Digital Design & Computer Arch.

Lecture 26a: Virtual Memory II

Prof. Onur Mutlu

ETH Zürich

Spring 2022
3 June 2022

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

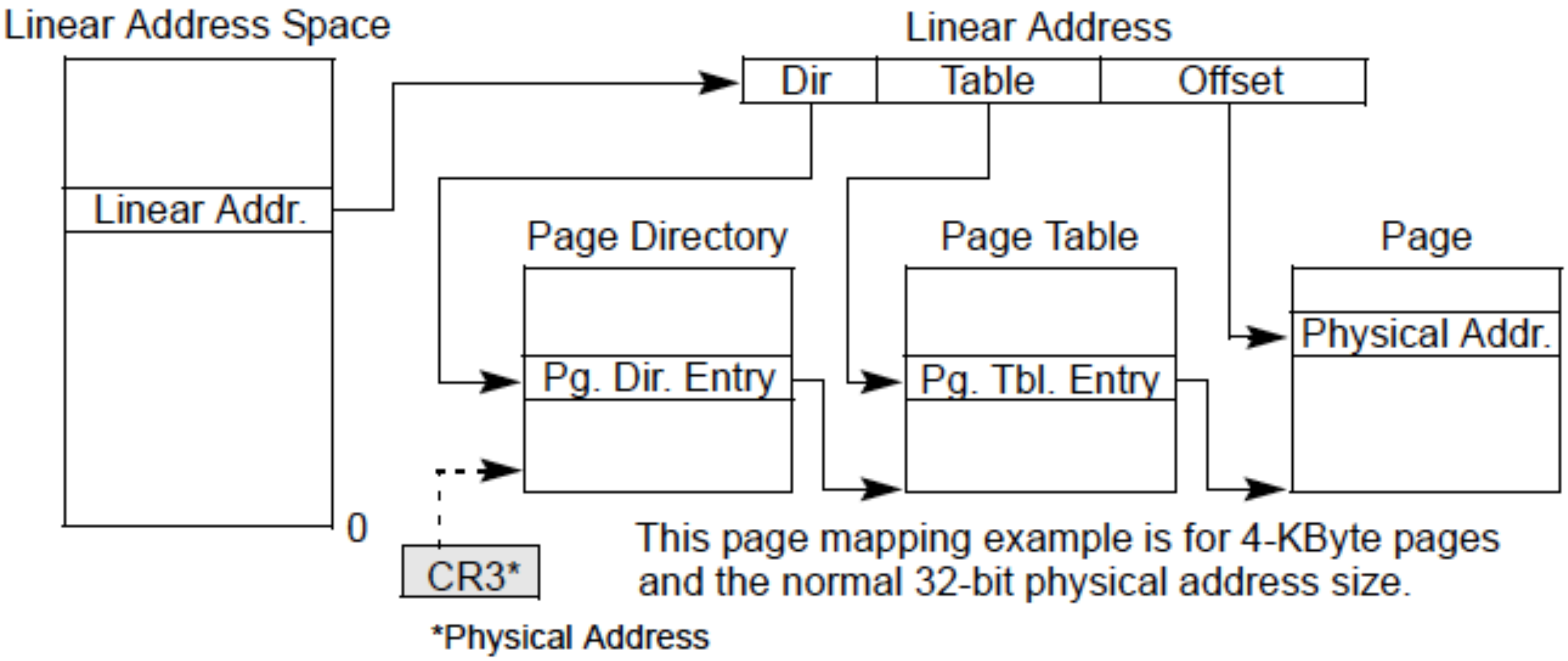
- How large is the page table?

- 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: Solution: Multi-Level Page Tables

Example from the x86 architecture



This page mapping example is for 4-KByte pages and the normal 32-bit physical address size.

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

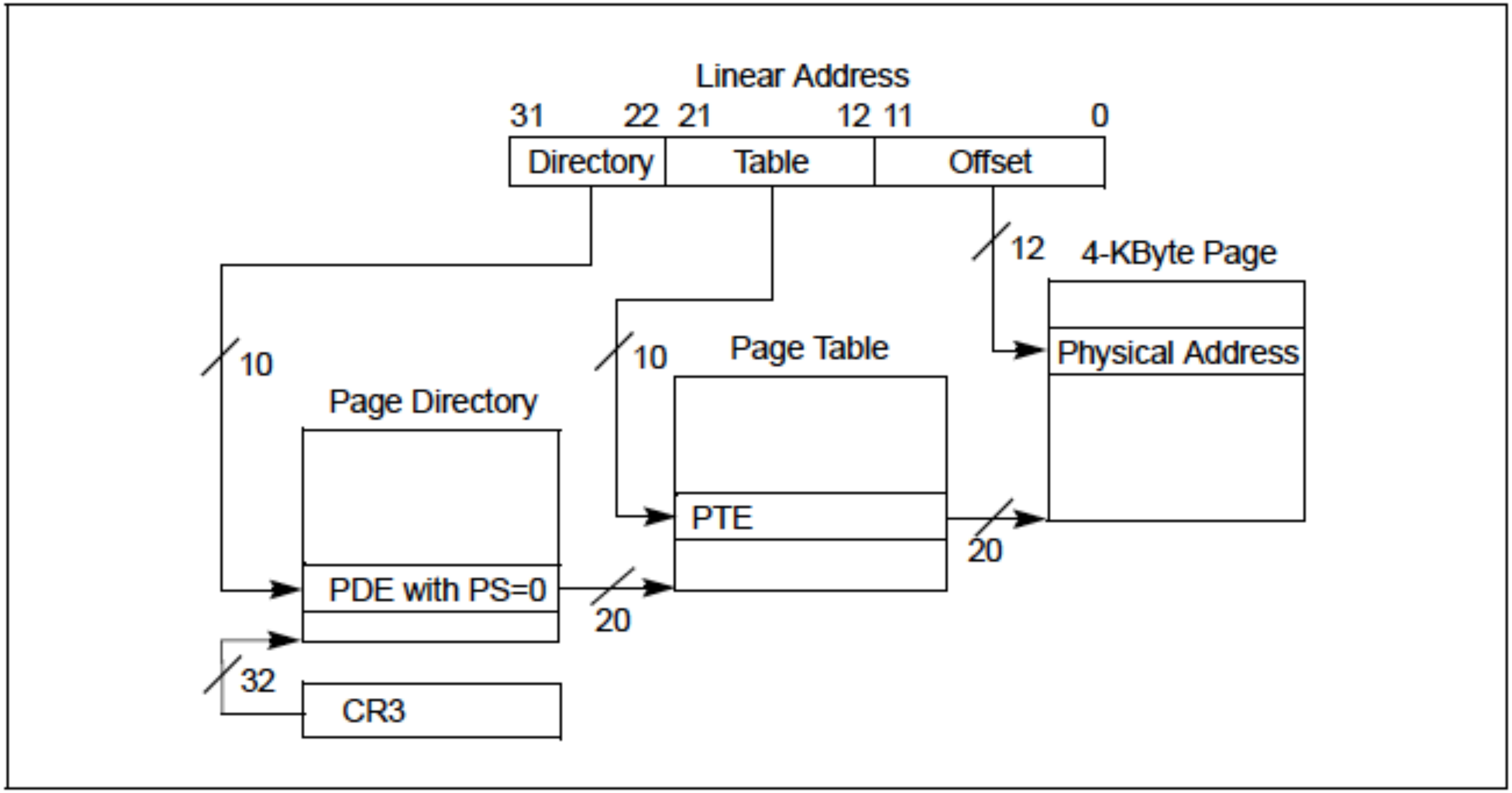


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

More on x86 Page Tables (II): Large Pages

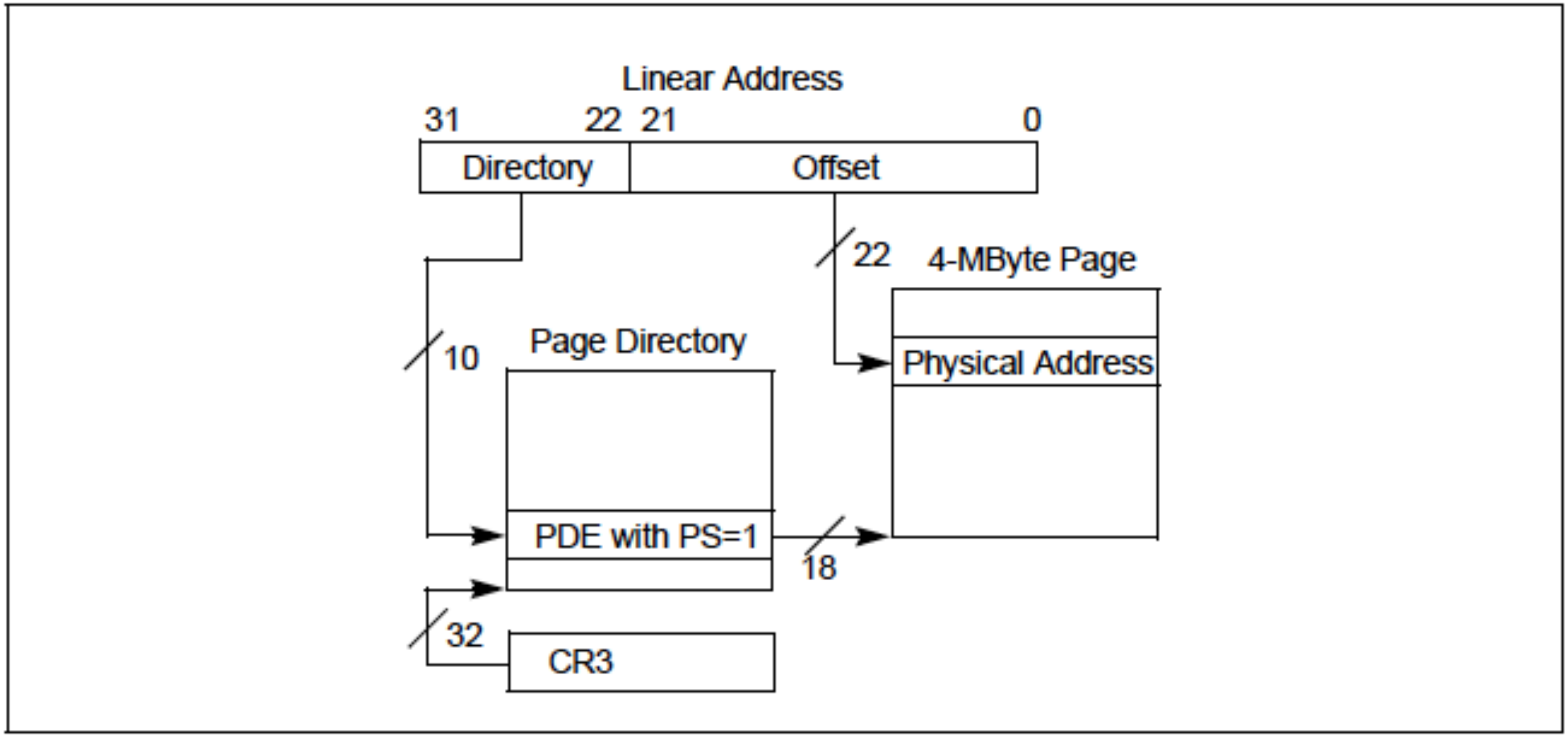
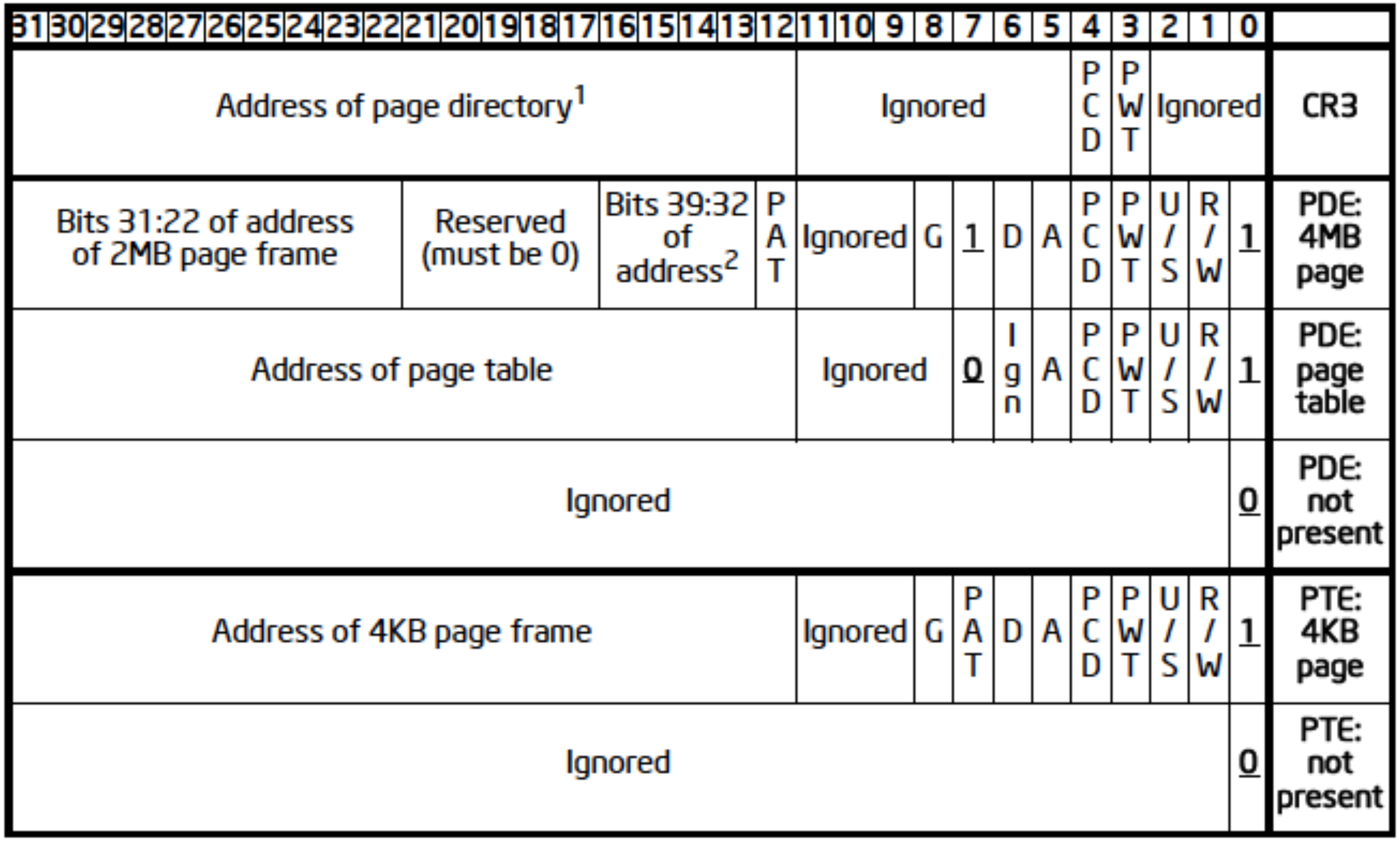


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

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.

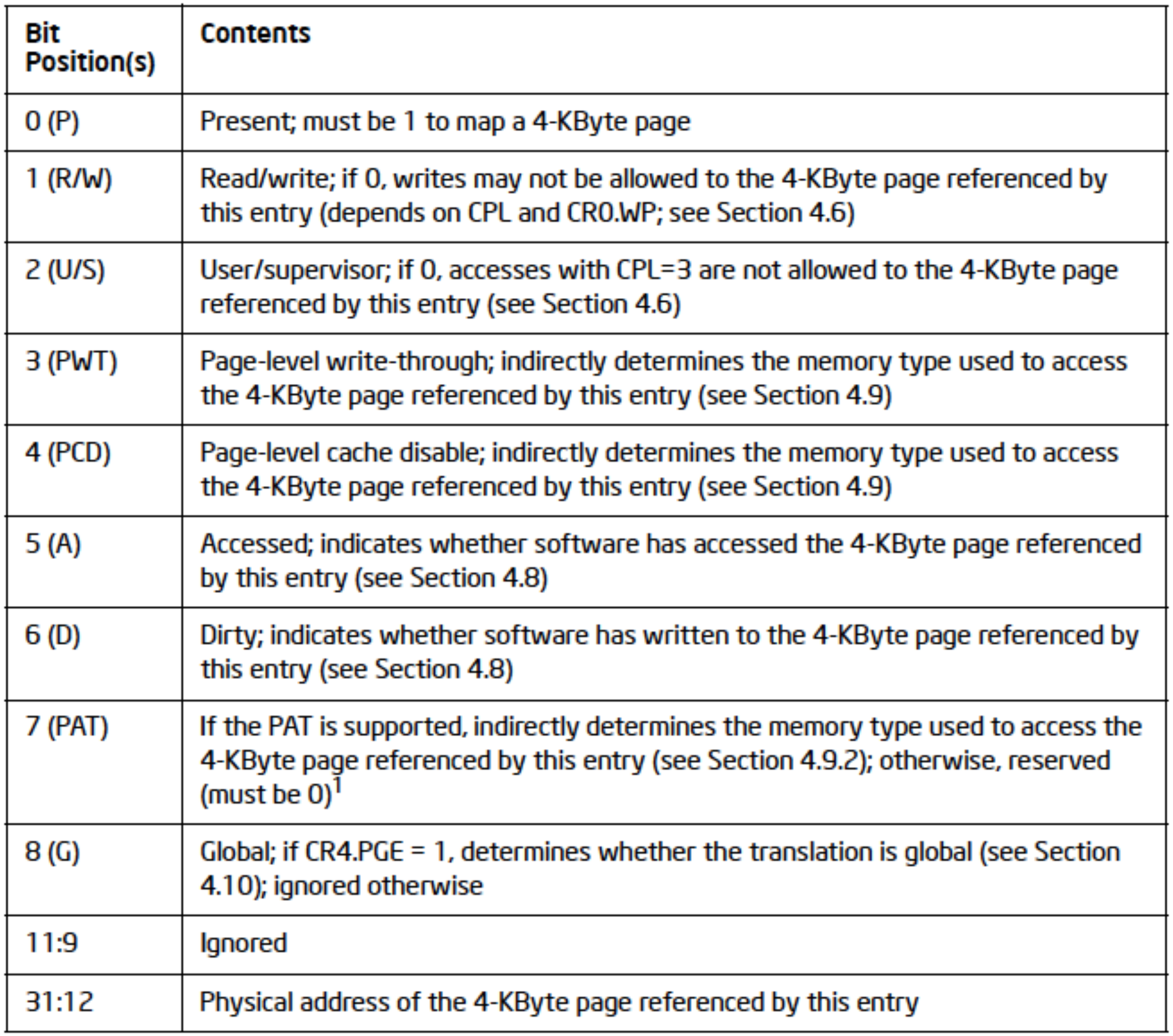


<table>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td colspan="20">Address of page directory[1]</td><td colspan="7">Ignored</td><td>PCD</td><td>PWT</td><td colspan="3">Ignored</td><td>CR3</td></tr>
<tr><td colspan="10">Bits 31:22 of address of 2MB page frame</td><td colspan="5">Reserved (must be 0)</td><td colspan="4">Bits 39:32 of address[2]</td><td>PAT</td><td colspan="3">Ignored</td><td>G</td><td>1</td><td>D</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PDE: 4MB page</td></tr>
<tr><td colspan="20">Address of page table</td><td colspan="4">Ignored</td><td>0</td><td>Ign</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PDE: page table</td></tr>
<tr><td colspan="31">Ignored</td><td>0</td><td>PDE: not present</td></tr>
<tr><td colspan="20">Address of 4KB page frame</td><td colspan="3">Ignored</td><td>G</td><td>PAT</td><td>D</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PTE: 4KB page</td></tr>
<tr><td colspan="31">Ignored</td><td>0</td><td>PTE: not present</td></tr>
</table>

