Principle of Locality: Memory Hierarchies

- Text and data are not accessed randomly
- Temporal locality
  - Recently accessed items will be accessed in the near future (e.g., code in loops, top of stack)
- Spatial locality
  - Items at addresses close to the addresses of recently accessed items will be accessed in the near future (sequential code, elements of arrays)
- Leads to memory hierarchy at two main interface levels:
  - Processor - Main memory -> Introduction of caches
  - Main memory - Secondary memory -> Virtual memory (paging systems)

Processor - Main Memory Hierarchy

- Registers: Those visible to ISA + those renamed by hardware
- (Hierarchy of) Caches: plus their enhancements
  - Write buffers, victim caches etc…
- TLB's and their management
- Virtual memory system (O.S. level) and hardware assists (page tables)
- Inclusion of information (or space to gather information) level per level
  - Almost always true

Questions that Arise at Each Level

- What is the unit of information transferred from level to level ?
  - Word (byte, double word) to/from a register
  - Block (line) to/from cache
  - Page table entry + misc. bits to/from TLB
  - Page to/from disk
- When is the unit of information transferred from one level to a lower level in the hierarchy?
  - Generally, on demand (cache miss, page fault)
  - Sometimes earlier (prefetching)

Questions that Arise at Each Level (c'ed)

- Where in the hierarchy is that unit of information placed?
  - For registers, directed by ISA and/or register renaming method
  - For caches, in general in L1
    - Possibility of hinting to another level (Itanium) or of bypassing the cache entirely, or to put in special buffers
- How do we find if a unit of info is in a given level of the hierarchy?
  - Depends on mapping;
  - Use of hardware (for caches/TLB) and software structures (page tables)

Questions that Arise at Each Level (c'ed)

- What happens if there is no room for the item we bring in?
  - Replacement algorithm; depends on organization
- What happens when we change the contents of the info?
  - i.e., what happens on a write?

Caches (on-chip, off-chip)

- Caches consist of a set of entries where each entry has:
  - line (or block) of data: information contents
  - tag: allows to recognize if the block is there
  - status bits: valid, dirty, state for multiprocessors etc.
- Cache Geometries
  - Capacity (or size) of a cache: number of lines * line size, i.e., the cache metadata (tag + status bits) is not counted
  - Associativity
  - Line size

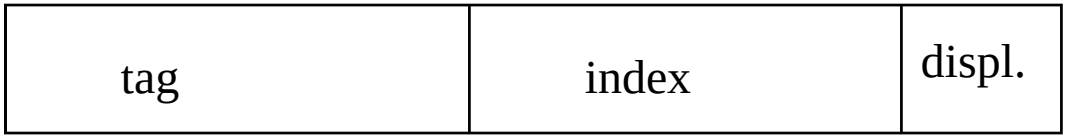
Cache Organizations

- Direct-mapped
- Set-associative
- Fully-associative

Cache Hit or Cache Miss?

- How to detect if a memory address (a byte address) has a valid image in the cache:
- Address is decomposed in 3 fields:
  - *line offset* or *displacement* (depends on line size)
  - *index* (depends on number of sets and set-associativity)
  - *tag* (the remainder of the address)
- The tag array has a width equal to *tag*

Hit Detection



Example: cache capacity *C*, line size *b*

Direct mapped: displ = $\log_2 b$; index = $\log_2(C/b)$; tag = 32 -index - displ

N -way S.A: displ = $\log_2 b$; index = $\log_2(C/bN)$; tag = 32 -index - displ

So what does it mean to have 3-way (N=3) set-associativity?

Replacement Algorithm

- None for direct-mapped
- Random or LRU or pseudo-LRU for set-associative caches
  - Not an important factor for performance for low associativity. Can become important for large associativity and large caches

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 higher level cache) (write-through) policy
- On a write miss, should we
  - Allocate a block as in a read (write-allocate)
  - Write only in memory (write-around)

The Main Write Options

- 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
  - Pro: consistent view of memory (better for I/O); no ECC required for cache
  - Con: more memory traffic (can be alleviated with write buffers)
- Write-back (aka copy-back)
  - On a write hit, write only in cache (requires dirty bit)
  - On a write miss, most often write-allocate (fetch on miss) but variations are possible
  - Pro-con reverse of write through

Classifying the Cache Misses: The 3 C's

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

Example of Cache Hierarchies

Processor	I-Cache	D-Cache	L2
Alpha 21064	(8KB, 1, 32)	(8KB, 1, 32) WT	Off-chip (2MB,1,64)
Alpha 21164	(8KB, 1, 32)	(8KB, 1, 32) WT	(96KB,3,64) WB
Alpha 21264	(64KB,2,64)	(64KB,2,64)	0ff-chip(16MB,1,64)
Pentium	(8KB,2,32)	(8KB,2,32) WT/WB	Off-chip up to 8MB
Pentium II	(16KB,2,32)	(16KB,2,32) WB	“glued”(512KB,8,32) WB
Pentium III	(16KB,2,32)	(16KB,4,32) WB	(512KB,8,32) WB
Pentium 4	Trace cache 96KB	(16KB,4,64) WT	(1MB, 8,128) WB

Cache Performance

- CPI $_{\text{contributed by cache}}$ = CPI_c

  = miss rate * number of cycles to handle the miss

- Another important metric

  Average memory access time = cache hit time * hit rate

  + Miss penalty * (1 - hit rate)

