

# 15-849 Datacenter Computing

Dimitrios Skarlatos

Fall 2021

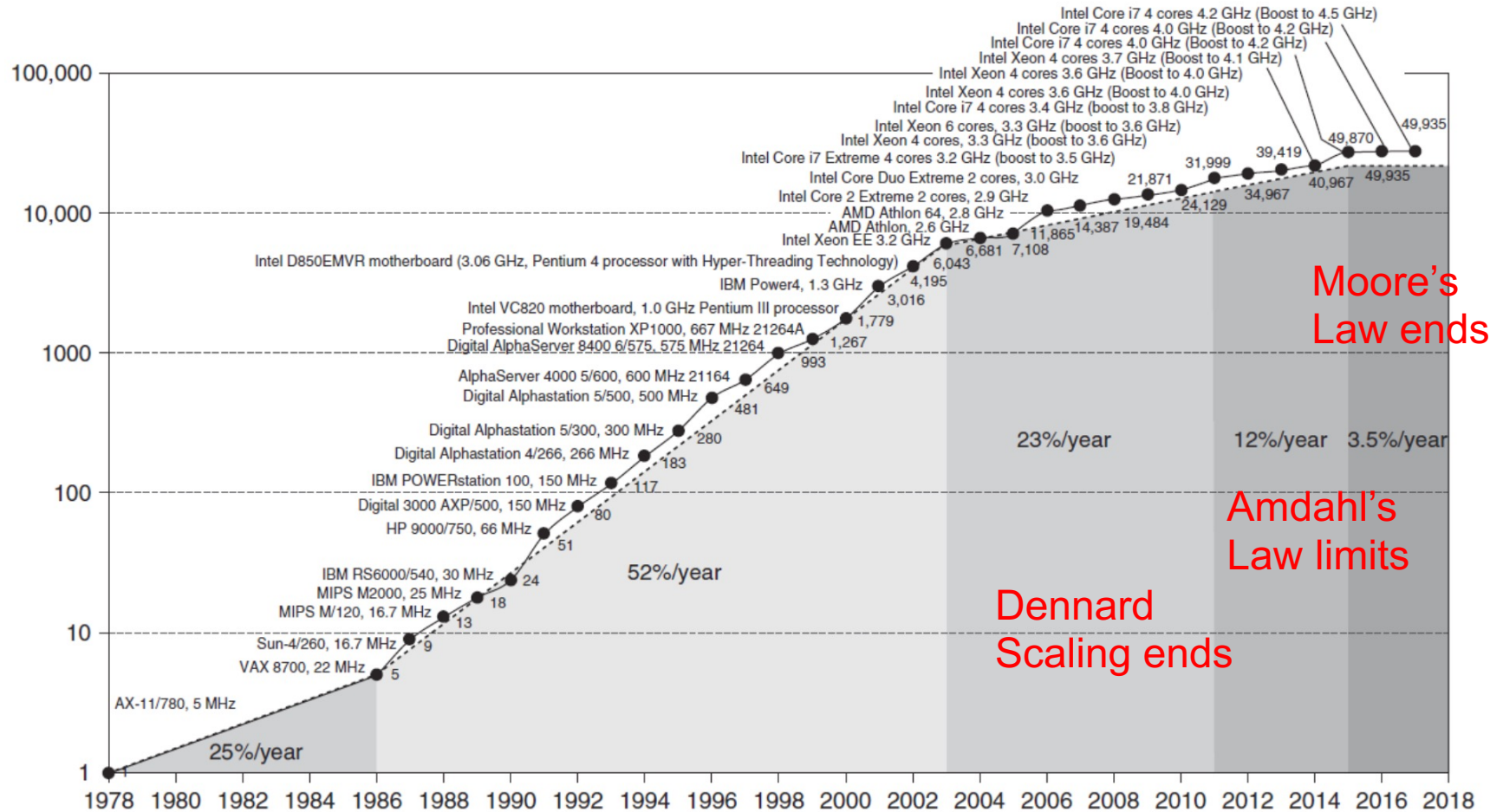
# Agenda

- Processor Design
- Memory Hierarchy
- Virtual Memory

Some Slides from:  
Computer Architecture A Quantitative Approach

# Processor Design

# Single Processor Performance

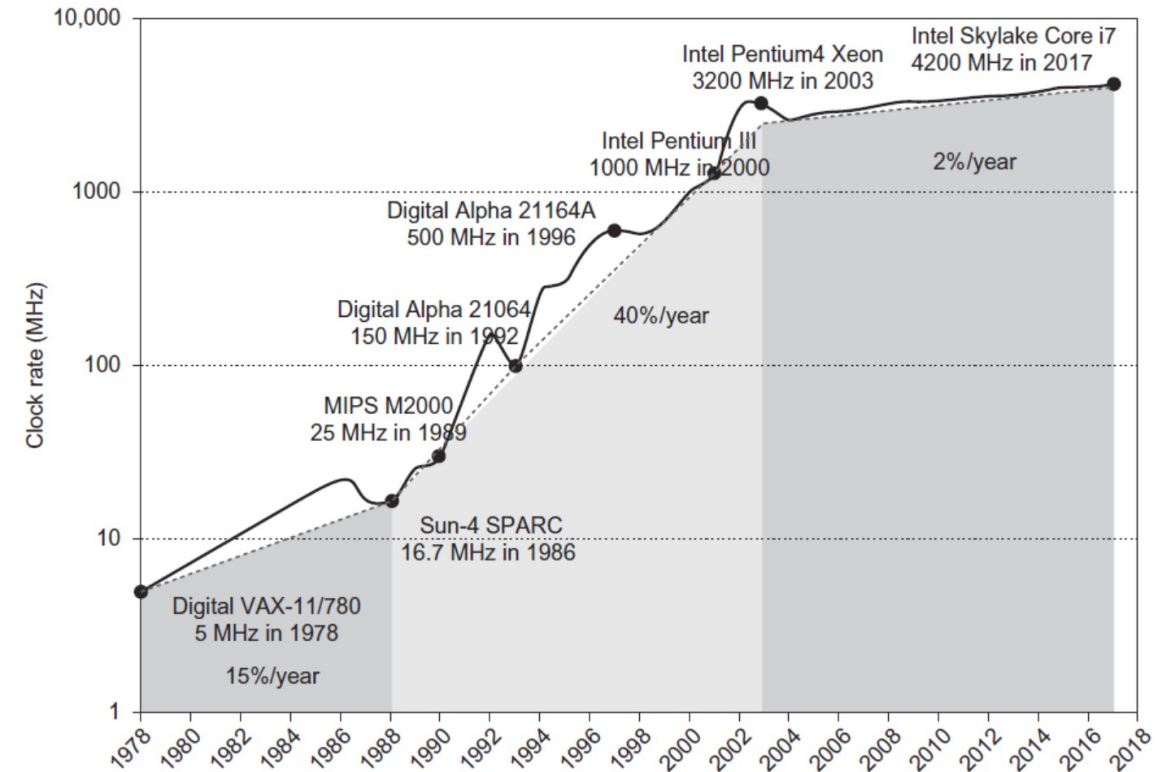


# Power

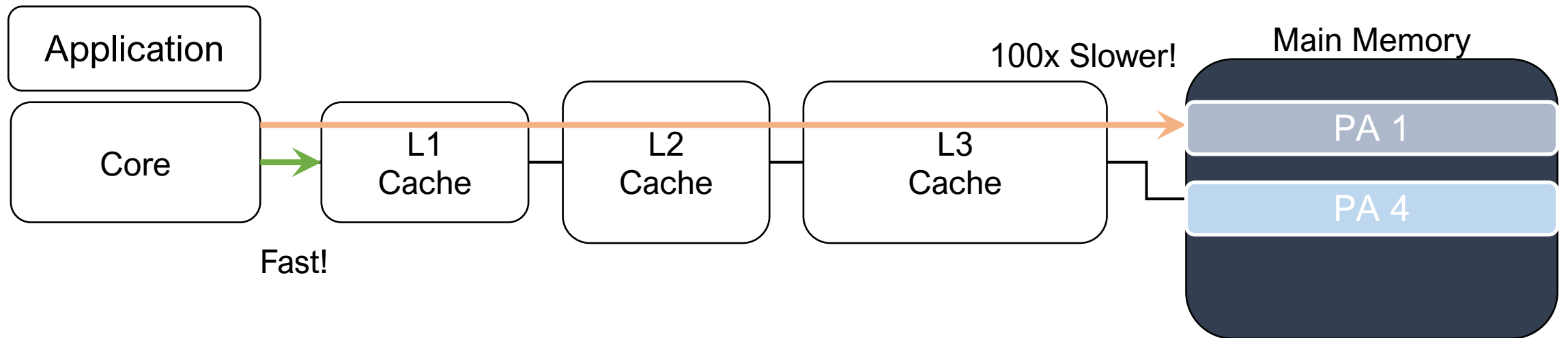
Intel 80386 ~ 2 W

Intel Core i7 ~130 W

Pretty much the limit of what  
can be cooled by air

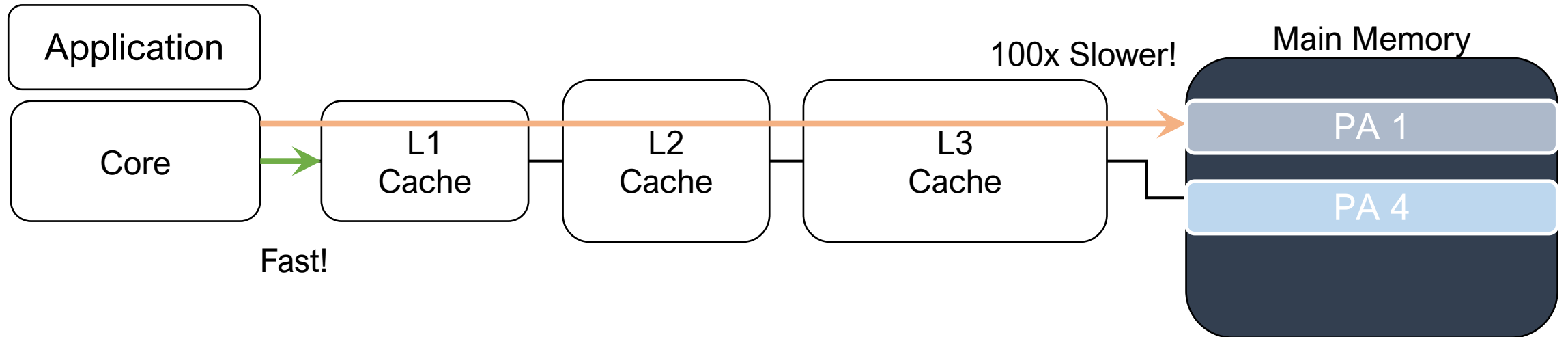
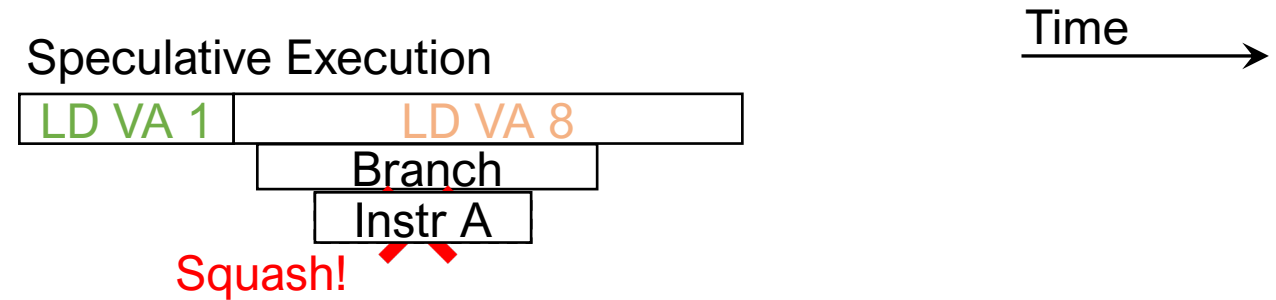
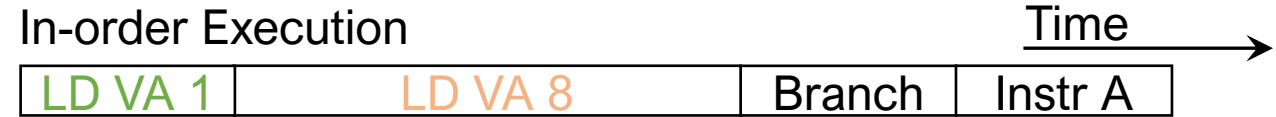


# The Problem



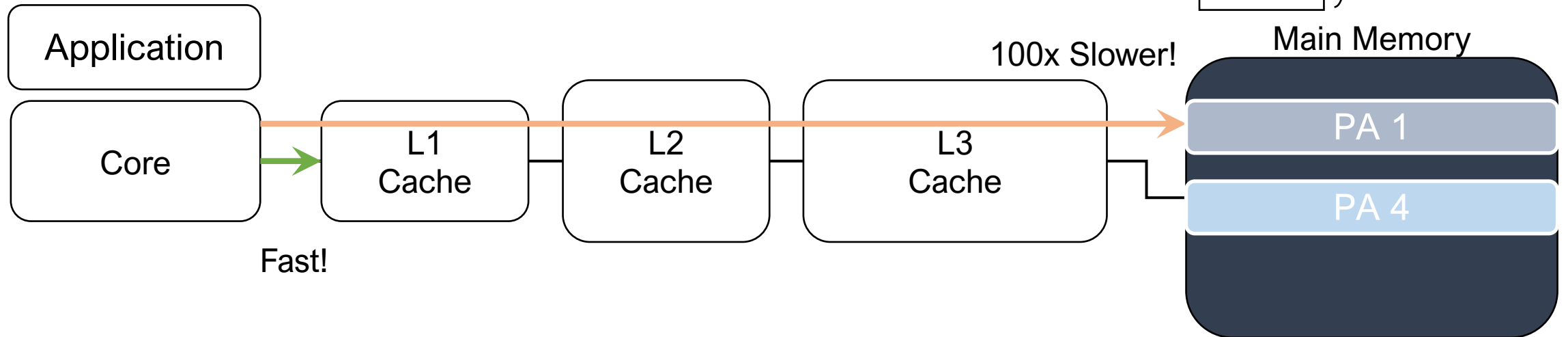
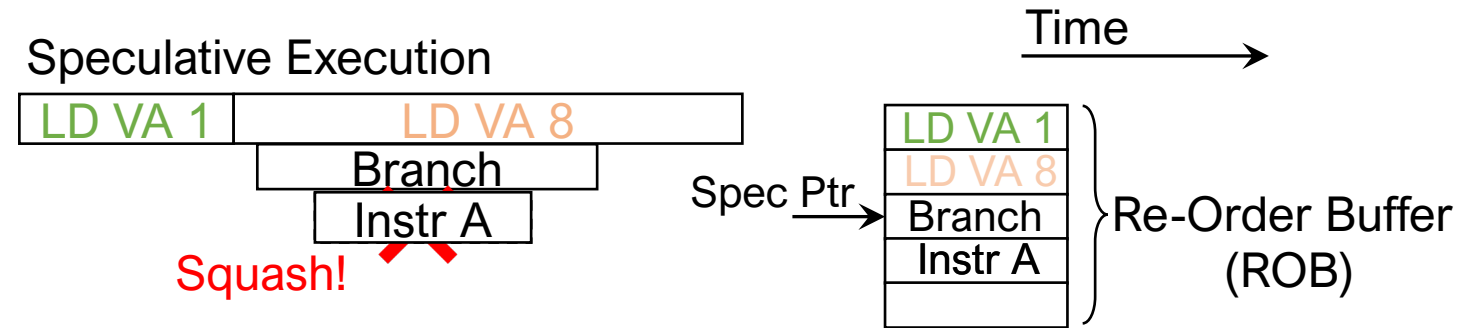
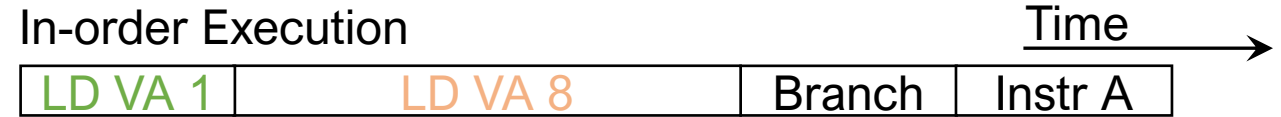
# Speculative Execution

```
1. Issue LD VA1
2. Issue LD VA8
3. if (...) {
4.     Instruction A
5. }
```



