Advanced Operating Systems (263-3800-00L) Caches and TLBs

Timothy Roscoe & Andrew Baumann

Based on slides by K Elphinstone & G Heiser (UNSW)

Overview

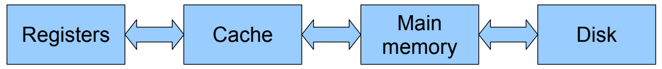
- Review of caches
  - Assume you know: direct mapping, associativity, …
- Cache addressing schemes
- Synonyms and Homonyms
- Cache management in Operating Systems
- Translation Lookaside Buffers
  - Coverage
- ARM cache/MMU architecture

Caching (review)

- Cache: something that remembers previous results
  - $\Rightarrow$can *sometimes* give the answer faster
  - E.g. memory caches, TLBs, etc.
- Work by *locality:*
  - Temporal locality: if I needed *x* recently, I'm likely to need it again soon.
  - Spatial locality: if I needed *x* recently, I'm likely to need something close by.

Memory caching (review)

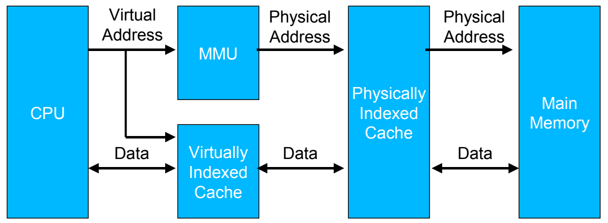
- Fast memory between registers & slow RAM
  - 1-5 vs. 10-100 cycles
- Holds recently used data and/or instructions
- Compensates for slow RAM if *hit rate* high (~90%)
- Hardware: (mostly) transparent to software
- Size: few kB – several MB
- Typically hierarchy (2-5 levels)
  - on- & off- chip



Cache organization

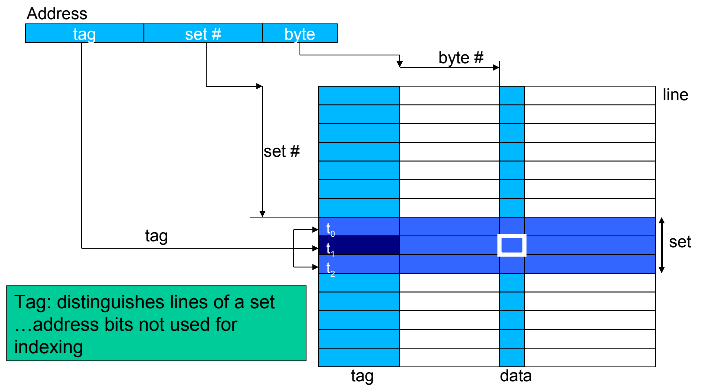
- Transfer units
  - registers ↔ L1 cache <= 1 word (1-16 bytes)
  - cache ↔ RAM (or cache) 16-32 bytes, or more
  - *Cache line*: also unit of storage allocation in cache
- Cache line associated information:
  - Valid bit
  - Modified bit
  - Tag
- Improves memory access by:
  - Absorbs reads (increases b/w, reduces latency)
  - Make writes asynchronous (hides latency)
  - Clusters reads & writes (hides latency)

Cache access



- *Virtually indexed:*
  - Lookup by virtual address
  - Concurrent with address translation
- *Physically indexed:*
  - Lookup by physical address

Cache indexing



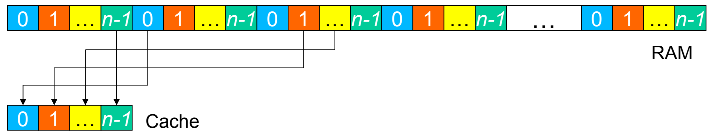
Tag: distinguishes lines of a set
…address bits not used for indexing

Cache addressing

- Address hashed ⇒ index of *line set*
- Associative looking of within set using tag
- *n* lines/set: *n-way set-associative cache*
  - *n=1:* direct mapped
  - *n=∞:* fully associative
  - *Usually n=1-4, occasionally 32-64*
- Hashing must be simple ⇒
  - Use least-significant bits of address

Cache mapping

- Different memory locations map to same cache line



  - Locations mapping to set *i* are said to be *of colour I*
  - *n*-way assoc. caches hold *n* lines of the same colour
- Cache miss types:
  - Compulsory miss: data cannot be in cache
  - Capacity miss: all cache entries in use by other data
  - Conflict miss: set mapped to address is full
    - Can't happen in fully-associative cache
  - Coherence miss: forced by hardware coherence protocol

