Lecture 3:

Advanced Caching Techniques

Department of Electrical Engineering
Stanford University

http://eeclass.stanford.edu/ee282

Announcements

- HW1 out on Wednesday
  - Make sure you have a group of 3 & start early
  - PA-1 is coming is a week too

Today’s Menu: Advanced Caching Techniques

- Understanding cache performance
  - Average memory access time
  - Types of misses (the 3 Cs)
  - Review: basic cache design choices

- How to reduce cache hit time
- How to reduce cache miss rate
- How to reduce cache miss penalty

Review of Cache Basics

- Why do we need caches and what's their goal?
- What's the basic idea of a cache and why does it work?
- How do we find something in a cache?
- What happens on a cache miss?
- What happens on a cache write?

Improving Cache Performance

- Goal: reduce the Average Memory Access Time (AMAT)
  – AMAT = Hit Time + Miss Rate * Miss Penalty

- Approaches
  – Reduce Hit Time
  – Reduce or Hide Miss Penalty
  – Reduce Miss Rate

- Notes:
  – There may be conflicting goals
  – Keep track of clock cycle time, area, and power consumption

Understanding Cache Misses: the 3 Cs

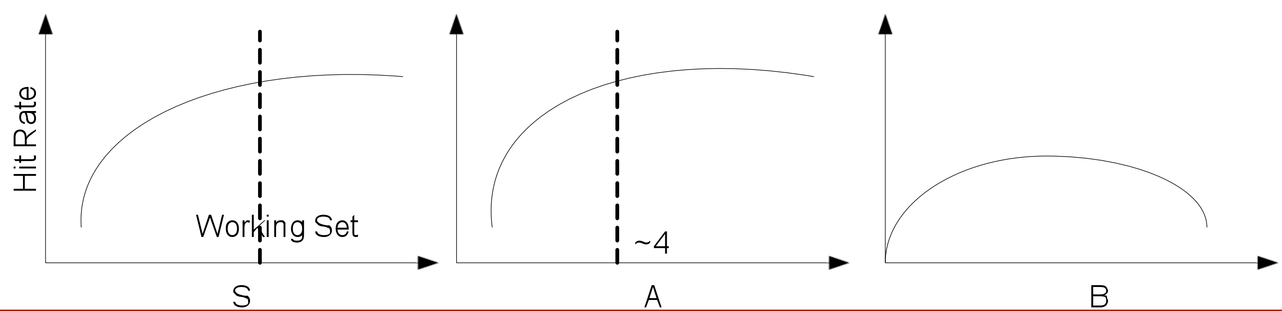
- Compulsory or cold misses
  – First access to an address within a program
  – Misses even with an infinite sized cache

- Capacity misses
  – Misses because cache not large enough to fit working set
    - Block replaced from cache and later accessed again
  – Misses in fully associative cache of a certain size

- Conflict or interference misses
  – Misses due to associativity
  – E.g. two addresses map to same block in direct mapped cache

Tuning Basic Cache Parameters: **S**ize, **A**ssociativity, **B**lock width

- Size:
  - Must be large enough to fit working set (temporal locality)
  - If too big, then hit time degrades
- Associativity
  - Need large to avoid conflicts, but 4-8 way is as good as FA
  - If too big, then hit time degrades
- Block
  - Need large to exploit spatial locality & reduce tag overhead
  - If too large, few blocks → higher misses & miss penalty

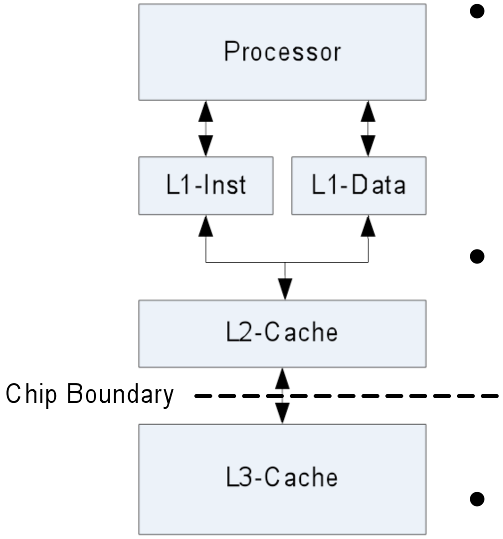


Basic Cache Policies (Write Miss)

	Write through				Write back	
	Write allocate		No write allocate		Write allocate	
Steps	fetch on miss	no fetch on miss	write around	write invalidate	fetch on miss	no fetch on miss
1	pick replacement	pick replacement			pick re-placement	pick re-placement
2				invalidate tag	[write back]	[write back]
3	fetch block				fetch block	
4	write cache	write partial cache			write cache	write partial cache
5	write memory	write memory	write memory	write memory		

- Which data access patterns benefit from each policy?

