CSE 490/590 Computer Architecture

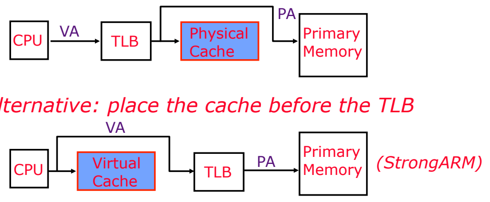
Virtual Memory II

Steve Ko
Computer Sciences and Engineering
University at Buffalo

Last time…

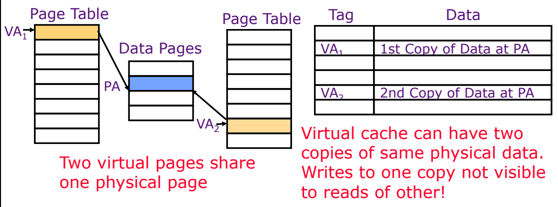
- Virtual memory organization
  - Linear page table
  - Hierarchical page table
- Page-table walk
  - Software or hardware
- TLB
  - Caches address translations

Virtual Address Caches



- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- *aliasing problems* due to the sharing of pages (-)
- maintaining cache coherence (-) (*see later in course*)

Aliasing in Virtual-Address Caches

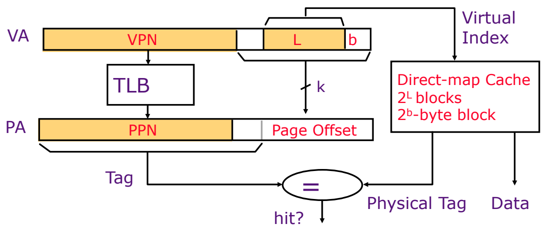

Page Table
VA1
Data Pages
PA
VA2
Page Table
Tag
Data
VA1
1st Copy of Data at PA
VA2
2nd Copy of Data at PA
Two virtual pages share one physical page
Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Disallow aliases to coexist in cache*

Software (i.e., OS) solution for direct-mapped cache

VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache (early SPARCs)

Concurrent Access to TLB & Cache

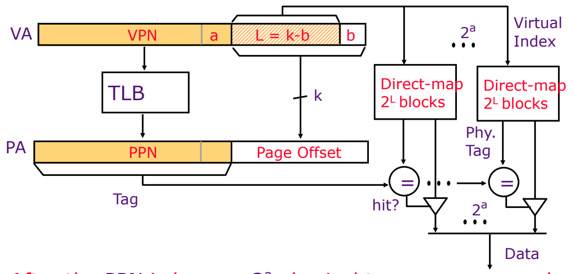


Index L is available without consulting the TLB
⇒ *cache and TLB accesses can begin simultaneously*
Tag comparison is made after both accesses are completed

Cases: $L + b = k, \quad L + b < k, \quad L + b > k$

Virtual-Index Physical-Tag Caches:

Associative Organization

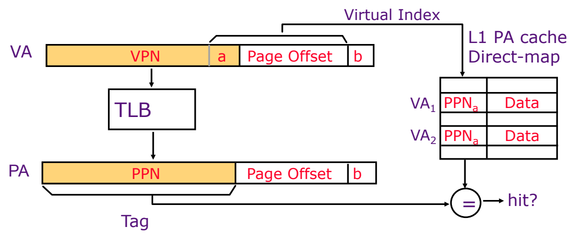


After the PPN is known, 2^a physical tags are compared

How does this scheme scale to larger caches?

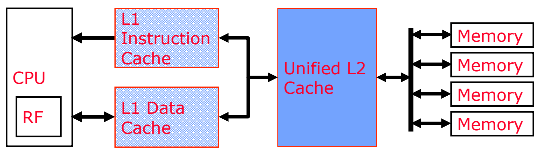
Concurrent Access to TLB & Large L1

The problem with L1 > Page size



Can VA_1 and VA_2 both map to PA ?

