

Digital Design & Computer Arch.

Lecture 26a: Virtual Memory II

Prof. Onur Mutlu

ETH Zürich

Spring 2021

4 June 2021

Readings

■ Virtual Memory

■ Required

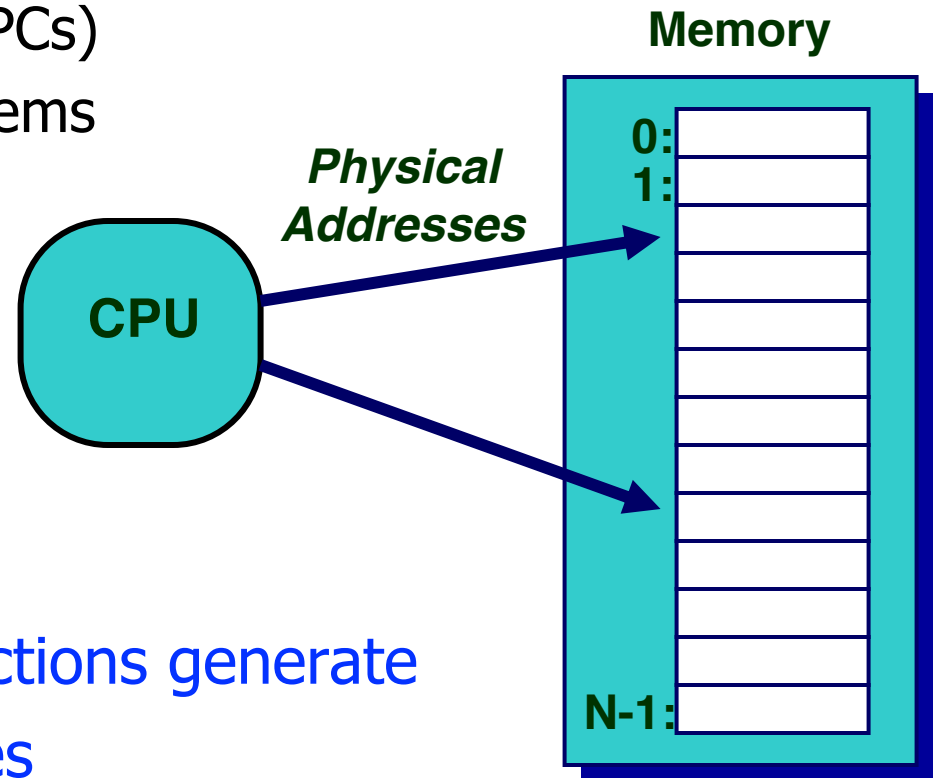
- H&H Chapter 8.4
- Kim & Mutlu, “**Memory Systems**,” Computing Handbook, 2014.
 - https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

■ Recommended

- Jacob & Mudge, “**Virtual Memory: Issues of Implementation**,” IEEE Computer, 1998.
- Hajinazar et al., “**The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework**,” ISCA 2020.

Recall: A System with Physical Memory Only

- Examples:
 - ❑ most Cray supercomputers
 - ❑ early personal computers (PCs)
 - ❑ many older embedded systems



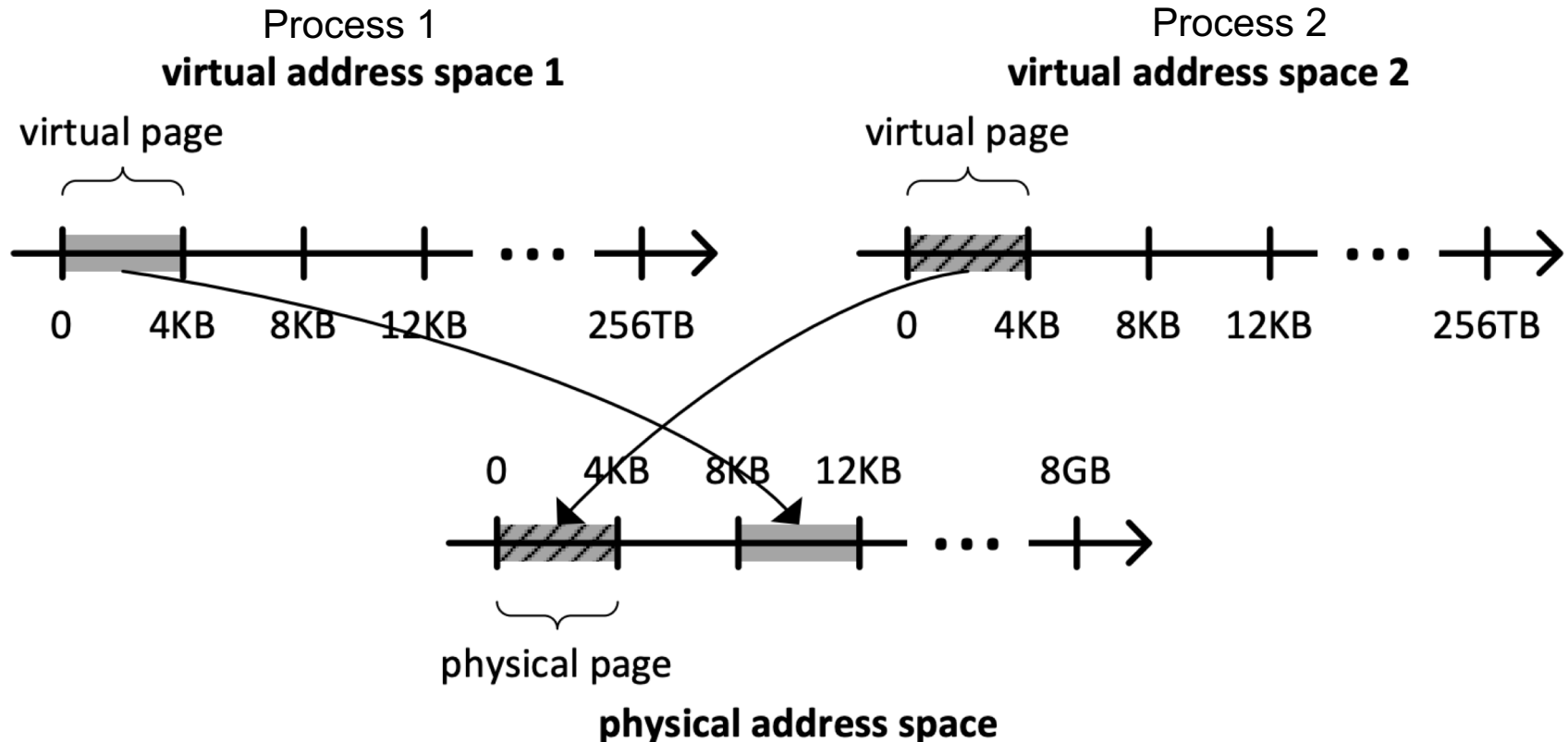
CPU's **load or store** instructions generate **physical** memory addresses

Recall: Virtual Memory

- Idea: Give each program the illusion of a large address space while having a small physical memory
 - So that the programmer does not worry about managing physical memory (within a process or across processes)
- Programmer can assume they have “infinite” amount of physical memory
- Hardware and software cooperatively and automatically manage the physical memory space to provide the illusion
 - Illusion is maintained for each independent process

Recall: Virtual Memory: Conceptual View

■ Illusion of large, separate address space per process

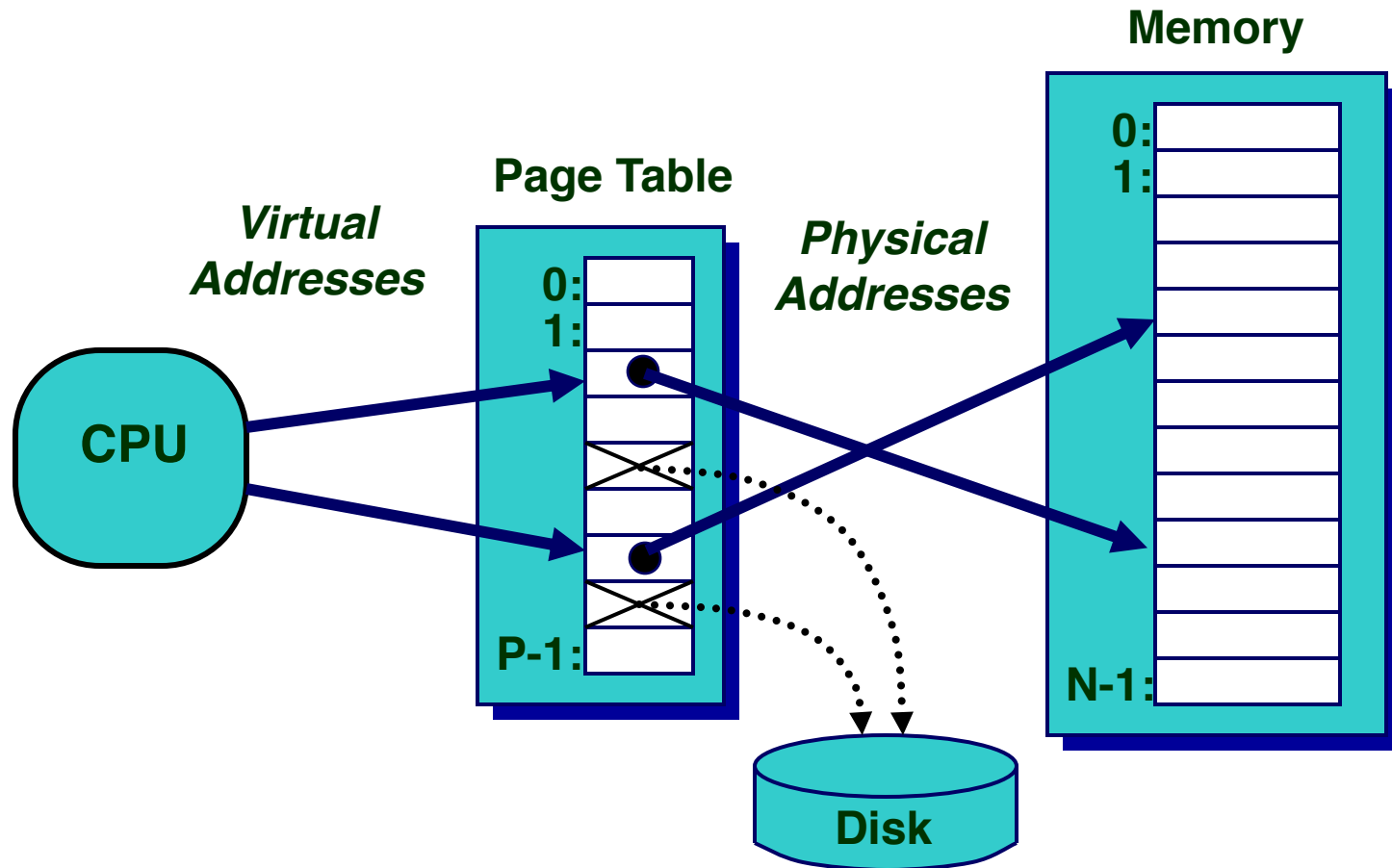


Requires **indirection and mapping** between virtual and physical spaces

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

Recall: 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)

Recall: Four Issues in Indirection and Mapping

- When to map a virtual address to a physical address?
 - When the virtual address is first referenced by the program
- What is the mapping granularity?
 - Byte? Kilo-byte? Mega-byte? ...
 - Multiple granularities?
- Where and how to store the virtual→physical mappings?
 - Operating system data structures? Hardware? Cooperative?
- What to do when physical address space is full?
 - Evict an unlikely-to-be-needed virtual address from physical memory

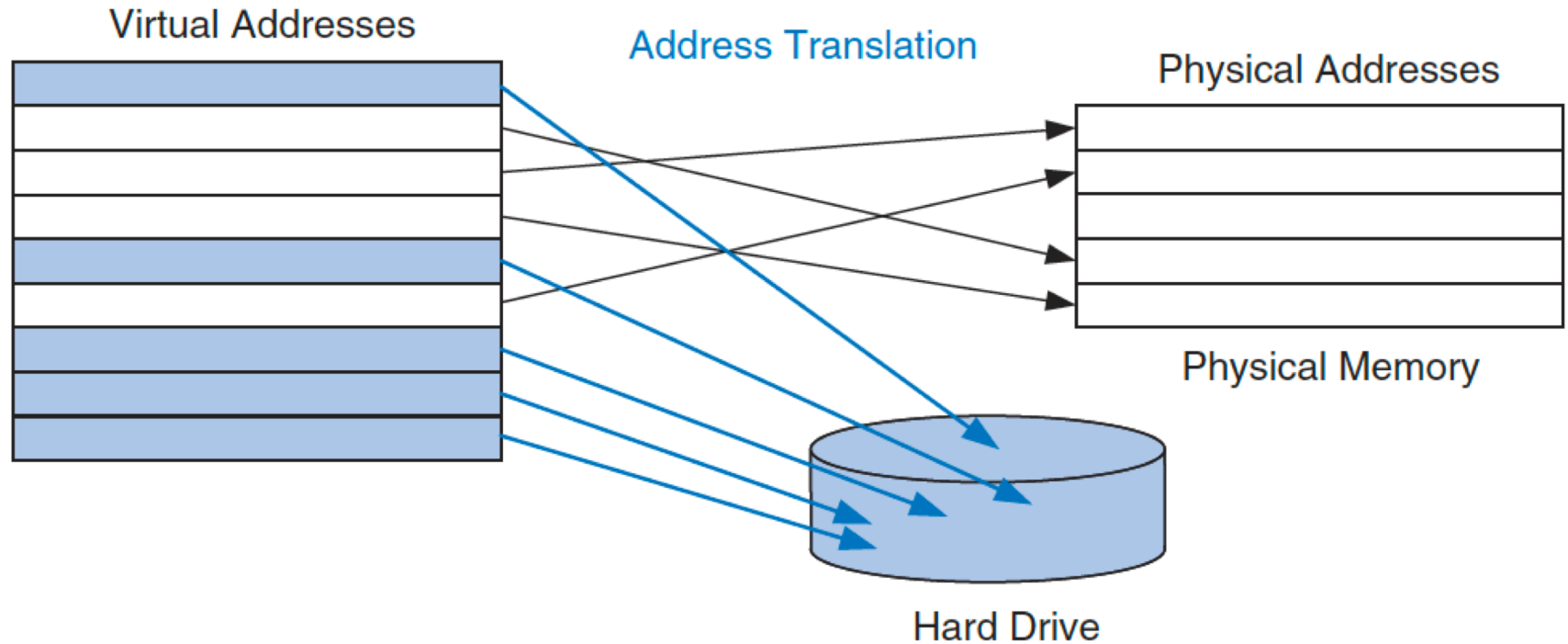
Recall: Physical Memory as a Cache

- In other words...
- Physical memory is a cache for pages stored on disk
 - In fact, it is a fully-associative cache in modern systems (a virtual page can potentially be mapped to any physical frame)
- Similar caching issues exist as we have covered earlier:
 - Placement: where and how to place/find a page in cache?
 - Replacement: what page to remove to make room in cache?
 - Granularity of management: large, small, uniform pages?
 - Write policy: what do we do about writes? Write back?

Recall: Virtual Memory Definitions

- **Page size**: the mapping granularity of virtual→physical address spaces
 - dictates the amount of data transferred from hard disk to DRAM at once
- **Page table**: table that stores virtual→physical page mappings
 - lookup table used to translate virtual page addresses to physical frame addresses (and find where the associated data is)
- **Address translation**: the process of determining the physical address from the virtual address

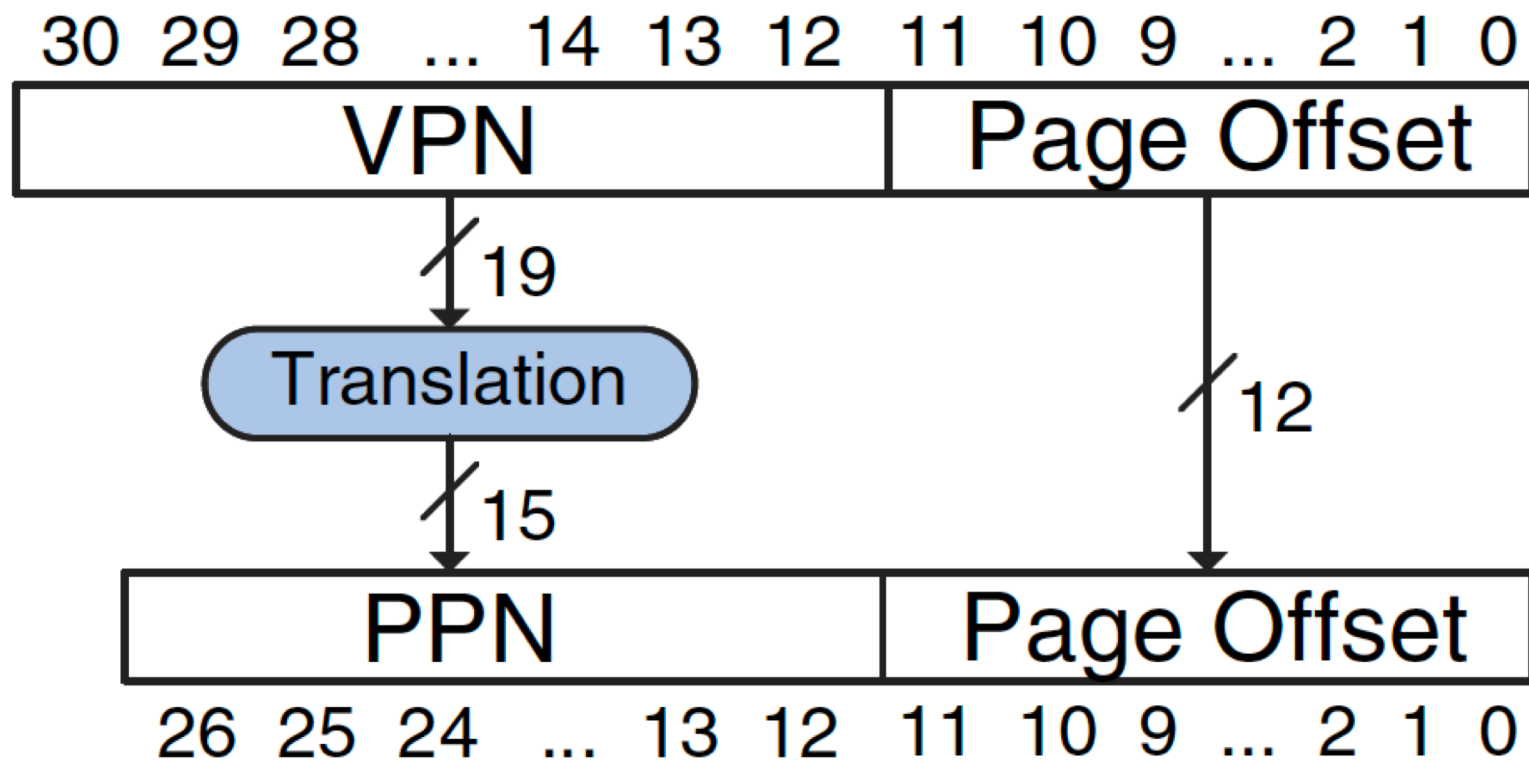
Recall: Virtual to Physical Mapping



- Most accesses hit in physical memory
- Programs see the large capacity of virtual memory

Recall: Address Translation

Virtual Address



Physical Address

Recall: Virtual Memory Example

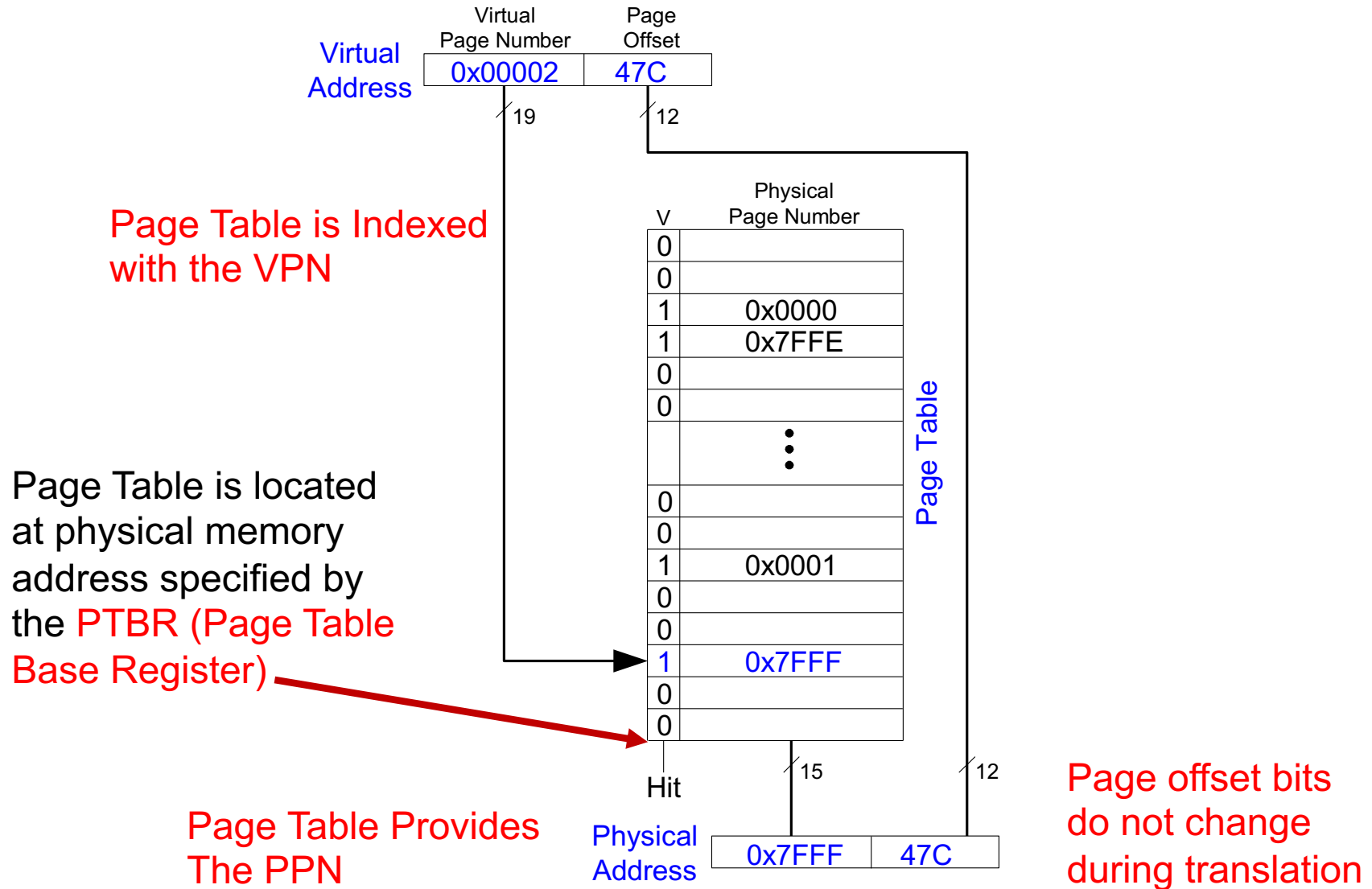
■ System:

- ❑ Virtual memory size: 2 GB = 2^{31} bytes
- ❑ Physical memory size: 128 MB = 2^{27} bytes
- ❑ Page size: 4 KB = 2^{12} bytes

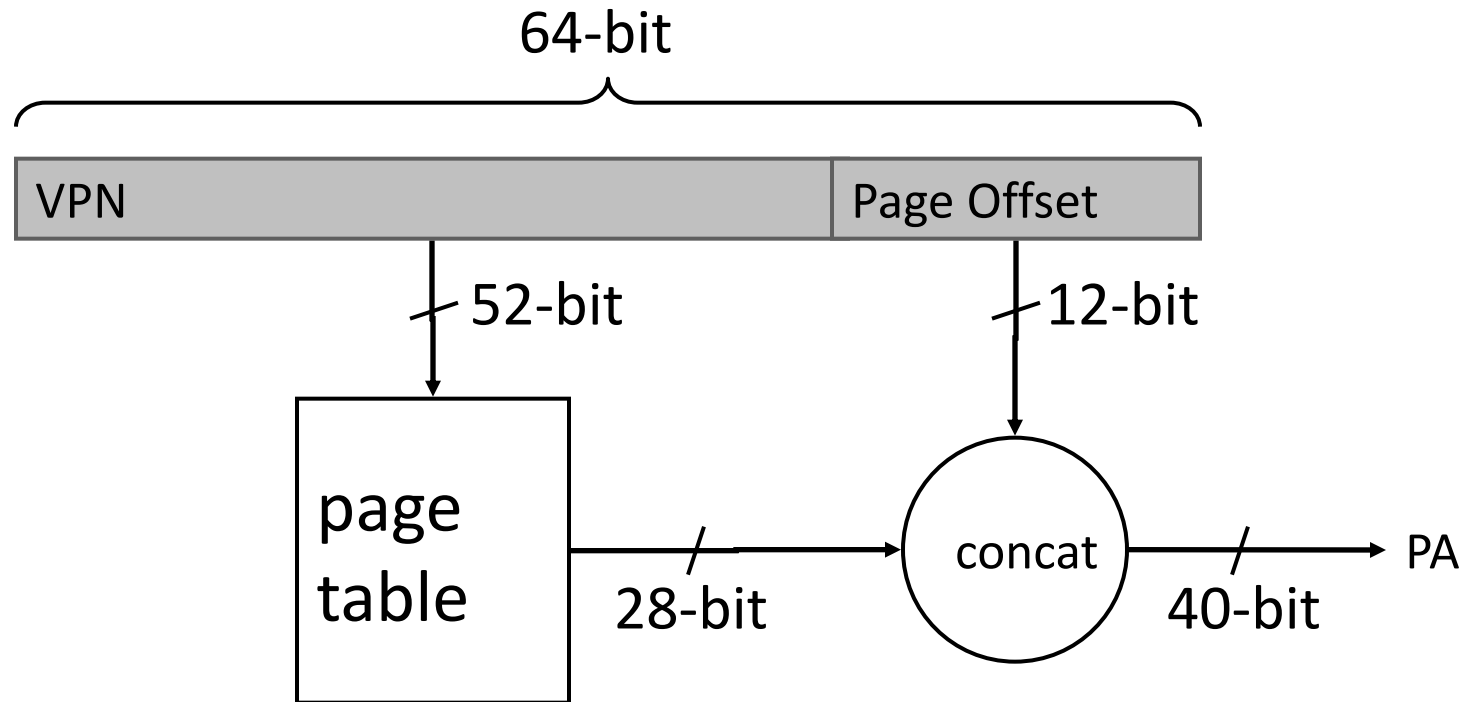
■ Organization:

- ❑ Virtual address: **31** bits
- ❑ Physical address: **27** bits
- ❑ Page offset: **12** bits
- ❑ # Virtual pages = $2^{31}/2^{12} = 2^{19}$ (VPN = 19 bits)
- ❑ # Physical pages = $2^{27}/2^{12} = 2^{15}$ (PPN = 15 bits)

Recall: Page Table Address Translation Example



Recall: Issue: Page Table Size



- Suppose 64-bit VA and 40-bit PA, how large is the page table?
 - **2^{52} entries x ~4 bytes $\approx 2^{54}$ bytes**
and that is for just one process!
and the process may not be using the entire VM space!

Page Table Challenges (I)

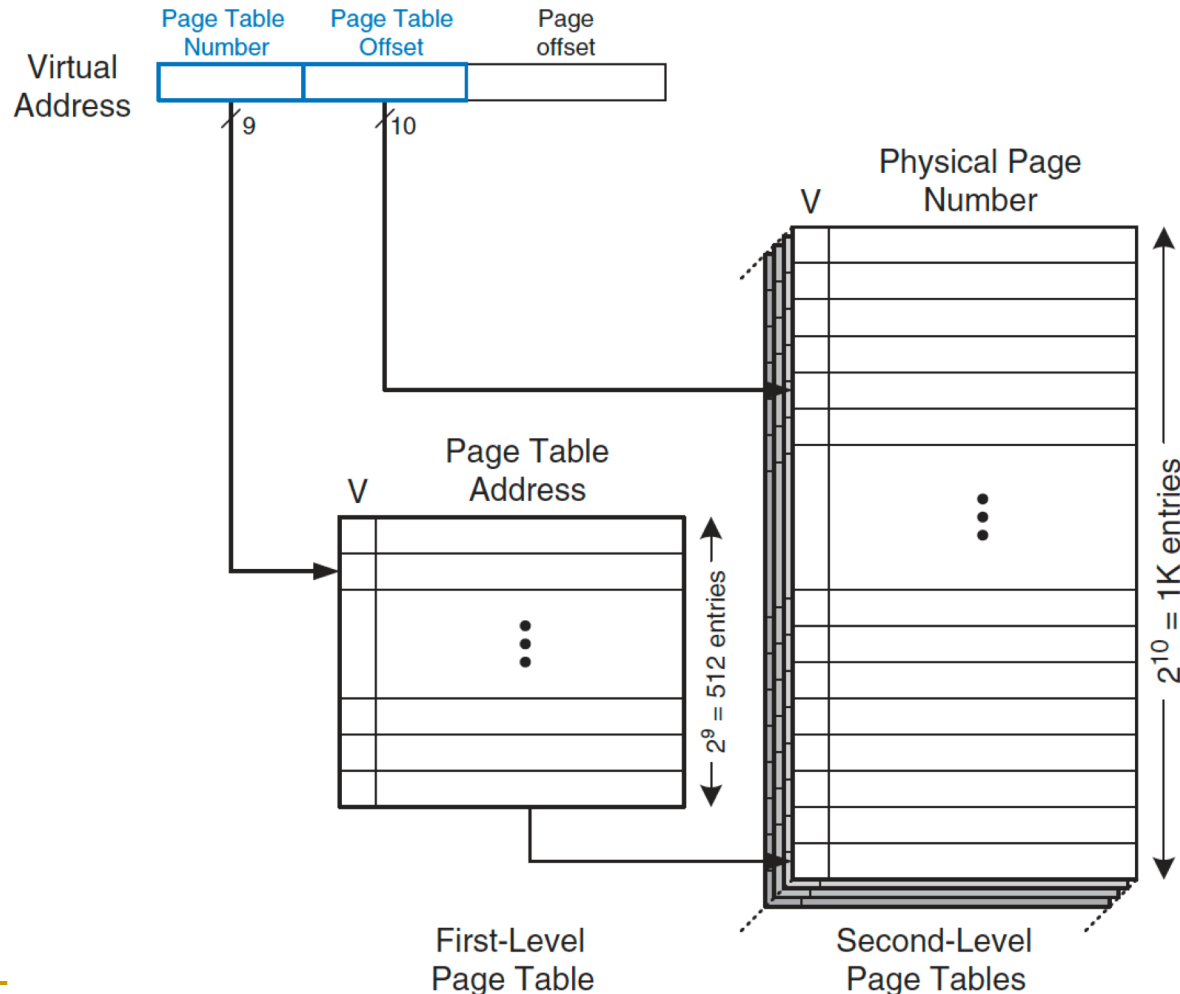
- Challenge 1: **Page table is large**
 - ❑ at least part of it needs to be located in physical memory
 - ❑ solution: **multi-level (hierarchical) page tables**

Multi-Level Page Tables

- Idea: Organize page table in a hierarchical manner such that only a small first-level page table has to be in physical memory
- Multi-level (hierarchical) page tables

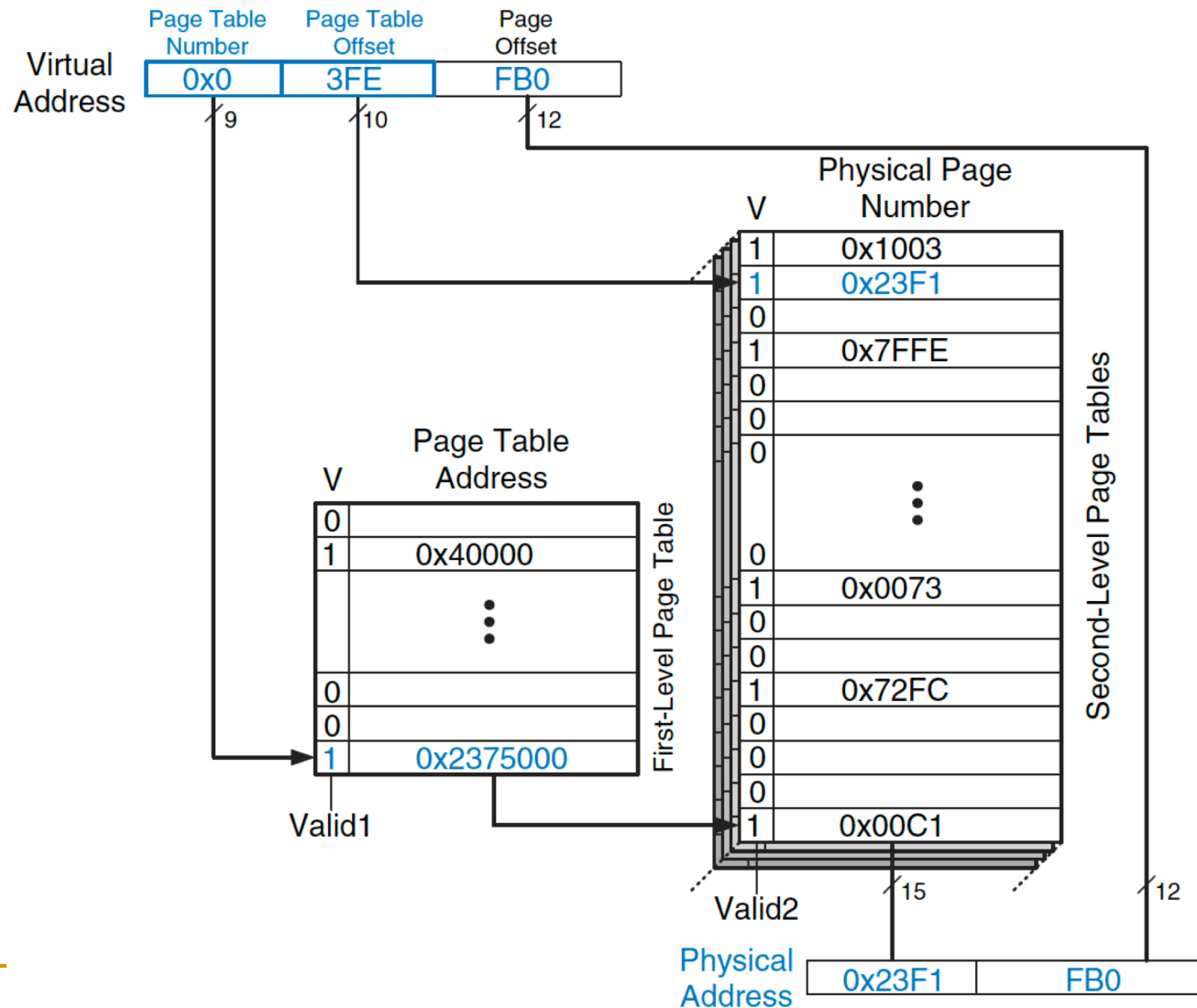
Multi-Level Page Table Example

- First-level page table has to be in physical memory
- Only the needed second-level page tables can be kept in physical memory



