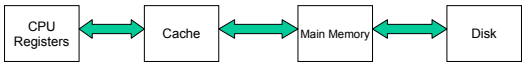
Caches

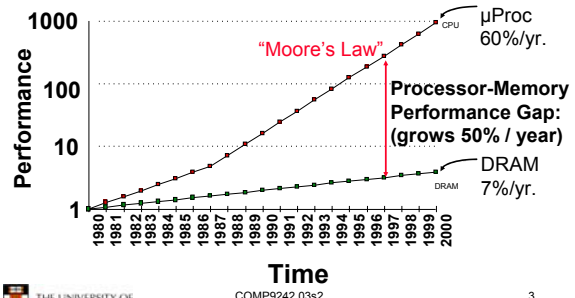
From a Mostly OS Software Perspective

Cache



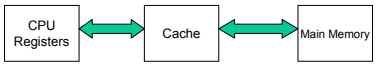
- Cache is fast memory placed between the CPU and main memory
  - 1 to a few cycles access time compared to RAM access time of tens – hundreds of cycles
- Holds recently used data or instructions to save memory accesses.
- Matches slow RAM access time to CPU speed if high hit rate
- (└ 90 %)
- Is hardware maintained and (mostly) transparent to software
- Sizes range from few kB to several MB.
- Usually a hierarchy of caches (2–5 levels), on- and off-chip.

Processor-DRAM Gap (latency)



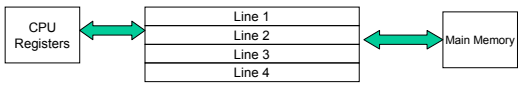
Slide originally from Dave Patterson, Parcon 98

Cache organisation



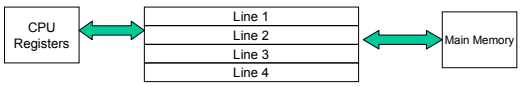
- Data transfer unit between registers and cache 1 word (1–8B)
  - Determined by data operand and instruction fetch size
- Date transfer unit between cache and RAM is a *cache line*, typically 16–32 bytes, sometimes 128 bytes and more.
  - Memory bus width * number of *burst* memory cycles
  - Large lines more efficient to load, but increases false sharing
- Line is unit of storage allocation in cache.

Cache organisation



- In addition to data, cache lines have control attributes
  - valid bit
  - modified bit
  - tag bit
    - for detecting matches

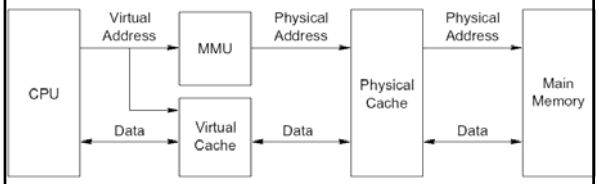
Cache organisation



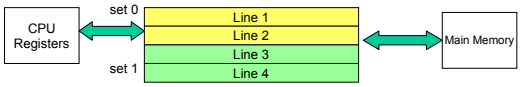
- Cache improves memory access by:
  - absorbing most reads (increases bandwidth, reduces latency),
  - making writes asynchronous (hides latency),
  - clustering reads and writes (hides latency).
    - burst read/write to RAM more efficient than single transactions

Cache Access

- **Virtually indexed:** looked up by *virtual address*, operates concurrently with address translation.
- **Physically indexed:** looked up by *physical address*

