Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 6: The Memory Hierarchy May 5, 2024

Practice Problems

Exercise 6.1. In the following, let r be the number of rows in a DRAM array, c the number of columns, b_r the number of bits needed to address the rows, and b_c the number of bits needed to address the columns. For each of the following DRAMs, determine the power-of-2 array dimensions that minimize $\max(b_r, b_c)$, the maximum number of bits needed to address the rows or columns of the array.

Organization	r	c	b_r	b_c	$\max(b_r, b_c)$
16×1	____	____	____	____	____
16×4	____	____	____	____	____
128×8	____	____	____	____	____
512×4	____	____	____	____	____
1024×4	____	____	____	____	____

Solution: The notation 16×1 means $d = 16$ supercells, each of width $w = 1$ bit. The number of rows r and columns c must satisfy $rc = d = 16$. Since 16 is a perfect square, the minimum is given when we pick the square root, so $r = c = 4$. Meanwhile, $d = 128$ is not a perfect square, and we must pick either $r = 8$ and $c = 16$, or $r = 16$ and $c = 8$. For $d = 512$, we can pick $r = 32$ and $d = 16$, and for $d = 1024$, we can pick $r = d = 32$.

Organization	r	c	b_r	b_c	$\max(b_r, b_c)$
16×1	4	4	2	2	2
16×4	4	4	2	2	2
128×8	16	8	4	3	4
512×4	32	16	5	4	5
1024×4	32	32	5	5	5

Exercise 6.2. What is the capacity of a disk with 2 platters, 10,000 cylinders, an average of 400 sectors per track, and 512 bytes per sector?

Solution: According Section 6.12, page 592, the capacity of a rotating disk is given by

$$\text{Capacity} = \frac{\text{\# bytes}}{\text{sector}} \times \frac{\text{average \# sectors}}{\text{track}} \times \frac{\text{average \# tracks}}{\text{surface}} \times \frac{\text{\# surfaces}}{\text{platter}} \times \frac{\text{\# platters}}{\text{disk}}$$

Recall that a cylinder is the collection of tracks on all the surfaces that are equidistant from the center of the spindle. Since the disk has 10,000 cylinders, this means there's an average of 10,000 tracks per surface. Therefore the capacity is:

$$\begin{aligned} \text{Capacity} &= \frac{512 \text{ bytes}}{\text{sector}} \times \frac{400 \text{ sectors}}{\text{track}} \times \frac{10,000 \text{ tracks}}{\text{surface}} \times \frac{2 \text{ surfaces}}{\text{platter}} \times \frac{2 \text{ platters}}{\text{disk}} && (1) \\ &= 8.192 \times 10^9 \text{ bytes} && (2) \\ &= 8.192 \text{ GB} && (3) \end{aligned}$$

Exercise 6.3. Estimate the average time (in ms) to access a sector on the following disk:

Parameter	Value
Rotational Rate	15,000 RPM
$T_{\text{avg seek}}$	8 ms
Average number of sectors/track	500

Solution: The average time to access a sector on a disk is the sum of the seek time, the rotational latency, and the transfer time. The seek time is how long it takes the actuator arm on a hard disk to move to the track that contains the target sector. The rotational latency is how long the drive waits for the first bit of the sector to pass under the head of the actuator arm. The transfer time is how long it takes to read the contents of the sector.

The average rotational latency is given by half of the maximum rotational latency, which in turn is given by:

$$T_{\text{max rotation}} = \frac{1}{\text{RPM}} \times \frac{60 \text{ secs}}{\text{min}}$$

Therefore,

$$T_{\text{avg rotation}} = \frac{1}{2} \times \frac{1 \text{ minute}}{15000 \text{ rotations}} \times \frac{60 \text{ secs}}{\text{min}} \times \frac{1000 \text{ ms}}{\text{second}} = 2 \text{ ms}$$

Meanwhile, the average transfer time is given by

$$\begin{aligned} T_{\text{avg transfer}} &= \frac{1}{\text{RPM}} \times \frac{1}{\text{average \# sectors/track}} \times \frac{60 \text{ secs}}{\text{min}} \\ &= \frac{\text{minute}}{15,000 \text{ rotations}} \times \frac{1}{500 \text{ sectors/track}} \times \frac{60 \text{ secs}}{\text{min}} \times \frac{1000 \text{ ms}}{\text{second}} \\ &= 0.008 \text{ ms} \end{aligned}$$

Therefore we have, the average access time is about $T_{\text{access}} = 8 \text{ ms} + 2 \text{ ms} + 0.008 \text{ ms} = 10.008 \text{ ms}$.

Exercise 6.4. Suppose that a 1 MB file consisting of 512-byte logical blocks is stored on a disk drive with the following characteristics:

Parameter	Value
Rotational rate	10,000 RPM
$T_{\text{avg seek}}$	5 ms
Average number of sectors/track	1,000
Surfaces	4
Sector size	512 bytes

For each case below, suppose that a program reads the logical blocks of the file sequentially, one after the other, and that the time to position the head over the first block is $T_{\text{avg seek}} + T_{\text{avg rotation}}$.

(a) *Best case*: Estimate the optimal time (in ms) required to read the file given the best possible mapping of logical blocks to disk sectors (i.e., sequential).

(b) *Random case*: Estimate the time (in ms) required to read the file if blocks are mapped randomly to disk sectors.

Solution:

(a) 1 MB is equivalent to 1024 KB, so with a logical block size of 512 bytes, it will take up about 2000 logical blocks on disk. Since the size of a block equals the size of a sector, and there are 1000 tracks per sector on average for the given disk, this means that a sector can hold about 1000 blocks. In the best case, the data for the file is mapped onto 2 tracks on two surfaces on the same platter with the increasing order of the logical blocks matching that of the sector. After the initial delay of $T_{\text{avg seek}} + T_{\text{avg rotation}}$ to place the head on the first block, it will take about $T_{\text{avg transfer}}$ to transfer all bytes in the sector, which is about $\frac{1}{1000} T_{\text{max rotation}}$.These latter two steps are performed 1000 times per track. Since the two surfaces are on the same platter, the head is already in place to start transferring the data from the second surface. Therefore, the optimal time would be about:

$$\begin{aligned} T_{\text{access, optimal}} &= T_{\text{avg seek}} + T_{\text{avg rotation}} + 2 \cdot 1000 \cdot T_{\text{avg transfer}} \\ &= T_{\text{avg seek}} + T_{\text{avg rotation}} + 2000 \cdot T_{\text{avg transfer}} \end{aligned}$$

The maximum rotational latency for the given disk is about

$$T_{\text{maximum rotation}} = \frac{1 \text{ minute}}{10000 \text{ rotations}} \times \frac{60 \text{ secs}}{\text{min}} \times \frac{1000 \text{ ms}}{\text{second}} = 6 \text{ ms}$$

The average transfer latency is about

$$\begin{aligned} T_{\text{avg transfer}} &= \frac{\text{minute}}{10,000 \text{ rotations}} \times \frac{1}{1,000 \text{ sectors/track}} \times \frac{60 \text{ secs}}{\text{min}} \times \frac{1000 \text{ ms}}{\text{second}} \\ &= 0.006 \text{ ms} \end{aligned}$$

Altogether, the optimal access time is about

$$\begin{aligned} T_{\text{access, optimal}} &= 5 \text{ ms} + \frac{1}{2} \cdot 6 \text{ ms} + 2000 \cdot 0.006 \text{ ms} \\ &= 20 \text{ ms} \end{aligned}$$

(b) For a random case, we may not have contiguous blocks for the file. The 2000 blocks for the file may be split evenly among all 4 surfaces, meaning there is about 500 blocks per surface. If each block is on a different track, it may take about

$$\begin{aligned} T_{\text{access, avg}} &= 2000 \times (T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}) \\ &= 2000 \times (5 + \text{ ms} + 3 \text{ ms} + 0.006 \text{ ms}) \\ &= 16012 \text{ ms} \\ &\approx 16 \text{ seconds} \end{aligned}$$

Exercise 6.5. As we have seen, a potential drawback of SSDs is that the underlying flash memory can wear out. For example, for the SSD in Figure 6.14 (shown below), Intel guarantees about 12 petabytes (128×10^{15} bytes) of writes before the drive wears out:

Reads		Writes	
Sequential read throughput	550 MB/s	Sequential write throughput	470 MB/s
Random read throughput (IOPS)	89,000 IOPS	Random write throughput (IOPS)	74,000 IOPS
Random read throughput (MB/s)	365 MB/s	Random write throughput (MB/s)	303 MB/s
Avg. sequential read access time	50 μs	Avg. sequential write access time	60 μs

Given this assumption, estimate the lifetime (in years) of this SSD for the following workloads:

(a) *Worst case for sequential writes*: The SSD is written to continuously at a rate of 470 MB/s (the average sequential write throughput of the device).

(b) *Worst case for random writes*: The SSD is written continuously at a rate of 303 MB/s (the average random write throughput of the device).

(c) *Average case*: The SSD is written to at a rate of 20 GB/day (the average daily write rate assumed by some computer manufacturers in their mobile computer workload simulations).

Solution:

(a) It would take

$$\begin{aligned} T_{\text{worst seq. writes}} &= 128 \times 10^{15} \text{ bytes} \cdot \frac{1 \text{ MB}}{10^6 \text{ bytes}} \cdot \frac{1 \text{ second}}{470 \text{ MB}} \cdot \frac{1 \text{ day}}{86400 \text{ seconds}} \cdot \frac{1 \text{ year}}{365 \text{ days}} \\ &\approx 8.63 \text{ years} \end{aligned}$$

(b)

$$\begin{aligned} T_{\text{worst ran. writes}} &= 128 \times 10^{15} \text{ bytes} \cdot \frac{1 \text{ MB}}{10^6 \text{ bytes}} \cdot \frac{1 \text{ second}}{303 \text{ MB}} \cdot \frac{1 \text{ day}}{86400 \text{ seconds}} \cdot \frac{1 \text{ year}}{365 \text{ days}} \\ &\approx 13.4 \text{ years} \end{aligned}$$

(c)

$$\begin{aligned} T_{\text{avg}} &= 128 \times 10^{15} \text{ bytes} \cdot \frac{1 \text{ GB}}{10^9 \text{ bytes}} \cdot \frac{1 \text{ day}}{20 \text{ GB}} \cdot \frac{1 \text{ year}}{365 \text{ days}} \\ &\approx 17534 \text{ years} \end{aligned}$$

Exercise 6.6. Using the data from the years 2005 to 2015 in Figure 6.15(c) on page 603, estimate the year when you will be able to buy a petabyte (10^{15} bytes) of rotating disk storage for $500 bytes. Assume actual dollars (no inflation).

Solution: Expressed in dollars per GB, the cost would be

$$\frac{\$500}{10^{15}\text{ bytes}} \cdot \frac{10^9\text{ byte}}{1\text{ GB}} = \frac{\$0.0005}{\text{GB}}$$

According to the table, the price for rotating disk storage in dollars per GB for the years 2005, 2010, and 2015 were \$5/GB, \$0.3/GB, and \$0.03/GB, indicating a decrease by a factor between 10 and 16.67. It would therefore take at least 5 years but no more than 10 years, so around 2025.

Exercise 6.7. Permute the loops in the following function so that it scans the three-dimensional array a with a stride-1 reference pattern.

```
int sumarray3d(int a[N][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

Solution: As it stands, the function has stride-N pattern. We simply ought to change the indexing scheme so that the outer loop indices appears first:

```
int sumarray3d(int a[N][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                sum += a[i][j][k];
            }
        }
    }
    return sum;
}
```

Exercise 6.8. The three functions in Figure 6.20 perform the same operation with varying degrees of spatial locality. Rank-order the functions with respect to the spatial locality enjoyed by each. Explain how you arrived at your ranking.

```
/* An array of structs */
#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;
    }
}
```

Solution: Function `clear1` has the best spatial locality. With respect to the `p` variable it has a stride-1 reference pattern. Furthermore because the the elements of the `vel` field appear contiguously followed by those of the `acc` field, its access pattern with respect to

`p[i].vel` and `p[i].acc` is also good. Within the `i` loop it has good temporal locality with respect to `p[i]` but poor with respect to `p[i].vec` and `p[i].acc`. The `clear2` function is the next best because of its spatial locality with respect to `p`, though it has poorer spatial locality with respect to `p[i].vel` and `p[i].acc` because it does not access their elements contiguously; its temporal locality is the same as `clear1`.

Overall, the ranking from best to worst is `clear1`, followed by `clear2`, and finally `clear3`.

Exercise 6.9. The following table gives the parameters for a number of different caches. For each cache, determine the number of cache sets (S), tag bits (t), set insex bits (s), and block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	1	_____	_____	_____	_____
2.	32	1024	8	4	_____	_____	_____	_____
3.	32	1024	32	32	_____	_____	_____	_____

Solution: The size (capacity) of a cache, C, is given by $C = S \times E \times B$. The number of addresses is 2^m, where m is the number of bits for a memory address. The number of cache sets is $S = 2^s$, the size of a data block is $B = 2^b$, where b bits represent the offset of a byte in a block, and the number of tag bits is $t = m - (b + s)$.

For example, for cache 1., $S = \frac{C}{BE} = \frac{1024}{4 \cdot 1} = 256 = 2^8$, which thus implies $s = 8$. Also $B = 4 = 2^2$, so $b = 2$. Finally, $t = 32 - (8 + 2) = 22$

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	1	256	22	8	2
2.	32	1024	8	4	32	24	5	3
3.	32	1024	32	32	1	27	0	5

Exercise 6.10. In the previous `dotprod` example, what fraction of the total references to `x` and `y` will be hits once we have padded array `x`?

Solution: There are 16 references, 8 for `x` and 8 for `y`. Of these, 2 are cold misses and 2 are. Therefore $\frac{3}{4}$ are hits.

Exercise 6.11. Imagine a hypothetical cache that uses the high-order s bits of an address as the set index. For such a cache, contiguous chunks are mapped to the same cache set.

(a) How many blocks are in each of these contiguous array chunks?

(b) Consider the following code that runs on a system with a cache of the form $(S, E, B, m) = (512, 1, 32, 32)$:

```
int array[4096];
for (i = 0; i < 4096; i++)
    sum += array[i];
```

What is the maximum number of array blocks that are stored in the cache at any point in time?

Solution: Ultimately I was unable to figure out this question because caches are challenging me for me to understand. The question also uses terms such as *chunks* and uses the term *blocks* to refer to *memory blocks* as well as *cache block*. Below I have rephrased the answer given at the end of the chapter in terms that make sense to me.

(a) Recall that if $S = 2^s$ is the number of cache sets, E is the number of lines in each cache set, $B = 2^b$ is the number of bytes in each, and t represents the number of tag bits that uniquely identify a block in each tag line, then $t = m - (s + b)$. The solution says that a contiguous array chunk consists of 2^t blocks. Therefore I think a contiguous array "chunk" refers to all of the bytes whose address belong to a given cache set.

(b) The cache in question has $S = 512 = 2^9$ cache sets and $B = 32 = 2^5$ bytes per block in any given line. This means $s = 9$ and $b = 5$, so $t = m - (s + b) = 18$. If we express the size of an array in terms of cache block units, then the first 2^{18} blocks of any array would map to the first cache set, and then the next 2^{18} blocks cache to the next. The array in questions has 4096 `int` values, each of which takes up 4 bytes. Altogether, the array takes up $4096 \cdot 4 = 2^{14}$ bytes. Since a block is $B = 2^5$ bytes, this means the array takes up $2^{14}/2^5 = 2^9 = 512$ blocks. This is less than $2^t = 2^{18}$, so all of these blocks will cache to the first cache set. This means that each time a new block ought to be loaded, the current one will be evicted. At any point in time, there is only one cache set, and hence one block since the current case is a direct-mapped cache ($E = 1$), with thus 1 block per line. That is, we have 1 block in use at any given time.

Exercise 6.12. The problems that follow will help reinforce your understanding of how caches work. Assume the following:

- The memory is byte-addressable
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is two-way associative ($E = 2$), with a 4-byte block size ($B = 4$) and eight sets ($S = 8$).

The contents of the cache are as follows, with all numbers given in hexadecimal notation.

	Line 0					
Set index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10
1	45	1	60	4F	E0	23
2	EB	0	—	—	—	—
3	06	0	—	—	—	—
4	C7	1	06	78	07	C5
5	71	1	0B	DE	18	4B
6	91	1	A0	B7	26	2D
7	46	0	—	—	—	—

	Line 1					
Set index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	00	0	—	—	—	—
1	38	1	00	BC	0B	37
2	0B	0	—	—	—	—
3	32	1	12	08	7B	AD
4	05	1	40	67	C2	3B
5	6E	0	—	—	—	—
6	F0	0	—	—	—	—
7	DE	1	12	C0	88	37

The following figure shows the format of an address (1 bit per box). Indicate (by labeling the diagram) the fields that would be useful in determining the following:

- CO. The cache block offset.
- CI. The cache set index.
- CT The cache tag.

12	11	10	9	8	7	6	5	4	3	2	1	0

Solution: Here $m = 13$, $S = 8 = 2^3$, and $B = 4 = 2^2$, so $s = 3$, $b = 2$, and $t = m-(s+b) = 8$. The cache block offset is given by bits 0 and 1, the two lowest order bits. The next three higher order bits, 2 through 4, are used for the cache set index. Finally, the cache tag is stored in bits 5 through 12:

CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO
12	11	10	9	8	7	6	5	4	3	2	1	0

Exercise 6.13. Suppose a program running on the machine in Problem 6.12 references the 1-byte word at address `0x0E34`. Indicate the cache entry accessed and the cache byte value returned in hexadecimal notation. Indicate whether a cache miss occurs. If there is a cache miss, enter "—" for "Cache byte returned."

(a) Address format (1 bit per box):

12	11	10	9	8	7	6	5	4	3	2	1	0

(b) Memory reference:

Parameter	Value
Cache block offset (CO)	0x_____
Cache set index (CI)	0x_____
Cache tag (CT)	0x_____
Cache hit? (Y/N)	_____
Cache byte returned	0x_____

Solution:

(a) The hexadecimal address `0x0E34` translates to binary address `0 1110 0011 0100`. We select the cache set with bits at positions 2, 3, and 4, which gives cache set index `101`, which is decimal 5 and hexadecimal `0x5`. Now the tag is formed by the upper 8 bits `0111 0001`, which is `0x71` in hexadecimal. Line 0 has a matching tag and its valid bit is set, so we have a cache hit. The offset bits are `00`, so we take the first word (byte), which is `0B`.

0	1	1	1	0	0	0	1	1	0	1	0	0
12	11	10	9	8	7	6	5	4	3	2	1	0

(b)

Parameter	Value
Cache block offset (CO)	`0x0`
Cache set index (CI)	`0x5`
Cache tag (CT)	`0x71`
Cache hit? (Y/N)	Y
Cache byte returned	`0B`

Exercise 6.14. Repeat Problem 6.13 for memory address `0x0DD5`.

Solution: Hexadecimal address `0x0DD5` is binary `0 1101 1101 0101`. The cache set index is given by binary `101`, or decimal 5, or hexadecimal `0x5`. The cache tag is given by binary `0110 1110`, which is hexadecimal `0x6E`. This matches the tag in line 1, for which the valid bit is not set and thus is a cache miss.

(a)

0	1	1	0	1	1	1	0	1	0	1	0	1
12	11	10	9	8	7	6	5	4	3	2	1	0

(b)

Parameter	Value
Cache block offset (CO)	`0x1`
Cache set index (CI)	`0x5`
Cache tag (CT)	`0x6E`
Cache hit? (Y/N)	N
Cache byte returned	—

Exercise 6.15. Repeat Problem 6.13 for memory address `0x1FE4`.

Solution: Hexadecimal `0x1FE4` is binary `1 1111 1110 0100`. The cache set index is `001`, or decimal 1, or hexadecimal `0x1`. The cache tag is given by `1111 1111`, or hexadecimal `0xFF`. This, too, is a cache miss, because no tags match in either line of set 1.

(a)

1	1	1	1	1	1	1	1	0	0	1	0	0
12	11	10	9	8	7	6	5	4	3	2	1	0

(b)

Parameter	Value
Cache block offset (CO)	`0x0`
Cache set index (CI)	`0x1`
Cache tag (CT)	`0xFF`
Cache hit? (Y/N)	N
Cache byte returned	—

Exercise 6.16. For the cache in Problem 6.12, list all of the hexadecimal addresses that will hit in set 3.

Solution: Set 3 has tag `0x06` in line 0 and tag `0x32` in line 1. However, only the latter has its valid bit set. Therefore, we seek all hexadecimal numbers with `0x32` as the upper digits. Since bits 2 through 4 must be fixed at binary `011` to get set 3, equivalently hexadecimal `0x3` Therefore the only two degrees of freedom are the two block bits, meaning we have 4 possible addresses that will hit set 3:

`0011 0010 011 00` $\rightarrow$ `0 0110 0100 1100` $\rightarrow$ `0x64C`

`0011 0010 011 01` $\rightarrow$ `0 0110 0100 1101` $\rightarrow$ `0x64D`

`0011 0010 011 10` $\rightarrow$ `0 0110 0100 1110` $\rightarrow$ `0x64E`

`0011 0010 011 11` $\rightarrow$ `0 0110 0100 1111` $\rightarrow$ `0x64F`

Exercise 6.17. Transposing the rows and columns of a matrix is an important problem in signal processing and scientific computing applications. It is also interesting from a locality point of view because its reference pattern is both row-wise and column-wise. For example, consider the following transpose routine:

```
typedef int array[2][2];

void transpose1(array dst, array src)
{
    int i, j;

    for (i = 0; i < 2; i++) {
        for (j = 0; j < 2; j++) {
            dst[j][i] = src[i][j];
        }
    }
}
```

Assume this code runs on a machine with the following properties:

- `sizeof(int)` = 4.
- The `src` array starts at address 0 and the `dst` array starts at address 16 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, and write-allocate, with block size of 8 bytes.
- The cache has a total size of 16 data bytes and the cache is initially empty.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively.

(a) For each `row` and `col`, indicate whether the access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

`dst` array			`src` array		
	Col. 0	Col. 1		Col. 0	Col.1
Row 0	m	_____	Row 0	m	_____
Row 1	_____	_____	Row 1	_____	_____

(b) Repeat the problem for a cache with 32 data bytes.

Solution: (a) Since the capacity of the cache is $C = 16$ bytes, the number of lines is $E = 1$ (direct-mapped), and the block size is $B = 8$, the cache has $S = \frac{C}{EB} = 2$ cache sets. Each block can fit two contiguous elements.

If a tupe is given by (i, j), then the access order is $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$. When the program tries to read `src[0][0]`, it will encounter a miss because the cache is cold, and it will all `src[0][0]` and `src[0][1]` into the block of the first set. When we attempt to write its value into `dst[0][0]`, we have a miss because `dst[0][0]` is not currently loaded. The write-allocate approach loads the corresponding block from memory, which in turn evicts the block containing `src[0][0]` and `src[0][1]` because it caches to the same index set.

When we go to access `src[0][1]`, we have another miss due to a cache conflict from the previous write. When we go on to write `dst[1][0]`, it indexes to set 1, which is cold, so we have a miss as a result, and `dst[1][0]` and `dst[1][1]` are loaded onto it as a result.

On to `src[1][0]`, we have a cache (miss) conflict, causing the `dst` values there to be evicted. When we try to store onto `dst[0][1]`, we have a cache (miss) conflict because set 0 already has `src[0][0]` and `src[0][1]` loaded. Finally, for `src[1][1]`, we have a cache hit due to the previous read, but then writing to `dst[1][1]` is a cache (miss) conflict.

`dst` array			`src` array		
	Col. 0	Col. 1		Col. 0	Col.1
Row 0	m	m	Row 0	m	m
Row 1	m	m	Row 1	m	h

(b) With 32 data bytes, we have $S = \frac{32}{8 \cdot 1} = 4$ index sets.

First, `src[0][0]`. Now `src[0][0]` and `src[0][1]` are loaded onto the block in the first cache set. Next, since the cache size is now 32 bytes and `dst` is at a 16 byte offset from `src`, it follows that `dst[0][0]` maps to the third cache index set. It is a miss because the cache is cold.

Next, `src[0][1]` maps to the first cache index set, so reading this gives a hit as a result of the previous read. Writing to `dst[1][0]` is a miss due to the cache being cold.

`src[1][0]` maps to the second cache index set, which is cold so it is a miss and it loads `src[1][0]` and `src[1][1]` onto it. The write to `dst[0][1]` maps to the third cache index set, which is a hit as a result of the very first write to `dst[0][0]`.

Finally, reading `src[1][1]` is a hit as a result of the previous read, and writing to `dst[1][1]` is a hit as a result of the second write.

`dst` array			`src` array		
	Col. 0	Col. 1		Col. 0	Col.1
Row 0	m	h	Row 0	m	h
Row 1	m	h	Row 1	m	h

Exercise 6.18. The heart of the recent hit game *SimAquarium* is a tight loop that calculates the average position of 256 algae. You are evaluating its cache performance on a machine with a 1024-byte direct-mapped data cache with 16-byte blocks ($B = 16$). You are given the following definitions:

```
struct algae_position {
    int x;
    int y;
}

struct algae_position grid[16][16];
int total_x = 0, total_y = 0;
int i, j;
```

You should assume the following:

- `sizeof(int)` = 4.
- `grid` begins at memory address 0.
- The cache is initially empty.

- The only memory accesses are to the entries of the grid `grid`. Variable `i`, `j`, `total_x`, and `total_y` are stored in registers.

Determine the cache performance for the following code:

```
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        total_x += grid[i][j].x;
    }
}

for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        total_y += grid[i][j].y;
    }
}
```

(a) What is the total number of reads?

(b) What is the total number of reads that miss in the cache?

(c) What is the miss rate?

Solution:

(a) Each loop does $16^2 = 256$ memory reads, for a total of 512 reads.

(b) Since the machine has a $C = 1024$-byte cache that is direct-mapped ($E = 1$) with $B = 16$ blocks, there are $S = \frac{1024}{16 \cdot 1} = 64$ cache sets. The code uses a stride-1 reference pattern, resulting in an average of $\min(1, \text{word size} \times k/B)$ misses per loop iteration. Letting the word size be 4 since we have an `int`, this suggests about $\frac{4 \cdot 1}{16} = \frac{1}{4}$ misses per iteration. Therefore about $\frac{1}{4}$ iterations in the first loop are misses, or 64 reads.

Recall that the fields of a `struct` are packed in sequence, possibly with pad bits for alignment. In this case no pad bits are necessary to get the 4-byte alignment demanded of the `int` values. This means, for example, that when `grid[0][0]` is loaded into the first cache set, the 16 bytes in that block have `grid[0][0].x`, `grid[0][0].y`, `grid[0][1].x`, and `grid[0][0].y`. Since the first loop only accesses the `x` field, we skip `y`, meaning this is a 2-stride access pattern. For such a pattern, the formula $\min(1, \text{word size} \times k/B)$ for the misses per loop iteration gives $4 \times 2/16 = \frac{1}{2}$, since `int` is a 4-byte word. That is, about half of the iterations will be misses, for a total of 128 misses for the first loop. Note that since there are 256 structs each of 8 bytes, they take up a total of 2048 bytes, so after the first 128 entries of the array is read, continuing to read the second will evict the previously loaded blocks.

When the second loop begins, we have cache misses due to conflicts instead of cache misses due to cold cache. A similar analysis yields 128 misses.

Altogether there is a total of 256 misses.

(c) The miss rate is $\frac{1}{2}$.

Exercise 6.19. Given the assumptions of Practice Problem 6.18, determine the cache performance of the following code:

```
for (int i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        total_x += grid[j][i].x;
        total_x += grid[j][i].y;
    }
}
```

(a) What is the total number of reads?

(b) What is the total number of reads that miss in the cache?

(c) What is the miss rate?

(d) What would the miss rate be if the cache were twice as big?

Solution:

(a) There are 512 reads; 256 iterations, 2 reads per iteration.

(b) This time the inverting of the indexes make it so that we access the elements in column-major order, so we access every 16th element of the array. Thus we have a miss every iteration, for a total of 256 reads. The justification is similar to that in Practice Problem 6.18.

In particular, note that the first 8 rows the array contains have of the elements, and that the cache can only fit half of the 2048-byte array. When $i = 0$, and $j = 0$ through $j = 7$, the accesses of type `grid[j][i]` are misses due to the cache being cold and the fact that the stride-16 pattern means we skip the cache blocks each time. When $j = 8$, we have skipped $16 \times 8 = 128$ array entries, each taking up 8 bytes, for a total of 1024 byte addresses. When we request `grid[8][0]`, we end up mapping to the same cache set as `grid[0][0]`, so we have a cache miss due to a conflict. Then when the first inner j loop ends and we move on to $i = 1$, the access `grid[0][1]` is a cache conflict because of the eviction that resulted from `grid[8][0]`. The pattern holds for all iterations, accessing field `x` results in a cache conflict thereafter, but accessing field `y` is a cache hit. Thus half of the reads are misses, or 256 reads.

(c) The miss rate is $\frac{1}{2}$.

(d) If the cache were twice as big, then the 2048-byte array, composed of 256 entries with 8-byte structs, will fit entirely in the cache. As a result, when $i = 0$ and $j = 8$ as described earlier, the memory reference `grid[8][0]` will not cache to the first cache index set, meaning that this cache hit will be due to a cold cache rather than a conflict. The significance is that the bytes loaded into the block in the first cache index set as

a result of the read to `grid[0][0]` does not cause an eviction. Therefore, after the first inner `j` loop and we switch to `i=1`, the read `grid[0][1]` is a cache hit and not a conflict. Similarly, `grid[1][1]` is a cache hit, and so is `grid[j][1]` for all j. Then when $i = 2$, we go back to having cold caches due to the request for field `x` and hits for `y` as a result of that read, but for $i = 3$, we have cache hits for all accesses.

The miss rate has halved, so altogether the miss rate is $\frac{1}{4}$.

Exercise 6.20. Given the assumptions of Practice Problem 6.18, determine the cache performance of the following code:

```
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; i++) {
        total_x += grid[i][j].x;
        total_y += grid[i][j].y
    }
}
```

(a) What is the total number of reads?

(b) What is the total number of reads that miss in the cache?

(c) What is the miss rate?

(d) What is the miss rate if the cache were twice as big?

Solution:

(a) Because the array takes up 2048 bytes, the iterations beginning at $i = 8$ cause evictions, and the misses due to that latter half of accesses is the same as the misses due to the first half.

For the access to `grid[0][0].x`, the cache is cold so we have a miss, but then we have a hit for `grid[0][0].y`. For `grid[0][1].x` and `grid[0][1].y` we have hits. Therefore, we have 1 miss every 2 iterations (every 4 reads) during the first 16 iterations, for a total of 8 misses out of the 32 reads. The pattern repeats for the next 7 iterations, so we have a total of 64 misses out of the first 256 reads from the first half of the iterations. Combining this with the second half reads to 128 misses out of 512 reads.

(b) The miss rate is $\frac{1}{4}$.

(c) This time there are no evictions because the cache is large enough to fit the entire array. However, we still have cache misses due to cold caches. Thus, the miss rate remains at $\frac{1}{4}$.

Exercise 6.23. Estimate the average time (in ms) to access a sector on the following disk:

Parameter	Value
Rotational rate	15,000 RPM
$T_{\text{avg seek}}$	4 ms
Average number of sectors/track	800

Solution: The maximum rotational latency is

$$\begin{aligned} T_{\text{max rotation}} &= \frac{1}{15000 \text{ RPM}} \cdot \frac{60 \text{ secs}}{1 \text{ min}} \cdot \frac{1000\text{ms}}{1 \text{ sec}} \\ &= 4.0 \text{ ms} \end{aligned}$$

The average rotational latency is $T_{\text{avg rotation}} = \frac{1}{2} T_{\text{max rotation}} = 2$ ms. The average transfer latency is given by

$$\begin{aligned} T_{\text{avg transfer}} &= \frac{1}{\text{RPM}} \cdot \frac{1}{\text{average \# sectors/track}} \cdot \frac{60 \text{ secs}}{1\text{min}} \cdot \frac{1000 \text{ ms}}{1 \text{ sec}} \\ &= \frac{1}{15000 \text{ RPM}} \frac{1}{800} \cdot \frac{60 \text{ secs}}{1\text{min}} \cdot \frac{1000 \text{ ms}}{1 \text{ sec}} \\ &= 0.005 \text{ ms} \end{aligned}$$

Therefore, the access time is about 6.005 ms.

Exercise 6.24. Suppose that a 2 MB file consisting of 512-byte logical blocks is stored on a disk drive with the following characteristics:

Parameter	Value
Rotational rate	15,000 RPM
$T_{\text{avg seek}}$	4 ms
Average number of sectors per track	1,000
Surfaces	8
Sectors size	512 bytes

For each case below, suppose that a program reads the logical blocks of the file sequentially, one after the other, and that the time to position the head over the first block is $T_{\text{avg seek}} + T_{\text{avg rotation}}$.

(a) *Best case*: Estimate the optimal (in ms) required to read the file over all possible mappings of logical blocks to disk sectors.

(b) *Random case*: Estimate the time (in ms) required to read the file if blocks are mapped randomly to disk sectors.

Solution:

(a) This is essentially Practice Problem 6.4. Since 2 MB is 2048 KBs, this means that the file and we have about 1000 sectors per track with 512 bytes per sector, the we need about 4 tracks to store the file, corresponding to about 4000 logical blocks. The idea is that in the best case, the data mapped onto 4 tracks, where 2 surfaces share one platter and 2 surfaces share a different platter, in a contiguous way. After the initial delay of $T_{\text{avg seek}} + T_{\text{avg rotation}}$ to place the head on the first block, it will take about $T_{\text{avg transfer}}$ to transfer all bytes in the sector, which is $\frac{1}{1000} T_{\text{max rotation}}$. These latter two steps are performed 1000 times per track. Thus we find the maximum rotational latency:

$$\begin{aligned} T_{\text{max rotation}} &= \frac{1}{15,000 \text{ RPM}} \times \frac{60 \text{ secs}}{1 \text{ min}} \times \frac{1000 \text{ ms}}{1 \text{ sec}} \\ &= 4 \text{ ms} \end{aligned}$$

Thus the optional access time is

$$\begin{aligned} T_{\text{access, optimal}} &= T_{\text{avg seek}} + \frac{1}{2} T_{\text{max rotation}} + 2 \cdot T_{\text{max rotation}} \\ &= 4 \text{ ms} + 2 \text{ ms} + 2 \cdot (4 \text{ ms}) \\ &= 14 \text{ ms} \end{aligned}$$

(b) For the random case, we may not have contiguous blocks; the 4000 logical blocks may be split evenly among all 8 surfaces, meaning there's about 500 blocks per surface. If each block is on a different track, it may take about:

$$\begin{aligned} T_{\text{access, avg}} &= 4000 \times (T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}) \\ &= 4000 \times (4 + \text{ ms} + 2 \text{ ms} + 0.002 \text{ ms}) \\ &= 24008 \text{ ms} \\ &\approx 24 \text{ seconds} \end{aligned}$$

Exercise 6.34. Consider the following matrix transpose routine:

```
typedef int array[4][4];

void transpose2(array dst, array src)
{
    int i, j;
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            dst[j][i] = src[i][j];
        }
    }
}
```

Assume this code runs on a machine with the following properties:

- `sizeof(int)` = 4
- The `src` array starts at address 0 and the `dst` array starts at address 64 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, write-allocate, with a block size of 16 bytes.
- The cache has a total size of 32 bytes, and the cache is initially empty.
- For each `row` and `col`, indicate whether access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

`dst` array

	Col. 0	Col. 1	Col. 2	Col. 3
Row 0	m	_____	_____	_____
Row 1	_____	_____	_____	_____
Row 2	_____	_____	_____	_____
Row 3	_____	_____	_____	_____

`src` array

	Col. 0	Col. 1	Col. 2	Col. 3
Row 0	m	_____	_____	_____
Row 1	_____	_____	_____	_____
Row 2	_____	_____	_____	_____
Row 3	_____	_____	_____	_____

Solution: We are given that the cache in question is a direct-cache, so $E = 1$. The block size is 16 bytes, so $B = 16$. Lastly, the total cache size is 32 bytes, meaning $C = 32$ bytes. Thus, $S = \frac{C}{EB} = 2$, meaning there are two cache sets.

When the program starts, it reads `src[0][0]`; the cache is cold, so it's a miss. Consequently, it loads 4 `int` values onto the first block (first cache index set): `src[0][0]`, `src[0][1]`, `src[0][2]`, and `src[0][3]`. Then, in writing to `src[0][0]`, we deal with a cache conflict, causing the same indices to be loaded.

Next, we read `src[0][1]`, which is a cache conflict again, and loads the initial four entries again. To store onto `dst[1][0]`, we load onto the second block (the only block int he second index set); this is a cold cache, so it's a miss.

Next, we move on to `src[0][2]`, which a hit since the previous load for `src[0][1]` already loaded this value. When trying to store it on `dst[2][0]`, we have a cache conflict.

Next, we move on to `src[0][3]`, which is a cache conflict because of the existing values of `dst` from the previous iteration. Lastly, we have a cache conflict for `dst[3][0]`, because of the iteration where we wrote to `dst[1][0]` on the second cache index set.

Proceeding this way we can fill out the rest of the table.

`dst` array

	Col. 0	Col. 1	Col. 2	Col. 3
Row 0	m	m	m	m
Row 1	m	m	m	m
Row 2	m	m	m	m
Row 3	m	m	m	m6

`src` array

	Col. 0	Col. 1	Col. 2	Col. 3
Row 0	m	m	h	m
Row 1	m	h	m	h
Row 2	m	m	h	m
Row 3	m	h	m	h

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

Solution: Unlike the previous Practice Problem, the cache is 4 times as big, so the number of cache index sets S is $S = \frac{C}{EB} = \frac{128}{16} = 8$.

The program starts reading `src[0][0]`, which is a cache miss because the cache is cold. This causes loading `src[0][0]`, `src[0][1]`, `src[0][2]`, and `src[0][3]` into the buffer cache. When storing onto `dst[0][0]`, there's a cold cache again as we load onto the 5th cache index set.

Next, when we read `src[0][1]`, there's a hit due to the previous read. Writing to `dst[1][0]` is a miss due to a cold cache, causing its data loaded to the 6th index set.

Reading from `src[0][2]` is a hit due to the previous read to `src[0][1]`, and writing to `dst[2][0]` is a miss due to a cold cache when attempting to write to the 7th cache index set.

Reading from `src[0][3]` is a hit like before, and writing to `dst[3][0]` is a miss as data is loaded onto to the cold, 8th cache index set.

Next we start the `i = 1` iteration. Reading from `src[1][0]` is a miss due to the 2nd cache index set being cold. The write to `dst[0][1]` is a hit before of the initial write to `dst[0][0]`, which has not been overwritten.

In fact, the rest of the reads and writes are hits because all caches have been warmed, `src` only ever maps to the first four caches, and `dst` only ever maps to the latter four caches in an alternating way.

`dst` array

	Col. 0	Col. 1	Col. 2	Col. 3
Row 0	m	h	h	h
Row 1	m	h	h	h
Row 2	m	h	h	h
Row 3	m	h	h	h

`src` array

	Col. 0	Col. 1	Col. 2	Col. 3
Row 0	m	h	h	h
Row 1	m	h	h	h
Row 2	m	h	h	h
Row 3	m	h	h	h

Exercise 6.36. This problem tests your ability to predict the cache behavior of C code. You are given the following code to analyze:

```
int x[2][128];
int i;
int sum = 0;

for (i = 0; i < 128; i++) {
    sum += x[0][i] * x[1][i];
}
```

Assuming we execute this under the following conditions:

- `sizeof(int)` = 4.
- Array `x` begins at memory address `0x0` and is stored in a row-major order.
- In each case below, the cache is initially empty.
- The only memory accesses are to the entries of the array `x`. All other variables are stored in registers.

(a) Case 1: Assume the cache is 512 bytes, direct-mapped, with 16-byte cache blocks. What is the miss rate?

(b) Case 2: What is the miss rate if we double the cache size to 1,024 bytes?

(c) Case 3: Now assume the cache is 512 bytes, two-way set associative using an LRU replacement policy, with 16-byte cache blocks. What is the cache miss rate?

(d) For case 3, will a large cache help to reduce the miss rate? Why or why not?

(e) For case 3, will a large block size help to reduce the miss rate? Why or why not?

Solution:

(a) We are given $C = 512$, $E = 1$, $B = 16$, so $S = \frac{C}{EB} = \frac{512}{16} = 32$.

When `i = 0`, we read from `x[0][0]`, causing us to index into cache index 0 and missing because of a cold cache. Since `x` has 128 elements, each of 4 bytes, accessing `x[1][0]` causes a conflict with the data just loaded. Every subsequent write causes either a cache miss or a conflict. The miss rate is `100%`.

(b) This time $C = 1024$ and $S = 64$. This is large enough to fit every element into the cache. Writes to `src[0][i]` map to caches with indices 0 through `31`, and writes to `src[1][]i` maps to caches indices 32 through 63; this offset means we no longer have cache conflicts. Every miss is now due to a cold cache. In particular, the first of every four iterations is a cache miss due to a cold cache when reading from each of `x[0][i]` and `x[1][i]`, while the next three iterations all lead to hits. The miss rate is now $\frac{1}{4}$.

(c) Now $E = 2$, so $S = \frac{C}{EB} = \frac{512}{2 \cdot 16} = 16$. That is, we have half as many cache index sets, but twice as many caches lines per set.

Reading from `x[0][0]` is a cold cache miss onto cache index set 0. Reading from `x[1][0]` also caches to the set of index 0, which does not contain the value we want. However there is an empty line, so we write to it. Next, reading `x[0][1]` and `x[1][0]` are both hits, and the same holds for indices `i = 2` and `i = 3`. Reading `x[0][i]` and `x[1][i]` repeats the pattern but at cache index 1. This continues until the 64th iteration, because having only 16 sets with 16 byte-blocks means we will eventually wrap around to set index 0. When we go to load the elements associated with `x[0][64]`, both lines are full. Since we have an LRU policy, this causes the need to unload `x[0][0]`, `x[0][1]`, `x[0][2]`, and `x[0][3]`, to replace them with `x[0][64]` through `x[0][67]`. The same occurs when we read from `x[1][64]`. Thus, in the first 64 iterations, 2 in

every 8 reads are cold cache misses. Meanwhile, in the latter 64 iterations, 2 in every 8 reads are cache conflict misses. Overall, the miss rate is $\frac{1}{4}$.

(d) No, because we cannot avoid cold cache misses.

(e) Yes. There would be less cache index sets, but the pattern would be similar to (c), where first misses are due to cold caches, and then they are due to the LRU policy, with elements of `x[0][i]` not conflicting with elements of `x[1][i]`. For example, if the cache block size was 32 bytes instead of 16, then could load 8 elements per block, leading to each 8 iterations doing 16 reads of which only 2 are misses. The miss rate would be about $\frac{1}{8}$ in this case.

Exercise 6.37. This is another problem that tests your ability to analyze the cache behavior of C code. Assume we execute the three summation functions in Figure 6.47 under the following conditions:

- `sizeof(int)` = 4.
- The machine has a 4 KB direct-mapped cache with a 16-byte block size.
- Within the two loops, the code uses memory accesses only for the array data. The loop indices and the value `sum` are held in registers.
- Array `a` is stored at memory address `0x8000000`

Fill in the table for the approximate cache miss rate for the two cases $N = 64$ and $N = 60$.

Function	$N = 64$	$N = 60$
`sumA`	_____	_____
`sumB`	_____	_____
`sumC`	_____	_____

```
typedef int array_t[N][N];

int sumA(array_t a)
{
    int i, j;
    int sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) {
            sum += a[i][j];
        }
    return sum;
}

int sumB(array_t a)
{
    int i, j;
    int sum = 0;
```

```
    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++) {
            sum += a[i][j];
        }
    return sum;
}

int sumC(array_t a)
{
    int i, j;
    int sum = 0;
    for (j = 0; j < N; j+=2)
        for (i = 0; i < N; i+=2) {
            sum += (a[i][j] + a[i+1][j]
                    + a[i][j+1] + a[i+1][j+1]);
    }
    return sum;
}
```

Solution: The cache size is $C = 4096$ bytes, with $B = 16$ bytes, and $E = 1$. The array has 4096 elements, each taking 4 bytes.

The `sumA` procedure uses a stride-1 access pattern. The first iteration is a cold cache miss onto the cache set with index 0, and the next three are hits. The next follows the same pattern. Since the cache cannot hold the entire array, eventually when we reach `i=16`, `j=0`, we wrap back around to the cache set with index 0, which this time results in a cache conflict. However, we have hits for `a[16][1]`, `a[16][2]`, and `a[16][3]`. This pattern continues; the miss rate is $\frac{1}{4}$.

`sumB` differs from the first in that we have a stride-64 pattern because we loop column-wise. The first 16 inner loop iterations are are cold cache misses. On the 17th inner loop iteration, we have a cache conflict. In fact, every iteration thereafter leads to a cache conflict. The miss rate is 100%.

For `sumC`, we still have `i` as the inner loop. However, we access two elements per iteration and increment by 2 each time. The reads from `a[0][0]` and `a[1][0]` both cause cold cache misses, whereas `a[0][1]` and `a[1][1]` are cache hits. Moving forward to `i=2`, we now read `a[2][0]`, `a[3][0]`, `a[2][1]`, and `a[3][1]`. Once again, we have two cold caches misses and two cache hits. When we reach `i=16`, we wrap back to the cache set with index 0, which causes a cache conflict due to the elements already being loaded onto the contained memory block. The pattern continues until the inner `i` loop continues. Thus far, the miss rate is about $\frac{1}{2}$. Then `j` increases by 2, and we begin from `i=0` again, where we begin by reading `a[0][2]` in the inner loop. However, though it was once in the cache during the first iteration, it has now been overwritten by the read from `a[48][0]` when we had `j=0`. Other than this cache conflict, the patterns is just the as before. The miss rate using this approach is $\frac{1}{2}$.

Exercise 6.38. 3M decides to make Post-its by printing yellow squares on white pieces of paper. As part of the printing process, they need to set the CMYK (cyan, magenta, yellow,

black) value for every point in the square. 3M hires you to determine the efficiency of the following algorithms on a machine with a 2048-byte direct-mapped data cache with 32-byte blocks. You are given the following definitions:

```
struct point_color {
    int c;
    int m;
    int y;
    int k;
};

struct point_color square[16][16];
int i, j;
```

Assume the following:

- `size(int)` = 4
- `square` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `square`. Variables `i` and `j` are stored in registers.

Determine the cache performance of the following code:

```
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        square[i][j].c = 0;
        square[i][j].m = 0;
        square[i][j].y = 1;
        square[i][j].k = 0;
    }
}
```

(a) What is the total number of writes?

(b) What is the total number of writes that miss in the cache?

(c) What is the miss rate?

Solution:

(a) There are 256 iterations, and each makes 4 writes, so there's a total of 1024 writes.

(b) The access to `square[0][0].c` is cold cache miss, which causes fields `m`, `y`, and `k` of the same array element to be loaded onto the cache block for the first cache index set. It also causes all four fields of `square[0][1]` to be loaded. The result is a cache miss

and 7 cache hits. When `j=2`, we have a cache miss again, followed by 7 hits. This pattern continues until we've scanned through the first 2048 bytes of the array. The entire array takes up $16 \cdot 16 \cdot 4 \cdot 4$ bytes, which is twice as big as the cache size. Thus, in the latter half of this double loop, when `i=8`, we get a cache conflict followed by 7 hits. The cache miss rate is $\frac{1}{8}$, so 128 writes miss the cache.

(c) $\frac{1}{8}$.

Exercise 6.39. Given the assumptions in Problem 6.38, determine the cache performance of the following code:

```
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        square[j][i].c = 0;
        square[j][i].m = 0;
        square[j][i].y = 1;
        square[j][i].k = 0;
    }
}
```

(a) What is the total number of writes?

(b) What is the total number of writes that miss in the cache?

(c) What is the miss rate?

Solution:

(a) The are 1024 writes as before.

(b) The inner loop is still `j`, but the indices into `square` have been swapped. The read on field `c` of `square[0][0]` is a miss, while the read for fields `m`, `y`, and `k` are hits. For the next iteration, the same occurs, because the stride-256 access pattern makes it so we don't benefit from the previous iteration's load. By `j=0` we begin to wrap around to cache set index 0, leading to cache conflicts. Overall the miss rate is $\frac{1}{4}$, so 256 of the writes miss.

(c) $\frac{1}{4}$.

Exercise 6.40. Given the assumptions in Problem 6.38, determine the cache performance of the following code:

```
for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
        square[i][j].y = 1;
    }
}
for (i = 0; i < 16; i++) {
```

```
    for (j = 0; j < 16; j++) {
        square[i][j].c = 0;
        square[i][j].m = 0;
        square[i][j].k = 0;
    }
}
```

(a) What is the total number of writes?

(b) What is the total number of writes that miss in the cache?

(c) What is the miss rate?

Solution:

(a) 1024 writes.

(b) The first loop does 256 writes, of which half are misses. On the second loop, we have 768 writes, of which 1 out of every 6 is a miss, or 128. Overall there are 384 misses.

(c) The miss rate is $\frac{2}{3}$.

Exercise 6.41. You are writing a 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is 640×480 array of pixels. The machine you are working on has a 64 KB direct-mapped cache with 4-byte lines. The C structures you are using are as follows:

```
struct pixel {
    char r;
    char g;
    char b;
    char a;
};

struct pixel buffer[480][640];
int i, j;
char *cptr;
char *iptr;
```

Assume the following:

- `sizeof(char)` = 1 and `sizeof(int)` = 4.
- `buffer` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `buffer`. Variables `i`, `j`, `cptr`, and `iptr` are stored in registers.

What percentage of writes in the following code will miss in the cache?

```
for (j = 0; j < 640; j++) {
    for (i = 0; i < 480; i++) {
        buffer[i][j].r = 0;
        buffer[i][j].g = 0;
        buffer[i][j].b = 0;
        buffer[i][j].a = 0;
    }
}
```

Solution: Each iteration reads the 4 fields of `buffer[i][j]` into the 4-byte buffer cache. The read for field `r` is a miss, while the rest are hits. The miss rate is $\frac{1}{4}$.

Exercise 6.42. Given the assumptions in Problem 6.41, what percentage of writes in the following code will miss in the cache?

```
char *cptr = (char *) buffer;
for (; cptr < (((char *) buffer) + 640 * 480 * 4); cptr++)
    *cptr = 0;
```

Solution: This loop does the same number of writes as the one on Practice Problem 6.41, in the same order, so the miss rate is again $\frac{1}{4}$.

Exercise 6.43. Given the assumptions in Problem 6.41, what percentage of writes in the following code will miss in the cache?

```
int *iptr = (int *)buffer;
for (; iptr < ((int *)buffer + 640*480); iptr++)
    *iptr = 0;
```

Solution: This version does a quarter of the number of writes as the previous version; they all miss in the cache, so the miss rate is 100%.

Exercise 6.45. In this assignment, you will apply the concepts you learned in Chapter 5 and Chapter 6 to the problem of optimizing code for a memory-intensive application. Consider a procedure to copy an transpose the elements of an $N \times N$ matrix of type `int`. That is, for source matrix S and destination matrix D, we want to copy each element $s_{i,j}$ to $d_{j,i}$. This code can be written with a simple loop,

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

where the arguments to the procedure are pointers to the destination (`dst`) and source (`src`) matrices, as well as the matrix size N (`dim`). Your job is to devise a transpose routine that runs as fast as possible.

Solution: The problem with the current implementation is that even though `src` has good spatial locality, `dst` does not, it having a stride-`dim` access pattern. As a result, each time the inner j-loop advances, we lose any hope of taking advantage of the cached elements. By using loop unrolling, however, we get a better use of cache. This is similar to `sumC` in Practice Problem 6.37. My implementation below uses 4×1 loop unrolling:

```
void transpose_optimized(int *dst, int *src, int dim)
{
    int i, j, k, l;
    for (i = 0; i < dim; i += 4)
        for (j = 0; j < dim; j+= 4) {
            int k = j * dim + i;
            int l = i * dim + j;

            dst[k] = src[l];
            dst[k + dim] = src[l + 1];
            dst[k + 2 * dim] = src[l + 2];
            dst[k + 3 * dim] = src[l + 3];
        }
}
```