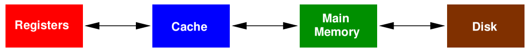
Caching

From an OS Perspective

COMP9242 2005/S2 Week 3

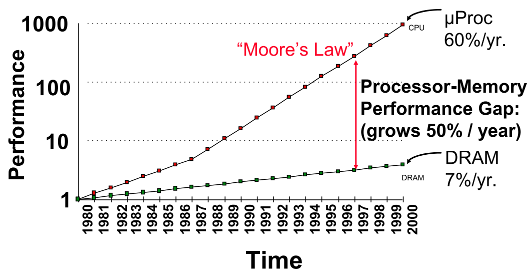
Caching



- 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* ($\geq 90\,\%$)
- 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.

Good overview in [Sch94].

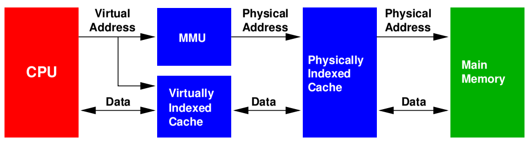
The Memory Wall



Cache Organisation

- Data transfer unit between registers and cache ≤ 1 word (1–16B)
- Data transfer unit between cache and RAM is *cache line*, 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).

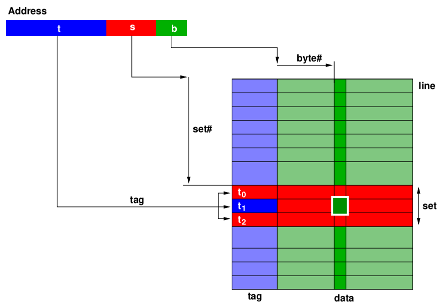
Cache Access



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

Physically indexed: looked up by *physical address*

Cache Indexing



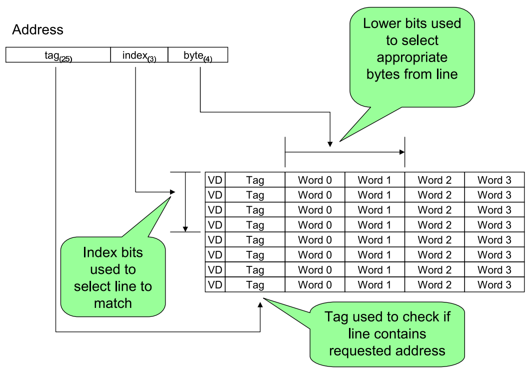
➜ The *tag* is used to distinguish lines of set...

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

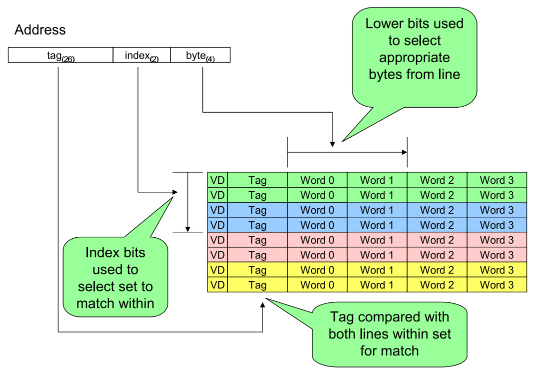
Cache Indexing

- 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 $n = 1 \dots 5$.
  - $n = 1$ is called *direct mapped*.
  - $n = \infty$ is called *fully associative*, unusual for CPU caches.
- Hashing must be simple (complex hardware is slow)
  ➜ use least-significant bits of address.

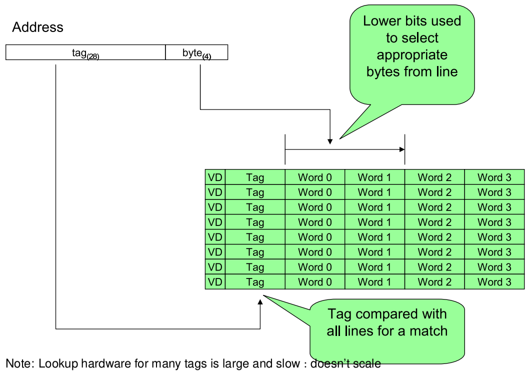
Cache Indexing: Direct Mapped



Cache Indexing: 2-Way Associative



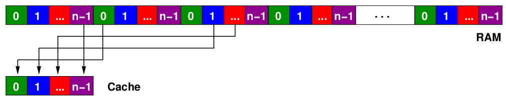
Cache Indexing: Fully Associative



Note: Lookup hardware for many tags is large and slow : doesn't scale

Cache Mapping

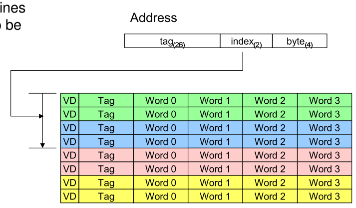
- Different memory locations map to same cache line:



  - All memory locations mapping to cache line # 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 corresponding to address is full, even though there may be unused lines in the cache.
    - ➜ Cannot occur on fully-associative cache

Cache Replacement Policy

- Indexing (using address) points to specific line set.
- On miss: all lines of set valid : *replace* existing line.
- Replacement strategy must be simple (hardware)
  - ➜ Note: dirty bit determines whether line needs to be written back
- Typical policies
  - ➜ Pseudo-LRU
  - ➜ FIFO
  - ➜ random
  - ➜ toss clean



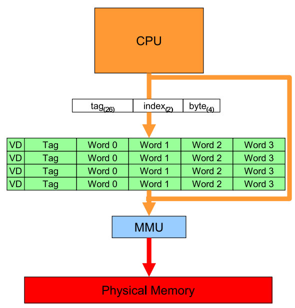
Cache Write Policy

- Treatment of store operations:
  - **write back:** Stores only update cache, memory is updated when dirty line is replaced (flushed).
    - ➜ clusters writes
    - ➜ memory is inconsistent with cache
  - **write through:** Stores update cache **and** memory immediately.
    - ➜ Memory is always consistent with cache.
- 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: VV Cache

Virtually-Indexed, Virtually-Tagged Cache

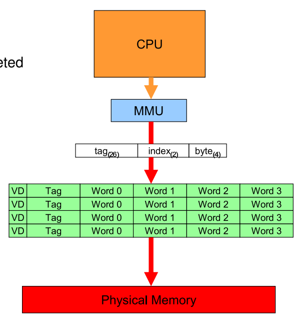
- Also called
  - ➜ virtually-addressed cache
- Also (misleadingly) called
  - ➜ virtual cache
  - ➜ virtual address cache
- uses virtual addresses only
  - ➜ can operate concurrently with MMU
- used on-core



Cache Types: VP Cache

Virtually-Indexed, Physically-Tagged Cache

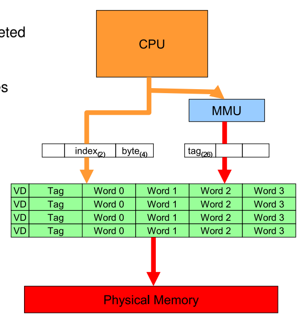
- virtual address for accessing line
- physical address for tagging
- needs address translation completed for retrieving data
- index concurrently with MMU, use MMU output for tag check
- typically used on-core



Cache Types: PP Cache

Physically Indexed, Physically-Tagged Cache

- only uses physical addresses
- needs address translation completed before begin of access
- typically used off-core
  - ➜ small physically-indexed caches do not require MMU lookup and can be used on-core
  - ➜ details later



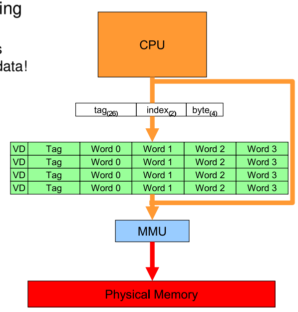
Cache Issues

- Caches are managed by hardware transparent to software
  → OS doesn't have to worry about them, ~~right?~~ Wrong!
- Software-visible cache effects:
  - performance
  - synonyms
    → can affect correctness!
  - homonyms
    → can affect correctness!

Virtually-Indexed Cache Issues

Homonyms

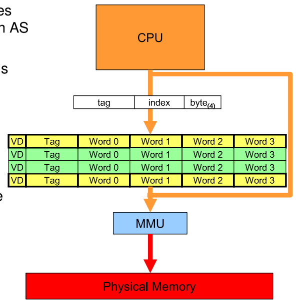
- Problem: VA used for indexing is context dependent
  → same VA refers to different PAs
  → tag does not uniquely identify data!
  → wrong data is accessed!
  → an issue for most OS!
- Homonym prevention:
  → flush cache on context switch
  → force non-overlapping address-space layout
  → *tag* VA with *address-space ID* (ASID)
    → makes VAs global
  → use physical tags



Virtually-Indexed Cache Issues

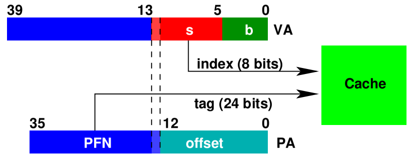
Synonyms (Aliases):

