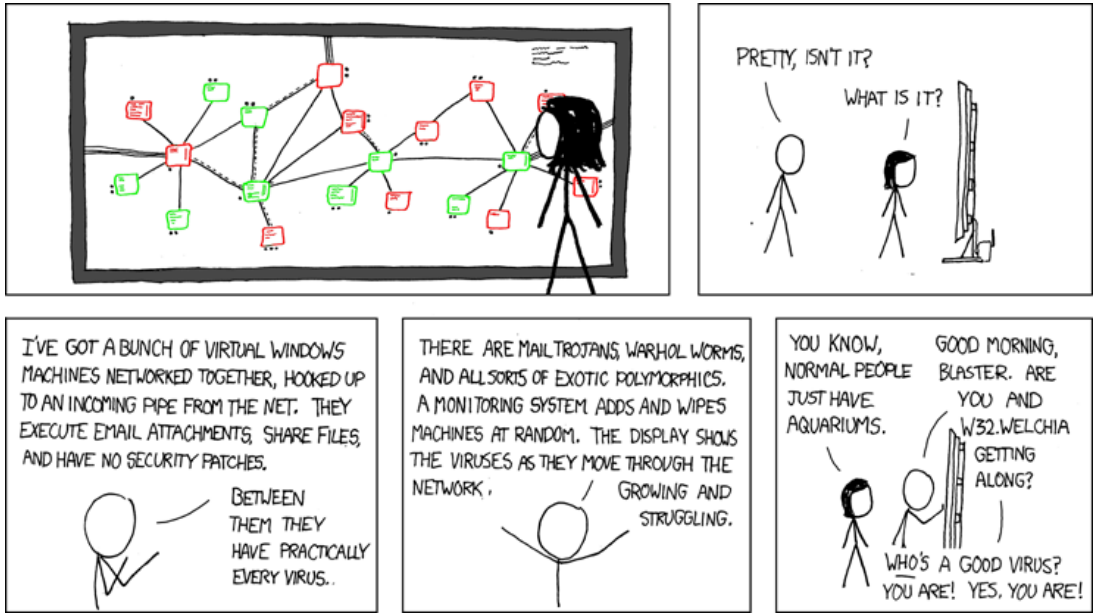
CS/ECE 752: Advanced Computer Architecture I

Prof. Matthew D. Sinclair

Virtual Memory

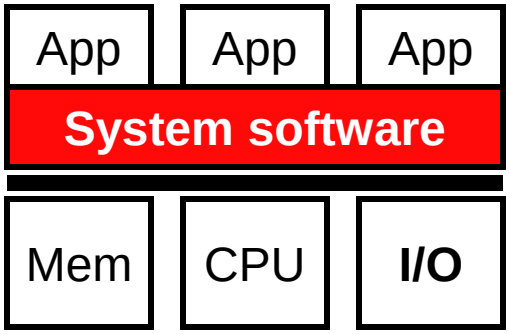


Slide History/Attribution Diagram:

Announcements 10/23/24

- Midterm grading update
  - 2A, 3A/B done too
  - 2B, 5B remain
- Proposals due Friday
- HW5 due Saturday
- HelioCampus mid-semester evals released tomorrow
  - Will look at and respond soon
  - I think you can still fill out today if you haven't already

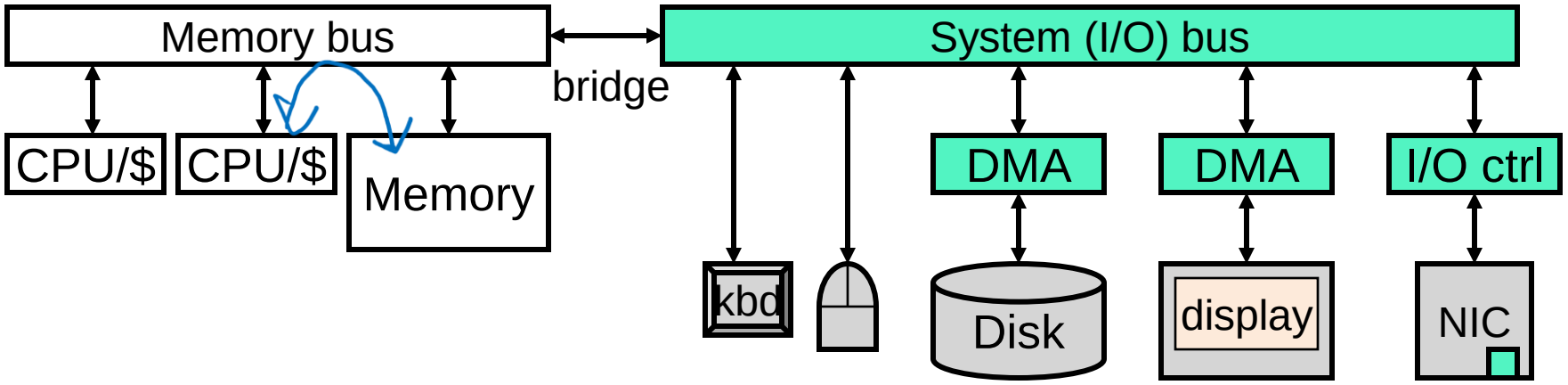
Today: Virtual Memory



- The operating system (OS)
  - A super-application
  - Hardware support for an OS
- Virtual memory
  - Page tables and address translation
  - TLBs and memory hierarchy issues

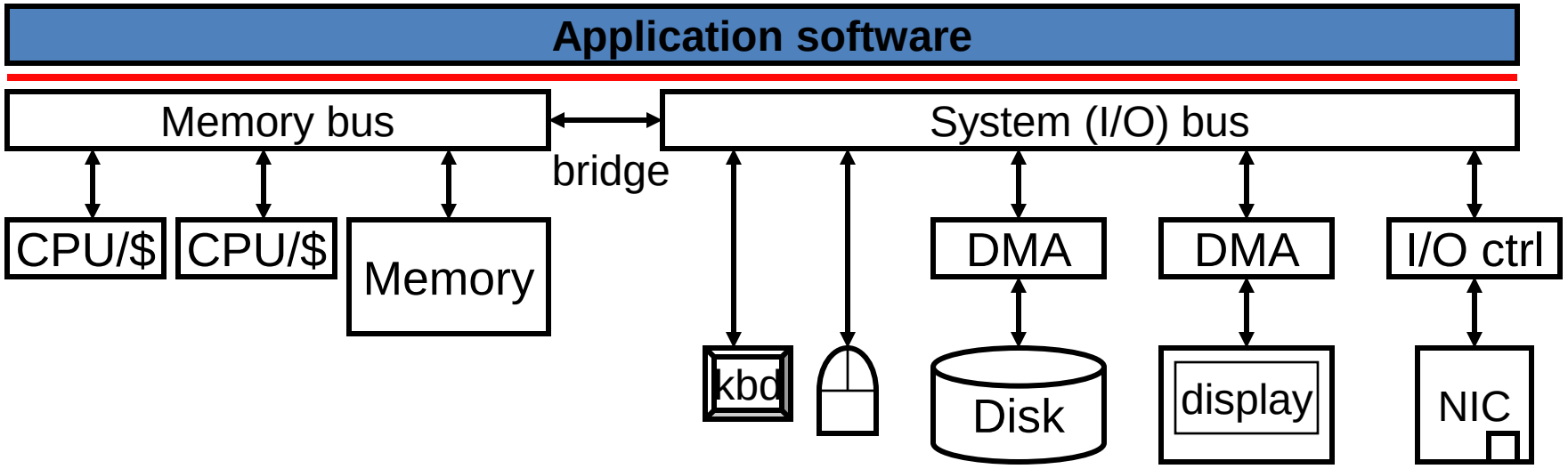
A Computer System: Hardware

- CPUs and memories
  - Connected by memory bus
- **I/O peripherals**: storage, input, display, network, …
  - With separate or built-in DMA
  - Connected by **system bus** (which is connected to memory bus)



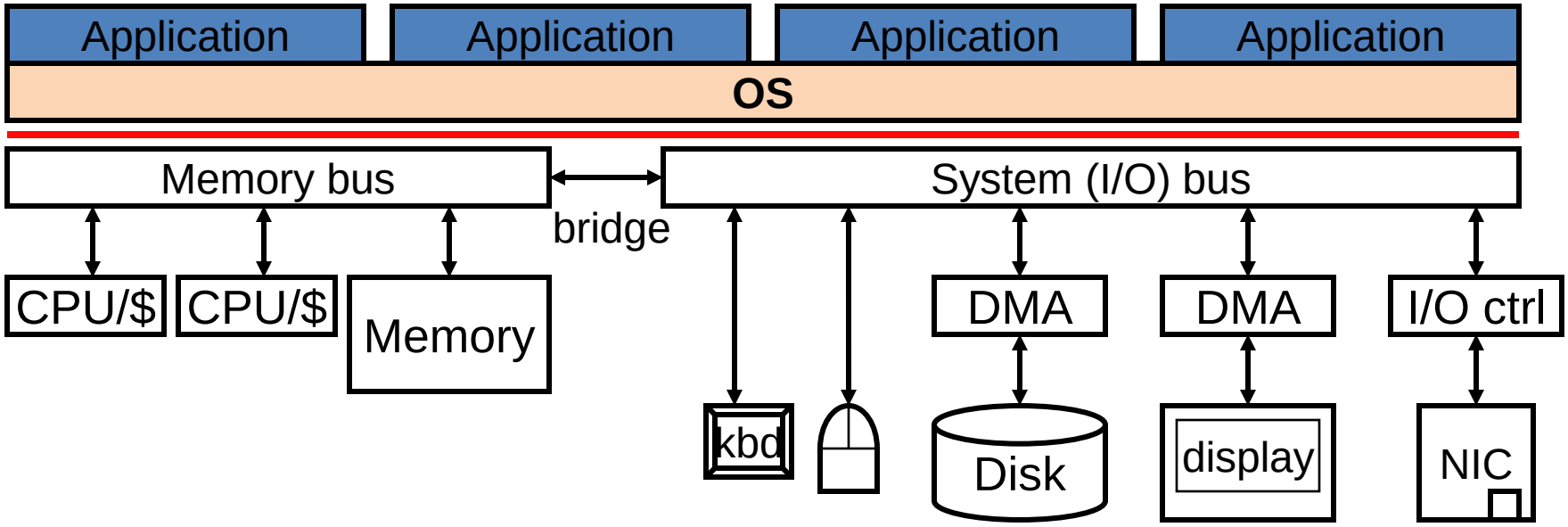
A Computer System: + App Software

- **Application software**: computer must do something
  - Pain to just use hardware directly – how to share?



A Computer System: + OS

- **Operating System (OS):** virtualizes hardware for apps
  - **Abstraction**: provides **services** (e.g., threads, files, etc.)
    + Simplifies app programming model, raw hardware is nasty
  - **Isolation**: gives each app illusion of private CPU, memory, I/O
    + Simplifies app programming model
    + Increases hardware resource utilization



Operating System (OS) and User Apps

- Sane system development requires a split
  - Hardware itself facilitates/enforces this split

- **Operating System (OS)**: a super-privileged process
  - Manages hardware resource allocation/revocation for all processes
  - Has direct access to resource allocation features
  - Aware of many low level hardware details
  - Aware of other processes
  - Talks directly to input/output devices (device driver software)

- **User-level apps**: ignorance is bliss
  - Unaware of most nasty hardware details
  - Unaware of other apps (and OS) — MPI
  - Explicitly denied access to resource allocation features

System Calls (sys calls)

- Controlled transfers to/from OS

- **System Call**: a user-level app "function call" to OS
  - Leave description of what you want done in registers
  - SYSCALL instruction (also called TRAP or INT)
    - Entry point into the OS for privileged operations
  - Basic operation
    - Processor jumps to requested handler
      - Sets privileged mode
    - OS code performs operation
    - OS does a "return from system call"
      - Unsets privileged mode

Interrupts – OS wants your attention

- Exceptions: synchronous, generated by running app
  - E.g., illegal insn, divide by zero, etc.
- **Interrupts**: asynchronous events generated externally
  - E.g., timer, I/O request/reply, etc.
- E.g. Timer: programmable on-chip interrupt
  - Initialize with some number of micro-seconds
  - Timer counts down and interrupts when reaches zero
- **"Interrupt" handling**: same mechanism for both
  - "Interrupts" are on-chip signals/bits
    - Either internal (e.g., timer, exceptions) or from I/O devices
  - Processor continuously monitors interrupt status, when one is high…
  - Hardware jumps to some preset address in OS code (interrupt vector)
  - Like an asynchronous, non-programmatic SYSCALL

Virtualizing Processors

- How do multiple apps (and OS) share the processors?
  - **Goal: applications think there are an infinite # of processors**

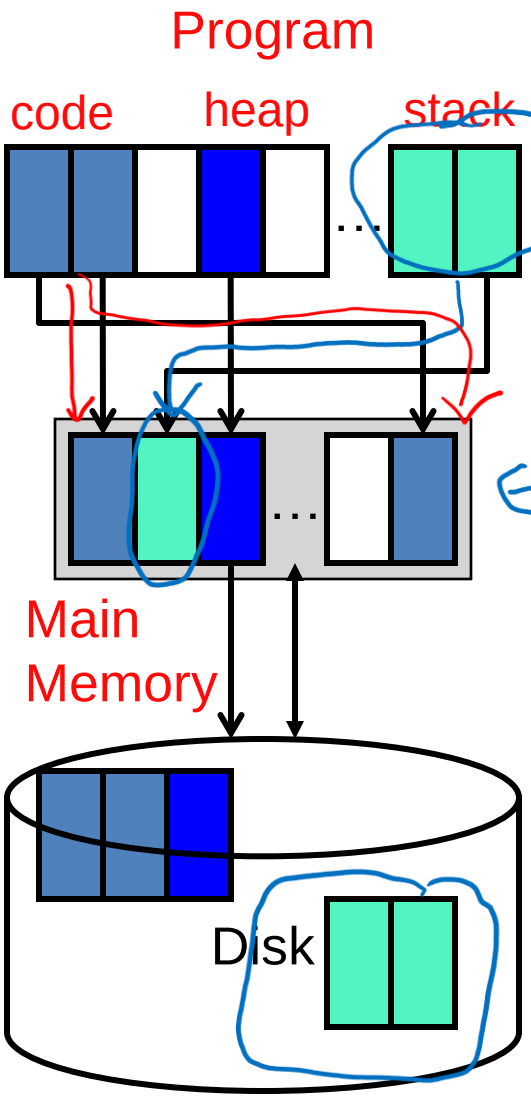
switch between apps

- Solution: time-share the resource
  - Trigger a **context switch** at a regular interval (~5ms)
    - **Pre-emptive**: app doesn't yield CPU, OS forcibly takes it
      + Stops greedy apps from starving others
  - **Architected state**: PC, registers
    - Save and restore them on context switches
    - Memory state?
  - **Non-architected state**: caches, predictor tables, etc.
    - Ignore or flush
- OS responsible to handle context switching
  - Hardware support is just a timer interrupt

Virtualizing Main Memory

- How do multiple apps (and the OS) share main memory?
  - **Goal: each application thinks it has infinite memory**
  - Biggest memory we have: Disk (so slooooow)
  - Faster memory DRAM (so expensive!)
- One app may want more memory than is in main memory
  - App's insn/data footprint > DRAM — database
  - **Requires main memory to act like a cache**
    - With disk as next level in memory hierarchy (slow)
    - Write-back, write-allocate, large blocks or "pages"
- Solution:
  - Part #1: treat memory as a "cache"
    - Store the overflowed blocks in "swap" space on disk
  - Part #2: add a level of indirection (address translation)

Virtual Memory



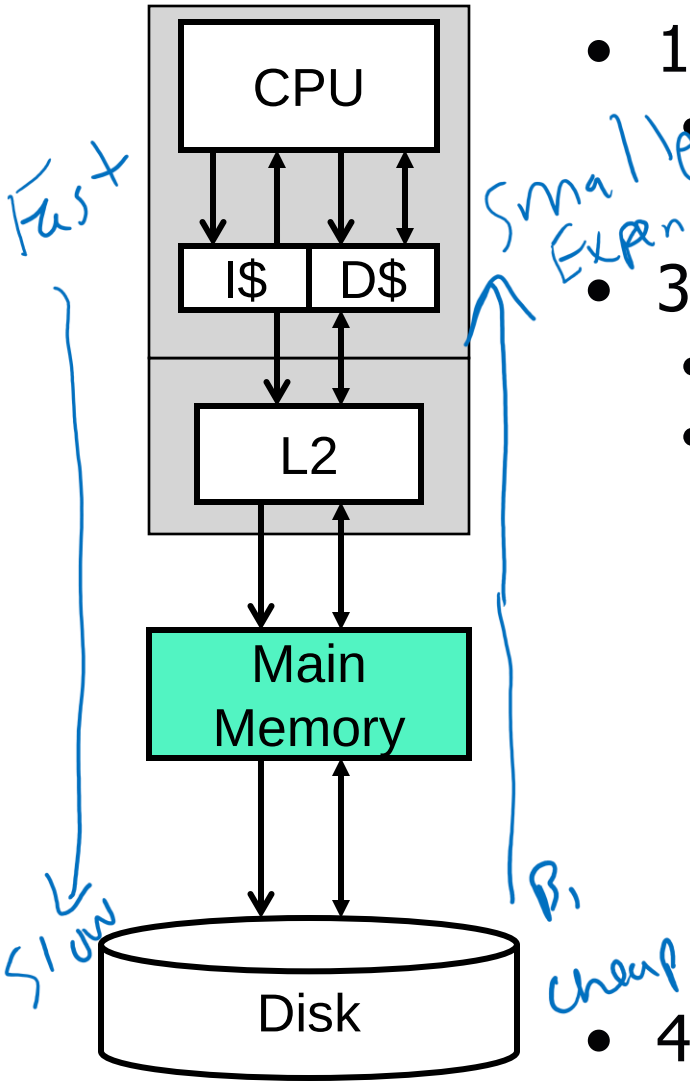
- Programs use **virtual addresses (VA)**
  - $0…2^N–1$
  - VA size also referred to as machine size
  - E.g., Pentium4 is 32-bit, SPARC is 64-bit
- Memory uses **physical addresses (PA)**
  - $0…2^M–1$ (typically M<N, especially if N=64)
  - 2^M is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - **Small granularity too expensive!**
  - By "system"
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap)

Historically….

