

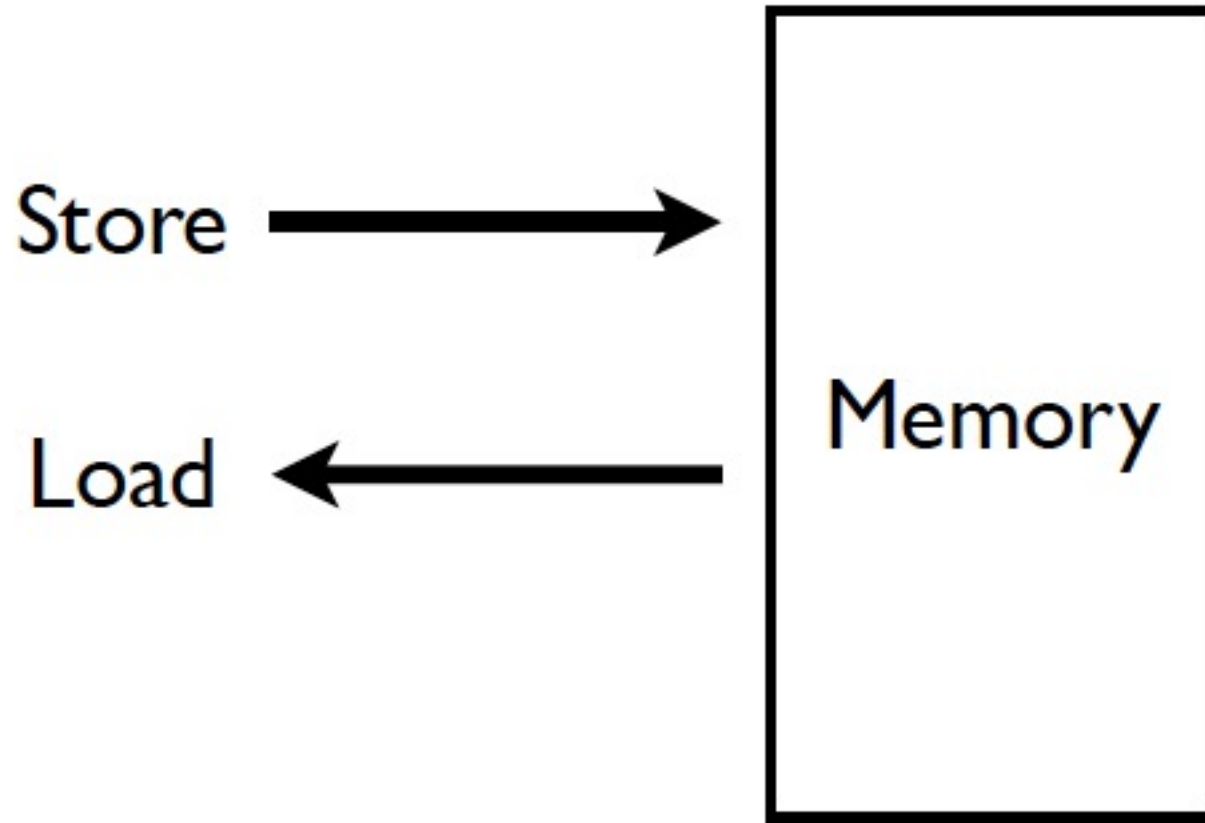
P&S HW/SW Co-design

Lecture 3: Virtual Memory (II)

Konstantinos Kanellopoulos
Prof. Onur Mutlu

ETH Zurich
Spring 2022
13 April 2022

Memory (Programmer's View)



Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

Abstraction: Virtual vs. Physical Memory

- **Programmer** sees **virtual memory**
 - Can assume the memory is “infinite”
 - Reality: **Physical memory** size is much smaller than what the programmer assumes
 - **The system** (system software + hardware, cooperatively) maps **virtual memory addresses** to **physical memory**
 - The system automatically manages the physical memory space **transparently to the programmer**
- + Programmer does not need to know the physical size of memory nor manage it → A small physical memory can appear as a huge one to the programmer → Life is easier for the programmer
- More complex system software and architecture

A classic example of the programmer/(micro)architect tradeoff

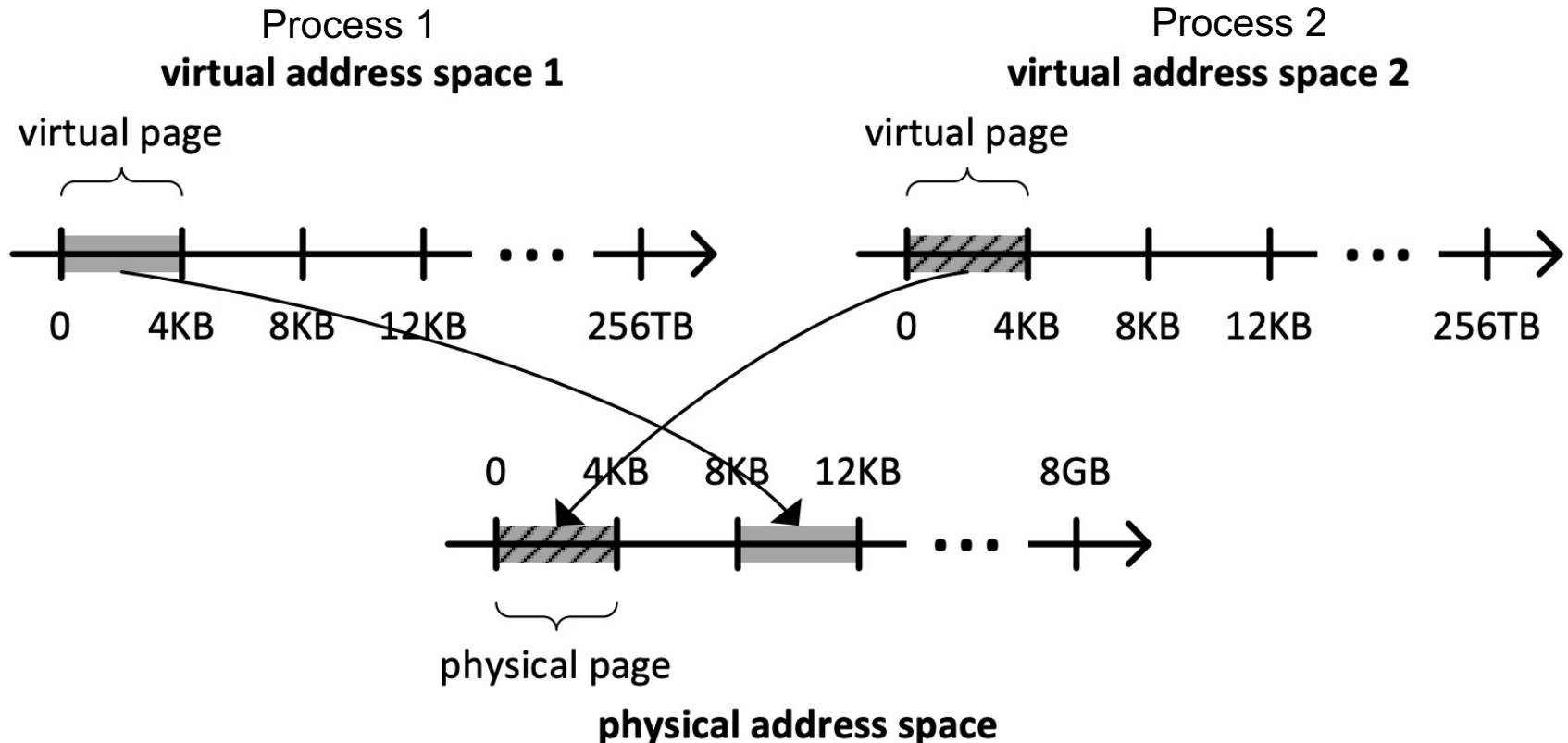
Benefits of Automatic Management of Memory

- Programmer does not deal with physical addresses
- Each process has its own independent mapping of virtual→physical addresses

- Enables
 - ❑ Code and data to be located anywhere in physical memory
(relocation and flexible location of data)
 - ❑ Isolation/separation of code and data of different processes in physical memory
(protection and isolation)
 - ❑ Code and data sharing between multiple processes
(sharing)

Virtual Memory: Conceptual View

■ Illusion of large, separate address space per process

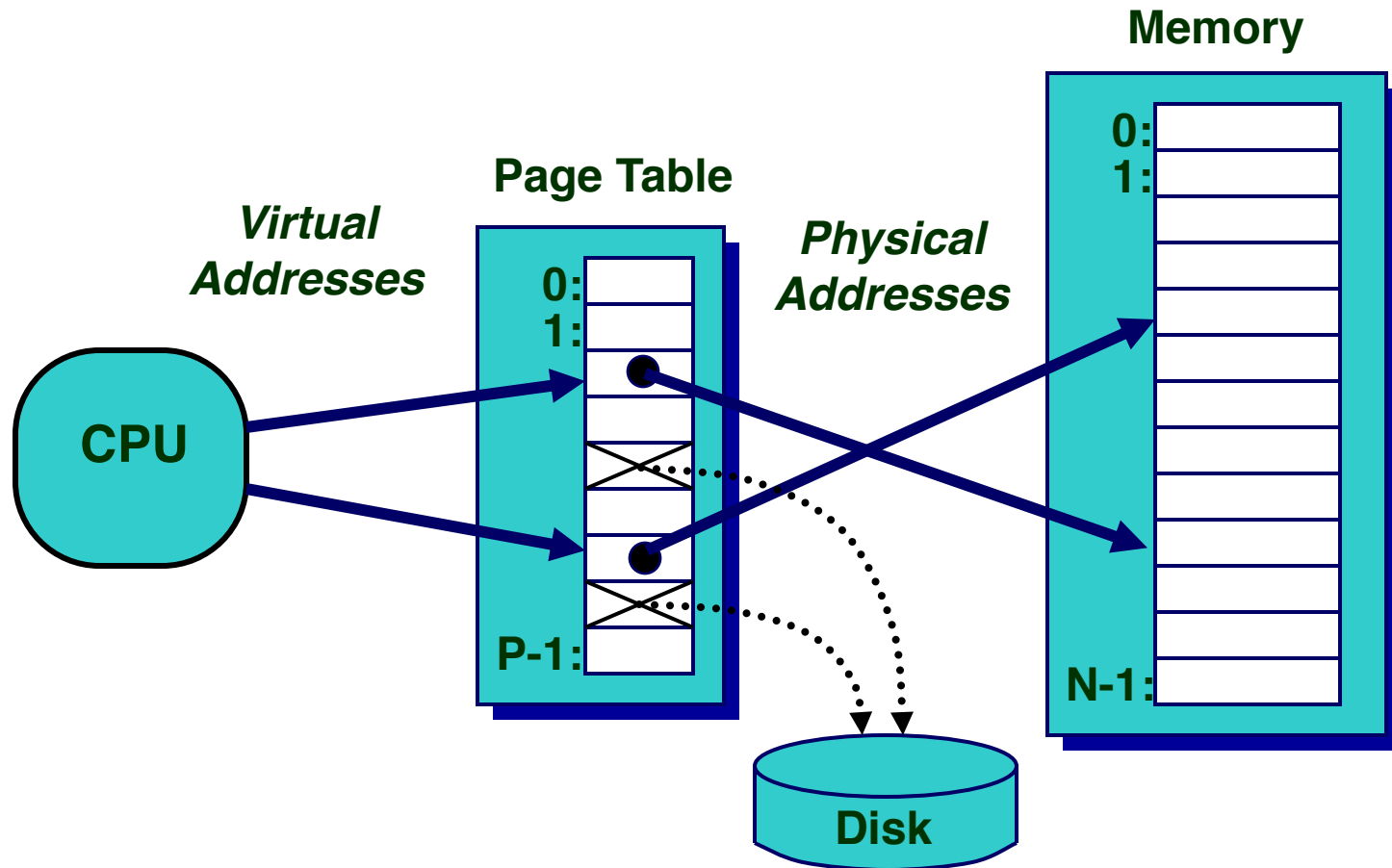


Requires **indirection and mapping** between virtual and physical spaces

Kim & Mutlu, "[Memory Systems](https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf)," Computing Handbook, 2014

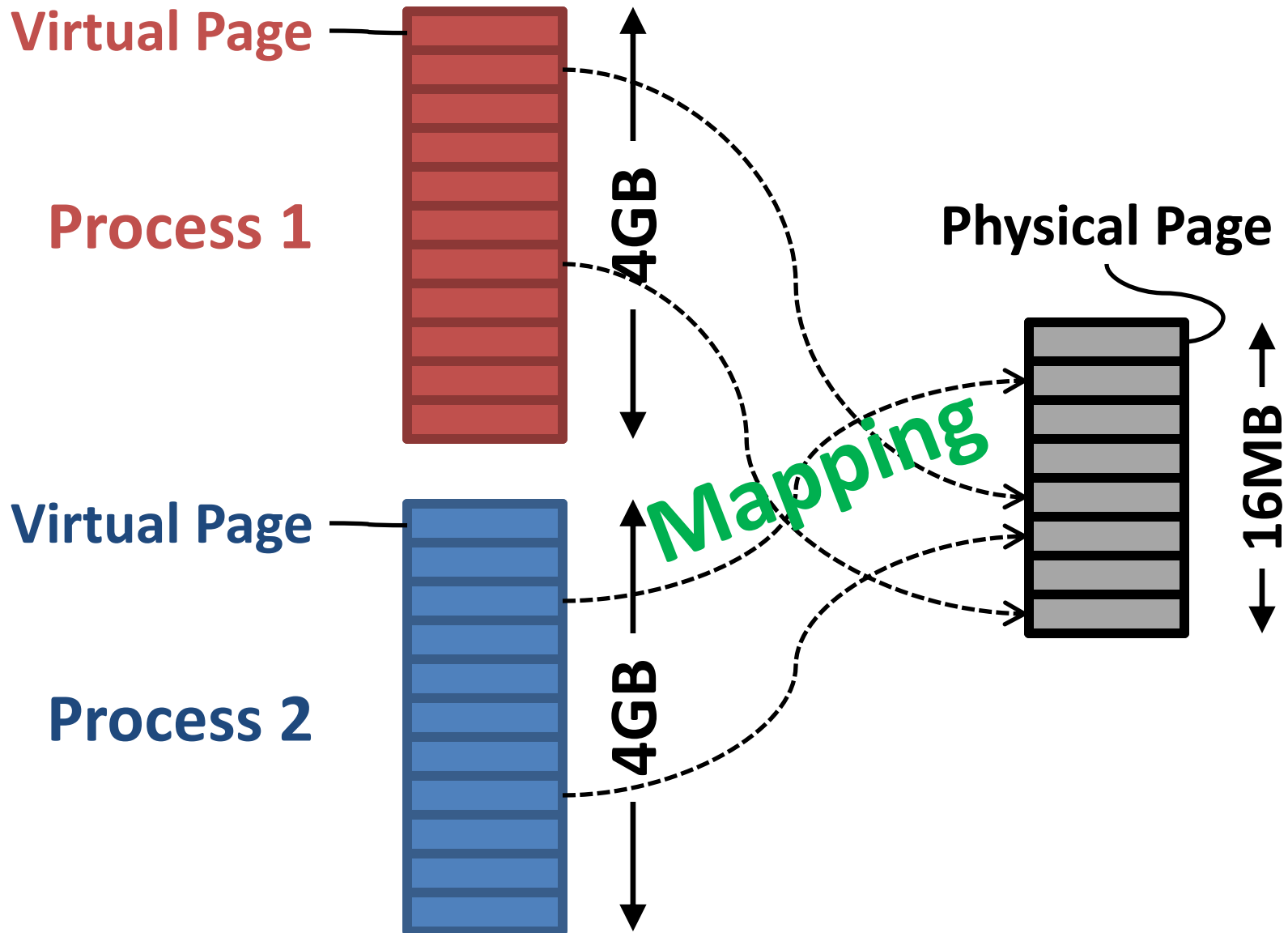
https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

A System with Virtual Memory (Page-based)



