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 block, or a sequence of cache blocks
- 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 block 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 etc.

Software Prefetching

- Use of special instructions (cache hints: touch in Power PC, load in register 31 for Alpha, prefetch in recent 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: Prefetch hits/ 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 Policies

- Loads (reads) occur twice as often as stores (writes)
- Miss ratio of reads and miss ratio of writes pretty much the same
- Although it is more important to optimize read performance, write performance should not be neglected
- Write misses can be delayed w/o impeding the progress of execution of subsequent instructions

A Sample of Write Mechanisms

- Fetch-on-write and Write-allocate
  - Proceed like on a read miss followed by a write hit: preferred method for write-back caches.
- No-fetch-on-write and no-write-allocate ("write-around")
  - The cache is not modified on write misses: preferred method for write-through caches
- No-fetch-on-write and write-allocate ("write-validate")
  - Write directly in cache and invalidate all other parts of the line being written. Requires a valid bit/writeable entity. Good for initializing a data structure. Assumedly the best policy for elimination of write misses in write-through caches but more expensive (dirty bits)

Write Mechanisms (c'ed)

- write-before-hit, no-fetch-on-write and no-write-allocate ("write invalidate")
  - The data is written in the cache totally in parallel with checking of the tag. On a miss, the rest of the line is invalidated as in the write-validate case
  - Possible only for direct-mapped write-through caches
- A mix of these policies can be (and has been) implemented
  - Dictate the policy on a page per page basis (bits set in the page table entry)
  - Have the compiler generate instructions (hints) for dynamic changes in policies (e.g. write validate on initialization of a data structure)

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

Coalescing Write Buffers and Write Caches

- Coalescing write buffers
  - Writes to an address (block) already in the write buffer are combined
    - Note the tension between writing the coalescing buffer to memory at high rate -- more writes -- vs. coalescing to the max -- but buffer might become full
- Extend write buffers to small fully associative write caches with WB strategy and dirty bit/byte.
  - Not implemented in any machine I know of

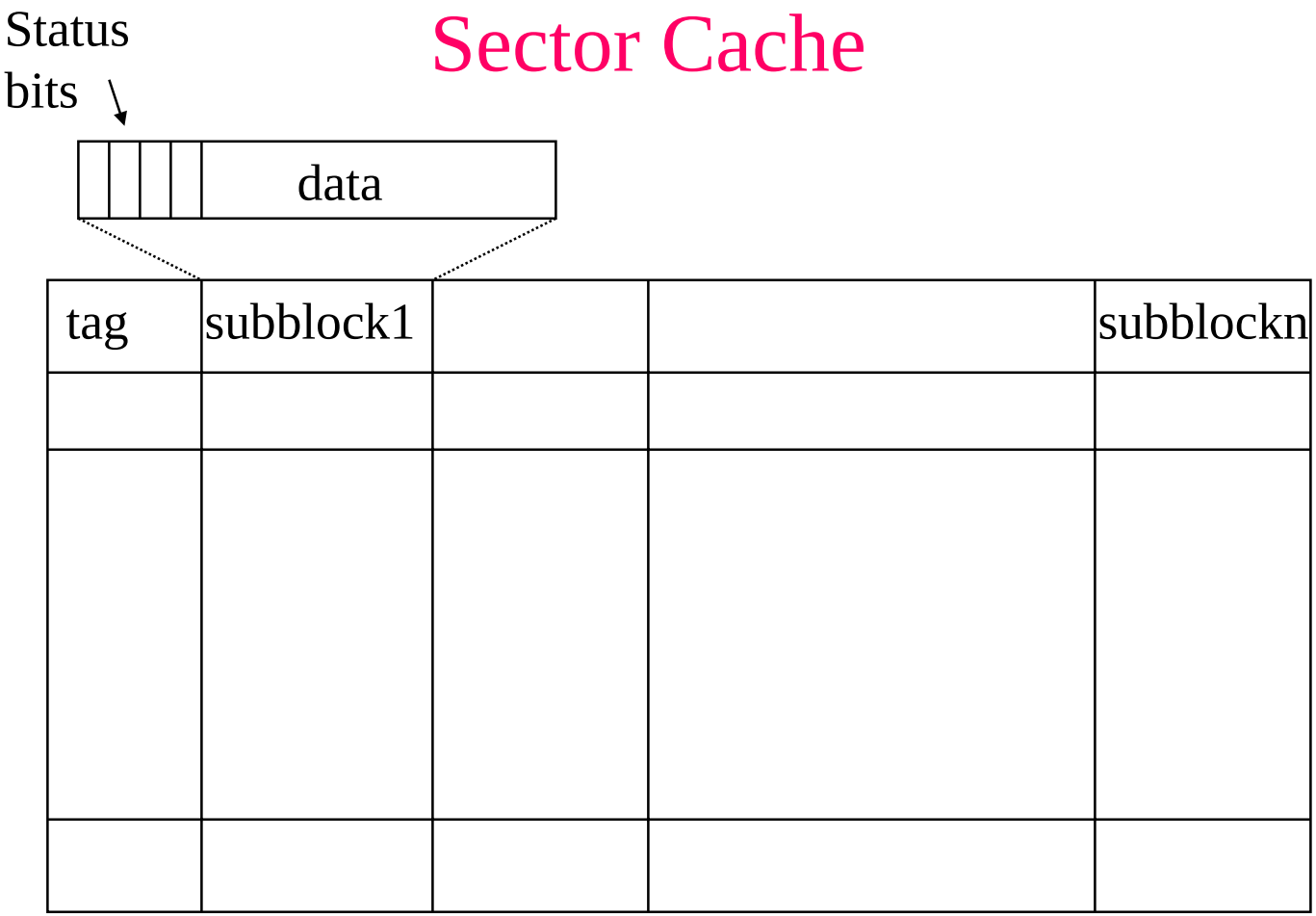
Critical Word First

- 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 block buffer to hold the incoming block (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 10 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 IV
- 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)

"Virtual" Address Caches

- Will get back to this after we study TLB's
- Virtually addressed, virtually tagged caches
  - Main problem to solve is the Synonym problem (2 virtual addresses corresponding to the same physical address).
- Virtually addressed, physically tagged
  - Advantage: can allow cache and TLB accesses concurrently
  - Easy and usually done for small L1, i.e., capacity < (page * ass.)
  - Can be done for larger caches if O.S. does a form of page coloring such that "index" is the same for synonyms

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