Comparing cache organizations

Like many architectural features, caches are evaluated experimentally
- As always, performance depends on the actual instruction mix, since different programs will have different memory access patterns
- Simulating or executing real applications is the most accurate way to measure performance characteristics

The graphs on the next few slides illustrate the simulated miss rates for several different cache designs
- Again lower miss rates are generally better, but remember that the miss rate is just one component of average memory access time and execution time

Associativity tradeoffs and miss rates

Earlier we saw, higher associativity ==> more complex HW
But a highly-associative cache will have a lower miss rate
- Each set has more blocks, so there’s less chance of a conflict between two addresses
- Overall, this will reduce AMAT and memory stall cycles
Cache size and miss rates

Cache size also has a significant impact on performance

- In a larger cache there’s less chance there will be of a conflict
- Again this means the miss rate decreases, so the AMAT and number of memory stall cycles also decrease

The complete Figure 5.30 depicts the miss rate as a function of both the cache size and its associativity

Block size and miss rates

Finally, the figure below shows miss rates relative to block size and overall cache size

- Smaller blocks do not take maximum advantage of spatial locality
- But if blocks are too large, there are fewer blocks available, and more potential conflicts misses
Review of Fields of Address

Address:

\[
\begin{array}{ccc}
\text{Tag} & \text{Index} & \text{Block Offset} & \text{Byte Offset} \\
\text{address_size - m – k - 2} & m & k & 2
\end{array}
\]

Size Determined By:
- Byte Offset: always 2 bits ‘cause we work w/words
- Block Offset: \(2^k\) words in block require k bits
- Index: \(2^m\) cache entries require m bits
- Tag: address_size - m – k -2

Memory and overall performance

How do cache hits/misses affect system performance?
- Assuming a hit time of one CPU clock cycle, program execution will continue normally on a cache hit.
- For cache misses, assume the CPU stalls to load from main memory.

The total number of stall cycles depends on the number of cache misses and the miss penalty

\[
\text{Memory stall cycles} = \text{Memory accesses} \times \text{miss rate} \times \text{miss penalty}
\]

To include stalls due to cache misses in CPU performance equations, we have to add them to the “base” number of execution cycles.

\[
\text{CPU time} = (\text{CPU execution cycles} + \text{Memory stall cycles}) \times \text{Cycle time}
\]
Performance example

Assume that 33% of the instructions in a program are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles.

Memory stall cycles = Memory accesses x Miss rate x Miss penalty
= 0.33 I x 0.03 x 20 cycles
= 0.2 I cycles

If I instructions are executed, then the number of wasted cycles will be 0.2 x I

This code is 1.2 times slower than a program with a “perfect” CPI of 1!

Memory systems are a bottleneck

CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

Processor performance traditionally outpaces memory performance, so the memory system is often the bottleneck.

EG, with a base CPI of 1, CPU time from the last page is:

\[
\text{CPU time} = (I + 0.2 I) \times \text{Cycle time}
\]

What if we could double the CPU performance so the CPI becomes 0.5, but memory performance remained the same?

\[
\text{CPU time} = (0.5 I + 0.2 I) \times \text{Cycle time}
\]

The overall CPU time improves by just 1.2/0.7 = 1.7 times!

✨ Speeding up only part of a system has diminishing returns
Basic main memory design

There are some ways to organize main memory to reduce miss penalties and help with caching.

Let’s assume the following:

3 steps are taken when a cache needs to load data from the main memory:

1. It takes 1 cycle to send an address to the RAM
2. There is a 15-cycle latency for each RAM access
3. It takes 1 cycle to return data from the RAM

In this setup, buses are all one word wide.

If the cache has 1 word (wd) blocks, then filling a block from RAM (i.e., the miss penalty) would take 17 cycles:

\[ 1 + 15 + 1 = 17 \text{ clock cycles} \]

- The cache controller sends the address to RAM, waits and receives the data.

Miss penalties for larger cache blocks

If the cache has four-word blocks, then loading a single block would need four individual main memory accesses, and a miss penalty of 68 cycles!

\[ 4 \times (1 + 15 + 1) = 68 \text{ clock cycles} \]
A wider memory

One way to decrease the miss penalty is to widen the memory and its interface to the cache, so multiple words are read from RAM in one shot.