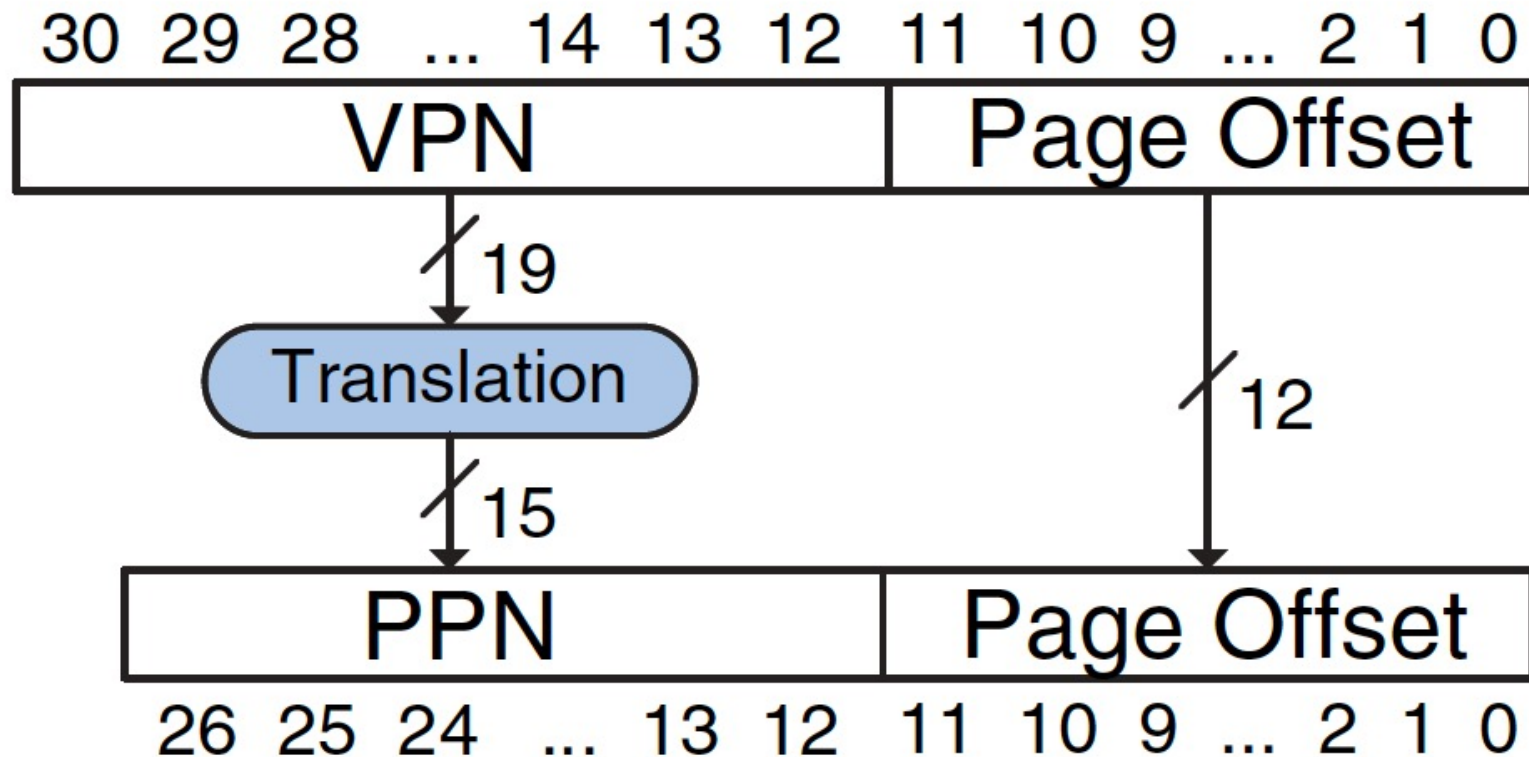
- **Address Translation:** The hardware converts virtual addresses into physical addresses via an OS-managed lookup table (page table)

Page-based Virtual-to-Physical Mapping



Address Translation

Virtual Address



Physical Address

Four-level Paging in x86-64

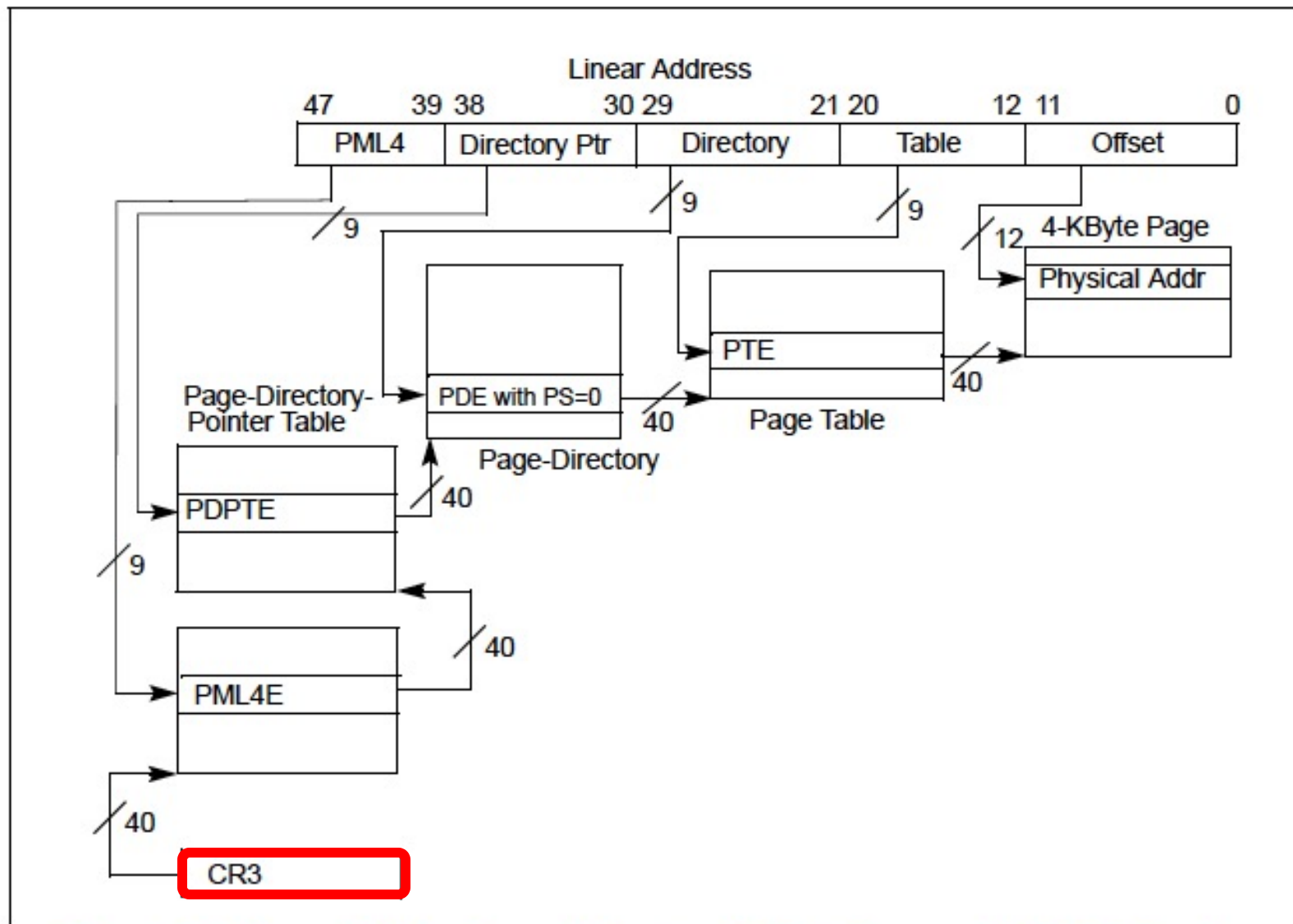


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Page Table Challenges

- Challenge 1: **Page table is large**
 - ❑ at least part of it needs to be located in physical memory
 - ❑ solution: **multi-level (hierarchical) page tables**
- Challenge 2: **Each instruction fetch or load/store requires at least two memory accesses:**
 1. one for address translation (page table read)
 2. one to access data with the physical address (after translation)
- Two memory accesses to service an instruction fetch or load/store greatly degrades execution time
 - ❑ Num. of memory accesses increases with multi-level page tables
 - ❑ **Unless we are clever... → speed up the translation...**

Supporting Virtual Memory

- Virtual memory **requires both HW+SW support**
 - Page Table is in memory
 - Can be cached in special hardware structures called Translation Lookaside Buffers (TLBs)
- The hardware component is called the **MMU** (memory management unit)
 - Includes Page Table Base Register(s), TLBs, page walkers
- **It is the job of the software** to
 - Populate page tables, decide what to replace in physical memory
 - Change the Page Table Base Register on context switch (to use the running thread's page table)
 - Handle page faults and ensure correct mapping

Three Major Issues in Virtual Memory

1. How large is the page table and how do we store and access it?
 2. How can we speed up translation & access control check?
 3. When do we do the translation in relation to cache access?
- There are many other issues we will not cover in detail
 - What happens on a context switch?
 - How can you handle multiple page sizes?
 - ...

Four-level Paging in x86-64

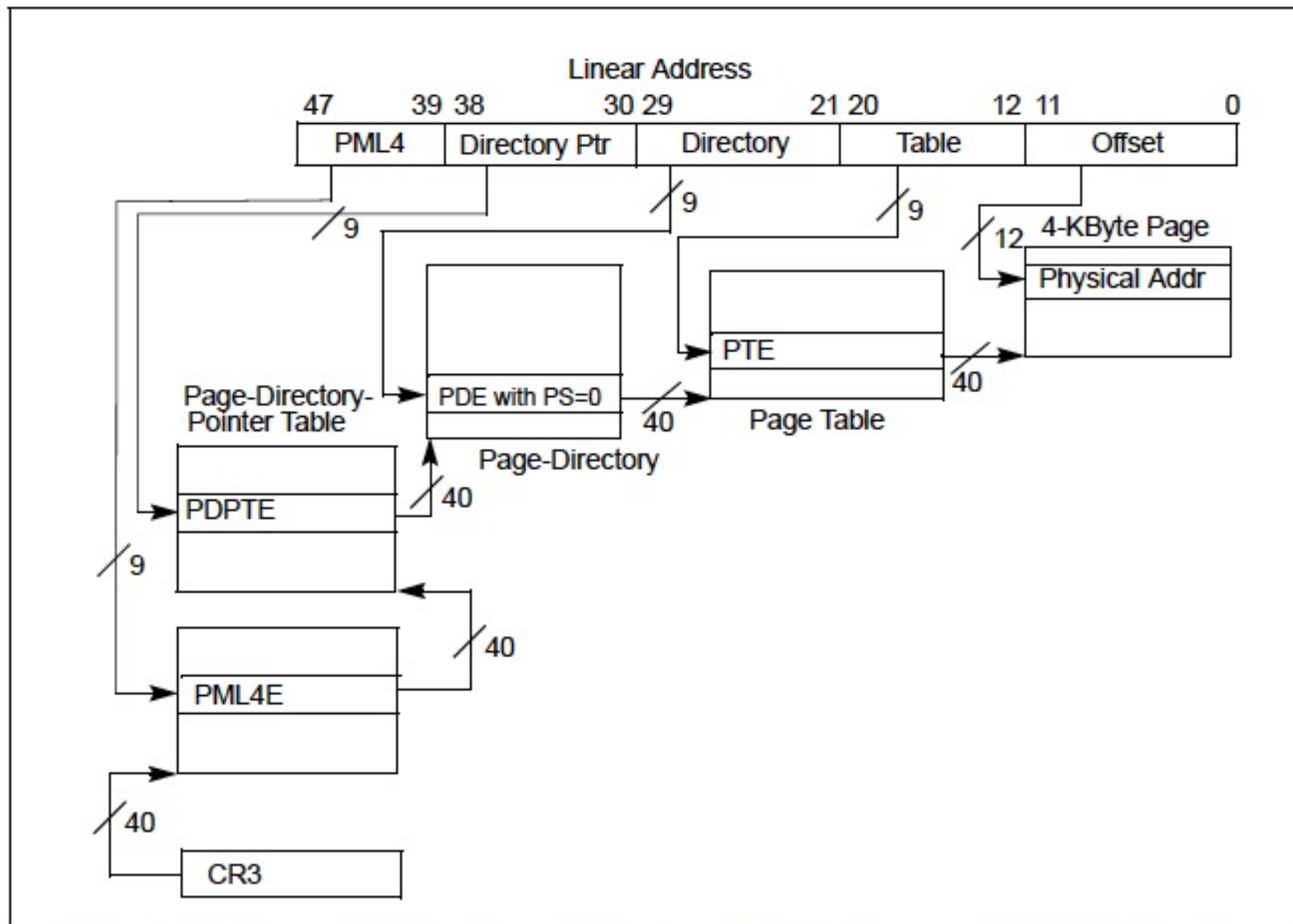


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Trade-Offs in Page Size

■ Large page size (e.g., 1GB)

- Pro: Fewer PTEs required → Saves memory space
 - Pro: Fewer Accesses during page table walk → Improves performance

 - Con: Cannot have fine-grained permissions
 - Con: Large transfers to/from disk
 - Even when only 1KB is needed, 1GB must be transferred
 - Waste of bandwidth/energy
 - Reduces performance
 - Con: **Internal fragmentation**
 - Even when only 1KB is needed, 1GB must be allocated
 - Waste of space
-