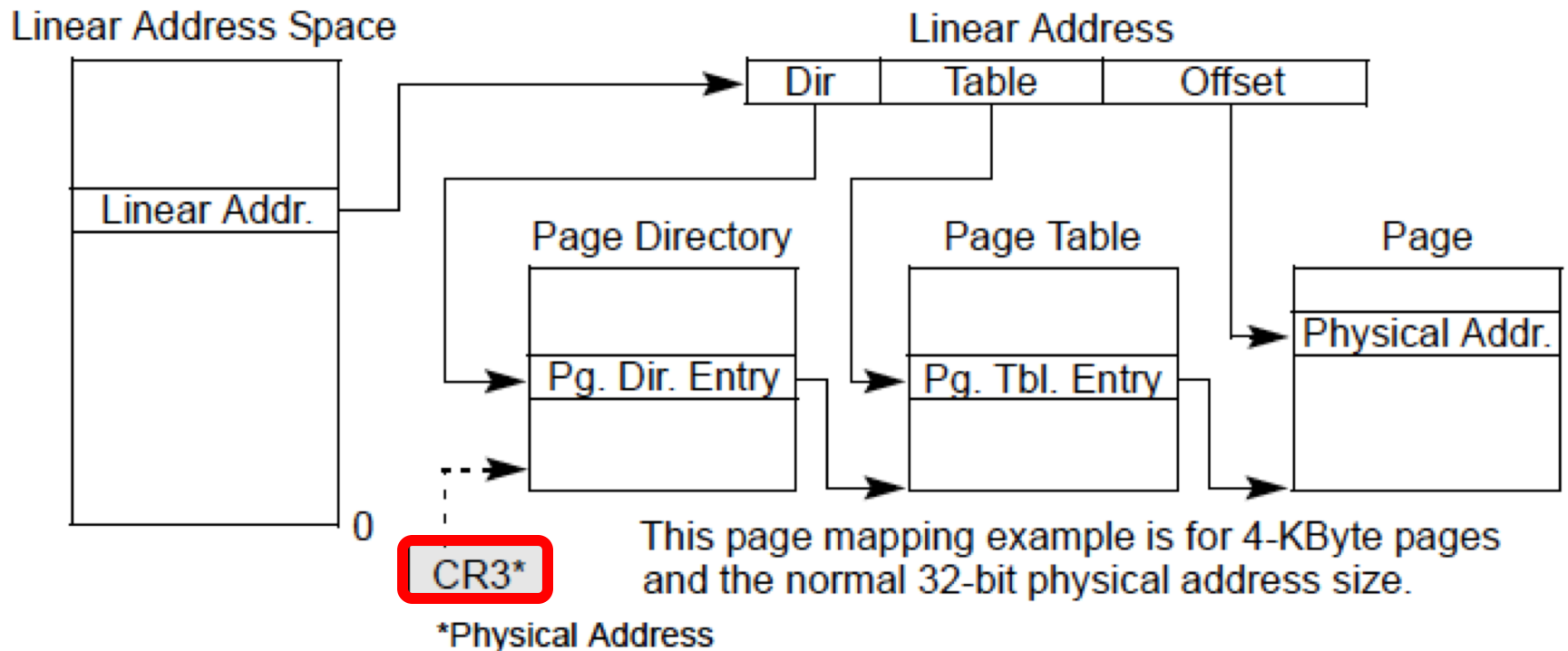
Multi-Level Page Table: Address Translation

- For N-level page table, we need N page table accesses to find the PTE



Multi-Level Page Tables from x86 Manual

Example from the x86 architecture



CR3: Control Register 3 (or Page Directory Base Register)

x86 Page Tables (I): Small Pages

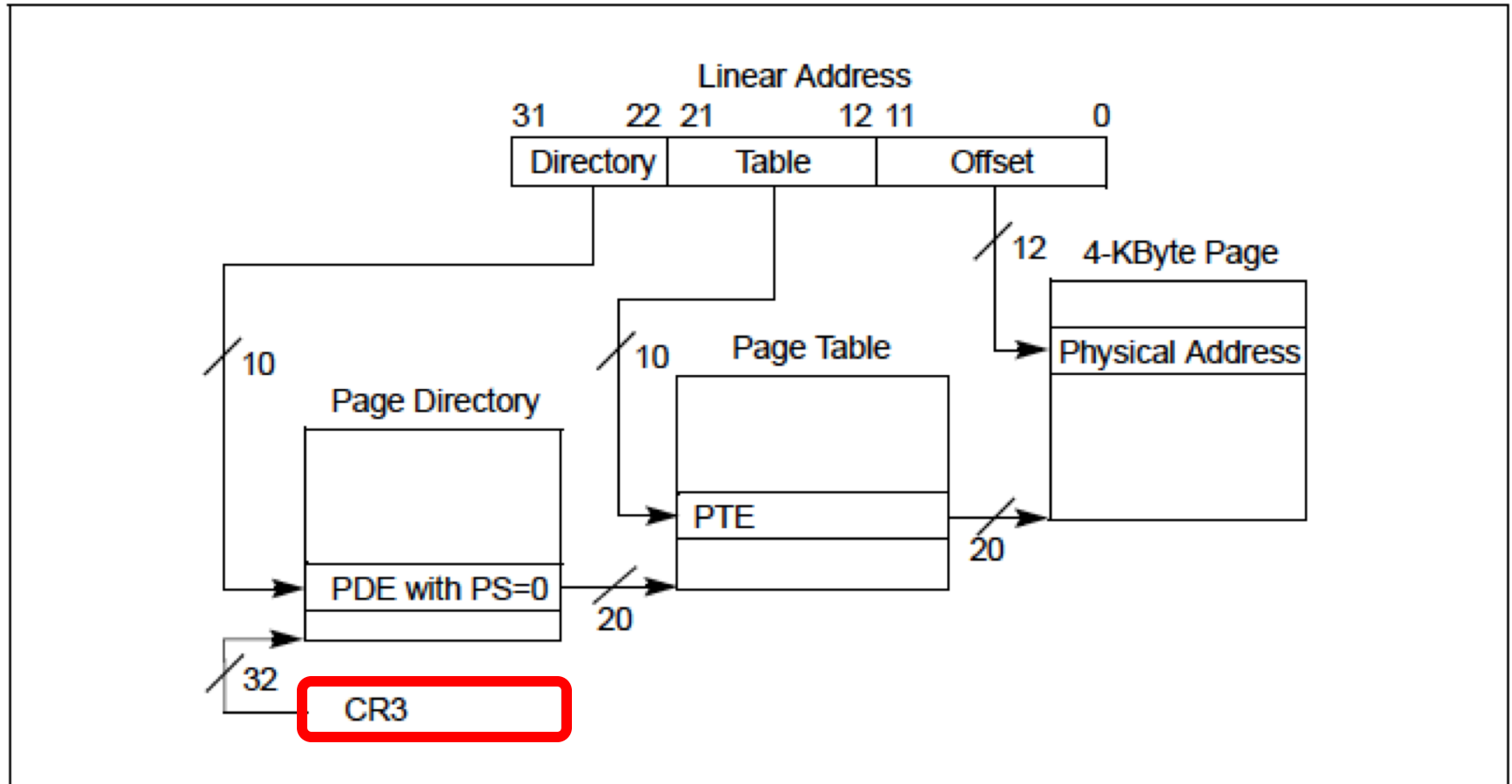


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

x86 Page Tables (II): Large Pages

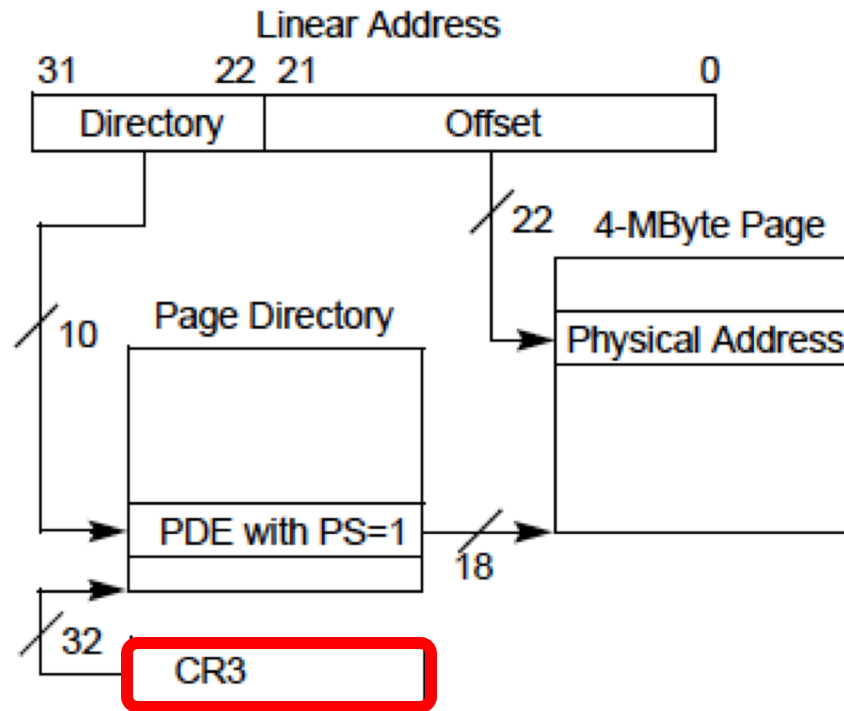


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

Four-level Paging in x86-64

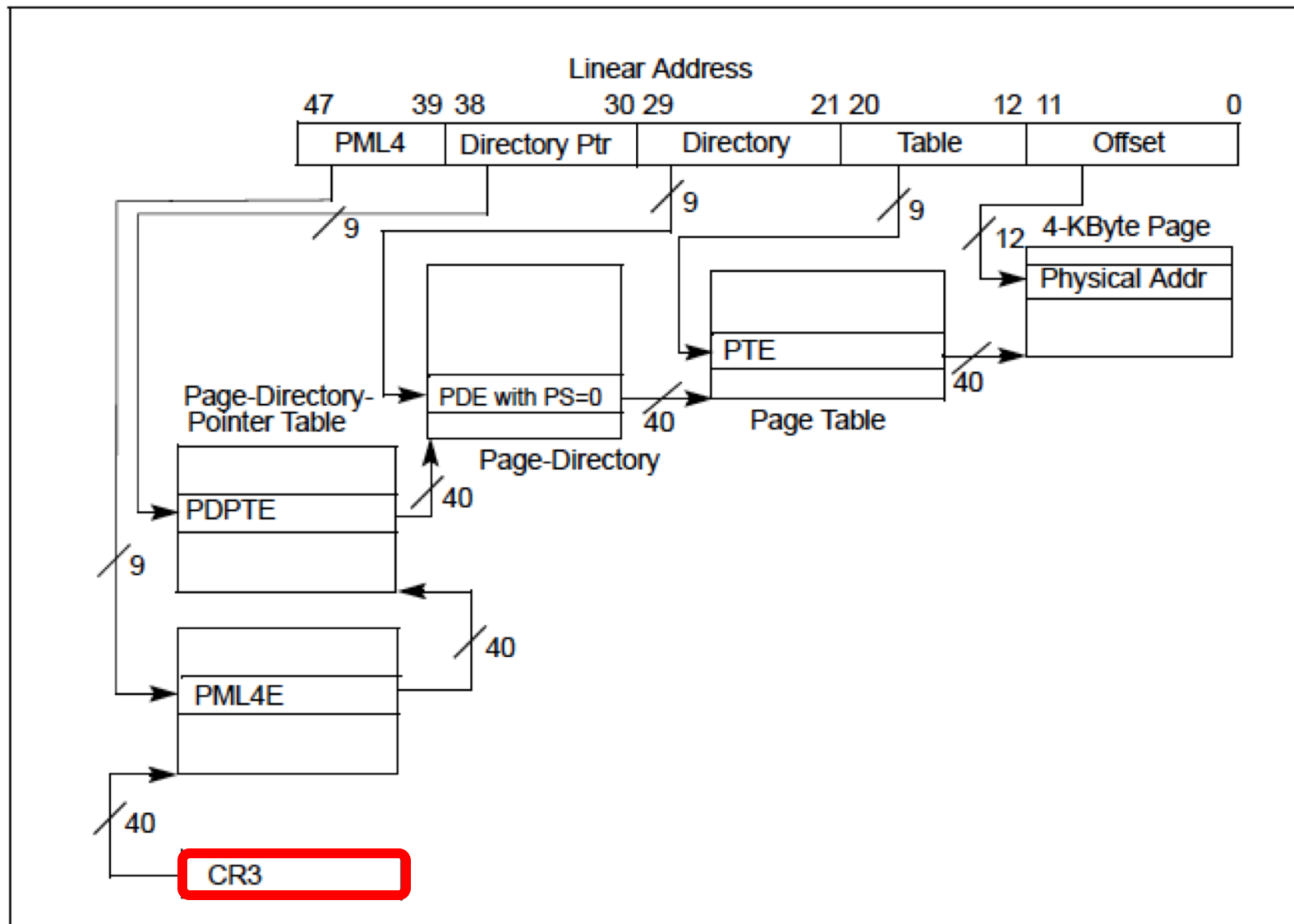


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Page Table Challenges (II)

- 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...**

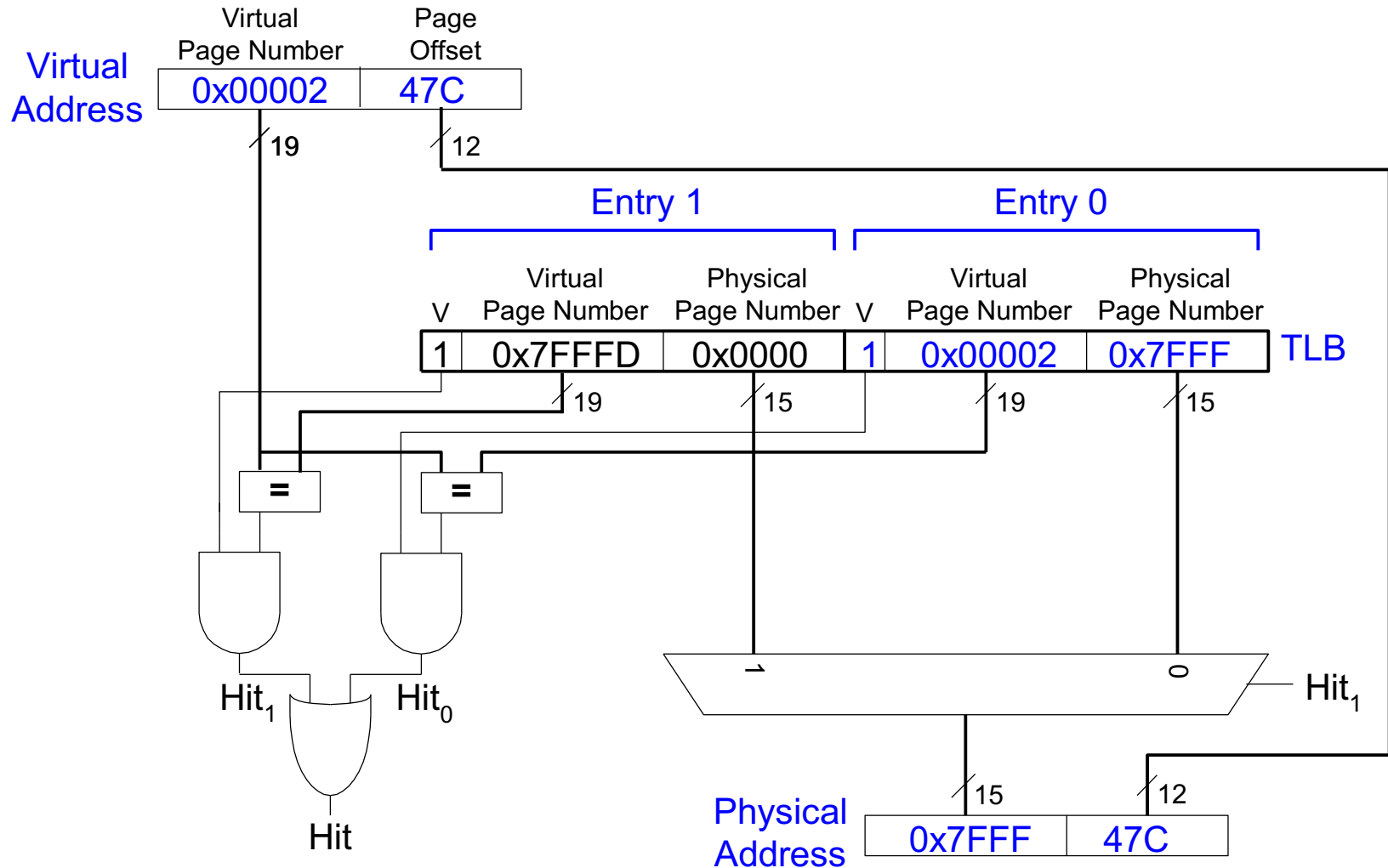
Translation Lookaside Buffer (TLB)

- Idea: Cache the page table entries (PTEs) in a hardware structure in the processor to speed up address translation
- Translation lookaside buffer (TLB)
 - Small cache of most recently used translations (PTEs)
 - Reduces number of memory accesses required for *most* instruction fetches and loads/stores to only one

Translation Lookaside Buffer (TLB)

- Page table accesses have a lot of temporal and spatial locality
 - ❑ Memory accesses have temporal and spatial locality
 - ❑ Large page sizes aid spatial locality (4KB, 8KB, MBs, GBs)
 - ❑ Consecutive instructions and loads/stores are likely to access same page
- TLB: cache of page table entries (i.e., translations)
 - ❑ Small: accessed in ~ 1 cycle
 - ❑ Typically 16 - 512 entries at level 1
 - ❑ Usually high associativity
 - ❑ $> 90\text{-}99\%$ hit rates typical (depends on workload)
 - ❑ Reduces number of memory accesses for most instruction fetches and loads/stores to only one

Example Two-Entry TLB



TLB is a Translation (PTE) Cache

- All issues we discussed in caching and prefetching lectures apply to TLBs
- Example issues:
 - Instruction vs. Data TLBs
 - Multi-level TLBs
 - Associativity and size choices and tradeoffs
 - Insertion, promotion, replacement policies
 - What to keep in which TLB and how to decide that
 - Prefetching into the TLBs
 - TLB coherence
 - Shared vs. private TLBs across cores/threads
 - ...

Virtual Memory Support and Examples

Supporting Virtual Memory

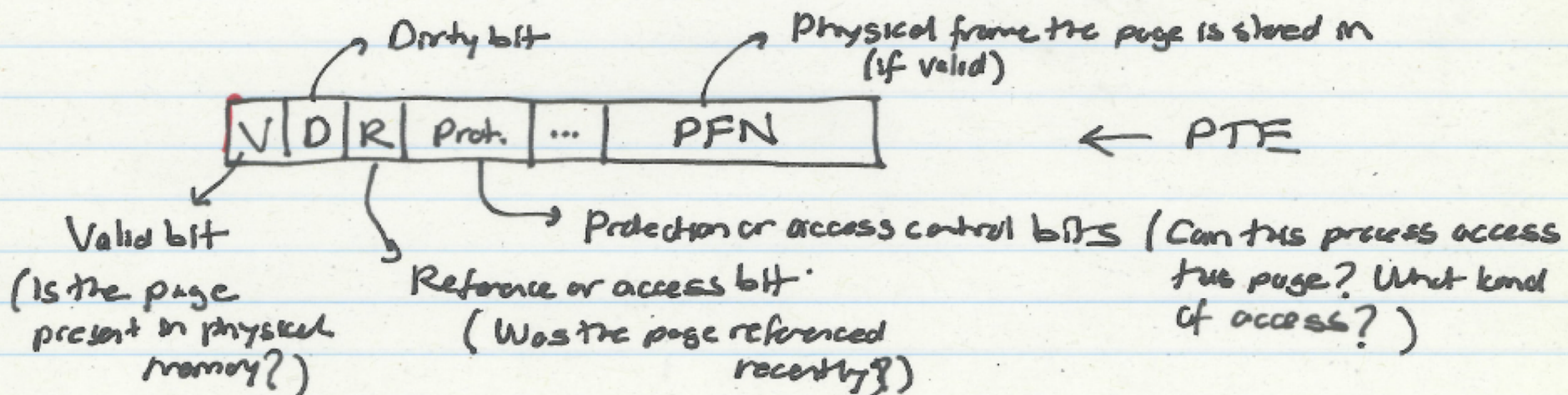
- Virtual memory **requires both HW+SW support**
 - Page Table is in memory
 - Can be cached in special hardware structures called Translation Lookaside Buffers (TLBs)
- The hardware component is called the **MMU** (memory management unit)
 - Includes Page Table Base Register(s), TLBs, page walkers
- **It is the job of the software** to leverage the MMU 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

Address Translation

- How to obtain the physical address from a virtual address?
- Page size specified by the ISA
 - VAX: 512 bytes
 - Today: 4KB, 8KB, 2GB, ... (small and large pages mixed together)
 - Trade-offs? (remember cache lectures)
- Page Table contains an entry for each virtual page
 - Called Page Table Entry (PTE)
 - What is in a PTE?

What Is in a Page Table Entry (PTE)?

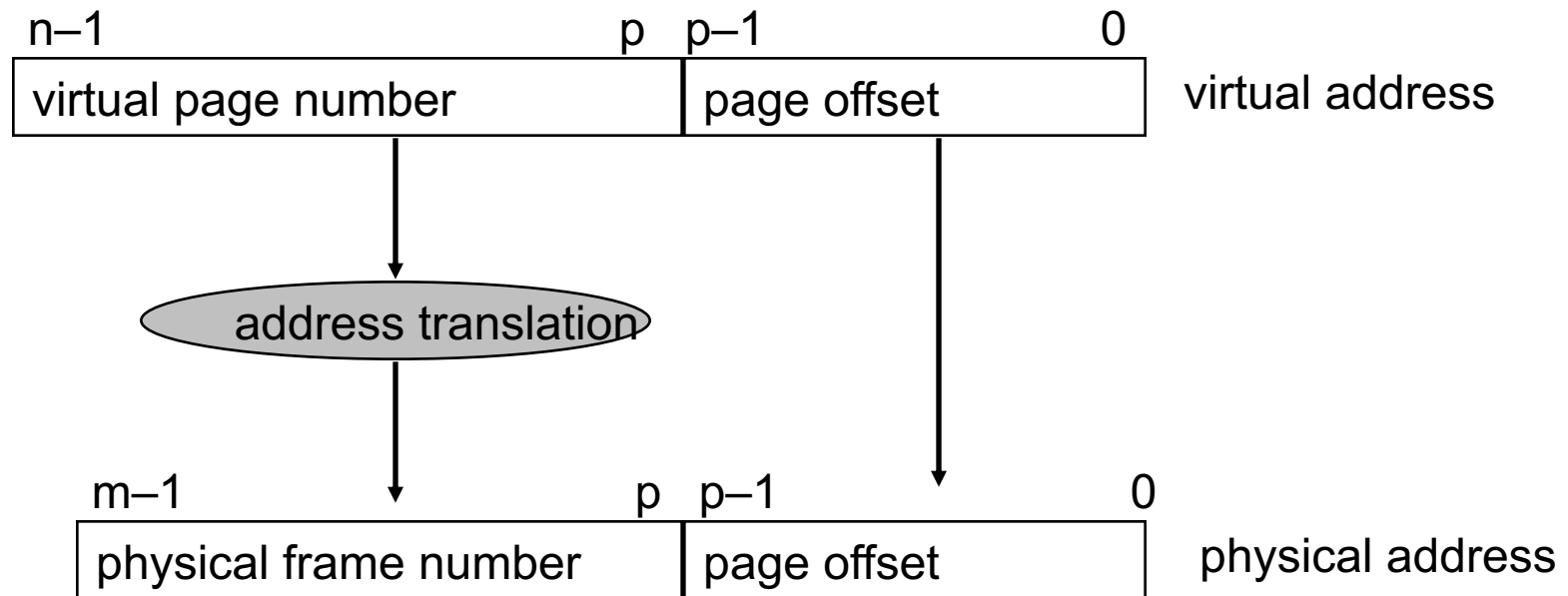
- Page table is the “tag store” for the physical memory data store
 - A mapping table between virtual memory and physical memory
- PTE is the “tag store entry” for a virtual page in memory
 - Need a **valid** bit → to indicate validity/presence in physical memory
 - Need **tag** bits (PFN) → to support translation
 - Need bits to support **replacement**
 - Need a **dirty** bit to support “write back caching”
 - Need **protection bits** to enable access control and protection



Recall: Address Translation (I)

■ Parameters

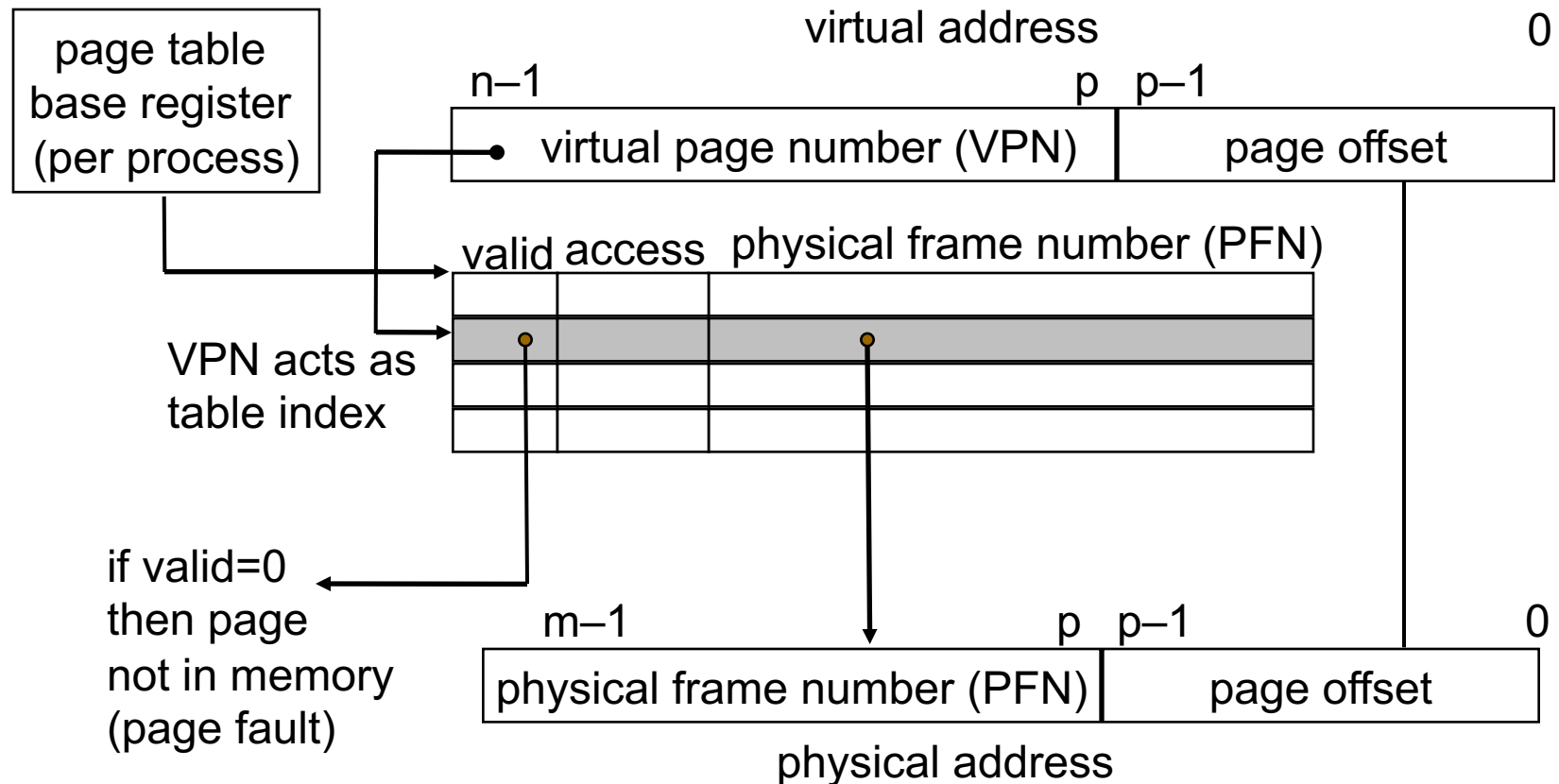
- $P = 2^p =$ page size (bytes)
- $N = 2^n =$ Virtual-address limit
- $M = 2^m =$ Physical-address limit



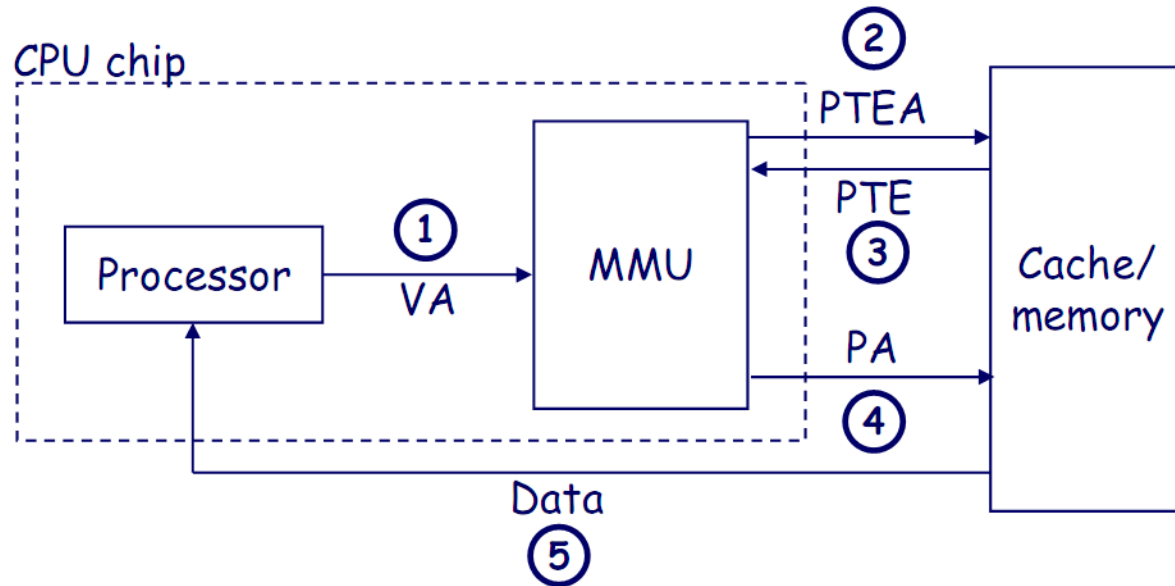
Page offset bits do not change as a result of translation

Recall: Address Translation (II)

- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)
- Page Table Entry (PTE) provides information about page

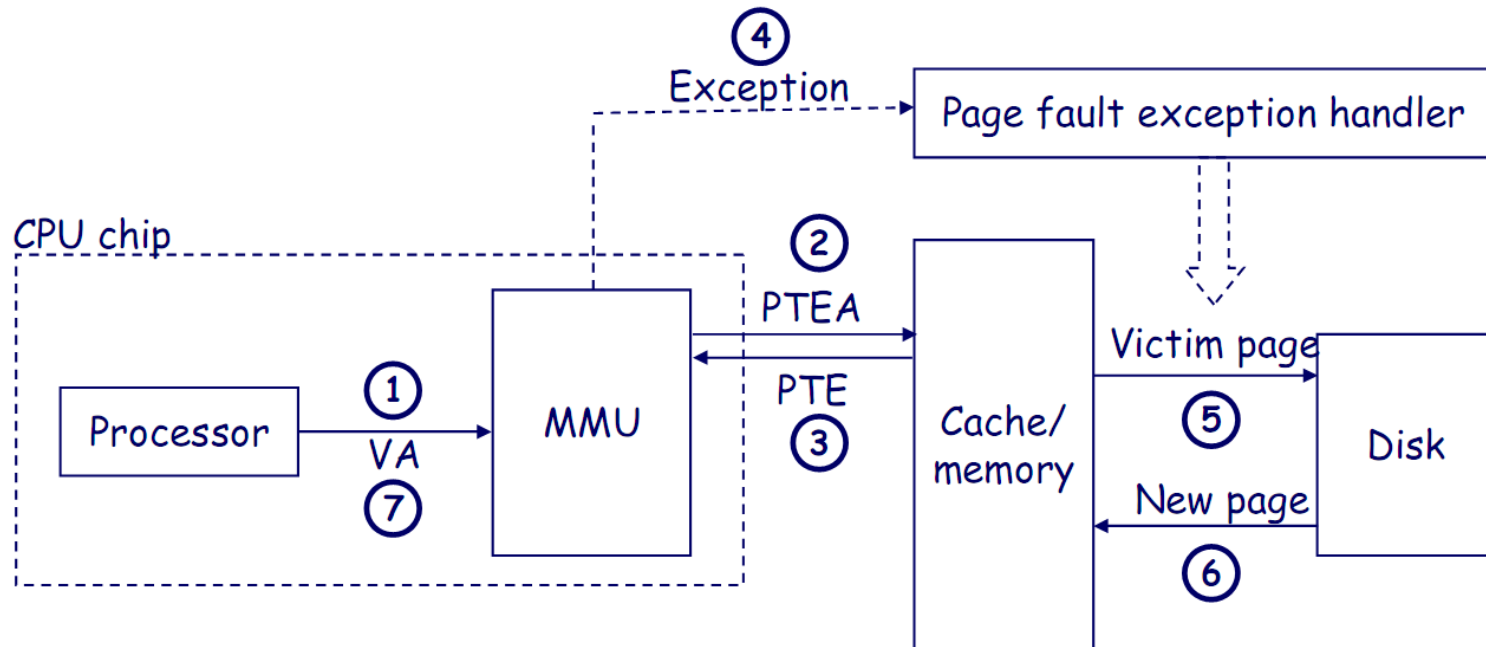


Address Translation: Page Hit



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to L1 cache
- 5) L1 cache sends data word to processor

Address Translation: Page Fault

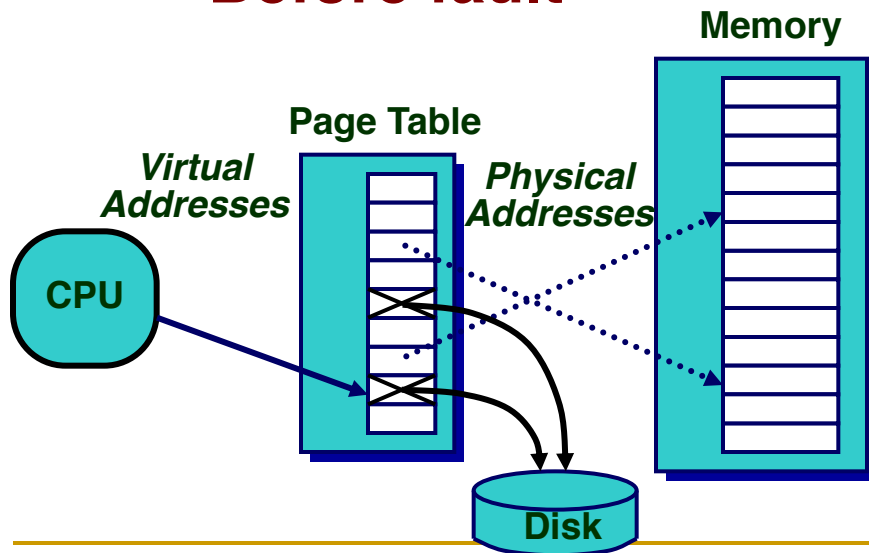


- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim, and if dirty pages it out to disk
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction.

