Section 8

Storage hierarchy

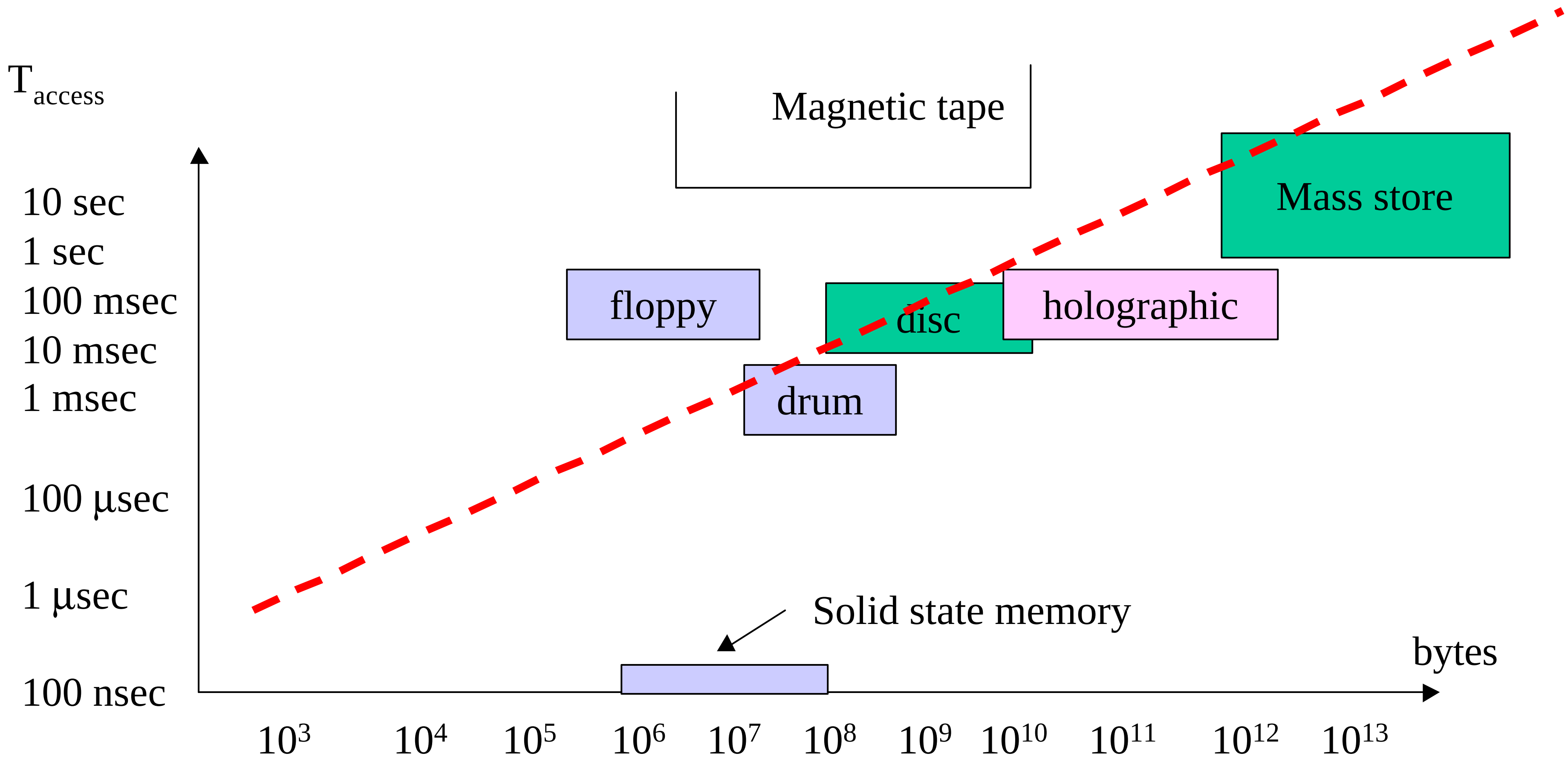
&

caches

Remember this?

(Section 2 --Ms devices)

Secondary Store Ms



Remarks

- Capacity and access time bear a vaguely linear relationship (log-log scale)
- I.e. we can have
  - big, slow memories and
  - small, fast memories.
- Obviously what we want is a . . .

Big, Fast memory!

- How to do it??

Thought ...

- A few things are accessed very frequently
  - tight code loops
  - current record
  - next record in sequence . . .
- Hold them in a small, fast store (memory)

Thought ctd. . .

