Lecture 13:

Virtual Memory

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2025

Today: All About Virtual Memory

- The virtual memory subsystem:
  - Fundamentals
  - How hardware supports virtual memory
  - Techniques to alleviate address translation latency
  - Translation coherence

The Address Translation Wall

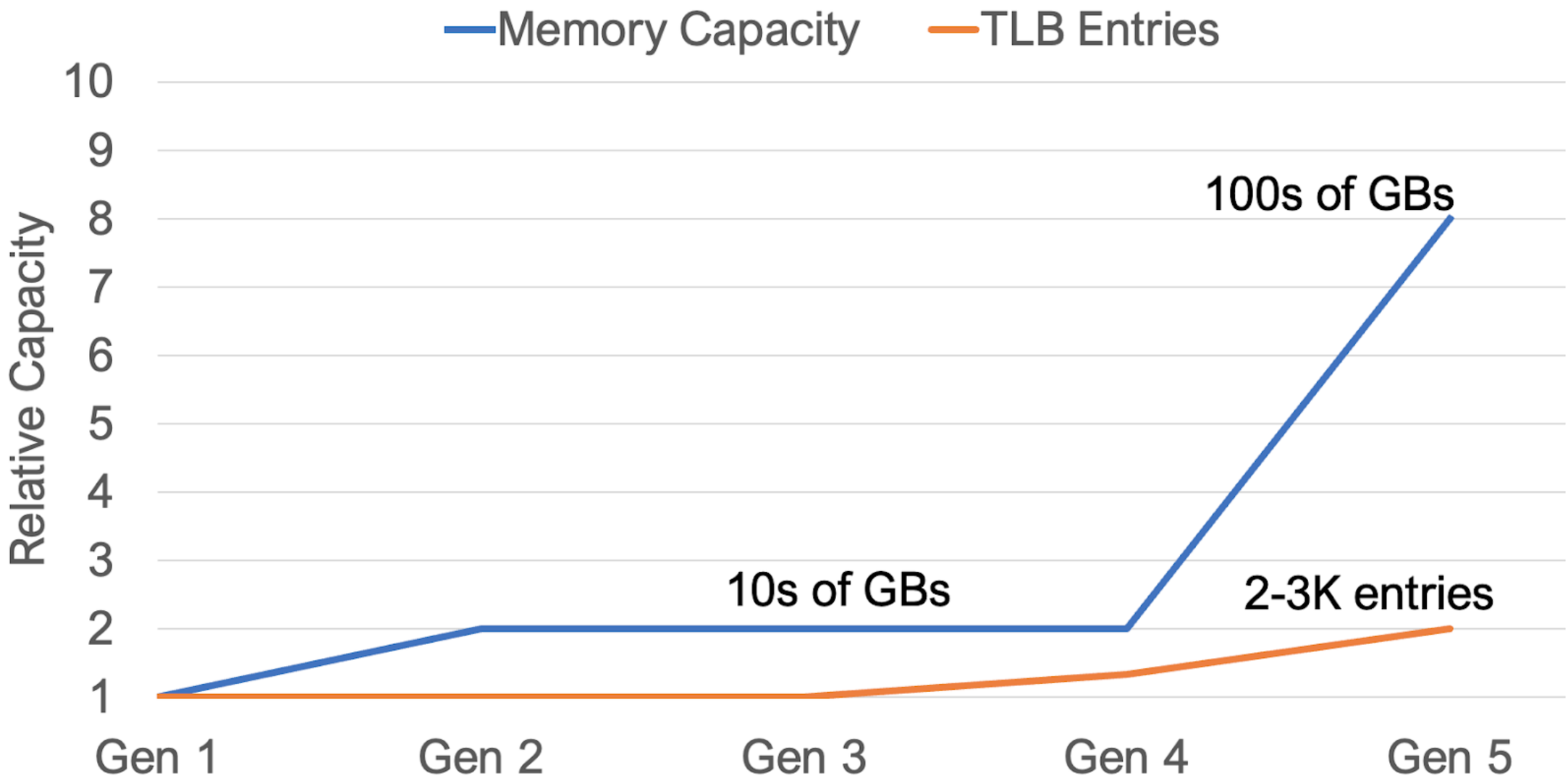


Image from: Zhao *et al*, "Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters." In Proceedings of ISCA'23.

Virtual memory in the real world

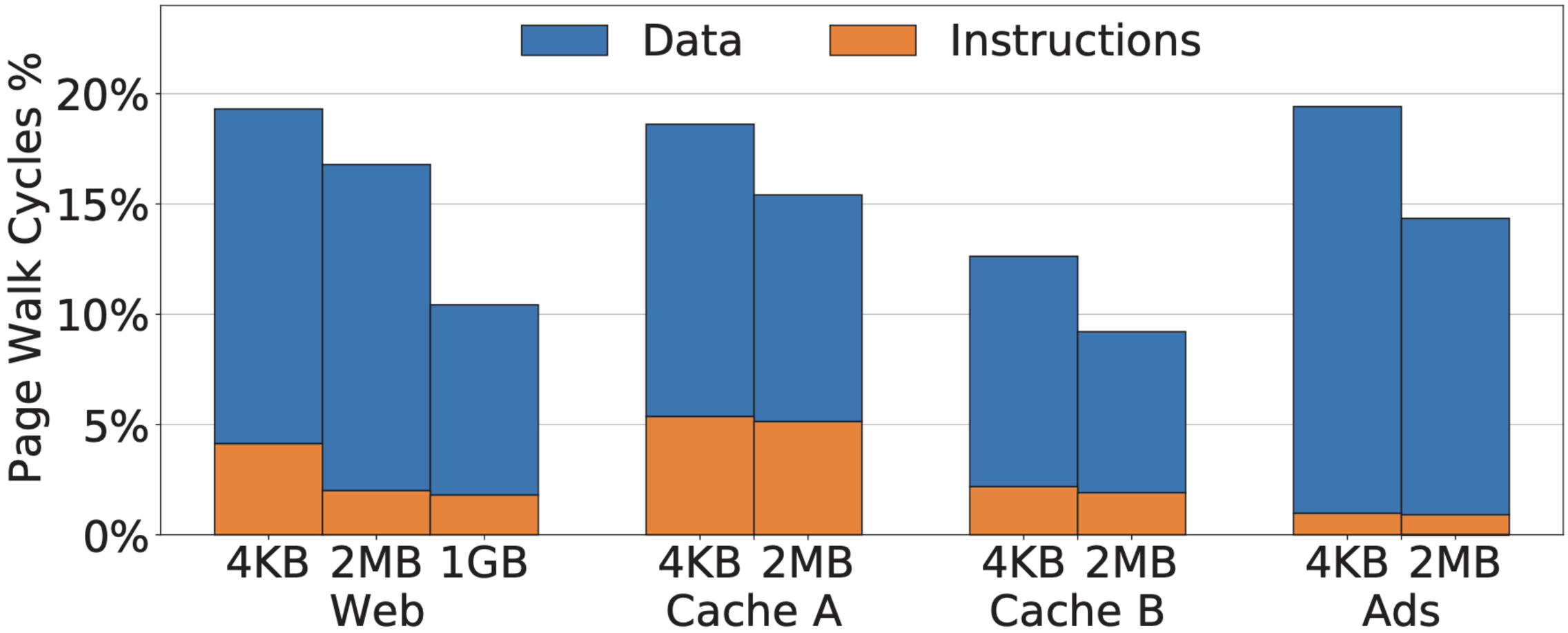
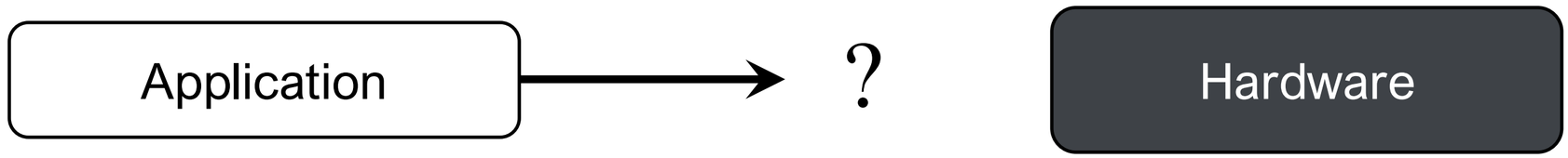


Image from: Zhao *et al*, "Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters." In Proceedings of ISCA'23.

Example

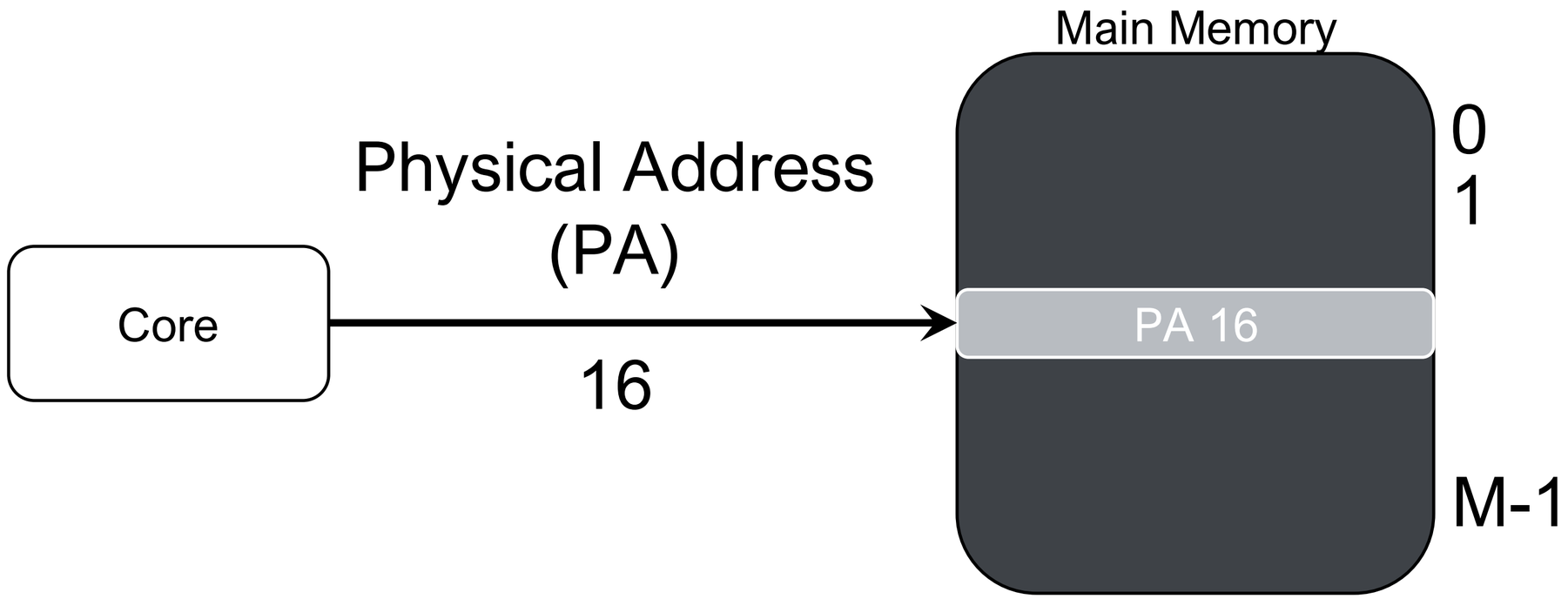
- **int array[10]**

- **Where does a value live in memory?**
- **How does the processor find it?**



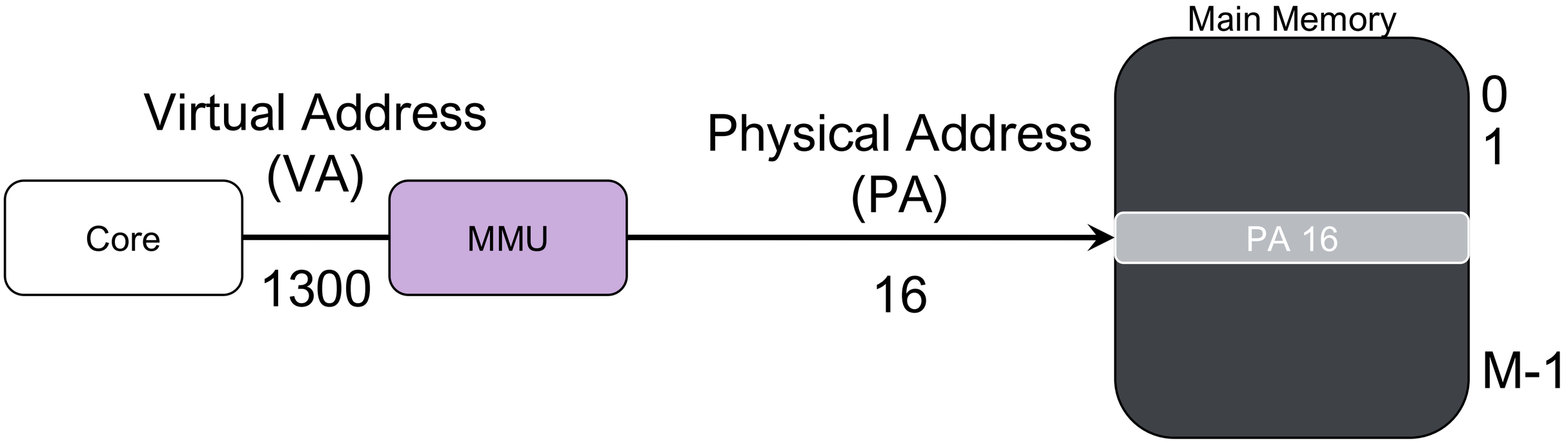
Physical addressing

- **Found in "simple" systems like embedded microcontrollers**
- **Limited programmability**



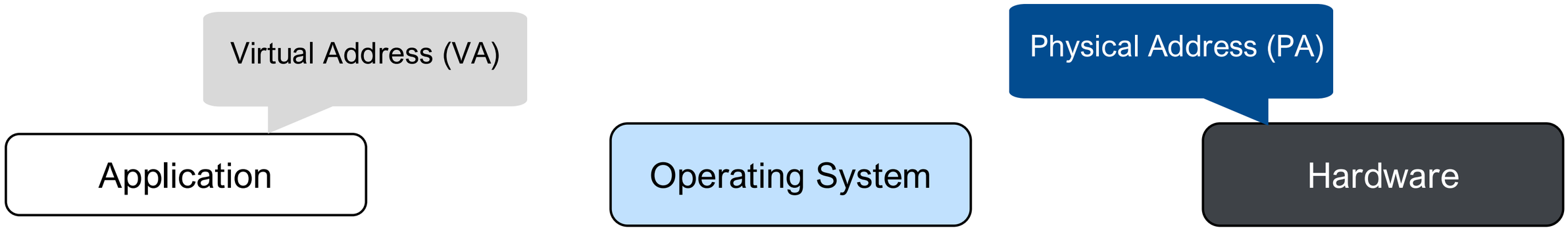
Virtual Addressing

- **Found in all modern systems! Servers, laptops, phones, tablets**
- **One of the great ideas in computer science**

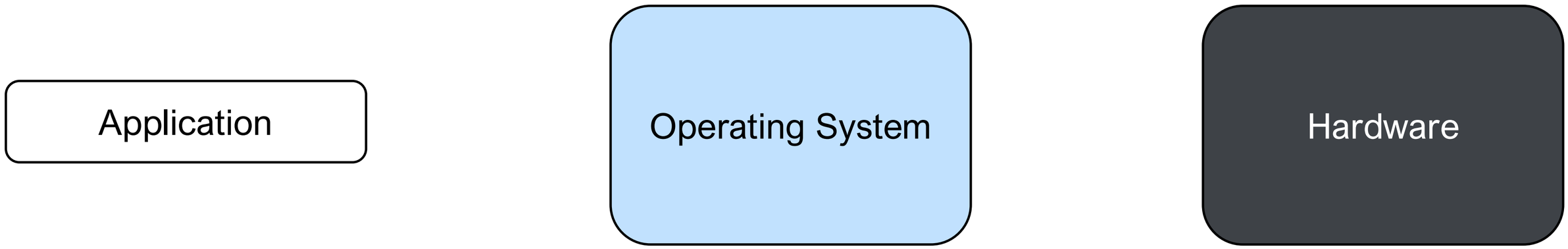


- **Efficient memory usage → DRAM caches part of the VA space**
- **Programmability → Each process gets a linear address space**