Cache write policy

- Write back: store only updates cache
  - Memory updated when dirty line replaced (flushed)
    - Clusters writes
    - Memory inconsistent with cache
    - Hard for multiprocessors
- Write through: store updates cache & memory
    - Memory always consistent with cache
    - Increased memory/bus traffic
- Store to line not currently in cache:
  - Write allocate: allocate new cache line & store
  - No allocate: store to memory & bypass cache
- Typical combinations:
  - Write-back & write-allocate
  - Write-through & no-allocate

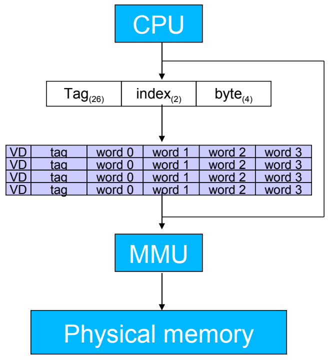
Cache addressing schemes

- So far, assumed cache *only* sees a virtual *or* physical address
- But indexing and tagging can use different addresses
  - Virtually-indexed, virtually tagged (VV)
  - Virtually-indexed, physically-tagged (VP)
  - Physically-indexed, virtually tagged (PV)
  - Physically-indexed, physically-tagged (PP)

Complex & rare

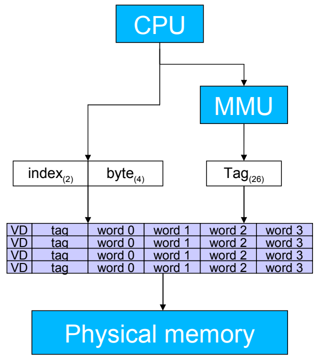
Virtually indexed, virtually tagged

- Also called:
  - virtually-addressed
- Also (misleadingly!)
  - Virtual cache
  - Virtual address cache
- Only uses virtual addresses
  - Operates concurrently with MMU
- Often used on-core


CPU
Tag(26)
index(2)
byte(4)
VD tag word 0 word 1 word 2 word 3
VD tag word 0 word 1 word 2 word 3
VD tag word 0 word 1 word 2 word 3
VD tag word 0 word 1 word 2 word 3
MMU
Physical memory

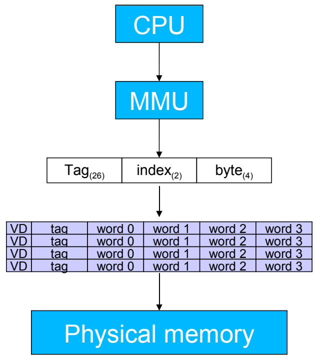
Virtually indexed, physically tagged

- Virtual addr ⇒line
- Physical addr ⇒tag
- Address translation required to retrieve data
- Index concurrently with MMU
- Use MMU output for check
- Typically used on-core



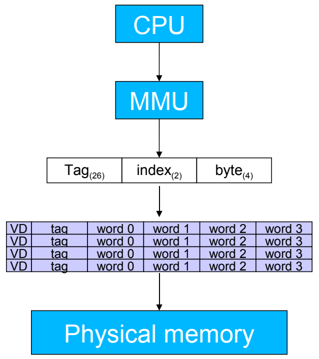
Physically indexed, physically tagged

- Only uses physical addresses
- Translation must complete before cache access can start
- Typically used off-core
- Note: page offset invariant under virtual address translation
  - Index bits ⊆ offset
  - Cache accessed without translation
  - Can be used on-core



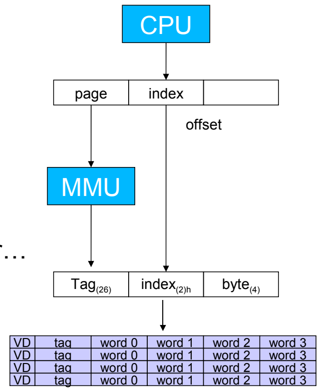
Physically indexed, physically tagged

- Only uses physical addresses
- Translation must complete before cache access can start
- Typically used off-core
- Note: page offset invariant under virtual address translation
  - Index bits ⊆ offset
  - Cache accessed without translation
  - Can be used on-core



Page offsets

See page translation later…



Cache issues

- Caches are hardware, transparent to software
- So, why worry about them in the OS?
- Well…
  - Performance
  - Synonyms
  - Homonyms
- And pretty much essential for <u>multiprocessors</u>
  - Can't really scale without them
  - A later lecture will cover MP/OS cache issues

Cache performance really matters.

