Reducing the Miss Ratio, cont.

Other software optimizations

Loop interchange

[H&P §5.5] Sometimes we can increase temporal locality by exchanging inner and outer

```c
for (j = 0; j < MAXJ; j++)
    for (i = 0; i < MAXI; i++)
        x[i][j] = c * x[i][j]
```

The problem is that `x[i][j]` and `x[i+1][j]`

```c
for (i = 0; i < MAXI; i++)
    for (j = 0; j < MAXJ; j++)
        x[i][j] = c * x[i][j]
```

This code has the same meaning, but is less likely to cause misses.

Loop fusion

Two loops with identical sets of iterations can be combined.

```c
for (j = 0; j < MAXJ; j++)
    for (i = 0; i < MAXI; i++)
        a[i][j] = cos(x[i][j]);

for (j = 0; j < MAXJ; j++)
    for (i = 0; i < MAXI; i++)
        z[i][j] = a[i][j] * d[i];
```

The problem is that `a[i][j]` is written

```c
for (j = 0; j < MAXJ; j++)
    for (i = 0; i < MAXI; i++) {
        a[i][j] = cos(x[i][j]);
        z[i][j] = a[i][j] * d[i];
    }
```

Here, the two loops have been “fused.”

Blocking
This strategy divides a large array into smaller regions so that a “block” of an array can fit into the cache.

Each iteration …

```
for (k = 0; k < N; k++)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      z[k] = z[k] + x[i][j] * y[k];
```

```
for (ii = 0; ii < N; ii += B)
  for (jj = 0; jj < N; jj += B)
    for (k = 0; k < N; k++)
      for (i = ii; i < min(ii+B, N); i++)
        for (j = jj; j < min(jj+B, N); j++)
          z[k] = z[k] + x[i][j] * y[k];
```

What’s the performance problem in the code on the left?

To attack this problem, blocking factor $B$ is selected so

**Reducing Hit Time**

**Avoiding address translation while indexing cache**

[H&P §5.7] The CPU generates *virtual* addresses, which correspond to locations in virtual memory.

In principle, the virtual addresses are translated to physical addresses using a page table.
But this is too slow, so in practice, a *translation lookaside buffer* (TLB) is used.

It is like a special cache that is indexed by page number.

If there is a hit on a page number, then the address of the page in memory (called the *page-frame address*) is immediately obtained.

Therefore, the TLB and the cache must be accessed sequentially.

This adds an extra cycle in case of a hit.

How can we avoid wasting this time?

Recall this diagram from Lecture 4:
We always need to read a line into a latch and then select the word (cf. the direct-mapped cache diagram in Lecture 4).

Now, if we know the index before address translation takes place, we can read the line into the latch while address translation is occurring.

Let’s take a look at address translation.

In this example, what is the page size?

How much physical memory is there?
Our goal is to allow the cache to be indexed before address translation completes.

In order to do that, we need to have the index field be *entirely contained* within the page offset.

Cache hit time reduces from two cycles to one!

… because the cache can now be indexed in parallel with TLB (although the tag match uses output from the TLB).

But there are some constraints...

- Suppose our cache is direct mapped. Then the index field just contains the line number. So, (line number || block offset) must fit inside the page offset.

What is the largest the cache can be?

- If we want to increase the size of the cache, what can we do?
Options:

- For new machines, select page size such that—
  \[ \text{page size} \geq \frac{\text{cache size}}{\text{associativity}} \]

- If page size is fixed, select associativity so that—
  \[ \text{associativity} \geq \frac{\text{cache size}}{\text{page size}} \]

**Example:** MC88110

- Page size = 4KB
- I-cache, D-cache are both: 8KB, 2-way set-associative
  \[ (4KB = 8KB / 2) \]

**Example:** VAX series

- Page size = 512B
- For a 16KB cache, need assoc. = (16KB / 512B) = 32-way set. assoc.!

*Moral:* Sometimes associativity is thrust upon you!

**Virtually addressed caches**

Rather than slow down every instruction by serializing address translation and cache access, it makes sense just to use a virtually addressed cache.

But virtually addressed caches have their own problems.

One problem is that

- physical addresses are unique within the system, while
- two processes may use the *same virtual address* to refer to *different data*.

So if a Process Q starts running while Process P’s data is in the cache, what could happen?
We can avoid this if we purge the cache each time we switch processes, but this has a tendency to drive up the miss rate dramatically.

Experiments by Agarwal (see p. 446 of H&P, Figure 5.25) showed that in a large cache (128 KB or larger), about 6 times more misses occurred when the cache was purged at process switches (aka. context swaps).

One way to solve this problem: Along with each tag, keep a process-ID field (PID), telling which process the block belongs to.

Then the cache only needs to be flushed when a PID is recycled.

Another problem is that, when memory is shared, two processes may use different virtual addresses to refer to the same physical data.

That is, they have different virtual names for the same block.

Why are duplicate copies of information bad? Think about what happens on a process switch. Assume
• The processor has been running Process 1.
• It then switches to Process 2.
• Later it switches back to Process 1.

This is called the *alias problem* (or the *synonym problem*).

To address this, we need an *antialiasing* mechanism.

*Example:* Alpha 21264 has a 64 KB instruction cache that is 2-way set-associative. Its pages are 8 KB.

• What is the size of the cache, as a power of two?
• The cache is two-way set associative, so how many bits are there in the index and block-offset fields combined?
• How many bits are in the page-offset field?
• By how many bits does the index field overlap the tag?

Therefore, if we know that a block is in the cache, but we do not know its virtual address, in how many *different* places could it possibly be stored?

When a cache miss occurs, the hardware simply checks the tags on each of these lines to see if any one of them matches the physical address being fetched. If one does, the line is invalidated.

**Direct-mapped caches**

Direct-mapped caches have lower hit times. So, if we use a direct-mapped cache,
• There is no need for search of members of the set.
• Might require larger caches for equal miss rate.
• Sometimes page size doesn’t let you do it (because cache would have to be too small)—unless you go to

Small caches
(See discussion of cache size.)

Write pipelining
Not all cache accesses are created equal—writes take an additional cycle over reads

1. Access cache array
2. word select/tag compare,
3. write word to block and cache.

Solution: Buffer the write and do (3) in the background!