Problems and Solutions for the Examination of EDA284 (2018-10-29)

Problem 1 (12p)

The goal of this exercise is to reason about three locking schemes designed to protect a shared data structures in three different scenarios. The underlying parallel computer is a sequentially consistent shared memory bus-based multiprocessor that uses the MSI-invalidate protocol to manage coherence across private caches.

The three scenarios under consideration are:

- Scenario #1: the shared data structure is accessed by a single thread.
- Scenario #2: the lock is accessed by multiple threads but lock contention is very low.
- Scenario #3: the lock is accessed by many threads, and lock contention is high.

The following codes show the implementation of the three locking schemes. In the following assume that R0=0, and that the lock is taken when its value is 1. The variable holding the lock is `_lock`.

Action/Label	**Lock #1**	**Lock #2**	**Lock #3**

`Lock:`	`T&S R1, _lock`
	`BNEZ R1, Lock`
		`LW R1, _lock`
		`BNEZ R1, Lock`
		`T&S R1, _lock`
		`BNEZ R1, Lock`
			`ADDI R1,R0,1`
			`LL R2,_lock`
			`SC R1,_lock`
			`BEQZ R1, Lock`
			`BNEZ R2, Lock`

`Unlock:`	`SW R0, _lock`	`SW R0, _lock`	`SW R0, _lock`

That task is to describe, for each of the three locking schemes:
(a) How well do the locking schemes apply to the three scenarios shown above?
(b) What synchronization hardware needs to be added to the basic bus-based multiprocessor (as described in Lecture 4) to support the locking schemes?
(c) Assume now that the memory consistency model is a synchronization-based memory consistency model such as weak ordering. Will the locking schemes still work correctly?

Solution to Problem 1:

The three locks correspond to a test-and-set based spinlock, a test-and-test-and-set spinlock, and a LL-SC spinlock that implements the same functionality as the test-and-set spinlock (the first four lines emulate a T&S and the fifth line checks if the lock was successfully taken).

(a) Performance and Efficiency

Scenario 1:
Since the data is only accessed by a single thread it is not necessary to actually implement mutual exclusion in the access. No lock should be used for this case. However, even if a lock is used, the overhead will be low, since the lock will always reside in the local cache and it will always be taken in the first try.

Scenario 2:

In scenario 2 multiple threads access the data structure. As long as contention is low Locks 1 and 3 are a slightly better solution as it takes only one atomic read-write to take the lock, while Lock 2 requires at least 1 load and 1 atomic read-write operation.

Scenario 3:
Given that the lock is highly contented, we assume that most threads do not immediately get access to the lock. Given that the coherence protocol is MSI, using Lock 1 will result in the lock migrating across all spinning threads. Each T&S generates an invalidation and a migration. Lock 2 is a better solution in this case as it exploits the 'S' state that allows multiple threads to share the value. All threads keep the lock locally in the 'S' state, and only when the lock is released by the holding thread is a invalidation generated. Hence there is only one invalidation each time a lock is release, instead of continuously by all the spinning threads (Lock 1). Since Lock #3 is an emulation of Lock #1, the behavior is similar to Lock #1.

(b) Hardware support

Locks 1 and 2 require HW necessary to implement atomic read-modify-write instructions:

- invalidation-based coherence protocol
- cache controller acquires cache line in ‘M’ state and performs T&S atomically

Lock 3 uses LL-SC which is typically found in load/store architectures (e.g. RISC pipelines). LL-SC support requires:

- LL-bit, set when LL is executed
- method to detect writes to cache line with lock (resets LL-bit)
- SC fails if LL-bit has been reset

(c) Memory Consistency

In weak ordering, accesses to synchronization data are treated differently by the hardware
from accesses to other shared and private data. Such operations act as memory barriers on all accesses, hence the behavior for programs using locks is equivalent to sequential consistency. Hence all three lock implementations are still correct under weak ordering.

Problem 2 (6p)

Amdahl's law and Gustafson's law are two speed-up models used to assess the expected impact of an architectural improvement, and to guide the efforts to improve the performance of parallel computer system.

(a) Describe Amdahl's law and provide the mathematical formulation to determine the speed-up as a function of the parallelizable program fraction and the number of processors used to run the program.

