
Virtual Memory

Virtual Memory

- The games we play with addresses and the memory behind them

Address translation

- decouple the names of memory locations and their physical locations
- arrays that have space to grow without pre-allocating physical memory
- enable sharing of physical memory (different addresses for same objects)
 - shared libraries, fork, copy-on-write, etc

Specify memory + caching behavior

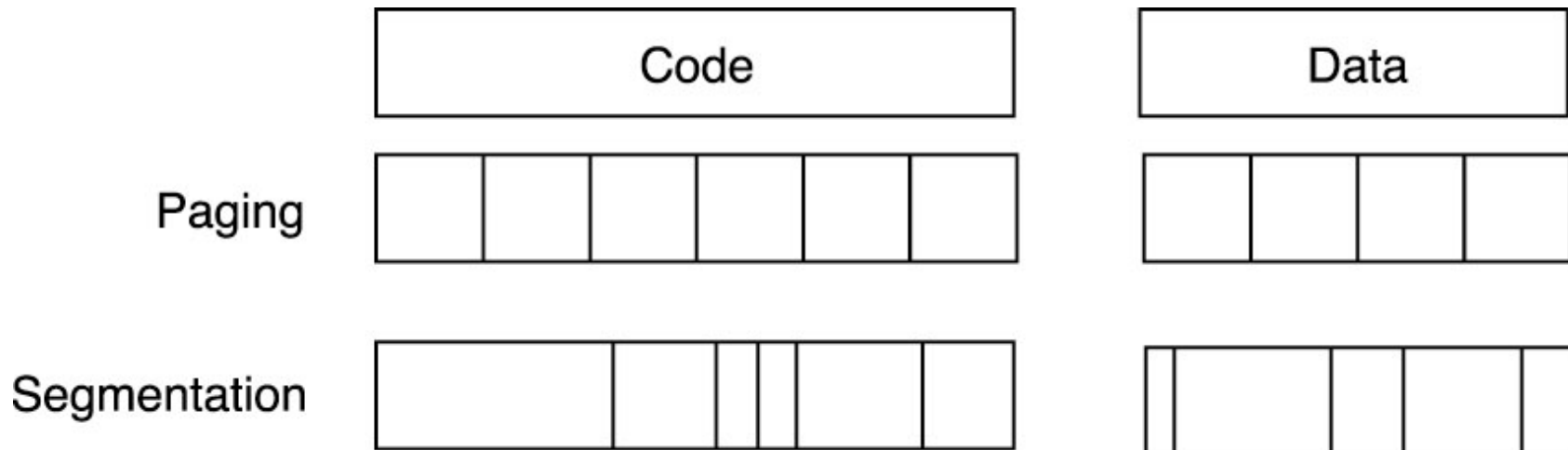
- protection bits (execute disable, read-only, write-only, etc)
- no caching (e.g., memory mapped I/O devices)
- write through (video memory)
- write back (standard)

Demand paging

- use disk (flash?) to provide more memory
- cache memory ops/sec: 1,000,000,000 (1 ns)
- dram memory ops/sec: 20,000,000 (50 ns)
- disk memory ops/sec: 100 (10 ms)
- demand paging to disk is only effective if you basically never use it
 - not really the additional level of memory hierarchy it is billed to be
 - but good for removing inactive jobs from DRAM

Paged vs Segmented Virtual Memory

- Paged Virtual Memory
 - memory divided into fixed sized pages
 - each page has a base physical address
- Segmented Virtual Memory
 - memory is divided into variable length segments
 - each segment has a base physical address + length



Virtual Memory

- The games we play with addresses and the memory behind them

Out of fashion



Segmentation

Paging

Address translation

- | | | |
|--|---|----|
| - decouple the <u>names</u> of memory locations and their physical locations | + | + |
| - arrays that have space to grow without pre-allocating physical memory | + | ++ |
| - enable sharing of physical memory (different addresses for same objects) | + | + |
| - shared libraries, fork, copy-on-write, etc | | |