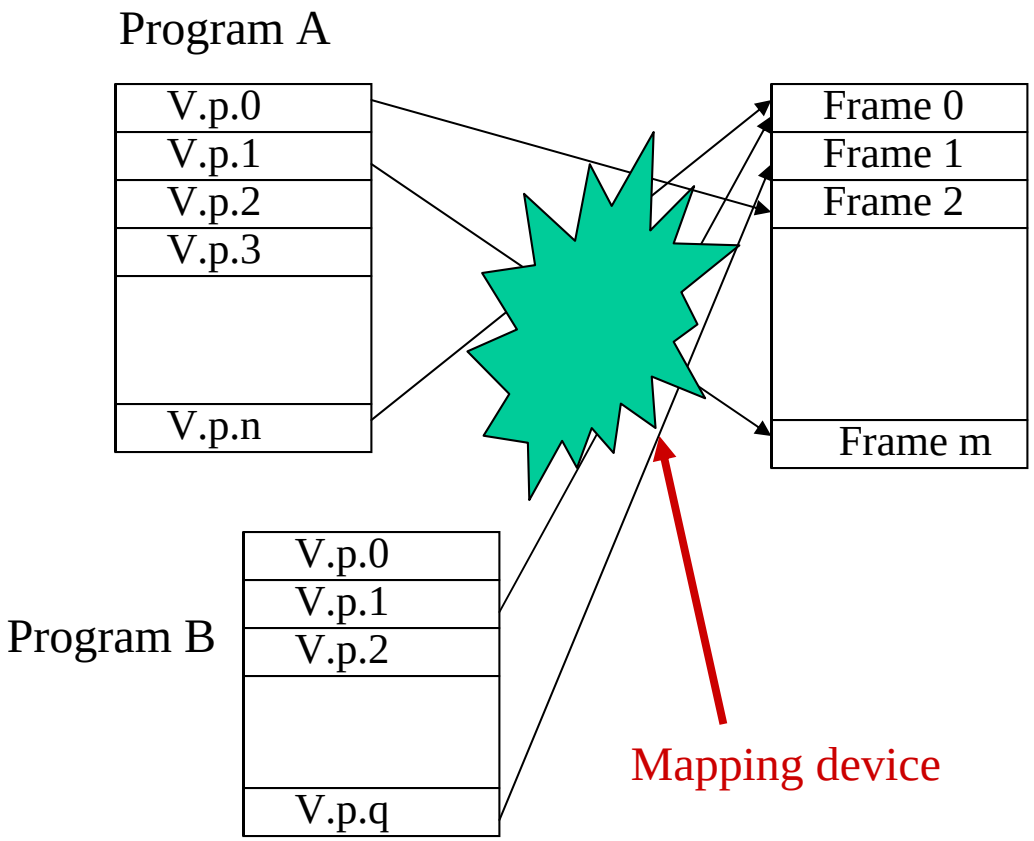
Improving Cache Performance

- To improve cache performance:
  - Decrease miss rate without increasing time to handle the miss (more precisely: without increasing average memory access time)
  - Decrease time to handle the miss w/o increasing miss rate
- A slew of techniques: hardware and/or software
  - Increase capacity, associativity etc.
  - Hardware assists (victim caches, write buffers etc.)
  - Tolerating memory latency: Prefetching (hardware and software), lock-up free caches
  - O.S. interaction: mapping of virtual pages to decrease cache conflicts
  - Compiler interactions: code and data placement; tiling

Improving L1 Cache Access Time

- Processor generates virtual addresses
- Can cache have virtual address tags?
  - What happens on a context switch?
- Can cache and TLB be accessed in parallel?
  - Need correspondence between page size and cache size + associativity
- What about virtually addressed physically tagged caches?

A Brief Review of Paging



Physical memory

Note: In general n, q >> m

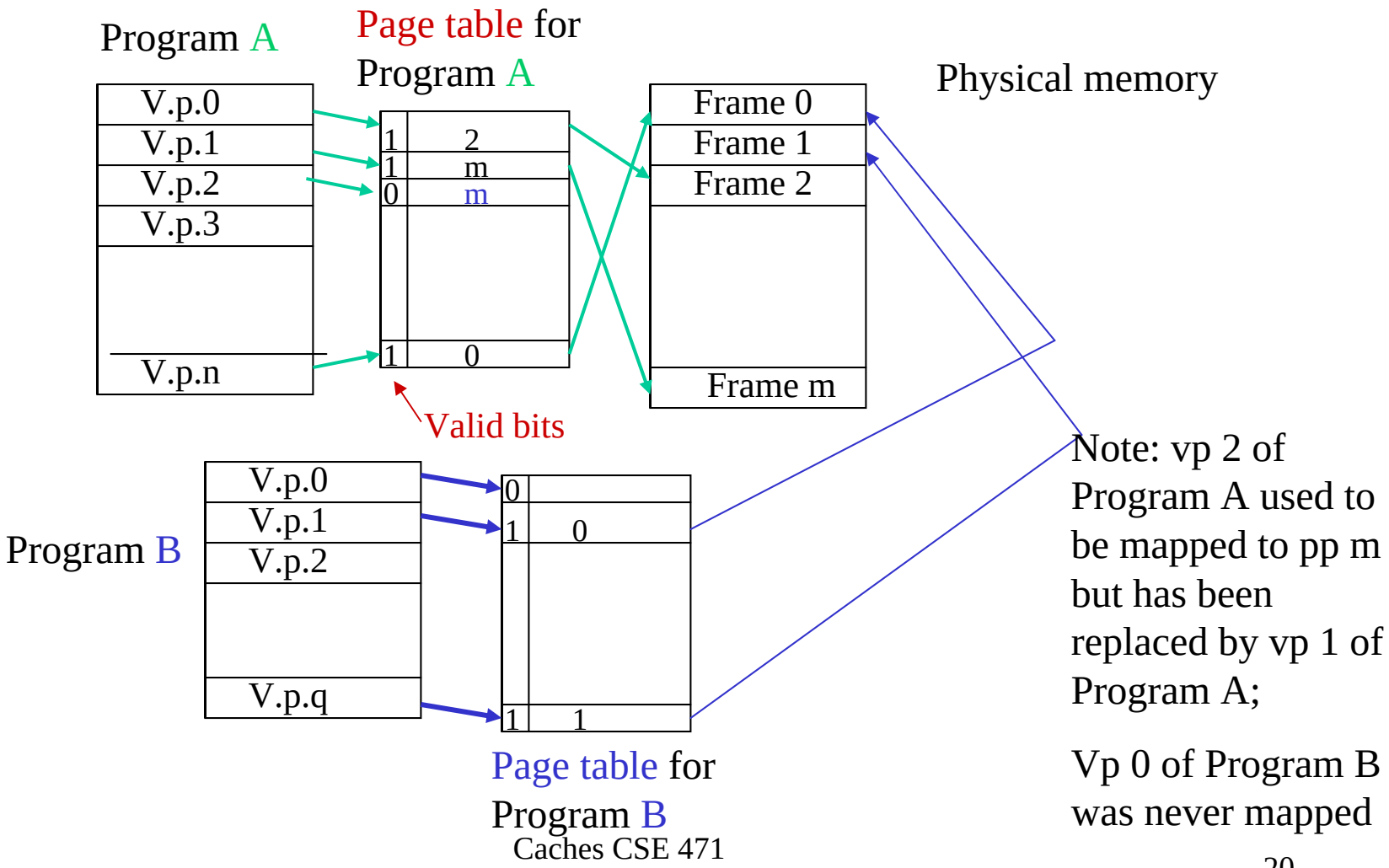
Not all virtual pages of a program are mapped at a given time

