Virtual Memory and Cache

Why virtual memory?

- With a 32-bit address you can access 4 GB of physical memory (you will never get the full memory though)
  - Seems enough for most day-to-day applications
  - But there are important applications that have much bigger memory footprint: databases, scientific apps operating on large matrices etc.
  - Even if your application fits entirely in physical memory it seems unfair to load the full image at startup
  - Just takes away memory from other processes, but probably doesn't need the full image at any point of time during execution: hurts multiprogramming
- Need to provide an illusion of bigger memory: Virtual Memory (VM)

Virtual memory

- Need an address to access virtual memory
  - Virtual Address (VA)
- Assume a 32-bit VA
  - Every process sees a 4 GB of virtual memory
  - This is much better than a 4 GB physical memory shared between multiprogrammed processes
  - The size of VA is really fixed by the processor data path width → width of org file
  - 64-bit processors (Alpha 21264, 21364; Sun UltraSPARC; AMD Athlon64, Opteron; IBM POWER4, POWER5; MIPS R10000 onwards; Intel Itanium etc.) provide bigger virtual memory to each process
  - Large virtual and physical memory is very important in commercial server market: need to run large databases

Addressing VM

- There are primarily three ways to address VM
  - Paging, Segmentation, Segmented paging
  - We will focus on flat paging only
- Paged VM
  - The entire VM is divided into small units called pages
  - Virtual pages are loaded into physical page frames as and when needed (demand paging)
  - Thus the physical memory is also divided into equal sized page frames
  - The processor generates virtual addresses
  - But memory is physically addressed: need a VA to PA translation

VA to PA translation

- The VA generated by the processor is divided into two parts: VPN | page offset
  - Page offset and Virtual page number (VPN)
  - Assume a 4 KB page: within a 32-bit VA, lower 12 bits will be page offset (offset within a page) and the remaining 20 bits are VPN (hence 1 M virtual pages total)
  - The page offset remains unchanged in the translation
  - Need to translate VPN to a physical page frame number (PPFN)
  - This translation is held in a page table resident in memory: so first we need to access this page table
  - How to get the address of the page table?

VA to PA translation

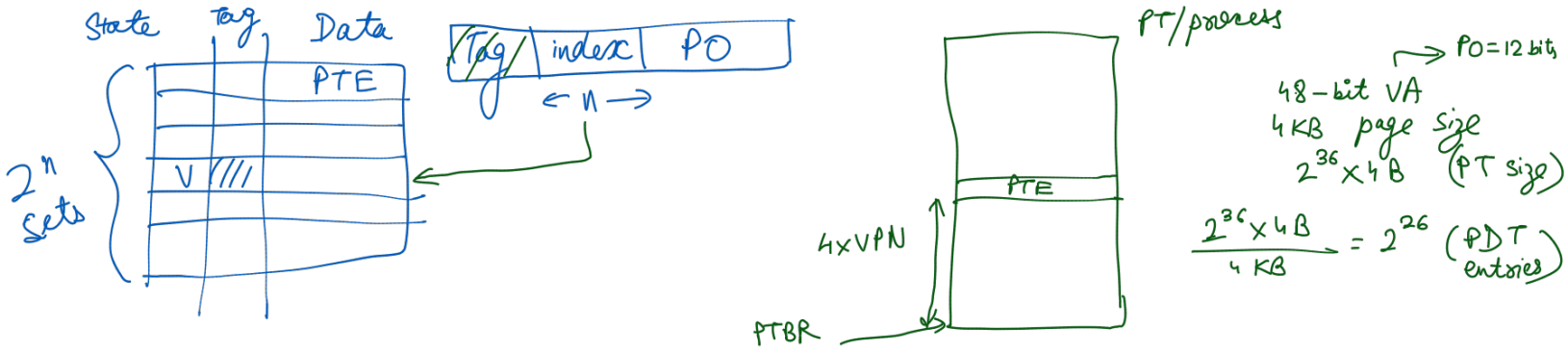
- Accessing the page table
  - The Page table base register (PTBR) contains the starting physical address of the page table (assuming a single-level page table)
  - PTBR (cr3 register in x86) is normally accessible in the kernel mode only
  - Assume each entry in page table is 32 bits (4 bytes)
  - Thus the required page table address is
    PTBR + (VPN << 2) Since each entry is 4 B.
  - Access memory at this address to get 32 bits of data from the page table entry (PTE)
  - These 32 bits contain many things: a valid bit, the much needed PPFN (may be 20 bits for a 4 GB physical memory), access permissions (read, write, execute), a dirty/modified bit etc.

Page fault

- The valid bit within the 32 bits tells you if the translation is valid
- If this bit is reset that means the page is not resident in memory: results in a page fault
- In case of a page fault the kernel needs to bring in the page to memory from disk
- The disk address is normally provided by the page table entry (different interpretation of 31 bits)
- Also kernel needs to allocate a new physical page frame for this virtual page
- If all frames are occupied it invokes a page replacement policy

VA to PA translation

- Page faults take a long time: order of ms
  - Need a good page replacement policy
- Once the page fault finishes, the page table entry is updated with the new VPN to PPFN mapping
- Of course, if the valid bit was set, you get the PPFN right away without taking a page fault
- Finally, PPFN is concatenated with the page offset to get the final PA PPFN | Offset
- Processor now can issue a memory request with this PA to get the necessary data
- Really two memory accesses are needed 1) PTE 2) actual data
- Can we improve on this?



