Lecture 9 Memory Hierarchy

Instructor: Prof. Yan Luo

Cache Performance

- Different measure: AMAT
- Average Memory Access time (AMAT)

  = HitTime×HitRate + MissTime×MissRate = HitTime×(1-MissRate)+ MissTime×MissRate

  = HitTime + (MissRate × MissPenalty)

  MissPenalty = MissTime - HitTime

- CPU time

  = (CPU execution clock cycles w/o memory stalls+Memory stall clock cycles) x cc

- $CPI = CPI_{ideal}$ + Memory stall per instruction
- Note: *memory hit time is included in execution cycles*.

Impact on Performance

- Suppose a processor executes at
  - Clock Rate = 200 MHz (5 ns per cycle)
  - Base CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- Suppose 10% miss rate in data memory and miss penalty is 50 cycles
- Suppose 1% miss rate in instructions memory, same miss penalty
- CPI = Base CPI + average stalls per instruction

  1.1(cycles/ins) +

  [0.30 (DataMops/ins)

  x 0.10 (miss/DataMop) x 50 (cycle/miss)] +

  [1 (InstMop/ins)

  x 0.01 (miss/InstMop) x 50 (cycle/miss)]

  = (1.1 + 1.5 + .5) cycle/ins = 3.1

- AMAT=1+(1/1.3×1% + 0.3/1.3×10%)×50 = 2.54

Impact of Change in cc

- Suppose a processor has the following parameters:
  - CPI = 2 (w/o memory stalls)
  - mem access per instruction = 1.5
- Compare AMAT and CPU time for a direct mapped cache and a 2-way set associative cache assuming:

	cc	Hit cycle	Miss penalty	Miss rate
Direct map	1ns	1	75 ns	1.4%
2-way associative	1.25ns(why?)	1	75 ns	1.0%

  - $AMAT_d$ = hit time + miss rate * miss penalty = 1*1 + 0.014*75 = 2.05 ns
  - $AMAT_2$ = 1*1.25 + 0.01*75 = 2 ns < 2.05 ns

  - CPU_d = (CPI*cc + mem. stall time)*IC = (2*1 + 1.5*0.014*75)IC = 3.575*IC
  - CPU_2 = (2*1.25 + 1.5*0.01*75)IC = 3.625*IC **>** CPU_d !
- Change in cc affects all instructions while reduction in miss rate benefit only memory instructions.

Miss Penalty for Out-of-Order Exe. Proc.

- In OOO processors, memory stall cycles are overlapped with execution of other instructions. Miss penalty should not include this overlapped part.

  mem stall cycle per instruction = mem miss per instruction x

  (total miss penalty – overlapped miss penalty)

- For the previous example. Suppose 30% of the 75ns miss penalty can be overlapped, what is the AMAT and CPU time?
  - Assume using direct map cache, cc=1.25ns to handle out of order execution.

  AMATd = 1*1.25 + 0.014*(75*0.7) = 1.985 ns

  CPU time =(2*1.25 + 1.5 * 0.014 * (75*0.7))*IC = 3.6025 IC < CPU_2

Improving CPU Performance

- In the past 10 years, there are over 5000 research papers on reducing the gap between the CPU and memory speeds.
- We will address some them in four categories:
  - Reducing the cache miss penalty
  - Reducing the miss rate
  - Reducing the cache miss penalty or miss rate via parallelism
  - Reducing the hit time

Cache Misses

- Types of cache misses
  - the three Cs:
    - *Compulsory*: first access to a block is a miss.
    - *Conflict*: collision misses, blocks map to the same set.
    - *Capacity*: replaced blocks that are later referenced, cache too small.
  - the fourth C:
    - *Coherence*: shared memory, invalid copies
- Relative effects
  - Fully associative placement: no conflict misses.
  - Larger block size *might* reduce compulsory misses.
  - Larger caches have lower capacity misses.

