Honours Computer Architecture 2020 Prac 3 – Hand in 5pm Thursday 10 September

You may confer on issues you do not understand but each answer must be your individual work. Note that you need to write enough so I can assess how well you understood – it you just write out an answer with no detail of how you got there, I can't assess understanding.

understanding of shared memory performance issues	15
clear and accurate calculation methods – show how you got there	15
correct answers	15
readable, well presented, minimal typos	5
Total	50

In these questions, where applicable, use the latencies in Table 5.1, p 89 of the notes. A multicore design shares data through the L3 cache; updates have to go to L3 before they are seen elsewhere. A MESI protocol is used.

1. A programmer has a loop initializing an array and splits it into two threads. The first thread executes:

```
for (int i = 0; i < N; i+=2)
   a[i] = init (i); // init even entries
```

   and the second executes:

```
for (int i = 1; i < N; i+=2)
   a[i] = init (i); // init odd entries
```

   a. Timing runs of the code shows that the 2-thread code runs twice as long as running the initialization without an extra thread. Explain how the memory access pattern could explain this.

      The two threads are updating even-numbered and odd-numbered array locations. If an int is 32 bits and a cache block is almost certainly bigger than this (e.g., 32 bytes – 8 times the size) then each time one thread updates a variable, there is good chance that it will either invalidate the block that the other thread needs to update or find the block invalidated by the other thread before it can update it. This is a classic example of *false sharing* – data smaller than block granularity that is not actually shared causing invalidation and write back traffic.

   b. Rewrite the two loops, changing the split of data between the two loops to fix the problem you identify in 1(a) above and explain why the memory access pattern will likely result in faster execution with your revised code.

      Some creative answers but any have to recognise that is ***false sharing***.

      Rewrite so that each thread updates half the data but not interleaved; the worst-case scenario is that the second thread updates entries close to the end of its half that share a cache block with the second thread when the latter initialises the first few entries but this is not likely if the threads start at about the same time as the second thread will have updated the entries that may share a cache block with the first thread at the start, whereas the first thread will only update any entries in a shared block at the end. You could go a step further and split the initialization at a cache block boundary – but you would need to know how big cache blocks are. In C, an array is represented as a pointer to its first element, so you could use this fact to work out how the array aligns with cache blocks.

```
for (int i = 0; i < N/2; i++)
   a[i] = init (i); // init first half of entries
```

and the second executes:

```
for (int i = N/2; i < N; i++)
   a[i] = init (i); // init last half entries
```

2. Explain how an M-lock can reduce memory delays compared with a spinlock. In your answer consider the following scenarios:

   The main thing to understand here is that although M-lock is more ***scalable*** and ***fair***, it has more overhead so for a very short critical section, it is not clear that it is a win.

   a. A very short critical section that only has one instruction that updates a shared data structure.

      In this scenario, the time spent in the critical section is extremely short so a spinlock will not incur too much extra time arising from shared-memory contention. However, the M-lock has more overhead and may therefore come out worse. If there are very many tasks waiting on the lock, since M-lock is more scalable, it could still be better than a spinlock in that scenario.

   b. A critical section in which numerous shared data structures are updated.

      Since the lock is held for longer, the scalability property of the M-lock will make it come out ahead even if the number of waiters is not very high.

3. Two cores attempt to modify the same location in memory at the same time, with a store word (`sw`) instruction. Outline the steps taken including bus transactions from the start of the first `sw` instruction to the completion of the second `sw` instruction. Assume at the start the core that wins the race to modify the block has it in the shared (S) state and it is *not* in the cache of the core that loses the race.

   Biggest issue here: not stating assumptions on what protocol is used and leaving out significant detail. MESI protocol is the one we study and this is an opportunity to understand it (noting that there are variations).

   Assume MESI protocol.

   Since the core that wins the race has it in S state, it issues a snoop on the bus. By wining the bus, it can immediately complete the store after changing state to E (exclusive). The `sw` instruction completes and state in the local cache changes to EM (exclusive and modified). Just by virtue of winning the bus, it can go ahead without delay to complete the write and change the state of its cache to E then M. The loop latency guarantees that there is more than enough time for this.

   The other core has to stall the store because it doesn't win the race to acquire the bus; seeing the snoop, it invalidates its copy so its own snoop is quashed since it no longer has the block in S state.

   As soon as the snoop completes, the losing core acquires the bus (it is important to understand that the bus serialises requests). It issues a write miss request.

   By the time the *first* snoop latency is complete, the winning core will have completed its `sw` instruction and changing the state of its copy of the block (part of the design is to ensure that the snoop latency is long enough for this to be true).

The write miss results in the winning core writing the block back to the highest level of the hierarchy that the other core sees (e.g., in a current Intel design, that would be L3, which is shared; L2 is not). Assume the shared level is L3. The write miss can be serviced either by waiting for L3 to update then an L3 to L2 and then L2 to L1 update, or by doing all of these in parallel to save time (current Intel architectures do this). Once the write miss is handled, the second core can now issue an invalidate and once the snoop latency has passed, mark the block exclusive and continue to execute the `sw` instruction, after which the block is marked M in addition to E.

There can be slight variations in the design – e.g. a write miss can be handled as two transactions, a read miss and an invalidate. However these two transactions need to be designed so that the previous winning core can't insert another invalidate between the read miss and the losing core's invalidate resulting in starvation.

Here is the approximate timeline (bus events roughly correspond to snoops):

clock	**bus event**	**core 0**	**core 1**
0		`sw` decoded	`sw` decoded
1	core 0 copy of block➛E; core 1 loses race to acquire bus for same purpose	`sw` execute	`sw` execute
2		sw mem (local copy of block E➛EM)	stall
3		sw wb	stall
… to end snoop latency		core 0 continues	stall
	core 1 write miss		stall
	write back starts	invalidates local copy of block EM➛I	stall
… to end write back and read miss latency	core 1 copy of block➛E; ensures no other core acquires block before `sw` completes	continues if it doesn't need this block	`sw` mem (local copy of block E➛EM)

Core 0 sees no further delays *unless* it also has a cache miss that gets held up behind core 1's miss on the shared block. Bus events are not visible to the pipeline unless they interfere with the cache state as seen by the CPU.