《计算机系统结构》课程直播

2020. 3.24

请将ZOOM名称改为“姓名”；

听不到声音请及时调试声音设备；签到将在课间休息进行

本次讲课内容

1. **虚拟存储器、快表**
   Virtual Memory, TLB
2. **如何用虚拟地址查找cache**
   Virtually indexed, physically tagged
3. **性能模型**
   Memory mountain

PART 01

虚拟存储器

Overview

上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

Virtual Memory： Motivation

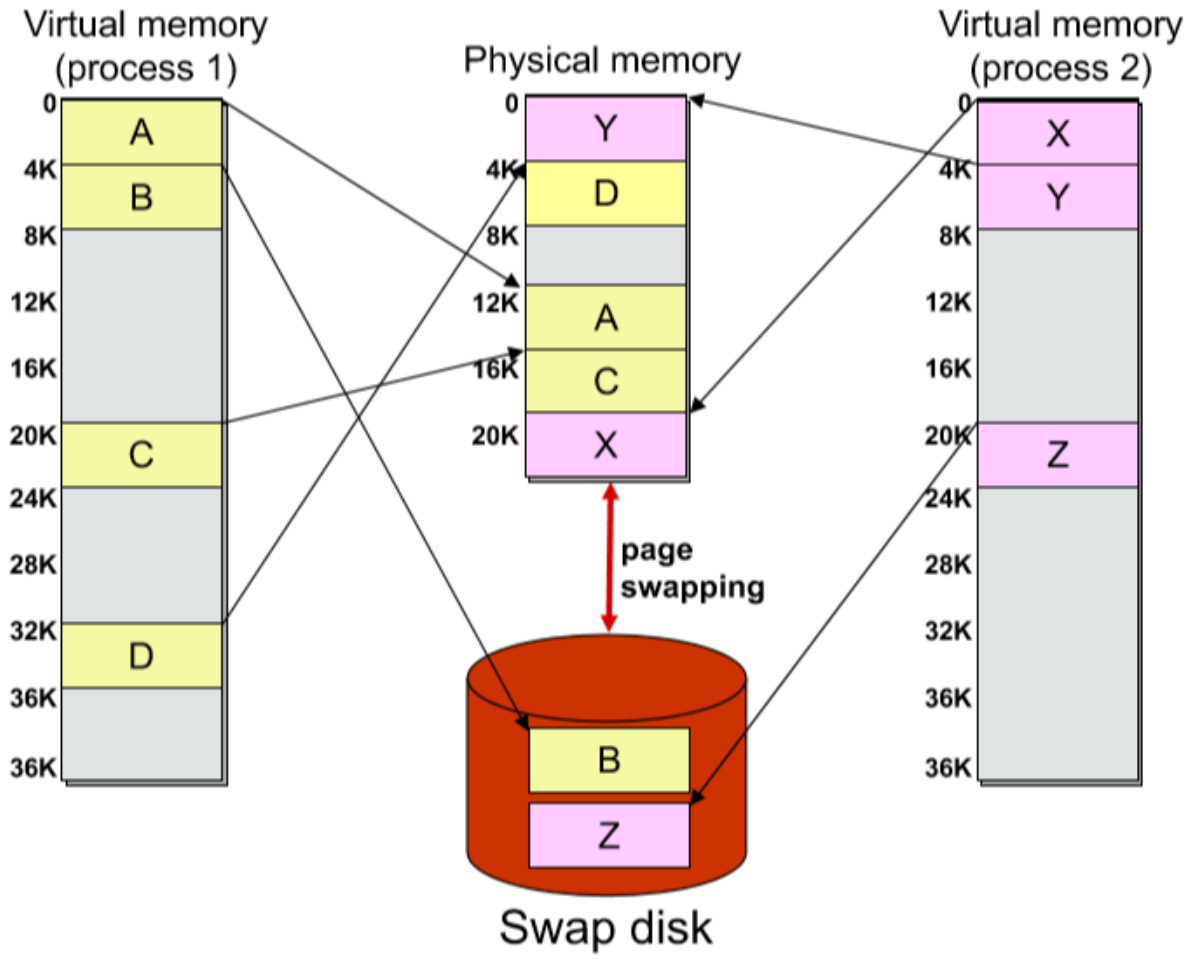
- **Each process** would like to see its **own, full, address space**
- **内存容量无法满足：** **c**learly impossible to provide **full** physical memory for all processes
- 程序一般只使用的部分代码和数据: Processes may **define a large address space but use only a small part** of it at any one time
- 内存保护:
  - **Processes would like their memory to be protected** from access and modification by other processes
  - The **operating system** needs to be protected from applications