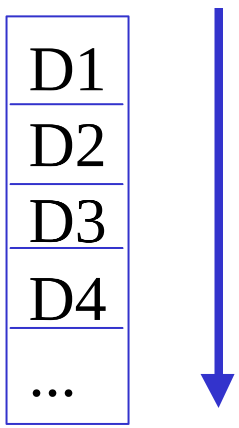
- Some things are accessed somewhat frequently
  - rest of currently-executing procedure or process
  - rest of current file
- put them in a bigger, slower store

Thought ctd. . .

- Some things are accessed hardly at all
- put them in a very slow, big, cheap store

This suggests:

- A hierarchy of storage devices:



Getting slower, bigger, cheaper per byte
as we descend the linear ordering or hierarchy

For instance?

The hierarchy:

- D1: bipolar transistor registers
  - few nsec access time
  - capacity maybe 256 Mb (8 chips)
  - cost maybe few hundred dollars

The hierarchy:

- D2: MOS transistor Mp
  - maybe 10-100 nsec access time
  - size a GB or so
  - cost a few hundred to few thousand dollars

The hierarchy:

- D3: solid state disc (modern)

    or rotating drum (old fashioned)

  – access time microseconds to a few msec
  – capacity a few 10s of GB
  – cost a few thousands of dollars

The hierarchy:

- D4: moving-head disc, for sure!
  - Cost a few hundred to few tens of thousands
  - access time tens to 100s of msec
  - capacity a few GB to a TB

The hierarchy:

- D5: arrays of moving head disc (storage farms)
- parameters: see D4

The hierarchy:

- D6: tapes or robot-arm served farms of chip or magnetic film devices
  - capacity : Terabytes to unbounded (tape warehouses)
  - access times: msec to seconds to hours (tape archives)
  - costs: megadollars

Next thought: time

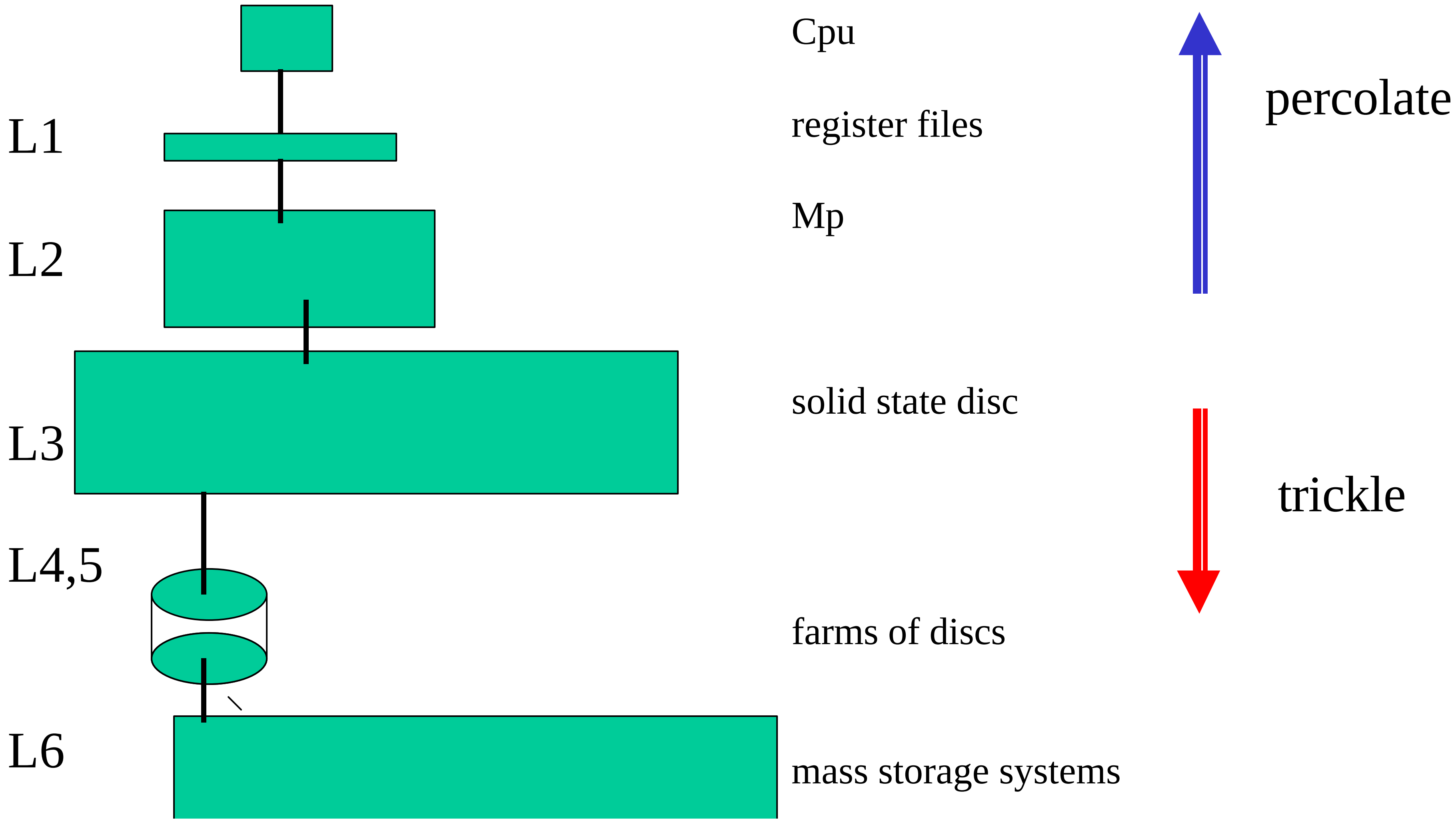
- Yesterday's popular item is today's old hat..
- I.e. items must
  - **rise up** in the hierarchy

