CS 351 Midterm 2 Exam Solutions

Notes:

- You must explain your answers to receive partial credit.
- You will lose points for incorrect extraneous information, even if the answer is otherwise correct.

Question 1 [25 points].

a. Write the term that fits each definition.

The range of bits of a memory address that are used to locate data within a cache block or page.

Offset

A mechanism for I/O handling that involves repeatedly checking status bits, usually using a while loop.

Polling

The exception raised when a requested virtual address is not mapped to physical memory.

Page fault

b. Write pseudocode that would have a 0% hit rate for a fully associative cache with LRU replacement. The cache has four 8-byte blocks.

If we assume that "array" is an array of 4B integers, this works:

```
while (true) {
        for (int i=0; i<10; i+=2) {
                array[i]++;
        }
}
```

We access elements 0, 2, 4, 6, 8, and then 0 again…but it has just been evicted! And so on, where we are always trying to access the least recently used element, which has just been evicted.

(c) How can cache designers reduce…

- compulsory/ "cold start" misses?

  The canonical answer is to increase the block size, bringing in a memory location's neighbors before they are explicitly requested. However, lots of people talked about implementing prefetching schemes of one kind or another, which is great and gets full credit!

- conflict misses?

  Increase associativity

- capacity misses?

  Increase the cache size. Decreasing the block size arguably works too, if the problem is that there aren't enough cache blocks; this answer also gets full credit.

(d) What is the distinction between synchronous and asynchronous bus protocols?

In synchronous protocols, sender and receiver use the same clock to coordinate transfers. In asynchronous protocols, there's no clock, so a handshaking protocol must be used to coordinate between sender and receiver.

Question 2 [15 points].

(a) What is the average memory access time for the following memory system?

- Level 1 cache: 91% hit rate, 1-cycle access time.
- Level 2 cache: 98% hit rate, 15-cycle access time.
- Memory: 140-cycle access time.

Cache hits and misses both require some number of cycles to access the cache (the access time). You may assume that all memory accesses are hits in main memory.

The main thing to remember is that you incur the access time for a cache regardless of whether it's a hit or a miss. So if we have an access that is not in cache at all, we first spend 1 cycle accessing L1 to find out that it's a miss. Then we spend 15 more cycles accessing L2 to find out that we've missed again, and finally 140 cycles accessing memory.

We have 3 cases…
L1 hit (0.91 probability): 1 cycle
L1 miss, L2 hit (0.09 * 0.98 probability): 1 cycle + 15 cycles
L1/L2 miss, memory hit (0.09 * 0.02 probability): 1 cycle + 15 cycles + 140 cycles

Altogether:
AMAT = 0.91 * 1 + 0.09 * (0.98*(1+15) + 0.02*(1+15+140) = $\underline{2.602}$ cycles

No individual memory access will take 2.602 cycles, but this should be the average over all memory accesses.

(b) Assume that the L2 cache is write-allocate and write-back and that there is no write buffer. If D is the percentage of evicted blocks that are dirty, how many cycles (in terms of D) do L2 write-back operations add to the average memory access time?

When we have a write-back operation, we have to first access L2 to read the block and then write it out to memory, incurring an extra L2 access and an extra memory access = 155 cycles. (If you interpreted write-back as only the writing part and said 140 cycles, that's fine.)

But we only need to do this write-back operation *if* we missed in L2 and have to bring in the new data and kick out old data. The probability that this happens is the probability of an L2 miss * the probability that the evicted block is dirty: in other words, 0.09 * 0.02 * D.

The number of cycles that get added to the average memory access time is

Cycles/write-back * probability of write-back = 155*0.09*0.02*D

= <u>0.279 * D</u>

Question 3 [20 points].

A byte-addressable machine with 32-bit memory addresses has a cache with the following properties:

- 16-byte cache blocks
- 8KB of data in the cache
- Direct-mapped
- Write-through

(a) How many cache blocks are there?

# cache blocks = (bytes in cache)/(bytes per block) = 8KB/16B = $2^{13}/2^4 = 2^9$ = <u>512</u>

(b) How many cache sets are there?

In a direct-mapped cache, the number of sets is the same thing as the number of blocks. (Remember, # sets = # blocks/associativity, and here the associativity is 1). So <u>512</u>.

(c) How many bits of **meta**data are required for each cache entry? Explain what each is for.

We need tag bits and a valid bit. We don't need LRU bits, since we have zero flexibility in where to locate a block. We also don't need a dirty bit, since the cache is write-through and we update memory immediately when we get a write.

# tag bits = address bits – index bits – offset bits

# index bits = enough bits to choose a cache set. Since there are 512 = 2^9 sets, we need 9 index bits to select among them.

# offset bits = enough bits to choose a byte within a cache block. Since there are 16 = 24 bytes/block, we need 4 offset bits to select among them.

# tag bits = 32 – 9 – 4 = 19.

Total metadata bits/block = 19 tag bits + 1 valid bit = <u>20</u>

(d) How many bits are needed to implement the cache (data and metadata)?

Per block, we have 20 bits of metadata and 16 BYTES (128 bits) of data = 148 bits/block. There are 512 blocks, so we need

148 * 512 = <u>75776 bits</u> or <u>74 Kb</u>

Question 4 [15 points].

A byte-addressable memory system has 48-bit virtual addresses and 32-bit physical addresses, with 8KB pages.

(a) How many virtual pages does each process's page table have?

Each page contains 8KB = 2^{13} bytes. To tell these bytes apart, we need 13 address bits. The remaining (48-13 = 35) bits of the virtual address give the virtual page number. If we have 35 bits for the page number, we have $\underline{2^{35}}$ virtual pages.

(b) How many physical pages does each process's page table have?

Same logic, except that now we're using the physical address, which has 32 bits. So we have 32-13 = 19 bits for the PPN and thus $\underline{2^{19}}$ physical pages.

(c) How many page table entries are needed?

We need one entry per virtual page, so $\underline{2^{35}}$ PTEs.

(d) How many bits (data and metadata) are needed for each page table entry?

We need the PPN and some metadata: at the very least, a valid bit, a dirty bit, and a use bit. We'll also need some number of protection bits (not specified in the problem so optional). The PPN is 19 bits, plus valid/dirty/use = 22 bits/PTE.

(e) How many pages does the page table occupy?

The number of *bytes* in the page table is the number of bytes/PTE times the number of PTEs. Rounding the 22 bits up to 3 bytes, we get 3 * 2^{35} bytes. The question asks about pages, so we divide by 8KB = 2^{13} bytes and get $\underline{3 * 2^{22}}$ pages.

Question 5 [10 points].

(a) Does a workload consisting of many small write accesses perform better on an N-disk RAID 1 system or an N-disk RAID 4 system, where N is the total number of disks (including the redundant ones)? **Explain** for credit.

RAID 1. RAID 4 writes bottleneck on the parity disk, which means that we get *zero* parallelism for a bunch of small writes. With RAID 1, we can have N/2 accesses going at the same time.

(b) Is RAID5 better or worse than RAID1 in the following areas? **Explain** for credit**.**

Reliability: RAID 5 is worse; you can lose a *maximum* of 1 disk safely, and it takes longer to rebuild that disk. With RAID 1, you can lose up to N disks safely.

Cost overhead: RAID 5 is better: only 1 disk is redundant, while RAID 1 has N/2 redundant disks.

Performance on small accesses: About the same: for reads, all disks can be read. For writes, 2 disks must be written in each case.

Performance on large accesses: RAID 5 is better because striping allows you to parallelize the access over all disks.

Question 6 [15 points].

A workload consists of repeated CPU processing overlapped with 16KB disk transfers. The system has the following properties:

- The CPU takes 5 ms to process the data
- The I/O bus, which is clocked at 300 MHz, takes 10 cycles to initiate a transfer and then transfers 32B of data per clock cycle.
- The disk has an average seek time of 3 ms
- The disk rotates at 10000 rpm
- The disk's average transfer speed is 30 MB/s
- The disk's controller overhead is 0.4 ms.

(a) How long does it take the disk to do an 8K transfer?

Disk time = controller latency + seek time + rotational time + transfer time

Controller latency = 0.4 ms (from problem)
Seek time = 3 ms (ditto)
Rotational time = 0.5 rotations on average / 10,000 (rotations/min) * 60,000 ms/min = 3 ms
Transfer time = 8KB / 30(MB/s) * 1000 ms/s = 0.26 ms

Total = 0.4 ms + 3 ms + 3 ms + 0.26 ms = <u>6.66 ms</u>

(b) If you could purchase a faster disk for $100 for a system dedicated to this workload, should you?

The CPU can process each 16KB transfer in 5 ms. The disk is going to take even longer than 6.66 ms, since the number in part (a) is for 8KB transfers rather than the 16KB transfers in our workload. The only question is whether the bus can keep up if we get a faster disk.

The bus takes 10 cycles to start up + 1 cycle per 32B = 10 + 16KB/32(B/cy) = 10 + $2^{14}/2^{5}$ cycles = 522 cycles. We divide this by the clock rate to get 0.00174 ms for each 16KB transfer – not a bottleneck at all. Therefore, it would be to our benefit to get a faster disk.