(b) Describe Gustafson's law conceptually and discuss what motivated its development

Solution to Problem 2

(a) Amdahl's law considers a problem of fixed size (or work) and states that the upper bound to performance is determined by the fraction of the problem (F) that can be parallelized. The formula for the speed-up for P processors is:

$$S_p = \frac{T_1}{T_p} = \frac{1}{1 - F + F/P} = \frac{P}{F + P(1-F)} < \frac{1}{1-F}$$

(b) Gustafson's law provides a speed-up formula considering a problem if fixed time (as opposed to Amdahl's law which considers a problem of fixed size). Amdahl's law suggests that adding resources to solve a fixed problem in less time quickly leads to diminishing returns. Gustafson's law takes the opposing view that, as faster and larger computers are developed, scientists/engineers use them to solve larger and more complex problems. Hence Gustafson's law fixes the time it takes to obtain a result, instead of the problem size.

Problem 3 (6p)

In this problem you are asked to describe basic metrics of interconnection networks and to analyze the performance of some basic network topologies.

(a) Describe in a few sentences the following latency and bandwidth measures used for interconnection networks: (a) routing distance, (b) network diameter, (c) bandwidth per node and (d) bisection bandwidth.

(b) Consider the following network topologies: a bus, a NxN crossbar switch, a bidirectional ring with *N* nodes, and a n-by-n 2D mesh such that *N*=*n***n*. The task is to determine the network diameter and bisection width of these four network topologies, as a function of *N* (or *n*).

Solutions to Problem 3

8.(a)

Routing distance: Nb of links traversed by a packet
Network diameter: Longest routing distance over all pairs of nodes
Bandwidth per node: Aggregate bandwidth across all links divided by the number of nodes
Bisection bandwidth: Bandwidth when all nodes in one network bisection communicate with nodes in the other bisection

8.(b)

Bus:
Network Diameter = 1
Bisection Width = 1 (or less, due to contention and arbitration)

Crossbar:
Network Diameter = 1
Bisection width = N

Bidirectional Ring:
Network diameter = N/ 2
Bisection width = 2

n-by-n Mesh:
Network diameter = 2 * (n-1)
Bisection width = n

Problem 4 (6p)

To effectively use multicores, simplifying the development of parallel software while providing high performance is an important goal. A recent proposal to achieve these goals is Hardware-based Transactional Memory (TM). Your task is to compare HW Transactional Memory with traditional single-global locks and fine-grained lock-based programming in terms of:

(a) Programming complexity
(b) Performance
(c) Hardware support

Write no more than 1-3 sentences for each item

Solution to problem 4

Answers are directly in the slides:
(a) Transactional memory (TM) simplifies shared memory programming by allowing to execute sets of writes (ST) and reads (LD) in an atomic way. Just annotate with Transactin_begin and Transaction_end. Complexity Similar to coarse-grained

(b) Performance is similar to fine-grained locking when conflict probability is low. When conflicts are high, performance decreases and can be even worse than a single global lock if most transactions need to restart.

(c) HW-based TM requires special hardware to keep speculative values in the L1 caches and release the values atomically when a transaction commits. Support for locks is much simpler as it focuses on providing atomic read-modify-write for single cache lines, instead of read/write sets.

Problem 5 (10p)

A design team needs to choose an appropriate cache coherence protocol to be used for a bus-based shared memory multiprocessor with a number of processor/private cache units connected by a shared single-transaction bus. The team has narrowed down the design to two protocol choices: MSI-invalidate and MESI-invalidate. Given the higher complexity of MESI, the designers consider that at least 5% better performance and 5% lower traffic need to be achieved to justify the implementation of the MESI protocol. In order to select the protocol the team needs to analyze the following representative sequence of accesses happening on the bus in the following order:

