

18-447 Lecture 18:

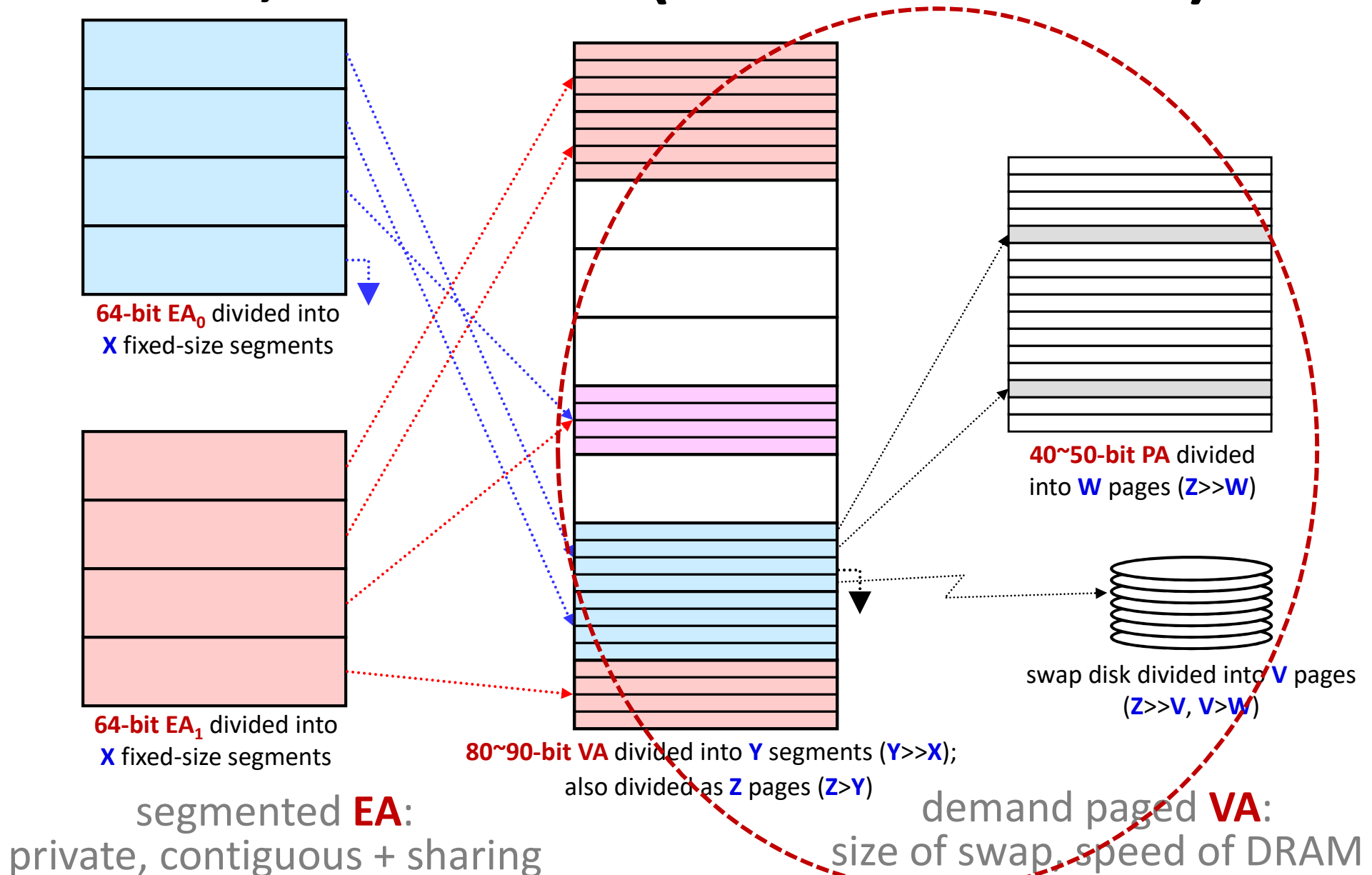
VM Mechanisms: Page Tables and TLBs

James C. Hoe
Department of ECE
Carnegie Mellon University

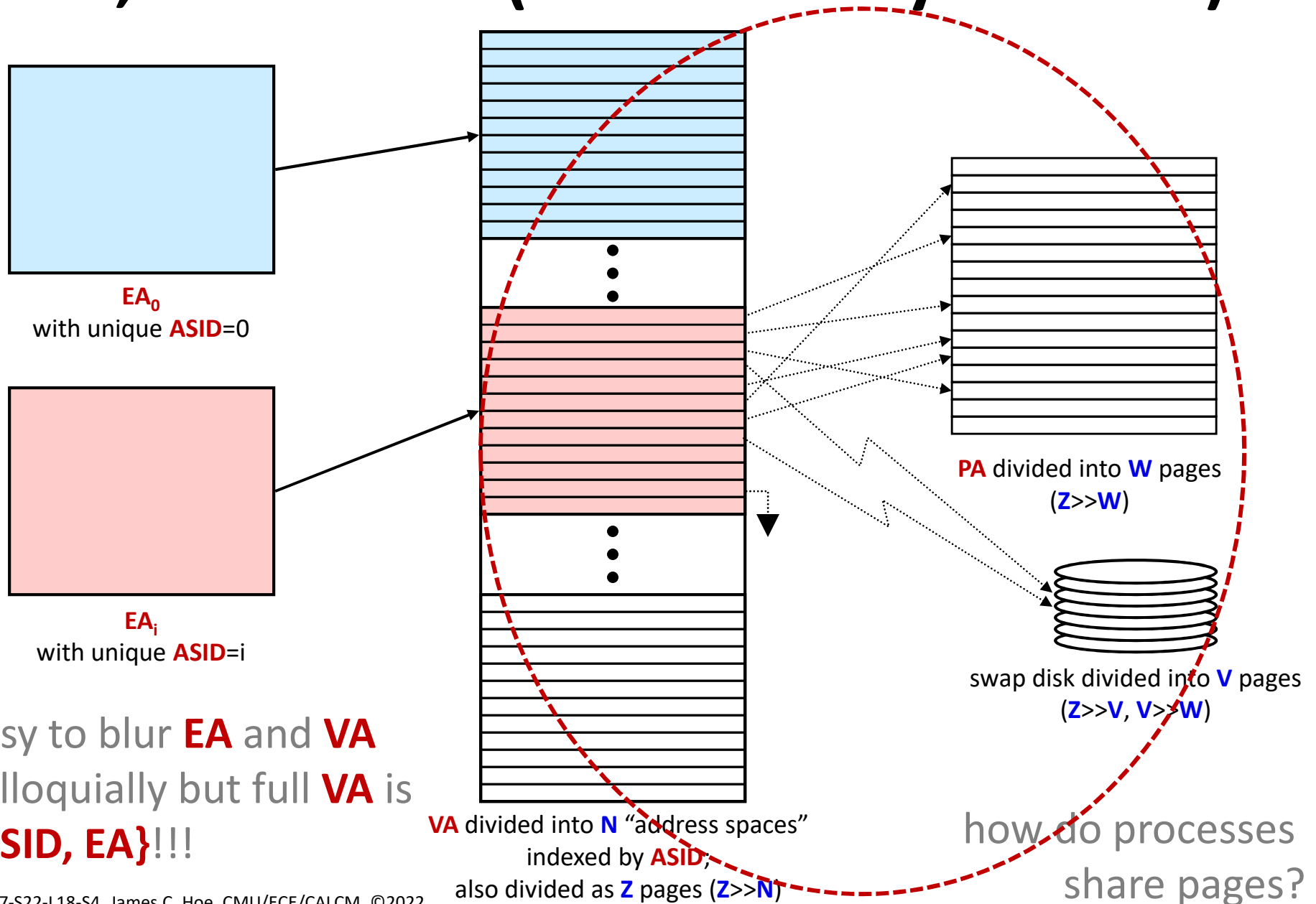
Housekeeping

- Your goal today
 - see the reality of page tables
 - delve into the many nuts and bolts of VM supports
- Notices
 - HW 5, due 4/4 (Handout #13)
 - Lab 3, due week 10
 - Midterm 2, Wed, 4/6
- **Required readings for L19**
 - “Virtual Memory in . . .” [Jacob&Mudge] (Canvas)
 - Meltdown→Mechanism (Wikipedia)

