Impact of Block Size

Large blocks are a mixed blessing. On the one hand, larger blocks can help increase the hit rate by exploiting any spatial locality that might exist in a program. However, for a given cache size, larger blocks imply a smaller number of cache lines, which can hurt the hit rate in programs with more temporal locality than spatial locality. Larger blocks also have a negative impact on the miss penalty, since larger blocks cause larger transfer times. Modern systems such as the Core i7 compromise with cache blocks that contain 64 bytes.

Impact of Associativity

The issue here is the impact of the choice of the parameter E, the number of cache lines per set. The advantage of higher associativity (i.e., larger values of E) is that it decreases the vulnerability of the cache to thrashing due to conflict misses. However, higher associativity comes at a significant cost. Higher associativity is expensive to implement and hard to make fast. It requires more tag bits per line, additional LRU state bits per line, and additional control logic. Higher associativity can increase hit time, because of the increased complexity, and it can also increase the miss penalty because of the increased complexity of choosing a victim line.

The choice of associativity ultimately boils down to a trade-off between the hit time and the miss penalty. Traditionally, high-performance systems that pushed the clock rates would opt for smaller associativity for L1 caches (where the miss penalty is only a few cycles) and a higher degree of associativity for the lower levels, where the miss penalty is higher. For example, in Intel Core i7 systems, the L1 and L2 caches are 8-way associative, and the L3 cache is 16-way.

Impact of Write Strategy

Write-through caches are simpler to implement and can use a *write buffer* that works independently of the cache to update memory. Furthermore, read misses are less expensive because they do not trigger a memory write. On the other hand, write-back caches result in fewer transfers, which allows more bandwidth to memory for I/O devices that perform DMA. Further, reducing the number of transfers becomes increasingly important as we move down the hierarchy and the transfer times increase. In general, caches further down the hierarchy are more likely to use write-back than write-through.

6.5 Writing Cache-Friendly Code

In Section 6.2, we introduced the idea of locality and talked in qualitative terms about what constitutes good locality. Now that we understand how cache memories work, we can be more precise. Programs with better locality will tend to have lower miss rates, and programs with lower miss rates will tend to run faster than programs with higher miss rates. Thus, good programmers should always try to

Aside Cache lines, sets, and blocks: What's the difference?

It is easy to confuse the distinction between cache lines, sets, and blocks. Let's review these ideas and make sure they are clear:

- A *block* is a fixed-size packet of information that moves back and forth between a cache and main memory (or a lower-level cache).
- A *line* is a container in a cache that stores a block, as well as other information such as the valid bit and the tag bits.
- A *set* is a collection of one or more lines. Sets in direct-mapped caches consist of a single line. Sets in set associative and fully associative caches consist of multiple lines.

In direct-mapped caches, sets and lines are indeed equivalent. However, in associative caches, sets and lines are very different things and the terms cannot be used interchangeably.

Since a line always stores a single block, the terms "line" and "block" are often used interchangeably. For example, systems professionals usually refer to the "line size" of a cache, when what they really mean is the block size. This usage is very common and shouldn't cause any confusion as long as you understand the distinction between blocks and lines.

write code that is *cache friendly*, in the sense that it has good locality. Here is the basic approach we use to try to ensure that our code is cache friendly.

1. *Make the common case go fast.* Programs often spend most of their time in a few core functions. These functions often spend most of their time in a few loops. So focus on the inner loops of the core functions and ignore the rest.
2. *Minimize the number of cache misses in each inner loop.* All other things being equal, such as the total number of loads and stores, loops with better miss rates will run faster.

To see how this works in practice, consider the sumvec function from Section 6.2:

```
int sumvec(int v[N])
{
    int i, sum = 0;

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

Is this function cache friendly? First, notice that there is good temporal locality in the loop body with respect to the local variables i and sum. In fact, because these are local variables, any reasonable optimizing compiler will cache them in the register file, the highest level of the memory hierarchy. Now consider the stride-1 references to vector v. In general, if a cache has a block size of B bytes, then a stride-k reference pattern (where k is expressed in words) results in an average of min $(1, (\textit{word size} \times k)/B)$ misses per loop iteration. This is minimized for $k = 1$, so the stride-1 references to v are indeed cache friendly. For example, suppose that v is block aligned, words are 4 bytes, cache blocks are 4 words, and the cache is initially empty (a cold cache). Then, regardless of the cache organization, the references to v will result in the following pattern of hits and misses:

v[i]	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
Access order, [h]it or [m]iss	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]

In this example, the reference to v[0] misses and the corresponding block, which contains v[0]–v[3], is loaded into the cache from memory. Thus, the next three references are all hits. The reference to v[4] causes another miss as a new block is loaded into the cache, the next three references are hits, and so on. In general, three out of four references will hit, which is the best we can do in this case with a cold cache.

To summarize, our simple sumvec example illustrates two important points about writing cache-friendly code:

- Repeated references to local variables are good because the compiler can cache them in the register file (temporal locality).
- Stride-1 reference patterns are good because caches at all levels of the memory hierarchy store data as contiguous blocks (spatial locality).

Spatial locality is especially important in programs that operate on multidimensional arrays. For example, consider the sumarrayrows function from Section 6.2, which sums the elements of a two-dimensional array in row-major order:

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

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

Since C stores arrays in row-major order, the inner loop of this function has the same desirable stride-1 access pattern as sumvec. For example, suppose we make the same assumptions about the cache as for sumvec. Then the references to the array a will result in the following pattern of hits and misses:

a[i][j]	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
$i = 0$	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]
$i = 1$	9 **[m]**	10 [h]	11 [h]	12 [h]	13 **[m]**	14 [h]	15 [h]	16 [h]
$i = 2$	17 **[m]**	18 [h]	19 [h]	20 [h]	21 **[m]**	22 [h]	23 [h]	24 [h]
$i = 3$	25 **[m]**	26 [h]	27 [h]	28 [h]	29 **[m]**	30 [h]	31 [h]	32 [h]

But consider what happens if we make the seemingly innocuous change of permuting the loops:

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

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

In this case, we are scanning the array column by column instead of row by row. If we are lucky and the entire array fits in the cache, then we will enjoy the same miss rate of 1/4. However, if the array is larger than the cache (the more likely case), then each and every access of `a[i][j]` will miss!

`a[i][j]`	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
$i = 0$	1 **[m]**	5 **[m]**	9 **[m]**	13 **[m]**	17 **[m]**	21 **[m]**	25 **[m]**	29 **[m]**
$i = 1$	2 **[m]**	6 **[m]**	10 **[m]**	14 **[m]**	18 **[m]**	22 **[m]**	26 **[m]**	30 **[m]**
$i = 2$	3 **[m]**	7 **[m]**	11 **[m]**	15 **[m]**	19 **[m]**	23 **[m]**	27 **[m]**	31 **[m]**
$i = 3$	4 **[m]**	8 **[m]**	12 **[m]**	16 **[m]**	20 **[m]**	24 **[m]**	28 **[m]**	32 **[m]**

Higher miss rates can have a significant impact on running time. For example, on our desktop machine, `sumarrayrows` runs 25 times faster than `sumarraycols` for large array sizes. To summarize, programmers should be aware of locality in their programs and try to write programs that exploit it.

Practice Problem 6.17 (solution page 701)

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 a 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	______	Row0	m	______
Row 1	______	______	Row 1	______	______

B. Repeat the problem for a cache with 32 data bytes.

Practice Problem 6.18 (solution page 702)

The heart of the recent hit game *SimAquarium* is a tight loop that calculates the average position of 512 algae. You are evaluating its cache performance on a machine with a 2,048-byte direct-mapped data cache with 32-byte blocks ($B = 32$). You are given the following definitions:

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

struct algae_position grid[32][32];
int total_x = 0, total_y = 0;
int i, j;
```

You should also 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 array `grid`. Variables `i`, `j`, `total_x`, and `total_y` are stored in registers.

Determine the cache performance for the following code:

```
for (i = 31; i >= 0; i--) {
    for (j = 31; j >= 0; j--) {
        total_x += grid[i][j].x;
    }
}

for (i = 31; i >= 0; i--) {
    for (j = 31; j >= 0; 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?

Practice Problem 6.19 (solution page 702)

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

```
for (i = 31; i >= 0; i--){
    for (j = 31; j >= 0; j--) {
        total_x += grid[j][i].x;
        total_y += grid[j][i].y;
    }
}
```

A. What is the total number of reads?

B. What is the total number of reads that hit in the cache?

C. What is the hit rate?

D. What would the miss hit be if the cache were twice as big?

Practice Problem 6.20 (solution page 702)

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

```
for (i = 31; i >= 0; i--){
    for (j = 31; j >= 0; j--) {
        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 hit in the cache?

C. What is the hit rate?

D. What would the hit rate be if the cache were twice as big?

6.6 Putting It Together: The Impact of Caches on Program Performance

This section wraps up our discussion of the memory hierarchy by studying the impact that caches have on the performance of programs running on real machines.

6.6.1 The Memory Mountain

The rate that a program reads data from the memory system is called the *read throughput*, or sometimes the *read bandwidth*. If a program reads n bytes over a period of s seconds, then the read throughput over that period is n/s, typically expressed in units of megabytes per second (MB/s).

If we were to write a program that issued a sequence of read requests from a tight program loop, then the measured read throughput would give us some insight into the performance of the memory system for that particular sequence of reads. Figure 6.40 shows a pair of functions that measure the read throughput for a particular read sequence.

The `test` function generates the read sequence by scanning the first `elems` elements of an array with a stride of `stride`. To increase the available parallelism in the inner loop, it uses 4×4 unrolling (Section 5.9). The `run` function is a wrapper that calls the `test` function and returns the measured read throughput. The call to the `test` function in line 37 warms the cache. The `fcyc2` function in line 38 calls the `test` function with arguments `elems` and estimates the running time of the `test` function in CPU cycles. Notice that the `size` argument to the `run` function is in units of bytes, while the corresponding `elems` argument to the `test` function is in units of array elements. Also, notice that line 39 computes MB/s as 10^6 bytes/s, as opposed to 2^{20} bytes/s.

The `size` and `stride` arguments to the `run` function allow us to control the degree of temporal and spatial locality in the resulting read sequence. Smaller values of `size` result in a smaller working set size, and thus better temporal locality. Smaller values of `stride` result in better spatial locality. If we call the `run` function repeatedly with different values of `size` and `stride`, then we can recover a fascinating two-dimensional function of read throughput versus temporal and spatial locality. This function is called a *memory mountain* [112].

Every computer has a unique memory mountain that characterizes the capabilities of its memory system. For example, Figure 6.41 shows the memory mountain for an Intel Core i7 Haswell system. In this example, the `size` varies from 16 KB to 128 MB, and the `stride` varies from 1 to 12 elements, where each element is an 8-byte `long int`.