- Original motivation: **capacity**
  - Atlas (1962): Fully-associative cache of pages, called *one-level store*
  - 16K words of core memory; 96K words of drum storage (≈ disk)

- Successful motivation: **compatibility**
  - IBM System 370: a family of mainframe computers with one software suite
  + Same program could run on machines with different memory sizes
    - Caching mechanism made it appear as if memory was 2^N bytes
    - Regardless of how much there actually was
  – Prior, programmers explicitly accounted for memory size

How to manage our memory as a cache?



- 1st/2nd levels: SRAM caches (I$, D$, L2)
  - Set Associative/Hardware Managed
- 3rd level: **main memory (DRAM)**
  - Made of DRAM
  - **Managed in software (how?)**
    - Block size? 4KB+
    - Associativity? Fully associative
    - Replacement policy? In software
    - Write-back vs. write-through? Write-back
    - Write-allocate vs. write-non-allocate? Write allocate
- 4th level: disk (swap space)
  - SSD/HDD/etc. -- SLOW!

Low $\%_{miss}$ At All Costs

- For a memory component: t_{hit} vs. $\%_{miss}$ tradeoff
- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access $\rightarrow$ minimal t_{hit} important
  - t_{miss} is not bad $\rightarrow$ minimal $\%_{miss}$ less important
  - Low capacity/associativity, write-back or write-thru
- Moving down (L2) emphasis turns to $\%_{miss}$
  - Infrequent access $\rightarrow$ minimal t_{hit} less important
  - t_{miss} is bad $\rightarrow$ minimal $\%_{miss}$ important
  - High capacity/associativity, write-back
- For memory, emphasis entirely on $\%_{miss}$
  - t_{miss} is disk access time (measured in ms, not ns)

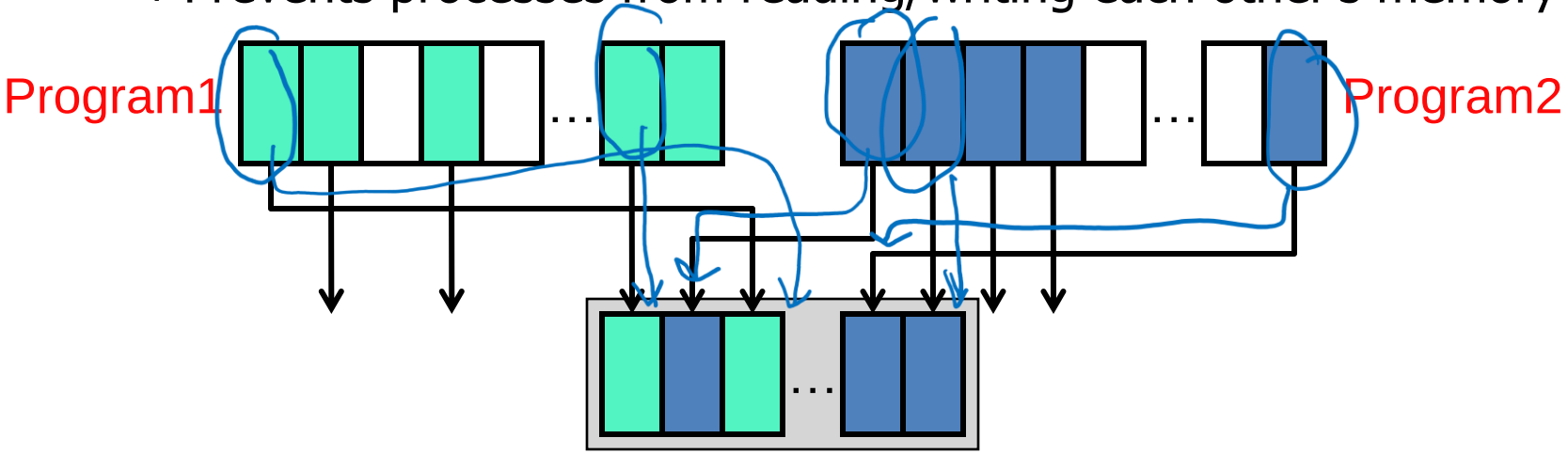
Memory Organization Parameters

Parameter	I$/D$	L2	Main Memory
t_{hit}	<1ns	10-20ns	**100ns**
t_{miss}	**10-20ns**	**100ns**	**10ms (10M ns)**
Capacity	8–64KB	256KB–20MB	**Up to 1TB**
Block size	16–**64B**	**64**–256B	**4+KB**
Associativity	1–8	4–16	**Full**
Replacement Policy	LRU/NMRU	LRU/NMRU/RRIP	**working set**
Write-through?	Either	No	**No**
Write-allocate?	Either	Yes	**Yes**
Victim buffer?	Yes	Maybe	**No**
Prefetching?	Either	Yes	**Software**

Uses of Virtual Memory

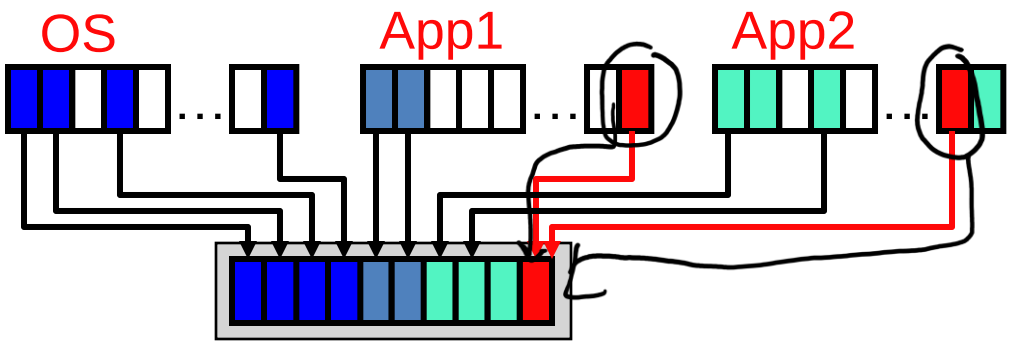
- Virtual memory is quite a useful feature
  - Automatic, transparent memory management just one use
  - "Functionality problems are solved by adding levels of indirection"
- Example: **program isolation and multiprogramming**
  - Each process thinks it has 2^N bytes of address space
  - Each thinks its stack starts at address 0xFFFFFFFF
  - System maps VPs from different processes to different PPs

    \+ Prevents processes from reading/writing each other's memory



More Uses of Virtual Memory

- **Protection**
  - Piggy-back mechanism to implement page-level protection
  - Map virtual page to physical page
    - … and to Read/Write/Execute protection bits in page table
  - Attempt to illegal access, to execute data, to write read-only data?
    - Exception → OS terminates program
- **Inter-process communication**
  - Map virtual pages in different processes to same physical page
  - Share files via the UNIX mmap() call



- **Don't duplicate shared libraries!**

In-Class Assignment

- With a partner, answer the following questions:
  - In OS (736) they taught you software (OS) should handle page faults. In Architecture (552, 752) we teach you hardware should handle page faults. Who is right?

Hardware: faster, less overhead, more area

OS: less area, slower, more app aware, flexibility

Balance between OS + HW

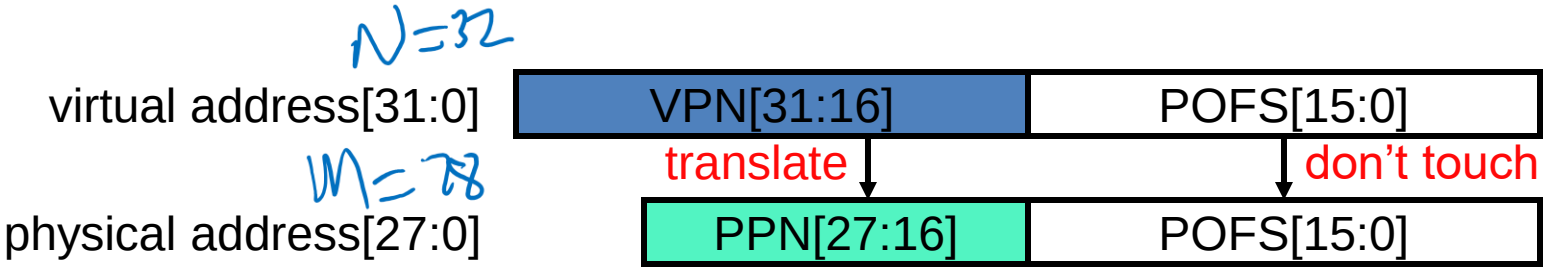
  - If nested paging is so expensive, why is it worth it?

Every process could have own PT → frag, wasted space

Lots of indirection

- In 3 minutes we'll discuss as a class

Address Translation



- VA→PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** and page offset (POFS)
  - Translate VPN into **physical page number (PPN)**
  - POFS is not translated
  - VA→PA = [VPN, POFS] → [PPN, POFS]

- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN
  - Maximum 256MB memory → 28-bit PA → 12-bit PPN

$\log_2(64k) \Rightarrow 16b$

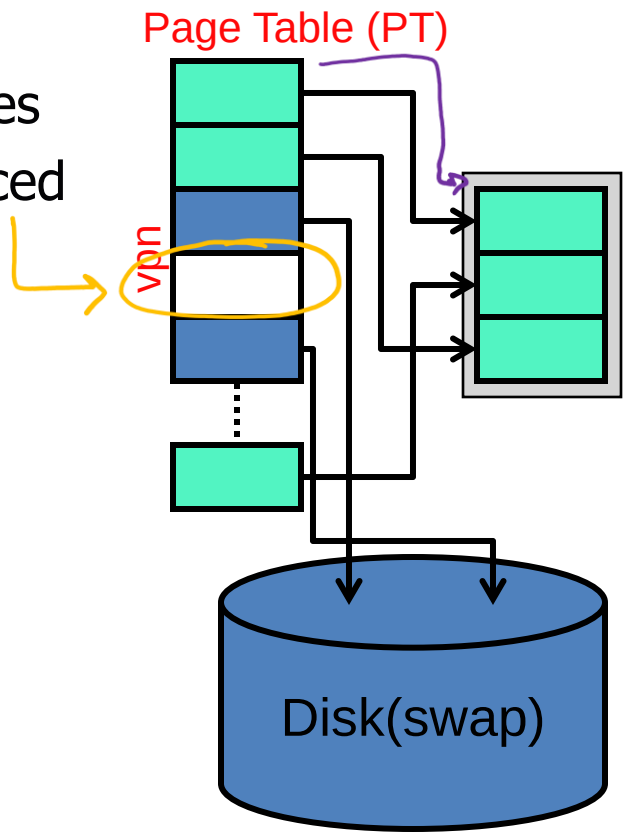
$32-16 = 16b$

$28-16 = 12b$

Mechanics of Address Translation

- How are addresses translated?
  - In software (now) but with hardware acceleration (a little later)
- Each process allocated a **page table (PT)**
  - **Managed by the operating system**
  - Maps VPs to PPs or to disk (swap) addresses
    - VP entries empty if page never referenced
  - Translation is table lookup

```
struct {
   union { int ppn, disk_block; }
   bool is_valid, is_dirty;
} PTE;
struct PTE pt[NUM_VIRTUAL_PAGES];

int translate(int vpn) {
  if (pt[vpn].is_valid)
     return pt[vpn].ppn;
}
```



Page Table Size

- How big is a page table on the following machine?
  - 4B page table entries (PTEs)
  - 32-bit machine
  - 4KB pages

  - 32-bit machine → 32-bit VA → 4GB virtual memory
  - 4GB virtual memory / 4KB page size → 1M VPs
  - 1M VPs * 4B PTE → 4MB (per process)

```
tjn@trapezo:~$ ps aux | wc -l
377
```

$\frac{2^{32}}{2^{16}} = 2^{16}$ VPs

$2^{16} \times 2^2 = 2^{18}$ B

- How big would the page table be with 64KB pages?
  - 256KB
- How big would it be for a 64-bit machine?
  - 18,014,398,509,481,984KB (oops)
- Page tables can get big
  - There are ways of making them smaller
  - PA = f(VA) ➔ many different data structures possible
    - Want a compact representation that has fast lookups!

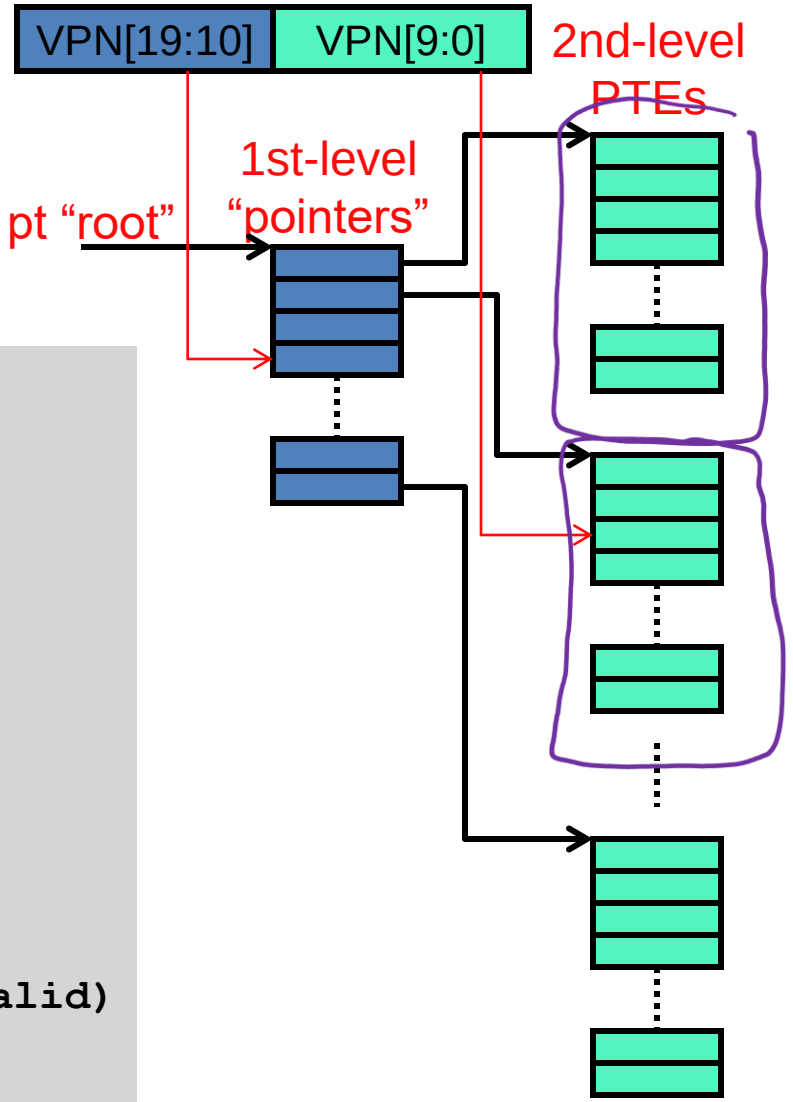
Multi-Level Page Table

- One way: **multi-level page tables**
  - Tree of page tables
  - Lowest-level tables hold PTEs
  - Upper-level tables hold pointers to lower-level tables
  - Different parts of VPN used to index different levels