EA, VA and PA (IBM Power view)

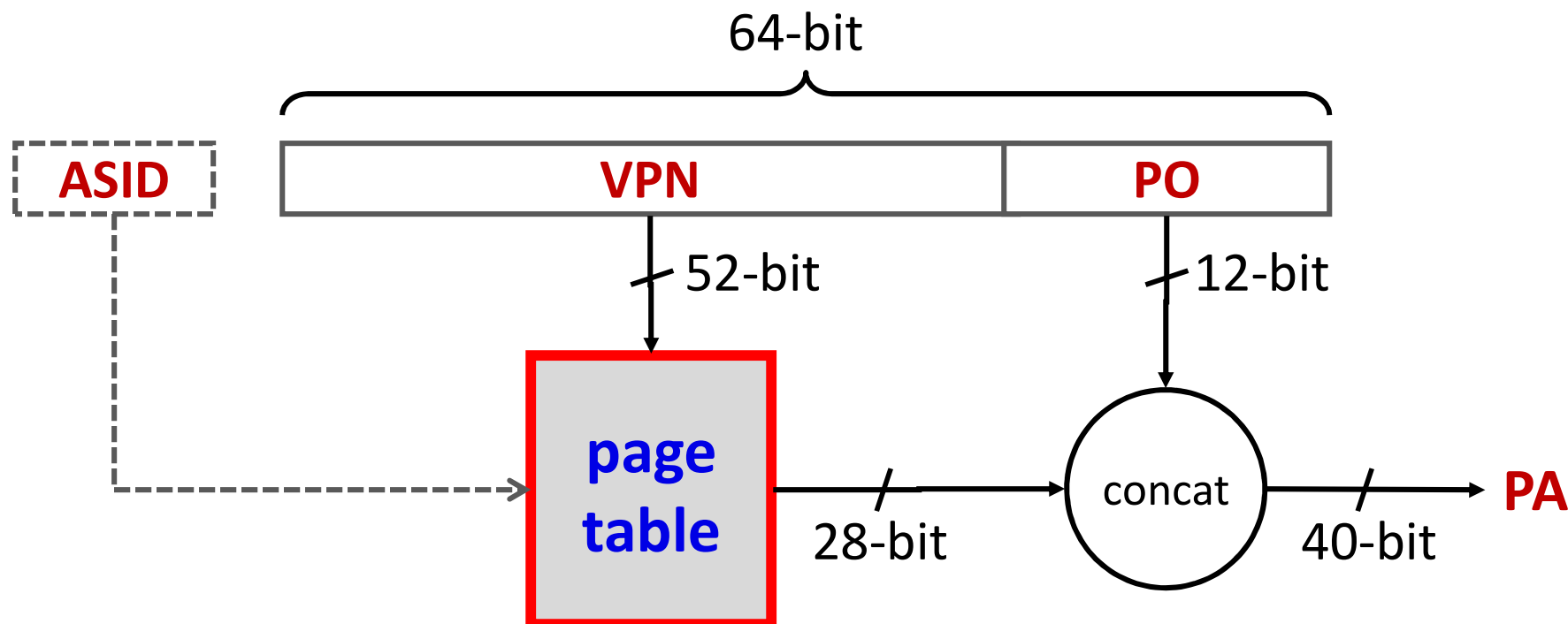


EA, VA and PA (almost everyone else)



Easy to blur **EA** and **VA**
colloquially but full **VA** is
{ASID, EA}!!!

Just one more thing: How large is the page table?



- A page table holds mapping from **VPN** to **PPN**
- Suppose 64-bit **VA** and 40-bit **PA**, how large is the page table? 2^{52} entries \times ~ 4 bytes $\approx 16 \times 10^{15}$ Bytes

And that is for just one process!!?

How large should it be?

