ECE 350 Real-time Operating Systems

Lecture 8: Caching

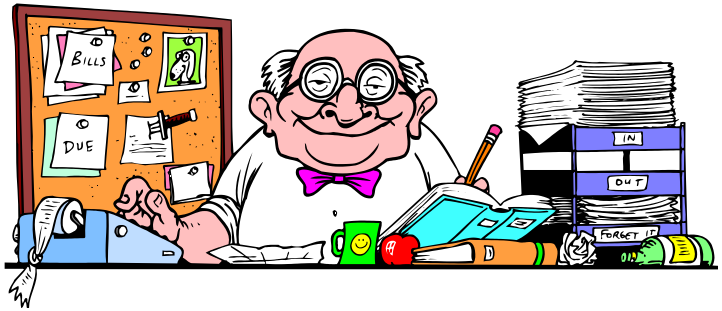
Prof. Seyed Majid Zahedi

https://ece.uwaterloo.ca/~smzahedi

Outline

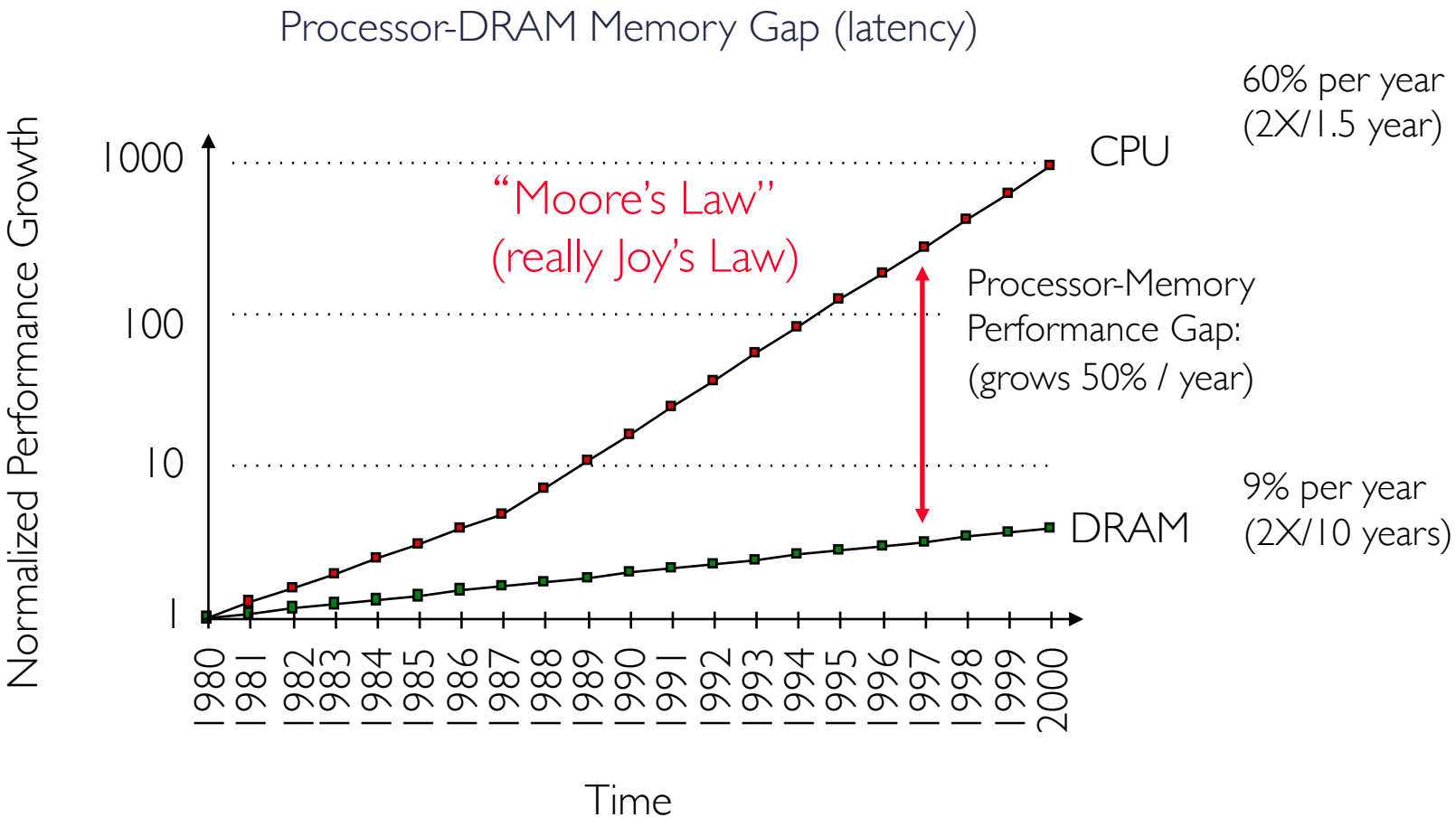
- Principle of locality
  - Temporal locality: Locality in time
  - Spatial locality: Locality in space
- Cache organizations
  - Direct mapped, set associative, fully associative
- Major categories of cache misses
  - Compulsory, conflict, capacity, coherence
- Translation lookaside buffer (TLB): caching applied to address translation
  - Cache relatively small number of PTEs
  - On TLB miss, page table is traversed

Caching Concept

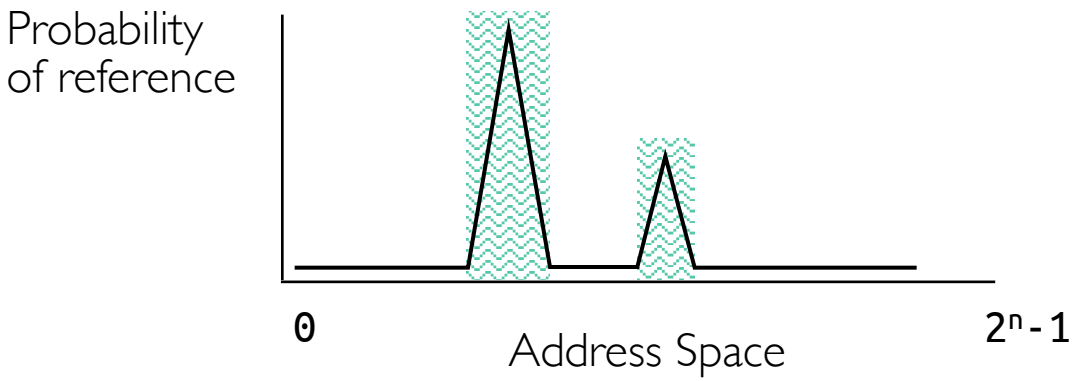


- Cache is repository for copies that can be accessed more quickly
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many techniques used today to make computers fast
  - We can cache memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Only good if
  - Frequent case is frequent enough and
  - Infrequent case is not too expensive

Why Bother with Caching?



Why Does Caching Help?



- Temporal locality (locality in time):
  - Cache recently accessed data items
- Spatial locality (locality in space):
  - Cache contiguous blocks

Some Terminology

- Block: group of spatially contiguous and aligned bytes (words)
  - Typical sizes are 32B, 64B, 128B
- Hit: access cache and find what we want
  - Hit time: time to hit (or discover miss)
- Miss: access cache and fail to find what we want
  - Miss time: time to satisfy miss
  - Misses are expensive (take a long time) ⇒ try to avoid them
  - But, if they happen, amortize their costs ⇒ bring in more than just specific word you want ⇒ bring in whole block (multiple words)

Some Terminology (cont.)

- Hit rate = num of hits / (num of hits + num of misses)
  - Miss rate = 1 – hit rate
  - High hit rate means high probability of finding what we want

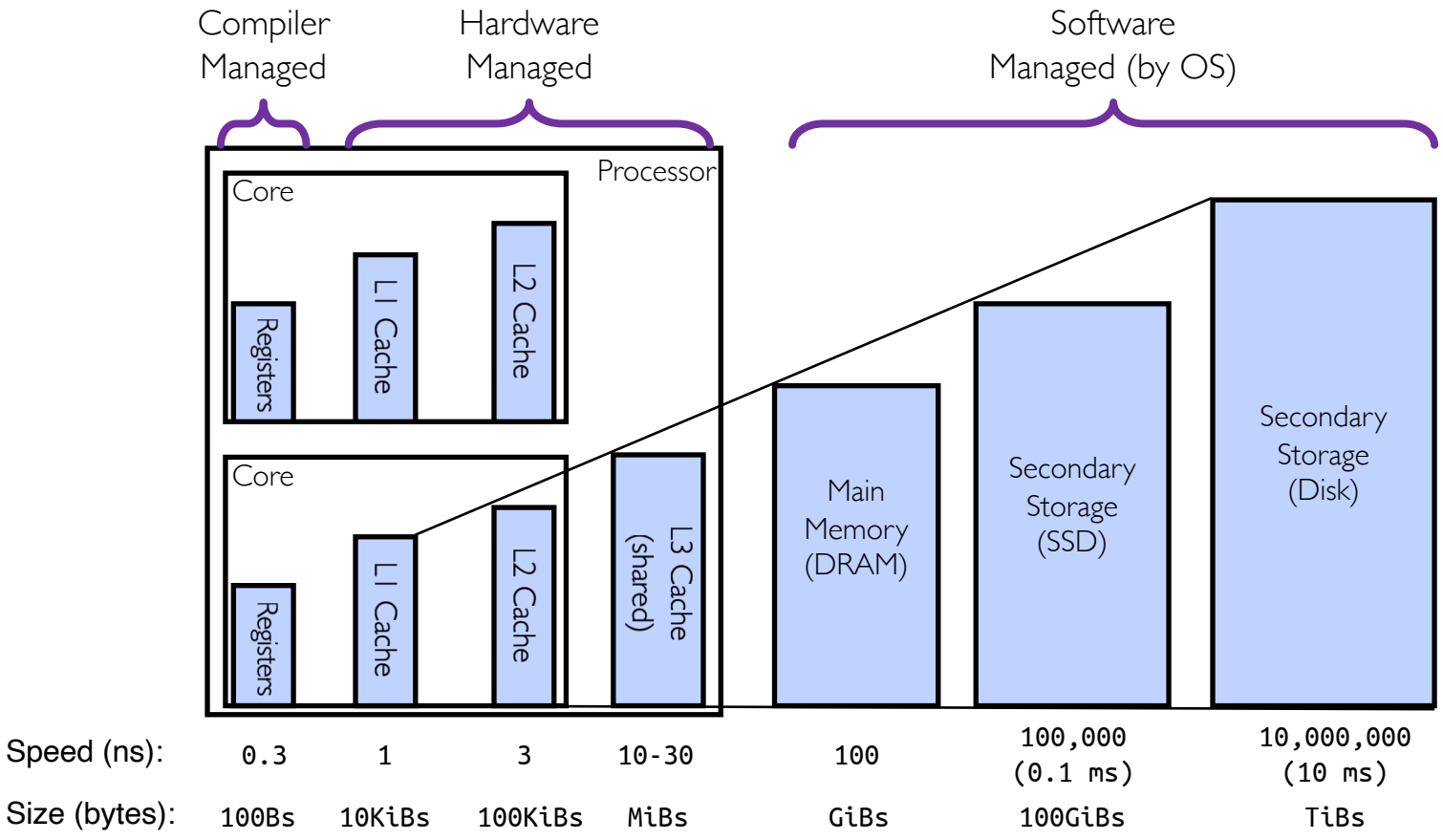
- Average access time = hit rate x hit time + miss rate x (hit time + miss time)
  = hit time + miss rate x miss time

- Problem: hard to get low hit time and miss rate in one memory structure
  - Large memory structures have low miss rate but high hit time
  - Small memory structures have low hit time but high miss rate

