Performance metrics for caches

Basic performance metrics: *hit rate h*.

For a given program its hit rate is defined as:

$$hit\,rate = \frac{number\ of\ memory\ references\ that\ hit\ in\ the\ cache}{total\ number\ of\ memory\ references}$$

We often talk about *miss rate m* which is (1 - h)

The average memory access time is defined as:

$$Average\ memory\ access\ time = h \times T_{cache} + (1 - h) \times T_{mem}$$

where T_{cache} is the time to access the cache and T_{mem} the time to access memory.

Typically, for on chip caches, h = 0.9 (m= 0.1), $T_{cache} = 1$ or 2, and T_{mem} ranges from 10 to 100.

This definition can be easily extended to cache hierarchies:
$Average\ memory\ access\ time = h_1 \times T_{cache-1} + (1 - h_1) \times h_2 \times T_{cache-2} + (1 - h_1) \times (1 - h_2) \times T_{mem}$

Taxonomy of misses (the three C's)

On a single processor system, the misses can be classified as:

1. *Compulsory* (or cold) misses: there will be a miss the first time you "touch" a block in main memory.
2. *Capacity* misses: the cache is not big enough to hold all the blocks you have referenced.
3. *Conflict* misses: two blocks are mapping to the same location (or set) and there is not room to have them in the same set at the same time

In fact, there is 4th "C": *Coherence* misses which occur in multiprocessor systems.

Parameters for cache design

Once you are given a budget in terms of number of transistors, there are many parameters that will influence the way you design a cache.

The goals are to have as great an h as possible without paying too much for T_{cache}

1. *Capacity* (or *Size*): at first glance the bigger the cache, the higher h. This is true most of the time but too big an on-chip cache could lengthen T_{cache} (this has happened only in the rather reent past where clock rates and on-chip cache capacities have increased tremendously)
2. *Associativity*: h is better for high-associativity but higher associativity can lengthen the detection of a hit or miss.
3. *Block size*: for a given application there is an optimal block size (to optimize h) but it is not the same for all applications
4. *Write policy*. There are several alternatives with, as expected, the most complex giving the best results (see later)
5. *Replacement algorithm*. (No problem with direct-mapped!). Not a very important factor for caches.
6. *Separation between I-caches and D-caches*. Almost mandatory for pipelined CPU's and on-chip caches. Not good for second and higher-level caches.

A sample of current caches in recent microprocessors

Micro	On-chip cache (I,D) KB	Line size (bytes)	Ass.	Write policy	Clock MHz
Alpha 21164	8,8	32	1	WT	300
Alpha 21264	64,64	32	2	?	500
Power PC 620	32,32	64	8	WB	133
MIPS R4400	16,16	32	1	WB	150
MIPS R8000	16,16	32	1	WB	75
MIPS R10000	32,32	64	2	WB?	200
Pentium	8,8	32	2	both	66-100
Pentium Pro	8,8	32	4I,2D	?	150-200

Don't quote me on these numbers!

The Alpha 21164 has an on-chip second level cache of 96Kbytes
The Alpha 21264 can support an L2 off-chip of up to 16 MB.

The MIPS R8000 has an enormous second level cache (off-chip) of up to 16 Mbytes. The on-chip data cache is for integer calculations only

Associativity

Advantages:

- Reduce conflict misses

Drawbacks

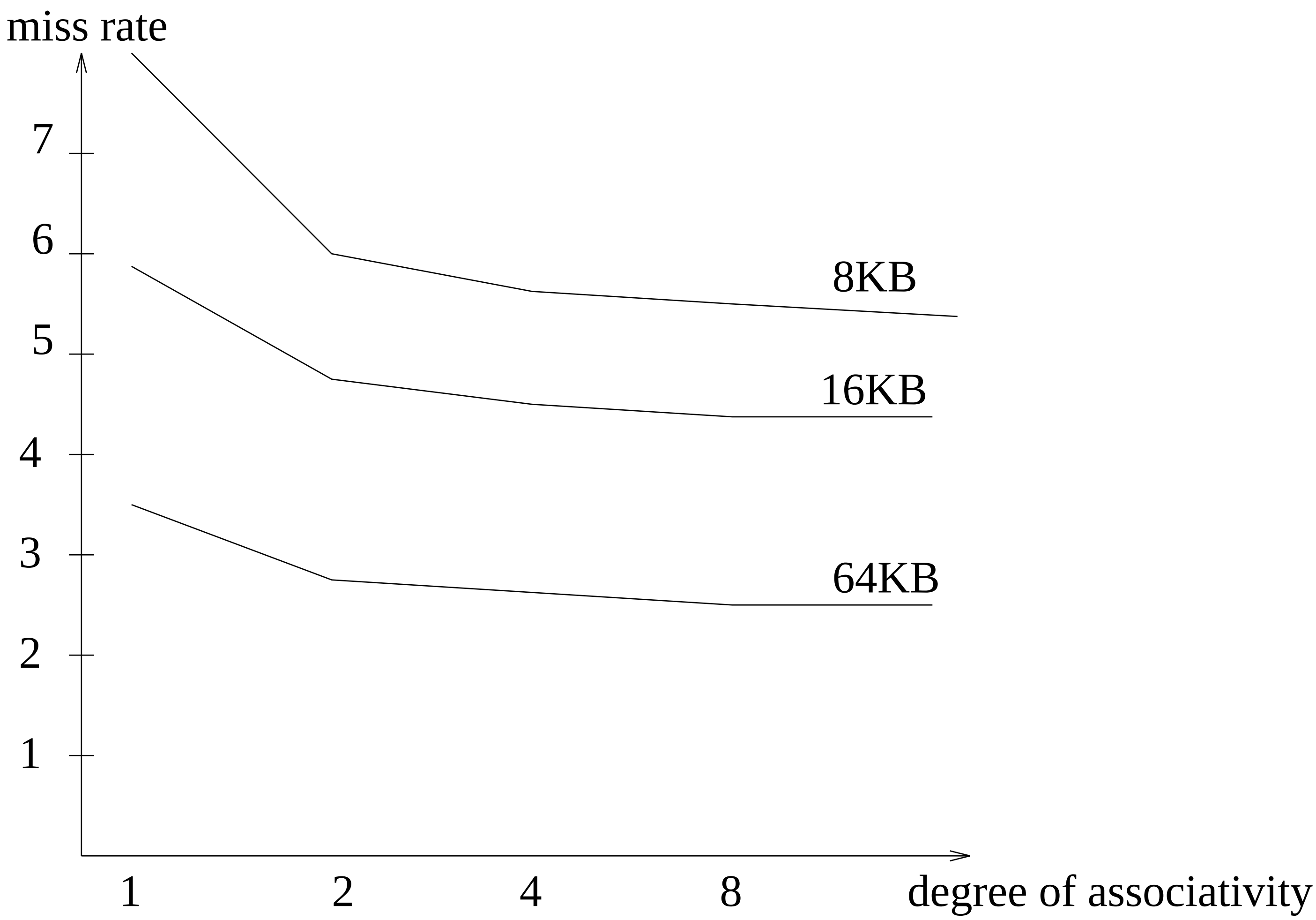
- Needs more comparators
- Access time is longer as set-associativity is larger (more logic in the multiplexor)
- Replacement algorithm is needed and could get more complex as associativity is larger

Performance vs. associativity

Typical curves.

Biggest gain when passing from direct-mapped to two-way.

Gains diminish as caches become larger



Block size

The block (line) size is the number of data bytes stored in one block of the cache.

On a cache miss the whole block is brought into the cache.

For a given cache capacity, large block sizes have advantages:

- Decrease the number of blocks, hence the amount of tag memory
- Decrease miss ratio **IF** the programs exhibit good spatial locality
- Increases transfer efficiency between cache and main memory