Virtual Memory： Basic idea

- **一个程序的地址空间可以分成多个大小相同的页（pages：例如4KB）**
  - Each process has its own Virtual Address Space, divided into fixed-sized pages
- **把程序当前使用的部分代码和数据保留在内存中，把其它部分存在磁盘上，需要时在内存和磁盘之间动态交换**
  - Virtual pages not recently used may be stored on disk
  - Extends the memory hierarchy out to the swap partition of a disk
  - Virtual memory: pages
  - Physical memory: frames
    - Virtual pages that are in use get mapped to pages of physical memory (called page frames)

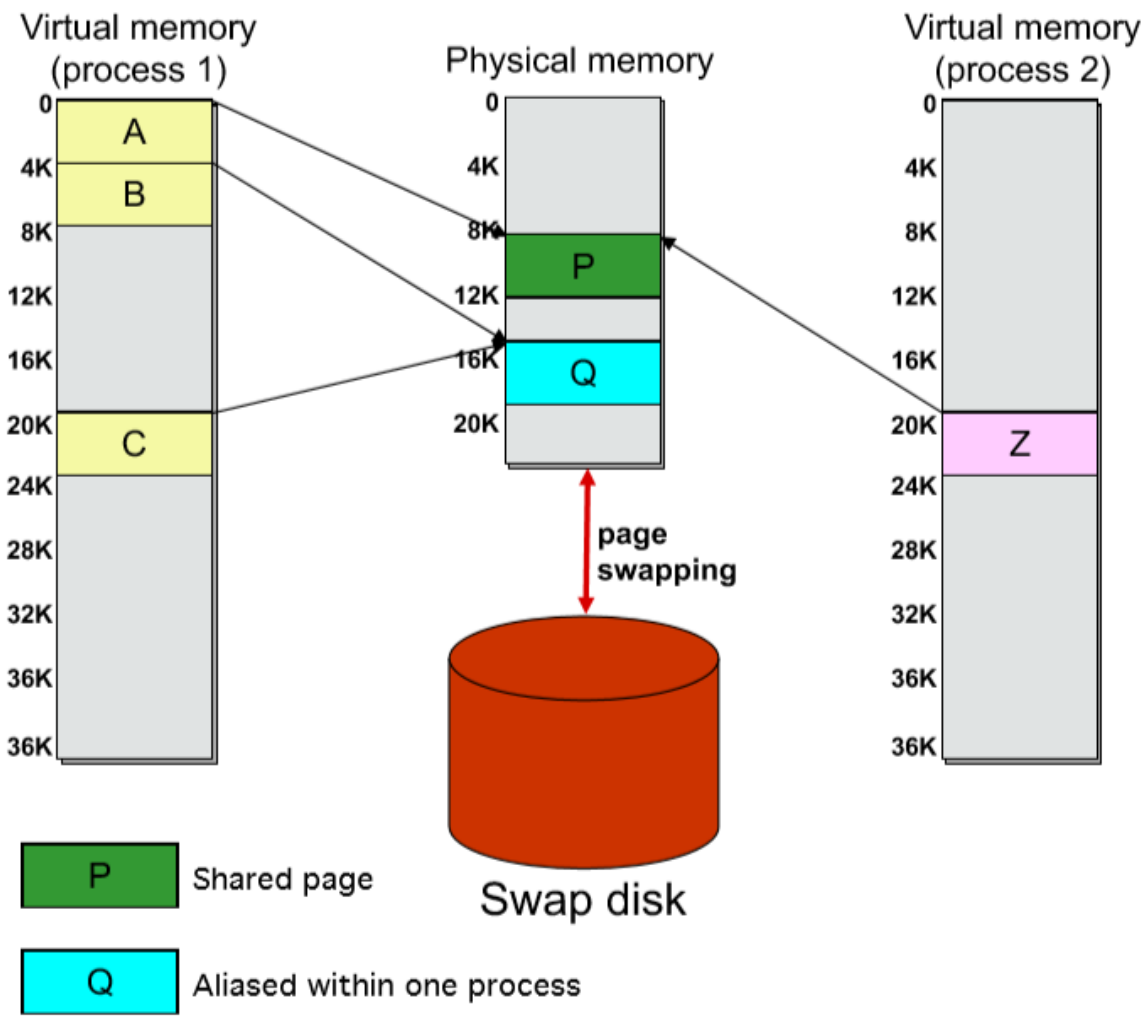
Virtual and Physical Memory

- Example 4K page size
- Process 1 has pages A, B, C and D
- Page B is held on disk
- Process 2 has pages X, Y, Z
- Page Z is held on disk
- Process 1 cannot access pages X, Y, Z
- Process 2 cannot access page A, B, C, D
- O/S can access any page (full privileges)



Sharing memory using Virtual Aliases

- Process 1 and Process 2 want to share a page of memory
- Process 1 maps virtual page A to physical page P
- Process 2 maps virtual page Z to physical page P
- Permissions can vary between the sharing processors.
- Note: Process 1 can also map the same physical page at multiple virtual addresses !!



Typical Virtual Memory Parameters

parameter	L1 cache	memory
Size	4KB-64KB	128MB-1TB
block/page	16-128 bytes	4KB-4GB
hit time	1-3 cycles	100-300 cycles
miss penalty	8-300 cycles	1M-10M cycles
miss rate	0.1-10%	0.00001-0.001%

H&P 5/e
Fig. B.20

- Modern OS's support several page sizes for flexibility.
- On Linux:
  - Normal pages: 4KB
  - Huge pages: 2MB or 1GB
- Virtual Memory miss is called a page fault