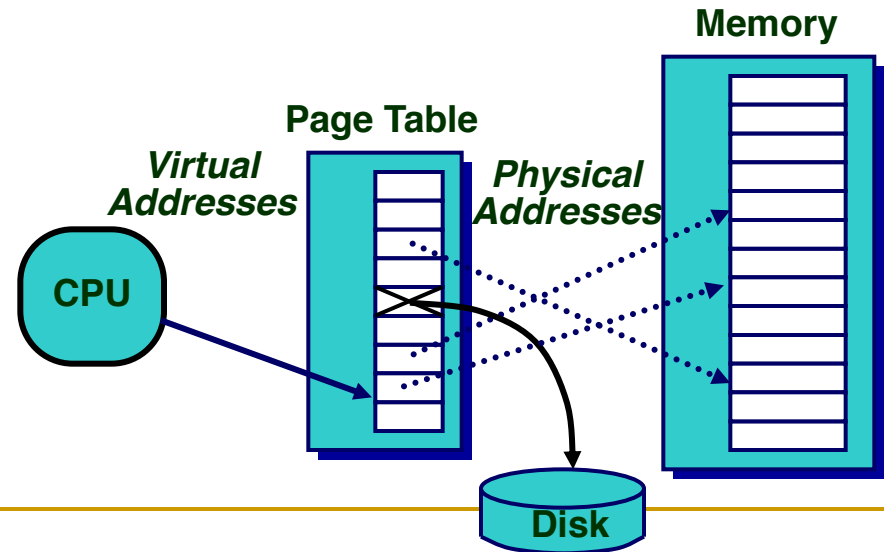
Page Fault (“A Miss in Physical Memory”)

- If a page is not in physical memory but disk
 - ❑ Page table entry indicates virtual page not in memory
 - ❑ Access to such a page triggers a page fault exception
 - ❑ OS trap handler invoked to move data from disk into memory
 - Other processes can continue executing
 - OS has full control over placement

Before fault

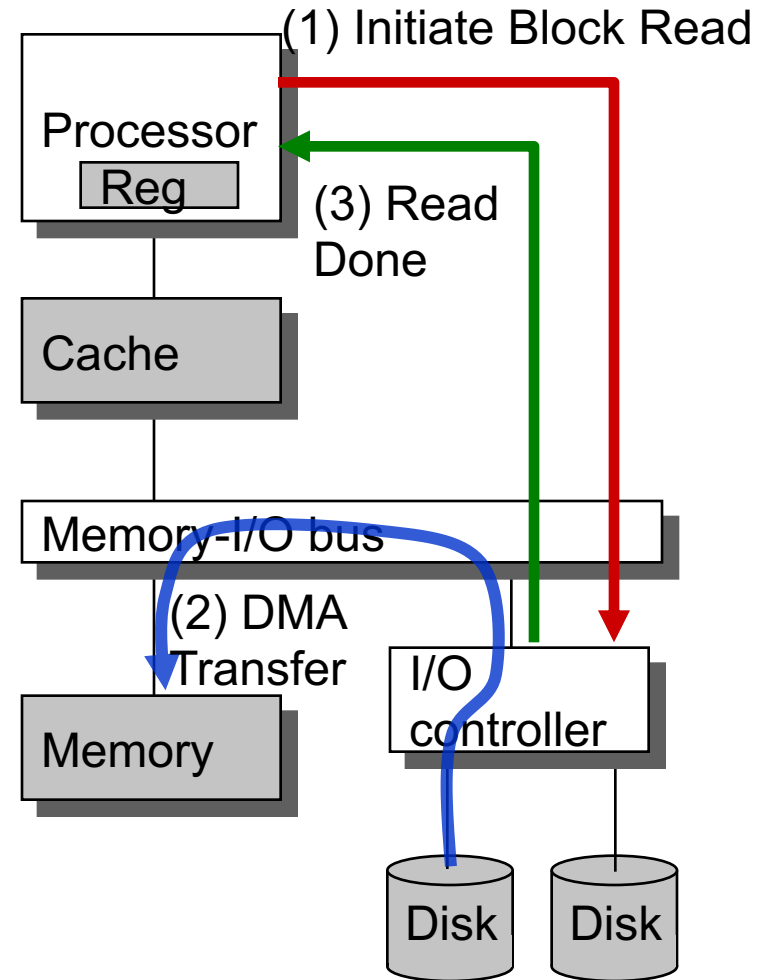


After fault



Servicing a Page Fault

1. Processor signals I/O controller
 - ❑ Read block of length P starting at disk address X and store starting at memory address Y
2. Disk-to-mem read occurs
 - ❑ Direct Memory Access (DMA)
 - ❑ Under control of I/O controller
3. Controller signals completion
 - ❑ Interrupts processor
 - ❑ OS resumes suspended process

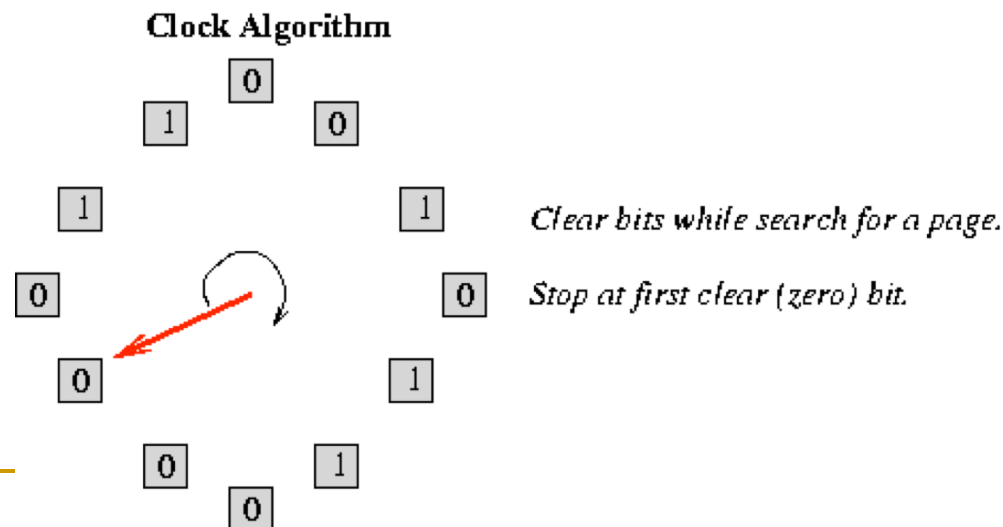


Page Replacement Algorithms

- If physical memory is full (i.e., list of free physical pages is empty), which physical frame to replace on a page fault?
- Is True LRU feasible?
 - 4GB memory, 4KB pages, how many possibilities of ordering?
- Modern systems use approximations of LRU
 - E.g., the CLOCK algorithm
- And, more sophisticated algorithms to take into account “frequency” of use
 - E.g., the ARC algorithm
 - Megiddo and Modha, “[ARC: A Self-Tuning, Low Overhead Replacement Cache](#),” FAST 2003.

CLOCK Page Replacement Algorithm

- Keep a **circular list of physical frames** in memory (OS does)
- Keep a **pointer** (hand) to the last-examined frame in the list
- When a page is accessed, set the R bit in the PTE
- When a frame needs to be replaced, replace the first frame that has the reference (R) bit not set, traversing the circular list starting from the pointer (hand) clockwise
 - ❑ During traversal, clear the R bits of examined frames
 - ❑ Set the hand pointer to the next frame in the list



Cache versus Page Replacement

- Physical memory (DRAM) is a cache for disk
 - Managed by system software via the virtual memory subsystem
- Page replacement is similar to cache replacement
- Page table is the “tag store” for physical memory data store
- What is the difference?
 - Required speed of access to cache vs. physical memory
 - Number of blocks in a cache vs. physical memory
 - “Tolerable” amount of time to find a replacement candidate (disk versus memory access latency)
 - Role of hardware versus software

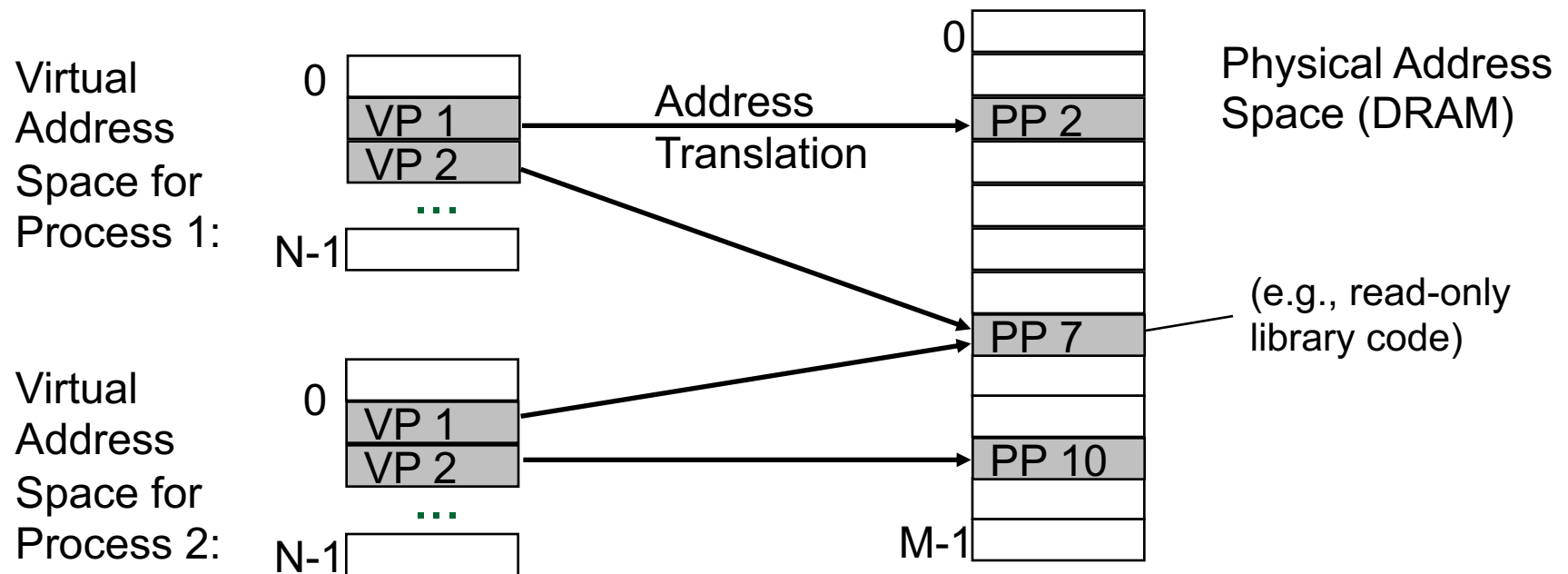
Memory Protection

Memory Protection

- Multiple programs (i.e., processes) run concurrently
 - Each process has its own page table
 - Each process can use its entire virtual address space without worrying about where other programs are
- A process can only access physical pages mapped in its page table – cannot overwrite memory of another process
 - Provides protection and isolation between processes
 - Enables access control mechanisms per page

Page Table is Per Process

- Each process has its own virtual address space
 - Full address space for each program
 - Simplifies memory allocation, sharing, linking and loading

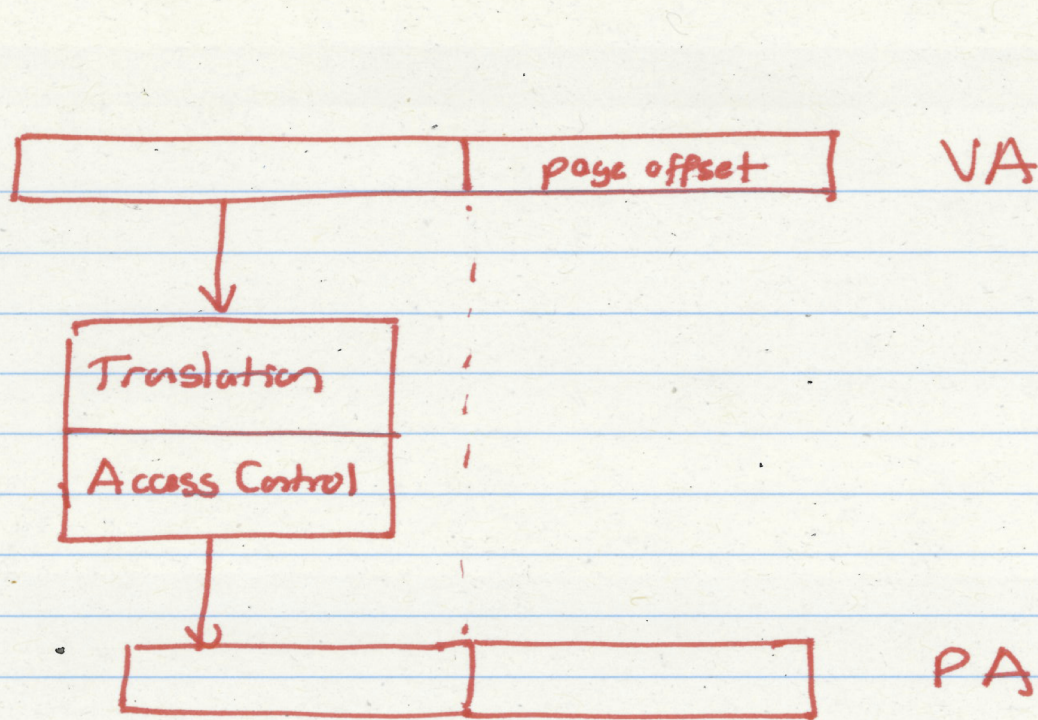


Access Protection/Control via Virtual Memory

Page-Level Access Control (Protection)

- Not every process is allowed to access every page
 - E.g., need supervisor (i.e., kernel) level privilege to access system pages
 - E.g., may not be able to execute “instructions” in some pages
 - Idea: Store access control information on a page basis in the process's page table
 - Enforce access control at the same time as translation
- Virtual memory system serves two functions today
- Address translation (for illusion of large physical memory)
 - Access control (protection)

Two Functions of Virtual Memory



Virtual
Memory

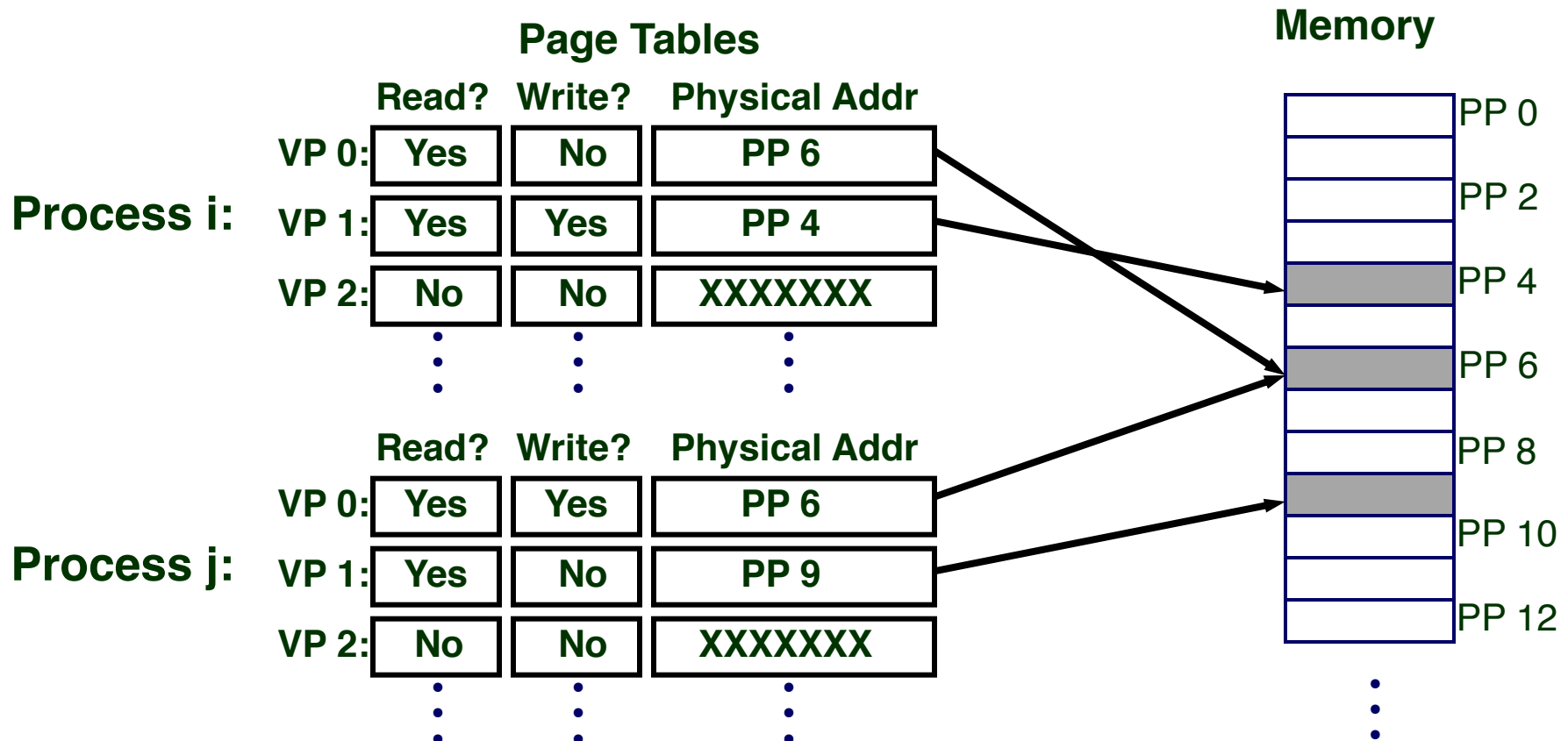
Two Functions
Today

1. Translation
2. Access control (protection)

PTE contains access control bits associated with the virtual page.

VM as a Tool for Memory Access Protection

- Extend Page Table Entries (PTEs) with permission bits
- Check bits on each access and during a page fault
 - If violated, generate exception (Access Protection exception)



Privilege Levels in x86

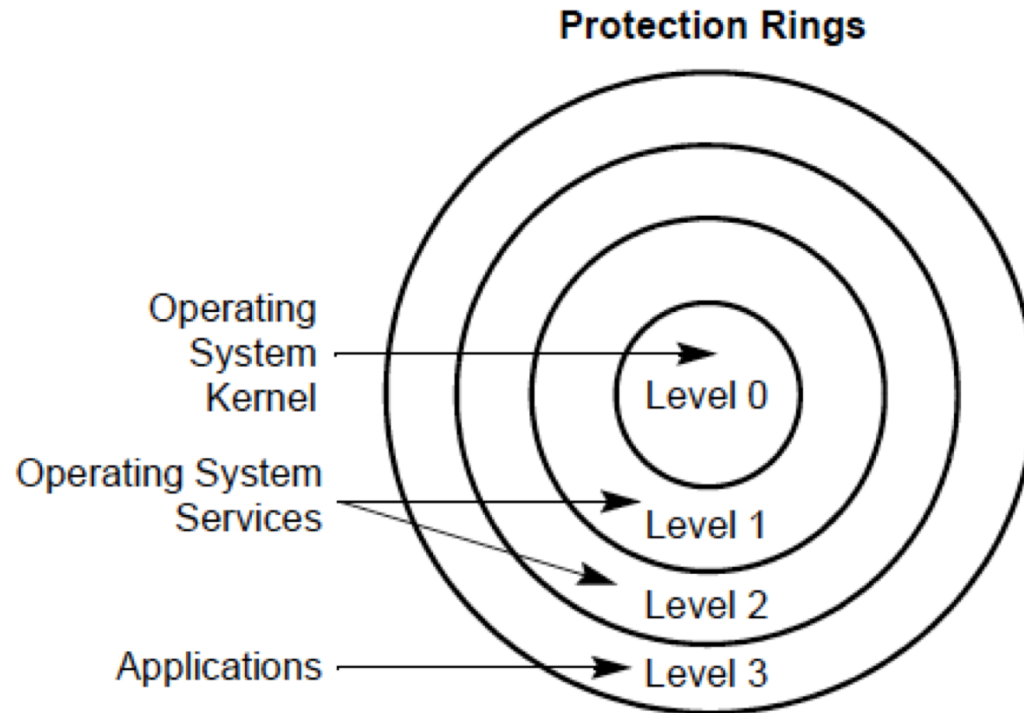


Figure 5-3. Protection Rings

Privilege Levels in x86

- Four **privilege levels** in x86 (referred to as **rings**)

- Ring 0: Highest privilege (operating system)
 - Ring 1: Not widely used
 - Ring 2: Not widely used
 - Ring 3: Lowest privilege (user applications)
- "Supervisor"**
- "User"**

- Supervisor = Kernel (in modern terminology)
-

x86: A Closer Look at the PDE/PTE

- **PDE:** Page Directory Entry (32 bits)
- **PTE:** Page Table Entry (32 bits)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0																																			
Address of page directory ¹																Ignored						P C D	PW T	Ignored		CR3									
Bits 31:22 of address of 2MB page frame										Reserved (must be 0)				Bits 39:32 of address ²				P A T	Ignored	G	1	D	A	P C D	PW T	U / S	R / W	1	PDE: 4MB page						
PDE	Address of page table &PT																Ignored				0	Ignored				P A T	1	D	A	P C D	PW T	U / S	R / W	1	PDE: page table
	Ignored																Ignored				0	PDE: not present													
PTE	Address of 4KB page frame PPN																Ignored				G	P A T	1	D	A	P C D	PW T	U / S	R / W	1	PTE: 4KB page				
	Ignored																Ignored				0	PTE: not present													

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Protection: PDE's Flags

- Protects all 1024 pages in a page table

Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table 4-4); otherwise, ignored

Protection: PTE's Flags

■ Protects one page at a time

Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0) ¹
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise

Page Level Protection in x86

Table 5-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write ⁺
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write ⁺
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write ⁺
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write ⁺
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write ⁺
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write ⁺
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write ⁺
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write ⁺
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write ⁺
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

Protection: PDE + PTE = ???

Table 5-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

NOTE:

* If CR0.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. If CR0.WP = 0, supervisor privilege permits read-write access.

Food for Thought: What If?

- Your hardware is unreliable and someone can flip the access protection bits
 - such that a user-level program can gain supervisor-level access (i.e., access to all data on the system)
 - by flipping the access control bit from user to supervisor!
- Can this happen?

Remember RowHammer?

One can
predictably induce errors
in most DRAM memory chips

Remember RowHammer?

- One can **predictably induce bit flips** in commodity DRAM chips
 - >80% of the tested DRAM chips are vulnerable
- First example of how a **simple hardware failure mechanism** can create a **widespread system security vulnerability**

WIRED

Forget Software—Now Hackers Are Exploiting Physics

BUSINESS	CULTURE	DESIGN	GEAR	SCIENCE
----------	---------	--------	------	---------

ANDY GREENBERG SECURITY 08.31.16 7:00 AM

SHARE



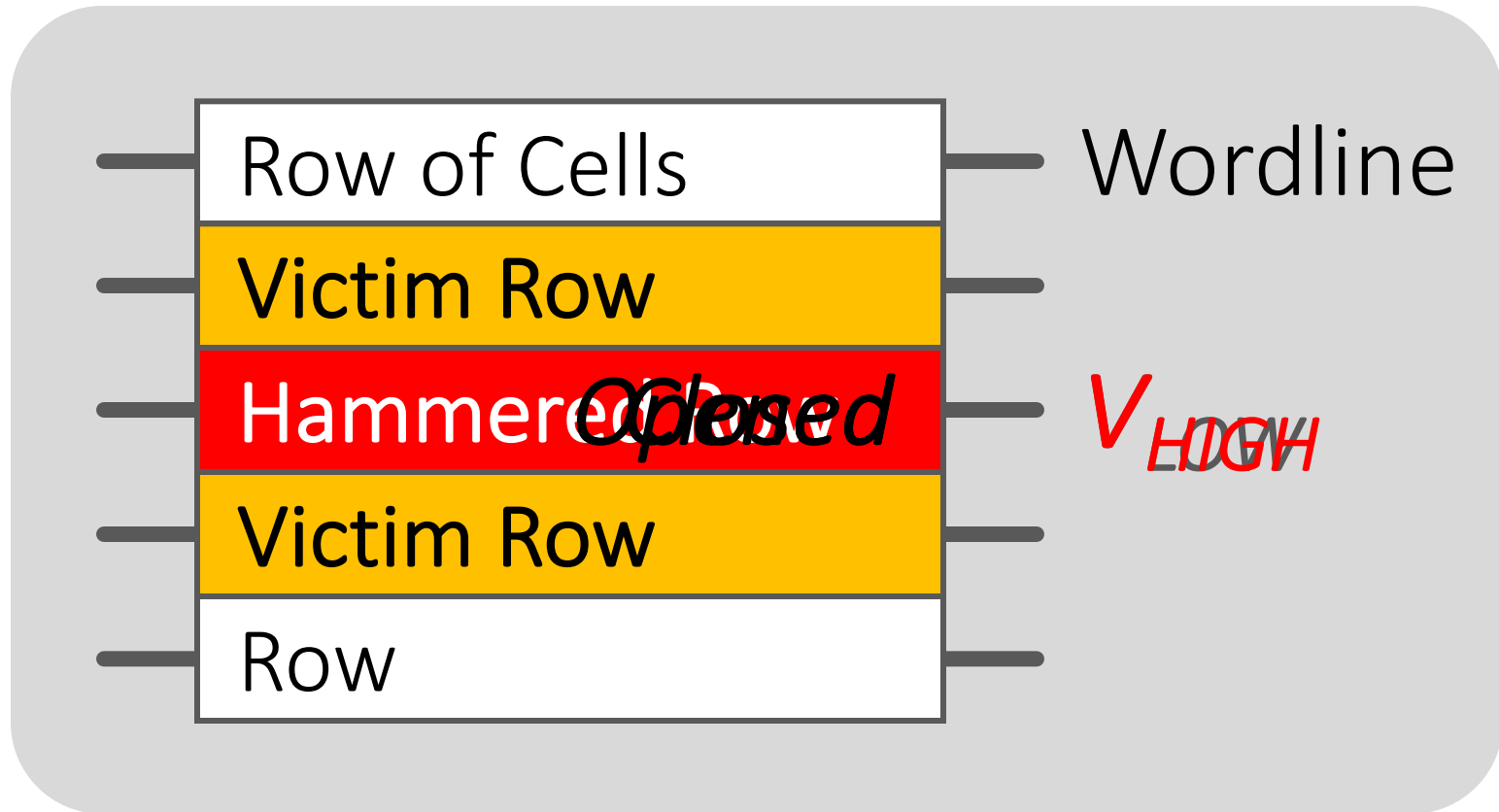
SHARE
18276



TWEET

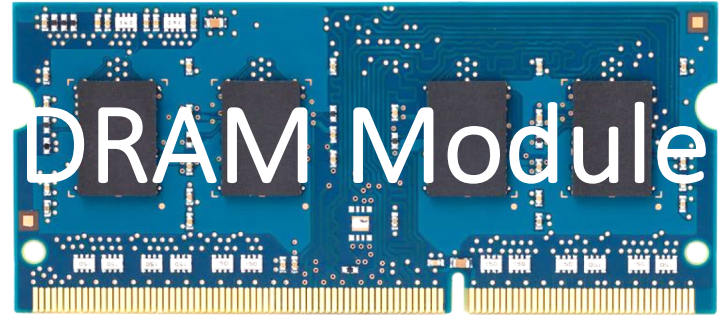
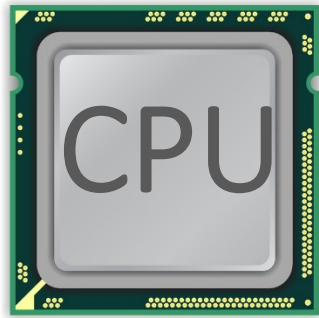
FORGET SOFTWARE—NOW HACKERS ARE EXPLOITING PHYSICS

Modern DRAM is Prone to Disturbance Errors

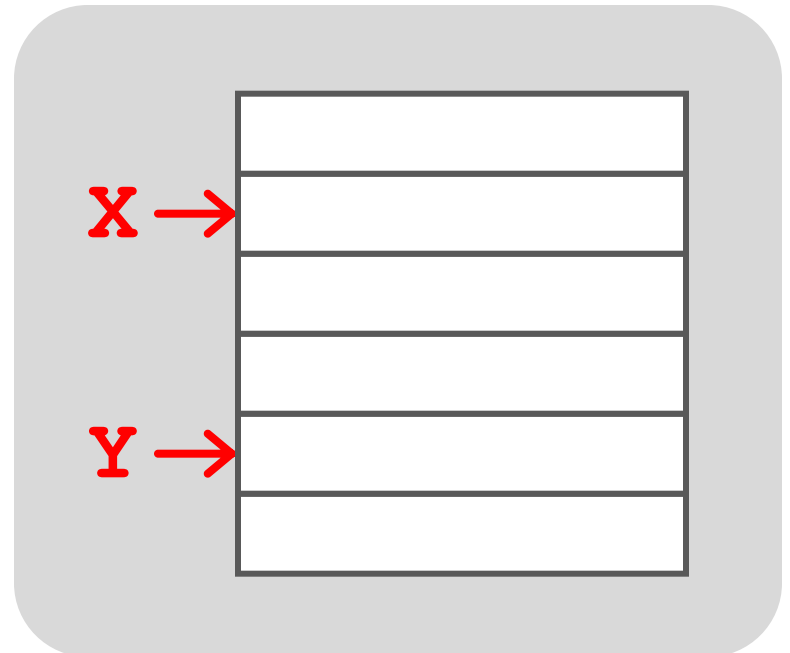


Repeatedly reading a row enough times (before memory gets refreshed) induces **disturbance errors** in adjacent rows in **most real DRAM chips you can buy today**

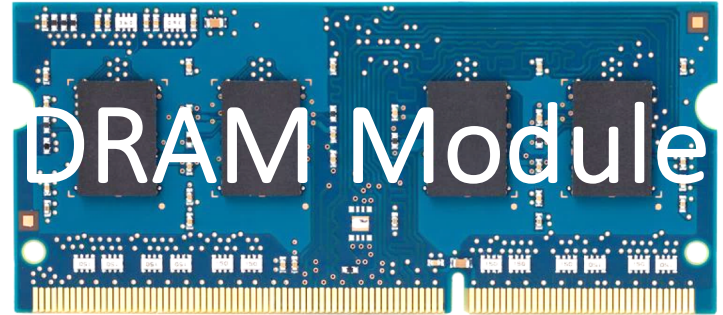
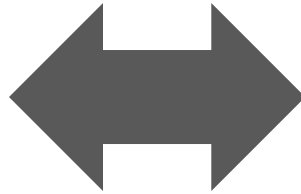
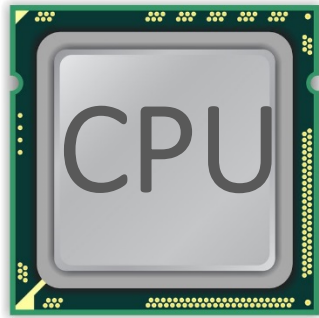
A Simple Program Can Induce Many Errors



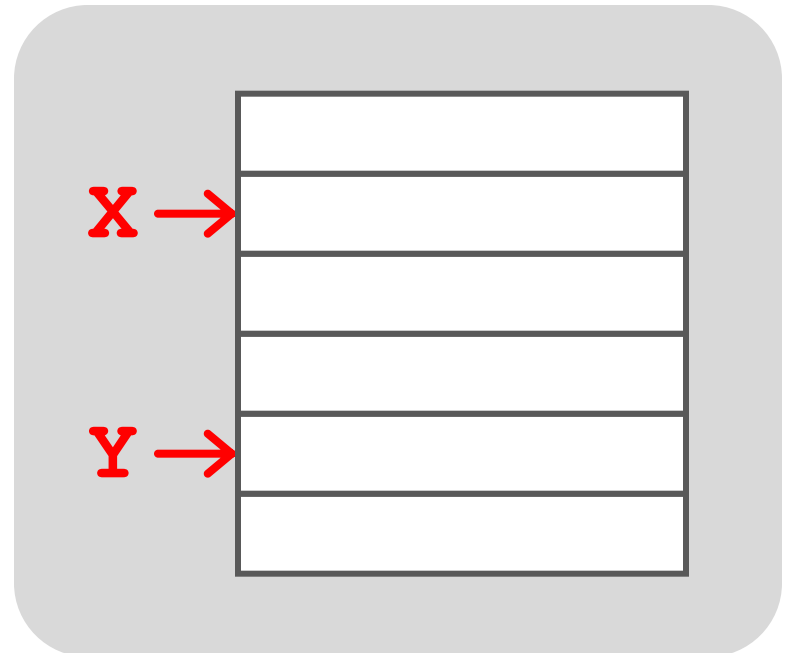
```
loop:  
  mov  (X),  %eax  
  mov  (Y),  %ebx  
  clflush (X)  
  clflush (Y)  
  mfence  
  jmp  loop
```



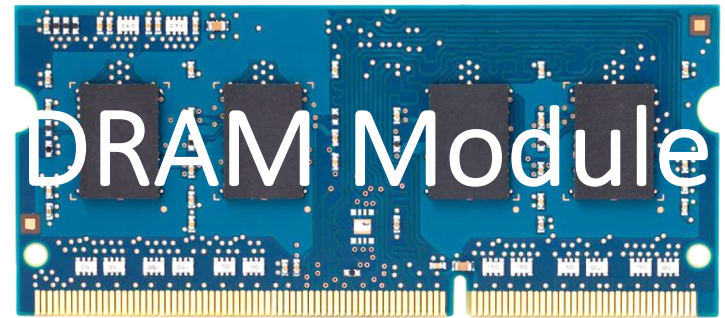
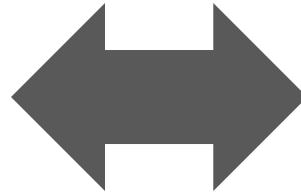
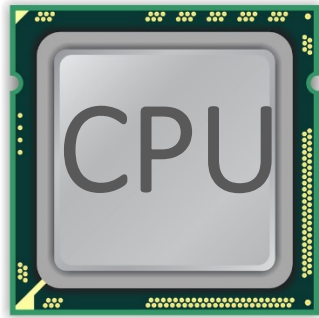
A Simple Program Can Induce Many Errors



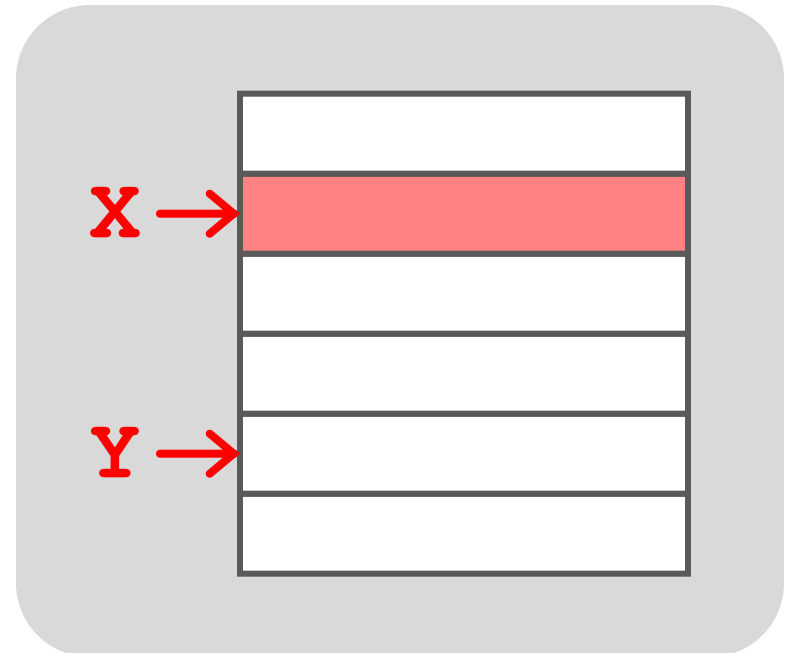
1. Avoid *cache hits*
 - Flush **X** from cache
2. Avoid *row hits* to **X**
 - Read **Y** in another row



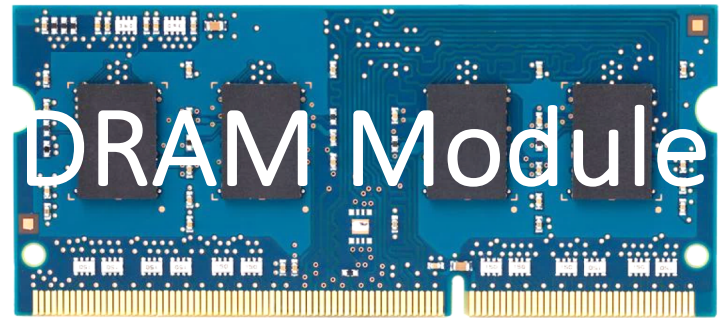
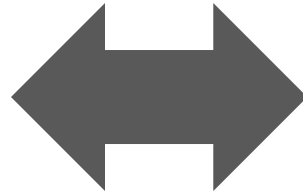
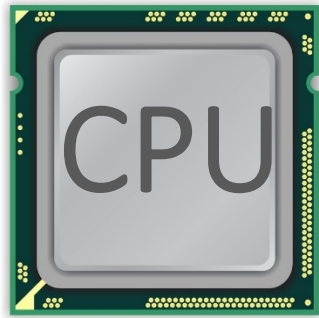
A Simple Program Can Induce Many Errors



