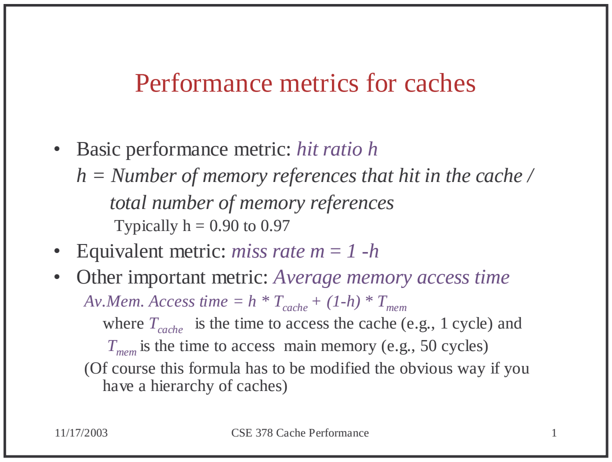


Performance metrics for caches

- Basic performance metric: *hit ratio h*

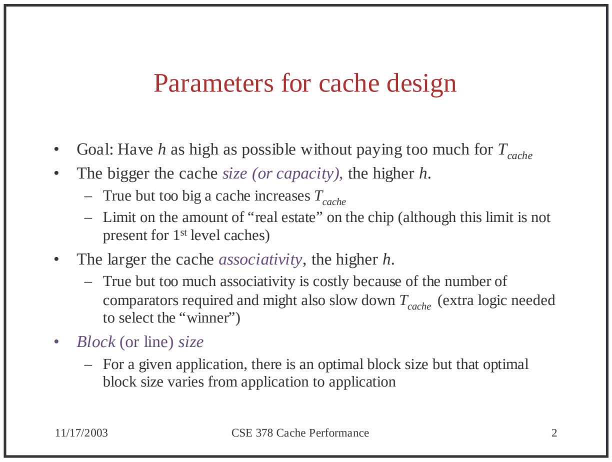
  h = Number of memory references that hit in the cache / total number of memory references

  Typically h = 0.90 to 0.97
- Equivalent metric: *miss rate m = 1 -h*
- Other important metric: *Average memory access time*

  $Av.Mem.\ Access\ time = h * T_{cache} + (1-h) * T_{mem}$

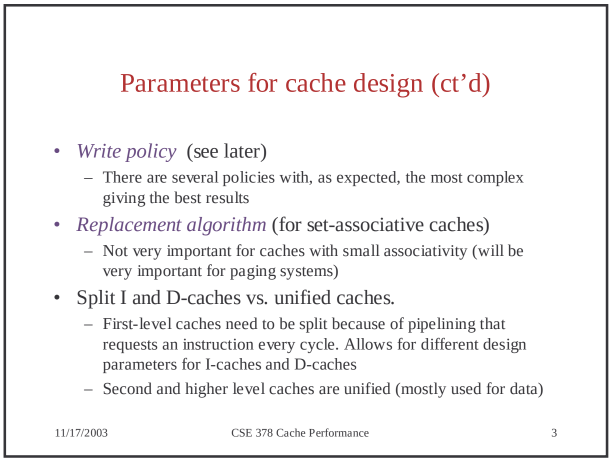
  where T_{cache} is the time to access the cache (e.g., 1 cycle) and T_{mem} is the time to access main memory (e.g., 50 cycles)

  (Of course this formula has to be modified the obvious way if you have a hierarchy of caches)



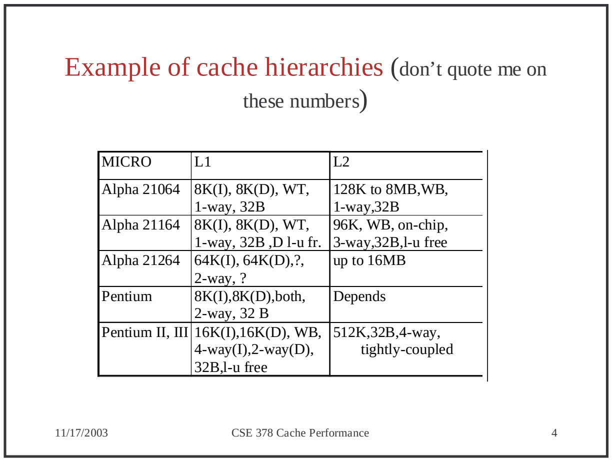
Parameters for cache design

- Goal: Have *h* as high as possible without paying too much for T_{cache}
- The bigger the cache *size (or capacity),* the higher *h*.
  - True but too big a cache increases T_{cache}
  - Limit on the amount of "real estate" on the chip (although this limit is not present for 1st level caches)
- The larger the cache *associativity*, the higher *h*.
  - True but too much associativity is costly because of the number of comparators required and might also slow down T_{cache} (extra logic needed to select the "winner")
- *Block* (or line) *size*
  - For a given application, there is an optimal block size but that optimal block size varies from application to application