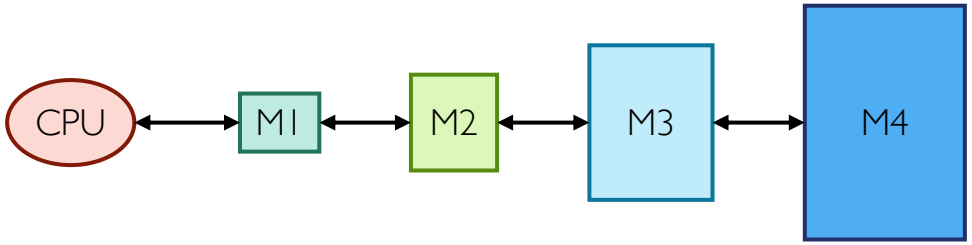
- Solution: use hierarchy of memory structures

Memory Hierarchy of Modern Computer Systems

- Goal: bring average memory access time close to L1's



Abstract Hierarchy Performance



Miss time at level X = Average access time at level X + 1

Avg. memory access time = Avg-time_{M1}

= $\text{Hit-time}_{M1} + (\text{Miss-ratio}_{M1} \times \text{Miss-time}_{M1})$

= $\text{Hit-time}_{M1} + (\text{Miss-ratio}_{M1} \times \text{Avg-time}_{M2})$

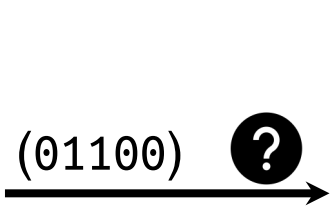
= $\text{Hit-time}_{M1} + (\text{Miss-ratio}_{M1} \times (\text{Hit-time}_{M2} + (\text{Miss-ratio}_{M2} \times \text{Miss-time}_{M2})))$

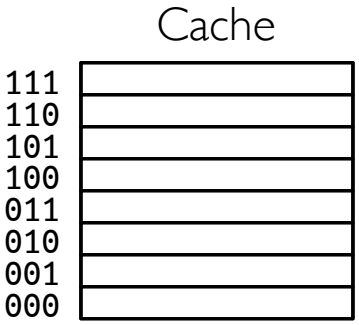
= $\text{Hit-time}_{M1} + (\text{Miss-ratio}_{M1} \times (\text{Hit-time}_{M2} + (\text{Miss-ratio}_{M2} \times \text{Avg-time}_{M3})))$

= …

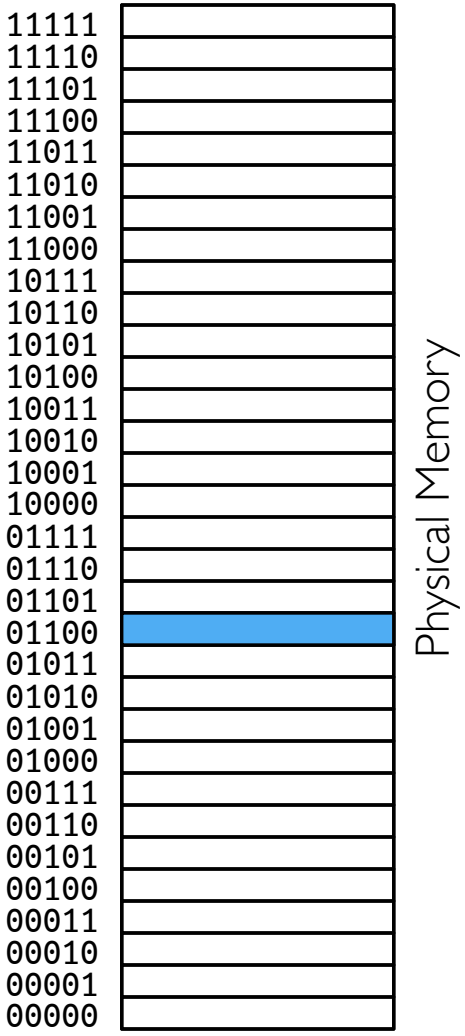
Caching Questions

- **8-byte** cache, **32-byte** memory, 1 block = **1 byte**
- Assume CPU accesses **01100**



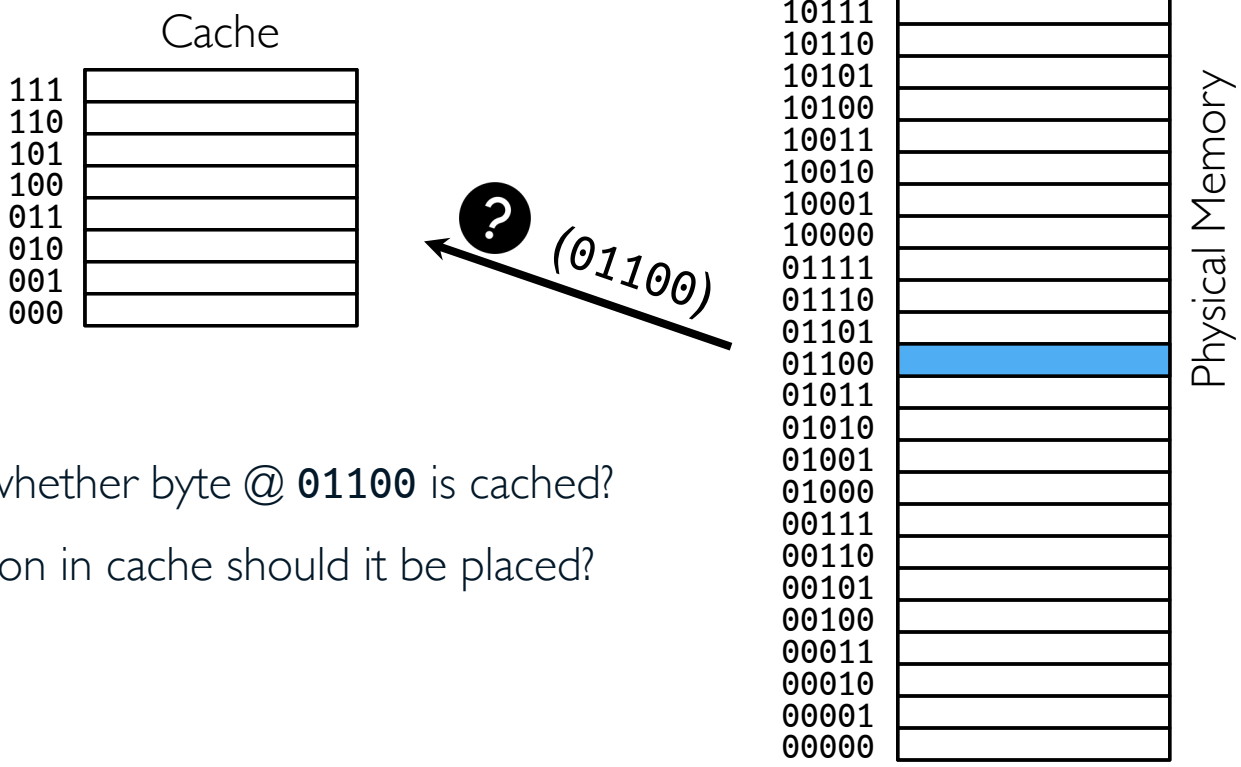


- How do you know whether byte @ **01100** is cached?



Caching Questions (cont.)

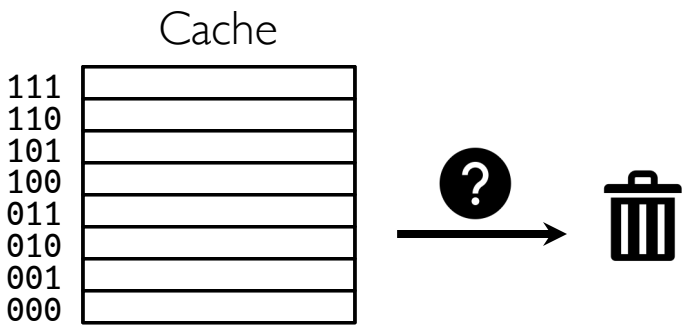
- **8-byte** cache, **32-byte** memory, 1 block = **1 byte**
- Assume CPU accesses **01100**

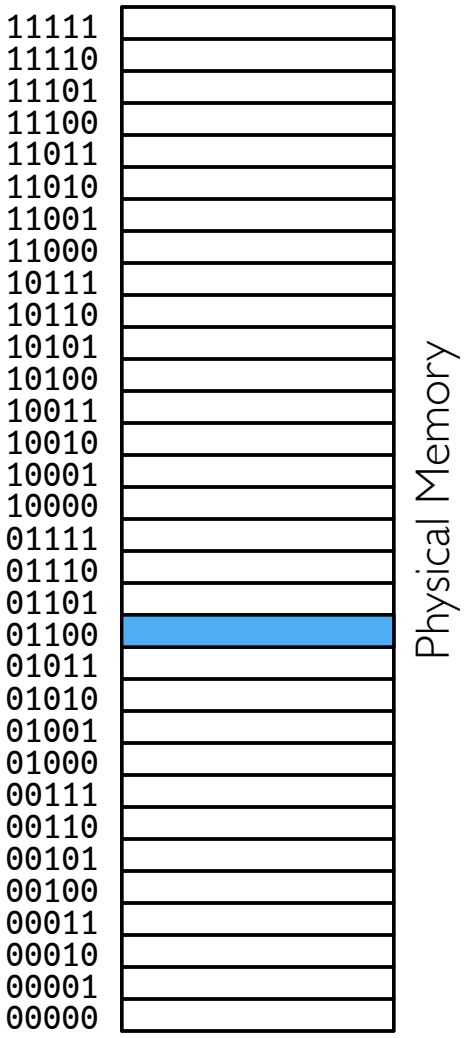


- How do you know whether byte @ **01100** is cached?
- If not, at which location in cache should it be placed?

Caching Questions (cont.)

- **8-byte** cache, **32-byte** memory, 1 block = **1 byte**
- Assume CPU accesses **01100**