# Speculative Execution

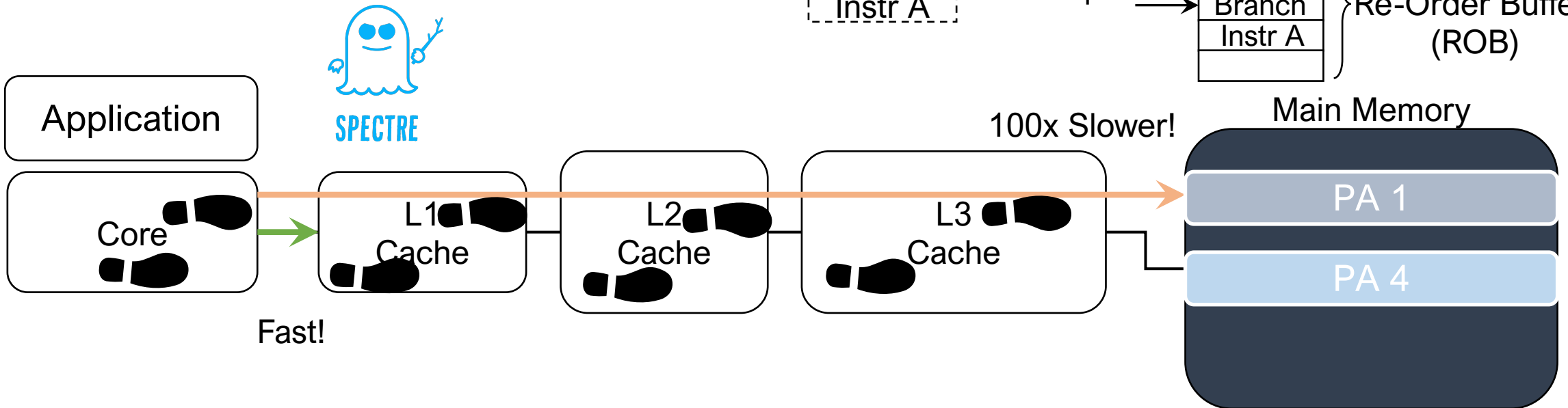
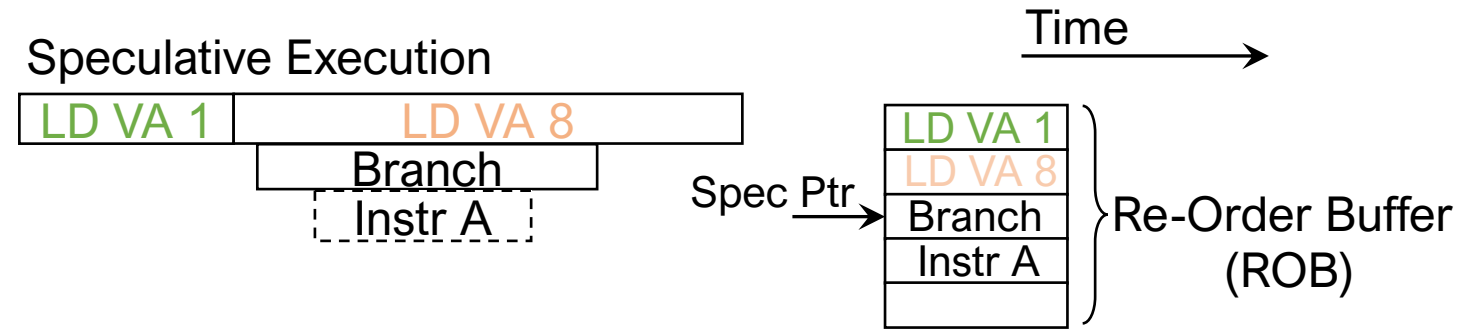
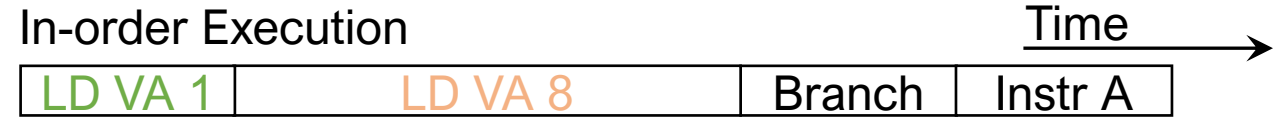
```
1. Issue LD VA1
2. Issue LD VA8
3. if (...) {
4.   Instruction A
5. }
```





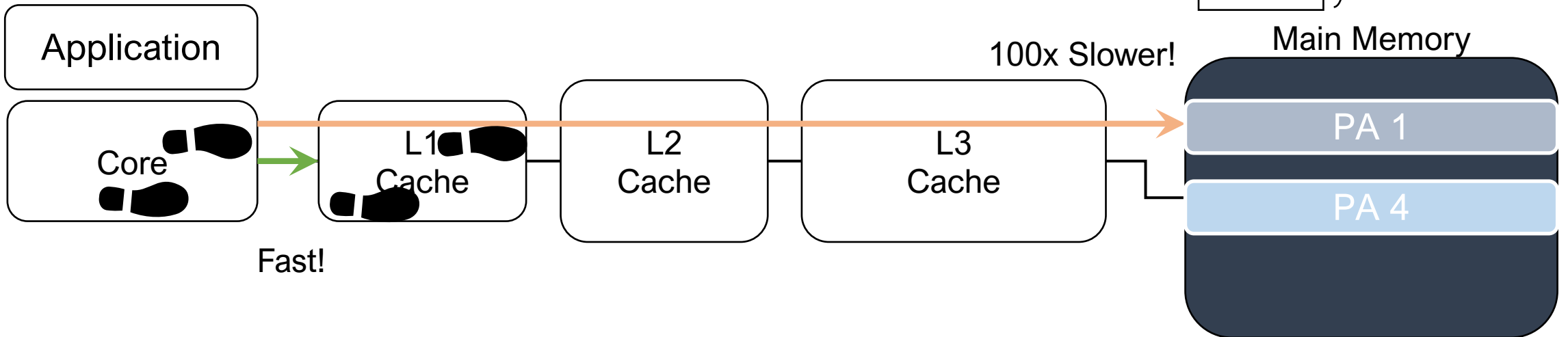
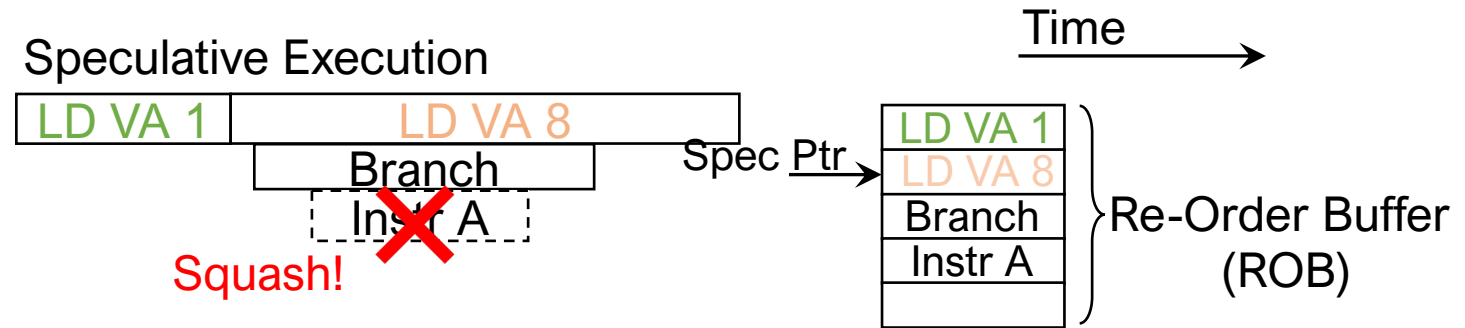
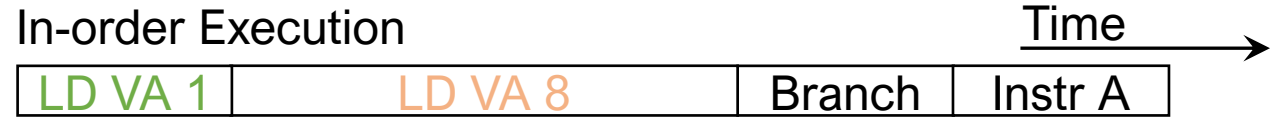
# Speculative Execution

```
1. Issue LD VA1
2. Issue LD VA8
3. if (...) {
4.   Instruction A
5. }
```



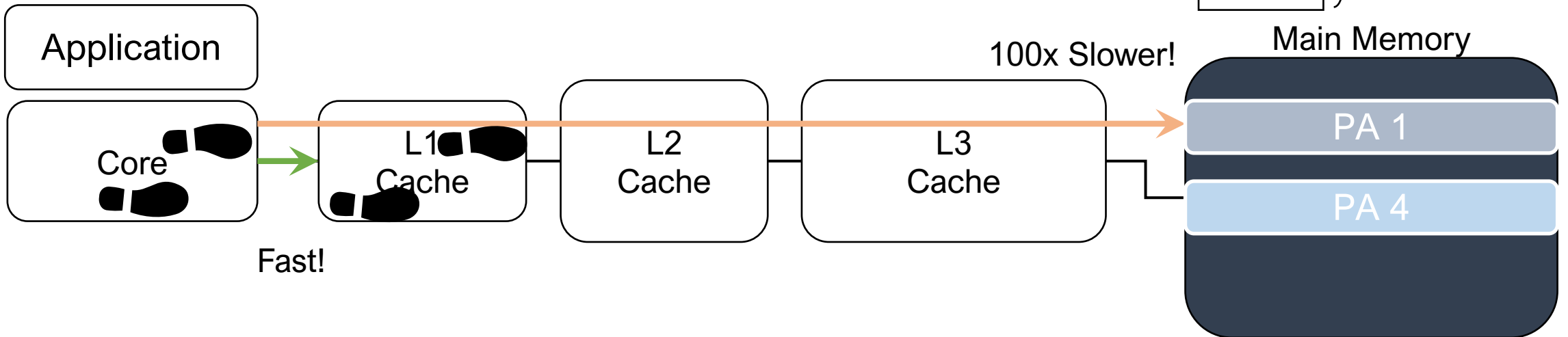
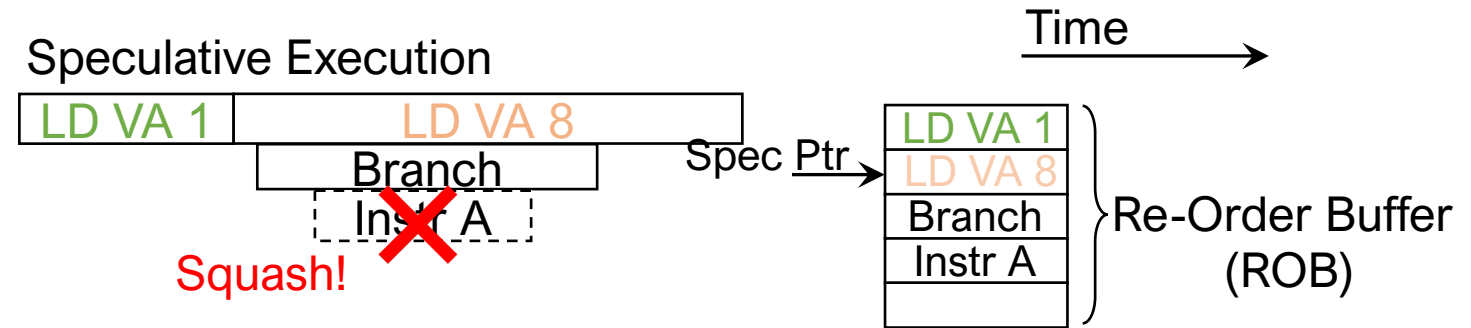
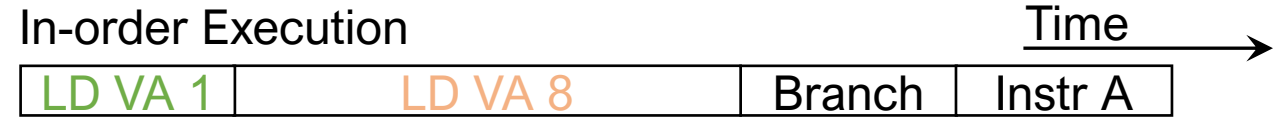
# Speculative Execution

```
1. Issue LD VA1
2. Issue LD VA8
3. if (...) {
4.   Instruction A
5. }
```



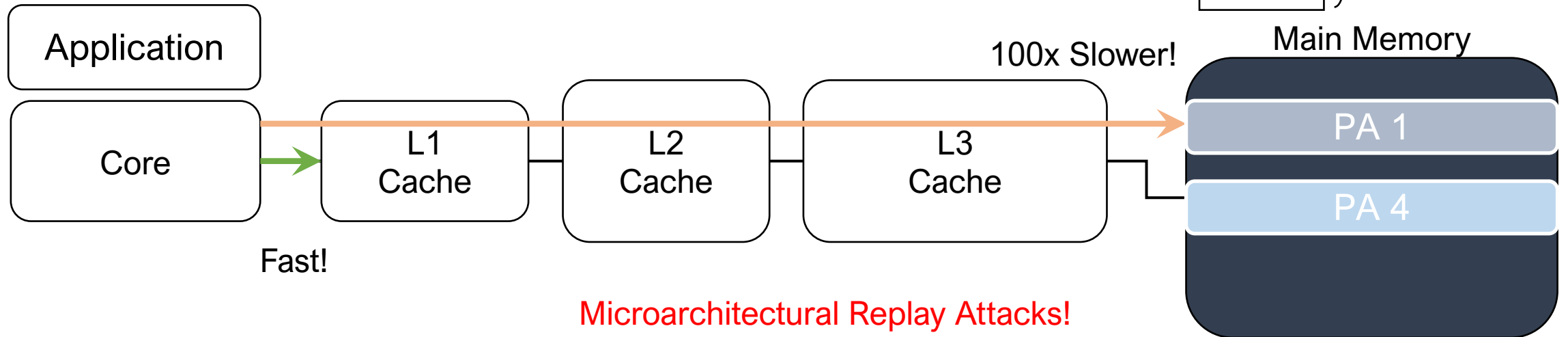
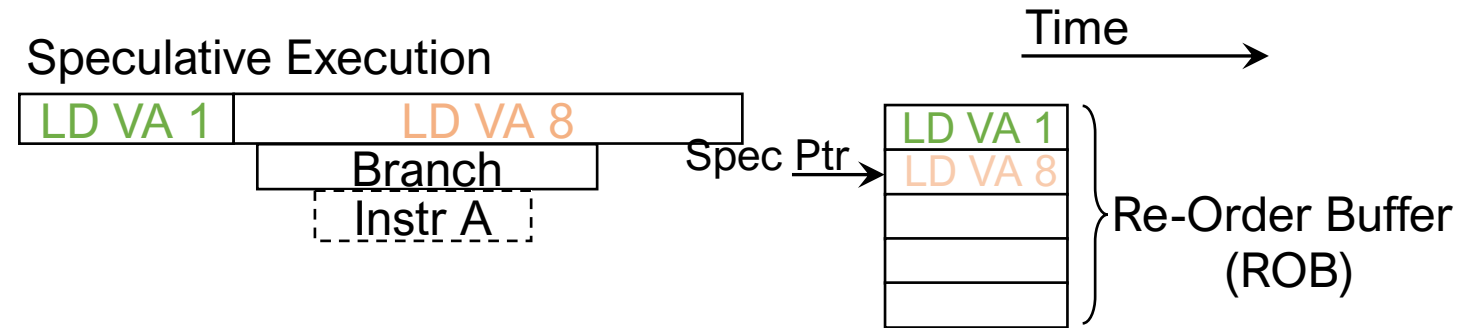
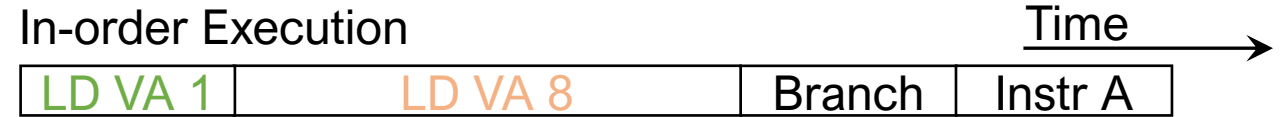
# Speculative Execution

```
1. Issue LD VA1
2. Issue LD VA8
3. if (...) {
4.   Instruction A
5. }
```