Parameters for cache design (ct'd)

- *Write policy* (see later)
  - There are several policies with, as expected, the most complex giving the best results
- *Replacement algorithm* (for set-associative caches)
  - Not very important for caches with small associativity (will be very important for paging systems)
- Split I and D-caches vs. unified caches.
  - First-level caches need to be split because of pipelining that requests an instruction every cycle. Allows for different design parameters for I-caches and D-caches
  - Second and higher level caches are unified (mostly used for data)



Example of cache hierarchies (don't quote me on these numbers)

MICRO	L1	L2
Alpha 21064	8K(I), 8K(D), WT, 1-way, 32B	128K to 8MB,WB, 1-way,32B
Alpha 21164	8K(I), 8K(D), WT, 1-way, 32B ,D l-u fr.	96K, WB, on-chip, 3-way,32B,l-u free
Alpha 21264	64K(I), 64K(D),?, 2-way, ?	up to 16MB
Pentium	8K(I),8K(D),both, 2-way, 32 B	Depends
Pentium II, III	16K(I),16K(D), WB, 4-way(I),2-way(D), 32B,l-u free	512K,32B,4-way, tightly-coupled

Examples (cont’d)

PowerPC 620	32K(I),32K(D),WB 8-way, 64B	1MB TO 128MB, WB, 1-way
MIPS R10000	32K(I),32K(D),l-u, 2-way, 32B	512K to 16MB, 2-way, 32B
SUN UltraSparcIII	32K(I),64K(D),l-u, 4-way	4-8MB 1-way
AMD K7	64k(I), 64K(D)	

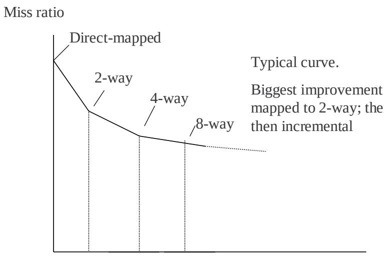
Back to associativity

- Advantages
  - Reduces conflict misses
- Disadvantages
  - Needs more comparators
  - Access time is longer (need to choose among the comparisons, i.e., need of a multiplexor)
  - Replacement algorithm is needed and could get more complex as associativity grows

Replacement algorithm

- None for direct-mapped
- Random or LRU or pseudo-LRU for set-associative caches
  - LRU = "least recently used": means that the entry in the set which has not been used for the longest time will be replaced (think about a stack)

Impact of associativity on performance



Typical curve.

Biggest improvement from direct-mapped to 2-way; then 2 to 4-way then incremental

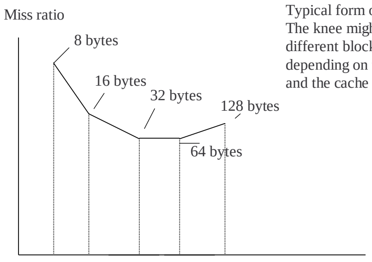
Impact of block size

- Recall block size = number of bytes stored in a cache entry
- On a cache miss the whole block is brought into the cache
- For a given cache capacity, advantages of large block size:
  - decrease number of blocks: requires less real estate for tags
  - decrease miss rate **IF** the programs exhibit good *spatial locality*
  - increase transfer efficiency between cache and main memory
- For a given cache capacity, drawbacks of large block size:
  - increase latency of transfers
  - might bring unused data **IF** the programs exhibit poor spatial locality
  - Might increase the number of conflict/capacity misses

Classifying the cache misses:The 3 C's

- Compulsory misses (cold start)
  - The first time you touch a block. Reduced (for a given cache capacity and associativity) by having large block sizes
- Capacity misses
  - The working set is too big for the ideal cache of same capacity and block size (i.e., fully associative with optimal replacement algorithm). Only remedy: bigger cache!
- Conflict misses (interference)
  - Mapping of two blocks to the same location. Increasing associativity decreases this type of misses.
- There is a fourth C: coherence misses (cf. multiprocessors)

Impact of block size on performance



Typical form of the curve. The knee might appear for different block sizes depending on the application and the cache capacity

Performance revisited

- Recall *Av.Mem. Access time* $= h * T_{cache} + (1-h) * T_{mem}$
- We can expand on T_{mem} as $T_{mem} = T_{acc} + b * T_{tra}$
  - where T_{acc} is the time to send the address of the block to main memory and have the DRAM read the block in its own buffer, and
  - T_{tra} is the time to transfer one word (4 bytes) on the memory bus from the DRAM to the cache, and b is the block size (in words) (might also depend on width of the bus)
- For example, if $T_{acc} = 5$ and $T_{tra} = 1$, what cache is best between
  - C1 ($b1 = 1$) and C2 ($b2 = 4$) for a program with $h1 = 0.85$ and $h2 = 0.92$ assuming $T_{cache} = 1$ in both cases.