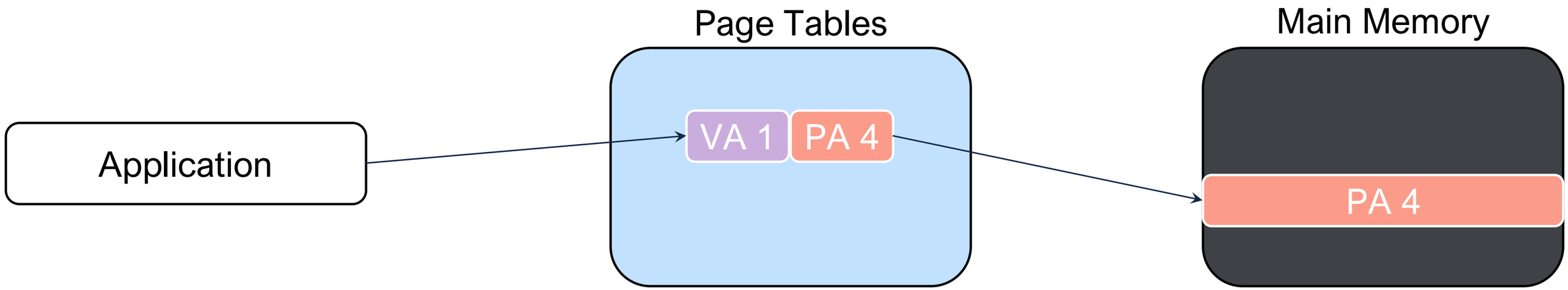
Virtual Memory Abstraction



Virtual Memory Abstraction

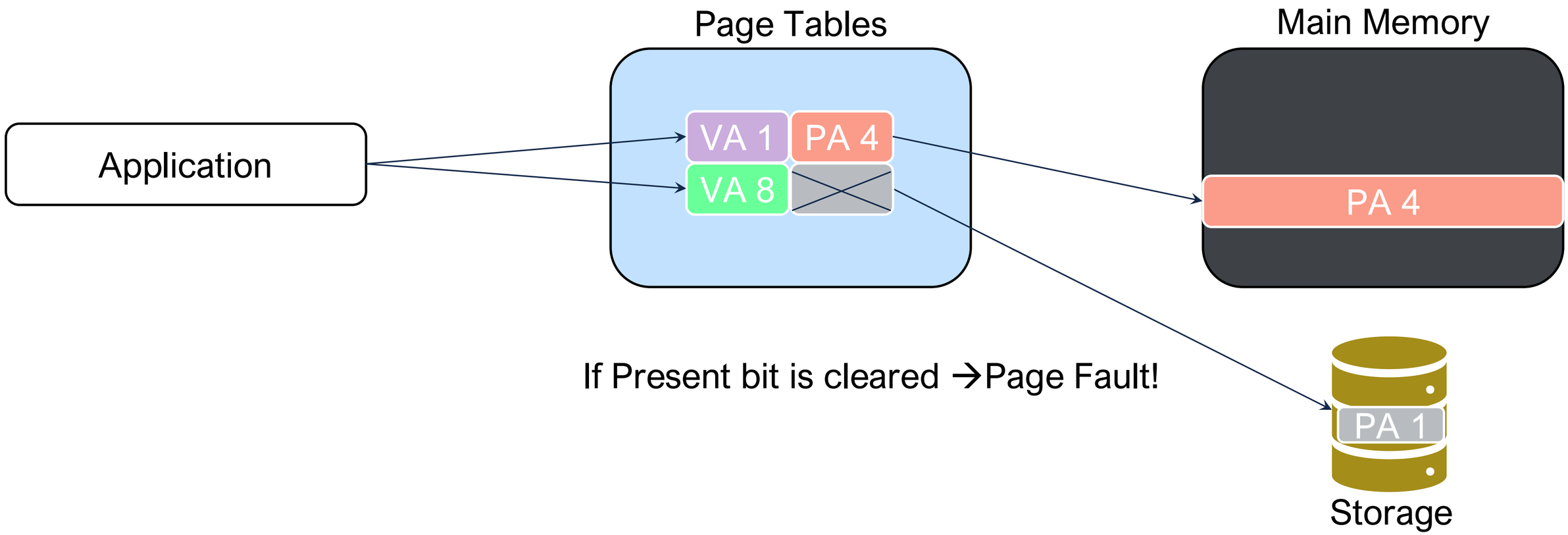


Virtual Memory Abstraction



Virtual Memory Abstraction

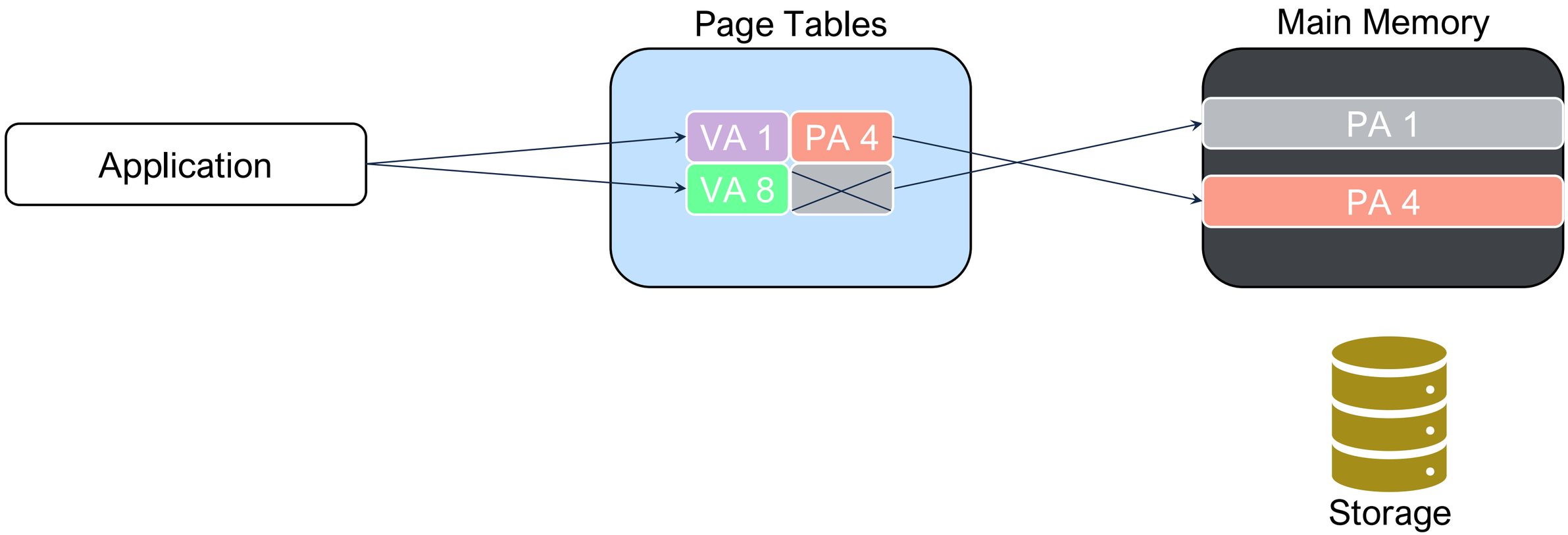
- Split the physical address space into pages
- Use page tables to map virtual-to-physical mappings at page granularity
- Page fault: Page is not in memory → OS handled fetching the page and updated the entry



Page fault: In this case we are discussing a major page fault. A minor page fault the page is in memory but not yet mapped to the address space of the process. Both handled by the OS.

Virtual Memory Abstraction

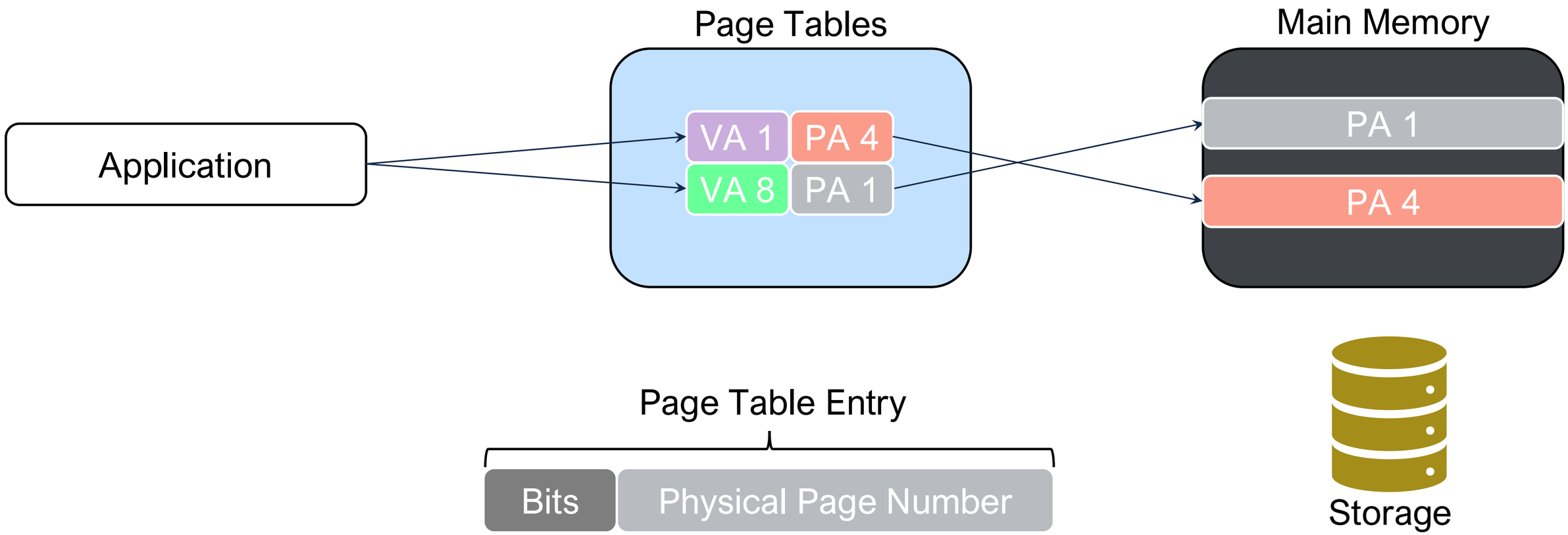
- Split the physical address space into pages
- Use page tables to map virtual-to-physical mappings at page granularity
- Page fault: Page is not in memory → OS handled fetching the page and updated the entry



Page fault: In this case we are discussing a major page fault. A minor page fault the page is in memory but not yet mapped to the address space of the process. Both handled by the OS.

Virtual Memory Abstraction

- Split the physical address space into pages
- Use page tables to map virtual-to-physical mappings at page granularity
- Page fault: Page is not in memory → OS handled fetching the page and updated the entry



Virtual Memory Abstraction

- Split the physical address space into pages
- Use p
- Page

...e entry

Table 4-20. Format of a 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 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
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)	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
(M–1):12	Physical address of the 4-KByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1 or CR4.PKS = 1, this may control the page's access rights (see Section 4.6.2); otherwise, it is not used to control access rights.
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Applic

emory

1

4

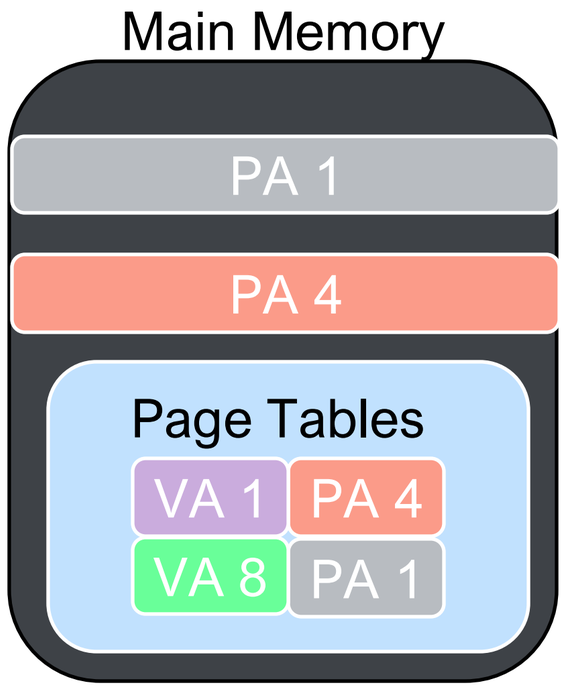
rage

Virtual Memory in Hardware

- How hardware supports virtual memory?

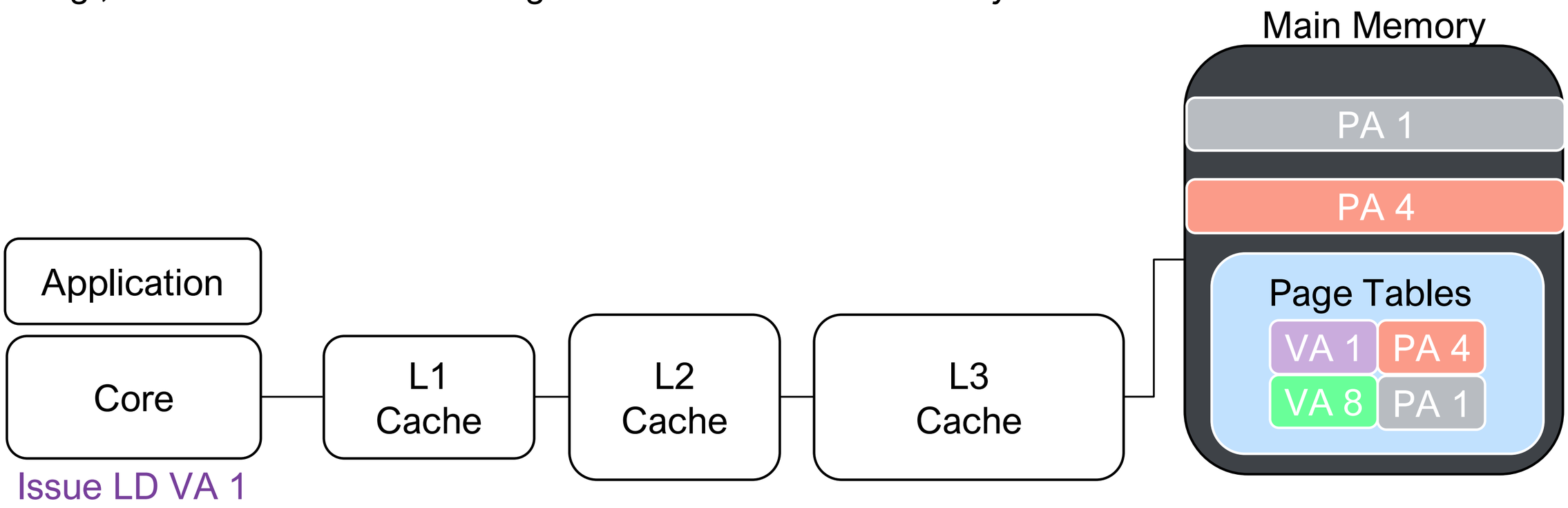
- What we will cover next:
  - TLB
  - Page walks
  - Interaction with caches

Virtual Memory Abstraction



How to Find Translations?

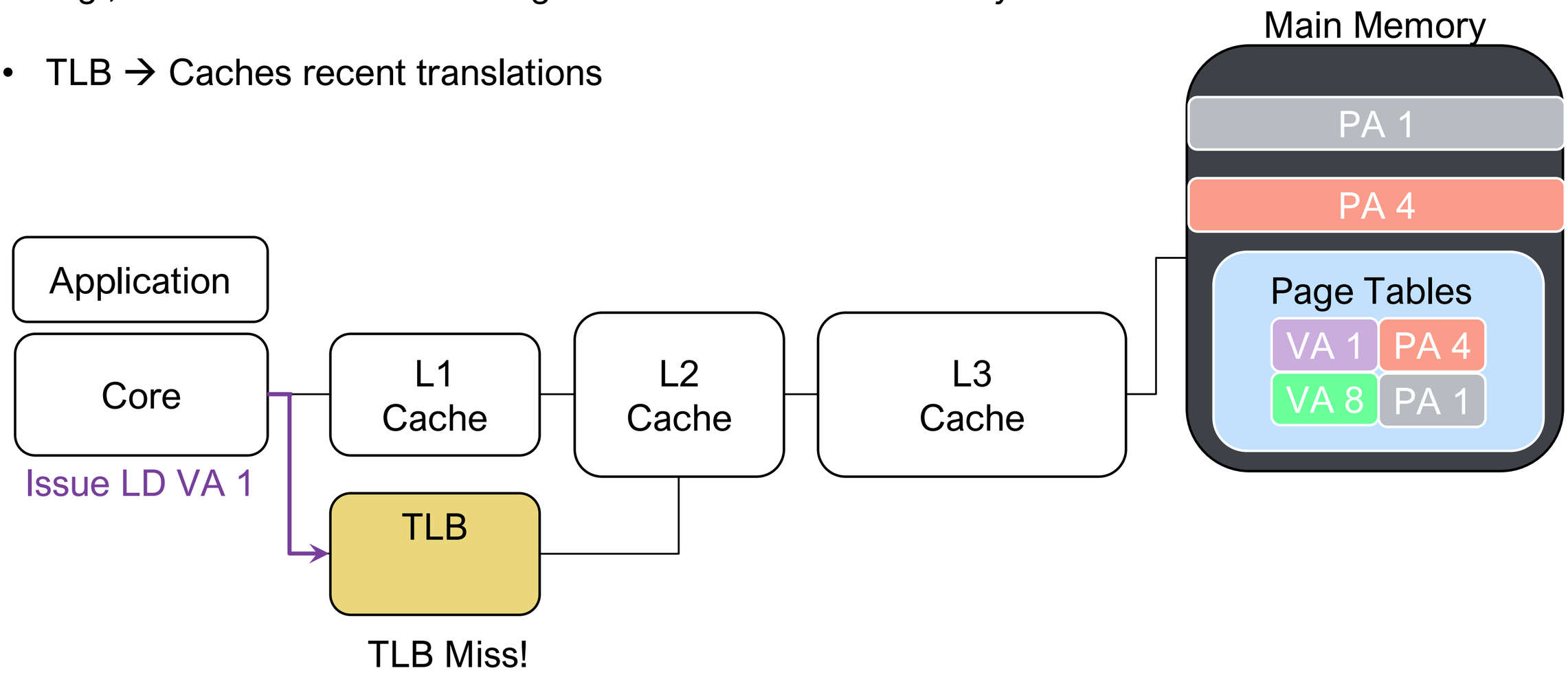
- Fetching each translation on a load would be expensive!
- How about locality? Translations cover pages but we operate on cache lines
- E.g., 4KB translations → Enough for 64 cache lines of 64 bytes



Translation Lookaside Buffer

- Fetching each translation on a load would be expensive!
- How about locality? Translations cover pages but we operate on cache lines
- E.g., 4KB translations → Enough for 64 cache lines of 64 bytes

- TLB → Caches recent translations

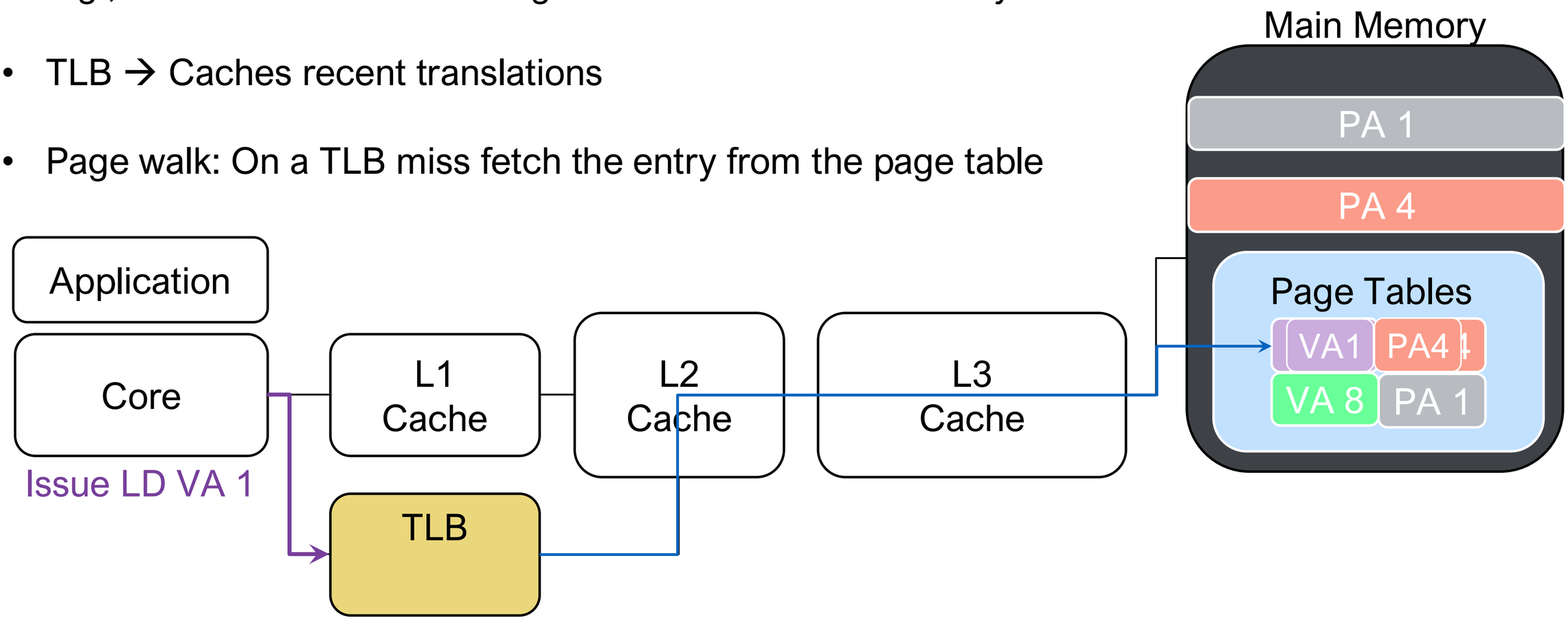


How to Fill the TLB? Page Walk

- Fetching each translation on a load would be expensive!
- How about locality? Translations cover pages but we operate on cache lines
- E.g., 4KB translations → Enough for 64 cache lines of 64 bytes