    as they get more popular (more frequently used) and
  - **fall down** as they get less frequently used

Details:

- Unit of data: segment or page
- page replacement algorithm:
  - chooses who to boot out of Level n (Ln) to make room for someone new from Level n-1
  - Least Recently Used (LRU) often very good
  - assignment: read about LRU

Story so far:



L1
L2
L3
L4,5
L6
Cpu
register files
Mp
solid state disc
farms of discs
mass storage systems
percolate
trickle

What about programmer's eye view?

- Sounds **very** complex: can we hide it?
- Yes!
- How??

Virtual Memory!

- Programmer sees one huge space of segments and/ or pages
- some of the content is high up, some low down
- it percolates up and trickles down as dictated by the page replacement algorithm
- invisible to application programmer **except for** wildly varying access time
- work done in O/S filesys and VM

Multics war story

The tape warehouse in Secaucus NJ

Making it work better: Caches

- Fundamental performance improver for
  - CPUs (instruction and operand caches)
  - World Wide Web
  - storage hierarchies etc etc

The idea

- Identify the things I need often and save them automatically in a small fast (cache) memory

- supply them quickly from the cache instead of slowly from the original source

Cache: How does it work?

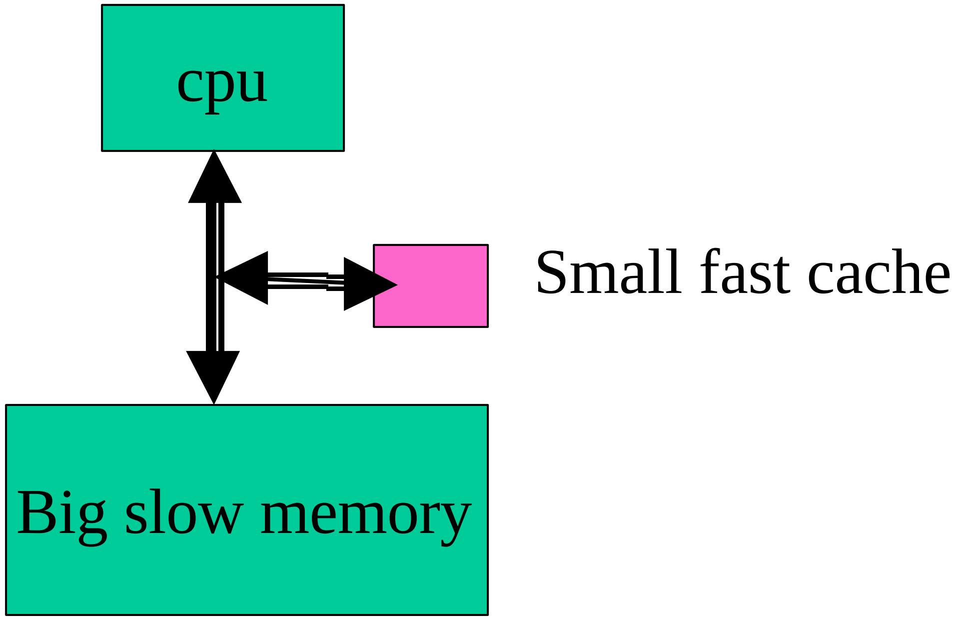
- a kind of associative store
- ordinary memory access protocol:
  - you give it Y
  - it gives you c(Y) = Z
- associative store protocol:
  - you give it Z
  - it gives you Y such that c(Y) = Z

Associative Store refined:

- You may not care where Z is in the store
- You do care
  - 1] if Z is present somewhere, and if so:
  - 2] a value associated with *tag* Z called the *payload*
- Huh?