TLB

- Why can't we cache the most recently used translations?
  - Translation Look-aside Buffers (TLB)
  - Small set of registers (normally fully associative)
  - Each entry has two parts: the tag which is simply VPN and the corresponding PTE
  - The tag may also contain a process id ASID
  - On a TLB hit you just get the translation in one cycle (may take slightly longer depending on the design)
  - On a TLB miss you may need to access memory to load the PTE in TLB (more later)
  - Normally there are two TLBs: instruction and data

Page table design

- Reserving a large contiguous memory space for the page table is wasteful
  - The program doesn't need the entire page table at any point in time
  - The entire page table may not be needed for running the program
- The page table is usually paged
  - Need a "page directory table" to locate pages of the page table
  - Since the page directory table is large, that is also paged
  - General principle: any level of page table that is larger than a page is paged to avoid reserving contiguous memory larger than a page

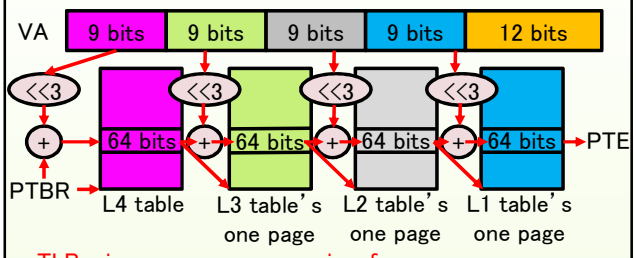
PD entries = $\frac{2^{36} \times 8B}{4 KB} = 2^{27}$ } L2

PD size = $2^{27} \times 8$ B

PD' entries = $\frac{2^{27} \times 8B}{4 KB} = 2^{18}$ size = $2^{18} \times 8B$ } L3

PD'' entries = $\frac{2^{18} \times 8B}{4 KB} = 2^9$ size = $2^9 \times 8B = 4 KB$ } L4

VA to PA translation



TLB misses are very expensive: four memory accesses

Two levels of TLB with very large L2 TLB

Small page structure caches (PSCs) or page walk caches (PWCs) to cache recently used L4, L3, L2 table's entries

TLB caches L1 entries

VA to PA translation

- Step#1: Look up L1 TLB; on a hit, return PTE
- Step#2: Look up L2 TLB; on a hit, insert PTE in L1 TLB and return PTE
- Step#3: Invoke hardware page walker to handle TLB miss (older processors used software TLB miss handlers, but with multi-level page table, that is slow)
  - Step#3A: Look up PSCs in parallel and identify the lowest level PSC that returns a hit among L4, L3, L2; suppose n is the lowest level that returns hit (n=5 if no hit in any PSC)
  - Step#3B: start retrieving entries at levels n-1, n-2, ⋯ one at a time by first looking up the traditional CPU cache hierarchy and on a miss main memory; continue until PTE is obtained; insert each level entry in PSCs and CPU caches
  - Step#3C: insert PTE in both L1 and L2 TLBs

Caches

- Once you have completed the VA to PA translation you have the physical address. What's next?
- You need to access memory with that PA
- Instruction and data caches hold most recently used (temporally close) and nearby (spatially close) data
- Use the PA to access the cache first
- Caches are organized as arrays of cache lines
- Each cache line holds several contiguous bytes (32, 64 or 128 bytes)

cache line = cache block

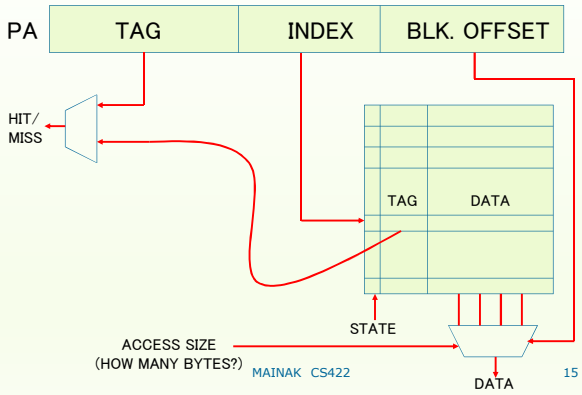
Addressing a cache

- The PA is divided into several parts

TAG	INDEX	BLK. OFFSET

- The block offset determines the starting byte address within a cache line
- The index tells you which cache line to access
- In that cache line you compare the tag to determine hit/miss

Addressing a cache



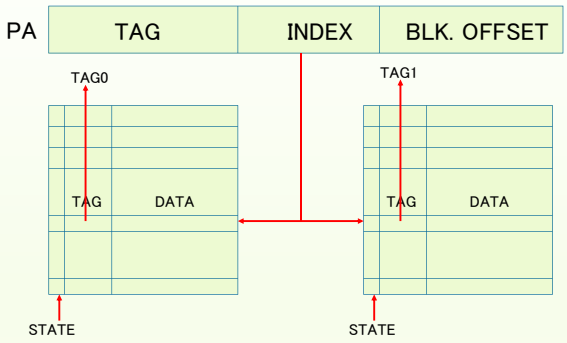
Addressing a cache

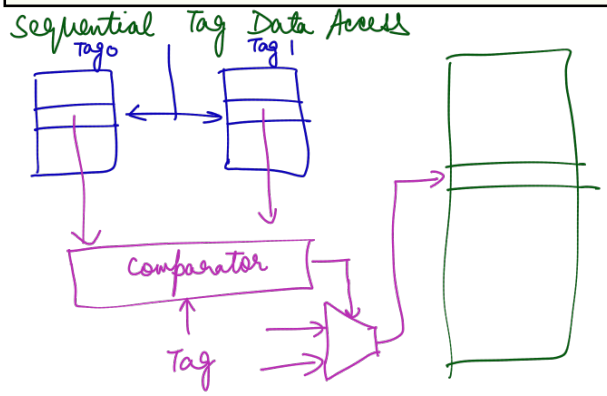
- An example
  - PA is 32 bits
  - Cache line is 64 bytes: block offset is 6 bits
  - Number of cache lines is 512: index is 9 bits
  - So tag is the remaining bits: 17 bits
  - Total size of the cache is 512*64 bytes i.e. 32 KB
  - Each cache line contains the 64 byte data, 17-bit tag, one valid/invalid bit, and several state bits (such as shared, dirty etc.)
  - Since both the tag and the index are derived from the PA this is called a physically indexed physically tagged cache

Set associative cache

- The example assumes one cache line per index
  - Called a direct-mapped cache
  - A different access to a line evicts the resident cache line
  - This is either a capacity or a conflict miss
- Conflict misses can be reduced by providing multiple lines per index
- Access to an index returns a set of cache lines
  - For an n-way set associative cache there are n lines per set
- Carry out multiple tag comparisons in parallel to see if any one in the set hits