# Speculative Execution

```
1. Issue LD VA1
2. Issue LD VA8
3. if (...) {
4.   Instruction A
5. }
```

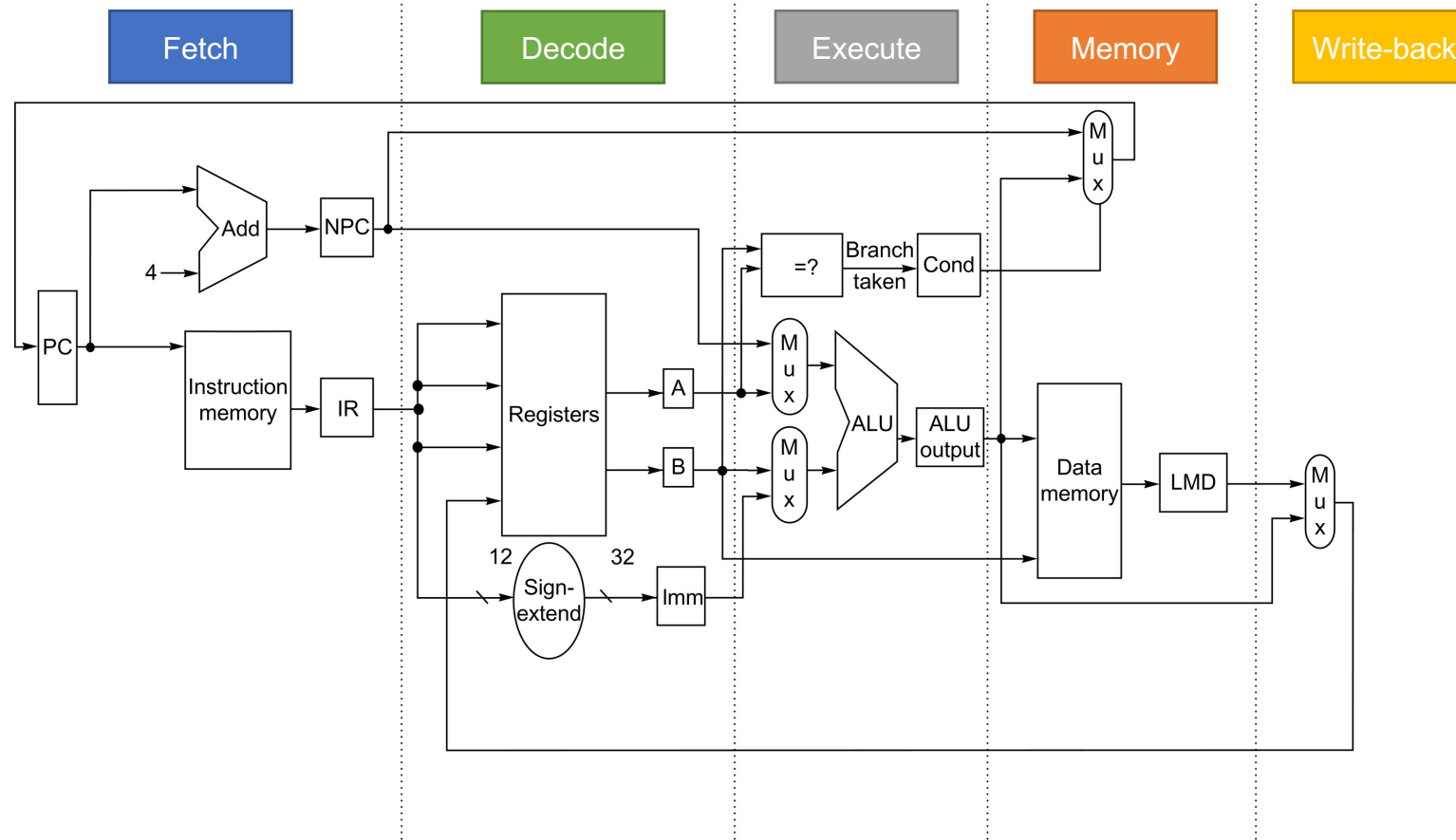


# How did we get here?

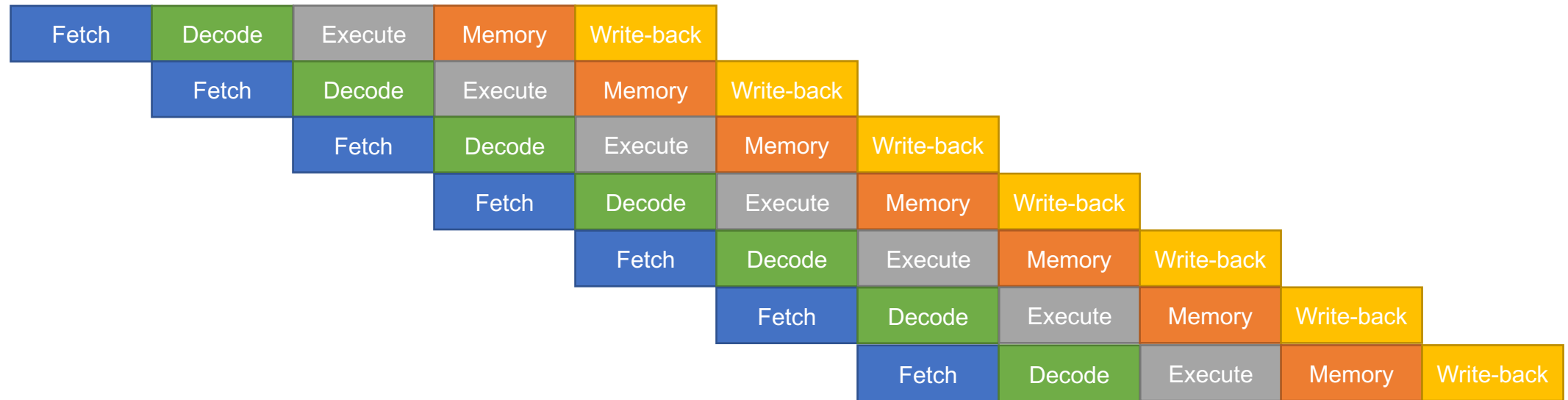
Pipelining become universal technique in 1985

- Overlaps execution of instructions
- Exploits “Instruction Level Parallelism”

# 5-stage Pipeline



# 5-stage Pipeline



# How did we get here?

Pipelining become universal technique in 1985

- Overlaps execution of instructions
- Exploits “Instruction Level Parallelism”

Beyond this, there are two main approaches:

- Hardware-based dynamic approaches
  - Used in server and desktop processors
  - Not used as extensively in PMP processors
- Compiler-based static approaches
  - Not as successful outside of scientific applications



# Instruction Level Parallelism (ILP)

When exploiting instr-level parallelism, goal is to minimize CPI

- Pipeline CPI =
  - Ideal pipeline CPI +
  - Structural stalls +
  - Data hazard stalls +
  - Control stalls

Parallelism with basic block is limited

- Typical size of basic block = 3-6 instructions
- Must optimize across branches

# Data dependency

Instruction  $j$  is data dependent on instruction  $i$  if

- Instruction  $i$  produces a result that may be used by instruction  $j$
- Instruction  $j$  is data dependent on instruction  $k$  and instruction  $k$  is data dependent on instruction  $i$

Dependent instructions cannot be executed simultaneously

# Data dependency

Dependencies are a property of programs

Pipeline organization determines detection and if it causes a stall

Data dependence conveys:

- Possibility of a hazard
- Order in which results must be calculated
- Upper bound on exploitable instruction level parallelism

# Stall Factors

## Data Hazards

- Read after write (RAW) ← True Dependence!
- Write after write (WAW)
- Write after read (WAR)

$$\begin{aligned} R2 &\leftarrow R0 + R1 \\ R3 &\leftarrow R2 + R4 \end{aligned}$$

## Control Dependence

- Ordering of instruction  $i$  with respect to a branch instruction
  - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
  - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

# Branch Prediction

## Basic 2-bit predictor:

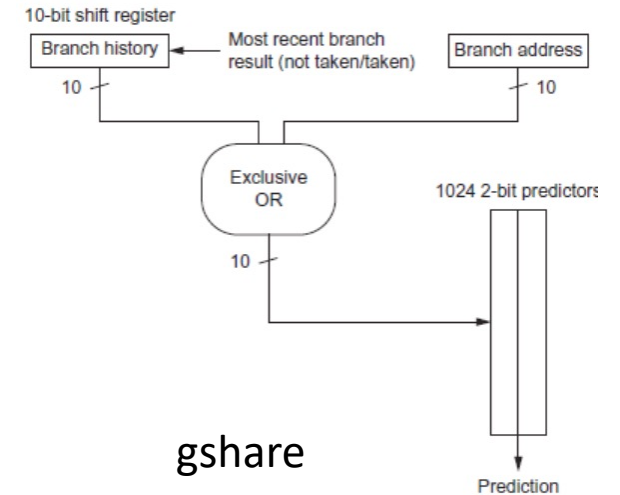
- For each branch:
  - Predict taken or not taken
  - If the prediction is wrong two consecutive times, change prediction

## Correlating predictor:

- Multiple 2-bit predictors for each branch
- One for each possible combination of outcomes of preceding  $n$  branches
  - $(m,n)$  predictor: behavior from last  $m$  branches to choose from  $2^m$   $n$ -bit predictors

## Tournament predictor:

- Combine correlating predictor with local predictor



# Hardware-Based Speculation

Execute instructions along predicted execution paths but only commit the results if prediction was correct

Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative

Need an additional piece of hardware to prevent any irrevocable action until an instruction commits

- I.e. updating state or taking an execution

# Reorder Buffer

Holds the result of instruction between completion and commit

Four fields:

- Instruction type: branch/store/register
- Destination field: register number
- Value field: output value
- Ready field: completed execution?

Type	Dst	Val	Rdy

Modify pipeline:

- Operand source is now reorder buffer entry instead of functional unit

# Reorder Buffer

## Issue:

- Allocate ROB, read available operands

## Execute:

- Begin execution when operand values are available

## Write result:

- Write result and ROB tag on bus

## Commit:

- When ROB reaches head of ROB, update register
- When a mispredicted branch reaches head of ROB, *discard all entries*





# Reorder Buffer

Register values and memory values are not written until commit

On misprediction:

- Speculated entries in ROB are cleared

Exceptions:

- Not recognized until it is ready to commit



# Speculative Execution

## How much to speculate

- Mis-speculation degrades performance and power relative to no speculation
  - May cause additional misses (cache, TLB)
- Prevent speculative code from causing higher costing misses (e.g. L2)

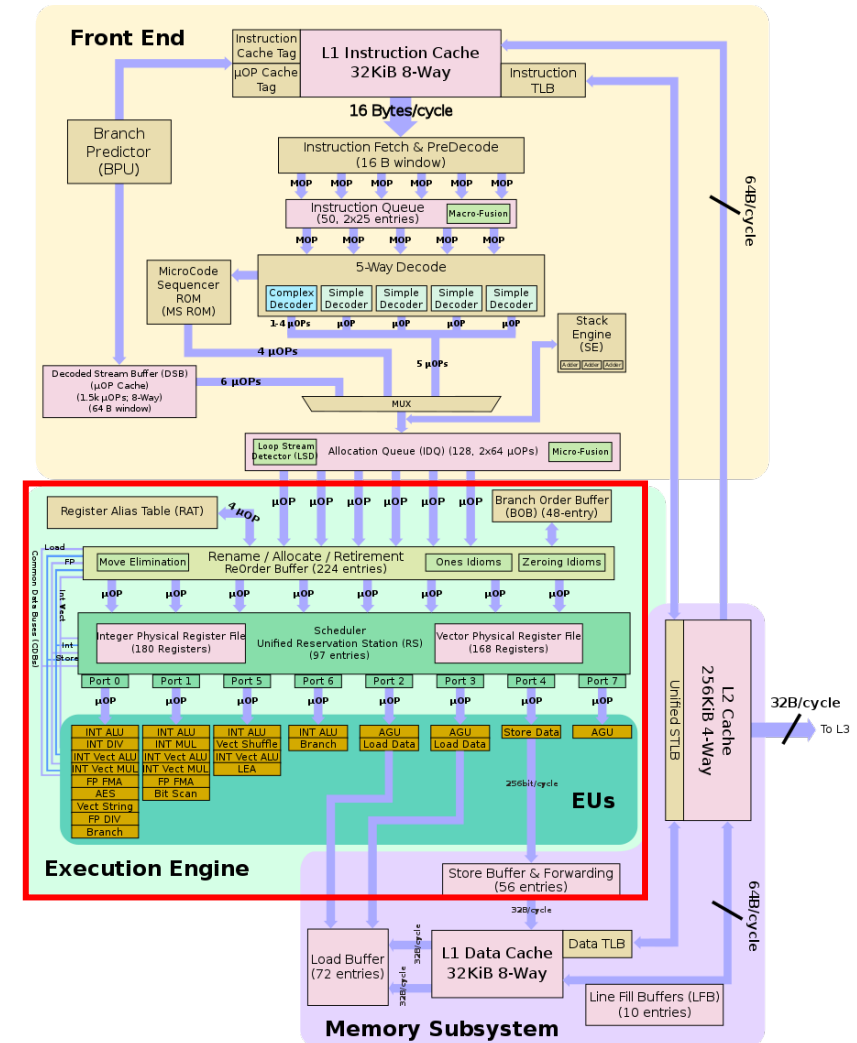
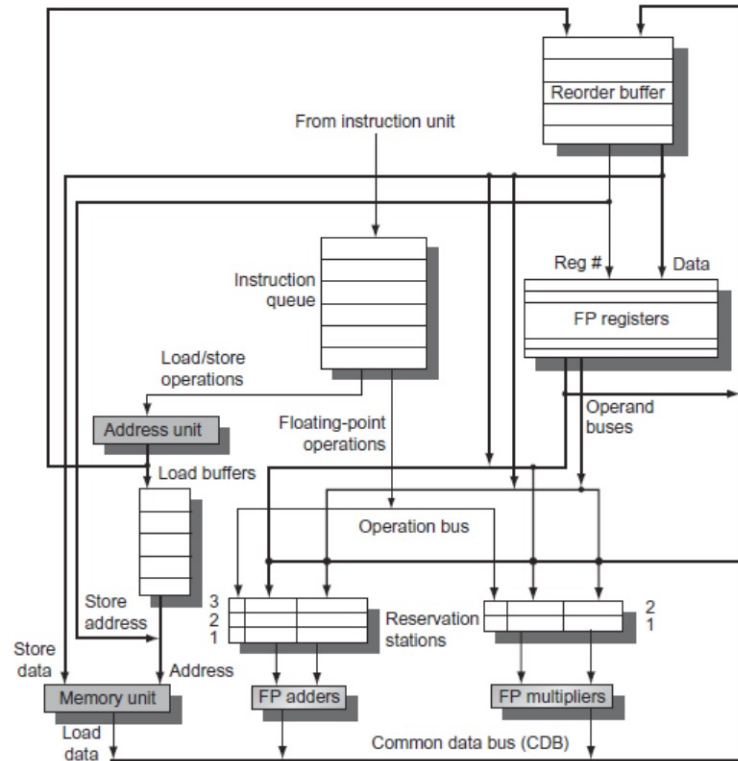
## Speculating through multiple branches

- Complicates speculation recovery

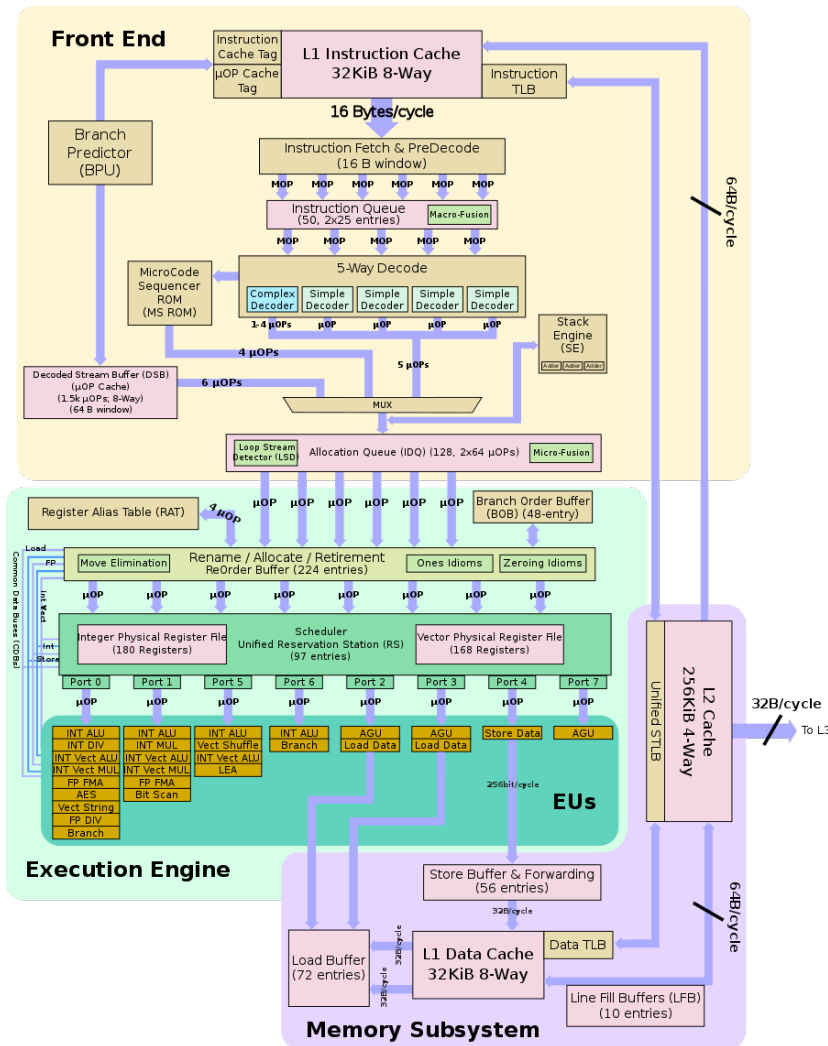
## Speculation and energy efficiency

- Note: speculation is only energy efficient when it significantly improves performance