- TLB → Caches recent translations

- Page walk: On a TLB miss fetch the entry from the page table



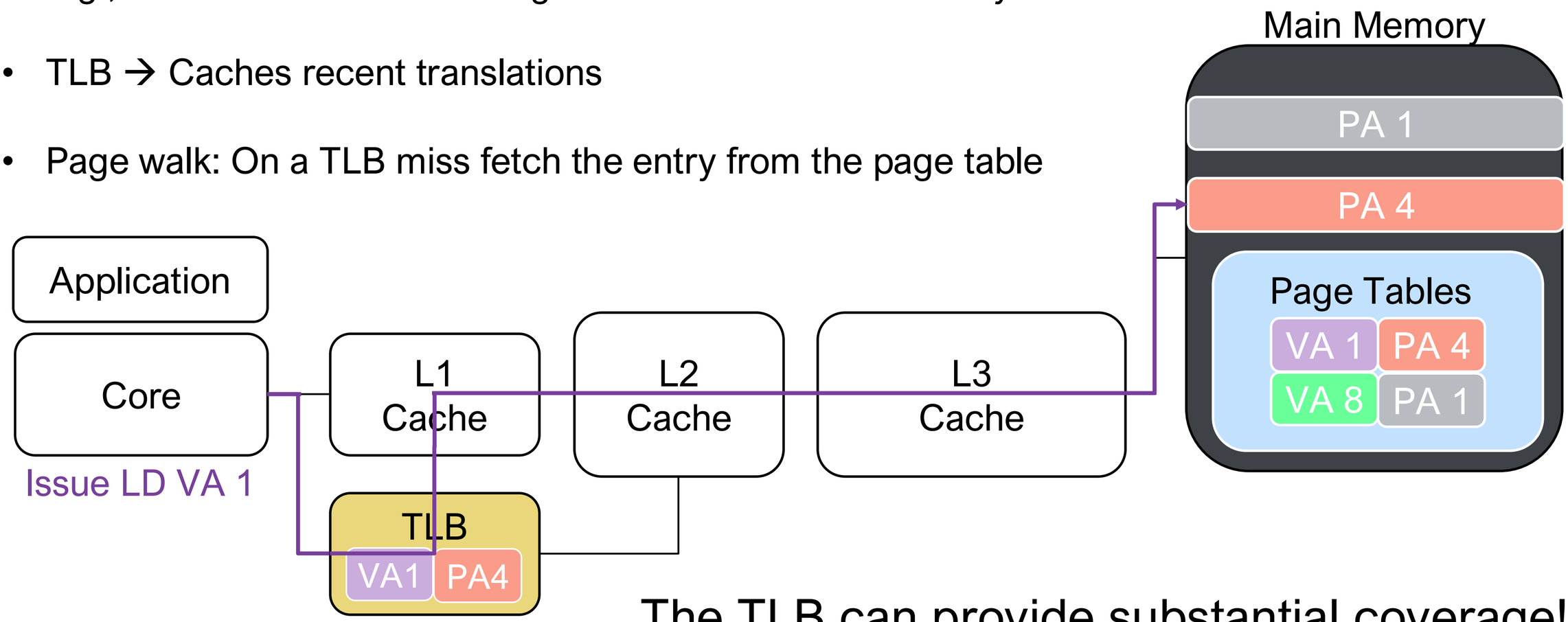
TLB Miss → "Page Walk" = Fetch entry from page table

TLB Hit

- Fetching each translation on a load would be expensive!
- How about locality? Translations cover pages but we operate on cache lines
- E.g., 4KB translations → Enough for 64 cache lines of 64 bytes

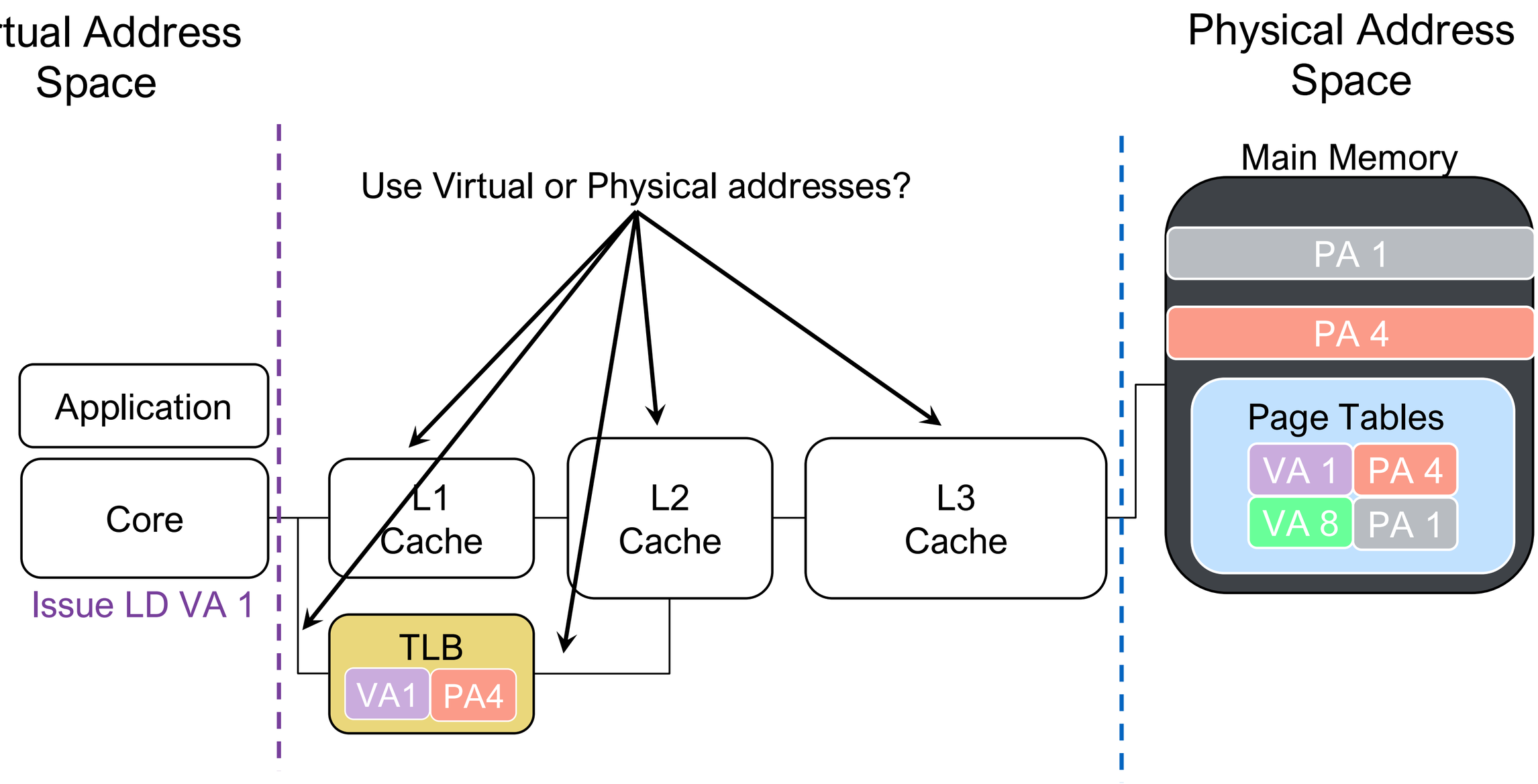
- TLB → Caches recent translations

- Page walk: On a TLB miss fetch the entry from the page table



The TLB can provide substantial coverage!
Both spatial and temporal locality at page granularity

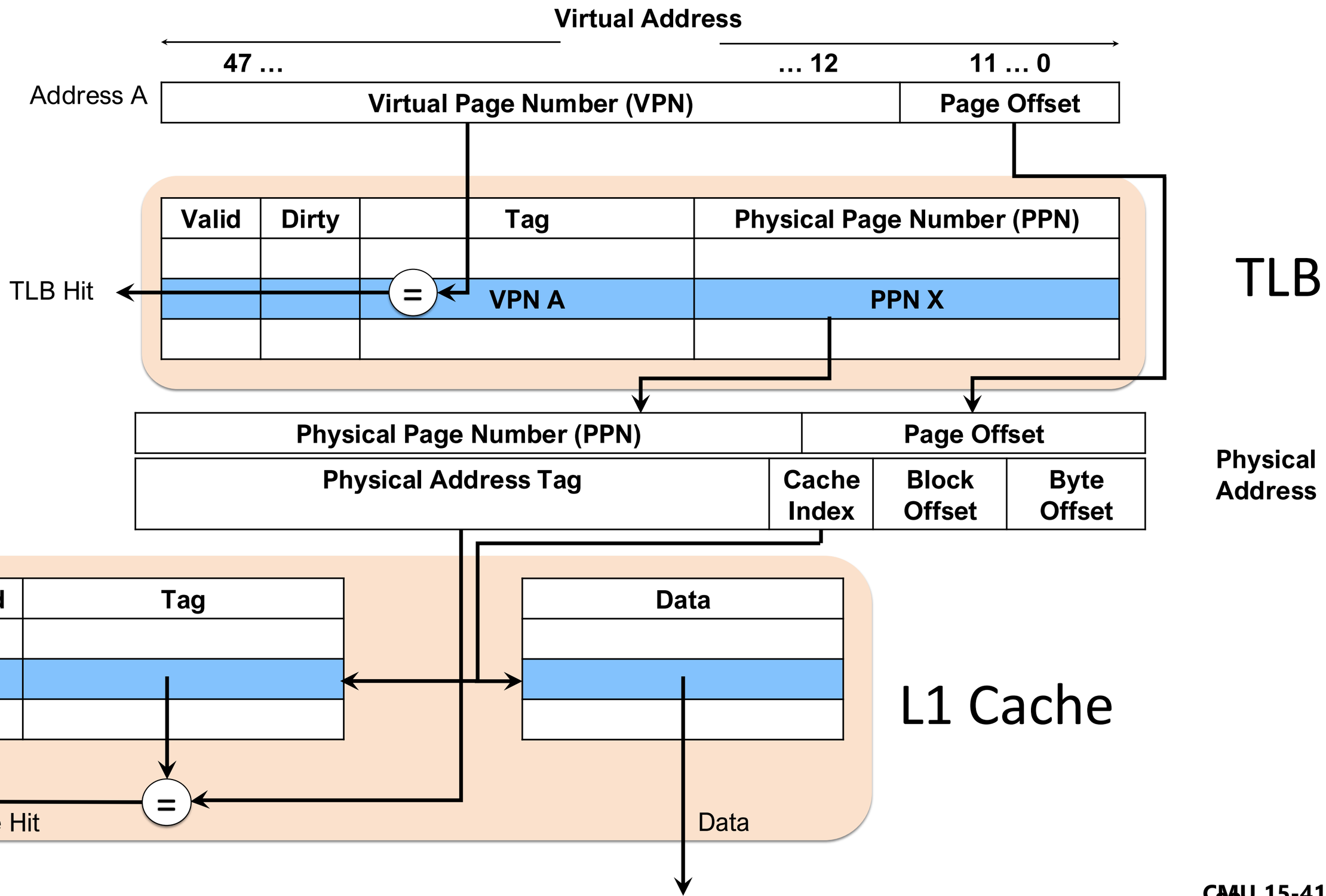
TLB Interaction with Caches



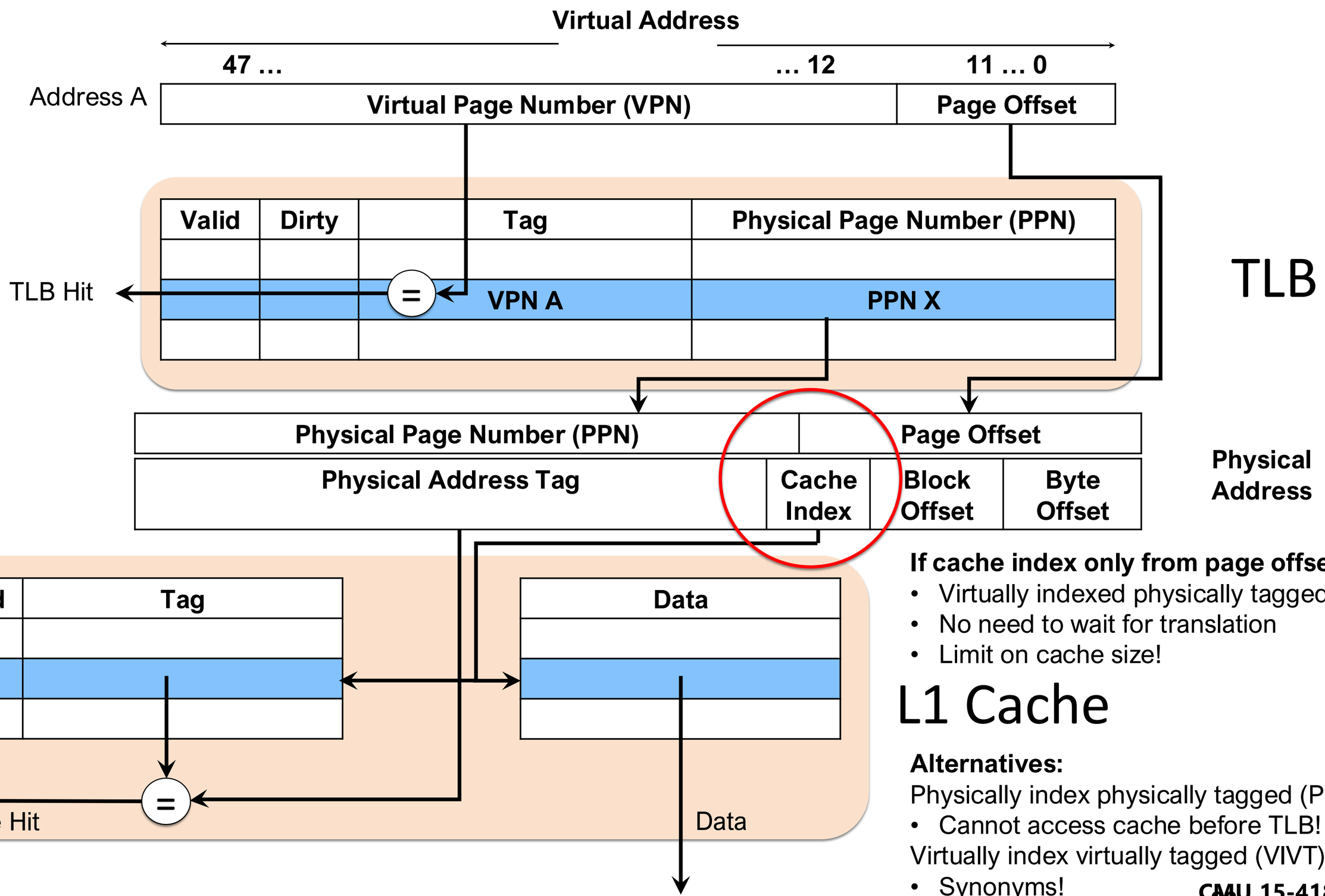
If cache uses physical addresses → Need to translate before cache lookup!

Alternative: use virtual address tag → Complications due to aliasing, need to track/flush conflicts

TLB Interaction with Caches



TLB Interaction with Caches



If cache index only from page offset

- Virtually indexed physically tagged (VIPT)
- No need to wait for translation
- Limit on cache size!

L1 Cache

Alternatives:

Physically index physically tagged (PIPT)

- Cannot access cache before TLB!

Virtually index virtually tagged (VIVT)

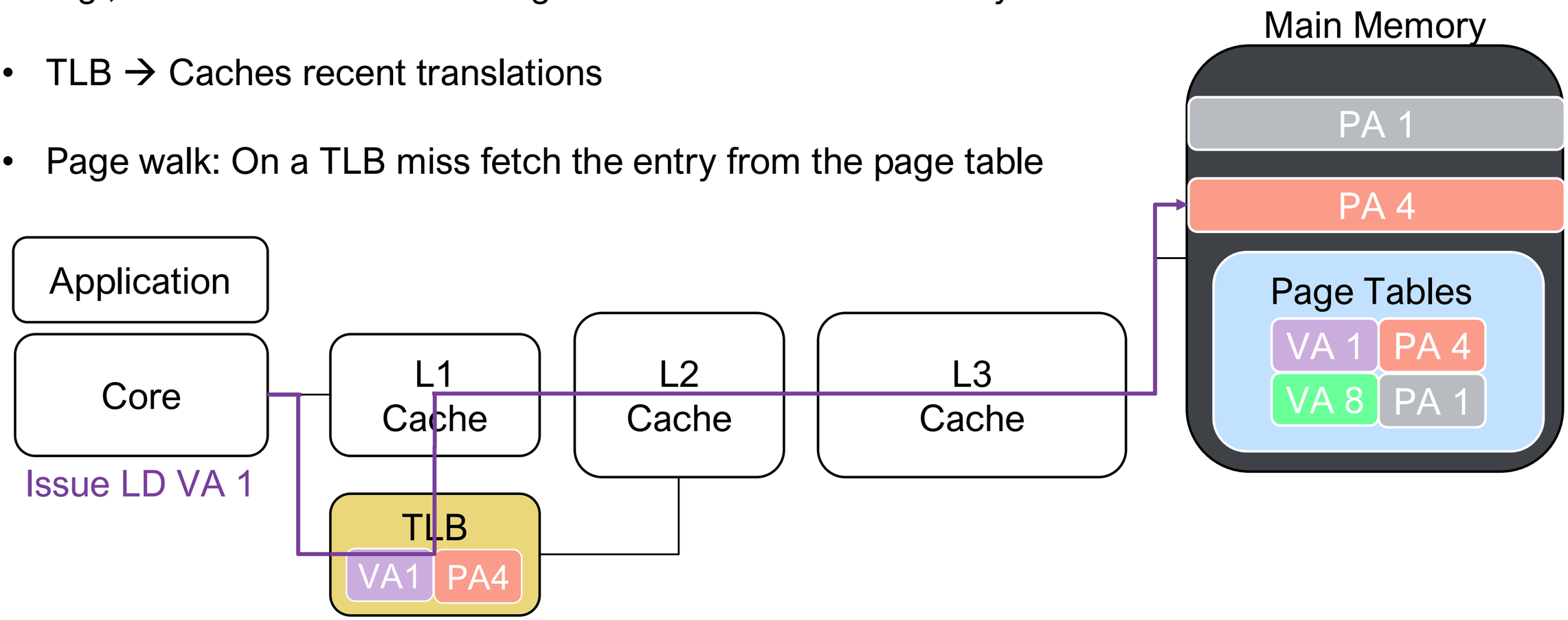
- Synonyms!