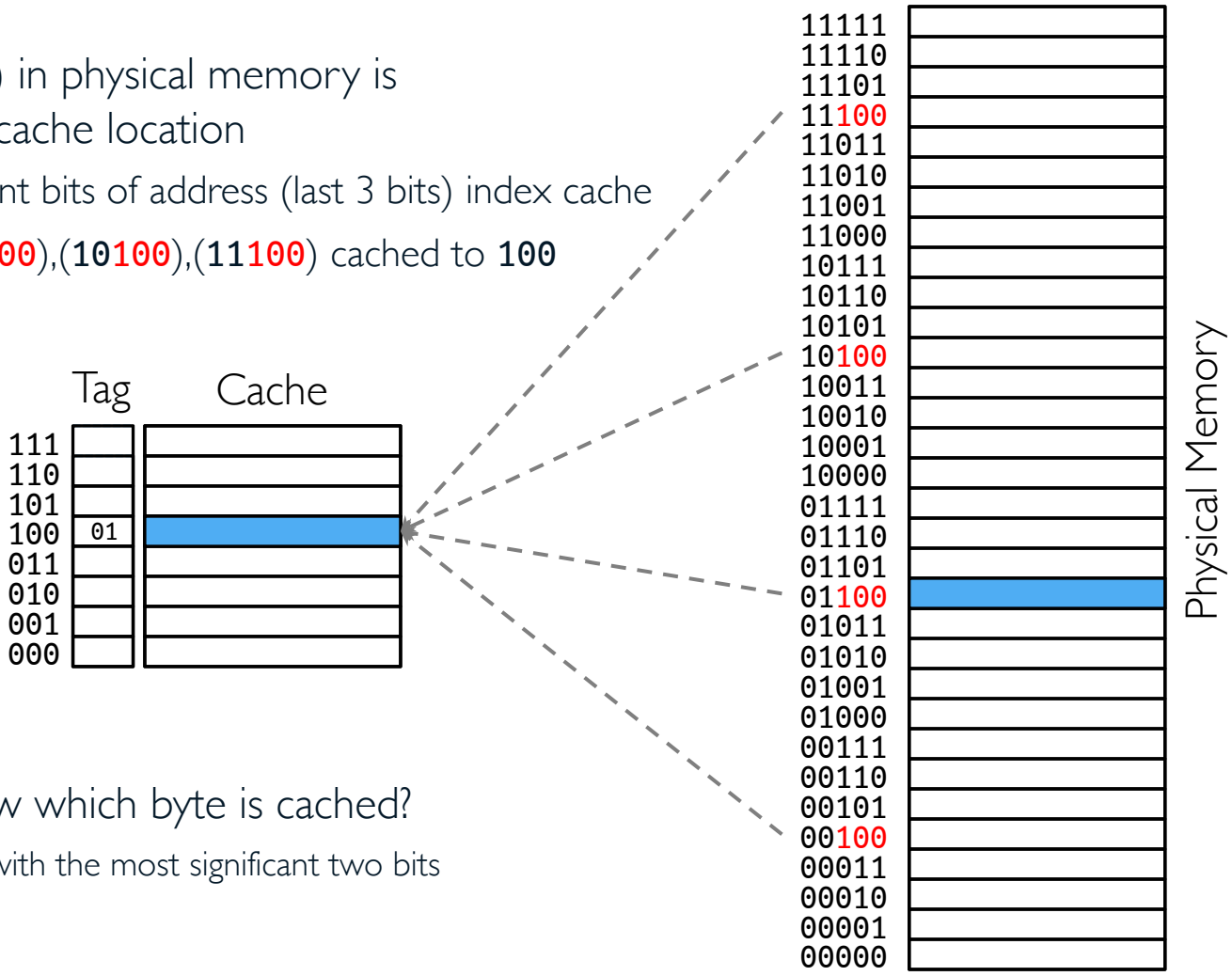
- How do you know whether byte @ **01100** is cached?
- If not, at which location in cache should it be placed?
- If cache is full, which cached byte should be evicted?

Where to Put Blocks in Cache?

- Divide cache into disjoint sets of blocks
  - There is 1-to-1 mapping from block address to set
- M-way set-associative cache: each set holds M number of blocks
  - E.g., 4 blocks per set ⇒ 4-way set-associative cache
- Fully-associative cache: whole cache has just one set
  - + Most flexible
  - − Longest access latency
- Direct-mapped cache: each set has one block (= 1-way set-associative)
  - + Least flexible
  - − Shortest access latency

Example: Direct-mapped Cache

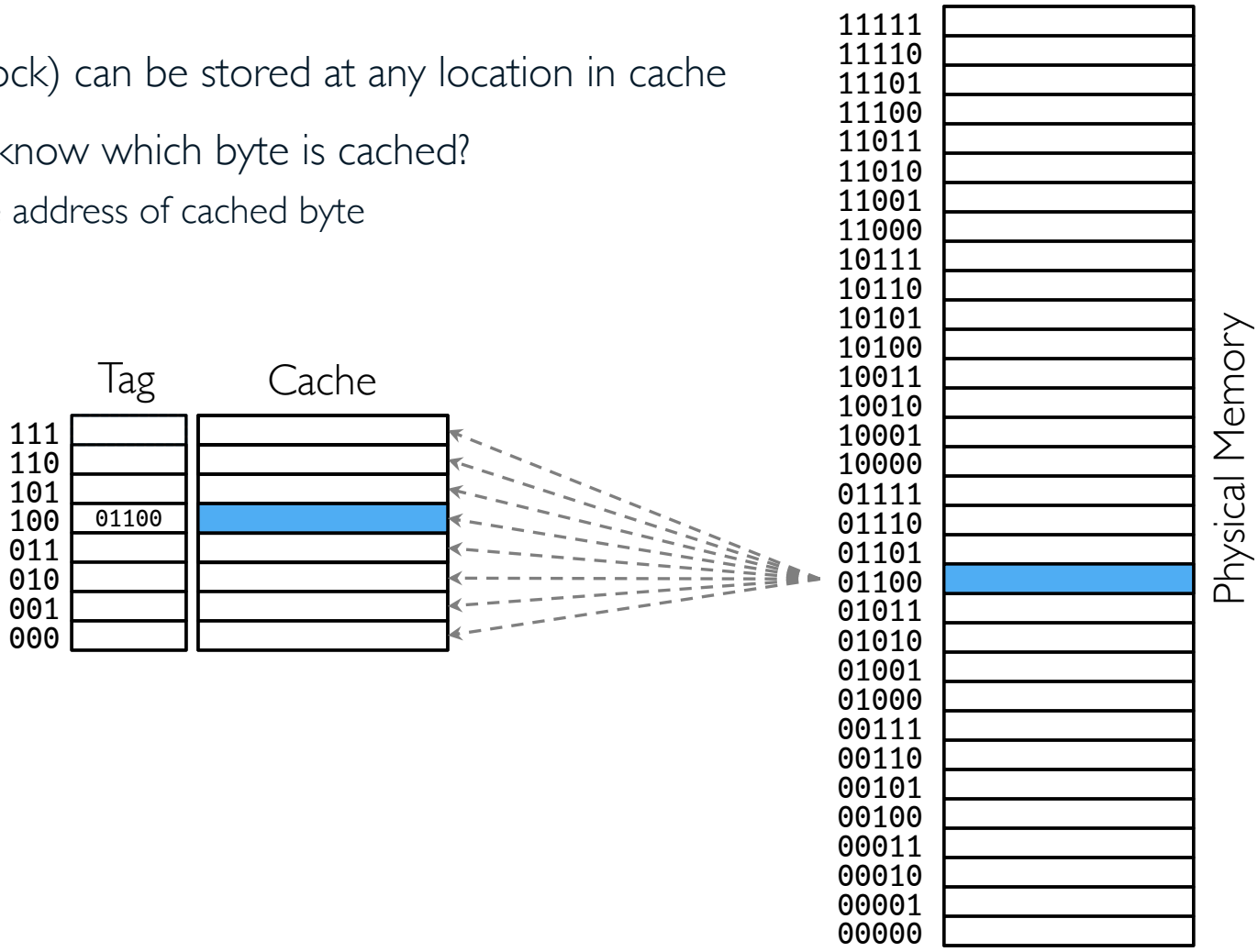
- Each byte (block) in physical memory is cached to single cache location
  - Least significant bits of address (last 3 bits) index cache
  - (**00100**),(**01100**),(**10100**),(**11100**) cached to **100**



- How do we know which byte is cached?
  - Tag each byte with the most significant two bits

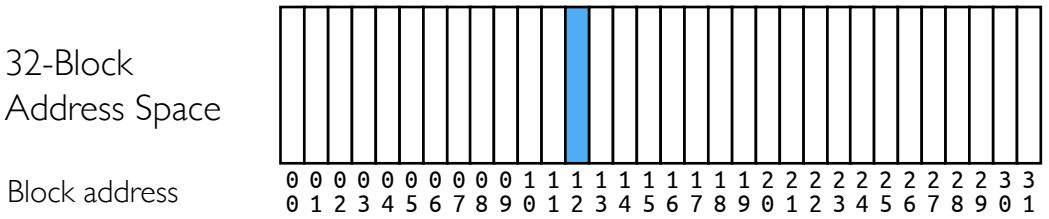
Example: Fully-associative Cache

- Each byte (block) can be stored at any location in cache
- How do you know which byte is cached?
  - Tag entire address of cached byte

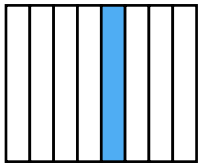


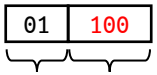
Where to Put Blocks in Cache? (cont.)

- Example: where is block 12 placed in 8-block cache?

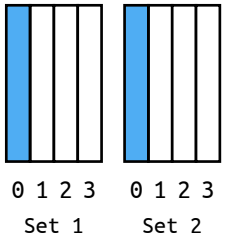


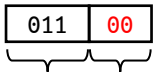
Direct mapped
Block 12 can go only into block 4 (12 mod 8)



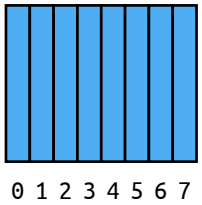


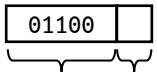
2-way set-associative
Block 12 can go anywhere in set 0 (12 mod 4)



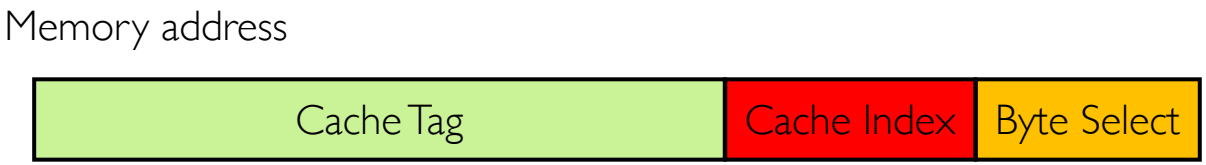


Fully associative
Block 12 can go anywhere





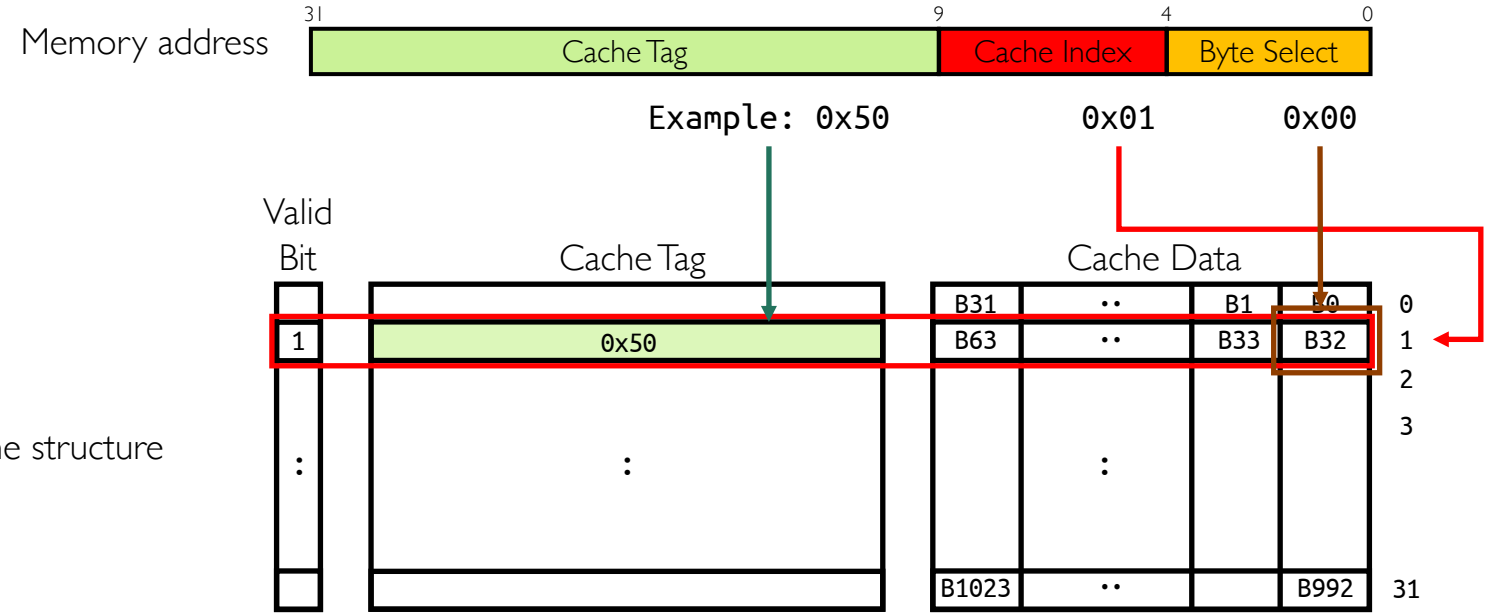
How is Block Found in Cache?