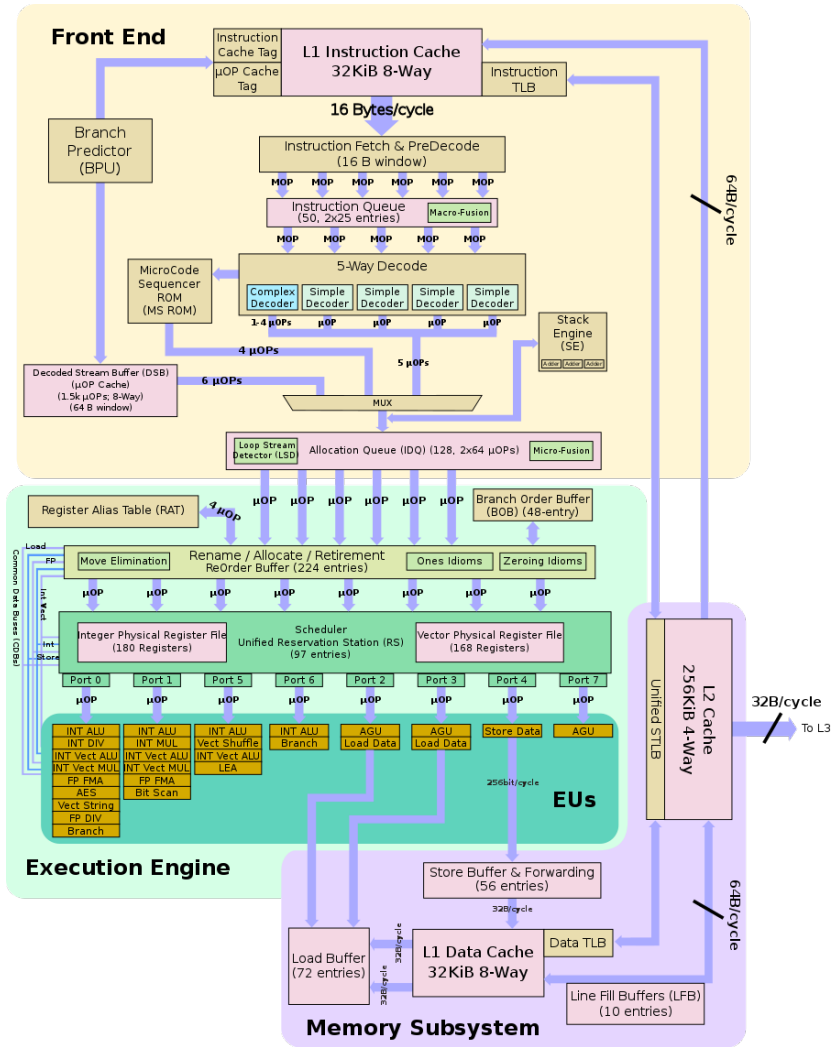
# Processor Pipeline



# Simultaneous Multithreading (SMT)



# Simultaneous Multithreading (SMT)

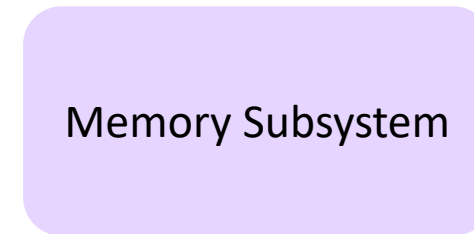
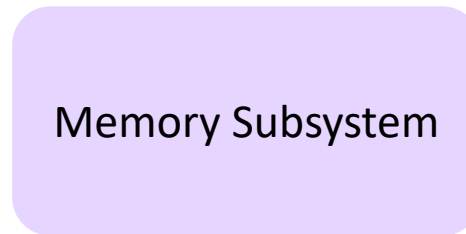
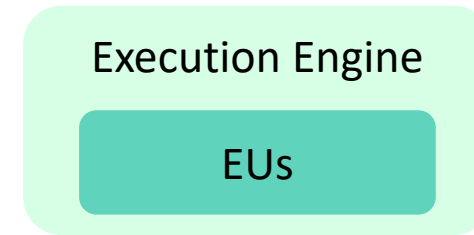
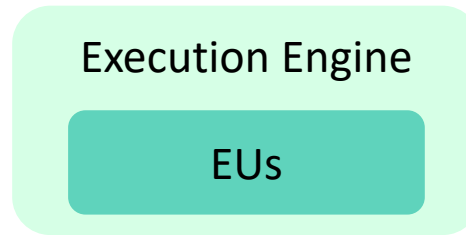
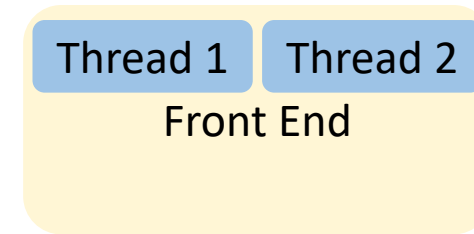
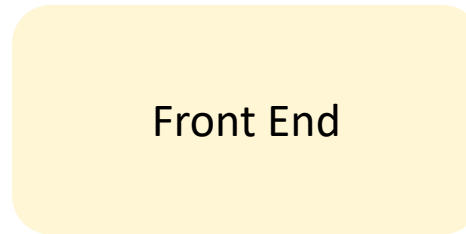
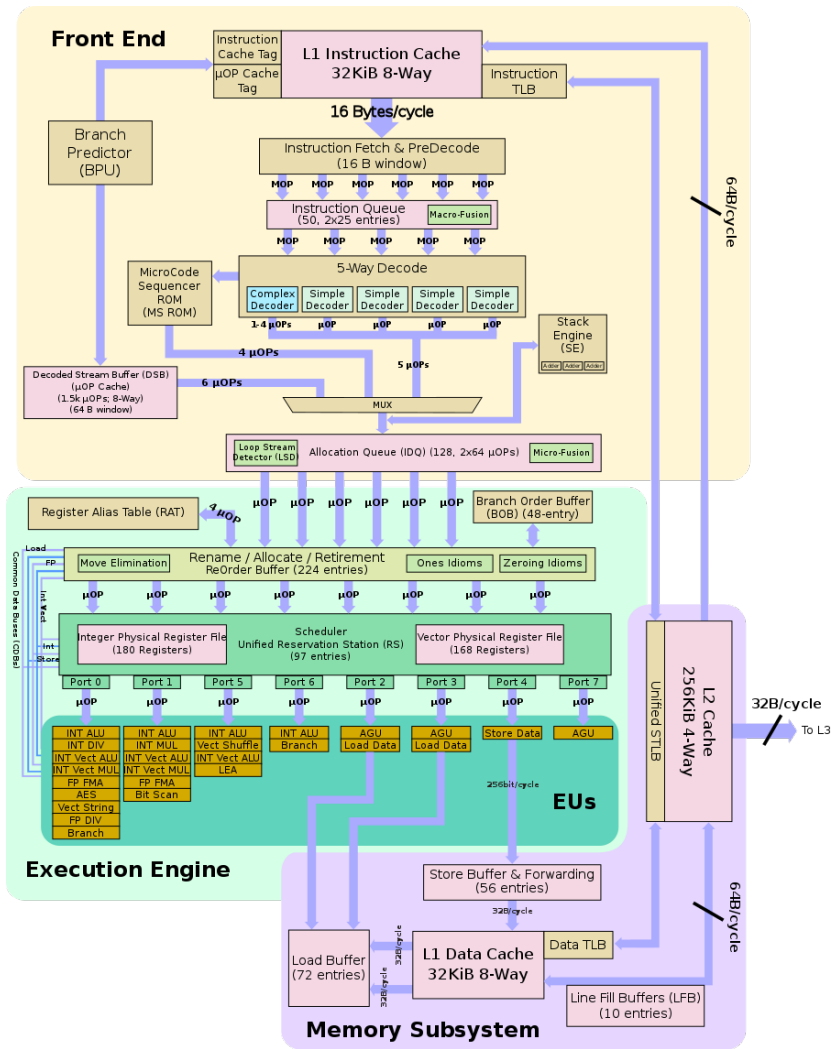


Front End

Execution Engine  
EUs

Memory Subsystem

# Simultaneous Multithreading (SMT)



# Security Considerations

Deep pipeline w/ variable timing per instr → Subnormal FP, PortSmash

Branch predictors → Foundation of many attacks (secret-dependent CF)

Load / Store queues → MemJam, MDS (RIDL, Fallout)

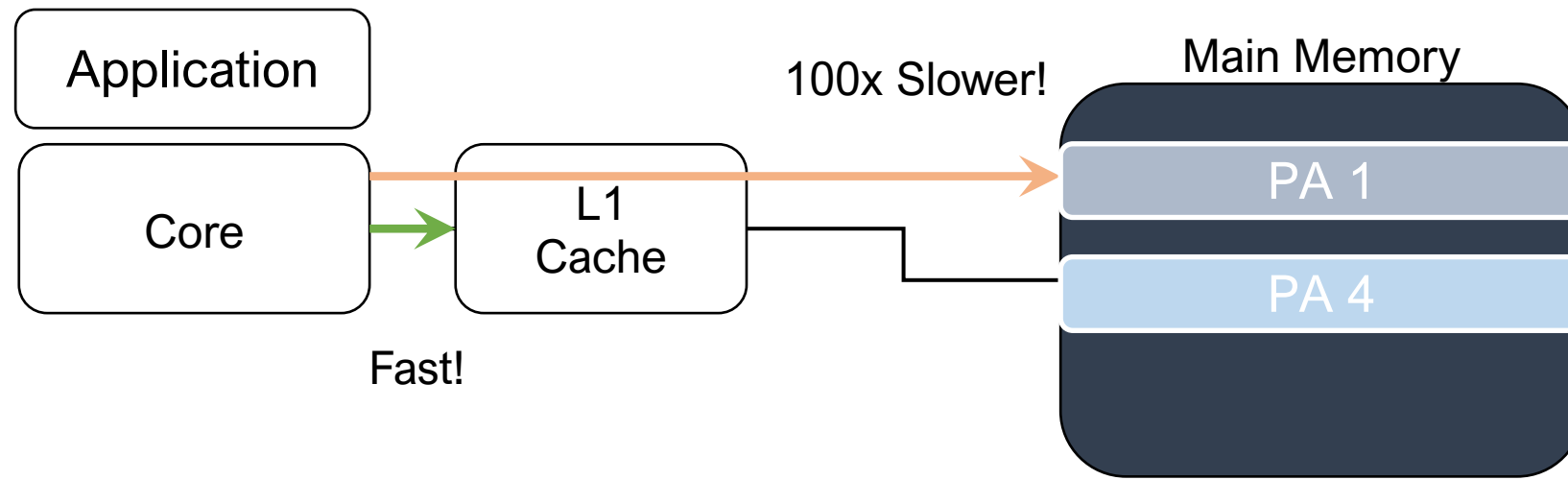
Out-of-Order Execution → Spectre, Meltdown, Microscope

Simultaneous multi-threading → Amplifies/Simplifies most attacks

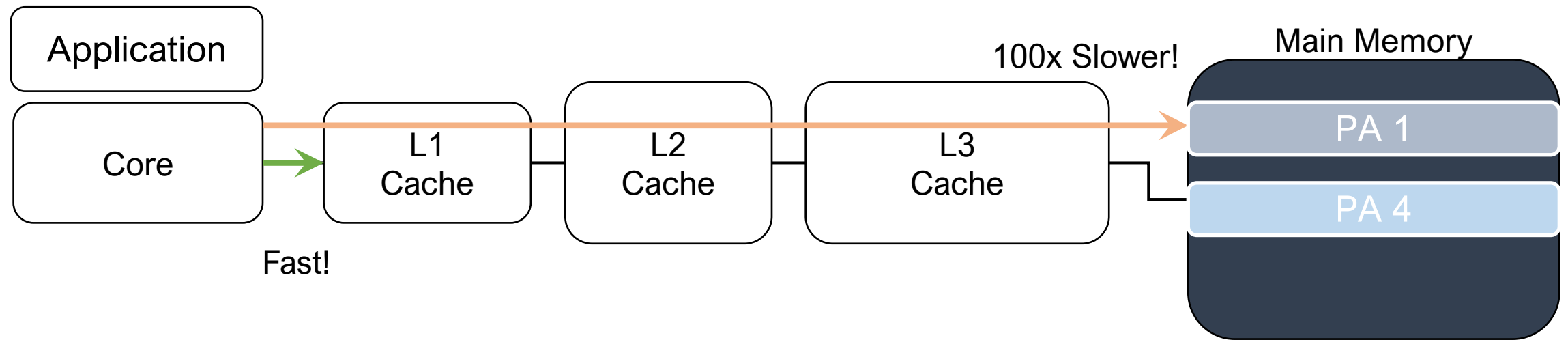
# Memory Hierarchy



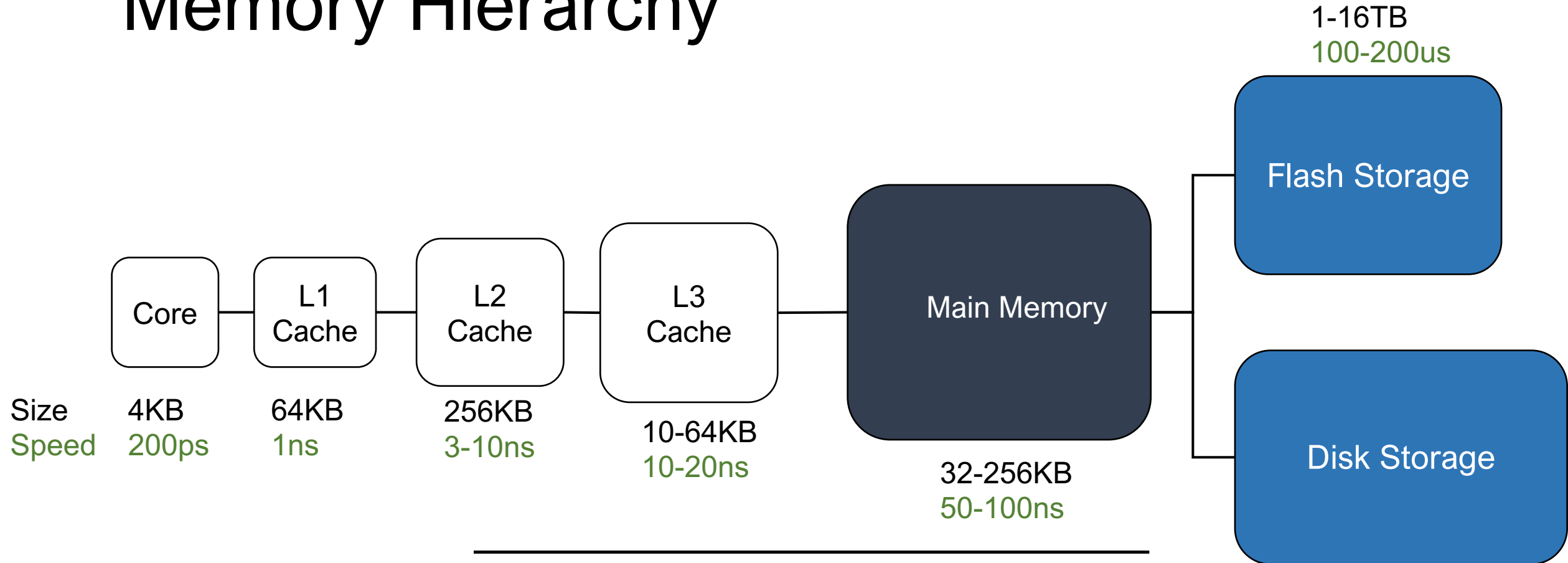
# The Problem



# Memory Hierarchy

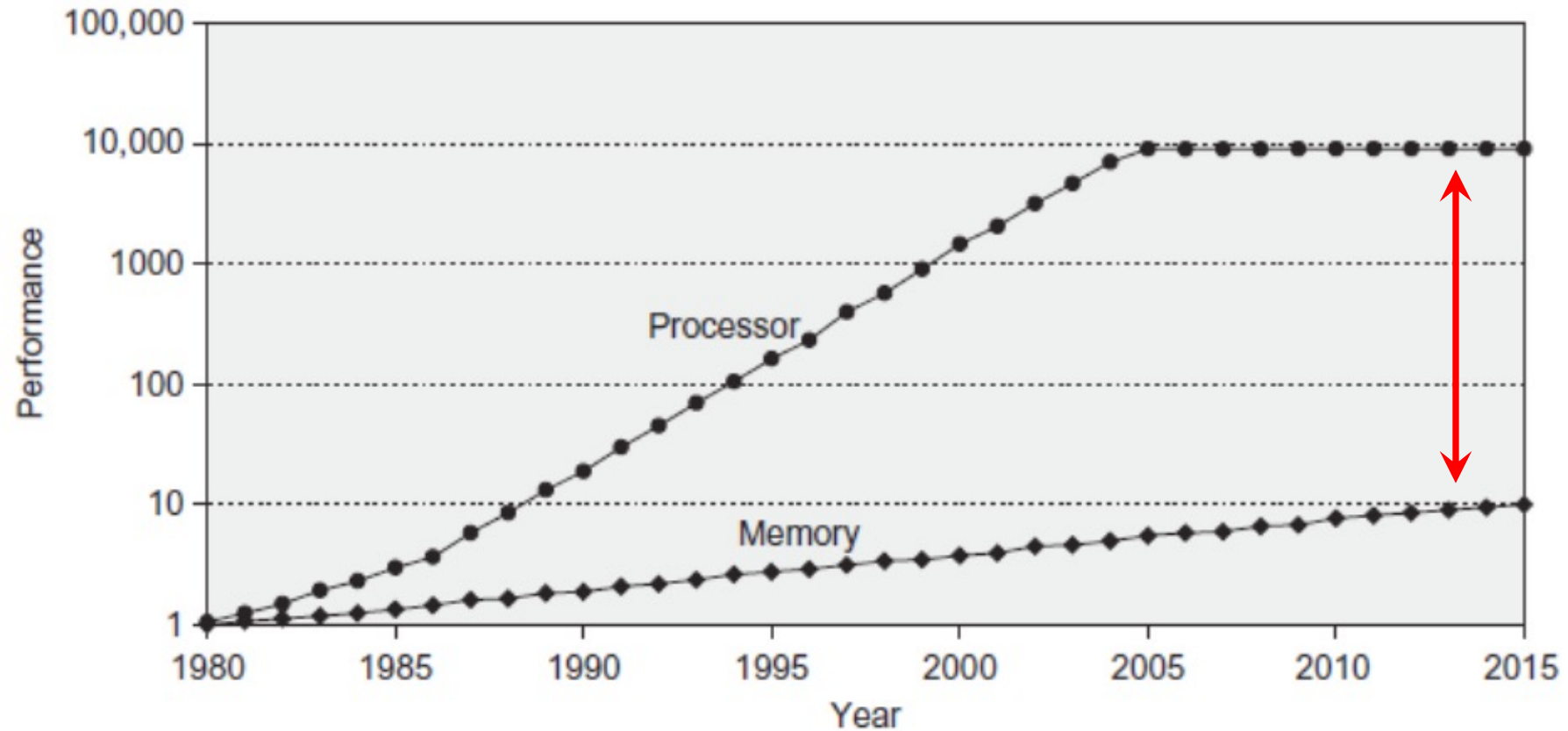


# Memory Hierarchy



Characteristic	L1	L2	L3
Size	32 KiB I/32 KiB D	256 KiB	2 MiB per core
Associativity	both 8-way	4-way	16-way
Access latency	4 cycles, pipelined	12 cycles	44 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorithm

# The Processor-Memory Gap



# Why we need a memory hierarchy?

Programmers want unlimited amounts of memory with low latency

Fast memory technology is more expensive per bit than slower memory

Solution: organize memory system into a hierarchy

- Entire addressable memory space available in largest, slowest memory
- Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed in steps up toward the processor

Temporal and spatial locality insures that nearly all references can be caches

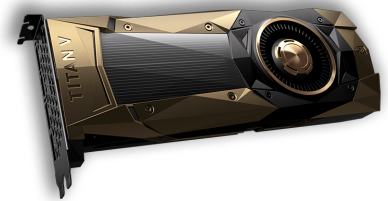
- Gives the allusion of a large, fast memory being presented to the processor

# Memory Hierarchy Design

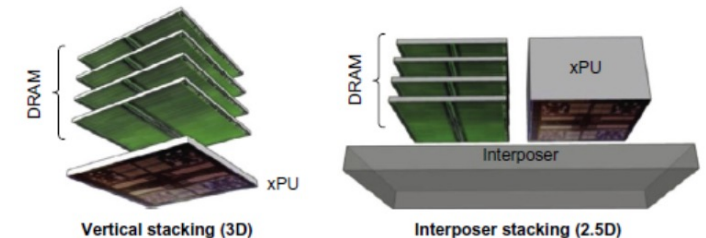
Memory hierarchy design becomes more crucial with recent multi-core processors:

- Aggregate peak bandwidth grows with # cores:
  - Intel Core i7 can generate two references per core per clock
  - Four cores and 3.2 GHz clock
    - 25.6 billion 64-bit data references/second +
    - 12.8 billion 128-bit instruction references/second
    - = 409.6 GB/s!
  - DRAM bandwidth is only 8% of this (34.1 GB/s)
  - Requires:
    - Multi-port, pipelined caches
    - Two levels of cache per core
    - Shared third-level cache on chip

GDDR5



HBM



# Memory Hierarchy Fundamentals

When a word is not found in the cache, a *miss* occurs:

- Fetch word from lower level in hierarchy, requiring a higher latency
- Lower level may be another cache or the main memory
- Also fetch the other words contained within the *block*
  - Takes advantage of spatial locality
- Place block into cache in any location within its *set*, determined by addr
  - Block address MOD number of sets in cache

# Memory Hierarchy Fundamentals

$n$  sets  $\Rightarrow$   $n$ -way set associative

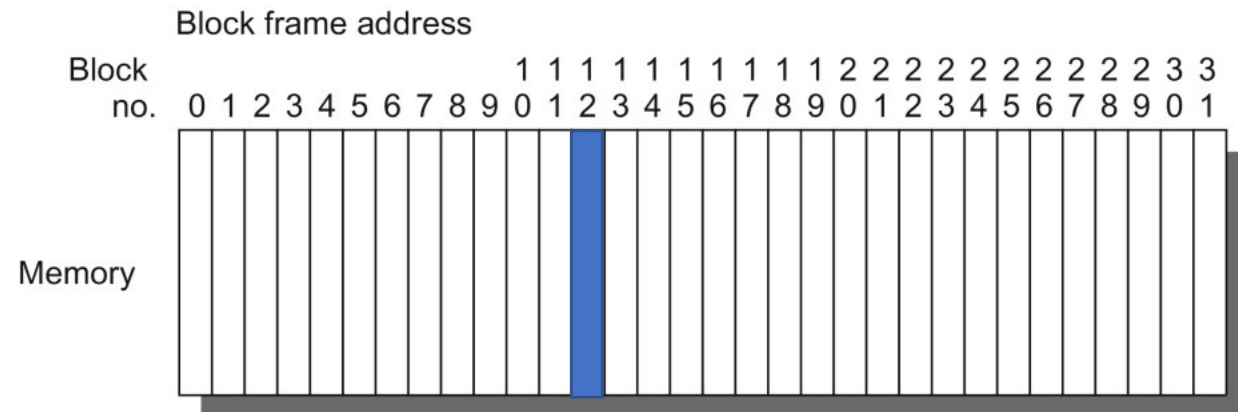
- *Direct-mapped cache*  $\Rightarrow$  one block per set
- *Fully associative*  $\Rightarrow$  one set

Writing to cache: two strategies

- *Write-through*
  - Immediately update lower levels of hierarchy
- *Write-back*
  - Only update lower levels of hierarchy when an updated block is replaced
- Both strategies use *write buffer* to make writes asynchronous

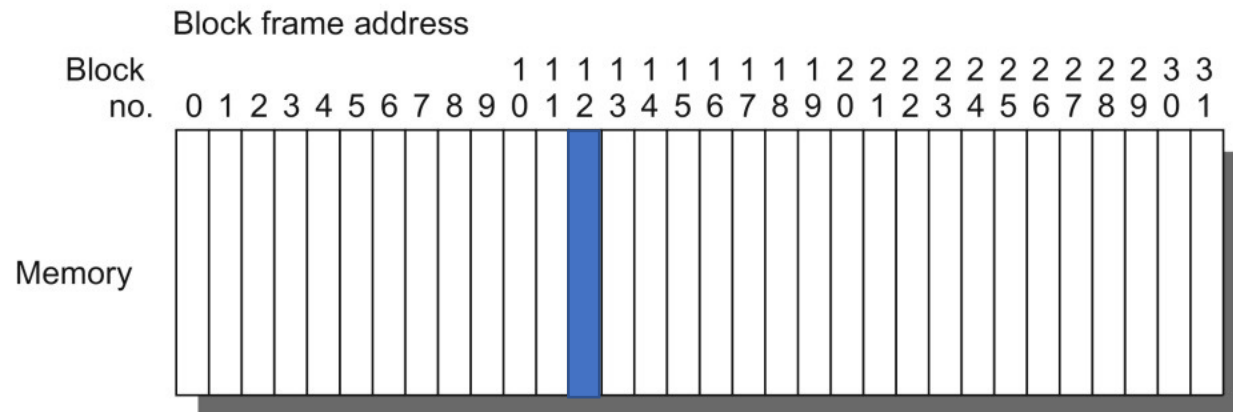
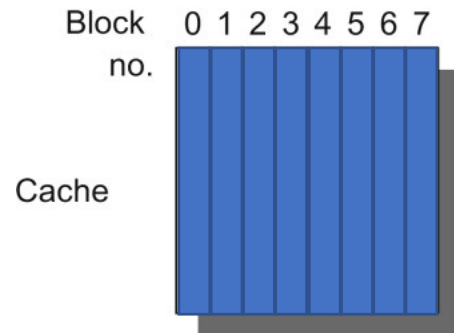


# Memory Hierarchy Fundamentals



# Memory Hierarchy Fundamentals

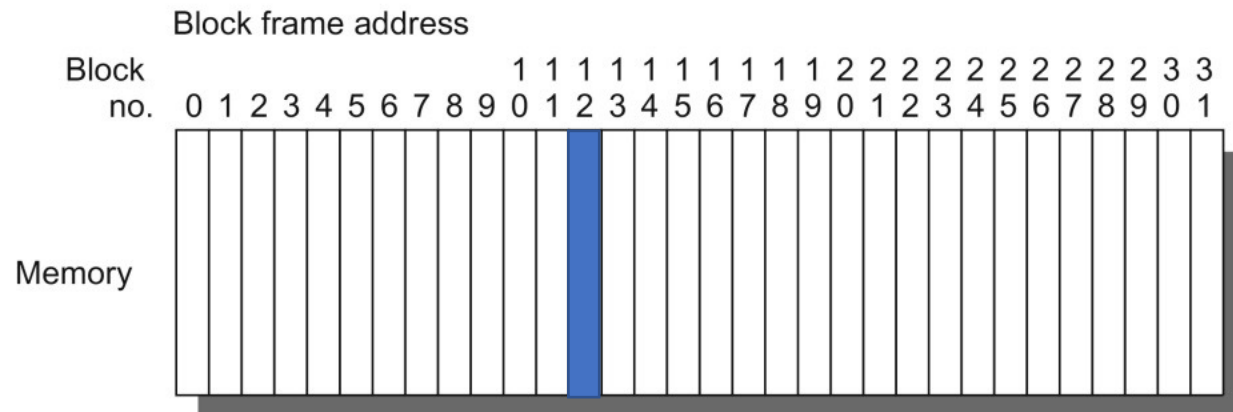
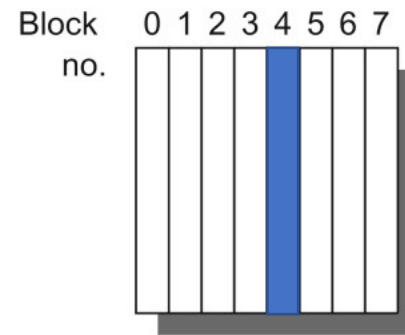
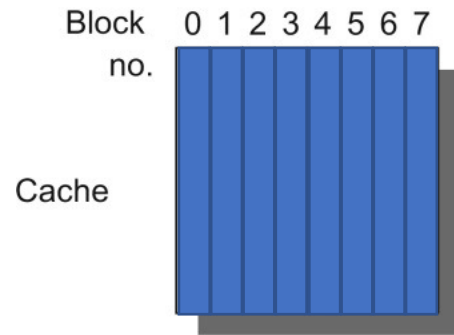
Fully associative:  
block 12 can go  
anywhere



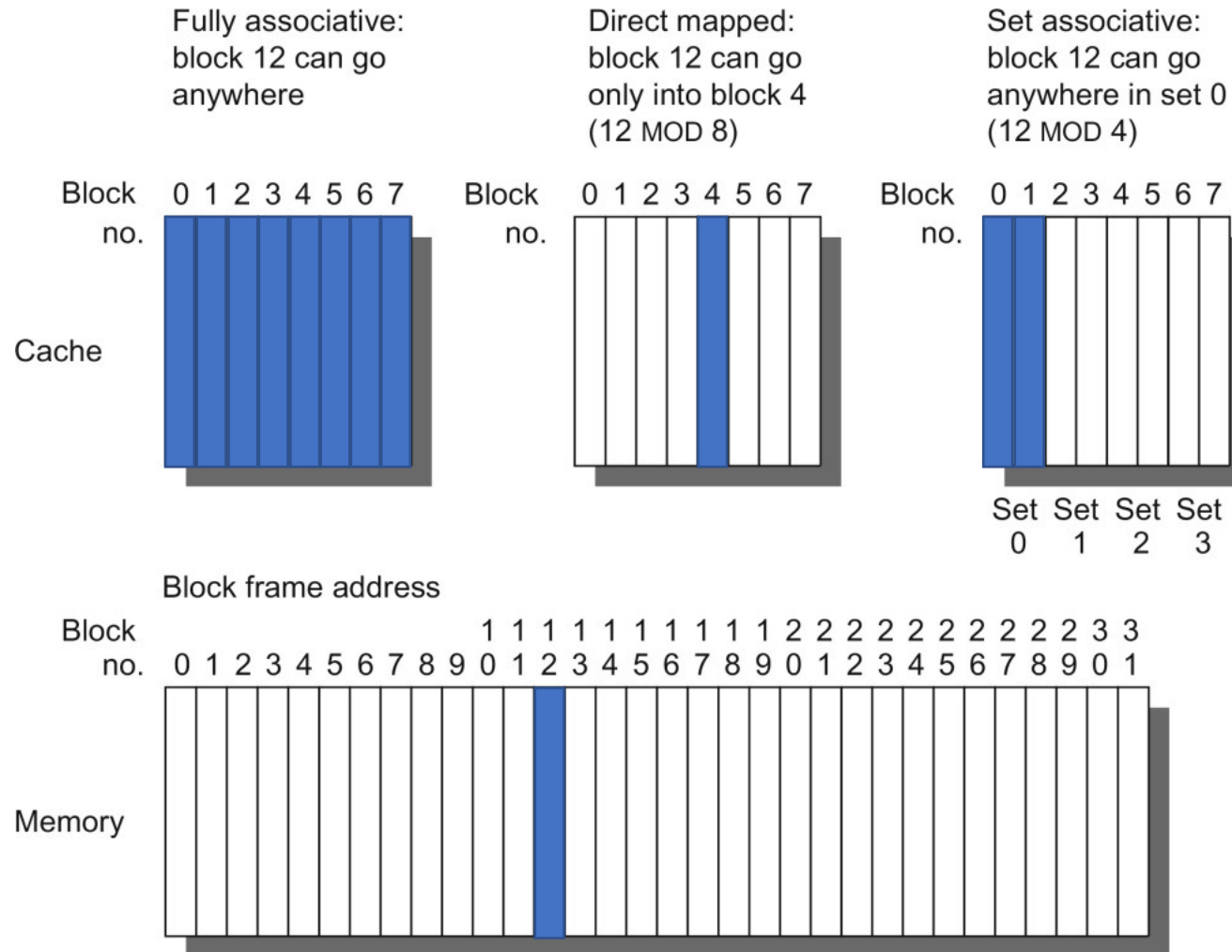
# Memory Hierarchy Fundamentals

Fully associative:  
block 12 can go  
anywhere

Direct mapped:  
block 12 can go  
only into block 4  
(12 MOD 8)



# Memory Hierarchy Fundamentals



# Memory Hierarchy Fundamentals

## Six basic cache optimizations:

1. Larger block size
  - Reduces compulsory misses
  - Increases capacity and conflict misses, increases miss penalty
2. Larger total cache capacity to reduce miss rate
  - Increases hit time, increases power consumption
3. Higher associativity
  - Reduces conflict misses
  - Increases hit time, increases power consumption
4. Higher number of cache levels
  - Reduces overall memory access time
5. Giving priority to read misses over writes
  - Reduces miss penalty
6. Avoiding address translation in cache indexing
  - Reduces hit time

# Security Considerations

Shared memory hierarchy

Variable latencies across levels

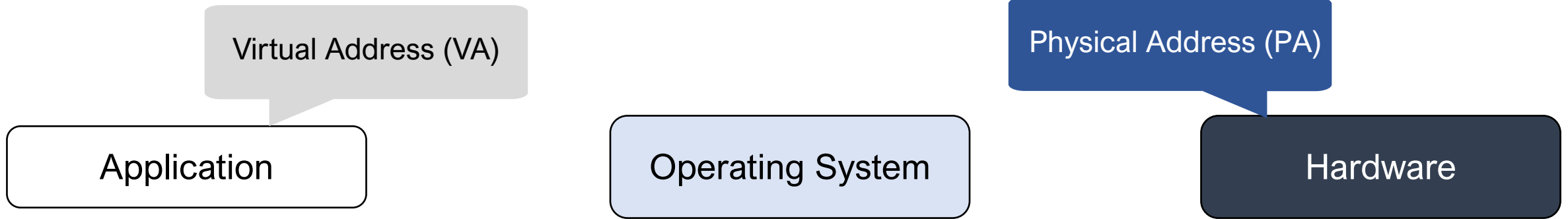
Sets, Ways, Banks, Slices

Cache timing & contention attacks  
Prime+Probe, Flush+Reload, Flush+Flush  
L1, L2, cross-core L3  
Inclusive and non-inclusive  
Directories  
Prefetchers  
Replacement policy  
Userspace, SGX