X86-64 Page Table: Accessing 4KB pages

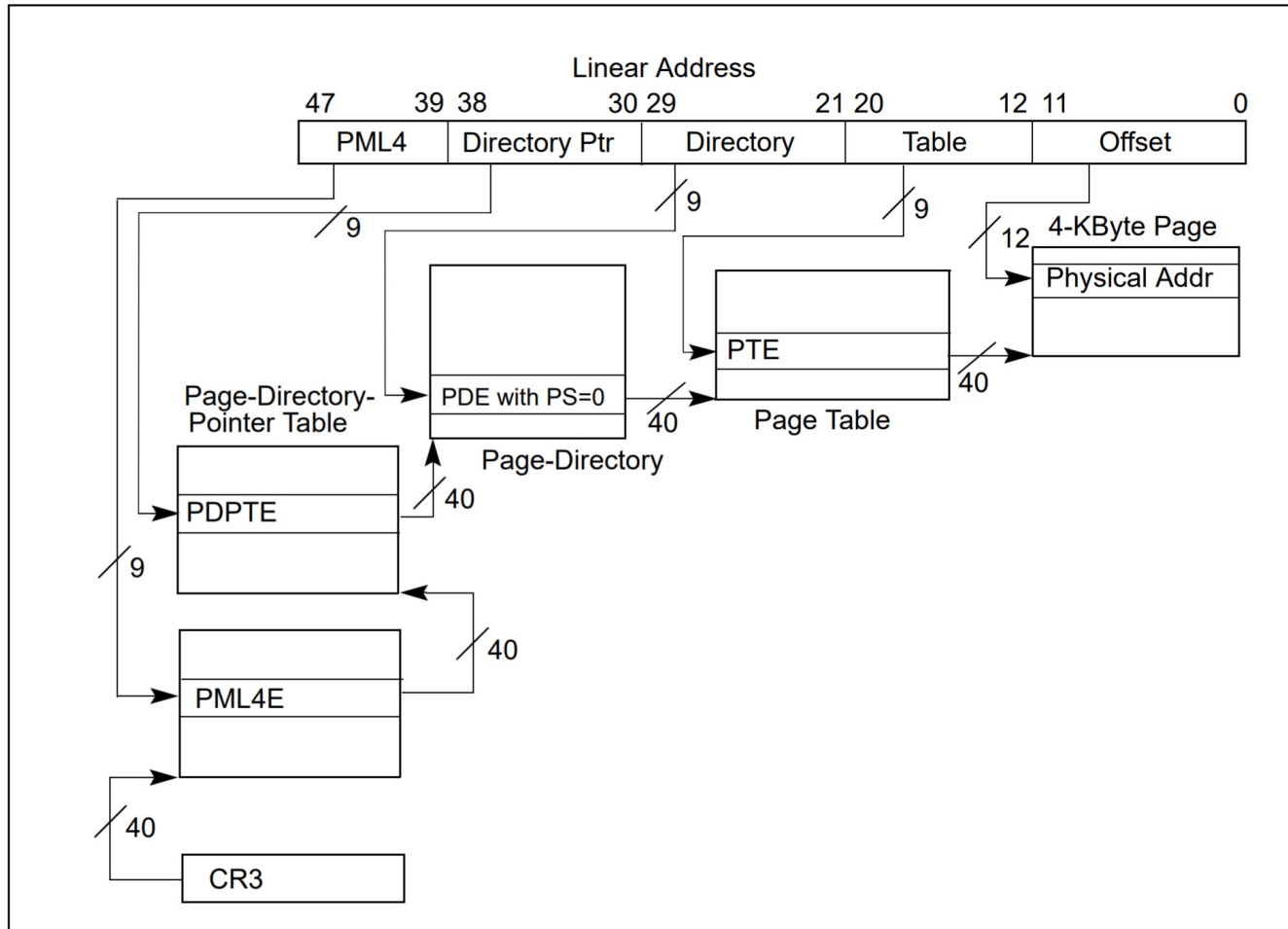


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

X86-64 Page Table: Accessing 2MB pages

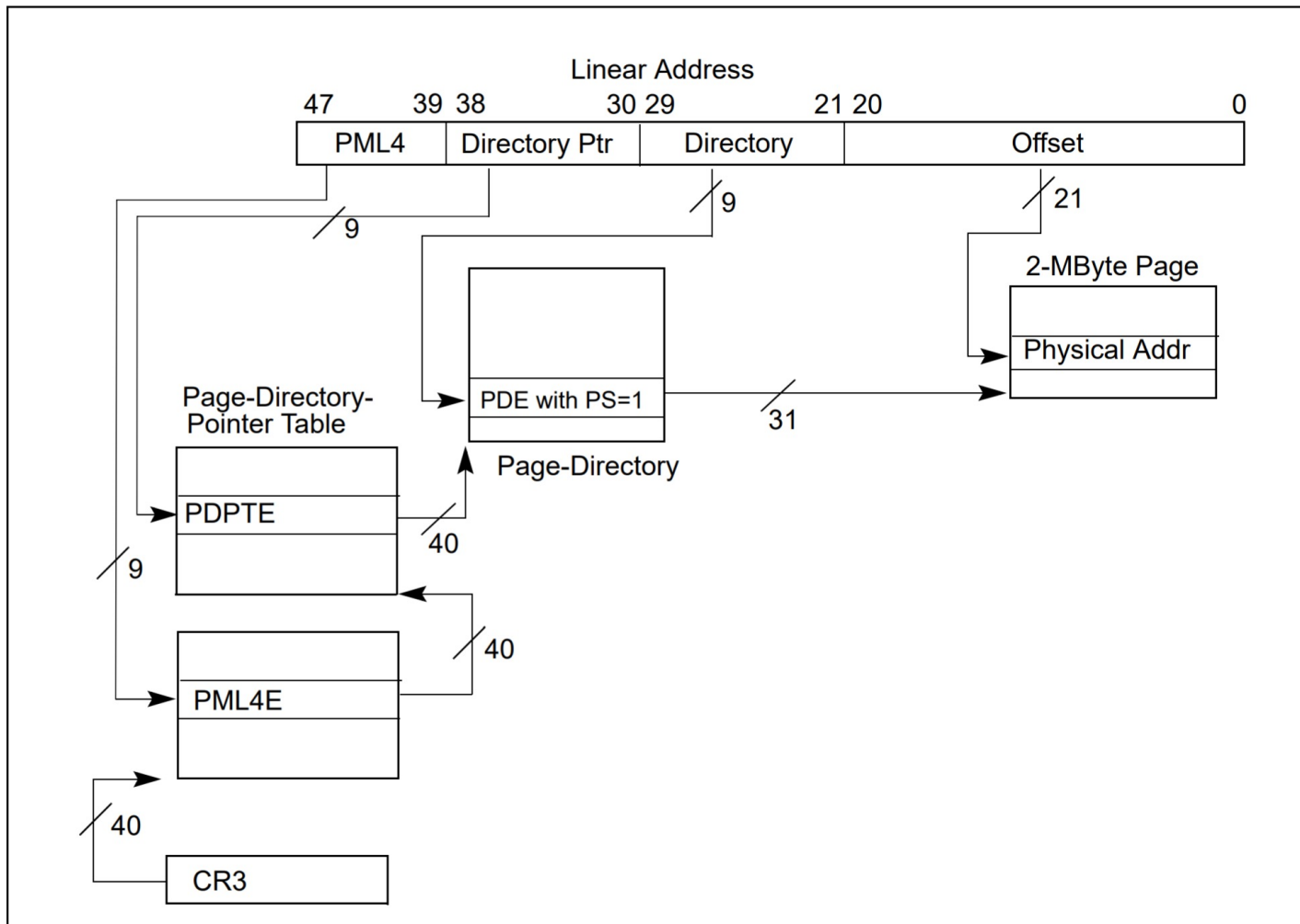


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

X86-64 Page Table: Accessing 1GB pages

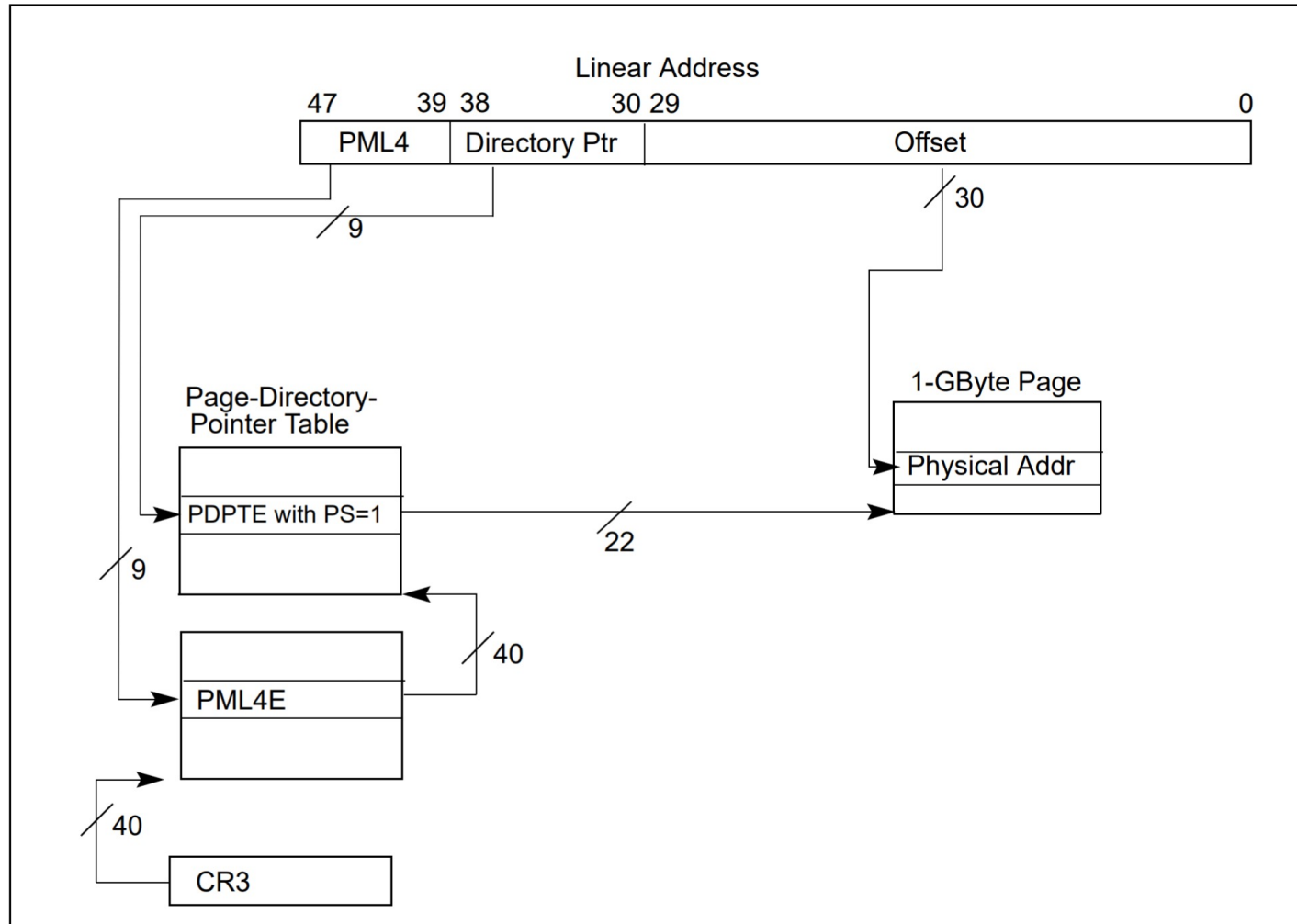


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

Three Major Issues in Virtual Memory

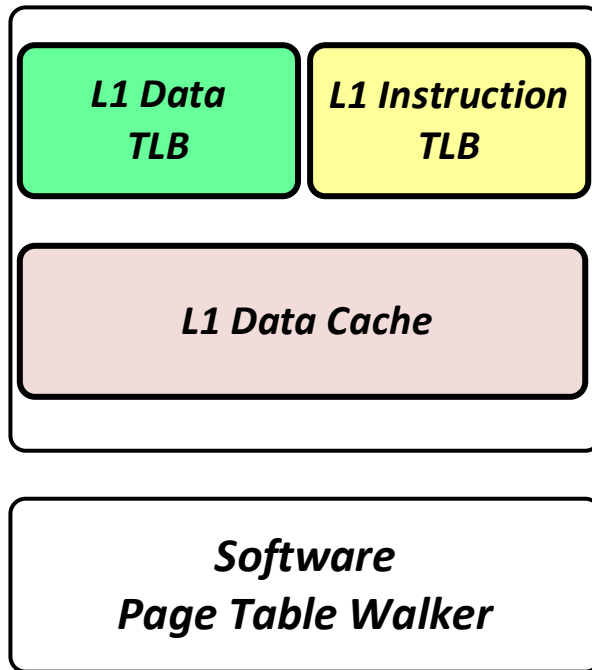
1. How large is the page table and how do we store and access it?
 2. How can we speed up translation & access control check?
 3. When do we do the translation in relation to cache access?
- There are many other issues we will not cover in detail
 - What happens on a context switch?
 - How can you handle multiple page sizes?
 - ...

Virtual Memory Issue II

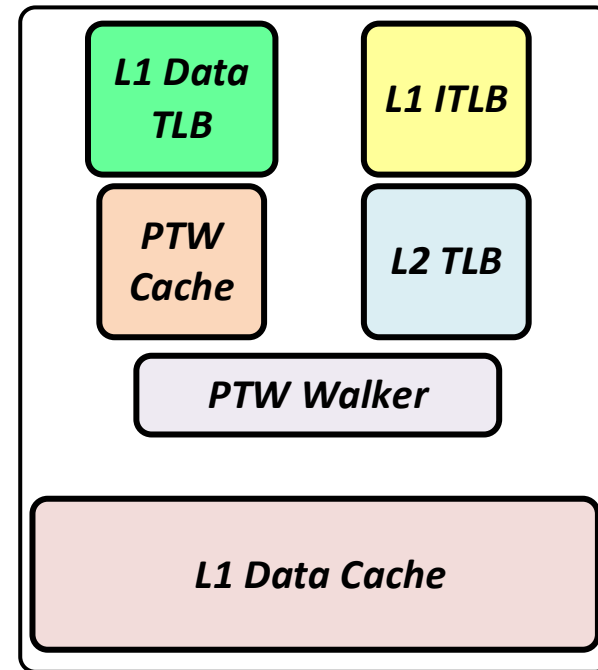
- How fast is the address translation?
 - How can we make it fast?
- Idea: Use a hardware structure called MMU that accelerates address translation

Evolution of Hardware Support for VM

Conventional Address Translation



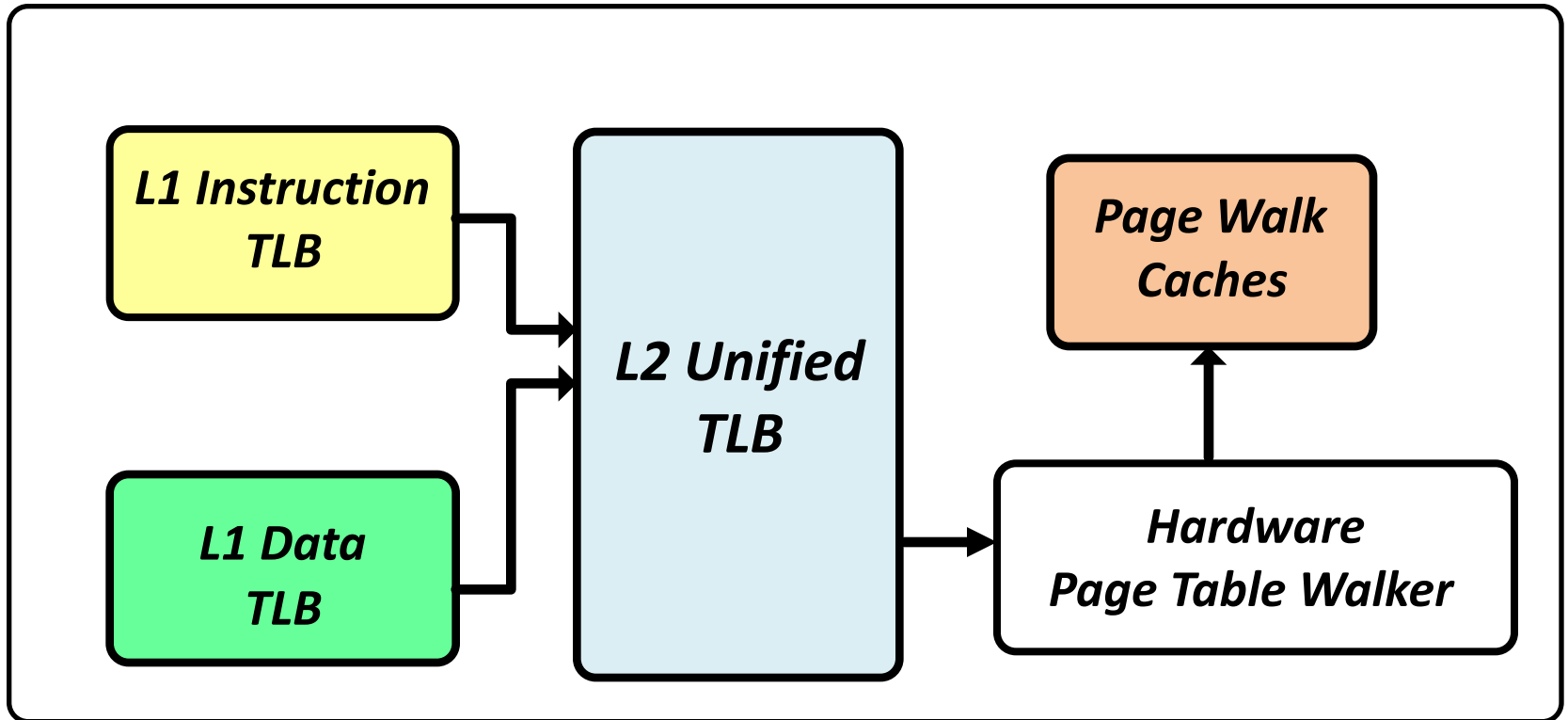
Modern Address Translation



Memory Management Unit

- The **Memory Management Unit (MMU)** is responsible for resolving address translation requests
 - One MMU per core (usually)
- MMU typically has three key components:
 - **Translation Lookaside Buffers** that cache recently-used virtual-to-physical translations (PTEs)
 - **Page Table Walk Caches** that offer fast access to the intermediate levels of a multi-level page table
 - **Hardware Page Table Walker** that sequentially accesses the different levels of the Page Table to fetch the required PTE

