Caches:
What Every OS Designer Must Know

COMP9242
2010/S2 Week 4

The Memory Wall

Multicore offsets stagnant per-core performance with proliferation of cores
→ Basic trend is unchanged

Caching

→ Cache is fast (1–5 cycle access time) memory sitting between fast registers and slow RAM (10–100 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- and off-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 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

- The tag is used to distinguish lines of set...
- Consists of high-order bits not used for 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 CPU caches)
- Hashing must be simple (complex hardware is slow)
  - use least-significant bits of address
Cache Indexing: Direct Mapped

- Lower 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

- Lower bits used to select appropriate bytes from line.
- Index bits used to select set to match within.
- Tag compared with all lines for a match.
- Tag compared with all lines within set for match.

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

Address Mismatch Problem: Aliasing

Page aliased in different address spaces
- AS₁: VA₁₂ = 1, AS₂: VA₁₂ = 0

One alias gets modified
- in a write-back cache, other alias sees stale data
- lost-update problem

Example: MIPS R4X00 Synonyms

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

Remember, location of data in cache determined by index
- tag only confirms whether it’s a hit!
- synonym problem if VA₁₂ ≠ VA₁₂
- similar issues on other processors, eg. ARM11 (set size 16KiB, page size 4KiB)

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

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)
→ Cache can use bus snooping to receive/supply DMA data
→ Usable as off-chip 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
  - transparent to OS.

TLB Issues: Associativity

- First TLB (VAX-11/780, [Clark, Emer 85]) was 2-way associative.
- Most modern architectures have fully associative TLBs.
- Exceptions:
  - i486 (4-way)
  - Pentium, P6 (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.

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>32-128kB</td>
</tr>
<tr>
<td>ix86</td>
<td>32-32+64</td>
<td>4KB+4MB</td>
<td>128-128+256kB</td>
</tr>
<tr>
<td>MIPS</td>
<td>96-128</td>
<td>4KB-16MB</td>
<td>384KB-....</td>
</tr>
<tr>
<td>SPARC</td>
<td>64</td>
<td>8KB-4MB</td>
<td>512KB-....</td>
</tr>
<tr>
<td>Alpha</td>
<td>32-128+128</td>
<td>6KB-4MB</td>
<td>256KB-....</td>
</tr>
<tr>
<td>RS/6000</td>
<td>32+128</td>
<td>4KB</td>
<td>128+512KB</td>
</tr>
<tr>
<td>Power-4/G5</td>
<td>128</td>
<td>4KB+16MB</td>
<td>512KB-....</td>
</tr>
<tr>
<td>PA-6000</td>
<td>96+96</td>
<td>4KB-64MB</td>
<td></td>
</tr>
<tr>
<td>Itanium</td>
<td>64+96</td>
<td>4KB-4GB</td>
<td></td>
</tr>
</tbody>
</table>

TLB coverage

- Memory sizes are increasing
- Number of TLB entries are more-or-less constant
- Page sizes are growing very slowly
  - 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 modern architectures have software-loaded TLBs
- General increase in TLB miss handling cost
- The TLB is becoming a performance bottleneck.
Address Space Usage vs. TLB Coverage

- Each TLB entry maps one virtual page
- On TLB miss, reloaded from page table (PT), which is in memory
  - Some TLB entries need to map page table
  - E.g. 32-bit page table entries, 4KiB pages
  - One PT page maps 4MiB
- 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
  - however, difficult to manage in the OS

Sparse Address-Space Use

- Ties up many TLB entries for mapping page tables

Origins of Sparse Address-Space Use

- Modern OS features:
  - memory-mapped files
  - dynamically-linked libraries
  - mapping IPC (server-based systems)...
- This problem gets worse 64-bit address spaces:
  - bigger page tables
- An in-depth study of such effects can be found in [Uhlig et al. 94]

Case Study: Context Switches on ARM

- Typical features of ARM v4/v5 cores with MMU:
  - Virtually-addressed split L1 caches
  - No L2 cache
  - No address-space tags in TLB or caches
  - Other features to be discussed later
- Representatives:
  - ARM7, StrongARM (ARMv4)
  - ARM9, Xscale (ARMv5)