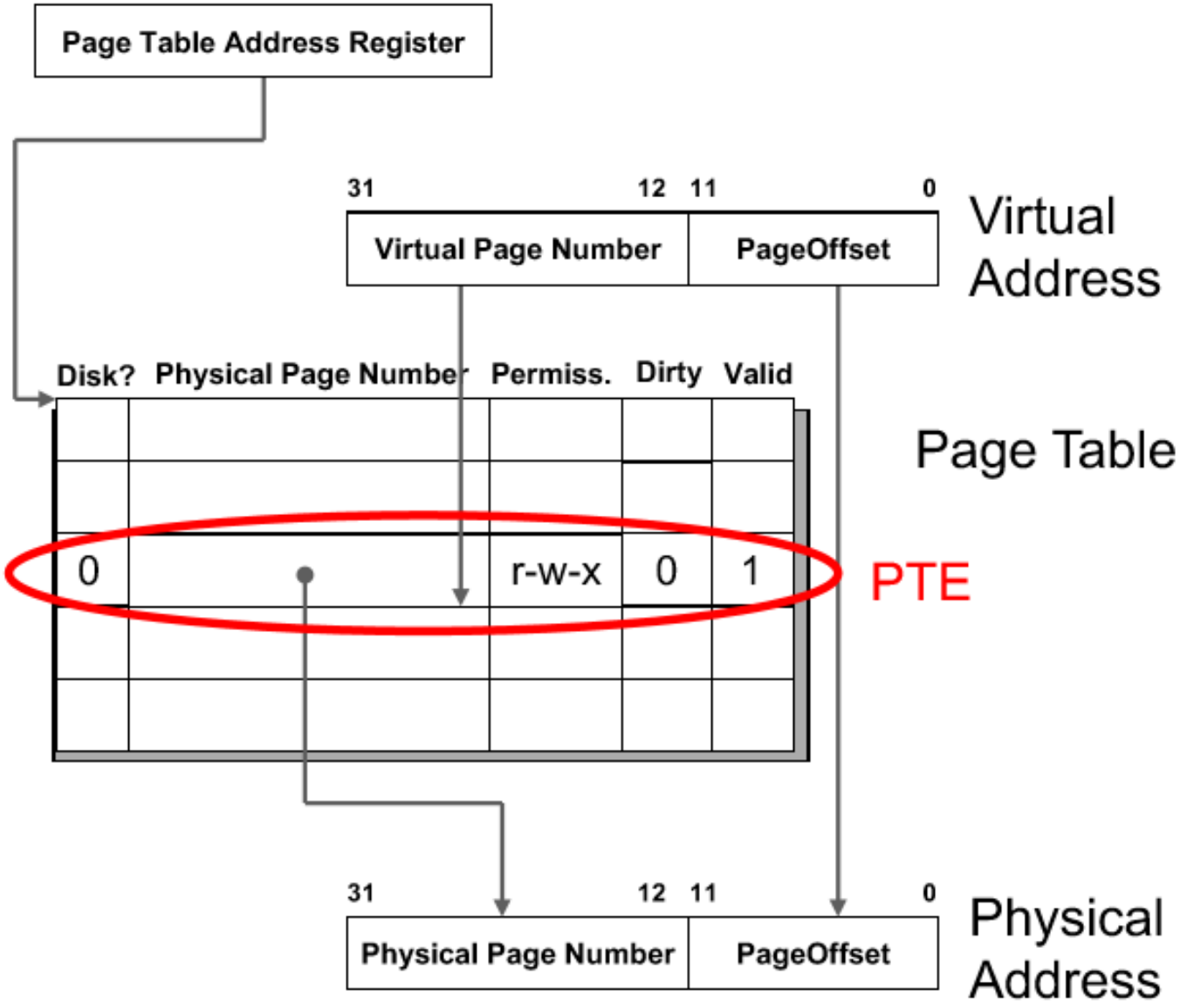
Memory Hierarchy Issues

- **Block size:** E.g., 64B for cache lines, **4KB for memory pages**
- **Block placement**: Where can a block be placed?
  - E.g., direct mapped for cache block, **fully associative for memory pages**
- **Block identification**: How can a block be identified?
  - E.g., hardware tag matching (e.g., cache), **OS page table**
- **Block replacement**: Which block should be replaced?
  - E.g., Random, **Least recently used (LRU),** Not recently used (NRU)： CLOCK置换算法
- **Write strategy:** What happens on a write?
  - E.g., write-through, **write-back,** write-allocate
- **Inclusivity**: whether next lower level contains all the data found in the current level
  - Inclusive, exclusive

Page Tables

Page Table Entry (PTE):

- Track access permissions for each page
  - Read, Write, Execute Bit
- Indicates if page is on disk, in which case Physical Page Number indicates location within swap file
- "Dirty" bit indicates if there were any writes to the page
- 4B per PTE in this example



Making Page Tables space-efficient

- The number of entries in the table is the number of virtual pages
- many! e.g., 4KB pages
- 2^{20}=1M entries for a 32b address space ,need 4MB/process
- 2^{52}entries for a 64b address space, need petabytes per process!

