Caches:
What Every OS Designer Must Know
@GernotHeiser
Copyright Notice

These slides are distributed under the Creative Commons Attribution 3.0 License

• You are free:
  • to share—to copy, distribute and transmit the work
  • to remix—to adapt the work

• under the following conditions:
  • Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:
    “Courtesy of Gernot Heiser, UNSW Sydney”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
The Memory Wall

Multicore offsets stagnant per-core performance with proliferation of cores

- Same effect on overall memory bandwidth
- Basic trend is unchanged
Use temporal & spatial locality to improve average memory latency

- Holds recently used data/instructions
- Load/fetch hits in cache ⇒ fast access
- Miss not much worse than no cache
- Key is high hit rate (>90%)

Fast: 1–3 cycles
Small: 32 KiB – 16 MiB

Slow: 10s–100s cycles
Large: GiB
Cache Organisation: Unit of Data Transfer

Line is also unit of allocation, holds data and
• valid bit
• modified (dirty) bit
• tag
• access stats (for replacement)

Reduce memory transactions:
• Reads – locality
• Writes – clustering
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 a hierarchy: L1, L2, …, LLC (last-level cache, next to RAM)
  - L1 may use virtual address, all others use physical only
Cache Indexing

- The *tag* is used to distinguish lines of a set...
- Consists of high-order bits not used for indexing
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=1–16
  - n = 1 is called direct mapped
  - 2 ≤ n ≤ ∞ is called set associative
  - n = ∞ is called fully associative
- Hashing must be simple (complex hardware is slow)
  - generally use least-significant bits of address

Many conflicts ⇒ low hit rate

Slow & power-hungry
Cache Indexing: Direct Mapped

- \(tag_{(25)}\)
- \(index_{(3)}\)
- \(offset_{(4)}\)

Offset bits used to select appropriate bytes from line

- Index bits used to select unique line
- Tag used to check whether line contains requested address
Cache Indexing: 2-Way Associative

Index bits used to select unique set to match within

Offset bits used to select appropriate bytes from line

Tag checked against both lines for match

<table>
<thead>
<tr>
<th>VD</th>
<th>Tag</th>
<th>Word 0</th>
<th>Word 1</th>
<th>Word 2</th>
<th>Word 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
<td>Word 2</td>
<td>Word 3</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
<td>Word 2</td>
<td>Word 3</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
<td>Word 2</td>
<td>Word 3</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
<td>Word 2</td>
<td>Word 3</td>
</tr>
</tbody>
</table>
Cache Indexing: Fully Associative

Tag\(_{(28)}\)  Offset\(_{(4)}\)

Offset bits used to select appropriate bytes from line

Tag compared with all lines for a match

Lookup hardware for many tags is large and slow ⇒ does not scale
Cache Associativity vs Paging

When index overlaps page number, a particular page can only reside in a specific subset of the cache!
Cache Mapping Implications

Multiple memory locations map to same cache line

A page can only reside in the part of the cache defined by its colour

If c index bits overlap page #, a page can only reside in $2^c$ of the cache

Cache is said to have $2^c$ colours

$2^c = \frac{\text{cache}\_\text{size}}{(\text{page}\_\text{size} \times \text{assoc})}$
Cache Misses

• *n-way* associative cache can hold *n lines* with the same *index* value
• More than *n lines* are competing for same index forces a miss!
• There are different types of cache misses ("the four Cs"):  
  • **Compulsory miss**: data cannot be in the cache (of infinite size)  
    • First access (after loading data into memory or cache flush)  
  • **Capacity miss**: all cache entries are in use by other data  
    • Would not miss on infinite-size cache  
  • **Conflict miss**: all lines *with the same index value* are in use by other data  
    • Would not miss on fully-associative cache  
  • **Coherence miss**: miss forced by hardware coherence protocol  
    • Covered later (multiprocessing lecture)
Cache Replacement Policy

- Indexing (using address) points to specific line set
- On miss (no match and all lines valid): *replace* existing line
  - Dirty-bit determines whether write-back needed
- Replacement strategy must be simple (hardware!)

Typical policies:
- LRU
- pseudo-LRU
- FIFO
- “random”
- toss clean

<table>
<thead>
<tr>
<th>Address</th>
<th>tag(_{26})</th>
<th>index(_{2})</th>
<th>byte(_{4})</th>
</tr>
</thead>
<tbody>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
</tr>
</tbody>
</table>
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

• So far pretended 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

Nonsensical except with weird MMU designs
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

• Usable for on-core L1
  • Rarely used these days

Permissions? Write back?

<table>
<thead>
<tr>
<th>VD</th>
<th>Tag</th>
<th>Word 0</th>
<th>Word 1</th>
<th>Word 2</th>
<th>Word 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
<td>Word 2</td>
<td>Word 3</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
<td>Word 2</td>
<td>Word 3</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
<td>Word 2</td>
<td>Word 3</td>
</tr>
</tbody>
</table>

MMU

Physical Memory
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
- Used for on-core L1

Use MMU for tag check & permissions
Physically-Indexed, Physically-Tagged Cache

- Only uses physical addresses
- Address translation result needed for lookup
- Only sensible choice for L2...LLC

Speed matters less after L1 miss

Page offset invariant under VA→PA:
- Index bits ⊆ offset bits
  ⇒ don’t need MMU for indexing!
- VP = PP in this case
  ⇒ fast, suitable for L1
- Single-colour cache!