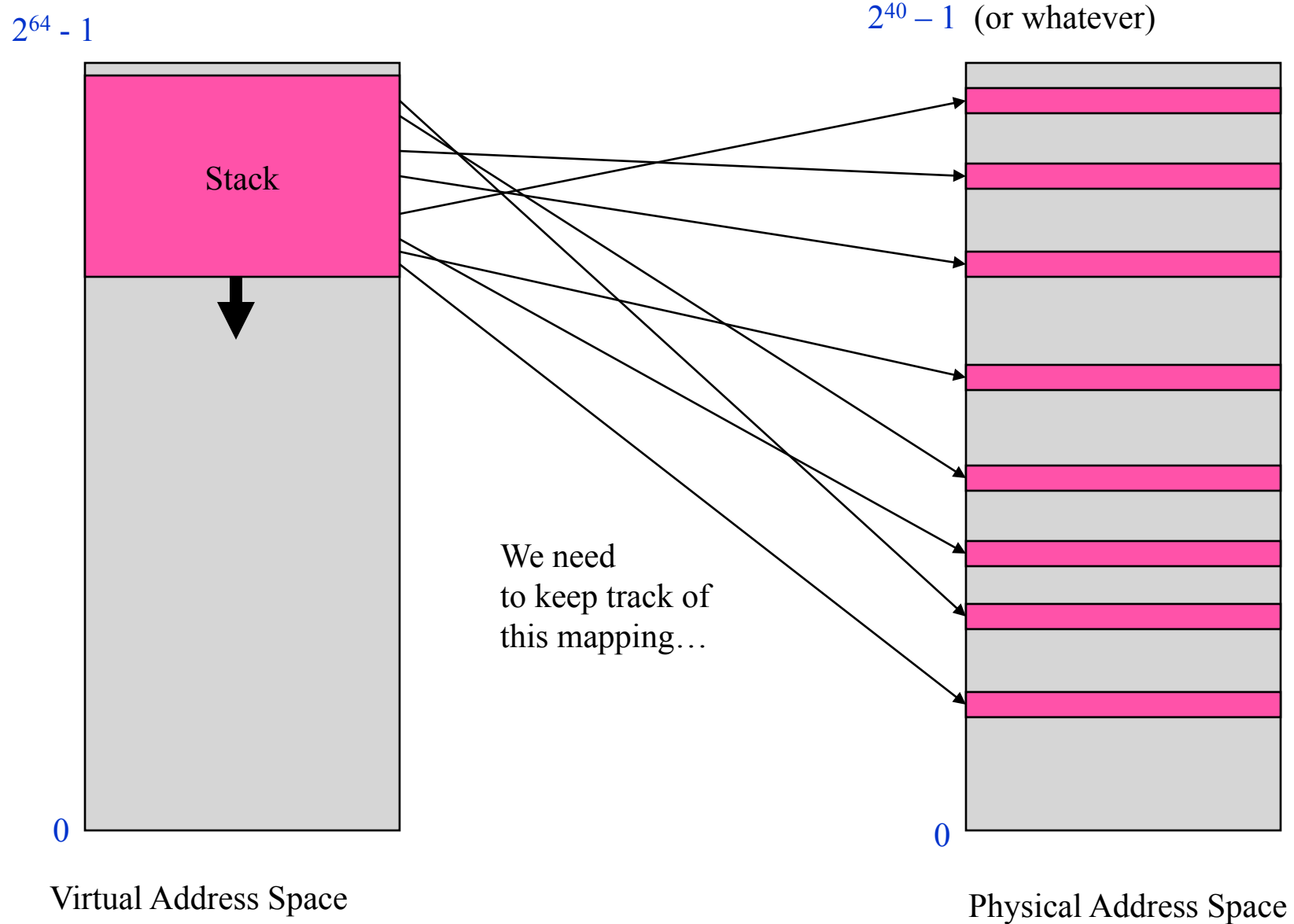
Specify memory + caching behavior

- | | | |
|---|----|---|
| - protection bits (execute disable, read-only, write-only, etc) | ++ | + |
| - no caching (e.g., memory mapped I/O devices) | ++ | + |
| - write through (video memory) | ++ | + |
| - write back (standard) | ++ | + |

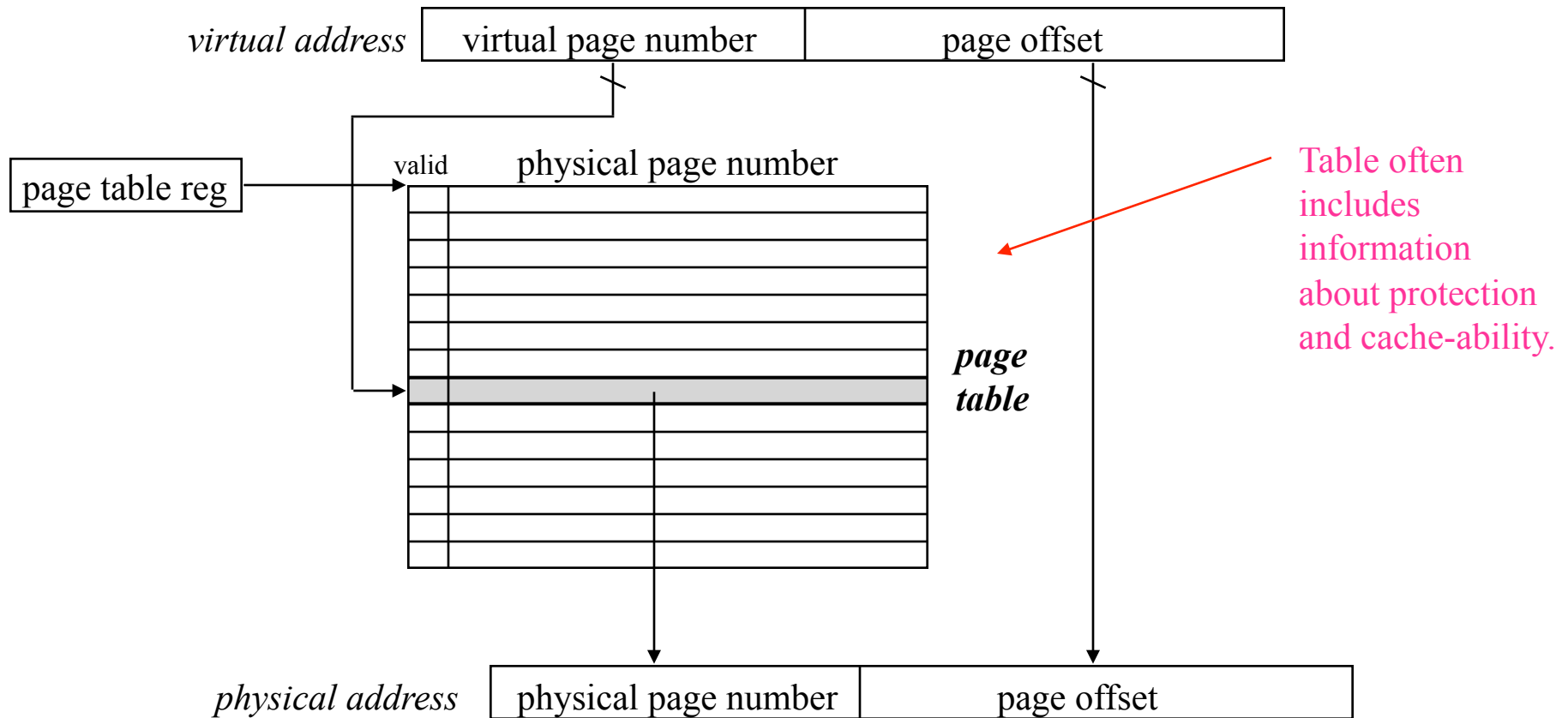
Demand paging

- | | | |
|--|---|----|
| - use disk (flash?) to provide more memory | + | ++ |
| - cache memory ops/sec: 1,000,000,000 (1 ns) | | |
| - dram memory ops/sec: 20,000,000 (50 ns) | | |
| - disk memory ops/sec: 100 (10 ms) | | |
| - demand paging to disk is only effective if you basically never use it | | |
| - not really the additional level of memory hierarchy it is billed to be | | |