2-way set associative





more # of comparators

Way ① :→ mux on

look the data + tag of all ways in parallel.
Feed them to a mux and select correct data by output of tag comparison

mux on critical path of telling whether hit/miss

Parallel Tag Data Access

Set associative cache

- When you need to evict a line in a particular set you run a replacement policy
  - LRU is a good choice: keeps the most frequently and most recently used lines (favors temporal locality)
  - Thus you reduce the number of conflict misses
- Two extremes of set size: direct-mapped (1-way) and fully associative (all lines are in a single set)
  - Example: 32 KB cache, 2-way set associative, line size of 64 bytes: number of indices or number of sets=32*1024/(2*64)=256 and hence index is 8 bits wide
  - Example: Same size and line size, but fully associative: number of sets is 1, within the set there are 32*1024/64 or 512 lines; you need 512 tag comparisons for each access

Cache hierarchy

- Ideally want to hold everything in a fast cache
  - Never want to go to the memory
- But, with increasing size the access time increases
- A large cache will slow down every access
- So, put increasingly bigger and slower caches between the processor and the memory
- Keep the most recently used data in the nearest cache: register file (RF)
- Next level of cache: level 1 or L1 (same speed or slightly slower than RF, but much bigger)
- Then L2: way bigger than L1 and much slower

$\frac{8 \times 2^{10}}{2^6} = \frac{8 \times 2^4}{4} = 2^5$

$128 \times 32\,b = 128 \times 4\,B = 512\,B$

Cache hierarchy

- Example: Intel Pentium 4 (Netburst) 32 bit archi
  - 128 registers accessible in 2 cycles IW=6 CW=3
  - L1 date cache: 8 KB, 4-way set associative, 64 bytes line size, accessible in 2 cycles for integer loads L/S →2
  - L2 cache: 256 KB, 8-way set associative, 128 bytes line size, accessible in 7 cycles
- Example: Intel Itanium 2 (code name Madison)
  - 128 registers accessible in 1 cycle
  - L1 instruction and data caches: each 16 KB, 4-way set associative, 64 bytes line size, accessible in 1 cycle
  - Unified L2 cache: 256 KB, 8-way set associative, 128 bytes line size, accessible in 5 cycles
  - Unified L3 cache: 6 MB, 24-way set associative, 128 bytes line size, accessible in 14 cycles

21

12 read ports 3 write ports

Local store BW = 2 (just every time) 2 reg

States of a cache line

- The life of a cache line starts off in invalid state (I)
- An access to that line takes a cache miss and fetches the line from main memory
- If it was a read miss the line is filled in shared state (S) [we will discuss it later; for now just assume that this is equivalent to a valid state]
- In case of a store miss the line is filled in modified state (M); instruction cache lines do not normally enter the M state (no store to Icache)
- The eviction of a line in M state must write the line back to the memory (this is called a writeback cache); otherwise the effect of the store would be lost

MAINAK CS422 22

$L1 \subseteq L2$

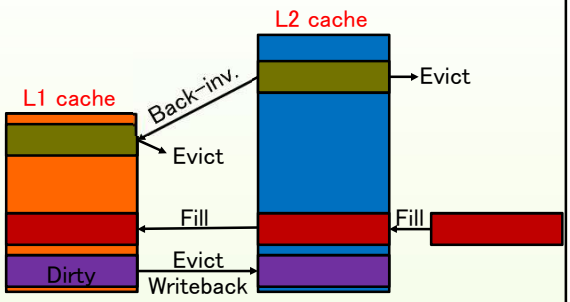
Inclusion policy

- Normally the contents of level n cache (exclude the register file) is a subset of the contents of level n+1 cache
  - Eviction of a line from L2 must ask L1 caches (both instruction and data) to invalidate that line if present
  - A store miss fills the cache line in both L2 and L1 caches, but the store really happens in L1 data cache; so L2 cache does not have the most up-to-date copy of the line
  - Eviction of an L1 line in M state writes back the line to L2
  - Eviction of an L2 line in exclusive (E) state first asks the L1 data cache to send the most up-to-date copy (if any), then it writes the line back to the next higher level (L3 or main memory)
  - Inclusion simplifies the on-chip coherence protocol (more later)

MAINAK CS422 23

if some block brought normally to L1 and L2 (in Valid state) and store in L1 then too send signal to L2 to make it E state.

Inclusion policy



$L1 \subseteq L2$

MAINAK CS422 24

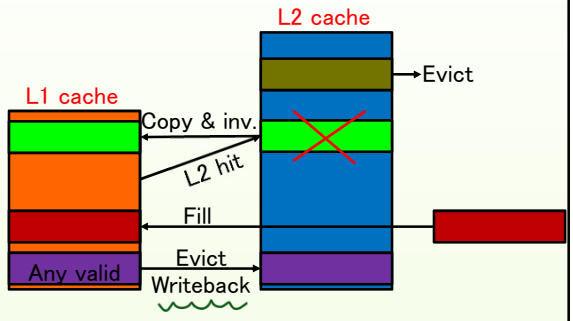
⊕ less L1-L2 traffic
⊕ on-chip cache coherence protocol
⊖ Inclusion victim
⊖ less $|L1 \cup L2|$

$|L_1| : |L_2| = ?$

⊕ More range of blocks

⊖ No LRU policy in L2 possible

Exclusion policy

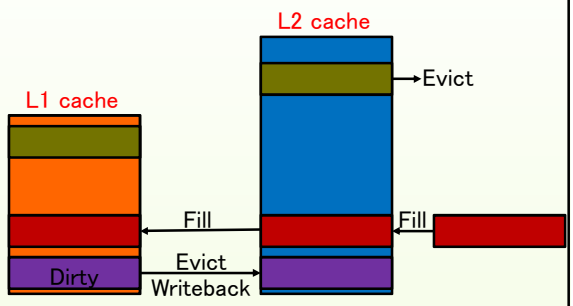


$L1 \cap L2 = \phi$

Exclusive hierarchy

- Drawbacks of inclusion
  - Back-invalidations and replication across levels of hierarchy
- Trade-off between inclusion and exclusion
  - Effective capacity vs. bandwidth
- Bandwidth-efficient exclusion
  - Bypass potential dead blocks
- Middle-ground
  - Non-inclusive/non-exclusive (NINE)