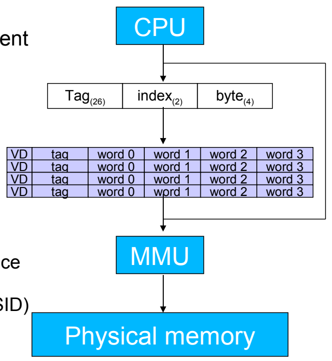
Access	Cycles	Normalized to L1	Per-hop cost
L1 cache	2	1	
L2 cache	15	7.5	
L3 cache	75	37.5	
Other L1/2/3	130	65	
Memory	~300	~150	
1-hop cache	190	95	60
2-hop cache	260	130	70

Hardware:

32-core AMD "Barcelona", 2.8GHz.

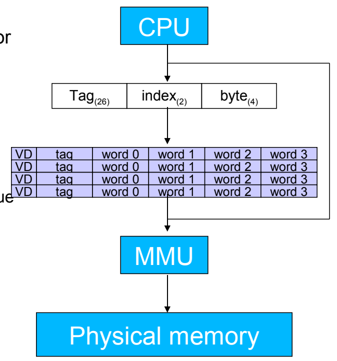
Virtually-indexed cache issues

- **Homonyms**: same names for different data
- VA used for indexing is context dependent
  - Same VA refers to different PAs
  - Tag does not uniquely identify data
  - Wrong data is accessed!
  - OS must prevent this!
- Homonym prevention:
  - Flush cache on context switch
  - Force non-overlapping address-space layout
  - ***Tag*** VA with ***address-space ID*** (ASID)
  - Use physical tags


CPU
Tag(26)
index(2)
byte(4)
VD tag word 0 word 1 word 2 word 3
VD tag word 0 word 1 word 2 word 3
VD tag word 0 word 1 word 2 word 3
VD tag word 0 word 1 word 2 word 3
MMU
Physical memory

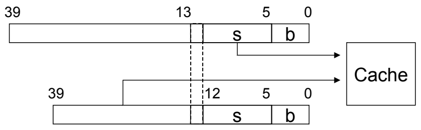
Virtually-indexed cache issues

- **Synonyms** (aliases): multiple names for same data
- Several VA map to the same PA
  - Frames shared between processes
  - Multiple mappings of frame within AS
- May access stale data:
  - Same data cached in several lines
  - On write, one synonym updated
  - Read on other synonym returns old value
  - Physical tags don't help!
  - ASIDs don't help!
- Are synonyms a problem?
  - Depends on page and cache size
  - No problem for r/o data
  - Or i-caches


CPU
Tag(26)
index(2)
byte(4)
VD tag word 0 word 1 word 2 word 3
VD tag word 0 word 1 word 2 word 3
VD tag word 0 word 1 word 2 word 3
VD tag word 0 word 1 word 2 word 3
MMU
Physical memory

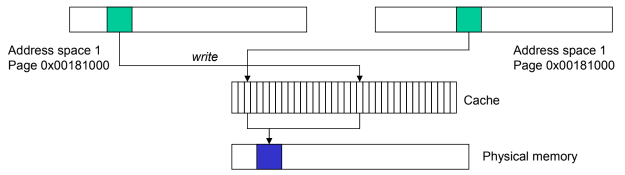
Example: MIPS R4x00 synonyms

- ASID-tagged, on-chip L1 VP cache
  - 16kB cache, 32B lines, 2-way set associative
  - 4kB (base) page size
  - Set size = 16kB/2 = 8kB > page size
  - Overlap of tag & index bits, but from different addresses!



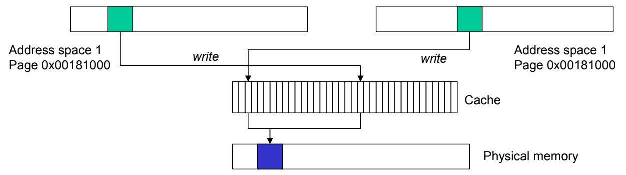
- Remember, location of data in cache determined by index
  - Tag only confirms whether it's a hit
  - Synonym problem iff $VA_{12} \neq VA'_{12}$
  - Similar issues on other processors, e.g. ARM11 (set size 16kB, page size 4kB)