In this example, programs A and B share frame 0 but with different virtual page numbers

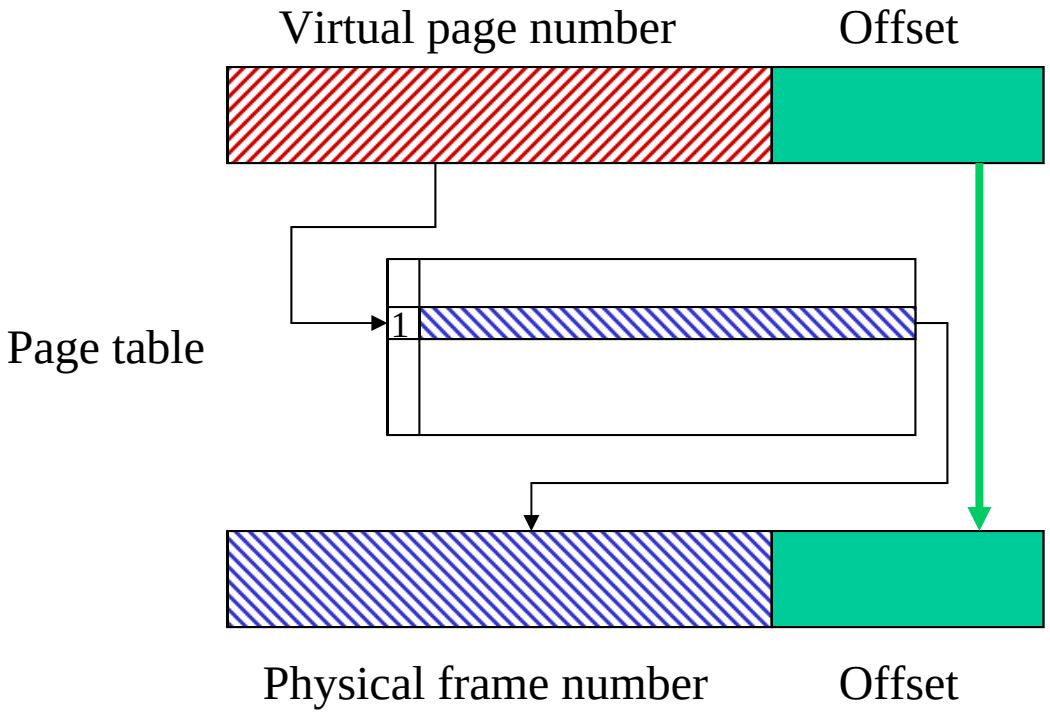
Mapping Device: Page Tables

- Page tables contain page table entries (PTE):
  - Virtual page number (implicit/explicit), physical page number,valid, protection, dirty, use bits (for LRU-like replacement), etc.
- Hardware register points to the page table of the running process
- Earlier system: contiguous (in virtual space) page tables; Now, multi-level page tables
- In some systems, inverted page tables (with a hash table)
- In all modern systems, page table entries are cached in a TLB