NINE policy



$L1 \not\subseteq L2,\ L1 \cap L2 \neq \phi$

The first instruction

- Accessing the first instruction
  - Take the starting PC
  - Access iTLB with the VPN extracted from PC: iTLB miss
  - Invoke iTLB miss handler
  - Calculate PTE address
  - If PTEs are cached in L1 data and L2 caches, look them up with PTE address: you will miss there also
  - Access page table in main memory: PTE is invalid: page fault
  - Invoke page fault handler
  - Allocate page frame, read page from disk, update PTE, load PTE in iTLB, restart fetch

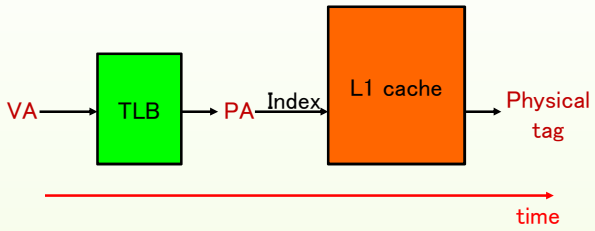
The first instruction

- Now you have the physical address
  - Access Icache: miss
  - Send refill request to outer levels: you miss everywhere
  - Send request to memory controller (north bridge)
  - Access main memory
  - Read cache line
  - Refill all levels of cache as the cache line returns to the processor
  - Extract the appropriate instruction from the cache line with the block offset
- This is the longest possible latency in an instruction/data access

TLB access

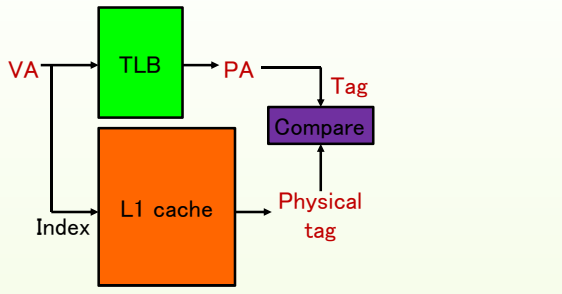
- For every cache access (instruction or data) you need to access the TLB first
- Puts the TLB in the critical path
- Want to start indexing into cache and read the tags while TLB lookup takes place
  - Virtually indexed physically tagged cache
  - Extract index from the VA, start reading tag while looking up TLB
  - Once the PA is available do tag comparison
  - Overlaps TLB reading and tag reading

TLB access: PIPT



Total time = TLB access time + L1 cache access time

TLB access: VIPT

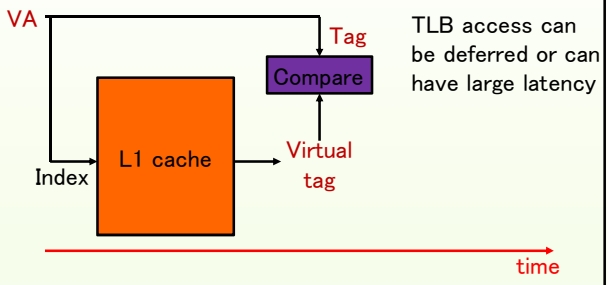


Total time = max(TLB access time, L1 cache access time)

can have bigger L1 TLB

needs ASID bits and security bits (r,w,x) in L1 cache.

TLB access: VIVT



TLB access can be deferred or can have large latency

Total time = L1 cache access time

even bigger TLB (will now be clubbed with L2 cache)

Handing synonyms

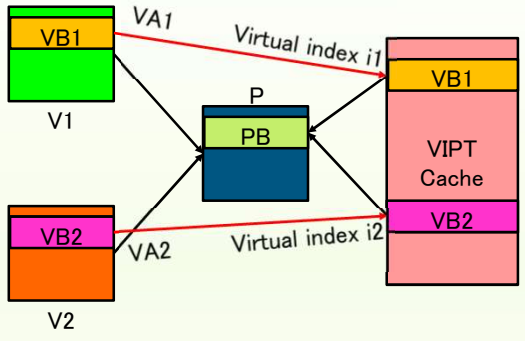
- A correctness problem arises in VIVT and VIPT caches
  - Consider two virtual pages V1 and V2 that map to the same physical page P
    - Such mapping is often used to share data/code between different processes
    - V1 and V2 are called synonyms because they refer to the same physical data
    - Now consider a particular cache block VB1 in V1 which gets modified through some access; suppose this block resides at virtual index i1 in the cache
    - Later another access touches the corresponding block VB2 in V2 through a different virtual index i2 and gets stale data

4KB pages
32 KB L1
64 B
8 way

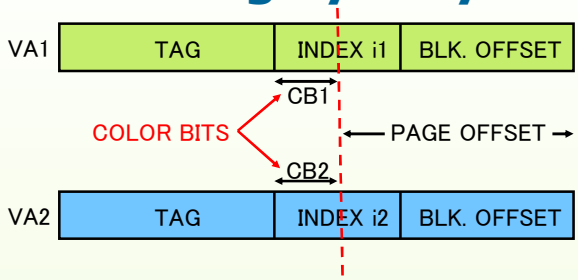
|PO| = 12 bits
|BO| = 6 bits
64 sets

assoc = 32 KB / (64 × 64B) = 8

Handing synonyms

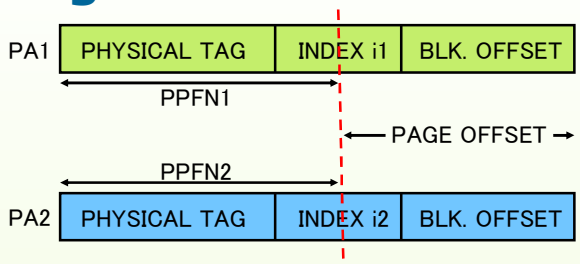


Handling synonyms



- Page offset part is same in both addresses
- i1 and i2 differ only in color bits
- Could use page coloring with help from OS to make i1=i2

