Caches: What Every OS Designer Must Know

COMP9242
2008/S2 Week 4

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

The Memory Wall

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 (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
Cache Indexing

The tag is used to distinguish lines of set...
... consists of the address bits not used for indexing.

Cache Indexing: Direct Mapped

Cache Indexing: 2-Way Associative

Cache Indexing: Fully Associative

Caching Index: Fully Associative

- 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 i lines of the same colour

Types of cache misses:
- Compulsory miss: data cannot be in cache (if infinite size)
  - first access (after flush)
- Capacity miss: all cache entries are in use by other data
- Conflict miss: set mapped to address is full
  - miss that would not happen on fully-associative cache
- Coherence miss: miss forced by hardware coherence protocol
  - multiprocessors

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).
  - Typical policies:
    - pseudo-LRU
    - FIFO
    - random
    - least 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

**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
    - Used on-core

**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 on-core

**Physically-Indexed, Virtually-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
  - index bits are subset of offset,
    - PP cache can be accessed without result of translation
  - fast and suitable for on-core use
**Cache Issues**

- Caches are managed by hardware transparent to software.
  - OS doesn't have to worry about them.
- Software-visible cache affects:
  - Performance
  - Address Mismatch Problem: Aliasing
    - Same name, different data
      - can affect correctness!
  - Synonyms (aliases) — multiple names
    - 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 OSs!
- 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

**Example: MIPS R4X00 Synonyms**

- ASID-tagged, on-chip L1 VP cache
  - 16KByte cache with 32B lines, 2-way set associative
  - 4KByte (basic) page size
  - set size = 16K/2 = 8 KiB
  - 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 iff VA1 = VA2
  - similar issues on other processors, eg. ARM11 (set size 16KiB, page size 4KiB)

**Address Mismatch Problem: Aliasing**

- Page aliased in different address spaces
  - different VA, same data
  - 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

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., R4600: ensure that VA = PA

Summary: VV Caches

- Fastest (don’t rely on TLB for retrieving data)
- Lower TLB backup for protection
- Suffer from synonyms and homonyms
- Requires flushing on context switch
- Makes context switches expensive
- May even require kernel–user switch
- No guarantee of no synonyms and homonyms
- Require TLB backup for write-back!
- Used on M68040, 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 synonyms
- 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 synonym problem
- No homonym 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
- Write buffer is a FIFO queue of incomplete stores
  - 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 (V) 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)

- 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

<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>Ibx</td>
<td>32–128</td>
<td>4kB–4MB</td>
<td>384KB–...</td>
</tr>
<tr>
<td>MIPS</td>
<td>96–128</td>
<td>4kB–16MB</td>
<td>512kB–...</td>
</tr>
<tr>
<td>SPARC</td>
<td>64</td>
<td>8kB–4MB</td>
<td>256kB–...</td>
</tr>
<tr>
<td>Alpha</td>
<td>32–128</td>
<td>8kB–8MB</td>
<td>128kB–512kB</td>
</tr>
<tr>
<td>RS/6000</td>
<td>32–128</td>
<td>4kB</td>
<td>512kB–...</td>
</tr>
<tr>
<td>Power-4/650</td>
<td>128</td>
<td>48kB–16MB</td>
<td>512kB–...</td>
</tr>
<tr>
<td>PA-8000</td>
<td>90–96</td>
<td>4kB–6MB</td>
<td></td>
</tr>
<tr>
<td>Itanium</td>
<td>64–90</td>
<td>4kB–4GB</td>
<td></td>
</tr>
</tbody>
</table>

Not much growth in 20 years!
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 (map 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
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 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
  - 10KB is mostly wasted on small processes (typical in embedded)
- Can replace by more appropriate (denser) data structure
- Saves 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>lat_ctx</td>
<td>11</td>
<td>25</td>
<td>0.6</td>
</tr>
<tr>
<td>lat_ctx</td>
<td>262</td>
<td>5</td>
<td>52</td>
</tr>
<tr>
<td>lat_ctx</td>
<td>228</td>
<td>45</td>
<td>6.6</td>
</tr>
<tr>
<td>lat_ctx</td>
<td>419</td>
<td>203</td>
<td>2.1</td>
</tr>
<tr>
<td>lat_ifio</td>
<td>596</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

- Imbench context-switch latency
- Imbench pipe bandwidth

Performance: Linux Microbenchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>Wombat</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>lat_ctx</td>
<td>12.7</td>
<td>0.97</td>
<td>1.3</td>
</tr>
<tr>
<td>lat_ctx</td>
<td>10.7</td>
<td>6.23</td>
<td>0.7</td>
</tr>
<tr>
<td>lat_ctx</td>
<td>13.7</td>
<td>20.96</td>
<td>0.7</td>
</tr>
<tr>
<td>lat_ctx</td>
<td>25.8</td>
<td>26.3</td>
<td>0.7</td>
</tr>
<tr>
<td>lat_ifio</td>
<td>106.7</td>
<td>106.7</td>
<td>1.0</td>
</tr>
<tr>
<td>lat_ifio</td>
<td>480.8</td>
<td>480.8</td>
<td>1.0</td>
</tr>
<tr>
<td>lat_ifio</td>
<td>410</td>
<td>410</td>
<td>1.0</td>
</tr>
<tr>
<td>lat_ifio</td>
<td>216.7</td>
<td>216.7</td>
<td>1.0</td>
</tr>
<tr>
<td>lat_ifio</td>
<td>17.2</td>
<td>17.2</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Native Linux vs OKL4/Wombat on PXA255 @ 400MHz
Issues: Sharing

- Wombat 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 MKES'07 paper
- 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
  - Need an API for this
- Allow sharing of TLB entries where HW supports it
  - ARM, segmented architectures (PowerPC, Itanium)
- Unified API abstracting over architecture differences
- TLB flushes are unnecessary overhead
  - Performance degradation, especially on I/O syscalls

ARM Domains for Sharing

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
  - E.g. for Wombat accessing user buffers
- Segment API solves this elegantly
  - And also enables use of HW support for TLB sharing