UT 3. Memory subsystem

Tema 3.1 Performance of the memory subsystem

A. Doménech, J. Duato, P. López, V. Lorente,
A. Pérez, S. Petit, J.C. Ruiz, S. Sáez, J. Sahuquillo

Department of Computer Engineering
Universitat Politècnica de València

Contents

Bibliography

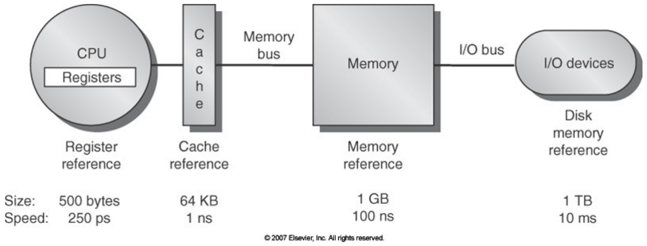
John L. Hennessy and David A. Patterson.
Computer Architecture, Fifth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2012.

Tema 3.1 Performance of the memory subsystem

Contents

Memory hierarchy

"...programmers would like to have access to an unlimited amount of fast memory..."
$\rightarrow$ There is not any technology with such capabilities
Solution: Hierarchical organization using different technologies.
Memory hierarchy: Memory is organized into several levels, where each level is smaller, faster, and more expensive than the lower level.



Memory hierarchy (cont.)

Why does the hierarchy achieve good performance?

- Objective: Speed close to the fastest level.
- Principle of locality. Programs tend to reuse code and data.
  - Temporal locality: Recently accessed data items will be accessed again soon.
    - It is usually high in the code: “Programs spend around 90% of their execution time running 10% of the code”.
  - Spatial locality: Items with nearby addresses tend to be accessed shortly after $\rightarrow$ block organization.

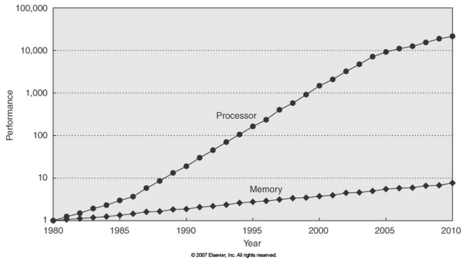
Why is it an efficient solution?

- Technologies with different characteristics are used.
  - Fast but expensive for performance. Used only for small caches.
  - Dense but cheaper (cost per bit) for capacity.

1. Memory hierarchy: a quick review.

Importance of the memory hierarchy

The difference in speed between the processor and the DRAM memory has grown exponentially during two decades (1985 - 2005) $\rightarrow$ cache hierarchy.



- In 1980, microprocessors were implemented without caches.
- In 2001, two on-chip cache levels.
- Currently many systems deploy three cache levels.

Requirements vary according to computer type

- Desktop computer: One user, one application.
  - Objective: Latency reduction.
- Servers. Multiple users, multiple applications.
  - Objectives: Throughput, protection.
- Embedded computers. One application, sometimes without operating system. Small main memory.
  - Objectives: Low power consumption. Real-time (important to know the worst-case performance).

Cache memories

“Cache: A safe place to hide or store things”

Cache: First level of the memory hierarchy.

The term *cache* is currently used when the stored information is reused: file caches, disk cache, name cache, etc.

- *Hit*: Processor finds the requested data in cache.
- *Miss*: Otherwise. The *block* that contains the requested word is fetched from main memory to cache.

Cache management (hits, misses, and replacements) is performed by hardware.

Virtual Memory

If the system supports virtual memory, the objects referenced by a program can be located either in main memory or disk.

The addressing space is divided into *pages* that must be in memory to enable access to them.

On a *page miss*, the entire page is transferred from disk to main memory.

Page misses are handled by *software* and do not stall the processor. The processor switches context, running another task while the disk is accessed.

Tema 3.1 Performance of the memory subsystem

Contents

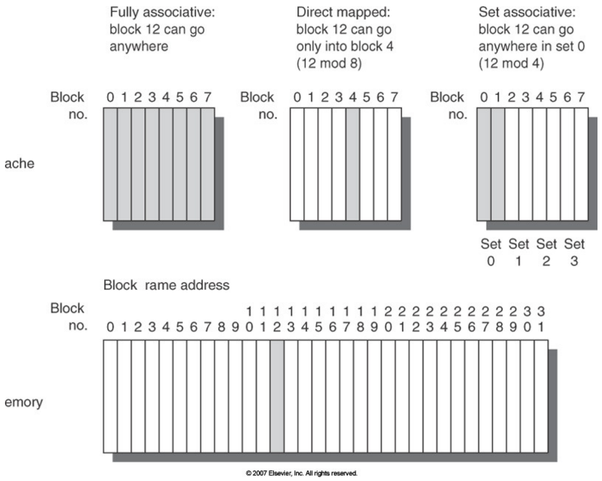
Cache memory characteristics

Any level of the memory hierarchy can be characterized by answering the following questions:

- Block placement. Where can a block be placed?
- Block identification. How is a block found?
- Replacement policy. Which block should be replaced on a cache miss?
- Write policy. What happens on a write?

