COMP9242 Advanced OS
S2/2017 W03:
Caches: What Every OS Designer Must Know
@GernotHeiser

The Memory Wall

Multicore offsets stagnant per-core performance with proliferation of cores
- Same effect on overall memory bandwidth
- Basic trend is unchanged

Caching

- Cache is fast (1–5 cycle access time) memory sitting between fast registers and slow RAM (10–100s cycles access time)
- Holds recently-used data or instructions to save memory accesses
- Matches slow RAM access time to CPU speed if high hit rate (> 90%)
- Is hardware maintained and (mostly) transparent to software
- Sizes range from few KiB to several MiB.
- Usually a hierarchy of caches (2–5 levels),
- On contemporary processors generally on-chip

Good overview of implications of caches for operating systems: [Schimmel 94]
Cache Organization

- Data transfer unit between registers and L1 cache: ≤ 1 word (1–16B)
- Cache line is transfer unit between cache and RAM (or lower cache)
  - typically 16–32 bytes, sometimes 128 bytes and more
- Line is also unit of storage allocation in cache
- Each line has associated control info:
  - valid bit
  - modified bit
  - tag
- Cache improves memory access by:
  - absorbing most reads (increases bandwidth, reduces latency)
  - making writes asynchronous (hides latency)
  - clustering reads and writes (hides latency)

Cache Access

- Virtually indexed: looked up by virtual address
  - operates concurrently with address translation
- Physically indexed: looked up by physical address
  - requires result of address translation
- Usually have hierarchy: L1 (closest to core), ... Ln (closest to RAM)
  - L1 may use virtual address, all others use physical only

Cache Indexing

- Address hashed to produce index of line set
- Associative lookup of line within set
- n lines per set: n-way set-associative cache
  - typically n = 1 … 5; some embedded processors use 32–64
  - n = 1 is called direct mapped
  - n = ∞ is called fully associative (unusual for I/D caches)
- Hashing must be simple (complex hardware is slow)
  - generally use least-significant bits of address (except L3 on recent x86)
Cache Indexing: Direct Mapped

Offset bits used to select appropriate bytes from line

Index bits used to select unique line to match

Tag used to check whether line contains requested address

Cache Indexing: 2-Way Associative

Offset bits used to select appropriate bytes from line

Index bits used to select unique set to match within

Tag checked against both lines for match

Cache Indexing: Fully Associative

Offset bits used to select appropriate bytes from line

Cache Mapping Implications

• Multiple memory locations map to the same cache line

• Locations mapping to cache set $i$ are said to be of colour $i$

• $n$-way associative cache can hold $n$ lines of the same colour

• Types of cache misses (“the four Cs”):
  – Compulsory miss: data cannot be in the cache (if infinite size)
    o first access (after flush)
  – Capacity miss: all cache entries are in use by other data
    o would not miss on infinite-size cache
  – Conflict miss: all lines of the correct colour are in use by other data
    o would not miss on fully-associative cache
  – Coherence miss: miss forced by hardware coherence protocol

Note: Lookup hardware for many tags is large and slow ⇒ does not scale

Tag compared against all lines for a match

Cache Mapping Implications

• Multiple memory locations map to the same cache line

• Locations mapping to cache set $i$ are said to be of colour $i$

• $n$-way associative cache can hold $n$ lines of the same colour

• Types of cache misses (“the four Cs”):
  – Compulsory miss: data cannot be in the cache (if infinite size)
    o first access (after flush)
  – Capacity miss: all cache entries are in use by other data
    o would not miss on infinite-size cache
  – Conflict miss: all lines of the correct colour are in use by other data
    o would not miss on fully-associative cache
  – Coherence miss: miss forced by hardware coherence protocol

Note: Lookup hardware for many tags is large and slow ⇒ does not scale
Cache Replacement Policy

- Indexing (using address) points to specific line set
- On miss (all lines of set are valid): replace existing line
- Replacement strategy must be simple (hardware!)
  - dirty bit determines whether line must be written back
  - typical policies:
    - LRU
    - pseudo-LRU
    - FIFO
    - random
    - toss clean

Cache Write Policy

- Treatment of store operations
  - write back: Stores only update cache; memory is updated once dirty line is replaced (flushed)
    - clusters writes
    - memory inconsistent with cache
    - multi-processor cache-coherency challenge
  - write through: stores update cache and memory immediately
    - memory is always consistent with cache
    - increased memory/bus traffic
- On store to a line not presently in cache (write miss):
  - write allocate: allocate a cache line and store there
    - typically requires reading line into cache first!
  - no allocate: store directly to memory, bypassing the cache
- Typical combinations:
  - write-back & write-allocate
  - write-through & no allocate

Cache Addressing Schemes

- For simplicity assumed so far that cache only sees one type of address: virtual or physical
- However, indexing and tagging can use different addresses!
- Four possible addressing schemes:
  - virtually-indexed, virtually-tagged (VV) cache
  - virtually-indexed, physically-tagged (VP) cache
  - physically-indexed, virtually-tagged (PV) cache
  - physically-indexed, physically-tagged (PP) cache
- PV caches can make sense only with unusual MMU designs
  - not considered any further