Tag size in VIPT cache



- Consider two different physical blocks (PA1≠ PA2)
- But virtual indices are same (i1=i2); they map to same set
- It could be possible that physical tag portions are same
- VIPT cache tags must include the entire PPFN

Memory op latency

- L1 hit: ~1 ns
- L2 hit: ~5 ns
- L3 hit: ~10–15 ns
- Main memory: ~70 ns DRAM access time + bus transfer etc. = ~110–120 ns
- If a load misses in all caches it will eventually come to the head of the ROB and block instruction retirement (in-order retirement is a must)
- Gradually, the pipeline backs up, processor runs out of resources such as ROB entries and physical registers
- Ultimately, the fetcher stalls: severely limits ILP

MLP

- Need memory-level parallelism (MLP)
  - Simply speaking, need to mutually overlap several memory operations
- Step 1: Non-blocking cache
  - Allow multiple outstanding cache misses
  - Mutually overlap multiple cache misses
  - Supported by all microprocessors today (Alpha 21364 supported 16 outstanding cache misses)
- Step 2: Out-of-order load issue
  - Issue loads out of program order (address is not known at the time of issue)
  - How do you know the load didn't issue before a store to the same address? Issuing stores must check for this memory-order violation

Out-of-order loads

sw 0(r7), r6

… /* other instructions */

lw r2, 80(r20)

- Assume that the load issues before the store because r20 gets ready before r6 or r7
- The load accesses the store buffer (used for holding already executed store values before they are committed to the cache at retirement)
- If it misses in the store buffer it looks up the caches and, say, gets the value somewhere
- After several cycles the store issues and it turns out that 0(r7)==80(r20) or they overlap; now what?

Load/store ordering

- Out-of-order load issue relies on speculative memory disambiguation
  - Assumes that there will be no conflicting store
  - If the speculation is correct, you have issued the load much earlier and you have allowed the dependents to also execute much earlier
  - If there is a conflicting store, you have to squash the load and all the dependents that have consumed the load value and re-execute them systematically
  - Turns out that the speculation is correct most of the time
  - To further minimize the load squash, microprocessors use simple memory dependence predictors (predicts if a load is going to conflict with a pending store based on that load's past behavior)

MLP and memory wall

- Today microprocessors try to hide cache misses by initiating early prefetches:
  - Hardware prefetchers try to predict next several load addresses and initiate cache line prefetch if they are not already in the cache
  - All processors today support prefetch instructions; so you can specify in your program when to prefetch what: this gives much better control compared to a hardware prefetcher
- Load value prediction?
- Even after doing all these memory latency remains the biggest performance bottleneck: memory wall

L1: $r \times p$ → miss rate of L1 × miss penalty of L1

L1+L2: $r h p' + r(1-h)p \leq rp$ (h = hit rate of L2)

L2 local miss rate = # miss in L2 / # access to L2

L2 global miss rate = # miss in L2 / # total mem access

Cache performance

- Three parameters govern the average memory access latency
  - (1 – Miss rate) × Hit time + Miss rate × Miss penalty
  - Overall execution time = Busy time + memory stall time
  - Busy time is determined mostly by ILP exploited by the processor
  - Cache misses become more expensive as processors get faster (why?) in same time, more cycles are lost
  - How do you calculate percentage of execution time lost on memory stalls in out-of-order processors?

Stall: cycles in which the processor could not commit anything

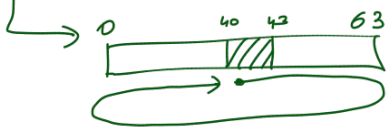
→ memory: Head of ROB is a cache miss

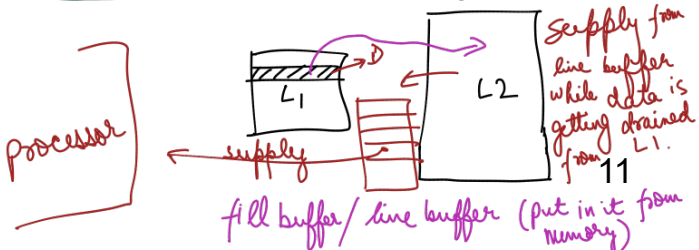
→ arithmetic stall: incl in busy time

Miss penalty

- Which instruction is more affected: load or store?
- How to reduce miss penalty?
  - Multi-level caches: equation for memory access latency?
    - Local miss rate and global miss rate
    - Misses per instruction → good for spatial locality
    - Inclusion/exclusion? larger block size in L2 but block size rem same due to false sharing
    - Large or fast L2? Block size and associativity? Comparison with L1? → larger to reduce conflicts
  - Critical word first: fetch the missed word first
  - Early restart: fetch in order, but start as soon as possible
    - Problems? (next access) start as soon as byte 40 comes
  - Prioritize loads: queue
    - Put stores in write buffers for write-through cache?
    - Fill before spill in writeback cache

not a golden rule





store — frees up resources for loads (of concern)

misses
a[0]
a[16]
a[32]
⋮

```
for (i=0; i<N; i++)
    x = a[i];
```

compiler can bring these accesses together

Miss penalty

- How to reduce miss penalty?
  - Write merges — BW optimisation
    - Send writes to the same cache line together (why useful?)
    - Benefit depends on width of data bus to next level
    - Same technique can be used for gathering uncached writes (what is this?) — to IO devices
  - Victim cache
    - A small fully associative buffer containing last few evicted cache lines
    - Victim cache hit swaps replaced cache line with it
    - Increases effective associativity of the cache
    - AMD Athlon has 8 victim buffers

32KB
inclusive
16 blocks
L1
Victim
L2
exclusive

Miss penalty

- Non-blocking caches
  - Also known as lockup-free caches
  - Make the loads non-blocking
  - Easy to make stores non-blocking (why?)
  - Allow hit under miss and miss under miss
  - Need some table to remember outstanding requests (one per line): MSHR (miss status holding registers) or MHT (miss handling table)
  - Replies snoop load queue to wake up matching instructions; they can now retry
  - Why MSHR is needed, given that load queue already has the outstanding instruction?
  - Size of MSHR determines degree of exposed MLP
  - Can compiler help in exposing more MLP?
  - Problems of increasing the size of MHT? (associative search) 24-32

