Module 9

Caches as a layer of memory hierarchy

Outline

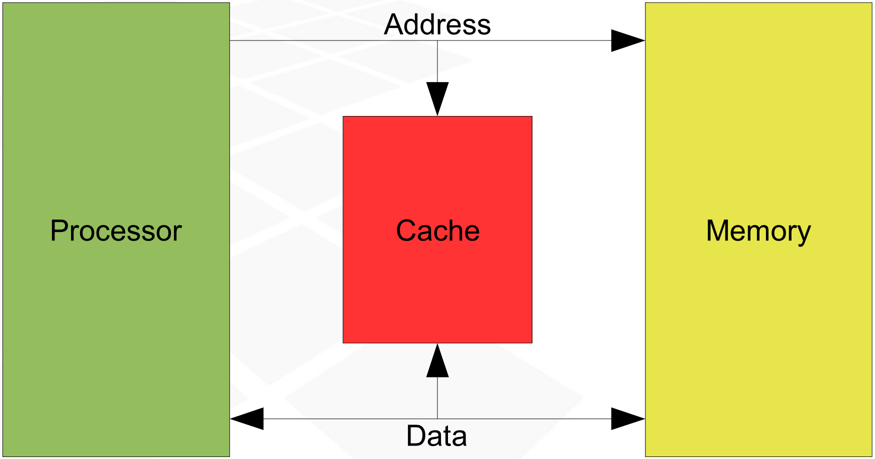
- Principle of operation
- Fully associative cache
- Direct-mapped cache
- Set-associative cache
- Cache efficiency model
- Caches in write cycles
- Inclusive and exclusive cache hierarchies
- Coherency of memory hierarchy

Caches - basics

- Layer of memory hierarchy placed between registers and memory
- Not visible in application programming model
  - In contemporary computers application software may have limited control of cache operation (hints)
- Buffers the references to main memory
- Necessary in every g.p. computer due to growing performance gap between processor and memory
- Introduced for the first time in IBM S/370 circa 1968

E-STUDIA INFORMATYCZNE

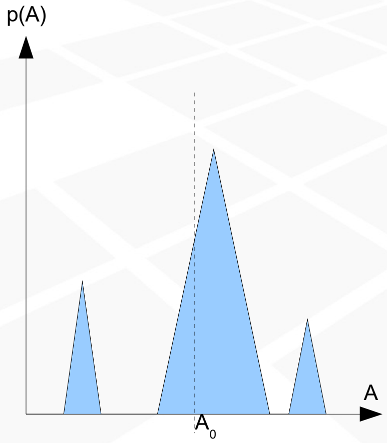
Cache



Cache operation

- During every memory reference the cache is checked for availability of data from a given address
- Data not found – *cache miss*
  - Data is fetched from memory and passed to processor
  - The data is also placed in the cache, together with its address
    - If the cache is already full – some item must be removed to make space for newly arriving data
  - During the next reference to the same address the data will be found in cache
- Data found in cache – *cache hit*
  - Data is fetched from the cache
  - There is no need to access the memory
  - Cache access is significantly faster than memory access

Locality of references



- During a limited period of time, the processor references only a small range of memory addresses
- The plot shows the probability distribution of accesses to addresses in address space during time ($t_{0,}$ $t_0+\Delta t$) with the assumption that in t_0 the processor accessed the address A_0

