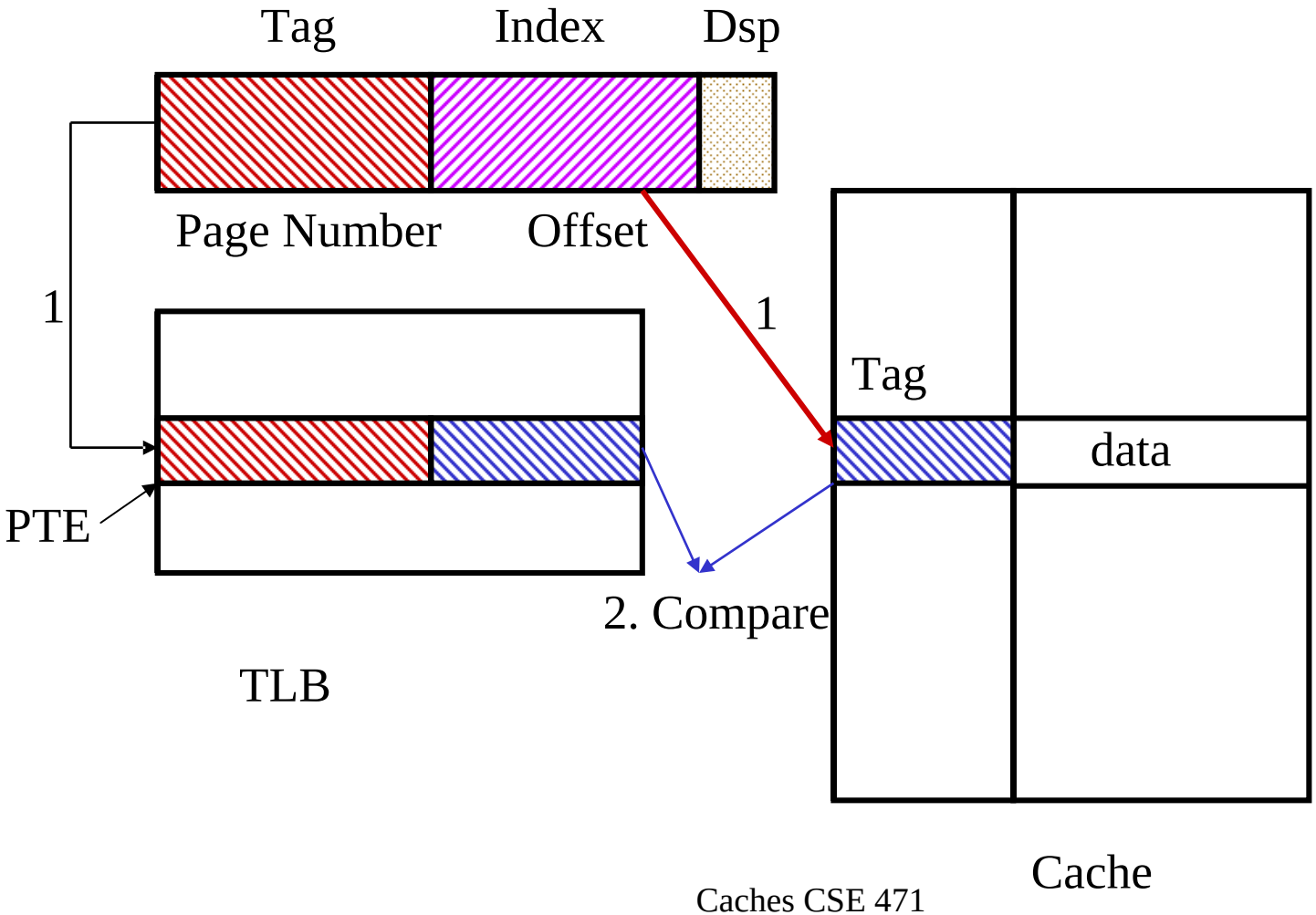
Speeding up L1 Access

- Cache can be (speculatively) accessed in parallel with TLB if its indexing bits are not changed by the virtual-physical translation
- Cache access (for reads) is pipelined:
  - Cycle 1: Access to TLB and access to L1 cache (read data at given index)
  - Cycle 2: Compare tags and if hit, send data to register