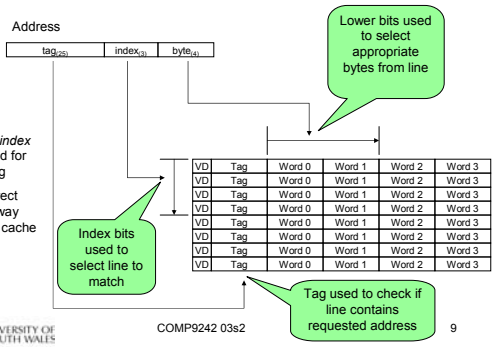


Cache Access Mechanics

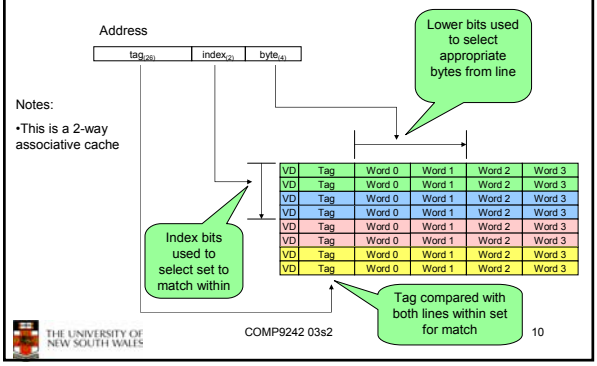


- Address is hashed to produce index of line set.
- Associative lookup of line within set n lines per set: n-way set associative cache.
  - Typically n = 1 . . . 5.
    - Above example is a two-way (n = 2) associative cache
  - n = 1 is called direct mapped.
  - n = ∞ is called fully associative, unusual for CPU caches.
- Hashing must be simple (complex hardware is slow)
  - ⇒ use least-significant bits of address.
    - ⇒ Gives a better hash as it has more varied distribution

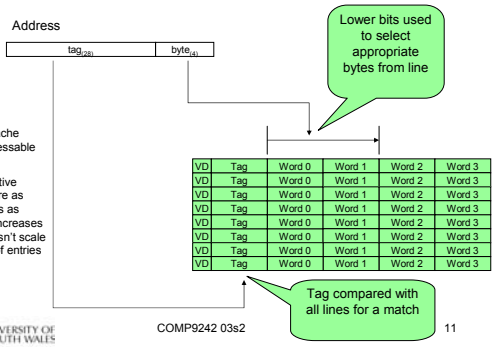
Cache Indexing Example



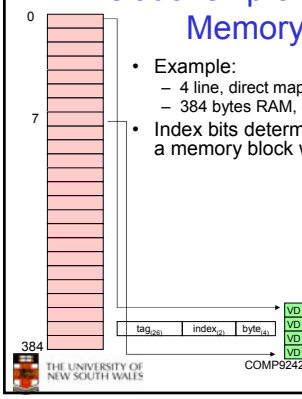
Cache Indexing Example



Cache Indexing Example



Relationship of Cache Line to Memory Blocks



- Example:
  - 4 line, direct mapped cache (64 byte)
  - 384 bytes RAM, divided into lines
- Index bits determine location within cache that a memory block will appear in

Relationship of Cache Line to Memory Blocks

- Different memory blocks map to the same cache line (or set of cache lines)
- All memory blocks mapping to cache line #*i* are said to be of colour *i*.
- *n*-way associative cache can hold *n* blocks of the same colour
  - In our example, only a single block of each colour can be held

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

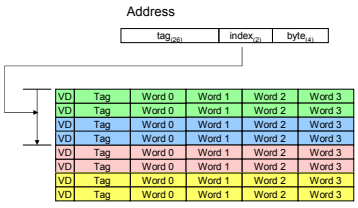
Cache Miss Types

- Capacity misses:
  - cache is full.
- Conflict misses:
  - set corresponding to address is full, even though there may be unused lines in the cache.
  - Higher associativity reduces conflict misses at the expense of access time (trade-off)

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

Cache Replacement Policy

- Index (physical or virtual) determines line set
  - Direct-mapped requires no policy
- If miss and all lines are valid, we must replace one
- Replacement policy must be simple as it's done in hardware
- Typical polices
  - Psuedo LRU
  - Random
  - FIFO
- Note: If existing line is dirty, we must write it back to memory