Implementing Virtual Memory



Address translation via Paging



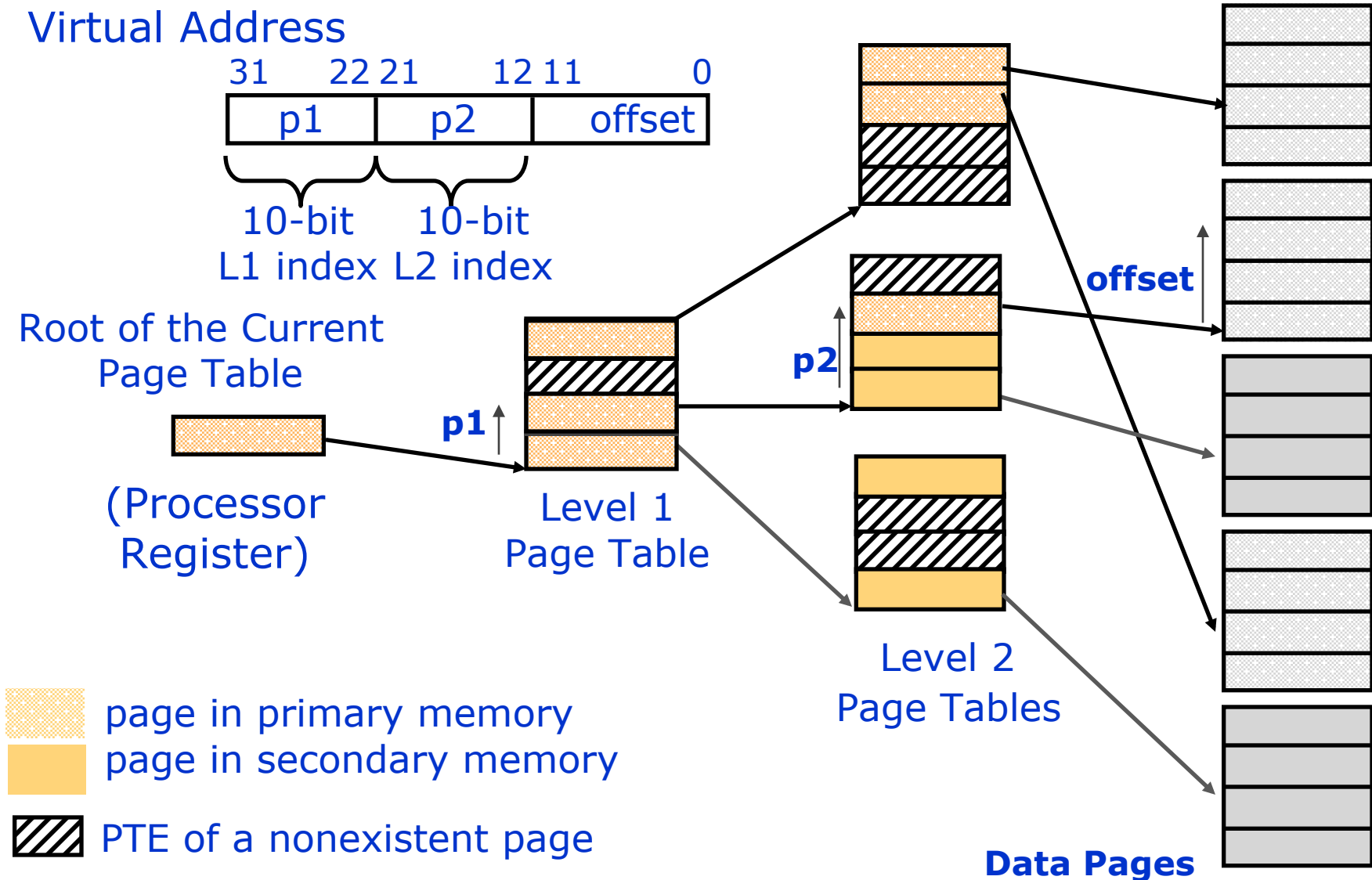
- all page mappings are in the page table, so hit/miss is determined solely by the valid bit (i.e., no tag)