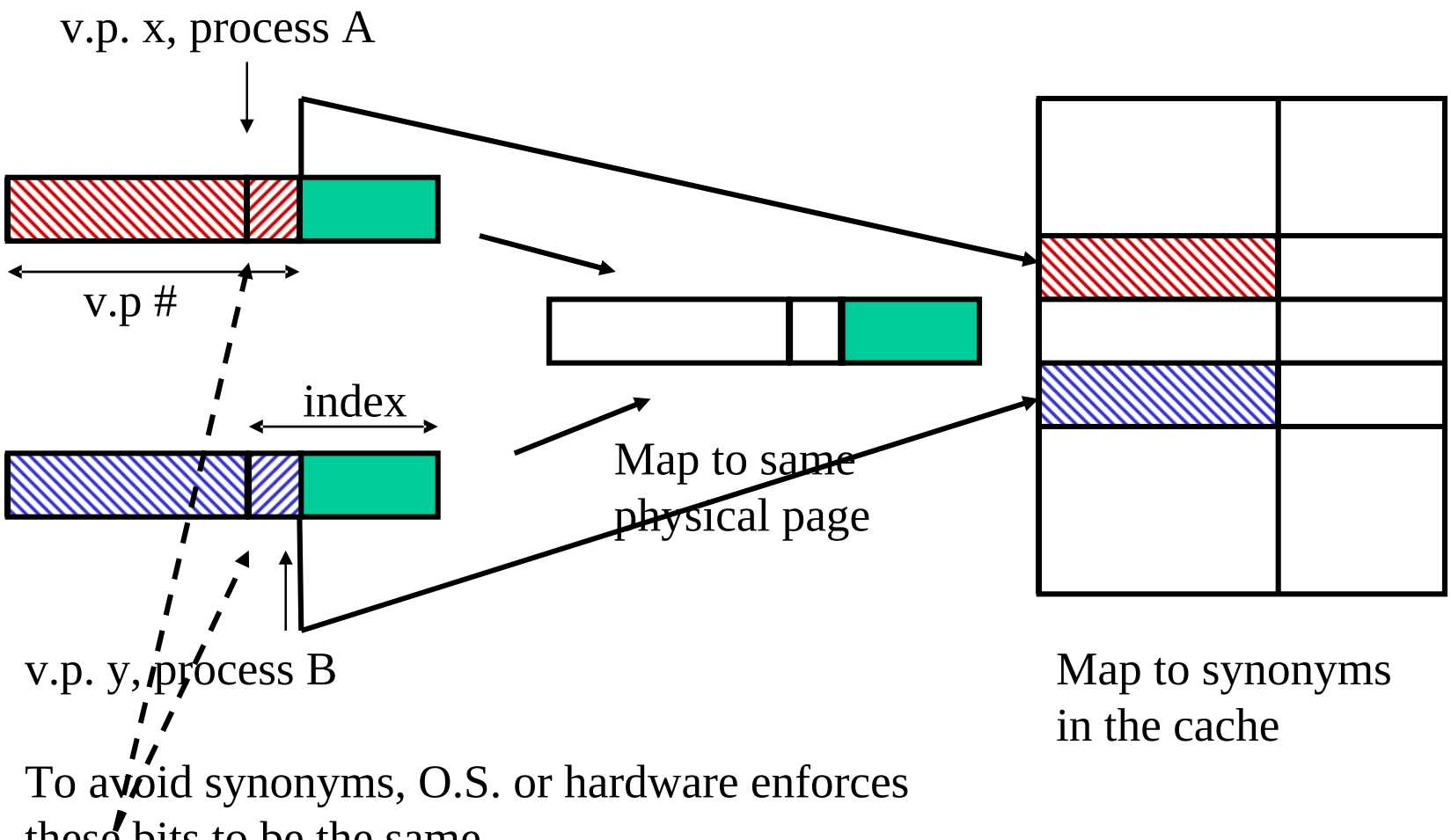
Virtually Addressed Cache



"Virtual" Caches

- Previous slide: Virtually addressed, physically tagged
  - Can be done for small L1, i.e., capacity < (page * ass.)
  - Can be done for larger caches if O.S. does a form of page coloring (see later) such that "index" is the same for synonyms (see below)
  - Can also be done more generally (complicated but can be elegant)
- Virtually addressed, virtually tagged caches
  - Synonym problem (2 virtual addresses corresponding to the same physical address). Inconsistency since the same physical location can be mapped into two different cache blocks
  - Can be handled by software (disallow it) or by hardware (with "pointers")
  - Use of PID's to only partially flush the cache

Synonyms


v.p. x, process A
v.p #
index
Map to same
physical page
v.p. y, process B
Map to synonyms
in the cache
To avoid synonyms, O.S. or hardware enforces
these bits to be the same

Obvious Solutions to Decrease Miss Rate

- Increase cache capacity
  - Yes, but the larger the cache, the slower the access time
  - Solution: Cache hierarchies (even on-chip)
  - Increasing L2 capacity can be detrimental on multiprocessor systems because of increase in coherence misses
- Increase cache associativity
  - Yes, but "law of diminishing returns" (after 4-way for small caches; not sure of the limit for large caches)
  - More comparisons needed, i.e., more logic and therefore longer time to check for hit/miss?
  - Make cache look more associative than it really is (see later)

What about Cache Line Size?

