## Problem 5.11

| | Processor 1 | Processor 2 | Processor 3 | Miss type |
|---|---|---|---|---|
| 1 | $R_A$ | | | cold |
| 2 | | $R_B$ | | cold |
| 3 | | | $R_C$ | cold |
| 4 | $W_A$ | | | |

| | Processor 1 | Processor 2 | Processor 3 | Miss type |
|---|---|---|---|---|
| 5 | | | $R_C$ | false-sharing miss |
| 6 | | $R_A$ | | true-sharing miss |
| 7 | $W_B$ | | | |
| 8 | | | $R_A$ | true-sharing miss |
| 9 | | | $R_B$ | |

**a)** It is obvious that the first three accesses result in cold misses as it is the first access by each processor to the block. The miss at time slot 5 is a false-sharing miss because word A will not be accessed for as long as the block is in the cache; it is invalidated at time-slot 7. As for the miss experienced by processor 2 at time slot 6, it is obviously a true-sharing miss as it brings in a new value in the cache. For the same reason, the miss at slot 8 is also a true-sharing miss as processor 3 will subsequently access word B.

**b)** The false-sharing miss at time-slot 5 can be ignored.

**c)**
Number of read requests: 6 (traffic: 6 x 38 bytes = 228 bytes)
Number of bus updates: 2 (traffic: 2 x 10 bytes = 20 bytes)
**Total traffic: 248 bytes**

The only non-essential traffic is the read request associated with the false-sharing miss: 38 bytes. The essential traffic is 248 - 38 bytes = 210 bytes. The fraction of essential traffic is 210/248 = 85%

## Problem 5.14

**a)** When the home node is the same as the requesting node the miss penalty is the same as the time it takes to process a directory request which is 50 cycles according to the assumptions.
Cache miss time: 1 + 50 = 51 cycles.
Traffic: There is no traffic outside the node.

**b)** In the case when the memory copy is dirty some other node will have to service the cache miss. The time it takes to do that involves a directory lookup (50 cycles) at the home node, a remote read request (20 cycles), a lookup at the remote node (50 cycles), a flush operation that brings a copy of the block back to the home node, which is the same as the local node (100 cycles), and finally installing the block copy (50 cycles). Hence,
Cache miss time: 1 + 50 + 20 + 50 + 100 + 50 cycles = 271 cycles
Traffic involves sending a remote read request and flushing the block.
Traffic: 6 + 6 + 32 = 44 bytes

**c)** This scenario involves sending a read request to the home node (BusRd), doing a directory lookup, flushing the block back, and installing it at the requesting node.
Cache miss time: 1 + 20 + 50 + 100+ 50 cycles = 221 cycles
Traffic: 6 + 6 + 32 bytes = 44 bytes

**d)** This scenario involves sending a read request to the home node, doing a directory lookup, flushing the block back to the requesting node, and installing it there.
Cache miss time: 1 + 20 + 50 + 100 + 50 cycles = 221 cycles
Traffic: 6 + 6 + 32 bytes = 44 bytes

**e)** This scenario involves sending a read request to the home node (20 cycles), doing a directory lookup (50 cycles), sending a remote read request to the remote node (20 cycles), doing a lookup at the remote node (50 cycles), flushing the block back to the home node (100 cycles), processing it at the directory (50 cycles), flushing the block back to the local node (100 cycles) and installing it there (50 cycles).
Cache miss time: 1 + 20 + 50 + 20 + 50 + 100 + 50 + 100 + 50 cycles = 441 cycles
Traffic: 6 + 6 + 6 + 32 + 6 + 32 bytes = 88 bytes

## Problem 5.15

**a)** Same as in Problem 5.14a.

**b)** Same as in Problem 5.14b

**c)** Same as in Problem 5.14c

**d)** Same as in Problem 5.14d

**e)** This scenario involves sending a read request to the home node, doing a directory lookup, sending a request to the remote node, flushing the block back to the requesting node, and installing it there.
Cache miss time: 1 + 20 + 50 + 20 + 50 + 100 + 50 cycles = 291 cycles
Traffic: 6 + 6 + 6 + 32 bytes = 50 bytes

**f)** The time to process a cache miss will be shorter only in the case when the requesting node, the home node, and the remote node are all different. In that case a four-hop transaction is converted into a three-hop transaction.

## Problem 5.17

**a) Intra-cluster protocol:**
A directory entry is associated with each 2-level cache block frame. Since there are eight processors per cluster, each presence-flag vector consists of eight bits. With a 32-byte block size, the overhead becomes 8/(32x8) = 1/32 = 0.03. The overhead is approximately 3%.