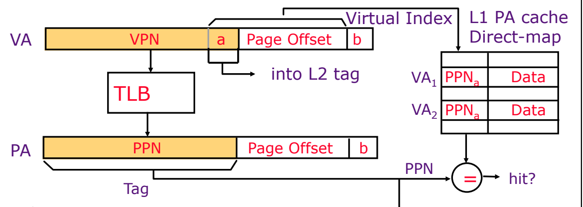
A solution via Second Level Cache



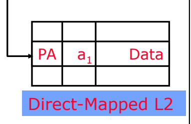
Usually a common L2 cache backs up both Instruction and Data L1 caches

L2 is "inclusive" of both Instruction and Data caches

Anti-Aliasing Using L2: *MIPS R10000*

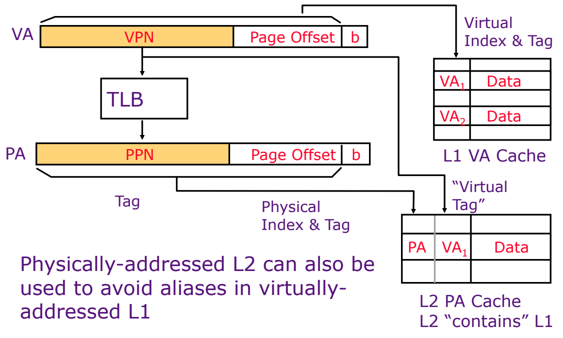


- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
- After VA2 is resolved to PA, a collision will be detected in L2.
- VA1 will be purged from L1 and L2, and VA2 will be loaded ⇒ *no aliasing !*



Virtually-Addressed L1:

Anti-Aliasing using L2



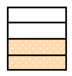
Physically-addressed L2 can also be used to avoid aliases in virtually-addressed L1

Page Fault Handler

- When the referenced page is not in DRAM:
  - The missing page is located (or created)
  - It is brought in from disk, and page table is updated
    Another job may be run on the CPU while the first job waits for the requested page to be read from disk
  - If no free pages are left, a page is swapped out
    Pseudo-LRU replacement policy
- Since it takes a long time to transfer a page (msecs), page faults are handled completely in software by the OS
  - Untranslated addressing mode is essential to allow kernel to access page tables

Swapping a Page of a Page Table

A PTE in primary memory contains primary or secondary memory addresses



A PTE in secondary memory contains *only* secondary memory addresses

⇒ a page of a PT can be swapped out only if none its PTE's point to pages in the primary memory

*Why?*____

Virtual Memory Use Today - 1

- Desktops/servers have full demand-paged virtual memory
  - Portability between machines with different memory sizes
  - Protection between multiple users or multiple tasks
  - Share small physical memory among active tasks
  - Simplifies implementation of some OS features
- Vector supercomputers have translation and protection but not demand-paging
- (Older Crays: base&bound, Japanese & Cray X1/X2: pages)
  - Don't waste expensive CPU time thrashing to disk (make jobs fit in memory)
  - Mostly run in batch mode (run set of jobs that fits in memory)
  - Difficult to implement restartable vector instructions

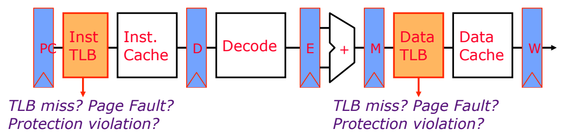
Virtual Memory Use Today - 2

- Most embedded processors and DSPs provide physical addressing only
  - Can't afford area/speed/power budget for virtual memory support
  - Often there is no secondary storage to swap to!
  - Programs custom written for particular memory configuration in product
  - Difficult to implement restartable instructions for exposed architectures

