the home node instead of sending out the INVW. Let the requesting processor send the INVW instead. The proxy message should include an encapsulated form of the INVW message.

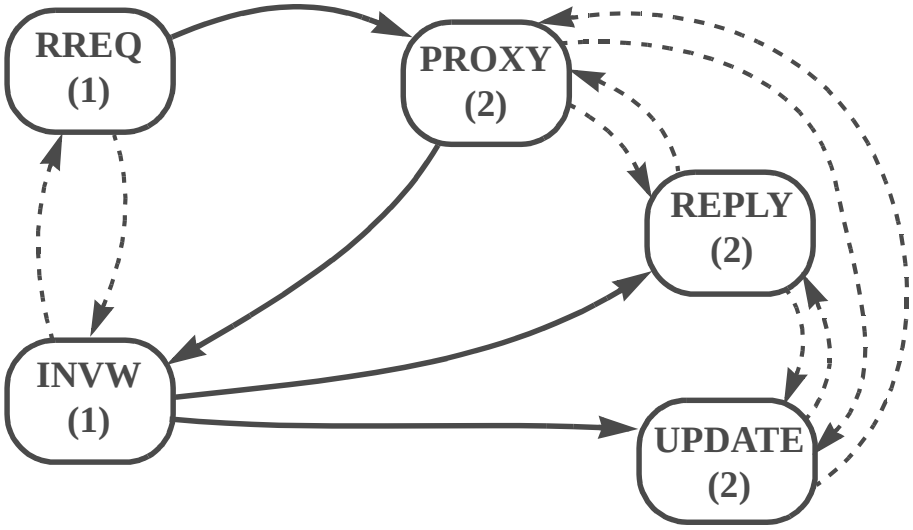
Note that this one proxy message is all that's needed, to reduce the depth 3 protocol to depth 2.

g) Draw a new diagram like the *printed* diagram at the beginning of this problem that shows the new use of the proxy message. Label the arcs in the graph with the time-order in which they occur.



Question 3 (continued)

h) Draw a message dependency diagram (as in part b) for these transactions. Be sure to include any queue dependencies. *Hint: this diagram will still include cycles.*



Note: protocol arcs are shown with solid arrows; queue arcs are shown with dotted arrows.

i) Given the fact that the number of outstanding requests from a process is limited, explain how a finite-sized transaction buffer can remove the cycles and the potential for deadlock, yielding a deadlock-free protocol on two networks.

Since the number of outstanding requests is limited, one can reserve sufficient buffer space to hold the proxy in the requesting node. This allows one to guarantee that the proxy can be taken out of the input queue, breaking the PROXY->INVW protocol arc. Note that when the proxy is buffered, there needs to be hardware that watches for sufficient output queue space to send the INVW.