  Most of these have negative impacts on hit time and therefore cycle time.
  - a direct mapped cache can be faster than a set associative one

Reducing Cache Miss Penalty (1)

1. Multilevel Caches – "the more the merrier"
   - Add another level behind L1 cache to speed up access from memory (why not combine the two levels into one? Because larger cache will incur longer access time and could stretch cc to hurt all instructions!)

$$\text{AMAT} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Local miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

$$\text{Average memory stall time} = \text{Miss rate}_{L1} \times \text{Hit time}_{L2} + \text{Miss rate}_{L1} \times \text{Local miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

$$\text{Average memory stalls per instruction} = \text{Miss per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Miss per instruciton}_{L2} \times \text{Miss penalty}_{L2}$$

Example

- For every 1000 instructions, 40 misses in L1 and 20 misses in L2;

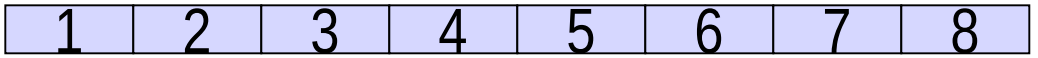
  Hit cycle in L1 is 1, L2 is 10; Miss penalty from L2 to memory is 100 cycles; there are 1.5 memory references per instruction. What is AMAT and average stall cycles per instruction?
  - AMAT = [1 + 40/1000 * (10 + 20/40 * 100)] *cc = 3.4cc
  - Average stall cycles per instruction = 1.5 * 40/1000 * 10 + 1.5 * 20/1000 * 100 = 3.6 cycles

- Note: We have not distinguished reads and writes. Access L2 only on L1 miss, i.e. write back cache

Reducing Cache Miss Penalty (2,3)

2. Critical word first – "impatience"

L2 cache block:

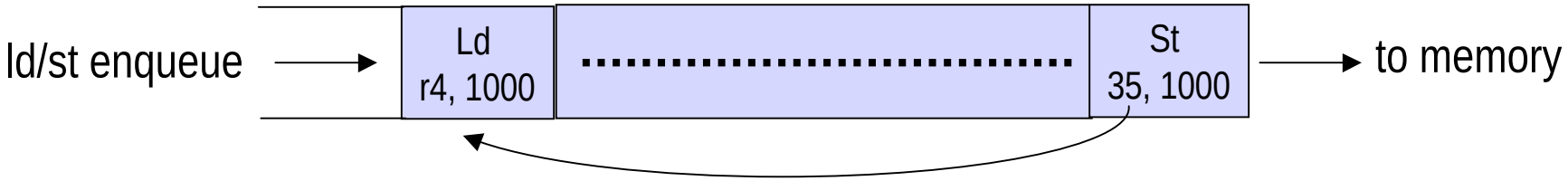


Requested word: 5

Critical word first:

(wrapped fetch)

3. Serves reads before writes have been completed – "preference"
   - Recall: In ooo processor, the reorder buffer contains loads/stores (waiting for address computation or memory) in program order.



Reducing Cache Miss Penalty (3,4)

- Complications with write buffer: it stores updated blocks but the L2 hasn't seen them yet! What will happen on a L1 read miss?
  - Conventionally, read miss stalls and waits until write buffer flushes its content to L2 and then access L2 (slow).
  - Alternatively, L1 read miss should check write buffer before going to L2 (faster).

4. Improving write buffer (in write through) efficiency – "companionship"
   - Two writes with the same address are coalesced.
   - Writes stall if no empty entries are present -> utilize the entries efficiently using *write merge*.

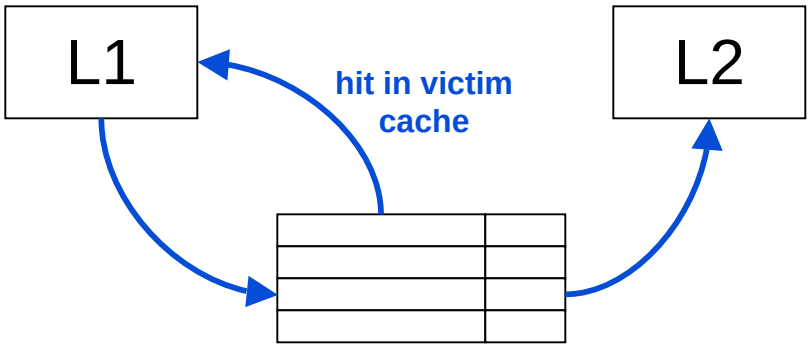
Write Merge

Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Reducing Cache Miss Penalty (5)

- Victim Cache – "recycling"
  - Holds victim blocks discarded from the L1 cache due to replacement.
  - Small (otherwise an L2) and fully associative.
  - Checked on a L1 miss. If found, block is swapped back to L1 (the block previously took its place is put into victim cache).
  - Works better for small L1 caches since it saves victim blocks from conflict misses.
  - Effective: a 4-entry vc can reduce ¼ of the misses in a 4KB L1 cache [Jouppi, 1990].