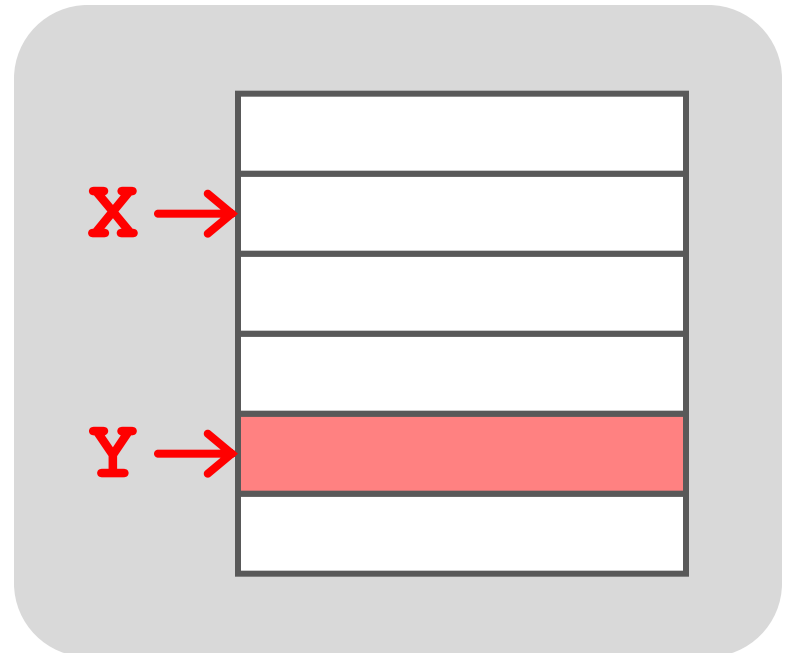
```
loop:  
  mov  (X), %eax  
  mov  (Y), %ebx  
  clflush (X)  
  clflush (Y)  
  mfence  
  jmp  loop
```



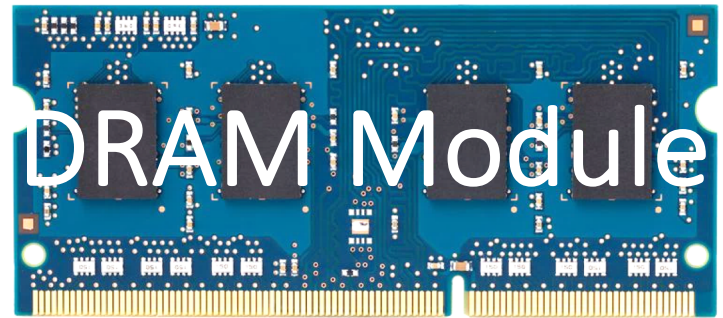
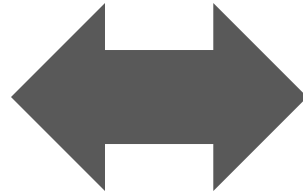
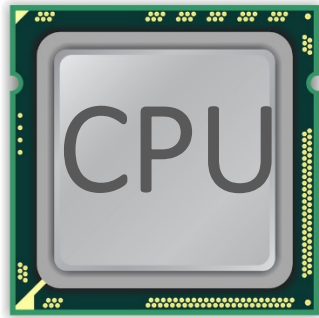
A Simple Program Can Induce Many Errors



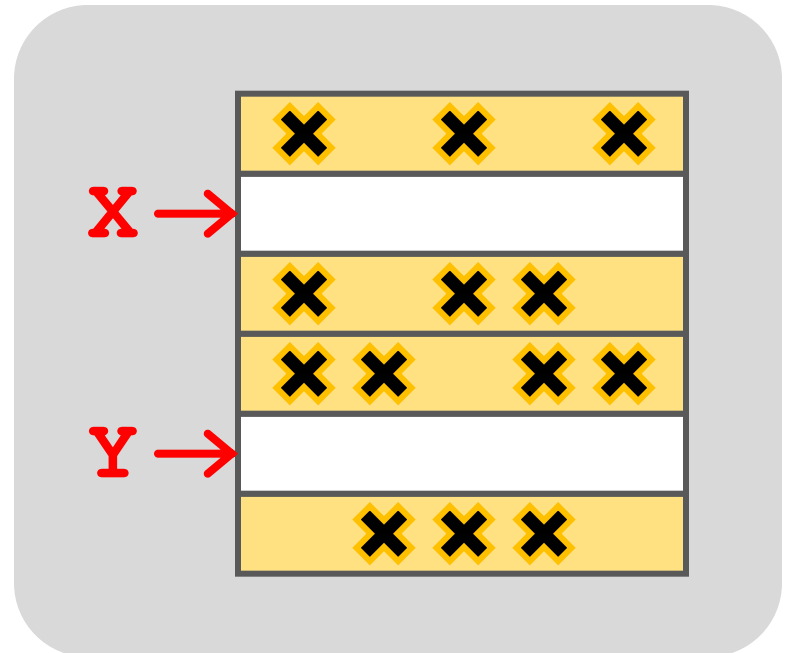
```
loop:  
  mov  (X), %eax  
  mov  (Y), %ebx  
  clflush (X)  
  clflush (Y)  
  mfence  
  jmp  loop
```



A Simple Program Can Induce Many Errors



```
loop:  
  mov  (X),  %eax  
  mov  (Y),  %ebx  
  clflush (X)  
  clflush (Y)  
  mfence  
  jmp  loop
```



Observed Errors in Real Systems

CPU Architecture	Errors	Access-Rate
Intel Haswell (2013)	22.9K	12.3M/sec
Intel Ivy Bridge (2012)	20.7K	11.7M/sec
Intel Sandy Bridge (2011)	16.1K	11.6M/sec
AMD Piledriver (2012)	59	6.1M/sec

A real reliability & security issue

One Can Take Over an Otherwise-Secure System

Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors

Abstract. Memory isolation is a key property of a reliable and secure computing system — an access to one memory address should not have unintended side effects on data stored in other addresses. However, as DRAM process technology

Project Zero

Flipping Bits in Memory Without Accessing Them:
An Experimental Study of DRAM Disturbance Errors
(Kim et al., ISCA 2014)

News and updates from the Project Zero team at Google

Exploiting the DRAM rowhammer bug to
gain kernel privileges (Seaborn, 2015)

Monday, March 9, 2015

Exploiting the DRAM rowhammer bug to gain kernel privileges

RowHammer Security Attack Example

- “Rowhammer” is a problem with some recent DRAM devices in which repeatedly accessing a row of memory can cause bit flips in adjacent rows (Kim et al., ISCA 2014).
 - Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors (Kim et al., ISCA 2014)
- We tested a selection of laptops and found that a subset of them exhibited the problem.
- We built two working privilege escalation exploits that use this effect.
 - Exploiting the DRAM rowhammer bug to gain kernel privileges (Seaborn+, 2015)
- One exploit uses rowhammer-induced bit flips to gain kernel privileges on x86-64 Linux when run as an unprivileged userland process.
- When run on a machine vulnerable to the rowhammer problem, the process was able to induce bit flips in page table entries (PTEs).
- It was able to use this to gain write access to its own page table, and hence gain read-write access to all of physical memory.

Google's Original RowHammer Attack

The following slides are from Mark Seaborn and Thomas Dullien's BlackHat 2015 talk

<https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf>

Kernel exploit

- x86 page tables entries (PTEs) are **dense and trusted**
 - They control access to physical memory
 - A bit flip in a PTE's physical page number can give a process access to a different physical page
- Aim of exploit: Get access to a page table
 - Gives access to all of physical memory
- Maximise chances that a bit flip is useful:
 - Spray physical memory with page tables
 - Check for useful, repeatable bit flip first

This slide is from Mark Seaborn and Thomas Dullien's BlackHat 2015 talk

<https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf>

x86-64 Page Table Entries (PTEs)

- Page table is a 4k page containing array of 512 PTEs
- Each PTE is 64 bits, containing:

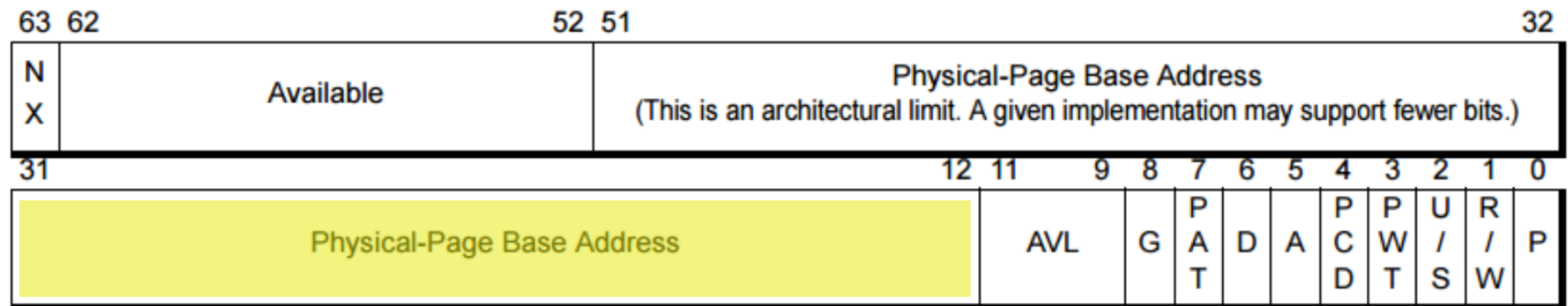
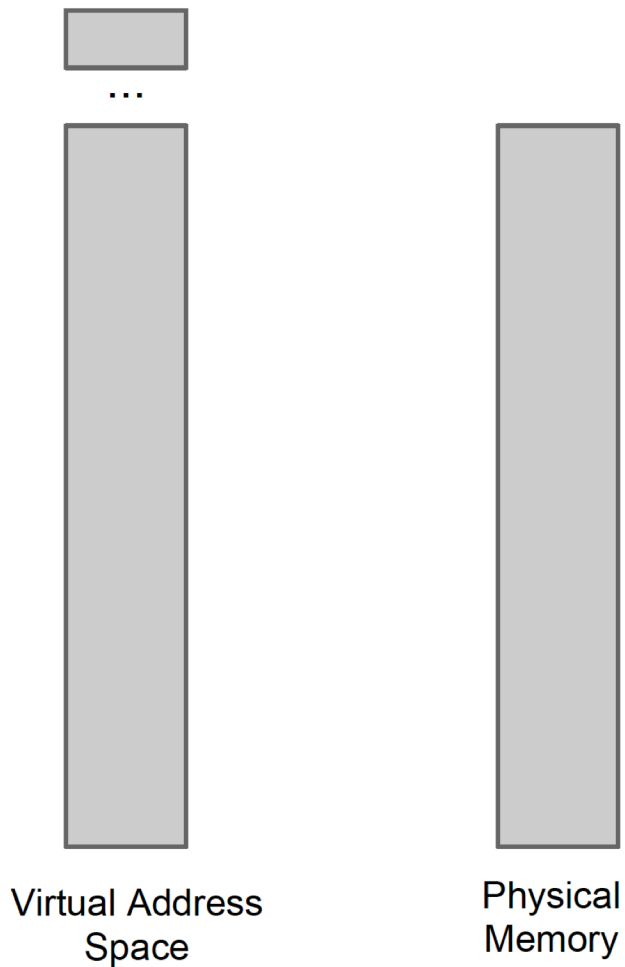


Figure 5-21. 4-Kbyte PTE—Long Mode

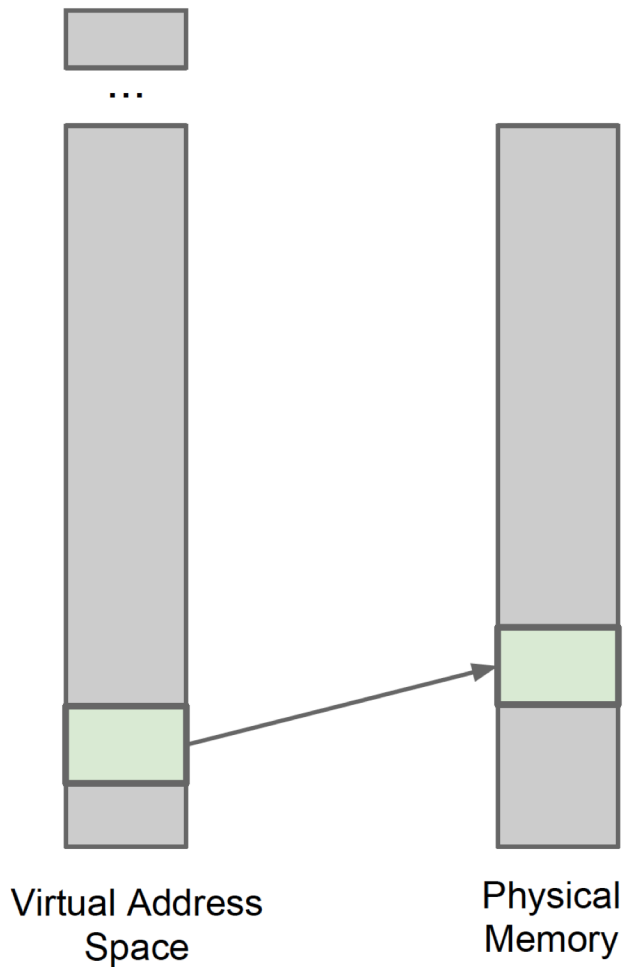
- Could flip:
 - “Writable” permission bit (RW): 1 bit → 2% chance
 - Physical page number: 20 bits on 4GB system → 31% chance

This slide is from Mark Seaborn and Thomas Dullien’s BlackHat 2015 talk



This slide is from Mark Seaborn and Thomas Dullien's BlackHat 2015 talk

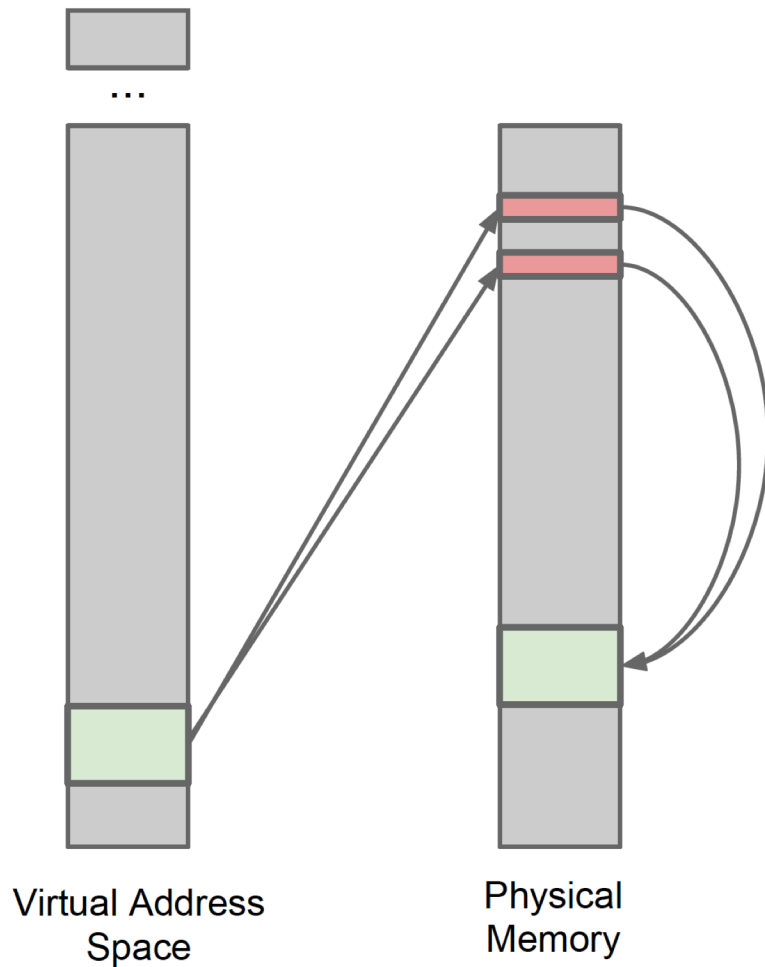
<https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf>



What happens when we map a file with read-write permissions?

This slide is from Mark Seaborn and Thomas Dullien's BlackHat 2015 talk

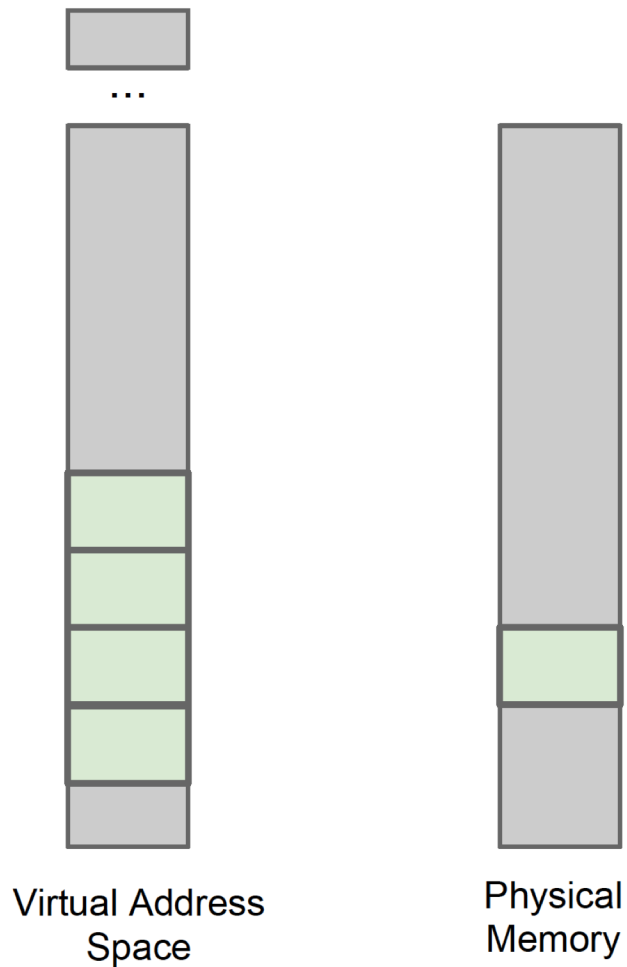
<https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf>



What happens when we map a file with read-write permissions? Indirection via page tables.

This slide is from Mark Seaborn and Thomas Dullien's BlackHat 2015 talk

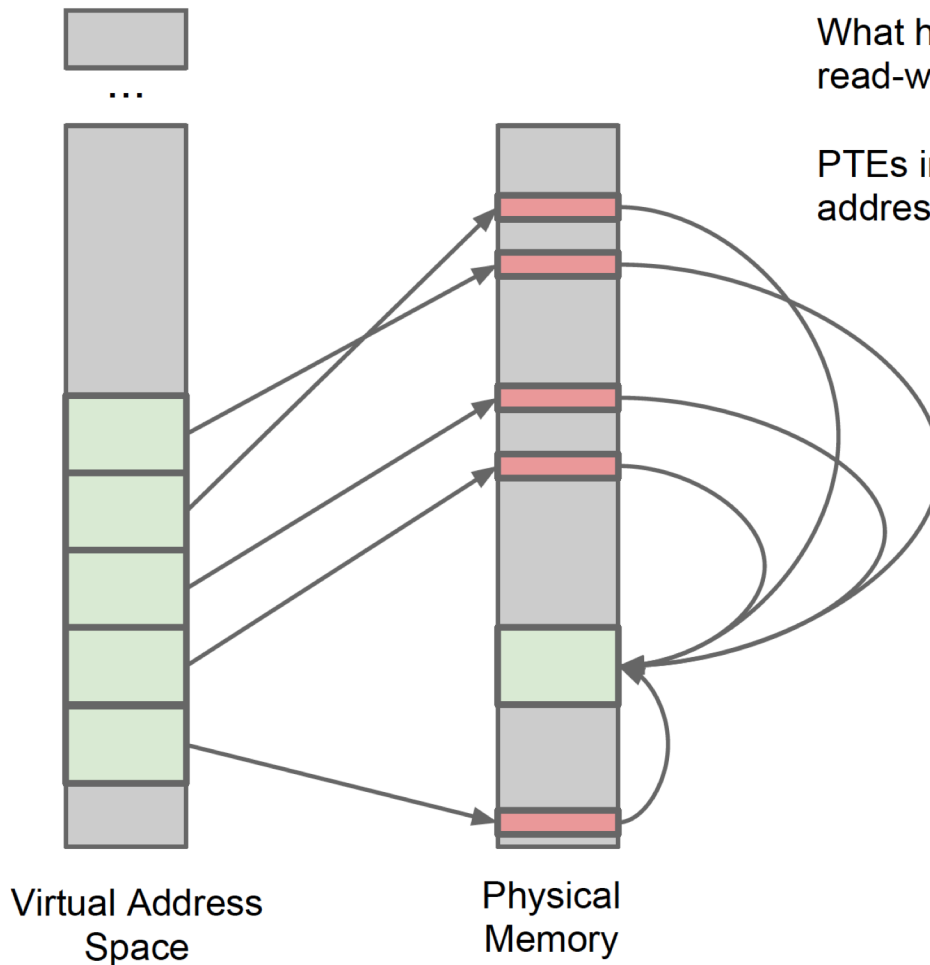
<https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf>



What happens when we repeatedly map a file with read-write permissions?

This slide is from Mark Seaborn and Thomas Dullien's BlackHat 2015 talk

<https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf>

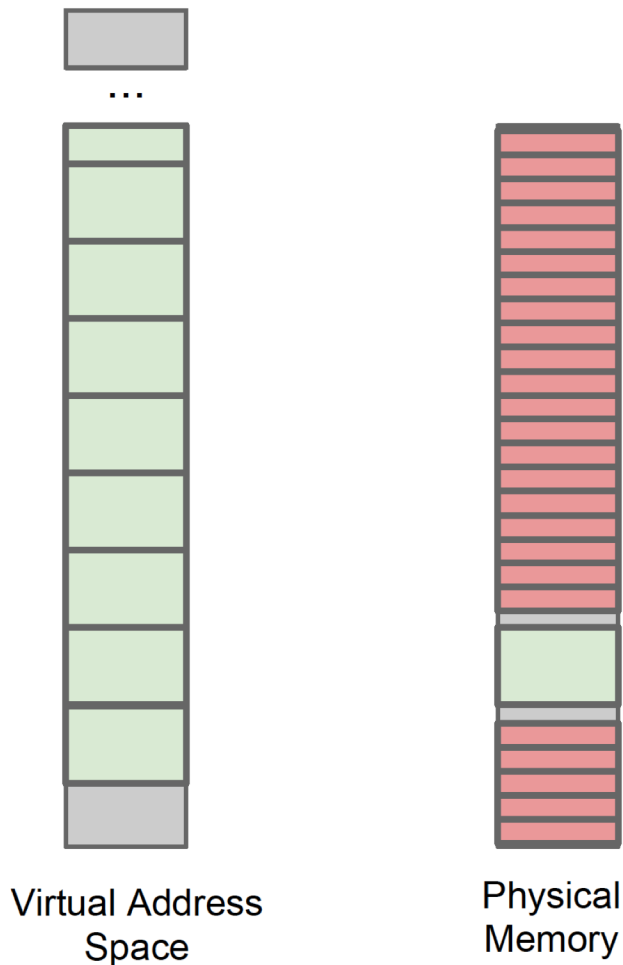


What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

This slide is from Mark Seaborn and Thomas Dullien's BlackHat 2015 talk

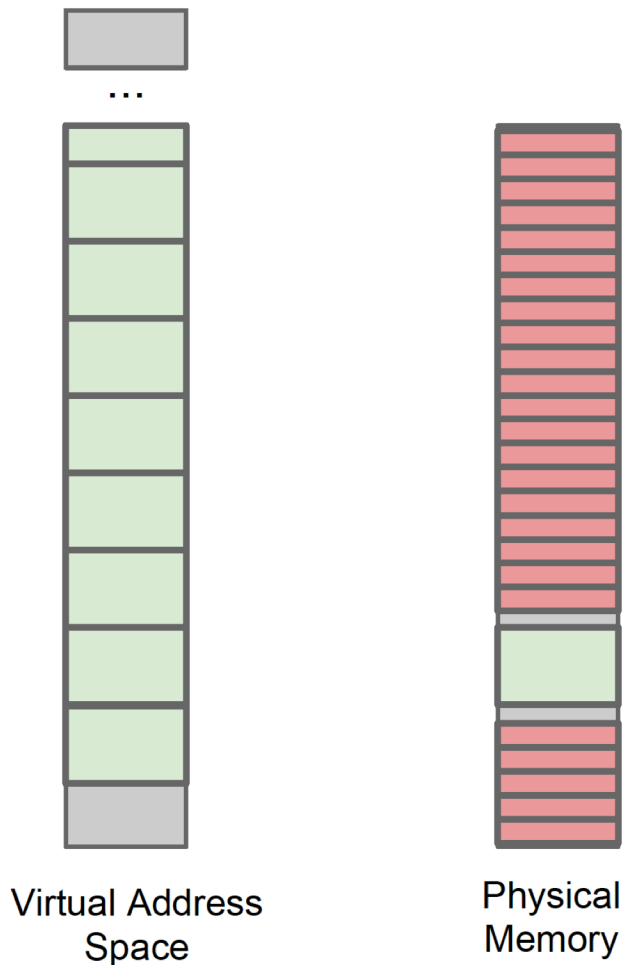
<https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf>



What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

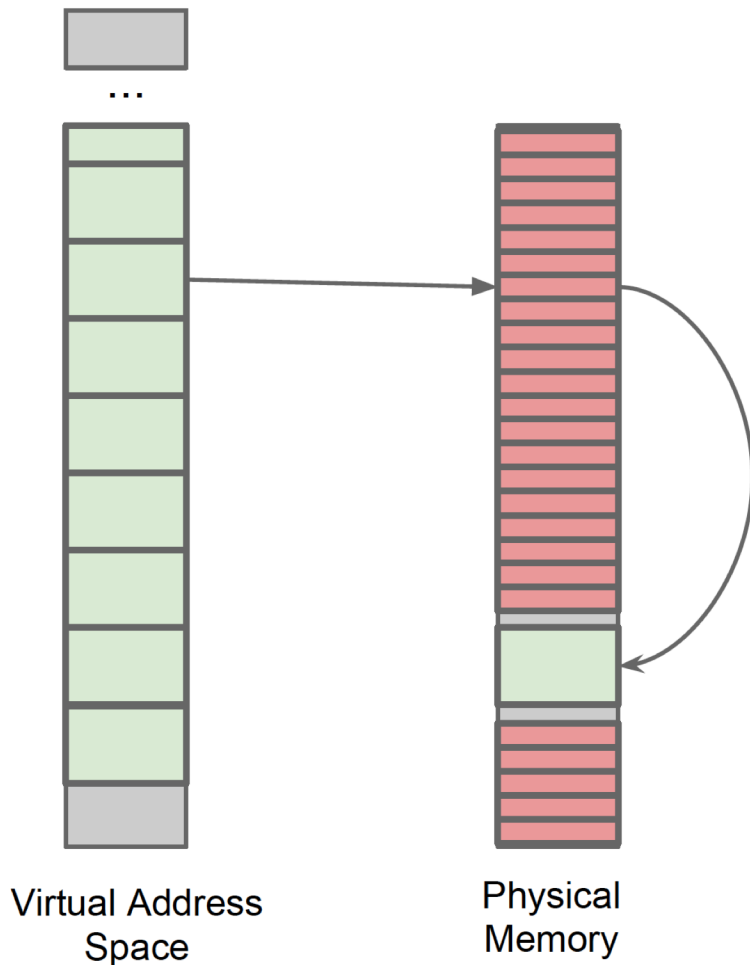


What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

Each of them points to pages in the same physical file mapping.



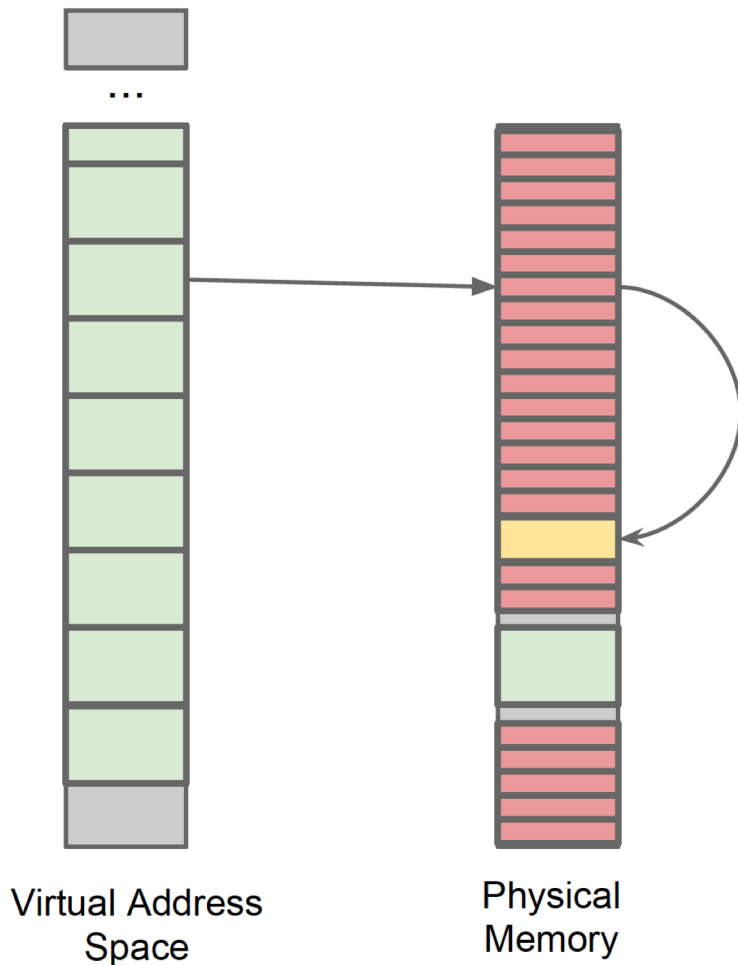
What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

Each of them points to pages in the same physical file mapping.

If a bit in the right place in the PTE flips ...



What happens when we repeatedly map a file with read-write permissions?

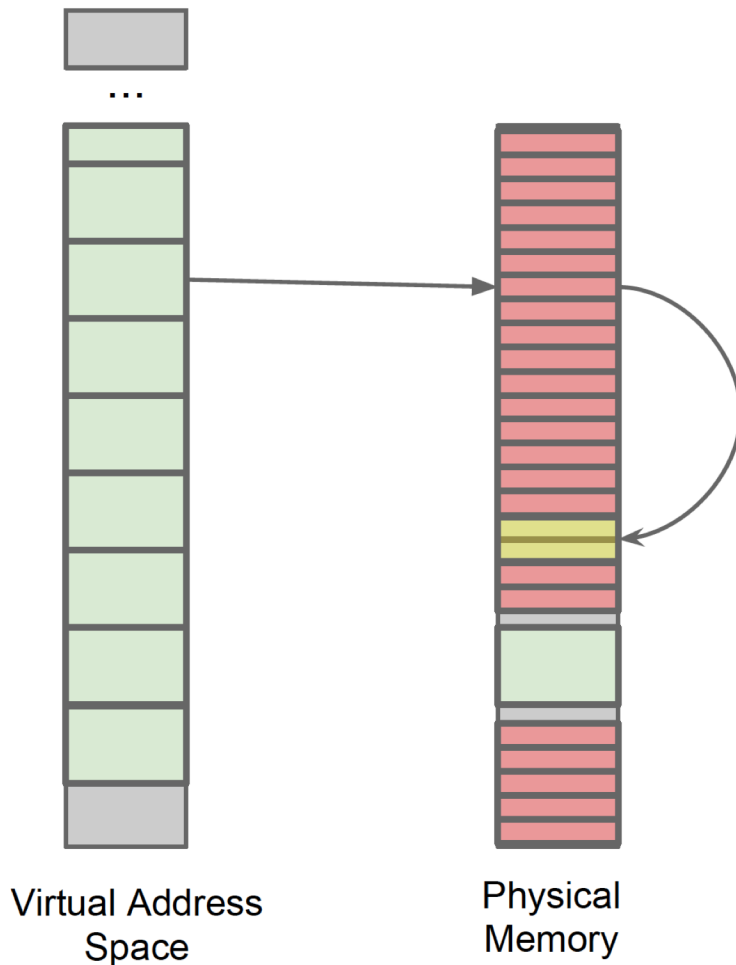
PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

Each of them points to pages in the same physical file mapping.

If a bit in the right place in the PTE flips ...

... the corresponding virtual address now points to a wrong physical page - with RW access.



What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

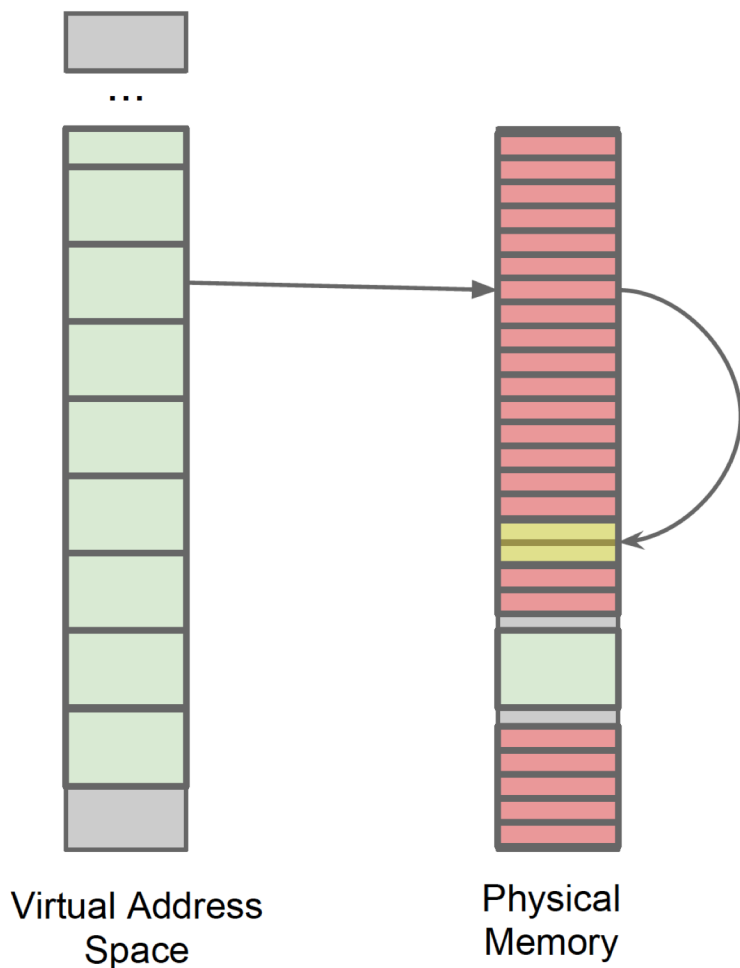
Each of them points to pages in the same physical file mapping.

If a bit in the right place in the PTE flips ...

... the corresponding virtual address now points to a wrong physical page - with RW access.

Chances are this wrong page contains a page table itself.

This slide is from Mark Seaborn and Thomas Dullien's BlackHat 2015 talk



What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

Each of them points to pages in the same physical file mapping.

If a bit in the right place in the PTE flips ...

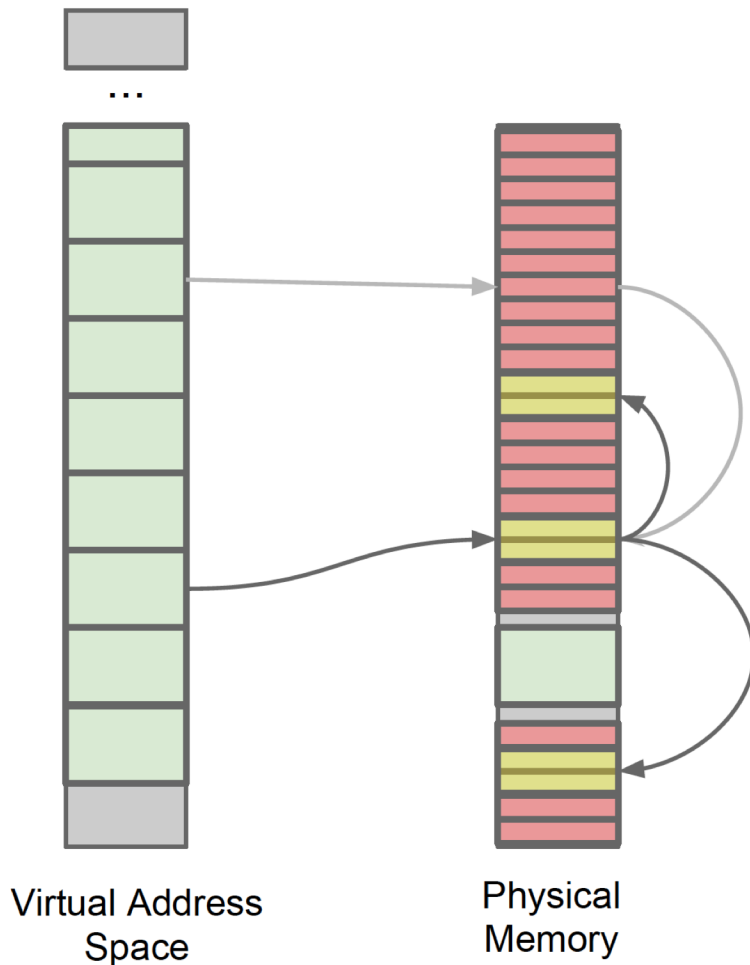
... the corresponding virtual address now points to a wrong physical page - with RW access.