- Example: two-level page table
  - Compute number of pages needed for lowest-level (PTEs)
    - 4KB pages / 4B PTEs $\rightarrow$ 1K PTEs/page
    - 1M PTEs / (1K PTEs/page) $\rightarrow$ 1K pages
  - Compute number of pages needed for upper-level (pointers)
    - 1K lowest-level pages $\rightarrow$ 1K pointers
    - 1K pointers * 32-bit VA $\rightarrow$ 4KB $\rightarrow$ 1 upper level page

Multi-Level Page Table

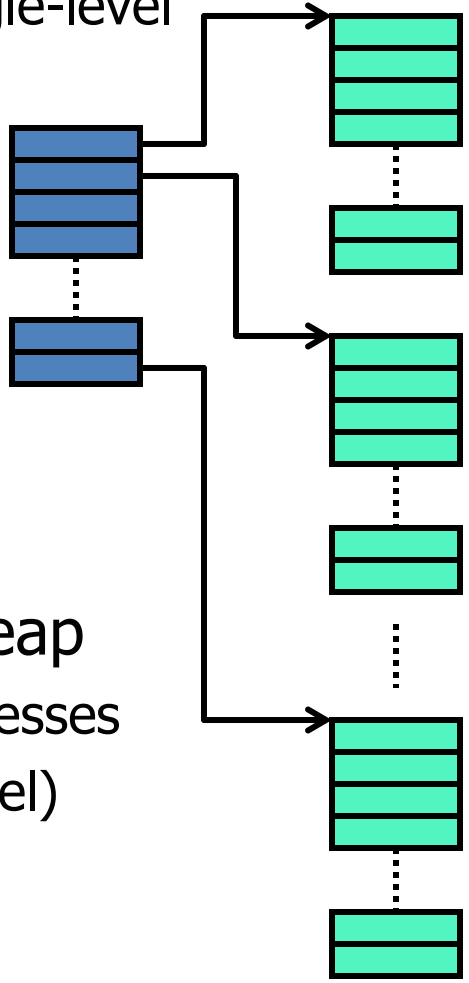
- 20-bit VPN
  - Upper 10 bits index 1st-level table
  - Lower 10 bits index 2nd-level table



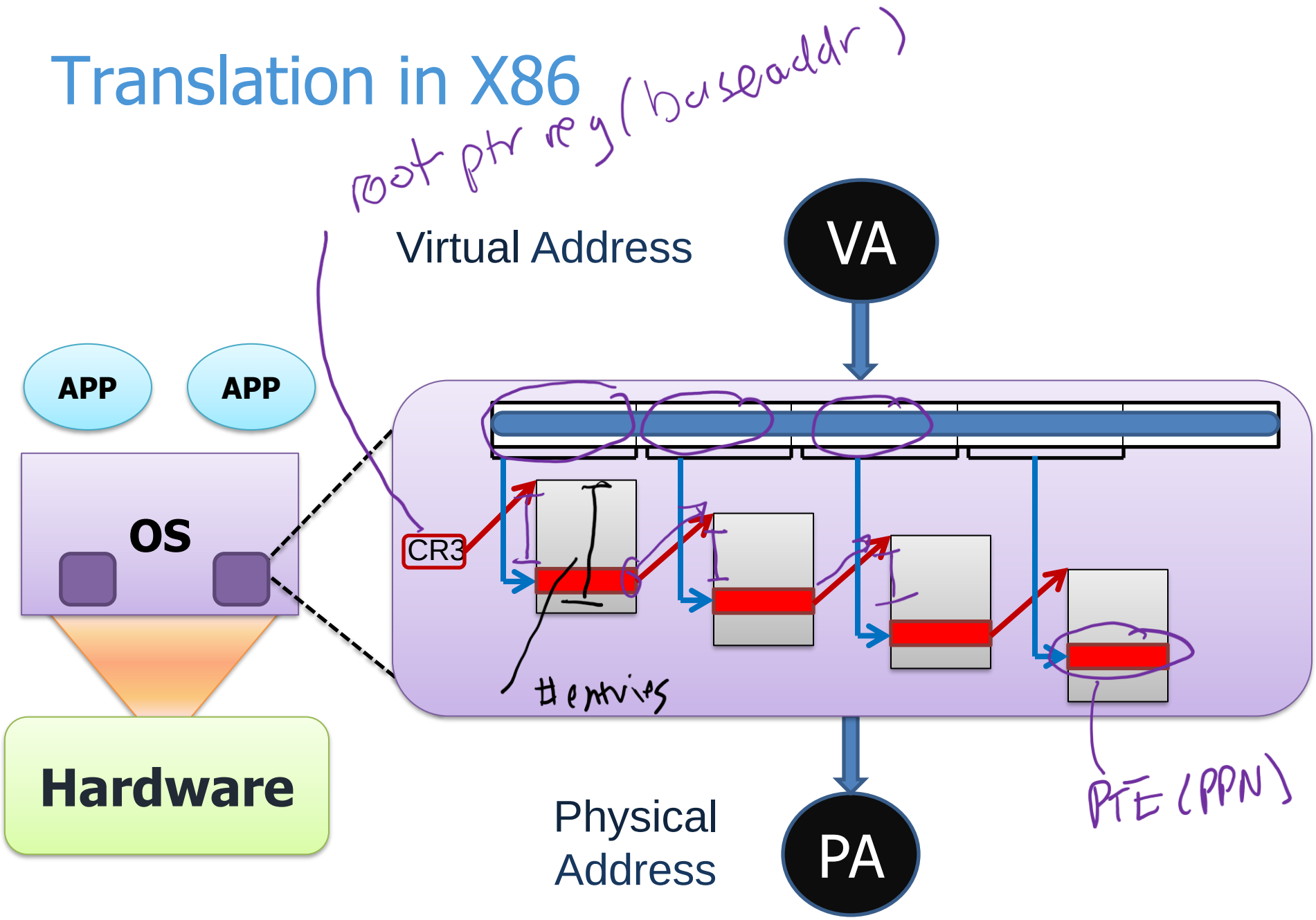

```
struct {
   union { int ppn, disk_block; }
   bool is_valid, is_dirty;
} PTE;

struct {
   struct PTE ptes[1024];
} L2PT;
struct L2PT *pt[1024];

int translate(int vpn) {
  struct L2PT *l2pt = pt[vpn>>10];
  if (l2pt && l2pt->ptes[vpn&1023].is_valid)
     return l2pt->ptes[vpn&1023].ppn;
}
```

Multi-Level Page Table (PT)

- Have we saved any space?
  - Isn't total size of 2nd level tables same as single-level table (i.e., 4MB)?
  - Yes, but…
- Large virtual address regions unused
  - Corresponding 2nd-level tables need not exist
  - Corresponding 1st-level pointers are null
- Example: 2MB code, 64KB stack, 16MB heap
  - Each 2nd-level table maps 4MB of virtual addresses
  - 1 for code, 1 for stack, 4 for heap, (+1 1st-level)
  - 7 total pages = 28KB (much less than 4MB)



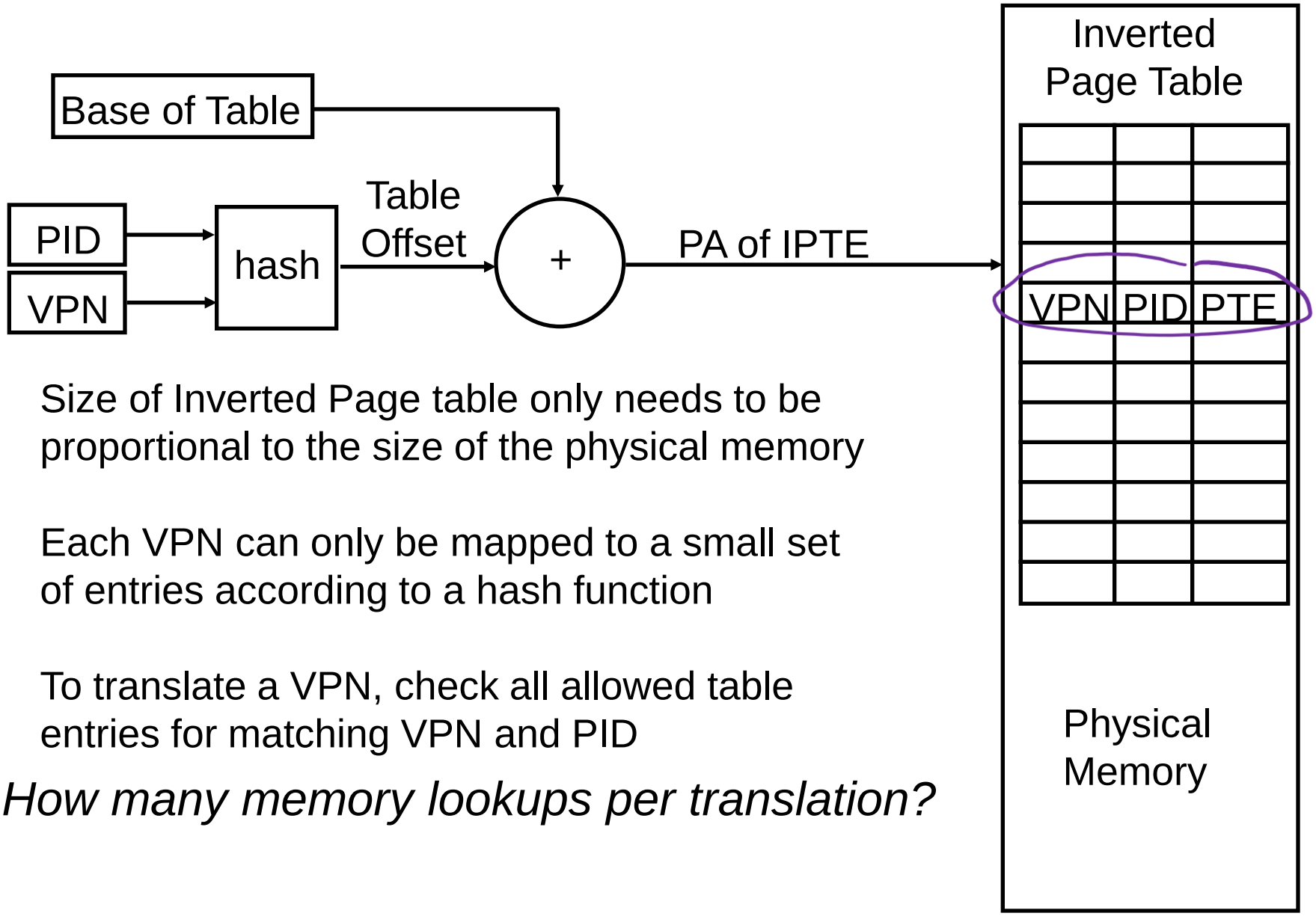
Translation in X86



Courtesy Jayneel Gandhi (Agile Paging)

At most mem accesses = 4

Alternative: Inverted/Hashed Page Tables



Size of Inverted Page table only needs to be proportional to the size of the physical memory

Each VPN can only be mapped to a small set of entries according to a hash function

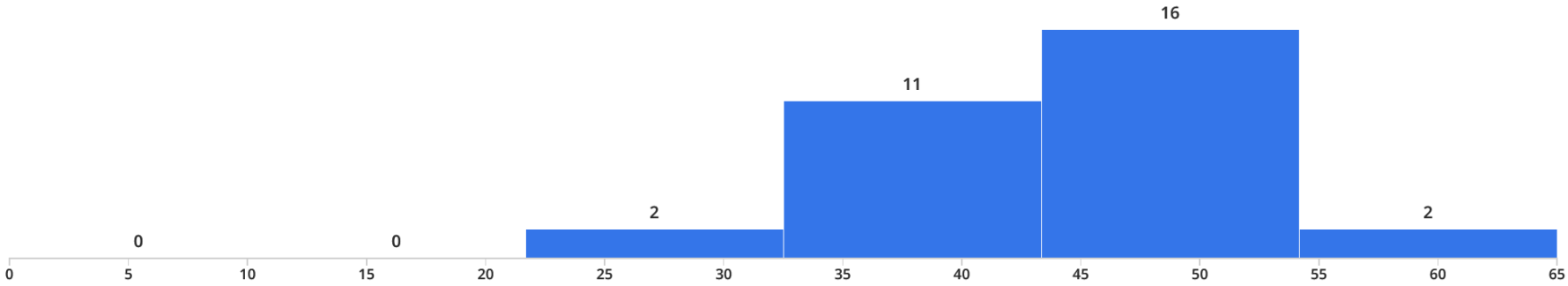
To translate a VPN, check all allowed table entries for matching VPN and PID

How many memory lookups per translation?

Announcements 10/24/24

- Midterm grades released
  - Regrade Requests due by 10/31/24
  - Grading:
    - Rotated who was graded first
    - Attempted to grade as anonymously as possible
  - My post-exam thoughts
- Proposals will be graded soon
- HW4 grading is through PLRU
- HW5 due Saturday (tomorrow)
- HW6 posted – due next Friday
- HelioCampus mid-semester evals released but …
  - With focus on getting grades out I didn't get a chance to look at yet
  - Will look at and respond soon

Midterm 1: Rough Breakdown



- Average: 67.2% (43.7 / 65 points)
- Rough Grades
  - A: 75% - 100% (> 48 points)
  - AB: 65% - 74.5% (41.6 - 47.9 points)
  - B: 50% - 63.5% (32 - 41.5 points)
  - <B: < 50% (< 32 points)

Midterm 1 Common Issues

- Power (1B)
  - Seemed to be the most commonly skipped problem
  - Beyond that: various issues with static vs. dynamic power
- P6 (5B)
  - A variety of issues
  - Missing issue (S) column and issuing things too soon or too late
  - Not allowing multiple instructions to go to different FUs (X) simultaneously
  - Missed spec saying have to wait one cycle to go into X when broadcast on CDB
  - Forgot to do in-order retire
  - Misidentified source/destination registers
  - Labeled operand as coming from RS instead of RF/ROB/CDB

Midterm 1: Last Words

- Don’t panic – lots of grades to go
  - But we can talk if it would help
- If you struggled with certain concepts, consult solutions, come to office hours, etc.
- For midterm 2 make sure to read directions carefully

Address Translation Mechanics

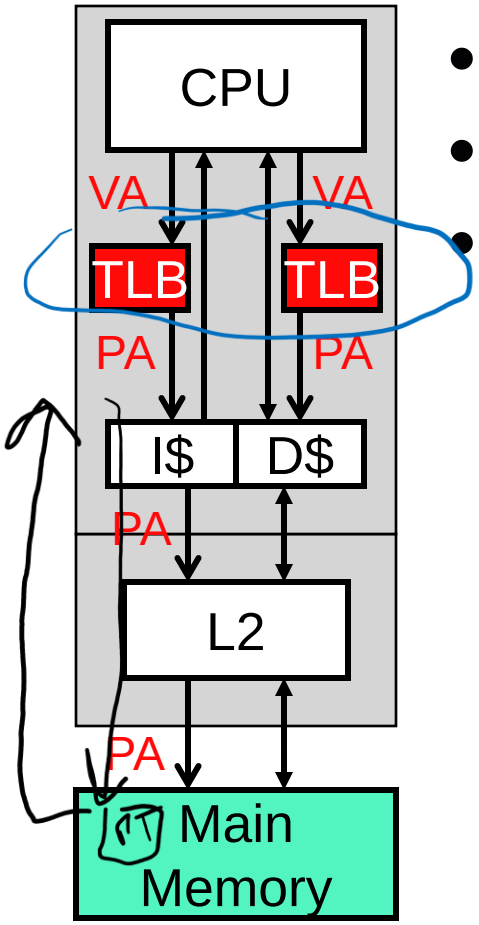
- Three remaining questions
  - **Who performs translation?**
  - **When do you translate?**
  - **Where does page table reside?**

- Conceptual view:
  - Who: OS
    - Disallow program from modifying its own page table entries
  - When: Translate virtual address before every cache access
    - Walk the page table for every load/store/instruction-fetch
  - Where: Memory

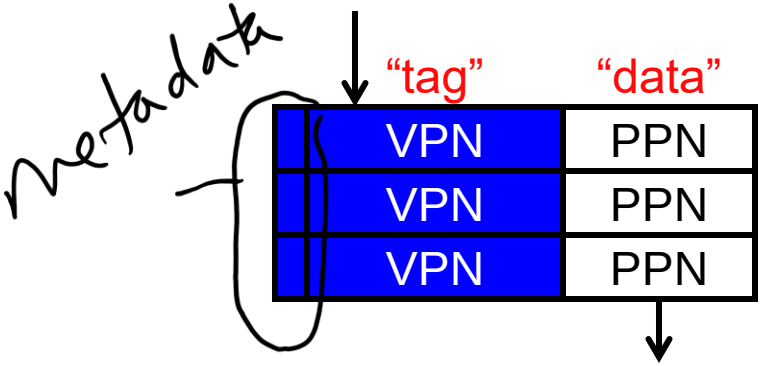
really slow if many PT "misses"

- Actual approach:
  - Cache translations in a "translation cache" to avoid repeated lookup

Translation Lookaside Buffer (TLB)



- Functionality problem? add indirection
- Performance problem? add cache
- Address translation too slow?
  - Cache translations in **translation lookaside buffer (TLB)** needs to be fast ← on crit. path
    - Small cache: 16–512 entries
    - Small TLBs often fully associative (<64)
  - \+ Exploits temporal locality in page table (PT)
  - What if an entry isn't found in the TLB?
    - Invoke TLB miss handler




```
if (tag = VPN)
    return PPN;
```

TLB Misses and Miss Handling

- **TLB miss:** requested PTE not in TLB, search page table
  - **Software routine**, e.g., Alpha, SPARC, MIPS
    - Special instructions for accessing TLB directly
    - Latency: one or two memory accesses + trap
  - **Hardware finite state machine (FSM),** e.g., x86
    - "Page table walker"
    - Store page table root in hardware register (CR3)
    - Page table root and table pointers are physical addresses
    - \+ Latency: saves cost of OS call
  - In both cases, reads use the the standard cache hierarchy
    - \+ Allows caches to help speed up search of the page table

multi levels of TLB

- **Nested TLB miss**: miss handler itself misses in the TLB
  - Solution #1: Allow recursive TLB misses (very tricky)
  - Solution #2: Lock TLB entries for page table into TLB (pinning pages)
  - *Solution #3: Avoid problem using physical address in page table*

TLB Performance

- TLB Reach = # TLB entries * Page size
  = 64 * 4KB = 256KB << L2 cache size

Solution #1: Big pages (e.g., 4MB)

TLB Reach = 256MB, but internal fragmentation

How to support both big and small pages?