CSE 490/590 Administrivia

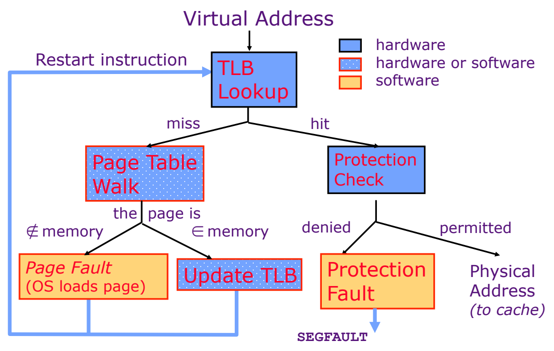
- Midterm on Friday, 3/4
- Project 1 deadline: Friday, 3/11
- Quiz 1 regrading → Jangyoung
- CSE machines are available for projects
  - Thin clients & SSH only for simulation
  - Linux & Windows machines @ 216 Bell for board

Address Translation in CPU Pipeline


PC
Inst TLB
Inst. Cache
D
Decode
E
+
M
Data TLB
Data Cache
W
TLB miss? Page Fault? Protection violation?
TLB miss? Page Fault? Protection violation?

- Software handlers need *restartable* exception on page fault or protection violation
- Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB
- Need mechanisms to cope with the additional latency of a TLB:
  - slow down the clock
  - pipeline the TLB and cache access
  - virtual address caches
  - parallel TLB/cache access

Address Translation: *putting it all together*

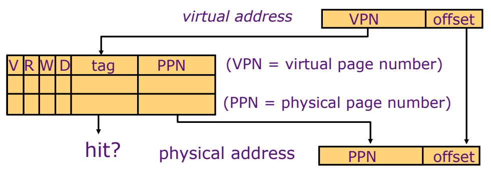


Translation Lookaside Buffers

Address translation is very expensive!
In a two-level page table, each reference becomes several memory accesses

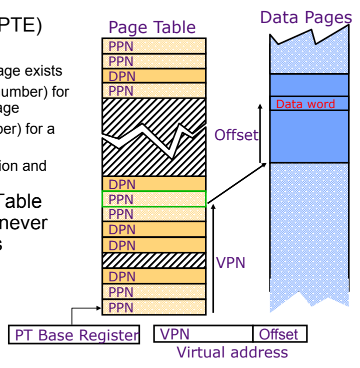
Solution: *Cache translations in TLB*

TLB hit	⇒ *Single Cycle Translation*
TLB miss	⇒ *Page-Table Walk to refill*

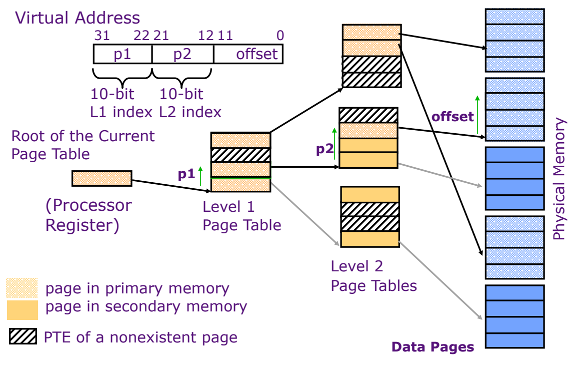


Linear Page Table

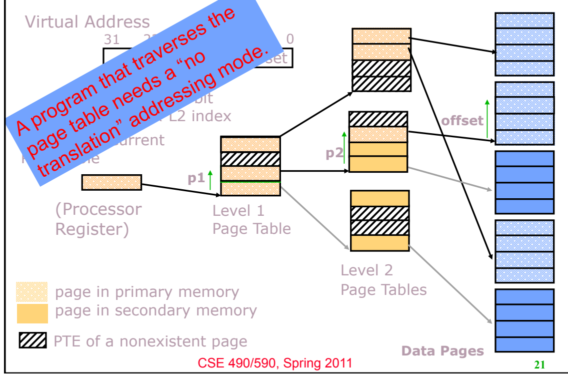
- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - PPN (physical page number) for a memory-resident page
  - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage
- OS sets the Page Table Base Register whenever active user process changes



Hierarchical Page Table



Hierarchical Page Table



A program that traverses the page table needs a “no translation” addressing mode.

Acknowledgements

- These slides heavily contain material developed and copyright by
  - Krste Asanovic (MIT/UCB)
  - David Patterson (UCB)
- And also by:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252