Multilevel Caches



- Motivation:
  - Optimize each cache for different constraints
  - Exploit cost/capacity trade-offs at different levels
- L1 caches
  - Optimized for fast access time (1-3 CPU cycles)
  - 8KB-64KB, DM to 4-way SA
- L2 caches
  - Optimized for low miss rate (off-chip latency high)
  - 256KB-4MB, 4- to 16-way SA
- L3 caches
  - Optimized for low miss rate (DRAM latency high)
  - Multi-MB, highly associative, embedded DRAM?

2-level Cache Performance Equations

- L1 AMAT = HitTimeL1 + MissRateL1 * Miss PenaltyL1
  - MissLatencyL1 is low, so optimize HitTimeL1
- MissPenaltyL1 = HitTimeL2 + MissRateL2 * MissPenaltyL2
  - MissLatencyL2 is high, so optimize MissRateL2
- MissPenaltyL2 = DRAMaccessTime + (BlockSize/Bandwidth)
  - If DRAM time high or bandwidth high, use larger block size

- L2 miss rate:
  - Global: L2 misses / total CPU references
  - Local: L2 misses / CPU references that miss in L1
  - The equation above assumes local miss rate

Multi-level Inclusion

- Inclusion: if data at L1 is always a subset of data at L2

- Advantages of maintaining multi-level inclusion
  - Easier cache analysis
    - Overall MissRate = $MissRate_{L1}$ x $LocalMissRate_{L2}$
  - Easier coherence checks for I/O & multiprocessors
    - Check the lowest level only to determine if data in cache

- Disadvantages
  - L2 replacements are complicated if L2 and L1 block sizes differ
  - Wasted space if L2 not much larger than L1
    - The motivation for non-inclusion for some AMD chips

How to Maintain Inclusion

- On L1 misses
  - Bring block in L2 as well
- On L2 evictions or invalidations
  - First evict all block(s) from L1
  - Can simplify by maintaining extra state in L2 indicates which blocks are also in L1 and where (cache way)
- L1 instruction cache inclusion?
  - For most systems, instruction inclusion is not needed (why?)
  - Bad for applications that stress the L2 capacity with small code
    - E.g. matrix multiply with huge matrices…

Reducing Cache Hit Time

- Techniques we have seen so far (most interesting for L1)
  - Smaller capacity
  - Smaller associativity

- Additional techniques
  - Wide cache interfaces
  - Pseudo-associativity

- Techniques that increase cache bandwidth (# of concurrent accesses)
  - Pipelined caches
  - Multi-ported caches
  - Multi-banked caches

Wide Cache Interfaces

- Idea: return multiple words with single cache access
  - 2 words to a full cache line
- Benefit: reduces hit time if multiple words must be read anyway
  - Reduce need for multi-cycle accesses
- Cost: more wires/pins
  - To transfer multiple words at once
- Usage:
  - Instruction caches: to satisfy wide processor fetch
  - L1→L2, L2 → L3, …: where whole cache lines are transferred

Pseudo Associative Caches

- Idea: search the N ways sequentially
  - First search in way 1, if hit pass the data to processor
  - If miss, search way 2 in 2nd cycle, …
- Advantage: Hit time of direct mapped, miss rate of N-way SA
  - Each cycle only 1 way can provide data (fast multiplexing)
- Disadvantage: multiple hit times to handle
  - Depending on which way produces hit
  - Optimization: start from MRU way or predict
- Usage
  - With L1 caches to reduce miss rate without affecting hit time
  - With external caches (L3) to reduce board traces needed

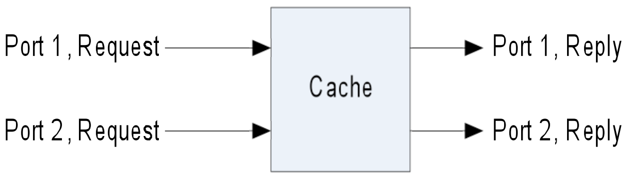
Multi-ported Caches

- Idea: allow for multiple accesses in parallel
  - Processor with many LSUs, I+D access in L2, …