The following is based on [Wiggins et al. 03], updated with [van Schaik 07]
ARM v4/v5 Memory Architecture

- Virtually-indexed, virtually-tagged caches

ARM Cache Issues

- Virtually-indexed, virtually-tagged caches
  - Contents are tied to address space
  - For coherency, flush caches on context switch
- Flushing is expensive!
  - Direct cost: 1k-18k cycles
  - Indirect cost: up to 54k cycles
- Permissions from TLB
  - Could avoid flushes if no address-space overlap
  - Infeasible in normal OS

ARM PID Relocation

- Processor supports relocation of small address spaces
  - Lowest 32MiB of AS get mapped to higher regions
  - Mapping slot selected by process-ID (PID) register
  - Re-mapping happens prior to TLB lookup
- Re-mapped address spaces don’t overlap
  - no need to flush caches on address-space switch
  - Sounds fine, but what about protection?

ARM v4/v5 TLB

- No address space tags on TLB entries
- However, 4 bit domain tag
- Domain access control register en/disables domains
**ARM TLB Issues**

- No address-space tag in TLB
  - Need to keep mappings from different AS separate
  - Flush TLB on context switch
- Flushing is expensive!
  - Direct cost: 1 cycle
  - Indirect cost: 3k cycles
- Permissions on cache data from TLB
  - TLB flush requires cache flush!
- Better: make use of *domains*
  - Use as poor man’s address-space tags
  - Play tricks with page tables

**Domains for Fast Address-Space Switch**

- Caching page directory mixes entries from different AS
- Tagged with per-address-space domain
- Hardware detects collisions (via DACR)
- Full performance if no overlap, flush on collisions
- Implementation details in paper

**Fast Address-Space Switching**

- Multiple ASs co-exist in top-level page table and TLB
- TLB and cache flushes are only required on collisions
  - minimised by the use of PID relocation
  - minimised by the use of a single-address-space layout (Iguana)
  - may happen as a result of:
    - address-space overflow (with PID relocation)
    - conflicting mappings (mmap with MAP_FIXED)
    - out of domains

**Fast Address-Space Switch Issues**

- Only 16 domains
  - Must recycle domains when exhausted
- User-level thread control blocks (UTCBs)
  - Aliased between user and kernel
  - There are ways to make this work
  - Better: let kernel determine UTCB location (as in OKL4)
OKL4 Implementation

- Kernel transparently assigns domains
  - 1 reserved for kernel, 15 available for user processes
- When out of domains, preempt one and flush
- Kernel keeps track of domains used since last flush
  - If not used since last flush, domain is clean
  - If possible, preempt clean domain, requires no cache flush
  - Otherwise preempt a random domain
- Kernel keeps per-domain bitmask of used CPD entries
  - Supports easy detection of AS collisions (at 1 MiB granularity)

Alternative Page Table Format

- Top level of AS’s page table is no longer hardware walked
  - 16KiB is mostly wasted on small processes (typical in embedded)
- Can replace by more appropriate (denser) data structure
- Save significant amount of kernel memory (up to 50%)
- Same benefit on ARM v6

Performance: Linux Microbenchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>Wombat</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>lmbench context-switch latency [us] smaller is better</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lat_ctx - s 0 1</td>
<td>11</td>
<td>20</td>
<td>0.6</td>
</tr>
<tr>
<td>lat_ctx - s 0 2</td>
<td>262</td>
<td>5</td>
<td>52</td>
</tr>
<tr>
<td>lat_ctx - s 0 10</td>
<td>298</td>
<td>45</td>
<td>6.6</td>
</tr>
<tr>
<td>lat_ctx - s 4 1</td>
<td>48</td>
<td>58</td>
<td>0.8</td>
</tr>
<tr>
<td>lat_ctx - s 4 10</td>
<td>419</td>
<td>203</td>
<td>2.1</td>
</tr>
<tr>
<td>lat_fifo</td>
<td>509</td>
<td>49</td>
<td>10</td>
</tr>
<tr>
<td>lat_unix</td>
<td>1015</td>
<td>77</td>
<td>13</td>
</tr>
</tbody>
</table>