<table>
<thead>
<tr>
<th>VD</th>
<th>Tag</th>
<th>Word 0</th>
<th>Word 1</th>
<th>Word 2</th>
<th>Word 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
<td>Word 2</td>
<td>Word 3</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
<td>Word 2</td>
<td>Word 3</td>
</tr>
<tr>
<td>VD</td>
<td>Tag</td>
<td>Word 0</td>
<td>Word 1</td>
<td>Word 2</td>
<td>Word 3</td>
</tr>
</tbody>
</table>

CPU → MMU → Physical Memory

tag\(_{26}\) index\(_{2}\) byte\(_{4}\)
Cache Issues

• Caches are managed by hardware transparently to software, so OS doesn’t have to worry about them, right? — Wrong!

• Software-visible cache effects:
  • performance
    • cache-friendly data layout
  • homonyms:
    • same address, different data
    • can affect correctness!
  • synonyms (aliases):
    • different address, same data
    • can affect correctness!

![Diagram showing cache issues]

VAS_1: A B
VAS_2: A C
PAS: A'B' C'

© Gernot Heiser 2019 – CC Attribution License
Virtually-Indexed Cache Issues

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
    • single-address-space OS
  • tag VA with address-space ID (ASID)
    • 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 or ASIDs don’t help!
• Are synonyms a problem?
  • depends on page and cache size (colours)
  • no problem for R/O data or I-caches
Example: MIPS R4x00 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 ⇒ 8 index bits (12..5)
    - overlap of tag bits and index bits, but from different addresses!

Remember, only index determines location of data!
- Tag only confirms hit
- Synonym problem iff VA_{12} ≠ VA'_{12}
- Problem of virtually-indexed cache with 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: Aliasing

- 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 replication, new write will overwrite old (hits same cache line)
  - with replication, 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
  - must flush cache before device write
  - must 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 A53: 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 I-caches on several other architectures (Alpha, Pentium 4)
- Not used on recent architectures

Historically used with shallow hierarchies to support bigger L1
Summary: ASID-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
- Not used on recent architectures
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
  • but mostly single-colour (pseudo-PP) or with HW alias prevention (Arm)
Summary: PP Caches

খSlowest
   ᵘrequires result of address translation before lookup starts
✓No synonym problem
✓No homonym problem
✓Easy to manage
✓Cache can use *bus snooping* for DMA/multicore coherency
✓Obvious choice for L2–LLC where speed matters less
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*
  - May exist at any cache level, or between cache and memory
- Can fetch intermediate values out of buffer
  - to service read 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 memory ordering**”, 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–LLC
- Each level reduces and clusters traffic
- L1 split into I- and D-caches
  - “Harvard architecture”
  - requirement of pipelining
- Other levels unified
- Chip multiprocessors:
  - Usually LLC shared chip-wide
  - L2 private (Intel) or clustered (AMD)
**ODROID-C2 (Cortex A53) System Architecture**

**L1 cache:**
- 32 KiB, 64-B lines
  - L1-I: 2-way, virtually addr.
  - L1-D: 4-way, “physically” addr.

**L2 cache:**
- 512 KiB, 16-way
  - 64-B lines, physical
Translation Lookaside Buffer (TLB)

- TLB is a (VV) 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 Size (I-TLB+D-TLB)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Size (I+D)</th>
<th>Assoc</th>
<th>Page Size</th>
<th>Coverage</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>512 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 (PPC)</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

TLB coverage

• Memory sizes are increasing
• Number of TLB entries are roughly constant
• Base page sizes are steady
  • 4 KiB (SPARC, Alpha used 8KiB)
  • 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!
• The TLB can become a bottleneck
Multi-Level TLBs

- Multi-level design (like I/D cache)
- Improve size-performance tradeoff

**Intel Core i7**

<table>
<thead>
<tr>
<th>L</th>
<th>I/D</th>
<th>Pages</th>
<th>Assoc</th>
<th>Entr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I</td>
<td>4 KiB</td>
<td>4-way</td>
<td>64</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>4 KiB</td>
<td>4-way</td>
<td>64</td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>2/4 MiB</td>
<td>fully</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>2/4 MiB</td>
<td>4-way</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>unif</td>
<td>4 KiB</td>
<td>4-way</td>
<td>512</td>
</tr>
</tbody>
</table>

**Arm A53**

<table>
<thead>
<tr>
<th>L</th>
<th>I/D</th>
<th>Pages</th>
<th>Assoc</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I</td>
<td>4 KiB–1 GiB?</td>
<td>full?</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>4 KiB–1 GiB?</td>
<td>full?</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>unif</td>
<td>4 KiB–512 MiB</td>
<td>4-way</td>
<td>512</td>
</tr>
</tbody>
</table>
Intel Core i7 (Haswell) Cache Structure

Source: Intel
Intel Haswell L3 Cache

Source: Intel