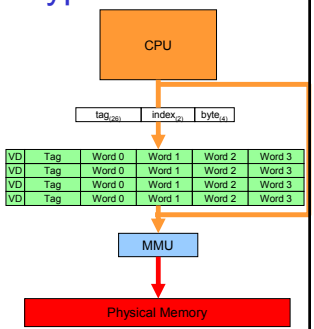


Cache Write Policy

- Treatment of store operations:
  - **write back:** Stores update cache only, memory updated when dirty line is replaced (flushed).
    - ✓ Consolidates multiple writes into a single memory update
    - × Memory is inconsistent with cache.
  - **write through:** Stores update cache (or bypass it) **and** memory immediately.
    - ✓ Memory is always consistent with cache.
    - × Slow writes to memory speed
- On store to a line not presently in cache, use:
  - **write allocate:** allocate a cache line to the data and store,
  - **no allocate:** store to memory and bypass cache.
- Usual combinations:
  - write-back & write-allocate,
  - write-through & no-allocate.

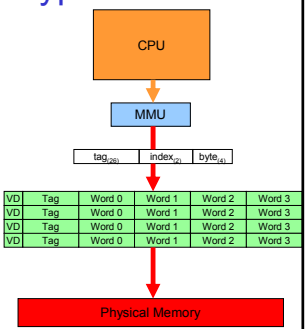
Cache Types

- Virtually indexed, virtually tagged
  - also called virtual cache
  - Can operate concurrently with MMU



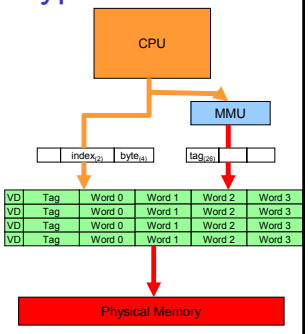
Cache Types

- Physically Index, Physically Tagged
  - also called physical cache
  - Must wait for MMU to complete prior to beginning cache lookup



Cache Types

- Virtually Index, Physically Tagged
  - Virtual address for indexing, physical address for tagging
  - Needs address translation to complete to retrieve data
  - Index concurrently with MMU, MMU output used to check tag



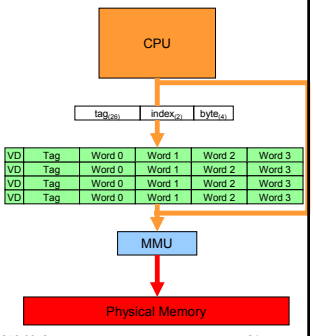
Cache Types

- Typically
  - On-chip caches are virtually indexed
  - Off-chip caches are physically indexed
    - MMU usually on-chip

Virtual Cache Issues

Homonyms

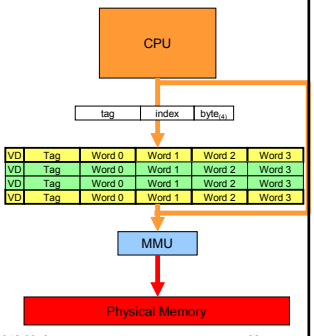
- Same VA corresponds to several PAs
  - standard situation in multitasking systems (not SASOS!)
- Problem: tag may not uniquely identify cache data!
  - Homonyms lead to cache accessing the wrong data!
- Homonym prevention:
  - tag virtual address with address-space ID (ASID)
    - disambiguates virtual addresses (makes them globally valid)
  - use physical tags
  - flush cache on context switch
  - use a SASOS



Virtual Cache Issues

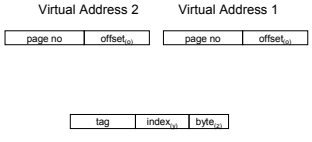
Synonyms (Aliases)

- Several VAs correspond to the same PA
  - frames shared between processes
  - same frame mapped multiply within the same address space
- Synonyms in cache may lead to accessing stale data:
  - same data may be cached in several lines
  - on write, one synonym is update
  - a subsequent read on the other synonym returns the old value
- Solutions?
  - Physical tagging doesn't help!
  - ASIDs don't help either!
  - Whether synonyms are a problem depends on cache size and page size!