- 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



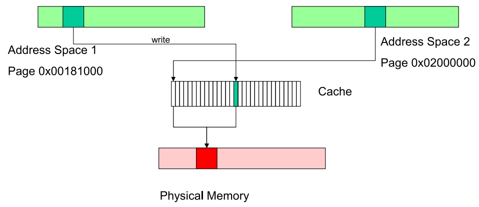
Example: MIPS R4x00 Synonyms

- ASID-tagged, on-chip L1 VP cache
  → 16kB cache with 32B lines, 2-way set associative
  → 4kB (base) page size



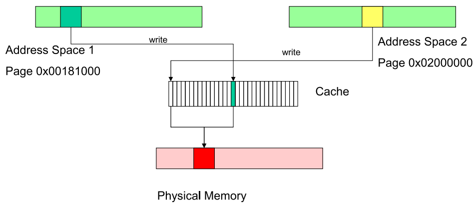
- Remember, location of data in cache determined by index
- Tag only confirms whether it's a hit!
- Synonym problem iff $VA_{12} \neq VA'_{12}$

Problem: Aliasing



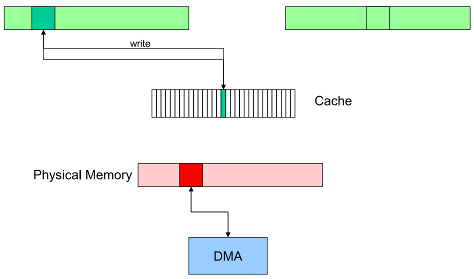
→ AS_1: $VA_{12} = 1$, AS_2: $VA_{12} = 0$

Problem: Re-Mapping



→ "Cache bomb": unaligned orphaned cache line can corrupt memory

Problem: DMA



→ CPU access inconsistent with device access

Synonyms...

- 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, ARMv4/5
- Used for I-caches on a number of architectures
  - ➜ Alpha, Pentium 4, ...

SUMMARY: VV CACHES WITH KEYS

- Add *address-space identifier* (ASID) as part of tag.
- On access compare with CPU's ASID register.
- Removes homonyms, creates 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 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

- 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 see [Wig03]

CACHE HIERARCHY

- Use a hierarchy of caches to balance memory accesses:
  - ➜ Small, fast, virtually indexed cache on top (L1 cache).
  - ➜ Large, slow, physically indexed cache at bottom (L2–L5).
- Each level reduces and clusters traffic.
- L1 cache tends to be 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 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, Itanium) 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-6 (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)

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

Not much growth in 20 years!

TLB coverage

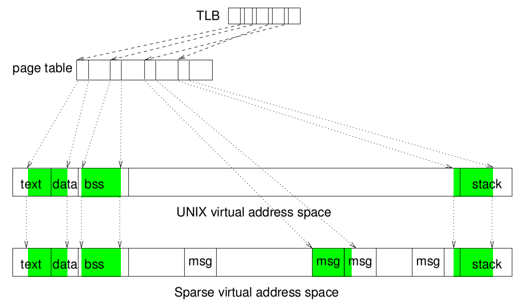
- Memory sizes are increasing.
- Number of TLB entries are more-or-less constant.
- Page sizes are growing *very* slowly.
  - ➜ Total amount of RAM mapped by TLB is not changing much.
  - ➜ Fraction of RAM mapped by TLB is shrinking dramatically.
- Modern architectures have very low *TLB coverage*.
- 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, 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.
- Superpages can be used to extend TLB coverage
  - ➜ however, difficult to manage in the OS

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].

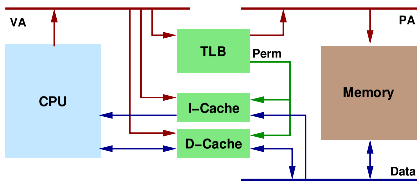
Case Study: Context Switches on ARM

ARM features:

- Virtually-indexed caches
- No address-space TLB tags
- Shared pages

The following is from [WTUH03]

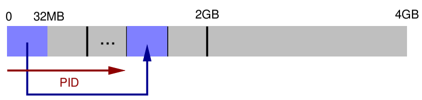
ARM Cache Architecture



Virtually-indexed caches:
- ➜ flush caches on context switch
- ➜ direct cost: 1k–18k cycles
- ➜ indirect cost: up to 54k cycles — up to 270μs at 200MHz!