Solution:

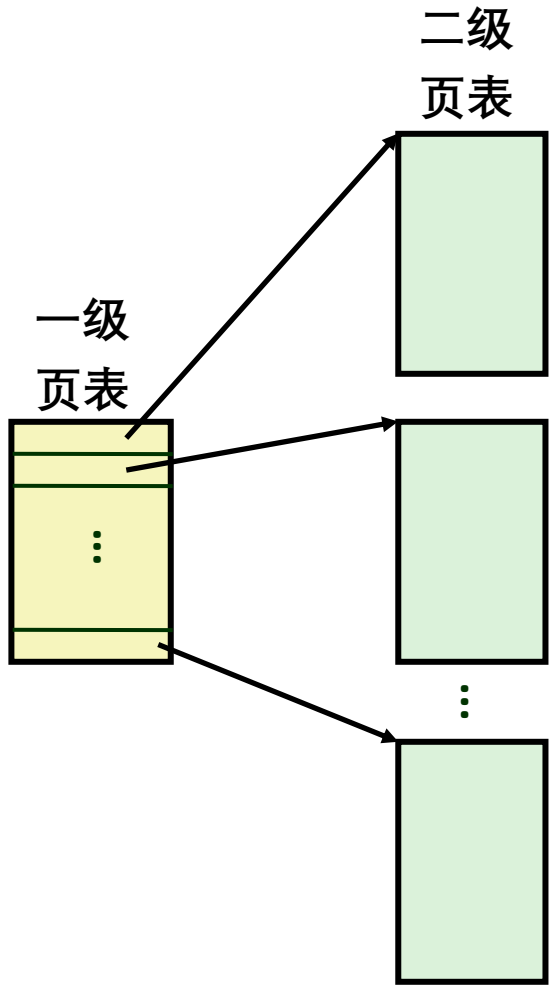
- hash virtual addresses to avoid maintaining a map from each virtual page (many) to physical frame (few).
- Resulting structure is called the inverted page table

Other (complementary) solutions:

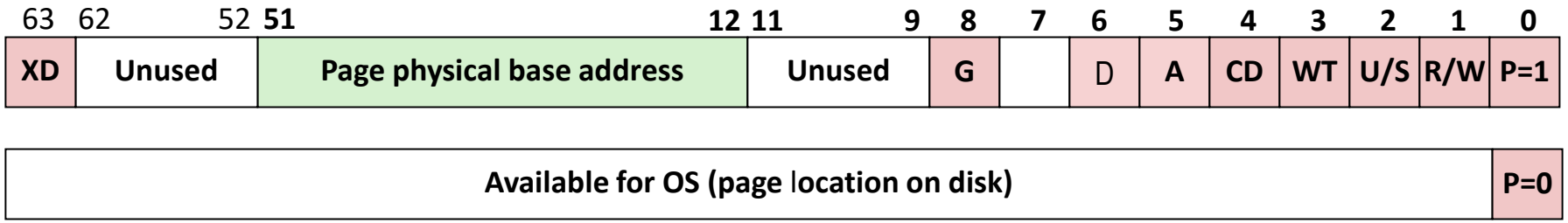
- Store PTs in the virtual memory of the OS, and swap out recently unused portions
- Use large pages

Making Page Tables space-efficient

- 解决方法: 多级页表
- 举例： 二级页表
  - 第一级页表:
    - 每一项指向一个页表,
    - 记录页表所在位置
  - 第二级页表:
    - 每一项指向一页
    - 记录页所在位置