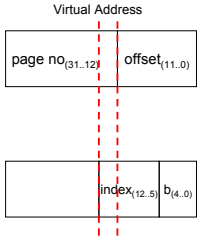
Virtually Indexed Cache Without Aliasing

- For synonyms, only the page number is different, the offset is the same
- If the cache index is formed only from the offset bits, the index will also be identical for aliased pages
  - All aliases will select the same cache line (or set)
  - Only one copy of the line can exist in the cache
- Restriction
  - $y+z \le o$
  - Limits cache size to *page size × associativity*



Large Virtually Indexed Cache Example: R4000

- 16 Kbyte I & D Cache, 2-way associative, 4Kbyte (minimum) page size, 32 bytes lines, physical tags.
  - 5 bits to index data within a line
  - 8 bits to index 256 sets
- Synonyms exist
  - VA_{12} forms part of index
  - References to aliases where $VA_{12} \neq VA'_{12}$ result in indexing of different cache sets



Avoiding Synonyms

- 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 VA12 = PA12

Fully Virtual Caches

- Fastest (don't rely on TLB for retrieving data)
  - however, needs TLB 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

Fully Virtual Caches with Keys

- Add *address-space identifier* (ASID) as part of tag.
  - On access compare with CPU's ASID register.
  - Turns homonyms into synonyms
- Removes homonym problem
- Potentially better context switching performance
- ASID recycling still requires cache flush
- Doesn't solve synonym problem
- Doesn't solve write-back problem

Virtually indexed, physically tagged caches

- Medium speed:
  - lookup in parallel with address translation
  - tag comparison after address translation
- No homonym problem
- Potential synonym problem
- More bits per tag (cannot leave off set-number bits)
  - increases area, latency, power consumption
- Used on most modern architectures for L1 cache

Physical Caches

- 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.
  - Attractive feature for intermediate level cache.
- Cache can use bus snooping to receive/supply DMA data
- Usable as off-chip cache with any architecture.

Cache Hierarchy

- Use a hierarchy of caches to balance memory performance and price:
  - Small, fast, virtually indexed cache on top (L1 cache).
  - Large, slow, physically indexed cache at bottom (L2–L5).
- Each level reduces and clusters traffic.
- High levels tend to be separated into instruction and data caches.
- Low levels tend to be unified.

Translation Lookaside Buffers

- TLB is a cache for page table entries.
- Can be:
  - Hardware loaded, transparent to OS, or
  - software loaded, maintained by OS.
- Can be:
  - Split, instruction and data TLBs, or
  - unified.
- Some architectures (MIPS, IA-64) 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, [CE85]) was 2-way associative.
- Most modern architectures have fully associative TLBs.
- Exceptions:
  - i486 (4-way),
  - Pentium, Pentium-Pro (4-way),
  - IBM RS/6000 (2-way).
- Superpages ⇒ fully associative TLB

SIZE (I-TLB + D-TLB)

- Note: not much growth in 20 years!

Architecture	*TLB Size*
VAX	64–256
ix86	32–32+64
MIPS	96–128
SPARC	64
Alpha	32–128+128
RS/6000	32+128
PA-8000	96+96
Itanium	64+96

TLB Coverage

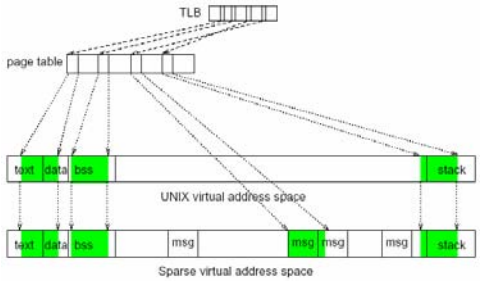
- Memory sizes are increasing.
- Number of TLB entries are more-or-less constant.
- Page sizes are growing very slowly.
  - ⇒ Total amount of RAM mapped by TLB is not changing much.
  - ⇒ Fraction of RAM mapped by TLB is shrinking dramatically.
- Modern architectures have very low TLB coverage.
- Also, many modern architectures have software-loaded TLBs.
  - ⇒ General increase in TLB miss handling cost.

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, 4kb pages.
  - One PT page maps 4Mb.
- 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.