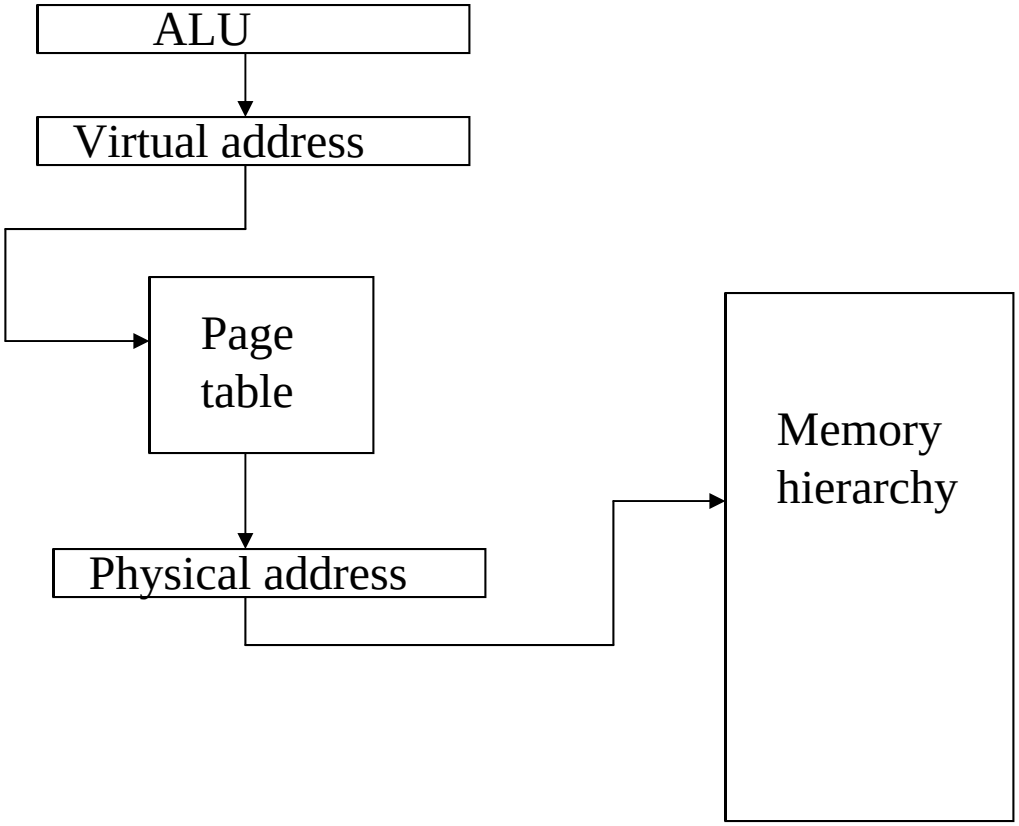
Illustration of Page Table



Virtual Address Translation



From Virtual Address to Memory Location (highly abstracted)



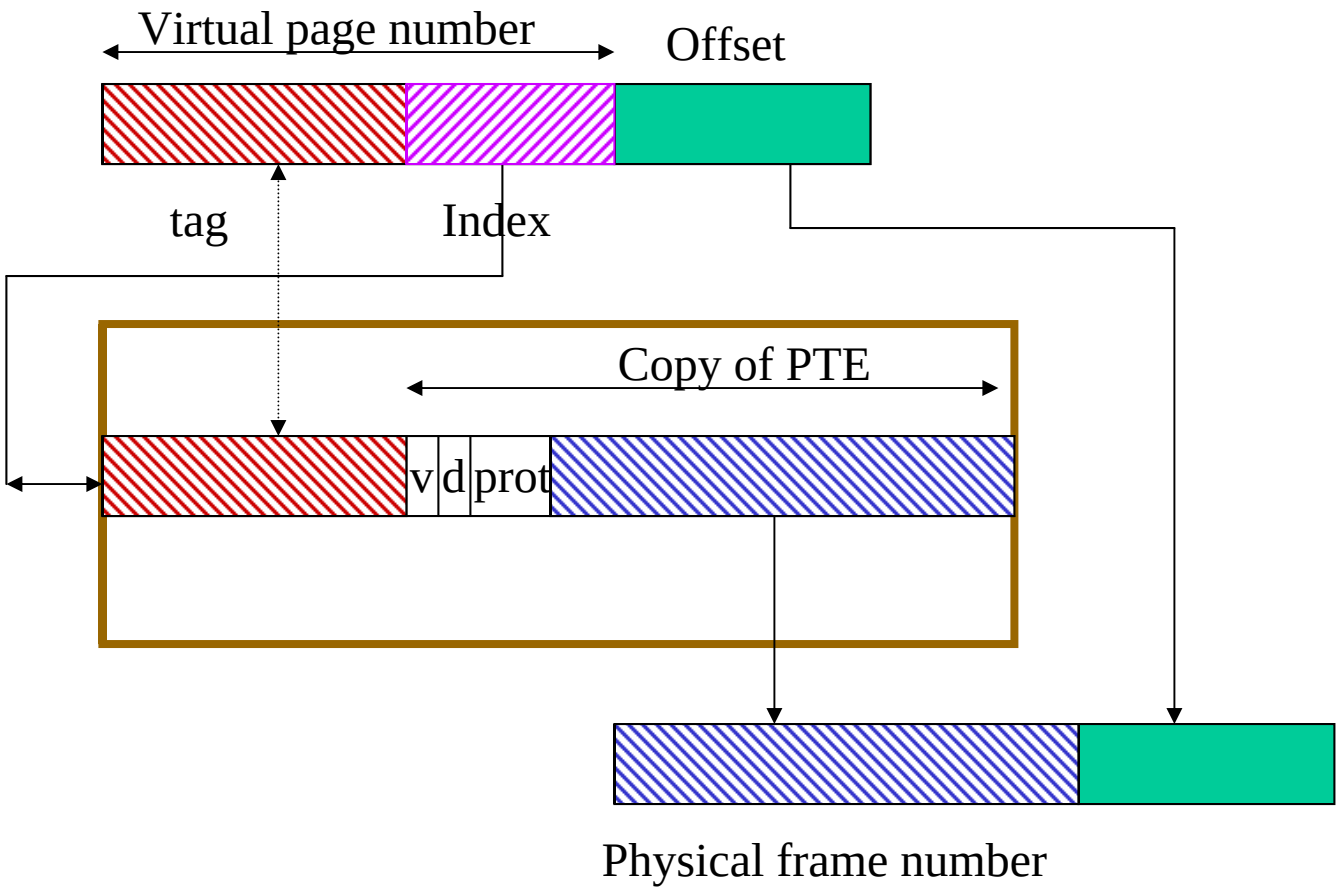
Translation Look-aside Buffers (TLB)

- Keeping page tables in memory defeats the purpose of caches
  - Needs one memory reference to do the translation
- Hence, introduction of caches to cache page table entries; these are the TLB's
  - There have been attempts to use the cache itself instead of a TLB but it has been proven not to be worthwhile
- Nowadays, TLB for instructions and TLB for data
  - Some part of the TLB's reserved for the system
  - Of the order of 128 entries, quite associative