- Byte select field used to select data within block
  - Offset of byte in block
- Cache index used to lookup candidate blocks in cache
  - Index identifies set
- Cache tag used to identify actual copy among candidate blocks
  - If no candidate matches, then declare cache miss

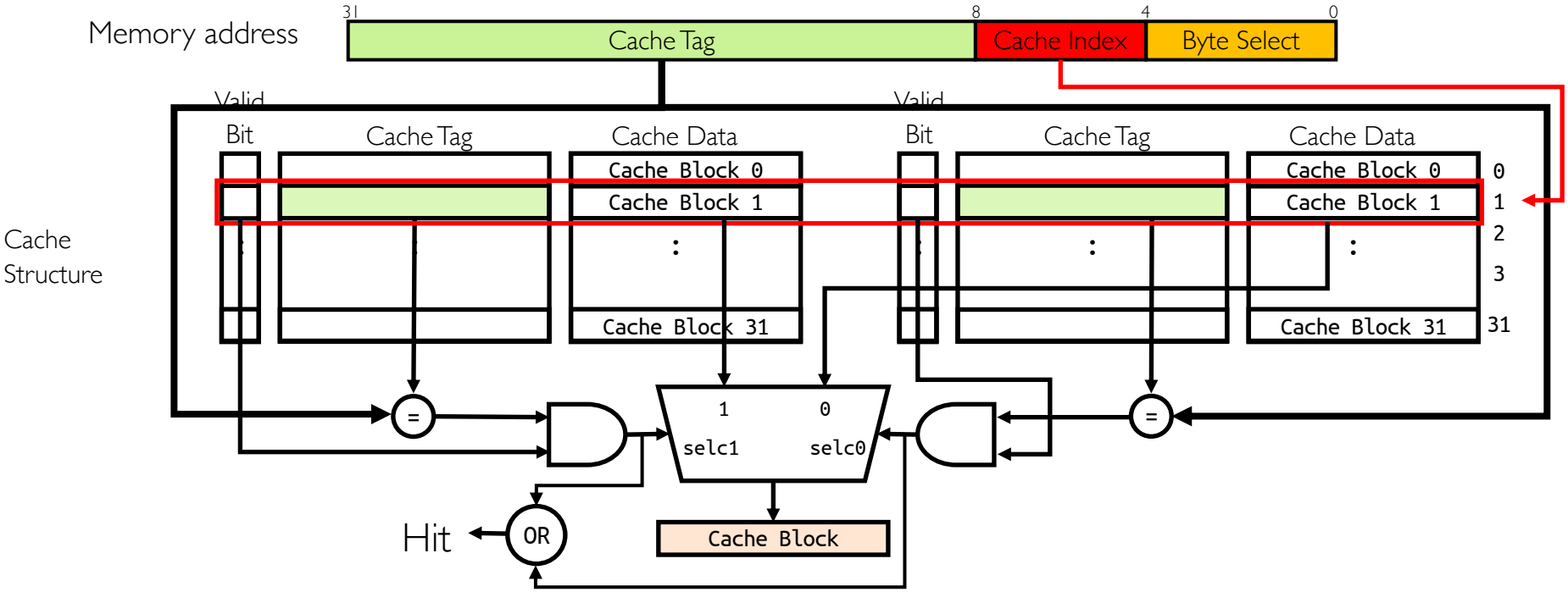
Direct Mapped Cache

- Direct mapped 2^N **byte** cache with block size of 2^M **bytes**
  - Uppermost (**32 - N**) **bits** of address are cache tag
  - Lowest **M bits** are byte select, rest are cash index
- Example: **1KiB** direct mapped cache with **32B** blocks
  - Log_2**32 = 5 bits** for byte select, **32 –** Log_2**1024 = 22 bits** for cache tag
  - **32 – 5 – 22 = 5 bits** for cache index



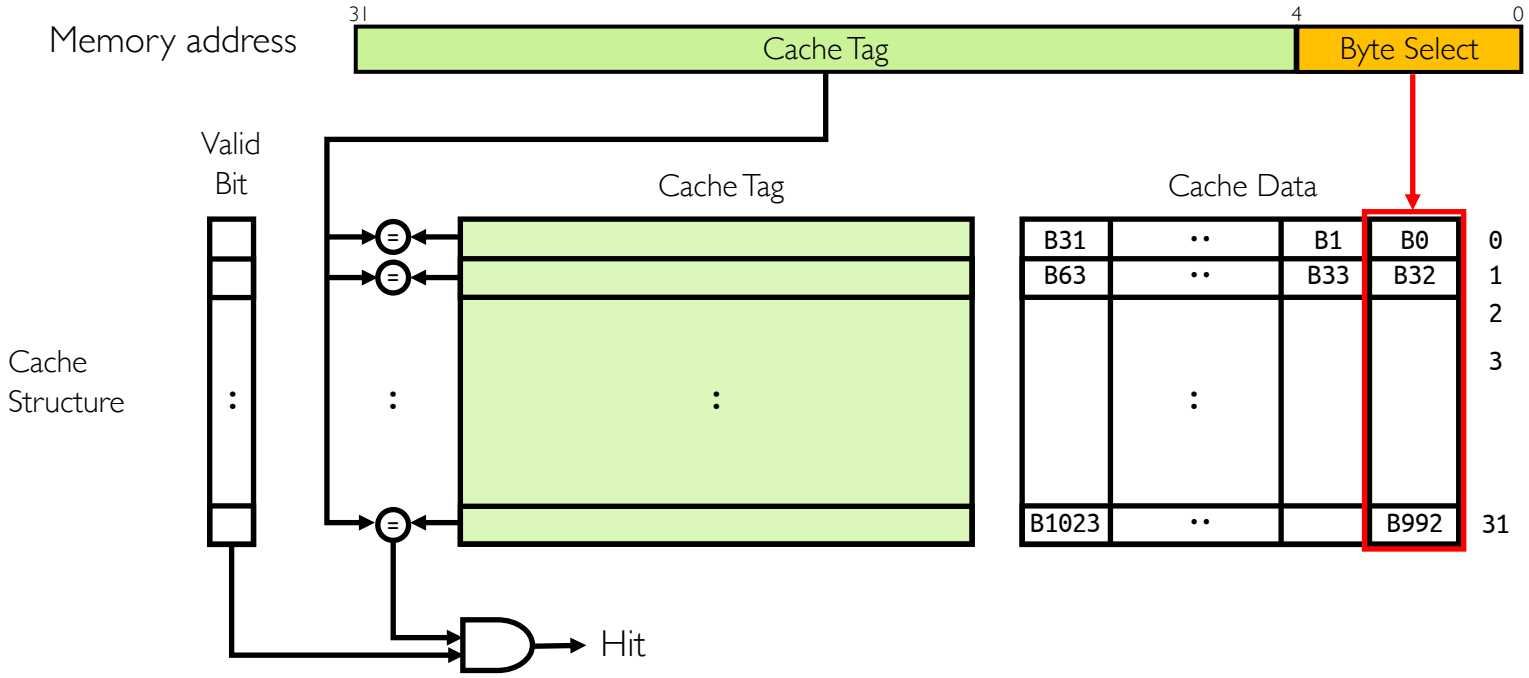
Set-Associative Cache

- 2^K-way set-associative **2^N byte** cache with block size of **2^M bytes**
  - Lowest **M bits** for byte select, (**32 - N + K**) **bits** for cache tag, rest for cache index
  - 2^K direct mapped caches operates in parallel
- Previous example, now with 2-way set-associativity
  - Cache Index selects "set" from cache, there are 16 sets ⇒ **4 bits** for index



Fully Associative Cache

- Every cache block can hold any memory block
  - Address does not include cache index
  - Compare cache tags of all cache blocks in parallel
- Previous example now with fully associative cache

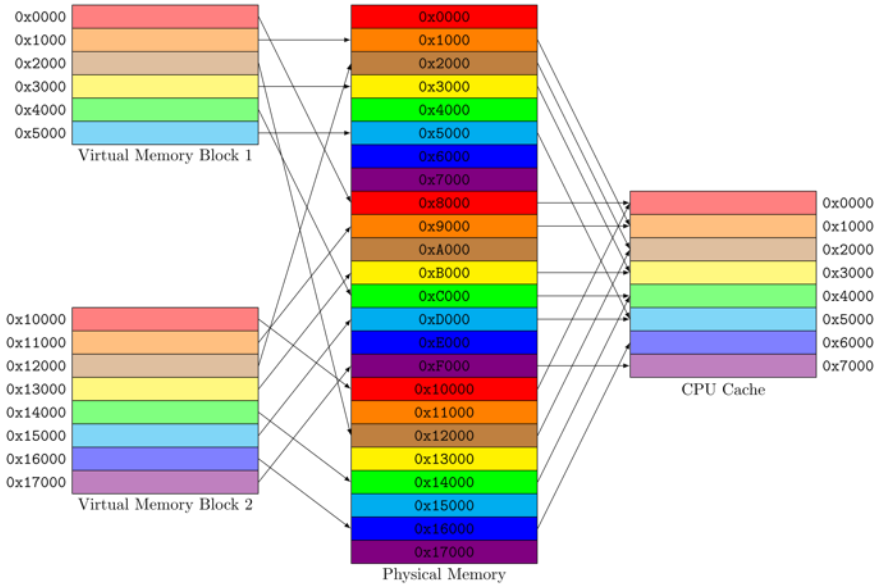


Effective Cache

- Consider 2-MiB, 8-way-set-associative cache and 4KiB physical pages
- Suppose HW uses low-order bits of physical address to index cache
- Suppose process A is allocated physical pages that are separated by 256KiB
- How much cache capacity process A can effectively use?
  - Bytes in memory that are separated by 256KiB are mapped to same cache set
  - A will only be able to use 32KiB (4KiB pages times 8-way set associativity)
  - A can only use less than 2% of cache!

Page Coloring

- Physical pages are given colors
- Pages with same color will be mapped to the same set in cache
  - In above example, there will be 64 different colors (256KiB divided by 4KiB pages)
- Kernel maps sequential virtual pages to physical pages with different colors
  - Sequential pages in virtual memory do not contend for the same cache set



en.wikipedia.org

Possible Sources of Cache Misses

- Compulsory (cold)
  - Cache hasn't seen this block before (start or migration of process)
  - "Cold" fact of life: not whole lot you can do about it
- Capacity
  - Cache cannot contain all blocks accessed by program
  - Solution: increase cache size
- Conflict (collision)
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (no conflict misses in fully associative cache)
- Coherence (invalidation)
  - Other process (e.g., I/O) updates memory

Replaced Policy on Cache Miss?