TLB Hit

- Fetching each translation on a load would be expensive!
- How about locality? Translations cover pages but we operate on cache lines
- E.g., 4KB translations → Enough for 64 cache lines of 64 bytes

- TLB → Caches recent translations

- Page walk: On a TLB miss fetch the entry from the page table

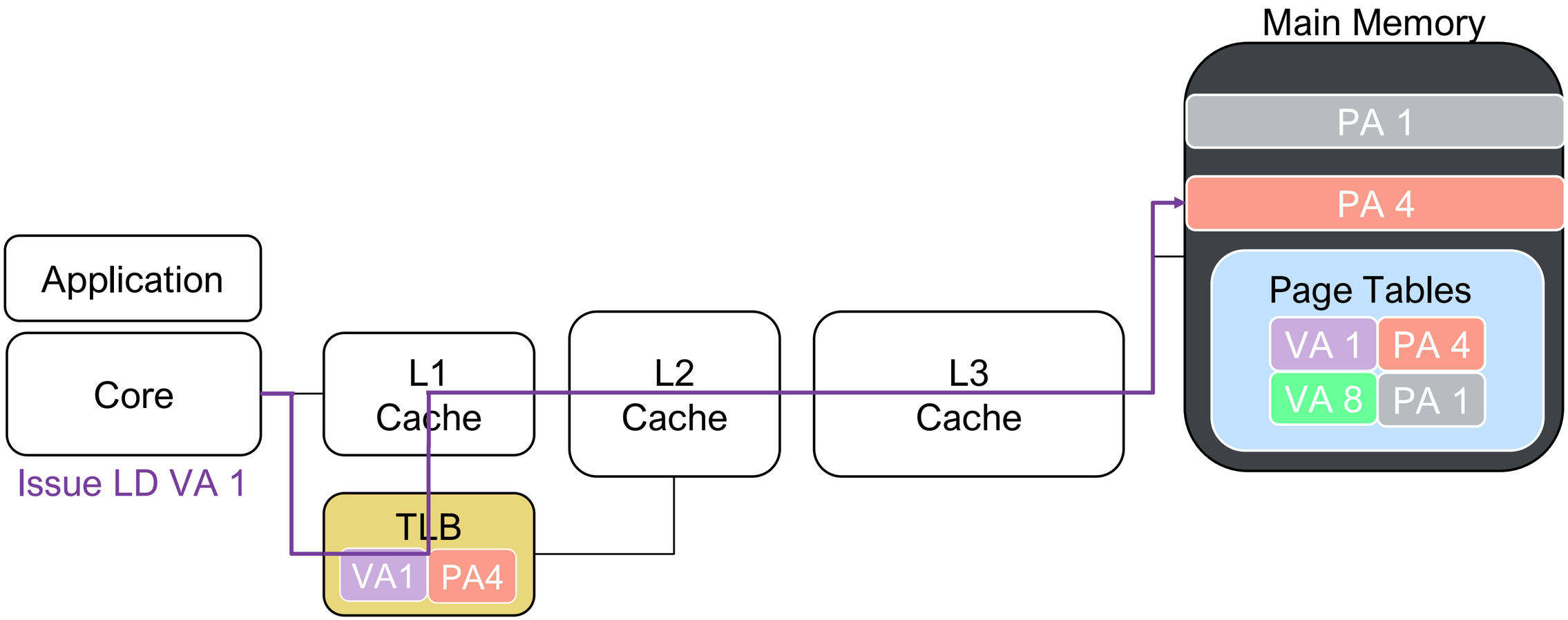


Page Tables and Translation Latency

- Page table structure of modern processors

- How to reduce the address translation latency?
  - Multi-level TLBs
  - Page Walk Caches (PWC)
  - Caching translation in data caches
  - Larger Translations

Multi-level Radix Page Tables

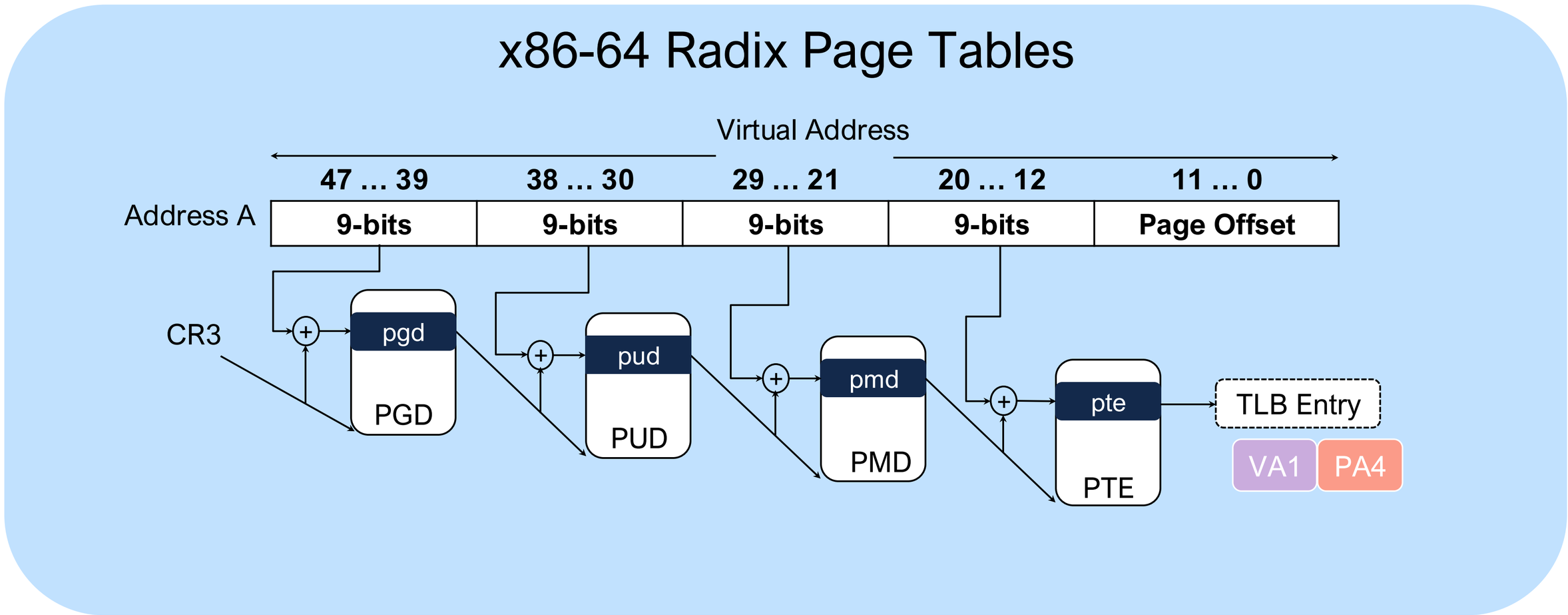


Multi-level Radix Page Tables

x86-64 Radix Page Tables

Multi-level Radix Page Tables

- Each page table is 4KB and each entry is 8 bytes → 512 entries per page
- We need 9 bits for index
- Base page size 4KB → Page offset 12 bits



PGD: Page Global Directory

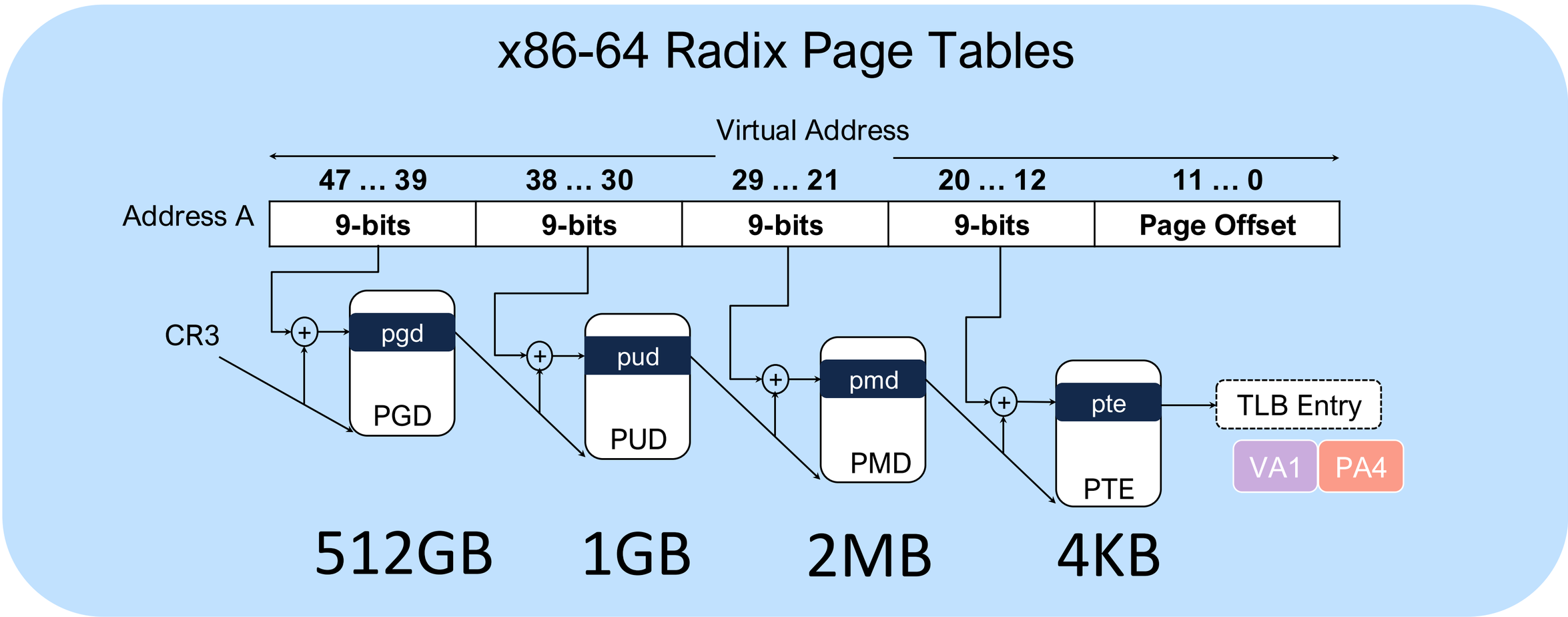
PGD: Page Upper Directory

PGD: Page Middle Directory

PTE: Page Table Entry

Multi-level Radix Page Tables

- Each page table is 4KB and each entry is 8 bytes → 512 entries per page
- We need 9 bits for index
- Base page size 4KB → Page offset 12 bits



Why? sparsely populated address space
(VA space >> PA space)

PGD: Page Global Directory

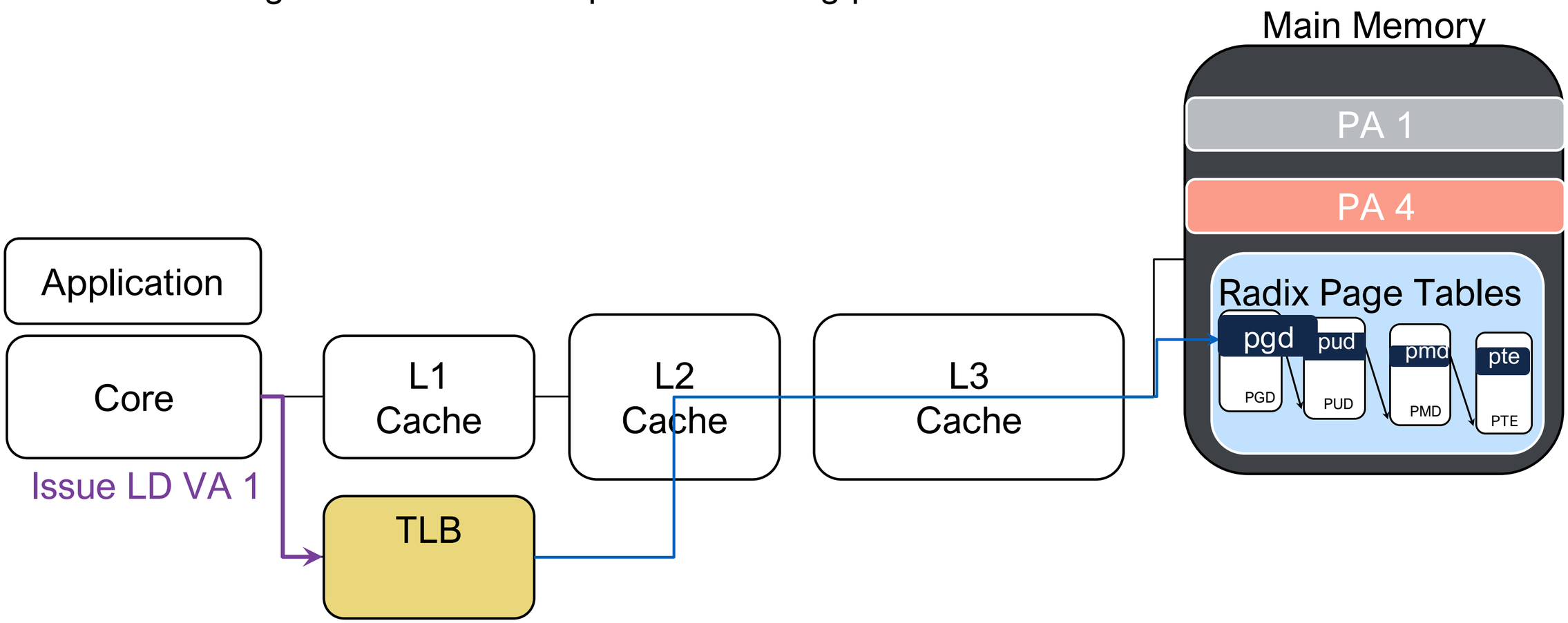
PGD: Page Upper Directory

PGD: Page Middle Directory

PTE: Page Table Entry

Multi-level Radix Page Tables

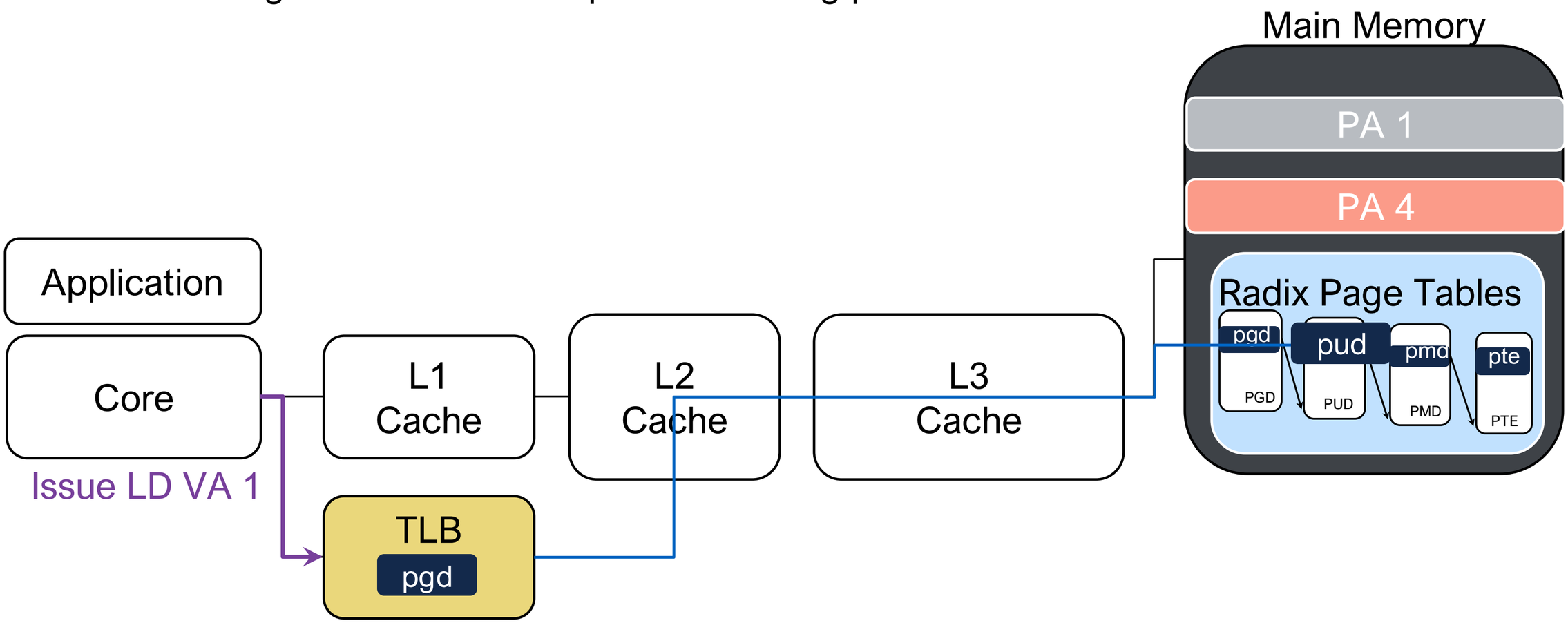
- Radix page tables: Tree hierarchy of page tables
- Benefit: Easy to support sparsely populated address space (VA space >> PA space)
- Drawback: Page walk becomes a pointer chasing process



TLB Miss → "Page Walk" = Fetch entry from radix page table

Multi-level Radix Page Tables

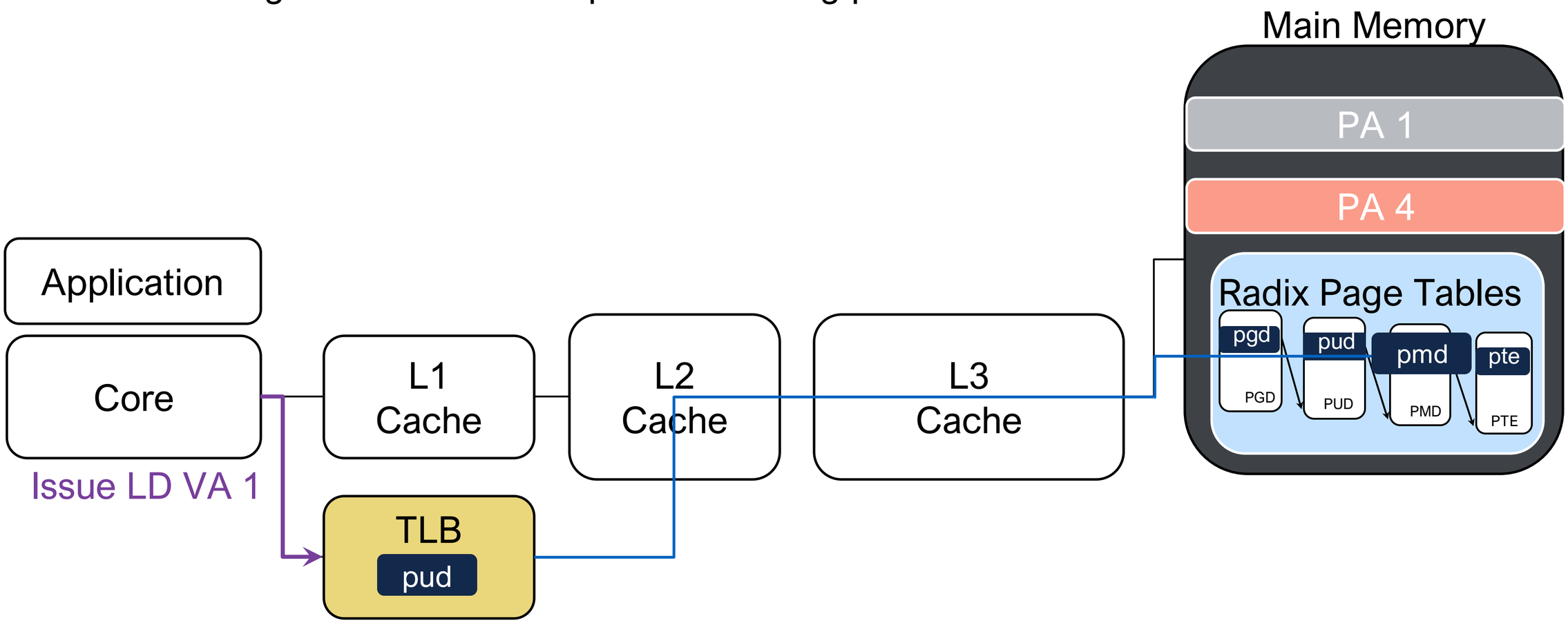
- Radix page tables: Tree hierarchy of page tables
- Benefit: Easy to support sparsely populated address space (VA space >> PA space)
- Drawback: Page walk becomes a pointer chasing process!