Sparse Address Space Use Ties Up PT Entries



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 [UNS+94].

Consistency Management for Virtually Indexed Caches

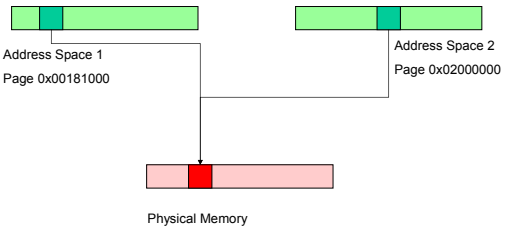
Bob Wheeler and Brian N. Bershad

Proc. 5th Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS - V)

Overview

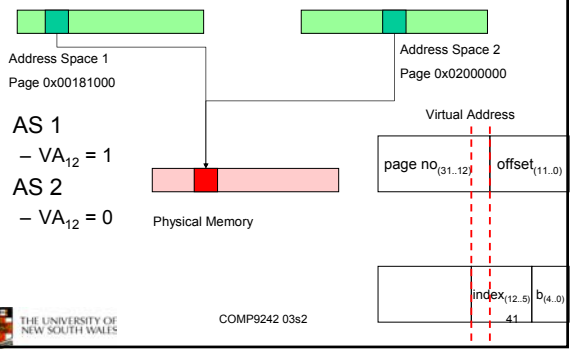
- Virtually indexed caches are attractive for their low-level performance advantages.
- Unattractive from the operating systems perspective
  - Aliasing
  - (Re-)Establishment of mappings to physical frames
  - DMA-based I/O

Problem: Aliasing



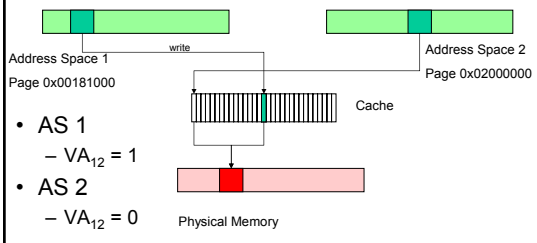
- Two pages mapped to the same frame

Recall R4000 Cache Indexing Scheme



- AS 1
  - $VA_{12} = 1$
- AS 2
  - $VA_{12} = 0$

Problem: Aliasing

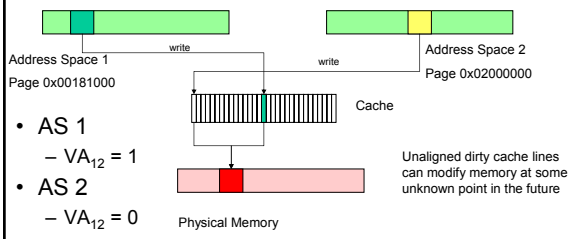


- AS 1
  - $VA_{12} = 1$
- AS 2
  - $VA_{12} = 0$

Solutions?

- Force alignment of mapping such that indexing always results same cache line accessed
  - Unduly restrictive
  - May require modification of applications
- Only allow read-only aliases
  - How do we modify data
- Flush the cache on context switches
  - Doesn't help with multiply mapped frames
  - Expensive

Problem: Re-mapping

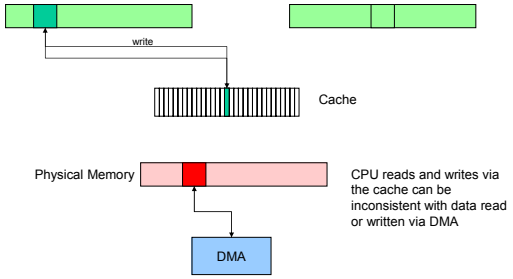


- AS 1
  - $VA_{12} = 1$
- AS 2
  - $VA_{12} = 0$

Solutions?

- Always flush cache after unmapping a page
  - May not be required if new mapping has same alignment, or
  - cache will be indirectly (partially) flushed via future activity
    - A future flush may be less expensive

Problem: DMA-based I/O



Solutions?