- Don't need to track entire **VA** space
 - total allocated **VA** space is 2^{64} bytes x # processes, but most of which not backed by storage
 - can't use more memory locations than physically exist (DRAM and swap disk)
- A clever page table should scale linearly with physical storage size and not **VA** space size
- Table cannot be too convoluted
 - a page table must be “walkable” by HW
 - a page table is accessed not infrequently

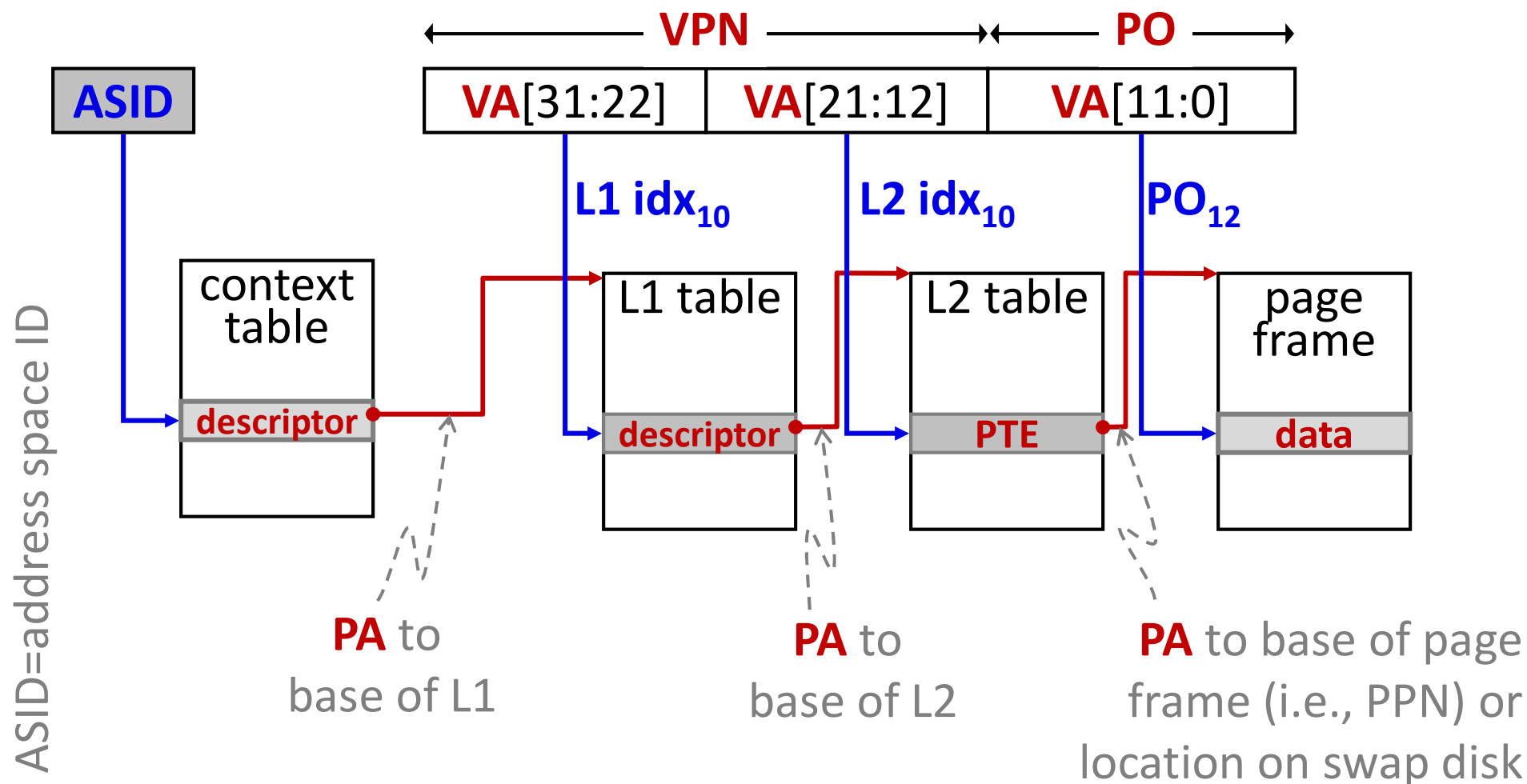
Two dominant schemes in use today:

hierarchical page table and *hashed page table*

*EA or VA
on this
slide?*

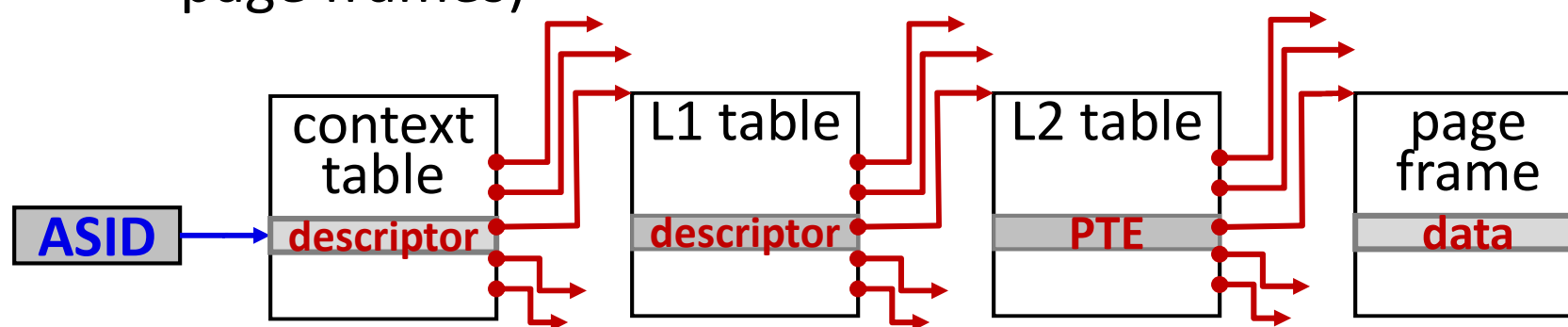
Hierarchical Page Table

- Hierarchical page table is a “tree” data structure in DRAM (and is cacheable)



Hierarchical page table is a tree

- For example on previous page
 - L1 table could have 1024 descendants (L2 tables)
 - each L2 table could have 1024 decedents (physical page frames)



- More levels can be used for larger **VA** space, but more memory references per translation
- Simple ratio btwn table sizes and page size (2, 1, 0.5) so tables demand-pageable btwn DRAM/disk

