Cache Memory

- memory hierarchy
- CPU memory request presented to first-level cache first
- if data NOT in cache, request sent to next level in hierarchy...
- and so on
Cache Hierarchy

- for a system with a first level cache and memory ONLY

\[ t_{\text{eff}} = h t_{\text{cache}} + (1-h) t_{\text{main}} \]

where

\[ t_{\text{eff}} = \text{effective access time} \]
\[ h = \text{probability of a cache hit [hit rate]} \]
\[ t_{\text{cache}} = \text{cache access time} \]
\[ t_{\text{main}} = \text{main memory access time} \]

- assuming \( t_{\text{main}} = 60\text{ns} \) and \( t_{\text{cache}} = 5\text{ns} \)

<table>
<thead>
<tr>
<th>( h )</th>
<th>( t_{\text{eff}}(\text{ns}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>5.0</td>
</tr>
<tr>
<td>0.99</td>
<td>5.6</td>
</tr>
<tr>
<td>0.98</td>
<td>6.1</td>
</tr>
<tr>
<td>0.89</td>
<td>11.1</td>
</tr>
<tr>
<td>0.50</td>
<td>32.5</td>
</tr>
<tr>
<td>0.00</td>
<td>60.0</td>
</tr>
</tbody>
</table>

- small changes in hit ratio [as \( h \to 1 \)] are amplified by \( t_{\text{main}}/t_{\text{cache}} \)

- if \( t_{\text{main}}/t_{\text{cache}} \) is 10 then a decrease of 1\% in \( h \) [as \( h \to 1 \)] results in a 10\% increase in \( t_{\text{eff}} \)
Temporal and Locality of Reference

- exploit the temporal locality and locality of reference inherent in *typical* programs

- high probability memory regions
  - recently executed code
  - recent stack accesses
  - recently accessed data

- if the memory references occur randomly, cache will have very little effect
- NB: see average on graph
A real example collected from an early IA32 PC running OSF/1
K-way Set Associative Cache with N Sets

N sets, K directories and L bytes per cache line

L = 16
4 x 32bit words per cache line
Searching a K-way Cache

- address mapped onto a particular set [set #]...
- by extracting bits from incoming address
- NB: tag, set # and offset

- consider an address that maps to set 1
- the set 1 tags of all K directories are compared with the incoming address tag simultaneously
- if a is match found [hit], corresponding data returned offset within cache line
- the K data lines in the set are accessed concurrently with the directory entries so that on a hit the data can be routed quickly to the output buffers
- if a match is NOT found [miss], read data from memory, place in cache line within set and update corresponding cache tag [choice of K positions]
- cache line replacement strategy [within a set] - Least Recently Used [LRU], pseudo LRU, random...
Caches

Searching a K-way Cache...

Incoming address (32 bits or 8 nybbles)

1234 001 8
  tag  set off

L=16, N=4096, K = K

NB: cache lines aligned on 16 byte boundaries
Cache Organisation

- the cache organisation is described using the following three parameters
  - L \(\text{bytes per cache line [cache line or block size]}\)
  - K \(\text{cache lines per set [degree of associativity K-way]}\)
  - N \(\text{number of sets}\)

- cache size \(L \times K \times N\) bytes

- \(N = 1\)
  - fully associative cache, incoming address tag compared with \textbf{ALL} cache tags
  - address can map to any one of the K cache lines

- \(K = 1\)
  - direct mapped cache, incoming address tag compared with \textbf{ONLY ONE} cache tag
  - address can be mapped \textbf{ONLY} onto a single cache line

- \(N > 1\) and \(K > 1\)
  - set-associative [\textbf{K-way cache}]
Write-Through vs Write-Back [Write-Deferred]

- WRITE-THROUGH
  - write hit
    update cache line and main memory
  - write miss
    update main memory ONLY [non write allocate cache]
      OR
    select a cache line [using replacement policy]
    fill cache line by reading data from memory
    write to cache line and main memory [write allocate cache]

NB: unit of writing [e.g. 4 bytes] likely to be much smaller than cache line size [e.g. 16 bytes]
Write-Through vs Write-Back [Write-Deferred]...

