QUESTION 1)

Bus:
2GB/sec * 1 Cash block/16B = 125M cash update/second

Processors
4 processors * 2.8G clock/sec * 1 instruction/2.2 cycles = 5.09G instructions/second

Assuming that ONLY stores will use the bus (i.e. ignoring instruction cash updates and data cash updates during reads)
Need 125M cash updates/sec (assumed to be all caused by Store instructions) / 5.09G instructions/sec *100% = 2.46%

QUESTION 2a)
There is no natural inclusion here. The problem lies in the different policies implemented for cache overwrite. Since the L1 is directly mapped and the L2 is set associative, it is possible that the L2 gets changed by set associatively, while the L1 remains the same.

Here is an example, using D = Data, S = set:
Read D1 => cache miss =>
    D1 is associated to S1, takes the first slot S1.1 in L2
    D1 is directly mapped to L1.1
Read D2 => cache miss =>
    D2 is associated to S1, takes the second slot S1.2 in L2
    D2 is directly mapped to L1.2
Read D3 => cache miss =>
    D3 is associated to S1, takes the third slot S1.3 in L2
    D3 is directly mapped to L1.3
Read D4 => cache miss =>
    D4 is associated to S1, takes the forth slot S1.4 in L2
    D4 is directly mapped to L1.4

Assuming that the slots are overwritten in a sequential manner
D2 is marked as invalid by another processor and the first processor needs to read it again
Read D2 => cache miss =>
    D2 is associated to S1, takes the first slot S1.1 in L2
    D2 is directly mapped to L1.2

We now have violation, because the D1 present in L1 at L1.1 is nowhere to be found in L2 (it was at S1.1, but S1.1 now contains D2)

QUESTION 2b)
Same as in a, since the cache overwrite policies differ between the 2 levels of caches, inclusion cannot be guaranteed. Using the exact same example as in a, we can show a violation. The fact that the cache lines are different from L1 to L2 also breaks the condition.

QUESTION 3)

Advantages:

- The processors cannot start thrashing the first level cash (L1 Cache deadlock). Each processors have their own L1 cache so no performance degradation due to sharing
- It is easier to build with regards to on chip real estate; sharing L1 caches puts some severe constraints to processor locations
- Sharing the L2 cache provides some great savings in terms of memory and cache coherence

Disadvantages:

- Need to worry about cache coherence between the separate cache entities; cache coherence protocols implemented might reduce performance

QUESITON 4a)

No instruction misses = ignore instruction caches

There will be a collision at the cache tag between the processor and a bus snoop every 3*5=15cc. Since the bus snoop has priority,
90% of the time, it will stall the processor for 1cc (Bus marks valid)
10% of the time, it will stall the processor for 2cc (Bus Marks invalid)
Performance loss due to cache tag contention:
[0.9*1cc + 0.1*2cc] / 15cc = 7.33%

QUESTION 4b)
Here, there is no contention due to both the cache controller and the processor reading the cache tag at the same time; the processor and the cache controller can both read their individual tags simultaneously.

Contention will only happen if the processor has a cache miss and needs to update the cache from memory:
Out of 3cc, the processor will miss 0.03 times which will require 2cc to change the cache tag from invalid to valid
0.03 * 2 / 3 = 2% contention

QUESTION 4c)
In the case if a collision: the processor wishes to read a cache value which is marked with 'VALID', at the same time, the bus wishes to convert 'VALID' to 'INVALID', because some other processor modified this value elsewhere already. If the processor gets priority, the data he reads is outdated. If the bus gets priority, it sets the cache line to invalid and forces the processor to go fetch the updated cache line.
Out of the 2 options, the second one, giving the bus snooping priority, seems to be a lot better.

QUESTION 5a)

The logic required to accommodate cache-to-cache transfer probably would have been too heavy to implement. When dealing with a large multiprocessors environment, cache lines would get interrupted all the time to get updated and this would either slow down the performance, or require a tremendous amount of logic to implement. Overall, we gain a simpler bus protocol at the cost of increased overhead due to sharing since invalid caches need to be updated from the memory.

QUESTION 5b)

We would require two extra signals:

- Cache-To-Cache (CTC): to signal a cache to cache transfer is taking place
- Don't Write Back (DWB): to notify the memory not to bother recording the CTC data

Here is a cache-to-cache transaction between 2 caches, say A and B.

1. Cache B has a modified cache line that has not yet been sent to memory, so labeled 'Modified'. Cache A requests this address from memory; cache B's controller snoops the request and realizes that it contains the data.
2. In order for cache B to stop cache A to get an invalid copy of the data, cache B's controller asserts the CTC line, which notifies the memory controller that it has the most recent version of the data and will provide it.
3. At the next clock pulse, cache B will dump the requested data on the bus. The memory controller, instead of putting the requested data on the bus, will actually read the data from the bus in order to get updated. Simultaneously, cache A will also read this data from the bus.
4. The CTC line is also connected to cache A's cache controller, such that when it is asserted and that cache A makes the read, it knows that it is reading from another cache and not memory, therefore, the cache line goes directly into 'shared' mode.
5. Finally, cache B, after having held the data on the bus for a clock pulse, de-asserts the CTC line and marks the cache line at hand as 'shared'

In the event that cache B had the data in 'Exclusive' mode, when it asserts CTC, it also asserts DWB. This signal notifies the memory controller to simply ignore the data that cache B will put on the bus because it already contains an updated value. Everything else remains the same.

QUESTION 5c)

It should work correctly because in either state Exclusive or Modified, the processor assumes that it is the only one which holds the most recent value of the data. If the controller snoops another processor requesting the same cache line on the bus, all we have to do is make a cache to cache transfer instead of a memory access (as described above) in order to update the second processor's cache line. Regardless of the original E or M state of the cache line, after the transaction, both caches will have the same up to date version of the data, and they will both be in shared mode.

I would assume they made this optimization to keep the design as simple as possible. If 2 states, even though not identical, represent the same assumptions made on the data, and that the final outcome is the same, what's the point of keeping track of both states individually…