Chances are this wrong page contains a page table itself.

An attacker that can read / write page tables ...

This slide is from Mark Seaborn and Thomas Dullien's BlackHat 2015 talk

<https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf>



What happens when we repeatedly map a file with read-write permissions?

PTEs in physical memory help resolve virtual addresses to physical pages.

We can fill physical memory with PTEs.

Each of them points to pages in the same physical file mapping.

If a bit in the right place in the PTE flips ...

... the corresponding virtual address now points to a wrong physical page - with RW access.

Chances are this wrong page contains a page table itself.

An attacker that can read / write page tables can use that to map **any** memory read-write.

This slide is from Mark Seaborn and Thomas Dullien's BlackHat 2015 talk

<https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf>

Exploit strategy

Privilege escalation in 7 easy steps ...

1. Allocate a large chunk of memory
2. Search for locations prone to flipping
3. Check if they fall into the “right spot” in a PTE for allowing the exploit
4. Return that particular area of memory to the operating system
5. Force OS to re-use the memory for PTEs by allocating massive quantities of address space
6. Cause the bitflip - shift PTE to point into page table
7. Abuse R/W access to all of physical memory

In practice, there are many complications.

This slide is from Mark Seaborn and Thomas Dullien's BlackHat 2015 talk

<https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf>

Security Implications



Security Implications



Rowhammer

It's like breaking into an apartment by repeatedly slamming a neighbor's door until the vibrations open the door you were after

More Security Implications (I)

“We can gain unrestricted access to systems of website visitors.”

www.iaik.tugraz.at ■

Not there yet, but ...



ROOT privileges for web apps!

29

Daniel Gruss (@lavados), Clémentine Maurice (@BloodyTangerine),
December 28, 2015 — 32c3, Hamburg, Germany



GATED
COMMUNITIES

Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript (DIMVA'16)

More Security Implications (II)

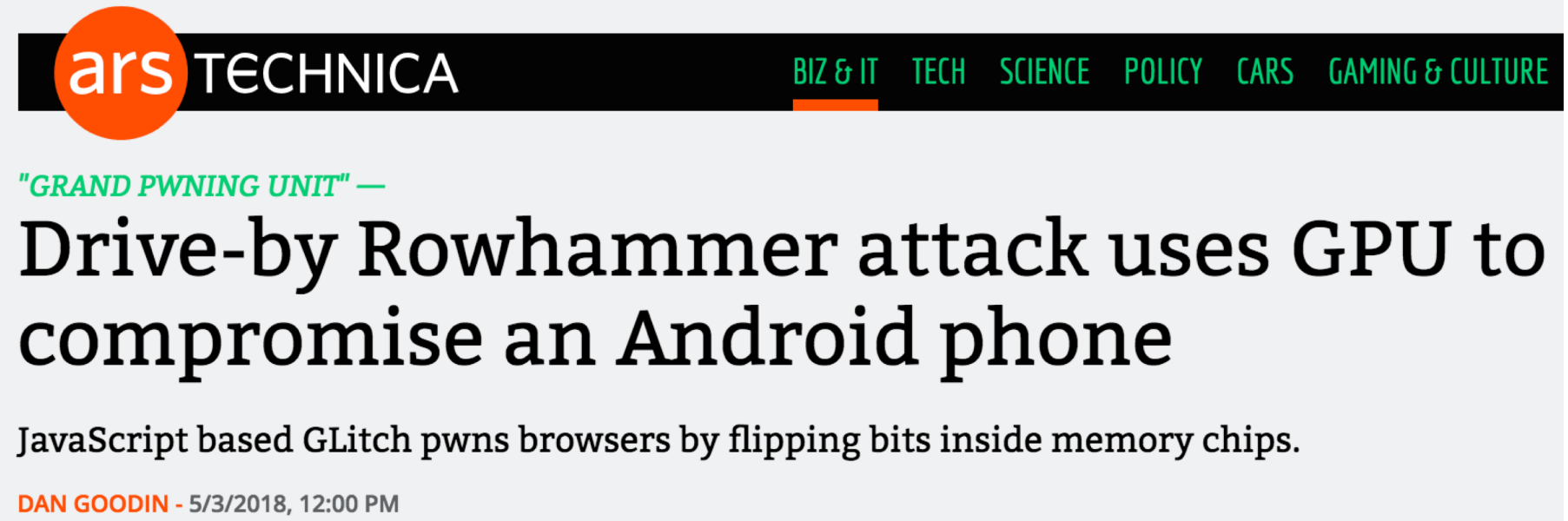
"Can gain control of a smart phone deterministically"



Drammer: Deterministic Rowhammer
Attacks on Mobile Platforms, CCS'16 86

More Security Implications (III)

- Using an integrated GPU in a mobile system to remotely escalate privilege via the WebGL interface



The screenshot shows the top of an Ars Technica article. The header includes the 'ars TECHNICA' logo and a navigation bar with links for 'BIZ & IT', 'TECH', 'SCIENCE', 'POLICY', 'CARS', and 'GAMING & CULTURE'. The article title is 'Drive-by Rowhammer attack uses GPU to compromise an Android phone', preceded by the sub-header '"GRAND PWINING UNIT" —'. A summary line reads: 'JavaScript based GLitch pwns browsers by flipping bits inside memory chips.' The author and date are listed as 'DAN GOODIN - 5/3/2018, 12:00 PM'.

ars TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE

"GRAND PWINING UNIT" —

Drive-by Rowhammer attack uses GPU to compromise an Android phone

JavaScript based GLitch pwns browsers by flipping bits inside memory chips.

DAN GOODIN - 5/3/2018, 12:00 PM

Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU

Pietro Frigo
Vrije Universiteit
Amsterdam
p.frigo@vu.nl

Cristiano Giuffrida
Vrije Universiteit
Amsterdam
giuffrida@cs.vu.nl

Herbert Bos
Vrije Universiteit
Amsterdam
herbertb@cs.vu.nl

Kaveh Razavi
Vrije Universiteit
Amsterdam
kaveh@cs.vu.nl

More Security Implications (IV)

■ Rowhammer over RDMA (I)

ars TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE

THROWHAMMER —

Packets over a LAN are all it takes to trigger serious Rowhammer bit flips

The bar for exploiting potentially serious DDR weakness keeps getting lower.

DAN GOODIN - 5/10/2018, 5:26 PM

Throwhammer: Rowhammer Attacks over the Network and Defenses

Andrei Tatar
VU Amsterdam

Radhesh Krishnan
VU Amsterdam

Elias Athanasopoulos
University of Cyprus

Cristiano Giuffrida
VU Amsterdam

Herbert Bos
VU Amsterdam

Kaveh Razavi
VU Amsterdam

More Security Implications (V)

■ Rowhammer over RDMA (II)



Nethammer—Exploiting DRAM Rowhammer Bug Through Network Requests



Nethammer: Inducing Rowhammer Faults through Network Requests

Moritz Lipp
Graz University of Technology

Daniel Gruss
Graz University of Technology

Misiker Tadesse Aga
University of Michigan

Clémentine Maurice
Univ Rennes, CNRS, IRISA

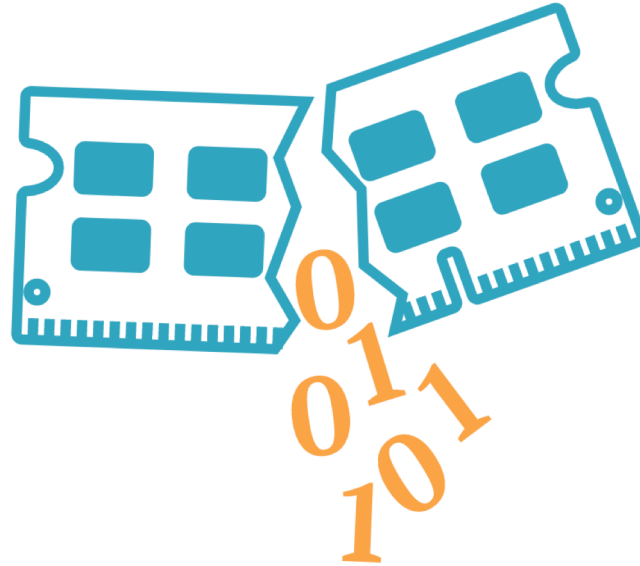
Michael Schwarz
Graz University of Technology

Lukas Raab
Graz University of Technology

Lukas Lamster
Graz University of Technology

More Security Implications (VI)

- IEEE S&P 2020



RAMBleed

RAMBleed: Reading Bits in Memory Without Accessing Them

Andrew Kwong
University of Michigan
ankwong@umich.edu

Daniel Genkin
University of Michigan
genkin@umich.edu

Daniel Gruss
Graz University of Technology
daniel.gruss@iaik.tugraz.at

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

More Security Implications (VII)

■ USENIX Security 2019

Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks

Sanghyun Hong, Pietro Frigo[†], Yiğitcan Kaya, Cristiano Giuffrida[†], Tudor Dumitraş

University of Maryland, College Park

[†]Vrije Universiteit Amsterdam



A Single Bit-flip Can Cause Terminal Brain Damage to DNNs

One specific bit-flip in a DNN's representation leads to accuracy drop over 90%

Our research found that a specific bit-flip in a DNN's bitwise representation can cause the accuracy loss up to 90%, and the DNN has 40-50% parameters, on average, that can lead to the accuracy drop over 10% when individually subjected to such single bitwise corruptions...

[Read More](#)

More Security Implications (VIII)

■ USENIX Security 2020

DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips

Fan Yao
University of Central Florida
fan.yao@ucf.edu

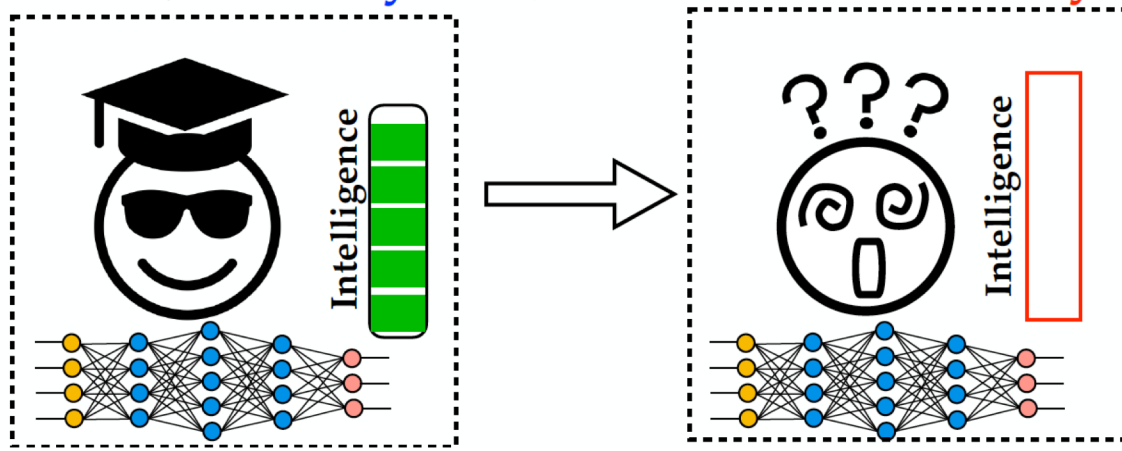
Adnan Siraj Rakin
Arizona State University
asrakin@asu.edu

Deliang Fan
Arizona State University
dfan@asu.edu

Degrade the inference accuracy to the level of Random Guess

Example: ResNet-20 for CIFAR-10, 10 output classes

Before attack, **Accuracy: 90.2%** After attack, **Accuracy: ~10% (1/10)**



More Security Implications?



Curious? First RowHammer Paper

- Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu,
"Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors"
Proceedings of the 41st International Symposium on Computer Architecture (ISCA), Minneapolis, MN, June 2014.
[[Slides \(pptx\)](#)] [[pdf](#)] [[Lightning Session Slides \(pptx\)](#)] [[pdf](#)] [[Source Code and Data](#)]

Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors

Yoongu Kim¹ Ross Daly* Jeremie Kim¹ Chris Fallin* Ji Hye Lee¹
Donghyuk Lee¹ Chris Wilkerson² Konrad Lai Onur Mutlu¹

¹Carnegie Mellon University ²Intel Labs

Curious? RowHammer: Now and Beyond...

- Onur Mutlu and Jeremie Kim,
[**"RowHammer: A Retrospective"**](#)
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) Special Issue on Top Picks in Hardware and Embedded Security, 2019.
[[Preliminary arXiv version](#)]
[[Slides from COSADE 2019 \(pptx\)](#)]
[[Slides from VLSI-SOC 2020 \(pptx\) \(pdf\)](#)]
[[Talk Video](#) (30 minutes)]

RowHammer: A Retrospective

Onur Mutlu^{§‡} Jeremie S. Kim^{‡§}
[§]ETH Zürich [‡]Carnegie Mellon University

RowHammer in 2020 (I)

- Jeremie S. Kim, Minesh Patel, A. Giray Yaglikci, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu,
"Revisiting RowHammer: An Experimental Analysis of Modern Devices and Mitigation Techniques"
Proceedings of the 47th International Symposium on Computer Architecture (ISCA), Valencia, Spain, June 2020.
[[Slides \(pptx\)](#)] [[pdf](#)]
[[Lightning Talk Slides \(pptx\)](#)] [[pdf](#)]
[[Talk Video](#) (20 minutes)]
[[Lightning Talk Video](#) (3 minutes)]

Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques

Jeremie S. Kim^{§†} Minesh Patel[§] A. Giray Yağlıkçı[§]
Hasan Hassan[§] Roknoddin Azizi[§] Lois Orosa[§] Onur Mutlu^{§†}
[§]*ETH Zürich* [†]*Carnegie Mellon University*

RowHammer in 2020 (II)

- Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi,
"TRRespass: Exploiting the Many Sides of Target Row Refresh"
Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, May 2020.
[[Slides \(pptx\)](#)] [[pdf](#)]
[[Lecture Slides \(pptx\)](#)] [[pdf](#)]
[[Talk Video](#)] (17 minutes)
[[Lecture Video](#)] (59 minutes)
[[Source Code](#)]
[[Web Article](#)]
Best paper award.
Pwnie Award 2020 for Most Innovative Research. [Pwnie Awards 2020](#)

TRRespass: Exploiting the Many Sides of Target Row Refresh

Pietro Frigo^{*†} Emanuele Vannacci^{*†} Hasan Hassan[§] Victor van der Veen[¶]
Onur Mutlu[§] Cristiano Giuffrida^{*} Herbert Bos^{*} Kaveh Razavi^{*}

RowHammer in 2020 (III)

- Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu,

"Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers"

Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, May 2020.

[[Slides \(pptx\)](#) ([pdf](#))]

[[Talk Video](#) (17 minutes)]

Are We Susceptible to Rowhammer?

An End-to-End Methodology for Cloud Providers

Lucian Cojocar, Jeremie Kim^{§†}, Minesh Patel[§], Lillian Tsai[‡],
Stefan Saroiu, Alec Wolman, and Onur Mutlu^{§†}
Microsoft Research, [§]ETH Zürich, [†]CMU, [‡]MIT

BlockHammer Solution in 2021

- A. Giray Yaglikci, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu,

"BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows"

Proceedings of the 27th International Symposium on High-Performance Computer Architecture (HPCA), Virtual, February-March 2021.

[[Slides \(pptx\)](#) ([pdf](#))]

[[Short Talk Slides \(pptx\)](#) ([pdf](#))]

[[Talk Video](#) (22 minutes)]

[[Short Talk Video](#) (7 minutes)]

BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows

A. Giray Yağlıkçı¹ Minesh Patel¹ Jeremie S. Kim¹ Roknoddin Azizi¹ Ataberk Olgun¹ Lois Orosa¹
Hasan Hassan¹ Jisung Park¹ Konstantinos Kanellopoulos¹ Taha Shahroodi¹ Saugata Ghose² Onur Mutlu¹

¹ETH Zürich

²University of Illinois at Urbana–Champaign

Google's Recent RowHammer Attack (May 2021)

Google Security Blog

The latest news and insights from Google on security and safety on the Internet

Introducing Half-Double: New hammering technique for DRAM Rowhammer bug

May 25, 2021

Research Team: Salman Qazi, Yoongu Kim, Nicolas Boichat, Eric Shiu & Mattias Nissler

Today, we are sharing details around our discovery of [Half-Double](#), a new Rowhammer technique that capitalizes on the worsening physics of some of the newer DRAM chips to alter the contents of memory.

Rowhammer is a DRAM vulnerability whereby repeated accesses to one address can tamper with the data stored at other addresses. Much like speculative execution vulnerabilities in CPUs, Rowhammer is a breach of the security guarantees made by the underlying hardware. As an electrical coupling phenomenon within the silicon itself, Rowhammer allows the potential bypass of hardware and software memory protection policies. This can allow untrusted code to break out of its sandbox and take full control of the system.

Google's Recent RowHammer Attack (May 2021)



- Given three consecutive rows A, B, and C, we were able to attack C by directing a very large number of accesses to A, along with just a handful (~dozens) to B.
- Based on our experiments, accesses to B have a non-linear gating effect, in which they appear to “transport” the Rowhammer effect of A onto C.
- This is likely an indication that the electrical coupling responsible for **Rowhammer** is a property of distance, **effectively becoming stronger** and longer-ranged as cell geometries shrink down.

The Story of RowHammer Lecture ...

- Onur Mutlu,
"The Story of RowHammer"
Keynote Talk at *Secure Hardware, Architectures, and Operating Systems Workshop (SeHAS)*, held with *HiPEAC 2021 Conference*, Virtual, 19 January 2021.
[Slides (pptx)] (pdf)
[Talk Video] (1 hr 15 minutes, with Q&A)]



The video player shows a presentation slide titled "The Story of RowHammer" by Onur Mutlu. The slide includes contact information: omutlu@gmail.com, <https://people.inf.ethz.ch/omutlu>, and the date 19 January 2021. It also mentions "SEHAS Keynote @ HiPEAC" and features logos for SAFARI, ETH zürich, and Carnegie Mellon. The video player interface shows a progress bar at 58:18 / 1:14:41 and a video feed of Onur Mutlu on the right. Below the player, the video title "The Story of Rowhammer - Secure Hardware, Architectures, and Operating Systems Keynote - Onur Mutlu" is displayed, along with 1,293 views, a premiere date of Feb 2, 2021, and 64 likes. The video URL is <https://www.youtube.com/watch?v=sgd7PHQQ1AI>. The SAFARI logo is in the bottom left, and the Onur Mutlu Lectures channel name (13.9K subscribers) is in the bottom left of the video player area. Buttons for "ANALYTICS" and "EDIT VIDEO" are in the bottom right of the video player area.

The Story of Rowhammer - Secure Hardware, Architectures, and Operating Systems Keynote - Onur Mutlu

1,293 views • Premiered Feb 2, 2021

<https://www.youtube.com/watch?v=sgd7PHQQ1AI>

ANALYTICS EDIT VIDEO

Detailed Lectures on RowHammer

- Computer Architecture, Fall 2020, Lecture 4b
 - RowHammer (ETH Zürich, Fall 2020)
 - <https://www.youtube.com/watch?v=KDy632z23UE&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=8>
- Computer Architecture, Fall 2020, Lecture 5a
 - RowHammer in 2020: TRRespass (ETH Zürich, Fall 2020)
 - https://www.youtube.com/watch?v=pwRw7QqK_qA&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=9
- Computer Architecture, Fall 2020, Lecture 5b
 - RowHammer in 2020: Revisiting RowHammer (ETH Zürich, Fall 2020)
 - <https://www.youtube.com/watch?v=gR7XR-Eepecg&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=10>
- Computer Architecture, Fall 2020, Lecture 5c
 - Secure and Reliable Memory (ETH Zürich, Fall 2020)
 - <https://www.youtube.com/watch?v=HvswnsfG3oQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=11>

Takeaway and Food for Thought

- If hardware is unreliable, higher-level security and protection mechanisms (as in virtual memory) may be compromised
- The root of security and trust is at the very low levels...
 - in the hardware itself
 - RowHammer, Spectre, Meltdown are recent key examples...
- What should we assume the hardware provides?
- How do we keep hardware reliable?
- How do we design secure hardware?
- How do we design secure hardware with high performance, high energy efficiency, low cost, convenient programming?

Plenty of exciting and highly-relevant research questions

Some Issues in Virtual Memory

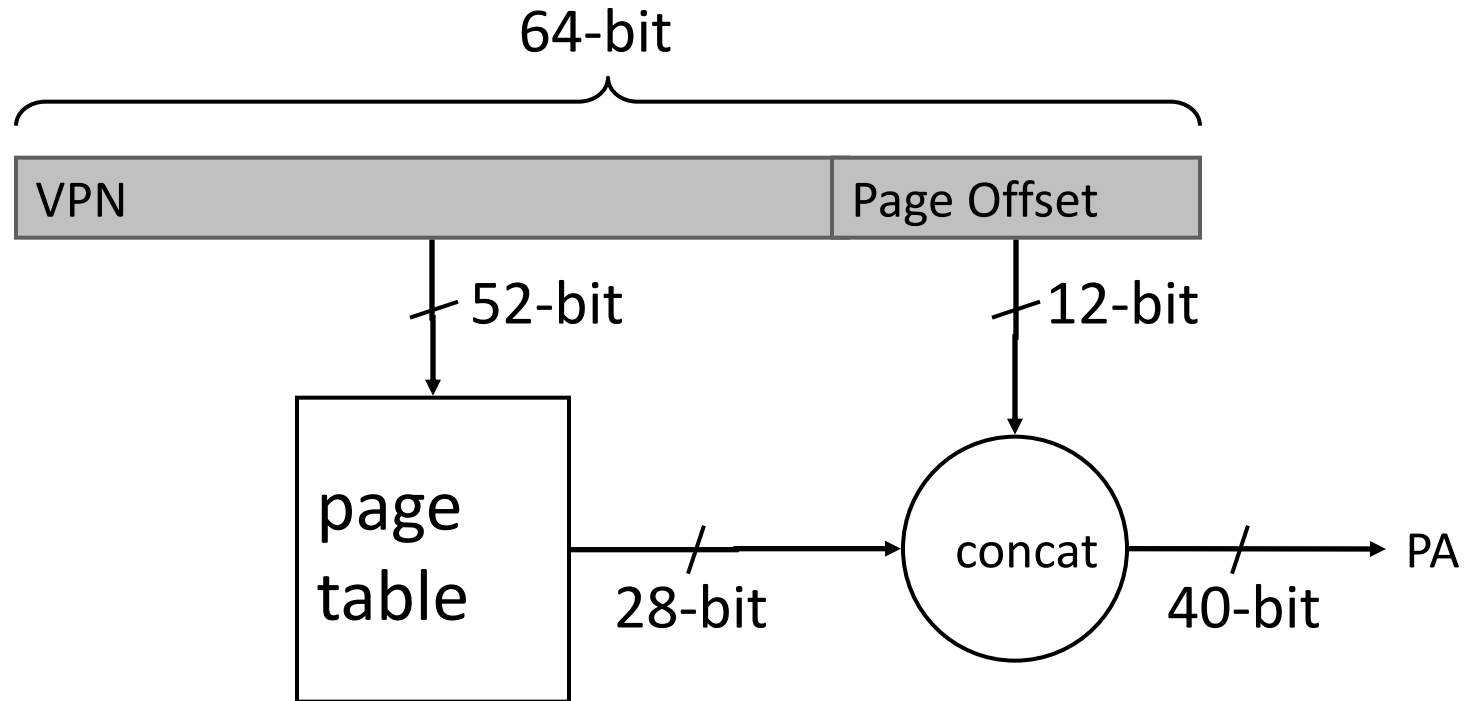
Three Major Issues in Virtual Memory

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 I

- How large is the page table?
- Where do we store it?
 - In hardware?
 - In physical memory? (Where is the PTBR?)
 - In virtual memory? (Where is the PTBR?)
- How can we store it efficiently without requiring physical memory that can store all page tables?
 - Idea: multi-level page tables
 - Only the first-level page table has to be in physical memory
 - Remaining levels are in virtual memory (but get cached in physical memory when accessed)

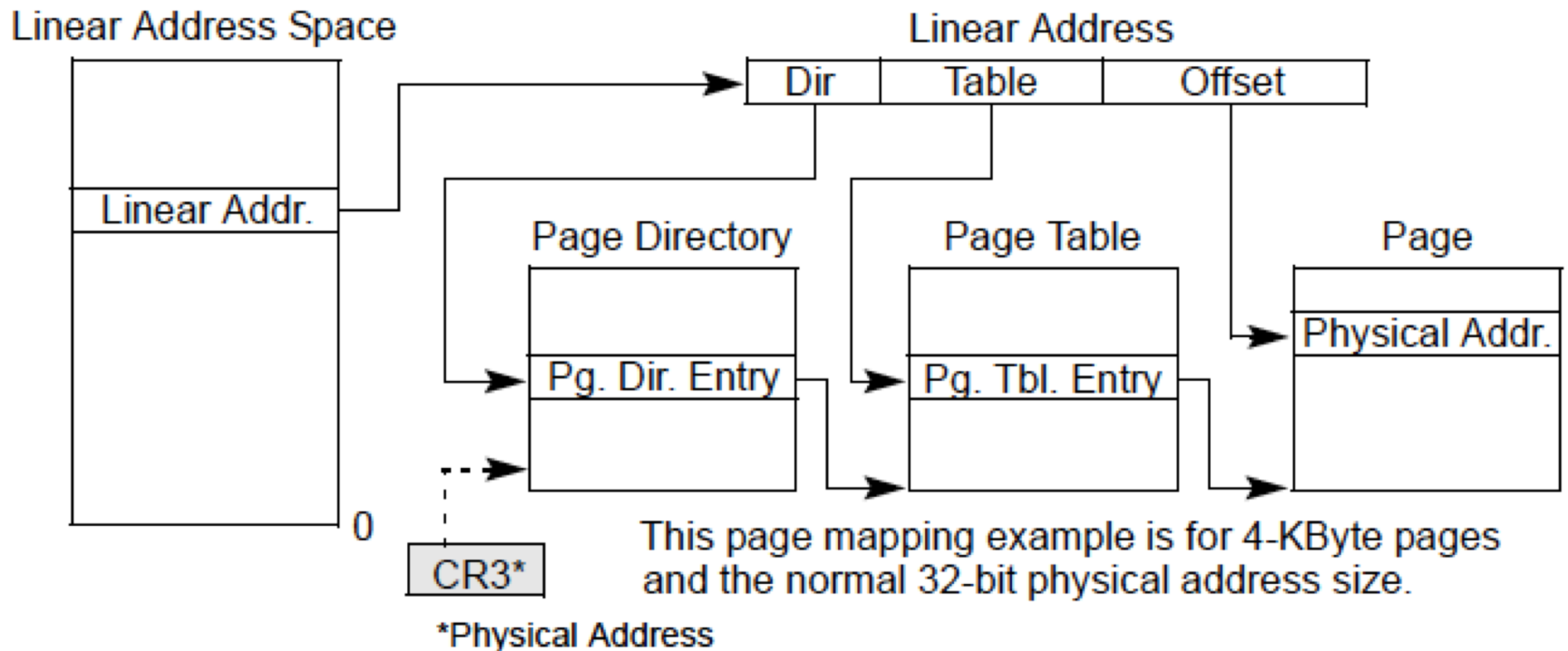
Recall: Issue: Page Table Size



- Suppose 64-bit VA and 40-bit PA, how large is the page table?
 - **2^{52} entries x ~4 bytes $\approx 2^{54}$ bytes**
and that is for just one process!
and the process may not be using the entire VM space!

Recall: Solution: Multi-Level Page Tables

Example from the x86 architecture



Page Table Access

- How do we access the Page Table?
- Page Table Base Register (CR3 in x86)
- Page Table Limit Register
- If VPN is out of the bounds (exceeds PTLR) then the process did not allocate the virtual page → access control exception
- Page Table Base Register is part of a process's context
 - Just like PC, status registers, general purpose registers
 - Needs to be loaded when the process is context-switched in

More on x86 Page Tables (I): Small Pages

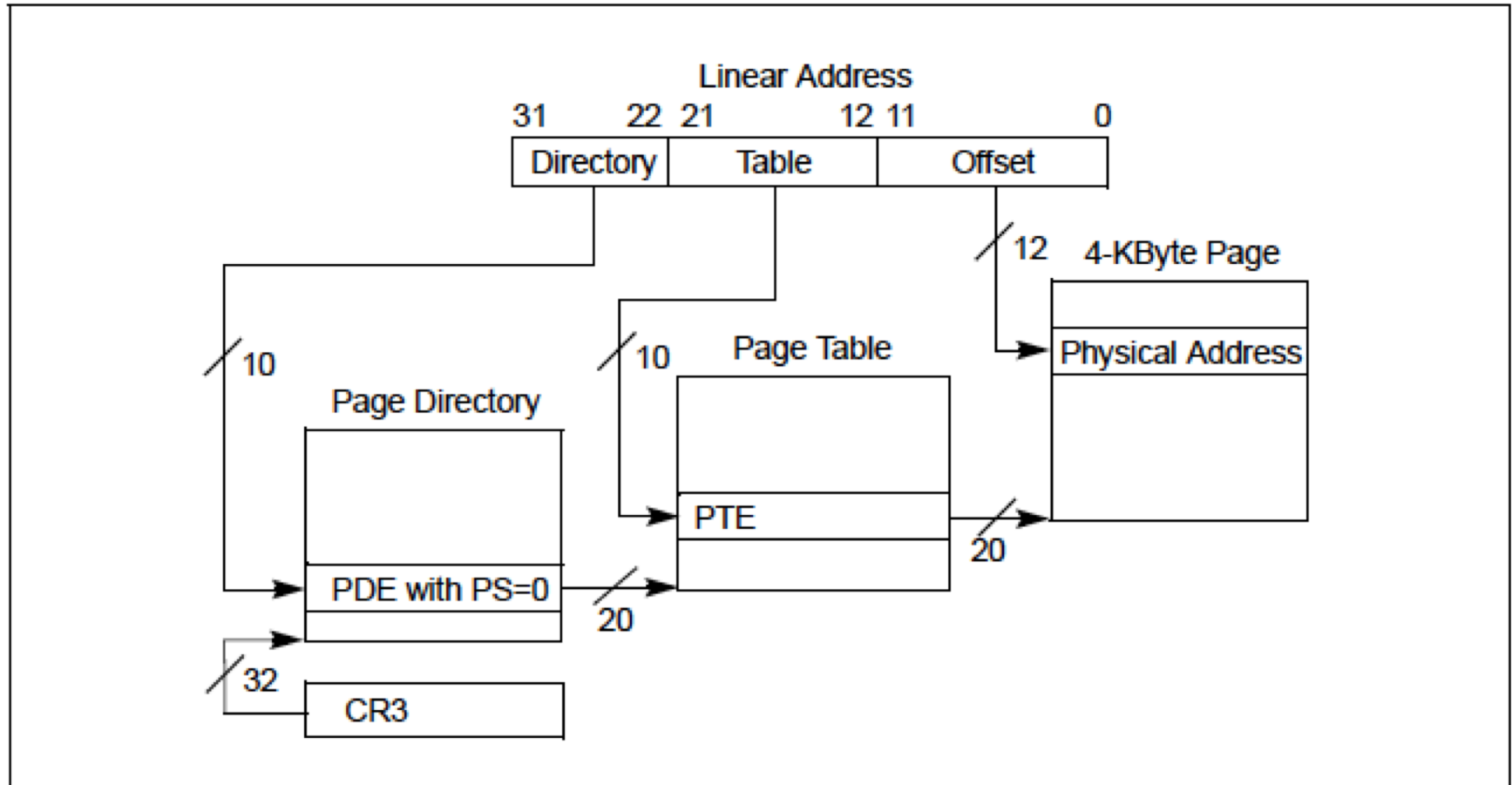


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

More on x86 Page Tables (II): Large Pages

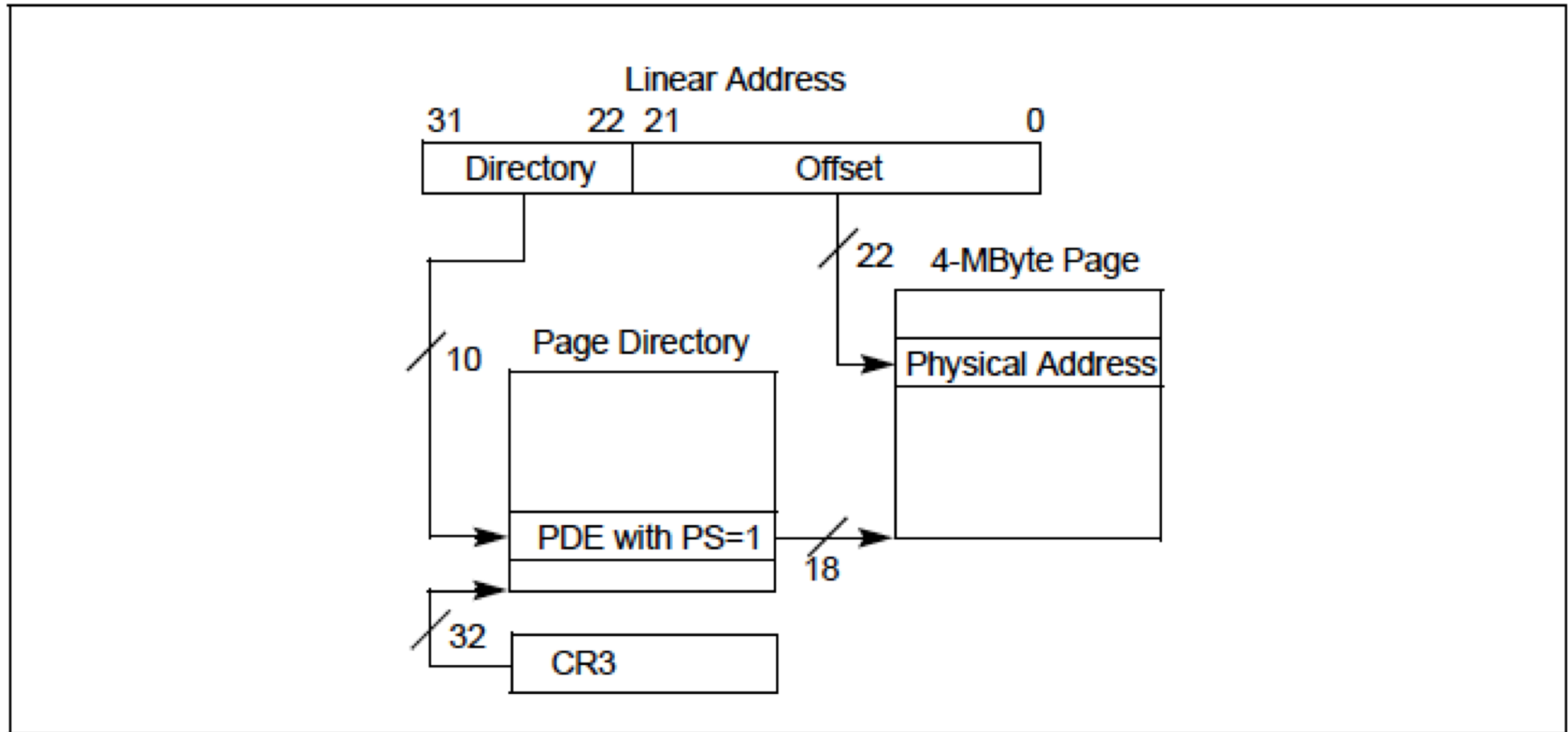


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

x86 Page Table Entries

Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are “not present”; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory ¹																Ignored				P C D	P W T	Ignored				CR3						
Bits 31:22 of address of 2MB page frame								Reserved (must be 0)				Bits 39:32 of address ²				P A T	Ignored	G	<u>1</u>	D	A	P C D	P W T	U / S	R / W	<u>1</u>	PDE: 4MB page					
Address of page table																Ignored				<u>0</u>	I g n	A	P C D	P W T	U / S	R / W	<u>1</u>	PDE: page table				
Ignored																												<u>0</u>	PDE: not present			
Address of 4KB page frame																Ignored	G	P A T	D	A	P C D	P W T	U / S	R / W	<u>1</u>	PTE: 4KB page						
Ignored																												<u>0</u>	PTE: not present			

