The Memory Hierarchy

Annalise Tarhan

September 19, 2020

6 Homework Problems

Note: I'm using the international edition of the textbook, which is explicitly disowned on the CS:APP website. Most of the time the difference is small, but there are some questions in this section that make less sense than they should.

6.22 Maximize the capacity of a disk where each track contains the same number of bits, determined by the size of the innermost ring. The radius of the disk is r and the radius of the hole is x*r.

The variable to be determined is x, the size of the hole expressed as a fraction of the size of the disk. Its possible values range from 0 to 1, and the value that maximizes the disk's capacity will be somewhere in between. We calculate the disk's capacity as the product of the number of equally sized tracks and the size of each track. Taking the derivative of the product and finding the critical point gives the capacity maximizing value of x.

In the following expressions, t is the number of tracks per unit of distance from the center. The value of t and its units are irrelevant for our purposes.

Number of tracks: $t(r - rx)$
Size of each track: $2\pi rx$
Capacity: $2\pi rtx(r - rx) = 2\pi r^2tx - 2\pi r^2tx^2$

$(2\pi r^2t - 2\pi r^2tx^2)dx = 0$
$2\pi r^2t - 4\pi r^2tx = 0$
$2\pi r^2t = 4\pi r^2tx$
$1 = 2x$
$x = 1/2$

To maximize the capacity of the disk, the radius of the hole should be half the radius of the disk.

6.23 Estimate the average time (in ms) to access a sector on a disk with a rotational rate of 12,000 RPM, an average seek time of 3 ms, and an average of 500 sectors per track.

$T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ read}$

$T_{avg\ seek} = 3$

$T_{avg\ rotation} = \frac{1}{2} * \frac{1}{RPM} * \frac{60000\ ms}{min} = \frac{1}{2} * \frac{1}{12000} * 60000 = \frac{5}{2}$

$T_{avg\ read} = \frac{1}{RPM} * \frac{1}{avg\ sectors\ per\ track} * \frac{60000\ ms}{min} = \frac{1}{12000} * \frac{1}{500} * 60000 = \frac{1}{100}$

$T_{access} = 3 + \frac{5}{2} + \frac{1}{100} \approx 5\frac{1}{2}$

6.24 Consider a 2MB file consisting of 512-byte logical blocks stored on a disk drive with the given characteristics.

6.24.1 Estimate the best case time required to read the file given the best possible mapping of logical blocks to disk sectors.

$T_{read} = T_{avg\ seek} + T_{avg\ rotation} + 512 * T_{avg\ read}$

$T_{avg\ seek} = 8$

$T_{avg\ rotation} = \frac{1}{2} * \frac{1}{RPM} * \frac{60000\ ms}{min} = \frac{1}{2} * \frac{1}{18000} * 60000 = 1\frac{2}{3}$

$T_{avg\ read} = \frac{1}{RPM} * \frac{1}{avg\ sectors\ per\ track} * \frac{60000\ ms}{min} = \frac{1}{18000} * \frac{1}{2000} * 60000 = \frac{1}{600}$

$T_{read} = 8 + 1\frac{2}{3} + 512 * \frac{1}{600} = 10.52\ ms$

6.24.2 Estimate the random case time required to read the file when the blocks are mapped randomly.

$T_{read} = 512 * (T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ read})$

$T_{read} = 512 * (8 + 1\frac{2}{3} + \frac{1}{600}) \approx 4950\ ms$

6.25 Fill in the table with the characteristics of differently sized caches.

Cache	m	C	B	E	S	t	s	b
1	32	1,024	4	4	64	23	7	2
2	32	1,024	4	256	1	30	0	2
3	32	1,024	8	1	128	21	8	3
4	32	1,024	8	128	1	28	0	4
5	32	1,024	32	1	32	20	6	6
6	32	1,024	32	4	8	23	3	6

6.26 Fil in the table with the characteristics of differently sized caches.

Cache	m	C	B	E	S	t	s	b
1	32	2,048	8	1	256	21	8	3
2	32	2,048	4	4	128	23	7	2
3	32	1,024	2	8	64	25	6	1
4	32	1,024	32	2	16	23	4	5