Solution #2: Two-level TLB

L1: 64-128 entries, L2: 512-2048 entries

Solution #3: Software TLB (aka TSB)

in memory TLB: 32K entries (or more)

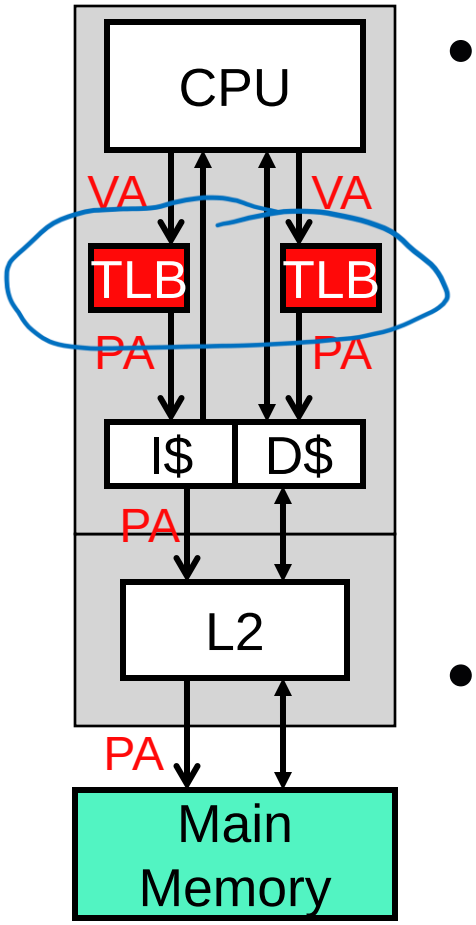
low-associativity (e.g., 2-way), longer hit time

Much faster than page table access

Page Faults

- **Page fault**: PTE not in page table
  - Page is simply not in memory
  - Starts out as a TLB miss, detected by OS handler/hardware FSM (trap) (PTW)
- **OS routine**
  - Choose a physical page to replace
    - **"Working set"**: more refined software version of LRU
      - Tries to see which pages are actively being used
      - Balances needs of all current running applications (avoid starvation)
    - If dirty, write to disk
  - Read missing page from disk
    - Takes so long (~10ms), **OS schedules another task**
  - Treat like a normal TLB miss from here

Physical (Address) Caches



- Memory hierarchy so far: **physical caches**
  - Indexed and tagged by Pas
    - Physically Indexed (PI)
    - Physically Tagged (PT)
  - Translate to PA at the outset
  + Cached inter-process communication works
    - Single copy indexed by PA
  – Slow: adds at least one cycle to t_{hit}
- Prior classes: VIPT, VIVT

PIPT

VIPT

L2 [Sandberg]

TLB Organization

- **Like caches**: TLBs also have ABCs
  - Capacity
  - Associativity (At least 4-way associative, fully-associative common)
  - What does it mean for a TLB to have a block size of two?
    - Two consecutive VPs share a single tag

- **Like caches**: there can be L2 TLBs
  - Why? Think about this…

- **Rule of thumb**: TLB should "cover" L2 contents (maybe LLC)
  - In other words: (#PTEs in TLB) * page size ≥ L2 size
  - Why? Think about relative miss latency in each…

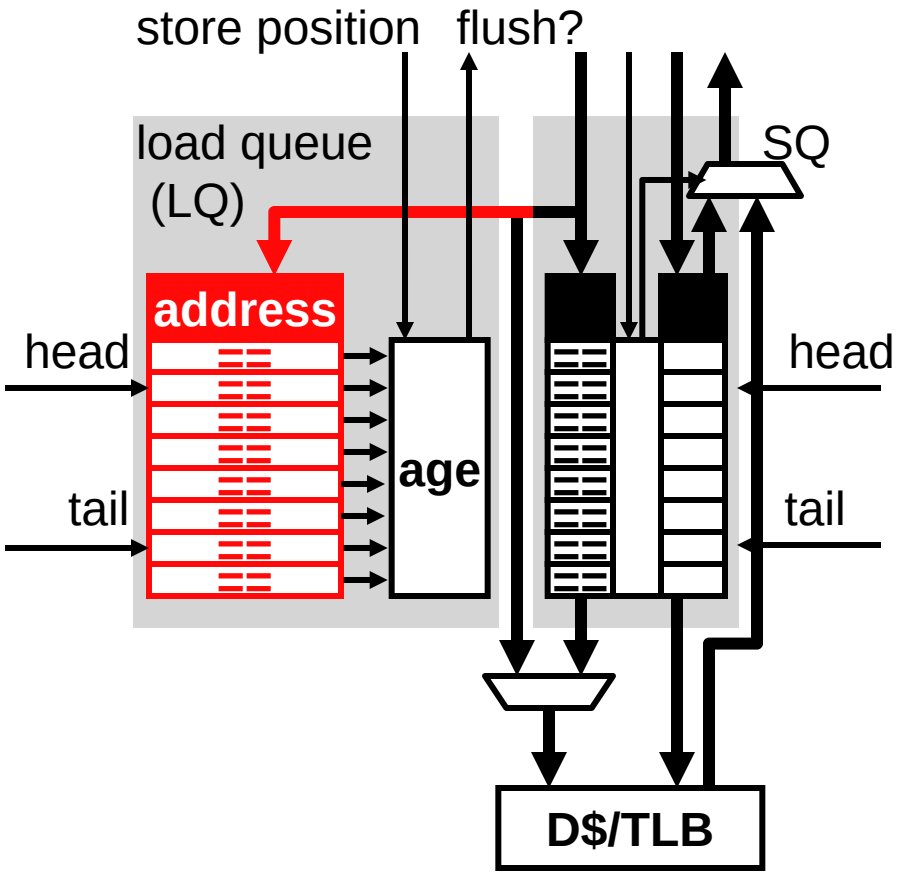
reach

$ miss: 1 cycle access → lower levels of mem.

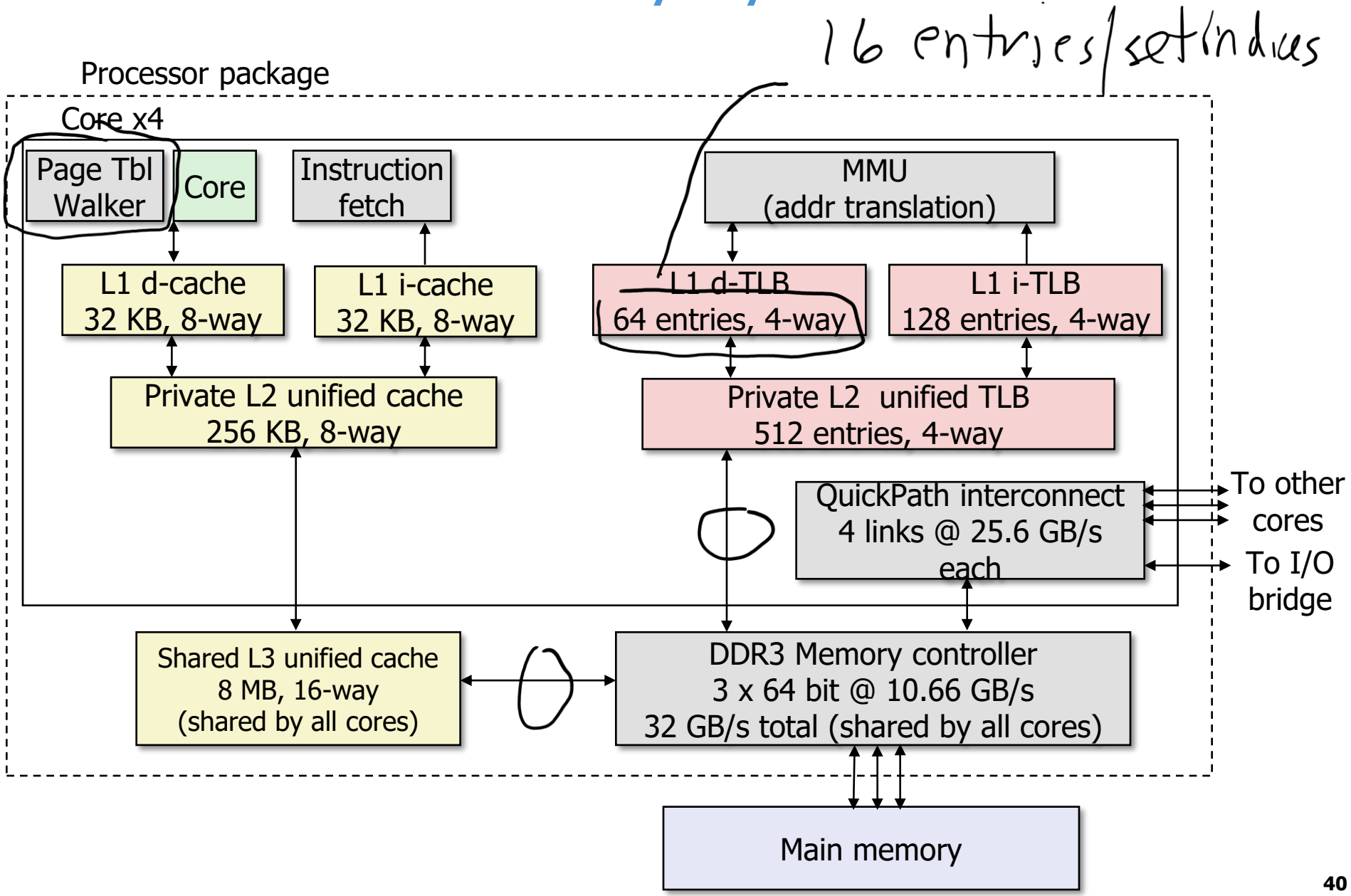
TLB miss: # levels of PT misses (max)

What addresses in the pipeline?

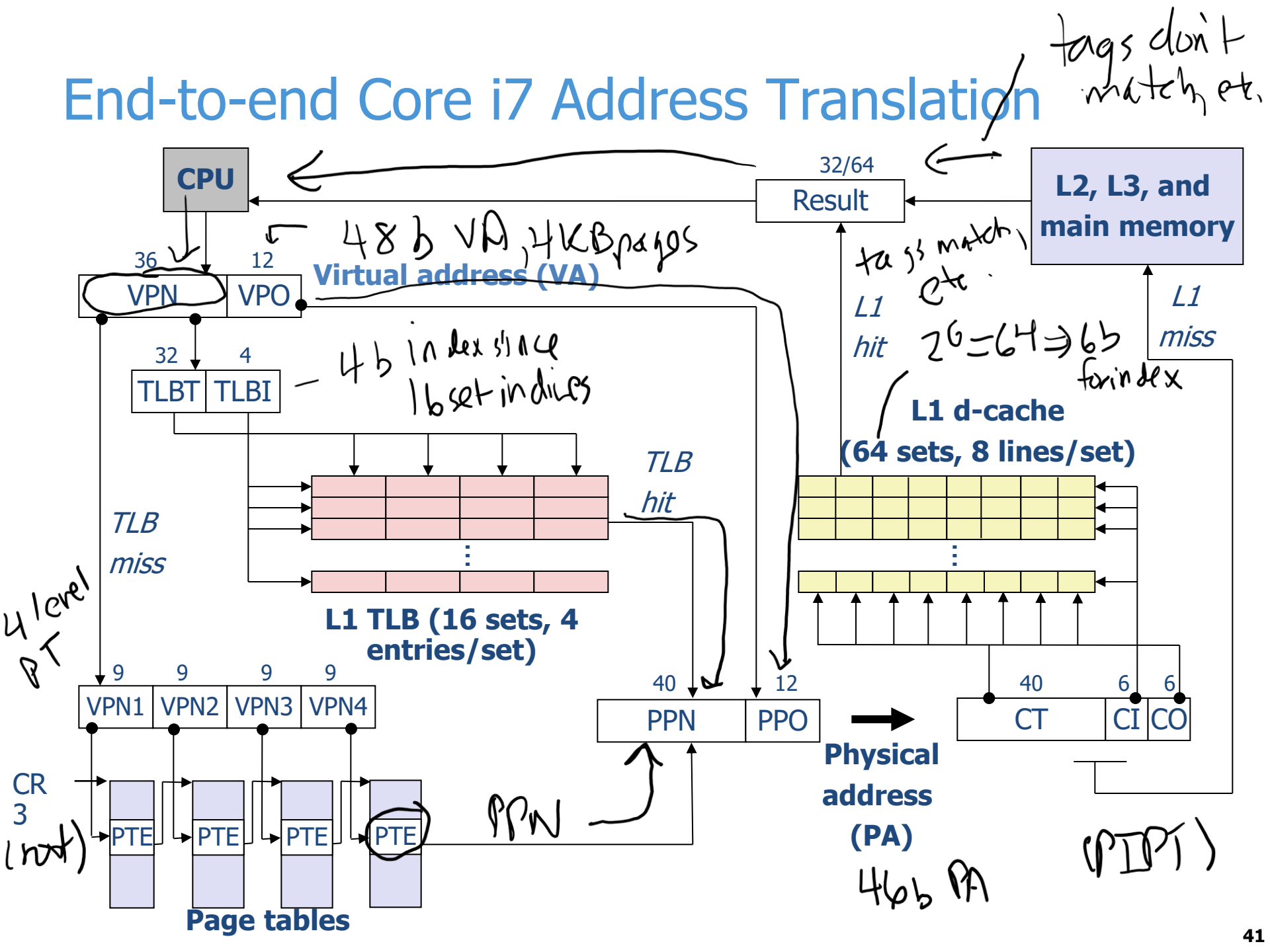
- What type of address for LSQ?
- (synonyms within a process -> same physical)



