CS61 Section Notes Solutions

Week 6 (Fall 2011)

Topics to be covered

- **Memory and Storage Technologies**
  - **DRAM**
  - **Hard Drives**
- **Caches**
  - **Direct Mapped Caches**
  - **Set Associative Caches**

Memory and Storage Technologies

DRAM

A $d \times w$ DRAM contains d supercells, each of which contains w bits. The d supercells are organized into a rectangular array of r rows and c columns (i.e. d=r*c). To access a supercell, we send a row access strobe (RAS), followed by a column access strobe (CAS). This identifies a unique supercell.

Question 1: What are the advantages of sending the address of a supercell in two steps? What are the disadvantages?

Answer 1: Sending the address in two steps reduces the number of address pins required on the chip. This is good, because pins are in short supply (they take a relatively large amount of physical space). However, it increases the time needed to send an address to the supercell.

Question 2: Given a DRAM with $d \times w$ supercells, how would you chose the dimensions of the rectangular array to minimize the number of address pins needed?

Answer 2: We need max ($\lceil \log_2 r \rceil$,$\lceil \log_2 c \rceil$) pins, where $r \times c = d$. This is minimized when r and c are close to the square root of d. That is, the squarer the rectangular array, the better.

Hard Drives

Question 3: Consider a disk with a rotational rate of 10,000 RPM, an average seek time of 8 ms, and an average of 500 sectors per track. Estimate the average time to read a random sector from disk. Do this by summing the estimates of the seek time, rotational latency, and transfer time.

Answer 3: The average access time is the sum of the seek time, rotational latency, and transfer time.
The seek time is given as 8ms. Once the head is in the right place, on average we will need to wait for half a rotation of the disk for the correct sector to come under the head. Thus, on average, the rotational latency is half the time it takes the disk to make a complete revolution. The disk spins at 10000 RPM, so it takes 1/10000 of a minute to make one revolution. Equivalently, (1000 ms/sec × 60 sec/minute) / 10000 RPM = 6 ms to make one revolution. So rotational latency is 3ms.
The transfer time is the time it takes for the head to read all of the sector. The head is now at the start of the sector, so how long does it take for the entire sector to go past the head? Since there are on average 500 sectors per track, we need 1/500th of a revolution of the disk to read the entire sector. We can work this out as (time for one revolution of disk) / 500 = 6ms / 500 = 0.012ms.

So the total time is 8ms + 3ms + 0.012ms ≈ 11ms. We can clearly see that getting to read the first byte of the sector takes a long time, but reading the rest of the bytes in the sector is essentially free.

Question 4: Disk controllers map logical blocks to physical locations on the disk. Suppose

that a 1MB file consisting of 512-byte logical blocks is stored on a disk drive with the following characteristics:

Rotational rate:	10,000 RPM
Average seek time	5ms
Average # sectors/track	1000
Surfaces	4
Sector size	512 bytes

Suppose that a program reads all the blocks of this file sequentially, and that the time to position the head over the first block is the average seek time plus the average rotational latency.

4a) What is the best case for mapping logical blocks to disk sectors? Estimate the time required to read the file in this best case scenario.

4b) Suppose that the logical blocks are mapped randomly to disk sectors. Estimate the time required to read the file in this scenario.

Answer 4: First, some basic properties: the time for one revolution is (1000 ms/sec × 60 sec/ minute) / 10000 RPM = 6 ms. The average rotational latency is thus 3ms.
Also, a 1MB file will need 2000 logical blocks.

4a) In the best case, the 2000 logical blocks are on the same cylinder (so the head will not need to move once it is in the correct position) in contiguous sectors. So, once the head moves to the correct position, it will take two full rotations (1000 sectors per rotation) to read the file. (We are ignoring the possibility of reading the data in parallel.) So the total time to read is (average seek time) + (average rotational latency) + (transfer time [2 full rotations]) = 5ms + 3ms + 12ms = 20ms.

4b) In this case, for each block, we will need to move the head, and wait for the correct sector to come under the head. So the total time to read the file is
2000 × ((average seek time) + (average rotational latency) + (transfer time [1/1000 of a full rotation]))
= 2000 × (5ms + 3ms + 0.006ms) = 16012 ms ≈ 16 seconds!!!!

Storage trends

Question 5: Given the following trends for the cost of rotating disk storage, estimate the year that you will be able to purchase a petabyte of storage (=10^{15} bytes = 1000 terabytes = 1 million gigabytes) for under $500. (In 2010 a petabyte of storage costs about $300,000).

Year	2000	2005	2010
$/MB	0.01	0.005	0.0003

Answer 5: In the 10 years between 2000 and 2010, the cost for 1 MB of rotating disk storage dropped by a factor of about 30 ≈ 2^5. So let's assume that the cost is dropping by a factor of 2 about every 2 years, and this trend continues.

In 2010, a petabyte of storage costs about $300,000. If the price halves 9 times, one petabyte will cost a bit more than $500 (since $300,000 / 2^9 ≈ $585). So it will take around 9 or 10

price-halvings to get below $500, which is around 18-20 years. So, around 2030 seems like a reasonable estimate.

(How much data do you think you generate in a week? Emails, photos, videos, ... If you had massive amounts of cheap storage, what could you record?)

Caches

Locality

Question 6. What are the two types of locality?

Answer 6. Temporal and spatial locality

Question 7. Why do we care about locality? In general, how does locality affect program execution time? How does increasing the size of the working set affect locality? What about changing the loop's "stride" distance?