Reading 4 words from memory at once needs just 17 cycles:

\[ 1 + 15 + 1 = 17 \text{ cycles} \]

The disadvantage is the cost of the wider buses—each additional bit of memory width requires another connection to the cache.

An interleaved memory

Another approach is to interleave the memory, or splitting it into “banks” accessible individually.

The main benefit is overlapping the latencies of accessing each word.

Eg, if main memory has 4 banks, each 1 word wide, then we can load 4 words in just 20 cycles:

\[ 1 + 15 + (4 \times 1) = 20 \text{ cycles} \]

Buses are still 1 word wide, so 4 cycles are needed to x-fer data.
This is cheaper than implementing a 4 bus, but not too much slower.
Interleaved memory accesses

Here is a diagram to show how the memory accesses can be interleaved:
- The magenta cycles represent sending an address to a memory bank.
- Each memory bank has a 15-cycle latency, and it takes another cycle (shown in blue) to return data from the memory.

This is the same basic idea as pipelining:
- As soon as we request data from one memory bank, we can request data from another bank as well.
- Each individual load takes 17 clock cycles, but four overlapped loads require just 20 cycles.

Which is better?

Increasing block size can improve hit rate (due to spatial locality), but transfer time increases. Which cache configuration would be better?

<table>
<thead>
<tr>
<th>Cache #1</th>
<th>Cache #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>32-bytes</td>
</tr>
<tr>
<td>Miss rate</td>
<td>5%</td>
</tr>
</tbody>
</table>

Assume both caches have single cycle hit times. Memory accesses take 15 cycles, and the memory bus is 8-bytes wide:
- i.e., an 16-byte memory access takes 18 cycles:
  1 (send address) + 15 (memory access) + 2 (two 8-byte transfers)

recall: \[ \text{AMAT} = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty}) \]
Writing Cache Friendly Code

Two major rules:
Replaced references to data are good (temporal locality)
Stride-1 reference patterns are good (spatial locality)
Example: cold cache, 4-byte words, 4-word cache blocks

```c
int sum_array_rows(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;
}

int sum_array_cols(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;
}
```

Miss rate = 1/4 = 25%  Miss rate = 100%

Adapted from Randy Bryant

Which is better?
Increasing block size can improve hit rate (due to spatial locality),
but transfer time increases. Which cache configuration would be better?

<table>
<thead>
<tr>
<th>Cache #1</th>
<th>Cache #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>32-bytes</td>
</tr>
<tr>
<td>Miss rate</td>
<td>5%</td>
</tr>
</tbody>
</table>

Assume both caches have single cycle hit times. Memory accesses take 15 cycles, and the memory bus is 8-bytes wide:
- i.e., a 16-byte memory access takes 18 cycles:
  1 (send address) + 15 (memory access) + 2 (two 8-byte transfers)

 Cache #1:
 Miss Penalty = 1 + 15 + 32B/8B = 20 cycles
 AMAT = 1 + (.05 * 20) = 2

 Cache #2:
 Miss Penalty = 1 + 15 + 64B/8B = 24 cycles
 AMAT = 1 + (.04 * 24) = 1.96
Caching Data Transfer Summary

Writing to a cache poses a couple of interesting issues:
- **Write-through** and **write-back** policies keep the cache consistent with main memory in different ways for write hits.
- **Write-around** and **allocate-on-write** are two strategies to handle write misses, differing in whether updated data is loaded into the cache.

Memory system performance depends upon the cache **hit time**, **miss rate** and **miss penalty**, as well as the actual program being executed:
- We can use these numbers to find the average memory access time.
- We can also revise our CPU time formula to include stall cycles.

\[
AMAT = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty})
\]

Summary (continued)

\[
\text{Memory stall cycles} = \text{Memory accesses} \times \text{miss rate} \times \text{miss penalty}
\]
\[
\text{CPU time} = (\text{CPU execution cycles} + \text{Memory stall cycles}) \times \text{Cycle time}
\]

The organization of a memory system affects its performance:
- The cache size, block size, and associativity affect the miss rate.
- We can organize the main memory to help reduce miss penalties. For example, **interleaved memory** supports pipelined data accesses.