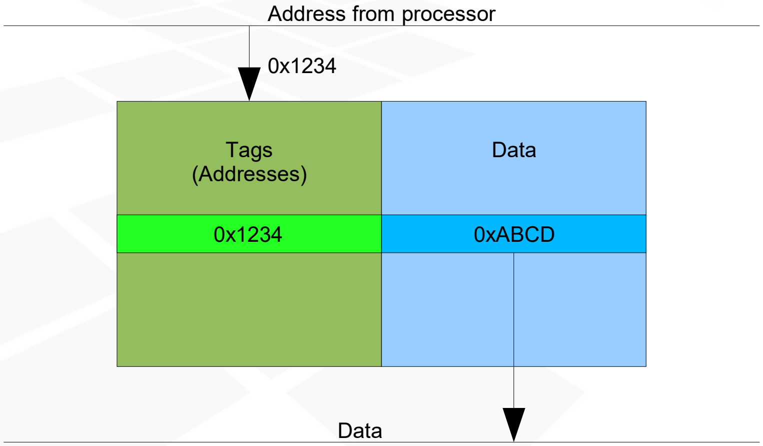
Principle of locality - conclusions

- The range of references is limited
  - The set of addresses accessed by the processor in some limited period of time is called **working set**
  - A relatively small buffer is capable of storing the content of a working set
- The references are performed repetitively to the same addresses
  - The referenced objects should be stored in a fast buffer since the processor will likely access them again
- The references to the consecutive addresses are very likely
  - While filling the buffer, it might be a good idea to fetch the content of several consecutive addresses in advance – they will probably be accessed soon

Fully associative cache

- The most intuitive cache - simple to understand
- Ineffective, difficult to implement
  - Low capacity
  - Not used nowadays
- Built using associative memory
  - No addressing needed for access
  - Data is accessed based on association (matching) of data stored in memory with pattern delivered from outside
  - The memory responds by outputting data matched with input pattern or information that no matching data was found
  - The operation of associative memory may be explained using the example of a phone book
    - We search for a known name
    - We read the phone number
    - We do not care about the position of a record in a phone book

Fully associative cache - model



Fully associative cache - characteristics

- Every cell may store data from any address
  - All the cells must be searched during every access
  - The cache may simultaneously store data from arbitrary addresses – high flexibility when compared with other cache architectures
- Selection of victims (lines to replaced upon miss) – LRU or random
  - LRU algorithm – difficult to implement in hardware
  - Random – varying results in various cases
- Every CAM cell has its own tag comparator
  - Hard to design
  - Low capacity (up to 32 KiB)
- With LRU algorithm, usually as soon as the size of working set exceeds cache capacity, all references result in misses

Cache – design details

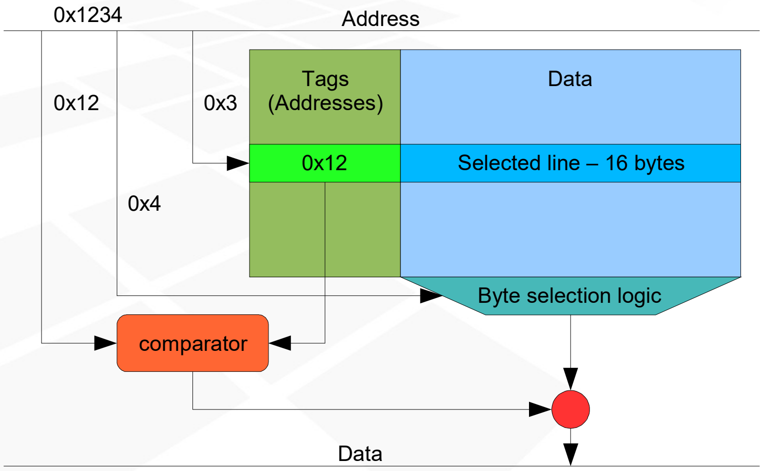
- Data is stored in big units instead of bytes or words
  - The units are called *lines*
  - The unit size is usually equal to 4× size of memory word
    - 32 or 64 bytes in contemporary computers
  - Lines are size-aligned
- The least significant bits of address are used to select a part of line
  - Other, more significant bits are used to check if the requested data is found in cache

Direct mapped cache

- Built using ordinary fast RAM and single equality comparator
  - Fast and very simple to implement
  - Thanks to low complexity it may achieve high capacity
- Simple but non-intuitive operation
  - Least significant bits of address select a part of line
  - “middle”, less significant part of address serves as the address of cache RAM; based on this value single line is selected in every access cycle
  - Eeach line contains address tag and data (and some other tags)
  - Address tag contains the most significant part of address of data stored in data field – the tag is compared with most significant part of address presented by the processor

Direct-mapped cache

Model: 16 lines, 16 bytes each → 256 bytes



Direct-mapped cache - operation

- During every access cycle one line is selected based on middle address bits
- The cache detects hit when the address tag matches the most significant bits of address sent by the processor
- In case of hit data is transferred from cache to the processor
- During miss the selected line is replaced
  - Most significant bits of address are stored in address tag
  - Data fetched from memory is stored in data field

