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

Obvious Solutions to Decrease Miss Rate

- Increase cache capacity
  - Yes, but the larger the cache, the slower the access time
  - Limitations for first-level (L1) on-chip caches
  - 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 Block Size?

- For a given application, cache capacity and associativity, there is an optimal cache block size
- Long cache blocks
  - 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 block is used – or reused)
  - Increase possibility of false-sharing in multiprocessor systems

What about Cache Block Size? (c'ed)

- In general, the larger the cache, the longer the best block size (e.g., 32 or 64 bytes for on-chip, 64, 128 or even 256 bytes for large off-chip caches)
- Longer block sizes in I-caches
  - Sequentiality of code
  - Matching with the IF unit

Example of (naïve) Analysis

- Choice between 32B (miss rate *m32*) and 64B block sizes (*m64*)
- Time of a miss:
  - send request + access time + transfer time
  - send request independent of block size (e.g., 4 cycles)
  - access time can be considered independent of block size (memory interleaving) (e.g., 28 cycles)
  - transfer time depends on bus width. For a bus width of say 64 bits transfer time is twice as much for 64B (say 16 cycles) than for 32B (8 cycles).
  - In this example, 32B is better if $(4+28+8)m32 < (4+28+16)\ m64$

Example of (naïve) Analysis (c'ed)

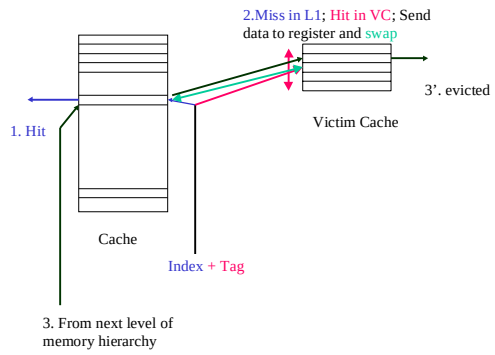
- Case 1. 16 KB cache: *m32* = 2.87, *m64* = 2.64
  - 2.87 * 40 < 2.64 * 48
- Case 2. 64KB cache: *m32* = 1.35, *m64* = 1.06
  - 1.35 * 40 > 1.06 * 48
- 32B better for 16KB and 64B better for 64KB
  - (Of course the example was designed to support the "in general the larger the cache, the longer the best block size " statement of two slides ago).

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 (see later)
  - 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

- First example (in this course) of an "hardware assist"
- Victim cache: Small fully-associative buffer "behind" the L1 cache and "before" the L2 cache
- Of course can also exist "behind" L2 and "before" main memory
- Main goal: remove some of the conflict misses in L1 direct-mapped caches (or any cache with low associativity)



Operation of a Victim Cache

- 1. Hit in L1; Nothing else needed
- 2. Miss in L1 for block 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

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 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.

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.
  - The preferred method for write-back caches.
- No-write-before-hit, no-fetch-on-write and no-write-allocate (called "write-around")
  - The cache is not modified on write misses
- 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 (called "write invalidate")
  - Possible only for direct-mapped write-through caches
  - 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
- 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

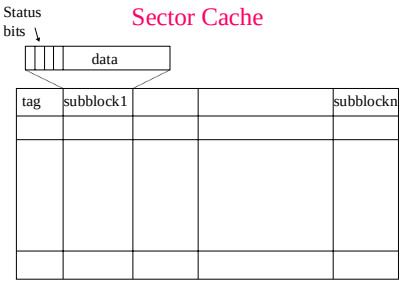
Critical Word First

- 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 recently 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
- 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 quite common now
- 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