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86 PTE (4KB page)

Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0) ¹
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

x86 Page Directory Entry (PDE)

Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)

X86-64 Page Table Entry Structure

6	6	6	6	5	5	5	5	5	5	5	5	5	5	5	5	M ¹	M-1			3	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	0	
Reserved ²																	Address of PML4 table (4-level paging) or PML5 table (5-level paging)															Ignored			P C W D	P W T	Ign.	CR3							
X D 3	Ignored										Rsvd.					Address of PML4 table															Ign.			R s v d	A	P C W D	P W T	R / W	1	PML5E: present					
Ignored																																		0	PML5E: not present										
X D 3	Ignored										Rsvd.					Address of page-directory-pointer table															Ign.			R s v d	A	P C W D	P W T	R / W	1	PML4E: present					
Ignored																																		0	PML4E: not present										
X D 3	Prot. Key ⁴	Ignored								Rsvd.					Address of 1GB page frame										Reserved					P A T	Ign.	G	1	D	A	P C W D	P W T	R / W	1	PDPTE: 1GB page					
X D 3	Ignored										Rsvd.					Address of page directory															Ign.			0	A	P C W D	P W T	R / W	1	PDPTE: page directory					
Ignored																																		0	PDPTE: not present										
X D 3	Prot. Key ⁴	Ignored								Rsvd.					Address of 2MB page frame										Reserved					P A T	Ign.	G	1	D	A	P C W D	P W T	R / W	1	PDE: 2MB page					
X D 3	Ignored										Rsvd.					Address of page table															Ign.			0	A	P C W D	P W T	R / W	1	PDE: page table					
Ignored																																		0	PDE: not present										
X D 3	Prot. Key ⁴	Ignored								Rsvd.					Address of 4KB page frame										Ign.					G	P A T	D	A	P C W D	P W T	R / W	1	PTE: 4KB page							
Ignored																																		0	PTE: not present										

Figure 4-11. Formats of CR3 and Paging-Structure Entries with 4-Level Paging and 5-Level Paging

X86-64 Page Table: Accessing 4KB pages

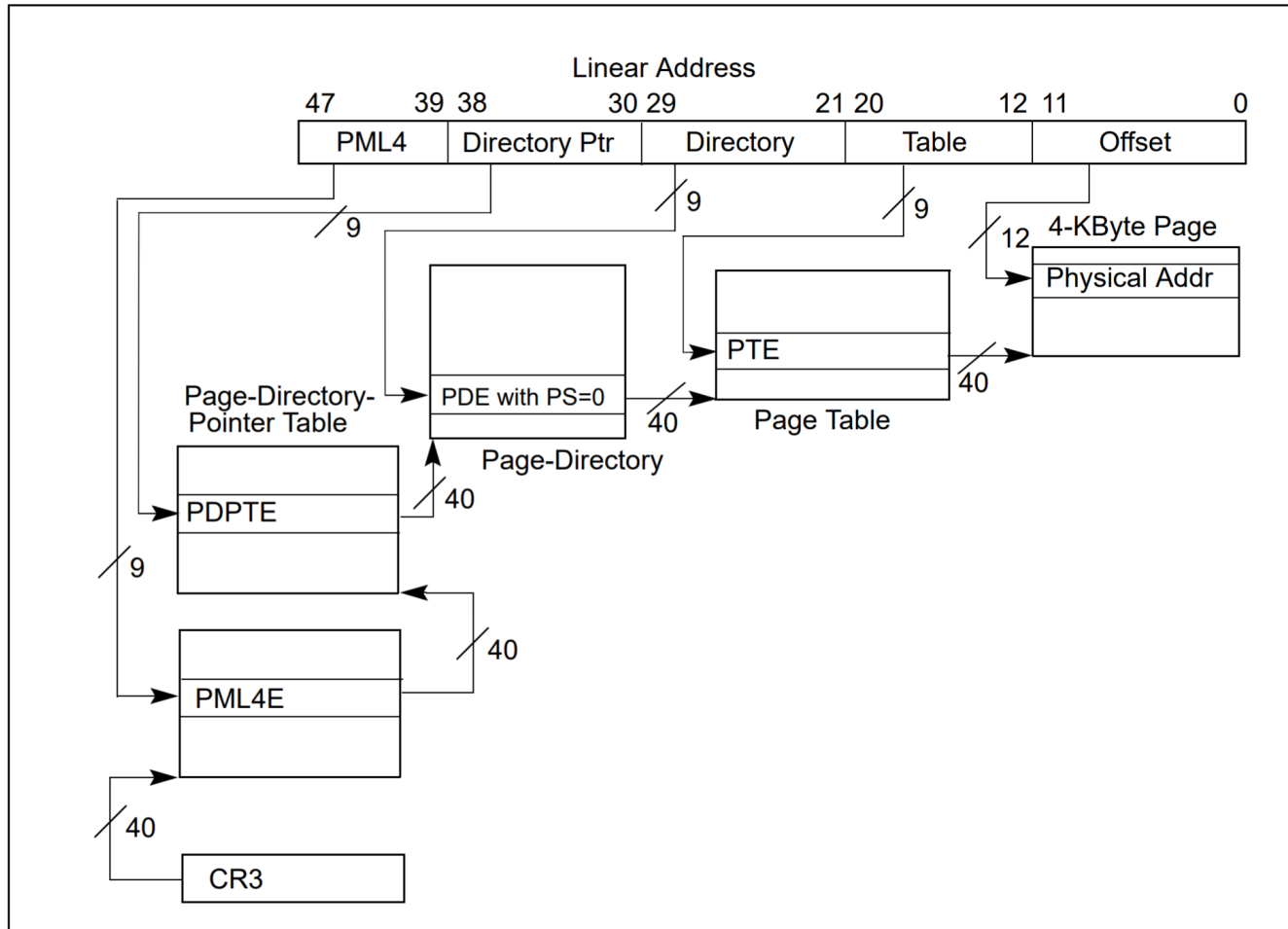


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

X86-64 Page Table: Accessing 2MB pages

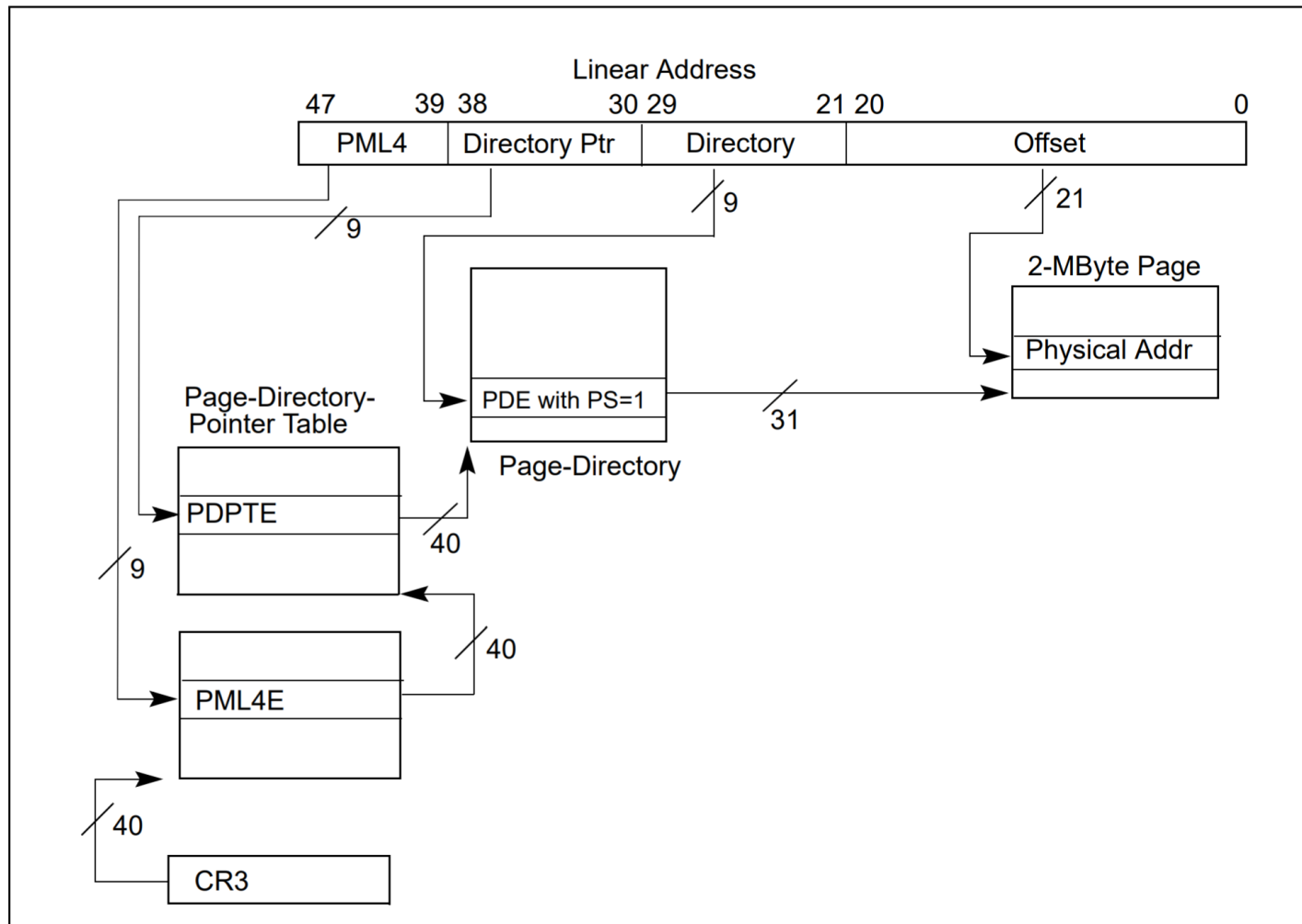


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

X86-64 Page Table: Accessing 1GB pages

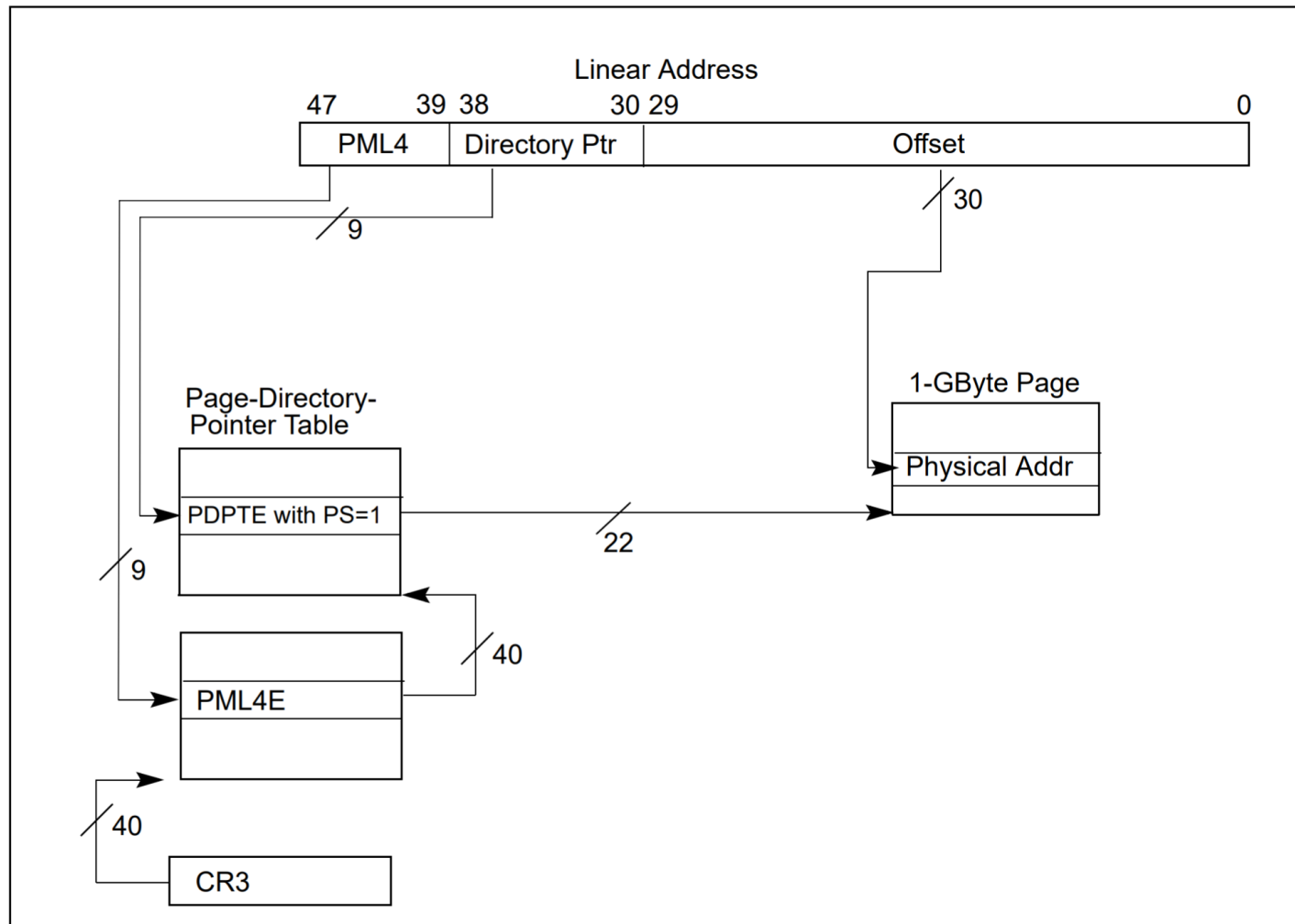


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

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?
 - ...

Recall: Translation Lookaside Buffer (TLB)

- Idea: Cache the page table entries (PTEs) in a hardware structure in the processor to speed up address translation
- Translation lookaside buffer (TLB)
 - ❑ Small cache of most recently used translations (PTEs)
 - ❑ Reduces number of memory accesses required for *most* instruction fetches and loads/stores to only one

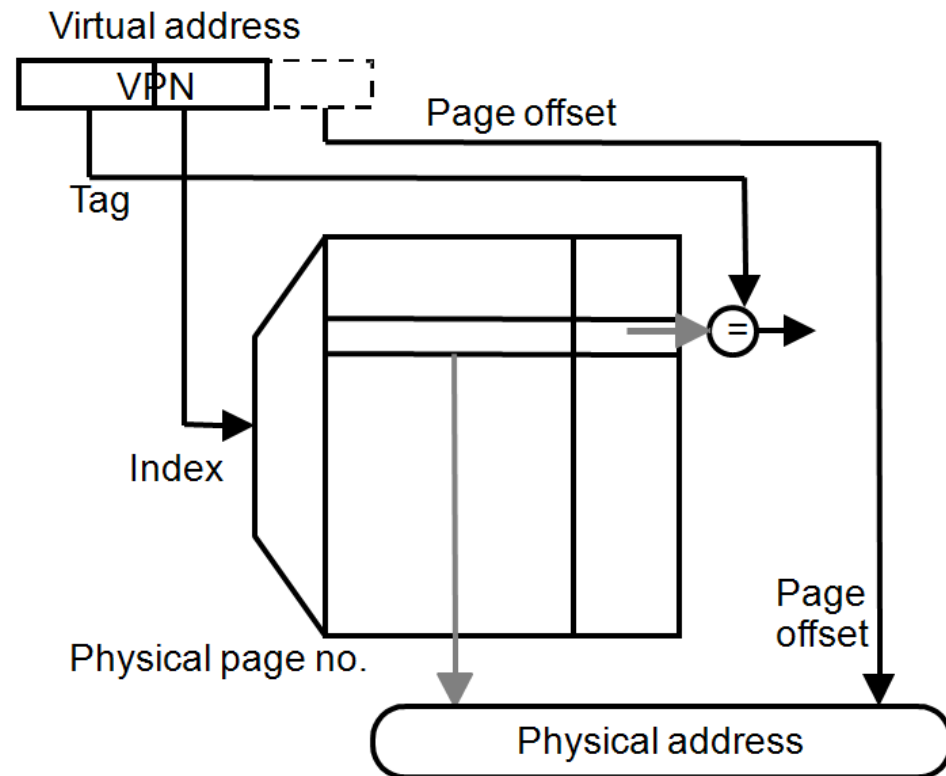
Virtual Memory Issue II

- How fast is the address translation?
 - How can we make it fast?
- Idea: Use a hardware structure that caches PTEs → Translation Lookaside Buffer (TLB)
- What should be done on a TLB miss?
 - What TLB entry to replace?
 - Who handles the TLB miss? HW vs. SW?
- What should be done on a page fault?
 - What virtual page to replace from physical memory?
 - Who handles the page fault? HW vs. SW?

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.



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 etched 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
-

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?
 - ...

Teaser: Virtual Memory Issue III

- When do we do the address translation?
 - Before or after accessing the L1 cache?

Address Translation and Caching

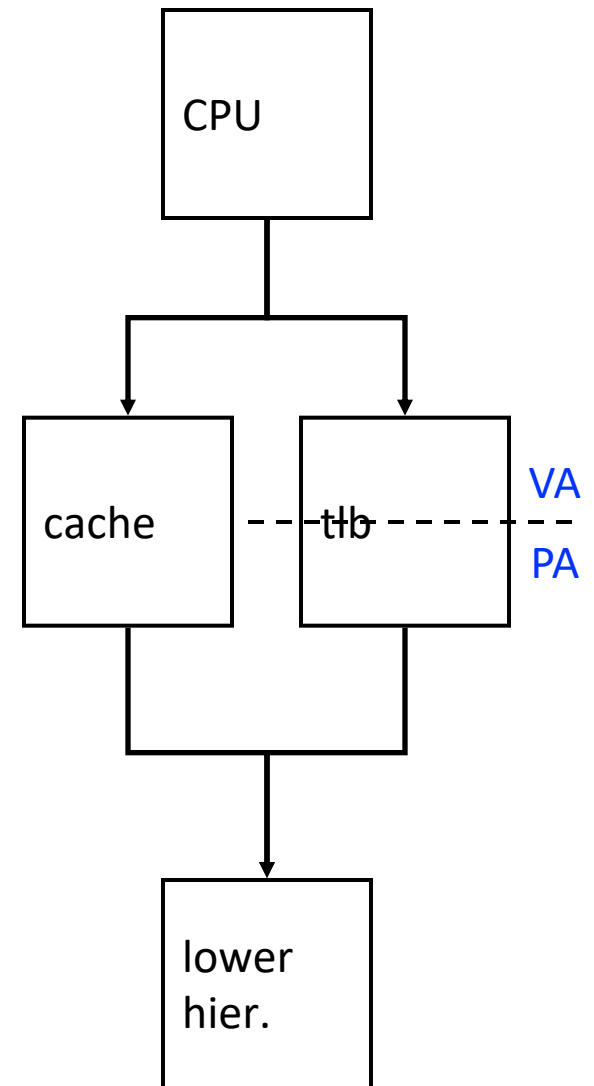
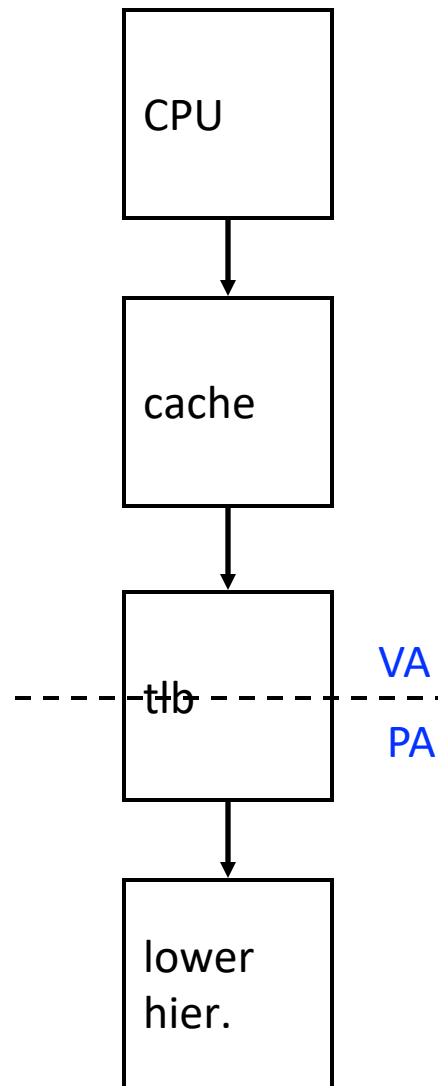
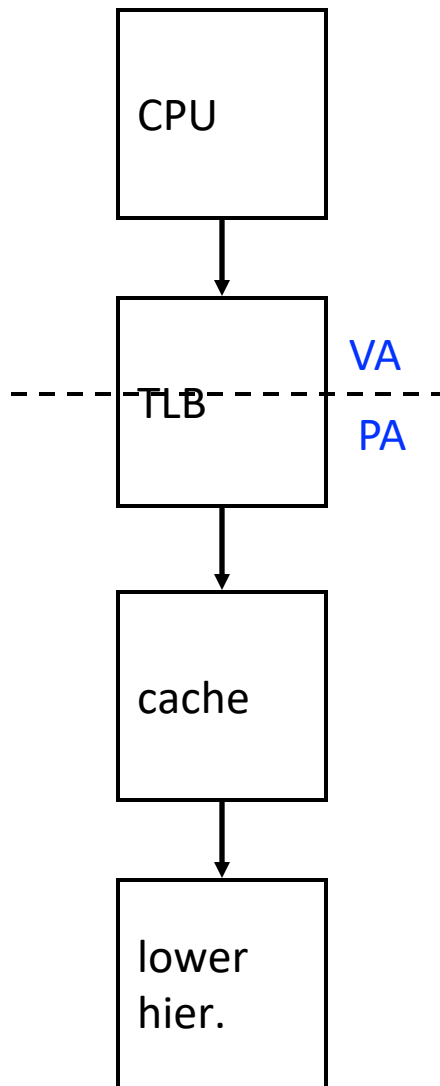
- When do we do the address translation?
 - Before or after accessing the L1 cache?
- In other words, is the cache virtually addressed or physically addressed?
 - Virtual versus physical cache
- What are the issues with a virtually addressed cache?
- **Synonym problem:**
 - Two different virtual addresses 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