L1
L2
MHT
L1 MHT ≤ L2 MHT
MHT

② Loads in the wrong path, so when a miss comes back, instr are gone; (unsolicited load)

MSHR needed to maintain exactly one req per cache line (otherwise, snoop entire load queue for each load issue)

Miss penalty

- Prefetching
  - Two types: hardware prefetching (pattern predictor assisted) and software prefetching (compiler assisted)
  - Three important questions
    - To dedicated prefetch buffer or cache?
    - What to prefetch? (danger of cache pollution)
    - When to prefetch? (danger of pre-mature eviction)
  - Can be applied to both instruction and data
  - Non-binding and binding data prefetch
  - Examples include single-stream stride prefetcher
    - A little more complicated: multi-stream stride prefetcher
  - Hardware prefetching is harder than software prefetching
  - Software prefetching needs support from ISA: prefetch instructions (must be non-blocking)
  - What about prefetches on wrong path? TLB misses?

just fill it in the cache

seen today

(not seen today)

Binding prefetch ~~lw $4, 0($10)~~ (correctness issues) remove this instr from IQ and put data in $4 directly.

A, A+2, A+4, ...

A, A+2, B, A+4, B+3, B+6, A+6, . . .

16 to 32 streams

Miss penalty

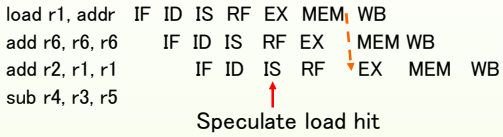
- Cold store misses
  - Initialization often writes to entire cache line
  - Write hint instruction in Alpha 21264
  - Needs compiler assistance

Miss rate

- Three types of misses: compulsory or cold, capacity, conflict (3 C's)
  - How to categorize these for a certain cache organization (i.e. for a given ABC)?
  - Victim cache targets what kind of miss?
- How to reduce miss rate?
  - Larger cache blocks for a fixed size cache
    - What is the trade-off?
    - Latency/bandwidth effects?
  - Larger caches (capacity)
  - Higher associativity (conflict)
  - Compiler optimizations (code layout and target alignment in icache, loop interchange, loop splitting, tiling for data)
  - Better replacement algorithms

Hit time

- What is the impact of hit time?
  - Load hit speculation and associated problems

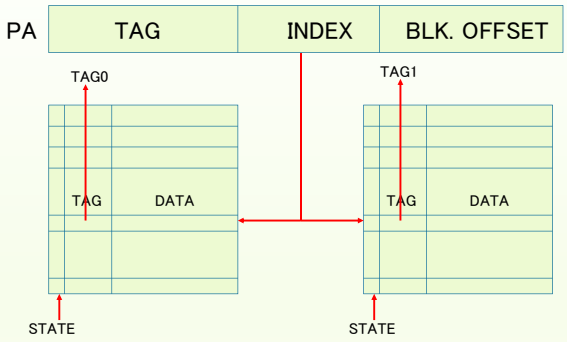


  - As load latency increases (i.e., equivalent to more MEM stages), deeper speculation is needed
    - A higher number of in-flight dependent instructions may have to be nullified on a wrong hit speculation

Hit time

- What is the impact of hit time?
  - Load hit speculation and associated problems
- How to reduce hit time?
  - Small and simple (e.g. direct mapped) cache
  - Virtual caches (protection/security/correctness issues?)
    - Virtually indexed virtually tagged (Alpha 21264 icache)
    - Virtually indexed physically tagged
    - Page coloring
  - Pipelined cache access
    - Access latency remains unchanged, but throughput improves
  - Way prediction (reduces power consumption also: 4300)
    - Alpha 21264: line and way prediction
  - Pseudo-associativity: fast and slow hits

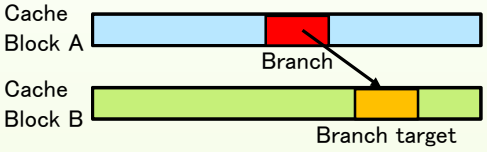
2-way pseudo SA



Access MRU way first; on a miss, access the other way

Trace cache

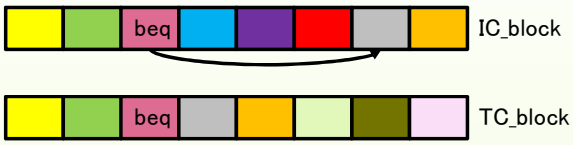
- Need to supply "good" instructions quickly
  - Often predicted target of a branch falls in a different cache block and may be in the middle of the block
  - Wastes trailing part of the cache block having the branch
  - Wastes header part of the target cache block



  - Worst part: cannot fetch from the target in the same cycle even if the prediction is correct (multi-ported icache is very hard to design at high frequency)

Trace cache

- Need to supply "good" instructions quickly
  - *Build traces dynamically* : Branch prediction validation becomes part of the hit test



  - Introduced in Pentium 4: it does not have an L1 instruction cache, only has a trace cache
  - While building the trace lines, Pentium 4 also translates IA32 instructions to micro-ops (off the critical path)
  - Downside of trace cache: may contain duplicate partial traces (why?)