TLB Miss → "Page Walk" = Fetch entry from radix page table

Multi-level Radix Page Tables

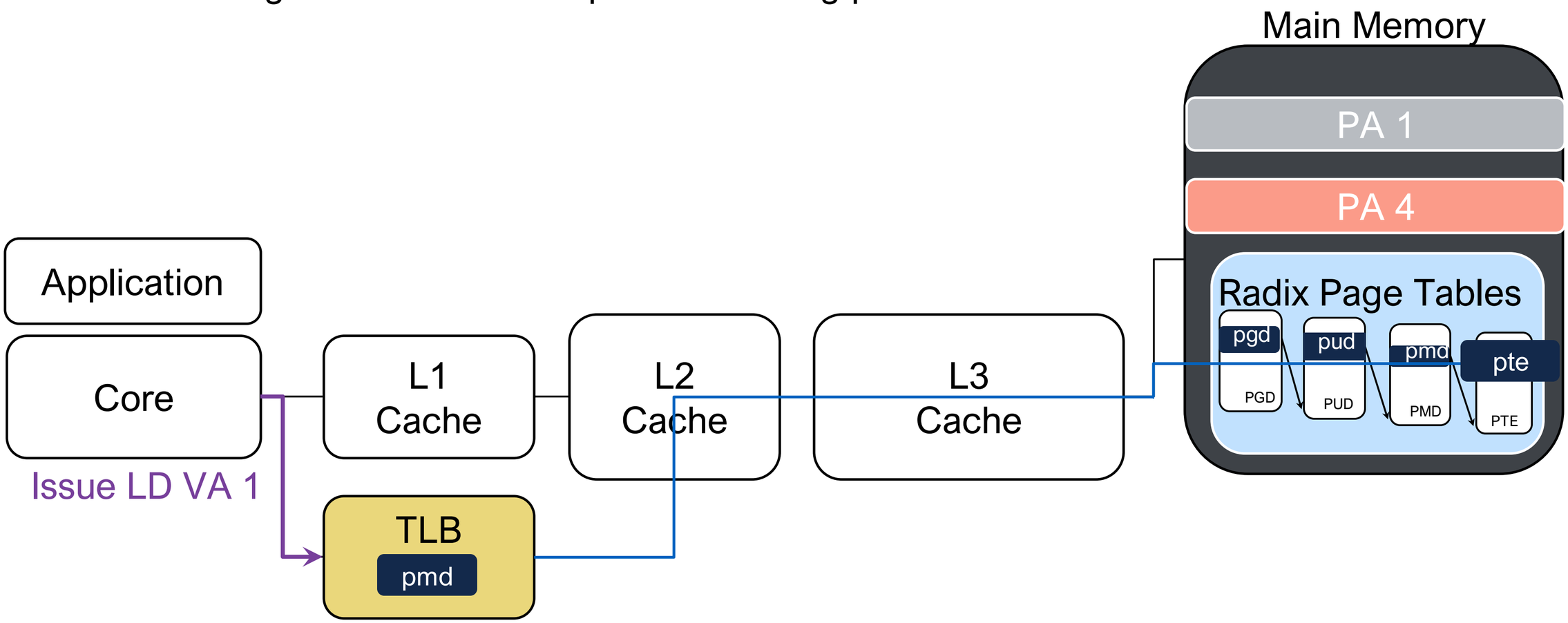
- Radix page tables: Tree hierarchy of page tables
- Benefit: Easy to support sparsely populated address space (VA space >> PA space)
- Drawback: Page walk becomes a pointer chasing process!



TLB Miss → "Page Walk" = Fetch entry from radix page table

Multi-level Radix Page Tables

- Radix page tables: Tree hierarchy of page tables
- Benefit: Easy to support sparsely populated address space (VA space >> PA space)
- Drawback: Page walk becomes a pointer chasing process!



TLB Miss → "Page Walk" = Fetch entry from radix page table

Reducing Translation Overhead

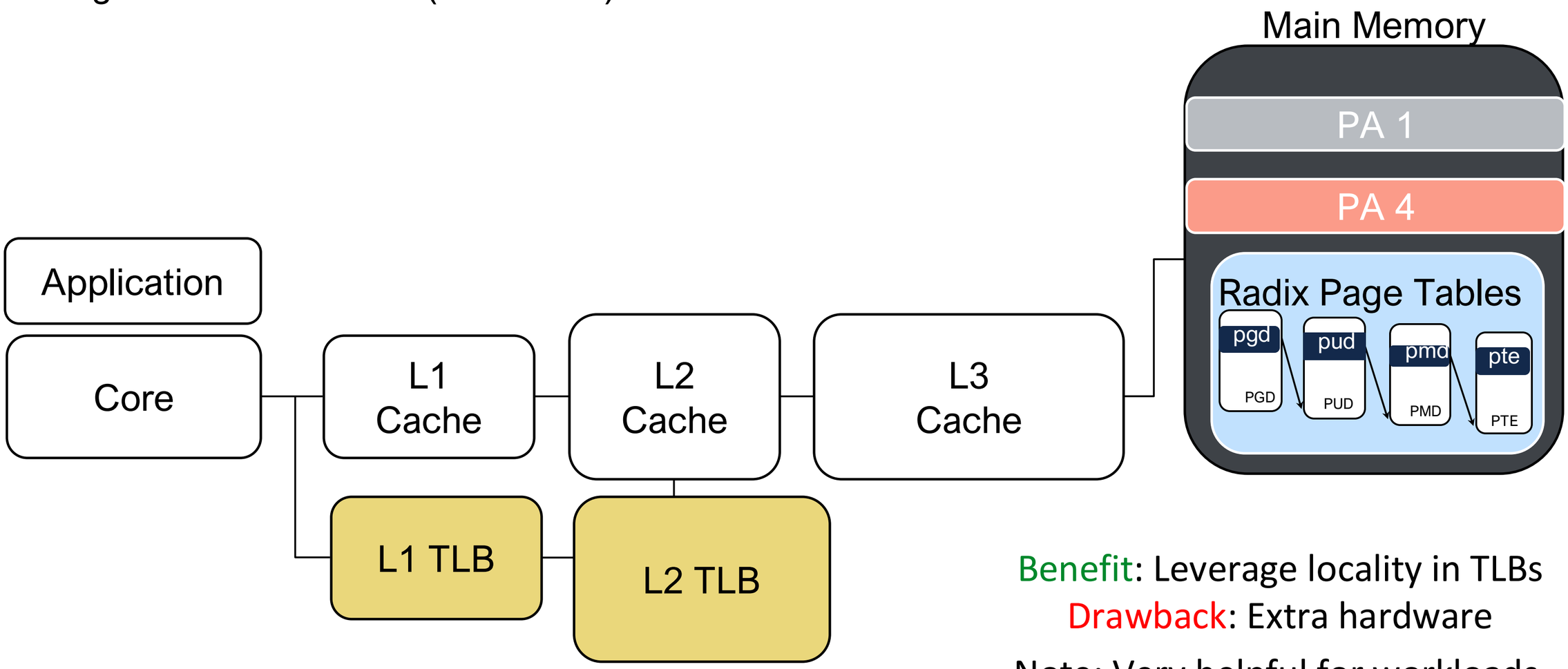
- How to reduce the address translation latency?

- Techniques we will cover next:
  - Multi-level TLBs
  - Page Walk Caches (PWC)
  - Caching translation in data caches
  - Larger Translations

Multilevel TLBs

How to avoid the cost of page walks?

- Separate Instruction and Data TLBs
- Larger TLBs → L2 TLB (mixed I+D)



Characteristic	Instruction TLB	Data DLB	Second-level TLB
Entries	128	64	1536
Associativity	8-way	4-way	12-way
Replacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Access latency	1 cycle	1 cycle	8 cycles
Miss	9 cycles	9 cycles	Hundreds of cycles to access page table

Intel i7 TLB structures

Benefit: Leverage locality in TLBs

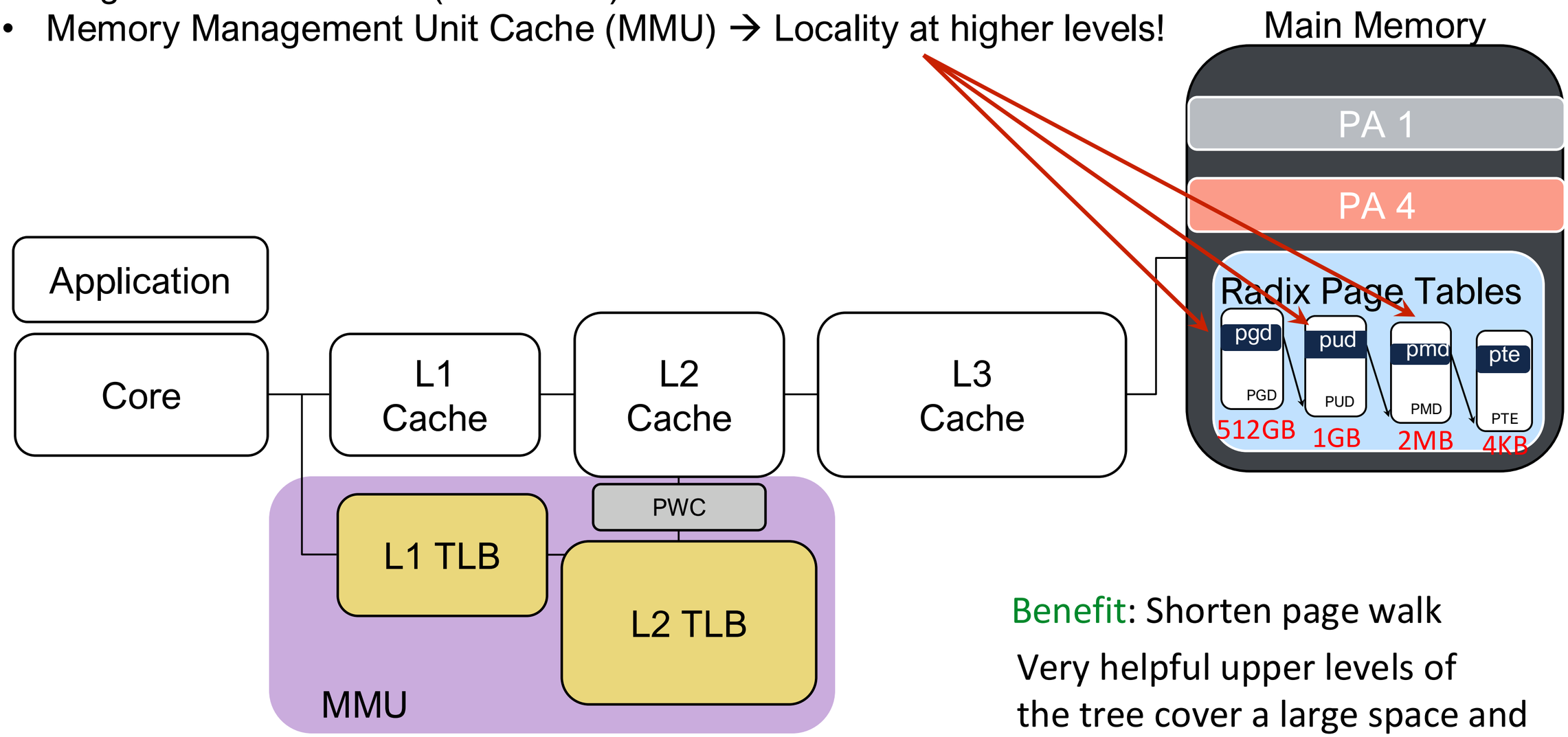
Drawback: Extra hardware

Note: Very helpful for workloads with access locality

Page Walk Cache

How to avoid the cost of page walks?

- Separate Instruction and Data TLBs
- Larger TLBs → L2 TLB (mixed I+D)
- Memory Management Unit Cache (MMU) → Locality at higher levels!



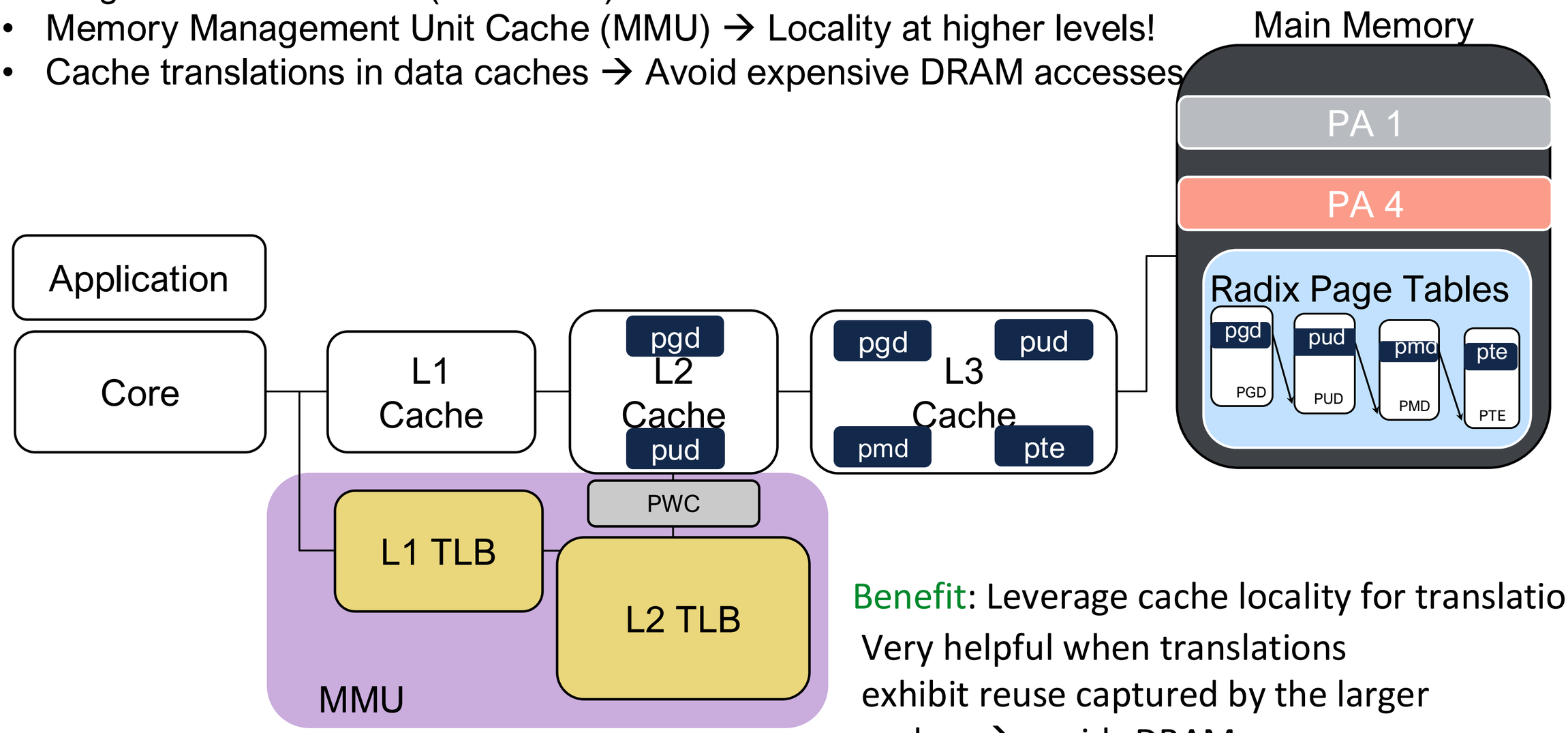
Benefit: Shorten page walk

Very helpful upper levels of the tree cover a large space and exhibit good locality

Translations in Data Caches

How to avoid the cost of page walks?