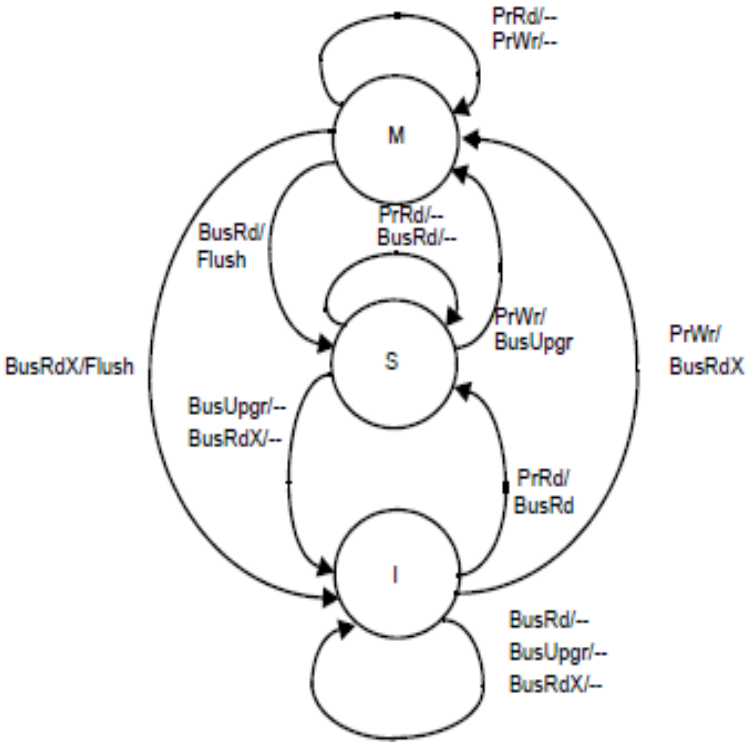
R1/X, R3/Z, R2/Y, W1/X, W2/Y, W3/Z, R1/Z, R4/Z, W2/Z, R3/Z

where the notation Ri/X, Wi/X means a read and write from processor i to cache block X. The latency and traffic costs of different operations considering a block size B bytes are:

Operation	**Latency**	**Bus Traffic**
Read hit	1	N/A
Write hit	1	N/A
Request serviced by next level (BusRd,Flush)	50	6+B
Bus upgrade (BusUpgr)	10	10
Snoop action	5	N/A

Assumptions:
1. A bus upgrade consists of transferring the request on the bus and making a snoop action in each cache (the latency of the latter is shown in the table)
2. There is no contention on the tag directory (duplicated tag directory)
3. X, Y and Z are not the same cache block
4. The block size B is 32 bytes
5. Read-exclusive requests (BusRdX) have the same latency and traffic as Read requests

The MSI state-transition diagram is shown below



To assess the performance of both protocols the team must construct a table with the latency (cycles) and bus traffic (bytes) needed to complete the aforementioned access sequence (shown below). Based on the aforementioned complexity-performance trade-off, which protocol will the design team ultimately choose?

Protocol	Latency	Traffic
MSI		
MESI		
Improvement		

Solution to Problem 5

MSI Protocol:

R1/X (miss/mem) → 5 cycles (snoop) + 50 cycles (BusRd) = 55 cycles / 6+B
R3/Z (miss/mem) → 5 cycles (snoop) + 50 cycles (BusRd) = 55 cycles / 6+B
R2/Y (miss/mem) → 5 cycles (snoop) + 50 cycles (BusRd) = 55 cycles / 6+B
W1/X (hit/upgrade) → 5 cycles (snoop) + 10 cycles (BusUpgr) = 15 cycles / 10
W2/Y (hit/upgrade) → 5 cycles (snoop) + 10 cycles (BusUpgr) = 15 cycles / 10
W3/Z (hit/upgrade) → 5 cycles (snoop) + 10 cycles (BusUpgr) = 15 cycles / 10
R1/Z (miss/cache) → 5 cycles (snoop) + 50 cycles (Flush) = 55 cycles / 6+B
R4/Z (miss/mem) → 5 cycles (snoop) + 50 cycles (BusRd) = 55 cycles / 6+B
W2/Z (miss/upgrade) → 5 cycles (snoop) + 50 cycles (BusRdX) = 55 cycles / 6+B
R3/Z (miss/cache) → 5 cycles (snoop) + 50 cycles (Flush) = 55 cycles / 6+B

Latency = 55 * 7 + 15*3 = 385 + 45 = 430 cycles
Traffic = (6+B)*7 + 10*3 = 72 + 7B
Given that B=32, the total traffic will be 296 bytes

MESI Protocol:

R1/X (I→E) → 5 cycles (snoop) + 50 cycles (BusRd) = 55 cycles / 6+B
R3/Z (I→E) → 5 cycles (snoop) + 50 cycles (BusRd) = 55 cycles / 6+B
R2/Y (I→E) → 5 cycles (snoop) + 50 cycles (BusRd) = 55 cycles / 6+B
W1/X (E→M) → 1 cycle / 0 bytes
W2/Y (E→M) → 1 cycle / 0 bytes
W3/Z (E→M) → 1 cycle / 0 bytes
R1/Z (I→S) → 5 cycles (snoop) + 50 cycles (Flush) = 55 cycles / 6+B
R4/Z (I→S) → 5 cycles (snoop) + 50 cycles (BusRd) = 55 cycles / 6+B
W2/Z (I→M) → 5 cycles (snoop) + 50 cycles (BusRdX) = 55 cycles / 6+B
R3/Z (I→S) → 5 cycles (snoop) + 50 cycles (Flush) = 55 cycles / 6+B

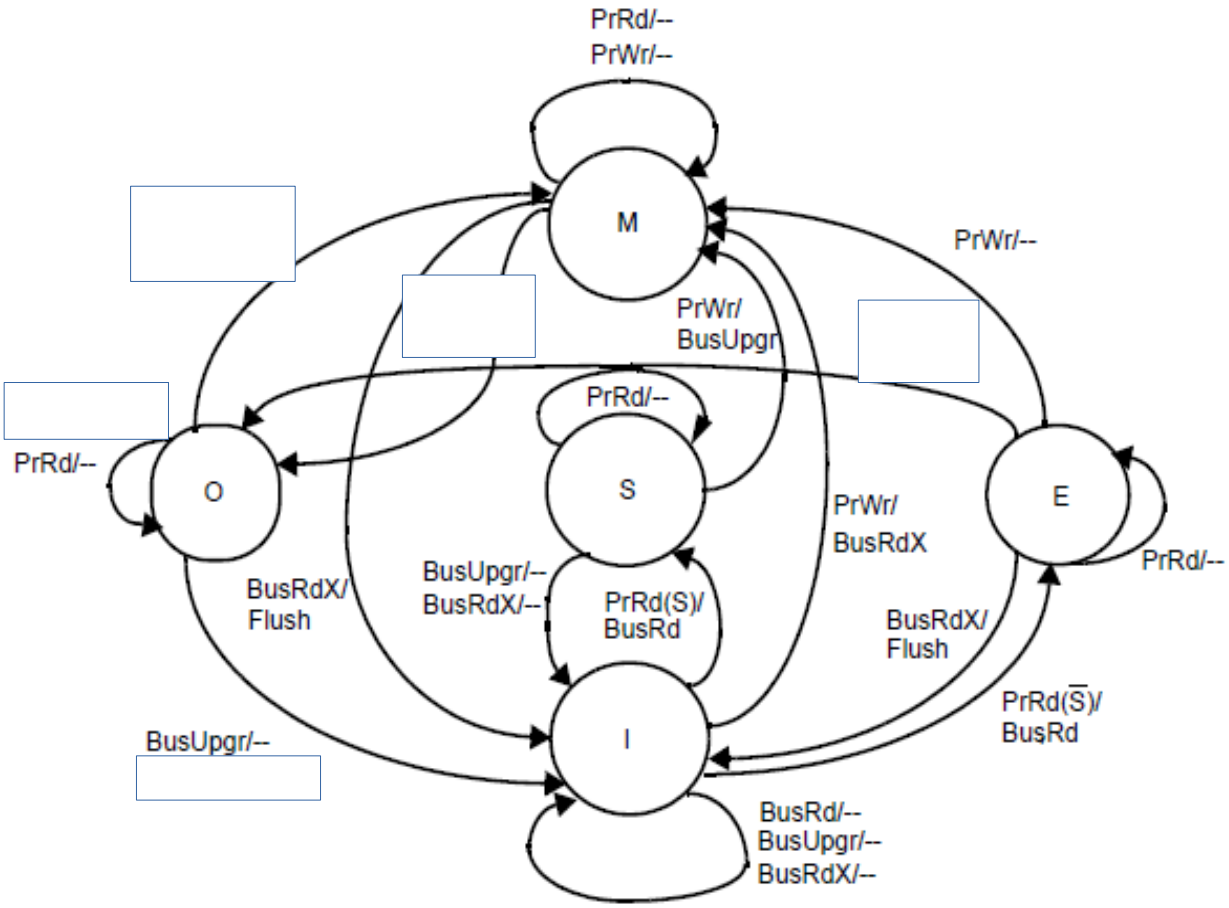
Latency = 55 * 7 + 3 = 388 cycles
Traffic = (6+B)*7 = 42 + 7B
Given the B=32, the total traffic will be 266 bytes.