Virtually-Indexed, Virtually-Tagged Cache

- Also called virtually-addressed cache
- Various incorrect names in use:
  - virtual cache
  - virtual address cache
- Uses virtual addresses only
- Can operate concurrently with MMU
- Still needs MMU lookup for access rights
- Writeback needs PA
  - TLB lookup?
- Used for on-core L1
Virtually-Indexed, Physically-Tagged Cache

- Virtual address for accessing line (lookup)
- Physical address for tagging
- Needs complete address translation for looking up retrieving data
- Indexing concurrent with MMU
- Use MMU output for tag check
- Used for on-core L1

Physically-Indexed, Physically-Tagged Cache

- Only uses physical addresses
- Address translation result needed to begin lookup
- Only choice for Ln, n>1
- Note: page offset is invariant under address translation
  - if index offset is a subset of the offset bits, PP cache lookup doesn’t need MMU result!
  - VP=PP in this case; fast and suitable for on-core use (L1)
  - This cache has single page colour

Cache Issues

- Caches are managed by hardware transparently to software
  - OS doesn’t have to worry about them, right?
- Software-visible cache effects:
  - performance
    - homonyms:
      o same address, different data
      o can affect correctness!
    - synonyms (aliases):
      o different address, same data
      o can affect correctness!

Homonyms – same name for different data:

- Problem: VA used for indexing is context-dependent
  - same VA refers to different PAs
  - tag does not uniquely identify data!
  - wrong data may be accessed
  - an issue for most OSes
- Homonym prevention:
  - flush cache on each context switch
  - force non-overlapping address-space layout
    o single-address-space OS
  - tag VA with address-space ID (ASID)
    o makes VAs global
Virtually-Indexed Cache Issues

Synonyms – multiple names for same data:

- Several VAs map to the same PA
  - frame shared between ASs
  - frame multiply mapped within AS
- May access stale data!
  - same data cached in multiple lines
    - If aliases differ in colour
      - 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 (colours)
  - no problem for R/O data or t-caches

Example: MIPS R4000 Synonyms

- ASID-tagged, on-chip VP cache
  - 16 KiB cache, 2-way set associative, 32 B line size
  - 4 KiB (base) page size
  - size/associativity = 16/2 KiB = 8 KiB > page size (2 page colours)
    - 16 KiB / (32 B/line) = 512 lines = 256 sets \(
      \Rightarrow \text{ 8 index bits (12..5)}
    \)
    - overlap of tag bits and index bits, but from different addresses!
- Remember, index determines location of data in cache
  - tag only confirms hit
  - synonym problem iff \( VA_{12} \neq VA_{12}' \)
  - similar issues on other processors where L1 cache has multiple colours

Address Mismatch Problem: Aliasing

- Page aliased in different address spaces
  - \( AS_1: VA_{12} = 1, AS_2: VA_{12} = 0 \)
- One alias gets modified
  - in a write-back cache, other alias sees stale data
  - lost-update problem

Address Mismatch Problem: Re-Mapping

- Unmap aliased page, remaining page has a dirty cache line
- Re-use (remap) frame for a different page (in same or different AS)
- Access new page
  - without mismatch, new write will overwrite old (hits same cache line)
  - with mismatch, alias may write back after remapping: "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

You’ll have to deal with this!

Avoiding Synonym Problems

- 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
- Restrict VM mapping so synonyms map to same cache set
  - eg on R4x00: ensure VA_{12} = PA_{12} – colour memory!
- Hardware synonym detection
  - e.g. Cortex A9: store overlapping tag bits of both addresses & check
  - “physically”-addressed

Summary: VV Caches

- Fastest (don’t rely on TLB for retrieving data)
  - still need TLB lookup for protection
  - … or alternative mechanism for providing protection
  - still need TLB lookup or physical tag for writeback
- Suffer from synonyms and homonyms
  - requires flushing on context switches
    - makes context switches expensive
    - may even be required on kernel→user switch
  - … or guarantee no synonyms and homonyms
- Used on MC68040, i860, ARM7/ARM9/StrongARM/Xscale
- Used for l-caches on several other architectures (Alpha, Pentium 4)

Summary: Tagged VV Caches

- Add ASID as part of tag
  - On access, compare with CPU’s ASID register
  - Removes homonyms
  - potentially better context-switching performance
  - ASID recycling still needs flush
  - Doesn’t solve synonym problem (but that’s less severe)
  - Doesn’t solve write-back problem
Summary: VP Caches

- Medium speed
  - Lookup in parallel with address translation
  - Tag comparison after address translation
- No homonym problem
- Potential synonym problem
- Bigger tags (cannot leave off set-number bits)
  - Increases area, latency, power consumption
- Used on most contemporary architectures for L1 cache

Summary: 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 index can be in parallel with translation:
    - All index bits come from page offset
    - Combines advantages of VV and PP cache
    - Useful for on-core L1 cache (Itanium, recent x86)
- Cache can use bus snooping to receive/supply DMA data
- Usable as post-MMU cache with any architecture

For an in-depths coverage see [Wiggins 03]