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)

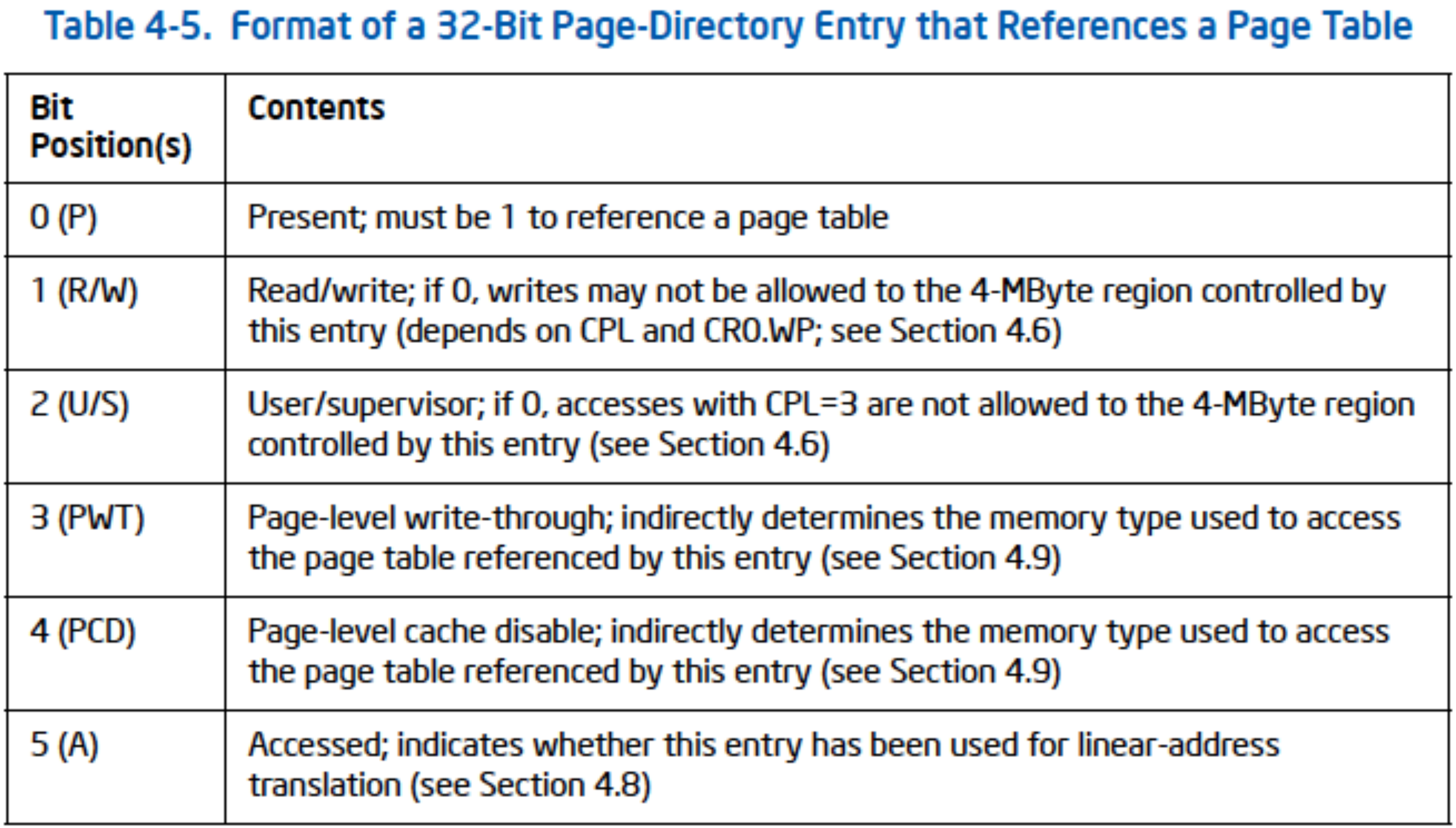
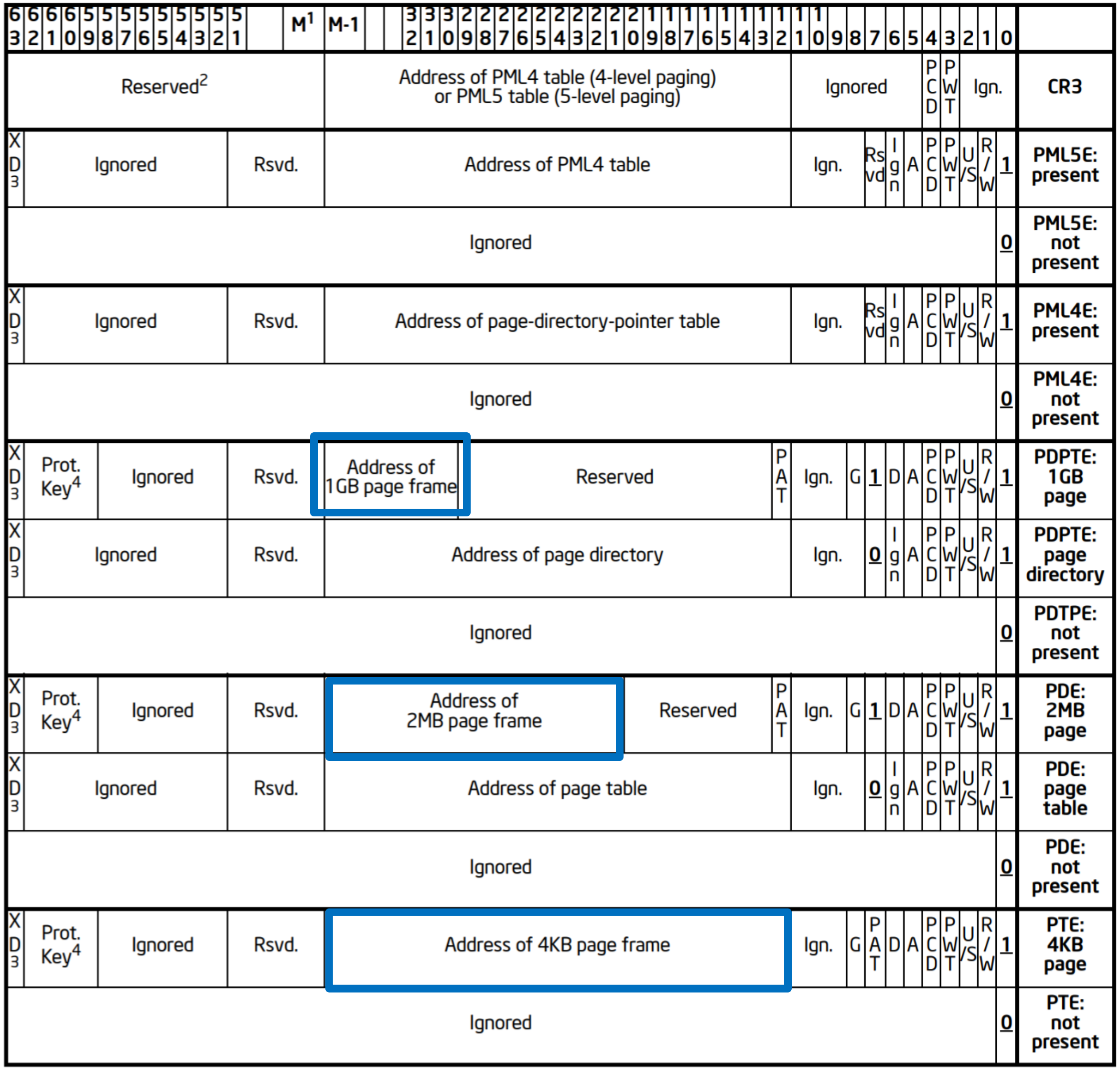


Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

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



63	62 … 59	58 … 52	51 … M[1]	M-1 … 30	29 … 21	20 … 13	12	11 … 9	8	7	6	5	4	3	2	1	0	
Reserved[2]				Address of PML4 table (4-level paging) or PML5 table (5-level paging)				Ignored					PCD	PWT	Ign.			**CR3**
XD[3]	Ignored		Rsvd.	Address of PML4 table				Ign.		Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	**PML5E: present**
Ignored																	0	**PML5E: not present**
XD[3]	Ignored		Rsvd.	Address of page-directory-pointer table				Ign.		Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	**PML4E: present**
Ignored																	0	**PML4E: not present**
XD[3]	Prot. Key[4]	Ignored	Rsvd.	Address of 1GB page frame	Reserved		PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	**PDPTE: 1GB page**
XD[3]	Ignored		Rsvd.	Address of page directory				Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	**PDPTE: page directory**
Ignored																	0	**PDTPE: not present**
XD[3]	Prot. Key[4]	Ignored	Rsvd.	Address of 2MB page frame		Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	**PDE: 2MB page**
XD[3]	Ignored		Rsvd.	Address of page table				Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**
Ignored																	0	**PDE: not present**
XD[3]	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame				Ign.	G	PAT	D	A	PCD	PWT	U/S	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

Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide, Part 1

X86-64 Page Table: Accessing 4KB pages

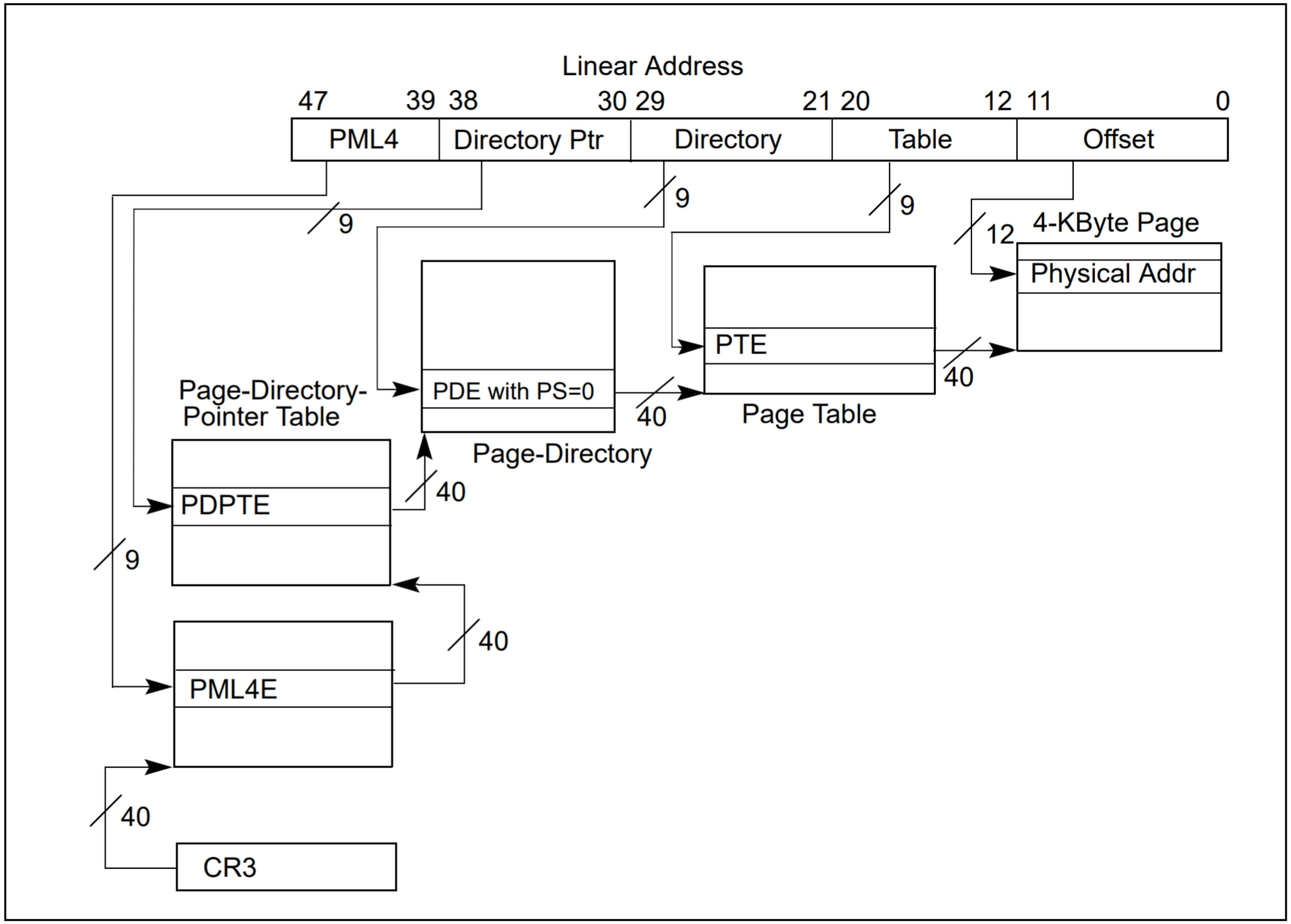


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1

X86-64 Page Table: Accessing 2MB pages

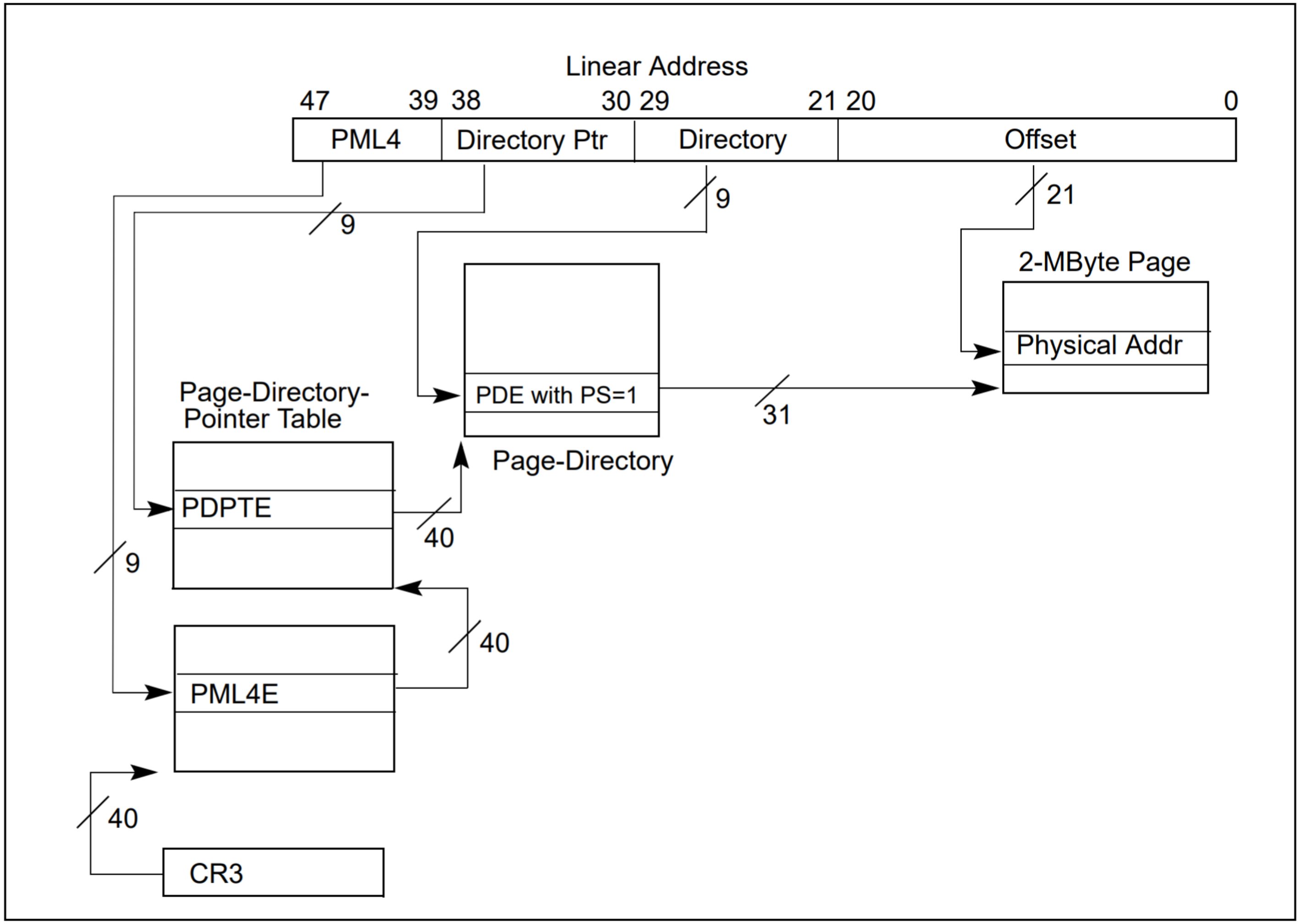


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1

X86-64 Page Table: Accessing 1GB pages

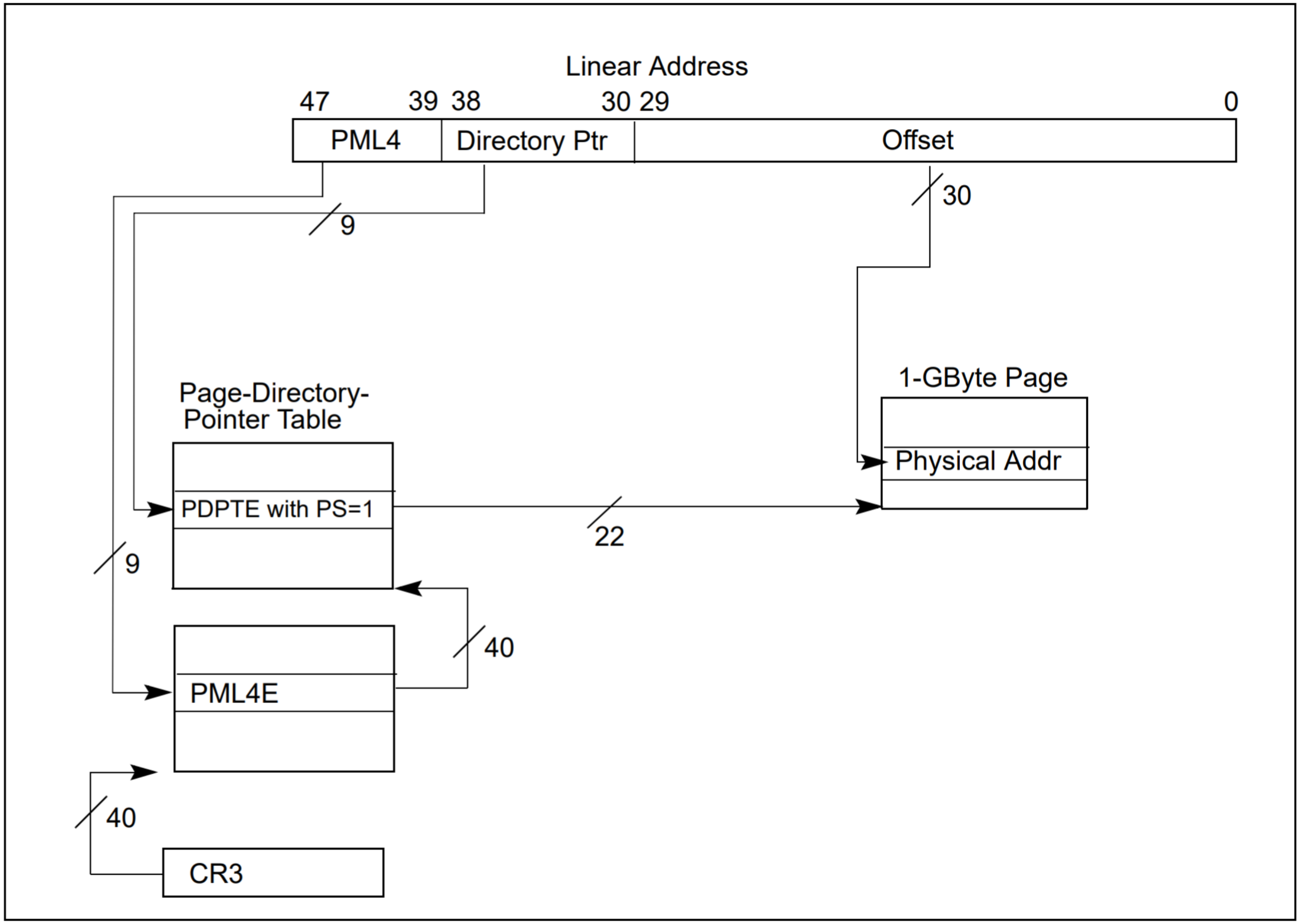


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1

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 Page Table Entries, i.e., recently used Virtual-to-Physical translations
  - Reduces the number of memory accesses required for *most* instruction fetches and loads/stores to only one TLB access

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?