Hierarchical page table is a sparse tree

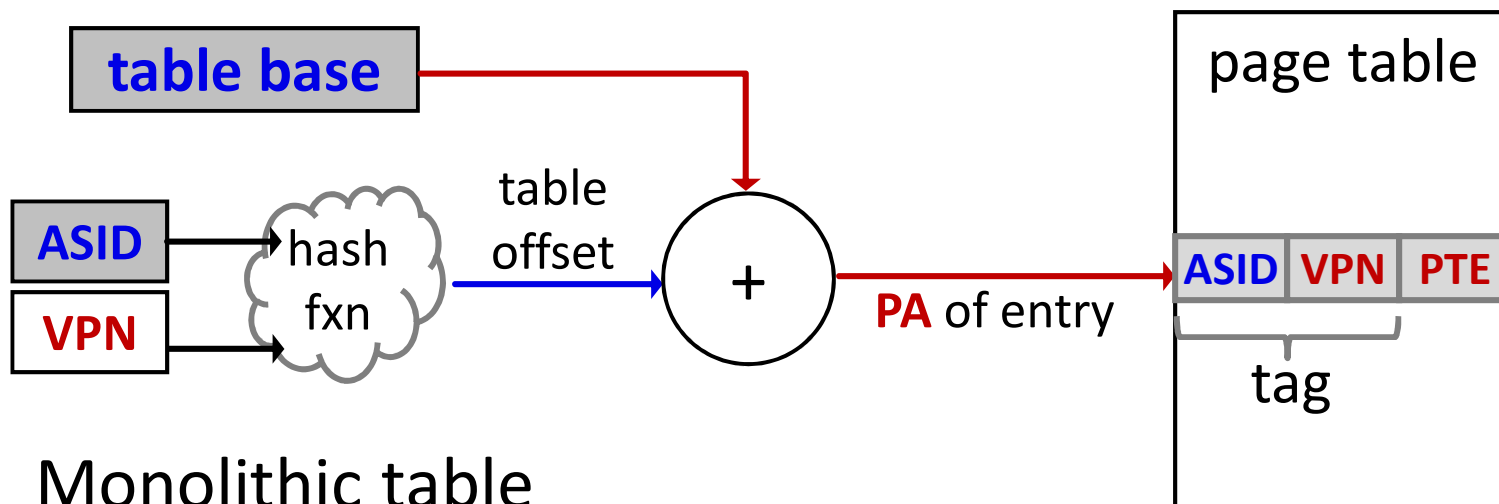
- Most virtual pages are not allocated; corresponding L2 entries point to null
- If a L2 table comprises entirely null pointers (no live descendants), itself does not need to exist; corresponding L1 entry points to null
- When more than 2 levels, an entire unused subtree is avoided
- Consider typical size ratio of **VA** to **PA**, the tree should be quite sparse for even the largest programs

How sparse?

Assume 32-bit **VA** with 4 MByte in use

- **Best Case:** one contiguous 4-MByte **VA** region aligned on 4 MByte boundaries
 - 1024 physical page frames used
 - needs 1 L2 table + 1 L1 table = 2 x 4KBytes
 - overhead $\approx \text{sizeof}(\text{PTE})$ per data page used, or 0.1%
- **Worst Case:** 1024 x 4-KByte **VA** regions; each is 4-MByte aligned
 - 1024 physical page frames used
 - needs 1K L2 tables (only 1 entry per L2 table used),
 - overhead $\approx \text{sizeof}(\text{L2 table})$ per data page, or 100%
- Locality says we should be closer to the best case

Hashed Page Table



- Monolithic table
 - indexed by hashing **VPN** and **ASID**,
 - e.g., $\text{index} = (\text{VPN} \oplus \text{ASID}) \% \text{table_size}$
- Entry “tagged” by **ASID** and **VPN** to detect collision
- Hashed table fast to access but not complete
 - lookup can fail even though page is valid
 - on a miss, consult a secondary complete table

How large is the hashed page table?

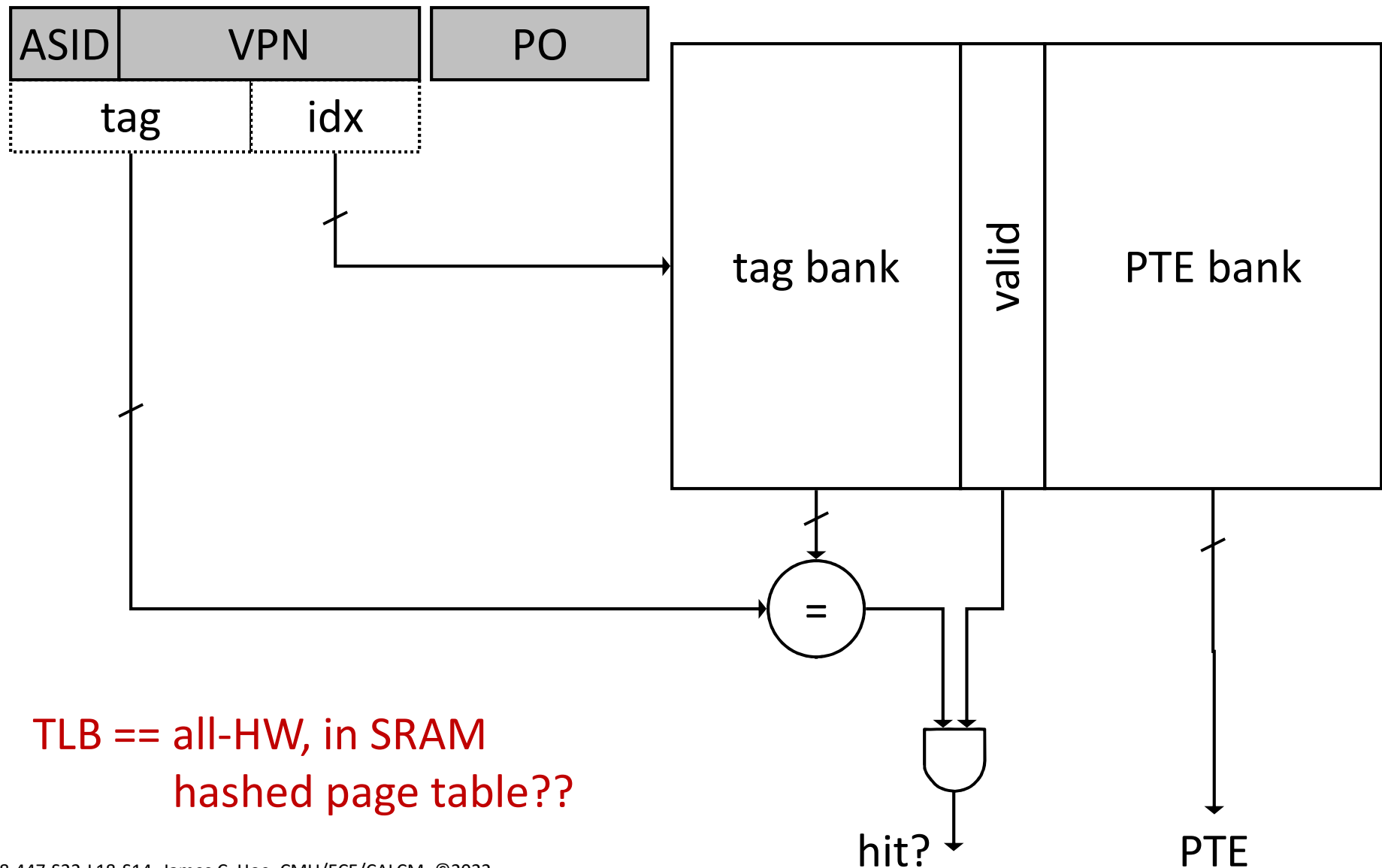
- Table size is an engineered choice, balancing storage overhead and hash collision
 - at least 1 entry per physical page
 - e.g., 1GB DRAM \Rightarrow 256K frames \Rightarrow 256K PTEs
 - typically some factor more to reduce collisions
- Original “inverted” page table
 - allocate 1 entry per physical page frame
 - use hashed index as **PPN** (a bit like direct-map . . .)
 - table entry contains only **VPN** tag