Native Linux vs OKL4/Wombat on PXA255 @ 400MHz
Performance: OKL4 with FASS vs Standard Linux

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>Wombat</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lmbench lat_proc procedure</code></td>
<td>0.21</td>
<td>0.21</td>
<td>1.0</td>
</tr>
<tr>
<td><code>lmbench lat_proc fork</code></td>
<td>0.67</td>
<td>0.82</td>
<td>0.8</td>
</tr>
<tr>
<td><code>lmbench lat_proc exec</code></td>
<td>17.40</td>
<td>26.00</td>
<td>0.7</td>
</tr>
<tr>
<td><code>lmbench lat_proc shell</code></td>
<td>465.00</td>
<td>588.00</td>
<td>0.8</td>
</tr>
<tr>
<td><code>lmbench bandwidth [MB/s]</code>, larger is better</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bw_files_rm 1024 io only</code></td>
<td>38.9</td>
<td>26.5</td>
<td>0.7</td>
</tr>
<tr>
<td><code>bw_mmap_rm 1024 mmap only</code></td>
<td>106.7</td>
<td>105.7</td>
<td>1.0</td>
</tr>
<tr>
<td><code>bw_mmap 1024 rd</code></td>
<td>416.4</td>
<td>412.4</td>
<td>1.0</td>
</tr>
<tr>
<td><code>bw_mmap 1024 wr</code></td>
<td>132.8</td>
<td>191.3</td>
<td>0.7</td>
</tr>
<tr>
<td><code>bw_mmap 1024 rdwr</code></td>
<td>218.8</td>
<td>216.5</td>
<td>1.0</td>
</tr>
<tr>
<td><code>bw_pipe</code></td>
<td>7.55</td>
<td>20.64</td>
<td>2.7</td>
</tr>
<tr>
<td><code>bw_unix</code></td>
<td>17.5</td>
<td>11.6</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Issues: Sharing

- Linux server and Linux app share data (argument buffers)
- Standard FASS scheme sees this as collisions
  - flushes caches and TLB
- Implemented `vspace` feature
  - allows identifying AS "families" with non-overlapping layout
  - sharing within family avoids cache flush
  - TLB still flushed
  - details in [van Schaik 07]
- TLB flushes are unnecessary overhead
  - performance degradation, especially on I/O syscalls

Better Approach to Sharing

Objectives

- Avoid TLB flushes on ARM v4/v5
  - need to use separate domain ID for shared pages
- Allow sharing of TLB entries where HW supports it
  - ARM, segmented architectures (PowerPC, Itanium)
- Unified API abstracting over architecture differences
**Arm Domains for Sharing**

- **PD**
  - PD₂
  - PD₁
- **LTP**
  - LTPₐ₀
  - LTP₂₀
  - LTP₁₀
  - LTP₁₁
- **CDP**
- **DACR**

**Better Approach to Sharing**

**Idea: Segments as an abstraction for sharing**
- Direct match on PowerPC, Itanium
- Maps reasonably well to ARM
  - Provided sharing is at same virtual address
  - Maps well to typical use

**Segment API Implementation (ARM)**

- Allocate unique domain ID first time a segment is mapped
  - Provided segment base and size is aligned to 1MiB
  - Domain is freed when segment is unmapped from last AS
  - Domain ID is enabled in DACR for all ASes mapping segment
- Will automatically share TLB entries for shared segments
  - Provided full access rights for all sharers
- Allows avoiding remaining aliasing problems

**Conclusions**

- Fast context-switching on ARM shows impressive results
  - Up to 50 times lower context-switching overhead
- Same mechanism supports reduction of kernel memory
  - Save 16KiB per process for top-level page table
  - This accounts for up to half of kernel memory!
- Shared pages still require TLB flush
  - Eg for Wombat accessing user buffers
- Segment API solves this elegantly
  - And also enables use of HW support for TLB sharing