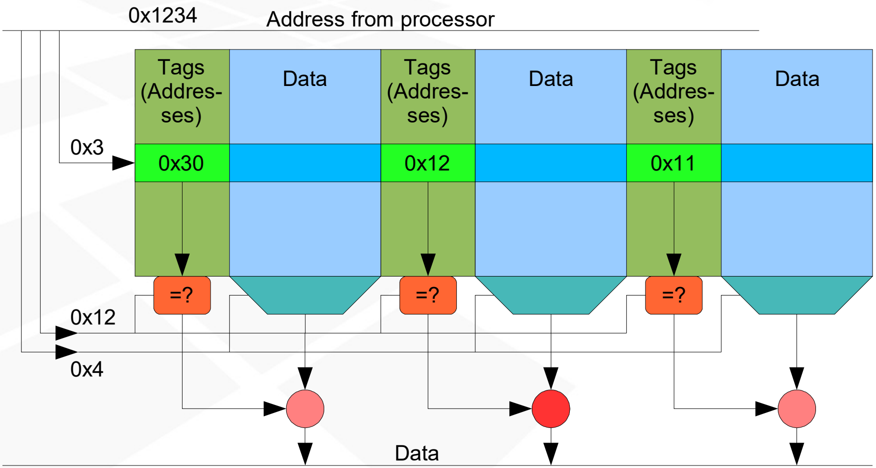
Direct-mapped cache - characteristics

- Low cost, high capacity, high efficiency
- No choice of replacement algorithm
  - Data from a given address may be stored in only one cache line, selected by some part of address
  - The cache cannot simultaneously store data from two addresses with identical middle parts
    - In practice the need to store such data does not appear very frequently
- In case of dense working set (program loop, data table) the direct-mapped cache speeds up the references as long as the working set size is smaller than 2× cache capacity
  - Much better than in case of fully associative cache

Set-associative cache

- Constructed from several direct-mapped caches (the component caches are called *blocks*)
- The number of possible locations of data from a given address is equal to the number of blocks
  - In every cycle the data is searched for in a single line of every block
  - The collection of lines selected by a given address is called a *set*
  - Every set behaves like a tiny fully-associative cache
- The number of blocks is called the cache *associativity*
  - The cache may be described as *two-way*, *four-way*
- Set-associative cache may also be treated as a connection of many small fully-associative caches

Set-associative cache



Set-associative cache - operation

- The cache design must guarantee that the data from a given address is never stored in more than one cache block
- During miss a line selected from the set is replaced
  - To select a line within the set, LRU algorithm may be used when the associativity is low
  - In case of higher associativity - pseudoLRU or random
- The characteristics of set-associative cache is similar to that of a direct-mapped one, but the problem of storing the data with identical middle part of address is solved
  - In that aspect the cache is similar to fully-associative one

Cache organizations - summary

- The most commonly used cache architecture is set-associative
  - Better characteristics than direct-mapped at marginally higher cost
  - If the access time is critical, associativity must be low
    - Higher associativity = slower operation
- Fully-associative caches are no longer used to store code and data
  - They are present in other parts of the computer

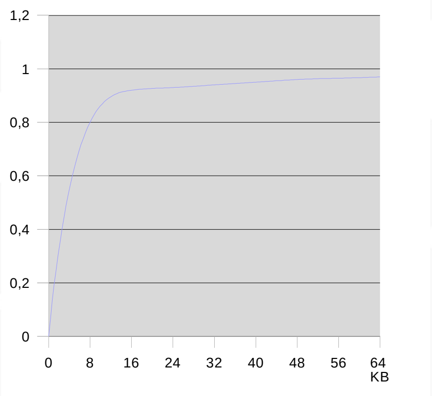
Hit ratio

- Defined as proportion of cache hits to total number of accesses in a given time period

$$h = \frac{n_{cache}}{n_{total}}$$

- Depends on:
  - Cache capacity
  - Cache organization and the replacement algorithm
  - The program being executed
    - For every cache it is possible to write a program yielding h = 0 and another with h arbitrarily close to 1
- Meaningful measurement and comparing the hit ratio of different caches requires defining the common workload set for testing
  - Usually the set contains several different programs, like compiler, text editor, database and computationally-intensive (matrix calculations)