Multi-ported cache

- Want support for multiple accesses to the cache in the same cycle
  - Instruction caches are normally not multi-ported
    - How can you fetch multiple instructions every cycle?
    - Bank interleaved caches
    - To the external world looks like a multi-ported cache, but memory cells are not really multi-ported
  - Do we need multiple data cache accesses per cycle?
    - Remember that memory is the bottleneck
    - Good to have better data cache throughput

I/O and coherency

- Does I/O go through cache or directly go to memory?
  - I/O through cache: loads from I/O devices are cached and stores to I/O devices are cached
    - These are respectively OS kernel's input and output buffer spaces
    - Reads from I/O devices are first stored in kernel's input buffer space by the system call handlers and then copied to user space
    - Writes to I/O devices are first copied from user space to kernel's output buffer space and later flushed to the I/O device space
  - I/O directly to memory: loads from I/O devices are deposited to memory via direct memory access (DMA) without caching and stores to I/O devices pick up data directly from memory without looking up cache
    - After the DMA, system call handler copies the input buffer contents into user space and these are cached

I/O and coherency

- Does I/O go through cache or directly go to memory?
  - I/O through cache poses no correctness issues, but may hurt performance (why?)
    - Correct because CPU sees the latest data read from I/O devices through caches (CPU accesses the caches first)
    - Performance problem arises from cache pollution due to I/O data which may not have much reuse
  - Direct I/O to memory may leave stale data in cache: problem of cache coherence
    - Consider two consecutive I/O read calls
    - At the end of the first read, the input buffer would be cached due to the copy operation from input buffer to user space executed by the system call handler
    - Second I/O read deposits data from I/O device to memory via DMA and then the copy operation executed by the system call handler sees cache hits to the stale input buffer contents

I/O and coherency

- Does I/O go through cache or directly go to memory?
  - Direct I/O to memory may leave stale data in cache: problem of cache coherence
    - Consider an I/O write call
    - System call handler copies user space data into OS kernel's output buffer space and these are cached
    - A DMA operation copies from kernel's output buffer into the I/O device ignoring the latest output buffer contents in the cache

I/O and coherency

- Does I/O go through cache or directly go to memory?
  - Direct I/O to memory may leave stale data in cache: problem of cache coherence
  - Couple of solutions for direct memory I/O
    - Software solution: syscall handler must ensure that cache lines belonging to the input buffer space are flushed to memory before starting I/O (processors come with privileged selective or full cache flush instructions; x86 ISA has a cache line flush instruction: *clflush addr*)
    - Hardware solution: hardware must check if some I/O device is reading/writing to a memory line that is cached; if yes, the cache line must be invalidated in the entire cache hierarchy and written back to memory in case it is dirty

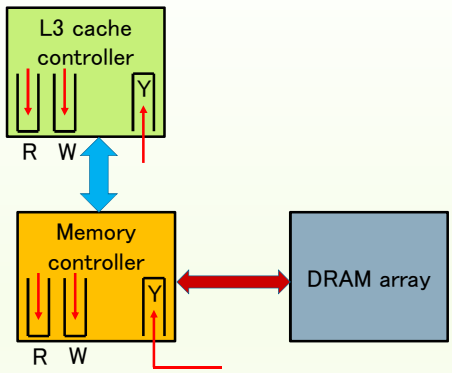
Segmented paging

- Alpha 21264 (segmented paging)
  - 8 KB page size: 13-bit page offset
  - Each PTE is 8 bytes wide (30-bit VPN requires 8 GB page table size)
  - Cannot hold the entire page table in memory: must swap in and out page table sections
  - Simple solution is to treat the page tables as normal memory and page them also
  - Alpha 21264 provides three levels of page tables
  - PTBR holds the base address of L1 page table
  - Each level can access 1024 PTEs (one page)
  - Remaining 21 bits of VA are used to identify the segment (supports three segments)

Main memory

- Outermost level of cache talks to main memory on a cache miss (I/O may have a separate path)
  - The interface is normally through a *memory controller (MC)*
  - Memory controller is connected to the *bus interface unit* which in turn connects the system bus (aka front-side bus) to the outermost cache controller (CC)
  - The cache controller usually puts a request in a buffer and the BIU grants the bus to that request at some point
  - BIU usually carries out an arbitration algorithm among various types of buffers (connecting MC to CC)
  - The request ultimately gets queued into the MC input queue
  - MC decodes the request and takes appropriate actions

Main memory



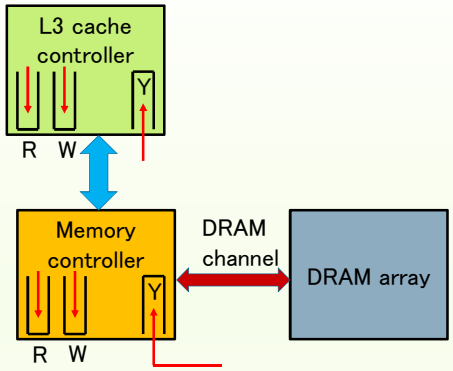
Main memory

- Steps involved in serving a cache miss from the outermost level (beyond miss detection)
  - Queue miss request in BIU buffer
    - How many request types (and hence how many different logical queues)?
  - BIU schedules the request and switches address, control, data (if needed) according to bus protocol
  - Request is fielded at the other end by MC and is put in a queue
  - MC picks requests out of this queue (normally in-order), decodes request type and address, and sends it to DRAM (dynamic random access memory)
  - DRAM access involves decoding row, decoding column, and reading out data (lumped together as access time)
  - Data reply is fielded by MC (DRAM bus connecting MC) and put in a BIU queue for scheduling

Main memory

- Latency and bandwidth
  - Understanding the trade-off
  - Compute-bound and memory-bound applications
  - Latency-bound applications
  - Bandwidth-bound applications
  - Easy to improve bandwidth: wide memory and bus, interleaving, banking, smart scheduling, …
  - Bank conflict and bank controllers
  - How many banks to support (relationship with access latency)? High density DRAM integration trend points to small number of banks
  - How to improve latency?
    - Architectural techniques: mostly revolves around "hiding" techniques
    - Lower-level techniques: deals with DRAM technology

DRAM technology



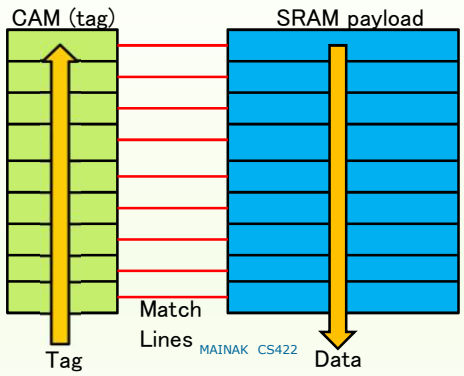
DRAM technology

- DRAM access
  - Row access strobe (RAS)
  - Column access strobe (CAS)
  - Why RAS and CAS are multiplexed?
  - Sense amplifiers
  - Refresh mechanism
  - DIMM (Dual inline memory module) organization
- DRAM cell layout
  - Wordline drive
  - Bitline charge/discharge
  - Access stack and implications of multi-porting