Paging Implementation

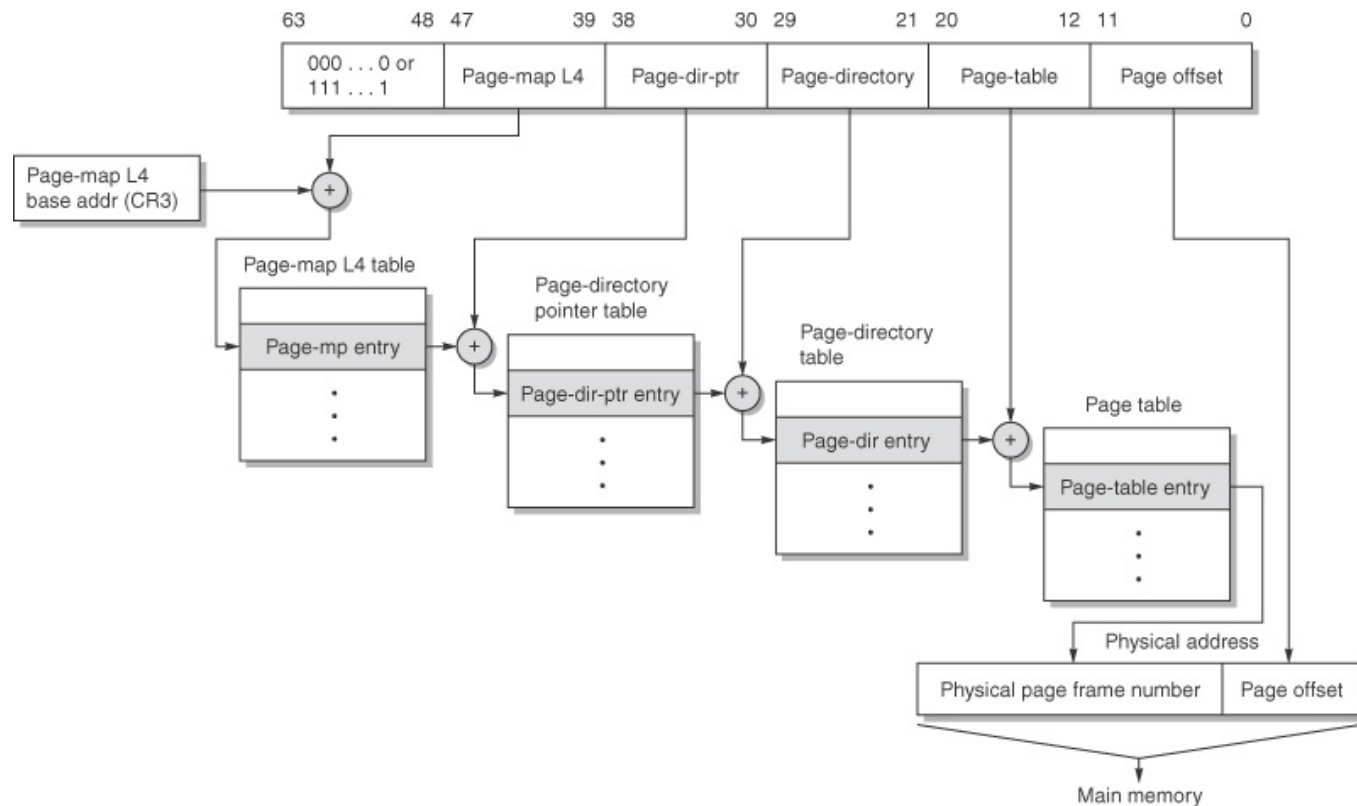
Two issues; somewhat orthogonal

- **specifying** the mapping with relatively little space
 - the larger the minimum page size, the lower the overhead
1 KB, 4 KB (very common), 32 KB, 1 MB, 4 MB ...
 - typically some sort of hierarchical page table (if in hardware)
or OS-dependent data structure (in software)
- making the mapping **fast**
 - TLB
 - small chip-resident cache of mappings from virtual to physical addresses
 - inverted page table (ala PowerPC)
 - fast memory-resident data structure for providing mappings

Hierarchical Page Table



Hierarchical Paging Implementation



© 2007 Elsevier, Inc. All rights reserved.

- depending on how the OS allocates addresses, there may be more efficient structures than the ones provided by the HW – however, a fixed structure allows the hardware to traverse the structure without the overhead of taking an exception
- a flat paging scheme takes space proportional to the size of the address space – e.g., $2^{64} / 2^{12} \times \sim 8 \text{ bytes per PTE} = 2^{55} \rightarrow \text{impractical}$

Paging Implementation

Two issues; somewhat orthogonal

- **specifying** the mapping with relatively little space

 - the larger the minimum page size, the lower the overhead

 - 1 KB, 4 KB (very common), 32 KB, 1 MB, 4 MB ...

 - typically some sort of hierarchical page table (if in hardware) or OS-dependent data structure (in software)

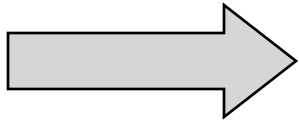
- making the mapping **fast**

 - TLB

 - small chip-resident cache of mappings from virtual to physical addresses

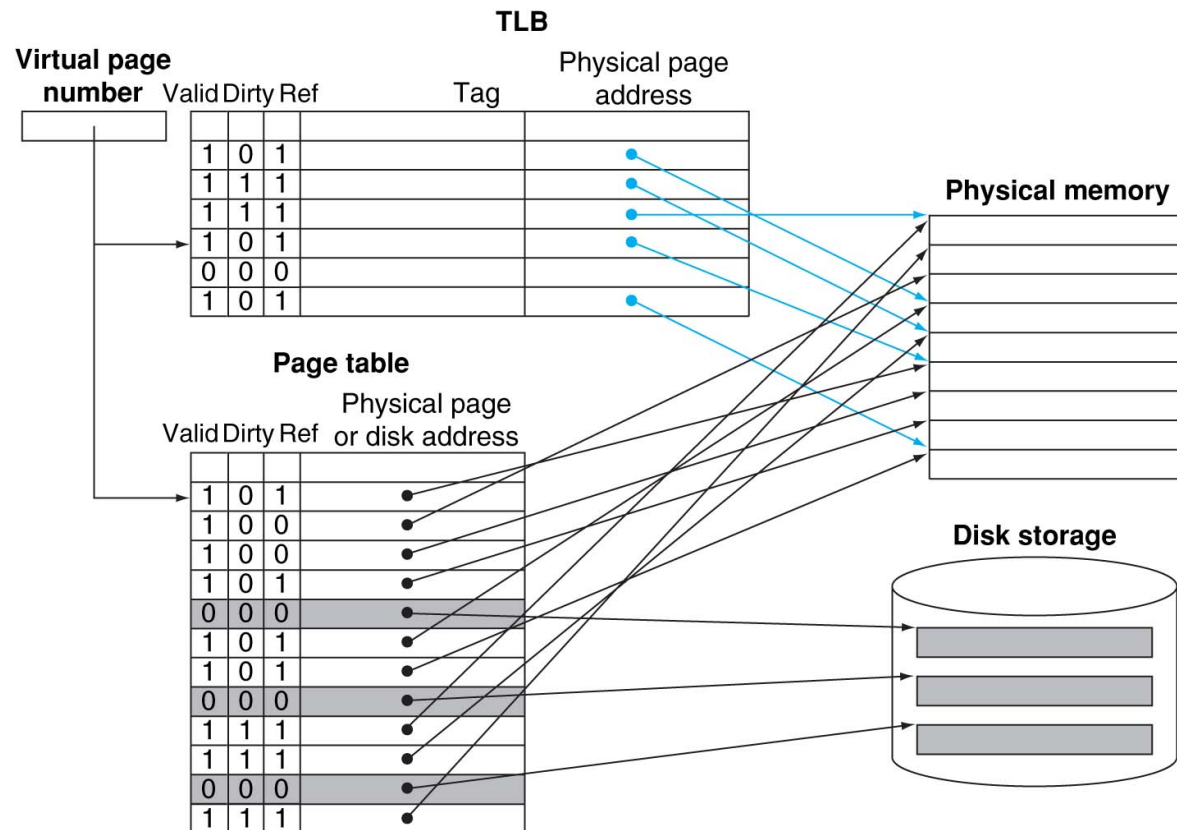
 - inverted page table (ala PowerPC)

 - fast memory-resident data structure for providing mappings

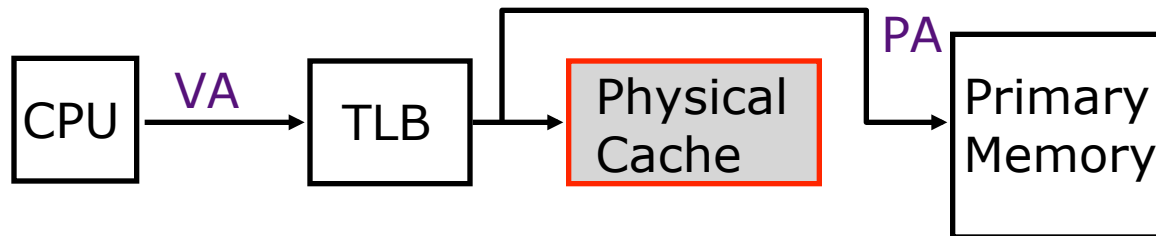


Translation Look-aside Buffer

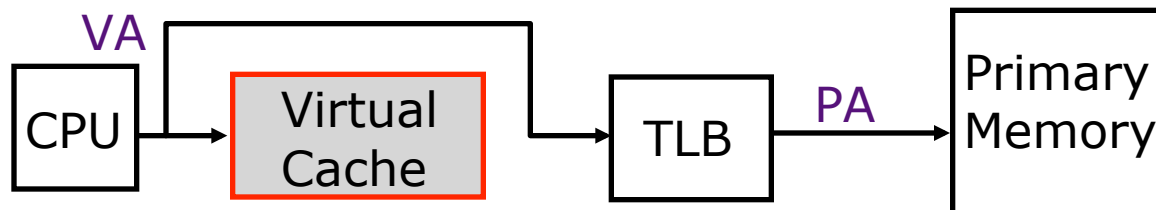
- A cache for address translations: translation lookaside buffer (TLB)