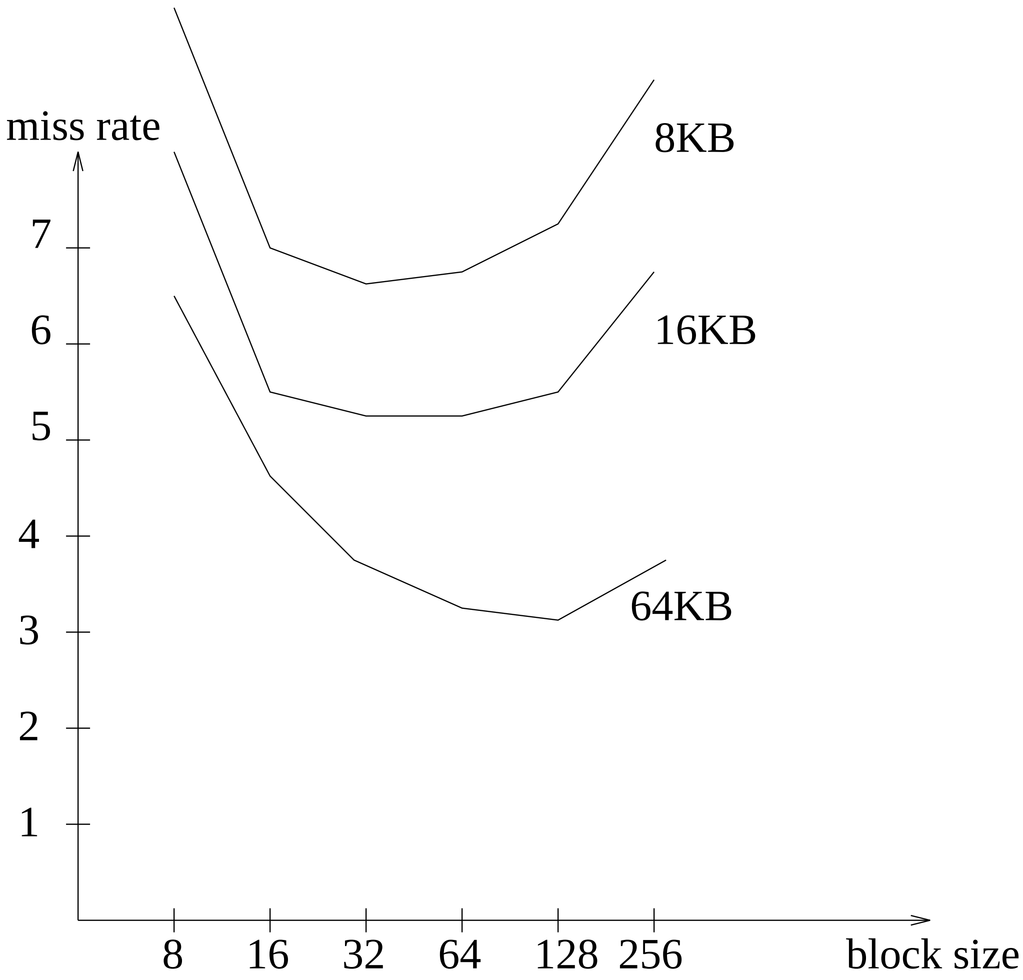
and drawbacks

- Increase latency of transfers
- Might bring unused data **IF** the programs do not exhibit good spatial locality
- Might increase the number of capacity misses

Performance vs. block sizes

Typical curves.

A good design criterion is 16-64 bytes for an on-chip cache and 64-256 for an off-chip cache.



Performance revisited

Recall that

$$Average\ memory\ access\ time = h \times T_{cache} + (1 - h) \times T_{mem}$$

Now we can expand on T_{mem} as

$$T_{mem} = T_{acc} + b \times T_{tra}$$

where T_{acc} is the time it takes to send the address of the block to main memory (DRAM) and have the DRAM access read the block in its own buffer, 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.

For example, if:
$T_{acc} = 5$ and $T_{tra} = 1$, what cache is best between C1 ($b_1 = 1$) and C2 ($b_2 = 4$) for a program where $h_1 = 0.85$ and $h_2 = 0.92$ assuming that $T_{cache} = 1$ in both cases.

Replacement

On a read miss, we bring data in the cache.

What if there is no room? We need to *replace* an item in the cache.

For direct-mapped cache, we have no choice. Replace the item that maps in the same place as the one brought in.

For set-associative (and fully associative) caches, we have several possible policies. However, the block replaced must belong to the same set as the block being brought in.

- random
- FIFO (i.e., replace the oldest one)
- LRU (Least recently used; replace the one not used for the longest time)

For caches, the replacement policy has little influence on performance.

What to do on writes

There are several choices in the case of *write hits* and in the case of *write misses*.

On a write hit, we'll update the value in the cache. But what about main memory.

Two basic choices: Update memory right away (*write-through*) or update memory only when the block is replaced (*write-back*).