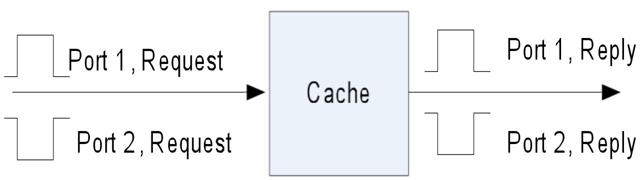
- Can be implemented in multiple ways
  - True multi-porting
  - Cache overclocking
  - Multiple cache copies
  - Line buffers
  - Multiple banks

- What is difficult about multiporting
  - Interaction between parallel accesses (especially for stores)

True Multiporting & Overclocking

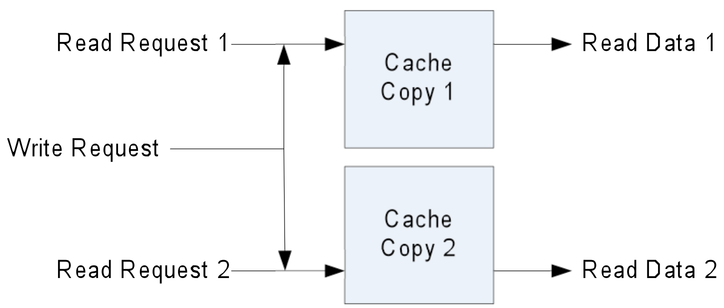


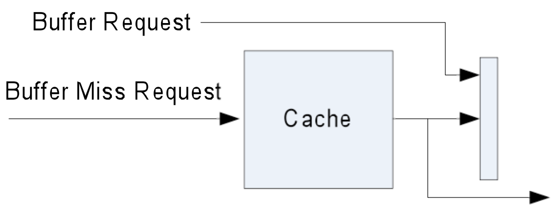
- True multiporting
  - Use 2-ported tag/data storage
  - Problem: large area increase
  - Problem: hit time increase



- Overclocking
  - Clock cache twice as fast as processor
    - Possible because caches are regular
  - One access per half cycle

Multiple Cache Copies & Line Buffers

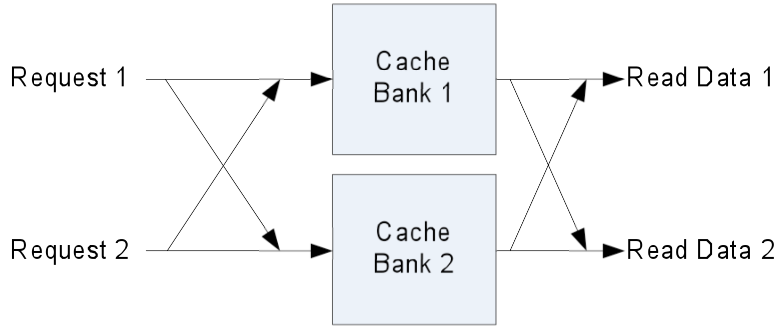




- Multiple cache copies
  - Two loads at the same time
  - Still only one store at a time
  - Twice the area, but same latency

- Line buffer or L0 cache
  - Store latest line accessed in buffer
  - Can do in parallel
    - An access to a new cache line
    - Multiple accesses that hit in buffer

Multi-banked Caches



- Partition address space into multiple banks
  - Bank0 caches addresses from partition 0, bank1 from partition 1…
  - Can use least or most significant address bits for partitioning
    - What are the advantages of each approach?
- Benefits: accesses can go in parallel if no conflicts
- Problems: conflicts, distribution network, bank utilization
- Usage:
  - Multi-ported L1, low latency L2

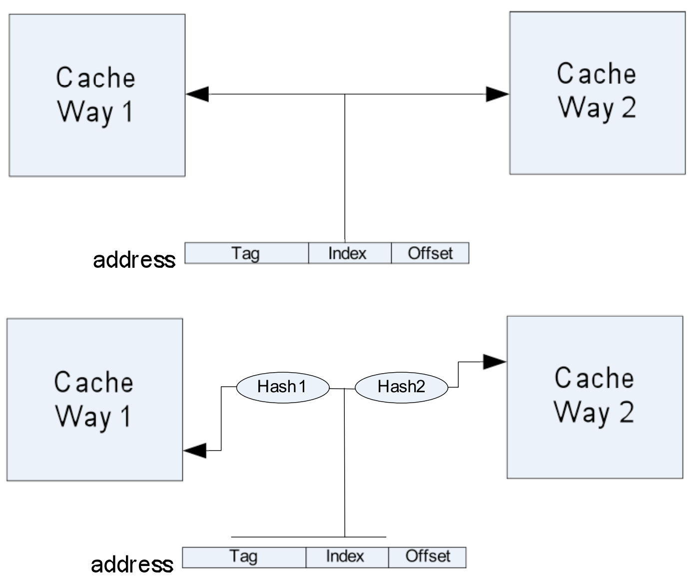
Reducing Miss Rate

- Techniques we have seen so far
  - Larger caches
    - Reduces capacity misses
  - Higher associativity
    - Reduces conflict misses
  - Larger block sizes
    - Reduces cold misses