- Separate Instruction and Data TLBs
- Larger TLBs → L2 TLB (mixed I+D)
- Memory Management Unit Cache (MMU) → Locality at higher levels!
- Cache translations in data caches → Avoid expensive DRAM accesses



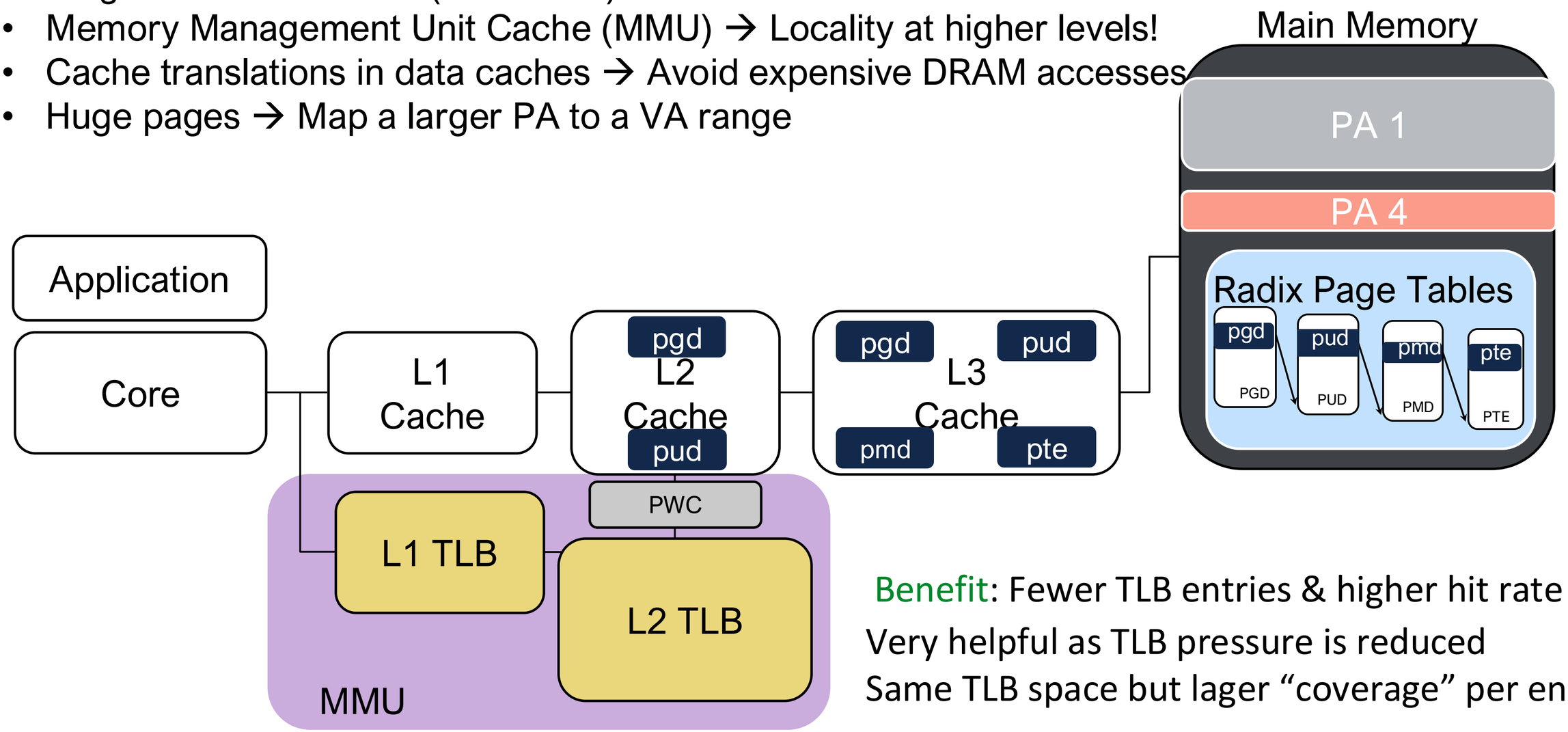
Benefit: Leverage cache locality for translations

Very helpful when translations exhibit reuse captured by the larger caches → avoids DRAM access.

Larger Translations (Huge Pages)

How to avoid the cost of page walks?

- Separate Instruction and Data TLBs
- Larger TLBs → L2 TLB (mixed I+D)
- Memory Management Unit Cache (MMU) → Locality at higher levels!
- Cache translations in data caches → Avoid expensive DRAM accesses
- Huge pages → Map a larger PA to a VA range

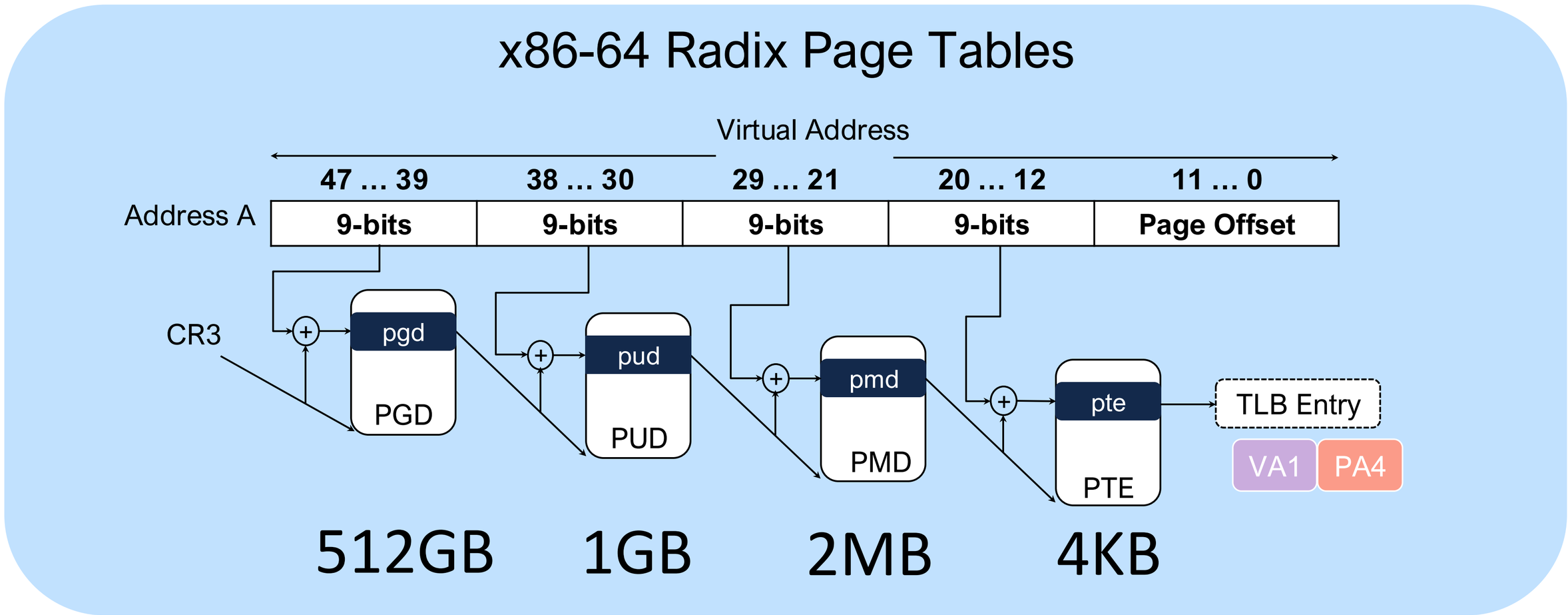


Benefit: Fewer TLB entries & higher hit rate

Very helpful as TLB pressure is reduced

Same TLB space but lager "coverage" per entry

Huge Pages with Radix Page Tables



Translation Coherence

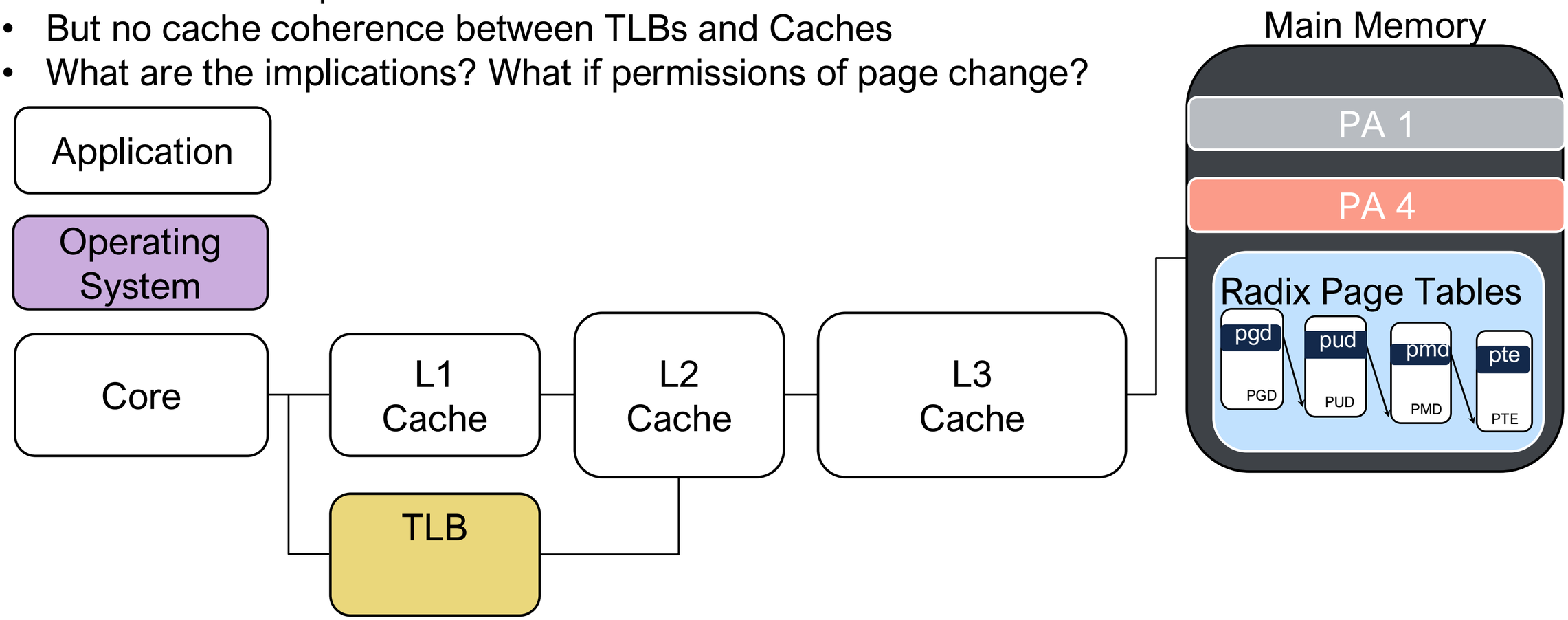
- OS or HW responsible for:
  - Loading the TLB with entries?
  - Removing entries from the TLB?
  - Create/update page tables?

- How to maintain coherence across TLBs?

Hardware Responsibilities

Hardware is responsible for:

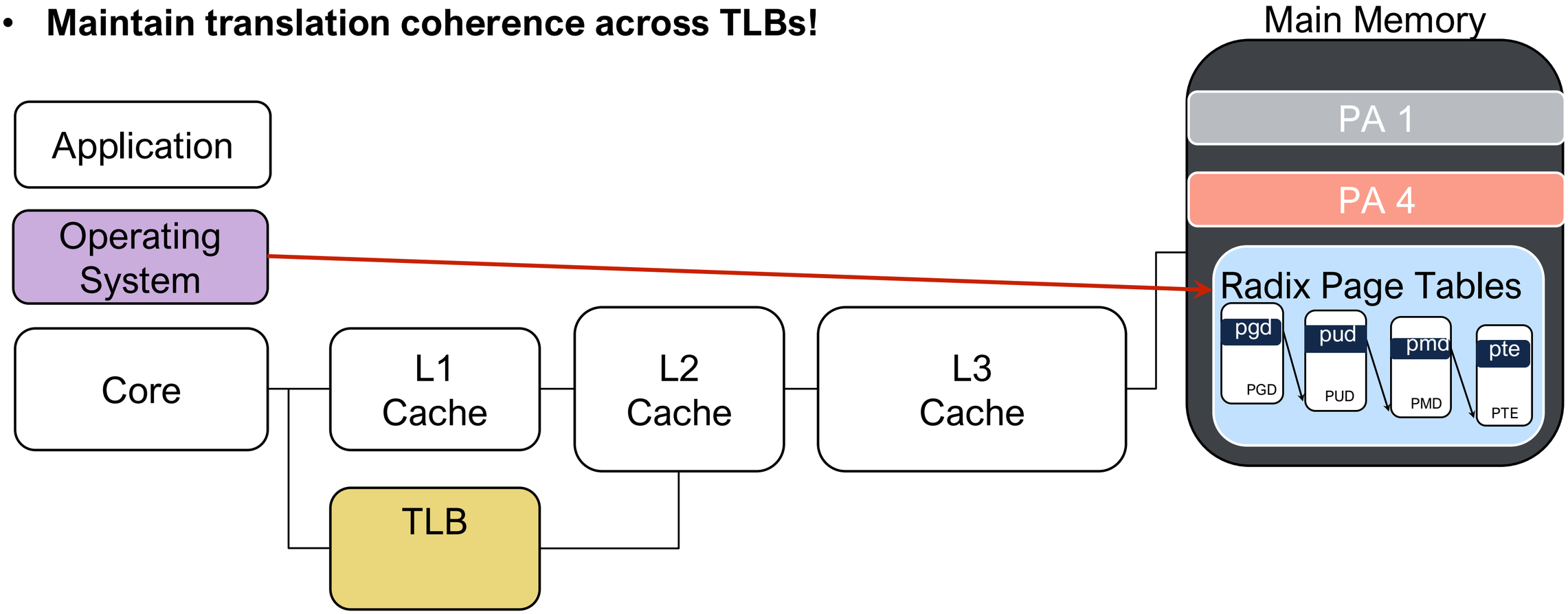
- Perform page walks and locate translations → Raise exception on error (e.g., Page fault)
- Load TLB and replace TLB entries due to conflicts
- But no cache coherence between TLBs and Caches
- What are the implications? What if permissions of page change?



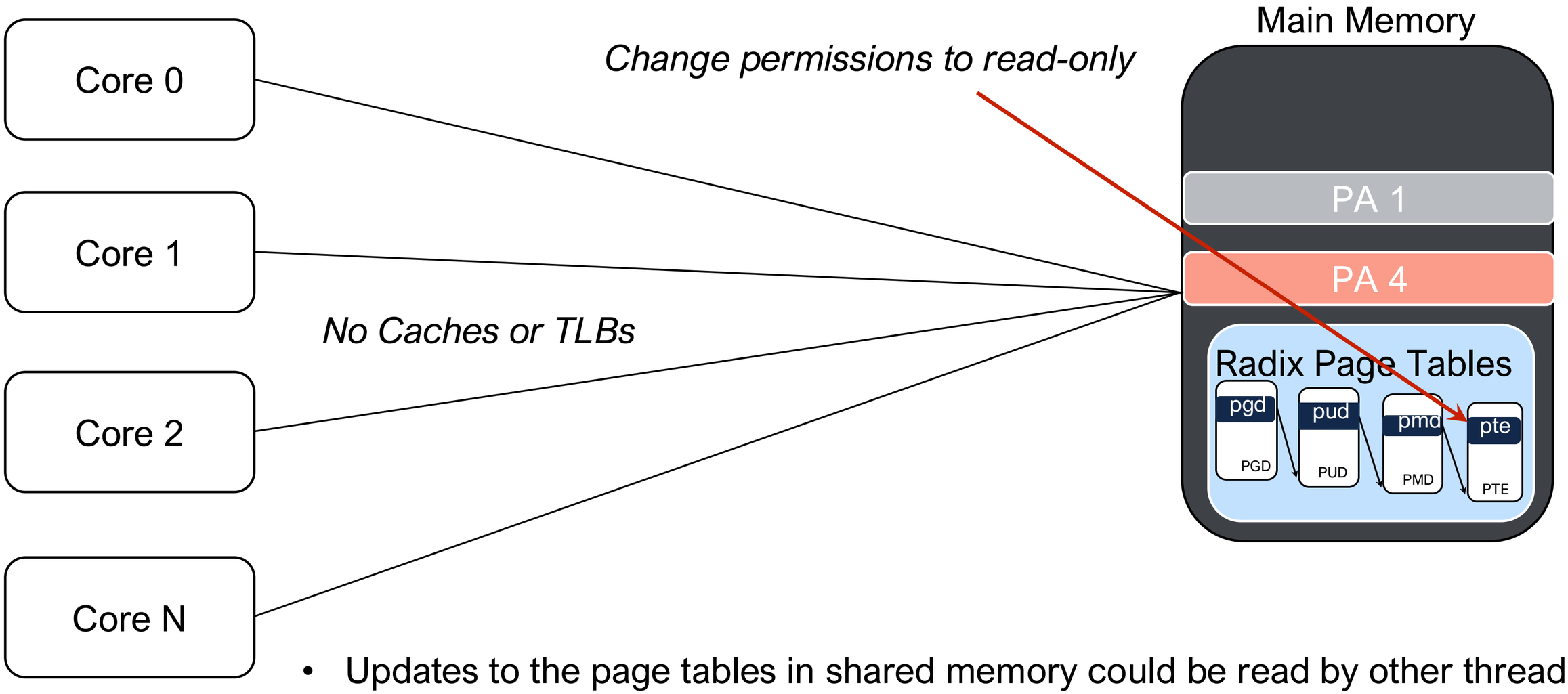
OS Responsibilities

OS is responsible for:

- Creating page tables entries → Mapping of memory to the address space
- Update page tables entries → When? Permissions, page movement, updates, paging…
- **Maintain translation coherence across TLBs!**

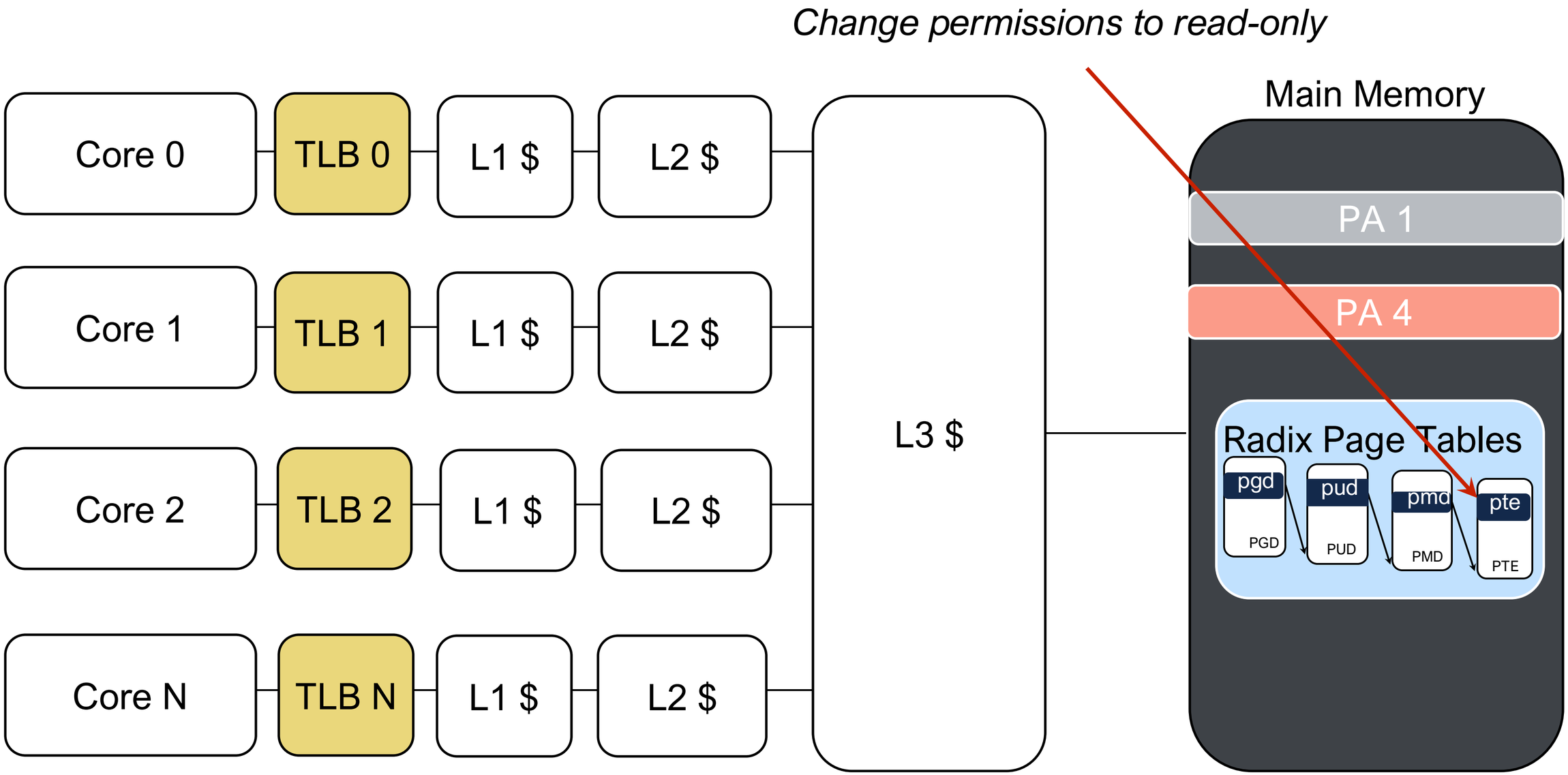


How Propagate PT Changes?