2. Cache structure and operation: a quick review

Block Placement



Fully associative: block 12 can go anywhere
Direct mapped: block 12 can go only into block 4 (12 mod 8)
Set associative: block 12 can go anywhere in set 0 (12 mod 4)
Block no.
0 1 2 3 4 5 6 7
ache
Set 0
Set 1
Set 2
Set 3
Block rame address
Block no.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
emory
© 2007 Elsevier, Inc. All rights reserved.

Block Placement (cont.)

- *Direct-mapped*. A block can only be stored in a certain cache line.

  `#line = #referenced_block MOD #cache_lines`
- *Fully-associative*. A block can be stored in any cache line.
- *n-way set-associative*. A block can only be stored in a certain set (consisting of n lines).

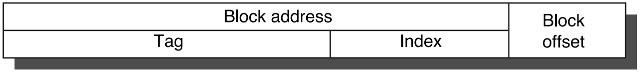
  `#set = #referenced_block MOD #cache_sets`

Block identification

Each block stored in cache has an associated tag.

To know if a block is in cache, the tag field of the block address is compared with the tags in the target set.

Parts of an address issued by the processor:



A valid bit (V) indicates whether a line contains valid information. Only tags having the valid bit set are compared.

Block identification (cont.)

How to compare?

- In parallel with all the tags in the set. Just a single comparison for direct mapping.
- Not necessary to include the *block offset* in the comparison, because the block is present or not as a whole.

For a given cache size, increasing the associativity increases the number of blocks per set and decreases the number of sets. So, the size of the index and tag decreases and increases, respectively.

Replacement policy

On a cache miss, the referenced block is fetched from main memory. If the target set is full, which block is removed?

Trivial in direct-mapped caches (just a single candidate).

Different strategies can be applied in (set-)associative caches:

- LRU: Less Recently Used. Exploits temporal locality.
- Pseudo LRU. Less costly than LRU, similar performance for a high number of ways.
- Random. The candidate is randomly selected. Easy to implement. Useful for structures with poor locality.

Write policies

Only one data item of the block is modified (byte, word, . . .).
The block must also be updated in main memory (MM).

Strategies in case of hit:

- *Write-through*. Data is written in both cache and MM.
  - Easiest to implement.
  - Main memory is always up to date.
- *Write-back*. Data is written just in cache.
  - The ”dirty” blocks (*dirty bit* set) are updated in MM when they are replaced.
  - Consumes less memory bandwidth than *Write-through*

Write policies (cont.)

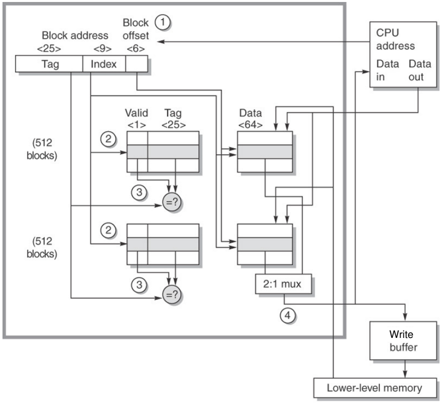
Strategies in case of miss:

- *Write allocate*. The block is brought to cache.
  Then, it is written by proceeding like in a cache hit. Usually combined with *write-back*.
- *No-write allocate*. The block is not brought to cache. It is just modified in the lower level where it is currently stored. Usually combined with *write-through* .

Write policies (cont.)

Example: Data cache in the old AMD Opteron (pre-Barcelona core)

- 64KB, 2 ways, 64B lines. LRU replacement
- Write-back, write-allocate



Write policies (cont.)

Cache operation

1. Processor sends the address (40 bits)
   = Block address: 34 bits + Block offset: 6 bits
2. Set is accessed with the index field:

$$2^{\text{Index}} = \frac{\text{Cache capacity}}{\text{Blocksize} \cdot \text{Num.ways}} = \frac{65536}{64 \cdot 2} = 512 = 2^9$$

   Tag: 34–9=25 bits
3. Both tags in the set are read and compared with the one sent by the processor. The data array is accessed at the same time.
4. On a hit, the multiplexer input is selected and the data are delivered to the processor.