Core i7 Level 4 Page Table Entries



Each entry references a 4K child page

P: **Child page is present in memory (1) or not (0)**

R/W: **Read-only or read-write access permission for child page**

U/S: **User or supervisor mode access**

WT: **Write-through or write-back cache policy for this page**

CD: **Cache disabled (1) or enabled (0)**

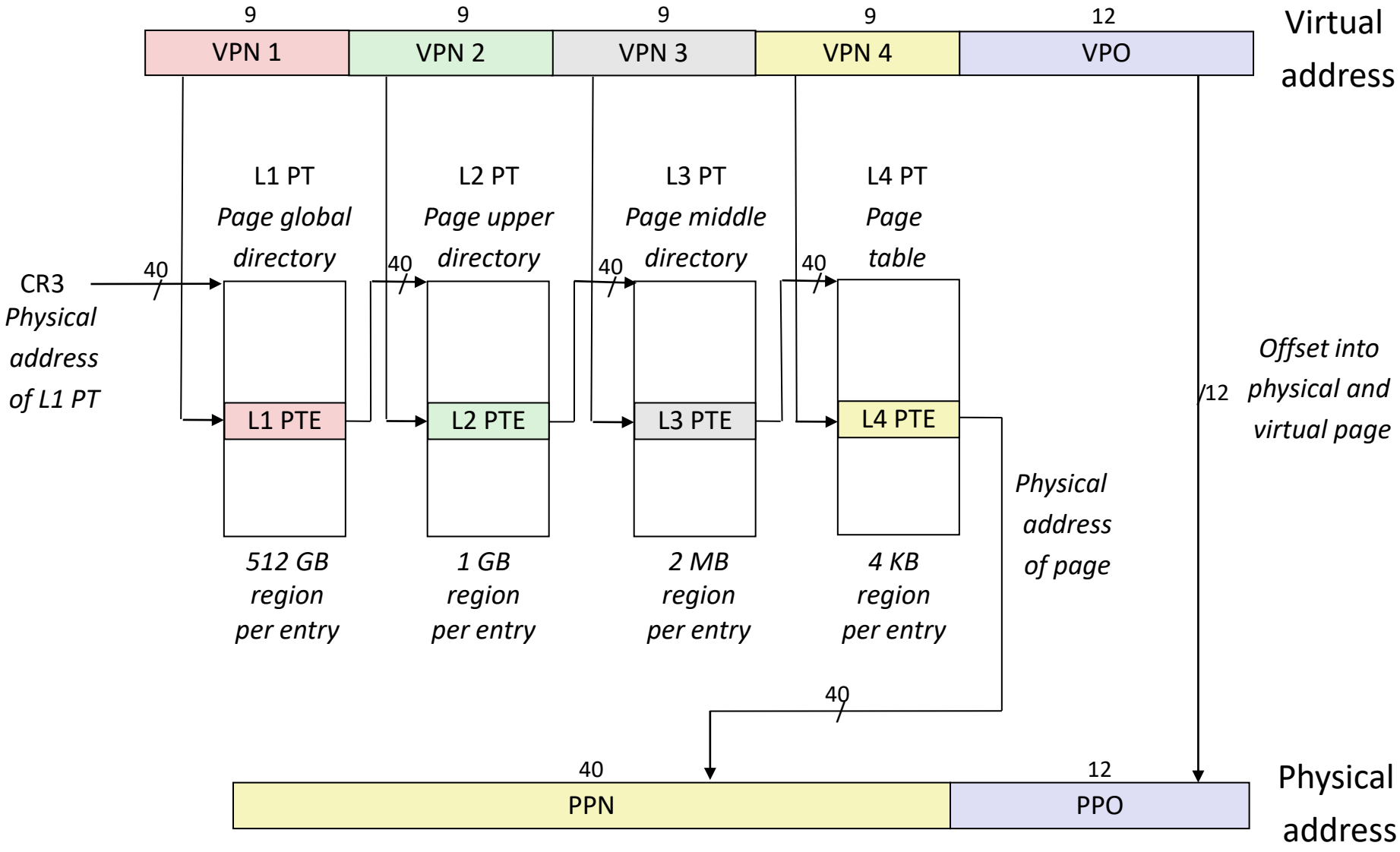
A: **Reference bit (set by MMU on reads and writes, cleared by software)**