**Inter-cluster protocol:**
Since there are 128 processors and eight processors per cluster, there are 16 clusters. Therefore, each directory entry at the memory contains 16 bits and yields twice as high an overhead as inside each cluster: Approximately 6%.

**b)** A limited-pointer directory protocol with two pointers inside each cluster and a coarse-vector directory protocol that partitions the clusters into groups of four clusters in each across clusters.

**Intra-cluster protocol**:
With eight processors per cluster each pointer contains 3 bits. With two pointers per 2-level cache block frame, the overhead is 6/(8 x 32) = 0.023. The overhead is approximately 2%.

**Inter-cluster protocol:**
There are 16 clusters. If we group them into groups of four clusters each, there are 4 groups and each presence flag in the coarse vector represents a group. There will be four bits in the presence flag vector which yields an overhead of 4/(8 x 32) = 1/64 = 0.016. The overhead is approximately 1.6%.

Since there are 128 processors and eight processors per cluster, there are 16 clusters. Therefore, each directory entry at the memory contains 16 bits and yields almost 3X as high an overhead as inside each cluster: Approximately 6%.

## Problem 8.2

Execution schedule for block (coarse grain) multithreading:

**Table 68: Block multithreading**

| Instruction (latency) | Dispatch (issue Q) | Issue | Register fetch | Exec start | Exec complete | CDB | Retire (T1) | Retire (T2) |
|---|---|---|---|---|---|---|---|---|
| X1(2) | 1(1) | 2 | 3 | 4 | 5 | 6 | 7 | |
| X2(2) | 1(2) | 4 | 5 | 6 | 7 | 8 | 9 | |
| X3(4) | 2(3) | 4 | 5 | 6 | 9 | 10 | 11 | |
| X4(2) | 2(4) | 8 | 9 | 10 | 11 | 12 | 13 | |
| X5(1,20) | 9(3) | 10 | 11 | 12 | 12 | 13 | 14 | |
| X6(1) | 9(4) | 11 | 12 | 13 | 13 | 14 | 15 | |
| Y1(2) | 15(1) | 16 | 17 | 18 | 19 | 20 | | 21 |
| Y2(3) | 15(2) | 18 | 19 | 20 | 22 | 23 | | 24 |
| Y3(2) | 16(3) | 21 | 22 | 23 | 24 | 25 | | 26 |
| Y4(2) | 16(4) | 23 | 24 | 25 | 26 | 27 | | 28 |
| Y5(3) | 24(3) | 25 | 26 | 27 | 29 | 30 | | 31 |
| Y6(1) | 24(4) | 28 | 29 | 30 | 30 | 31 | | 32 |

Execution schedule for interleaved (fine grain) multithreading:

**Table 69: Speculative scheduling with interleaved multithreading**

| Instruction (latency) | Dispatch | Issue | Register fetch | Exec start | Exec complete | CDB | Retire (T1) | Retire (T2) |
|---|---|---|---|---|---|---|---|---|
| X1(2) | 1(1) | 2 | 3 | 4 | 5 | 6 | 7 | |
| X2(2) | 1(2) | 4 | 5 | 6 | 7 | 8 | 9 | |
| Y1(2) | 2(3) | 3 | 4 | 5 | 6 | 7 | | 8 |
| Y2(3) | 2(4) | 5 | 6 | 7 | 9 | 10 | | 11 |
| X3(4) | 8(3) | 9 | 10 | 11 | 14 | 15 | 16 | |
| X4(2) | 8(4) | 13 | 14 | 15 | 16 | 17 | 18 | |
| Y3(2) | 11(3) | 12 | 13 | 14 | 15 | 16 | | 17 |
| Y4(2) | 11(4) | 14 | 15 | 16 | 17 | 18 | | 19 |
| X5(1,20) | 17(3) | 18 | 19 | **20** | 20 | 21 | 22 | |
| X6(1) | 17(4) | 19 | 20 | 21 | 21 | 22 | 23 | |
| Y5(3) | 19(3) | 20 | 21 | 22 | 24 | 25 | | 26 |
| Y6(1) | 19(4) | 23 | 24 | 25 | 25 | 26 | | 27 |

Looking at the issue, execution start and retire stages, interleaved multithreading is five cycles faster than block multithreading.

If the clock rate for interleaved multithreading is 20% slower, the time taken by interleaved multithreading is 27x1.2=32.4 and is a bit of slower than block multithreading.