Hit ratio



- The plot shows approximate relation between cache size and hit ratio
- Between 0 and 0.9 the hit ratio depends mainly on cache capacity
  - 0.9 is achieved at the capacity of about 8 KiB
- Above 0.9 the architecture and replacement algorithm become significant factors
  - Higher associativity = higher hit ratio

Hit ratio and cache efficiency

- Hit ratio is only a coefficient
- It does not express the efficiency growth resulting from cache presence
- The cache serves the purpose of optimizing the memory access
- Cache efficiency may be expressed in terms of memory access speed
  - Average throughput in MB/s
  - Average access time in time units per transfer
    - Nanoseconds
    - Processor cycles

Average access time

- Average access time of memory hierarchy composed of single cache and main memory:

  $$t_{AVG} = h \cdot t_{CACHE} + (1 - h) \cdot t_{MEM}$$

  h – cache hit ratio
  m = 1-h – miss ratio
- The cache connected to the processor must be designed in such way that it is able to supply data as fast as the processor requires – without any delays
  - Therefore we assume $t_{CACHE} = 1$

Average access time of memory hierarchy

	tMEM			
h	5	10	50	100
0,9	1,4	1,9	5,9	10,9
0,95	1,2	1,45	3,45	5,95
0,99	1,04	1,09	1,49	1,99

- The table shows average access time in relation to hit ratio and memory access time
- The values reflect the actual processor slowdown in reference to ideal case (single cycle access)
- In the red zone the processor is a few times slower than it should be (according to its clock frequency)

E-STUDIA INFORMATYCZNE

Cache efficiency - conclusions

- Intuitively high hit ratio does not translate to balanced cooperation between the processor and memory hierarchy
  - Speed gap between cache and memory is more significant than hit ratio
- In contemporary computers memory access time may be over 100 times longer than processor cycle
  - The table shows that even very high hit ratio does not compensate for speed gap being that big
- A single cache may compensate the difference in access time not exceeding 10×
- To reduce average access time we should improve the miss response time
  - This may be achieved by replacing the memory with the next cache level + memory
  - This way multilevel cache hierarchy is created

Multilevel caches

- Symbols:
  - L1 – level one cache, closest to the processor
  - L2 – level two cache
  - L3 – level three cache
- Efficiency – 2 cache levels
  - Assume the following parameter values:
    - L1 – access time 1, hit ratio 0.96
    - L2 – access time 5, hit ratio 0.99
    - Main memory – access time 100 cycles
  - Average access time for memory – L2 cache:

$$t_{AVGL2}=h_{L2}\cdot t_{L2}+(1-h_{L2})\cdot t_{MEM}=0.99\cdot 5+0.01\cdot 100=5.95$$

  - Average access time for the whole memory hierarchy:

$$t_{AVG}=h_{L1}\cdot t_{L1}+(1-h_{L1})\cdot t_{AVGL2}=0.96\cdot 1+0.04\cdot 5.95\approx 1.2$$

Multilevel caches

- The required speed of L1 cache is a limiting factor for its capacity and associativity
  - Bigger cache is slower
  - Higher associativity = longer access time
- L2 cache may be slower than L1 (for example 5 times), so:
  - Its associativity may be higher
  - It may be significantly bigger than L1
- If L2 together with memory does not provide satisfactory average access time, L3 cache may be introduced, bigger and slower than L2

Cache operation during write cycles

- So far we have only considered memory reads
- Two possible reactions of cache during write cycles:
  - Transparent write – data is always written to memory and if hit occurs – also to cache
  - Write back – memory writes are deferred until absolutely necessary
- Write back variants:
  - No allocation during write miss – during write miss data is written only to memory, during write hit – only to cache
  - Allocation on write miss – data always written only to the cache
- With write back, removing the line from cache may require writing it from cache to memory