Protocol	Latency	Traffic
MSI	430 cycles	296 bytes
MESI	388 cycles	266 bytes
Improvement	~10%	~10%

Given the complexity-performance criteria, the designers will ultimately choose the MESI protcol.

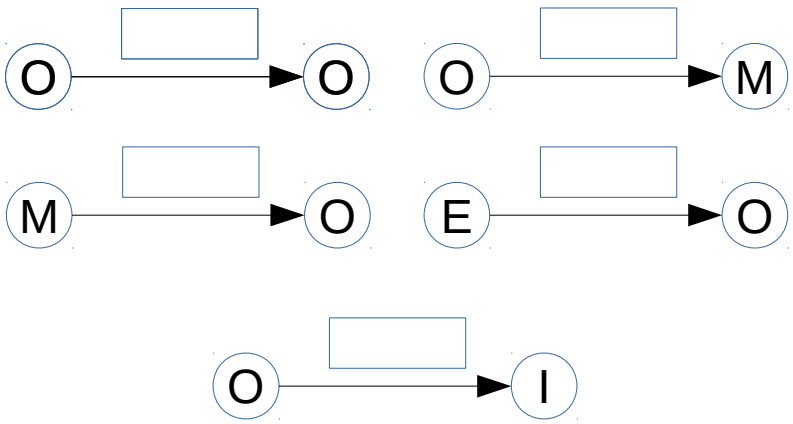
Problem 6

The MOESI protocol adds a new state called Owner (O). A designer has been given the task to implement the MOESI protocol by adding the necessary transitions on top of MESI. The designer has so far enumerated all the transitions and added them to the MESI diagram (as shown below), but not all coherence protocol actions have yet been determined.



Your task is to:

(a) Describe the meaning of the state (O)wner, and show one example in which it improves performance compared to MSI or MESI.
(b) Add the missing protocol actions to the five transitions shown below