Virtually Addressed vs. Physically Addressed Caches

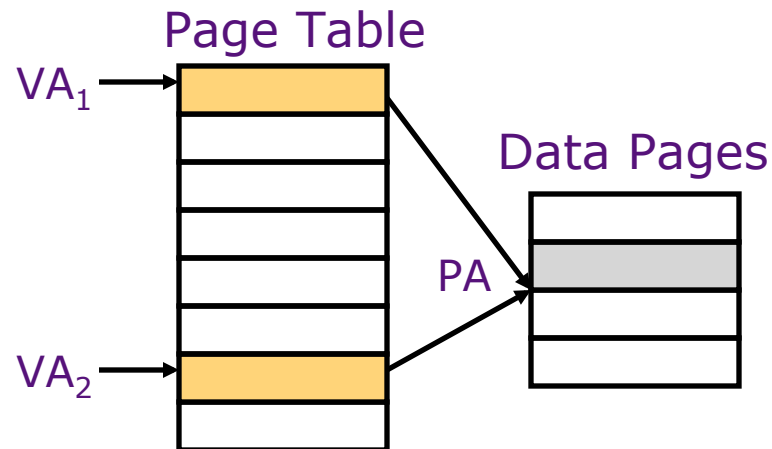


Alternative: place the cache before the TLB



- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch
(one approach: store address space identifiers (ASIDs) included in tags) (-)
- even then, *aliasing problems* due to the sharing of pages (-)

Aliasing in Virtually-Addressed Caches



Two virtual pages share one physical page

Tag	Data
VA ₁	1st Copy of Data at PA
VA ₂	2nd Copy of Data at PA

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*

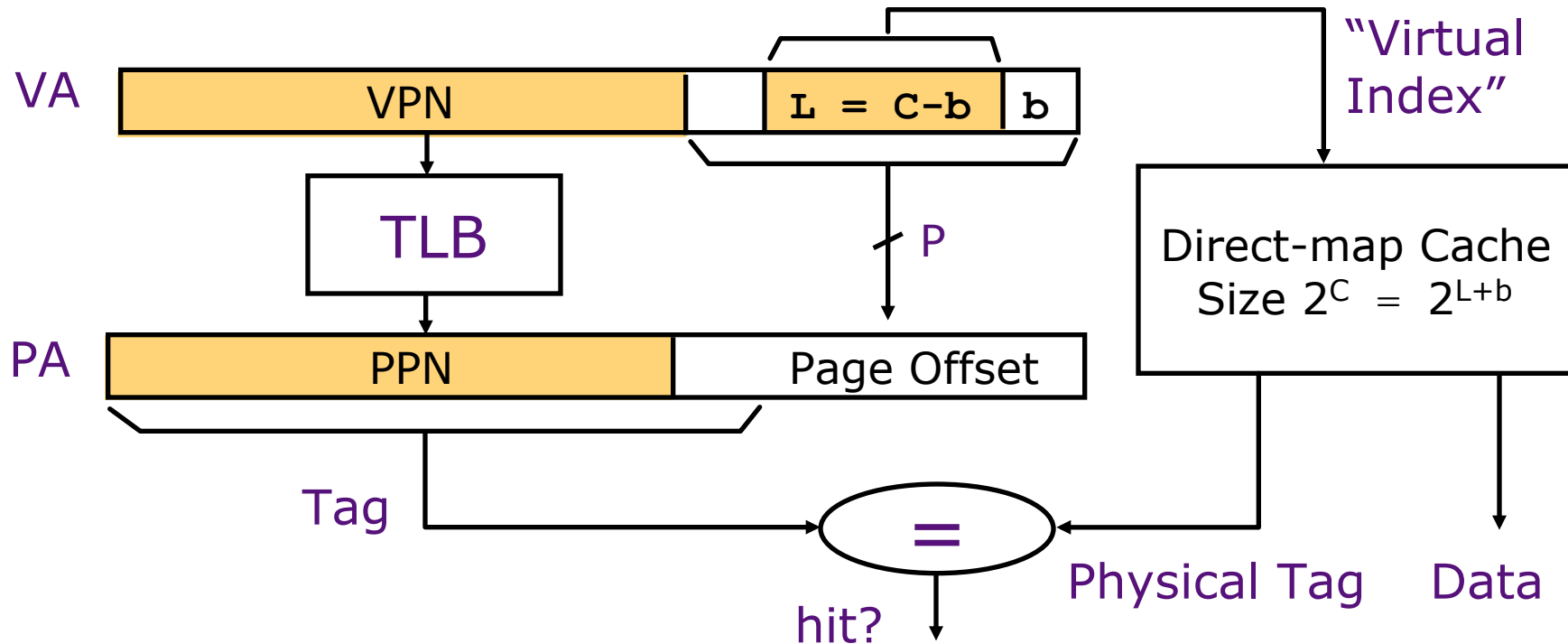
Software (i.e., OS) solution for direct-mapped cache

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)

Alternative: ensure that OS-based VA-PA mapping keeps those bits the same

Virtually Indexed, Physically Tagged Caches

key idea: page offset bits are not translated and thus can be presented to the cache immediately



