6/19/2021

**Hafiz Amir Ijaz**

SP18-BCS-026

**Computer Architecture**

**Assignment -4**

**Exercise Problems**

# **5.19**

**Ans:-**

The general form of Amdahl’s law is

Speedup = Execution Time old/Execution Time new

All that needs to be done to compute the formula for speedup in this multiprocessor case is derive the new execution time. The exercise states that for the portion of the original execution time that can use I processors is given by F (I,p). If we let Execution time old be 1, then the relative time for the application on p processors is given by summing the times required for each portion of the execution time that can be sped up using I processors, where I is between 1 and p. These yields

Execution time\_new

Substituting this value for Execution time\_new into the speedup equation makes Amdahl’s Law a function of the available processors, p.

# **5.20**

**Ans:-**

**a. (i)** 64 processors arranged ‘a’ as a ring: largest number of communication hops = 32 -> communication cost = (100 + 10 x 32)ns = 420ns.

**ii)** 64 processors arranged as 8 x 8 processor grid: largest number of communication hops= 14 -> communication cost = (100 +10 x 14) ns = 240ns

**iii)** 64 processors arranged as a hypercube: largest number of hops = 6(log264) -> communication cost = (100 + 10 x 6)ns = 160 ns.

**b.** Base CPI = 0.5 cpi

(i) 64 processors arranged a as a ring: Worst case CPI = 0.5 + 0.2/100 x (420) = 1.34 cpi

(ii) 64 processors arranged as 8x8 processor grid: Worst case CPI = 0.5 + 0.2/ 100 x (240) = 0.98 cpi

(iii) 64 processors arranged as a hypercube: Worst case CPI = 0.5 + 0.2/ 100 x (160) = 0.82 cpi the average CPI can be obtained by replacing the largest number of communications hops in the above calculation by h, the average numbers of communications hops. That latter number depends on both the topology and the application.

**c.** Since the CPU frequency and the number of instructions executed did not change, the answer can be obtained by the CFI for each of the topologies (worst case or average) by the base (no remote communication)

# **5.21**

**Ans:-**

To keep the figures from becoming cluttered, the coherence protocol is split into two parts as was done in Figure 5.6 in the text. Figure S.34 presents the CPU portion of the coherence protocol, and Figure S.35 presents the bus portion of the protocol. In both of these figures, the arcs indicate transitions and the text along each arc indicates the stimulus and bus action that occurs during the transition between states. Finally, like the text, we assume a write hit is handled as a write miss. Figure S.34 presents the behavior of state transitions caused by the CPU itself. In this case, a write to a block in either the invalid or shared state causes us to broad-cast a "write invalidate" to flush the block from any other caches that hold the block and move to the exclusive state. We can leave the exclusive state through either an invalidate from another processor, or a read miss generated by the CPU. In the shared state only a write by the CPU or an invalidate from another processor can move us out of this state. In the case of transitions caused by events external to the CPU, the state diagram is fairly simple, as shown in Figure S.35. When another processor writes a block that is resident in our cache, we unconditionally invalidate the corresponding block in our cache. This ensures that the next time we read the data, we will load the updated value of the block from memory. Also, whenever the bus sees a read miss, it must change the state of an exclusive block to share as the block is no longer exclusive to a single cache.

The major change introduced in moving from a write-back to write-through cache is the elimination of the need to access dirty blocks in another processor's caches. As a result, in the write-through protocol it is no longer necessary to pro-vide the hardware to force write back on read accesses or to abort pending memory accesses. As memory is updated during any write on a write-through cache, a processor that generates a read miss will always retrieve the correct information from memory. Basically, it is not possible for valid cache blocks to be incoherent with respect to main memory in a system with write-through caches.

Diagram

Description automatically generatedDiagram

Description automatically generated

# **5.26**

**Ans:-**

The problem illustrates the complexity of cache coherence protocols. In this case, this could mean that the processor P1 evicted that cache block from its cache and immediately requested the block in subsequent instructions. Given that the write-back message is longer than the request message, with networks that allow out-of-order requests, the new request can arrive before the write back arrives at the directory. One solution to this problem would be to have the directory wait for the write back and then respond to the request. Alternatively, the directory can send out a negative acknowledgment (NACK). Note that these solutions need to be thought out very carefully since they have potential to lead to deadlocks based on the particular implementation details of the system. Formal methods are often used to check for races and deadlocks.

# **5.27**

**Ans:-**

If replacement hints are used, then the CPU replacing a block would send a hint to the home directory of the replaced block. Such hint would lead the home directory to remove the CPU from the sharing list for the block. That would save an invalidate message when the block is to be written by some other CPU. Note that while the replacement hint might reduce the total protocol latency incurred when writing a block, it does not reduce the protocol traffic.