Reducing Miss Rate (1-3)

1. Larger block size
   - Takes advantage of locality
   - But, longer miss penalty and maybe more conflict misses (w/ same cache size)
   - Block size increase should not reach the point where miss rate increases.
2. Larger cache size
   - Longer hit time – suitable for lower level caches.
3. Higher associativity
   - 2:1 cache rule of thumb: a direct-mapped cache of size N has about the same miss rate as a 2-way set-associative cache of size N/2 (for cache size < 128KB)
   - Longer hit time

Improving an aspect of AMAT comes at the expense of another!

Reducing Miss Rate (4,5)

4. Pseudoassociative cache – a direct-mapped cache having same hit rate as a 2-way set-associative cache
   - If the first access is a miss, try an alternative block (by modifying an address portion)
   - A normal hit time and a pseudohit time – in addition to the miss penalty

Reducing Hit Time

1. Use small and simple L1 cache
   - Small hardware is faster
   - Direct map cache is faster
2. Pipeline writes
   - In writes: must check tag BEFORE write is done.
   - Separate tag check and data write, delay data write with respect to tag check: back to back writes (Alpha 21064).
3. Trace cache, another type of I-cache (Pentium 4)
   - Stores dynamic instruction sequence (trace) instead of static sequence.
   - Include multiple taken branches and make them into straight line code
     - Reduce I-cache misses due to fetch branch target since the target now is just the next instruction
   - Downside:
     - Instruction may repeatedly occur in trace cache – wasting space

Reducing Hit Time

4. Avoid address translation
   - Addresses sent to cache need to be translated from virtual addresses to physical addresses – done by ***translation lookaside buffer*** (TLB)
   - Translation occur before going to the L1 cache – cost time
   - Virtually addressed cache:
     - + Index the cache using virtual addresses, avoid TLB accesses, save time
     - – Context switch causes flushing the entire cache
     - – Alias: different virtual addresses may map into same physical address – need protection mechanism!
5. Way prediction – approaching hit time of a direct-mapped cache for set associative caches
   - Do not compare all the tags.
   - Predict one and access the way just as a direct-mapped cache.
   - Prediction is done in the previous cache access (extra prediction bits are maintained).
   - On a miss, all the rest ways are compared as a normal set-associative cache – takes longer time.
   - Alpha 21264 instruction cache

Parallelize Mem. Access with Execution

1. Nonblocking cache (lockup-free)
   - Continue to service cache accesses during a miss – works for ooo processors.
   - Significantly increase the complexity of the cache controller.
2. Hardware prefetching
   - Get the data or instruction before it is accessed
   - Does not slow down other cache activities
     - Continue to serve other instructions or data while waiting for the prefetched data – normally nonblocking.
   - Never replace useful data (use additional buffers).
   - Stream buffer for instructions
     - On I-cache miss, check stream buffer.
     - Stream buffer hit →move the block into I-cache, refill (prefetch) stream buffer with ***next*** block.
     - Stream buffer miss → fetch target block into I-cache and ***next*** block into stream buffer

Parallelize mem. Access with Execution

3. Compiler-controlled prefetching
   - Compiler inserted *prefetch* instructions; two types:
     - *register prefetch*: data loaded to registers
     - *cache prefetch*: data to CM.
   - Either type can be
     - *faulting* or
     - *non-faulting, i.e.* allowed to cause a page fault or not.
   - The most effective is "*semantically invisible*" prefetches
     - Does not change register content and non-faulting.
   - Prefetching is effective with loop unrolling (if miss penalty is low) or software pipelining (large penalty).
   - But,
     - instruction overhead (such overhead can not exceed the benefits)
     - Maybe the data or instruction is already in the cache
     - Is the prefetch early enough for the data to arrive by the time it is needed