Writing in a cache

- On a write hit, should we write:
  - In the cache only (*write-back*) policy
  - In the cache and main memory (or next level cache) (*write-through)* policy
- On a cache miss, should we
  - Allocate a block as in a read (*write-allocate)*
  - Write only in memory *(write-around)*

Write-through policy

- Write-through (aka store-through)
  - On a write hit, write both in cache and in memory
  - On a write miss, the most frequent option is write-around, i.e., write only in memory
- Pro:
  - consistent view of memory ;
  - memory is always coherent (better for I/O);
  - more reliable
    - memory units typically store extra bits with each word to detect/correct errors (“ECC” = Error-correcting code)
    - ECC not required for cache if write-through is used
- Con:
  - more memory traffic (can be alleviated with *write buffers*)

Write-back policy

- Write-back (aka copy-back)
  - On a write hit, write only in cache (
    - requires *dirty* bit to say that value has changed
  - On a write miss, most often *write-allocate* (fetch on miss) but variations are possible
  - We write to memory when a *dirty block* is replaced
- Pro-con reverse of write through

Cutting back on write backs

- In write-through, you write only the word (byte) you modify
- In write-back (when finally writing to memory), you write the entire block
  - But you could have one dirty bit/word so on replacement you’d need to write only the words that are dirty

Hiding memory latency

- On write-through, the processor has to wait till the memory has stored the data
- Inefficient since the store does not prevent the processor to continue working
- To speed-up the process, have *write buffers* between cache and main memory
  - write buffer is a (set of) temporary register that contains the contents and the address of what to store in main memory
  - The store to main memory from the write buffer can be done *while* the processor continues processing
- Same concept can be applied to dirty blocks in write-back policy

Coherency: caches and I/O

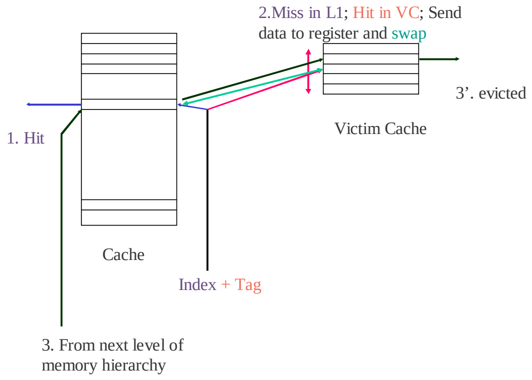
- In general I/O transfers occur directly to/from memory from/to disk
- The problem: what if the processor and the I/O are accessing the same words of memory?
  - Want processor and I/O to have a "coherent" view of memory
- Similar coherence problem arises with multiple CPUs
  - Each CPU accesses the same memory, but keeps its own cache

Preserving coherences with I/O

- What happens for memory to disk
  - With write-through memory is up-to-date. No problem
  - With write-back: memory is not up-to-date. Before I/O is done, need to "purge" cache entries that are dirty and that will be sent to the disk
- What happens from disk to memory
  - The I/O may change a memory location that is currently in the cache
  - The entries in the cache that correspond to memory locations that are read from disk must be *invalidated*
  - Need of a valid bit in the cache (or other techniques)

Reducing Cache Misses with more "Associativity" -- Victim caches

- Example of an "hardware assist"
- Victim cache: Small fully-associative buffer "behind" the cache and "before" main memory
- Of course can also exist if cache hierarchy (behind L1 and before L2, or behind L2 and before main memory)
- Main goal: remove some of the conflict misses in direct-mapped caches (or any cache with low associativity)



2.Miss in L1; Hit in VC; Send data to register and swap

3’. evicted

Victim Cache

1. Hit

Cache

Index + Tag

3. From next level of memory hierarchy

Operation of a Victim Cache

- 1. Hit in L1; Nothing else needed
- 2. Miss in L1 for block at location *b*, hit in victim cache at location *v:* swap contents of *b* and *v* (takes an extra cycle)
- 3. Miss in L1, miss in victim cache : load missing item from next level and put in L1; put entry replaced in L1 in victim cache; if victim cache is full, evict one of its entries.
- Victim buffer of 4 to 8 entries for a 32KB direct-mapped cache works well.