- WRITE-BACK [WRITE-DEFERRED]
  - write hit
    
    update cache line ONLY
    *ONLY update main memory when cache line is flushed or replaced*

  - write miss
    
    select a cache line [using replacement policy]
    *write-back previous cache line to memory if dirty/modified*
    fill cache line by reading data from memory
    write to cache line ONLY

NB: unit of writing [e.g. 4 bytes] likely to be much smaller than cache line size [e.g. 16 bytes]
Typical Cache Miss Rates

- data from *Hennessy and Patterson*
- shows *miss rate* rather than *hit rate*
- *miss rate* more interesting!
- note how data [address trace] was collected
- trace fed through a software cache model with
  - $L = 32$
  - LRU replacement policy

![Table of Cache Sizes and Miss Rates](image)

**FIGURE 8.12** Total miss rate for each size cache and percentage of each according to the "three Cs." Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases. Hill [1987] measured this trace using 32-byte blocks and LRU replacement. It was generated on a VAX-11 running Unix by mixing three systems’ traces, using a multiprogramming workload and three user traces. The total length was just over a million addresses; the largest piece of data referenced during the trace was 221 KB. Figure 8.13 (page 452) shows the same information graphically. Note that the 2:1 cache rule of thumb (inside front cover) is supported by the statistics in this table: a direct-mapped cache of size $N$ has about the same miss rate as a 2-way-set-associative cache of size $N/2$.
Typical Cache Miss Rates

- plot of *miss rate vs cache size* using Hennessy and Patterson data

- note that the 2:1 cache rule of thumb

  "the miss rate of a direct mapped cache of size X is about the same as a 2-way set-associative cache of size X/2"

- rule supported by data [although not perfectly]
The 3 Cs

- *Hennessy and Patterson* classify cache misses into 3 distinct types
  - compulsory
  - capacity
  - conflict

- total misses = compulsory + capacity + conflict

- assume an address trace is being processed through a cache model

- **compulsory** misses are due to addresses appearing in the trace for the first time, the number of unique cache line addresses in trace [reduce by prefetching data into cache]

- **capacity** misses are the additional misses which occur when simulating a fully associative cache [reduce by increasing cache size]

- **conflict** misses are the additional misses which occur when simulating a non fully associative cache [reduce by increasing cache associativity K]

- see *Hennessy and Patterson* data
Direct Mapped vs Associative Caches

• will an associative cache always outperform a direct mapped cache of the same size?

• consider two caches

K=4, N=1, L=16 [64 byte fully associative]
K=1, N=4, L=16 [64 byte direct mapped]

LxKxN equal...

and the following repeating sequence of 5 addresses

a, a+16, a+32, a+48, a+64, a, a+16, a+32...

• increase address by 16 each time, as this is the line size [L = 16]

• caches can contain 4 addresses, sequence comprises 5 addresses
• 5 addresses won't fit into 4
**Direct Mapped vs Associative Caches**

- **Fully associative**: only 4 addresses can fit in the 4-way cache so, due to the LRU replacement policy, every access will be a miss.

- **Direct mapped**: since ONLY addresses \( a \) and \( a+64 \) will conflict with each other as they map to the same set [set 0 in diagram], there will be 2 misses and 3 hits per cycle of 5 addresses.
Direct Mapped vs Associative Caches...

- the 3 Cs means that the conflict misses can be negative!

- consider previous example with 10 addresses [5 address sequence repeated twice]

<table>
<thead>
<tr>
<th></th>
<th>fully associative</th>
<th>direct mapped</th>
</tr>
</thead>
<tbody>
<tr>
<td>compulsory</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>capacity</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>conflict</td>
<td>0</td>
<td>-3</td>
</tr>
<tr>
<td>total</td>
<td>10 misses</td>
<td>7 misses</td>
</tr>
</tbody>
</table>

- calculate conflict misses from total, compulsory and capacity misses which are known

- conflict misses = total misses – compulsory misses - capacity misses

- for direct mapped cache, conflict misses = $7 - 5 - 5 = -3$