Write through

On each write we update both the cache and the memory.

Advantages:

- Memory is always current (*coherent*); easier for I/O see below
- More reliable (no need to have an error detecting device in the cache)

Drawbacks:

- Requires more memory bandwidth since every write has to go to memory

Write back

On a write hit, we update only in the cache.

We only write to memory when the block is replaced

Requires a *dirty bit* per block to indicate if the block has been written to since it was loaded in.

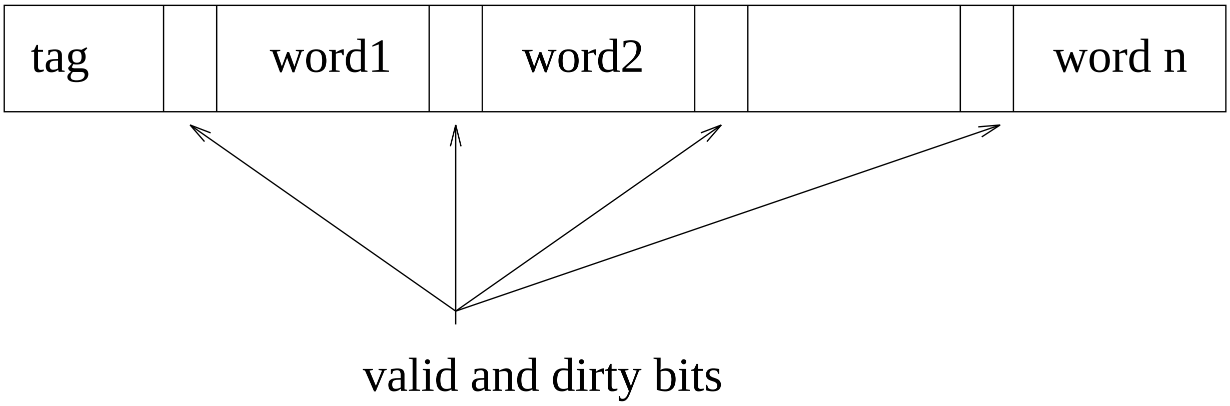
On replacement if the *dirty bit* is set, we write back otherwise we don't (the block is *clean*).

On replacement the *dirty bit* of the new block is reset. It will be set at the first write.

Note that at replacement we write back a block, not a word but see next slide.

Cutting back on write backs – Sector caches

For blocks with multiple words, have one dirty bit and one valid bit per word



Then we only need to write back the *words* that are dirty.

We can extend this design so that instead of “wordn” we have “subblock n” and only subblocks are loaded in on a cache miss.
Advantage: less tag memory.
Drawback: more complex control.

What to do on a write miss

Again , we have choices: write only in memory (no-fetch or *write-around* policy) or bring data in the cache and then write it (*write-allocate*)

Note that this is independent of *write-back* and *write-through*.

On *write-allocate*, *write-back* we need to write back the replaced block if it is dirty. On *write-around* we'll still need to write-back dirty blocks on read misses.

Hiding memory latency

On *write-through*, the processor has to wait till the memory has stored the data (i.e., the cache works at memory speed)

To speed-up the process we can have *write buffers* between the cache and main memory.

A *write buffer* is a (set of) temporary register where the register contains the contents of what to store and where to store it in main memory. The store to memory can be done while the processor continues its work (of course the processor will stall if the write buffer is full and there is another store).

Similarly, dirty blocks in a write back strategy can be stored in a buffer while the block replacing them is being fetched.

I-caches, D-caches and Unified caches

Early on, caches were unified (i.e., stored both instructions and data)

RISC's and pipeline implementations require that an instruction be fetched every cycle; hence split of on-chip caches into I-caches and D-caches.

Otherwise, there would be conflicts for cache access at every memory (load-store) operation.

Large off-chip caches are unified (better utilization of the cache)

I-caches should be simpler because they are read-only (not quite true; how about debuggers?)

Coherency problem: I/O

As we'll see, in general, I/O transfers occur directly to/from memory from/to disk.

With a write-through cache, the data transferred from memory to disk is OK. The data transferred from disk to memory must invalidate corresponding entries in the cache.

With a write-back cache, we have the same problem from disk to memory plus the fact that when the data is transferred from memory to disk, the data must be current. This might require *flushing* part of the cache.