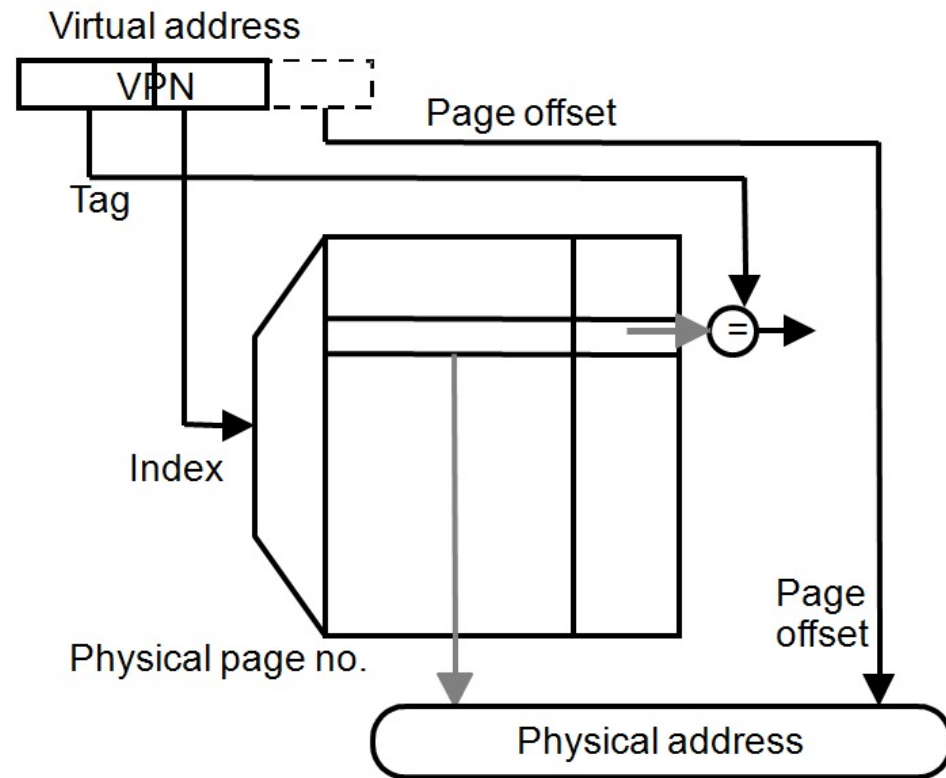
Intel Skylake: MMU



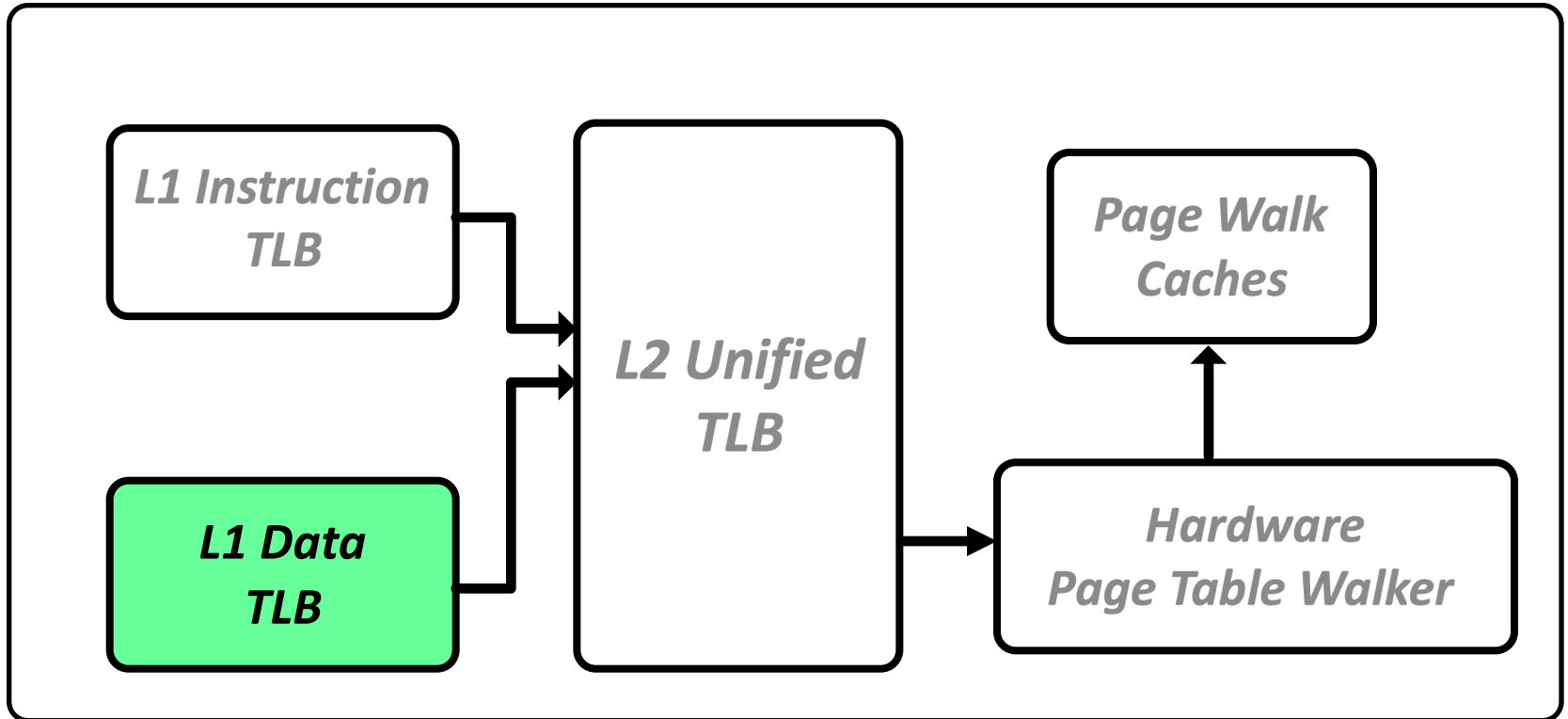
Speeding up Translation with a TLB

- A cache of address translations
 - Avoids accessing the page table on every memory access
- **Index** = lower bits of VPN (virtual page #)
- **Tag** = unused bits of VPN + process ID
- **Data** = a page-table entry
- **Status** = valid, dirty

The usual cache design choices (placement, replacement policy, multi-level, etc.) apply here too.



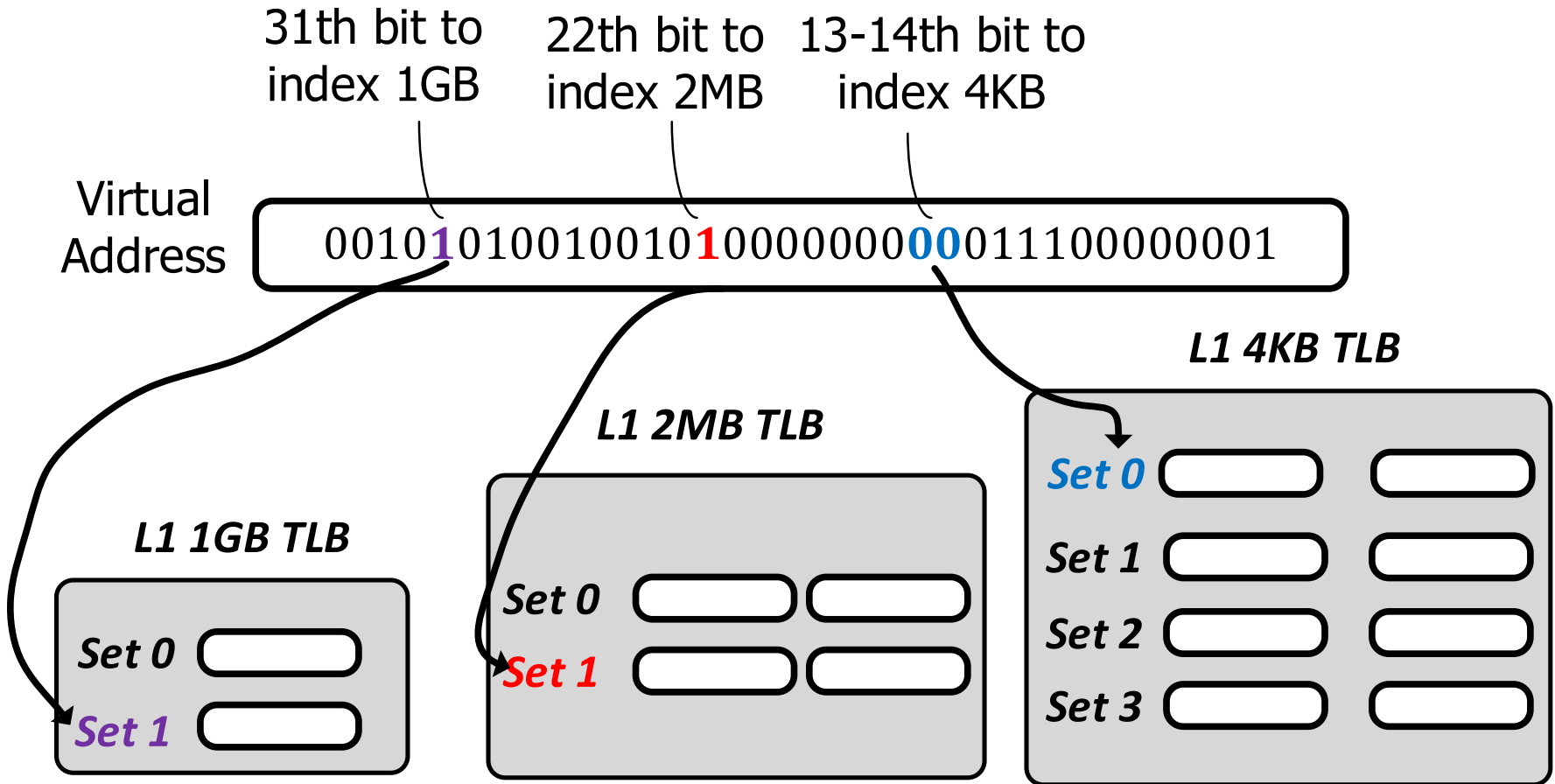
Intel Skylake: L1 Data TLB



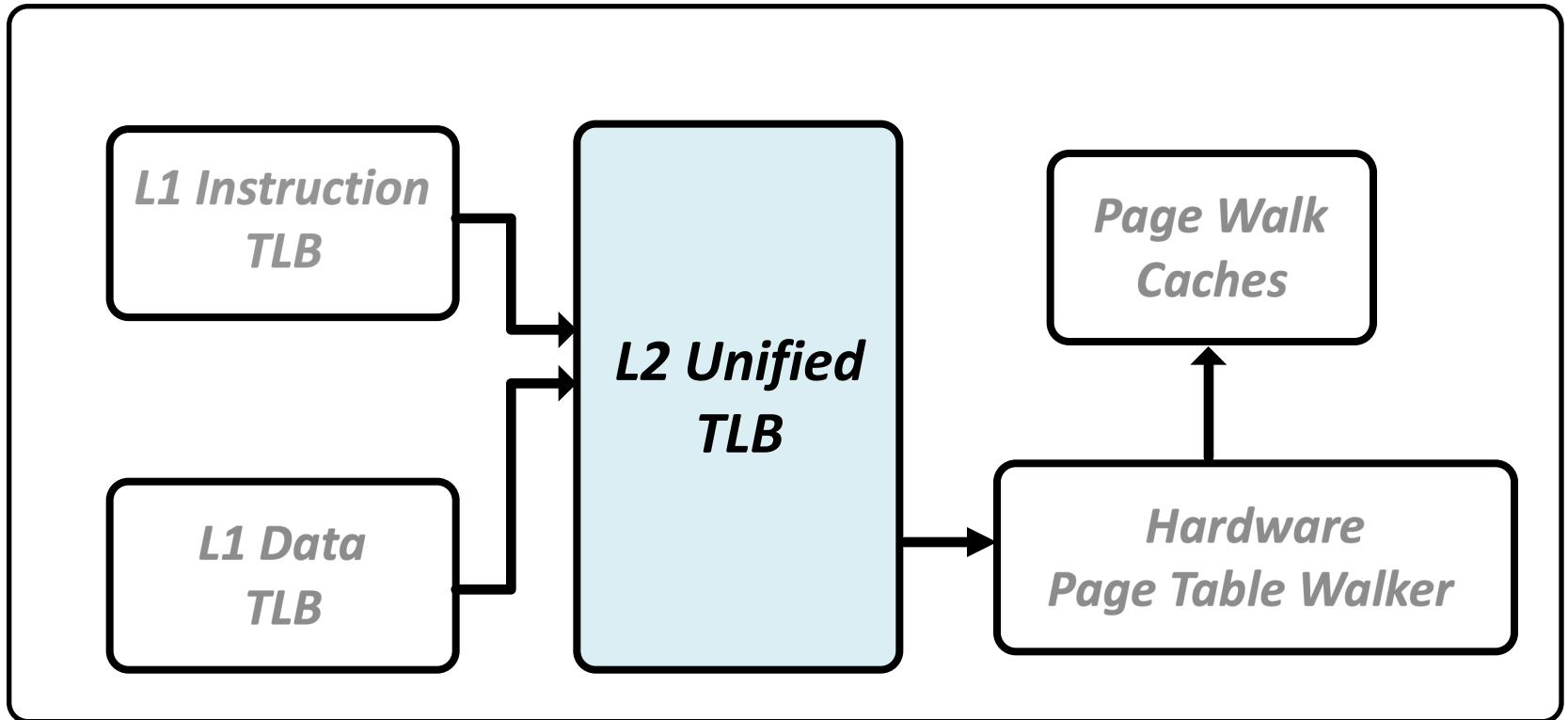
Intel Skylake: L1 Data TLB

- Separate L1 Data TLB structures for **4KB**, **2MB**, and **1GB** pages
- L1 DTLB
 - **4KB**: 64-entry, 4-way, 1 cycle access, 9 cycle miss
 - **2MB**: 32-entry, 4-way, 1 cycle access, 9 cycle miss
 - **1GB**: 4 entry, fully-associative
- Virtual-to-physical mappings are inserted in the corresponding TLB after a TLB miss
- During a translation request, all three L1 TLBs are looked up in parallel

L1 Data TLB: Parallel Lookup Example



Intel Skylake: L2 Unified I/D TLB



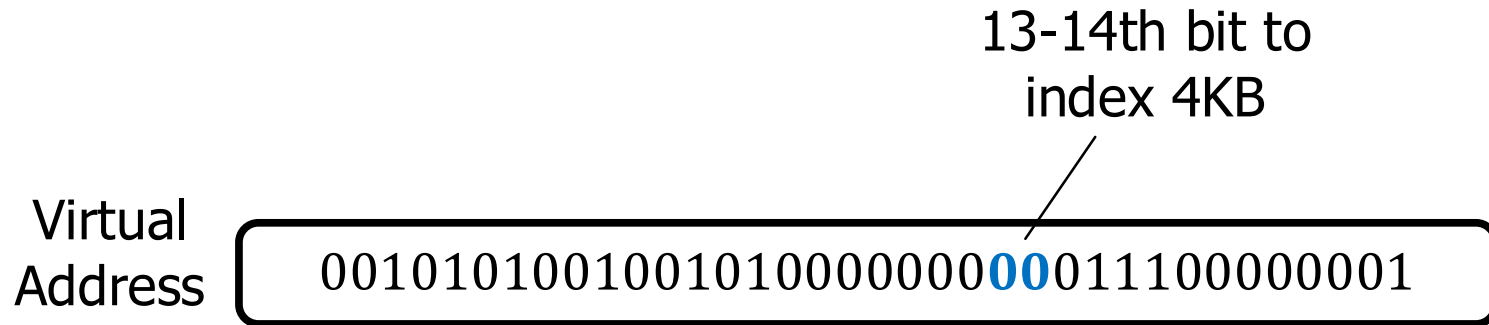
Intel Skylake: L2 Unified TLB

- L2 Unified TLB caches translations for both instr. and data
 - private per individual core
- 2 separate L2 TLB structures for 4KB/2MB and 1GB pages
- L2 TLB
 - **4KB/2MB**: 1536-entry, 12-way, 14 cycle access, 9 cycle miss
 - **1GB**: 16-entry, 4-way, 1 cycle access, 9 cycle miss penalty
- Challenge: How can the L2 TLB support both 4KB and 2MB pages using a single structure?
(Not enough publicly available information for Intel Skylake)