See backup slides for more



physical cache

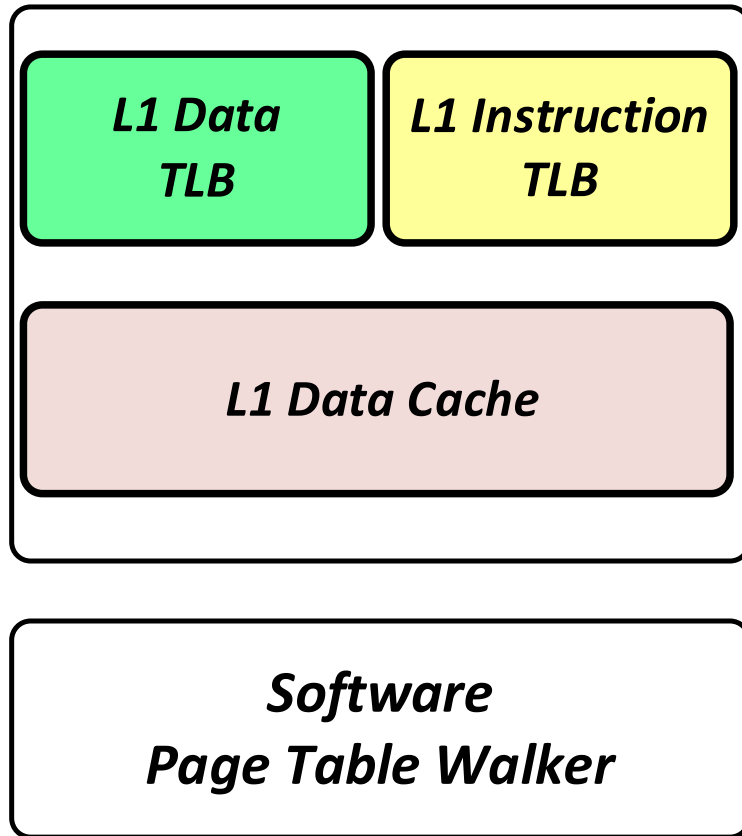
virtual (L1) cache

virtual-physical cache¹³¹

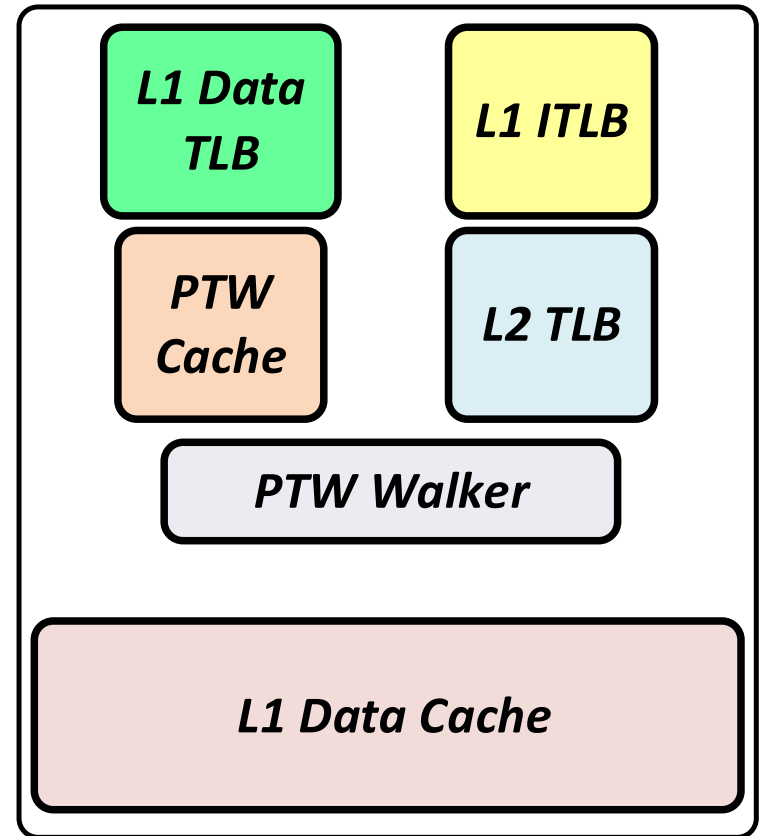
A Modern Example Virtual Memory System

Evolution of Address Translation

Simple Address Translation



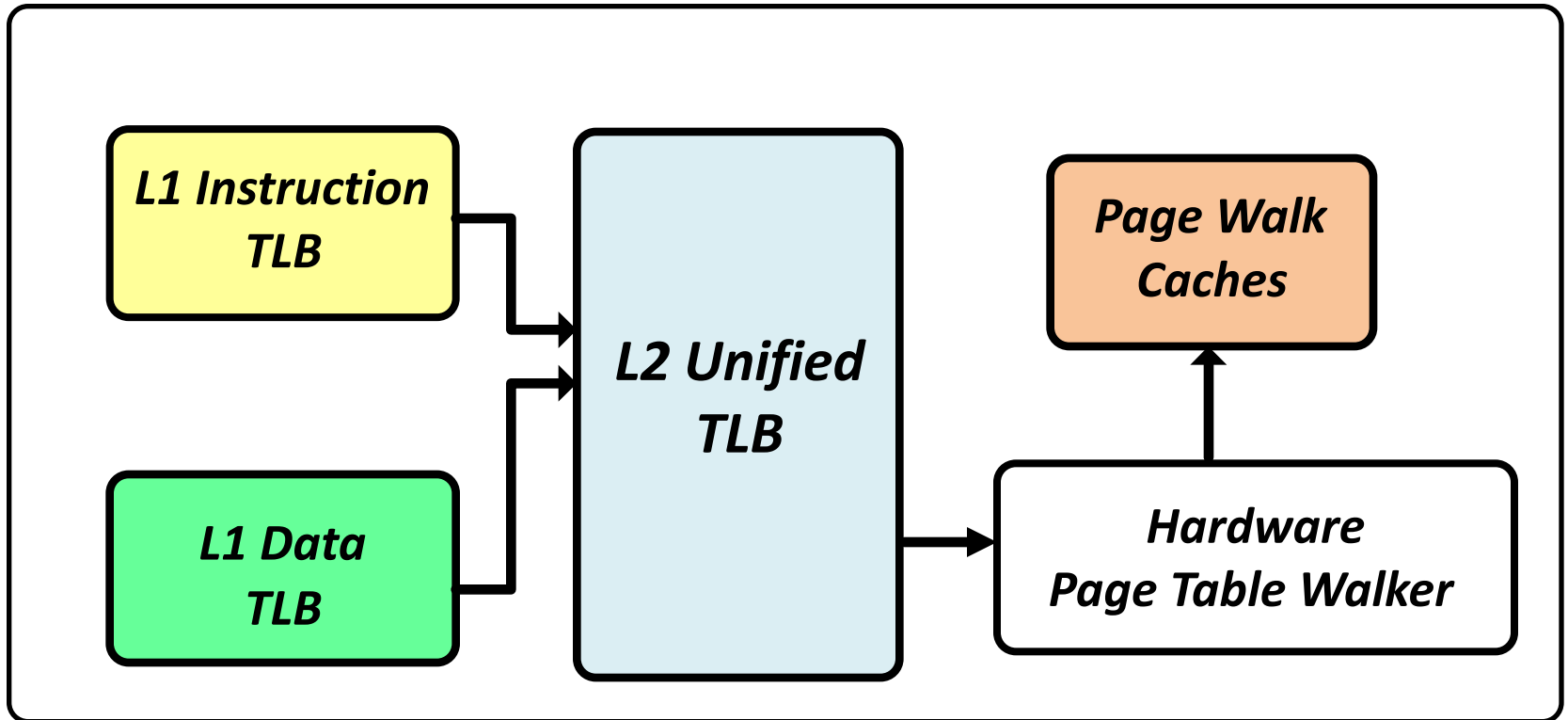
Modern Address Translation



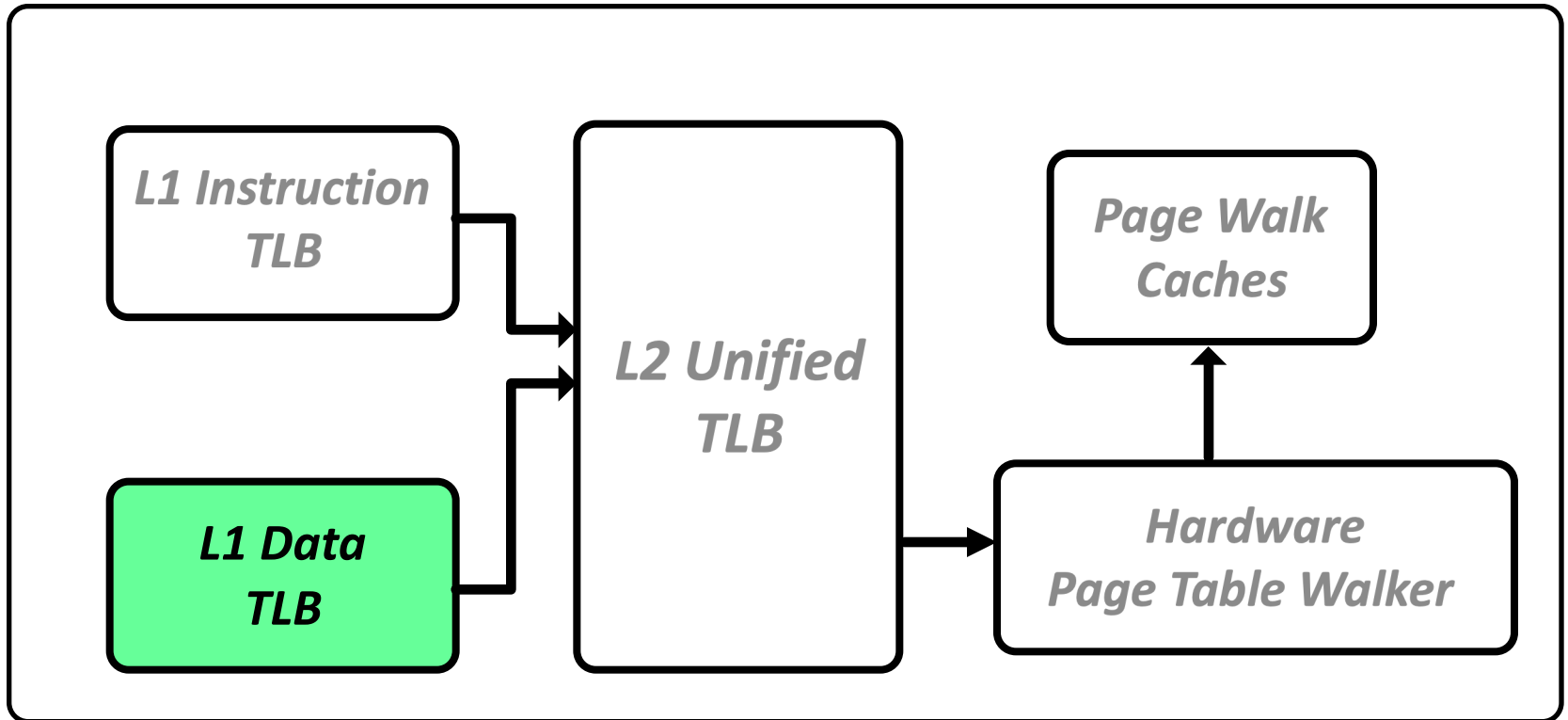
Memory Management Unit

- 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



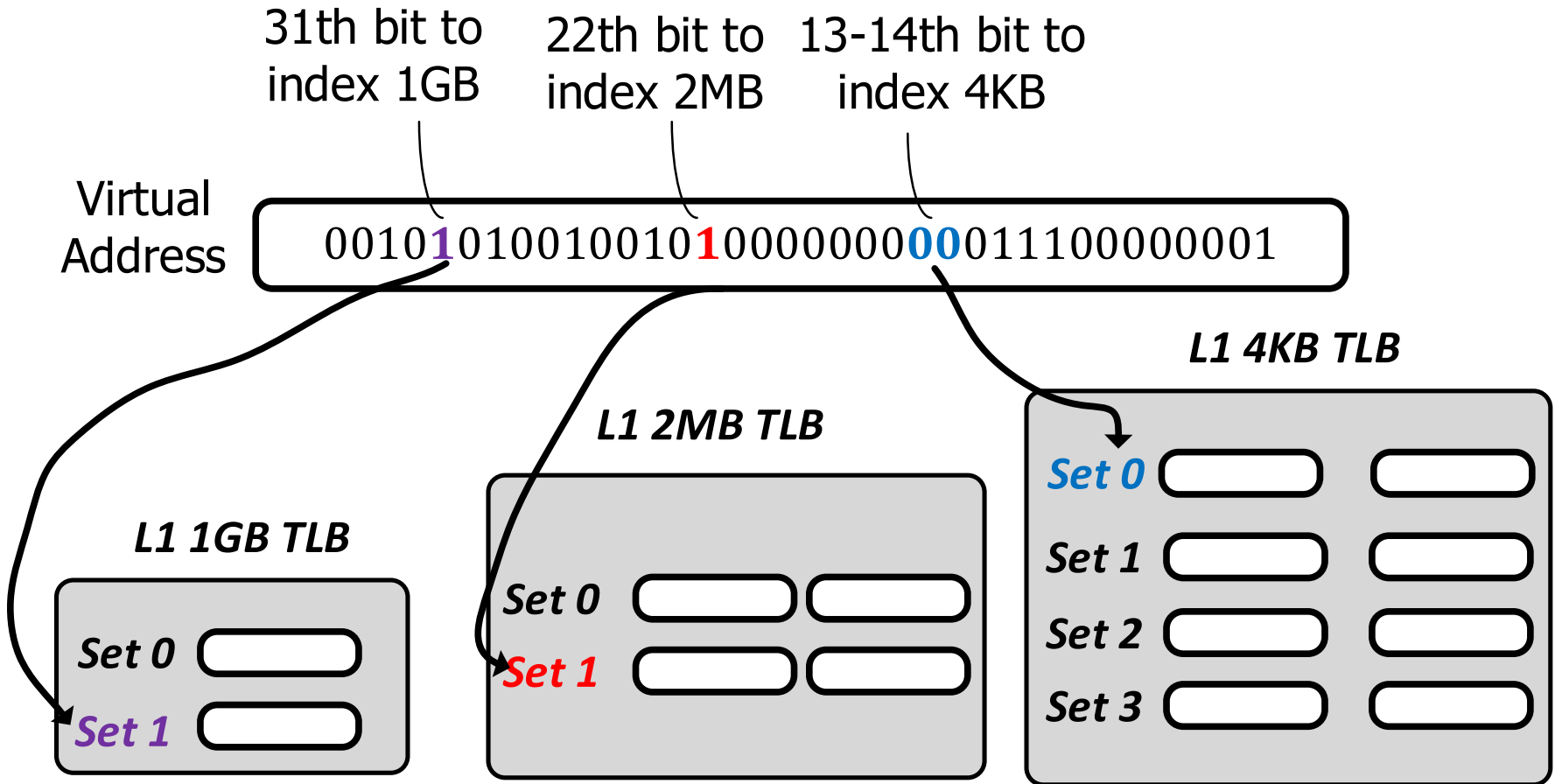
Intel Skylake: L1 Data TLB



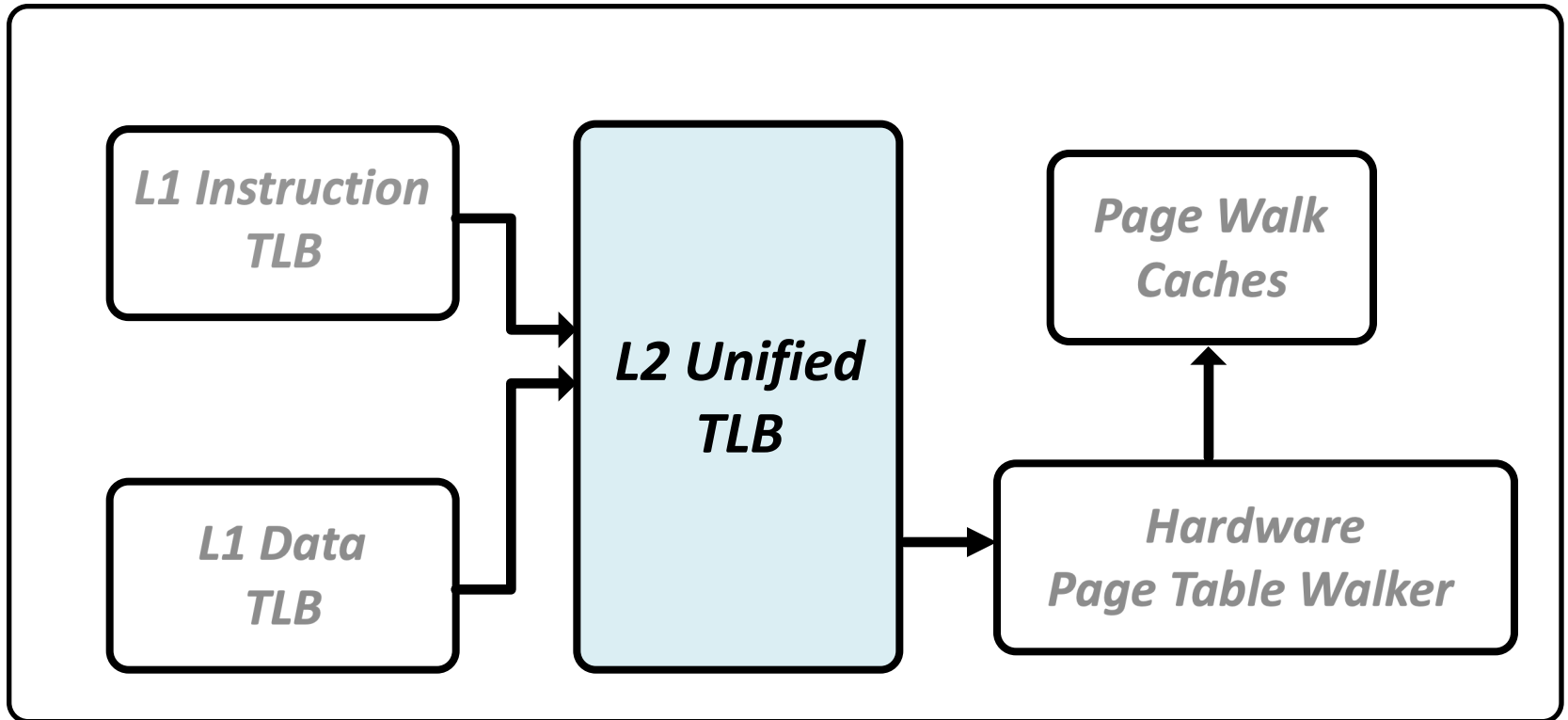
Intel Skylake: L1 Data TLB

- 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



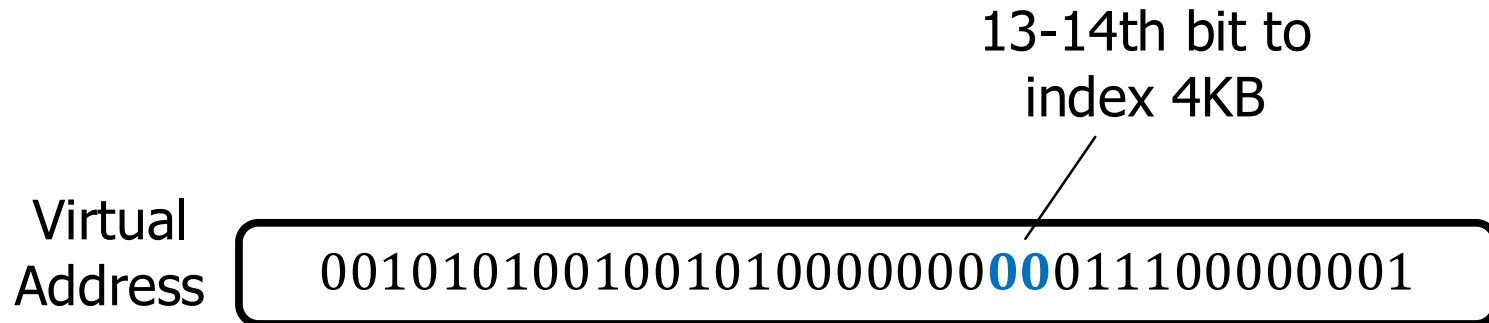
Intel Skylake: L2 Unified TLB

- 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)

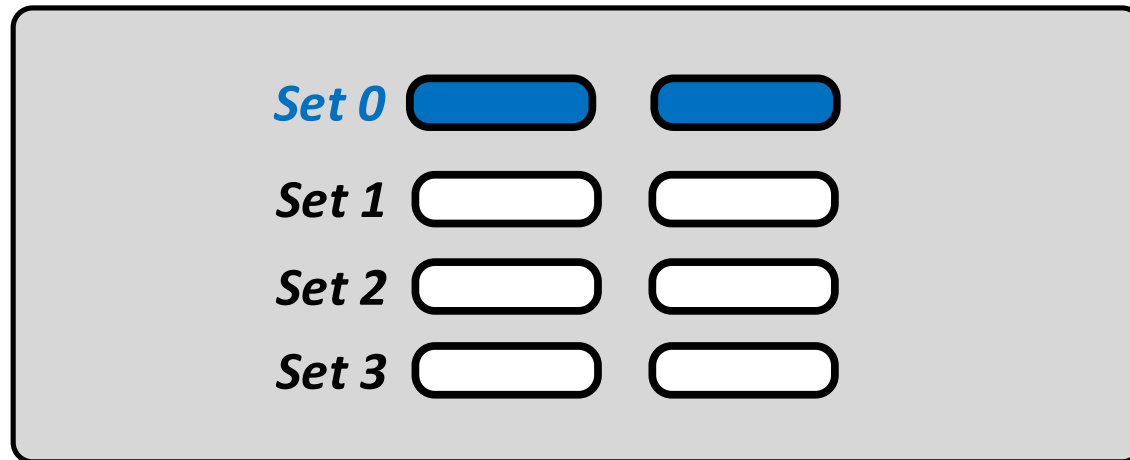
L2 Unified TLB: Accessing the TLB

- 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

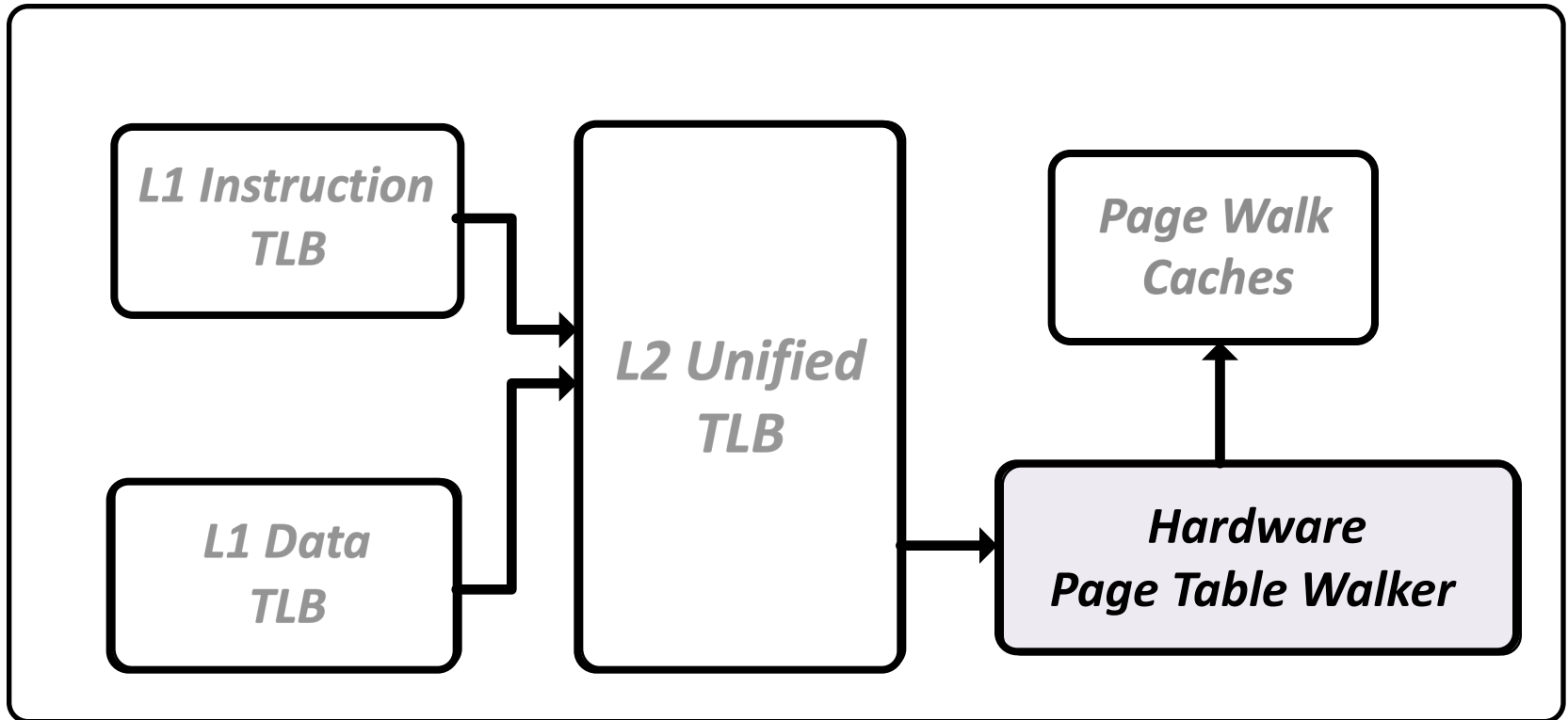
Set 0	<input type="text"/>	<input type="text"/>
Set 1	<input type="text"/>	<input type="text"/>
Set 2	<input type="text"/>	<input type="text"/>
Set 3	<input type="text"/>	<input type="text"/>

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

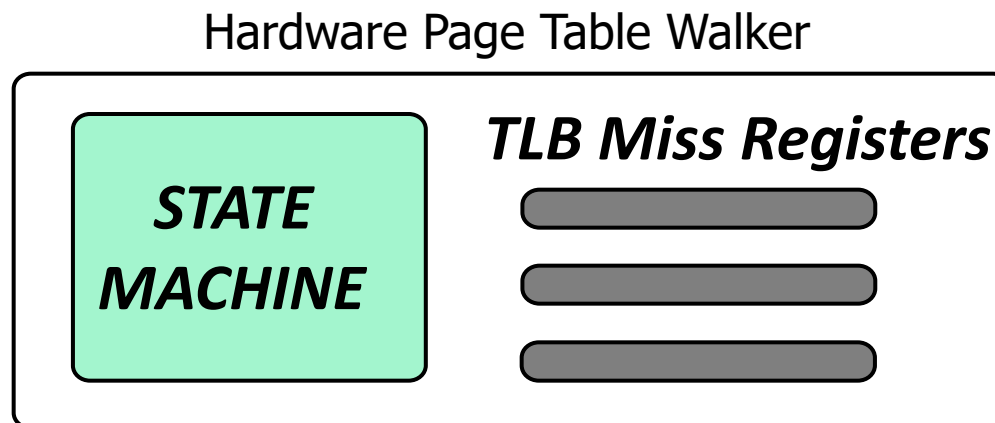
Tradeoffs are similar to “associativity in time” (also called pseudo-associativity)

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

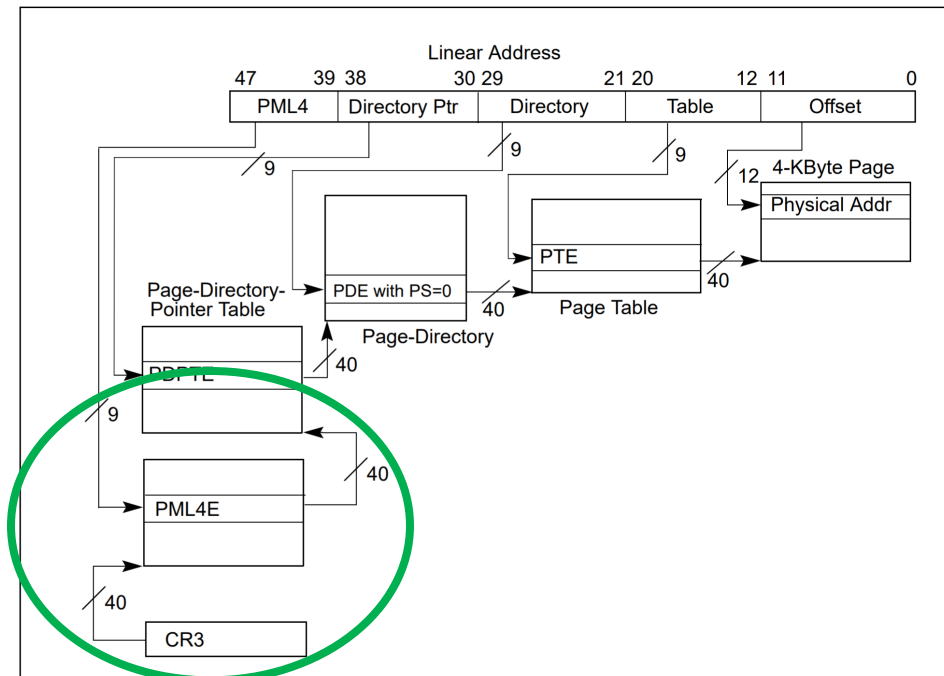


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

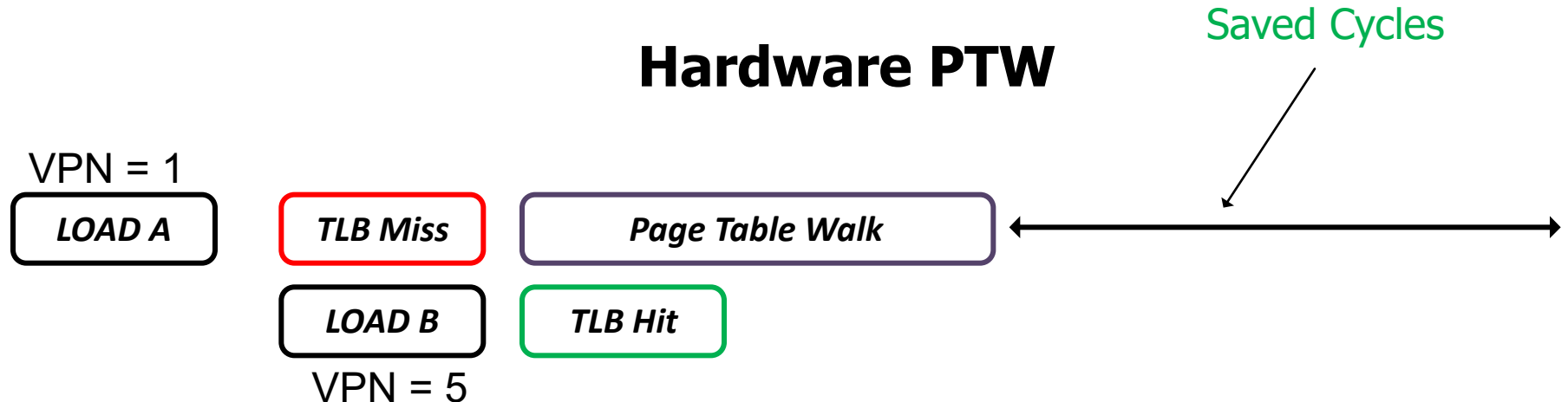
Hardware Page Table Walker (IV)

- Hardware PTWs allow overlapping TLB misses with useful computation

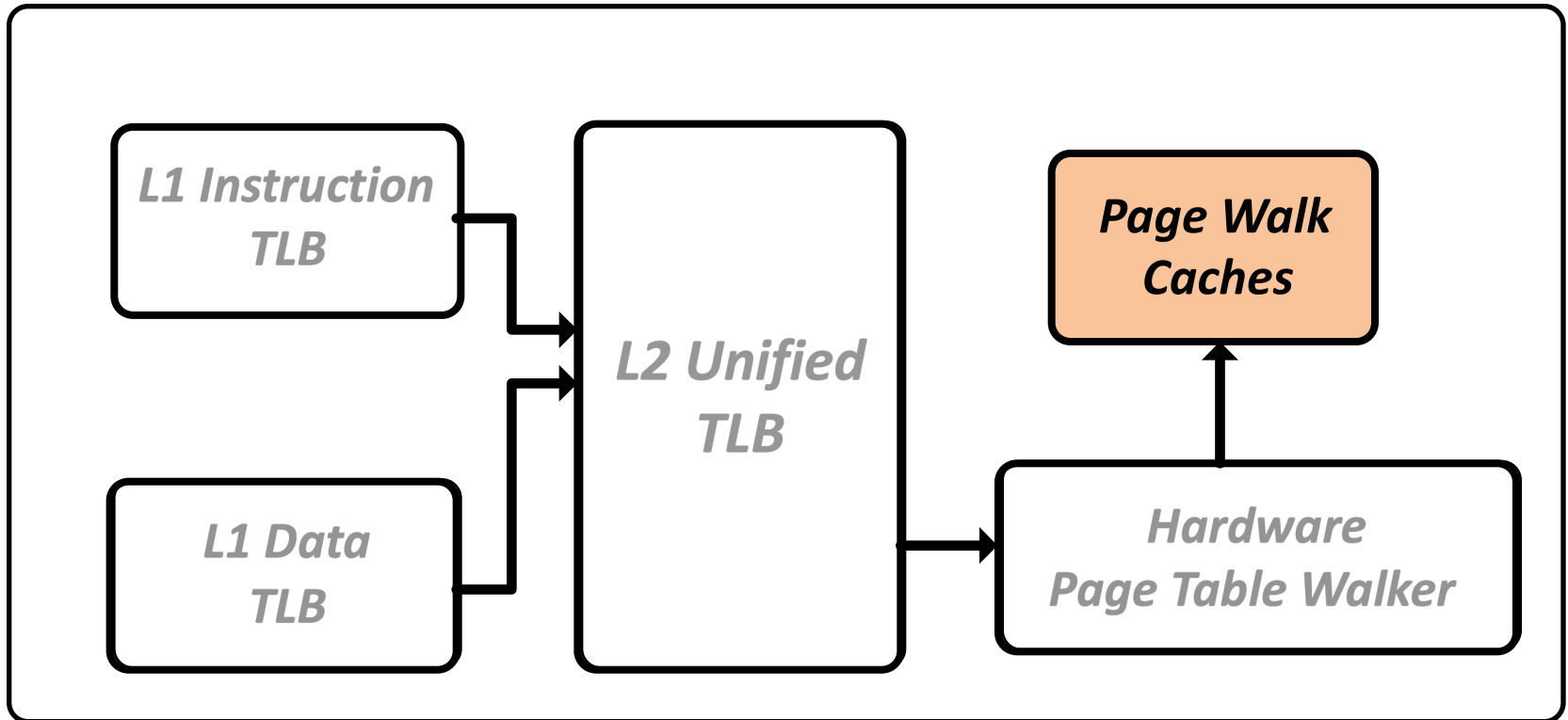
Software PTW



Hardware PTW



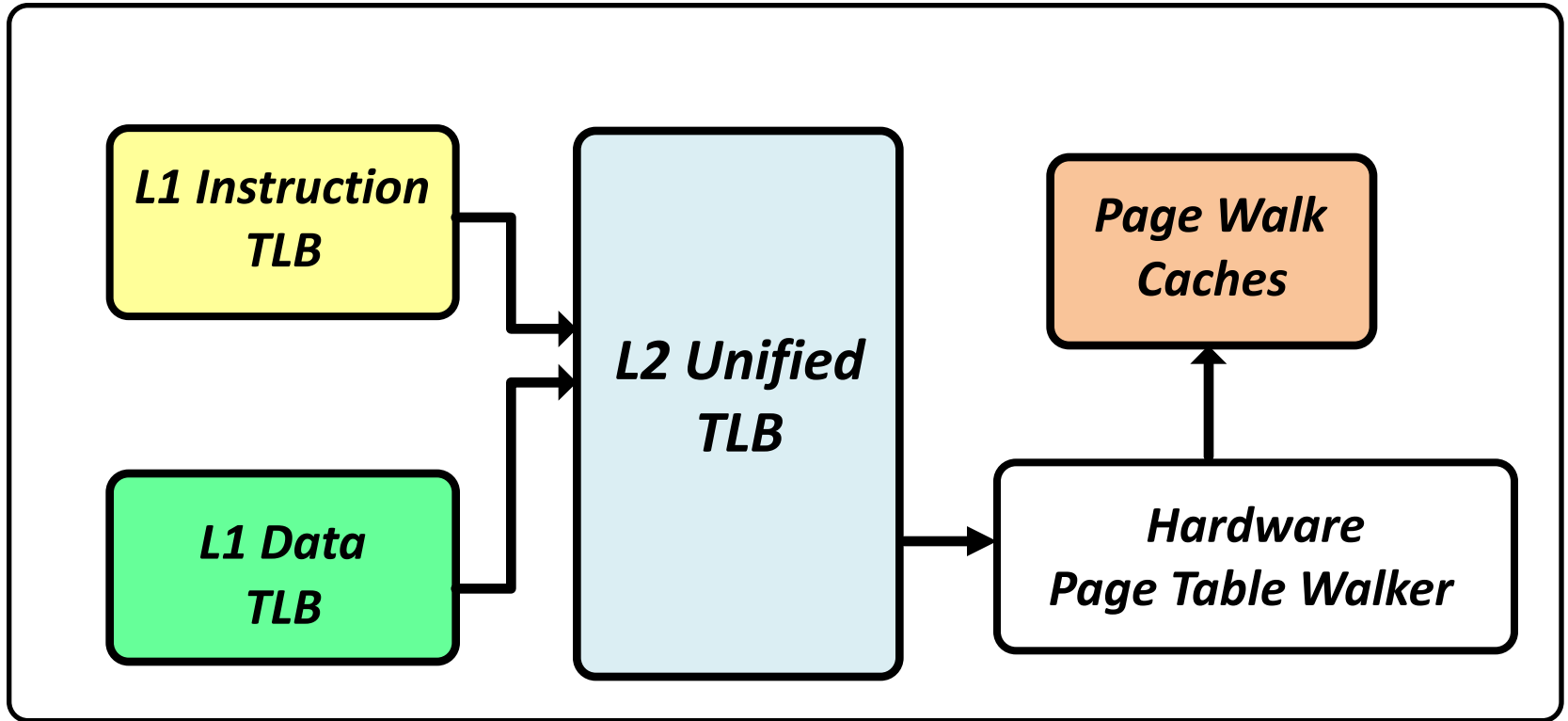
Page Walk Caches



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
 - 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 Summary

Virtual Memory Summary

- Virtual memory gives the illusion of “infinite” capacity
- A subset of virtual pages are located in physical memory
- A page table maps virtual pages to physical pages – this is called address translation
- A TLB speeds up address translation
- Multi-level page tables keep the page table size in check
- Using different page tables for different programs provides memory protection

There is More... We Will Not Cover...

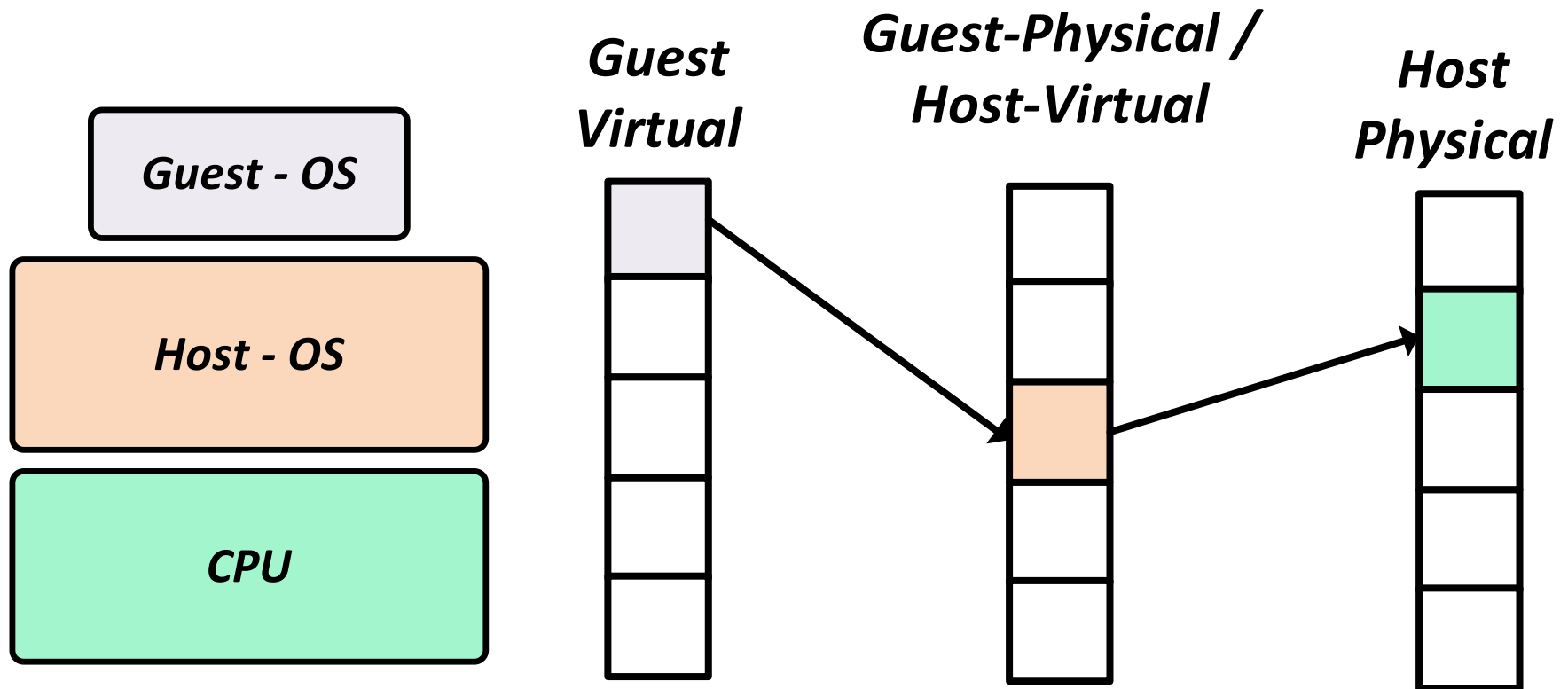
- How to handle virtualized systems?
 - Virtual machines running programs
 - Hypervisors

- Alternative page table structures
 - Hashed page tables
 - Inverted page tables
 - ...

- ...

Virtual Memory in Virtualized Environments

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



Virtual Memory: Parting Thoughts

- VM is one of the most successful examples of
 - ❑ architectural support for programmers
 - ❑ how to partition work between hardware and software
 - ❑ hardware/software cooperation
 - ❑ programmer/architect tradeoff
- Going forward: How does virtual memory scale into the future? Four key trends:
 - ❑ Increasing, huge physical memory sizes (local & remote)
 - ❑ Hybrid physical memory systems (DRAM + NVM + SSD)
 - ❑ Many accelerators in the system addressing physical memory
 - ❑ Virtualized systems (hypervisors, software virtualization, local and remote memories)

Rethinking Virtual Memory

Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo Francisco de Oliveira Jr., Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu, **"The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework"**

Proceedings of the 47th International Symposium on Computer Architecture (ISCA), Virtual, June 2020.

[[Slides \(pptx\)](#) ([pdf](#))]

[[Lightning Talk Slides \(pptx\)](#) ([pdf](#))]

[[ARM Research Summit Poster \(pptx\)](#) ([pdf](#))]

[[Talk Video](#) (26 minutes)]

[[Lightning Talk Video](#) (3 minutes)]

[[Lecture Video](#) (43 minutes)]

The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework

Nastaran Hajinazar^{*†} Pratyush Patel[⌘] Minesh Patel^{*} Konstantinos Kanellopoulos^{*} Saugata Ghose[‡]
Rachata Ausavarungnirun[⊙] Geraldo F. Oliveira^{*} Jonathan Appavoo[◇] Vivek Seshadri[▽] Onur Mutlu^{*‡}

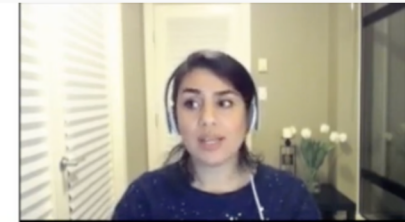
^{*}ETH Zürich [†]Simon Fraser University [⌘]University of Washington [‡]Carnegie Mellon University

[⊙]King Mongkut's University of Technology North Bangkok [◇]Boston University [▽]Microsoft Research India

Lectures on Virtual Memory

Challenges

- **Three examples** of the **challenges** in adapting conventional virtual memory frameworks for increasingly-diverse systems:
 - Requiring a **rigid page table structure**
 - High address **translation overhead** in virtual machines
 - **Inefficient** heterogeneous memory **management**



SAFARI 9:22 / 42:44

12



ETH ZÜRICH HAUPTGEBÄUDE

Computer Architecture - Lecture 12c: The Virtual Block Interface (ETH Zürich, Fall 2020)

726 views • Oct 31, 2020

16 0 SHARE SAVE ...



Onur Mutlu Lectures
16.5K subscribers

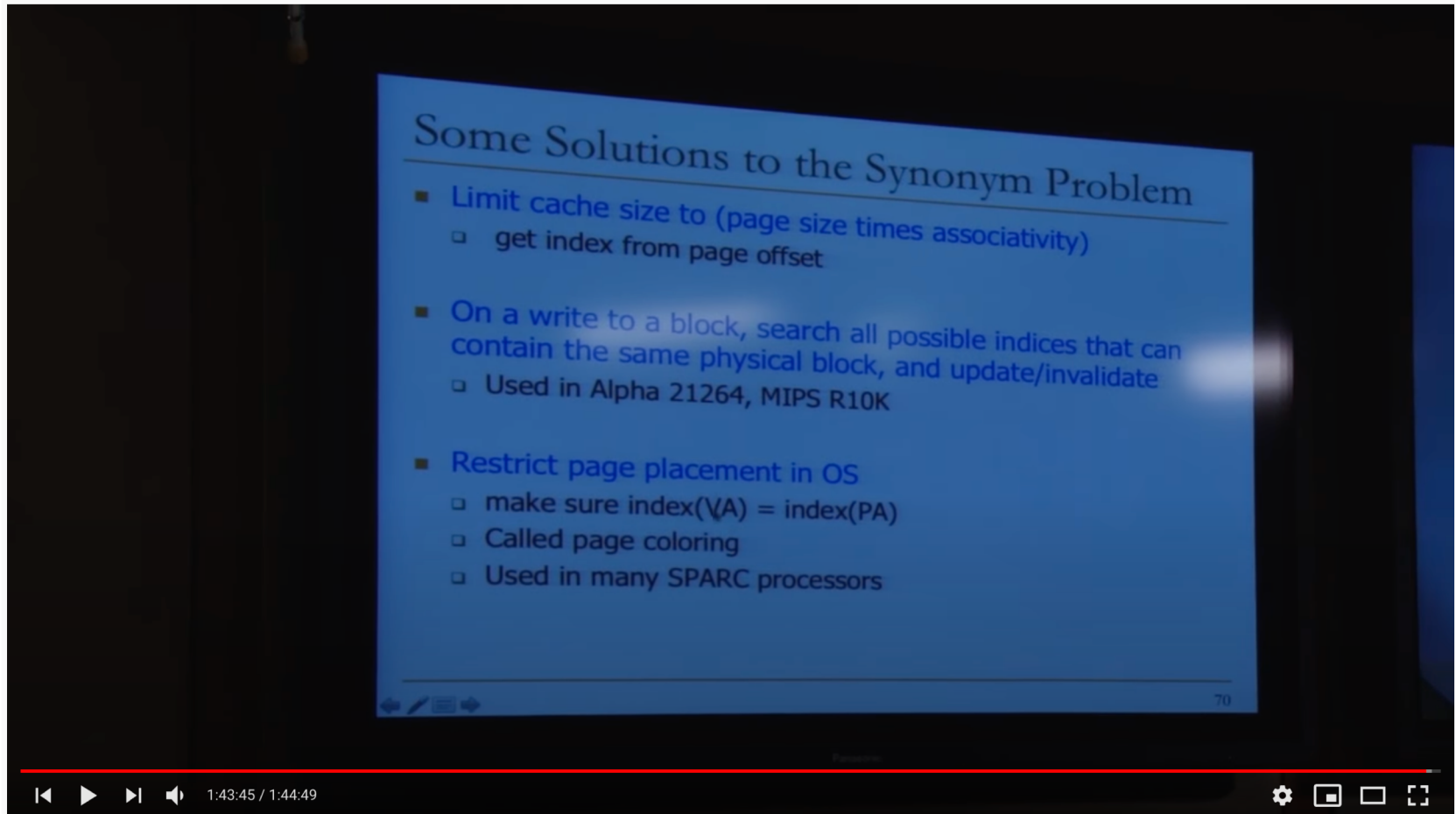
ANALYTICS

EDIT VIDEO

SAFARI

<https://www.youtube.com/watch?v=2RhGMpY18zw&list=PL5PHm2jkkXmi5Cxxl7b3JCL1TWybTDtKq&index=22>

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

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>

Digital Design & Computer Arch.