- Additional techniques
  - Skew associative caches
  - Victim caches

Skew Associative Caches

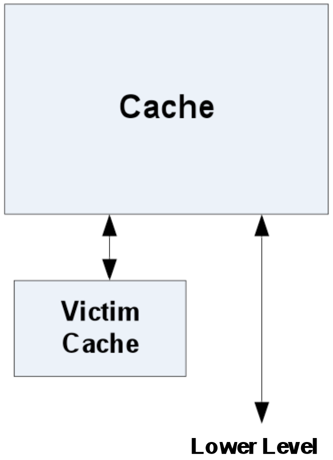
- Idea: reduce conflict misses by using different indices in each cache way
  - N-way cache: conflicts when N+1 blocks have same index bits in address



- Different indices though hashing
  - E.g. XOR index bits with some tag bits
  - E.g. reorder some index bits
- Benefit: indices are randomized
  - Less likely two blocks have same index
  - Conflict misses reduced and cache better utilized
  - May be able to reduce associativity
- Cost: latency of hash function

Victim Cache

- Small FA cache for blocks recently evicted from L1
  - Accessed on a miss in parallel or before the lower level
  - Typical size: 4 to 16 blocks (fast)



- Benefits
  - Captures common conflicts due to low associativity or ineffective replacement policy
  - Avoids lower level access
- Notes
  - Helps the most with small or low-associativity caches
  - Helps more with large blocks

Reducing Miss Penalty

- Techniques we have seen so far
  - Multi-level caches

- Additional techniques
  - Sub-blocks
  - Critical word first
  - Write buffers
  - Non-blocking caches

Sub-blocks

V	**Subblock0**	**V**	**Subblock1**	**V**	**Subblock2**	**V**	**Subblock3**

- Idea: break cache line into sub-blocks with separate valid bits
  – But the still share a single tag

- Low miss latency for loads:
  – Fetch required subblock only

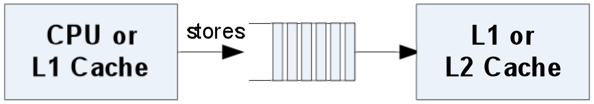
- Low latency for stores:
  – Do not fetch the cache line on the miss
  – Write only the sub-block produced, the rest are invalid
  – If there is temporal locality in writes, this can save many refills

Critical Word First

- Idea: fetch requested word or subblock first
  - And then the rest of the cache block
  - Useful when blocks are large and bandwidth low
  - Not that useful if program has spatial locality

- Why critical word first works: early CPU or L1 restart:
  - Return data to CPU/L1 as soon as requested word/subblock arrives
  - Don’t wait for the whole block to arrive in L1 cache

Write Buffers



- Write buffers allow for a large number of optimizations
- Write through caches
  - Stores don't have to wait for lower level latency
  - Stall store only when buffer is full
- Write back caches
  - Fetch new block before writing back evicted block
- CPUs and caches in general
  - Allow younger loads to bypass older stores
  - Beware of dependencies…