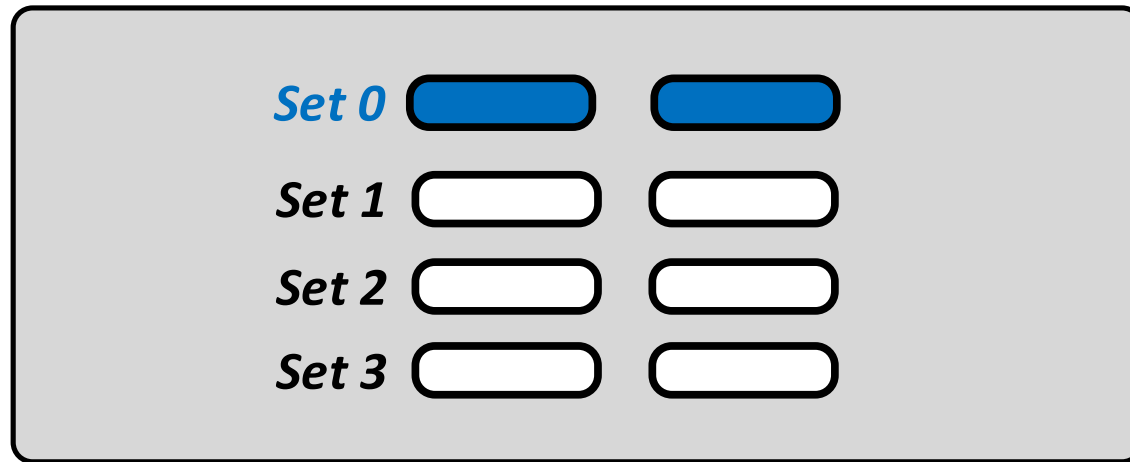
L2 Unified TLB: Accessing the TLB

- The 4KB/2MB structure of the L2 TLB is **probed in 2 steps**
- **Step 1:** Assume the page size is **4KB**, calculate the index bits and access the L2 TLB
 - If the tag matches, it is a hit. If the tag does not match, go to Step 2.
- **Step 2:** Assume the page size is **2MB**, **re-calculate** the index and access the L2 TLB.
 - If the tag matches, it is a hit. If the tag does not match, it is an L2 TLB miss.
- **General algorithm:**
Re-calculate index and probe TLB for all remaining page sizes

Step 1: Calculate index for 4KB



L2 TLB



Step 2: Re-calculate index for 2MB

22th-23th bit to
index 2MB

Virtual
Address

0010101001001**01**000000000011100000001

L2 TLB

Set 0	<input type="text"/>	<input type="text"/>
Set 1	<input type="text"/>	<input type="text"/>
Set 2	<input type="text"/>	<input type="text"/>
Set 3	<input type="text"/>	<input type="text"/>

L2 TLB: N-Step Index Re-Calculation

- Pros:
 - + Simple and practical implementation
- Cons:
 - Varying L2 TLB hit latency (faster for 4KB, slower for 2MB)
 - Slower identification of L2 TLB Miss as all page sizes need to be tested
- Potential Optimizations:
 1. **Parallel Lookup:** Look up for 4KB and 2MB pages in parallel
 2. **Page Size Prediction:** Predict the probing order

Tradeoffs are similar to “associativity in time” (also called pseudo-associativity)

Handling TLB Misses

- The TLB is small; it cannot hold **all** PTEs
 - Some translation requests will inevitably miss in the TLB
 - Must access memory to find the required PTE
 - Called **walking the page table**
 - Large performance penalty
 - Better TLB management & prefetching can reduce TLB misses
 - Who handles TLB misses?
 - Hardware or software?
-

Handling TLB Misses (II)

- Approach #1. **Hardware-Managed** (e.g., x86)
 - The hardware does the **page walk**
 - The hardware fetches the PTE and inserts it into the TLB
 - If the TLB is full, the entry **replaces** another entry
 - Done transparently to system software
 - Can employ specialized structures and caches
 - E.g., page walkers and page walk caches

 - Approach #2. **Software-Managed** (e.g., MIPS)
 - The hardware raises an exception
 - The operating system does the **page walk**
 - The operating system fetches the PTE
 - The operating system inserts/evicts entries in the TLB
-

Handling TLB Misses (III)

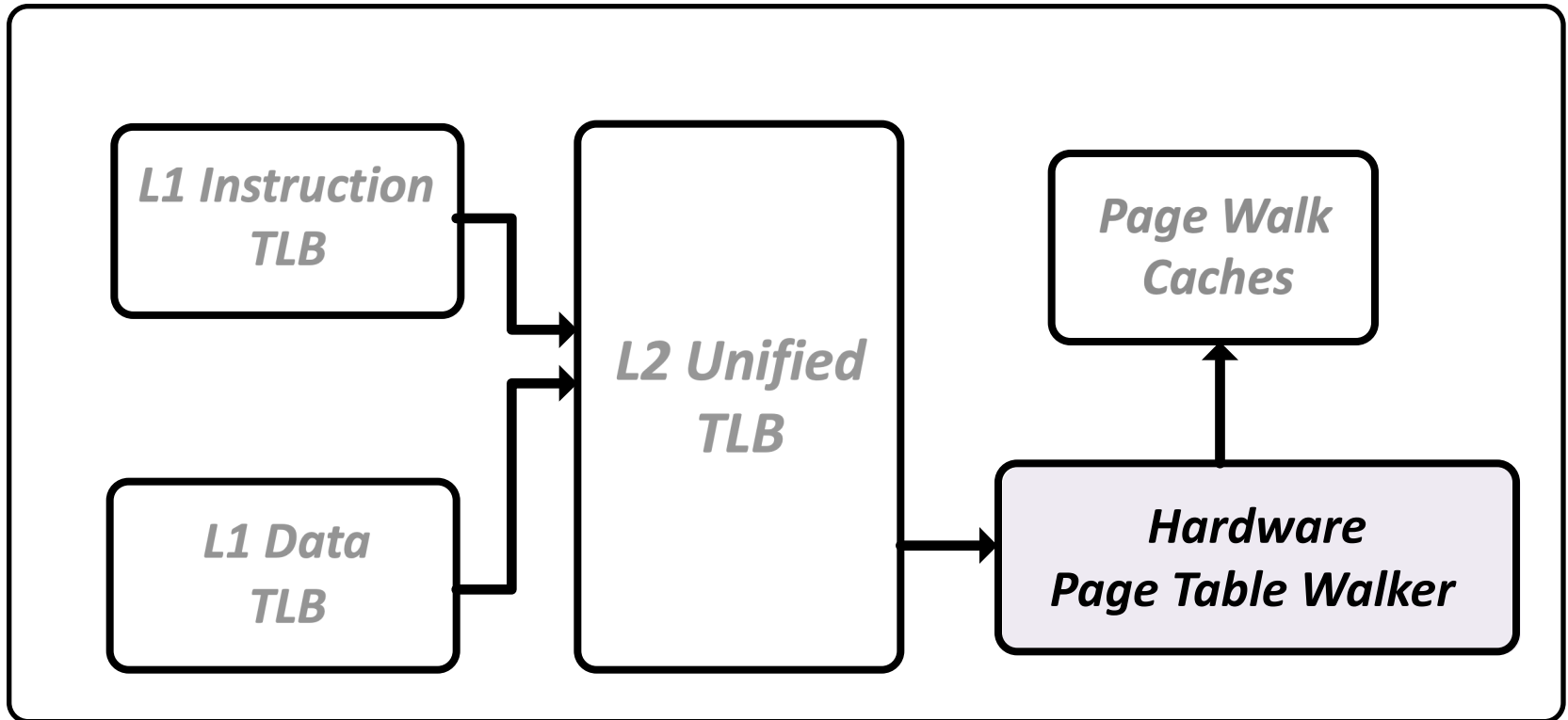
■ Hardware-Managed TLB

- + No exception on TLB miss. Instruction just stalls
- + Independent instructions may continue
- + No extra instructions/data brought into caches
- Page directory/table organization is fetched into the system:
OS has little flexibility in deciding these

■ Software-Managed TLB

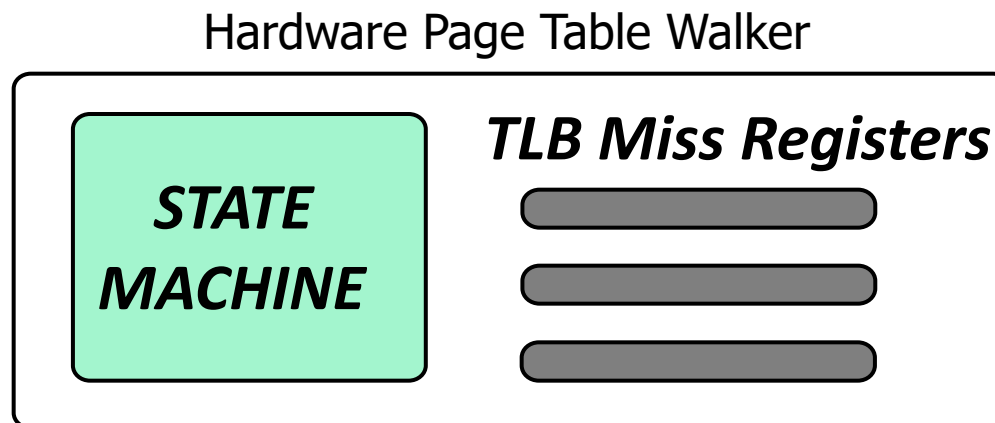
- + The OS can define the page table organization
 - + More sophisticated TLB replacement policies are possible
 - Need to generate an exception → performance overhead due to pipeline flush, exception handler execution, extra instructions brought to caches
-

Hardware Page Table Walker



Hardware Page Table Walker (I)

- A per-core hardware component that **walks the multi-level page table to avoid expensive context switches & SW handling**
- HW PTW consists of 2 components:
 - A state machine that is designed to be aware of the architecture's page table structure
 - Registers that keep track of outstanding TLB misses



Hardware Page Table Walker (II)

■ Pros:

- + Avoids the need for context switch on TLB miss
- + Overlaps TLB misses with useful computation
- + Supports concurrent TLB misses

■ Cons:

- Hardware area and power overheads
- Limited flexibility compared to software page table walk

Hardware Page Table Walker (III)

- PTW accesses the CR3 register that maintains information about the physical address of the root of the page table (PML4)
- PTW concatenates the content of CR3 with the first 9 bits of the virtual address

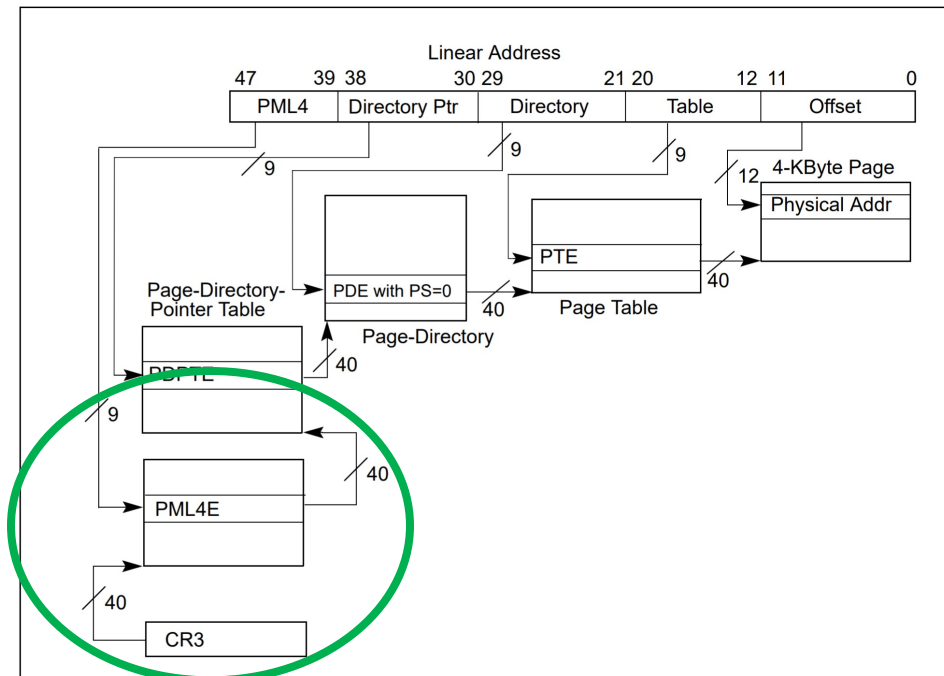


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

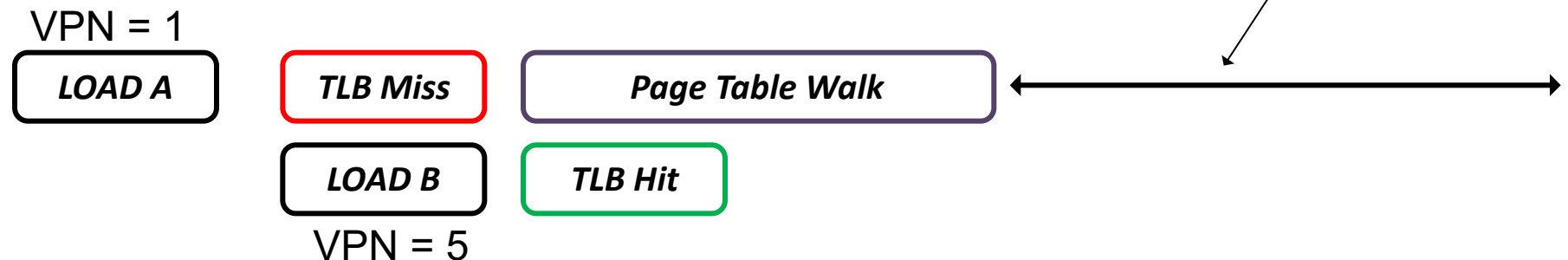
Hardware Page Table Walker (IV)