Intel Core i7 Memory System

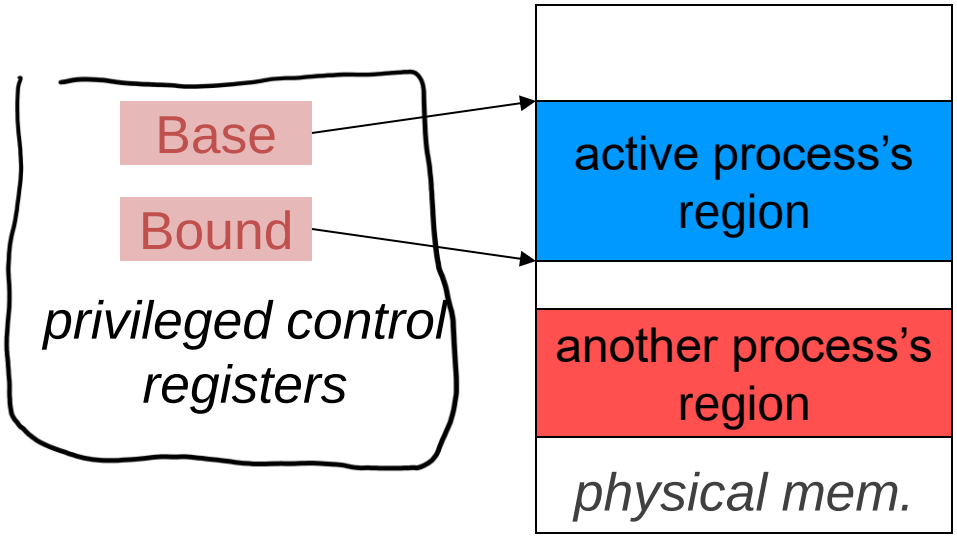


End-to-end Core i7 Address Translation



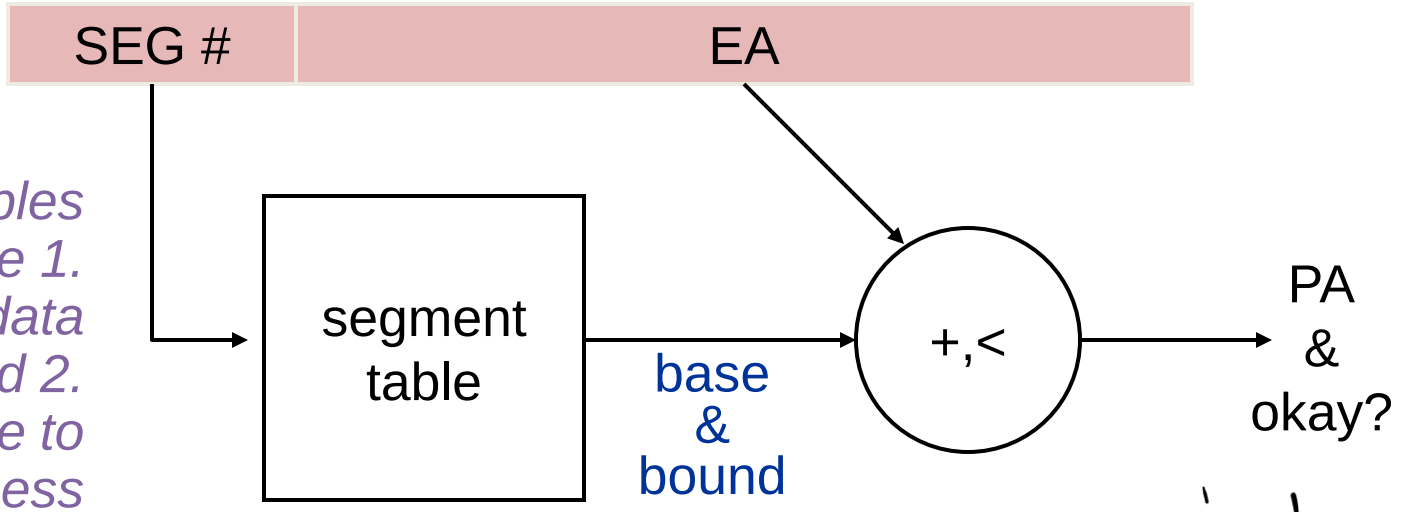
Alternative to VM: base/bound registers

- Each process is given a non-overlapping, contiguous physical memory region
- When a process is swapped in, OS sets base to the start of the process's memory region and bound to the end of the region
- On memory references, HW translation & protection check
- PA = EA + base
- provided (PA < bound),
- else violations



Also Segmented Address Space

- segment == a base and bound pair
- segmented addressing gives each process multiple segments
  - initially, separate code and data segments
    - *2 sets of base-and-bound reg's for inst and data fetch*
    - *allowed sharing code segments*
  - became more and more elaborate: *code, data, stack, etc.*
  - also (ab)used as a way for an ISA with a small EA space to address a larger physical memory space



segment tables must be 1. privileged data structures and 2. private/unique to each process

Cheaper translation, but fragmentation? + easier to manipulate

Virtual Memory

- Virtual memory ubiquitous today
  - Certainly in general-purpose (in a computer) processors
  - But even many embedded (in non-computer) processors support it

- Several forms of virtual memory
  - **Paging** (aka flat memory): equal sized translation blocks
    - Most systems do this
  - **Segmentation**: variable sized (overlapping?) translation blocks
    - x86 used this rather than 32-bits to break 16-bit (64KB) limit
    - Memory allocation not fun… (fragmentation?)
  - **Paged segments**: too many indirections…
  - **Today:** X86-64 does not use segmentation in 64-bit mode
    - forces all segment registers (CS,SS,DS,ES to 0-2^64)

Memory Protection and Isolation

- Important role of virtual memory today

- Virtual memory protects applications from one another
  - OS uses indirection to isolate applications
  - One buggy program should not corrupt the OS or other programs
  + Comes "for free" with translation
  – However, the protection is limited

– What about protection from…
    - Viruses and worms?
      - Stack smashing
    - Malicious/buggy services?
      - Other applications with which you want to communicate

Page-Level Protection

```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty, permissions;
} PTE;
```

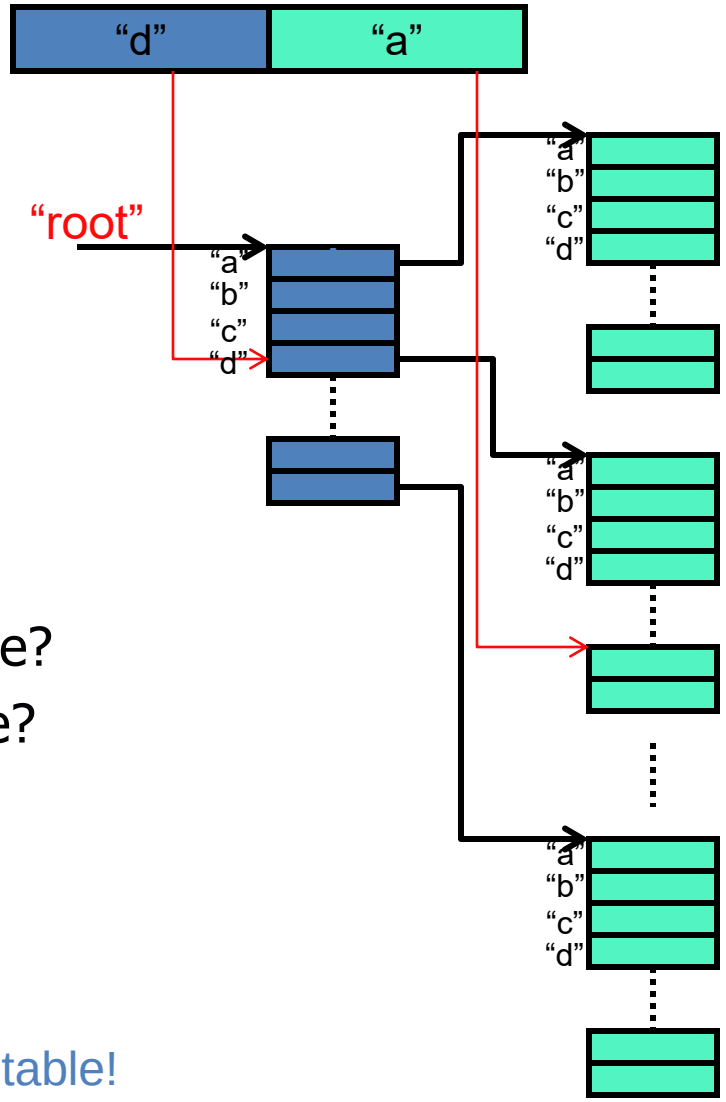
- **Page-level protection**
  - Piggy-backs on translation infrastructure
  - Each PTE associated with permission bits: **R**ead, **W**rite, e**X**ecute
    - **Read/execute (RX)**: for code
    - **Read (R)**: read-only data
    - **Read/write (RW)**: read-write data
  - TLB access traps on illegal operations (e.g., write to **RX** page)
  - To defeat stack-smashing? Set stack permissions to **RW**
    - Will trap if you try to execute `&buf[0]`

  – Unfortunately, hackers have many other tricks (return oriented programming)

Bonus

Here's an interesting datastructure!

- What is a "**trie**" data structure
  - Also called a "prefix tree"
- What is it used for?
- What properties does it have?
  - How is it different from a binary tree?
  - How is it different than a hash table?



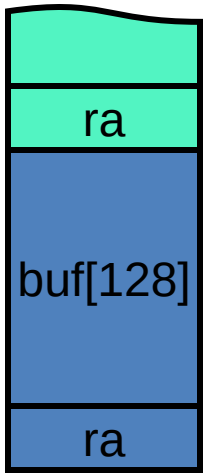
Better worst case # accesses compared to hash table!

Typical I/O Device Interface

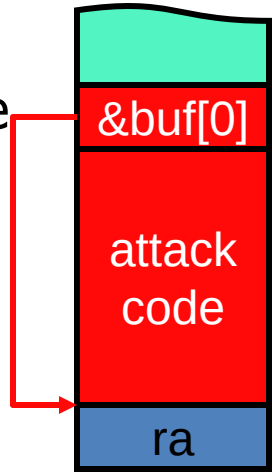
- Operating system talks to the I/O device
  - Send commands, query status, etc.
  - Software uses special uncached load/store operations
  - Hardware sends these reads/writes across I/O bus to device
- Direct Memory Access (DMA)
  - For big transfers, the I/O device accesses the memory directly
  - Example: DMA used to transfer an entire block to/from disk
- Interrupt-driven I/O
  - The I/O device tells the software its transfer is complete
  - Tells the hardware to raise an "interrupt" (door bell)
  - Processor jumps into the OS
  - Inefficient alternative: polling

Stack Smashing via Buffer Overflow

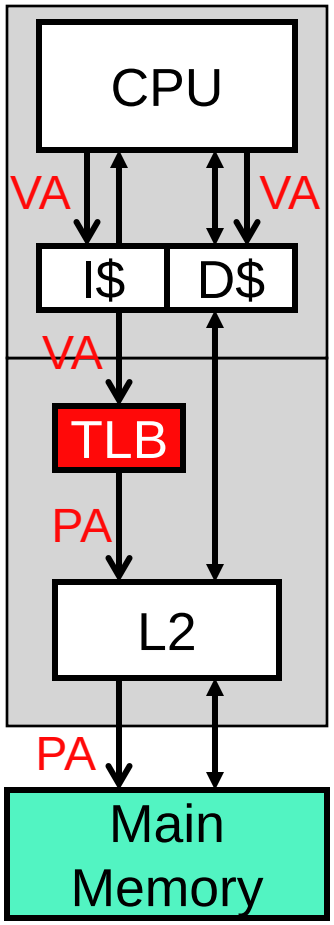
```
int i = 0;
char buf[128];
while ((buf[i++] = getc()) != '\n') ;
return;
```



- Stack smashing via buffer overflow
  - Oldest trick in the virus book
  - Exploits stack frame layout and…
  - Sloppy code: **length-unchecked copy to stack buffer**
  - "Attack string": **`code`** (128B) + **`&buf[0]`** (4B)
  - Caller return address replaced with pointer to attack code
    - Caller return…
    - …executes attack code at caller's privilege level

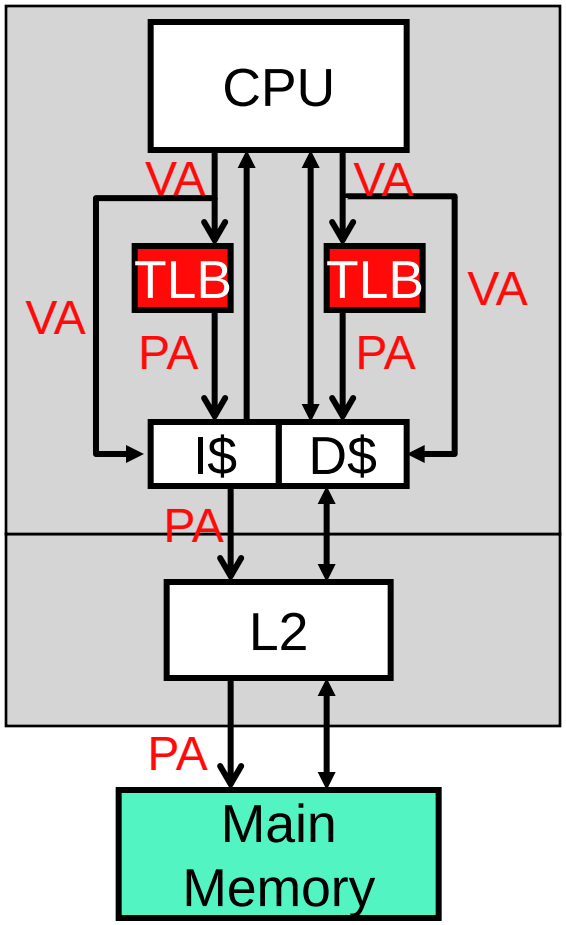


Virtual Address Caches (VI/VT)

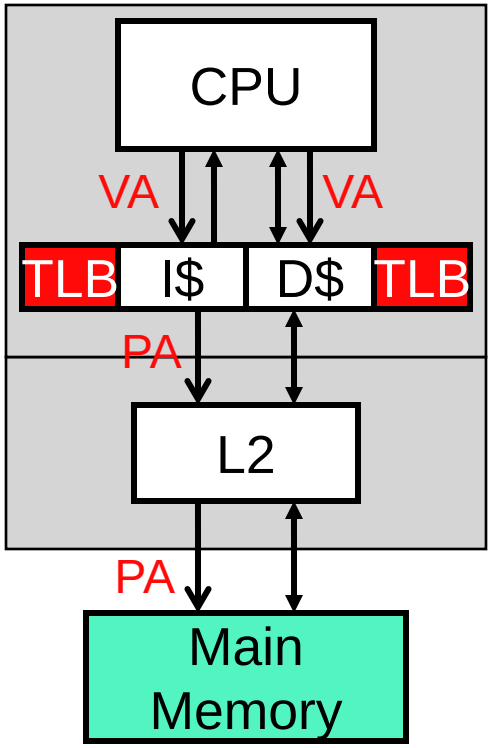


- Alternative: **virtual caches**
  - Indexed and tagged by VAs (VI and VT)
  - Translate to PAs only to access L2
  - \+ Fast: avoids translation latency in common case
  - – Problem: VAs from ***different processes*** are distinct physical locations (with different values) (**homonyms**: same address different meaning)
- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags
- Does inter-process communication work?
  - **Synonyms**: multiple VAs map to same PA
    - Can't allow same PA in the cache twice
  - Can be handled, but very complicated (bonus)

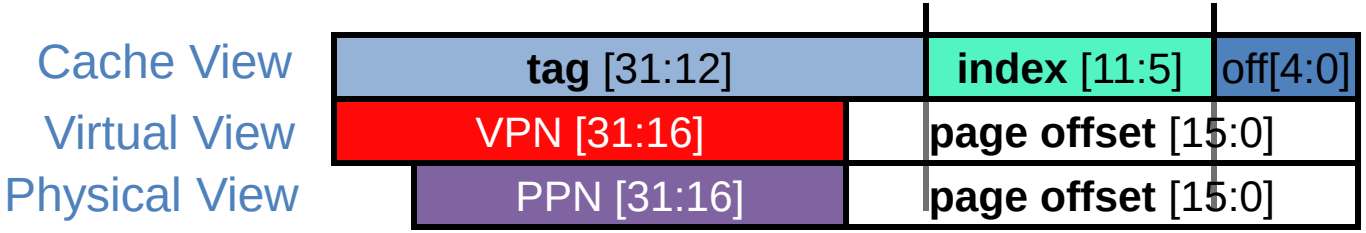
PIVT



Parallel TLB/Cache Access (VI/PT)