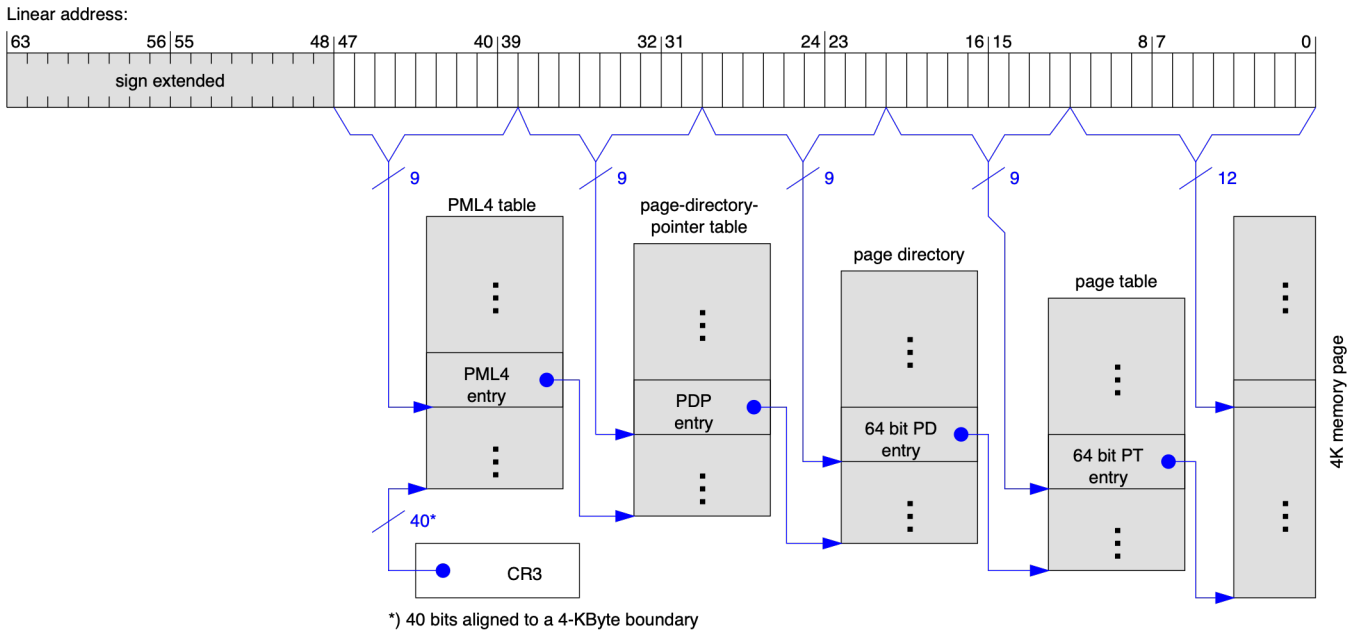
- Easy for direct mapped: only one possibility
- For set associative or fully associative
  - Random
  - Least Recently Used (LRU, more on this later)

What Happens on Write?

- Write-through: write to both cache and lower-level memory
  - + Read misses cannot result in writes
  - – Processor held up on writes unless writes are buffered

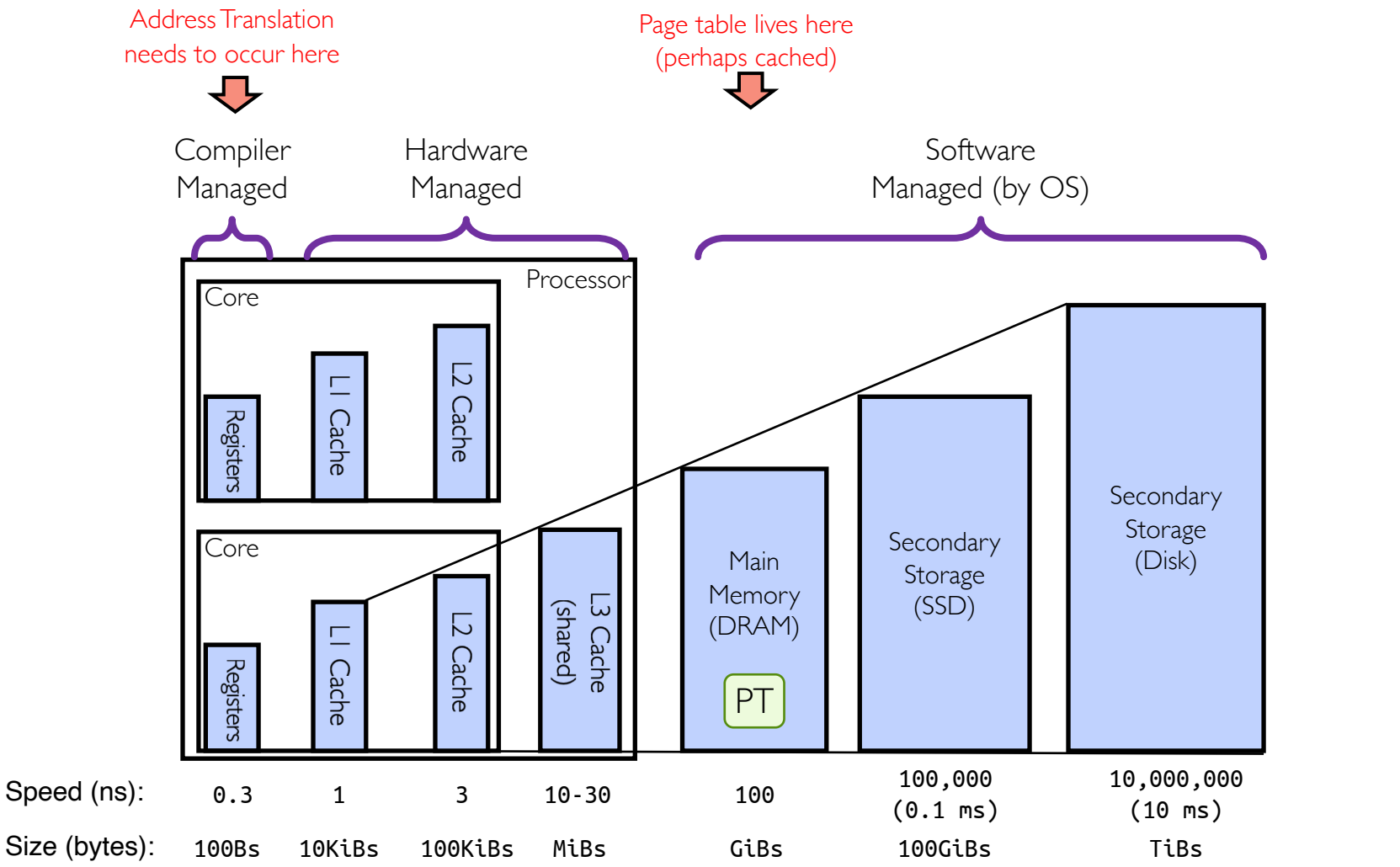
- Write-back: write only to cache
  - Modified cache block is marked dirty
  - On replacement, dirty block is written to lower-level memory
  - + Repeated writes are not sent to DRAM
  - + Processor not held up on writes
  - – More complex
  - – Read miss may require writeback of dirty data

Caching Address Translations



- Cannot afford to translate on every access
  - At least five DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially resides on disk!
  - Even worse, what if we use caches to make memory access faster than DRAM access?

Recall: Memory Hierarchy



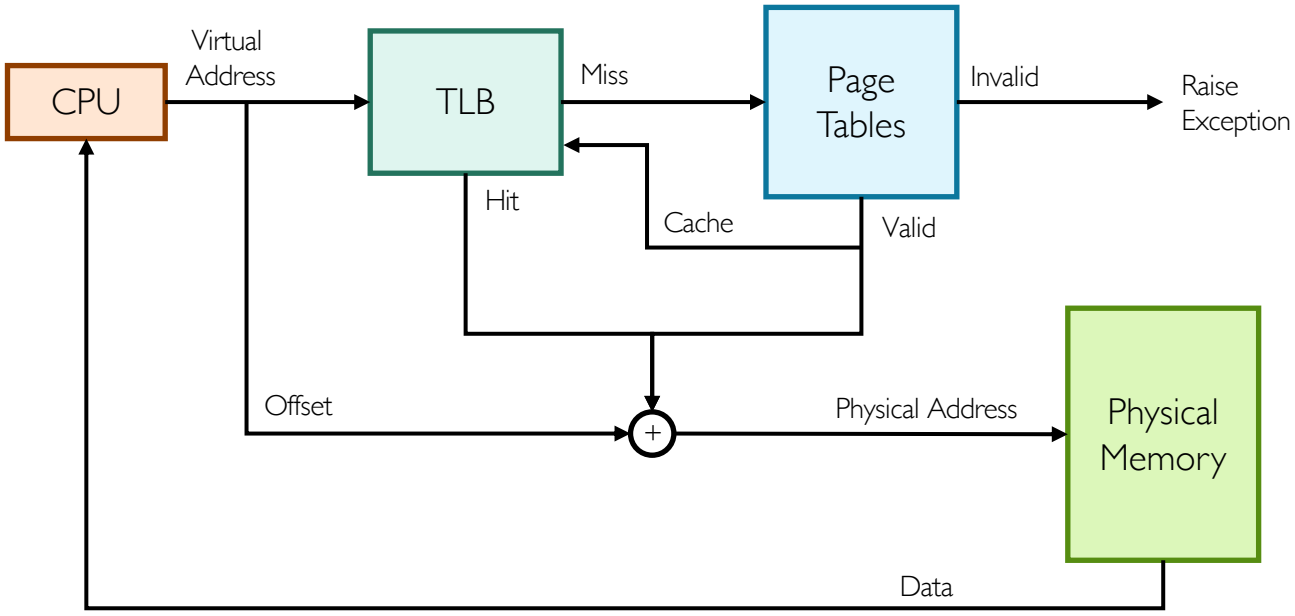
Translation Lookaside Buffer (TLB)

- Main idea: cache recent virtual page number to physical page number translations

- TLB hit provides physical address without reading any of page tables!
  - Caches end-to-end result
  - Even if translation involved multiple levels

- Does page locality exist?
  - Instruction accesses: sequential accesses ⇒ Frequent accesses to the same page ⇒ Yes!
  - Stack accesses: definite locality of reference ⇒ Yes!
  - Data accesses: less page locality, but still some ⇒ Yes, so so!

TLB: Caching Applied to Address Translation



TLB Consistency with PTEs

- If PTE permission is reduced: TLB entry should be invalidated
  - Early computers discarded entire TLB
  - Modern architectures allow removal of individual entries
- If PTE permission is added: nothing needs to be done
  - E.g., changing invalid to read-only or read-only to read-write
  - Any reference would cause exception, OS removes TLB entry
- If PTE is invalidated: TLB entry should be invalidated too
  - E.g., swapping out page from memory to disk (more on this later)

Accessed and Dirty Bits

- TLB entries generally don't have accessed bit
  - If address is cached in TLB, it already should have been accessed
  - Page-table walk sets accessed bit of PTE on TLB miss
- TLB entries do have dirty bit
  - When write misses in TLB, page-table walk sets dirty bit in PTE and TLB
  - Even when write hits in TLB, page-table walk is necessary to set dirty bit if it isn't set
  - If dirty bit is set in TLB, no page-table walk is necessary (saving memory bandwidth)
- Do we really need dirty bit in PTE and TLB?
  - No! OS can emulate it (e.g., BSD Unix)
  - Initially, mark all pages as read-only
  - On write, trap to OS, set software dirty bit, and mark page as read-write
- Do we really need access bit in PTE?
  - No! OS can emulate it
  - Initially, mark all pages as invalid
  - On read, trap to OS (invalid), set software access bit, and mark page read-only
  - On write, trap to OS (invalid or read-only), set software access and dirty bits, mark page read-write

Homonyms

- Definition: single virtual address mapped to different physical addresses
- Problem: TLB entries are invalid after context switching to another process
- Solution 1: invalidate all TLB entries
  - + Simple
  - − Expensive (what if switching frequently between processes)
- Solution 2: tag each TLB entry with *process-context identifier* (PCID)
  - + Less expensive
  - − Needs extra hardware

TLB Shootdown

Processor 1 TLB

	Process ID	VirtualPage	PageFrame	Access
=	0	0x0053	0x0003	R/W
=	1	0x40FF	0x0012	R/W

Processor 2 TLB

	Process ID	VirtualPage	PageFrame	Access
=	0	0x0053	0x0003	R/W
=	0	0x0001	0x0005	Read

Processor 3 TLB

	Process ID	VirtualPage	PageFrame	Access
=	1	0x40FF	0x0012	R/W
=	0	0x0001	0x0005	Read

- If processor1 updates process 0's PTE for page `0x53`, then it should
  - Remove old entry from its TLB
  - Send inter-processor interrupt to other processors telling them to remove their old entries
- Shootdown is complete once all processors verify that their old entry is removed
- TLB shootdown overhead increases linearly with number of processors

What Happens on TLB Miss?