- Hardware PTWs allow overlapping TLB misses with useful computation

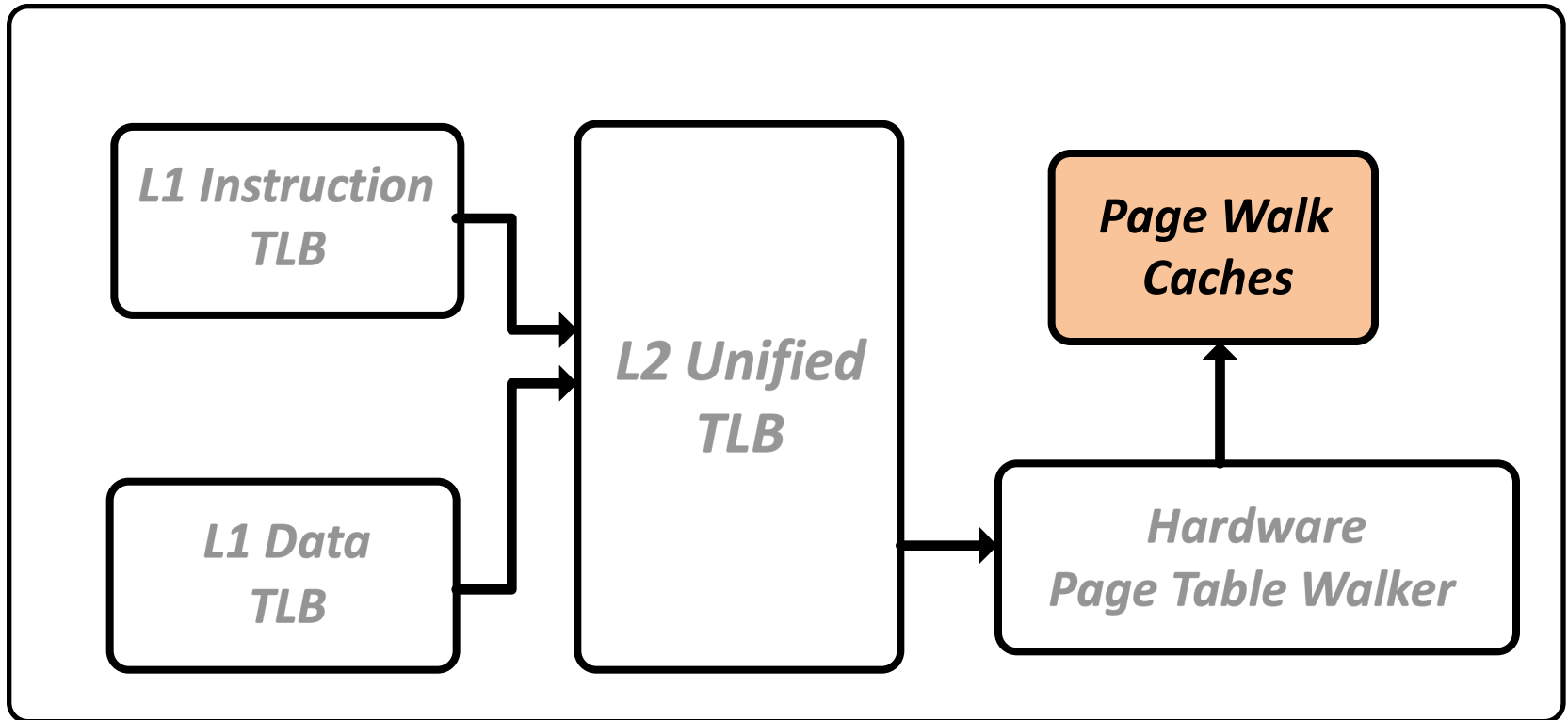
Software PTW



Hardware PTW



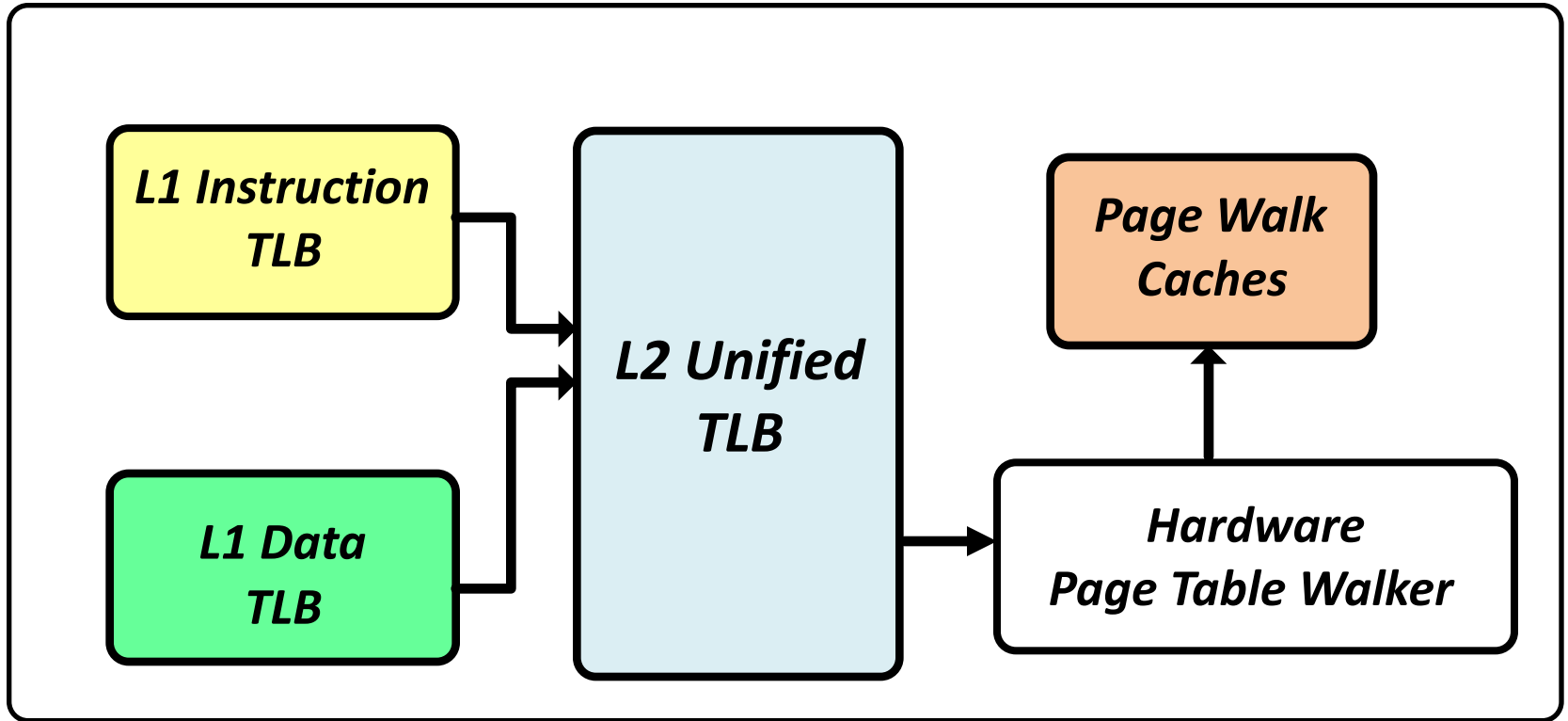
Page Walk Caches



Page Walk Caches

- Page Walk Caches cache translations from non-leaf levels of a multi-level page table to accelerate page table walks
- Page Walk Caches are low-latency caches that provide faster access to the page table levels
 - Faster compared to accessing the regular cache/memory hierarchy for every page table walk

Intel Skylake: MMU



Modern Virtual Memory Designs

	A14 "Firestorm" (iPhone 12 Pro)	Intel/AMD/ARM
Decode width	8	4, 5 (Samsung M3), 5 (Cortex-X1)
ROB size	630	352 (Intel Willow Cove)
Load/store queue size	~148 outstanding loads ~106 outstanding stores	Intel Sunny Cove (128-LQ, 72-SQ) AMD Zen3 (64-LQ, 44-SQ)
L1-TLB	256 entries	64 entries
L2-TLB	3072 entries	1536 entries
Page size	16KB	4KB
L1-I cache	192KB	48KB (Intel Ice Lake)
L1-D cache	128KB, 3-cycles	32KB (Intel/AMD), 4-cycles
L2 cache	8MB shared across two big-cores, ~16-cycles	1MB (Intel Cascade Lake)
L3 cache	16MB shared across all CPU cores and integrated GPU	1.375 MB/core

Virtual Memory and Cache Interaction

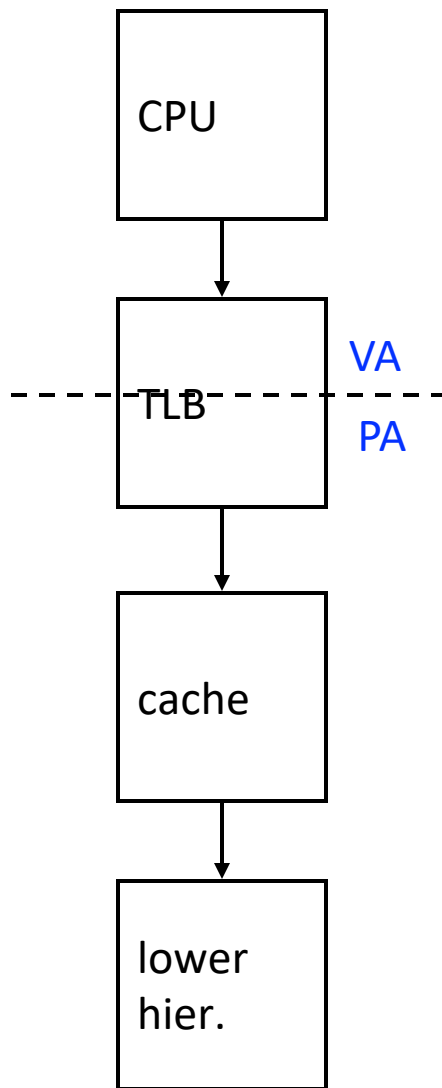
Address Translation and Caching

- When do we do the address translation?
 - Before or after accessing the L1 cache?
- In other words, is the cache virtually addressed or physically addressed?
 - Virtual versus physical cache
- What are the issues with a virtually addressed cache?
- **Synonym problem:**
 - Two different virtual addresses can map to the same physical address → same physical address can be present in multiple locations in the cache → can lead to inconsistency in data

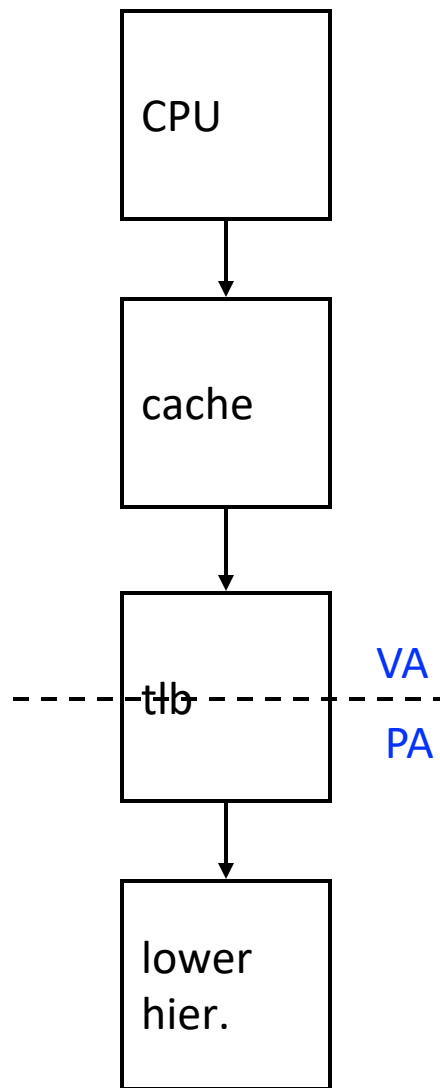
Homonyms and Synonyms

- **Homonym: Same VA can map to two different PAs**
 - Why?
 - VA is in different processes
- **Synonym: Different VAs can map to the same PA**
 - Why?
 - Different pages can share the same physical frame within or across processes
 - Reasons: shared libraries, shared data, copy-on-write pages within the same process, ...
- Do homonyms and synonyms create problems when we have a cache?
 - Is the cache virtually or physically addressed?