- Updates to the page tables in shared memory could be read by other threads
- But this would be a very slow system! Why?

How Propagate PT Changes?

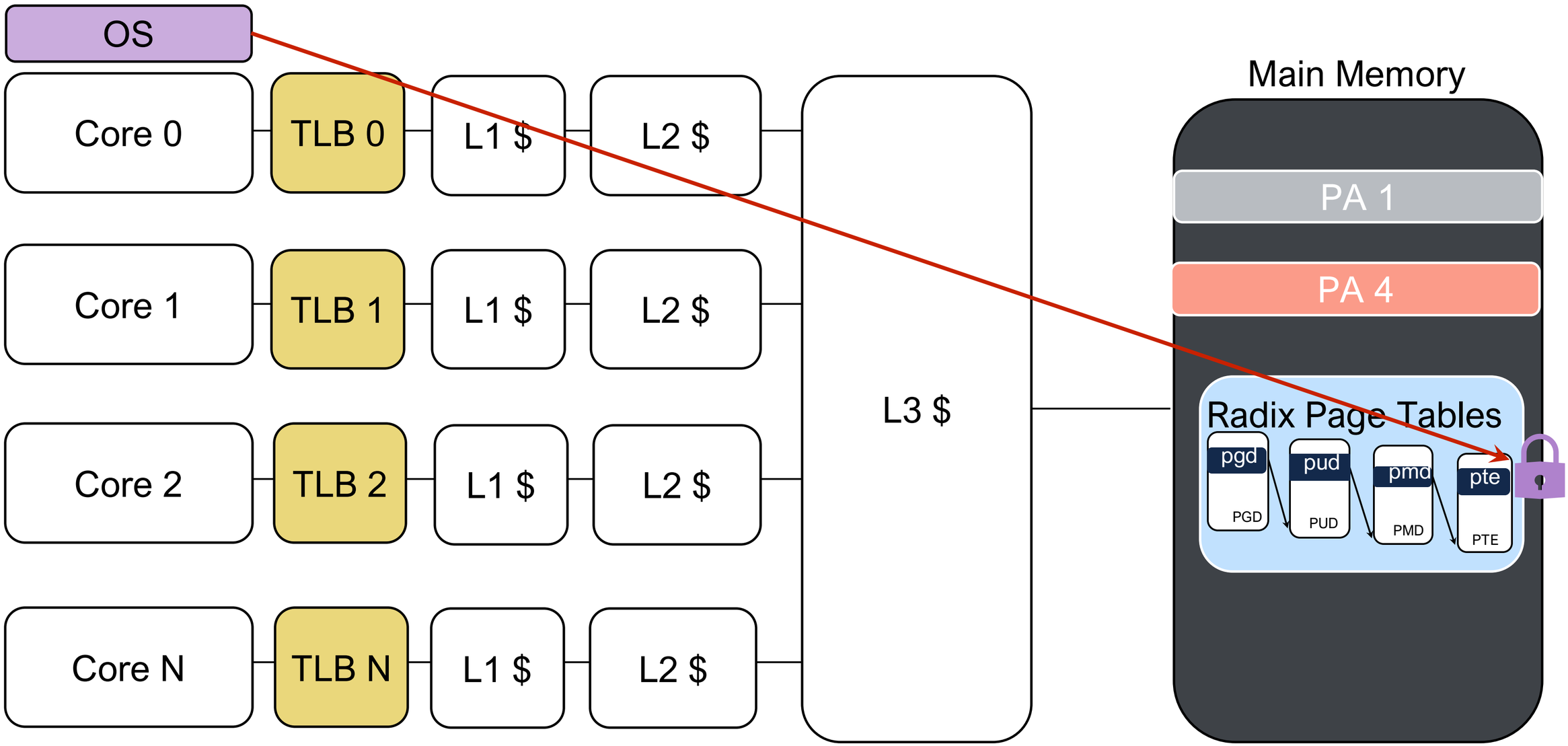


- TLB Shootdown:
  - OS intervenes and flushes relevant entries from the TLBs!

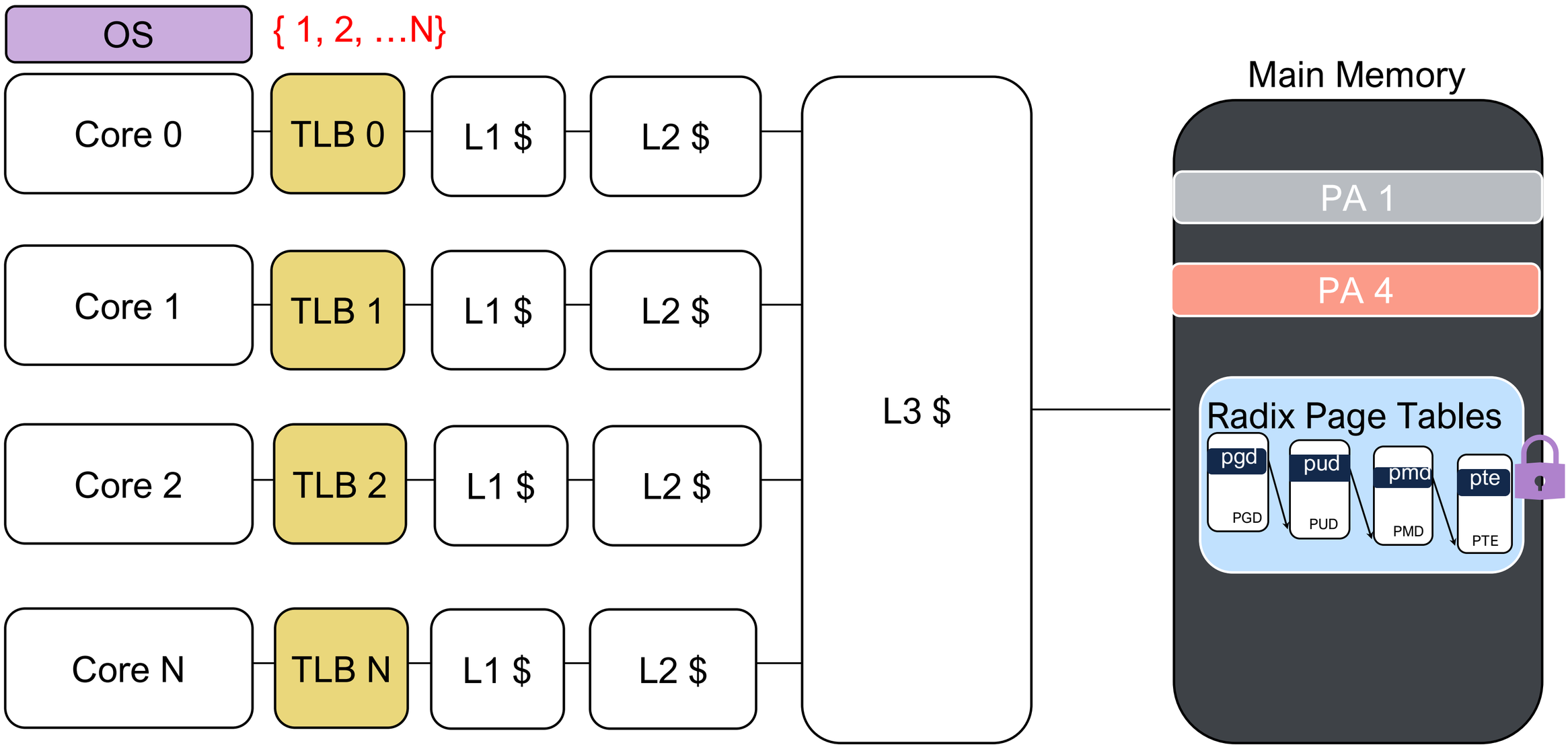
TLB Shootdown Steps

1. OS on initiating core locks the page table entry (PTE)
2. OS generates list of cores that may be using this PTE
3. Initiating core sends Inter-Processor Interrupt (IPI) to other cores
   - Requesting that they invalidate their corresponding TLB entry
4. Initiating core invalidates local TLB entry; waits for acks
5. Other cores receive interrupts; execute interrupt handler
   - Invalidate TLBs
   - Send an ack back to the initiating core
6. Once initiating core receives all acks → unlocks PTE