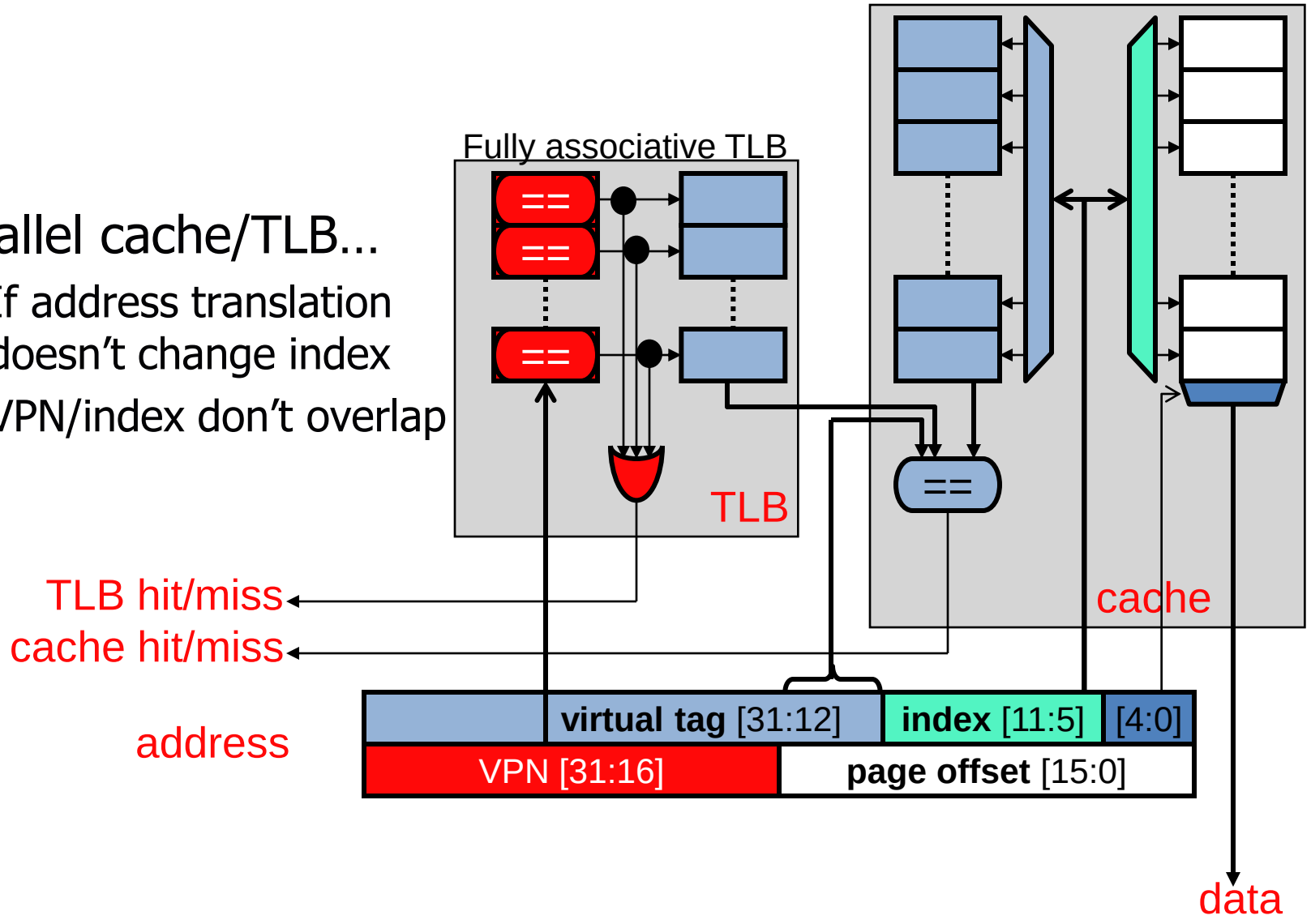


- Compromise: **access TLB in parallel**
  - *In small caches, index of VA and PA the same*
    - *VI == PI*
  - Use the VA to index the cache
  - Tagged by PAs
  - Cache access and address translation in parallel
  - \+ No context-switching/aliasing problems
  - \+ Fast: no additional t_{hit} cycles
  - Common organization in processors today

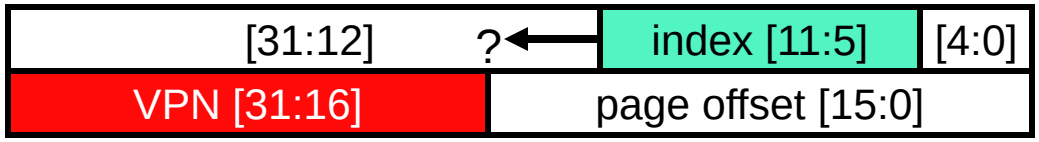


Parallel Cache/TLB Access

- Parallel cache/TLB…
  - If address translation doesn't change index
  - VPN/index don't overlap



Cache Size And Page Size



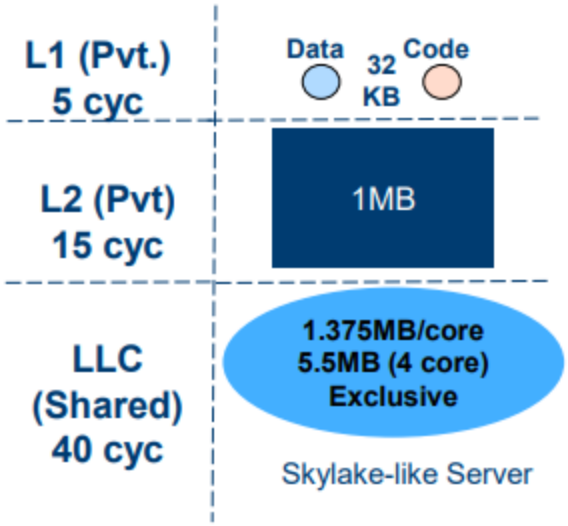
- Relationship between page size and L1 cache size
  - Forced by non-overlap between VPN and IDX portions of VA
    - Which is required for TLB access
  - **Rule: (cache size) / (associativity) ≤ page size**
  - Result: associativity increases allowable cache sizes
  - Example: Pentium 4, 4KB pages, 8KB, 2-way SA L1 data cache
- If cache is too big, same issues as virtually-indexed caches

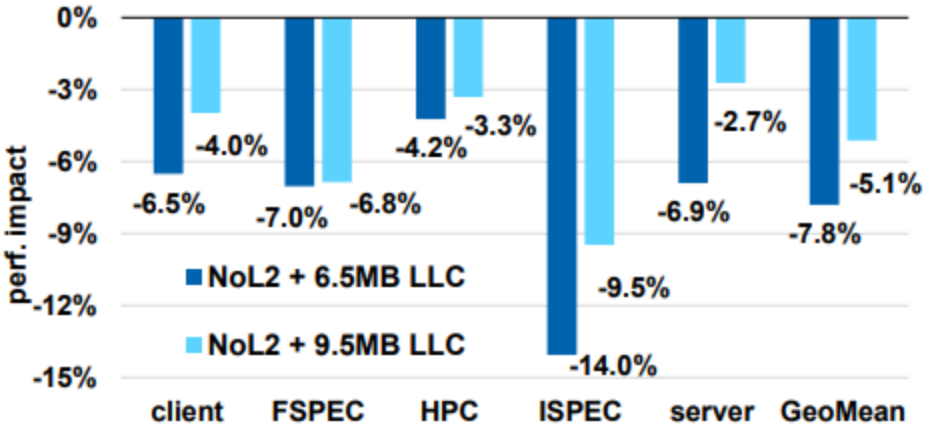
(not really VM, but let's talk about it)

- Criticality Aware Cache Hierarchies
  - Problem?
  - Insight?
  - Approach?
  - Benefits?

Popular Three Level Cache Hierarchy

- Cache capacity ↔ Access latency
  - Target low *average* latency
- Large distributed LLC, high latency
  - Lower L2 latency important



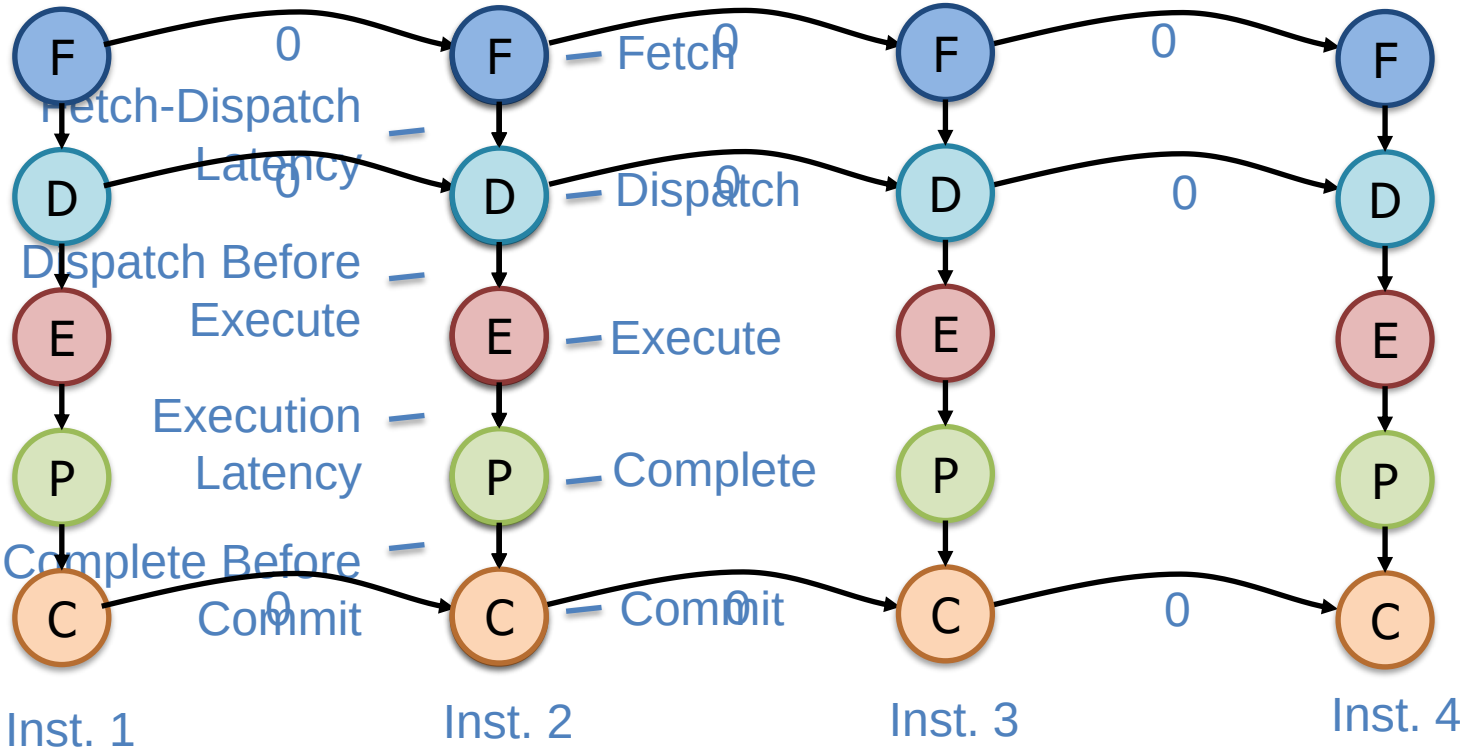


Background: μArch Dependence Graphs

[Fields, ISCA 2001]

- Represent ex… nodes.
- Edges betwe… encies.