- Hardware-traversed page tables
  - On TLB miss, hardware walks through current page tables to fill TLB (could be multiple levels)
    - Valid PTE: Hardware fills TLB and processor never notices
    - Invalid PTE: CPU raises page fault ⇒ Kernel decides what to do next

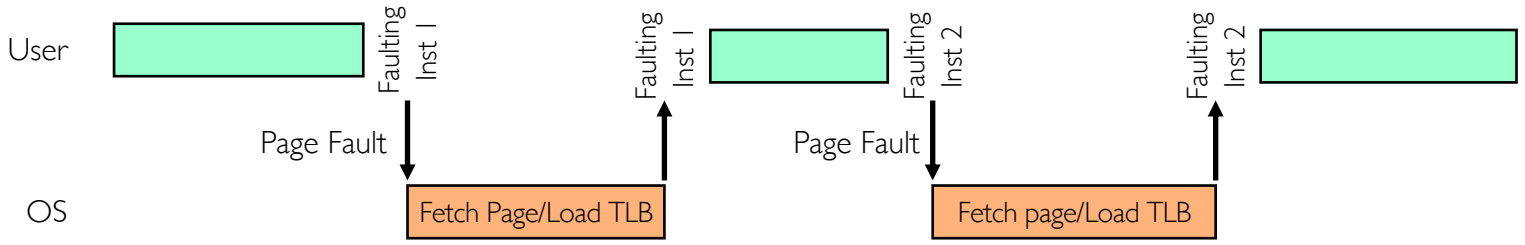
- Software-traversed page tables
  - On TLB miss, CPU raises TLB fault
  - Kernel walks through page table(s) to find PTE
    - Valid PTE: Fills TLB and returns from fault
    - Invalid, internally calls page fault handler

TLB Fault and Page Fault Exceptions

- Unlike interrupts, exceptions are synchronous, caused by particular instruction
  - E.g., TLB fault or page fault, divide by zero
- In general, faulting instruction needs to be *restarted* after exception is handled
- Side effects of faulting instruction need to be undone
  - Example: `push 10`
    - What if page fault occurs when write to stack pointer?
    - Was `sp` incremented before or after page fault?
- Partially executed instructions should also be undone in out-of-order execution
  - Example 1:
    ```
    mul r1, r2, r3
    bne r1, r4, loop
    ld r4,(r5)
    ```
    - What if it take many cycles to see if `r1 = r4`, but load has already caused page fault?
  - Example 2:
    ```
    div r1, r2, r3
    ld r3, (r4)
    ```
    - What if it takes many cycles to discover divide-by-zero, but load has already caused page fault?

Precise Exceptions

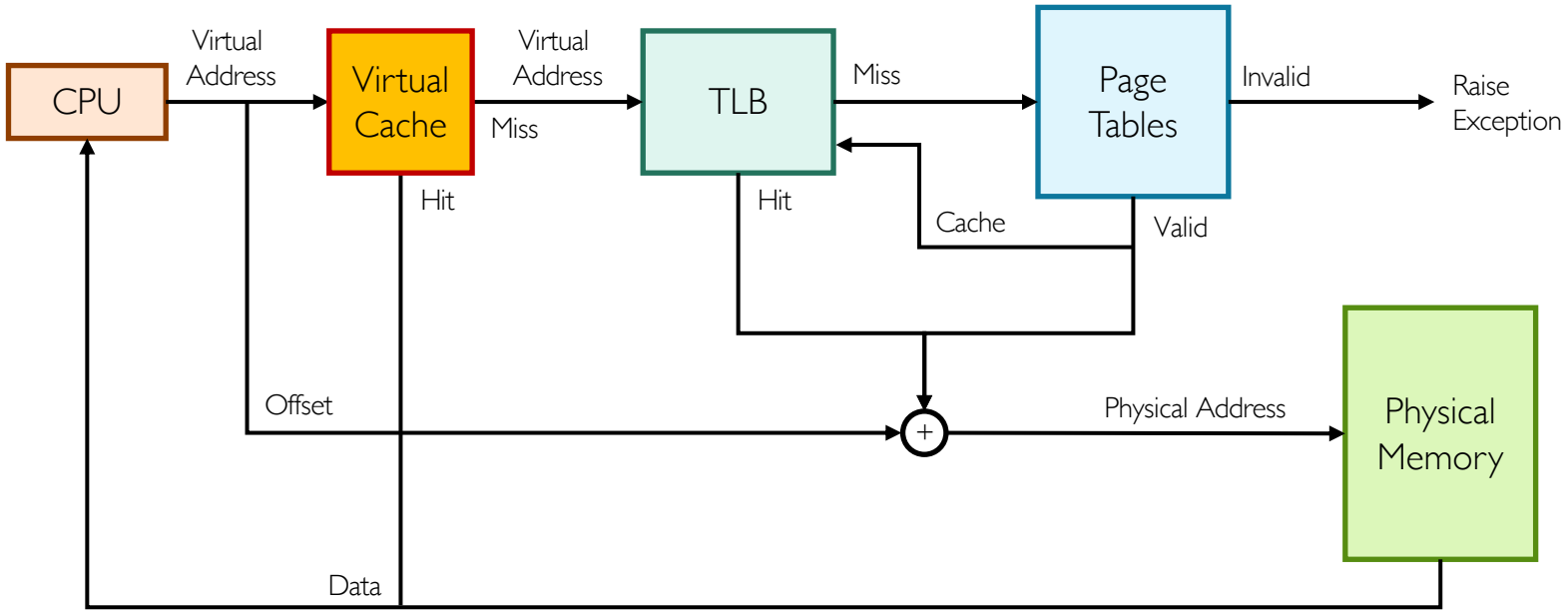
- Instructions retire when their results become visible in architectural state
  - E.g., processor registers, memory, etc.
  - Architectural state ≠ micro-architectural state (e.g., caches)
- To implement precise exceptions, instructions should retire in program's sequential order
  - Execution could still be out-of-order
- Exception should only be raised when faulting instruction tries to retire
  - All instructions before faulting instruction should have already retired
- When exception is raised, architectural state should be preserved
  - Faulting instruction and all following instructions act as if they have not even started



Improve Efficiency Even More!

- TLB improves performance by caching recent translations
- How to improve performance even more?
  - Multi-level TLBs
- What is the cost of first-level TLB miss?
  - Second-level TLB lookup
- What is the cost of second-level TLB miss?
  - x86: 2-4 level page table walk

Virtually-addressed Cache

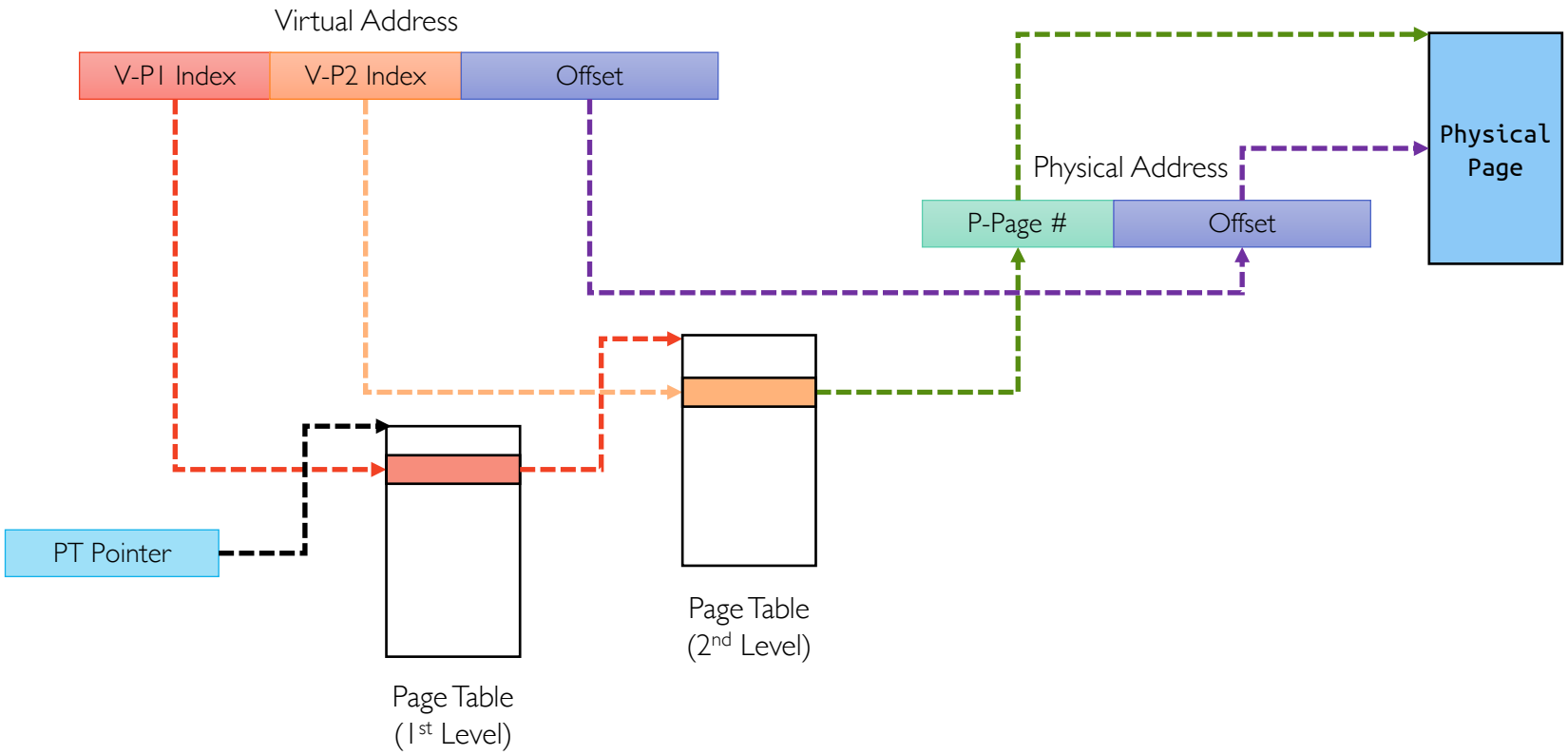


- Too slow to access TLB before looking up address in memory
- Instead, add virtually-addressed cache (virtual cache)
- In parallel, access TLB to generate physical address in case of cache miss

