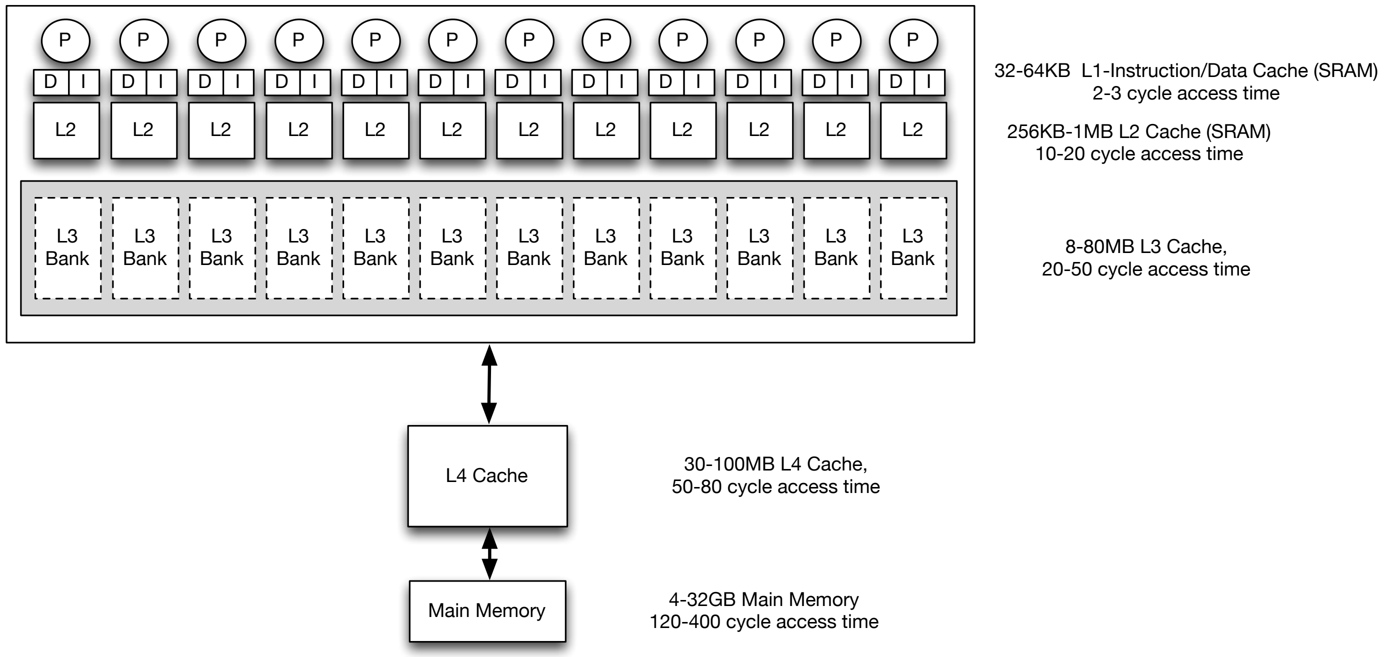
Chapter 6
Introduction to Memory Hierarchy Organization

Module 6.1 – Basic Cache Architecture 1

Example Configuration (2013)


P
D
I
L2
L3 Bank
L4 Cache
Main Memory
32-64KB L1-Instruction/Data Cache (SRAM) 2-3 cycle access time
256KB-1MB L2 Cache (SRAM) 10-20 cycle access time
8-80MB L3 Cache, 20-50 cycle access time
30-100MB L4 Cache, 50-80 cycle access time
4-32GB Main Memory 120-400 cycle access time

- Time to access the main memory used to be several clock cycles now several hundreds of cycles
- Keep small but fast storage near the processor: caches
  - Successful due to temporal & spatial locality
  - Hit rate of 70-95% is not uncommon

Cache Organization

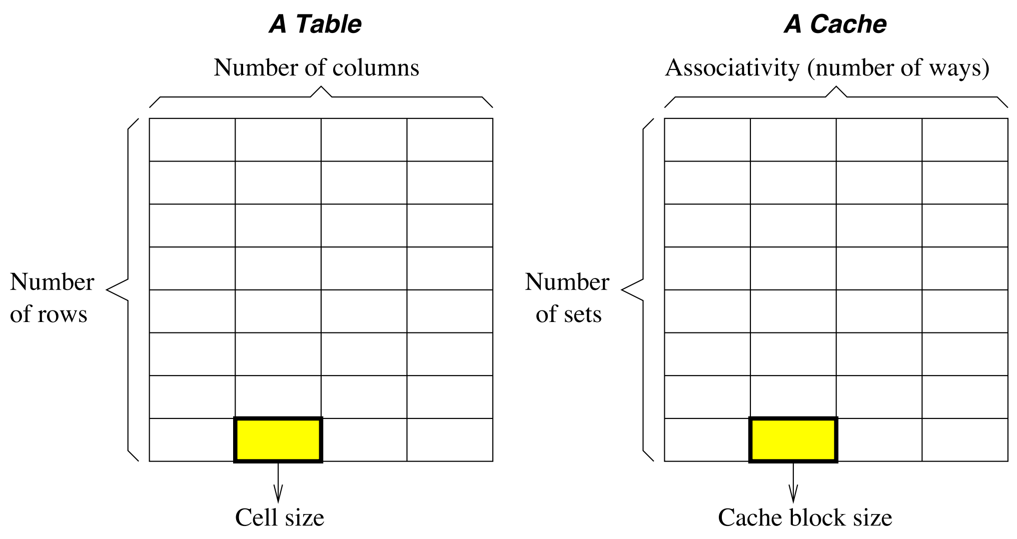
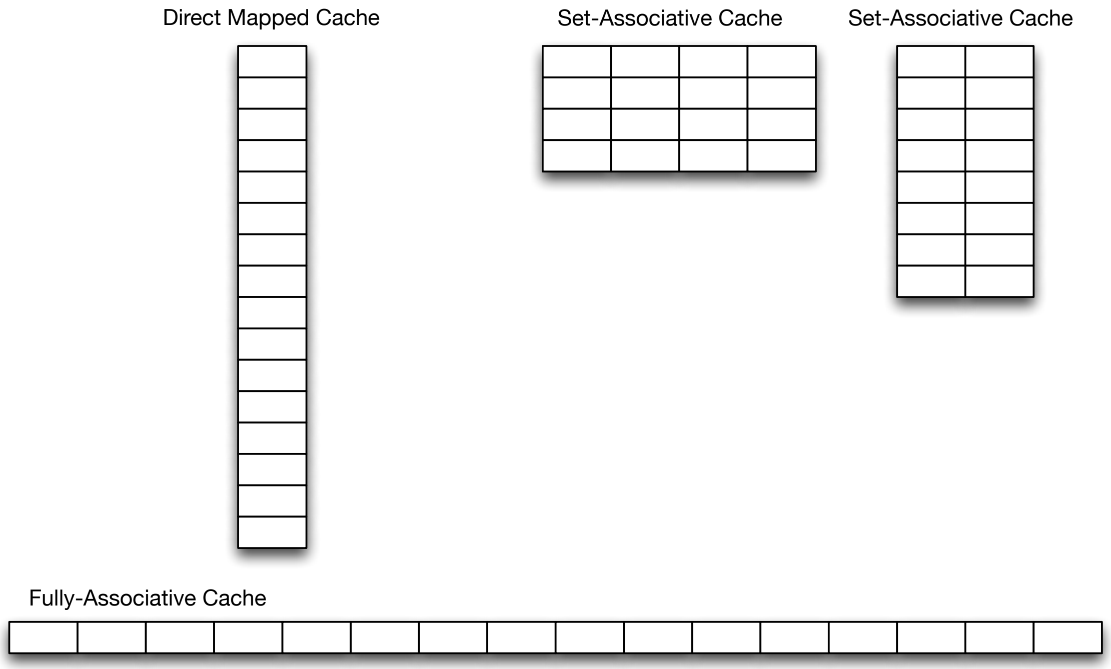


