The Memory Wall

Caching

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, [Institution]", where [Institution] is one of "UNSW" or "NICTA"

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

COMP9242 S2/2012 W02

©2011 Gernot Heiser UNSW/NICTA. Distributed under Creative Commons Attribution License
Cache Organization

- Data transfer unit between registers and L1 cache: ≤ 1 word (1–16B)
- Cache **line** is transfer unit between cache and RAM (or slower 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

Cache Indexing

- Address is 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)
  - use least-significant bits of address
Cache Indexing: Direct Mapped

- Tag used to check whether lines contain requested address
- Lower bits used to select appropriate bytes from line
- Index bits used to select unique line to match

Cache Indexing: 2-Way Associative

- Tag compared with both lines within set for match
- Lower bits used to select appropriate bytes from line

Caching Index: Fully Associative

- Tag compared with all lines within set for match
- Note: Lookup hardware for many tags is large and slow => does not scale
- Lower bits used to select appropriate bytes from line

Cache Mapping

- Different memory locations map to 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:
  - Compulsory miss: data cannot be in cache (of infinite size)
  - Capacity miss: all cache entries are in use by other data
  - Conflict miss: set of the right colour is full
  - 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 valid ⇒ must replace existing line.
- Replacement strategy must be simple (hardware)
  - Dirty bit determines whether line needs to be written back
  - Typical policies:
    - pseudo-LRU
    - FIFO
    - random
    - toss clean

Cache Write Policy

- Treatment of store operations:
  - write back: Stores update cache only
    - memory is updated once dirty line is replaced (flushed)
      - clusters writes
      - memory is inconsistent with cache
      - unsuitable for (most) multiprocessor designs
  - 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, use:
  - write allocate: allocate a cache line to the data and store
    - typically requires reading line into cache first!
  - no allocate: store to memory and bypass cache

Typical combinations:
- write-back & write-allocate
- write-through & no-allocate

Cache Addressing Schemes

- For simplicity, discussion so far assumed cache sees only one kind 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 only make sense with complex and unusual MMU designs
  - not considered here any further

Virtually-Indexed, Virtually-Tagged Cache

- Also called virtually-addressed cache
- Also (incorrectly) called virtual cache
- virtual address cache
- Uses virtual addresses only
  - can operate concurrently with MMU
  - still needs MMU
  - lookup to determine access rights
- Used for on-core L1
**Virtually-Indexed, Physically-Tagged Cache**

- Virtual address for accessing line
- Physical address for tagging
- Needs address translation completed for retrieving data
- Indexing concurrent with MMU, use MMU output for tag check
- Typically used for on-core L1

**Physically-Indexed, Physically-Tagged Cache**

- Only uses physical addresses
- Needs address translation completed before begin of access
- Typically used off-core
- Note: page offset is invariant under virtual-address translation
  - if index bits are *subset* of offset, PP cache can be accessed without result of translation!
  - VP and PP cache become the same in this case
  - fast and suitable for on-core use (esp. L1)

**Cache Issues**

- Caches are managed by hardware transparent to software
  - OS doesn’t have to worry about them, right? Wrong!
- Software-visible cache effects:
  - performance
    - *homonyms*: same name, different data
    - can affect correctness!
  - *synonyms*: different name, same data
    - can affect correctness!

**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 is accessed!
  - an issue for most OS!
- Homonym prevention:
  - flush cache on context switch
  - force non-overlapping address-space layout
    - *tag* VA with *address-space ID* (ASID)
    - makes VAs *global*
  - use physical tags
Virtually-Indexed Cache Issues

Synonyms (aliases) — multiple names for same data:
- Several VAs 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

Example: MIPS R4x00 Synonyms
- ASID-tagged, on-chip L1 VP cache
  - 16 KiB cache with 32 B lines, 2-way set associative
  - 4 KiB (base) page size
  - size/associativity = 16 KiB/2 = 8 KiB > page size
  - 16 KiB / (32 B/line) = 512 lines = 256 sets ⇒ 8 index bits (12..5)
  - overlap of tag and index bits, but come from different addresses!