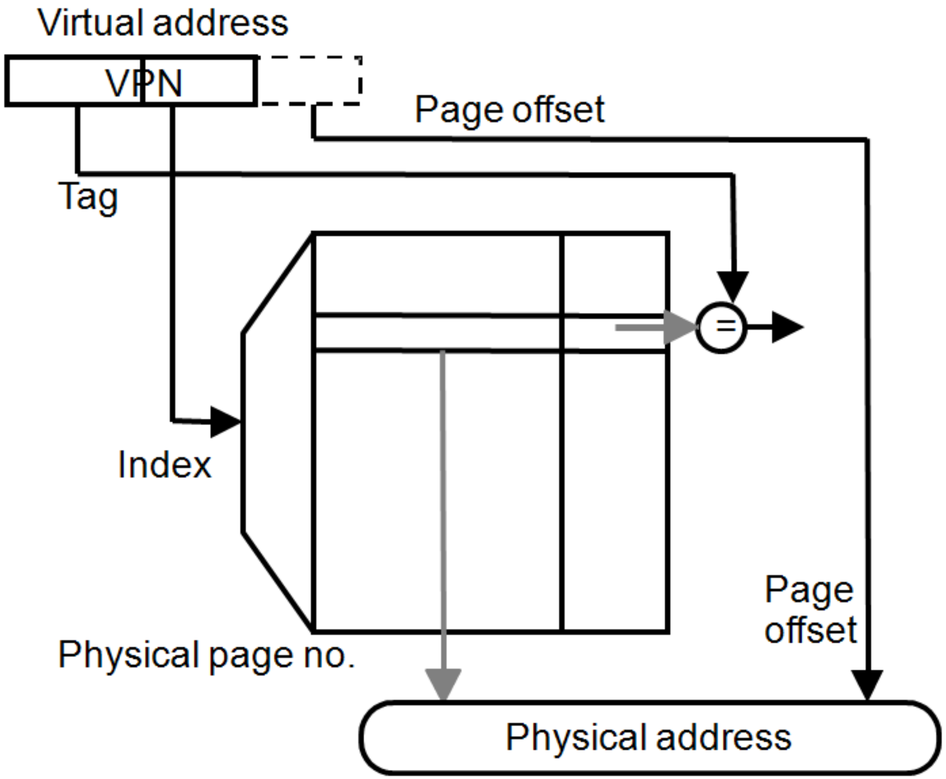
- What should be done on a page fault?
  - What virtual page to replace from physical memory?
  - Who handles the page fault? HW vs. SW?

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 execute and help tolerate latency

  + 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 oganization

  + 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

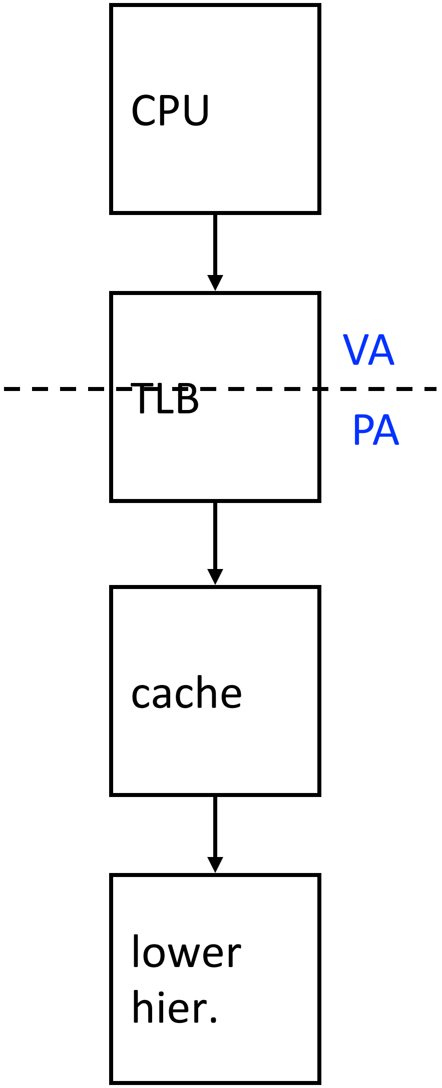
- Homonym: Same VA can map to two different PAs
  - Why?
    - VA is in different processes

- 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, …

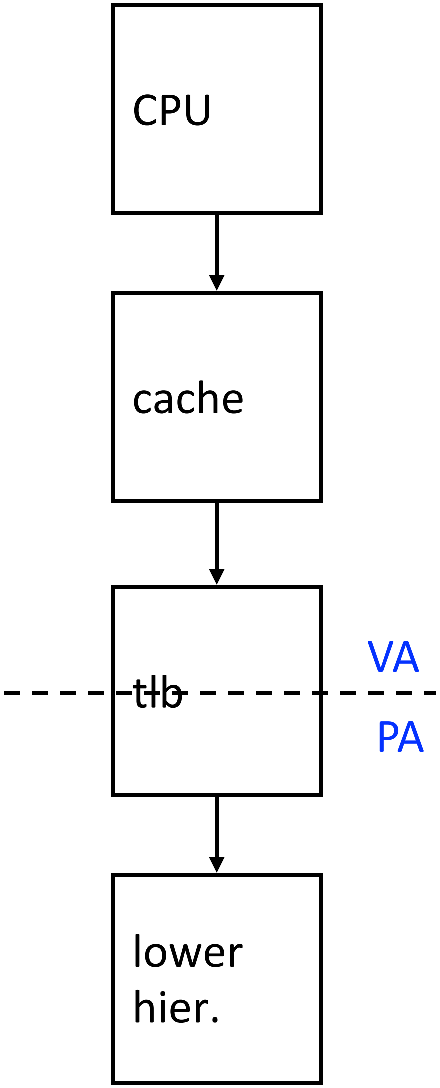
- Do homonyms and synonyms create problems when we have a cache?
  - Is the cache virtually or physically addressed?

Cache-VM Interaction

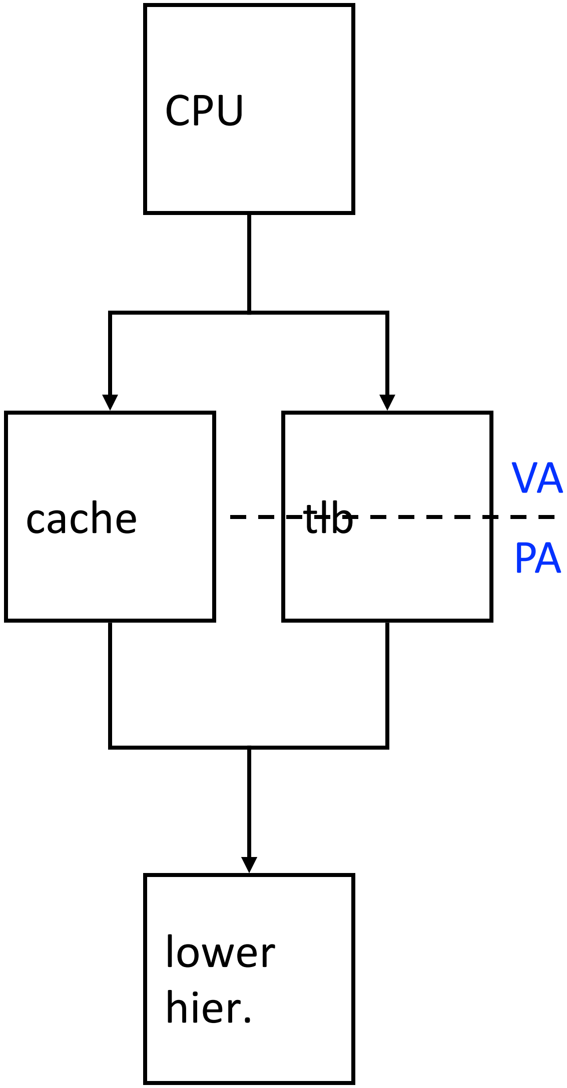
See backup slides for more



physical cache



virtual (L1) cache

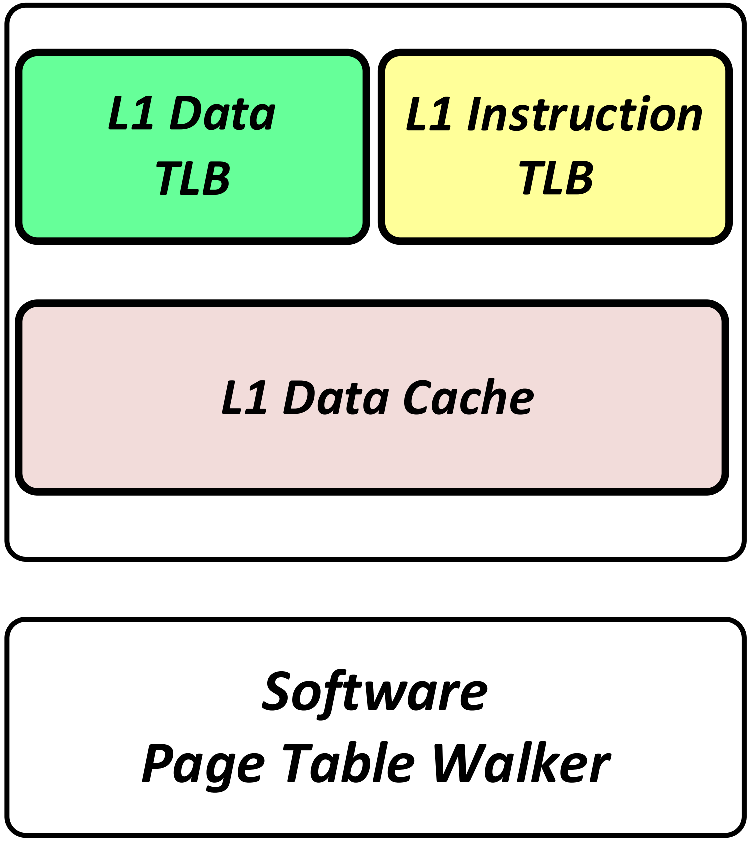


virtual-physical cache

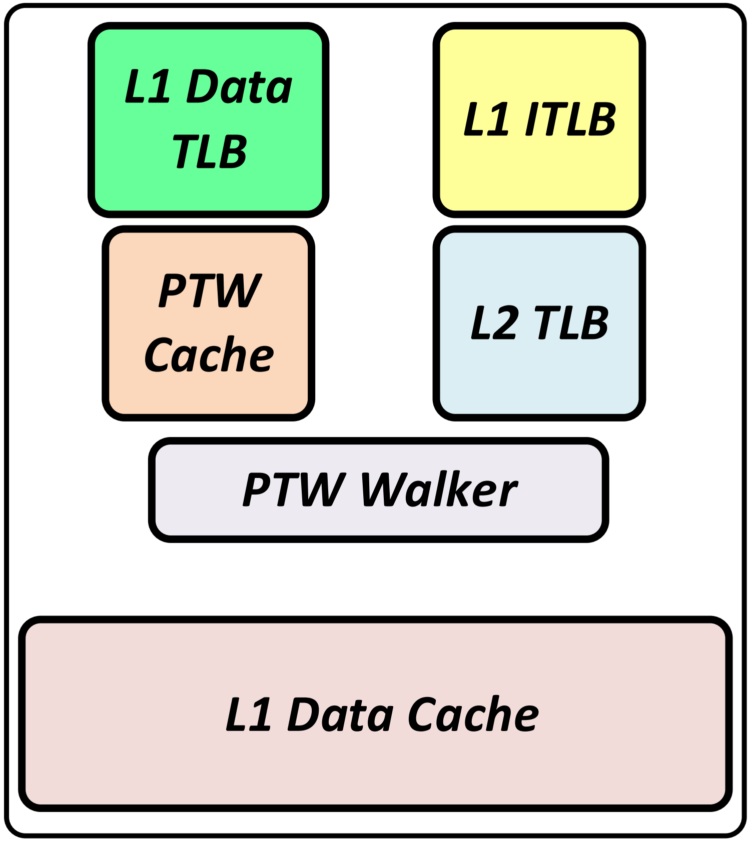
A Modern Example Virtual Memory System

Evolution of Address Translation

Simple Address Translation



Modern Address Translation

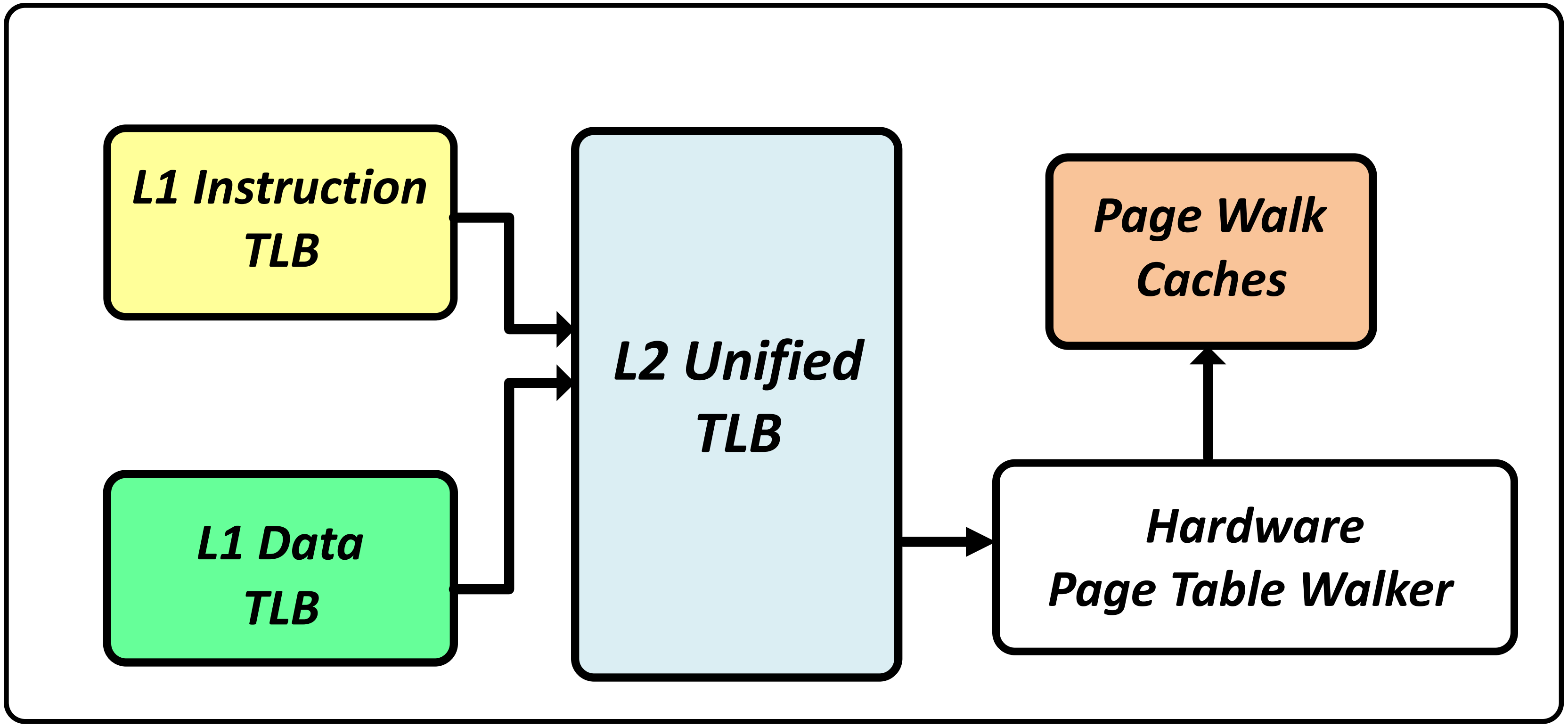


Memory Management Unit

- The **Memory Management Unit (MMU) is** responsible for resolving address translation requests
  - One MMU per core (usually)