Write Buffer

- Store operations can take a long time to complete
  - Eg if a cache line must be read or allocated
- Can avoid stalling the CPU by buffering writes
- Write buffer is a FIFO queue of incomplete stores
  - Also called store buffer or write-behind buffer
  - Typically between cache levels, cache and memory
- Can also read intermediate values out of buffer
  - To service load of a value that is still in 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 revisited in SMP context

Cache Hierarchy

- Hierarchy of caches to balance memory accesses:
  - Small, fast, virtually-indexed L1
  - Large, slow, physically indexed L2–L5
- Each level reduces and clusters traffic
- L1 typically split into I- and D-caches
  - "Harvard architecture"
  - Requirement of pipelining
- Other levels generally unified
- Chip multiprocessors:
  - Usually L3 shared chip-wide
  - L2 private, clustered or shared chip-wide
Sabre (Cortex A9) System Architecture

Translation Lookaside Buffer (TLB)
- TLB is a (V) cache for page-table entries
- TLB can be
  - hardware loaded, transparent to OS
  - software loaded, maintained by OS
- TLB can be:
  - split: I- and D-TLBs
  - unified

TLB Issues: Associativity
- First TLB (VAX-11/780 [Clark, Emer 85]) was 2-way associative
- Most modern architectures have fully-associative TLBs
- Exceptions:
  - Intel x86: 4-way
  - IBM RS/6000: 2-way
- Reasons:
  - modern architectures tend to support multiple page sizes
    o “superpages”
    o better utilises TLB entries
  - TLB lookup done without knowing the page’s base address
  - set associativity loses speed advantage

TLB Size (I-TLB + D-TLB)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>TLB Size inst + data</th>
<th>TLB Assoc</th>
<th>Page Size</th>
<th>Coverage (base page)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAX-11</td>
<td>64–256</td>
<td>2</td>
<td>0.5 KiB</td>
<td>32–128 KiB</td>
</tr>
<tr>
<td>ix86</td>
<td>32i + 64d</td>
<td>4</td>
<td>4 KiB + 4 MiB</td>
<td>128 KiB</td>
</tr>
<tr>
<td>MIPS</td>
<td>96–128</td>
<td>full</td>
<td>4 KiB – 16 MiB</td>
<td>384–512 KiB</td>
</tr>
<tr>
<td>SPARC</td>
<td>64</td>
<td>full</td>
<td>8 KiB – 4 MiB</td>
<td>256 KiB</td>
</tr>
<tr>
<td>Alpha</td>
<td>32–128i + 128d</td>
<td>full</td>
<td>8 KiB – 4 MiB</td>
<td>256 KiB</td>
</tr>
<tr>
<td>RS/6000</td>
<td>32i + 128d</td>
<td>2</td>
<td>4 KiB</td>
<td>256 KiB</td>
</tr>
<tr>
<td>Power-4 (G5)</td>
<td>1024</td>
<td>4</td>
<td>4 KiB</td>
<td>512 KiB</td>
</tr>
<tr>
<td>PA-8000</td>
<td>96i + 96d</td>
<td>full</td>
<td>4 KiB – 64 MiB</td>
<td>384 KiB</td>
</tr>
<tr>
<td>Itanium</td>
<td>64i + 96d</td>
<td>full</td>
<td>4 KiB – 4 GiB</td>
<td>384 KiB</td>
</tr>
<tr>
<td>ARMv7 (A9)</td>
<td>64–128</td>
<td>1–2</td>
<td>4 KiB – 16 MiB</td>
<td>256–512 KiB</td>
</tr>
<tr>
<td>x86 (Skylake)</td>
<td>L1:128i+64d; L2:1536</td>
<td>4</td>
<td>4 KiB + 2/4 MiB</td>
<td>1 MiB</td>
</tr>
</tbody>
</table>

Not much growth in 40 years!
**TLB Size (I-TLB + D-TLB)**

**TLB coverage**
- Memory sizes are increasing
- Number of TLB entries are roughly constant
- Page sizes are steady
  - 4 KiB, although larger on SPARC, Alpha
  - OS designers have trouble using superpages effectively
- Consequences:
  - total amount of RAM mapped by TLB is not changing much
  - fraction of RAM mapped by TLB is shrinking dramatically!
  - Modern architectures have very low TLB coverage!
- Also, many 64-bit RISC architectures have software-loaded TLBs
  - general increase in TLB miss handling cost
- The TLB can become a bottleneck

**Multi-Level TLBs**
- As with CPU caches, multi-level translation caches for better size/performance tradeoffs
- ARM Cortex A9:
  - F/A 32-entry I-TLB
  - F/A 32-entry D-TLB
  - 2-way 64–128 entry L2 TLB
- Intel Core i7:
  - F/A 8-entry I-TLB, holds 4KiB or 2/4MiB mappings
  - 4-way, 64-entry D-TLB, holds 4KiB mappings, separate TLB for 1-GiB
  - 8-way, 1024 entry L2 TLB, holds 4KiB or 2MiB mappings

**Intel Core i7 (Haswell) Cache Structure**

**Intel Haswell L3 Cache**

Source: Intel