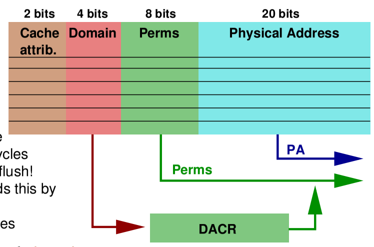
Permissions from TLB lookup:
- ➜ could avoid flushes if no address-space overlap
- ➜ unfeasible in standard OS

StrongARM PID Relocation



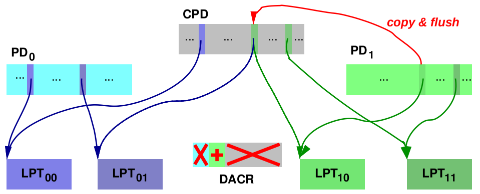
- Processor supports relocation of small address spaces:
  - ➜ lowest 32MB gets remapped into lower half of address space
  - ➜ mapping slot selected by *process-ID* (PID) register
  - ➜ re-mapping happens prior to cache or TLB lookup
- Re-mapped address spaces don't overlap:
  - ➜ no need to flush caches on address-space switch
  - ➜ used by Windows CE

ARM TLB



- TLB has no PID tag
  - ➜ must *flush* the TLB on context switches!
  - ➜ direct cost: 1 cycle
  - ➜ indirect cost: 3k cycles
  - ➜ TLB flush : cache flush!
  - ➜ Windows CE avoids this by
    - ➜ no protection!
    - ➜ max 32 processes
- Better: make use of *domains*
  - ➜ impose additional access restrictions

Fast Context Switch



PD$_0$ page table   Shared top-level page table   PD$_1$ page table

➜ Process switch replaces page table : must flush TLB
➜ Can avoid flush if keep top-level page table
➜ Use *domain access-control register* to share page table (&TLB) entries
➜ Flush TLB lazily when necessary (on collisions)

Fast Address-Space Switching

- Multiple address-spaces co-exist in top-level page table and TLB
- TLB and cache flushes only required on collisions:
  - ➜ minimised by use of PID relocation
  - ➜ may happen as result of
    - ➜ address-space overflow
    - ➜ conflicting mappings (`mmap()` with `MAP_FIXED`)
    - ➜ out of domains (only 16 available, some reserved)
- Domain recycling
  - ➜ preempt domain from some process
  - ➜ invalidate all CPD entries belonging to domain
  - ➜ flush TLB & caches if process had writable mappings
    - ➜ relatively infrequent, except with large number of active processes

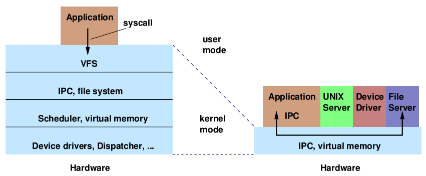
Implementation: Two Systems

Linux: monolithic open-source OS

➜ low to medium context-switching rates (dependent on application)

L4Ka::Pistachio: portable implementation of L4 microkernel

➜ high context-switching rates
➜ context-switching costs are extremely critical



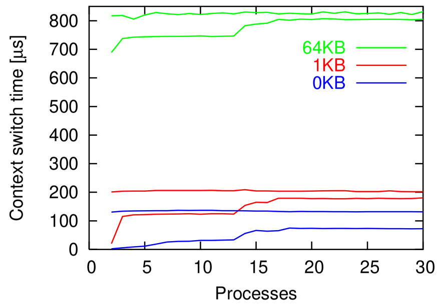
Linux Implementation

- Use PID-relocation for small (< 32MB) address spaces
  - ➜ includes code, stack and initial heap
  - ➜ extra heap allocated via `mmap()` in upper 2GB
  - ➜ shared libraries `mmap()`-ed in upper 2GB
- Large address spaces ($> 32MB$) not relocated
- Use domains for sharing top-level page table
  - ➜ 3 domains reserved by kernel, 13 available for user processes
  - ➜ use domain preemption if running out of domains

Performance: Linux lmbench Results

Benchmark	*original*	*fast*	*improvement*
lmbench hot potato latency [μs]			
`fcntl`	39	25	1.56±0.4
`fifo`	263	15.6	17±0
`pipe`	257	15.4	17±0
`unix`	511	30.7	16±0
lmbench hot potato bandwidth [MB/s]			
`pipe`	8.77	14.76	1.70±0.01
`unix`	12.31	12.94	1.05±0.00
lmbench process creation latency [μs]			
`fork`	4061	3650	1.10±0.00
`exec`	4321	3980	1.08±0.01
`shell`	54533	51726	1.05±0.01