Inorder Fetch/Dispatch/Commit

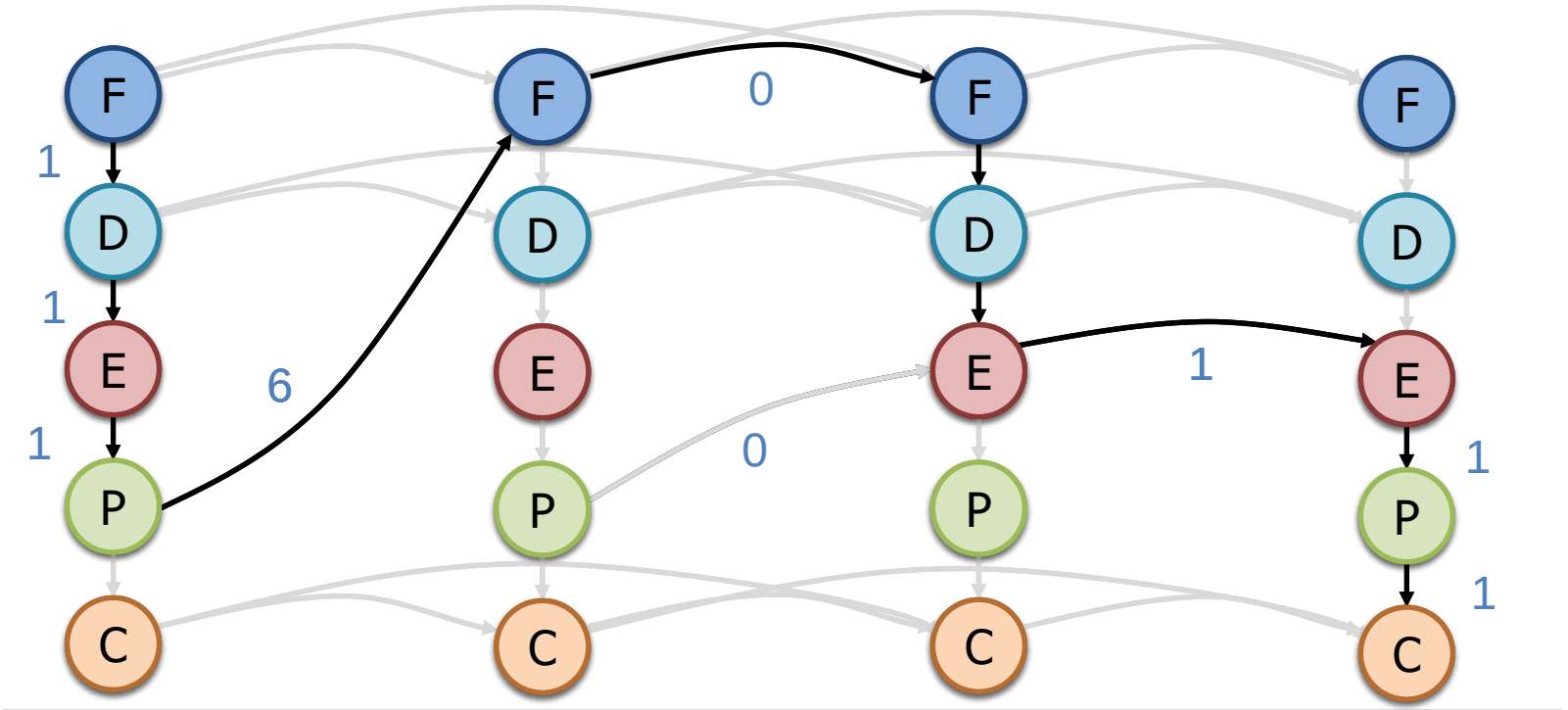


Background: μArch Dependence Graphs

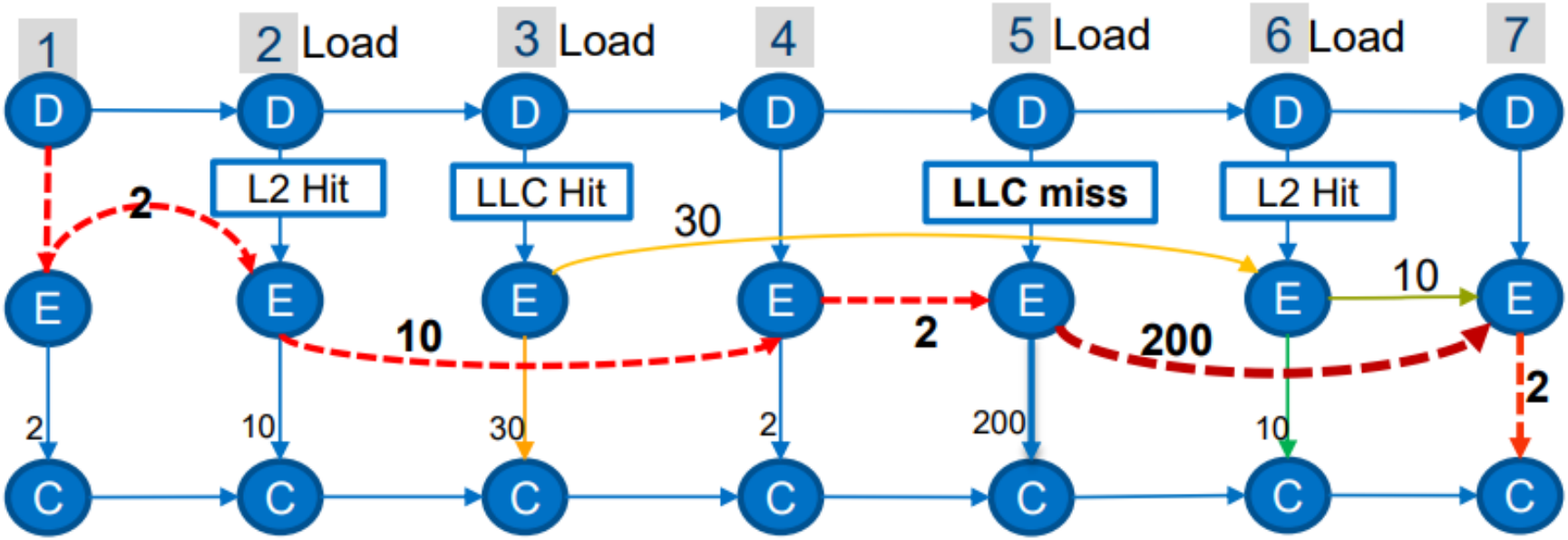
Branch Mispredic

Critical Path → Execution Time

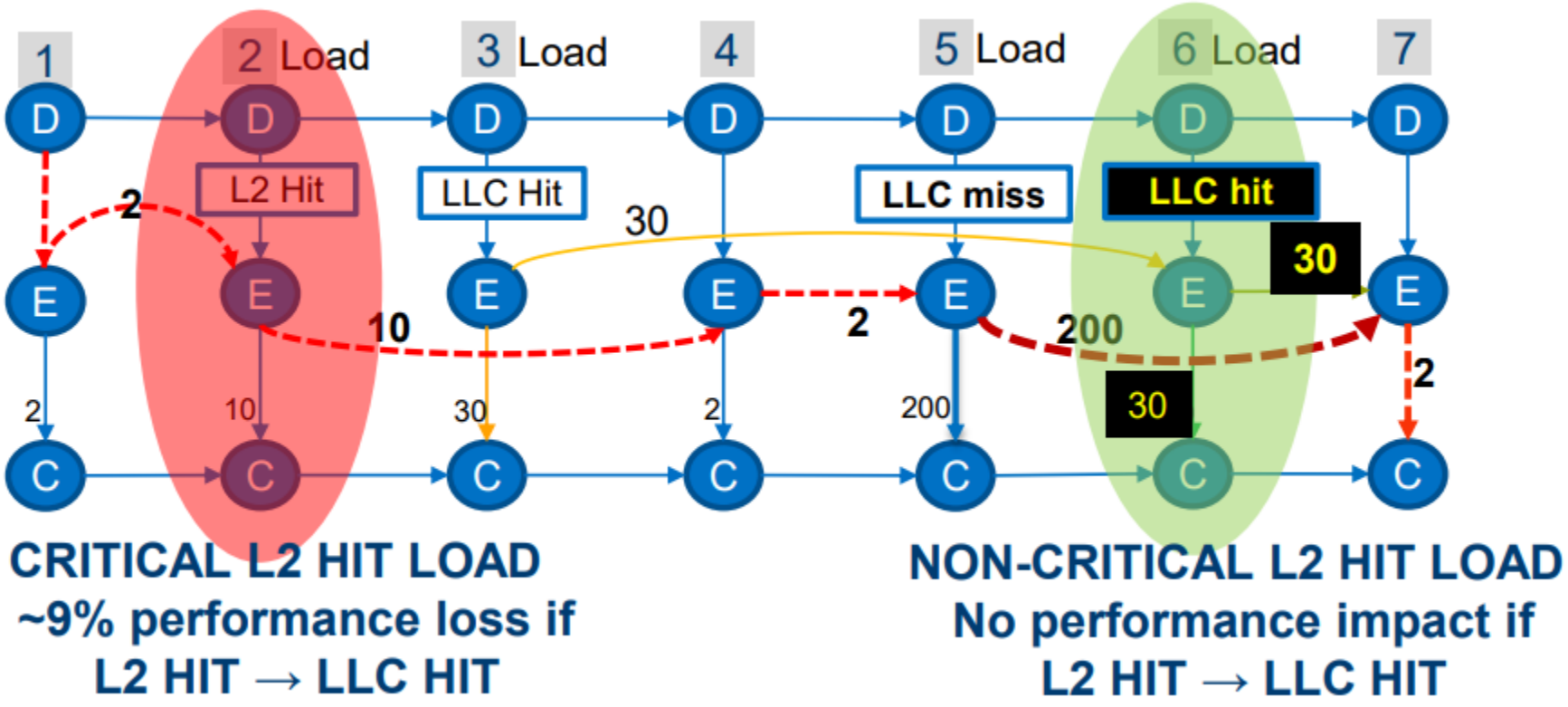
urce dence



- Reduced complexity model with only
  - D: Dispatch
  - E: Execute
  - C: Commit



Claim: Only critical L2 Hits matter



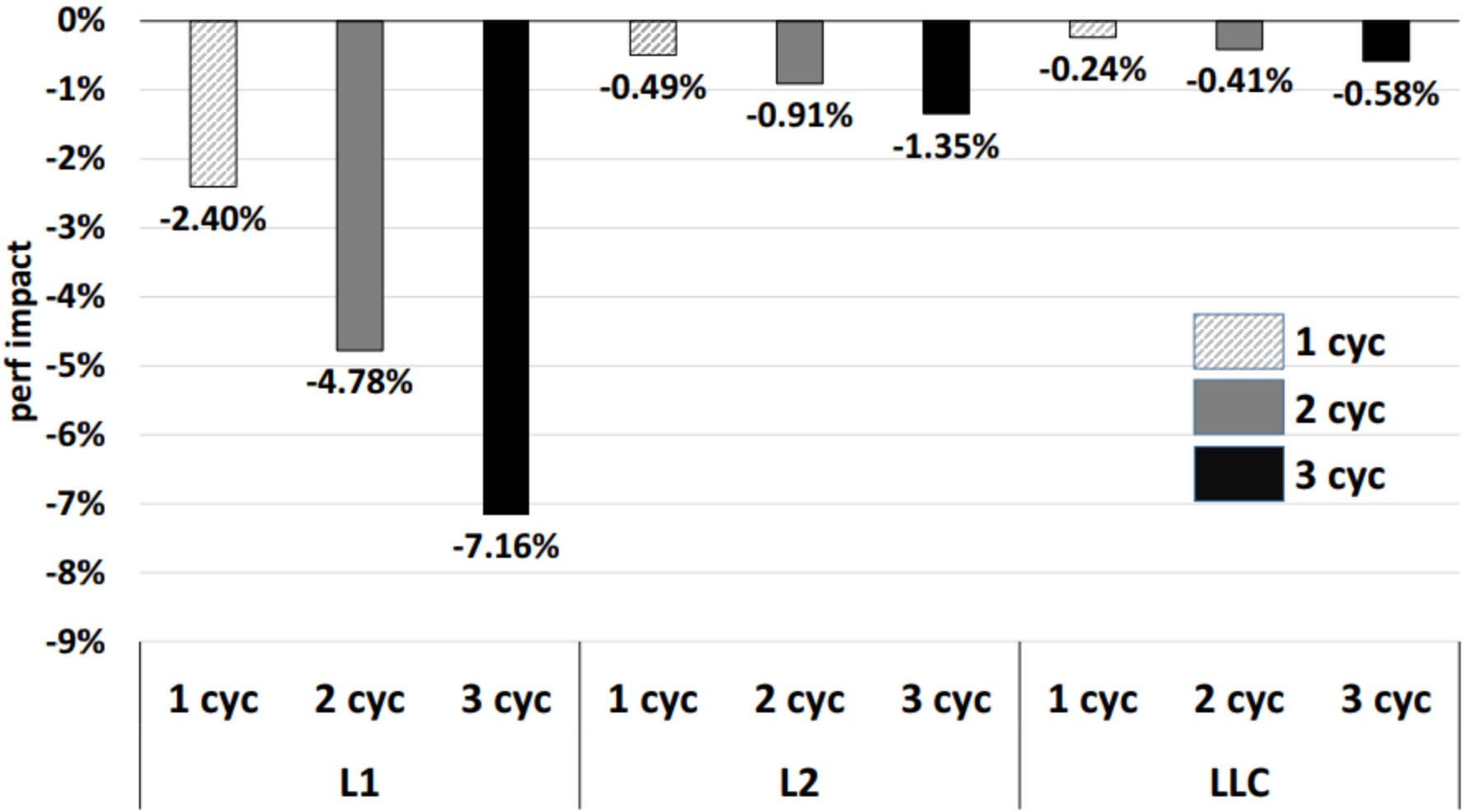


Figure 3. Impact of latency increase in L1, L2 and LLC

Intuition: Latency matters much more for L1 cache!

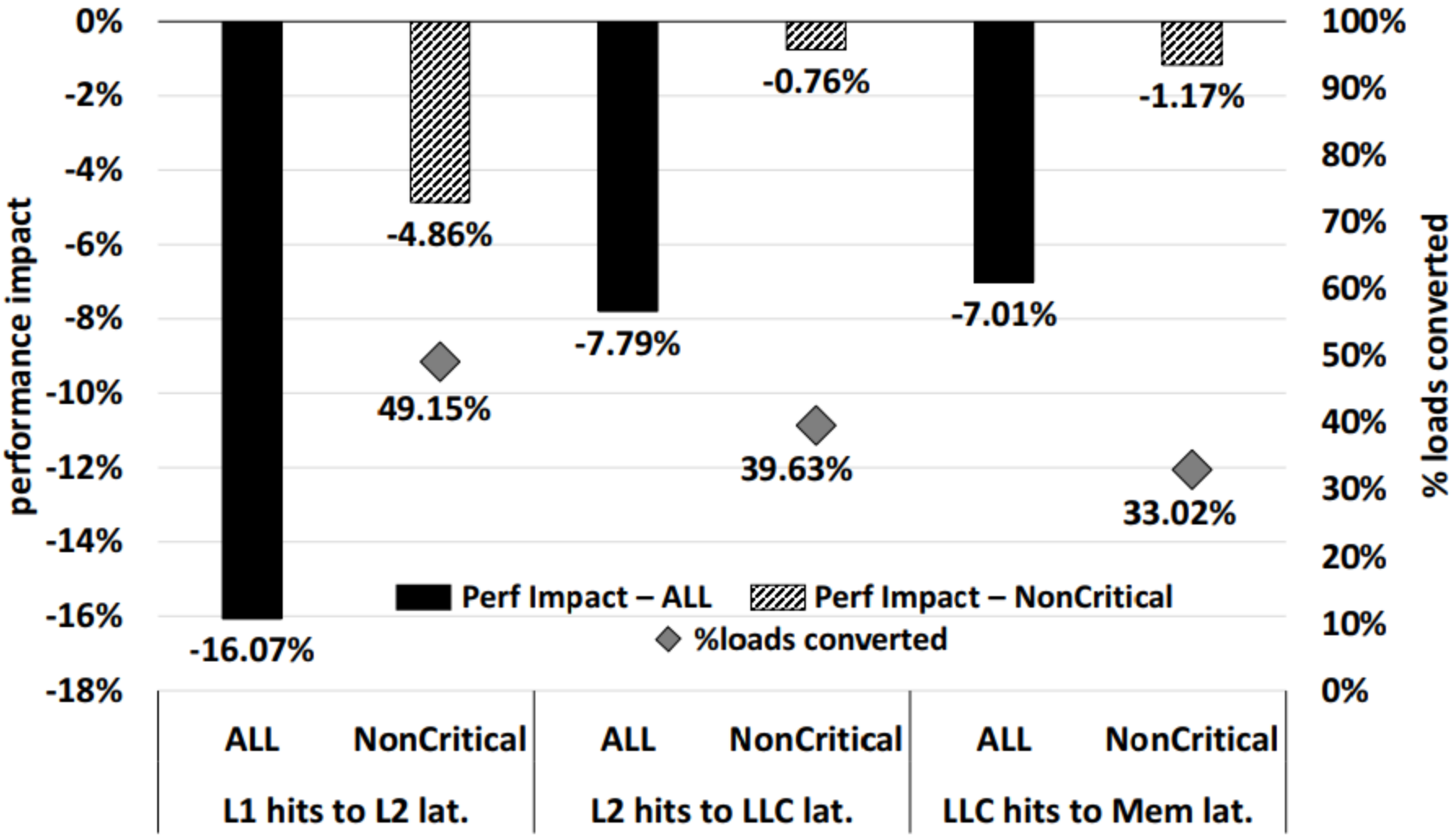
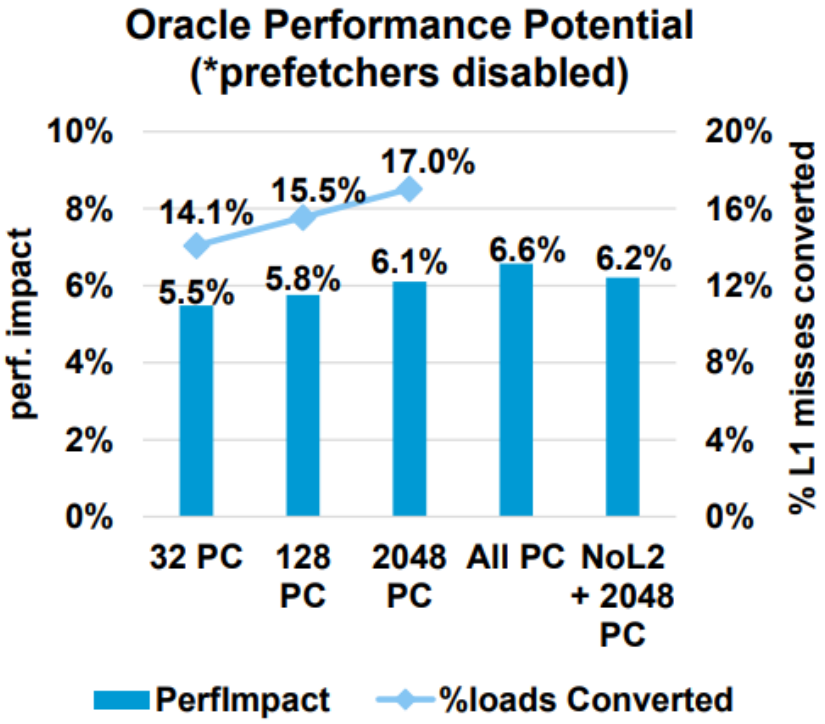


Figure 4. Impact of increasing non-critical load latency

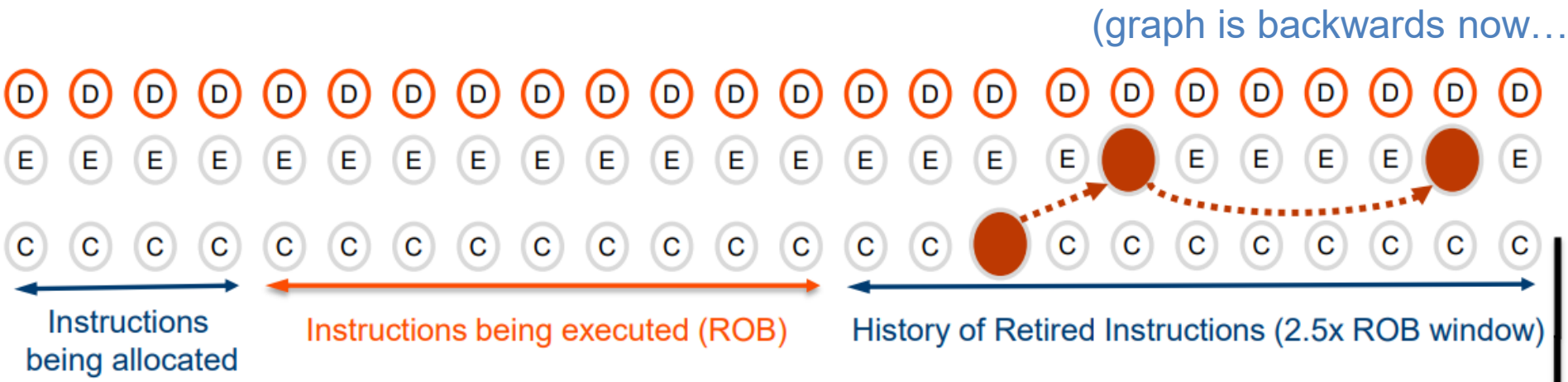
Intuition: L2 cache least sensitive to non-critical access latency

Criticality Aware Tiered Cache Hierarchy (CATCH)

- Track critical load PCs
  - Served from non-L1 on-die caches
- Prefetch critical loads into L1
  - Accelerate the critical path