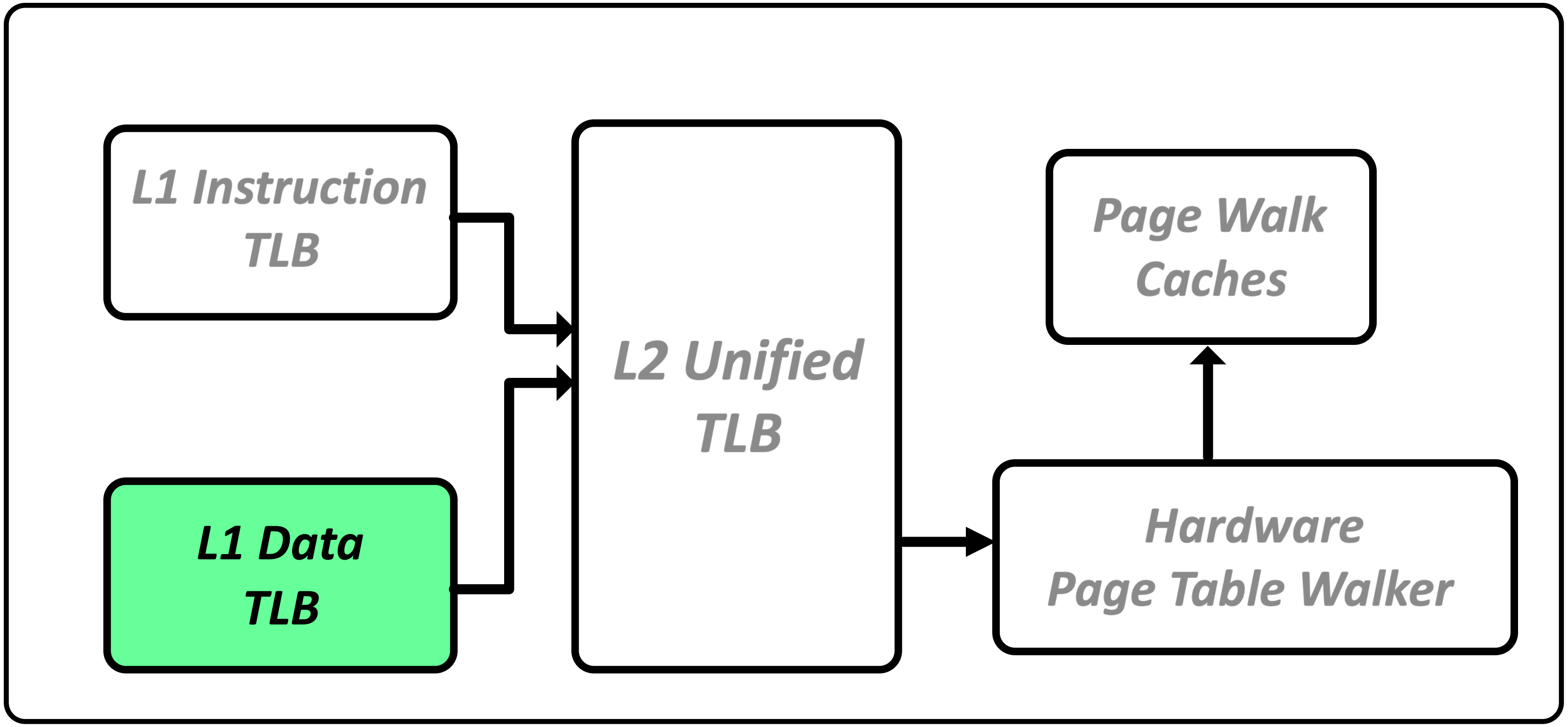
- MMU typically has three key components:
  - **Translation Lookaside Buffers** that cache recently-used virtual-to-physical translations (PTEs)
  - **Page Table Walk Caches** that offer fast access to the intermediate levels of a multi-level page table
  - **Hardware Page Table Walker** that sequentially accesses the different levels of the Page Table to fetch the required PTE

Intel Skylake: MMU



https://www.7-cpu.com/cpu/Skylake.html

Intel Skylake: L1 Data TLB



L1 Instruction TLB

L1 Data TLB

L2 Unified TLB

Page Walk Caches

Hardware Page Table Walker

Intel Skylake: L1 Data TLB

- Separate L1 Data TLB structures for 4KB, 2MB, and 1GB pages

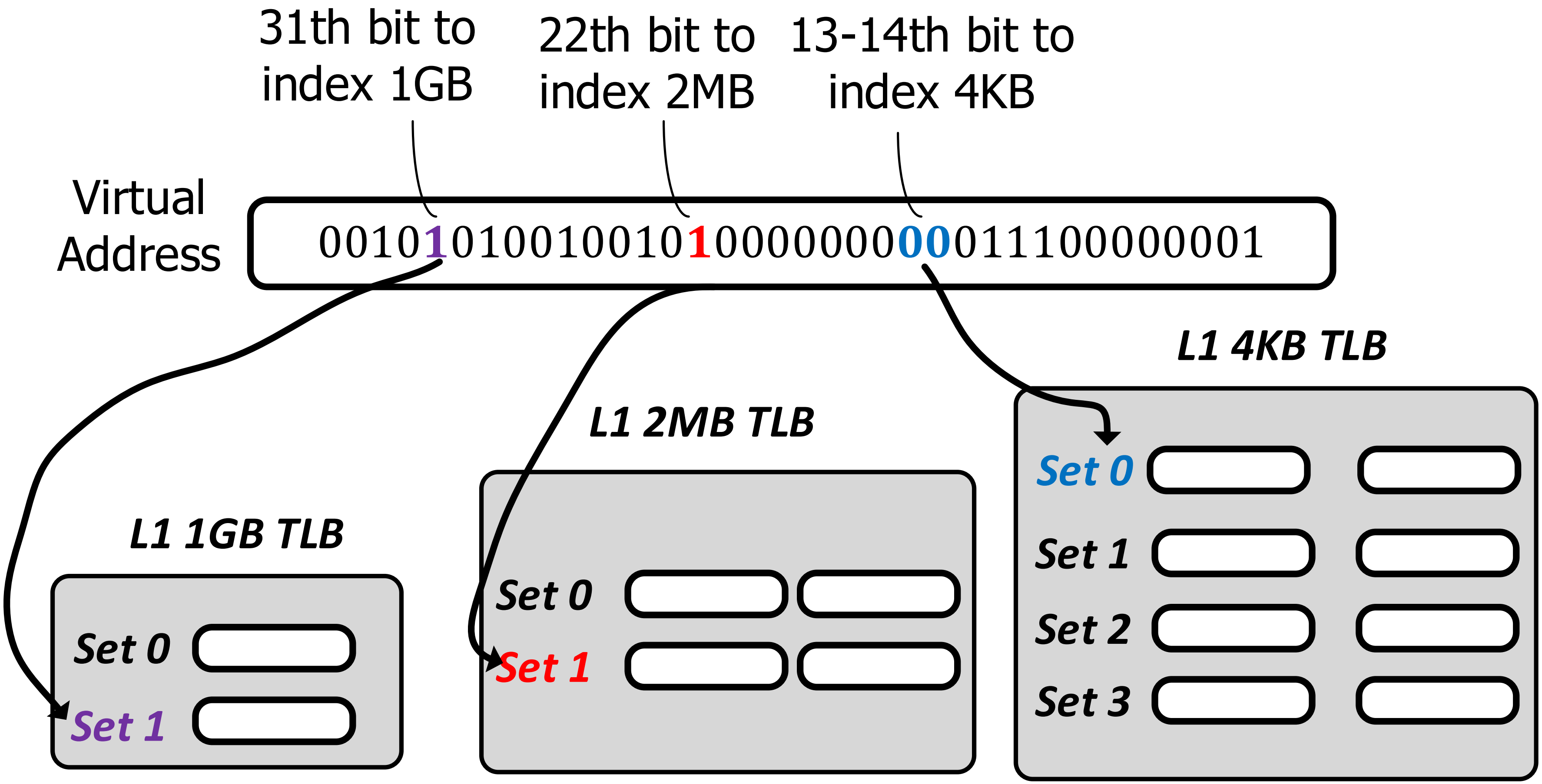
- L1 DTLB
  - **4KB**: 64-entry, 4-way, 1 cycle access, 9 cycle miss
  - **2MB**: 32-entry, 4-way, 1 cycle access, 9 cycle miss
  - **1GB**: 4 entry, fully-associative

- 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

https://www.7-cpu.com/cpu/Skylake.html

L1 Data TLB: Parallel Lookup Example



31th bit to index 1GB

22th bit to index 2MB

13-14th bit to index 4KB

Virtual Address

00101**1**010010010**1**0000000**00**011100000001

L1 1GB TLB

Set 0

Set 1

L1 2MB TLB

Set 0

Set 1

L1 4KB TLB

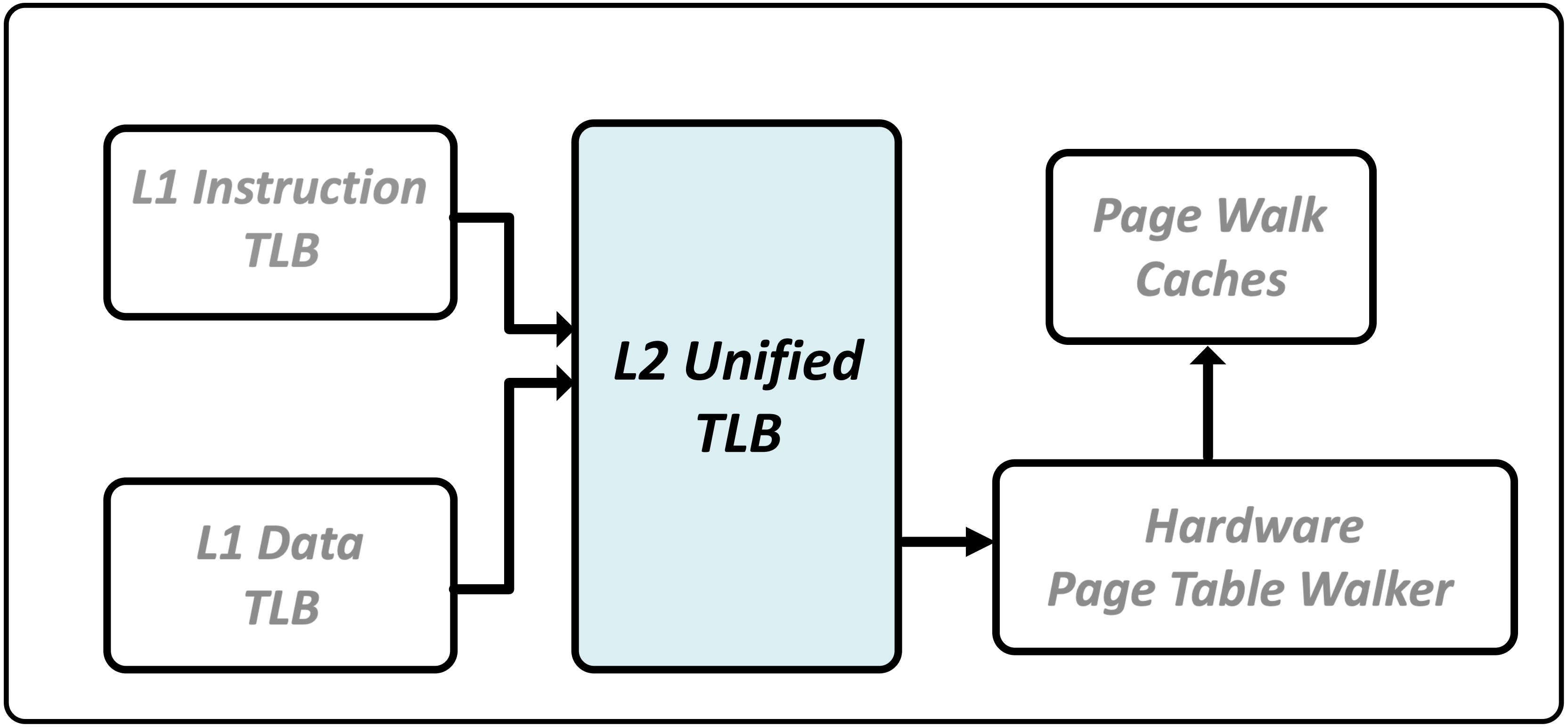
Set 0

Set 1

Set 2

Set 3

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)

https://www.7-cpu.com/cpu/Skylake.html

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.

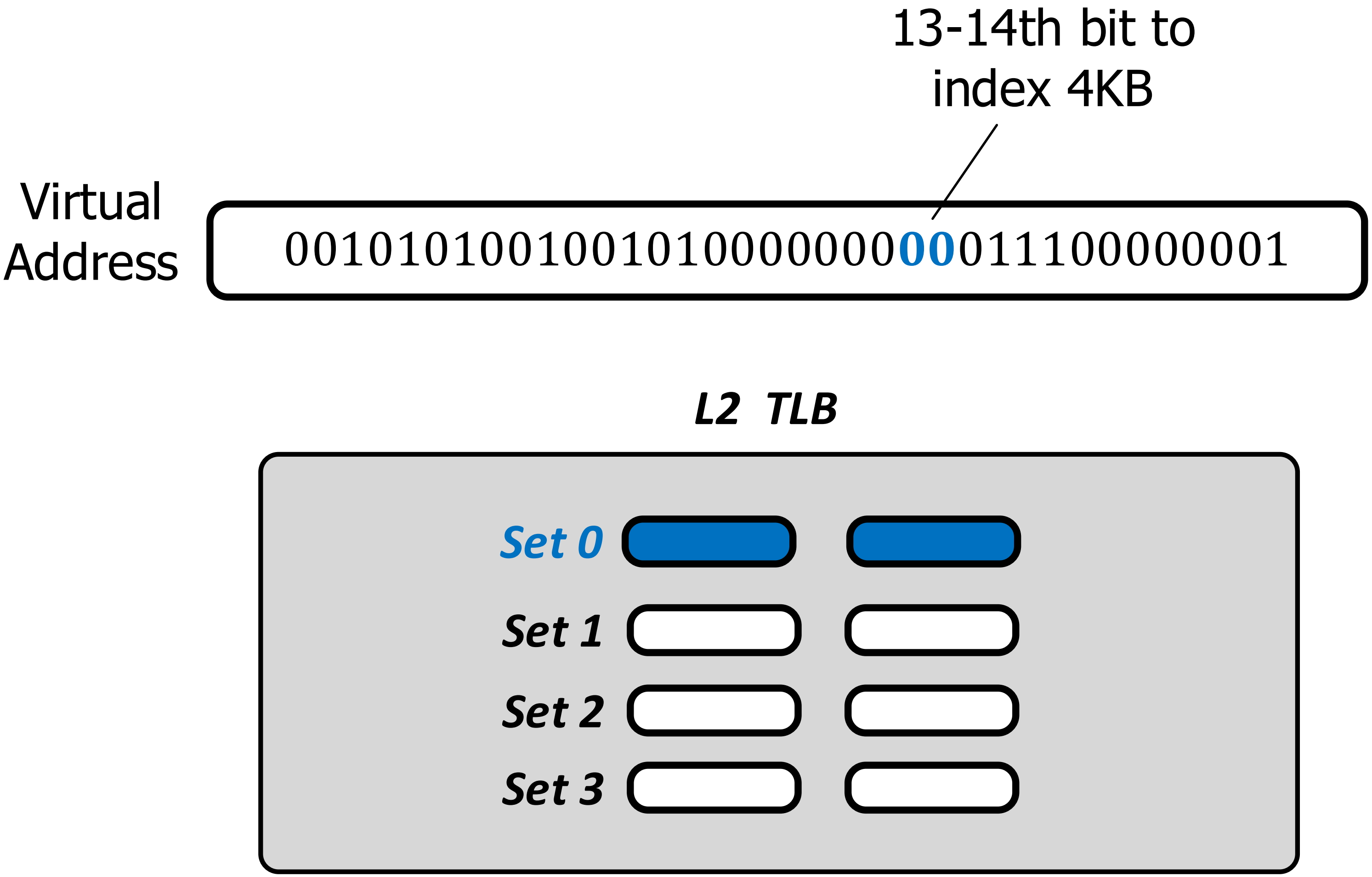
- Step 2: Assume the page size is 2MB, **re-calculate** the index and access the L2 TLB.
  - If the tag matches, it is a hit. If the tag does not match, it is an L2 TLB miss.

- **General algorithm**:

  Re-calculate index and probe TLB for all remaining page sizes

Similar to "associativity in time" (also called pseudo-associativity)

Step 1: Calculate Index for 4KB

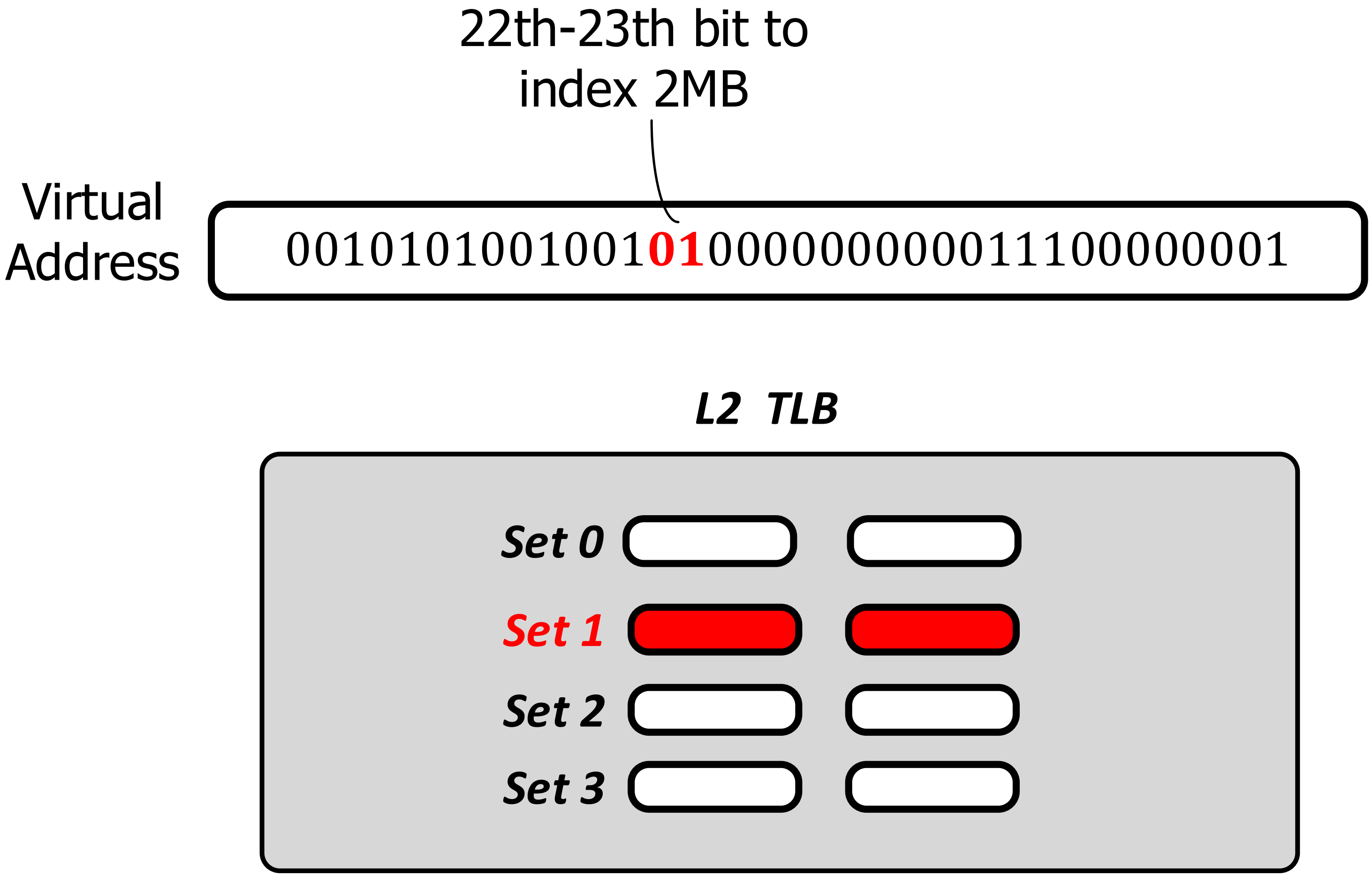


13-14th bit to index 4KB

Virtual Address

0010101001001010000000**00**011100000001

Step 2: Re-calculate Index for 2MB



22th-23th bit to index 2MB

Virtual Address

001010100100101000000000011100000001

L2 TLB

Set 0

Set 1

Set 2

Set 3

L2 TLB: N-Step Index Re-Calculation

- Pros:

  + Simple and practical implementation

- Cons:

  - Varying L2 TLB hit latency (faster for 4KB, slower for 2MB)

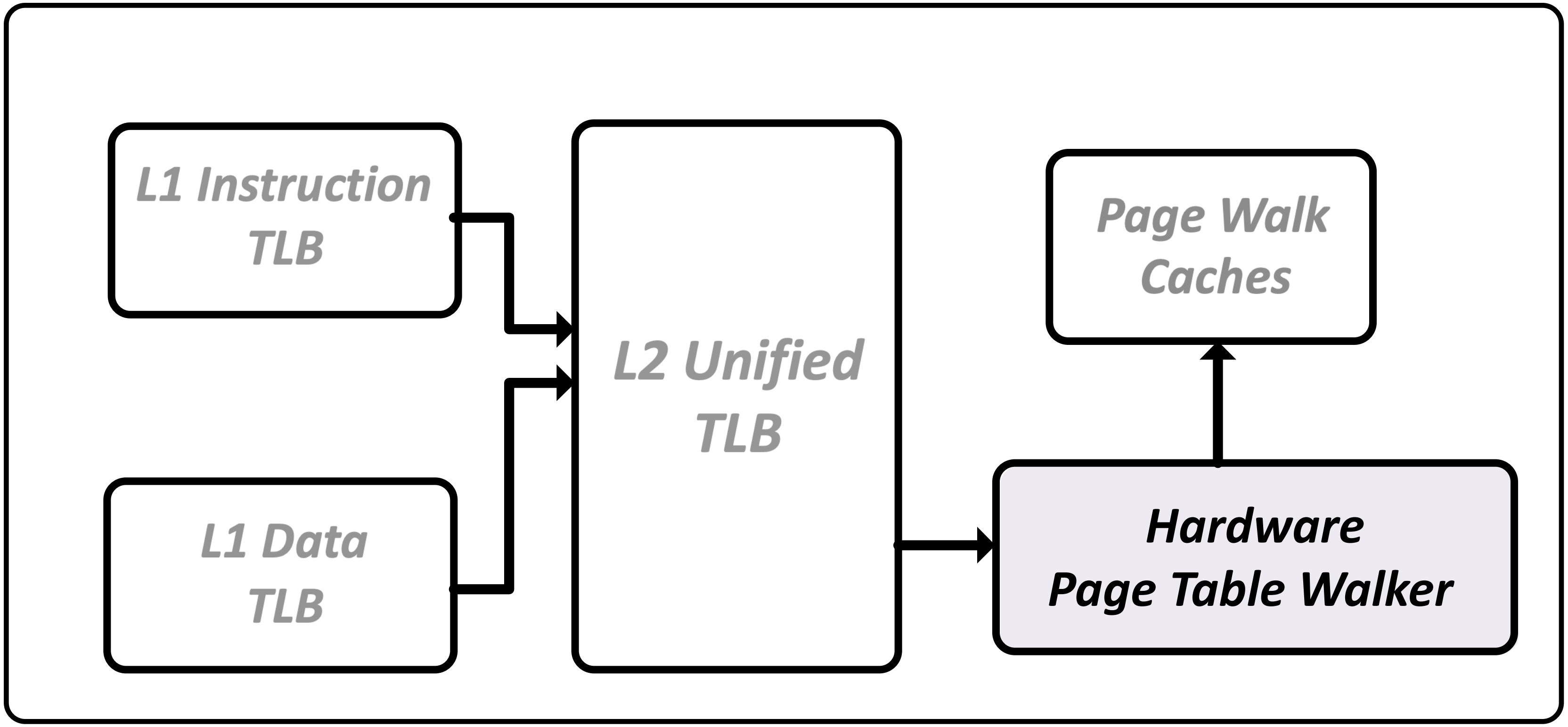
  - Slower identification of L2 TLB Miss as all page sizes need to be tested

- 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

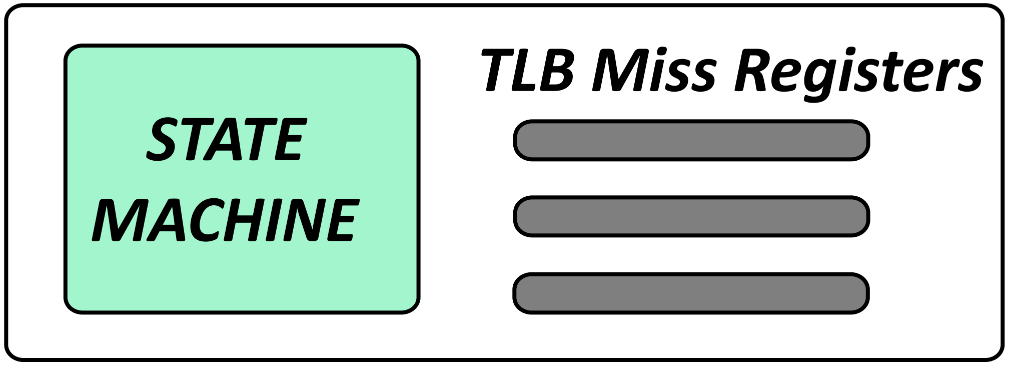


Hardware Page Table Walker (I)

- 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



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

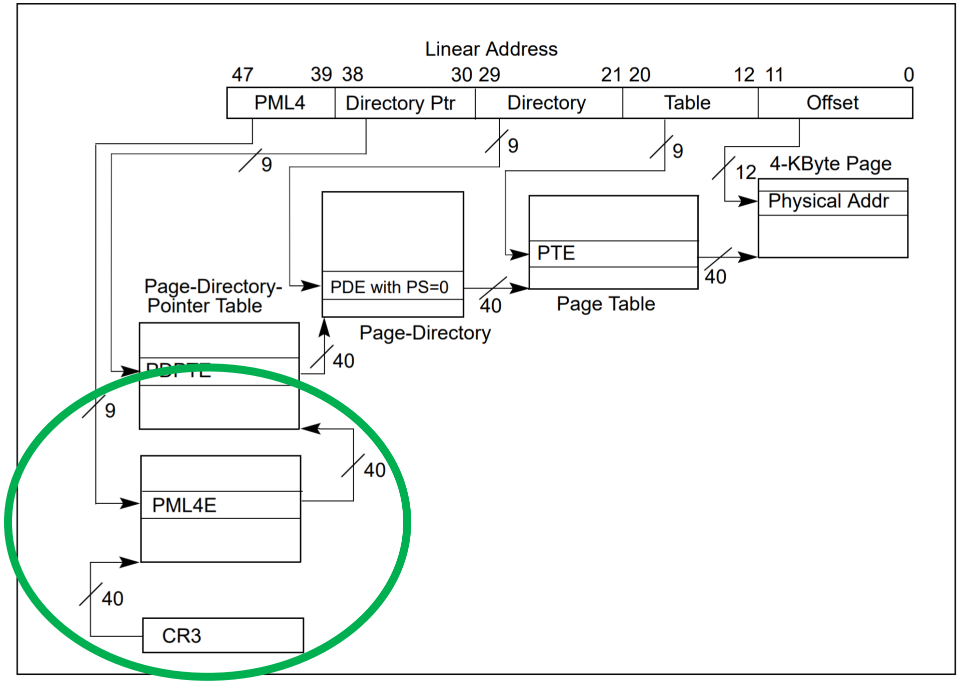
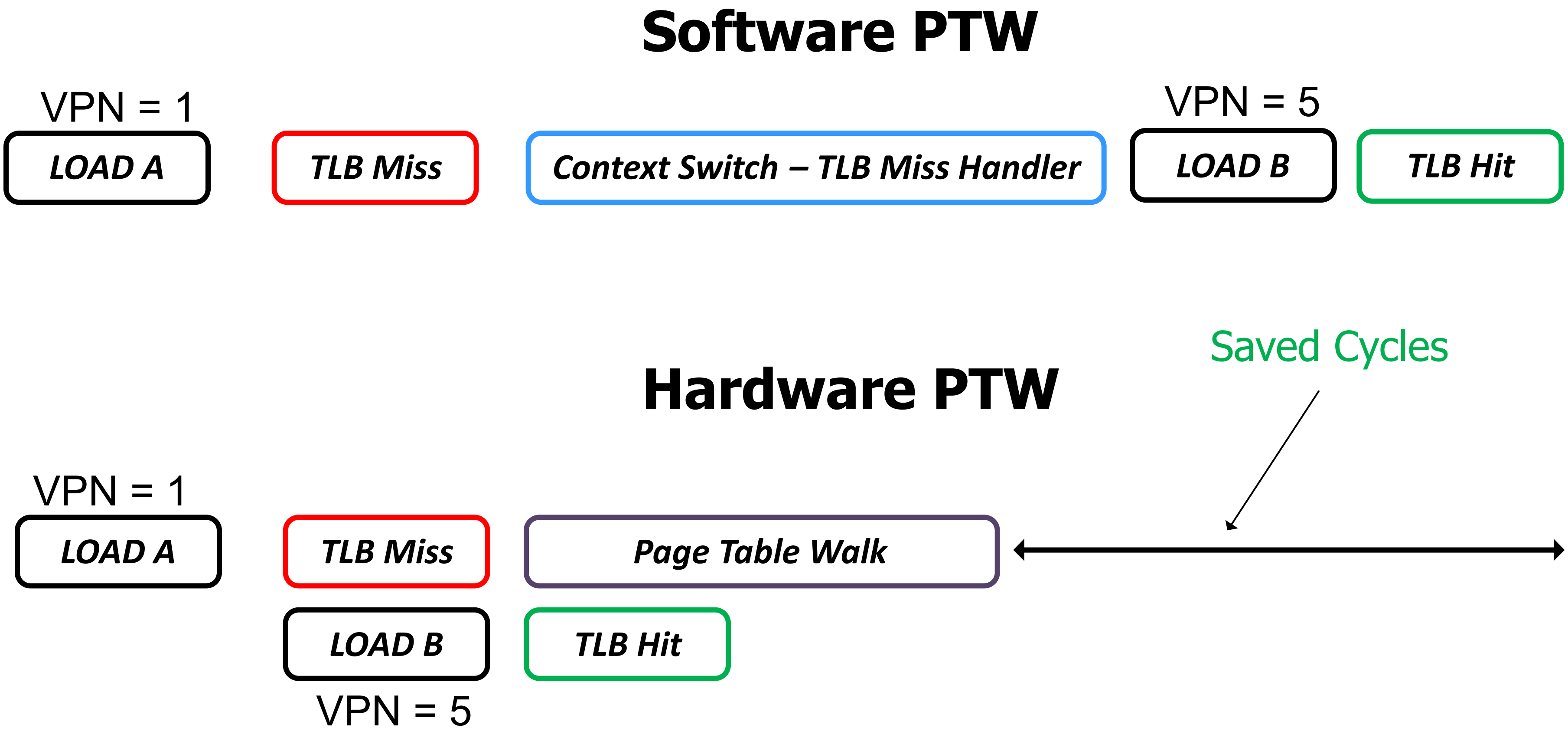


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1

Hardware Page Table Walker (IV)

- Hardware PTWs allow overlapping TLB misses with useful computation



Software PTW

VPN = 1

LOAD A

TLB Miss

Context Switch – TLB Miss Handler

VPN = 5

LOAD B

TLB Hit

Hardware PTW

Saved Cycles

VPN = 1

LOAD A

TLB Miss

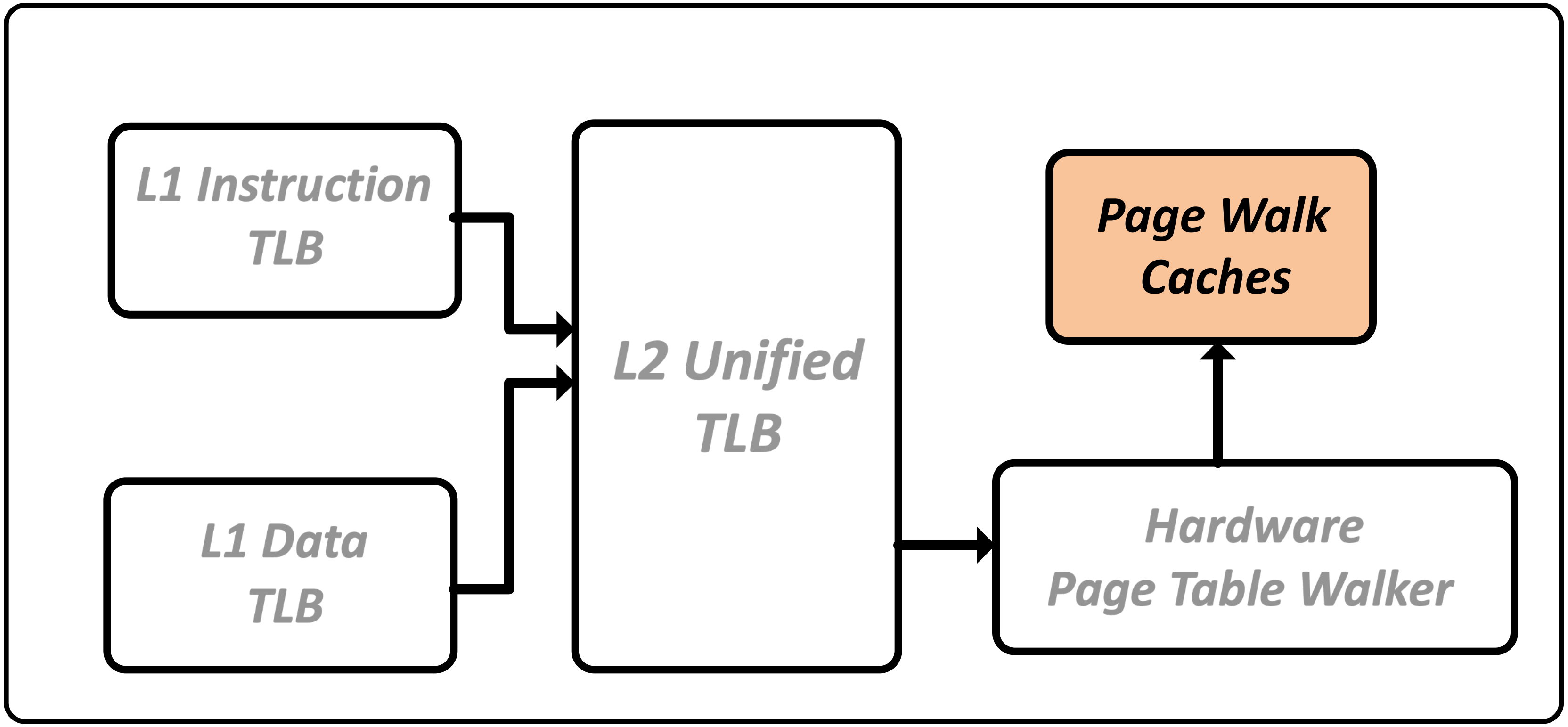
Page Table Walk

LOAD B

TLB Hit

VPN = 5

Page Walk Caches

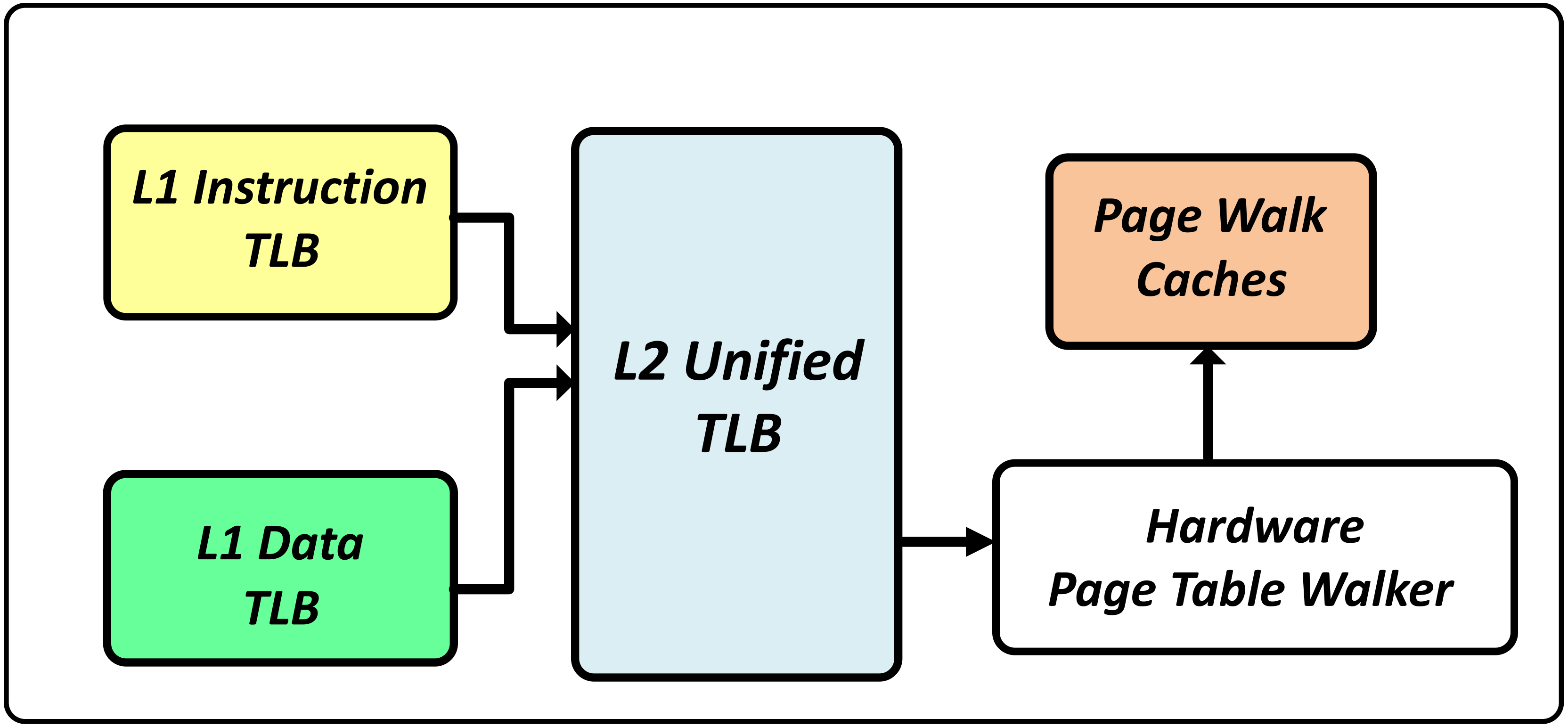


Page Walk Caches

- 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

https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2
https://news.ycombinator.com/item?id=25257932