SRAM technology

- Used in all on-chip storage
  - Register file, cache, any table
  - DRAM emphasizes cost and capacity: needs high density
  - SRAM is more concerned with speed
  - SRAM never multiplexes RAS and CAS; in fact in most cases the entire row is read out through sense amplifiers
- Little digression
  - Content addressable memory (CAM): counterpart of RAM
  - Read all rows and apply some function on contents of all rows (most common is comparison against some data)
  - Match lines from rows usually drive the wordlines of a connected RAM; example: TLB tag CAM, issue queue

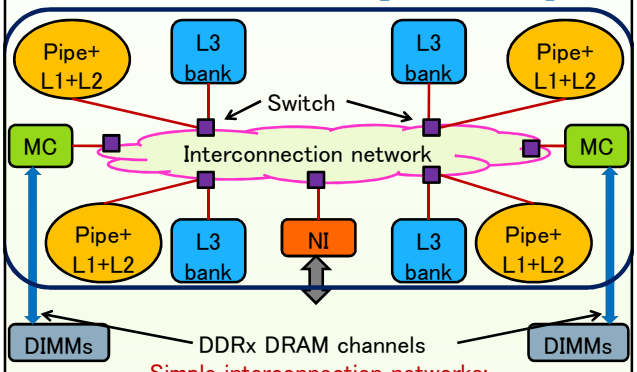
CAM



DRAM enhancements

- Simple architectural enhancements
  - Fast page mode: exploit multiple accesses to row buffers (also called DRAM pages) without intervening RAS
  - Smart MC scheduler for clustering open page accesses
  - Small embedded SRAMs to hold recently accessed pages
  - Synchronous interface to MC: leads to the concept of SDRAM (synchronous DRAM); PC100, PC133, PC150
    - Bandwidth? (assume 64-bit DIMM interface)
  - Double data rate (DDR) SDRAM: exploit both edges of clock; PC1600, PC2100, PC2400 at 2.5 V
  - DDR2 SDRAM: higher clock rate (266 MHz, 333 MHz, 400 MHz), better integration, lower voltage/power consumption (1.8 V)
  - DDR3 SDRAM: 533 MHz, 666 MHz, 800 MHz at 1.5 V

Floor of today's chips



Simple interconnection networks:
Linear bus, ring, 2D mesh (usually bidirectional)

Cache/TLB parameters

- How to infer the cache and TLB parameters of a machine
  - Often useful for program optimization
  - Three possibilities
    - Generic: reverse-engineering code snippets
      - Small programs that stress certain parts of the memory hierarchy (due to Saavedra-Barrera, PhD dissertation, 1992)
      - Often need to have some idea about the cache/TLB hierarchy to interpret the results correctly
    - OS specific: dmidecode for linux
      - Dumps the DMI (desktop management interface) table in human readable format
      - Also known as SMBIOS (system management BIOS) table
    - Hardware specific: cpuid instruction in x86
      - EAX=0x2 OR EAX=0x4 and ECX=0, 1, ⋯

Cache/TLB parameters

- Reverse-engineering cache/TLB parameters using small programs
  - Inferring cache capacity: write a loop that repeatedly goes over an array and measure total time to execute the loop; repeat with gradually increasing array size
    - A prominent jump will be seen in the measured time when the array size exceeds cache capacity
  - Inferring cache block size and associativity: write a loop that repeatedly goes over an array and measure total time; repeat with gradually increasing access stride
    - Fix array size that overflows cache capacity
    - Measured time will gradually increase with increasing stride; attains maximum at a certain stride value giving B
    - Measured time drops sharply when the accessed array elements fit in a cache set revealing associativity

Cache/TLB parameters

- Reverse-engineering cache/TLB parameters

  For each S in {1, 2, 4, 8, …}

    Repeat K*S times

      Access array of N elements with stride S

- For a particular S
  - Number of elements accessed = K*S*N/S = K*N = constant
  - Let element size be E and cache block size be B bytes
  - Number of elements in a block = B/E
  - No. of elements accessed per block = B/(E*S) if E*S ≤ B
  - Number of blocks touched = (K*N)/(B/(E*S)) = K*N*E*S/B
  - Array overflows cache capacity; so, no. of misses is same as number of blocks touched (increases with S if E*S ≤ B)
  - Number of hits per block = B/(E*S) – 1 (decreases with S)
  - Number of hits is zero when S = B/E (loop time is maximum)

73

Cache/TLB parameters

- Reverse-engineering cache/TLB parameters

  For each S in {1, 2, 4, 8, …}

    Repeat K*S times

      Access array of N elements with stride S

- For a particular S when E*S > B
  - Number of elements accessed = K*S*N/S = K*N = constant
  - No. of blocks touched = ((K*S)*(N*E)/B)/(S*E/B) = K*N
  - Array overflows cache capacity; so, no. of misses is same as number of blocks touched (constant) until the distinct blocks fit in the cache
  - Number of hits per block = 0 (loop time remains constant)
  - No. of distinct blocks touched = ((E*N)/B)/(S*E/B) = N/S
  - Let cache capacity be C bytes and associativity A
  - Number of elements per cache way = C/(E*A)

74

Cache/TLB parameters

- Reverse-engineering cache/TLB parameters

  For each S in {1, 2, 4, 8, …}

    Repeat K*S times

      Access array of N elements with stride S

- For a particular S when E*S > B
  - No. of distinct blocks touched = ((E*N)/B)/(S*E/B) = N/S
  - Number of elements per cache way = C/(E*A)
  - Let's assume that the array size is an integral multiple of the cache capacity i.e., N*E = kC for some positive integer k
  - When S = kC/(E*A), number of distinct blocks touched = (N*E*A)/(kC) = A
  - Since the stride S = kC/(E*A) is an integral multiple of the no. of elements per cache way, all these A blocks map to the same cache set and miss once (sharp drop in loop time)

75

Cache/TLB parameters

- Typical output from reverse-engineering code