6.27 Consider the cache from Problem 6.12.

6.27.1 List all of the hex memory addresses that will hit in set 1.

0x4510 0x4511 0x4512 0x4513
0x3810 0x3811 0x3812 0x3813

6.27.2 List all of the hex memory addresses that will hit in set 6.

0x9160 0x9161 0x9162 0x9163

6.28 Consider the cache from Problem 6.12.

6.28.1 List all of the hex memory addresses that will hit in set 2.

N/A

6.28.2 List all of the hex memory addresses that will hit in set 4.

0xC740 0xC741 0xC742 0xC743
0x0540 0x0541 0x0542 0x0543

6.28.3 List all of the hex memory addresses that will hit in set 5.

0x7150 0x7151 0x7152 0x7153

6.28.4 List all of the hex memory addresses that will hit in set 7.

0xDE70 0xDE71 0xDE72 0xDE73

6.29 Consider the following cache.

6.29.1 Indicate which field each part of the memory address would be used to determine.

CO: 0-1 CI: 2-3 CT: 4-11

6.29.2 For each of the following memory addresses, indicate if it will be a cache hit or miss.

Read 0x834 Miss

Write 0x836 Hit

Read 0xFFD Hit C0

6.30 Consider the following cache.

6.30.1 What is the size of this cache in bytes?

$C = S * E * B = 8 * 4 * 4 = 128$

6.30.2 Indicate which field each bit of the memory address would be used to determine.

C0: 0-1 CI: 2-4 CT: 5-12

6.31 Consider the cache from Problem 6.30.

6.31.1 What is the address in bits of the 1-byte word at 0x071A?

0011100011010

6.31.2 What information is accessed at that address?

Block offset (CO) 0x2

Index (CI) 0x6

Cache tag (CT) 0x38

Cache hit? No

Cache byte returned -

6.32 Consider the cache from Problem 6.30.

6.32.1 What is the address in bits of the 1-byte word at 0x16E8?

1011011101000

6.32.2 What information is accessed at that address?

Block offset (CO) 0x0

Index (CI) 0x2

Cache tag (CT) 0xB7

Cache hit? No

Cache byte returned -

6.33 For the cache in Problem 6.30, list the eight memory addresses that will hit in set 2.

0x1788 0x1789 0x178A 0x178B
0x12C8 0x12C9 0x12CA 0x12CB

6.34 Consider the following matrix transpose routine.

6.34.1 For each row and col, indicate whether the access to src[row][col] and dst[row][col] is a hit or a miss.

This is a direct mapped cache with a 16-byte block size and two sets. It can hold two array rows at a time.

	D Col 0	D Col 1	D Col 2	D Col 3	S Col 0	S Col 1	S Col 2	S Col 3
Row 0	m	m	m	m	m	m	H	m
Row 1	m	m	m	m	m	H	m	H
Row 2	m	m	m	m	m	m	H	m
Row 3	m	m	m	m	m	H	m	H

6.35 Repeat Problem 6.34 for a cache with a total size of 128 data bytes.

This cache is also direct mapped with a 16-byte block size, but there are now eight sets, so it can hold eight array rows.

	D Col 0	D Col 1	D Col 2	D Col 3	S Col 0	S Col 1	S Col 2	S Col 3
Row 0	m	H	H	H	m	H	H	H
Row 1	m	H	H	H	m	H	H	H
Row 2	m	H	H	H	m	H	H	H
Row 3	m	H	H	H	m	H	H	H

6.36 Based on the C code, estimate the miss rates for the following cases:

6.36.1 Cache is 512 bytes, direct-mapped, with 16-byte cache blocks

100% Elements x[0][i] and x[1][i] map to exactly the same place, so each access overwrites the previous one.

6.36.2 Cache size is doubled to 1024 bytes

25% Now, x[0] and x[1] map to different sets of sets.

6.36.3 Cache is 512 bytes, two-way set associative using an LRU replacement, with 16-byte cache blocks

25% Elements x[0][i] and x[1][i] still map to the same set, but there is now room for both of them.

6.36.4 For case 3, will a larger cache size help to reduce the miss rate? Why or why not?

No. Misses only occur when a block is read for the first time.