Viewed out of context, the table seems indexed by **PPN** and returns **VPN**, hence the misnomer

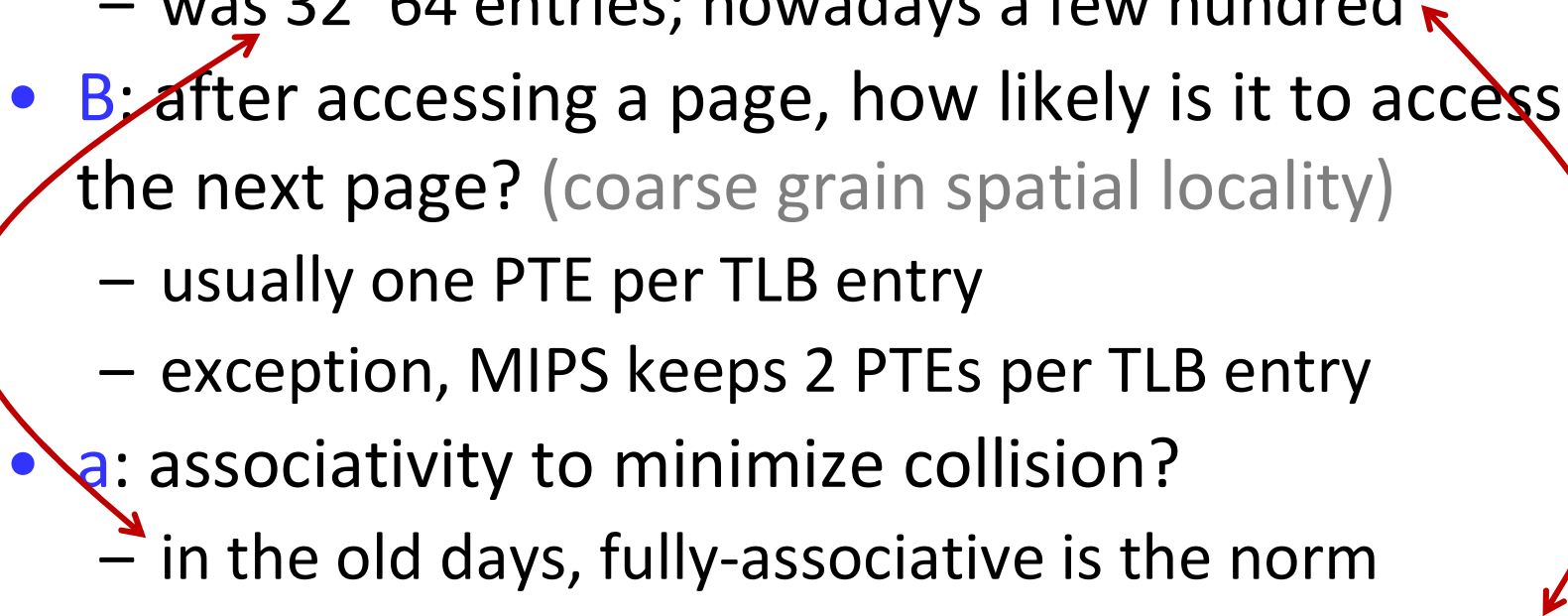
Translation Look-Aside Buffer (TLB)

- Every user memory access requires a translation
 - table walk requires its own memory accesses
 - can't possibly be walking the table on every access
- Keep a “cache” of recently used translations
- Similar “tagged” lookup structure as cache
 - same design considerations: A/B/C, replacement policy, split vs. unified, L1/L2, etc.
 - TLB entry:
 - tag:** address tag (from **VA**), **ASID**
 - PTE:** **PPN** & **protections**
 - misc:** valid, dirty, etc.

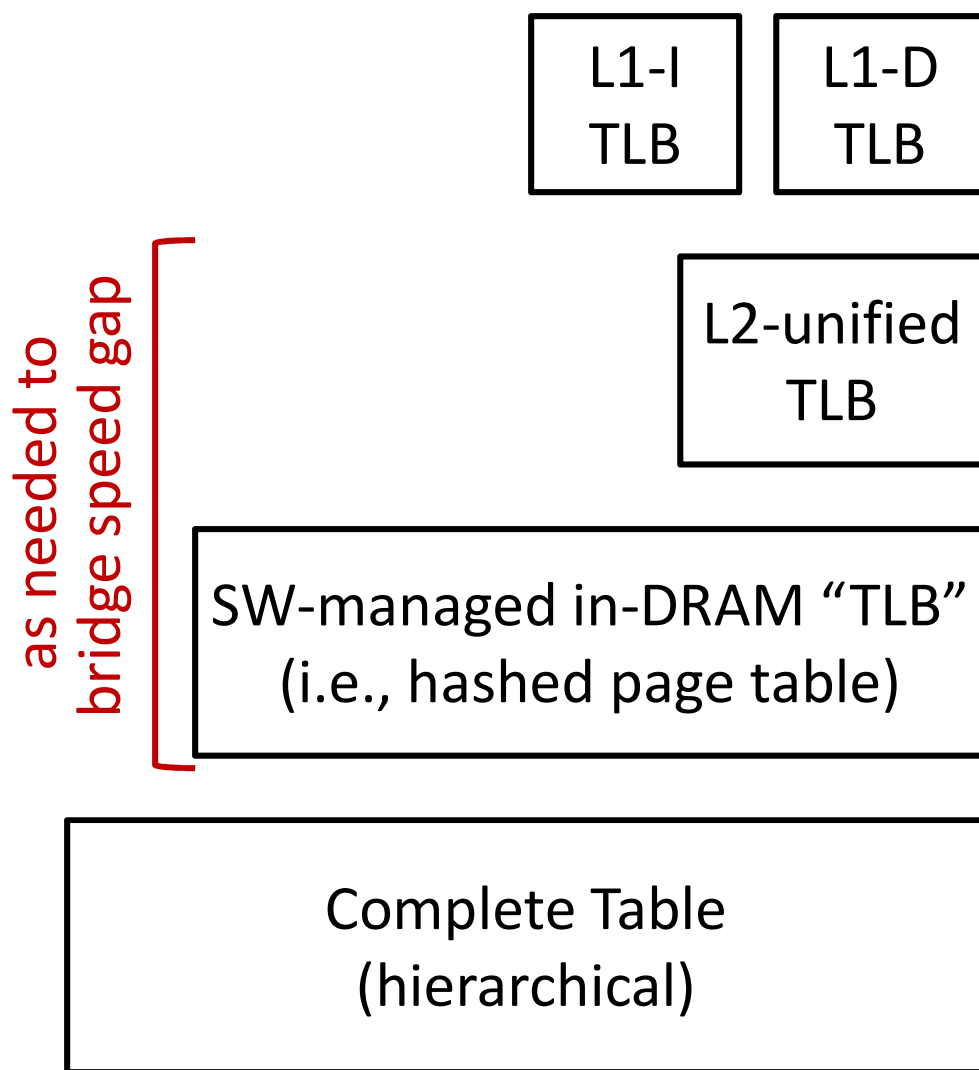
Direct-Mapped TLB (bad example)