- Flush the cache prior to any DMA
  - Expensive
  - Might not be required
    - DMA reads maybe be targeting DMA writes

Goal

- We would like a solution that
  - Ensures stale data is never read
  - All reads return the most recent write
- Aims to minimise the number of cache flushes required
- Avoids placing restrictions on use of mapping
  - Though encourages the avoidance of aliasing

Developing a solution

- A model for activity
  - CPU-read, CPU-write
    - Encompasses CPU-caused events that affect cache state
  - DMA-read, DMA-write
    - Encompasses DMA-caused events that affect cache state
  - Flush, Purge
    - Encompasses operations on the cache itself
      - Flush: empty the cache (line) writing back any dirty lines
      - Purge: simply empty the cache ignoring dirty lines

Developing a Solution

- Cache-line states
  - E: Empty
  - P: Present
  - D: Dirty
  - S: Stale
    - Line is out of date with respect to the most recent update
      - Allows us to postpone (maybe avoid) cache operations compared to *eager* approach of not allowing stale data to exist.

A Correct Solution

- A solution is correct if CPU or DMA never reads stale data
- Example:
  - Read to a stale cache line results in a purge to empty first

Cache State Transition Table

Operation on target address	Target cache line	All other similarly mapped but unaligned cache lines
CPU-read	E → P P → P D → D S $\xrightarrow{purge}$ E → P	E → E P → P D $\xrightarrow{flush}$ E S → S
CPU-write	E → D P → D D → D S $\xrightarrow{purge}$ E → D	E → E P → S D $\xrightarrow{flush}$ E S → S
DMA-read	E → E P → P D $\xrightarrow{flush}$ E S → S	E → E P → P D $\xrightarrow{flush}$ E S → S
DMA-write	E → E P → S D $\xrightarrow{purge}$ E S → S	E → E P → S D $\xrightarrow{purge}$ E S → S
Purge	E → E P → E D → E S → E	E → E P → P D → D S → S
Flush	E → E P → E D → E S → E	E → E P → P D → D S → S

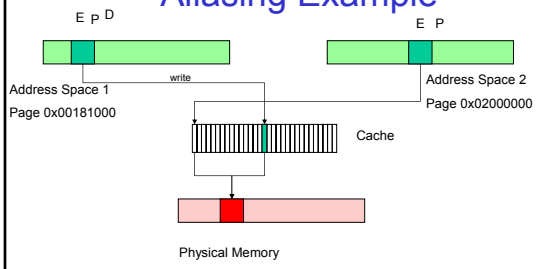
Implementing Model

- We need a method to detect (and control) access to the differently aligned mappings to physical memory
  - Detection allows us to take preventative action if access would result in inconsistent data.
- Idea
  - Use the virtual memory hardware to control access
    - ⇒ Perform cache consistency on a page basis (instead of line-by-line)
    - ⇒ Keep state associated with pages
    - ⇒ Avoid access to stale pages
  - DMA is controlled by kernel (can invoke consistency primitives directly)

Implementing the model

- Keep track of
  - all mappings to a frame
  - their *cache state*
  - their alignment

Aliasing Example



See the Paper for further details

Your project and the U4600

- The U4600 has same cache architecture described
  - It has the same aliasing problems
  - Same problems with DMA
  - Additionally, the D-cache and I-cache are not kept consistent by hardware
- There is potential for you to cause obscure, very-difficult-to-find bugs by not considering the cache when managing your mapping.
- Fortunately, we'll give you a kernel with caching switched off
  - Enabling caching is an optional challenge worth attempt
    - Once everything else is running
  - You are not required to use DMA
    - any DMA done by the network driver should handle the cache

References

[CE85] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *Trans. Comp. Syst.*, 3:31–62, 1985.

[Sch94] Curt Schimmel. *UNIX Systems for Modern Architectures*. Addison Wesley, 1994.

[UNS+94] Richard Uhlig, David Nagle, Tim Stanley, Trevor Mudge, Stuart Sechrest, and Richard Brown. Design tradeoffs for software-managed TLBs. *Trans. Comp. Syst.*, pages 175–205, 1994.