Main Memory (DRAM) → DRAM-based timing (DRAMAs), Rowhammer

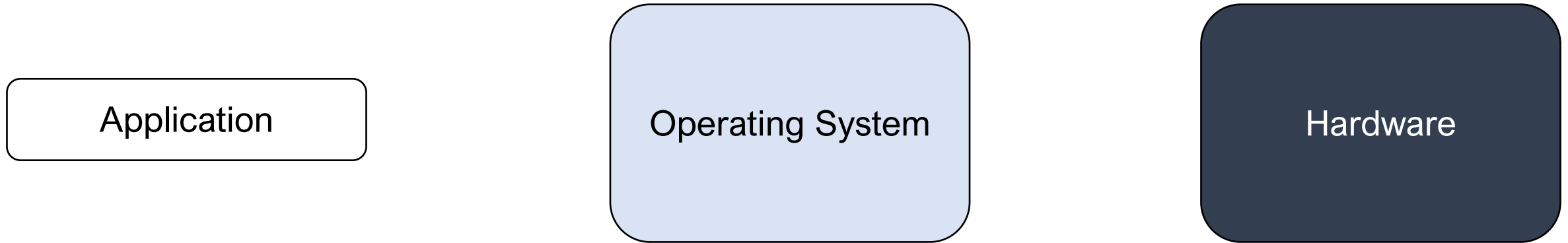
# Virtual Memory

# Virtual Memory Abstraction

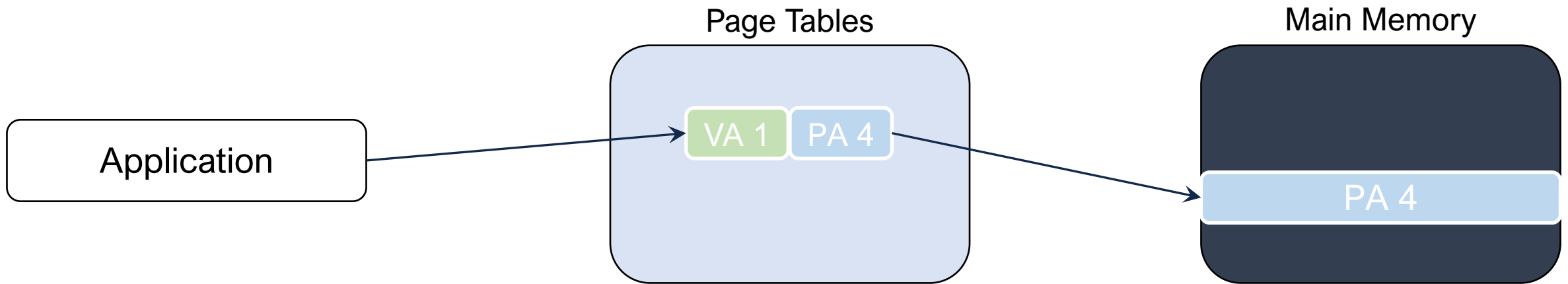




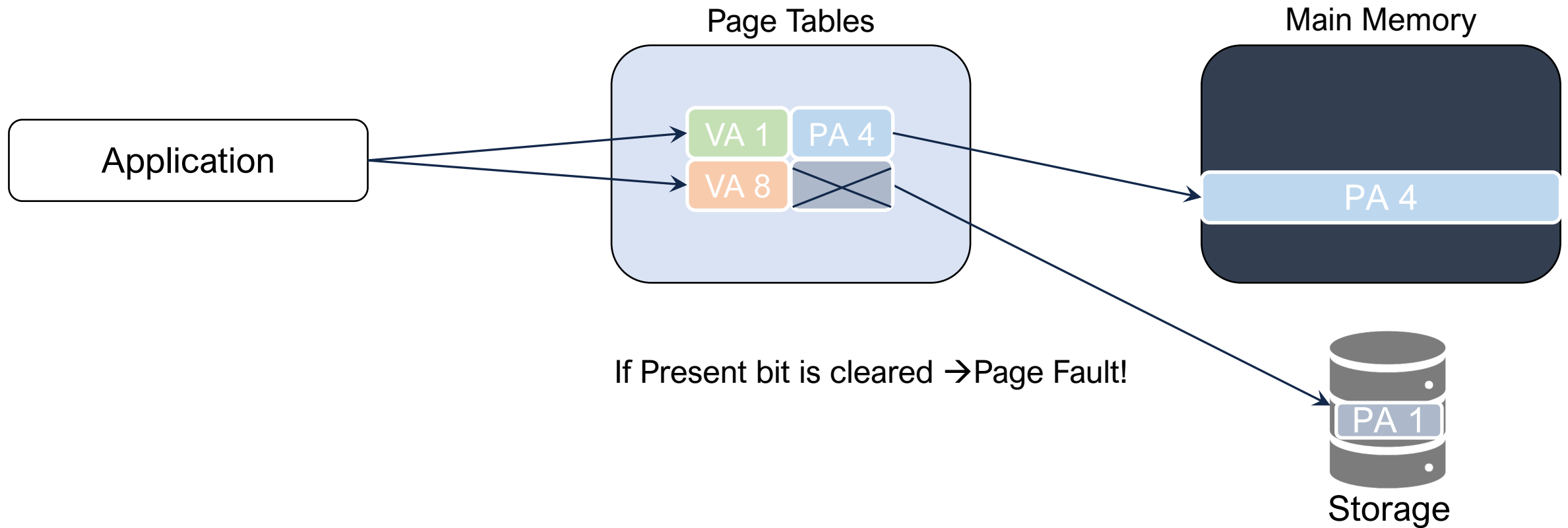
# Virtual Memory Abstraction



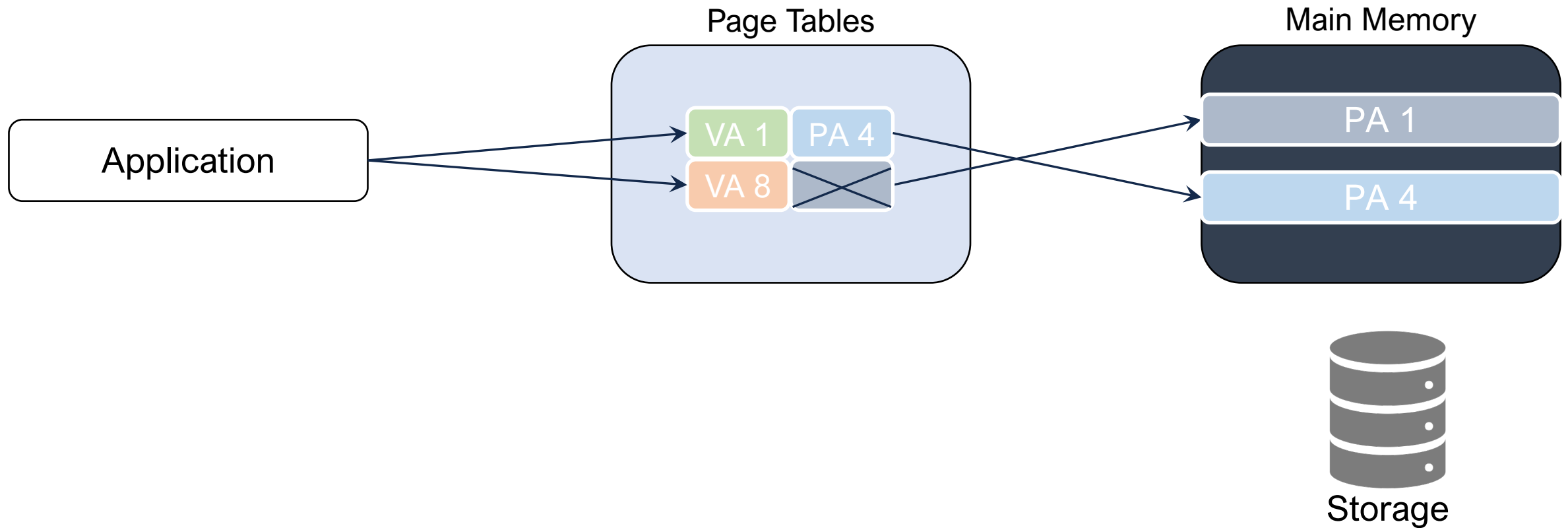
# Virtual Memory Abstraction



# Virtual Memory Abstraction



# Virtual Memory Abstraction

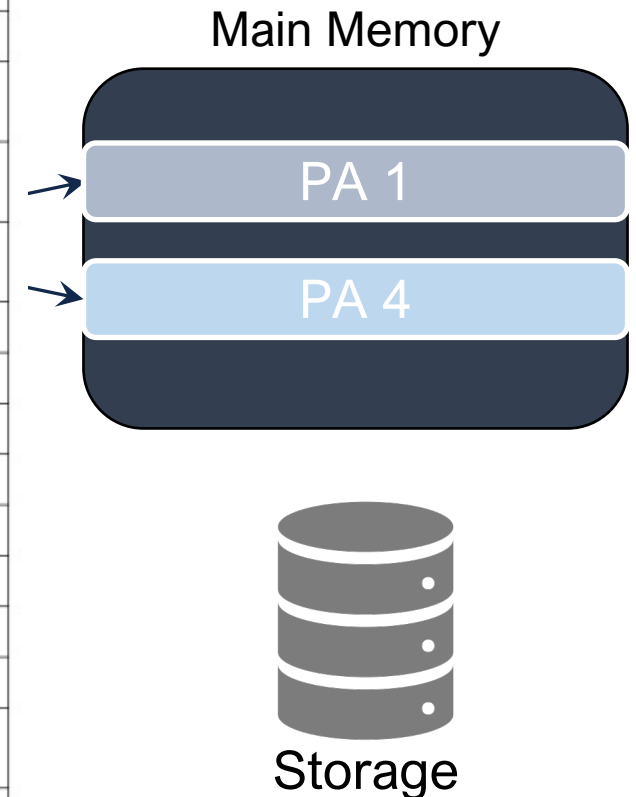


# Virtual Memory Abstraction

Table 4-20. Format of a Page-Table Entry that Maps a 4-KByte Page

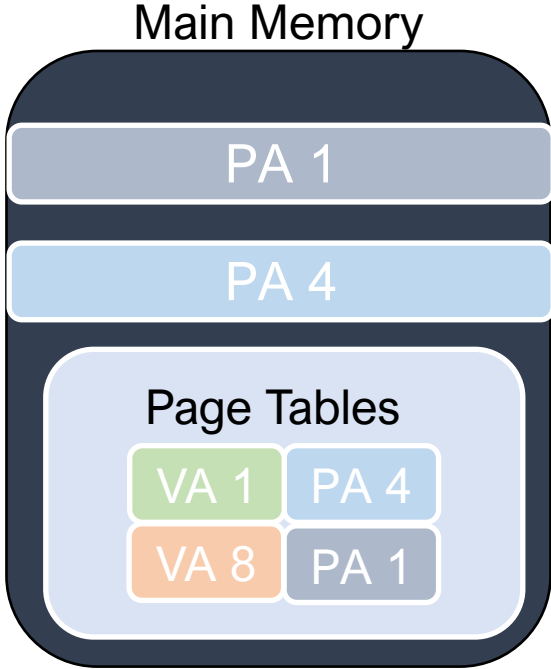
Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see 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)