TLB Design

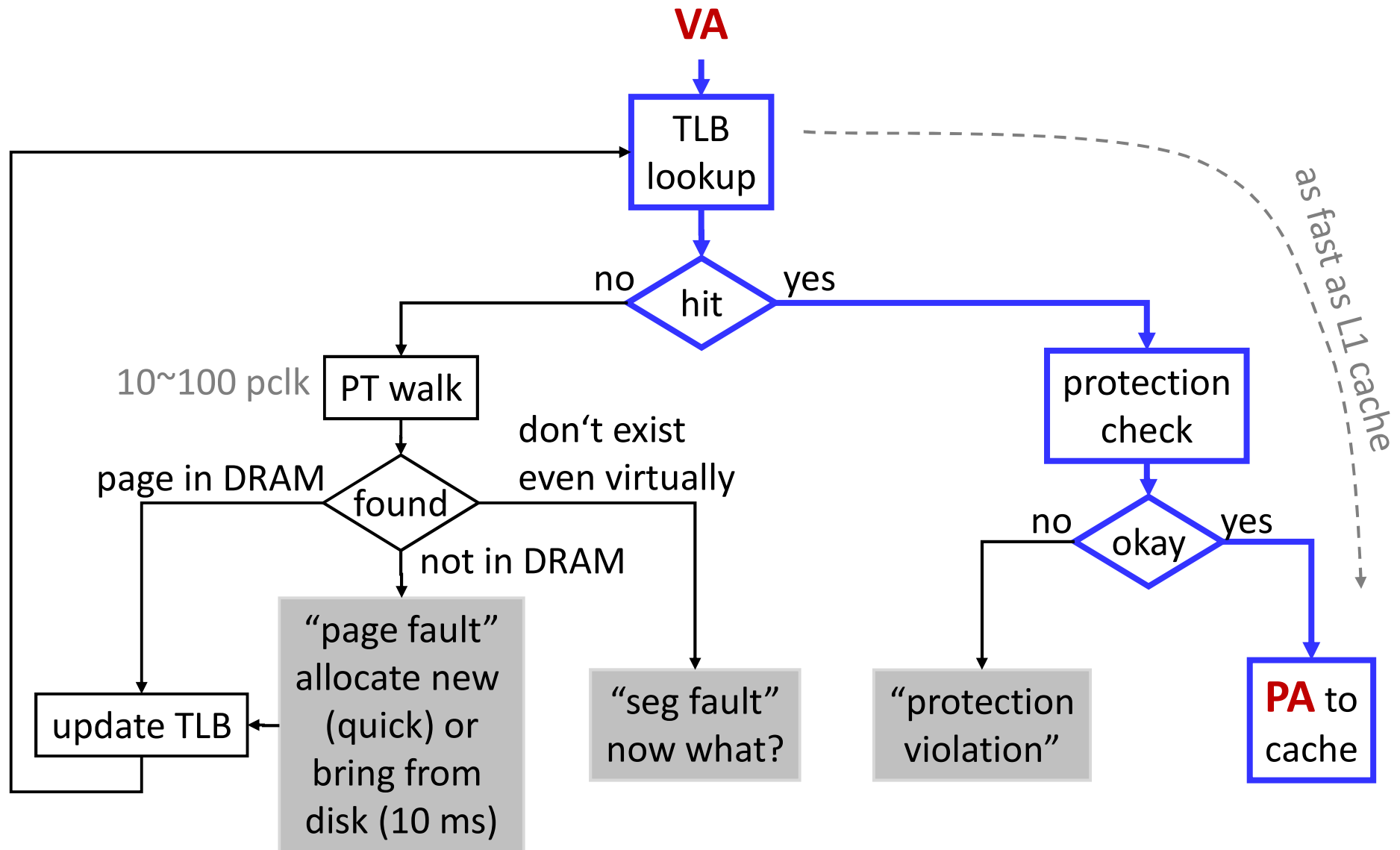
- C: L1 I-TLB should cover same footprint as L1 I-cache, e.g., if L1 I-cache is 64KB
 - L1 I-TLB needs minimum 16 pages but only if working set always use entire pages
 - was 32~64 entries; nowadays a few hundred
 - B: after accessing a page, how likely is it to access the next page? (coarse grain spatial locality)
 - usually one PTE per TLB entry
 - exception, MIPS keeps 2 PTEs per TLB entry
 - a: associativity to minimize collision?
 - in the old days, fully-associative is the norm
 - nowadays, 2~4-way-associative is more common
- 