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…

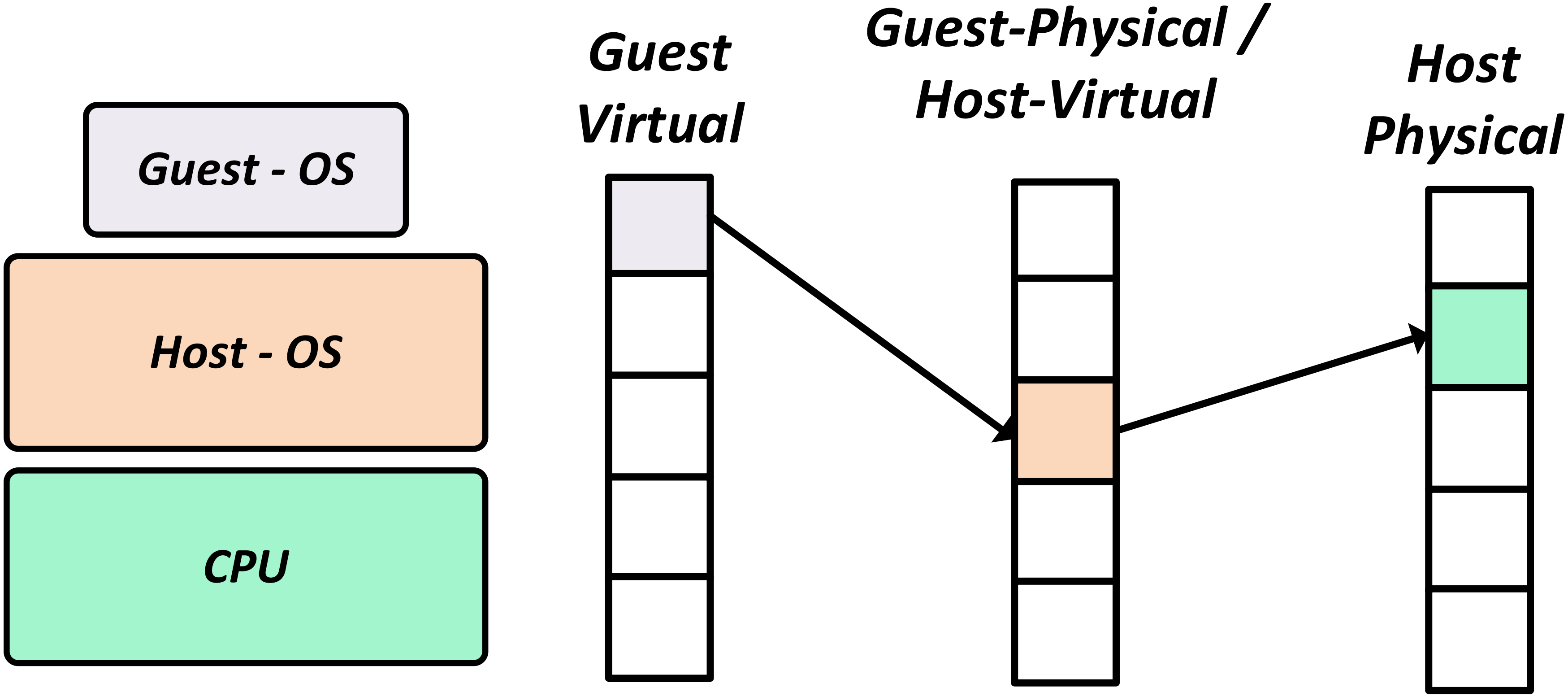
- How to handle virtualized systems?
  - Virtual machines running programs
  - Hypervisors

- 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



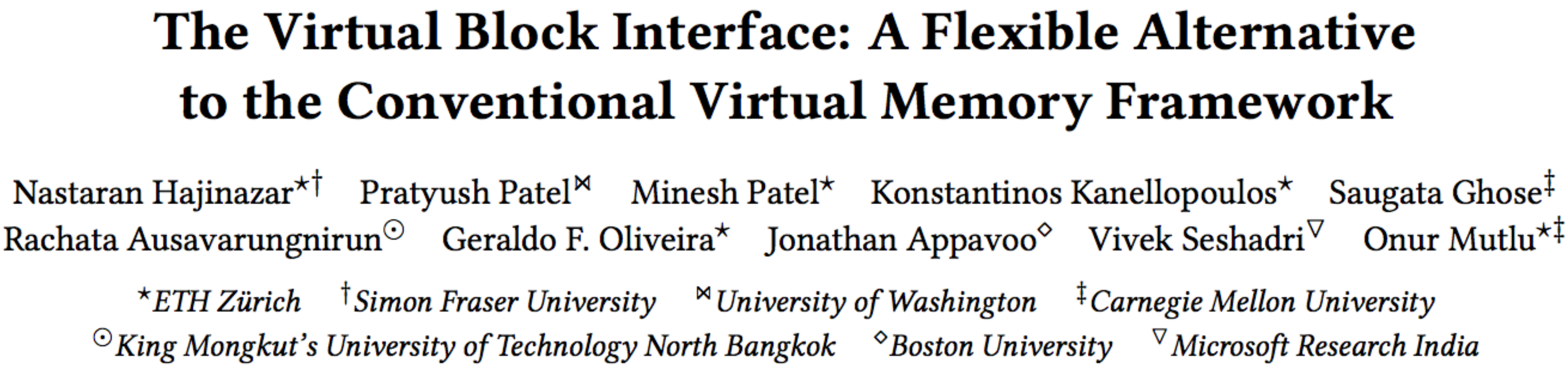
Virtual Memory: Parting Thoughts

- 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)]

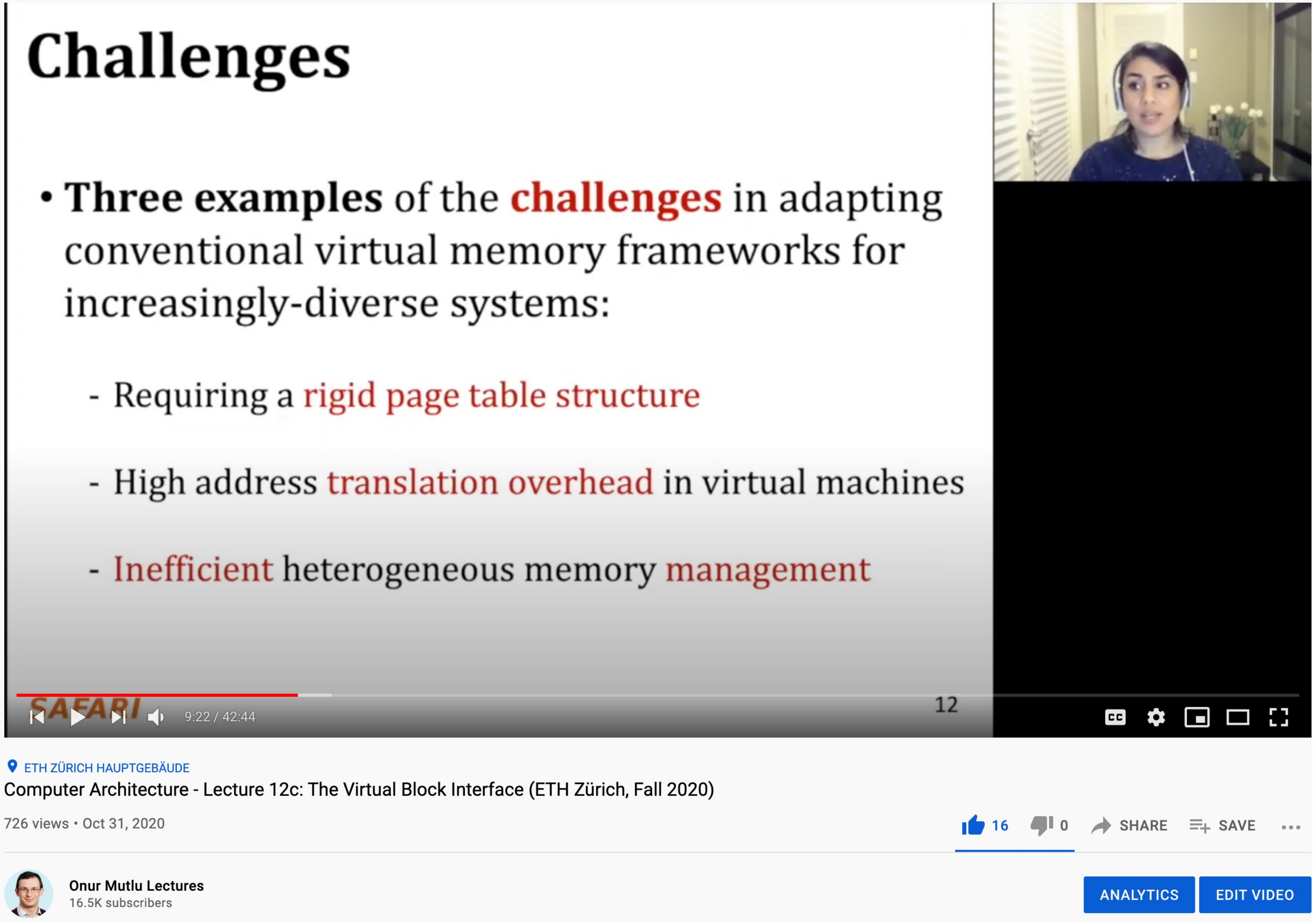


The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework

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

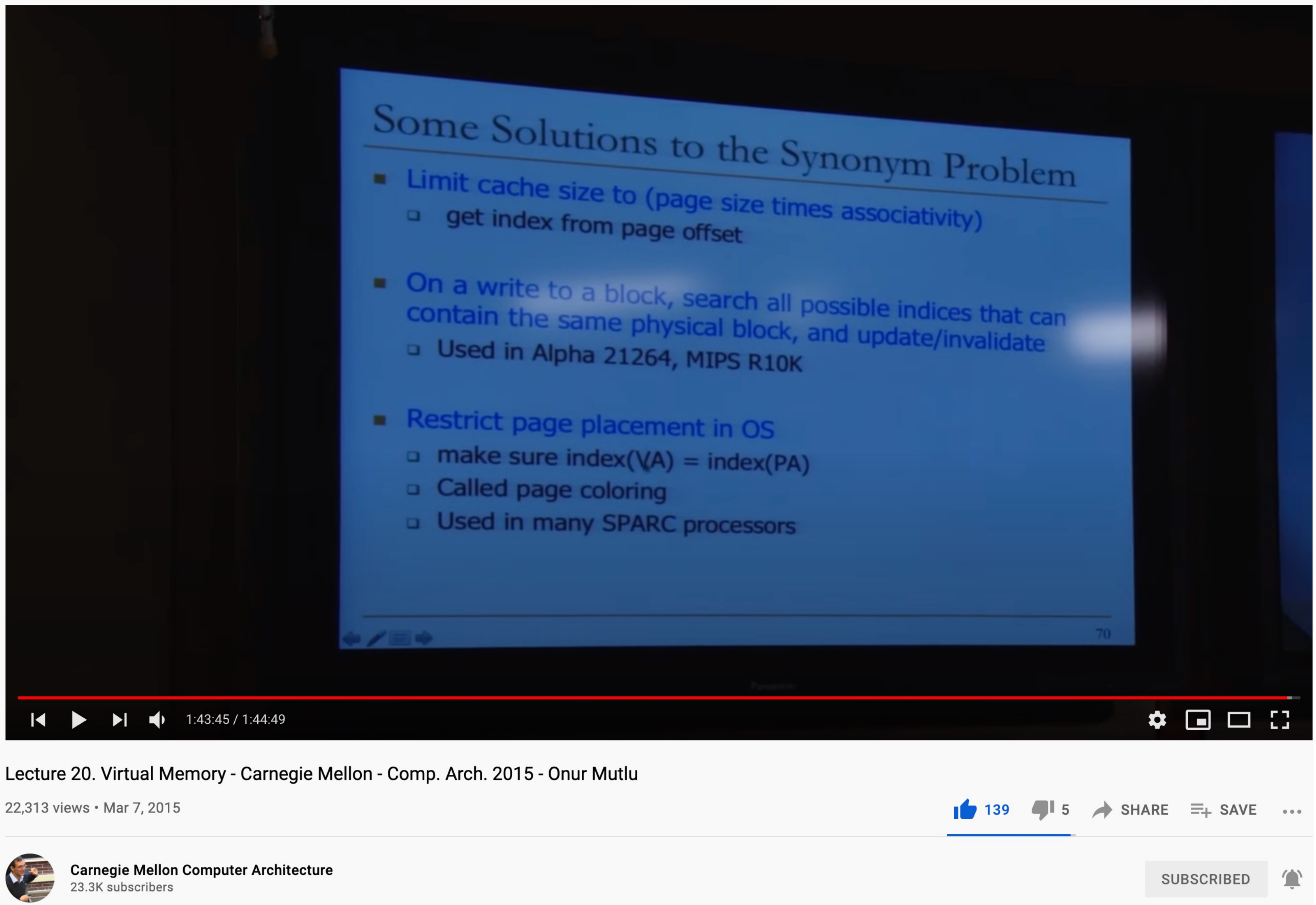
- **Three examples** of the **challenges** in adapting conventional virtual memory frameworks for increasingly-diverse systems:
  - Requiring a rigid page table structure
  - High address translation overhead in virtual machines
  - Inefficient heterogeneous memory management

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

726 views • Oct 31, 2020

https://www.youtube.com/watch?v=2RhGMpY18zw&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&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 index(VA) = index(PA)
  - Called page coloring
  - Used in many SPARC processors

70

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

22,313 views • Mar 7, 2015

Carnegie Mellon Computer Architecture

23.3K subscribers

https://www.youtube.com/watch?v=PPR7YrBi7IQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=24

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 2022
3 June 2022

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

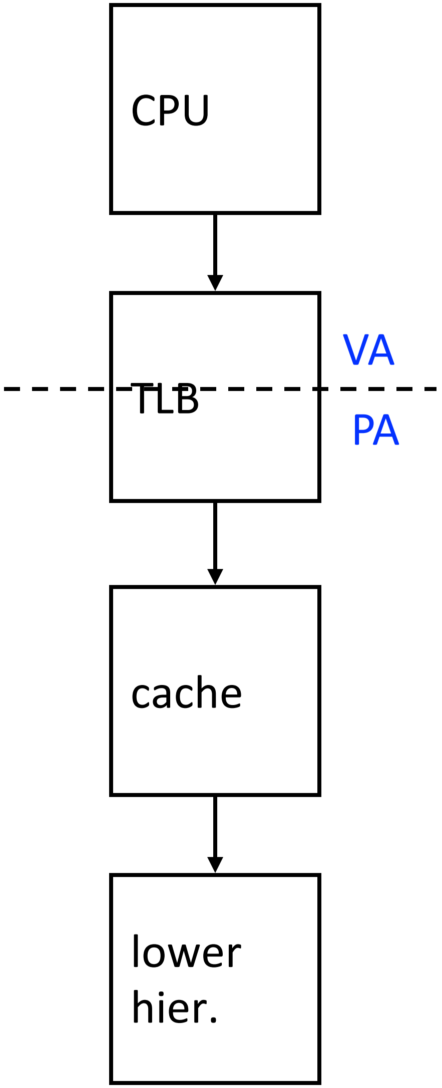
Homonyms and Synonyms

- Homonym: Same VA can map to two different PAs
  - Why?
    - VA is in different processes

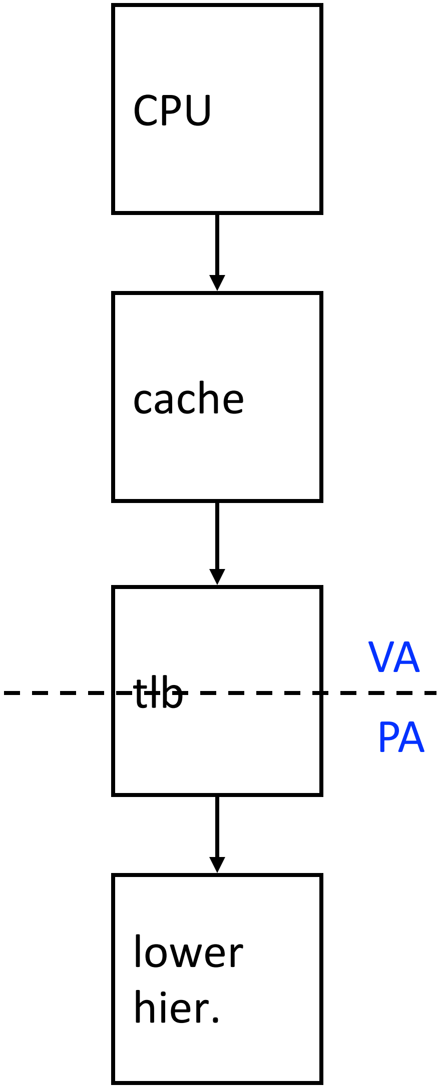
- Synonym: Different VAs can map to the same PA
  - Why?
    - Different pages can share the same physical frame within or across processes
    - Reasons: shared libraries, shared data, copy-on-write pages within the same process, …

- Do homonyms and synonyms create problems when we have a cache?
  - Is the cache virtually or physically addressed?