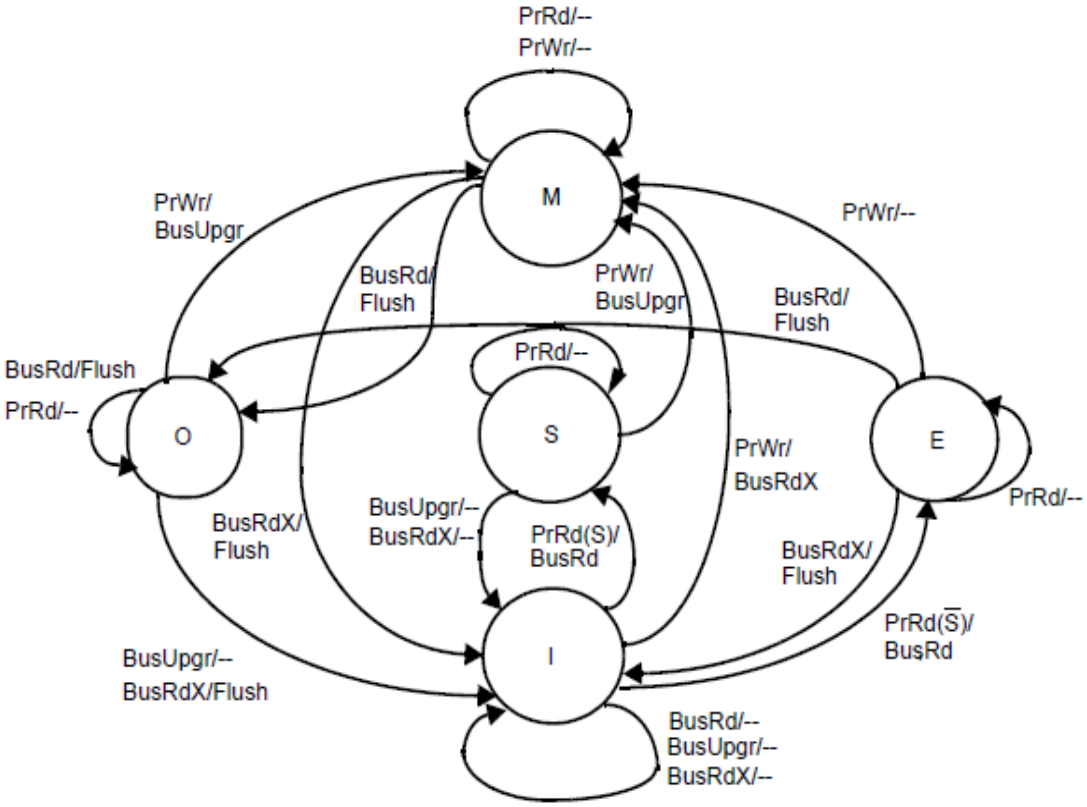


Solution to Problem 6

(O) adds the notion of ownership of a cache block. The states E and M are also (implicitly) owner. This state is used for ownership of blocks in the S (shared) state. Blocks in state O will supply a copy of the block when another cache requests it. Compared to MSI/MESI it:

1. enables sharing of dirty data and removes unnecessary writebacks
2. reduces memory bandwidth in CMPs by allowing on-chip caches to reply to BusRd requests.

The missing transitions are indicated in the following diagram for MOESI.



Problem 7 (8p)

Consider a future 8-way CMP on which you can power off some cores to allow the rest to operate at a higher frequency. You can use either 1, 2, 4, or 8 cores at the respective frequencies:

- when only one core is running it operates at 0.3ns clock cycle
- when two cores are running they operate at 0.4ns clock cycle
- when 4 cores are running they operate at 0.6ns clock cycle,
- when all 8 cores are running they operate at 1ns clock cycle,

Consider a partially parallel application consisting of a serial part of 100 Instructions and a parallel part that can be parallelized at will which is 200 Instructions. Parallelizing however requires you to create threads in the serial part of the application and 10 Instructions are added for every thread you create. The serial and parallel part of the application all run one instruction per cycle regardless of the frequency.

(a) Calculate the runtime of the application for the above processor when using 1, 2, 4 or 8 cores at their respective frequency without reconfiguring the processor while the application is running. Which configuration is better?

(b) What if you could reconfigure the processor to run the serial part with one core at 0.3 ns clock cycle and then configure it to 2 or 4 or 8 cores for the parallel part? What is the runtime for each case? Consider for all cases that creating each thread takes 10 Instructions that are added to the serial part (and run on the single core at 0.3ns clock cycle).

Solution to problem 7

(a)
1-Core: 100 Instr serial + 200 Instr parallel = 300 Instr at 0.3 ns = 90ns
2-Core: 100 Instr serial + (100 Instr parallel) + 20 Instr to create 2 threads = 220 Instr at 0.4 ns = 88ns
4-Core: 100 Instr serial + (50 Instr parallel) + 40 Instr to create 4 threads = 190 Instr at 0.6ns = 114 ns
8-Core: 100 Instr serial + (25 Instr parallel) + 80 Instr to create 8 threads = 205 Instr at 1ns = 205ns
→ 2 Core is the best configuration

(b)
Serial part is always 100 Instr at 0.3ns = 30ns
The parallel part :
2-Core: 20 Instr to create the threads at 03.ns + (100 Instr parallel at 0.4) = 6ns + 40ns = 46ns
4-Core: 40 Instr to create the threads at 03.ns + (50 Instr parallel at 0.6) = 12ns + 30ns = 42ns
8-Core: 80 Instr to create the threads at 03.ns + (25 Instr parallel at 1) = 24ns + 25ns = 49ns

So the total runtime for the reconfigurable processor is
2-Core: 30ns + 46ns = 71ns
4-Core: 30ns + 42ns = 67ns
8-Core: 30ns + 49ns = 79ns
→ 4-Core is the best