6.36.5 For case 3, will a larger block size help to reduce the miss rate? Why or why not?

Yes. The only miss occurs when a block is first loaded, so if twice as many elements are loaded at a time, the miss rate will be cut in half.

6.37 Fill in the miss rates for each sum function for each of the possible array sizes.

sumB has a terrible miss rate for N=64 because every sixteenth row uses the same address. This is a consequence of the size of each row being a power of two. Reading two sequential elements at a time cuts the miss rate in half, but it still doesn't perform nearly as well as the array where N=60.

Function	N=64	N=60
sumA	25%	25%
sumB	100%	25%
sumC	50%	25%

6.38 Evaluate the cache performance of the given code for a 1,024-byte direct mapped data cache with 16-byte blocks.

What is the total number of writes? 1,024

What is the total number of writes that hit in the cache? 768

What is the hit rate? 75%

6.39 Evaluate the cache performance of the given code.

What is the total number of writes? 1,024

What is the total number of writes that hit in the cache? 768

What is the hit rate? 75%

Note: This problem is intended to demonstrate that arrays should be traversed row by row instead of column by column, but the given cache is big enough to hold the entire array, so there is no penalty.

6.40 Evaluate the cache performance of the given code.

What is the total number of writes? 1,024

What is the total number of writes that hit in the cache? 768

What is the hit rate? 75%

Note: This problem is intended to demonstrate that good code should have good temporal locality, accessing all parts of square[i][j] at the same time, instead of spreading them out. Because of the way the problem was written, though, the cache is big enough to hold the entire array, so there is no penalty.

6.41 What percentage of the writes in the given code will hit in the cache?

7/8

6.42 What percentage of writes in the given code will hit in the cache?

7/8

6.43 What percentage of writes in the given code will hit in the cache?

1/2

6.44 Use the mountain program to estimate the sizes of the caches on your system.

The clearest part of this data is that the L1 cache holds at least 32 kilobytes of data, since the last two rows indicate much faster run times. The L2 cache size is less clear. It must hold at least 128 kilobytes, and less than 512 kilobytes, but the run times for 256 kilobytes is in the middle. The gap between 4 megabytes and 8 megabytes is significant, so L3 must have a capacity of at least 4 megabytes. Despite the ambiguity, I conclude the following: L1-32k, L2-256k, L3-4m.

My computer is a MacBook Pro (15-inch, 2017) with a 2.8 GHz Quad-Core Intel Core i7, whose specifications indicate a 32k L1 cache, a 256k L2 cache, and a 6m L3 cache. Cool!

	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13	s14	s15
128m	5268	3169	2276	1713	1348	1198	1001	879	832	641	460	524	257	415	465
64m	4646	2837	2244	1685	1375	1166	1016	831	694	475	585	411	414	495	409
32m	4745	2898	2010	1439	1185	1024	613	392	366	481	496	581	269	256	468
16m	4870	2830	2100	1619	1085	999	899	949	928	910	874	794	843	834	810
8m	6352	4028	3332	2615	2085	1150	1340	1311	1326	1257	1314	1202	1225	1306	1278
4m	10026	6936	5707	4698	3732	3278	2873	2580	2545	2442	2319	2192	2125	2060	2014
2m	10545	7596	6198	5137	4286	3678	3217	2846	2684	2525	2372	2231	2141	2067	2031
1024k	10529	7505	6160	5124	4293	3690	3236	2863	2496	2703	2605	2523	2469	2383	2350
512k	10377	7858	6511	5283	4410	3801	3337	2937	2831	2744	2656	2593	2582	2562	2568
256k	11442	8944	8276	7511	6659	5918	5117	4623	4612	4601	4799	5211	4985	4755	5429
128k	12050	9763	9492	9537	9011	8154	7291	7370	6438	6725	6579	6391	6467	6474	6286
64k	12087	9700	9494	8933	7924	6943	5800	5669	5253	5535	5343	5038	5515	5514	9796
32k	13991	13653	13143	13024	13002	12583	12158	11736	12423	12178	11817	13382	12000	11584	13236
16k	13861	13452	13065	12603	12133	12757	10935	13299	12727	13000	12306	11870	12990	12062	12000