D: **Dirty bit (set by MMU on writes, cleared by software)**

G: **Global page (don't evict from TLB on task switch)**

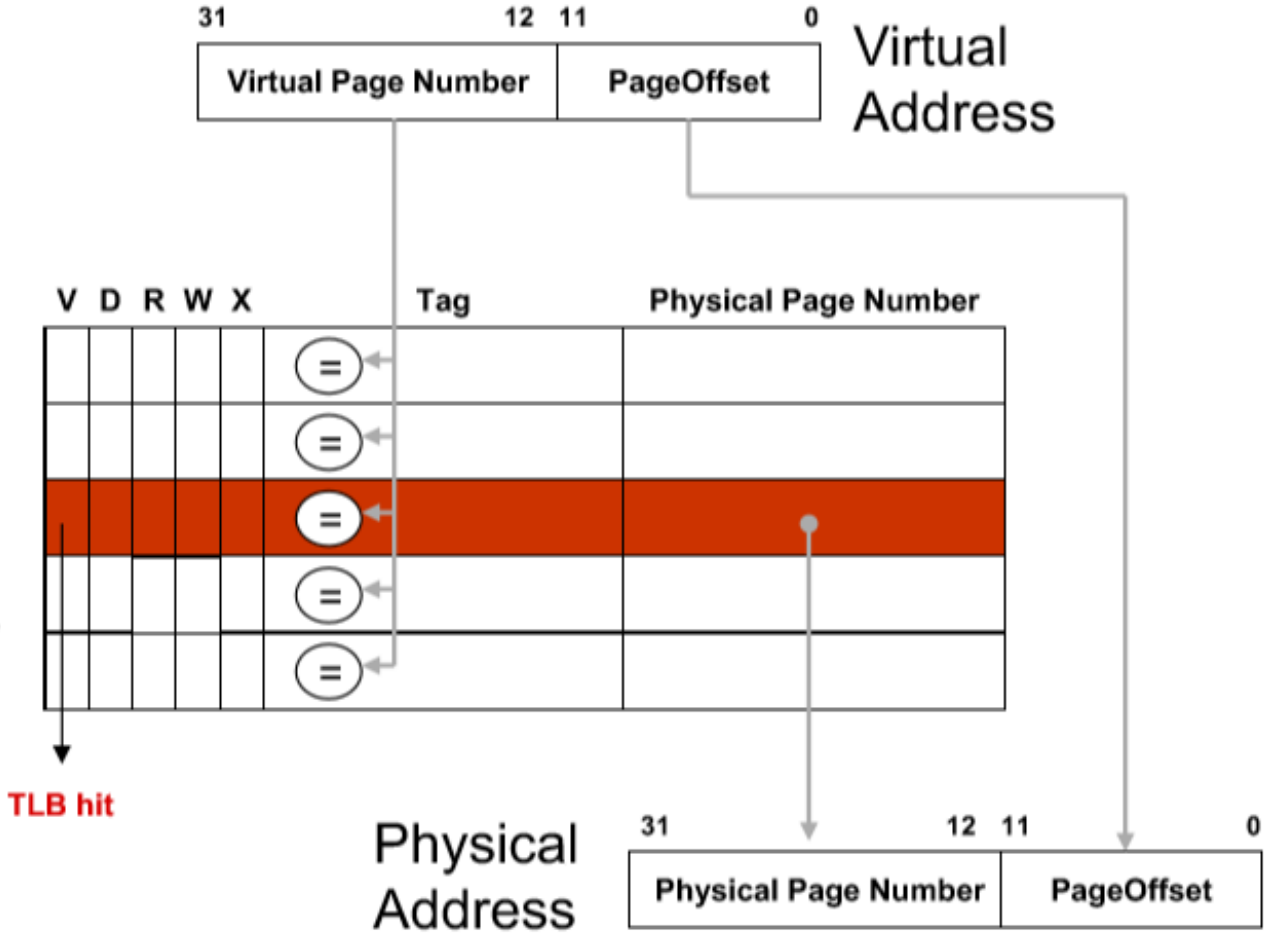
Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