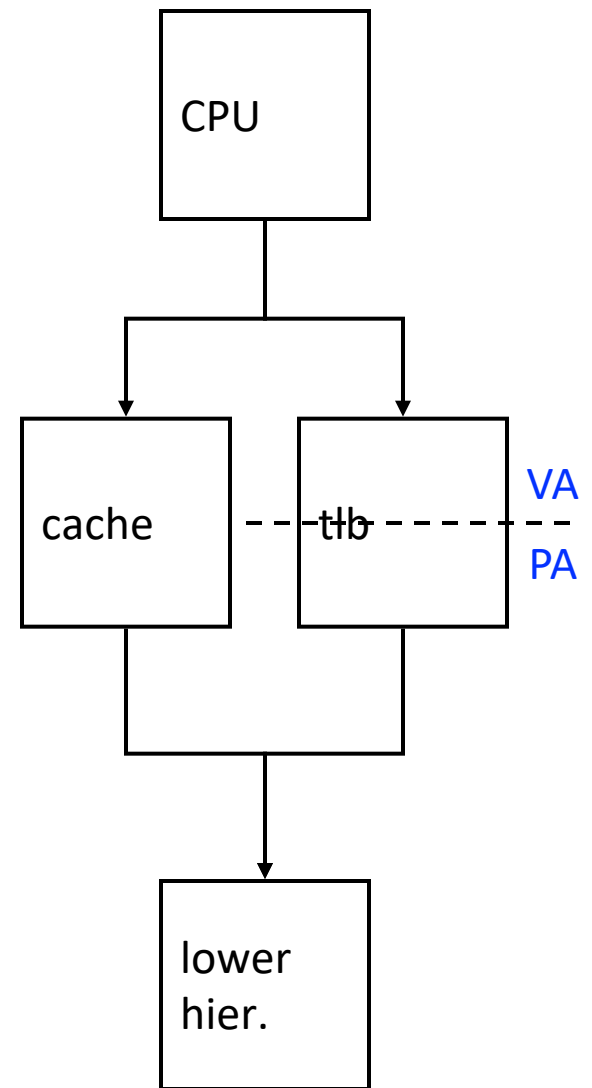
Cache-VM Interaction



physical cache

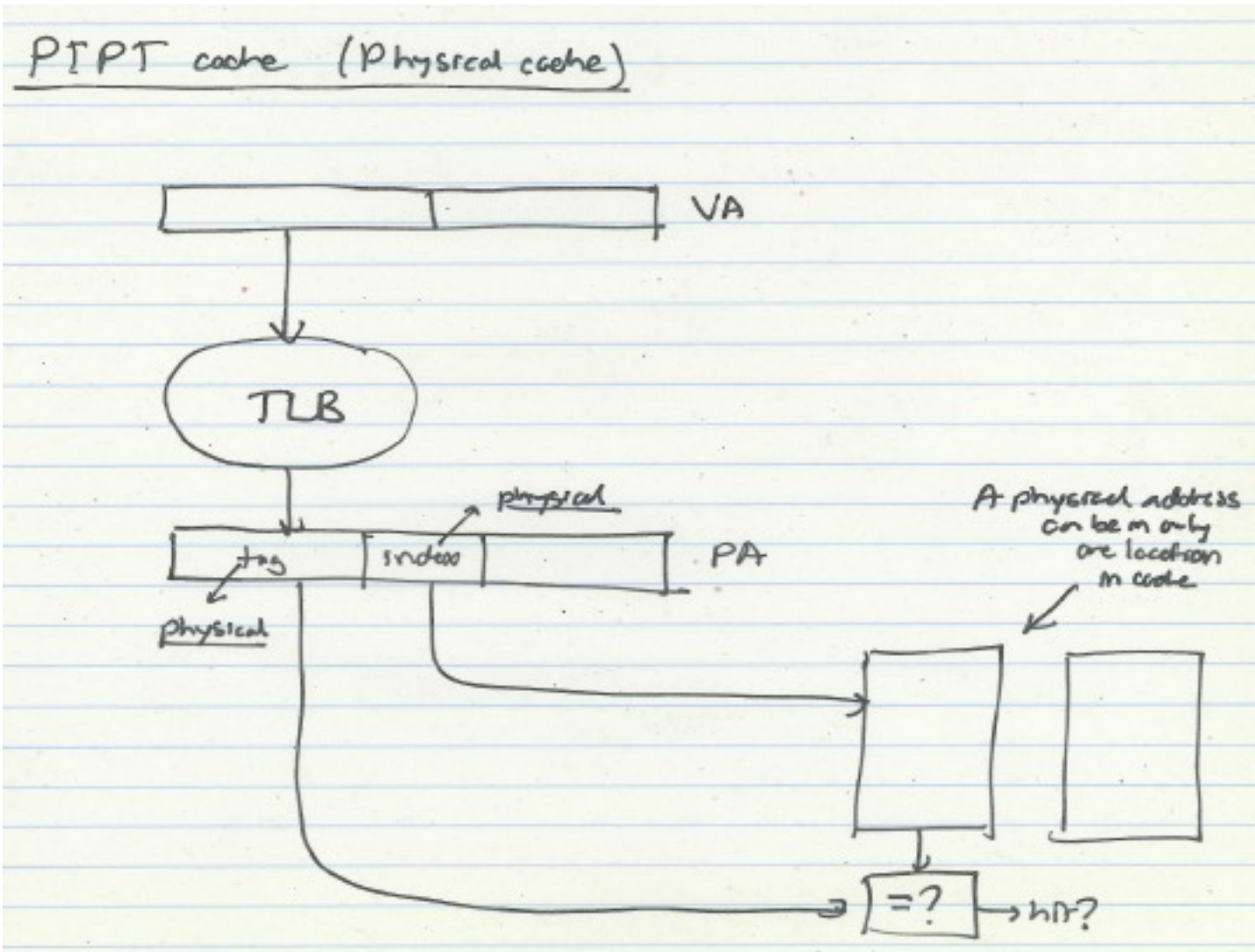


virtual (L1) cache



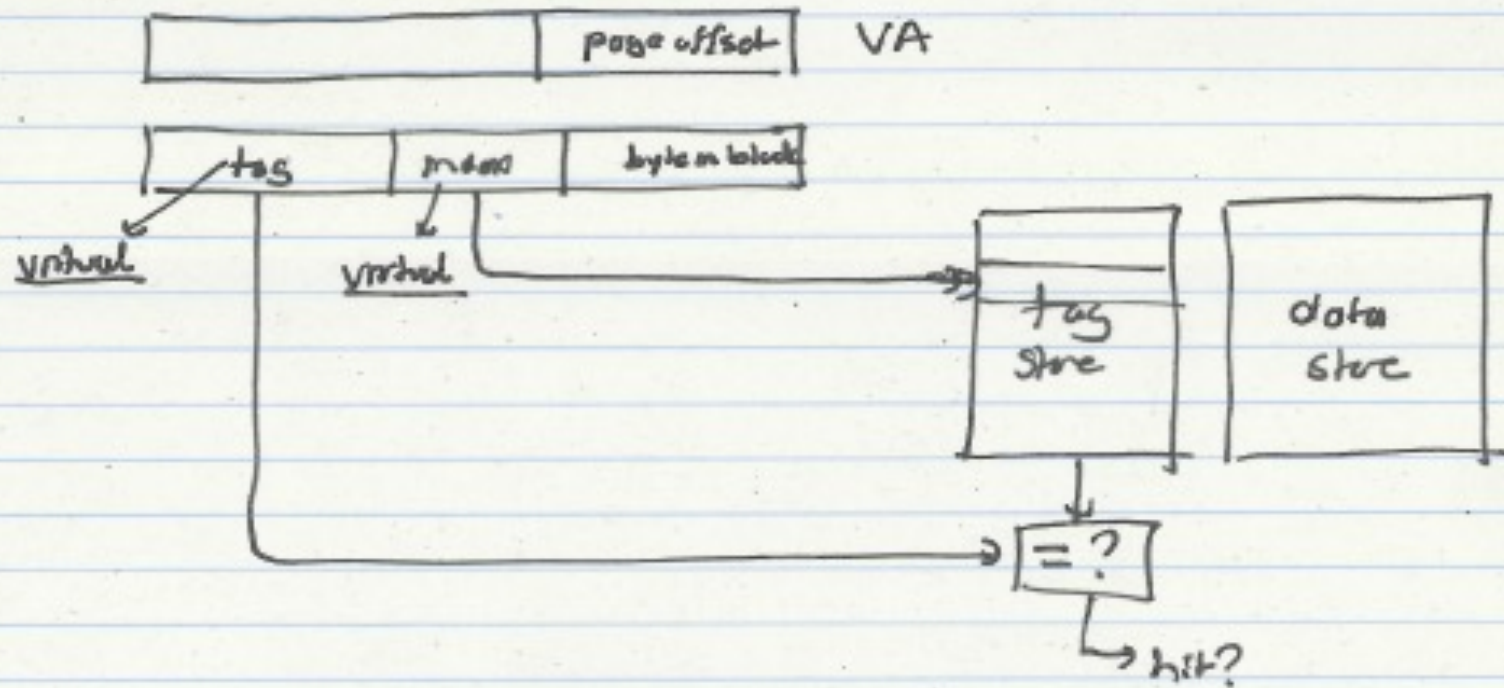
virtual-physical cache 49

Physical Cache



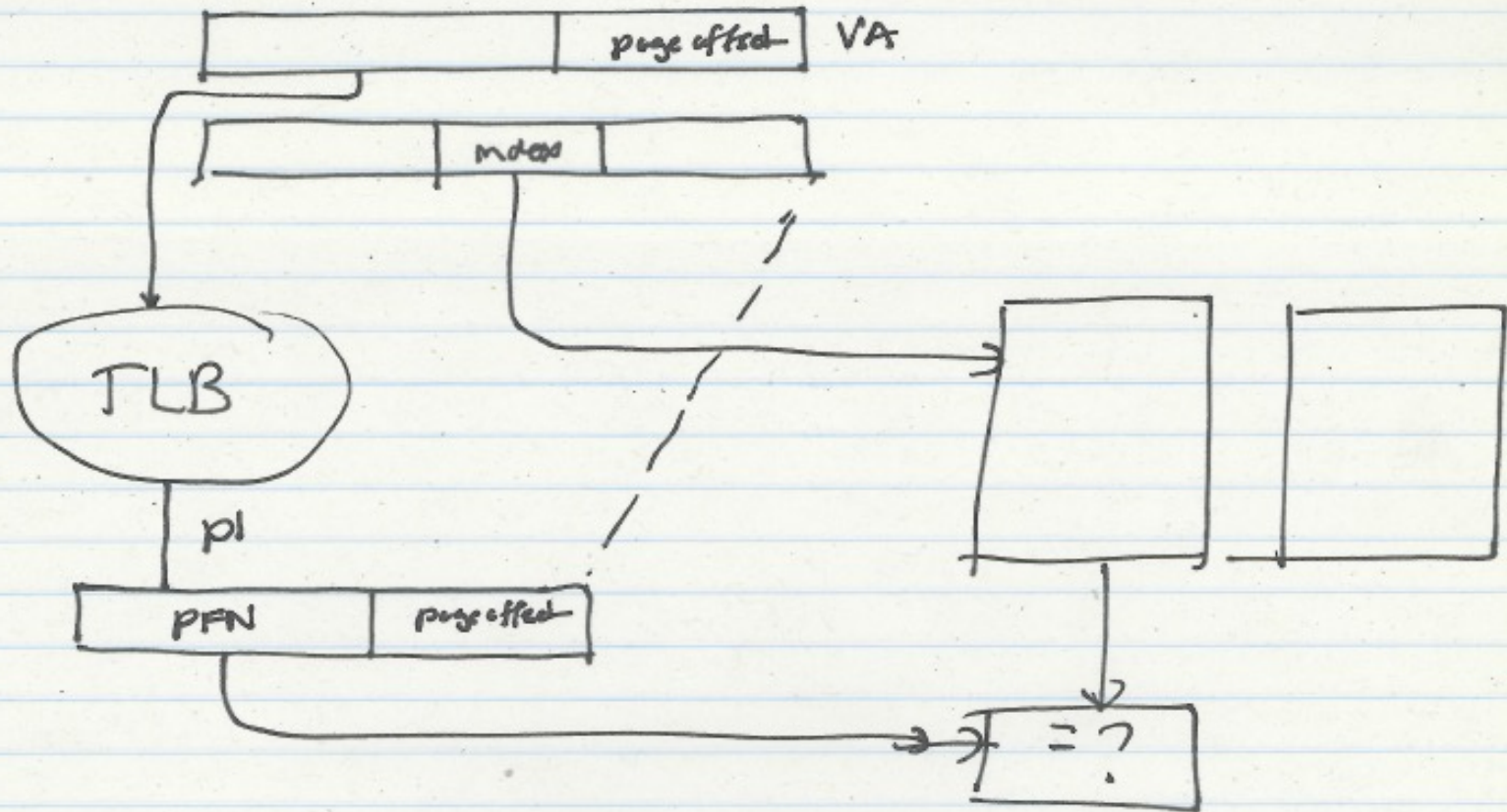
Virtual Cache

VINT cache (Virtual Cache)



Virtual-Physical Cache

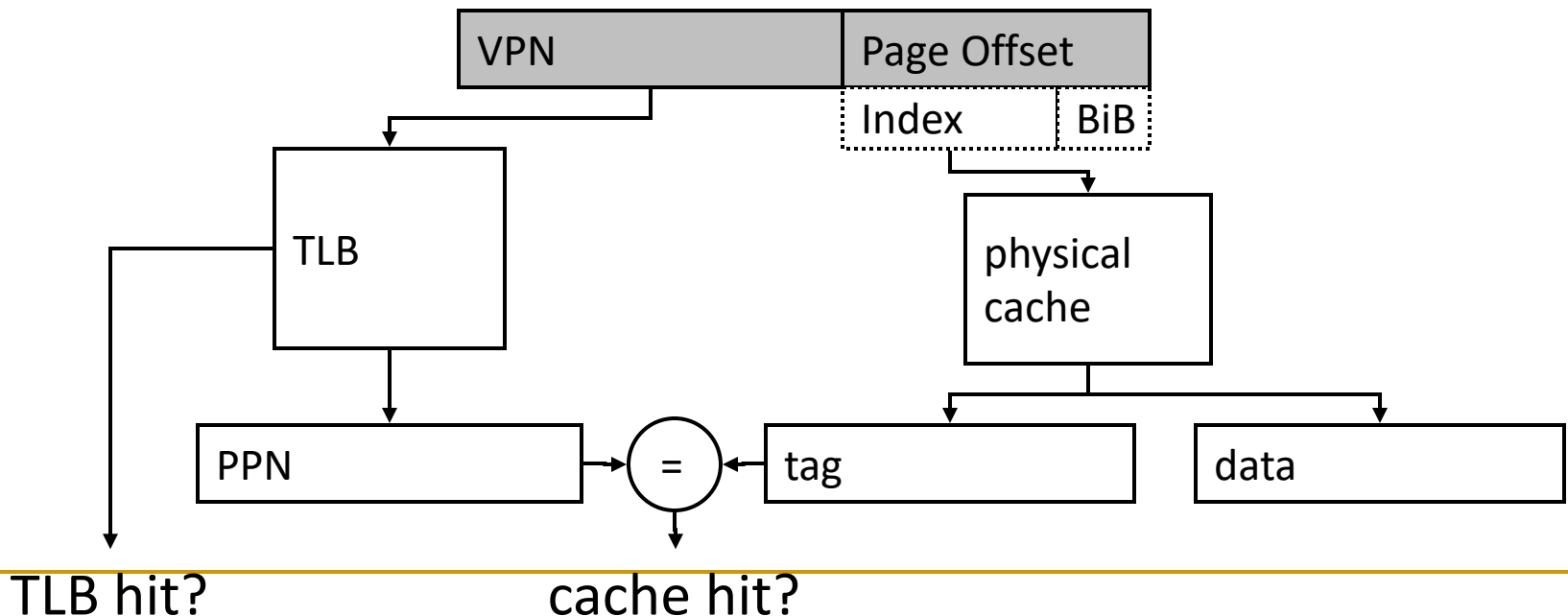
VIPT cache



Where can the same physical address be in the cache?

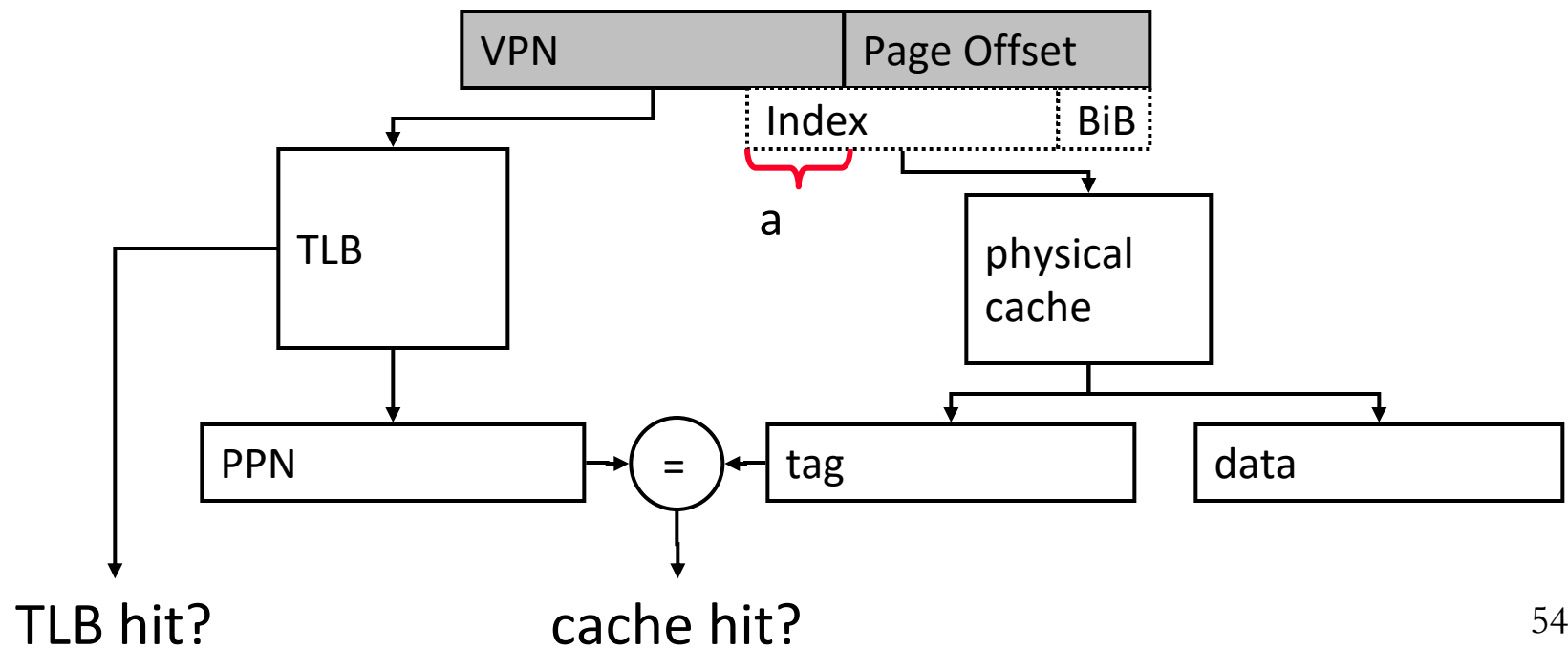
Virtually-Indexed Physically-Tagged

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



Virtually-Indexed Physically-Tagged

- If $C > (\text{page_size} \times \text{associativity})$, the cache index bits include VPN
⇒ Synonyms can cause problems
 - The same physical address can exist in two locations
- Solutions?