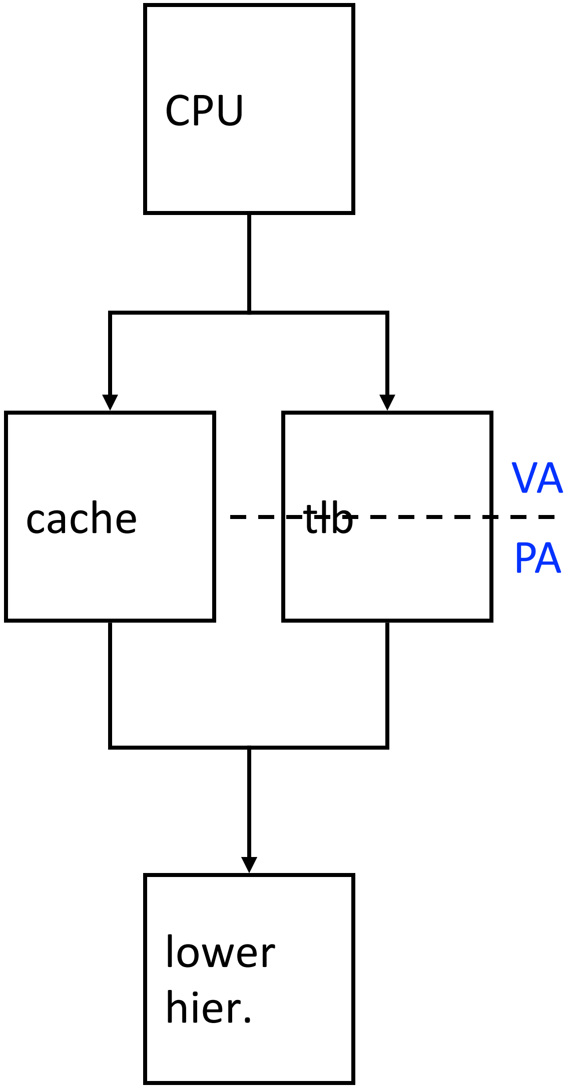
Cache-VM Interaction



physical cache

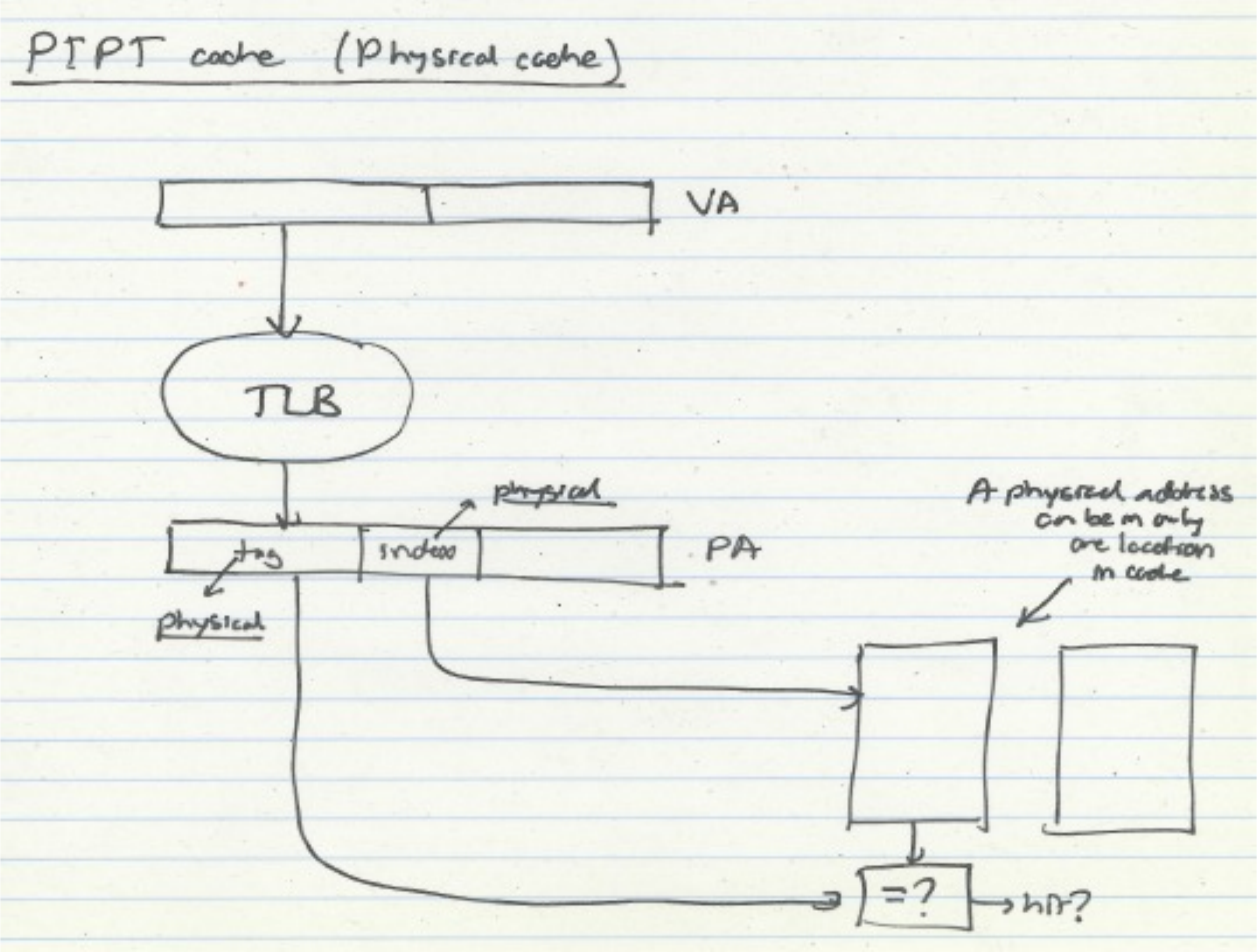


virtual (L1) cache

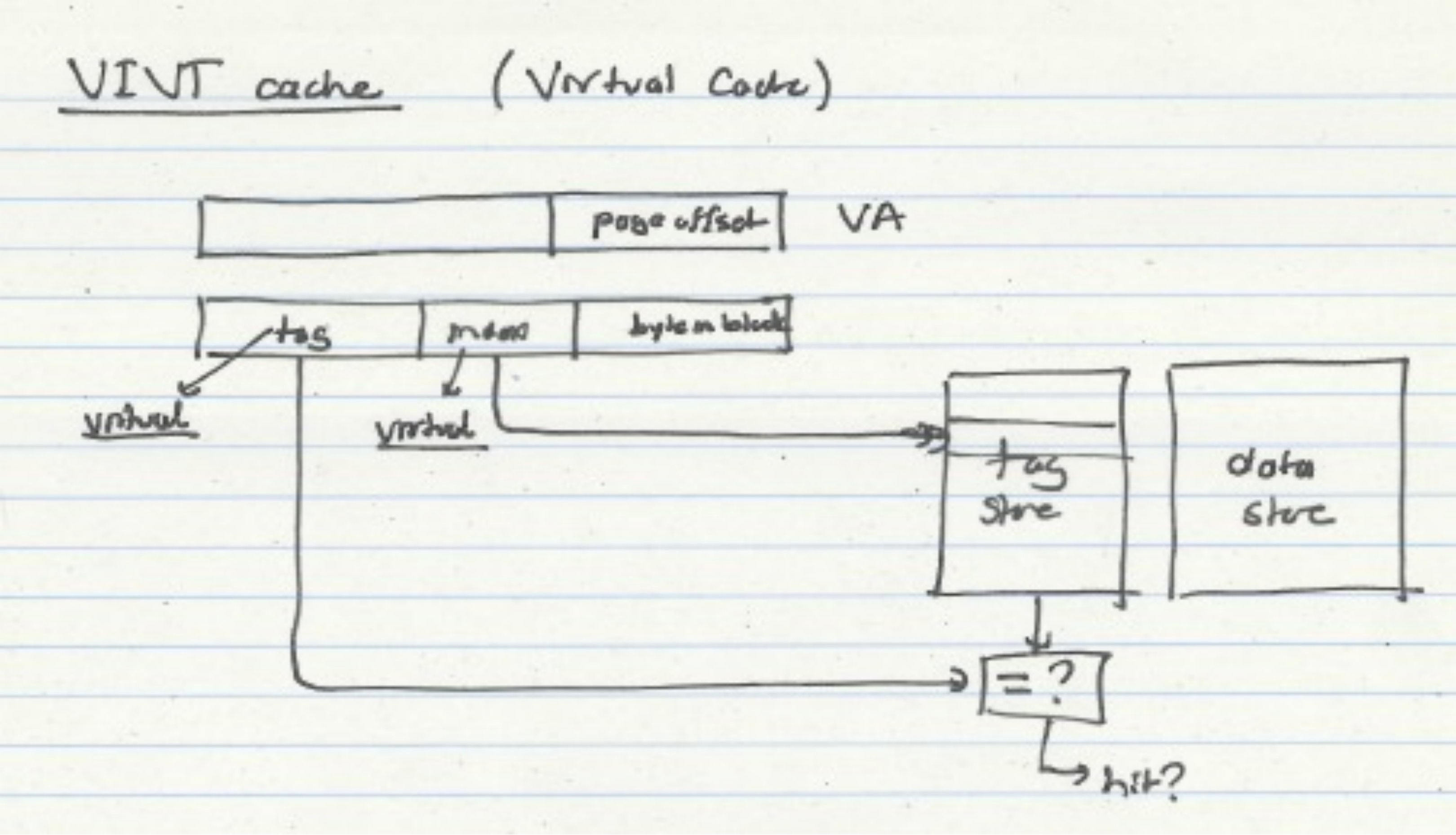


virtual-physical cache

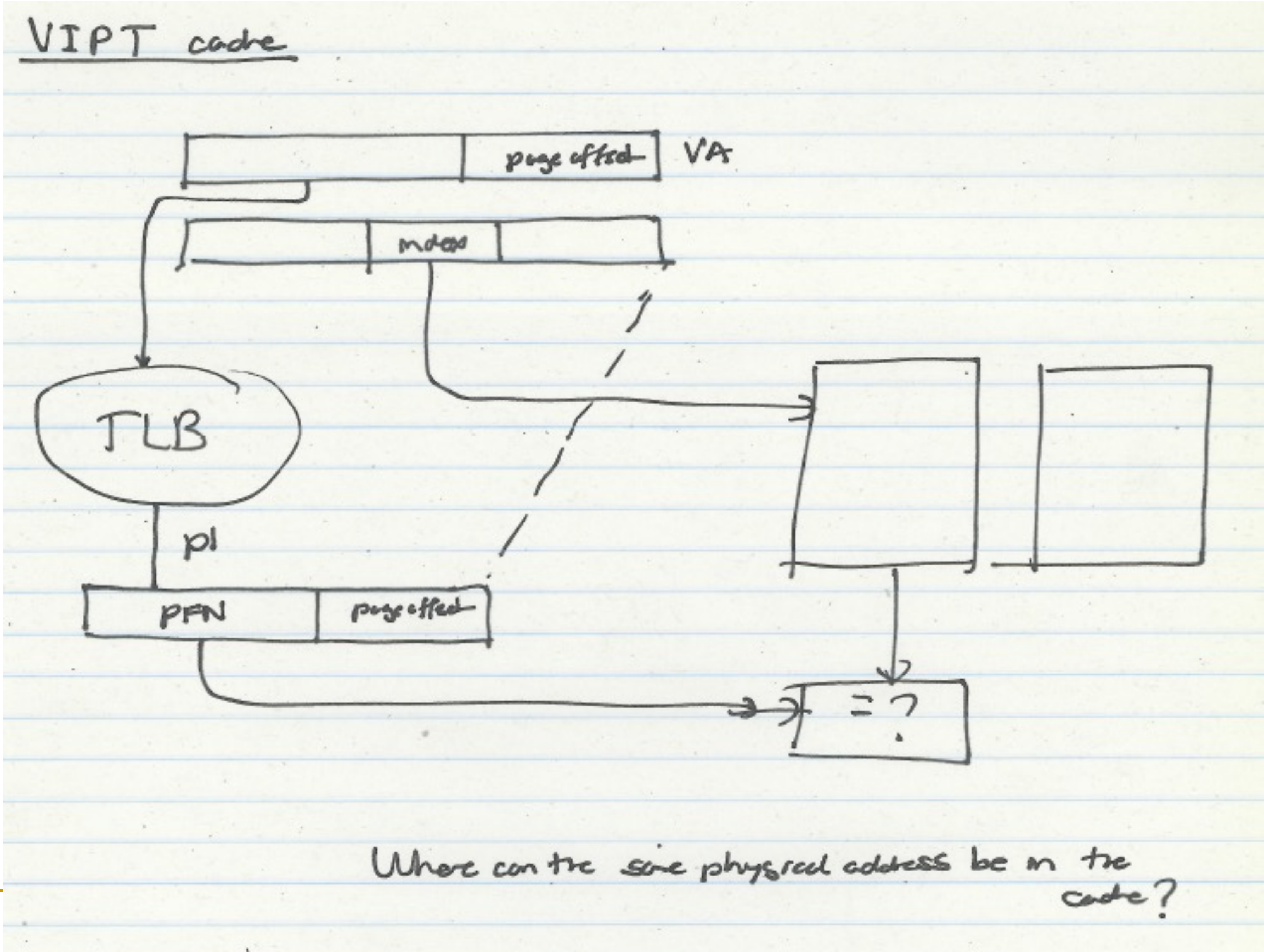
Physical Cache


PIPT cache (Physical cache)
VA
TLB
physical
tag
index
PA
physical
A physical address can be in only one location in cache
=?
hit?

Virtual Cache



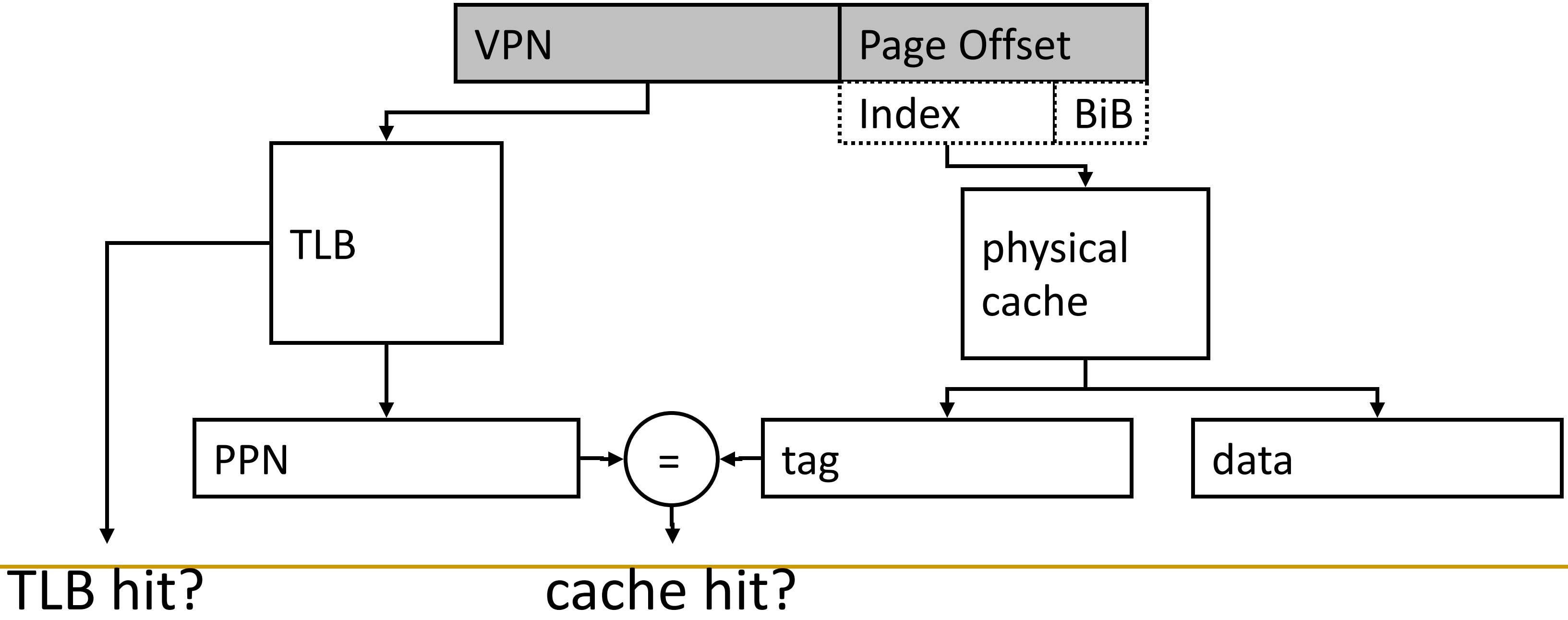
Virtual-Physical Cache


VIPT cache

Where can the same physical address be in the cache?

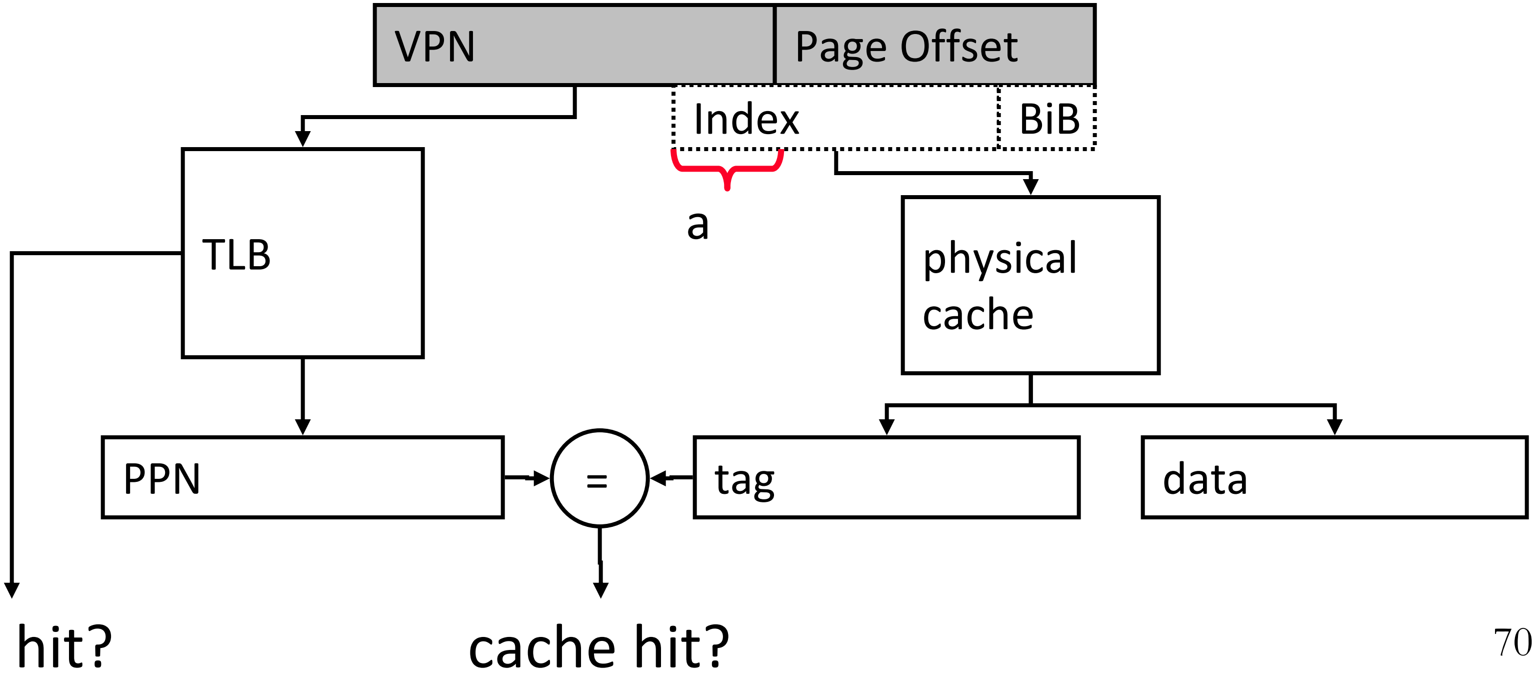
Virtually-Indexed Physically-Tagged

- If (index-bits + byte-in-block-bits < page-offset-bits), the cache index bits come only from page offset (same in VA and PA)
  - Also implies Cache Size ≤ (page size × associativity)
- 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 (index-bits + byte-in-block-bits < page-offset-bits), 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 index(VA) = index(PA)
  - Called page coloring
  - Used in many SPARC processors