# **5.28**

**Ans:-**

**a.** Considering first the storage requirements for nodes that are caches under the directory subtree: The directory at any level will have to allocate entries for all the cache blocks cached under that directory's subtree. In the worst case (all the CPUs under the subtree are not sharing any blocks), the directory will have to store as many entries as the number of blocks of all the caches covered in the subtree. That means that the root directory might have to allocate enough entries to reference all the blocks of all the caches. Every memory block cached in a directory will represented by an entry <block address, k-bit vector>, the k-bit vector will have a bit specifying all the subtrees that have a copy of the block. For example, for a binary tree an entry <m, 11> means that block m is cached under both branches of the tree. To be more precise, one bit per subtree would be adequate if only the valid/invalid states need to be recorded; however to record whether a block is modified or not, more bits would be needed. Note that no entry is needed if a block is not cached under the subtree. If the cache block has m bits (tag + index) then and s state bits need to be stored per block, and the cache can hold b blocks, then the directories at level L-1 (lowest level just above CPU's) will have to hold k x b entries. Each entry will have (m + k x s) bits. Thus each directory at level L-1 will have (mkb + k2bs) bits. At the next level of the hierarchy, the directories will be k times bigger. The number of directories at level i is k'. To consider memory blocks with a home in the subtree cached outside the subtree. The storage requirements per directory would have to be modified.

Diagram

Description automatically generated

**b.** Simulation

# **5.29**

**Ans:-**

Test and set code using load linked and store conditional.

MOV R3, #1

LL R2, 0 (R1)

SC R3, 0(R1)

Typically this code would be put in a loop that spins until a 1 is returned in R3.

# **5.30**

**Ans:-**

Assume a cache line that has a synchronization variable and the data guarded by that synchronization variable in the same cache line. Assume a two processor system with one processor performing multiple writes on the data and the other processor spinning on the synchronization variable. With an invalidate protocol, false sharing will mean that every access to the cache line ends up being a miss resulting in significant performance penalties.

# **5.31**

**Ans:-**

The monitor has to be place at a point through which all memory accesses pass. One suitable place will be in the memory controller at some point where accesses from the 4 cores converge (since the accesses are uncached anyways). The monitor will use some sort of a cache where the tag of each valid entry is the address accessed by some load-linked instruction. In the data field of the entry, the core number that produced the load-linked access -whose address is stored in the tag field- is stored.

This is how the monitor reacts to the different memory accesses.

■ Read not originating from a load-linked instruction:

Bypasses the monitor progresses to read data from memory

■ Read originating from a load-linked instruction:

Checks the cache, if there is any entry with whose address matches the read address even if there is a partial address match, the matching cache entry is invalidated and a new entry is created for the new read. If there is no matching entry in the cache, then a new entry is created. In either case the read progresses to memory and returns data to originating core.

■ Write not originating from a store-conditional instruction: o Checks the cache, if there is any entry with whose address matches the write address even if there is a partial address match (for example, read [0:7] and write [4:11] overlap match in addresses [4:7]), the matching cache entry is invalidated. The write progresses to memory and writes data to the intended address.

■ Write originating from a store-conditional instruction:

Checks the cache, if there is any entry with whose address matches the write address even if there is a partial address match (for example, read [0:7] and write [4:11] overlap match in addresses [4:7]), the core number in the cache entry is compared to the core that originated the write.

If the core numbers are the same, then the matching cache entry is invalidated, the write proceeds to memory and returns a success signal to the originating core. In that case, we expect the address match to be perfect —not partial- as we expect that the same core will not issue load-linked/store conditional instruction pairs that have overlapping address ranges. If the core numbers differ, then the matching cache entry is invalidated, the write is aborted and returns a failure signal to the originating core. This case signifies that synchronization variable was corrupted by another core or by some regular store operation.

# **5.33**

**Ans:-**

Inclusion states that each higher level of cache contains all the values present in the lower cache levels, i.e., if a block is in Ll then it is also in L2. The problem states that L2 has equal or higher associativity than LI, both use LRU, and both have the same block size.

When a miss is serviced from memory, the block is placed into all the caches, i.e., it is placed in Ll and L2. Also, a hit in Ll is recorded in L2 in terms of updating LRU information. Mother key property of LRU is the following. Let A and B both be sets whose elements are ordered by their latest use. If A is a subset of B such that they share their most recently used elements, then the LRU element of B must either be the LRU element of A or not be an element of A.

This simply states that the LRU ordering is the same regardless if there are 10 entries or 100. Let us assume that we have a block, D, that is in LI, but not in L2. Since D initially had to be resident in 1.42, it must have been evicted. At the time of eviction D must have been the least recently used block. Since an 1.2 eviction took place, the processor must have requested a block not resident in Ll and obviously not in L2. The new block from memory was placed in L2 (causing the eviction) and placed in LI causing yet another eviction. L1 would have picked the least recently used block to evict. Since we know that D is in LI, it must be the LRU entry since it was the LRU entry in L2 by the argument made in the prior paragraph. This means that L I would have had to pick D to evict. This results in D not being in Ll which results in a contradiction from what we assumed. If an element is in Ll it has to be in L2 (inclusion) given the problem's assumptions about the cache.