

## Caches: What Every OS Designer Must Know

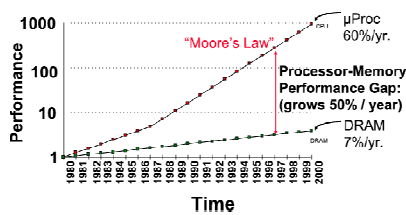
COMP9242  
2009/S2 Week 5

### Copyright Notice

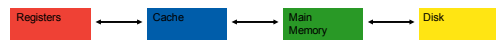
These slides are distributed under the Creative Commons Attribution 3.0 License

- 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



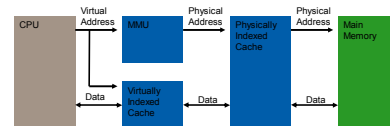
- Cache is fast (1–5 cycle access time) memory sitting between fast registers and slow RAM (10–100 cycles access time)
- Holds recently-used data or instructions to save memory accesses
- Matches slow RAM access time to CPU speed if high hit rate (90%)
- Is hardware maintained and (mostly) transparent 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:  $\leq 1$  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

UNSW

→ The *tag* is used to distinguish lines of set...

→ ... consists of the address bits not used for indexing.

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

### Cache Indexing

UNSW

→ 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  $n = 1 \dots 5$ , some embedded processors use 32–64
- $n = 1$  is called *direct mapped*.
- $n = \text{No. lines}$  is called *fully associative* (unusual for CPU caches)

→ Hashing must be simple (complex hardware is slow)

- use least-significant bits of address

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

### Cache Indexing: Direct Mapped

UNSW

Lower bits used to select appropriate bytes from line

Index bits used to select line to match

Tag used to check whether lines contains requested address

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

### Cache Indexing: 2-Way Associative

UNSW

Lower bits used to select appropriate bytes from line

Index bits used to select set to match within

Tag compared with both lines within set for match

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

### Caching Index: Fully Associative

UNSW

Lower bits used to select appropriate bytes from line

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

Tag compared with all lines for a match

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

### Cache Mapping

UNSW

→ 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)
  - first access (after flush)
- **Capacity miss:** all cache entries are in use by other data
- **Conflict miss:** set of the right colour is full
  - miss that would not happen on fully-associative cache
- **Coherence miss:** miss forced by hardware coherence protocol
  - multiprocessors

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

### Cache Replacement Policy

UNSW

- 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

Address

tag <sub>set</sub>		index <sub>set</sub>		byte <sub>set</sub>	
--------------------	--	----------------------	--	---------------------	--

VD	Tag	Word 0	Word 1	Word 2	Word 3
VD	Tag	Word 0	Word 1	Word 2	Word 3
VD	Tag	Word 0	Word 1	Word 2	Word 3
VD	Tag	Word 0	Word 1	Word 2	Word 3
VD	Tag	Word 0	Word 1	Word 2	Word 3
VD	Tag	Word 0	Word 1	Word 2	Word 3
VD	Tag	Word 0	Word 1	Word 2	Word 3
VD	Tag	Word 0	Word 1	Word 2	Word 3
VD	Tag	Word 0	Word 1	Word 2	Word 3

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 13

### Cache Write Policy

UNSW

- 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

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 14

### Cache Addressing Schemes

UNSW

- 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

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 15

### Virtually-Indexed, Virtually-Tagged Cache

UNSW

- 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 on-core

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 16

### Virtually-Indexed, Physically-Tagged Cache

UNSW

- 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

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 17

### Physically-Indexed, Physically-Tagged Cache

UNSW

- 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

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 18

## Cache Issues

UNSW

→ 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!

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 19

## Virtually-Indexed Cache Issues

UNSW

**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

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 20

## Virtually-Indexed Cache Issues

UNSW

**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

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 21

## Example: MIPS R4X00 Synonyms

UNSW

→ 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 iff  $VA_{12} \neq VA'_{12}$
- similar issues on other processors, eg. ARM11 (set size 16KiB, page size 4KiB)

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 22

## Address Mismatch Problem: Aliasing

UNSW

→ 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

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 23

## Address Mismatch Problem: Re-Mapping

UNSW

→ Unmap page with a dirty cache line

→ Re-use (remap) frame for a different page (in same or different AS)

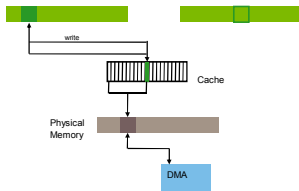
→ Write to new page

- without mismatch, new write will overwrite old (hits same cache line)
- with mismatch, order can be reversed: "cache bomb"

©2008 Gemot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 24

## DMA Consistency Problem

UNSW



- 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

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

25

## Avoiding Synonym Problems

UNSW

- 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  $VA_{12} = PA_{12}$

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

26

## Summary: VV Caches

UNSW

- ✓ 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, ...

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

27

## Summary: Tagged VV Caches

UNSW

- 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

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

28

## Summary: VP Caches

UNSW

- 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

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

29

## Summary: PP Caches

UNSW

- 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]

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License

30

## Write Buffer

UNSW

- 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

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 31

## Cache Hierarchy

UNSW

- 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

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 32

## Translation Lookaside Buffer (TLB)

UNSW

- 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

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 33

## TLB Issues: Associativity

UNSW

- 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
  - superpage TLBs are fully-associative

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 34

## TLB Size (I-TLB + D-TLB)

UNSW

Architecture	TLB Size	Page Size	TLB Coverage
VAX	64-256	512B	32-128KIB
ix86	32-32+64	4KIB-4MIB	128-128+256KIB
MIPS	96-128	4KIB-16MIB	384KIB-...
SPARC	64	8KIB-4MIB	512KIB-...
Alpha	32-128+128	8KIB-4MIB	256KIB-...
RS/6000	32+128	4KIB	128+512KIB
Power-4/G5	128	4KIB+16MIB	512KIB-...
PA-8000	96+96	4KIB-64MIB	
Itanium	64+96	4KIB-4GIB	

Not much growth in 20 years!

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 35

## TLB Size (I-TLB + D-TLB)

UNSW

**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

©2008 Gerrot Heiser UNSW/NICTA/OKL. Distributed under Creative Commons Attribution License 36

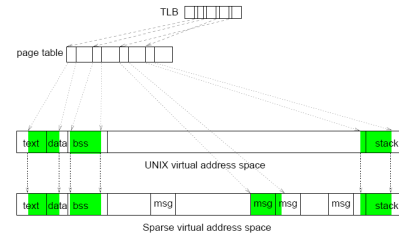
## Address Space Usage vs. TLB Coverage

UNSW

- 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

UNSW



## Origins of Sparse Address-Space Use

UNSW

- 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]