Translation Hierarchy: Speed vs Coverage

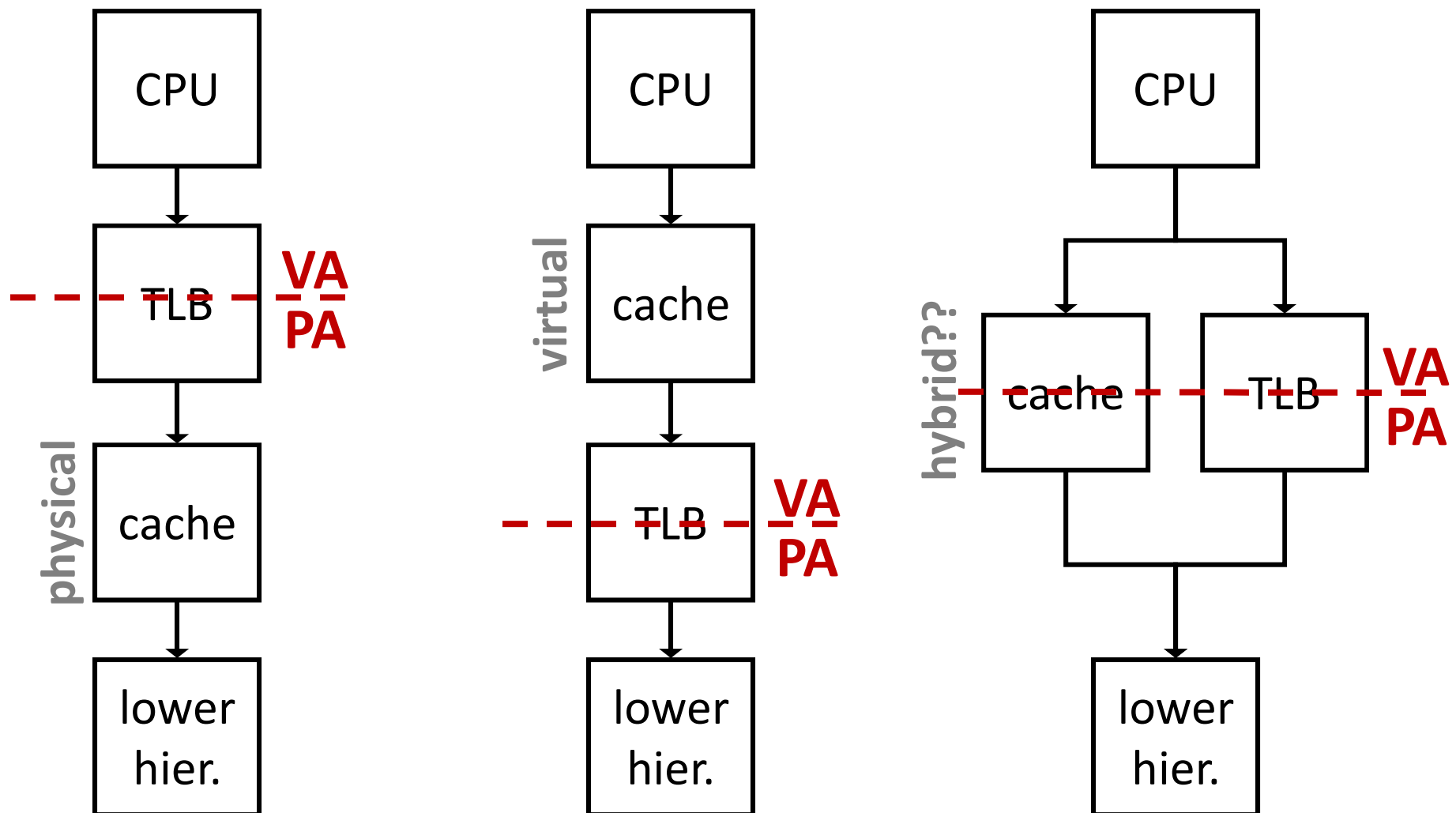


- L1 TLB: speed-driven, split I/D, small **C**, high **a**
accessed every cycle
- L2/L3/etc: capacity-driven, still on-chip, still HW
- “Soft TLB” or “Hashed Page Table”: in DRAM, SW managed, HW or SW walked—*fewer memory lookups than hier. table on large addresses*
- Hierarchical complete table at bottom

VA to PA Translation Flow Chart



How should VM and Cache Interact?



Only a question for L1 caches

Virtual Caches

- Even with TLB, translation takes time
- Naively, memory access time in the best case is
 $\text{TLB hit time} + \text{cache hit time}$
- Why not access cache with virtual addresses; only translate on a cache miss to DRAM
make sense if $\text{TLB hit time} \gg \text{cache hit time}$
- Virtual caches in SUN SPARC ISA, circa 1990
 - CPU fast enough for off-chip SRAM access to take multiple cycles
 - dies size large enough to include on-chip L1 caches
 - MMU and TLB still separate chip

These conditions no longer hold

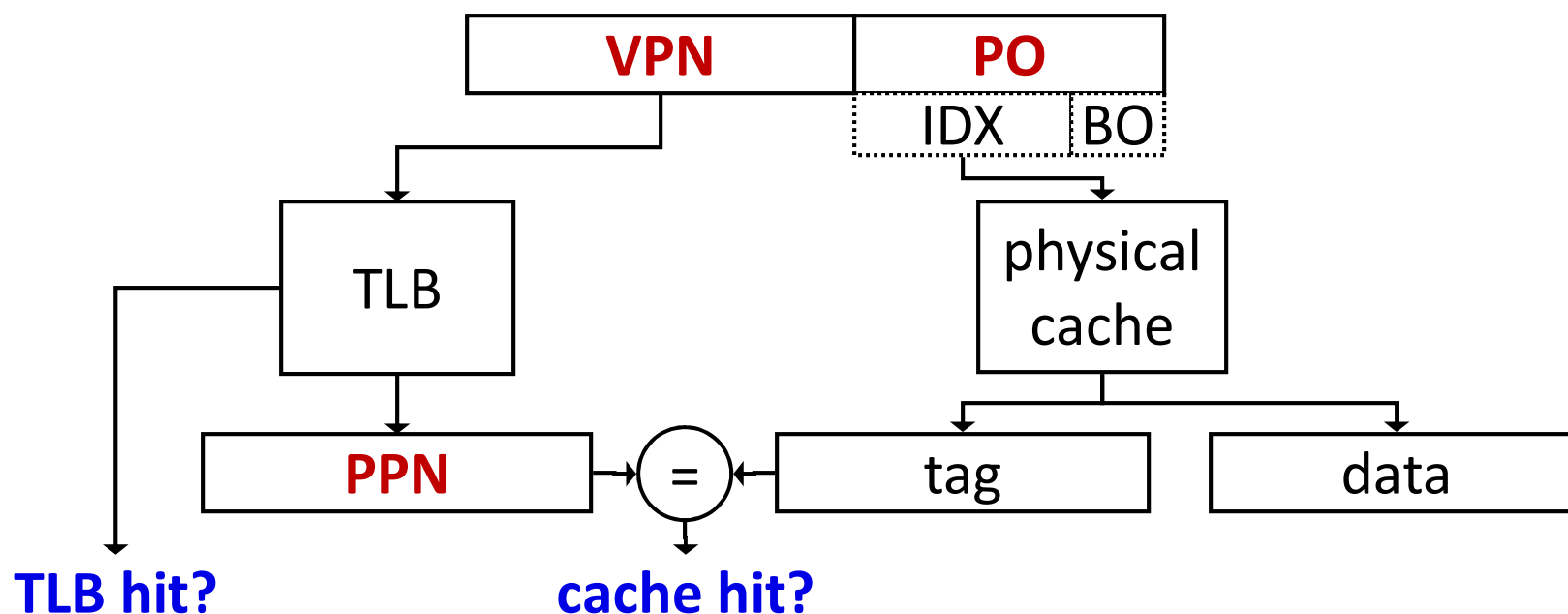
Resolving Synonym and Homonym in Virtual Caches