Criticality detection in hardware



Buffer execution DDG (Fields et. al.) on instruction retire

Enumerate critical path every 2x ROB instruction retires

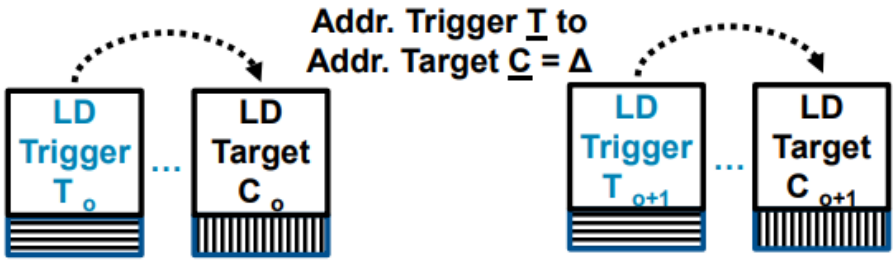
Optimizations: Area of DDG, Fast enumeration of critical path

32 entry Critical Load PC Table

TACT: Data Prefetchers

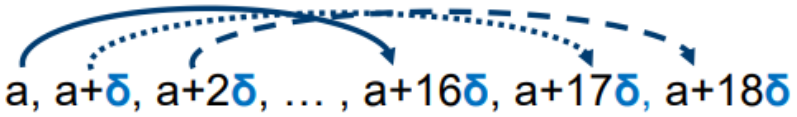
"Cross" Prefetcher:

- Trigger load PC address @ constant delta from Target/Critical PC

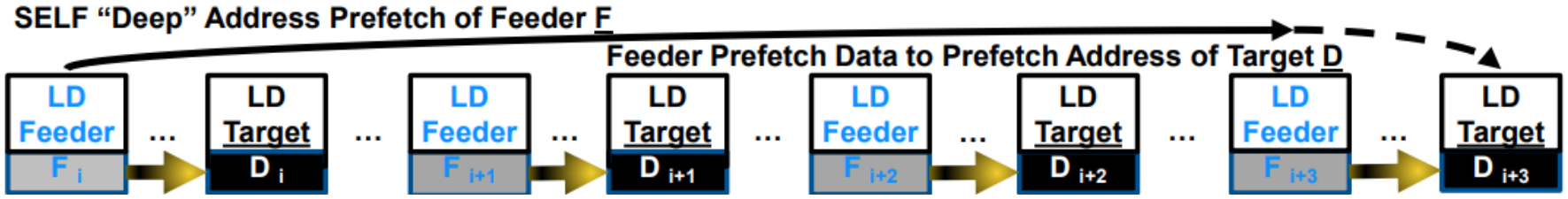


"Self" Deep Prefetcher:

- Upto deep prefetch distance 16

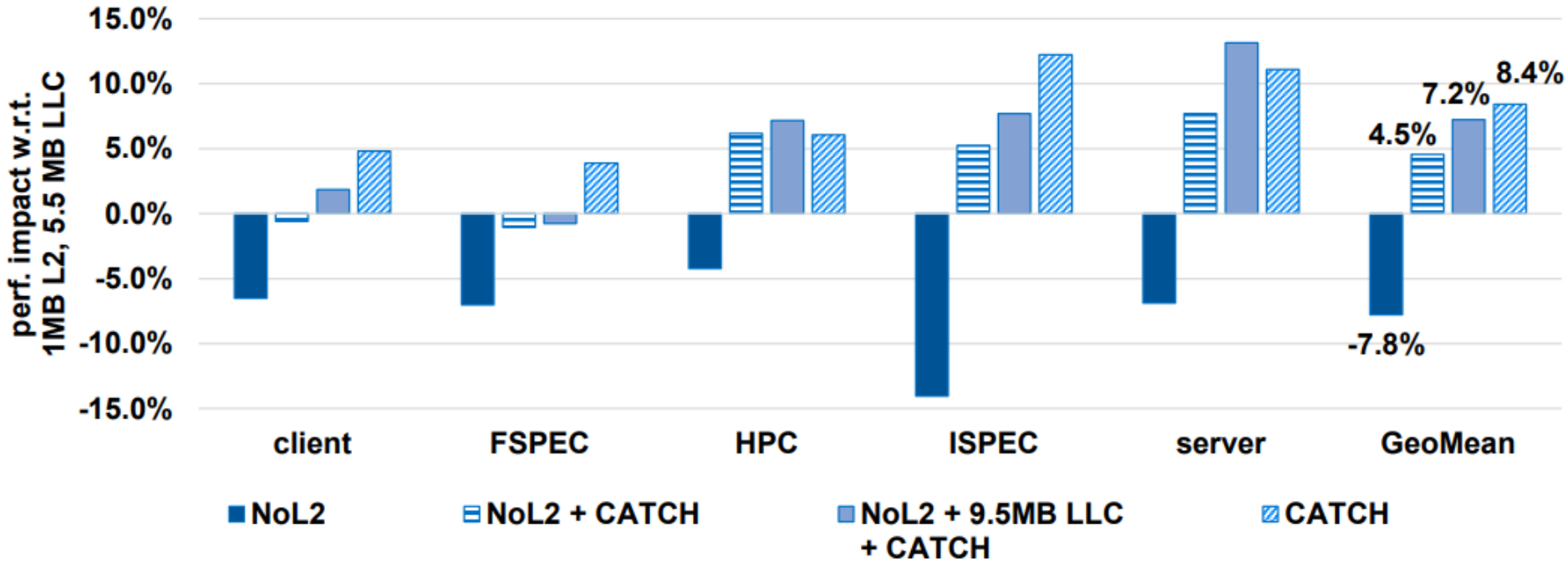


"Feeder" Prefetcher: Address of Target/Critical load = ***M*** * Data of Feeder load + ***C***



Large 1MB L2, Exclusive 1.375 MB LLC per Core

ST GeoMean Performance Impact



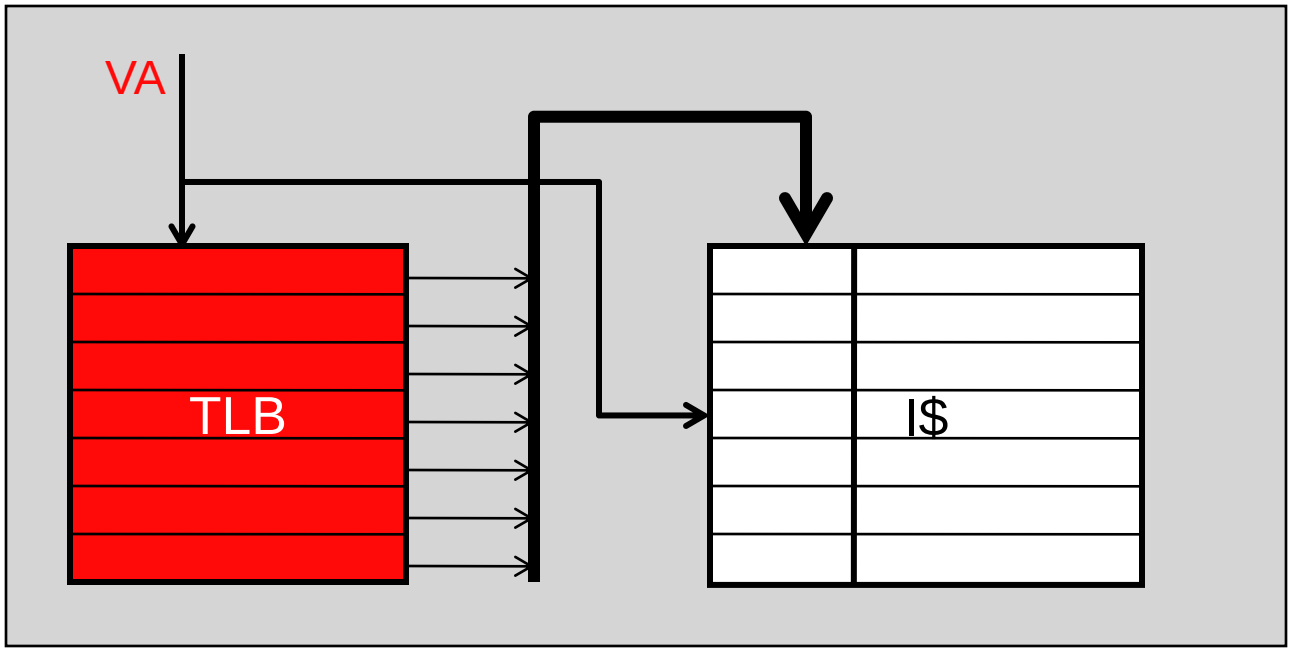
Backups

Slide History/Attribution Diagram:

Safe and Efficient Services

- Scenario: module (application) A wants service B provides
  - A doesn't "trust" B and vice versa (e.g., B is kernel)
  - How is service provided?
- Option I: conventional call in same address space
  - \+ Can easily pass data back and forth (pass pointers)
  - – Untrusted module can corrupt your data
- Option II: trap or cross address space call
  - – Copy data across address spaces: slow, hard if data uses pointers
  - \+ Data is not vulnerable
- Page-level protection helps somewhat, but…
  - Page-level protection can be too coarse grained
  - If modules share address space, both can change protections

Itanium Prevalidated tags



- I$ tag is bit vector, not address tag
  - match TLB location for hit
- TLB miss ➔ I$ miss
- TLB size ➔ tag size (32 entries/32 bits in Itanium 2)

Digital Rights Management

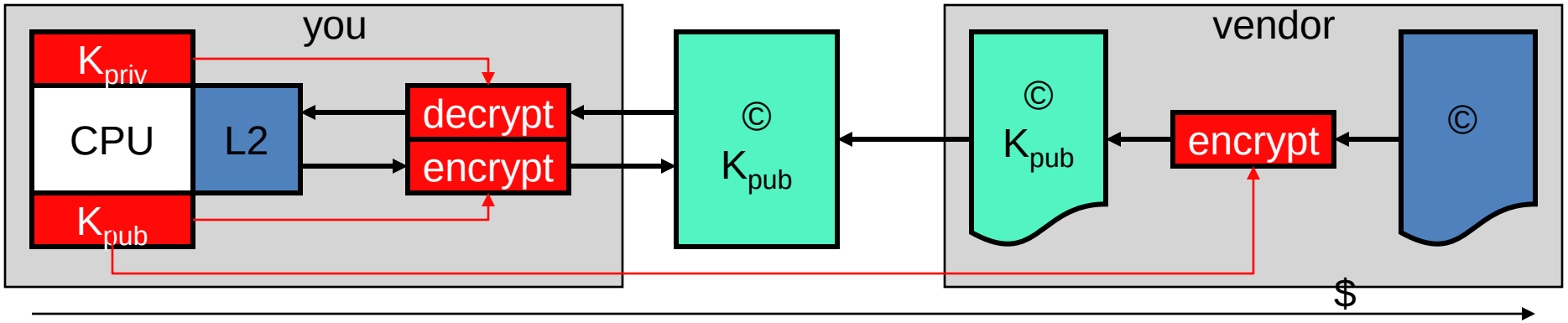
- **Digital rights management**
  - Question: how to enforce digital copyright?
    - Electronically, not legally
  - "Trying to make bits un-copiable is like trying to make water un-wet"

  - Suppose you have some piece of copyrighted material ©…
    - You can easily make a copy of ©
  - But, what if © is encrypted?
    - In order to use ©, you must also have the decryptor
    - Can hack decryptor to spit out unencrypted ©
    - Or hack OS to look at decryptor's physical memory

Aside: Public-Key Cryptography

- **Public-key cryptography**
  - Asymmetric: pair of keys
    - **K_{pub}**: used for encryption, published
    - **K_{priv}**: used for decryption, secret
    - acrypt(acrypt(M, K_{pub}), K_{priv}) = acrypt(acrypt(M, K_{priv}), K_{pub}) = M
    - Well-known example: RSA
- Two uses
  - **Encryption**
    - Someone sends you encrypted message M: C = acrypt(M, K_{pub})
    - You are the only one that can decrypt it
  - **Authentication/Digital Signature**
    - You send someone a chosen plaintext M
    - They "sign" it by sending back DS = acrypt(M, K_{priv})
    - If acrypt(DS, K_{pub}) = M, then they are who K_{pub} says they are

Research: XOM



- **eXecute Only Memory (XOM)**
  - Stanford research project [Lie+, ASPLOS'00]
  - Two registers: K_{priv}, K_{pub} different for every chip (Flash program)
    - Software can get at K_{pub}, but K_{priv} is hardware's secret
  - Hardware encryption/decryption engine on L2 fill/spill path
  - Vendor sells you acrypt(©, K_{pub})
    + Even if someone copies it, they won't have K_{priv} to decrypt it
    - Plaintext © only exists on-chip
    + Even OS can never see plaintext ©

XOM: Not Quite

- Performance consideration
  - Asymmetric en-/de-cryption is slow, **symmetric** (one key) faster
    - E.g., DES, AES (Rijndael)
    - – Problem: can't publish encryption key without also...
- XOM Take II
  - Vendor chooses random symmetric key K_{sym}
  - Sells you scrypt(©, K_{sym}) + acrypt(K_{sym}, K_{pub})
  - Two-stage decryption
    - Decrypt K_{sym} using K_{priv}: slow (but for one piece of data)
    - Decrypt © using K_{sym}: fast
  - Note: SSL does the same thing
    - Uses asymmetric cryptography to choose symmetric session key

Error Detection: Parity

- **Parity**: simplest scheme
  - $f(data_{N-1\ldots0}) = XOR(data_{N-1}, \ldots, data_1, data_0)$
  + Single-error detect: detects a single bit flip (common case)
    - Will miss two simultaneous bit flips…
    - But what are the odds of that happening?
  – Zero-error correct: no way to tell which bit flipped

Error Correction: Hamming Codes

- **Hamming Code**
  - H(A,B) = number of 1's in A^B (number of bits that differ)
    - Called "Hamming distance"
  - Use D data bits + C check bits construct a set of "codewords"
    - Check bits are parities on different subsets of data bits
  - $\forall$codewords A,B H(A,B) $\geq \alpha$
    - No combination of α–1 transforms one codeword into another
    - For simple parity: $\alpha = 2$
  - Errors of δ bits (or fewer) can be detected if $\alpha = \delta + 1$
  - Errors of β bits or fewer can be corrected if $\alpha = 2\beta + 1$
  - Errors of δ bits can be detected and errors of β bits can be corrected if $\alpha = \beta + \delta + 1$

SEC Hamming Code

- **SEC**: single-error correct
  - $C = \log_2 D + 1$
  - \+ Relative overhead decreases as D grows

- Example: $D = 4 \rightarrow C = 3$
  - $d_1\ d_2\ d_3\ d_4\ \mathbf{c_1\ c_2\ c_3} \rightarrow \mathbf{c_1\ c_2}\ d_1\ \mathbf{c_3}\ d_2\ d_3\ d_4$
  - $c_1 = d_1 \wedge d_2 \wedge d_4$, $c_2 = d_1 \wedge d_3 \wedge d_4$, $c_3 = d_2 \wedge d_3 \wedge d_4$
  - Syndrome: $c_i \wedge c'_i = 0$? no error : points to flipped-bit
- Working example
  - Original data = 0110 $\rightarrow c_1 = 1, c_2 = 1, c_3 = 0$
  - Flip d_2 = 0010 $\rightarrow c'_1 = 0, c'_2 = 1, c'_3 = 1$
    - Syndrome = 101 (binary 5) $\rightarrow$ 5th bit? D_2
  - Flip $c_2 \rightarrow c'_1 = 1, c'_2 = 0, c'_3 = 0$
    - Syndrome = 010 (binary 2) $\rightarrow$ 2nd bit? c_2

SECDED Hamming Code

- **SECDED**: single error correct, double error detect
  - $C = \log_2 D + 2$
  - Additional parity bit to detect additional error
- Example: $D = 4 \rightarrow C = 4$
  - $d_1\ d_2\ d_3\ d_4\ \mathbf{c_1\ c_2\ c_3} \rightarrow \mathbf{c_1\ c_2}\ d_1\ \mathbf{c_3}\ d_2\ d_3\ d_4\ \mathbf{c_4}$
  - $c_4 = c_1 \wedge c_2 \wedge d_1 \wedge c_3 \wedge d_2 \wedge d_3 \wedge d_4$
  - Syndrome == 0 and $c'_4 == c_4 \rightarrow$ no error
  - Syndrome != 0 and $c'_4 != c_4 \rightarrow$ 1-bit error
  - Syndrome != 0 and $c'_4 == c_4 \rightarrow$ 2-bit error
  - Syndrome == 0 and $c'_4 != c_4 \rightarrow c_4$ error
- Many machines today use 64-bit SECDED code
  - C = 8 (one additional byte, 12% overhead)
  - ChipKill - correct any aligned 4-bit error
    - If an entire DRAM chips dies, the system still works!