Problem 8 (6p)

We consider a scalable implementation of a shared memory multiprocessor using a set of nodes such that each contains a processor, a private cache, and a portion of the memory. Cache coherence is maintained using a directory cache protocol. The following costs apply:

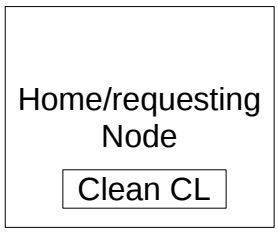
- The block size is 64 bytes.
- The time it takes to process a directory request is 20 cycles.
- The time it takes to install a new entry in a directory is also 20 cycles.
- A remote read request takes 10 Cycles and induces 8 bytes of traffic
- A flush of a cache line from one node to another costs 40 Cycles and 8 + 64 Bytes
- Ignore the time it takes to read/write to a cache.

Also:

- The home node is the one that holds the directory entry for a cache line.
- The requesting node is the node that produces a cache miss for a cache line.
- The remote node is the one that holds the dirty copy of a cache line when the cache line is dirty.

Your task is to describe the sequence of steps, as well as the number of cycles and traffic, needed to handle a cache miss for the following scenarios:

(a) The home node is the same as the requesting node and the memory copy is clean.

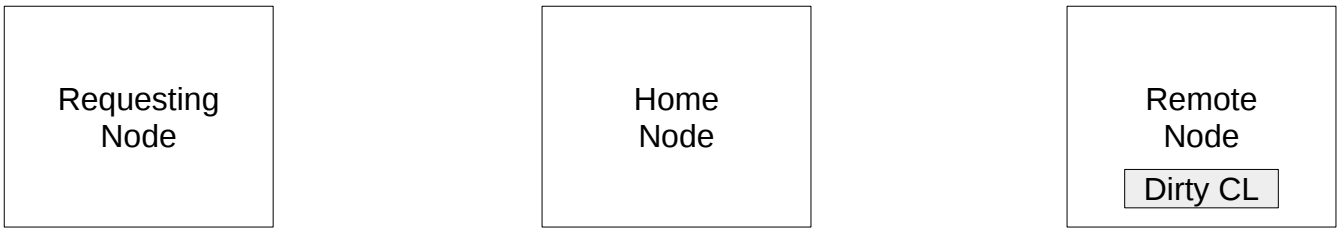


(b) The home node is the same as the requesting node and the memory copy is dirty

(c) The home node is different from the requesting node and the memory copy is clean

(d) The home node is different from the requesting node, the memory copy is dirty and the remote node is is the same as the home node

(e) The home node is different from the requesting node, the memory copy is dirty and the remote node is different from the home node



Solutions to Problem 8

(a)
1. Lookup home node dir (20 Cycles)

(b)
1. Lookup local dir (20 Cycles)
2. Remote read from home/requesting node to remote node (10 Cycles, 8 Bytes)
3. Directory lookup in remote node (20 Cycles)
4. Flush cacheline from remote node to home/requesting node (40 Cycles, 8 + 64 Bytes)
5. Install on home/requesting node directory (20 Cycles)

(c)
1. Remote read from requesting to home (10 Cycles, 8 Bytes)
2. Lookup home node directory (20 Cycles)
3. Flush cacheline from home node to requesting node (40 Cycles, 8 + 64 Bytes)
4. Install on requesting node directory (20 Cycles)

(d)
1. Remote read from requesting node to home/remote (10 Cycles, 8 Bytes)
2. Lookup home/remote node directory (20 Cycles)
3. Flush cacheline from home/remote node to requesting node (40 Cycles, 8 + 64 Bytes)
4. Install on requesting node directory (20 Cycles)

(e)
1. Remote read from requesting node to home node (10 Cycles, 8 Bytes)
2. Directory lookup in home node (20 Cycles)
3. Remote read from home node to remote node (10 Cycles, 8 Bytes)
4. Directory lookup in remote node (20 Cycles)
5. Flush cacheline from remote to home node (40 Cycles, 8 + 64 Bytes)
6. Install on home node directory (20 Cycles)
7. Flush cacheline from home to requesting node (40 Cycles, 8 + 64 Bytes)
8. Install on requesting node directory (20 Cycles)