Index L is available without consulting the TLB

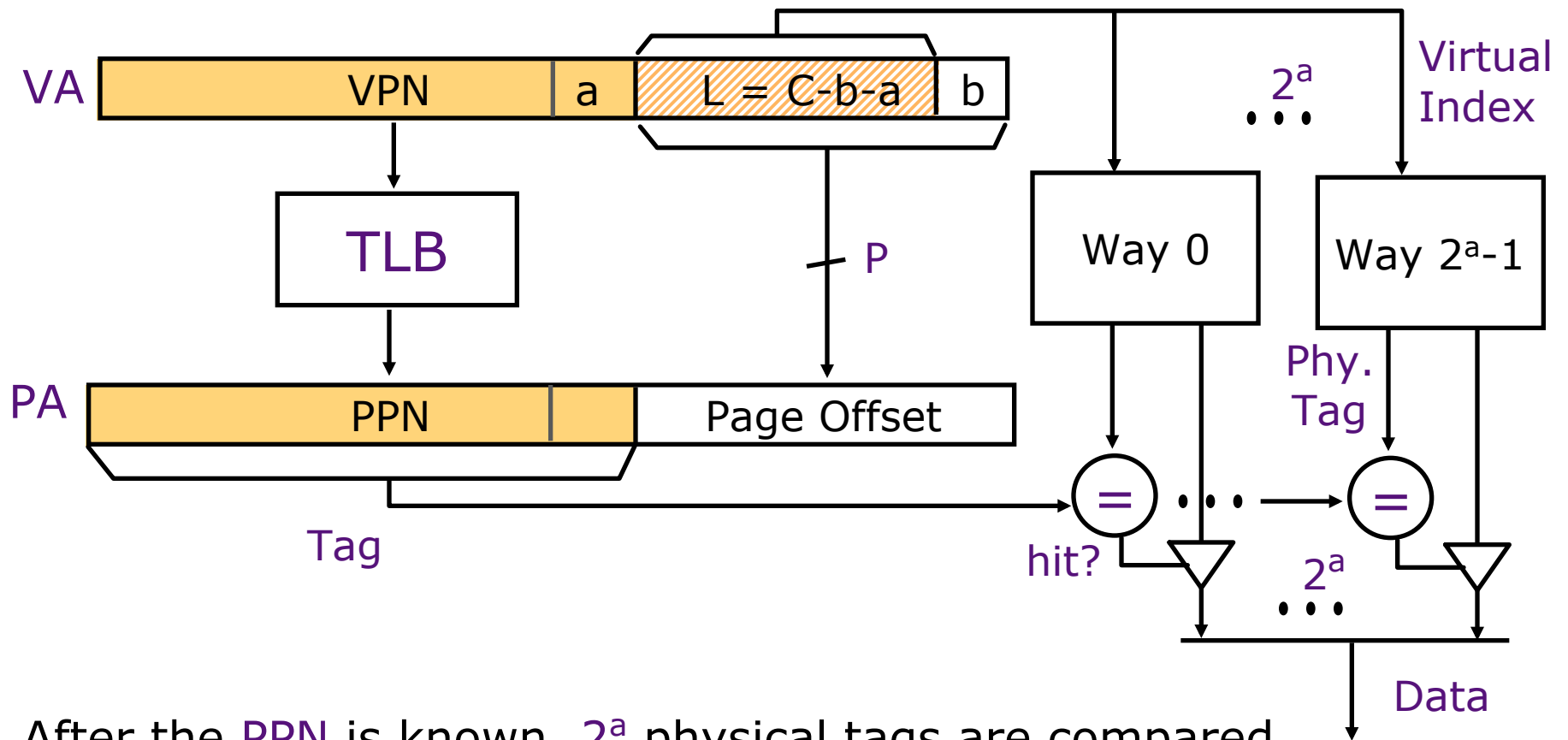
⇒ *cache and TLB accesses can begin simultaneously*

Tag comparison is made after both accesses are completed

Work if Cache Size ≤ Page Size (→ C ≤ P)

because then all the cache inputs do not need to be translated

Virtually-Indexed Physically-Tagged Caches: Using Associativity for Fun and Profit



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

Increasing the associativity of the cache reduces the number of address bits needed to index into the cache -

Work if: Cache Size / $2^a \leq$ Page Size

($\rightarrow C \leq P + A$)

Sanity Check: Core 2 Duo + Opteron

Core 2 Duo: 32 KB, 8-way set associative, page size $\geq 4\text{K}$

32 KB $\rightarrow C = 15$

8-way $\rightarrow A = 3$

4K $\rightarrow P \geq 12$

$C \leq P + A ?$

$\rightarrow 15 \leq 12 + 3 ?$

$\rightarrow \text{True}$

Sanity Check: Core 2 Duo + Opteron

Core 2 Duo: 32 KB, 8-way set associative, page size $\geq 4K$

32 KB $\rightarrow C = 15$

8-way $\rightarrow A = 3$

4K $\rightarrow P \geq 12$

$C \leq P + A ?$

$\rightarrow 15 \leq 12 + 3 ?$

\rightarrow True

Opteron: 64 KB, 2-way set associative, page size $\geq 4K$

64 KB $\rightarrow C = 16$

2-way $\rightarrow A = 1$

4K $\rightarrow P \geq 12$

$C \leq P + A ?$

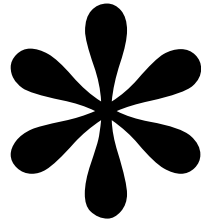
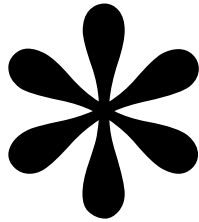
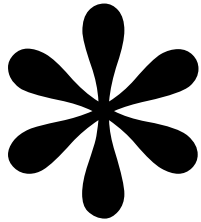
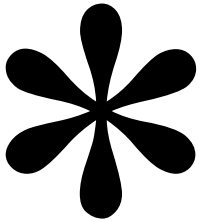
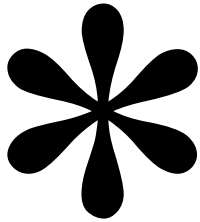
$\rightarrow 16 \leq 12 + 1 ?$

$\rightarrow 16 \leq 13 \rightarrow$ False

Solution:

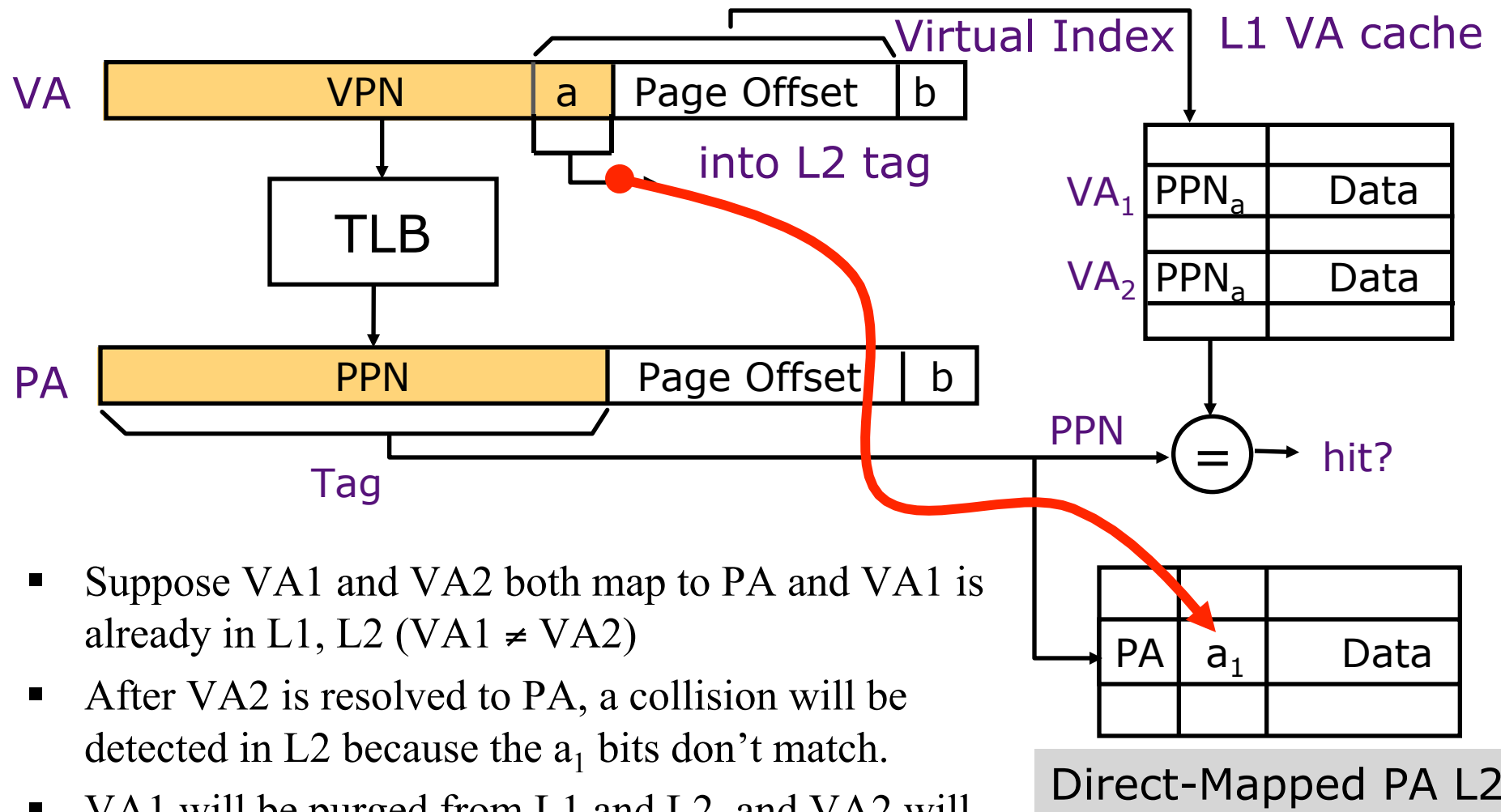
On cache miss, check possible locations of aliases in L1 and evict the alias, if it exists.

In this case, the Opteron has to check $2^3 = 8$ locations.



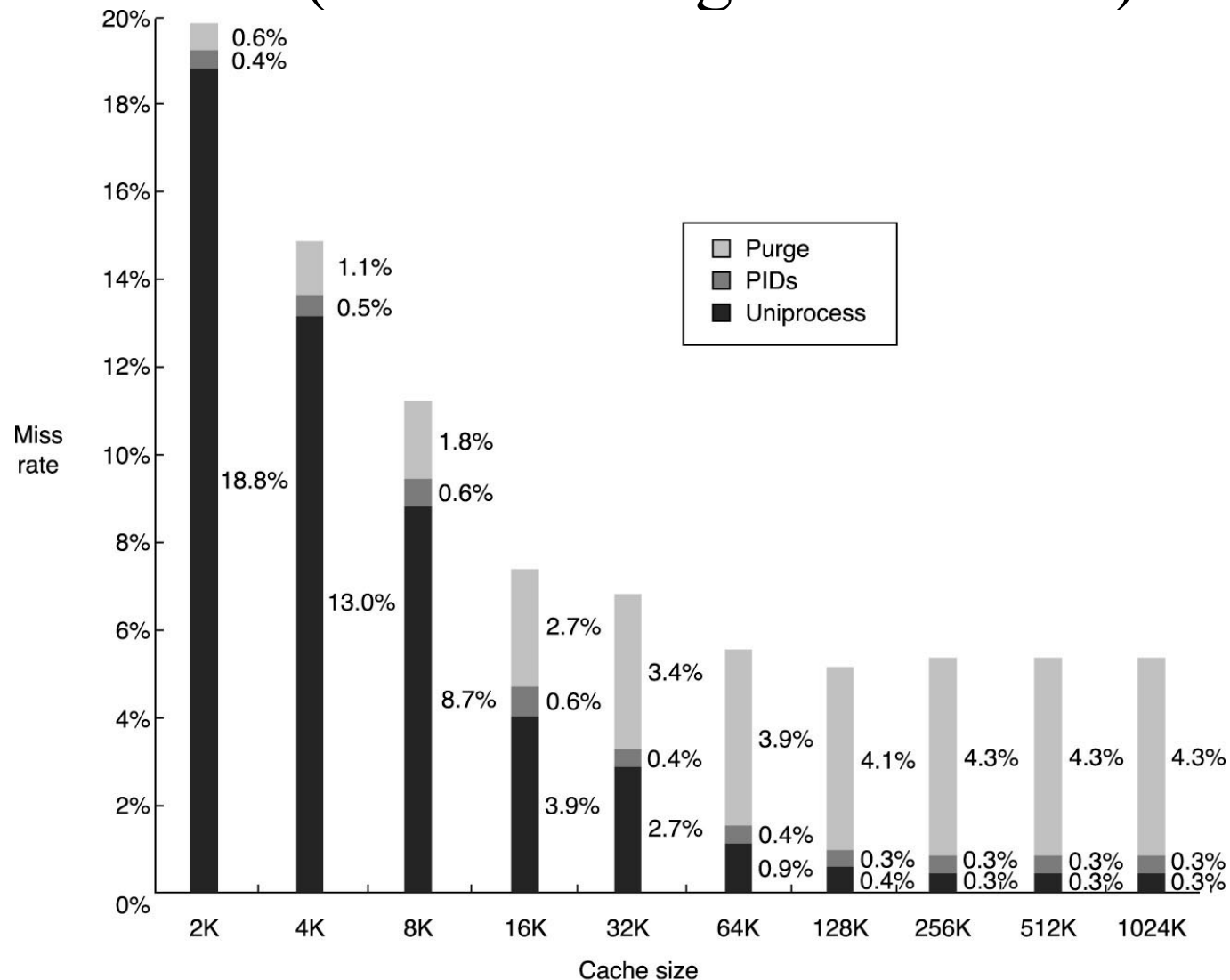
Anti-Aliasing Using Inclusive L2: *MIPS R10000-style*