Figure 5.2: Analogy of a table and a cache.

- Similar to a table
  - a set = a row
  - a way (or associativity) = a column
  - a cache line = a cell in the table
- Unique to cache
  - Placement policy: which line to place a block from lower level memory
    - A block is placed in one of the ways of a set
  - Replacement policy: which block to evict when a new block is placed
    - A block in the set is selected for replacement

Various Cache Associativities



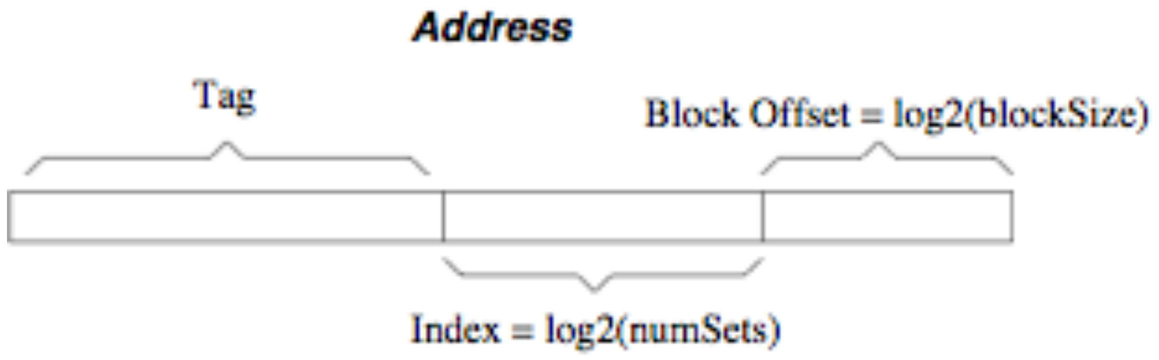
- Direct mapped cache
  - a block can be placed in only one line in the cache
- Fully associative cache
  - a block can be placed in any line in the cache
- Set-associative cache
  - a block can be placed in one of the ways of a set

Trade-offs in Associativity

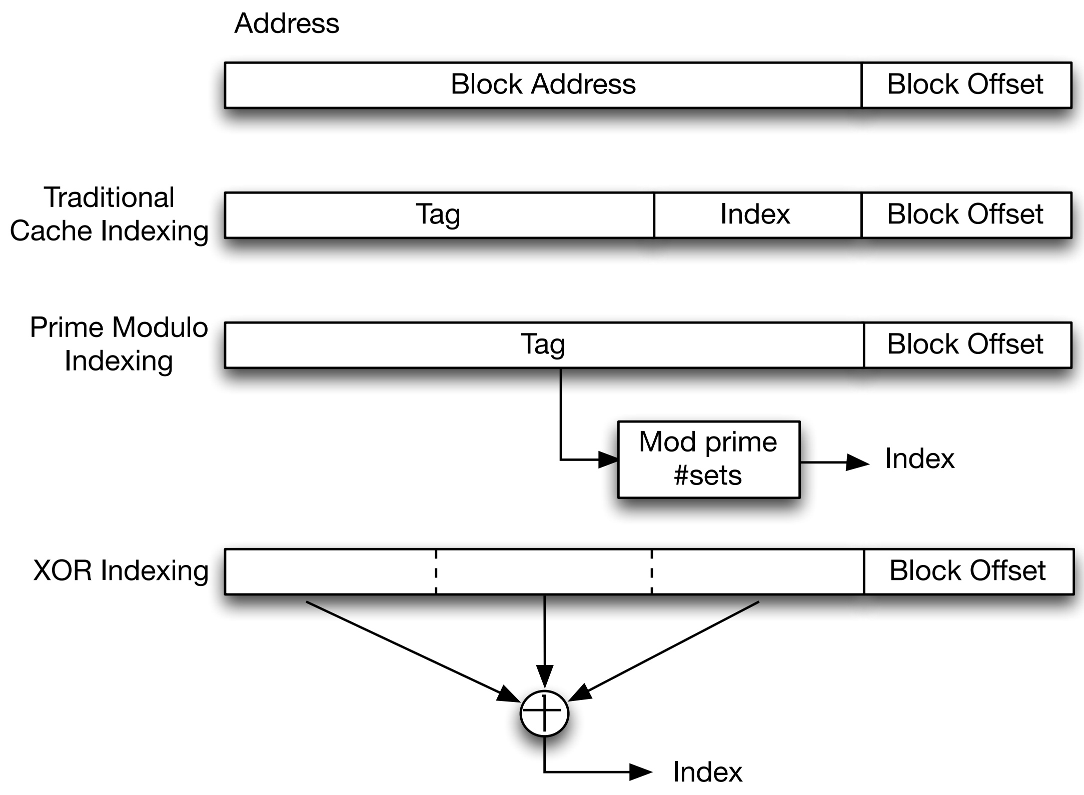
- Fully associative cache
  - Flexible placement of block in the cache
  - Usually has a lower cache miss rate (fraction of cache accesses that do not find the block in the cache)
  - But:
  - power hungry, must search the entire cache to find the block
- Direct-mapped cache
  - Rigid placement of a block in the cache
  - Usually has a higher cache miss rate
  - But: power efficient, only need to search in one place to find a block

How to Map a Block to a Set

- Achieved by a cache indexing function
  - setIdx = blkAddr **mod** numSets
- If numSets = power of 2, the indexing function does not require computation
- Typically, the index bits are selected from higher order bits right after the block offset bits
  - e.g. for a cache with 2048 sets and 64 byte blocks, the block offset uses 6 lowest order bits, and the index bits are taken from the next higher 11 bits

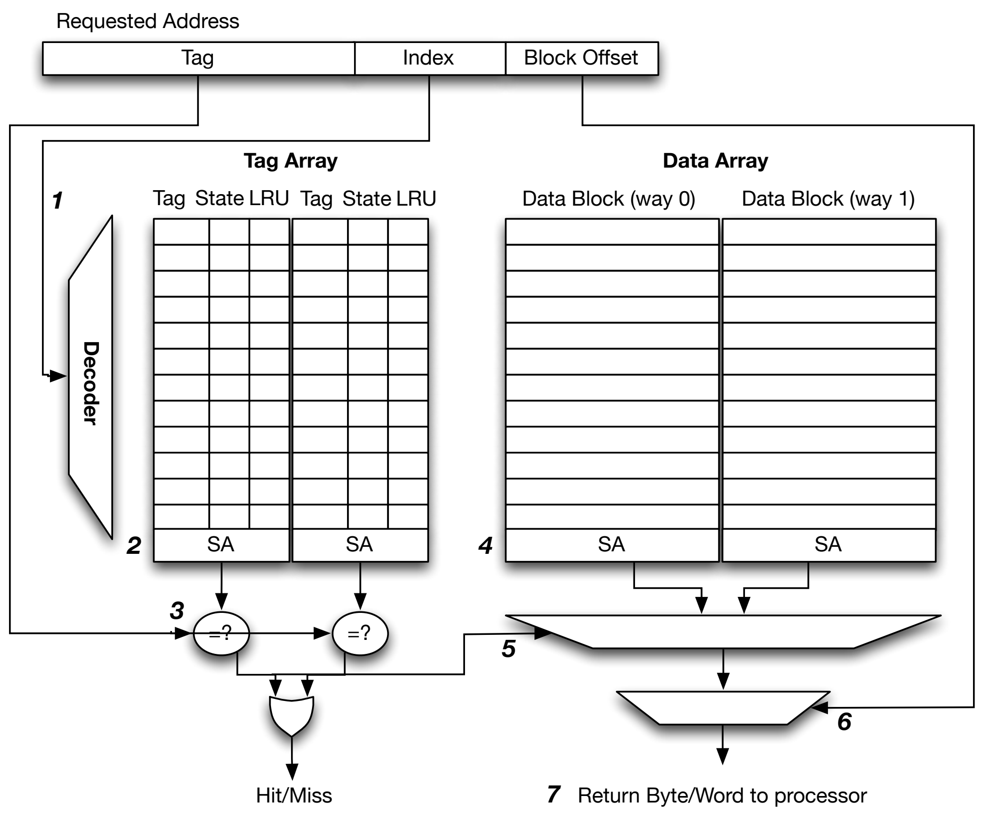


Other Cache Mapping Functions



- Prime modulo is resistant to non-uniform set utilization due to strided accesses
- XOR randomizes set mapping "cheaply"

Locating a Block in the Cache



Step 1: map block address to set

Step 2: Read out tags from set

Step 3: Compare address tag with stored tags: match means hit, mismatch means miss

Step 4: Read out data from set

Step 5: If hit (Step 3), select the block from the way that hits.

Step 6: Select the requested byte/word

Step 7: Return byte/word to processor

Replacement Policy

- Which block to evict to make room for a new block?
  - LRU: select the least recently used/accessed block
    - Usually perform well
    - Except when the working set is slightly larger than the cache
    - Expensive to implement for high associativity
  - FIFO: select the earliest block that was brought in
  - Pseudo-LRU: approximation to LRU
    - Behaves similar to LRU for half the most recently used blocks
    - Cheaper to implement than LRU
    - Idea: keep a tree of bits that records the most recent path to the block

LRU Implementation

- Assign a row and column to each way
- If hit in a way, set the row, unset the column
- Number of 1's in rows = MRU order

Cache Content:

way0	way1	way2	way3
A	B	C	D

Access Stream:

B, C, A, D

LRU Matrix Content

Initially

	way0	way1	way2	way3
way0	0	0	0	0
way1	0	0	0	0
way2	0	0	0	0
way3	0	0	0	0

After access to B

	way0	way1	way2	way3
way0	0	**0**	0	0
way1	**1**	**0**	**1**	**1**
way2	0	**0**	0	0
way3	0	**0**	0	0

After access to C

	way0	way1	way2	way3
way0	0	0	**0**	0
way1	1	0	**0**	1
way2	**1**	**1**	**0**	**1**
way3	0	0	**0**	0

After access to A

	way0	way1	way2	way3
way0	**0**	**1**	**1**	**1**
way1	**0**	0	0	1
way2	**0**	1	0	1
way3	**0**	0	0	0

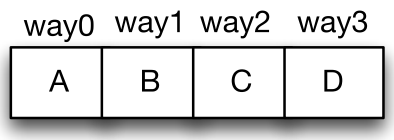
After access to D

	way0	way1	way2	way3
way0	0	1	1	**0**
way1	0	0	0	**0**
way2	0	1	0	**0**
way3	**1**	**1**	**1**	**0**

Pseudo-LRU Implementation

- LRU implementation = O(way^2), too expensive
- Approximate LRU with pseudo-LRU
- Pseudo LRRU complexity = O(way)

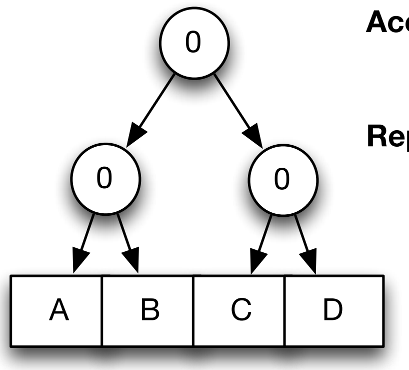
Cache Content:



Access Stream:

B, C, A, D, E

Pseudo LRU Tree

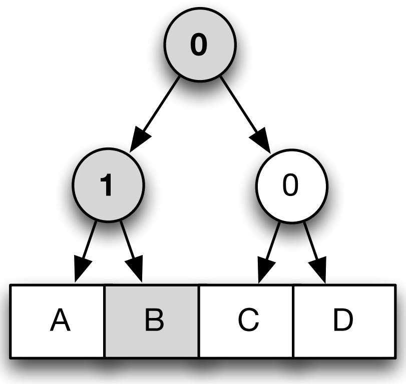


Access:
0 = Left, 1 = Right

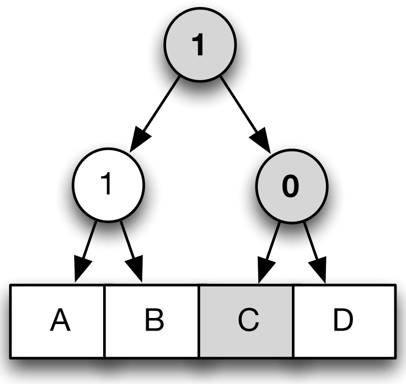
Replacement:
0 = Right, 1 = Left

Pseudo-LRU Replacement

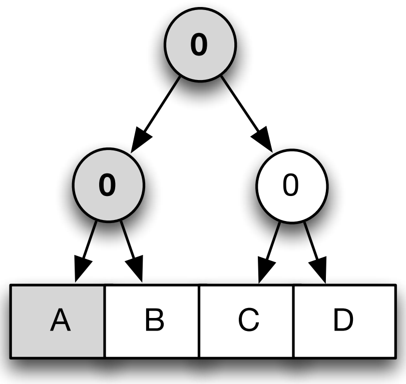
After access to B



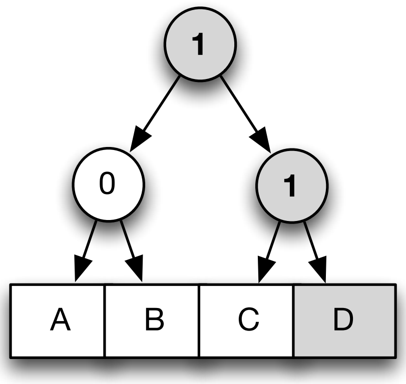
After access to C



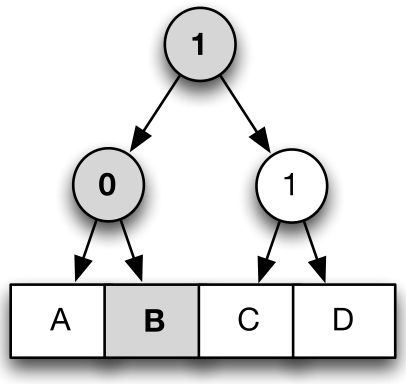
After access to A



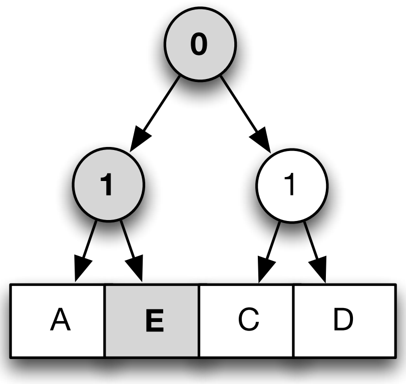
After access to D



Finding a block to replace
to make room for E
Block B is selected



After access to E,
E replaces B,
Bits flipped along the path



Write Policy: Write Through vs. Write Back

- When is a new value propagated to the lower level?
  - Immediately after a write: write through policy
  - Only when we need to, when the block is replaced from the cache: write back policy
- Write back policy tends to conserve bandwidth
  - Multiple writes on a block ->
    - multiple propagation in write through
    - only one propagation in write back
- Typical organization
  - L1 cache: write through
    - Bandwidth between L1 and L2 plentiful
    - L1 only protected by parity
  - L2 cache: write back
    - Bandwidth from L2 to lower level (often off-chip) memory is limited

Write policy: Write Allocate vs. No-Allocate

- Is a block allocated on write?
  - yes: write allocate
  - no: write no-allocate
  - in either case, block is always allocated on a read request
- Rationale for write no-allocate
  - If a block is going to be overwritten completely, why bother bringing it into the cache?
- Pitfalls for write no-allocate
  - The block that is written may be read or written again soon, would have been better to allocate the block
  - Larger cache block increases the odds of writing the block multiple times (at different bytes)
- Typical organization: write allocate

Module 6.1 – Basic Cache Architecture 2

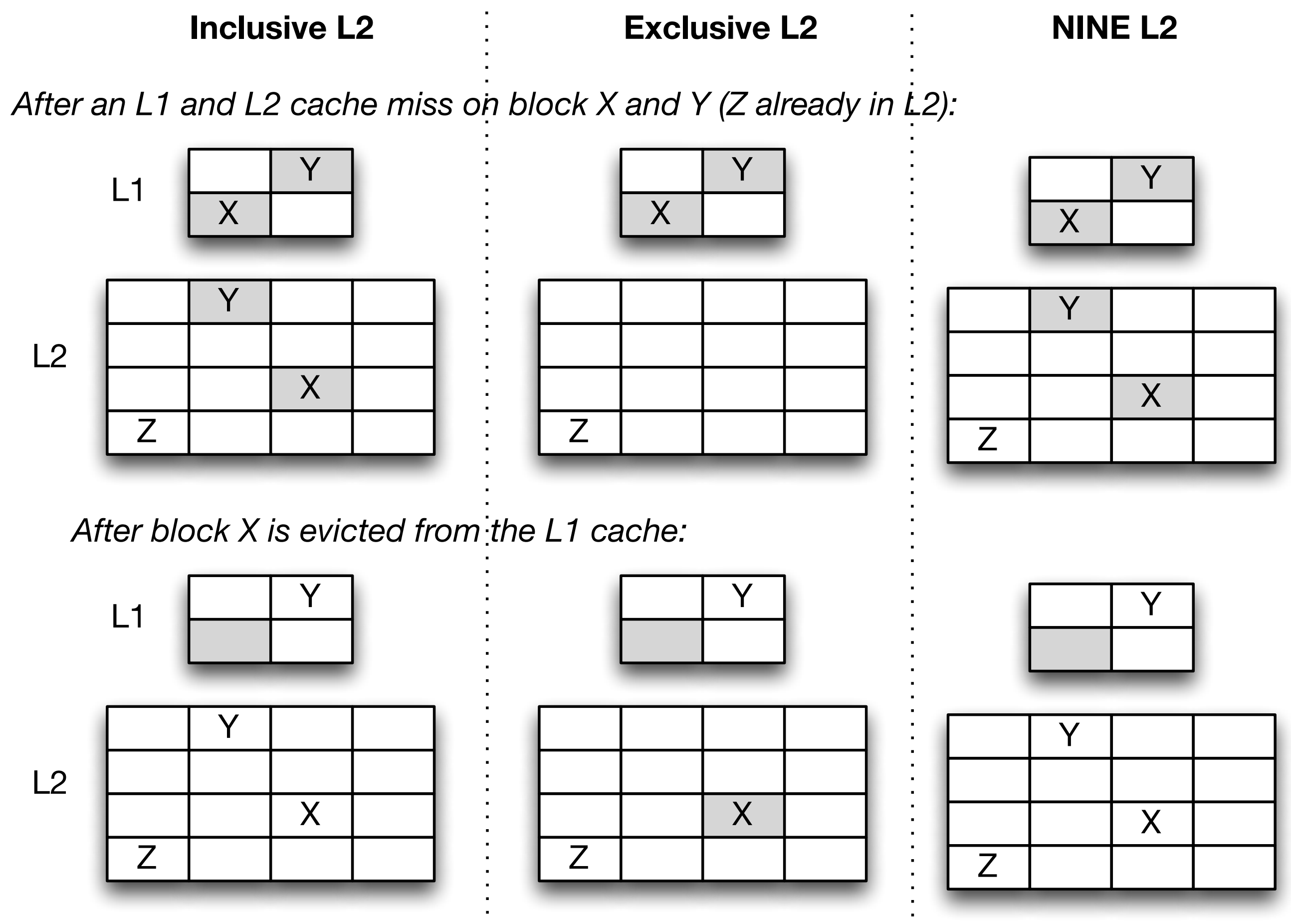
Inclusion Policy in Multi-Level Caches

- Should a block in the upper level cache be allocated in the lower level cache as well?
  - yes, always: Inclusion property
  - not always: Inclusion property not enforced
  - never: Exclusion
- If inclusion is maintained, should a block value the same in all levels? yes: value inclusion
- Why inclusion?
  - On an external request, we only need to check the lower level cache
    - If not found in the lower level, the upper level cache cannot possibly have the block
  - Reduces contention for cache tags at the upper level

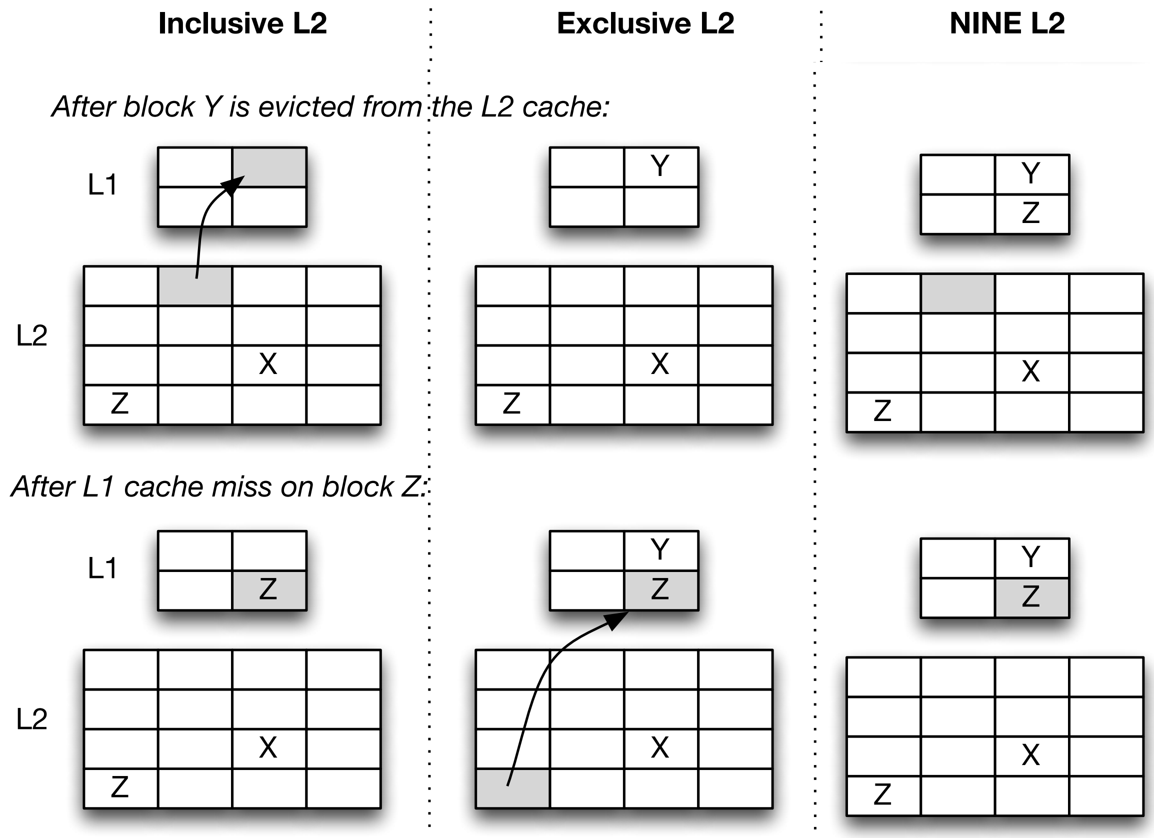
Inclusion Property

- Pitfalls for inclusion
  - Wasted cache capacity, esp. in multicore
    - e.g. AMD Barcelona quad core: 4x512KB L2 caches, 2MB L3 cache
- Protocol for enforcing inclusion
  - A fetched block is always placed in the lower level cache as well as the upper level cache
  - When a block in the lower level cache is replaced, probe the upper level cache to replace its copy as well

Comparing Inclusion Policies


Inclusive L2
Exclusive L2
NINE L2
After an L1 and L2 cache miss on block X and Y (Z already in L2):
L1
L2
X
Y
Z
After block X is evicted from the L1 cache:
L1
L2
X
Y
Z

Comparing Inclusion Policies



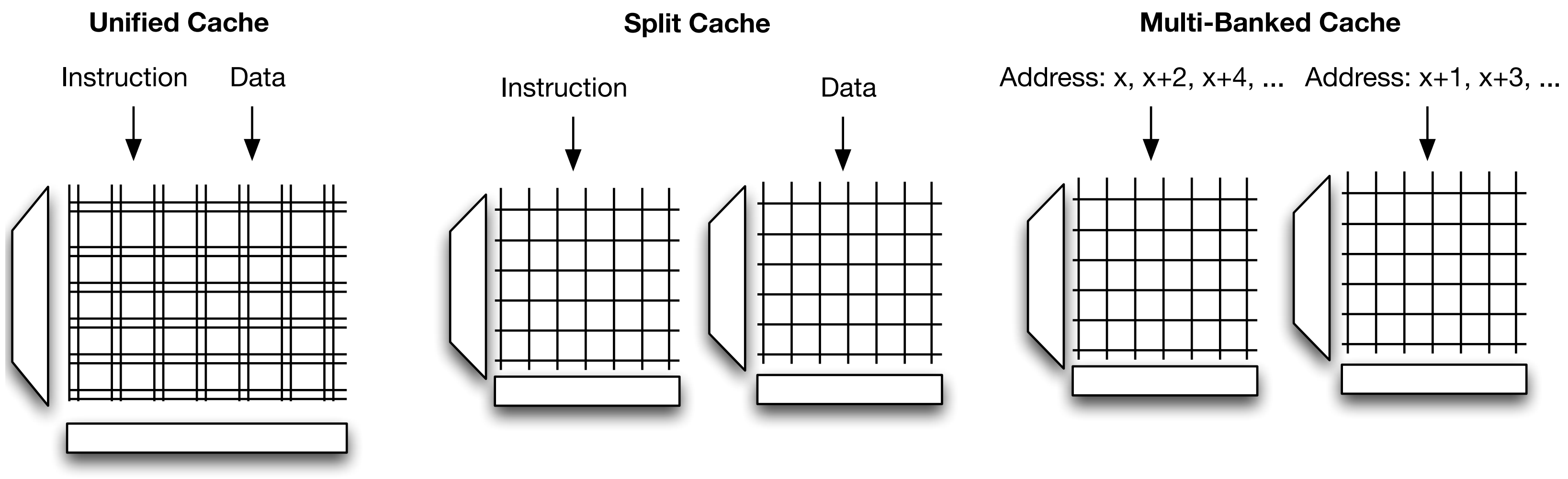
Inclusive L2
Exclusive L2
NINE L2
After block Y is evicted from the L2 cache:
L1
L2
Y
Z
X
After L1 cache miss on block Z:

Split Cache Organization

- Consider a unified L1 cache that holds instructions and data:
  - Each cycle, it must be accessed to supply instructions to the pipeline
  - Each cycle, several load/store instructions may access it
  - Hence, it needs to have many ports -> expensive and slow
- Solution: split into Instruction Cache + Data Cache
  - Instruction and data access no longer contending for the ports
- Less applicable for lower level caches: its access frequency is low

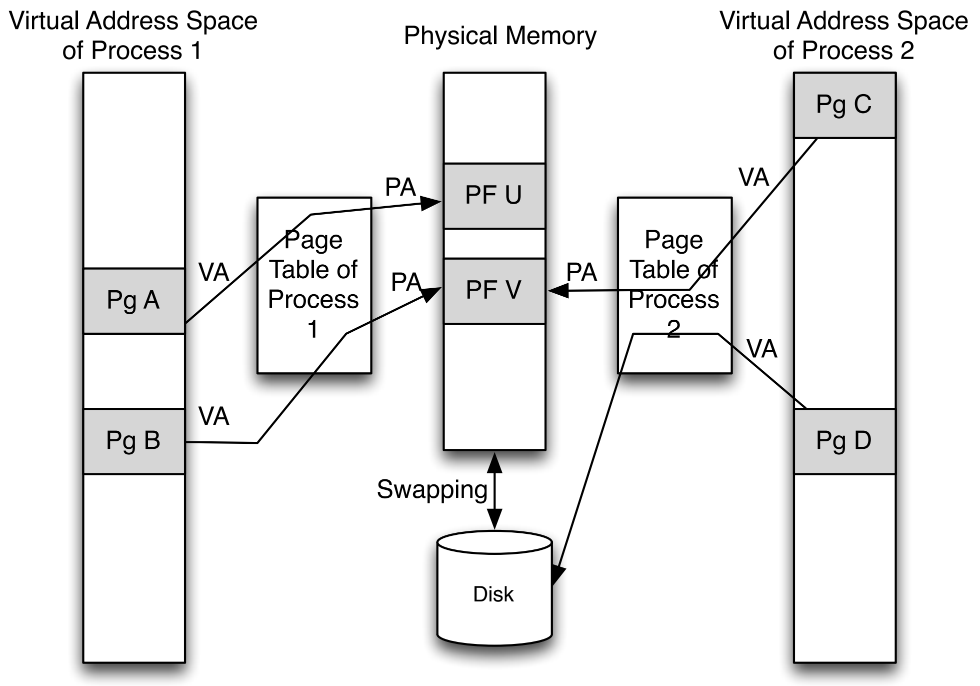
Illustrating Unified vs. Split vs. Multi-banked Cache

- Another way to split a cache is based on addresses => multi-banked cache

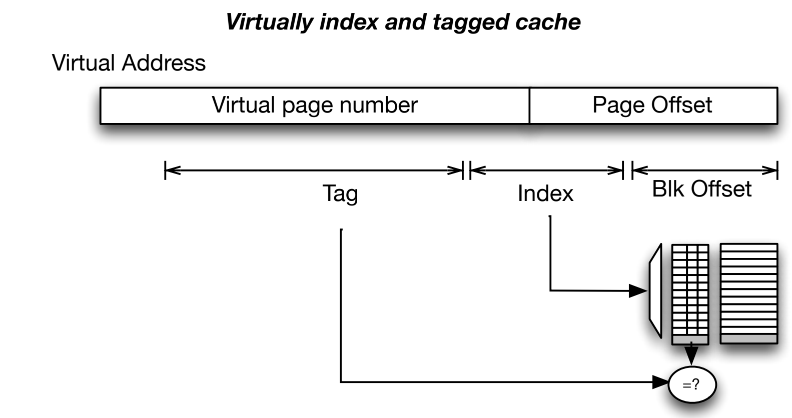


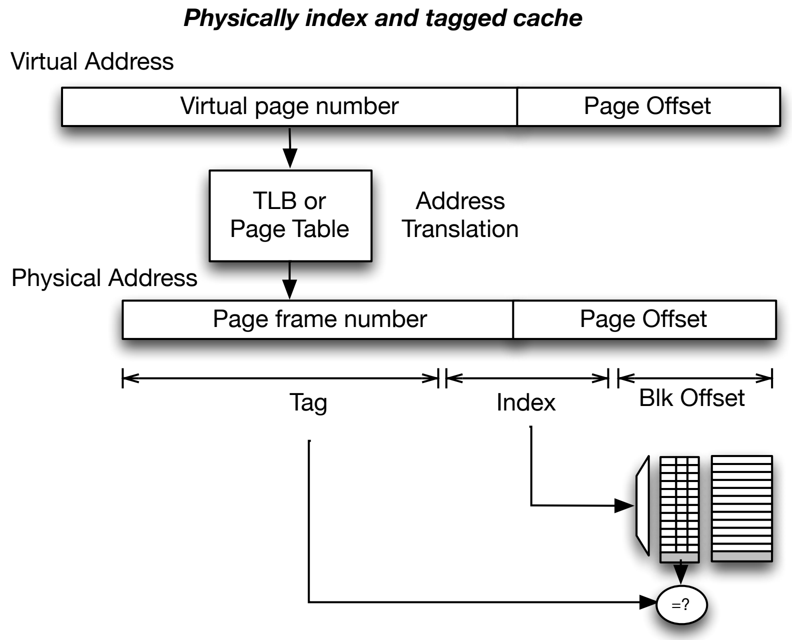
Cache Addressing

- Virtual memory system allows
  - » Program assumes it has access to the entire address space
  - » Multiple programs to be resident simultaneously in the system
- Achieved by separating addresses seen by program (virtual address) with actual addresses in memory (physical address)
  - » OS manages VA -> PA address translation at page granularity



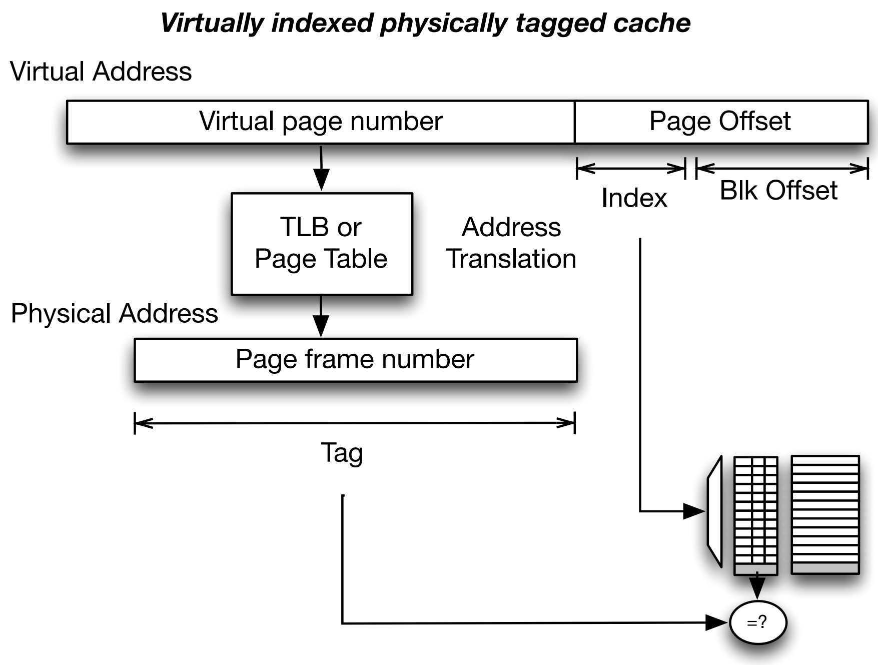
Cache Addressing





- Should physical or virtual address be used in addressing cache?
- Virtual
  - Can access cache directly
  - Content invalid on context switch
- Physical
  - Content valid across switches
  - Latency too high (TLB access first)
- What is a good compromise?

Cache Addressing



- Virtual indexing physical tagging
- Key insight: page offset bits equal in VA and PA
- Key feature: limit cache index using page offset bits
- Capacity/associativity limited

Types of Cache Misses

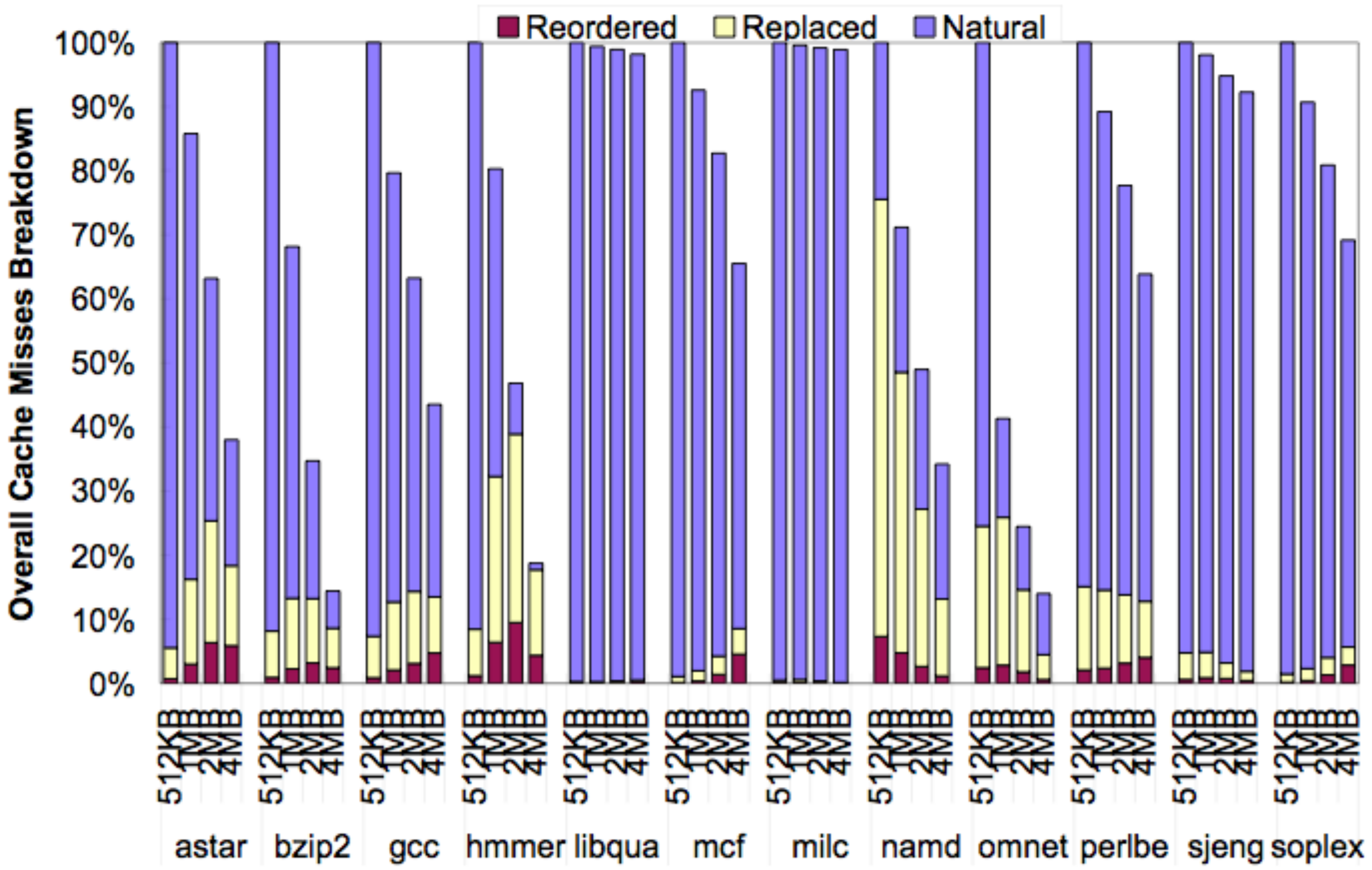
- Compulsory
  - misses required to bring blocks into the cache for the first time
- Conflict
  - misses that occur due to insufficient cache associativity
- Capacity
  - misses that occur due to limited cache size

- Coherence
  - misses that occur due to invalidation by other processors
- System Related
  - misses due to system activities such as system calls, interrupts, context switches, etc.

Factors Affecting Cache Misses

Parameters	Compulsory	Conflict	Capacity
Larger cache size	unchanged	unchanged	reduced
Larger block size	reduced	unclear	unclear
Larger associativity	unchanged	reduced	unchanged

Context Switch Misses



- Natural misses are reduced with larger cache size
- Context switch misses increase with larger cache size (until total working sets fit)