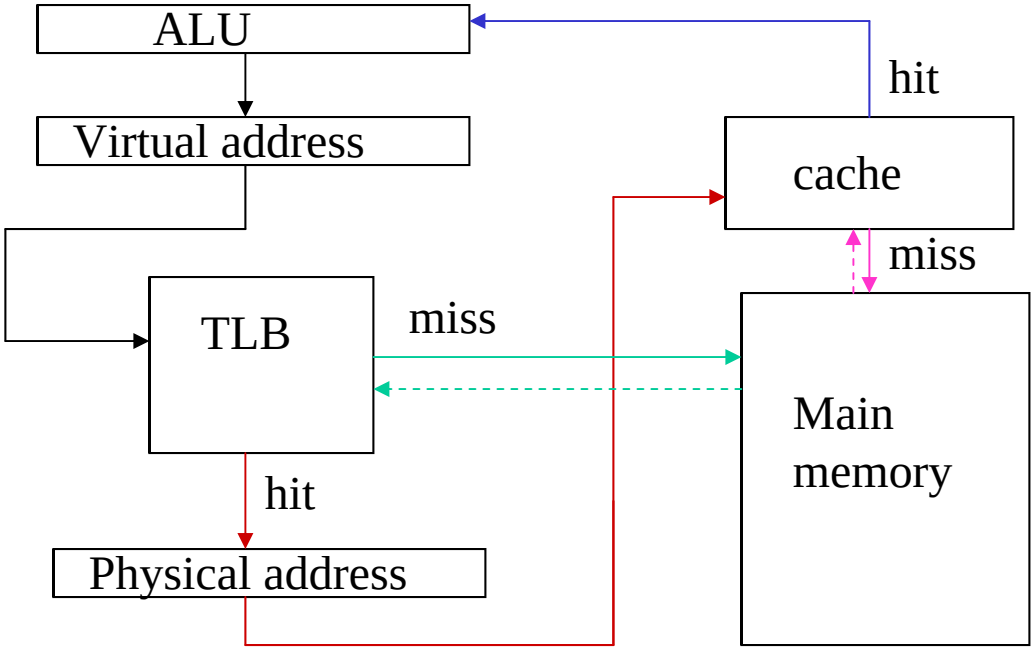
TLB's

- TLB miss handled by hardware or by software (e.g., PAL code in Alpha) or by a combination HW/SW
  - TLB miss 10's-100's cycles -> no context-switch
- Addressed in parallel with access to the cache
- Since smaller, goes faster
  - It's on the critical path
- For a given TLB size (number of entries)
  - Larger page size -> larger mapping range

TLB organization



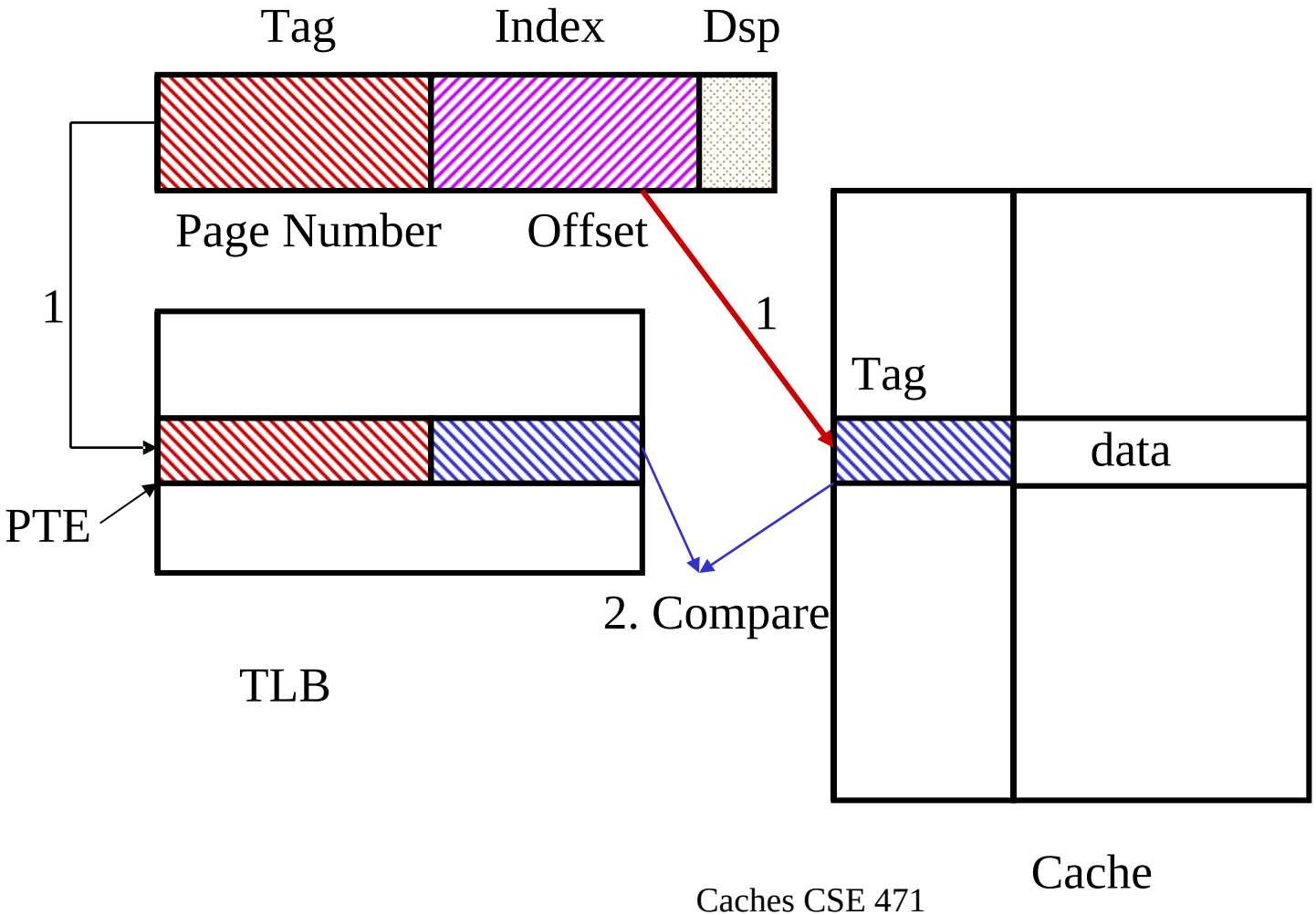
From Virtual Address to Memory Location (highly abstracted; revisited)