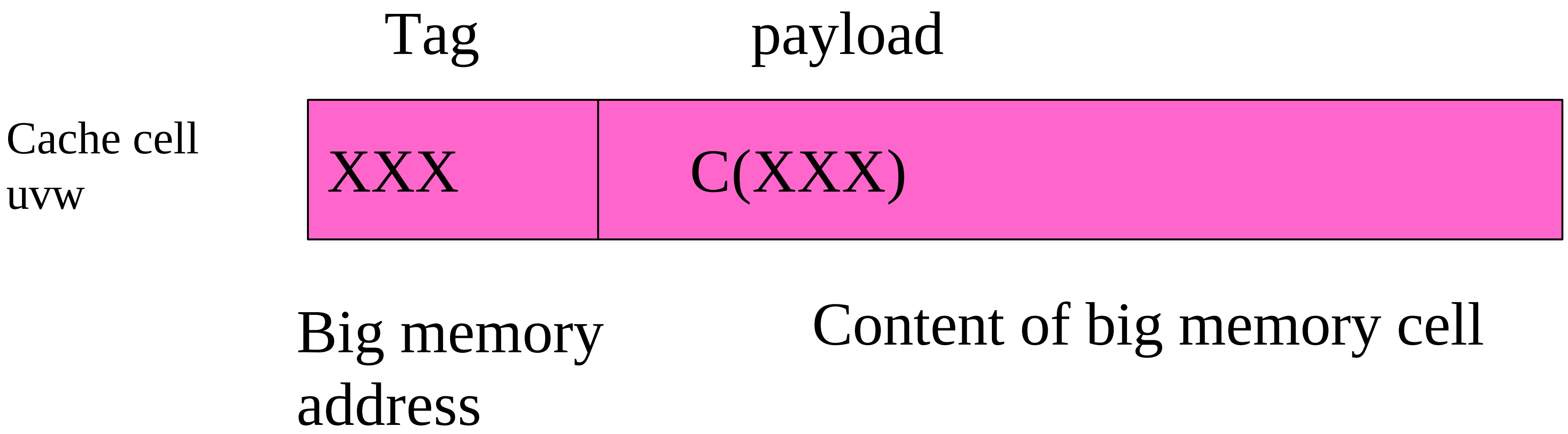
Associative store as cache

- Suppose we have a big slow memory and we have cached some popular items



- now we need c(XXX). We need to know:

- 1] is c(XXX) in the cache??
- 2] If yes, what is it?
- So, in the cache we stored:



Retrieval from cache

- 1] present tag value XXX
  - cache searches EVERY cell for tagvalue = XXX
  - if it finds XXX in some cell x , it returns the corresponding payload value
  - but payload value = c(XXX) where XXX is an address in the Big Slow Memory

Trick: using an ordinary memory as a cache

- Hashcode XXX (the address in Big Memory) to compute x (the address in cache) uniquely
- simple hash function:
  - use the last n bits of X to be x (Simple & fast!)
  - note
    - sizeof(X) >> sizeof(x)
    - homomorphism: many XXX map to the same x

- Which one is it?
  - Store the remaining (N-n) bits of XXX as a tag in c(x) -- tells us **which** XXX is present in x
- also store c(XXX) in x:

Cache cell x	First N- n bits of XXX	C(XXX) in big memory

Example:

Addr	content
123	abc
133	def

Big Memory

Addr	content
0	
1	
2	
3	12 abc
4	
5	
…	
9	

cache

Cache's big problem

- Rapidly changing data …
- cache *consistency*
- provide a validity bit in each cache word, turned OFF if we know the data is stale...

Exploit spatial locality reference

- What is it?
  - If you just touched word X, you will probably next touch one of X-2, X-1, X, X+1, X+2
- So don't just put X, c(X) in the cache
- put a block (4-8 words) of pairs

X-2	c(X-2)
. . .	
X+2	c(X+2)

in the cache

PHBreak

- Read P&H Sections 7.2 & 7.3 for
  - discussion of cache basics
  - formulas for calculating performance and a real example (DEC workstation using MIPS chip)