Address mismatch problems: Aliasing



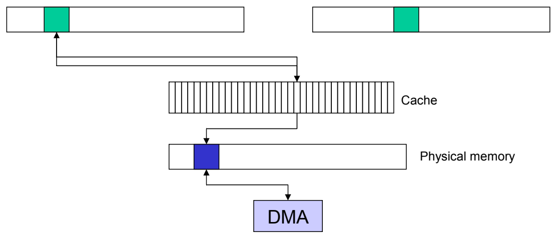
- Page aliased in different address spaces
  - AS1: $VA_{12} = 1$, AS2: $VA_{12} = 0$
- One alias gets modified
  - In a write-back cache, other alias sees stale data
  - Lost-update problem

Address mismatch problems: Remapping



- Unmap page with dirty cache line
- Re-use (remap) frame to a different page (in same or different AS)
- Write to a new page
  - Without mismatch, new write overwrites old (hits same cache line)
  - With mismatch, order can be reversed: "cache bomb"

DMA consistency problem



- DMA (normally) uses physical addresses and bypasses cache
  - CPU access inconsistent with device access
  - Need to flush cache before device write
  - Need to invalidate cache before device read

Avoiding synonym problems

- Hardware synonym detection
- Flush cache on context switch
  - Doesn't help for aliasing *within* address space
- Detect synonyms and ensure
  - All read-only, OR
  - Only one synonym mapped at a time
- Restrict VM mapping so synonyms map to same cache set
  - E.g. on R4x00, ensure that $VA_{12}=PA_{12}$

Summary: VV caches

- Fastest: don't rely on TLB for retrieving data
  - Still need TLB lookup for protection
  - Or other mechanism to provide protection
- Suffer from synonyms and homonyms
  - Requires flush on context switch
    - Makes context switches expensive
    - May even be required on kernel ⇒user switch
  - … or guarantee of no synonyms or homonyms
- Require TLB lookup for write-back!
- Used on i860, ARM7/ARM9/StrongARM/XScale
- Used for i-cache on many architectures
  - Alpha, Pentium-4, etc.

Summary: VV caches with keys

- Add *address space identifier* (ASID) part of tag
- On access compare with CPU's ASID register
- Removes homonyms, creates synonyms
  - Potentially better context switching performance
  - ASID recycling still requires a cache flush
- Doesn't solve synonym problem
  - (but that's less serious)
- Doesn't solve write-back problem!

VP caches

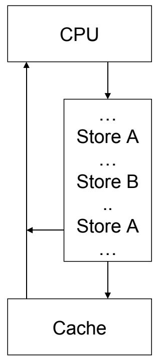
- Medium speed
  - Lookup in parallel with address space translation
  - Tag comparison after address translation
- No homonym problem
- Potential synonym problem
- Bigger tags (can't leave off set-number bits)
  - Increases area, latency, power consumption
- Used on most modern L1 data caches

PP caches

- Slowest
  - Requires result of address translation before lookup starts
- No synonym problem
- No homonym problem
- Easy to manage
- If small or highly associative (all index bits come from page offset), indexing can be in parallel with address translation
  - Potentially useful for L1 cache, e.g. Itanium
- Cache can use *bus snooping* to receive/supply DMA data
- Usable as off-chip cache with any architecture

Write buffers

- Store operations take long to complete
  - E.g. if cache line must be read or allocated
- Can avoid stalling CPU by buffering writes
- ***Write buffer*** is FIFO queue of incomplete stores
  - also called ***store buffer*** or ***write-behind*** *buffer*
- Can also read intermediate values out of buffer
  - To service load of a value that is still in the write buffer
  - Avoids unnecessary stalls of load operations
- Implies that memory contents are temporarily stale
  - On a multiprocessor, CPUs see different order of writes
  - "weak store order", to be revisiting in SMP context!

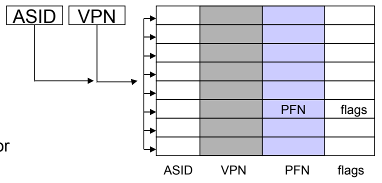


Summary

- The OS has to *manage caches* if it is to provide:
  - Correctness
  - Performance
- Interactions between caches and memory translation are complex and subtle
- OSes typically try to hide these from the user

Translations Lookaside Buffer (TLB)

- TLB is a (VV) cache for page table entries
- TLB can be
  - Hardware loaded, transparent to OS, or
  - Software loaded, maintained by OS
- TLB can be:
  - Split, instruction and data TLBs, or
  - Unified
- Modern, high-performance architectures use a hierarchy of TLBs:
  - Top-level TLB is hardware-loaded from lower levels
  - Transparent to OS