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

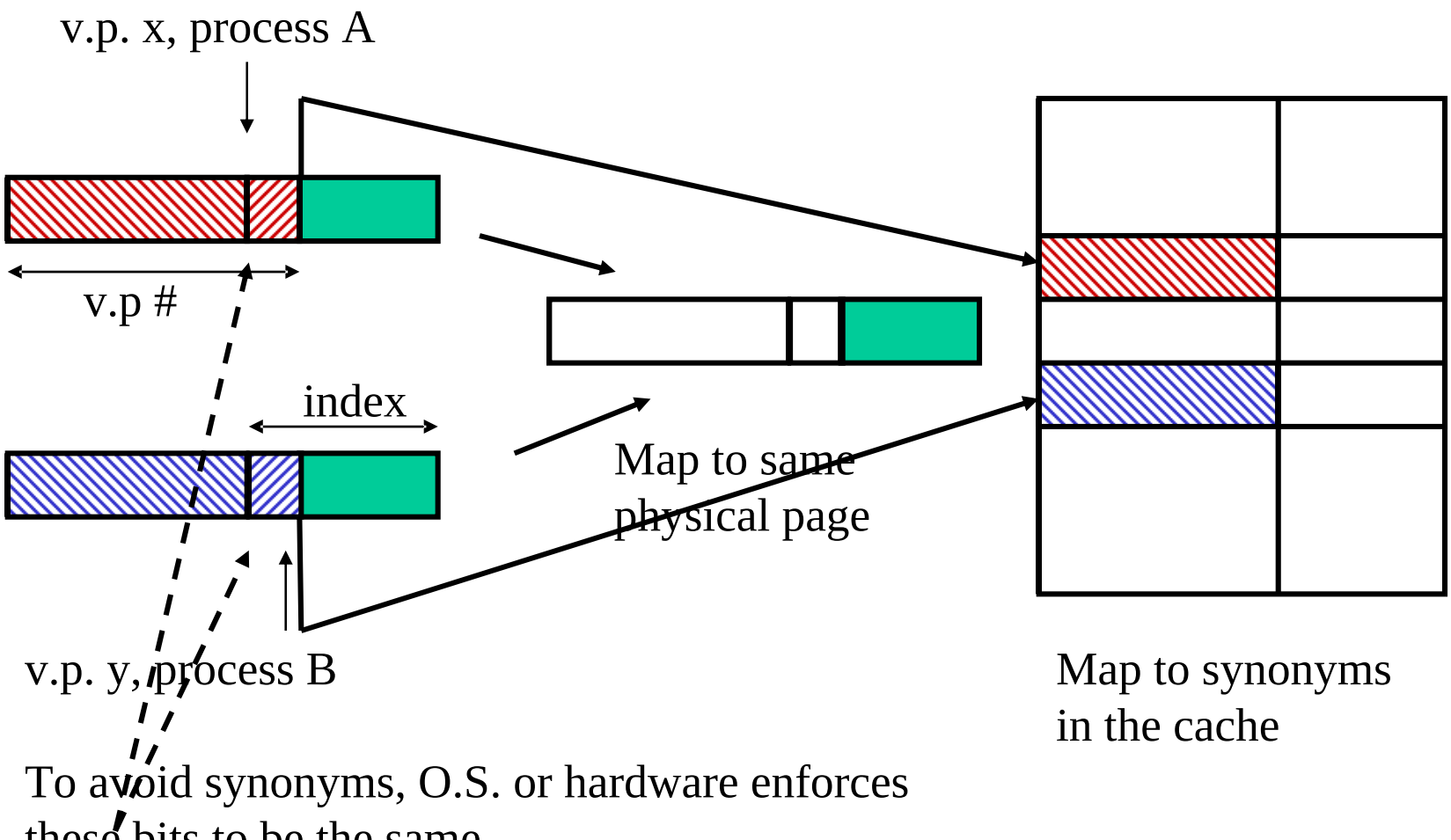
Virtually Addressed Cache



"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?