Once again, ensure the invariant that only one copy of physical address is in virtually-addressed L1 cache at any one time. The physically-addressed L2, which includes contents of L1, contains the missing virtual address bits that identify the location of the item in the L1.



- 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 because the a₁ bits don't match.
- VA1 will be purged from L1 and L2, and VA2 will be loaded ⇒ *no aliasing* !

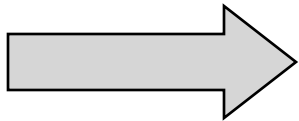
Why not purge to avoid aliases? Purging's impact on miss rate for context switching programs (data from Agarwal / 1987)



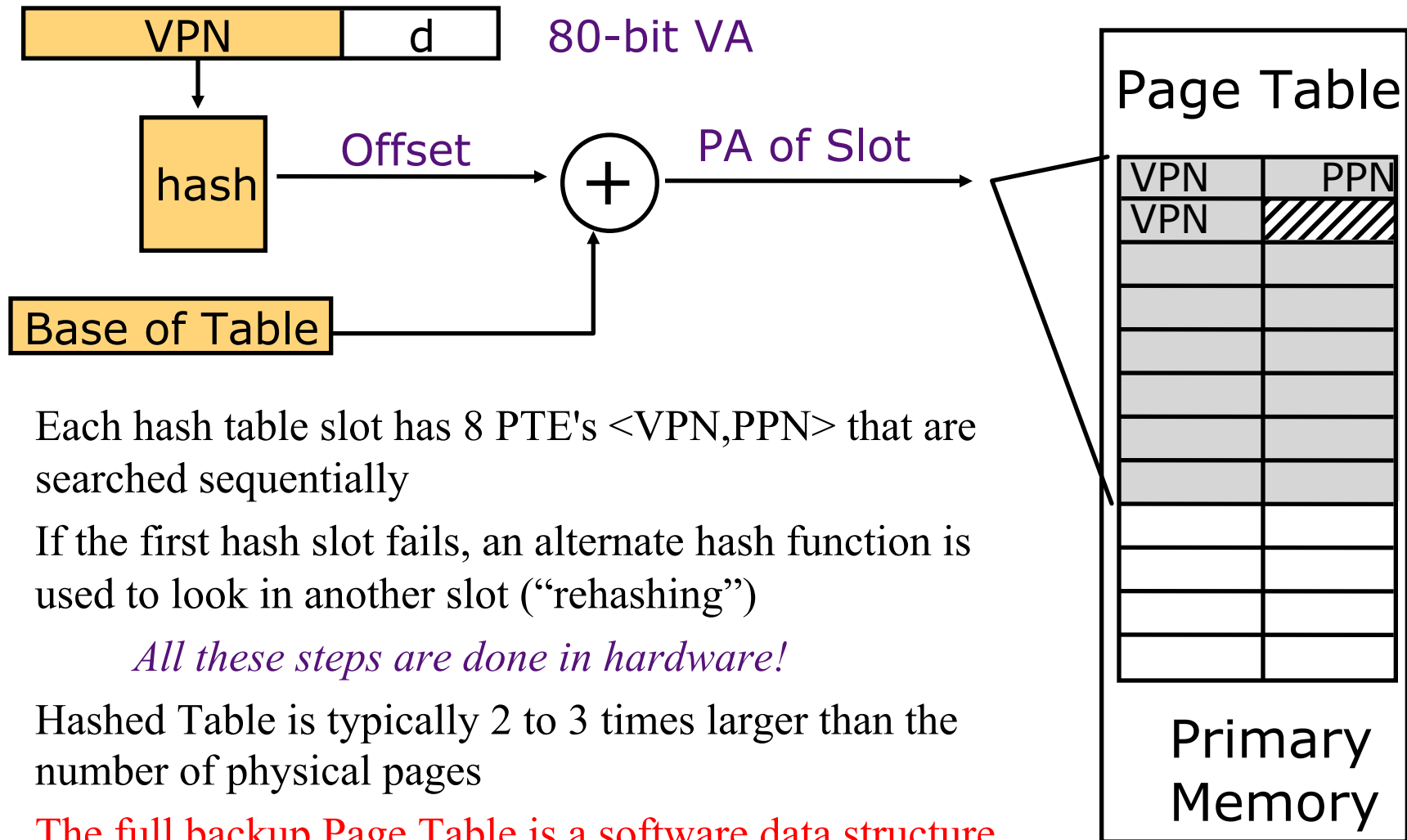
Paging Implementation

Two issues; somewhat orthogonal

- **specifying** the mapping with relatively little space
 - the larger the minimum page size, the lower the overhead
1 KB, 4 KB (very common), 32 KB, 1 MB, 4 MB ...
 - typically some sort of hierarchical page table (if in hardware)
or OS-dependent data structure (in software)
- making the mapping **fast**
 - TLB
 - small chip-resident cache of mappings from virtual to physical addresses
 - inverted page table (ala PowerPC)
 - fast memory-resident data structure for providing mappings



Power PC: Hashed Page Table



- Each hash table slot has 8 PTE's $\langle \text{VPN}, \text{PPN} \rangle$ that are searched sequentially
- If the first hash slot fails, an alternate hash function is used to look in another slot (“rehashing”)
All these steps are done in hardware!
- Hashed Table is typically 2 to 3 times larger than the number of physical pages
- **The full backup Page Table is a software data structure**