E-STUDIA INFORMATYCZNE

Write policies

Cache write policy	Read hit	Read miss	Write hit	Write miss
Write-thru (WT)	P←C	C←M, P←C	P→C, P→M	P→M
Write back, no write miss allocation (WB/NWA)	P←C	C→M, C←M, P←C	P→C	P→M
Write back, write miss allocation (WB/WA)	P←C	C→M, C←M, P←C	P→C	C→M, C←M, P→C

C – cache, P – processor, M - memory

Red – transactions making the cache not consistent with memory

Blue – victims spilled from cache to memory

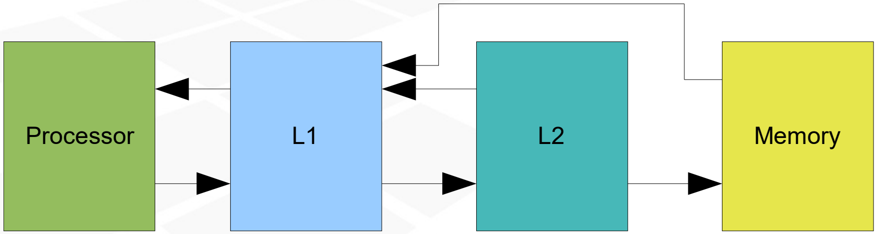
Inclusive caches

- Implemented until approx. 2000
- Data flow: memory ↔ L2 ↔ L1 ↔ processor
- Every object contained in an upper layer is also present in lower layers
- Effective capacity of caches is equal to the capacity of the biggest cache
- For a lower layer cache to operate reasonably, its capacity must be bigger than the capacity of upper layer

Exclusive caches

- Newer approach, after 2000
- Typically L2 cache is filled with the objects removed from L1
  - L2 it is called *victim cache*
    - Victim is a line removed due to replacement policy
- Main data flow: memory→L1↔processor, L1↔L2→memory
- L2 contains objects not present in L1
- During L1 miss causing L2 hit lines are exchanged between L1 and L2
- Effective total capacity is a sum of capacities of all levels
- L2 capacity may be the same as L1
- L2 associativity should be higher than L1
  - Otherwise the victims could not be effectively stored in L2
- Examples – AMD K7, K8; Intel Pentium 4, Core family

Exclusive caches – main data paths



Exclusive caches – L1 read miss handling

- L2 hit:
  - data exchanged between L1 and L2 – victim from L1 goes to L2, taking place of a line moved from L2 to L1
  - no need to use a replacement algorithm for L2
- L2 miss:
  - victim from L2 spilled to memory (if modified)
  - victim from L1 evicted to L2
  - new line read from memory to L1

Coherency of memory hierarchy

- Memory hierarchy must be coherent:
  - Every access to a given address must yield the same data value, regardless of the layer being accessed
- The coherency poses a problem when there is more than one access path to the memory hierarchy, like:
  - Harvard-Princeton processor with separate instruction and data cache
  - Two processors with separate L1
  - Processor with cache and peripheral controller accessing the memory directly (not through cache)
- Coherency does not require the contents of all memory hierarchy levels to be identical. It is enough to guarantee that every access will return the current value.

Methods of achieving coherency

- Invalidation of whole cache content upon detection of an external access to memory
  - Used when caches were small (< 1 KiB)
- Selective invalidation of a line potentially storing data from the address being accessed
  - Used until approx. 1990 with caches up to 8 KiB
- Software invalidation of a whole cache after external access
  - Executed by OS after completion of peripheral transfer
  - Limited application due to efficiency handicap
- Selective change of line state after detecting the access to the address range stored in a line
  - Used presently, complex implementation
  - Require complex coherency protocols

E-STUDIA INFORMATYCZNE

Cache coherency protocols

- State machine implemented for each cache line separately
- Line states:
  - M - modified – line is valid and contains the only valid copy in the whole system (memory content is invalid)
  - E – exclusive – line is valid and contains a valid copy of memory; no other cache contains the same line
  - I – invalid
  - S – shared – line is valid, there is more than one cache containing this line
  - O – owned - line is valid, the same copy in several caches, other caches have state set to S, memory content is invalid
- Protocols – names originate from the set of states
  - MEI, MESI, MOESI
- More states = less invalidations = better efficiency