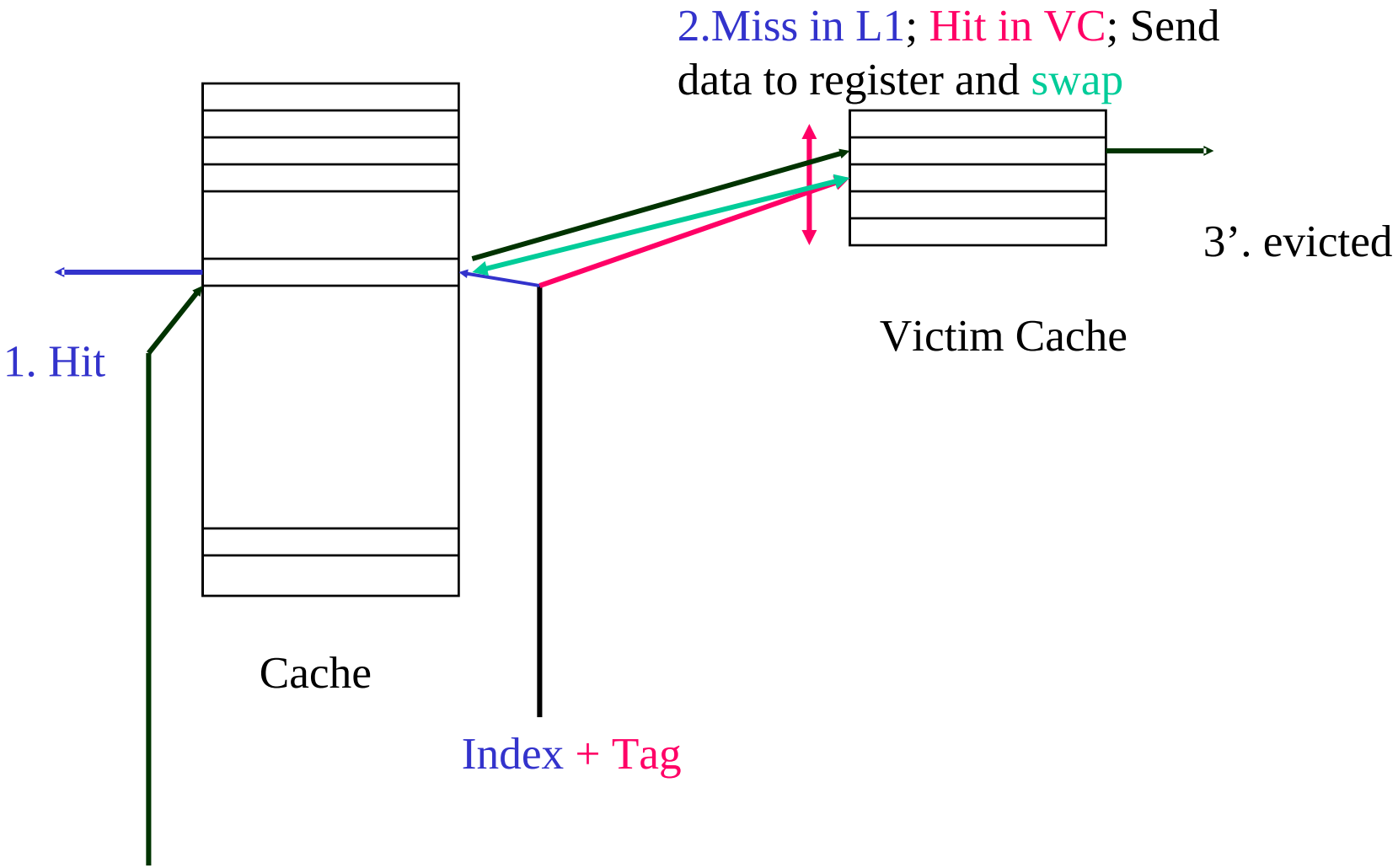
- For a given application, cache capacity and associativity, there is an optimal cache line size
- Long cache lines
  - Good for spatial locality (code, vectors)
  - Reduce compulsory misses (implicit prefetching)
  - But takes more time to bring from next level of memory hierarchy (can be compensated 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

Software Prefetching

- Use of special instructions (cache hints: touch in Power PC, load in register 31 for Alpha, prefetch in Intel micros)
- Non-binding prefetch (in contrast with proposals to prefetch in registers).
  - If an exception occurs, the prefetch is ignored.
- Must be inserted by software (compiler analysis)
- Advantage: no special hardware
- Drawback: more instructions executed.

Metrics for Prefetching

- Coverage: Useful prefetches/ number of misses without prefetching
- Accuracy: useful prefetches/ number of prefetches
- Timeliness: Related to number of hit-wait prefetches
- In addition, the usefulness of prefetching is related to how critical the prefetched data was

Techniques to Reduce Cache Miss Penalty

- Give priority to reads -> Write buffers
- Send the requested word first -> critical word or wrap around strategy
- Sectored (subblock) caches
- Lock-up free (non-blocking) caches
- Cache hierarchy

Write Buffers

- Reads are more important than:
  - Writes to memory if WT cache
  - Replacement of dirty lines if WB
- Hence buffer the writes in write buffers
  - Write buffers = FIFO queues to store data
  - Since writes have a tendency to come in bunches (e.g., on procedure calls, context-switches etc), write buffers must be "*deep*"

Write Buffers (c'ed)

- Writes from write buffer to next level of the memory hierarchy can proceed in parallel with computation
- Now loads must check the contents of the write buffer; also more complex for cache coherency in multiprocessors
  - Allow read misses to bypass the writes in the write buffer