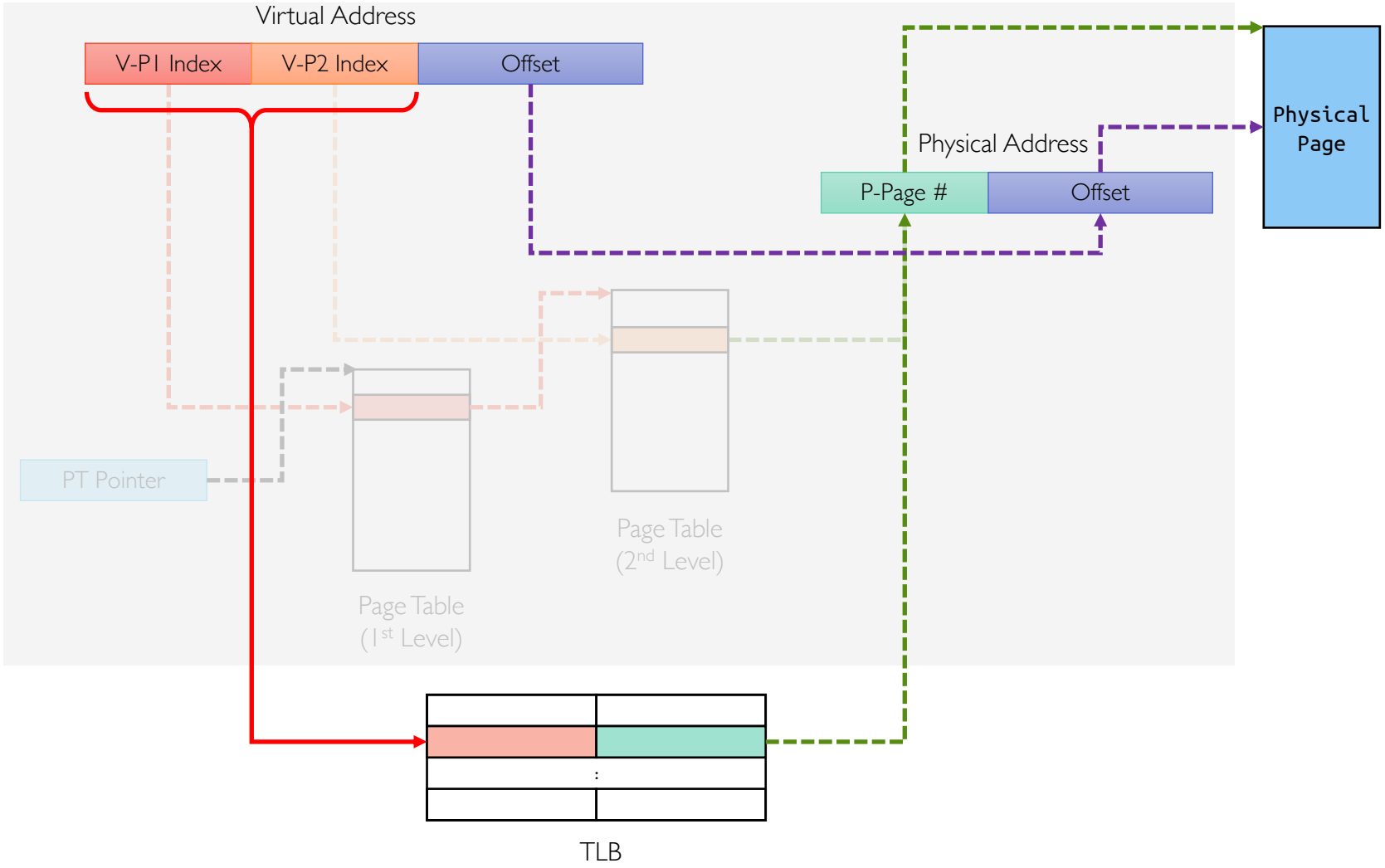
Synonym

- Definition: different virtual addresses mapped to same physical address
  - Could be in same virtual address space or different virtual address spaces
  - E.g., `mmap()` same file multiple times in same process or once in multiple processes
- Aliasing problem: synonyms could be mapped to different locations in virtual cache
- Typical solution: virtually-indexed-physically-tagged virtual cache
  - Map synonyms to the same cache set (kernel ensures assigned VAs agree in index bits)
  - Tag each virtual cache block by physical address
  - Lookup virtual cache and TLB in parallel
  - Update/invalidate other copies if physical address from TLB matches multiple entries
- Synonym problem could affect any HW structure that deals with memory accesses
  - E.g., load with synonym VA misses in store buffer if entries are tagged by VA and PID

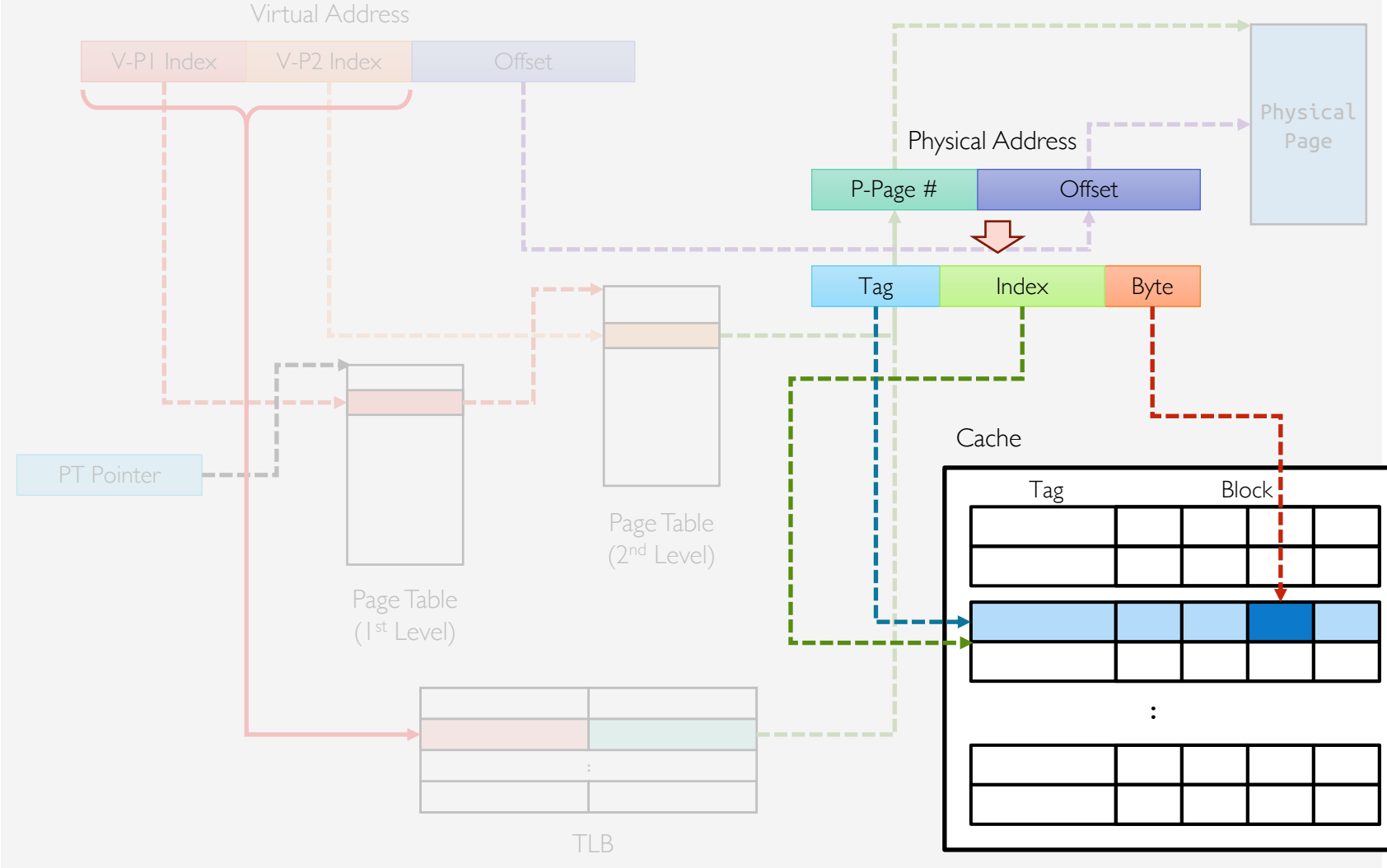
Putting it Together: Address Translation