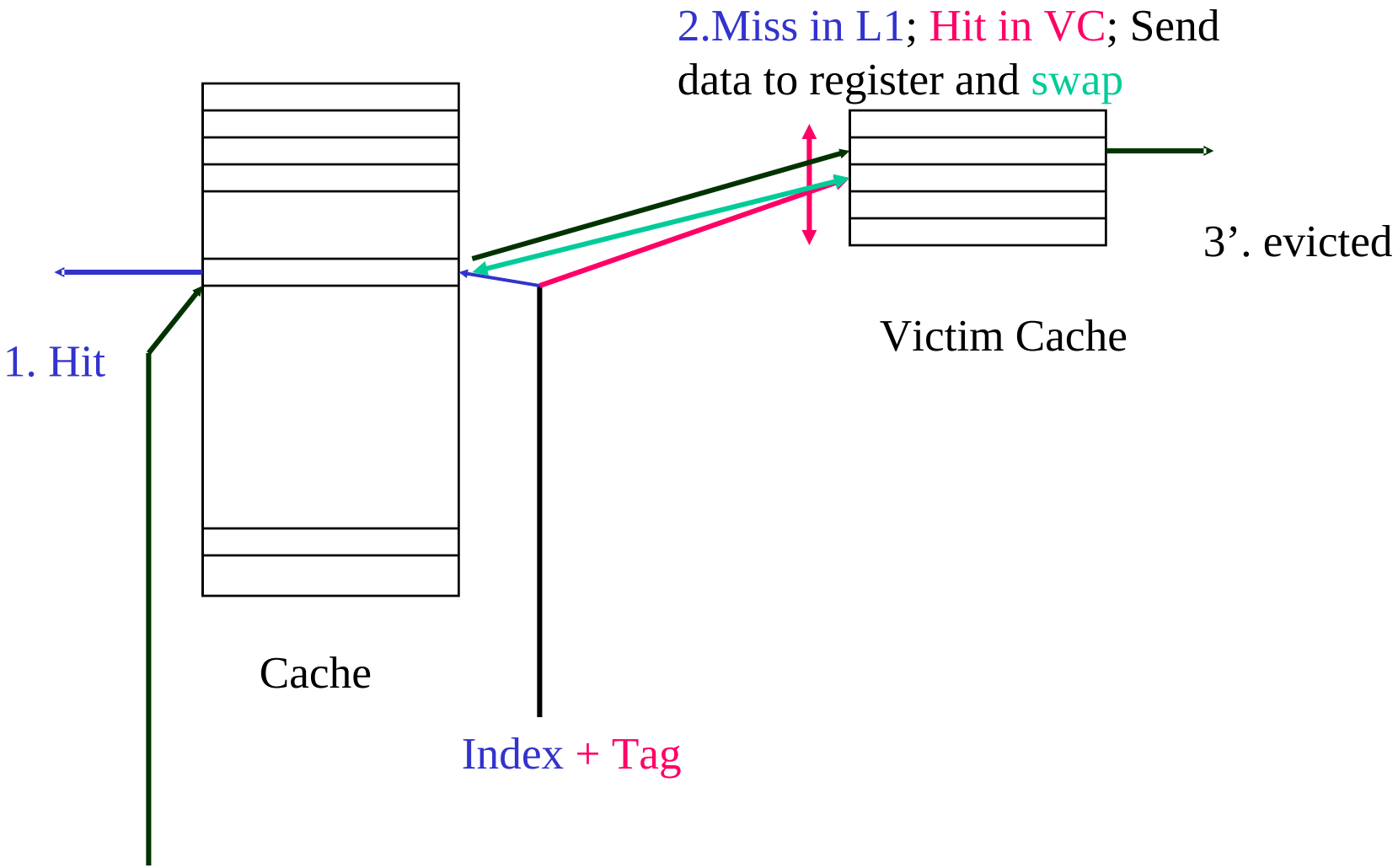
- For a given application, cache capacity and associativity, there is an optimal cache line size
- Long cache lines
  - Good for spatial locality (code, vectors)
  - Reduce compulsory misses (implicit prefetching)
  - But takes more time to bring from next level of memory hierarchy (can be compensated by "critical word first" and subblocks)
  - Increase possibility of fragmentation (only fraction of the line is used – or reused)
  - Increase possibility of false-sharing in multiprocessor systems