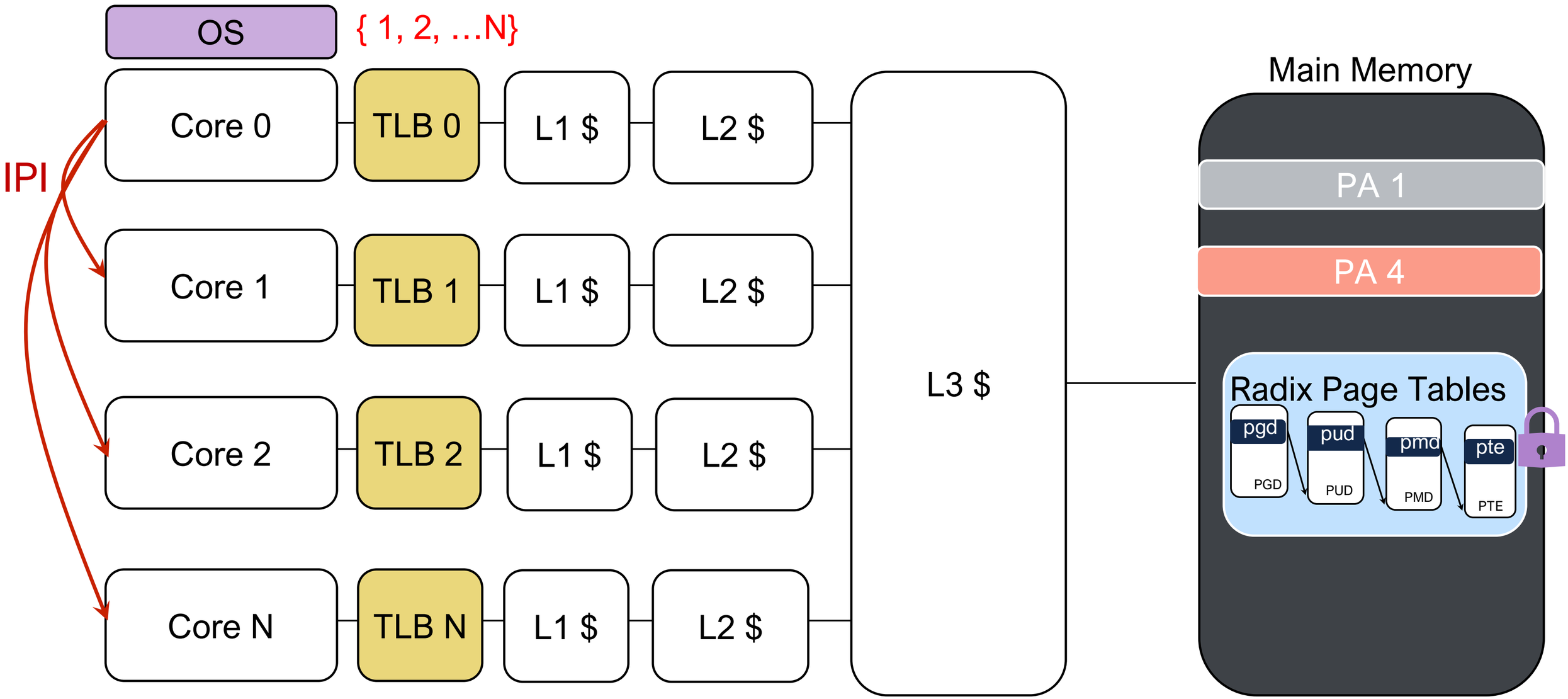
1. Lock Page Table Entry



2. Generate list of cores

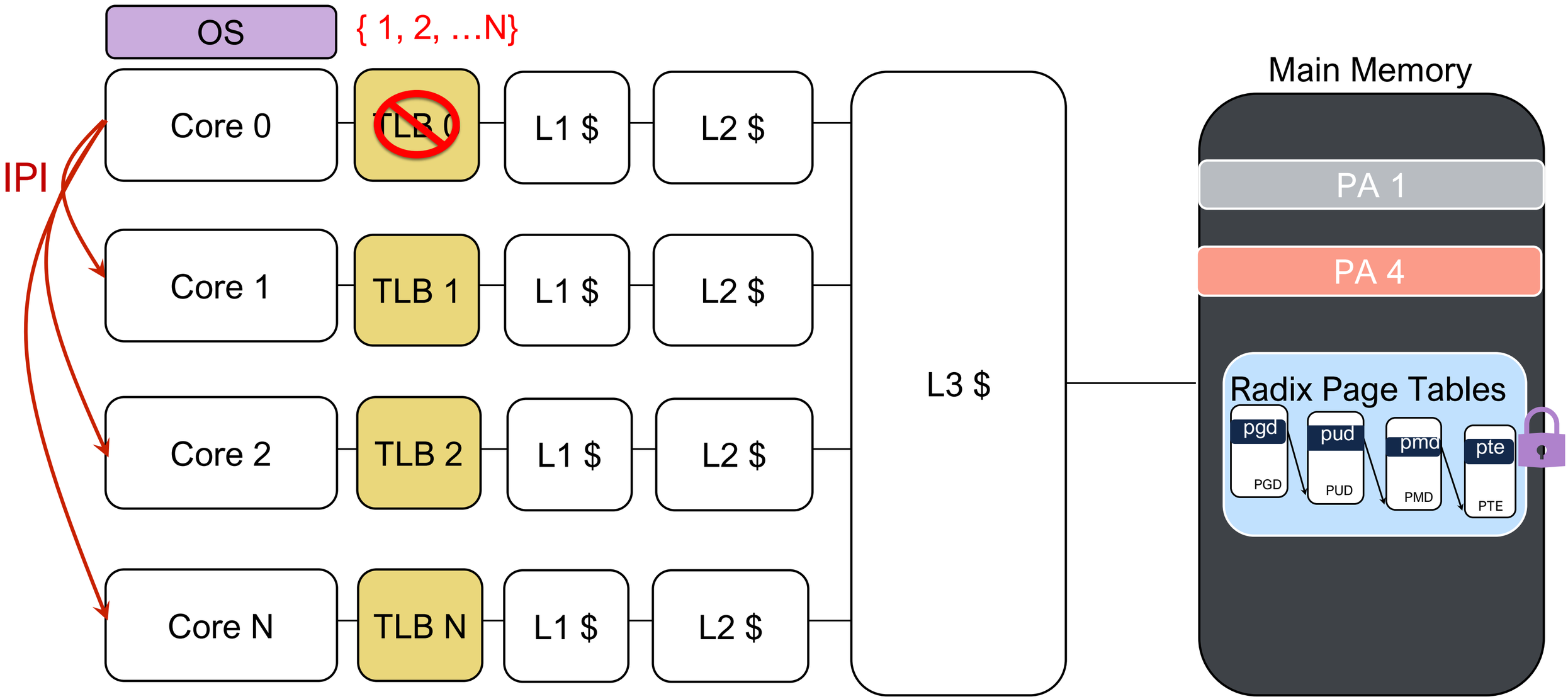


3. Send IPIs to Other Cores

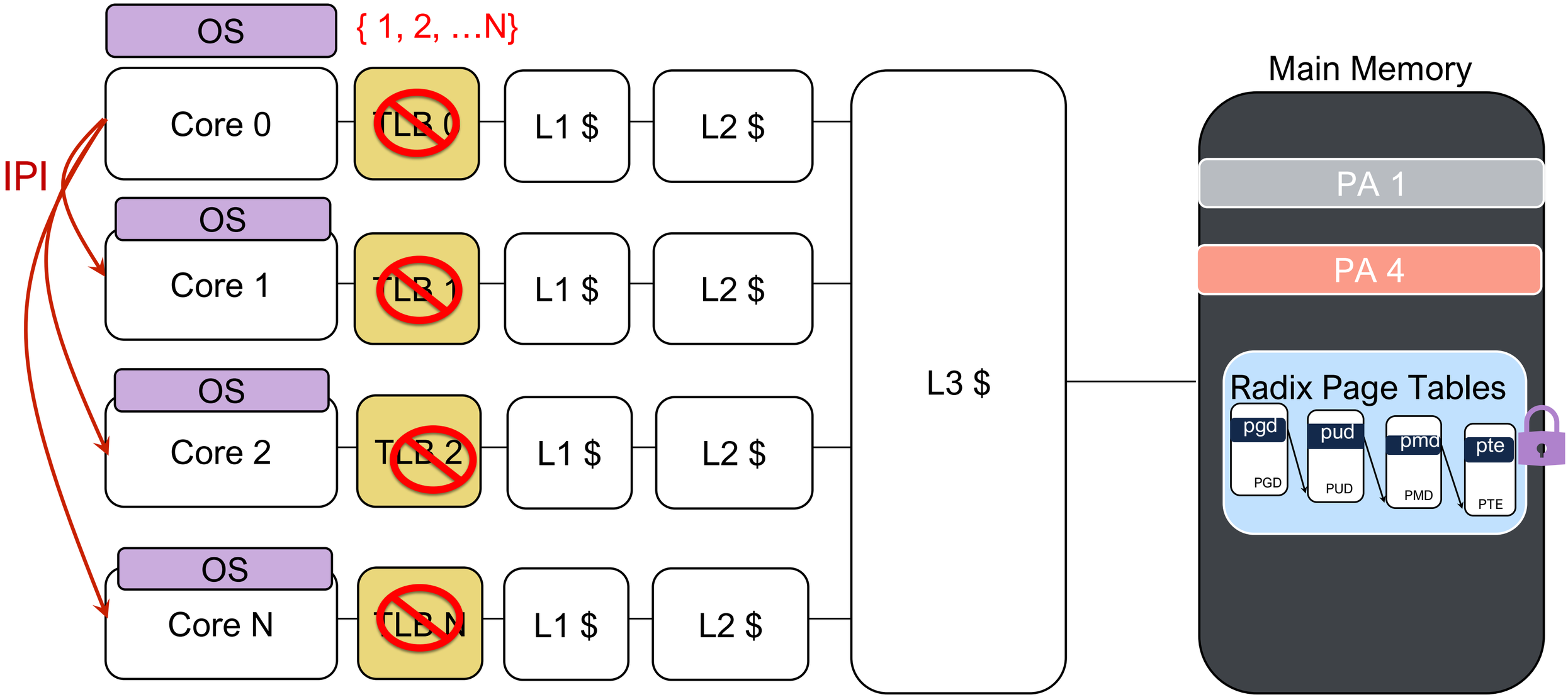


IPI: Inter-Processor Interrupts

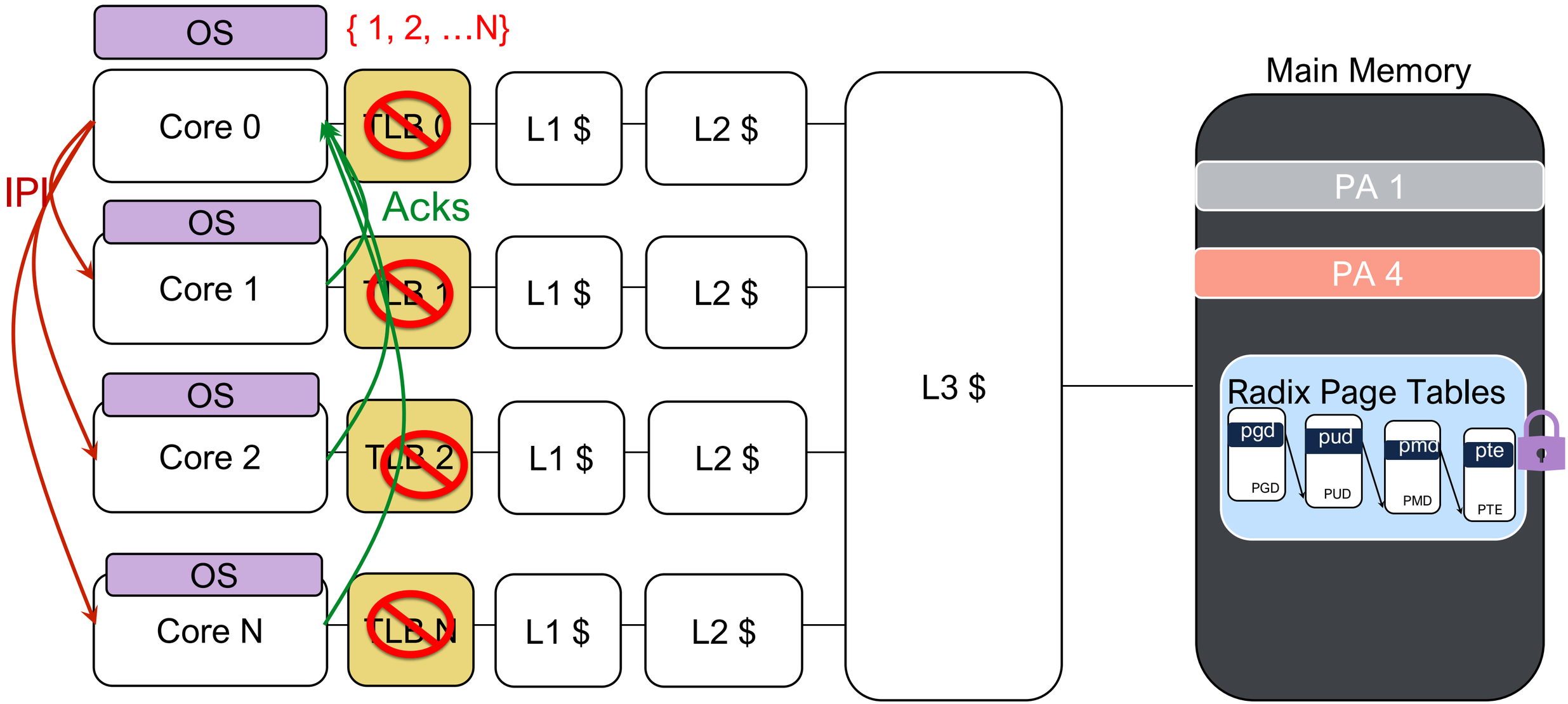
4. Invalidate local TLB



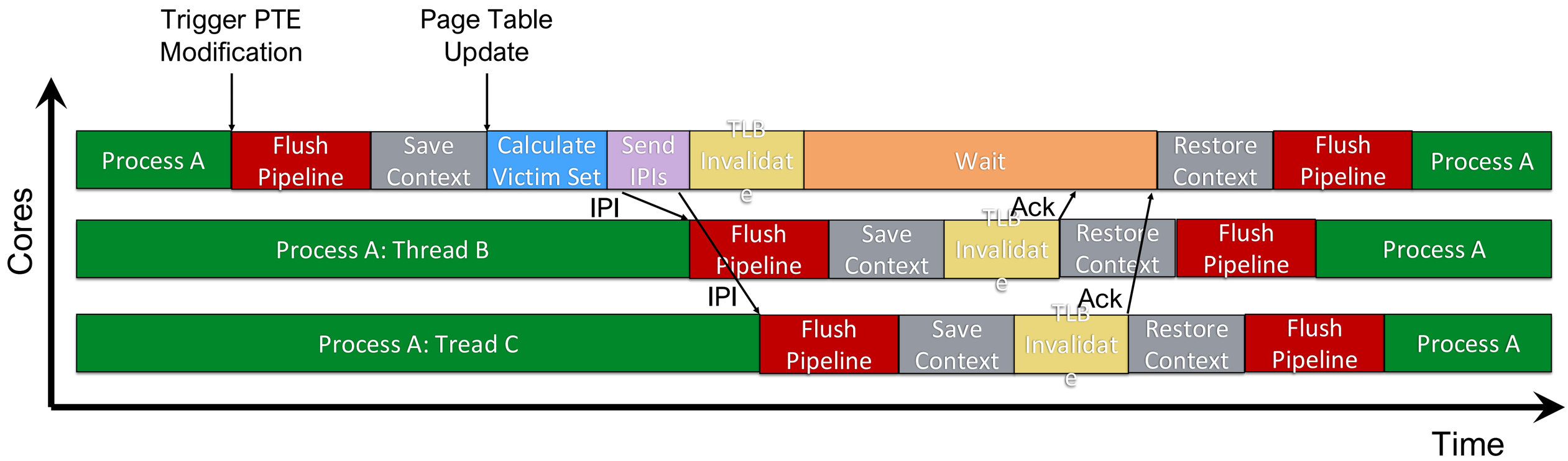
5. Interrupt Handler Invalidate TLBs



6. Receive all acks and unlock PTE

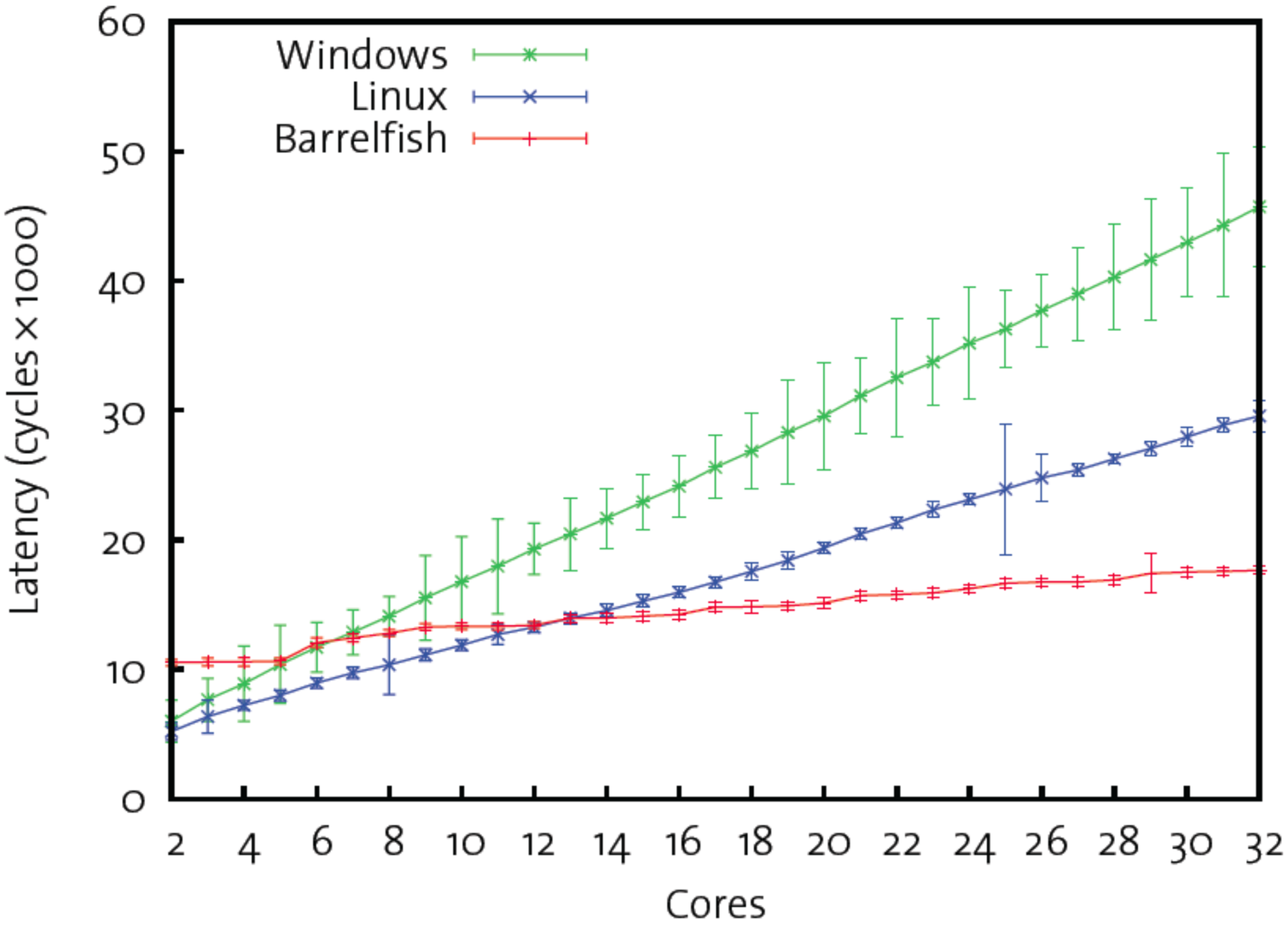


Timeline of TLB Shootdown



Substantial performance impact on multithreaded applications

Performance of TLB Shootdown



- Expensive operation
  - e.g., over 10,000 cycles on 8 or more cores
- Gets more expensive with increasing number of cores

Image from: Baumann *et al*,"The Multikernel: a New OS Architecture for Scalable Multicore Systems." In Proceedings of SOSP '09.

Further readin

https://dl.acm.org/doi/pdf/10.1145/3

ASPLOS II - 1987

Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures

Richard Rashid, Avadis Tevanian, Michael Young, David Golub,
Robert Baron, David Black, William Bolosky, and Jonathan Chew

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

This paper describes the design and implementation of virtual memory management within the CMU Mach Operating System and the experiences gained by the Mach kernel group in porting that system to a variety of architectures. As of this writing, Mach runs on more than half a dozen uniprocessors and multiprocessors including the VAX family of uniprocessors and multiprocessors, the IBM RT PC, the SUN 3, the Encore MultiMax, the Sequent Balance 21000 and several experimental computers. Although these systems vary considerably in the kind of hardware support for memory management they provide, the machine-dependent portion of Mach virtual memory consists of a single code module and its related header file. This separation of software memory management from hardware support has been accomplished without sacrificing system performance. In addition to improving portability, it makes possible a relatively unbiased examination of the pros and cons of various hardware memory management schemes, especially as they apply to the support of multiprocessors.

1. Introduction

While software designers are increasingly able to cope with variations in instruction set architectures, operating system portability continues to suffer from a proliferation of memory structures. UNIX systems have traditionally addressed the problem of VM portability by restricting the facilities provided and basing implementations for new memory management architectures on versions already done for previous systems. As a result, existing versions of UNIX, such as Berkeley 4.3bsd, offer little in the way of virtual memory management other than simple paging support. Versions of Berkeley UNIX on non-VAX hardware, such as SunOS on the SUN 3 and ACIS 4.2 on the IBM RT PC, actually simulate internally the VAX memory mapping architecture -- in effect treating it as a machine-independent memory management specification.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864, monitored by the Space and Naval Warfare Systems Command under contract N00039-85-C-1034.

Over the last two years CMU has been engaged in the development of a portable, multiprocessor operating system called Mach. One of the goals of Mach has been to explore the relationship between hardware and software memory architectures and to design a memory management system that would be readily portable to multiprocessor computing engines as well as traditional uniprocessors.

Mach provides complete UNIX 4.3bsd compatibility while significantly extending UNIX notions of virtual memory management and interprocess communication [1]. Mach supports:

- *large, sparse virtual address spaces,*
- *copy-on-write virtual copy operations,*
- *copy-on-write and read-write memory sharing between tasks,*
- *memory mapped files and*
- *user-provided backing store objects and pagers.*

This has been accomplished without patterning Mach's internal memory representation after any specific architecture. In fact, Mach makes relatively few assumptions about available memory management hardware. The primary requirement is an ability to handle and recover from page faults (for some arbitrary page size).

As of this writing, Mach runs on more than half a dozen uniprocessors and multiprocessors including the entire VAX family of uniprocessors and multiprocessors, the IBM RT PC, the SUN 3, the Encore MultiMax and the Sequent Balance 21000. Implementations are in progress for several experimental computers. Despite differences between supported architectures, the machine-dependent portion of Mach's virtual memory subsystem consists of a single code module and its related header file. All information important to the management of Mach virtual memory is maintained in machine-independent data structures and machine-dependent data structures contain only those mappings necessary to running the current mix of programs.

Mach's separation of software memory management from hardware support has been accomplished without sacrificing system performance. In several cases overall system performance has measurably improved over existing UNIX implementations. Moreover, this approach makes possible a

From Virtual Memory to Virtual Machines

Supports isolation and security

Sharing hardware among many unrelated users

Enabled by raw speed of processors, making the overhead more acceptable

Allows different ISAs and OS to be presented to user programs

- **“System Virtual Machines”**
- **SVM software is called “virtual machine monitor” or “hypervisor”**
- **Individual virtual machines run under the monitor are called “guest VMs”**

VMM Requirements

Guest software should:

- **Behave on as if running on native hardware**
- **Not be able to change allocation of real system resources**

VMM should be able to “context switch” guests

Hardware must allow:

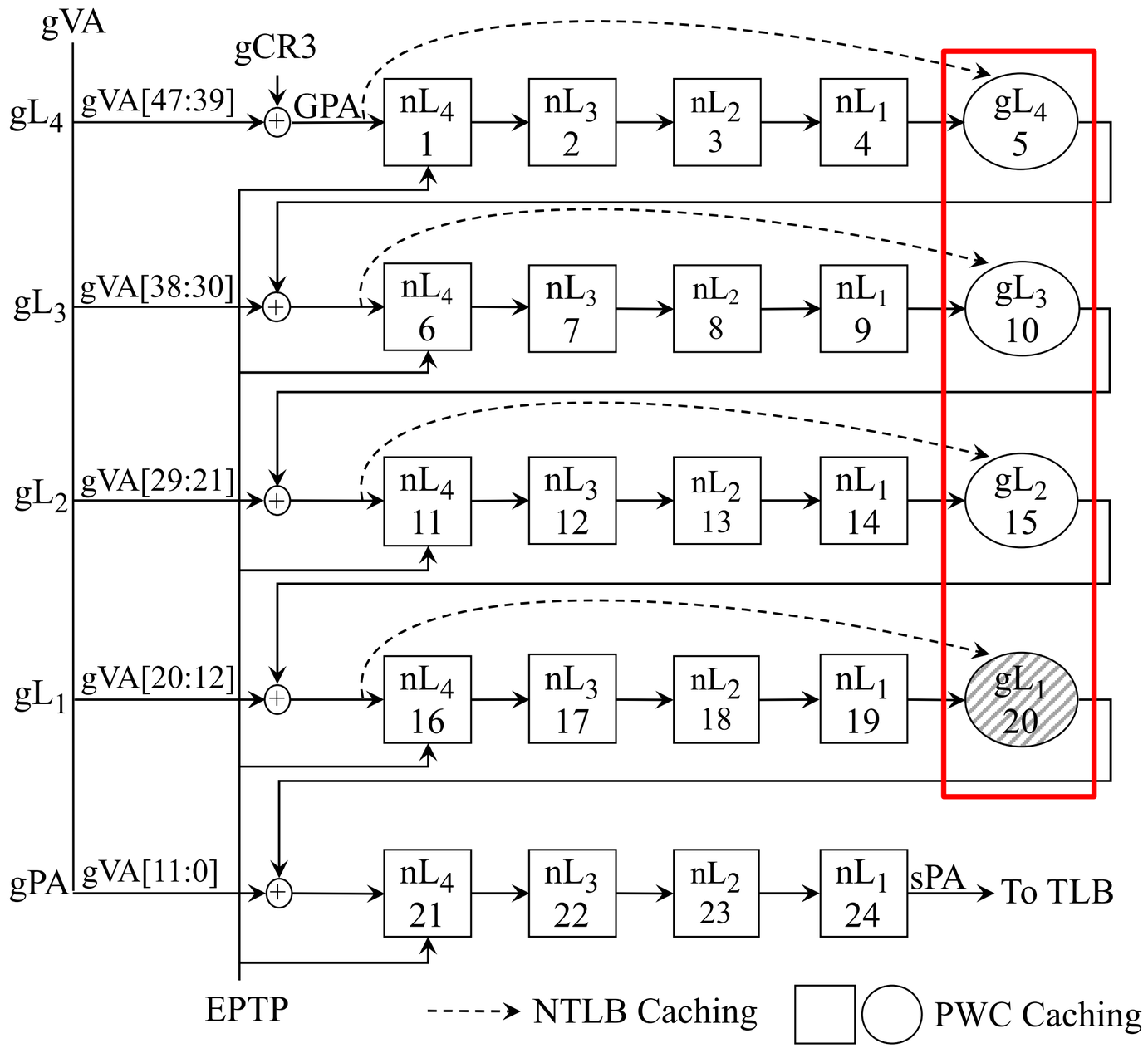
- **System and use processor modes**
- **Privileged subset of instructions for allocating system resources**

Impact of VMs on Virtual Memory

Each guest OS maintains its own set of page tables

- VMM adds a level of memory between physical and virtual memory called “real memory”
- VMM maintains shadow page table that maps guest virtual addresses to physical addresses
  - Requires VMM to detect guest’s changes to its own page table
  - Occurs naturally if accessing the page table pointer is a privileged operation

Impact of VMs on Virtual Memory



Virtualization Extensions

Objectives:

- **Avoid flushing TLB**
- **Use nested page tables instead of shadow page tables**
- **Allow devices to use DMA to move data**
- **Allow guest OS's to handle device interrupts**
- **For security: allow programs to manage encrypted portions of code and data**