Answer 7. In general, programs with good locality run faster than programs with poor locality. Increasing the size of the working set affects temporal locality, since we can't hold all the data we are working with in data at once. Increasing the stride hurts spatial locality since we are accessing pieces of data that are farther apart from each other. With very long stride lengths, we will never access the same cache line twice in a row.

Question 8. Look at the following code.

```
#define N 1000

typedef struct {
   int vel[3];
   int acc[3];
} point;

point p[N];

void clear1(point* p, int n)
{
   int i, j;

   for (i = 0; i < n; i++) {
       for (j=0; j < 3; j++)
                   p[i].vel[j] = 0;
       for (j=0; j < 3; j++)
                   p[i].acc[j] = 0;
   }
}
```

```
void clear2(point* p, int n)
{
   int i, j;

   for (i = 0; i < n; i++) {
       for (j=0; j < 3; j++) {
                   p[i].vel[j] = 0;
                   p[i].acc[j] = 0;
       }
   }
}

void clear3(point* p, int n)
{
   int i, j;

   for (j=0; j < 3; j++) {
       for (i = 0; i < n; i++)
                   p[i].vel[j] = 0;
        for (i = 0; i < n; i++)
                   p[i].acc[j] = 0;
   }
}
```

How would your order these functions from that which displays the most spatial locality to that which displays the least?

Answer 8. Function `clear1` accesses the array using a stride-1 reference pattern so it has the best spatial locality. Funciton `clear2` scans each of the N structs in order, but within each struct it hops around, so it has worse spatial locality than `clear1`. Function `clear3` hops around within each struct, but also hops from struct to struct, so `clear3` exibits the worst spatial locality of the three functions.

Direct-Mapped Caches

Direct-mapped caches have only one cache line per set (*E = 1*). Consider a cache with *B = 16* and *S = 2*. Assume floats are 4 bytes. For simplicity, also assume (incorrectly!) that `x[]` starts at memory address 0x0 and `y[]` at address 0x20 (so the two arrays are adjacent in memory). Consider the following code.

```
float dotprod(float x[8], float y[8])
{
      float sum = 0.0;
      int i;

      for (i = 0; i < 8; ++i)
      sum += x[i] * y[i];
      return sum;
}
```

Question 9. How many bits are used to determine the set to which an address will belong? Which bits of the address are used to determine the set?

Answer 9. Since there are a total of 2 sets, you need one bit to determine to which set an address will belong. Since each block is $16=2_4$ bytes, 4 bits are needed for the block offset. If a memory address is 32 bits long, the first 27 bits (starting from the most significant) will be the tag, the next will be the set offset and the last 4 the block offset.

Question 10. Step through the access pattern of the function as it executes. Is the cache efficiently used?

Answer 10. The first memory access is to `x[0]`, at address 0x0. The first 16 bytes of `x` are loaded into set 0. The next access is to `y[0]`, at address 0x20, which also maps to set 0, so `x` is evicted. The next access to `x` is still in the first 16 bytes, so must evict `y` from set 0, and so on. Set 1 is not used until `x[4]` (at address 0x10) is accessed, at which point the same process repeats using set 1. Unfortunately, the cache is used poorly in this example.

Set Associative Caches

The code we just saw causes the cache to *thrash*. A cache *thrashes* when it repeatedly loads and then evicts the same set of cache blocks. We can alleviate this issue with a cache that has more than one cache line per set. A cache with $1 < E < C/B$ is called an E-way set associative cache.

Question 11. Suppose we run the code in Questions 8-9 using a 2-way set associative cache. How will this change the program's use of the cache?

Answer 11. Each set of the cache now has two lines which can store data. Thus, `x[0]-x[3]` will be loaded into set 0, line 0, `y[0]-y[3]` will be loaded into line 1, and neither will need to be evicted when the next elements of both arrays are read. The same process will repeat with set 1 when array elements 4-7 are accessed. Adding one more line to each set increased our number of cache hits from 0 to 12!

Of course, adding cache lines adds hardware complexity. The cache must check the tag and valid bit of each line in the appropriate set. Since, within a set, any line can contain any block, the cache design must also include a policy for deciding which line to eject when a conflict miss occurs. Many such policies are used, all of which have pros and cons depending on the situation.

Question 12 What are the advantages of a LRU (Least Recently Used) cache eviction policy? The disadvantages? What approaches help overcome these shortcomings?

Answer 12 A LRU eviction policy makes it less likely that we will evict a cache line that is currently part of a stride-1 access pattern. However, such a policy takes additional time and hardware. We can save on space by using a NRU (Not Recently Used) policy since we only need to keep track of whether a cache line was recently accessed. We can save on both time and space by randomly ejecting a cache line. (Follow Up: Why might we want to used a LRU policy despite the overhead in lower levels of the cache? The miss penalty is very high if we need to go to DRAM or disk, so it is especially important to minimize misses with good replacement policies).

Question 13 Another possible cache eviction policy is MRU (Most Recently Used): the most recently used line in a set is chosen for eviction. In general, is this a good eviction policy? Is

there any situation where this would be a good policy?

Answer 13. The policy is cheap to implement. But in general it will be a poor policy, as it does not help the performance of code that exhibits good spatial locality. There may be some cases where MRU does well, but these are unlikely to be common. For example, a linear search of an array that will not be accessed again, so the cache is not filled with data that will only be used once.

Question 14 What are two policies for writing data back to memory and what are the advantages of each?

Answer 14 One policy is write-back and another policy is write-through. Write-back avoids needing to pass data over the bus for every time we want to write data, but requires us to keep track of which cache lines need to be written when they are evicted. Write-through allows us to avoid keeping track of “dirty” but puts additional strain on the memory bus.