Impact of Associativity

- "Old" conventional wisdom
  - Direct-mapped caches are faster; cache access is bottleneck for on-chip L1; make L1 caches direct mapped
  - For on-board (L2) caches, direct-mapped are 10% faster.
- "New" conventional wisdom
  - Can make 2-way set-associative caches fast enough for L1. Allows larger caches to be addressed only with page offset bits
  - Looks like time-wise it does not make much difference for L2/L3 caches, hence provide more associativity (but if caches are extremely large there might not be much benefit)

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

- Victim cache: Small fully-associative buffer "behind" the L1 cache and "before" the L2 cache
- Of course can also exist "behind" L2 and "before"L3 or main memory
- Main goal: remove some of the conflict misses in L1 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 line 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.

Bringing more Associativity -- Column-associative Caches

- Split (conceptually) direct-mapped cache into two halves
- Probe first half according to index. On hit proceed normally
- On miss, probe 2nd half ; If hit, send to register and swap with entry in first half (takes an extra cycle)
- On miss (on both halves) go to next level, load in 2nd half and swap
- Slightly more complex than that (need one extra bit in the tag)

Skewed-associative Caches

- Have different mappings for the two (or more) banks of the set-associative cache
- First mapping conventional; second one “dispersing” the addresses (XOR a few bits)
- Hit ratio of 2-way skewed as good as 4-way conventional.

Reducing Conflicts --Page Coloring

- Interaction of the O.S. with the hardware
- In caches where the cache size > page size * associativity, bits of the physical address (besides the page offset) are needed for the index.
- On a page fault, the O.S. selects a mapping such that it tries to minimize conflicts in the cache .

Options for Page Coloring

- Option 1: It assumes that the process faulting is using the whole cache
  - Attempts to map the page such that the cache will access data as if it were by virtual addresses
- Option 2: do the same thing but hash with bits of the PID (process identification number)
  - Reduce inter-process conflicts (e.g., prevent pages corresponding to stacks of various processes to map to the same area in the cache)
- Implemented by keeping “bins” of free pages

Tolerating/hiding Memory Latency

- One particular technique: prefetching
- Goal: bring data in cache *just in time* for its use
  - Not too early otherwise cache pollution
  - Not too late otherwise "hit-wait"cycles
- Under the constraints of (among others)
  - Imprecise knowledge of instruction stream
  - Imprecise knowledge of data stream
- Hardware/software prefetching
  - Works well for regular stride data access
  - Difficult when there are pointer-based accesses

Why, What, When, Where

- Why?
  - cf. goals: Hide memory latency and/or reduce cache misses
- What
  - Ideally a semantic object
  - Practically a cache line, or a sequence of cache lines
- When
  - Ideally, just in time.
  - Practically, depends on the prefetching technique
- Where
  - In the cache or in a prefetch buffer

Hardware Prefetching

- Nextline prefetching for instructions
  - Bring missing line and the next one (if not already there)
- OBL "one block look-ahead" for data prefetching
  - As *Nextline* but with more variations -- e.g. depends on whether prefetching was successful the previous time
- Use of special assists:
  - Stream buffers, i.e., FIFO queues to fetch consecutive lines (good for instructions not that good for data);
  - Stream buffers with hardware stride detection mechanisms;
  - Use of a reference prediction table
  - "Content-less" prefetching etc.

Memory Hierarchy in Power 4/5

Cache	Capacity	Associativity	Line size	Write policy	Repl. alg	Comments
L1 I-cache	64 KB	Direct/2-way	128 B		LRU	Sector cache (4 sectors)
L1 D-cache	32 KB	2-way/4-way	128 B	Write-through	LRU	
L2 Unified	1.5 MB/2 MB	8-way/10-way	128 B	Write-back	Pseudo-LRU	
L3	32 MB/36MB	8-way/12-way	512 B	Write-back	?	Sector cache (4 sectors)

Latency	P4 (1.7 GHz)	P5 (1.9 GHz)
L1 (I and D)	1	1
L2	12	13
L3	123	87
Main Memory	351	220

Sequential & Stride Prefetching in Power 4/5

- When prefetch line i from L2 to L1
  - Prefetch lines (i+1) and (i+2) from L3 to L2
  - Preftch lines (i=3),…(i+6) from main memry to L3