Assignment 10

Alex Clemmer
Student number: u0458675

Problem 1:

6.3.1 The book's example [pg. 577] contains a constant for controller overhead, while the question contains a "controller transfer rate". I will calculate exactly how I was told to by the TAs.

$$\begin{aligned} \textbf{(a)} \quad & 11.0 \text{ ms} + \frac{0.5 \text{ rotation}}{7,200 \text{ RPM}} + \frac{1 \text{ KB}}{34 \text{ MBytes/sec}} + \frac{1\text{K B}}{480\text{Mbits/sec}} \approx 15.1975 \text{ ms} \\ \textbf{(b)} \quad & 9.0 \text{ ms} + \frac{0.5 \text{ rotation}}{7,200 \text{ RPM}} + \frac{1 \text{ KB}}{30 \text{ MBytes/sec}} + \frac{1\text{KB}}{500 \text{ Mbits/sec}} \approx 13.2013 \text{ ms} \end{aligned} \tag{1}$$

6.3.2 Our best time would contain no seek overhead and no rotational latency. Thus:

$$\begin{aligned} \textbf{(a)} \quad & \frac{2 \text{ KB}}{34 \text{ MBytes/sec}} + \frac{2 \text{ KB}}{480\text{Mbits/sec}} \approx 0.061711519607 \text{ ms} \\ \textbf{(b)} \quad & \frac{2 \text{ KB}}{30 \text{ MBytes/sec}} + \frac{2 \text{ KB}}{500\text{Mbits/sec}} \approx 0.06920016666 \text{ ms} \end{aligned} \tag{2}$$

6.3.3 There is a clear and dominant factor here, and it is seek time. Whether or not we get data fast or slow is pretty much contingent upon how long it takes to seek. After that, the dominant factor is the rotational latency. Obviously we don't want to downplay how much time this takes compared to the other two factors, but even worst-case it still only takes around half the total time that an average-case seek would take.

Also worth noting is that increasing block size doesn't really change the read or write time by that much. In other words, for reasonably-sized blocks, the size does not appear to matter that much.

Problem 2:

6.6.1

$$\begin{aligned} \textbf{(a)} \quad & \frac{1 \text{ KB}}{34 \text{ MBytes/sec}} + \frac{1 \text{ KB}}{480 \text{ Mbits/sec}} \approx 0.0308557598 \text{ ms} \\ \textbf{(b)} \quad & \frac{1 \text{ KB}}{30 \text{ MBytes/sec}} + \frac{1\text{KB}}{500\text{MBits/sec}} \approx 0.03460008333 \text{ ms} \end{aligned} \tag{3}$$

6.6.2

$$\begin{aligned} \textbf{(a)} \quad & \frac{0.5 \text{ KB}}{34 \text{ MBytes/sec}} + \frac{0.5 \text{ KB}}{480\text{Mbits/sec}} \approx 0.0154279 \text{ ms} \\ \textbf{(b)} \quad & \frac{0.5 \text{ KB}}{30 \text{ MBytes/sec}} + \frac{0.5 \text{ KB}}{500\text{Mbits/sec}} \approx 0.0173000 \text{ ms} \end{aligned} \tag{4}$$

6.6.3 Obviously, the larger the memory, the more decode logic you have to wade through, not to mention the fact that propagating addresses around and through the Flash "array". What may be surprising is that the dropoff appears to be so precipitous. Remember though, that in this example, the size increases by powers of two, while the dropoff does not decline at this rate. So the effects are not as severe as one might think.

Problem 3:

3-A So, first, the assumption I'm going to make is that one bit is sent per cycle. To compute this, we start by seeing how far a signal will travel in a single cycle:

$$\frac{2 \cdot 10^8 \text{ m/s}}{100 \cdot 10^6 \text{ MHz}} = 2 \text{ meters} \tag{5}$$

Then, we evaluate the "capacity" of the 10 meter wire. Assuming the rate of signal propagation across the wire is constant, that means that immediately as one is introduced at one end, another will disappear at the other end. Thus at any point there should be **5 bits** moving across the wire.

3-B The first thing we look at is how much wire the signal covers in one cycle:

$$\frac{2 \cdot 10^8 \text{ m/s}}{10 \cdot 10^9 \text{ MHz}} = 0.02 \text{ meters} \tag{6}$$

Using similar logic to above we get $\frac{1500 \text{ meters}}{0.02 \text{ meters/bit}} = 75,000 \text{ bits}$

3-C This is tricky. First we'll do **(a)**. Let's start with what we know. There is 1 request of 100 bytes that is sent out. Since we are writing, there is an overhead of 0.03 ms. The other end must read, which also incurs this overhead. In other words, the signal gets sent to the receiver, and then the server takes 0.03 ms to process this on top of transmission time. At this point, our total should look like this:

$$0.03 \text{ ms} + \frac{100 \cdot 10.2 \text{ bits}}{100 \cdot 10^6 \text{ bits/sec}} + \frac{100 \cdot 2^{10} \text{ bits}}{100 \cdot 10^6} + 0.03 \text{ ms} \tag{7}$$

Since I do not see anything about the amount of time it takes to find the data on disk or respond, I will not account for it here, although you should note that this is DEFINITELY NOT negligible. So, assuming that there is no data retrieval wait, we have another 0.03 ms that we must wait before we can write back in response.

Now we have to send the actual message. That's 100,000 bytes plus the overhead for each byte. And then at the end of the line, there's another read overhead. So, to our previous equation, we should add this:

$$\ldots + 0.03 \text{ ms} + \frac{100,000 \cdot 10.2 \text{ bits}}{100 \cdot 10^6 \text{ bits/sec}} + \frac{100,000 \cdot 2^{10} \text{ bits}}{100 \cdot 10^6} + 0.03 \text{ ms} \tag{8}$$

For **(a)**, this gives us 1.0352342 seconds $+4 \cdot 0.03$ ms, for a **total of 1.0353542**.
For **(b)**, the same sort of derivation follows, for a **total of 0.010472342**.

3-D The bottleneck is definitely *not* the speed of the signal. The bottleneck is how many time we send a signal per second. If you wanted to decrease the communication latency, I would start by increasing the frequency of the signal.

Problem 4:

7.6.1 So there are $(m \times p \times n)$ multiplications, which leaves us with $(m \times p \times (n-1))$ additions. Unfortunately, we will be adding up the products, which means that we need to wait until at least two are available per addition. Thus the speedup should be in the ballpark of 4 times.

7.6.2 In this case, we are basically mapping different elements to the same cache line. Thus, each update ends up causing a cache miss, and so any speedup gained is diminished by a factor of 3 multiplied by the costs associated with a cache miss.

7.6.3 The simplest way is to traverse the matrix via columns instead of rows. That should pretty much eliminate the problem we were having because then the elements will end up mapped to different cache lines.

The only other thing I can think of would be to process the matrix index (i, j) and $(i + 1, j)$ on the same core. Remember that we are dealing with false sharing, and thus this is important.

Problem 5: