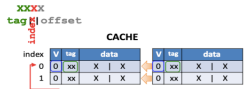
Temporal locality: the same thing again soon.
Spatial locality: something near that thing soon.
$t_{avg} = t_{hit} + \%miss \times t_{miss}$. n bits entries -> 2^n 个。
entry=row=**cache line**=cache block。lookup 顺序：index-> tag->V。

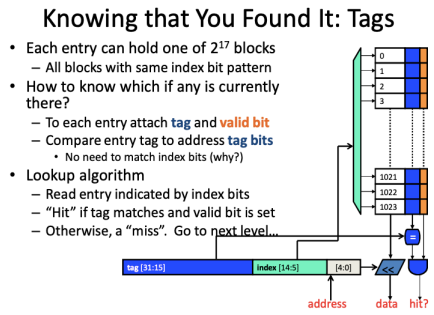
increase block size。一行里能放两个。

Fully-Associative Caches. 舍弃 index。用 tag 来找

Pros: no conflicts, Cons: t_{hit} 增加
Set-Associative Caches:

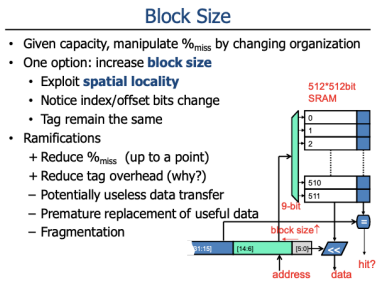


两个表存同一个 index 但是 tag 不同的情况
Misses: **Cold** (never seen address), **Conflicts** (cache associativity is too low), Capacity (cache is too small)
ABC of Caches: Associative 上升(Conflicts 下降, hit time 上升); Blocksize 上升(cold 下降，Conflict 上升); Capacity 上升(Capacity miss 下降，hit time 上升)



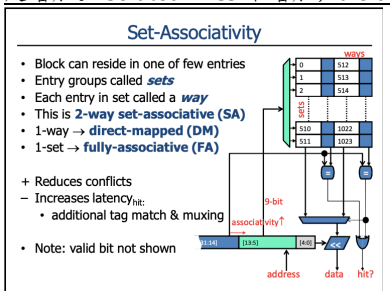
5 bits offset, 10 bits indext, 17 bits tag, 1bit valid
Overhead (17+1)*1024=2.2KB
tags. 2.2/32=6%
CPII$ = %miss * t_{miss}
CPID$ = $\%_{load/store}$ * %miss * t_{miss}

Cache Controller: Finite State Machine
i. Remember miss address
ii. Accesses next level of memory
iii. Wait for response
iv. Write data/tag into proper location
v. All of this happens on the fill path -> some times called back side



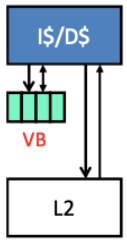
Increase Capacity -> %miss 下降，hit time 上升
减少 tag overhead.
9-bit index, 6-bit offset, 17-bit tag
+Spatial prefetching
-Interference

Blocksize 增加, t_{miss} 不一定增加。isolated miss 不增加，a cluster of miss will suffer



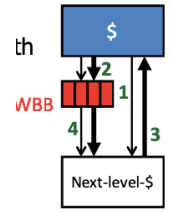
5-bit offset, 9-bit index, 18-bit tag
index to find set
read data/tags in parallel.
match and valid bit hit!

Replacement Policy: Random, FIFO, LRU (least recently used), **NMRU** (not most recently used), Belady's
3C: Cold(infinite cache 都可以出现)，Capacity(FA 都能出现)，Conflict(除了前面都是)，Coherence(miss due to external invalidations)
VictimBuffer(VB):small fully-associative cache
i. sits on I$/D$ miss path
ii. small so very fast (比如只有 8 个 entries)
iii. Blocks kicked out of I$/D$ placed in VB
iv. On miss, check VB: hit? Place block back to I$/D$



并没有减少%miss，但是减少了平均的 tmiss
Overlapping Misses: **Lockup Free Cache**
Lockup free: allow other accesses while miss is pending
就是在第一个语句 miss 的时候，第二个语句可以直接进行对 Processors can go ahead despite D$ miss (out-of-order)
Implementation: miss status holding register (**MSHR**)
1) miss address2) chosen entry3) requesting instructions
Common scenario: hit under miss -> handle hits while miss is pending -> easy; Less Common: miss under miss -> need multiple MSHRs -> search to avoid frame conflicts
Loop interchange: spatial locality; Loop blocking: temporal locality; Loop fusion: multiple consecutive loops (合并 loop)
Prefetching: put blocks in cache proactively/speculatively
Stride-based sequential prefetching(Can also do N blocks ahead to hide more latency); Address-prediction(Needed for non-sequential data: lists, trees); Correlating predictor(Large table stores (miss-addr -> next-miss-addr) pairs); Content-directed or dependence-based prefetching; Jump pointer; Cache-conscious layout/malloc
Write-through: immediately send the write to the next level, require bus bw. next-level handle small writes
即上一层的 t_{miss} = 下一层的 t_{avg} ; Upper components (I$/D$) emphasize low t_{hit}; Moving down emphasize low %miss.

Write-back: when block is replaced; need dirty bits, 2nd-level cache use it
Write-allocate: fill the block from next level, then write it
Write-non-allocate: just write to next level, no allocate



Split (insns and data in different caches); Unified
Inclusion (A block in the L1 is always in the L2); exclusion
Spectre targets branch predictions; **Meltdown** targets exception handling; **side channel attack**(unintended info), covert channel attack(Altering a system so that it will disclose info). Spectre and Meltdown are both covert channel attacks
Flush and Reload Attack(learn reload condition).
Spectre:Train predictor to take one path, switch to another; protected address -> no exception(due to no commit)
vulnerable machine: Cache, OoO, Branch Predictor.
Meltdown: access illegal memory, fail, mem in cache-> KPTI
Multithread trade latency for throughput; No state: ALUs; Persistent hard state ("context"): PC registers; Persistent soft state: cache, bp; Transient state: pipeline latches
Coarse-Grain Multithreading (CGMT) preferred thread; no pipeline partition; tolerate only long latencies
Fine-Grain Multithreading (FGMT) Switch threads every cycle (round-robin); Dynamic pipeline partition. more thread
Simultaneous Multithreading (SMT) Out-of-order + FGMT（Aka hyper-threading).一个 cycle 里多个语句 Physical regfile and insn buffer shared at fine-grain； Physically ordered Structure (ROB/LSQ) shared.
physical registers = (#threads * #arch-regs) + #in-flight insns
map table entries = (#threads * #arch-regs)
Caches are shared naturally; TLBs need explicit threads IDs to be shared; BTB: Thread IDs cost low; BHT: Thread IDs cost high; Ordered soft-state should be replicated (BHR, RAS)
Multicore: multiple separate pipelines
Multithreaded processor: a single large pipeline
CPU and memories connected by memory bus; System Bus; NIC=Network Interface Controller; DMA (direct memory access); OS: a super-privileged process; **Exceptions:** synchronous, generated by running app; **Interrupts:** asynchronous events generated externally
Virtualizing Processors: time-share the resource -> timer
motivation: large VM size, Simple management, protection

VM: treat mem as a Cache + add a level of indirection (address translation); write back, write allocate, large blocks and pages

Page Faults: Page table entry indicates virtual address not in memory; OS has full control over placement

Logically: translation performed before every insn fetch, load, store; Physically: hardware acceleration removes translation overhead; Isolation, Protection, Inter-process communication

Separate Virtual Address Space; Each process has its own virtual address space; trap into OS if violation occurs

VA->PA: translation; Split VA into virtual page number (VPN) & page offset (POFS); VA -> PA = [VPN, POFS] -> [PPN, POFS]

Each process allocated a page table (PT);Page Table Size: 32-bit VA->4GB vitual mem->4GB/4KB page size=1M PTEs->1M PTEs*4B-per-PTE=4MB

Multi-Level Page Table: Lowest-level tables hold PTEs; Upper-level hold pointer to lower-level

ii. Example: 2MB code, 64KB stack, 16MB heap
   1) Each second-level maps 4MB of virtual address
   2) 1 for code, 1 for stack, 4 for heap, (+1 1st-level)
   3) 7 total pages = 28KB (<<4MB) ?

Most caches are physically addressed, not care protection

Page Table Entries (PTE) can be cached

Hardware truism

Functionality problem? add indirection

Performance problem? add cache

Translation Lookaside Buffer (TLB): cache translation; small cache: 16-64 entries; Associative (4+way or fully associative)

Physical caches: 1) Indexed and tagged by physical address; 2）not share caches between app and OS; 3) Cached inter-process communication works; 4) Slow: adds at least one cycle to t_{hit}

TLB: 1) define by virtual addresses;2) Indexed and tagged by virtual addresses;3) Flush across context switches;4)Or extend with process identifier (x86)

2 way to look at VA: Cache: tag+index+offset; TLB:VPN+POFS

VPN/index: must not overlap; Only if (cache size) / (associativity) <= page size, Index bits same in virtual & physical addresses。 assocaitivity allows bigger caches

TLB: Capacity; Associativity (At least 4-way, fully associative common); Block size: 如果是 2 表明，2 consecutive VPs share a single tag; Rule of thumb: TLB should "cover" L2 contents

TLB Misses: Software-managed TLB, ARM (+keep page table format flexible; -one or two mem accesses + OS call (pipeline flush); Hardware-managed TLB, x86 (+latency: save cost of OS call (avoid pipeline flush; -page table format is hard-coded)

Multicores: Shared Memory Multiprocessors; Thread-Level Parallelism (TLP); Collection of asynchronous tasks; Data shared "loosely";

Uniprocessor Concurrency: Software "thread";Independent flow of execution; System software (OS) manages threads; Quickly swapping threads gives illusion of concurrent execution;

Multithreaded Programming Model: (Each thread has a private stack frame for local variables); A thread switch can occur at any time(Pre-emptive multithreading by OS)

Shared Memory Implementation: Multiplexed uniprocessor; Runtime system or OS swap threads -> Interleaved, but no parallelism; Hardware Multithreading(Tolerate pipeline latencies, higher efficiency -> Same interleaved shared-memory model); Multiprocessing: Multiply execution resources, higher peak performance; Same interleaved shared-memory model; Foreshadowing: allow private caches, furthre disentangle cores

Simplest Multiprocessor: Replicate entire processor pipeline

Low-level primitive: lock(acquire(lock) and release(lock)); higher level: semaphore, mutex; Barrier synchronization

Spin lock: Software lock implementation: acquire is not atomic

SYSCALL lock: Kernel can disable interrupt; (-Large system call overhead)

Better Spin Lock: use atomic Swap; A0: swap r1,0(&lock); A1: bnez r1,A0 (if busy, no change; if free, acquire it)

RISC Test-And-Set: ll/sc: load-locked / store-conditional; (ll r1,0(&lock); // potentially other insn; sc r2,0(&lock)); 每一次都要 store 一次，overhead 严重，useless

Test-and-Test-and-Set Locks
First test, then swap
Lock release by one processor -> create "free for all" by others

```
A0: ld r1,0(&lock)
A1: bnez r1,A0
A2: addi r1,1→r1
A3: swap r1,0(&lock)
A4: bnez r1,A0
```

Software queue lock: Passes lock from one processor to the next, in order (only next see the lock)

Coarse-grain locks: correct, but slow(one clock); **Fine-grain locks:** parallel, but difficult(one per record); **Multiple locks**(must acquire both id_from, id_to locks, may deadlock, Solution: Always acquired multiple locks in same order)

Transcational Memory: No locks, just shared data; Execute critical section speculatively, abort on conflicts

Cache Incoherence: write-back 有滞后性，访问值出错

Write-through 可以立即写，但是每个 core 的 cache 不会更新

Bus-based: All processors see all requests at the same time, same order; 3 processor-initiated events (ld,st,WB); 2 remote-initiated events(LdMiss, StMiss);

VI (valid-invalid) protocol: aka MI 自己 V，别人 I, send dirty

Only 1 cached copy allowed in entire system, overhead 重

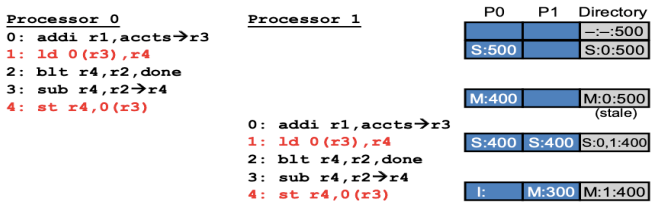
MSI (modified-shared-invalid); V state -> M + S

Upgrade miss(Delay to acquire write permission to read-only block); Coherence miss(Miss to a block evicted by another processor's requests)

MESI: Exclusive Clean.Load transition to E if no other processors is caching the block, otherwise S

Problem 1: N^2 bus traffic(All N processors send their misses to all N-1 other processors); N^2 processor snooping bandwidth

Directory Coherence Protocols(Extend memory to track caching info; MSI Directory Protocol



Flip Side: Latency; Unshared; More hops; Complexity

Write: don't want to stall-> WBB but lead to strange behavior

Sequential consistency (SC) (MIPS)(All see in program order)

Processor consistency (PC) (x86) (can defer but in order)

Release consistency (RC) (ARM) (un-ordered)

Fences: stall execution until write buffers are empty

Flynn's Taxonomy

SSID	Single Instruction, Single data	Traditional uniprocessor
SIMD	Single Instruction, Multiple data	Vector machines, graphic engines
MIMD	Multiple Instruction, Multiple data	Multicores
SPMD	Single Program, Multiple data	MIMD machine, each node executes the same code
MISD	Multiple Instruction, Single data	Systollic data

1st tier are top-of-rack (ToR) switches; Custom Interconnect (Known topology, trusted environment): Mesh, Torus

Infiniband Network: Multiple vendors, low latency, Remote Direct Memory Access (RDMA); MPI (Message Passing Interface) is de facto standard

SIMD, 对一个指令使用多个 data 进行执行

MISD, Systollic Arrays, 对一个 data 用多个指令重复执行

Tensor;$t_{MISS,L2} = t_{HIT,M}$

64 bytes X 8 / 128 bits/transfer = 4 bus transfers/access

2.66 GHz processor freq. / 133 MHz bus freq. = 20 processor clocks per transfer； 4 transfers X 20 clocks/transfer = 80 clocks to transfer data from main memory

CPI = CPI_{BASE} + (%load/store) x (%miss) x (clocks/miss)