6.45 Optimize the following transpose routine to run as fast as possible.

My version divides the array into blocks equal in size to cache blocks. Instead of transposing one row at a time, it transposes one block at a time. This has the effect of only loading each chunk of each array into the cache once, increasing the hit rate for the destination array significantly.

```
void transpose(int *dst, int *src, int dim) {
int block_size = 64;
int ints_per_block = block_size/sizeof(int);
int full_blocks = dim / ints_per_block;

int two_dim = dim*2;
int three_dim = dim*3;

// i and j traverse the array by blocks
for (int i = 0; i < full_blocks; i++) {
        int i_big_skip = i * ints_per_block * dim;
        int i_little_skip = i * ints_per_block;

for (int j = 0; j < full_blocks; j++) {
        int j_big_skip = j * ints_per_block * dim;
        int j_little_skip = j * ints_per_block;

// a and b traverse each block, stride 4
for (int a = 0; a < ints_per_block; a += 1) {
        int a_skip = a * dim;

for (int b = 0; b < ints_per_block-3; b += 4) {
        int b_skip = b * dim;

        int source = j_big_skip+i_little_skip+a_skip+b;
        int dest = i_big_skip+j_little_skip+b_skip+a;
        dst[dest] = src[source];
        dst[dest + dim] = src[source+1];
        dst[dest + two_dim] = src[source+2];
        dst[dest + three_dim] = src[source+3];
        }
}}}

// In case dim isn't a multiple of the block size,
// traverse remaining
int already_traversed = ints_per_block * full_blocks;

// Traverse ends of already (mostly) traversed rows
```

```
for (int i = already_traversed; i < dim; i++) {
for (int j = 0; j < already_traversed; j++) {
        dst[j*dim + i] = src[i*dim + j];
        }
}

// Traverse ends of already (mostly) traversed columns
for (int j = already_traversed; j < dim; j++) {
for (int i = 0; i < already_traversed; i++) {
        dst[j*dim + i] = src[i*dim + j];
        }
}

// Traverse block at the bottom right
for (int i = already_traversed; i < dim; i++) {
for (int j = already_traversed; j < dim; j++) {
        dst[j*dim + i] = src[i*dim + j];
        }
}}
```

6.46 Optimize the following routine, which converts a directed graph to an undirected graph, represented by an adjacency matrix, to run as fast as possible.

The case where the array's dimensions aren't a multiple of the block size is ignored because the benefit to tedium ratio is just too low.

```
void col_convert(int *G, int dim) {
        int block_size = 64;
        int ints_per_block = block_size/sizeof(int);
        int full_blocks = dim / ints_per_block;

        // First, traverse the diagonal
        for (int i = 0; i < full_blocks; i++) {
                int block_offset = i * ints_per_block;
                int top_left_corner =
                block_offset * dim + block_offset;

        for (int a = 0; a < ints_per_block-1; a++) {
                int a_skip = a * dim;

        for (int b = a+1; b < ints_per_block; b++) {
                int b_skip = b * dim;

                int top = top_left_corner + a_skip + b;
```

```
                int bottom = top_left_corner + b_skip + a;

                int either = G[top] || G[bottom];
                G[top] = either;
                G[bottom] = either;
                }
        }}

        // i and j traverse the blocks above the diagonal
        for (int i = 0; i < full_blocks-1; i++) {
                int i_little_skip = i * ints_per_block;
                int i_big_skip = i_little_skip * dim;

        for (int j = i+1; j < full_blocks; j++) {
                int j_little_skip = j * ints_per_block;
                int j_big_skip = j_little_skip * dim;

        // a and b traverse each block, stride 2
        for (int a = 0; a < ints_per_block; a++) {
                int a_skip = a * dim;

        for (int b = 0; b < ints_per_block - 1; b += 2) {
                int b_skip = b * dim;

                int top_right =
                i_big_skip + j_little_skip + b_skip + a;
                int bottom_left =
                j_big_skip + i_little_skip + a_skip + b;

                int either0 =
                G[top_right] || G[bottom_left];
                int either1 =
                G[top_right+dim] || G[bottom_left+1];

                G[top_right] = either0;
                G[top_right+dim] = either1;
                G[bottom_left] = either0;
                G[bottom_left+1] = either1;
                }
        }
}}}
```