L1-D Cache in Intel Skylake

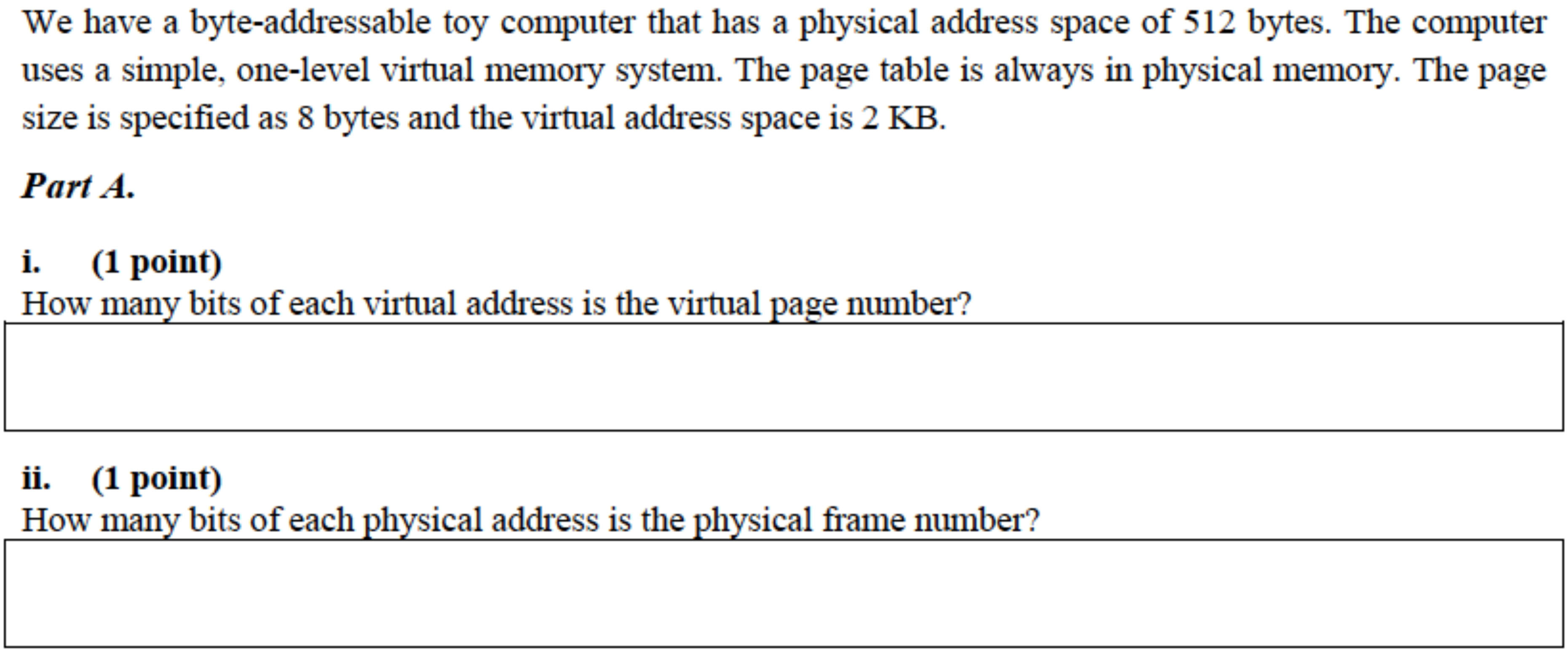
- 32 KB, 64B cacheline size, 8-way associative, 64 sets

- Virtually-indexed physically-tagged (VIPT)

- #set-index bits (6) + #byte-in-block-bits (6) = log2(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://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:

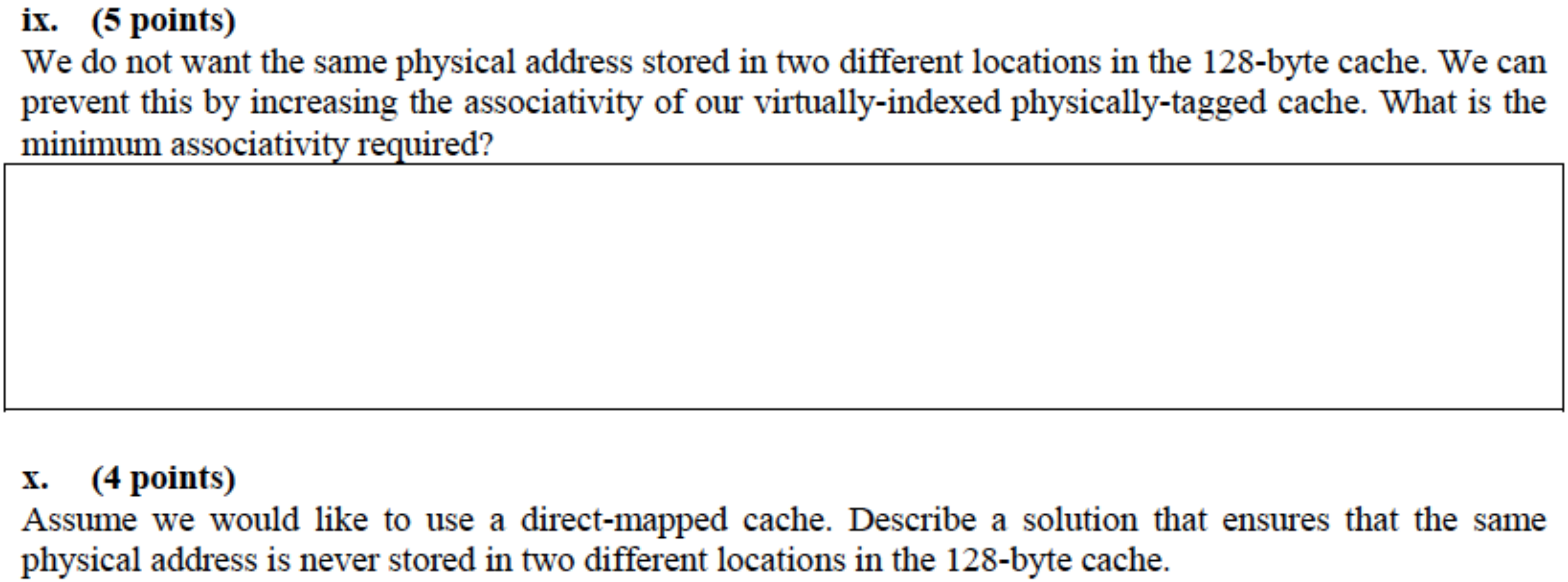
PROCESS 0	
Virtual Page	Physical Frame
Page 0	Frame 0
Page 3	Frame 7
Page 7	Frame 1
Page 15	Frame 3

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

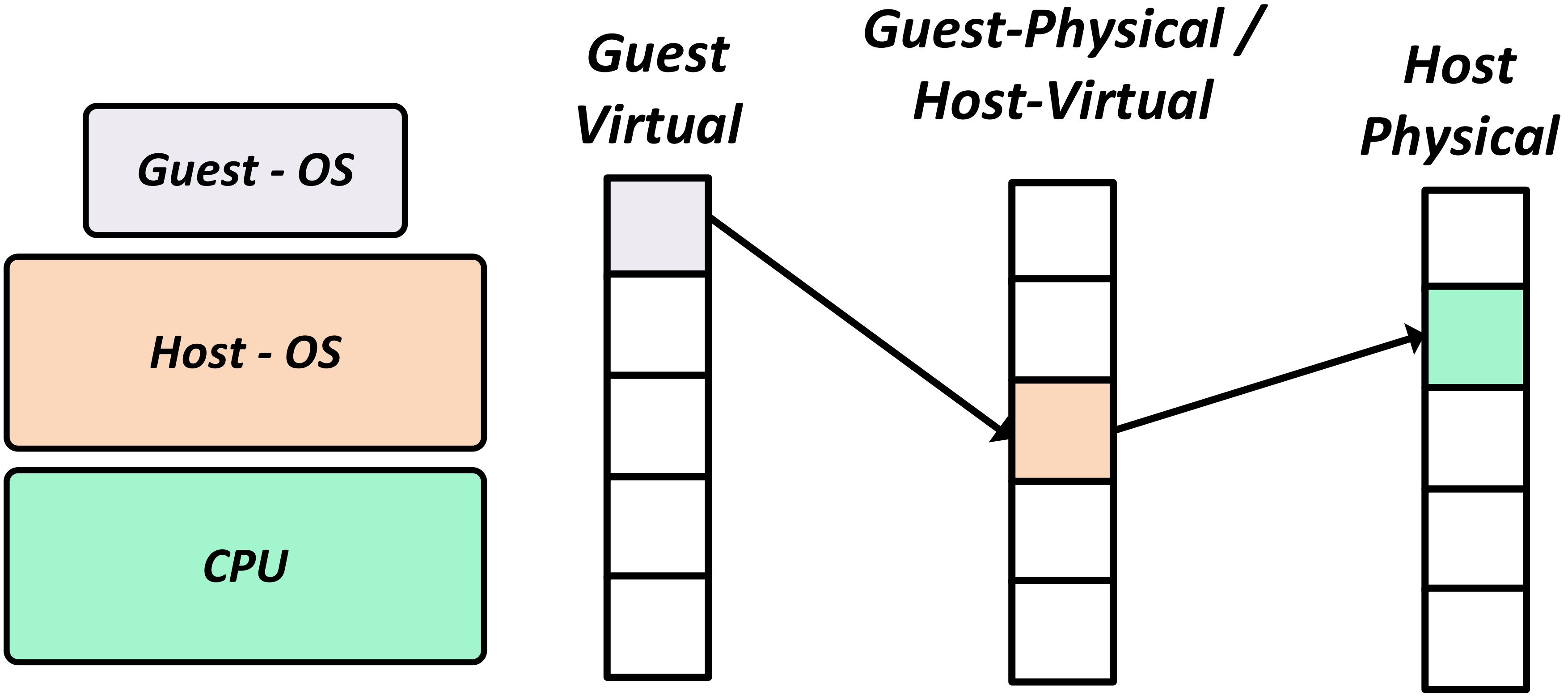
  - Con: Cannot have fine-grained permissions
  - Con: Large transfers to/from disk
    - Even when only 1KB is needed, 1GB must be transferred
    - Waste of bandwidth/energy
    - Reduces performance
  - Con: **Internal fragmentation**
    - Even when only 1KB is needed, 1GB must be allocated
    - Waste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

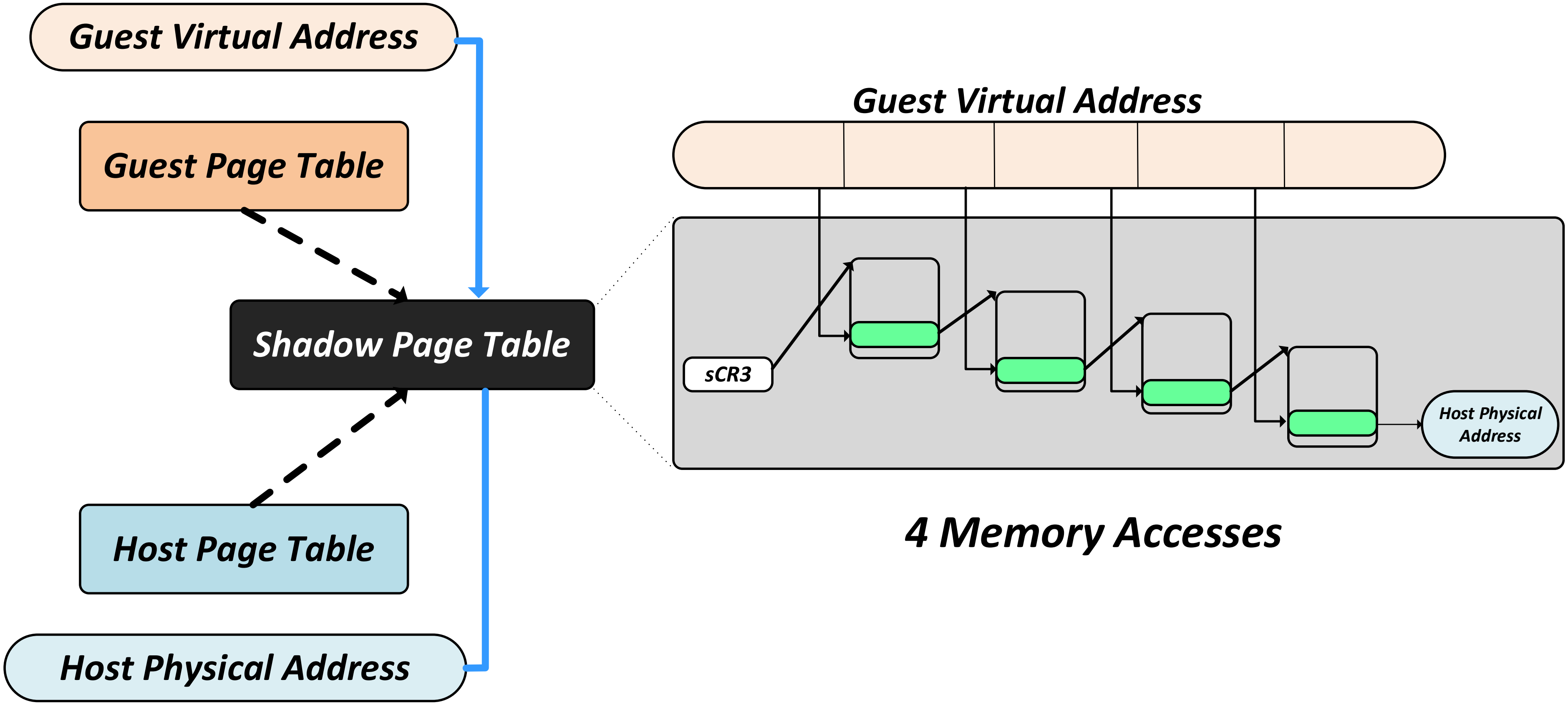
- Pros:

  + 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



Guest Virtual Address

Guest Page Table

Shadow Page Table

Host Page Table

Host Physical Address

Guest Virtual Address

sCR3

Host Physical Address

4 Memory Accesses

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

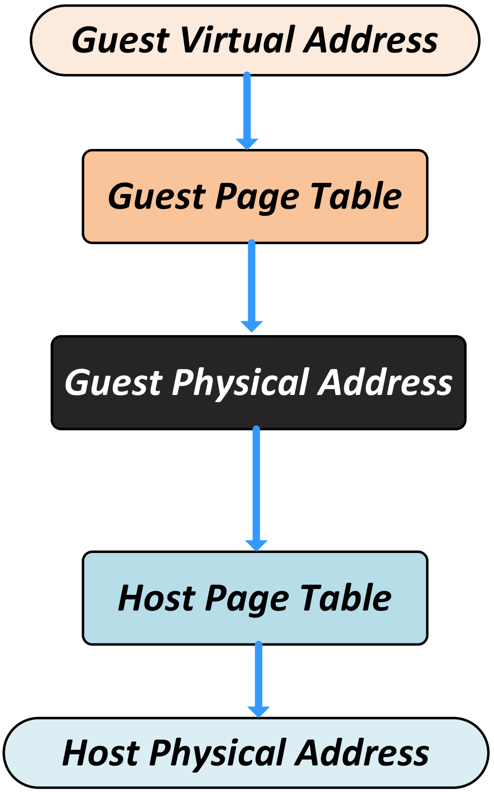
- Pros:

  + Easy for the system to maintain/update two page tables

- Cons:

  - TLB Misses are more costly (up to 24 memory accesses)

Nested Paging



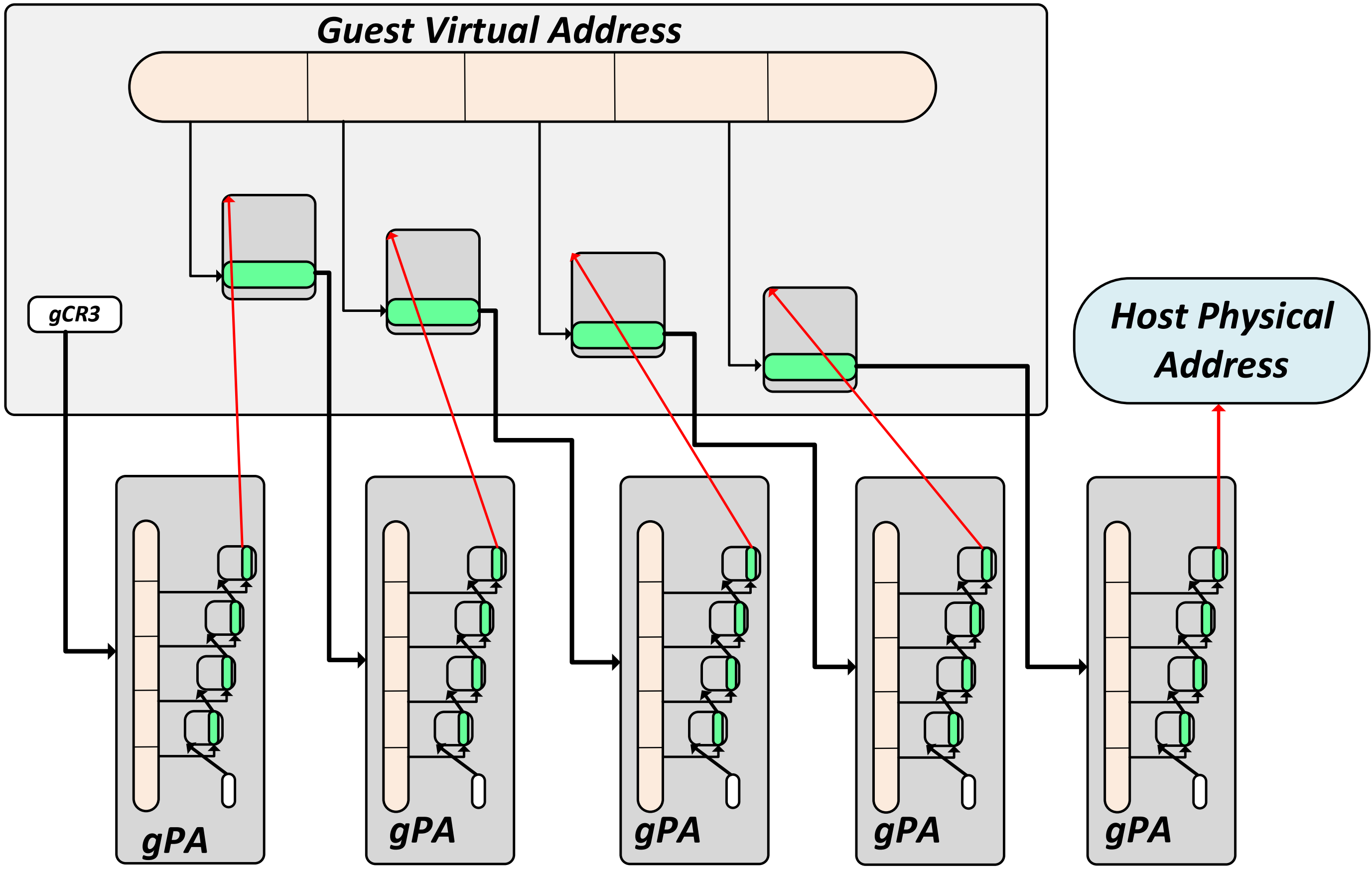
Guest Virtual Address

Guest Page Table

Guest Physical Address

Host Page Table

Host Physical Address



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