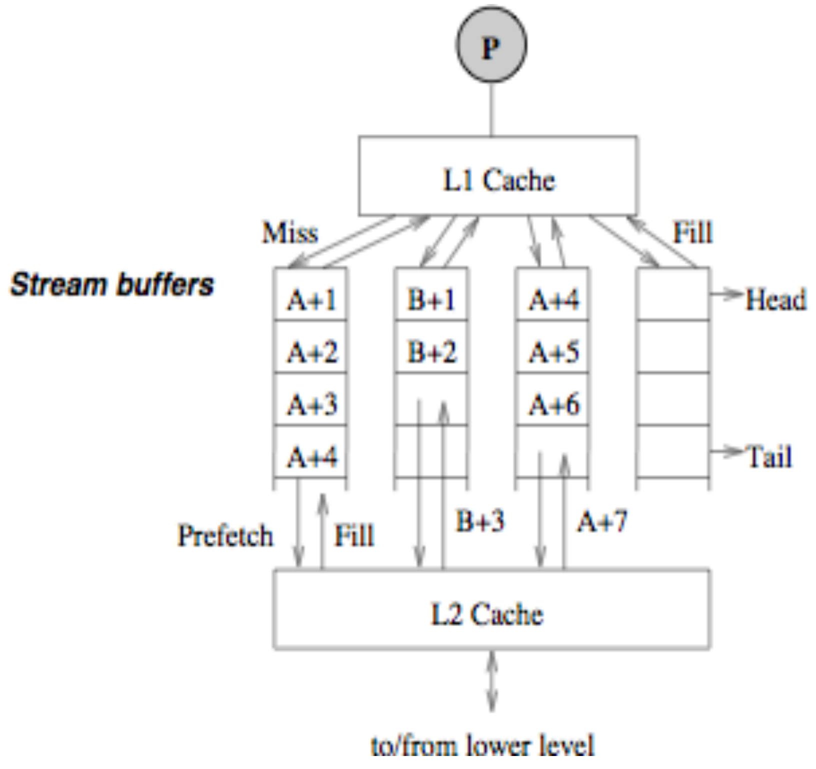
Cache Performance Metrics

- Average access time (AAT)
  - AAT = L1T + L1MR * L2T + L1MR * L2MR * L2MT
    - LxT = Level-x access time
    - LxMR = Level-x miss rate
    - LxMT = Level-x miss penalty
- Cycles per Instruction
  - cpi = (1-h2-hm) * cpi0 + h2 * t2 + hm * tm
    - h2 = instructions that access L2 cache
    - hm = instructions that access memory
    - cpi0 = cpi of instructions that do not access L2 cache
    - t2 = average penalty of an instruction that accesses L2 cache
    - tm = average penalty of an instruction that accesses memory

Prefetching

- A technique to bring data into the cache before the program accesses it
  - Software prefetching: using special instructions inserted by the compiler into code
  - Hardware prefetching: using an engine that detects access patterns and prefetch data
- Important metrics
  - Coverage = fraction of misses prefetched
  - Accuracy = fraction of prefetches that are useful
  - Timeliness
- Basic techniques
  - Sequential prefetching: detect sequential accesses to cache blocks, extrapolate trend and prefetch
  - Stride prefetching

Stream Buffers [Jouppi 90]

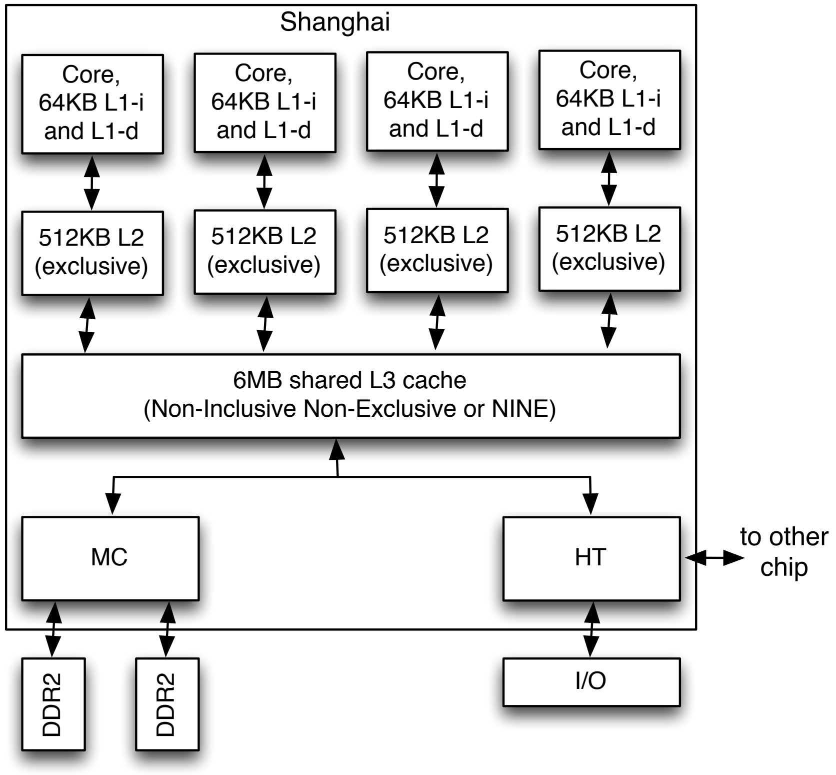


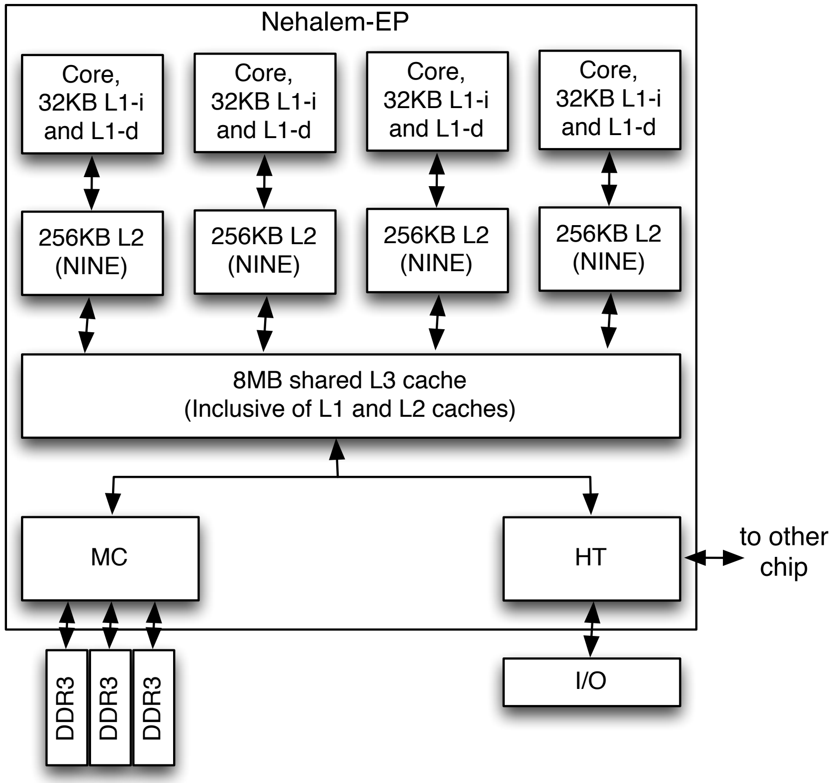
- On a miss, check the stream buffers for a match
  - If match, move block to cache
  - Otherwise, allocate a new stream buffer
- Start prefetch the continuation of the stream until the stream buffer is full
- Variants of stream buffers are widely implemented (Pentium, PowerPC, etc.)

Prefetching in Multiprocessor Systems

- Harder to be timely
- Too early
  - May be evicted before accessed (uniprocessor)
  - May evict a more useful block (uniprocessor)
  - May be invalidated by other processors (multiprocessor)
  - May cause downgrade on other processor (multiprocessor)

Example: AMD Shanghai and Intel Nehalem





Alternative View of Memory Hierarchy

- Cache: everything (placement, replacement) is managed by hardware
- vs. Scratchpad: everything is managed by software

- Servers, general purpose systems use caches
- But many embedded systems use scratchpad
  – Scratchpad provides predictability of hits/misses
  – Important for ensuring real time property

Other Topics

- Pipelined access
- Sequential tag-data lookups, parallel tag-data lookups
  - Way predicting cache
- Stack distance profile
- Nonblocking cache
- Write back queue and write combining