Address Mismatch Problem: Aliasing
- Page aliased in different address spaces
  - AS₁: VA₁ = 1, AS₂: VA₁ = 0
- One alias gets modified
  - in a write-back cache, other alias sees stale data
  - lost-update problem

Address Mismatch Problem: Re-Mapping
- Unmap page with a dirty cache line
- Re-use (remap) frame for a different page (in same or different AS)
- Write to new page
  - without mismatch, new write will overwrite 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

You’ll have to deal with this!

**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
- Restrict VM mapping so synonyms map to same cache set
  - E.g., 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 flushing on context switch
  - Makes context switches expensive
  - May even be required on kernel–user switch
  - ... or guarantee of no synonyms and homonyms
- Require TLB lookup for write-back!
- Used on MC68040, i860, ARM7/ARM9/StrongARM/Xscale
- Used for i-caches on a number of architectures
  - Alpha, Pentium 4, ...

**Summary: Tagged VV Caches**

- Add address-space identifier (ASID) as part of tag
- On access compare with CPU’s ASID register
- Removes homonyms
  - Potentially better context switching performance
  - ASID recycling still requires cache flush
- Doesn’t solve synonym problem (but that’s less serious)
- 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 modern 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 (all index bits come from page offset) indexing can be in parallel with address translation.
  - Potentially useful for L1 cache (used on Itanium, Intel Core i7)
- ✓ Cache can use bus snooping to receive/supply DMA data
- ✓ Usable as post-MMU cache with any architecture

For an in-depth coverage of caches see [Wiggins 03]

**Write Buffer**

- Store operations can take a long time to complete
  - e.g. 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
- 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 instruction and data caches.
  - requirement of pipelining
- Low levels tend to be unified.
- Chip multiprocessors (multicores) often share on-chip L2, L3
Translation 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, typically split
    - transparent to OS
  - second level is hardware- or software-loaded

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 (superpages) - better utilises TLB entries
  - TLB lookup done without knowing the page’s base address
  - set-associativity loses speed advantage
  - superpage TLBs are fully-associative
- x86 uses separate L1 TLBs for each page size
  - 1 each I-TLB and D-TLB for 4 KiB and 2/4 MiB (total 4 L1 TLBs)
  - unified L2 TLB
  - all 4-way associative

TLB Size (I-TLB + D-TLB)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>TLB Size</th>
<th>Page Size</th>
<th>TLB Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAX</td>
<td>64–256</td>
<td>512B</td>
<td>32–128KiB</td>
</tr>
<tr>
<td>x86</td>
<td>32–128+64</td>
<td>4KiB+4MiB</td>
<td>128–128+256KiB</td>
</tr>
<tr>
<td>MIPS</td>
<td>96–128</td>
<td>4KiB–16MiB</td>
<td>384KiB–…</td>
</tr>
<tr>
<td>SPARC</td>
<td>64</td>
<td>8KiB–4MiB</td>
<td>512KiB–…</td>
</tr>
<tr>
<td>Alpha</td>
<td>32–128+128</td>
<td>6KiB–4MiB</td>
<td>256KiB–…</td>
</tr>
<tr>
<td>RS/6000</td>
<td>32+128</td>
<td>4KiB</td>
<td>128+512KiB</td>
</tr>
<tr>
<td>Power-4/G5</td>
<td>128</td>
<td>4KiB+16MiB</td>
<td>512KiB–…</td>
</tr>
<tr>
<td>PA-6000</td>
<td>96+96</td>
<td>4KiB–64MiB</td>
<td></td>
</tr>
<tr>
<td>Itanium</td>
<td>64+96</td>
<td>4KiB–4GiB</td>
<td></td>
</tr>
</tbody>
</table>

Not much growth in 20 years!

TLB Size (I-TLB + D-TLB)

- Memory sizes are increasing
- Number of TLB entries are more-or-less constant
- Page sizes are steady (4 KiB, although larger on SPARC, Alpha)
  - 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 performance bottleneck