TLB issues: Associativity

- First TLB (VAX-11/780) was 2-way associative
- Most modern architectures have fully associative TLBs
- Exceptions:
  - i486, Pentium, P6, … (4-way)
  - IBM RS/6000 (2-way)
- Reasons:
  - Modern architectures tend to support multiple page sizes (superpages)
    - Better utilizes TLB entries
  - TLB lookup done without knowing page's base address
  - Set-associativity loses speed advantage
  - Hence superpage TLBs are fully-associative

TLB Size (i-TLB + d-TLB)

Architecture	TLB Size	Page Size	TLB Coverage
VAX	64-256	512B	32-128kB
ia32 / x86 (typical)	32-32+64	4kB+4MB	128-128+256kB
MIPS	96-128	4kB-16MB	384kB - …
SPARC	64	8kB-4MB	512kB - …
Alpha	32-128+128	8kB-4MB	256kB - …
RS/6000	32+128	4kB	128+512kB
Power4/G5	128	4kB+16MB	512kB - …
PA-8000	96+96	4kB-64MB	
Itanium	64+96	4kB-4GB	

Not grown much in 20 years!

TLB Coverage:

- Memory sizes are increasing
- # TLB entries more-or-less constant
- Pages sizes are growing *very* slowly
  - Total RAM mapped by TLB is not changing much
  - Fraction of RAM mapped by TLB is shrinking lots
- Modern architectures have very low *TLB coverage*
- Also, many modern architectures have software-loaded TLBs
  - General increase in TLB miss penalty (handling cost)
- The TLB is becoming a performance bottleneck

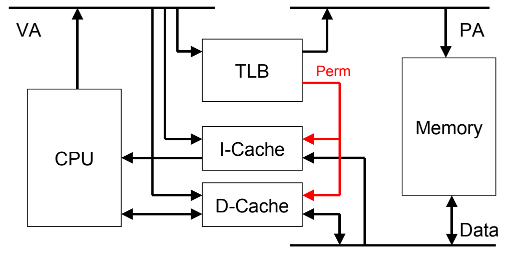
Address space usage vs. TLB coverage

- Each TLB entry maps 1 virtual page
- On TLB miss, reloaded from page table (PT), which is in memory
  - Some TLB entries needed to map page tables
  - Eg. 32-bit page table entries, 4kB pages.
  - One PT page maps 4MB
- Traditional UNIX process has 2 regions of allocated virtual address space
  - Low end: text, data, heap
  - High end: stack
  - 2-3 PT pages are sufficient to map most address spaces
- Superpages can be used to extend TLB coverage
  - But difficult to manage in the OS

Concrete example: ARM

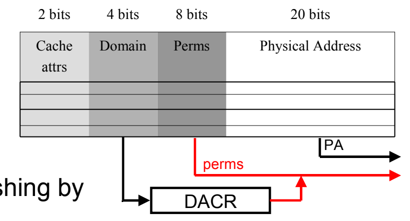
- Typical features of ARM MMU cores:
  - Virtually-indexed L1 split caches
  - No L2 cache
  - No address-space tags (ASIDs) in TLB or caches
- Warning:
  - Under some circumstances, L4 does not flush caches on a context switch.
  - Instead, uses *domain bits*

ARM cache architecture



- Virtually-indexed caches:
  - flush caches on context switch
  - direct cost: 1k-18k cycles
  - indirect cost: up to 54k cycles

ARM TLB format



- TLB has no PID tag
  - Must *flush* TLB on context switches
  - Direct cost: 1 cycle
  - Indirect cost: 3k cycles
  - TLB flush ⇒cache flush
- Windows CE avoids flushing by
  - No protection!
  - Max 32 processes
- Better: use *domains*
  - Impose additional access restrictions
  - Simulate address space tags
  - Flush TLB lazily on collisions

ARM domains

- Every PTE is in a *domain*.
  - There are 16 in total.
- Each domain has a 2-bit field in the DACR specifying access
  - Access rights to many pages can be changed at once
  - Access faults ⇒trap to kernel
- Exercise: (!)
  - Work out how to use this to share page tables between processes, and avoid most cache/TLB flushes on context switch

Summary

- OS Management of the TLB is critical for
  - Correctness
  - Performance
- Hardware is diverse
  - Many processors/MMUs have unusual features
  - Effective use of these requires thought and ingenuity
  - And isn't portable
- Next week: more on page table structures