Application

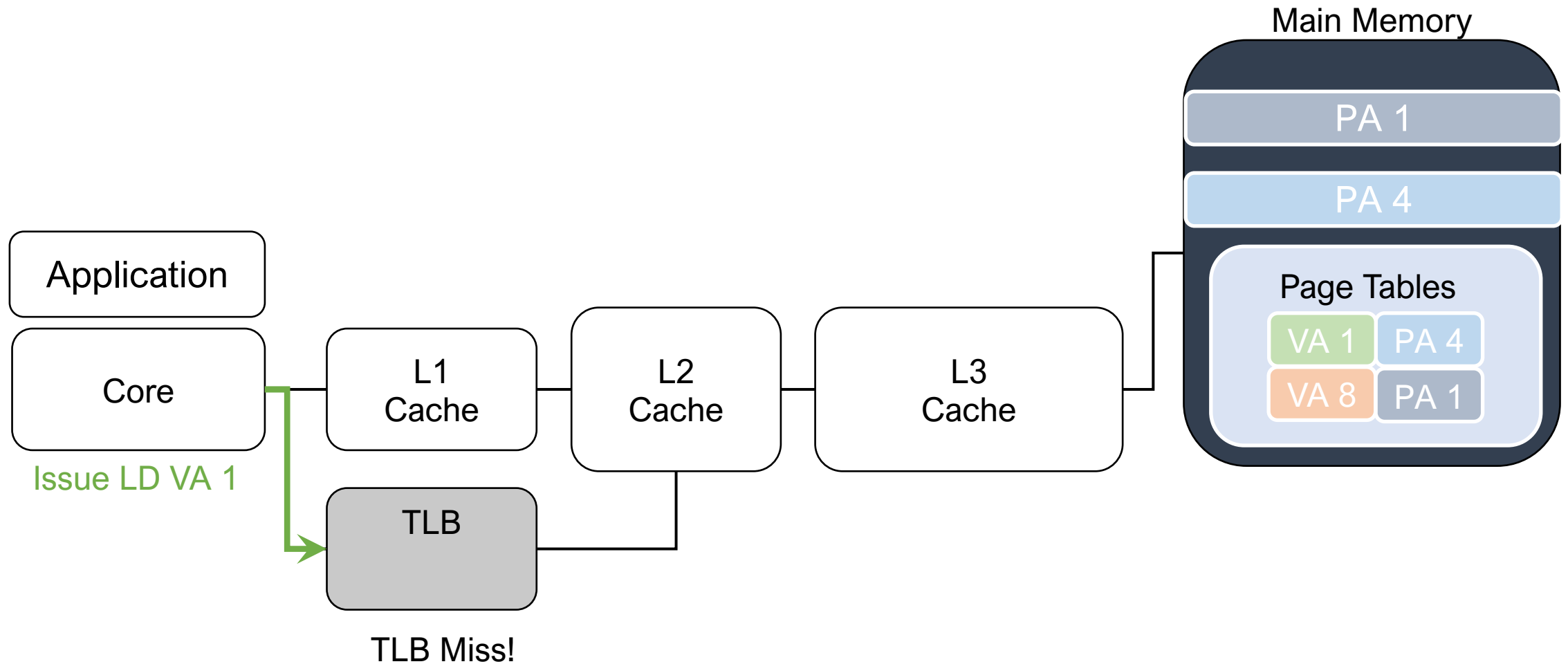


# Virtual Memory Abstraction

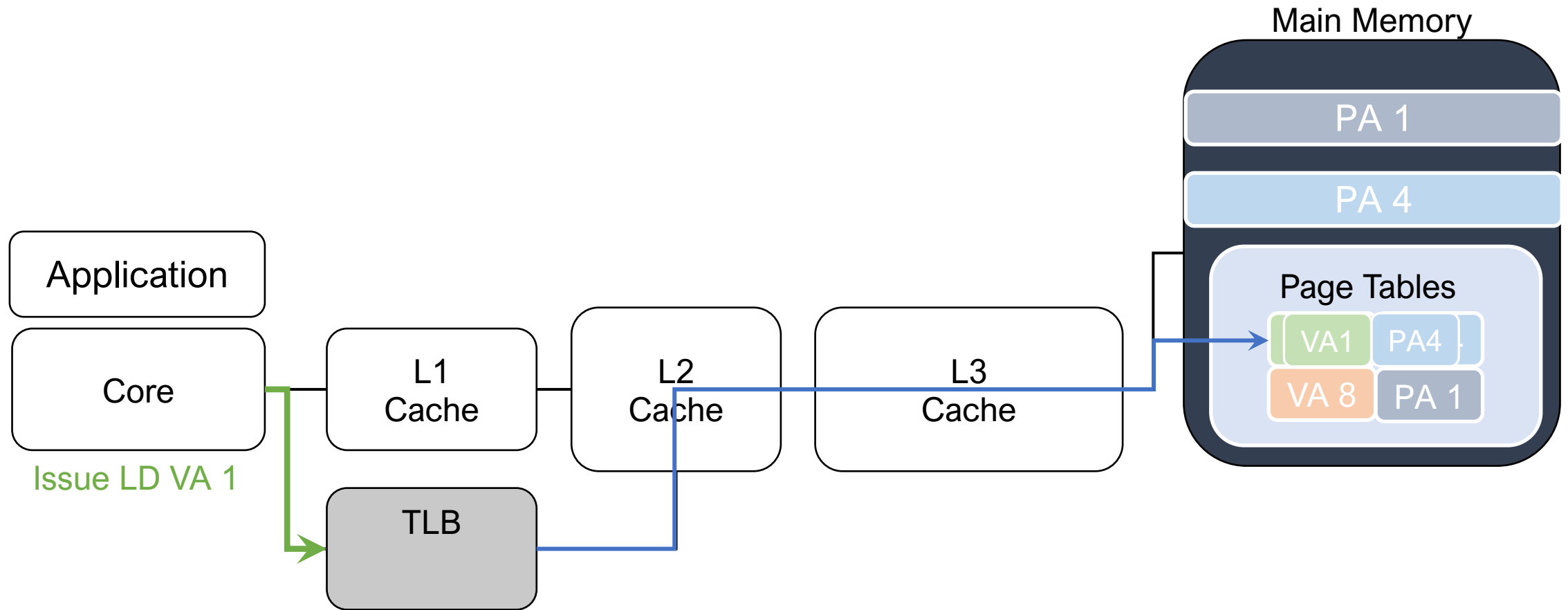
Application



# Virtual Memory Translation



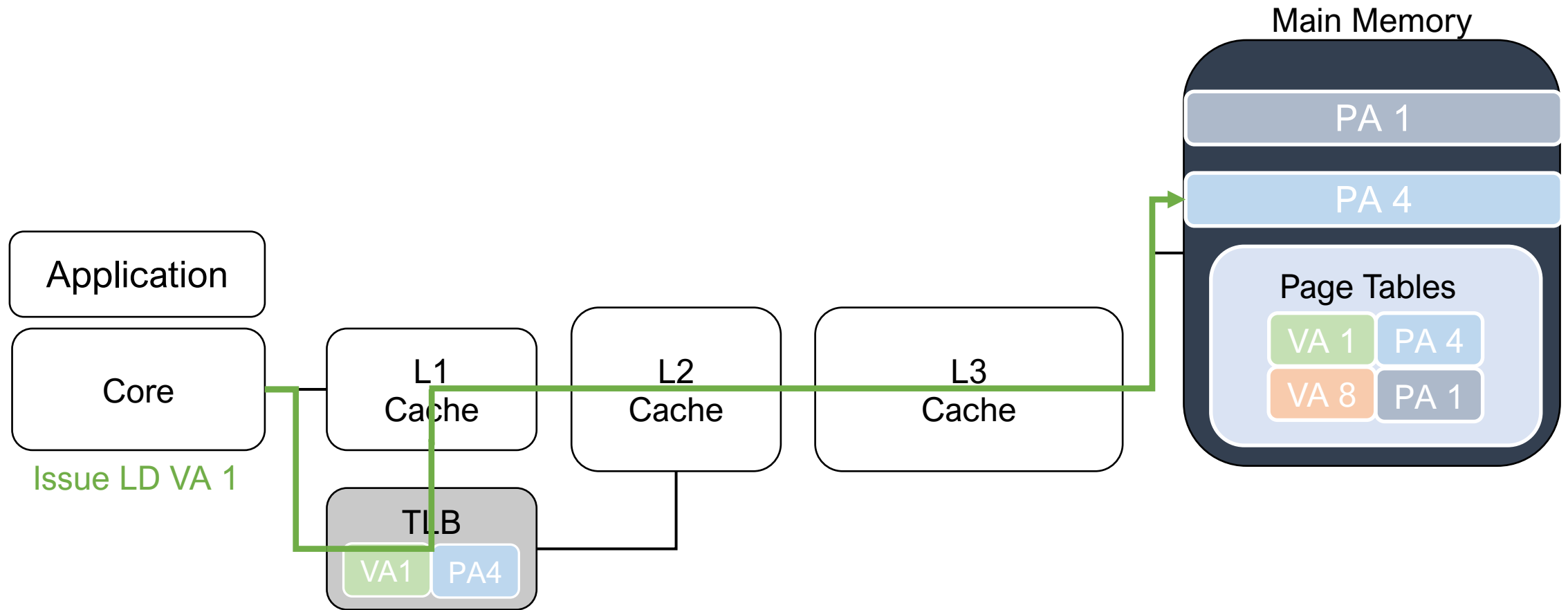
# Virtual Memory Translation



TLB Miss → “Page Walk” = Fetch entry from page table



# Virtual Memory Translation

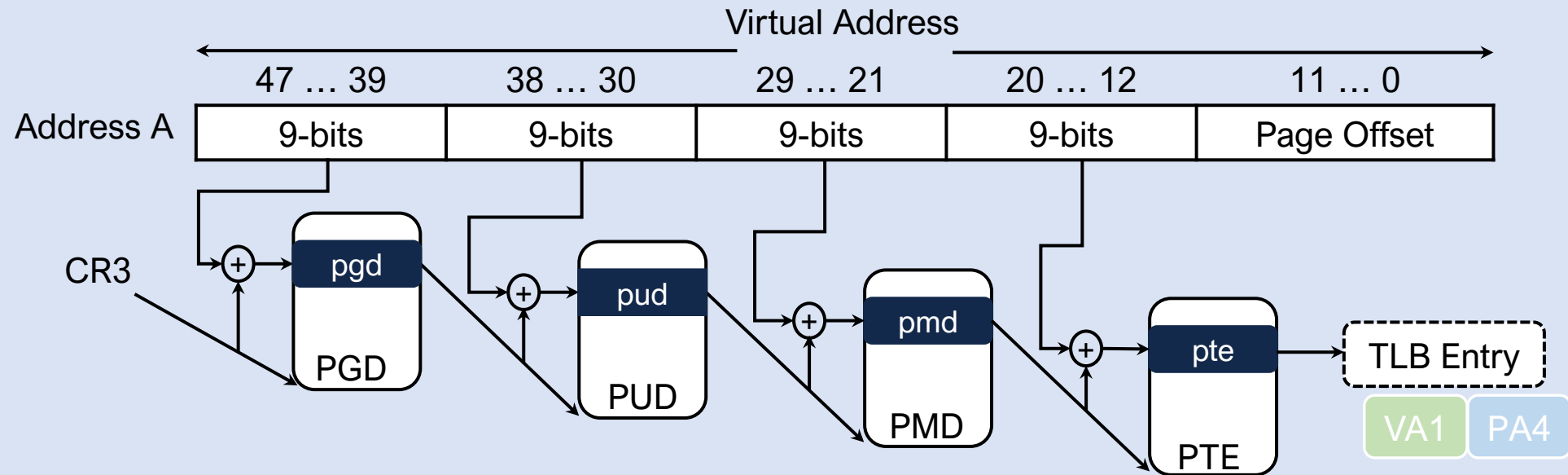


# Virtual Memory Translation

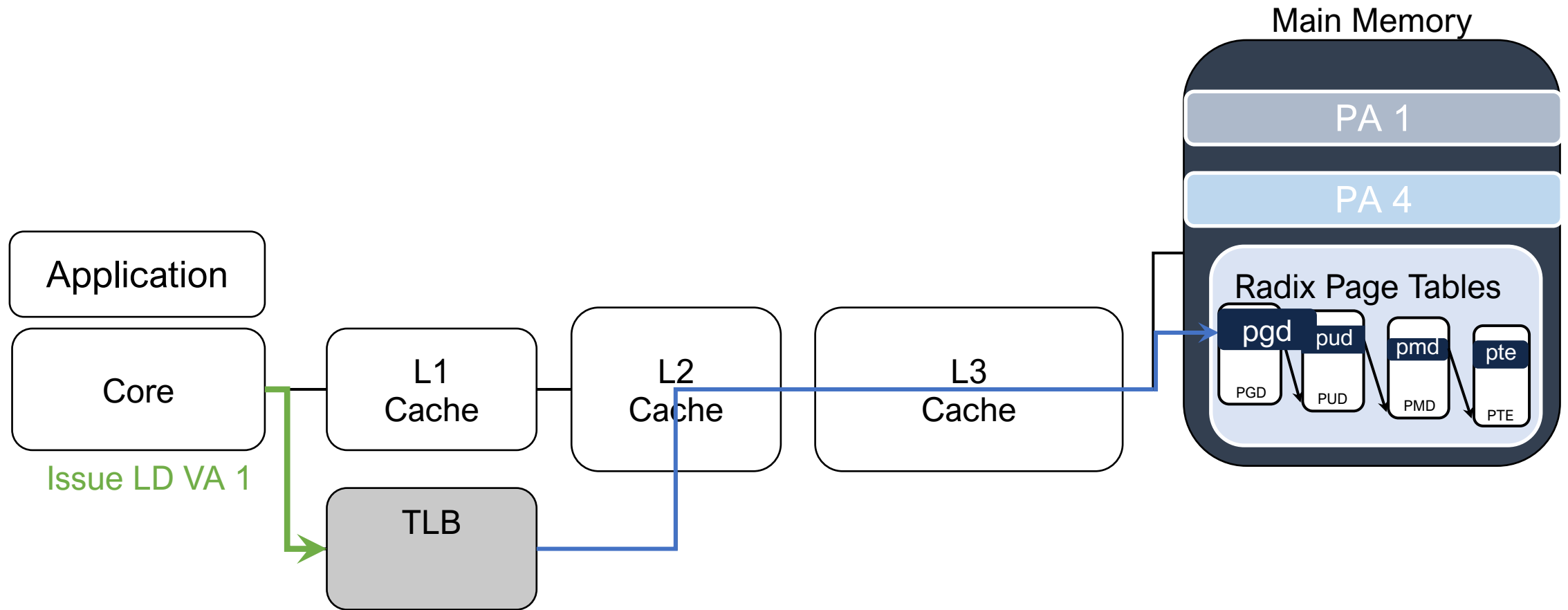
x86-64 Radix Page Tables

# Virtual Memory Translation

## x86-64 Radix Page Tables

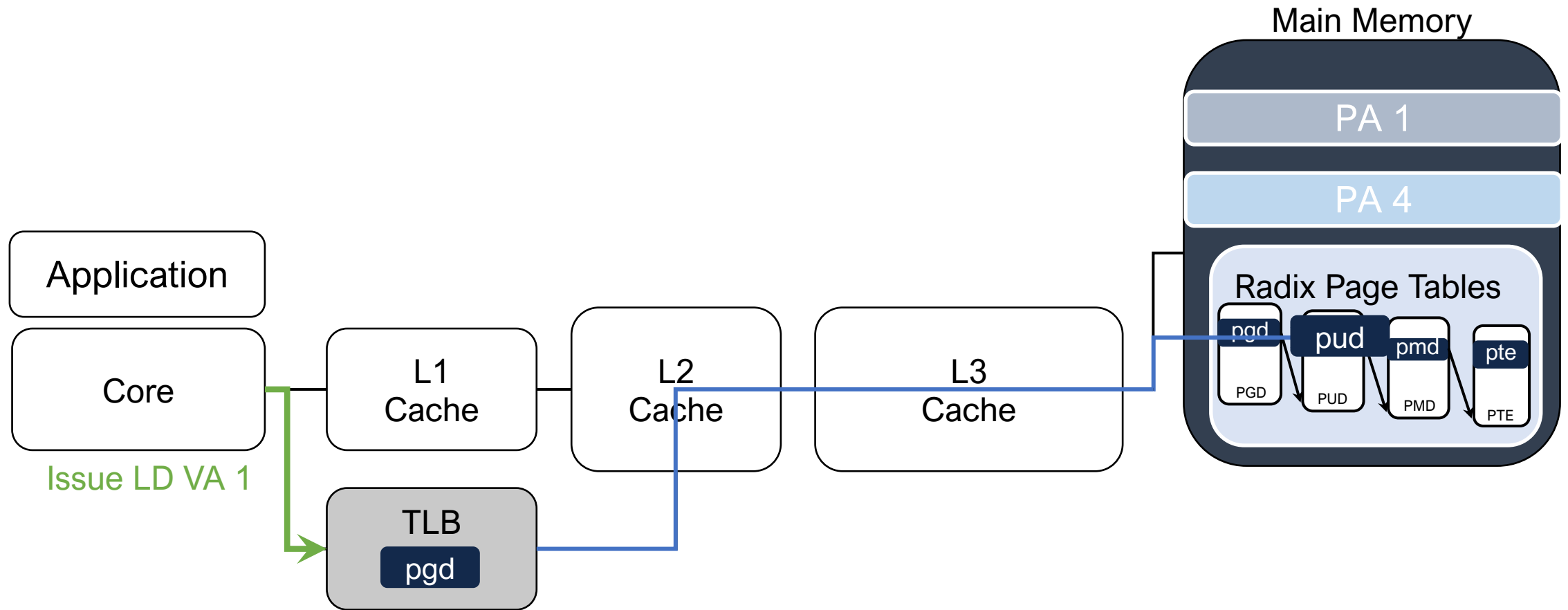


# Virtual Memory Translation



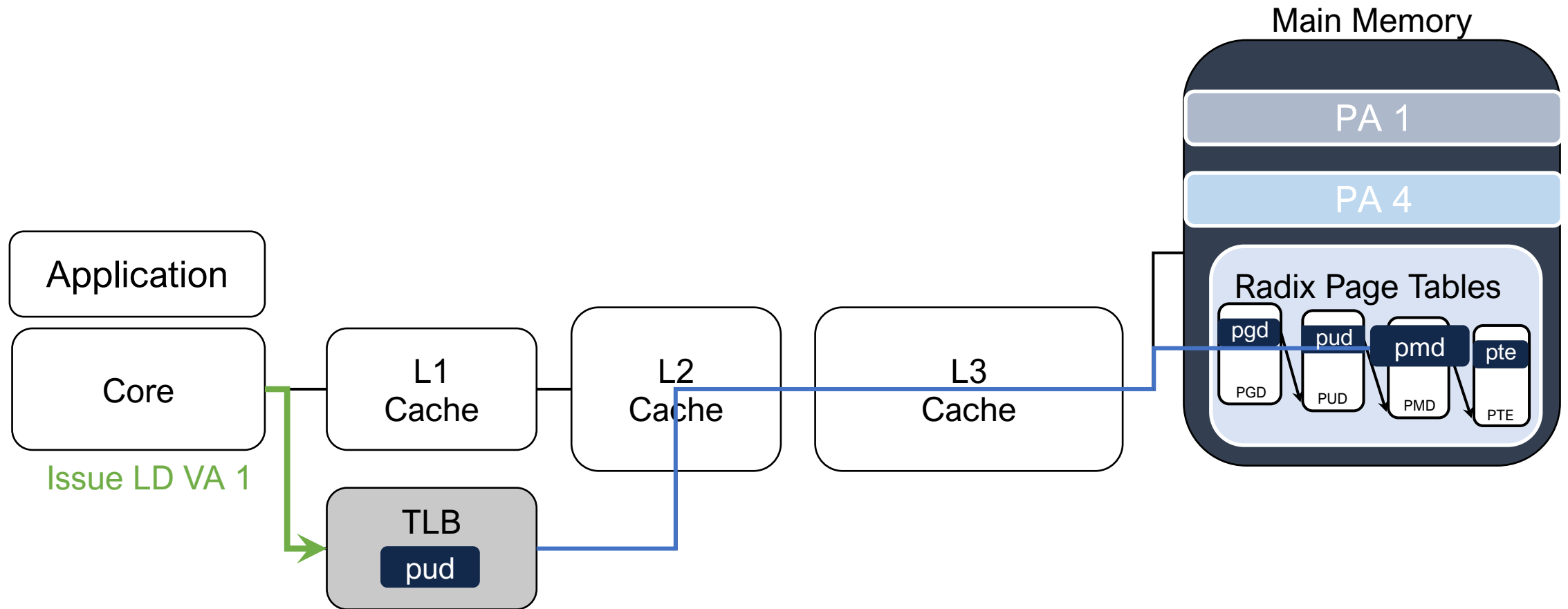
TLB Miss → **“Page Walk”** = Fetch entry from radix page table

# Virtual Memory Translation



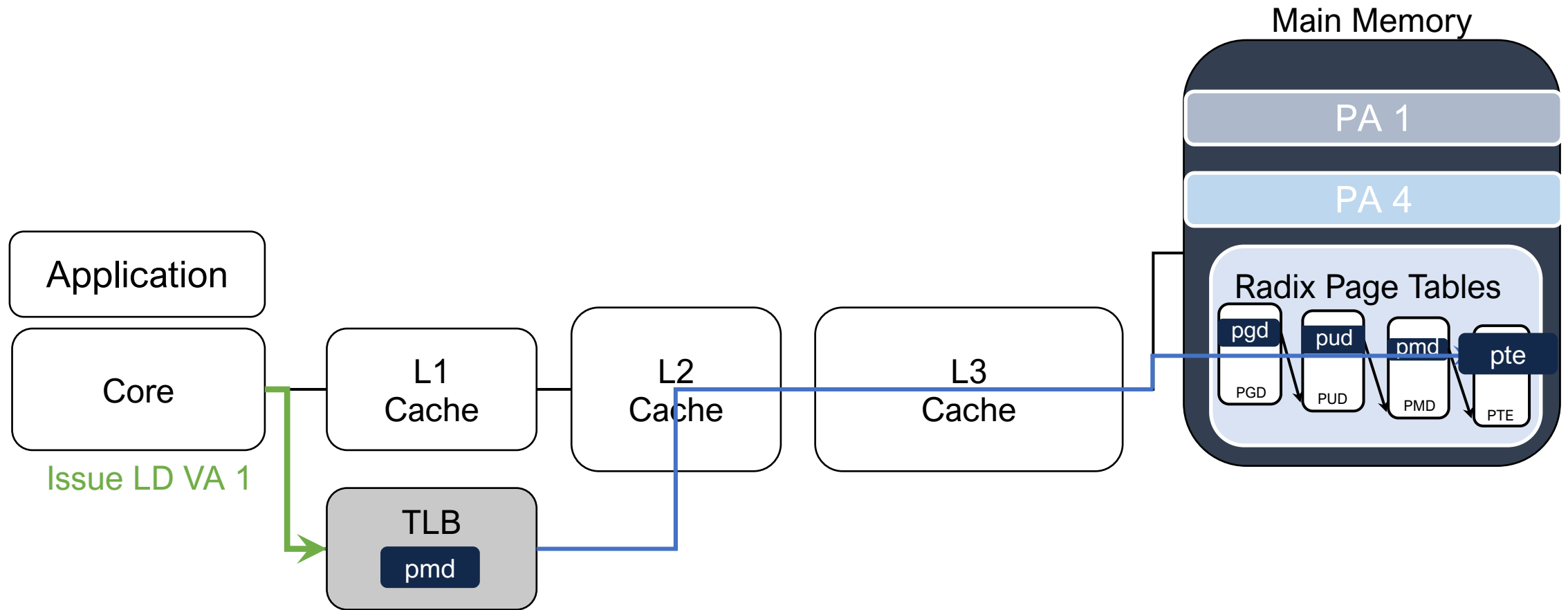
TLB Miss → **“Page Walk”** = Fetch entry from radix page table