Lecture 26a: Virtual Memory II

Prof. Onur Mutlu

ETH Zürich

Spring 2021

4 June 2021

Backup Slides

More on Issues in Virtual Memory

Virtual Memory and Cache Interaction

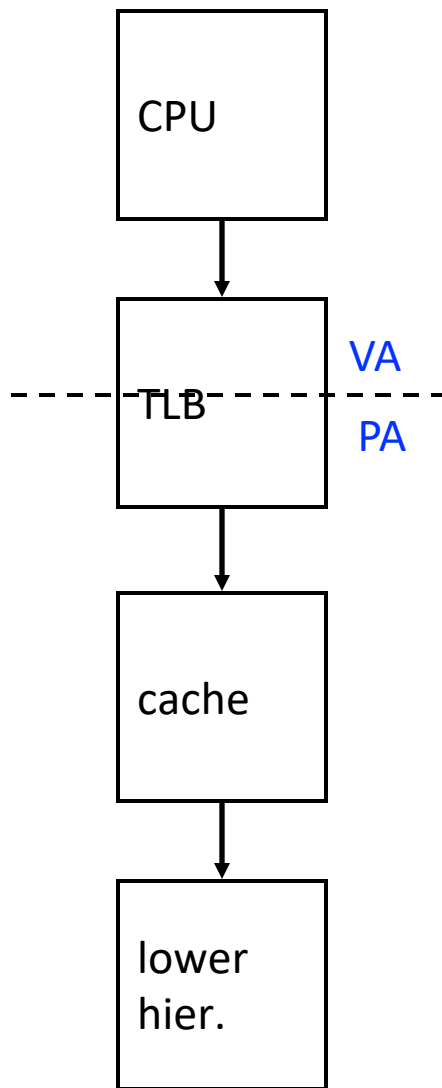
Address Translation and Caching

- 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

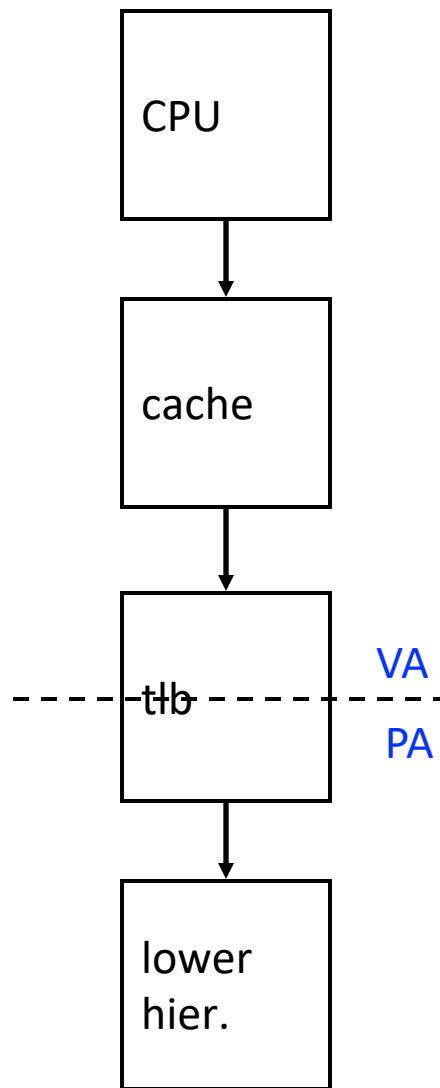
Homonyms and Synonyms

- **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?

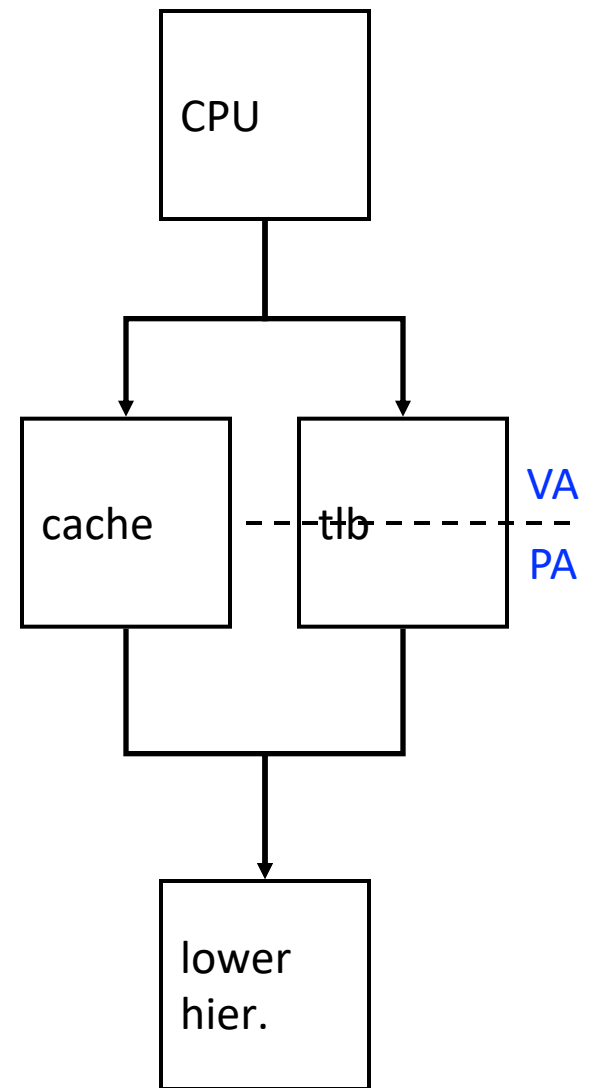
Cache-VM Interaction



physical cache

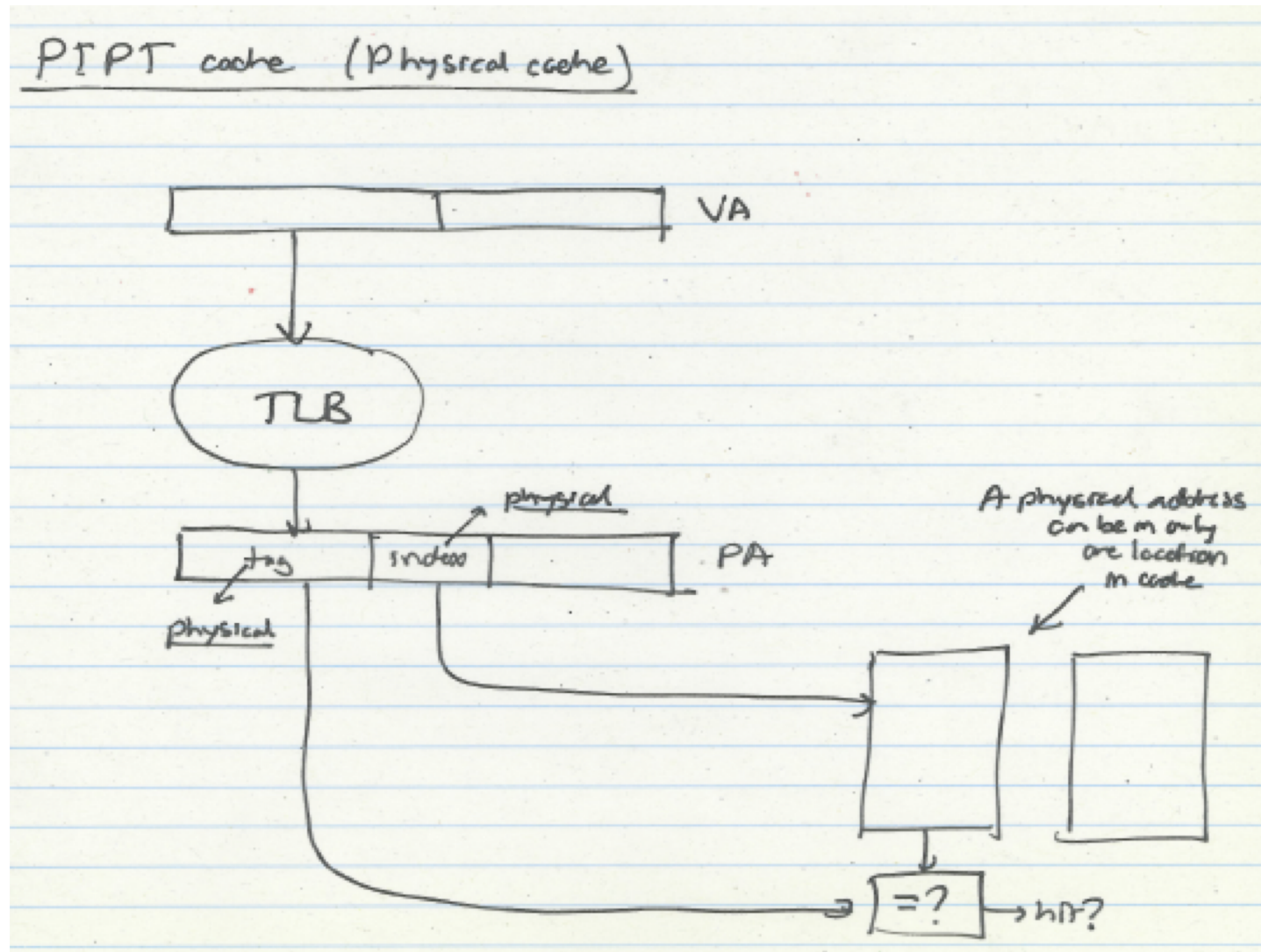


virtual (L1) cache

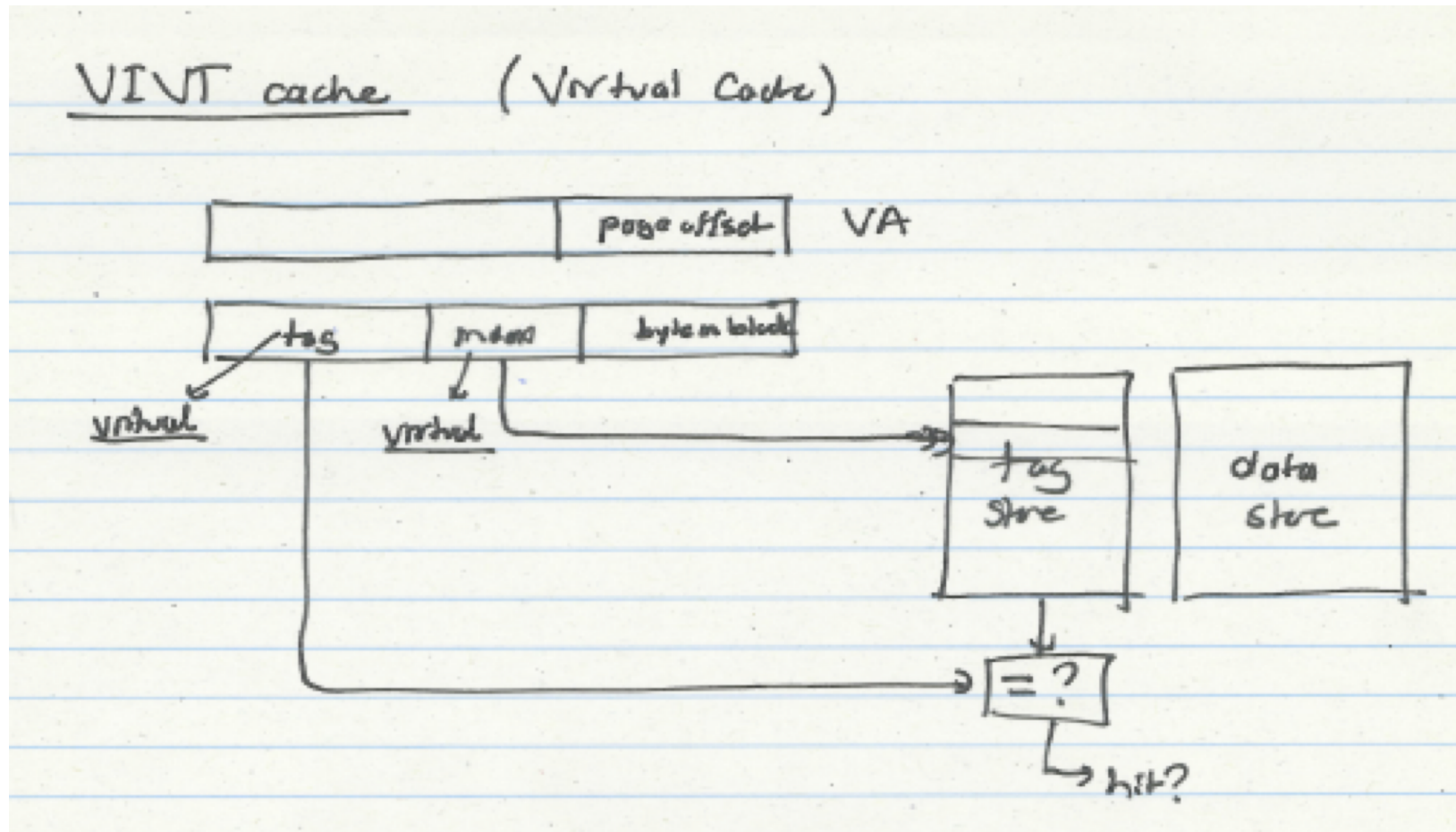


virtual-physical cache

Physical Cache

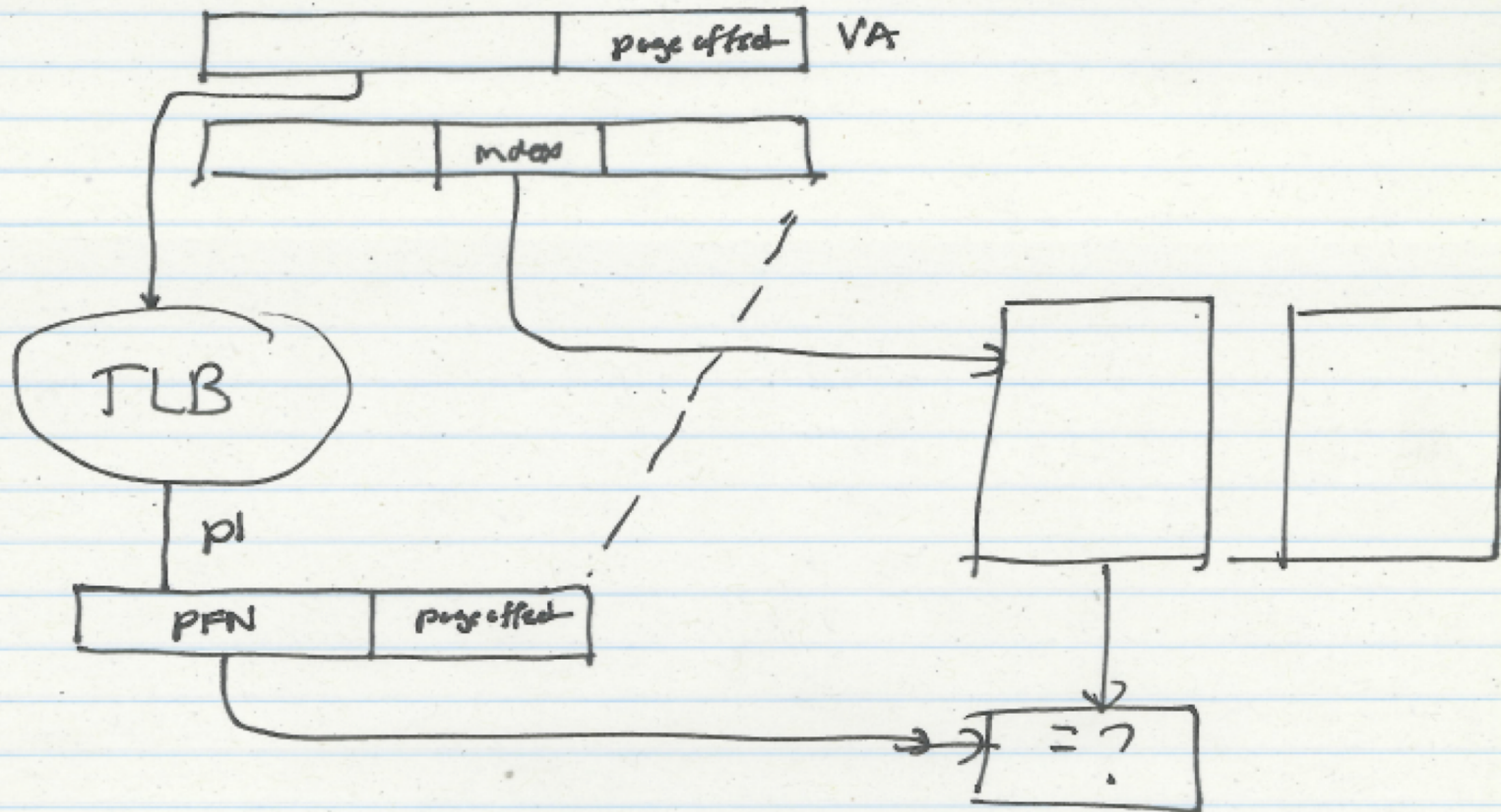


Virtual Cache



Virtual-Physical Cache

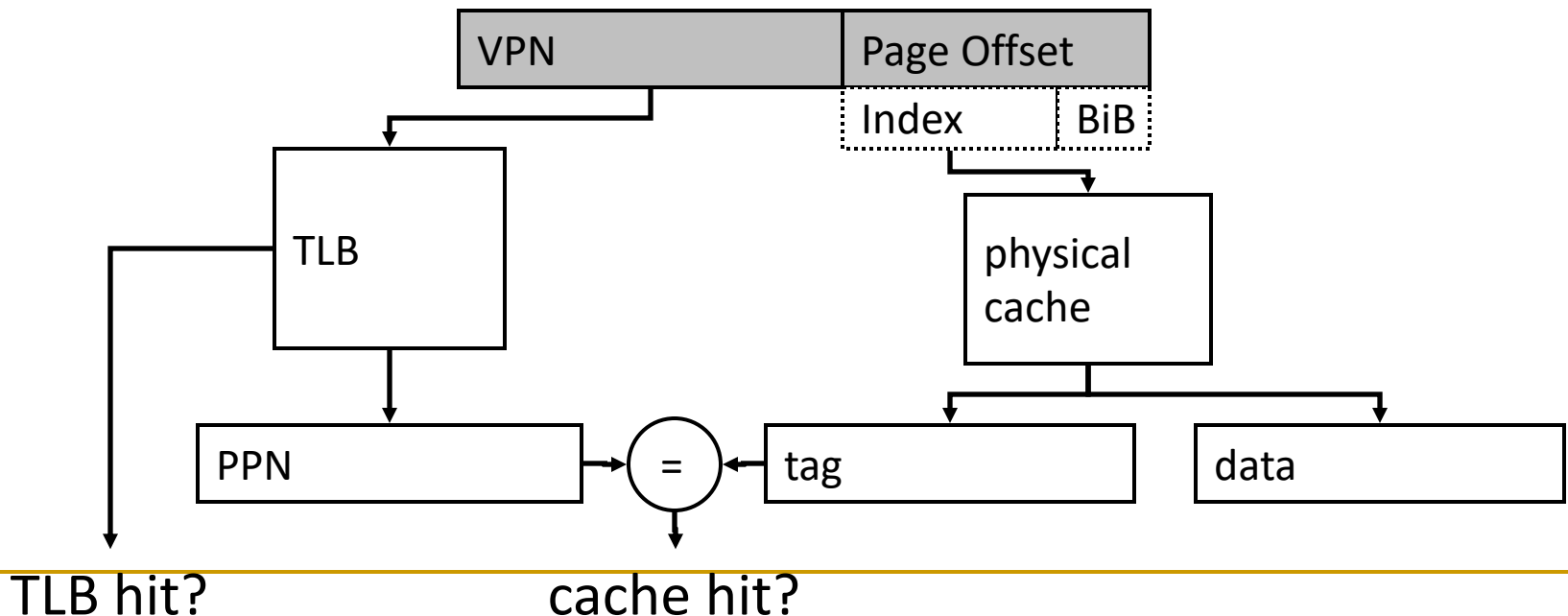
VIPT cache



Where can the same physical address be in the cache?

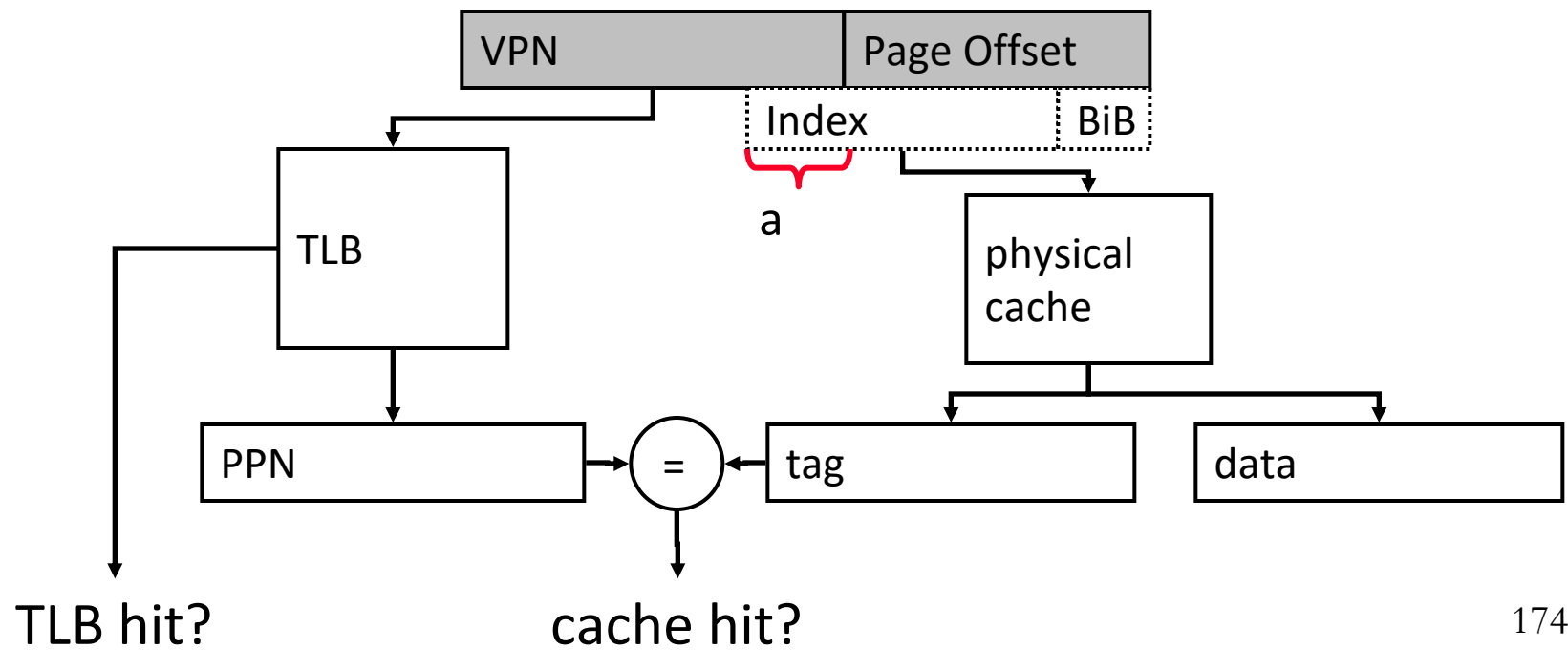
Virtually-Indexed Physically-Tagged

- 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>

An Exercise (I)

We have a byte-addressable toy computer that has a physical address space of 512 bytes. The computer uses a simple, one-level virtual memory system. The page table is always in physical memory. The page size is specified as 8 bytes and the virtual address space is 2 KB.

Part A.

i. (1 point)

How many bits of each virtual address is the virtual page number?

ii. (1 point)

How many bits of each physical address is the physical frame number?

We would like to add a 128-byte *write-through* cache to enhance the performance of this computer. However, we would like the cache access and address translation to be performed simultaneously. In other words, we would like to index our cache using a virtual address, but do the tag comparison using the physical addresses (virtually-indexed physically-tagged). The cache we would like to add is direct-mapped, and has a block size of 2 bytes. The replacement policy is LRU. Answer the following questions:

iii. (1 point)

How many bits of a virtual address are used to determine which byte in a block is accessed?

iv. (2 point)

How many bits of a virtual address are used to index into the cache? Which bits exactly?

v. (1 point)

How many bits of the virtual page number are used to index into the cache?

vi. (5 points)

What is the size of the tag store in bits? Show your work.

Part B.

Suppose we have two processes sharing our toy computer. These processes share some portion of the physical memory. Some of the virtual page-physical frame mappings of each process are given below:

PROCESS 0	
Virtual Page	Physical Frame
Page 0	Frame 0
Page 3	Frame 7
Page 7	Frame 1
Page 15	Frame 3

PROCESS 1	
Virtual Page	Physical Frame
Page 0	Frame 4
Page 1	Frame 5
Page 7	Frame 3
Page 11	Frame 2

vii. (2 points)

Give a complete physical address whose data can exist in two different locations in the cache.

viii. (3 points)

Give the indexes of those two different locations in the cache.

An Exercise (Concluded)

ix. (5 points)

We do not want the same physical address stored in two different locations in the 128-byte cache. We can prevent this by increasing the associativity of our virtually-indexed physically-tagged cache. What is the minimum associativity required?

x. (4 points)

Assume we would like to use a direct-mapped cache. Describe a solution that ensures that the same physical address is never stored in two different locations in the 128-byte cache.

A Potpourri of Issues

Trade-Offs in Page Size

■ Large page size (e.g., 1GB)

- Pro: Fewer PTEs required → Saves memory space
 - Pro: Fewer TLB misses → Improves performance

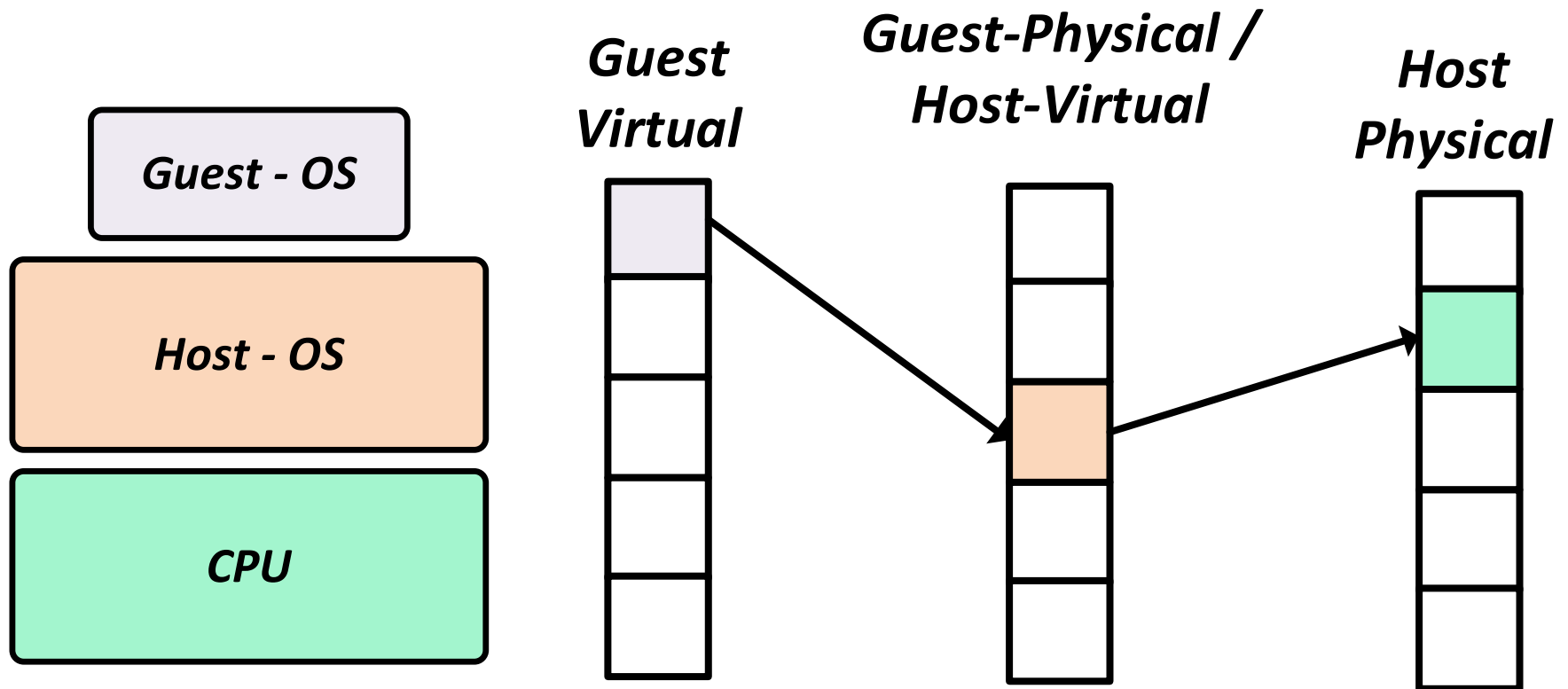
 - Con: Cannot have fine-grained permissions
 - Con: Large transfers to/from disk
 - Even when only 1KB is needed, 1GB must be transferred
 - Waste of bandwidth/energy
 - Reduces performance
 - Con: **Internal fragmentation**
 - Even when only 1KB is needed, 1GB must be allocated
 - Waste of space
 -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

Some System Software Tasks for VM

- Keeping track of which physical frames are free
- Allocating free physical frames to virtual pages
- Page replacement policy
 - When no physical frame is free, what should be removed?
- Sharing pages between processes
- Copy-on-write optimization
- Page-flip optimization

Virtual Memory in Virtualized Environments

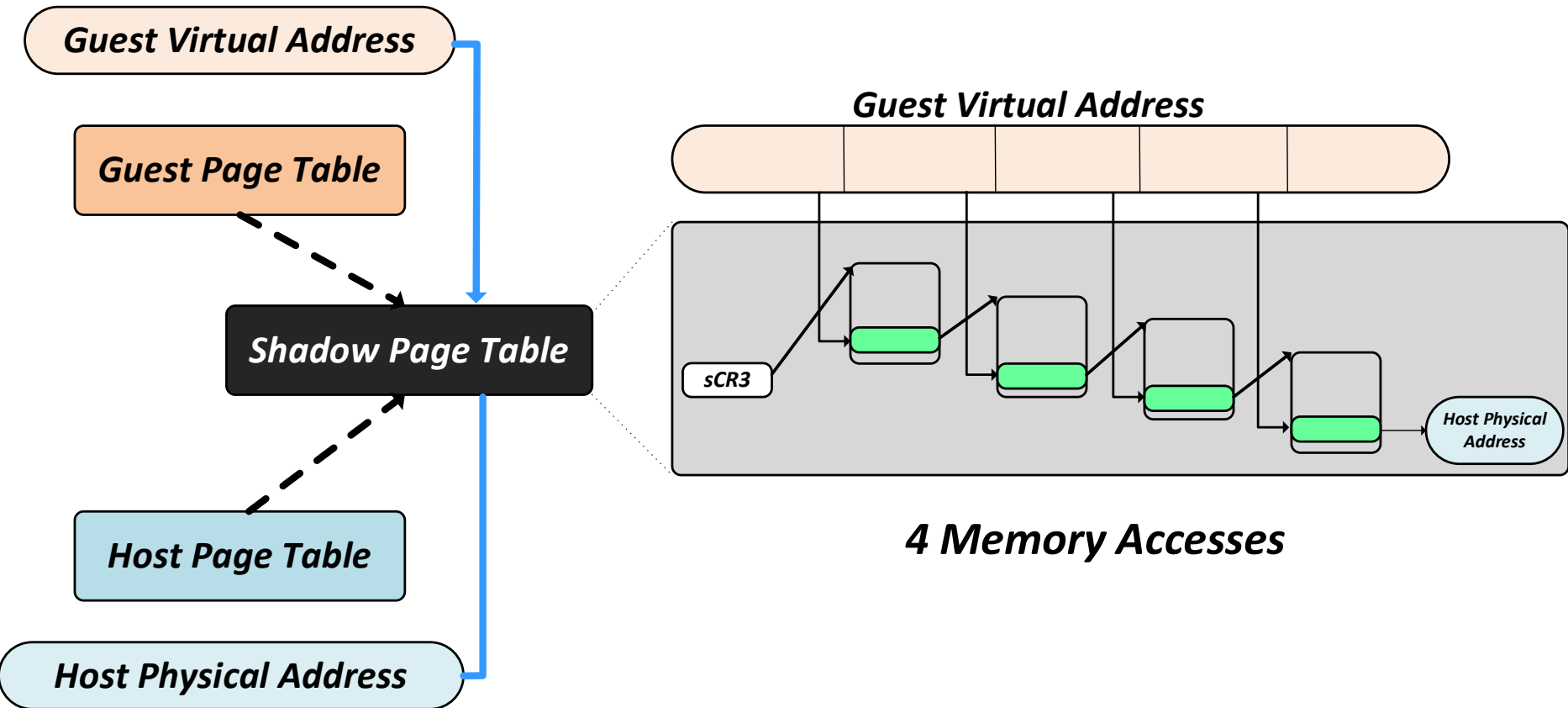
- Virtualized environments (e.g. Virtual Machines) need to have an additional level of address translation



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 handles every update to Guest and Host page tables

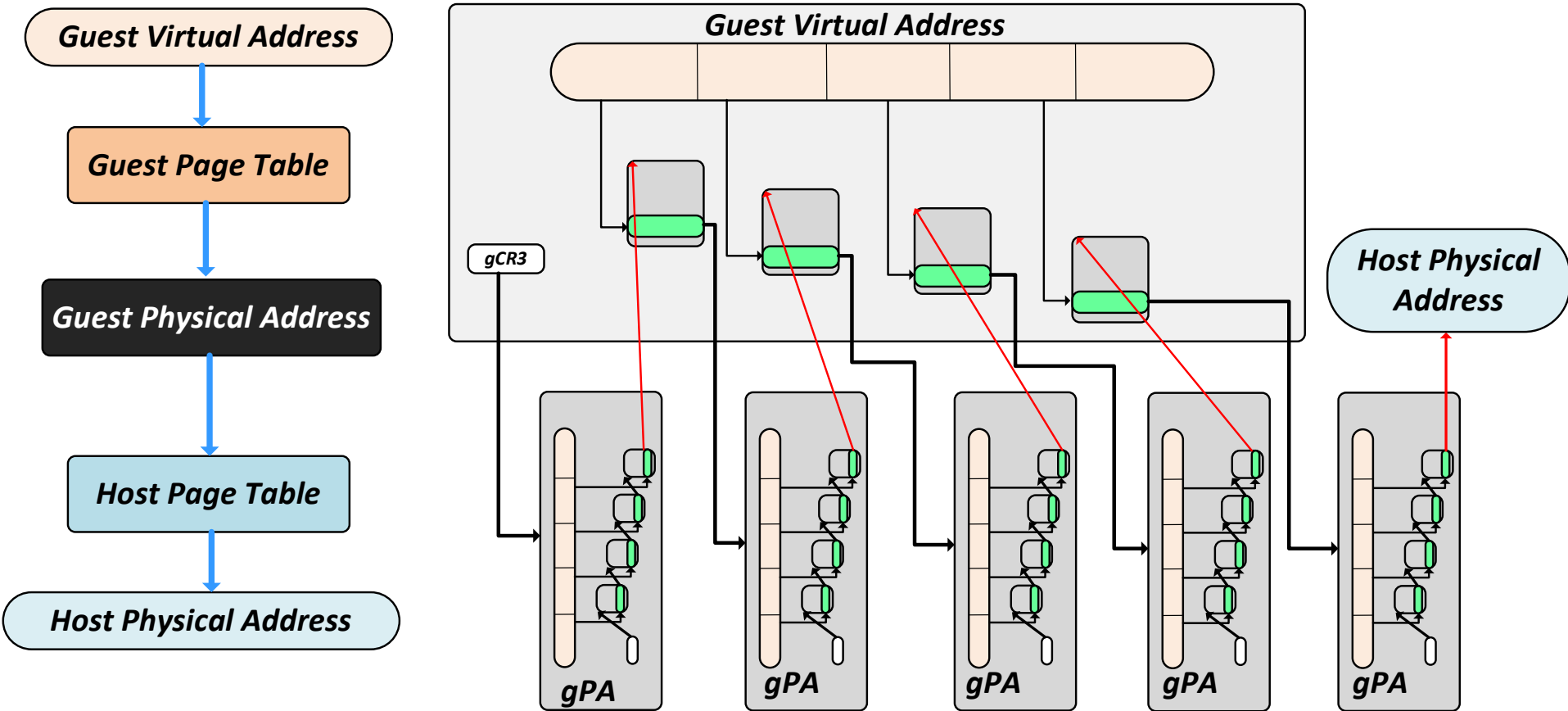
Shadow Paging



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