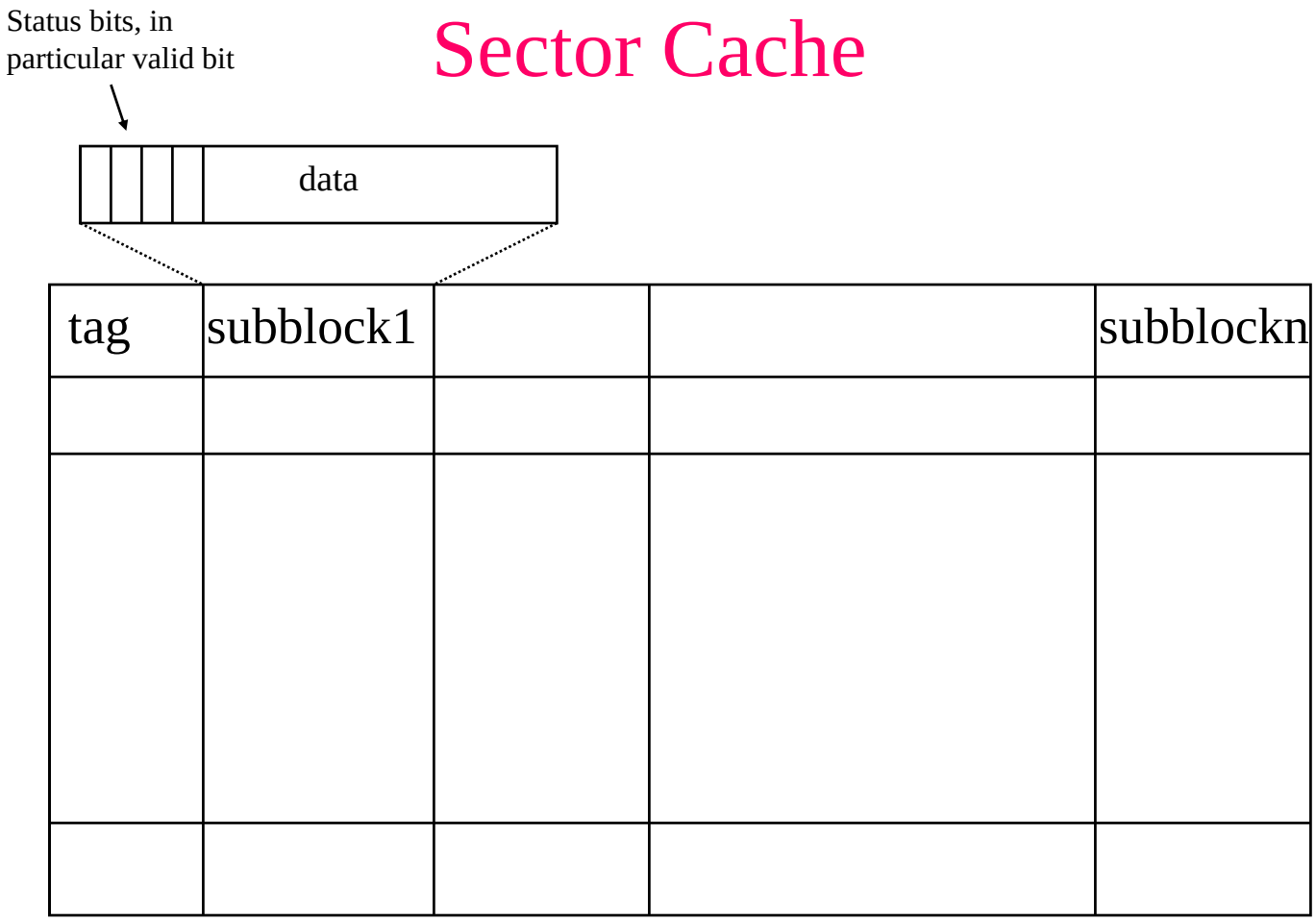
Critical Word First

- Send first, from next level in memory hierarchy, the word for which there was a miss
- Send that word directly to CPU register (or IF buffer if it's an I-cache miss) as soon as it arrives
- Need a one line buffer to hold the incoming line (and shift it) before storing it in the cache

Sectored (or subblock) Caches

- First cache ever (IBM 360/85 in late 60's) was a sector cache
  - On a cache miss, send only a subblock, change the tag and invalidate all other subblocks
  - Saves on memory bandwidth
- Reduces number of tags but requires good spatial locality in application
- Requires status bits (valid, dirty) per subblock
- Might reduce false-sharing in multiprocessors
  - But requires metadata status bits for each subblock
  - Alpha 21164 L2 uses a dirty bit/16 B for a 64B block size

Sector Cache



Lock-up Free Caches

- Proposed in early 1980's but implemented only within the last 15 years because quite complex
- Allow cache to have several outstanding miss requests (hit under miss).
  - Cache miss "happens" during EX stage, i.e., longer (unpredictable) latency
  - Important not to slow down operations that don't depend on results of the load
- Single hit under miss (HP PA 1700) relatively simple
- For several outstanding misses, require the use of MSHR's (Miss Status Holding Register).

MSHR's

- The outstanding misses do not necessarily come back in the order they were detected
  - For example, miss 1 can percolate from L1 to main memory while miss 2 can be resolved at the L2 level
- Each MSHR must hold information about the particular miss it will handle such as:
  - Info. relative to its placement in the cache
  - Info. relative to the "missing" item (word, byte) and where to forward it (CPU register)

Implementation of MSHR's

- Quite a variety of alternatives
  - MIPS 10000, Alpha 21164, Pentium Pro, III and 4
- One particular way of doing it:
  - Valid (busy) bit (limited number of MSHR's – structural hazard)
  - Address of the requested cache block
  - Index in the cache where the block will go
  - Comparator (to prevent using the same MSHR for a miss to the same block)
  - If data to be forwarded to CPU at the same time as in the cache, needs addresses of registers (one per possible word/byte)
  - Valid bits (for writes)

Cache Hierarchy

- Two, and even three, levels of caches in most systems
- L2 (or L3, i.e., board-level) very large but since L1 filters many references, “local” hit rate might appear low (maybe 50%) (compulsory misses still happen)
- In general L2 have longer cache blocks and larger associativity
- In general L2 caches are write-back, write allocate

Characteristics of Cache Hierarchy

- Multi-Level inclusion (MLI) property between off-board cache (L2 or L3) and on-chip cache(s) (L1 and maybe L2)
  - L2 contents must be a superset of L1 contents (or at least have room to store these contents if L1 is write-back)
  - If L1 and L2 are on chip, they could be mutually exclusive (and inclusion will be with L3)
  - MLI very important for cache coherence in multiprocessor systems (shields the on-chip caches from unnecessary interference)
- Prefetching at L2 level is an interesting challenge (made easier if L2 tags are kept on-chip)

Impact of Branch Prediction on Caches

- If we are on predicted path and:
  - An I-cache miss occurs, what should we do: stall or fetch?
  - A D-cache miss occurs, what should we do: stall or fetch?
- If we fetch and we are on the right path, it's a win
- If we fetch and are on the wrong path, it is not necessarily a loss
  - Could be a form of prefetching (if branch was mispredicted, there is a good chance that that path will be taken later)
  - However, the channel between the cache and higher-level of hierarchy is occupied while something more pressing could be waiting for it