Contents

Average access time

$T_{\text{access}} = Ht + MR \times MP$

where Ht: cache hit time
MR: miss rate
MP: miss penalty

Execution Time equation
extended to account for the cache latency:

$$T_{\text{ex}} = T_{\text{ex cpu}} + T_{\text{extra memory}}$$

where $T_{\text{ex cpu}}$ includes the cache hit time and $T_{\text{extra memory}}$ the time to handle misses (assuming that misses stall the processor)

3. Performance evaluation of the memory subsystem.

- $T_{\text{ex cpu}} = \text{I} \times \text{CPI} \times \text{T}$
- $T_{\text{extra memory}} = \text{Memory stall cycles} \times \text{T}$
- Memory stall cycles = Num. misses × Miss penalty = M × MP
- Num. misses = Instructions $\times \frac{\text{Accesses}}{\text{Instruction}} \times$ Miss rate = I × API × MR

Replacing:

$$T_{\text{extra memory}} = \text{I} \times \text{API} \times \text{MR} \times \text{MP (in cycles)} \times \text{T}$$

3. Performance evaluation of the memory subsystem.

Considering that read (R) and write (W) accesses have different miss rate and miss penalty:

- *RPI*, *RMR*, *RMP*: average number of accesses per instruction, miss rate, and miss penalty, respectively, for read accesses
- *WPI*, *WMR*, *WMP*: average number of accesses per instruction, miss rate, and miss penalty, respectively, for write accesses

$T_{\text{extra memory}} = (\mathrm{I}\times\mathrm{RPI}\times\mathrm{RMR}\times\mathrm{RMP}\times\mathrm{T}) + (\mathrm{I}\times\mathrm{WPI}\times\mathrm{WMR}\times\mathrm{WMP}\times\mathrm{T})$

where,

$$RMR = \frac{RM}{I+loads} \quad and \quad WMR = \frac{WM}{stores}$$

- If the penalties are identical (RMP = WMP = MP), we can compute the unified miss rate (MR'):

3. Performance evaluation of the memory subsystem.

$$T_{\text{extra memory}} = (I \times (RPI \times RMR \times RMP + WPI \times WMR \times WMP) \times T)$$

$$T_{\text{extra memory}} = (I \times (RPI \times RMR + WPI \times WMR) \times MP \times T)$$

Since $API = RPI + WPI$,

$$T_{\text{extra memory}} = (I \times API \times (\frac{RPI \times RMR + WPI \times WMR}{API}) \times MP \times T)$$

comparing with the general expression,

$$T_{\text{extra memory}} = (I \times API \times MR' \times MP \times T)$$

we conclude that the unified miss rate for read and write accesses is,

$$MR' = \frac{RPI \times RMR + WPI \times WMR}{API} = \frac{RPI \times RMR}{API} + \frac{WPI \times WMR}{API}$$

3. Performance evaluation of the memory subsystem.

On the other hand, if we consider that we have separate caches for instructions (I) and data (D), with different miss rates and miss penalties:

- *IPI*, *IMR*, *IMP*: average number of accesses per instruction, miss rate, and miss penalty, respectively, for instruction cache accesses
- *DPI*, *DMR*, *DMP*: average number of accesses per instruction, miss rate, and miss penalty, respectively, for data cache accesses

$$T_{\text{extra memory}} = (I \times IPI \times IMR \times IMP \times T) + (I \times DPI \times DMR \times DMP \times T)$$

where,

$$IMR = \frac{IM}{I} \;\; and \;\; DMR = \frac{DM}{loads + stores}$$

- Usually *IPI* is equal to 1.
- It is also possible to compute the unified miss rate for instructions and data if the penalties are the same.

3. Performance evaluation of the memory subsystem.

- Finally, we can consider separate caches (I + D) as well as different miss parameters for read and write accesses by combining the corresponding formulas.
- For the instruction cache we only need to consider read accesses.
- Let D_R and D_W represent read and write accesses, respectively, to the data cache:

$$T_{\text{extra memory}} = (I \times IPI \times IMR \times IMP \times T) + (I \times D_R PI \times D_R MR \times D_R MP \times T) + (I \times D_W PI \times D_W MR \times D_W MP \times T)$$

where,

$$IMR = \frac{IM}{I} \;,\; D_R MR = \frac{D_R M}{loads} \;\; and \;\; D_W MR = \frac{D_W M}{stores}$$