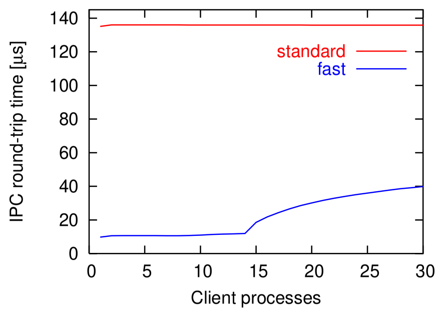
Linux lat_ctx Context Switch Costs



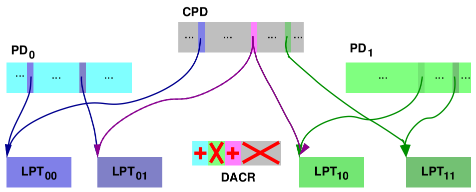
L4 Implementation

- All mappings fully under user control
  - ➜ no specific steps required to avoid collisions
- Use domains for sharing top-level page table
  - ➜ 1 domain reserved by kernel, 15 available for user processes
  - ➜ use domain preemption if running out of domains
  - ➜ keep track of domain use since last flush
    - ➜ if not used since last flush, domain is *clean*
    - ➜ clean domains can be preempted without cache flush

L4 Message-Passing Performance

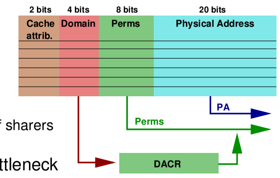


SHARED PAGES



- Shared (rather than conflicting) mappings:
  - ➜ use separate domain for shared mappings
  - ➜ use this to tag shared page table entries
  - ➜ can keep shared mappings valid across context switch

SHARED TLB ENTRIES

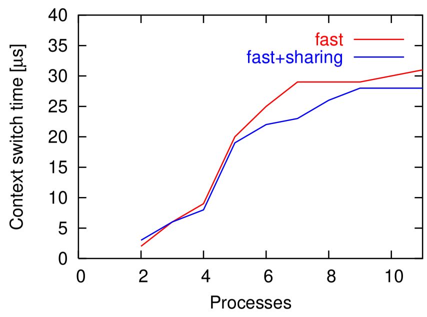


- Domains are TLB tags
  - ➜ sharing page table entries
    : sharing TLB entries
  - ➜ Single TLB entry for shared page, irrespective of number of sharers
- TLB coverage is a major bottleneck
- TLB entry sharing
  - ➜ reduces the consumption of TLB entries
  - ➜ reduces competition for TLB entries between address spaces
  - ➜ reduces indirect context-switching costs

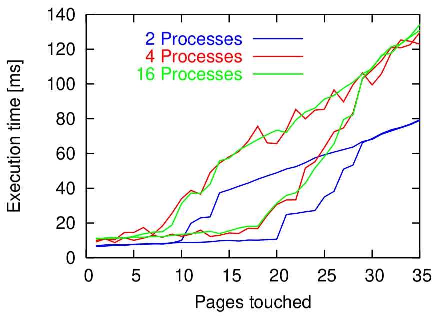
IMPLEMENTATION ON LINUX

- Transparent to user code
  - ➜ allocate separate domain for each shared region
- For shared libraries use shared domain
  - ➜ text segment mapped to *preferred link address* in upper 2GB
  - ➜ private linking as fall-back if address already in use
  - ➜ data segment relocated to process-specific region (lower 32MB)
- Some tricks required to make this work
  - ➜ see paper for details

LINUX `LAT_CTX` CONTEXT SWITCH COSTS



TAKING IT TO THE EXTREME



"Extreme" benchmark maximises context switching and sharing

CONCLUSIONS

- Fast context switching using domains is a clear winner
  - ➜ orders of magnitude reduction in context-switching overhead on SA-1100
  - ➜ never seen a case where "fast" kernel was slower than vanilla
  - ➜ similar improvements expected on XScale
- TLB entry sharing is of marginal benefit
  - ➜ ... except in extreme conditions
  - ➜ typical UNIX workloads don't exhibit high enough context switching rates
  - ➜ similar experience was made on Itanium
  - ➜ benefits may be more significant on microkernel
  - ➜ however, SGI has benchmarks which significantly benefit from TLB sharing

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.

[Wig03] Adam Wiggins. A survey on the interaction between caching, translation and protection. Technical Report UNSW-CSE-TR-0321, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Aug 2003.

[WTUH03] Adam Wiggins, Harvey Tuch, Volkmar Uhlig, and Gernot Heiser. Implementation of fast address-space switching and TLB sharing on the StrongARM processor. In *8th Asia-Pacific Comp. Syst. Arch. Conf*, Aizu-Wakamatsu City, Japan, Sep 2003. Springer Verlag.