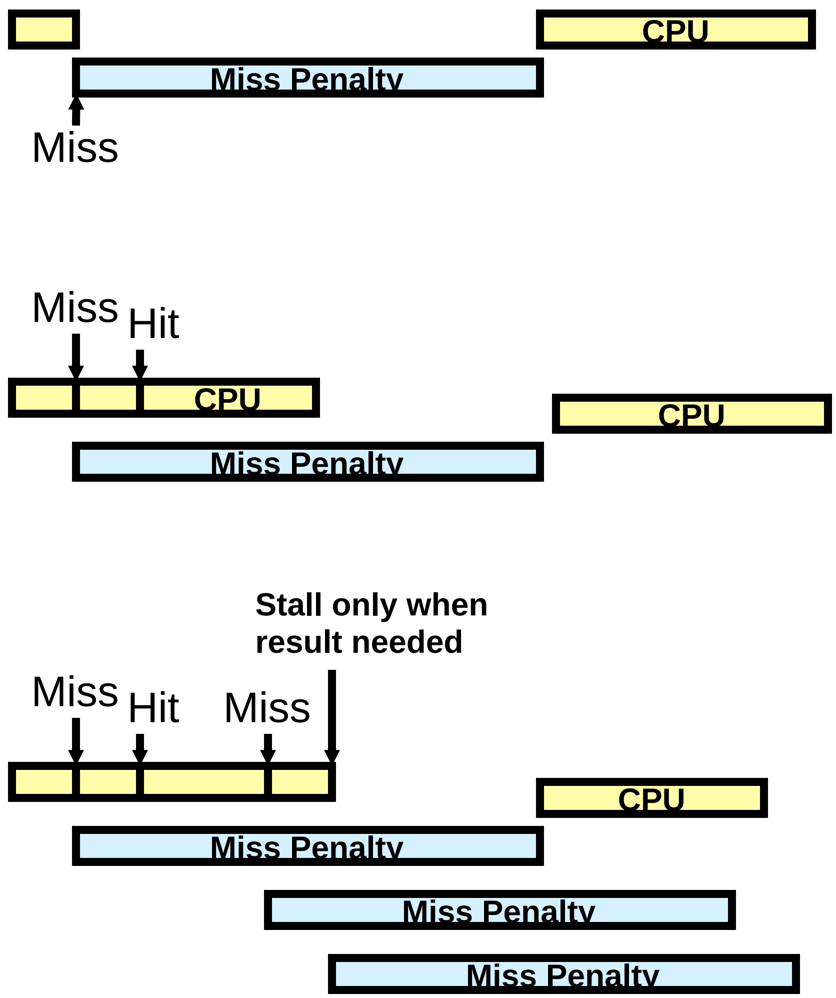
Write Buffer Design

- Size: 2-8 entries are typically sufficient for caches
  - But an entry may store a whole cache line
  - Make sure the write buffer can handle the typical store bursts…
    - Analyze your common programs, consider bandwidth to lower level
- Coalescing write buffers
  - Merge adjacent writes into single entry
  - Especially useful for write-through caches
- Dependency checks
  - Comparators that check load address against pending stores
    - If match there is a dependency so load must stall
  - Optimization: load forwarding
    - If match and store has its data, forward data to load…

Non-blocking or Lockup Free Caches

- Idea:
  - Allow for hits while serving a miss (hit-under-miss)
  - Allow for more than one outstanding miss (miss-under-miss)
- When does it make sense (for L1, L2, …)
  - When the processor can handle >1 pending load/store
    - This is the case with superscalar processors
  - When the cache serves >1 processor or other cache
  - When the lower level allows for multiple pending accesses
    - Multi-banked, split transaction busses, pipelining, …
- What is difficult about non-blocking caches:
  - Handling multiple misses at the time
  - Handling loads to pending misses
  - Handling stores to pending misses

Potential of Non-blocking Caches



Stall CPU on miss

Hit under miss

Multiple out-standing misses

Miss Status Handling Register

- Keeps track of
  - Outstanding cache misses
  - Pending load & stores that refer to that cache block
- Fields of an MSHR
  - Valid bit
  - Cache block address
    - Must support associative search
  - Issued bit (1 if already request issued to memory)
  - For each pending load or store
    - Valid bit
    - Type (load/store) and format (byte/halfword/…)
    - Block offset
    - Destination register for load OR store buffer entry for stores

MSHR

1	27	1
Valid	**Block Address**	**Issued**

1	3	5	5	
Valid	**Type**	**Block Offset**	**Destination**	Load/store 0
Valid	**Type**	**Block Offset**	**Destination**	Load/store 1
Valid	**Type**	**Block Offset**	**Destination**	Load/store 2
Valid	**Type**	**Block Offset**	**Destination**	Load/store 3

Non-block Caches: Operation

- On a cache miss:
  - Search MSHRs for pending access to same cache block
    - If yes, just allocate new load/store entry
  - (if no) Allocate free MSHR
    - Update block address and first load/store entry
  - If no MSHR or load/store entry free, stall
- When one word/sub-block for cache line become available
  - Check which load/stores are waiting for it
    - Forward data to LSU
    - Mark loads/store as invalid
  - Write word in the cache
- When last word for cache line is available
  - Mark MSHR as invalid

Summary

- How to reduce cache hit time
  - Smaller cache, lower associativity, wide interfaces, pseudo -associativity
  - Multi-ported and multi-banked caches

- How to reduce cache miss rate
  - Larger caches, higher associativity, skew associativity, victim cache

- How to reduce cache miss penalty
  - Multi-level caches, sub-blocks, critical word first, write buffers, non-blocking caches