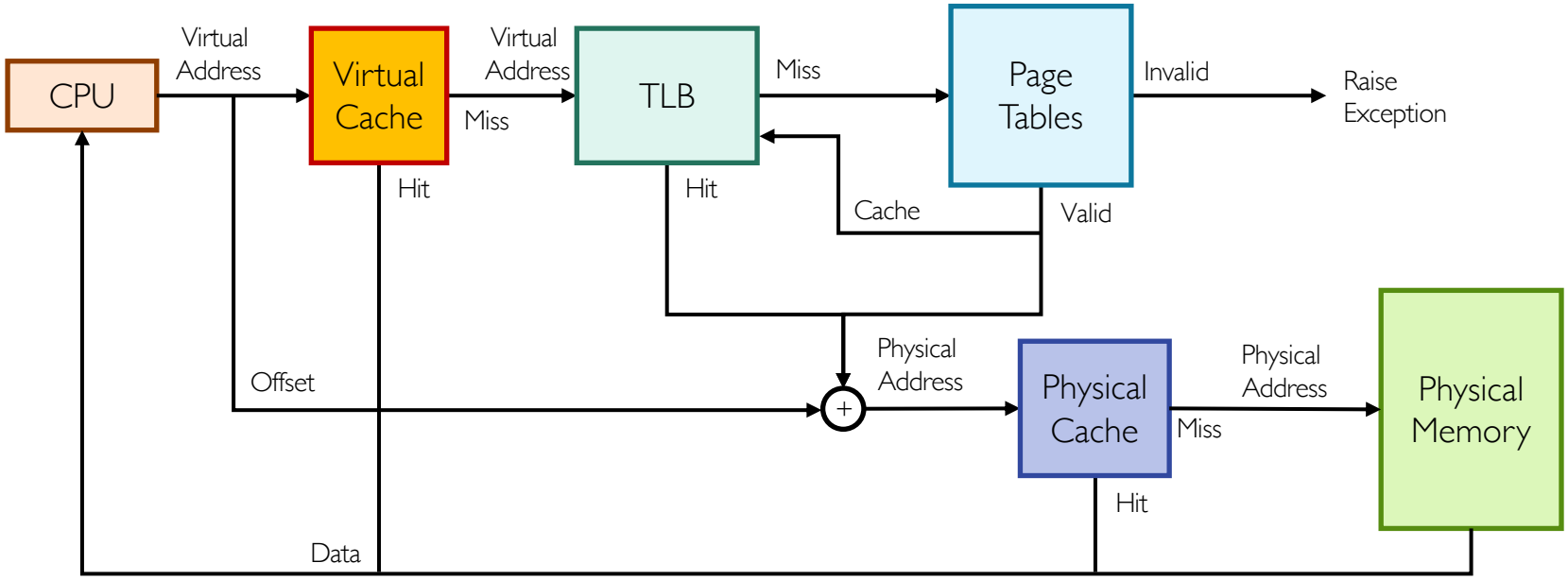
Putting it Together: TLB



Putting it Together: Physical Cache

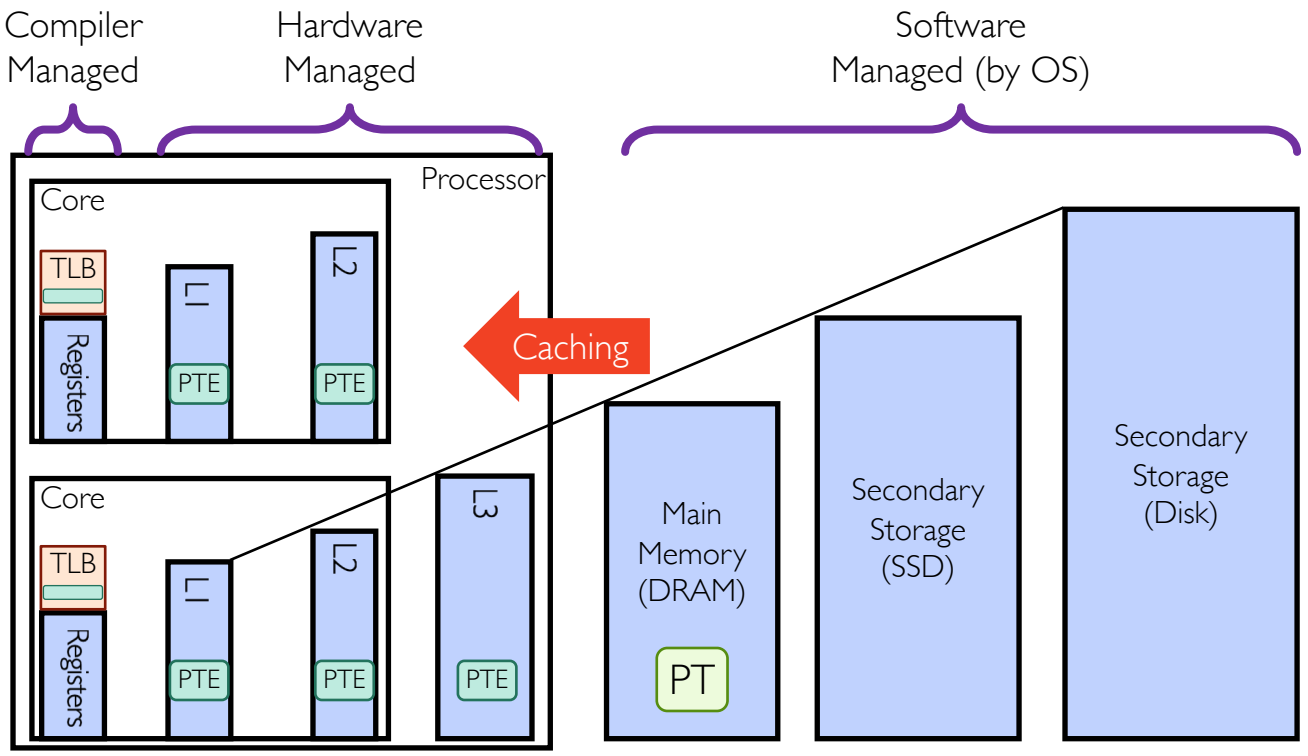


Putting it Together: Page Table, TLB, and Caches

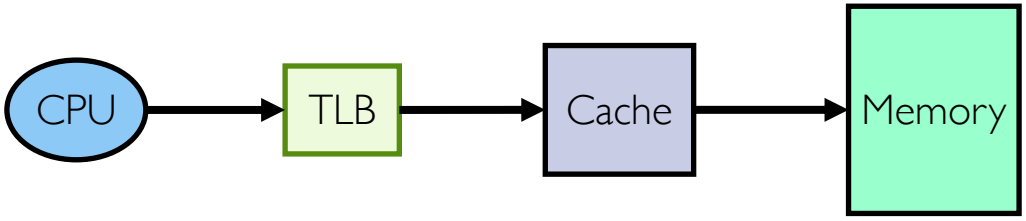


Recall: Memory Hierarchy

- Can TLB misses get resolved without ever going to main memory?



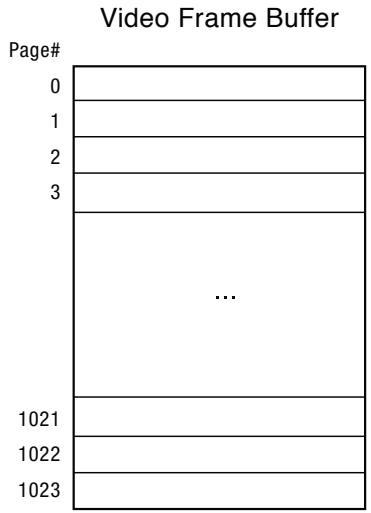
TLB Set Associativity



Average memory access time = hit-time_{TLB} + (miss-ratio_{TLB} x miss-time_{TLB})

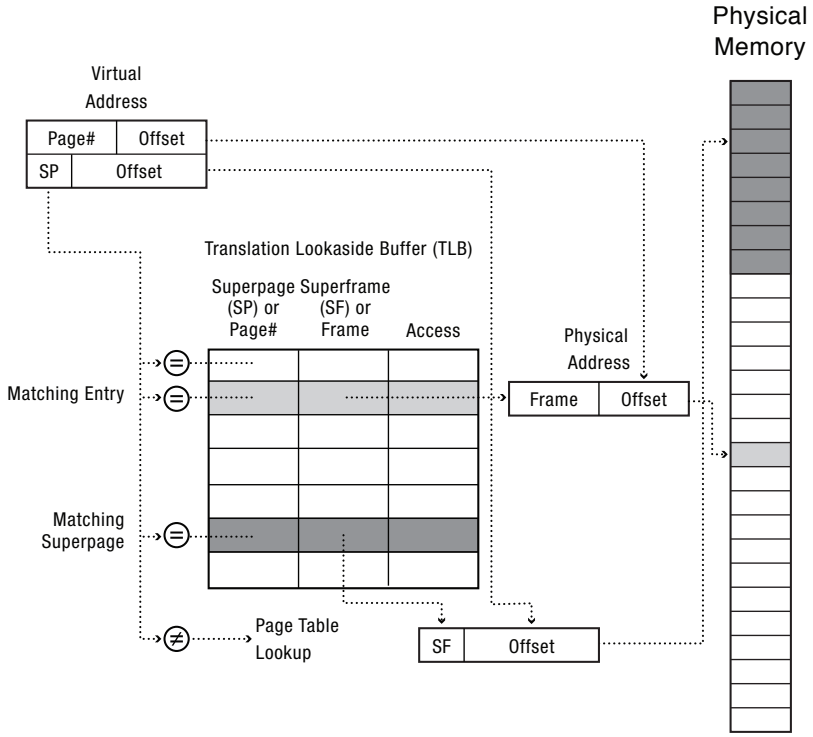
- TLB hit time is added to all memory accesses
- Should TLB be direct-mapped or have low associativity?
  - No! TLB needs to have very few conflicts!
  - Miss time is extremely high!
- TLBs are typically fully-associative
  - Significantly reduce conflict misses in return for slightly higher hit time

Do TLBs Always Improve Performance?



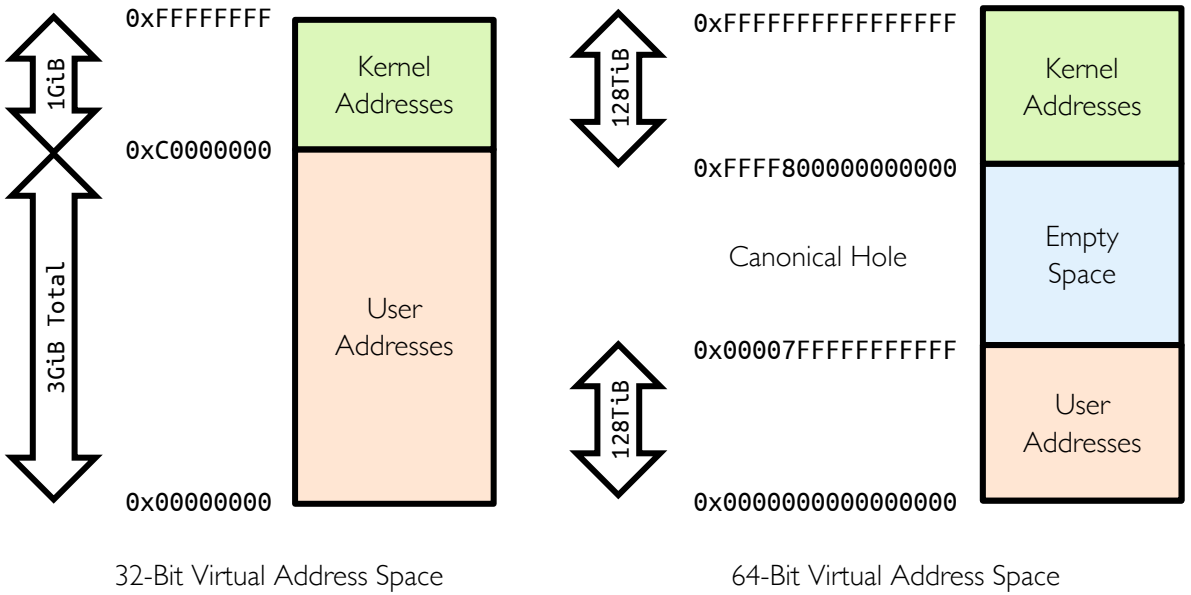
- Example: for HD displays, video frame buffer could be large
  - E.g., 4k display: 32 bits x 4K x 3K = 48MiB (spans 12K of 4KiB pages)
- Even large on-chip TLB with 256 entries cannot cover entire display
- Each horizontal line of pixels could be on on page
- Drawing vertical line could require loading a new TLB entry

Superpages: Improving TLB Hit Rate



- Reduce number of TLB entries for large, contiguous regions of memory
  - Represent 2 adjacent 4KB pages by single 8KB superpage
- By setting a flag, TLB entry can be a page or a supperpage
  - E.g., in x86: 4KB (12 bits offset), 2MB (21 bits offset), or 1GB (30 bits offset)

Linux Virtual Memory Map Prior to KPTI Patch



- Address space of user process includes kernel memory
- Kernel memory is protected from user process by *owner bit*
- + On system calls or interrupts, kernel page tables are always present
  - Mitigating context-switch overheads (e.g., TLB flush, page-table swapping, etc.)
- – It exposes serious security vulnerabilities that have been exploited by various attacks
  - E.g. Meltdown and Spectre attacks

Meltdown Attack: Background

- Branches can significantly slow down out-of-order execution
  - E.g., processor has to wait for many cycles to determine direction of conditional jump
- To speed up out-of-order execution, modern processors implement branch predictors (BP)
  - BP predicts whether conditional branch will be taken before its execution
  - Predictions are usually based on previous executions of the branch
  - Processor executes next instructions speculatively
  - On branch misprediction, processor rolls back speculatively-executed instructions
    - Their results do not change architectural state

Meltdown Attack

- Meltdown was announced in 2018
- It affects Intel x86, IBM Power, and some ARM processors

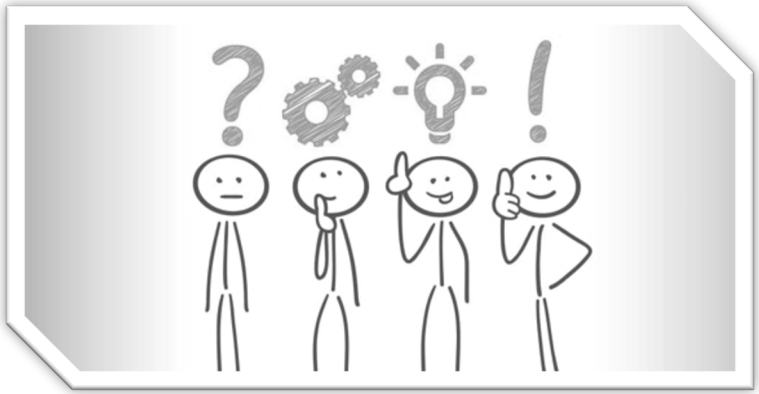
```
char array[256 * 4096];   // Skip 4096 to avoid HW cache prefetch
flush_cache(array);       // Make sure array is not cached
try {                     // catch and ignore SIGSEGV (illegal access)
     char result = *(char *)kernel_address;   // Segmentation fault
     char dummy = array[result * 4096];       // Leak info!
} catch(){;}              // Could use signal() to catch SIGSEGV
for (int i = 0; i < 256; i++)
     if (is_in_cache(array[i * 4096]))
          printf("%d\n"); // found byte at kernel_address
```

- Kernel page-table isolation (KPTI) patch was released to mitigate Meltdown
  - Without PCID tag in TLB, KPTI needs to flush TLB twice on syscall and interrupts (800% overhead!)
  - Need at least kernel v4.14 which utilizes PCID tag in new HW to avoid flushing

Summary

- Principle of locality
  - Programs access small portion of address space at any instant of time
- Cache organizations
  - Direct-mapped, set-associative, fully-associative
- Three (+1) major categories of cache misses
  - Compulsory, conflict, capacity, coherence
- TLB: caching applied to address translations
  - Cache relatively small number of PTEs
  - Fully associative (since conflict misses expensive)
  - On TLB miss, page table is traversed and if PTE is invalid, cause page fault
  - On change in page table, TLB entries must be invalidated

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Ousterhout, Sorin, Asanovic, Culler, Stoica, Silberschatz, Joseph, and Canny