Some Solutions to the Synonym Problem

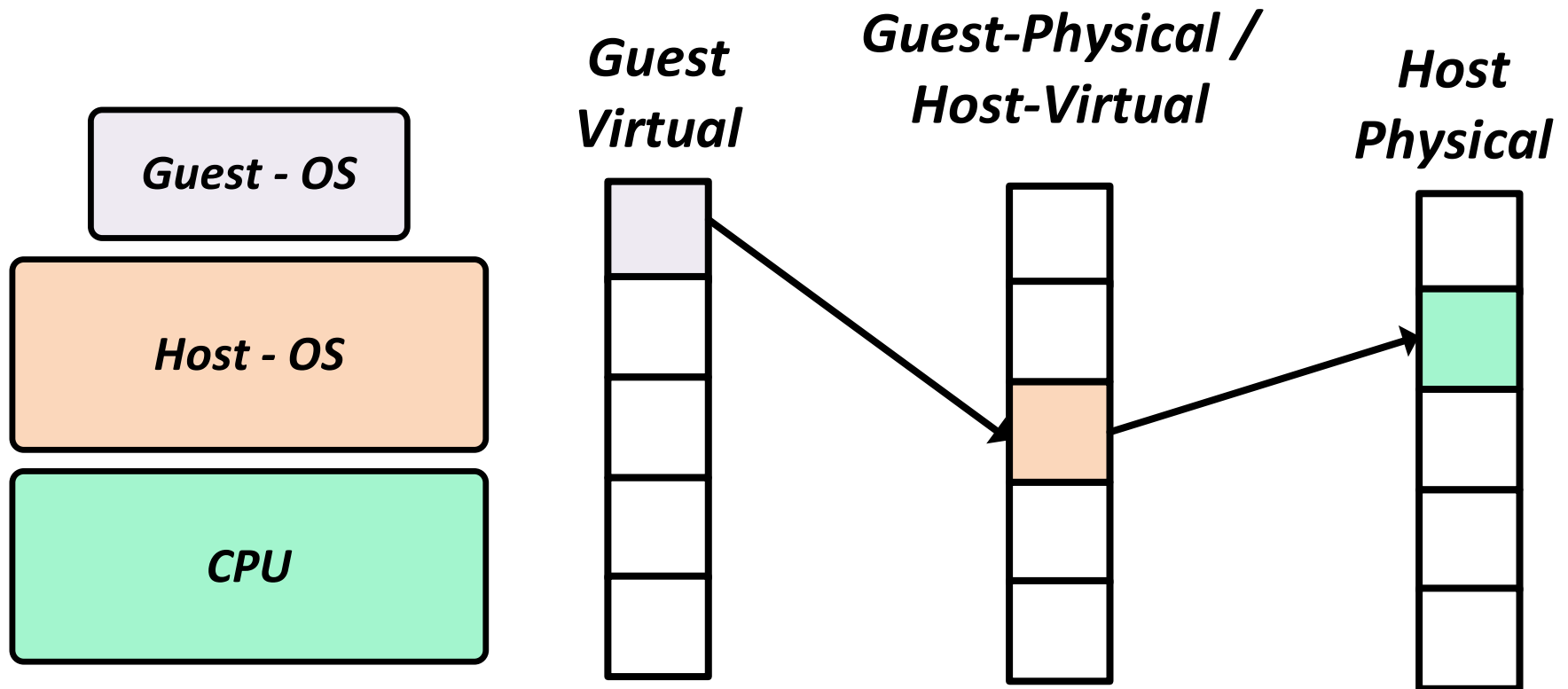
- Limit cache size to (page size times associativity)
 - get index from page offset
- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
 - Used in Alpha 21264, MIPS R10K
- Restrict page placement in OS
 - make sure $\text{index}(\text{VA}) = \text{index}(\text{PA})$
 - Called page coloring
 - Used in many SPARC processors

L1-D Cache in Intel Skylake

- 32 KB, 64B cacheline size, 8-way associative, 64 sets
- Virtually-indexed physically-tagged (VIPT)
- #set-index bits (6) + #offset-bits (6) = $\log_2(\text{Page Size})$
 - No synonym problem
- "SEESAW: Using Superpages to Improve VIPT Caches, Parasar+, ISCA'18
- [https://en.wikichip.org/wiki/intel/microarchitectures/skylake_\(server\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server))
- <https://uops.info/cache.html>
- <https://www.7-cpu.com/cpu/Skylake.html>

Virtual Memory in Virtualized Environments

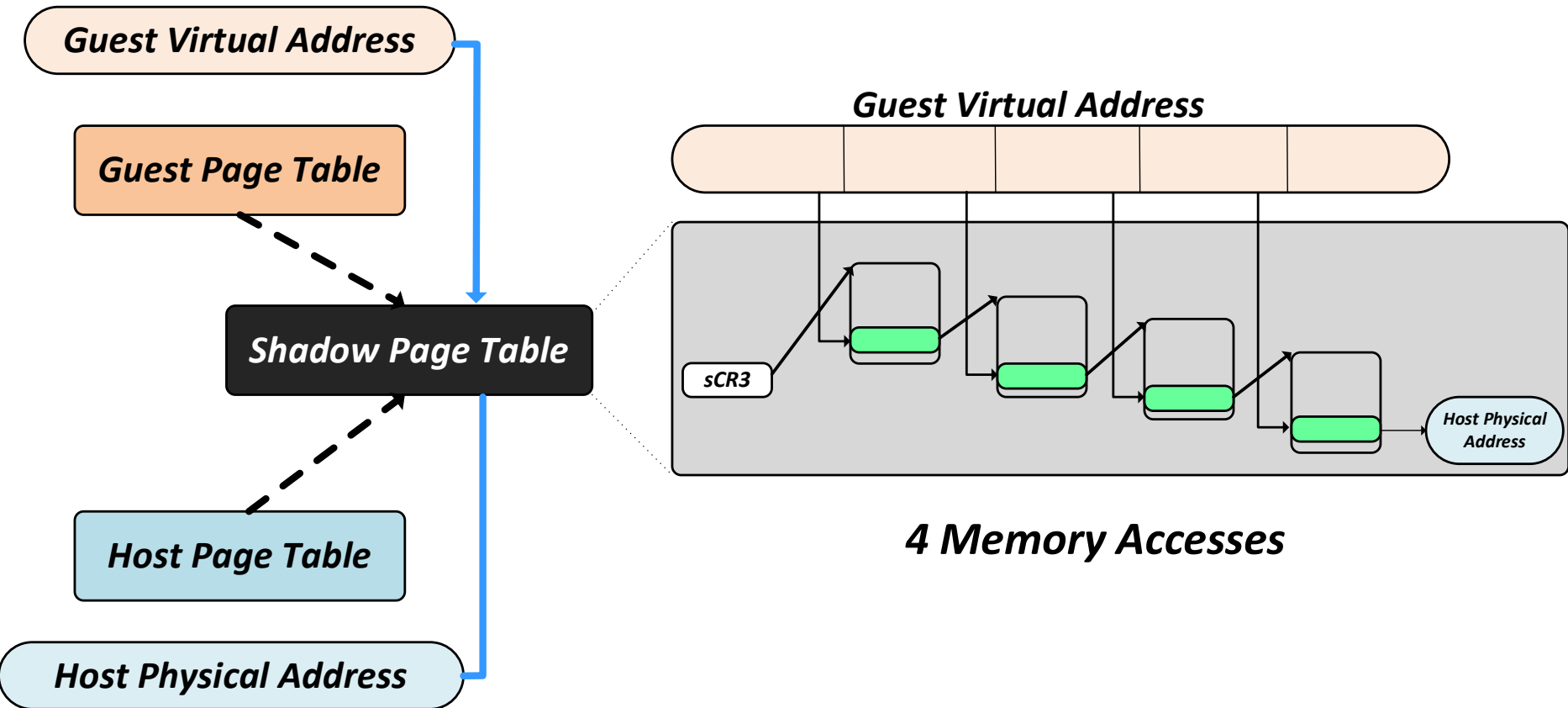
- Virtualized environments (e.g. Virtual Machines) need to have an additional level of address translation



Shadow Paging

- System maintains a new shadow page table which maps guest-virtual page directly to host-physical page
- Guest-virtual to Guest-physical page table is read-only for the Guest OS
- Pros:
 - + Fast TLB Miss / Page Table Walk
- Cons:
 - To maintain a consistent shadow page table, the system (e.g., VMM) handles every update to Guest and Host page tables

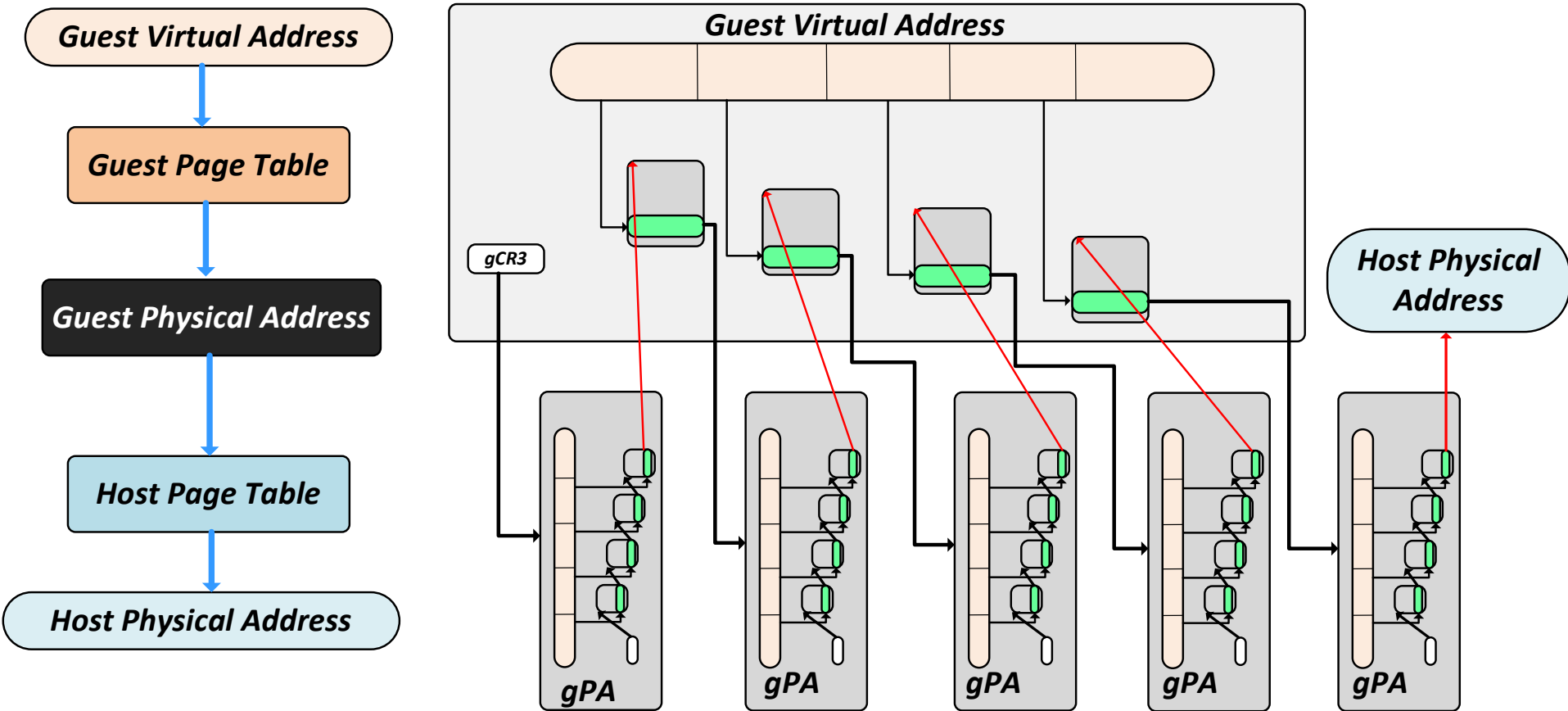
Shadow Paging



Nested Paging

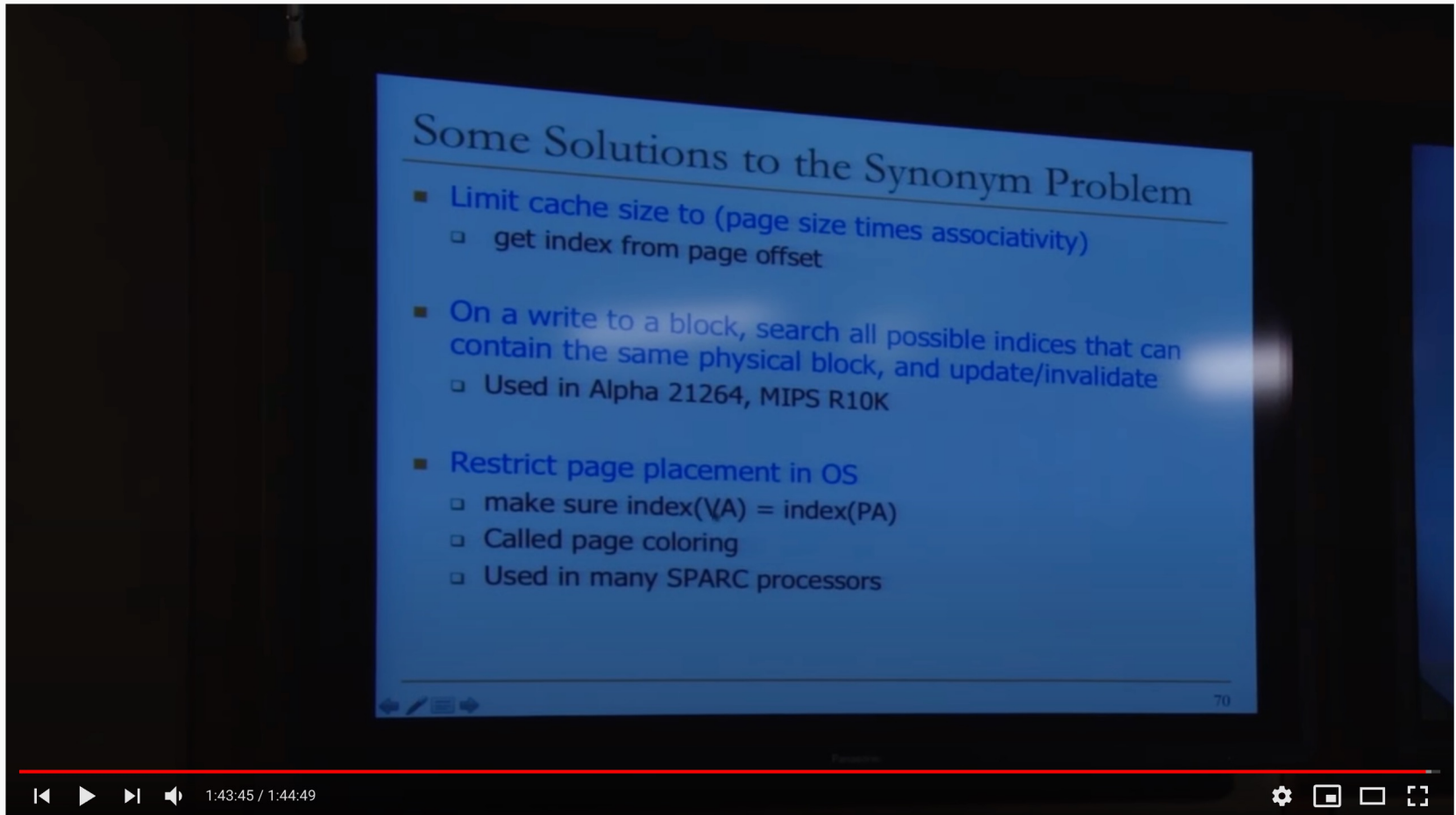
- Nested paging is the widely used hardware technique to virtualize memory in modern systems
- Two-dimensional hardware page-table walk:
 - For every level of Guest Page table
 - Perform a 4-level Host Page table walk
- Pros:
 - + Easy for the system to maintain/update two page tables
- Cons:
 - TLB Misses are more costly (up to 24 memory accesses)

Nested Paging



$5 + 5 + 5 + 5 + 4 = 24$ Memory Accesses

Lectures on Virtual Memory



Some Solutions to the Synonym Problem

- Limit cache size to (page size times associativity)
 - get index from page offset
- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
 - Used in Alpha 21264, MIPS R10K
- Restrict page placement in OS
 - make sure $\text{index}(\text{VA}) = \text{index}(\text{PA})$
 - Called page coloring
 - Used in many SPARC processors

70

Lecture 20. Virtual Memory - Carnegie Mellon - Comp. Arch. 2015 - Onur Mutlu

22,313 views • Mar 7, 2015

139 5 SHARE SAVE ...



Carnegie Mellon Computer Architecture
23.3K subscribers

SUBSCRIBED



Lectures on Virtual Memory

■ Computer Architecture, Spring 2015, Lecture 20

- Virtual Memory (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=2RhGMpY18zw&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=22>

■ Computer Architecture, Fall 2020, Lecture 12c

- The Virtual Block Interface (ETH, Fall 2020)
- <https://www.youtube.com/watch?v=PPR7YrBi7IQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=24>

P&S HW/SW Co-design

Lecture 3: Virtual Memory (II)

Konstantinos Kanellopoulos
Prof. Onur Mutlu

ETH Zurich
Spring 2022
13 April 2022