# Virtual Memory Translation



TLB Miss → **“Page Walk”** = Fetch entry from radix page table

# Virtual Memory Translation

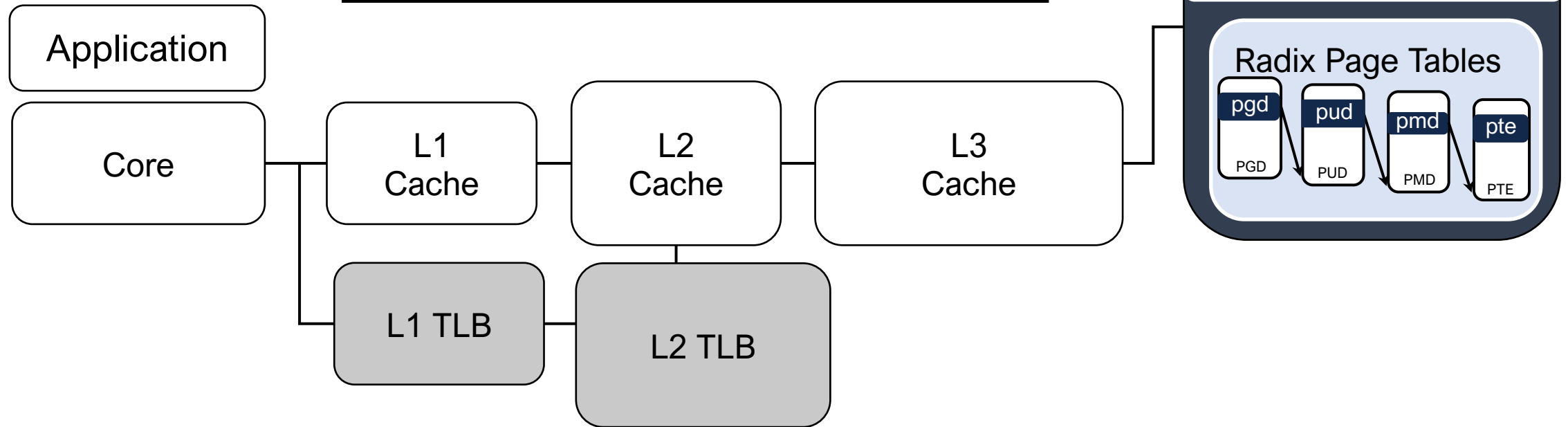


TLB Miss → **“Page Walk”** = Fetch entry from radix page table

# Multilevel TLBs

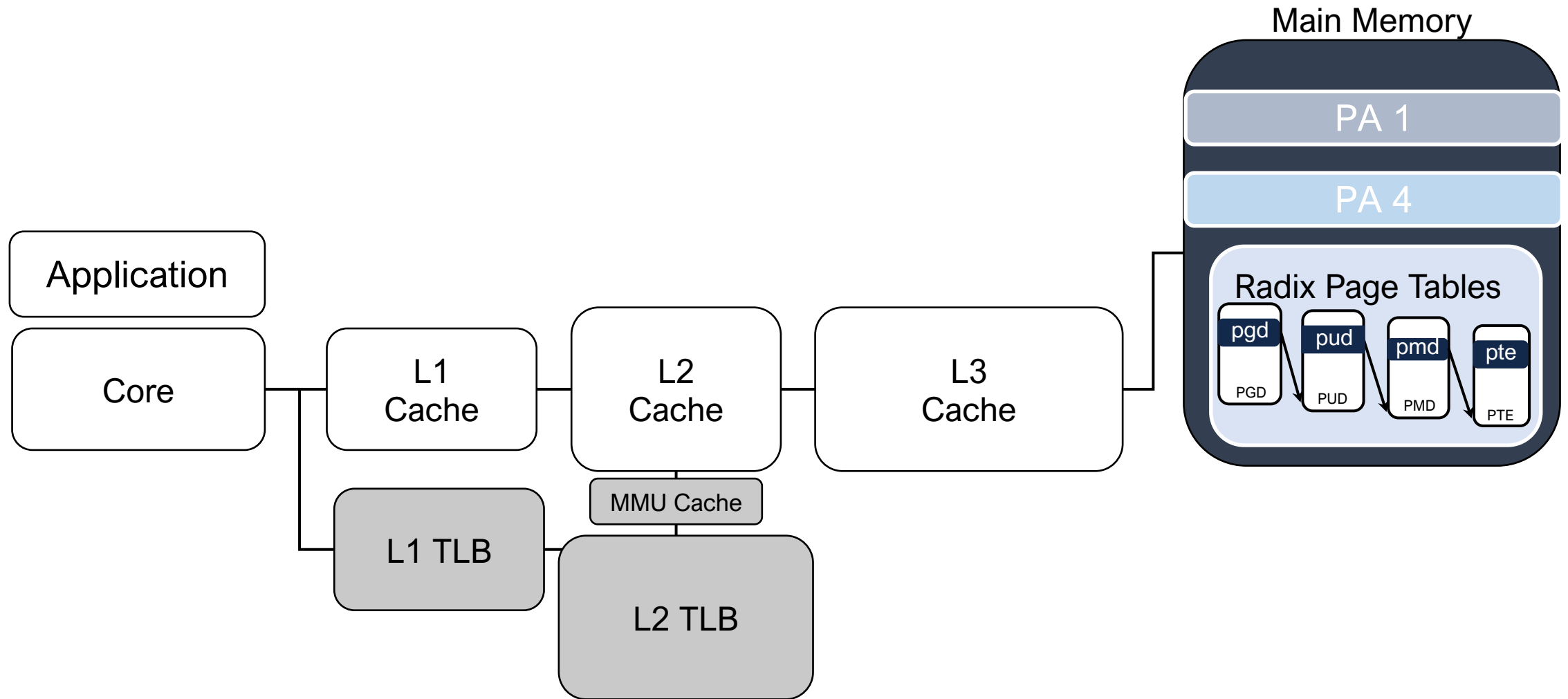
Intel i7 TLB structures

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

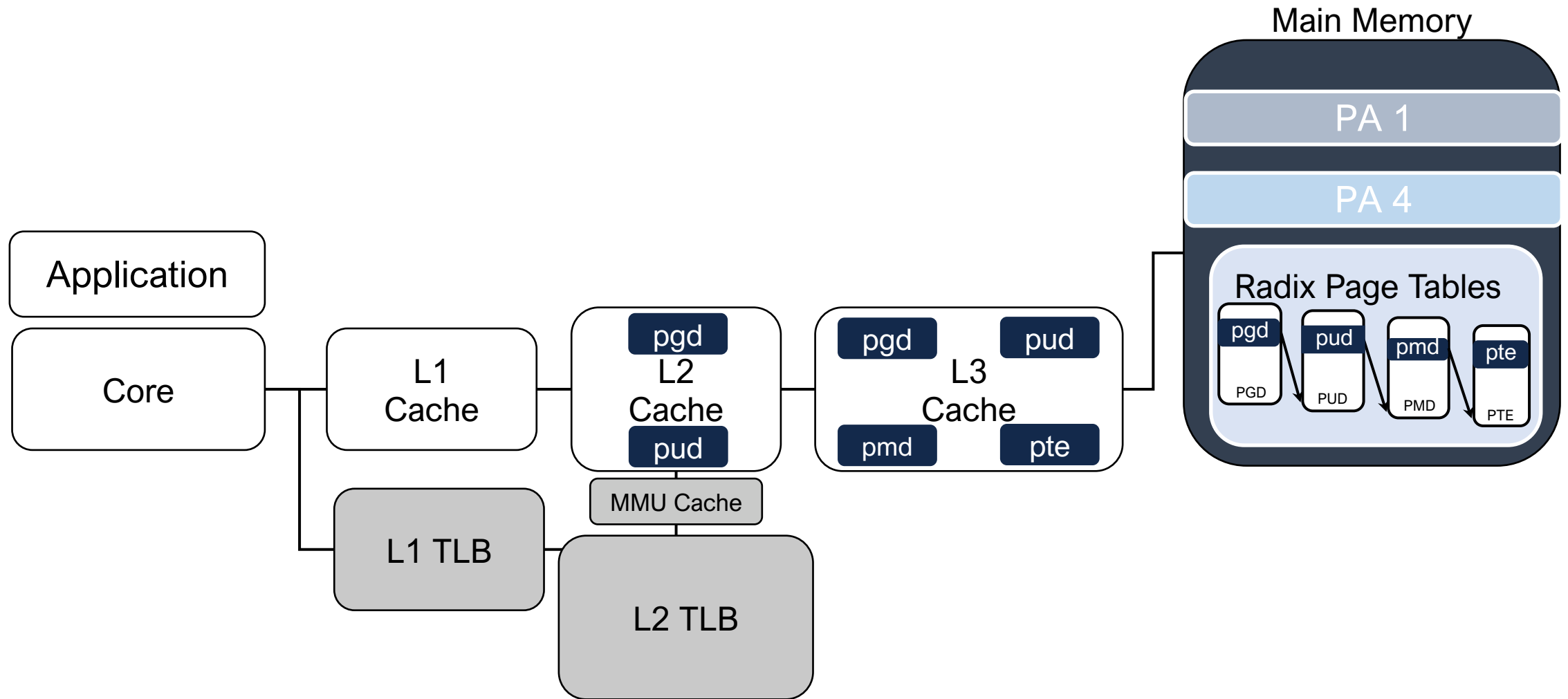





# Memory Management Unit Cache

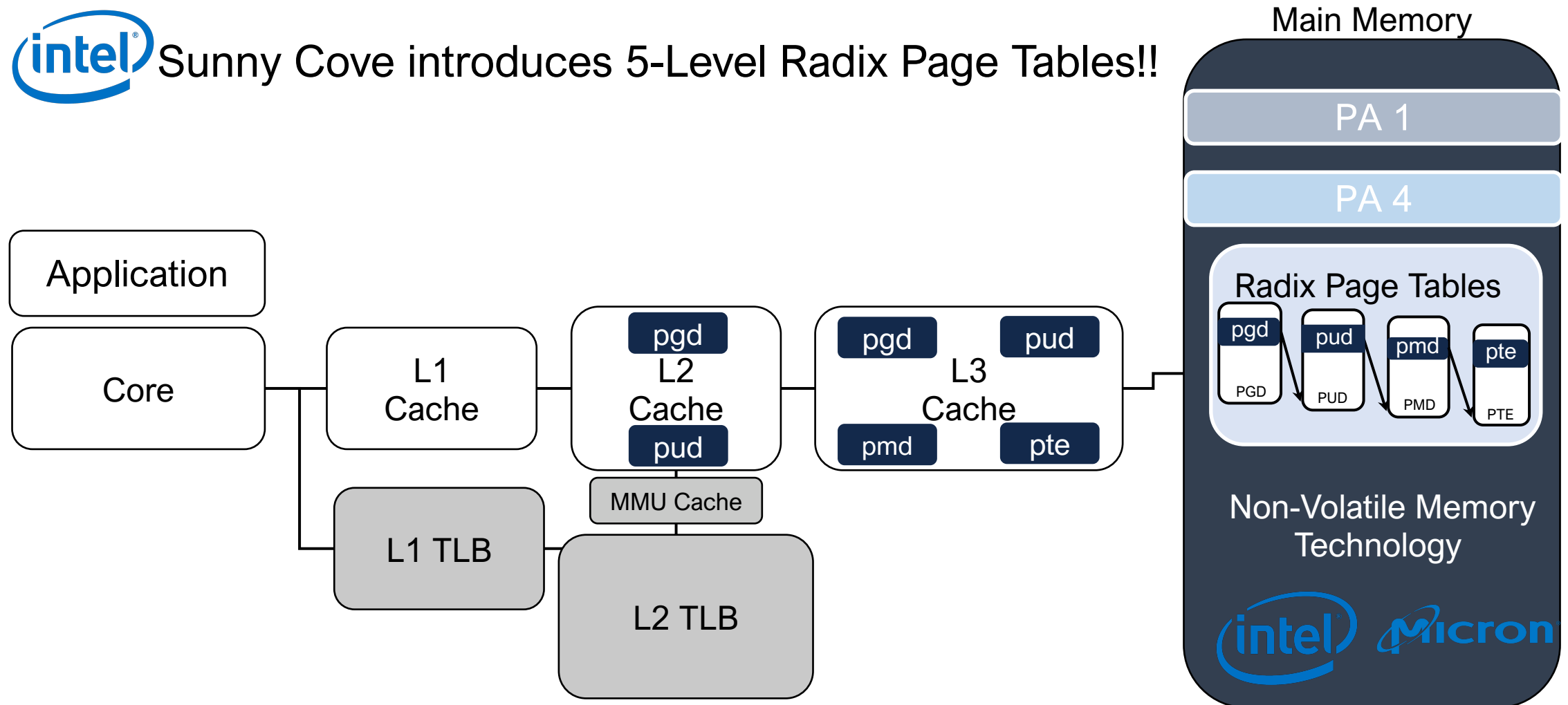


# Translations in Data Caches



# Even More Memory is Here!

 Sunny Cove introduces 5-Level Radix Page Tables!!



# Virtual Memory

## Protection via virtual memory

- Keeps processes in their own memory space

## Role of architecture

- Provide user mode and supervisor mode
- Protect certain aspects of CPU state
- Provide mechanisms for switching between user and supervisor modes
- Provide mechanisms to limit memory accesses
- Provide TLB to translate addresses

# 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:

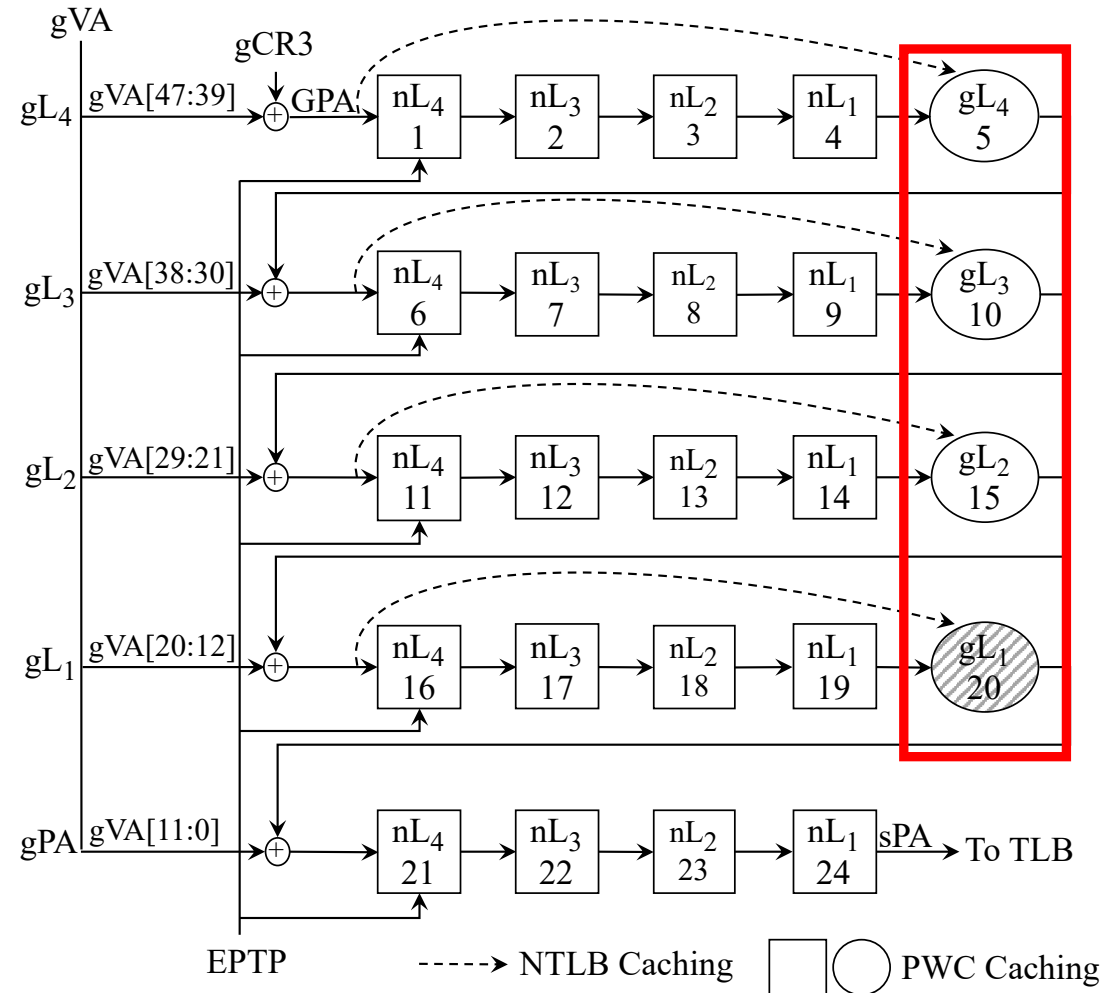
- System and user processor modes
- Privileged subset of instructions for allocating system resources

# 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

# Security Considerations

Present bit → Controlled Channel Attacks

Write & Execute permissions → Buffer overflow/Code Injection

Monitor MMU/Paging/MM → Several Side-channels (Leaky Cauldron)

Manipulate:

- Physical page number
- Data pages
- TLB-shootdowns

} Integrity violations

TLB → TLBleed, side-channel amplifier

# Next Up → Side-channels in the Cloud!

Check paper schedule

- <https://www.cs.cmu.edu/~15849/schedule.html>

Fill preference form

- <https://forms.gle/JZ93UQvwtepL9KKm7>

**Schedule will be finalized by Monday!**