- **Homonyms:** same sound different meaning
 - same **EA** (in different processes) → different **PAs**
 - flush virtual cache between context; or include **ASID** in cache tag
- **Synonyms:** different sound same meaning
 - different **EAs** (from the same or different processes) → same **PA**
 - **PA** could be cached twice under different **EAs**
 - writes to one cached copy not reflected in the other cached copy

*Resolve by ensuring only 1 such **EA** in cache at a time*

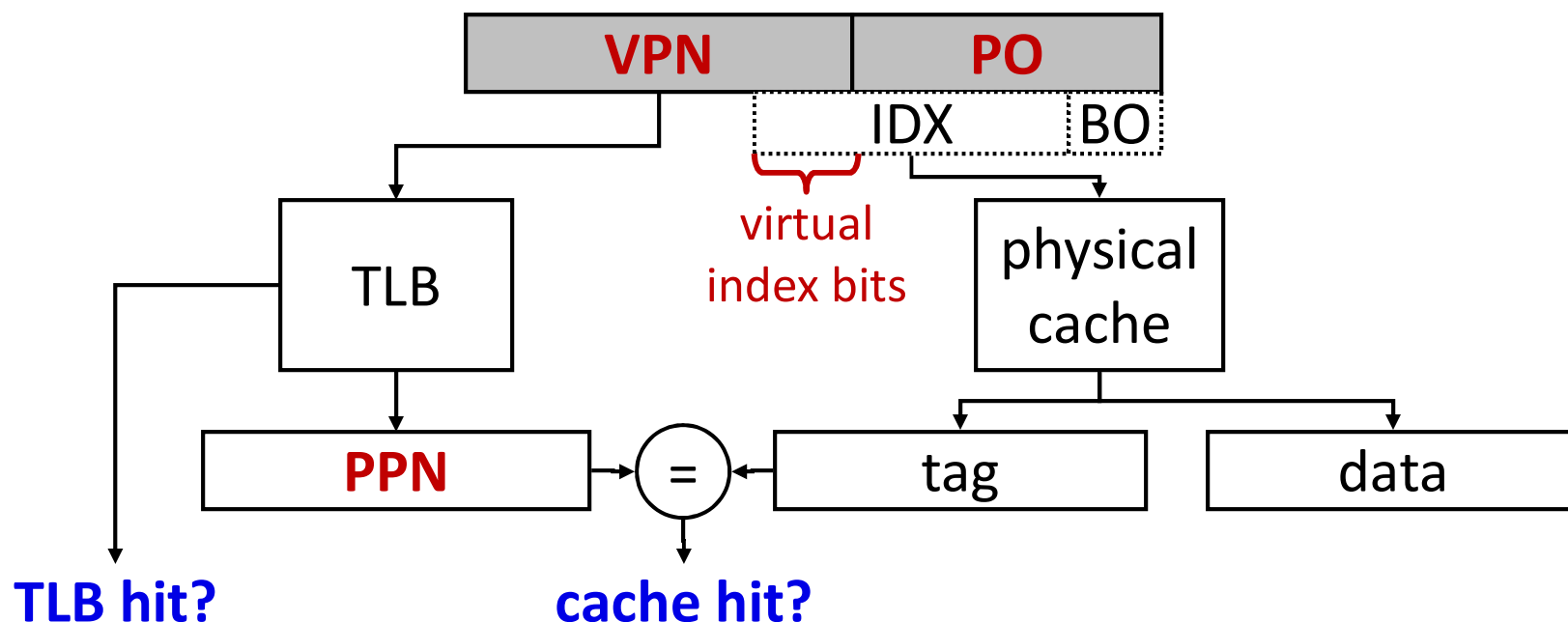
(misnomer) Virtually-Indexed Physically-Tagged

- If $C \leq (\text{page_size} \times \text{associativity})$, cache index bits come only from page offset
- If both cache and TLB are on chip
 - index both SRAMs concurrently using **PO** from **VA**
 - check cache tag (physical) against TLB at the end



“Large” Virtually-Indexed Caches

- If $C > (\text{page_size} \times \text{associativity})$, cache index bits include **VPN** \Rightarrow synonyms can cause problems
- Solutions to contain “virtual” index in page offset
 - increase associativity, 4KB page x 8 way = 32KB
 - increase page size



R10000's True Virtually Index Cache

- 32KB, 2-Way L1 D-cache
 - needs 10 bits of index + 4 bits of block offset
 - highest 2 index bits are **VA**[13:12] or **VPN**[1:0]
- Direct-mapped L2
 - L2 is inclusive of L1
 - **VPN**[1:0] is kept and checked as a part of L2 tag
- Given synonyms A_{VA} and B_{VA} that differs in **VPN**[1:0]
 - suppose A_{VA} accessed first so cached in L1 and L2
 - when accessing B_{VA} later
 1. B_{VA} indexes to a different block in L1 and misses
 2. B_{VA} indexes to the same block as A_{VA} in physical L2
 3. L2 detects synonym when comparing **VPN** portion of tag; L2 evicts A_{VA} from L1 before reloading B_{VA}

Interactions of VM and DMA

- A contiguous block in **VA**
 - is not guaranteed contiguous in **PA**
 - may not be in memory at all
- Software solutions
 - kernel copies from user buffer to pinned, contiguous buffer before DMA, or
 - user allocate special pinned and consecutive pages for zero-copy DMA
- Smarter DMA engines follow a “linked list” of commands for moving non-contiguous blocks
- Virtually-addressed I/O bus with I/O MMU

**Read “Virtual memory in
contemporary microprocessors” by
Jacob and Mudge before coming to
next Lecture!!!**