Core i7 Page Table Translation



Fast address translation: TLB

- Typically a small, fully-associative cache of Page Table Entries (PTE)
- Tag given by VPN for that PTE
- PPN taken from PTE
- Valid bit required
- D bit (dirty) indicates whether page has been modified
- R, W, X bits indicate Read, Write and Execute permission
- Permissions are checked on every memory access
- Physical address formed from PPN and Page Offset
- TLB Exceptions:
  - TLB miss (no matching entry)
  - Privilege violation
- Often separate TLBs for Instruction and Data references



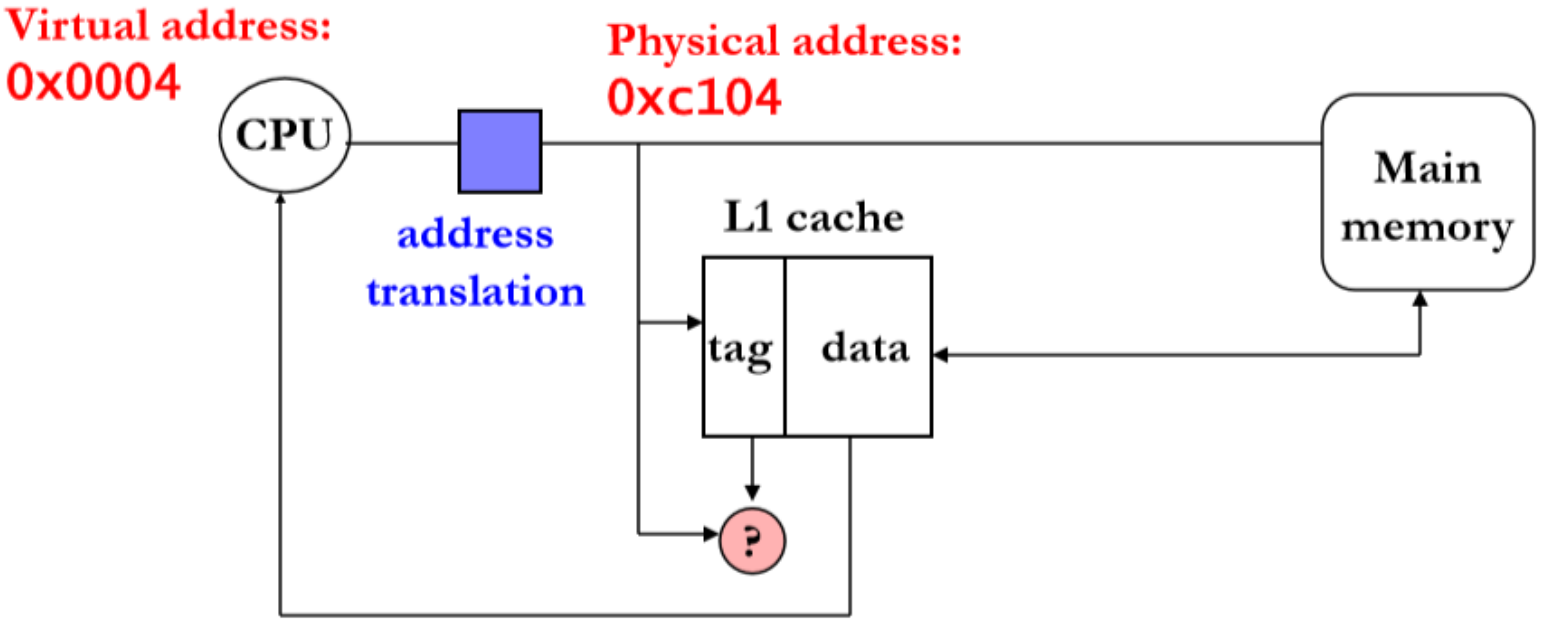
PART 02

How to address a cache in a virtual-memory system

Address a cache in a virtual-memory system

Option 1: physically-addressed caches → perform address translation before cache access

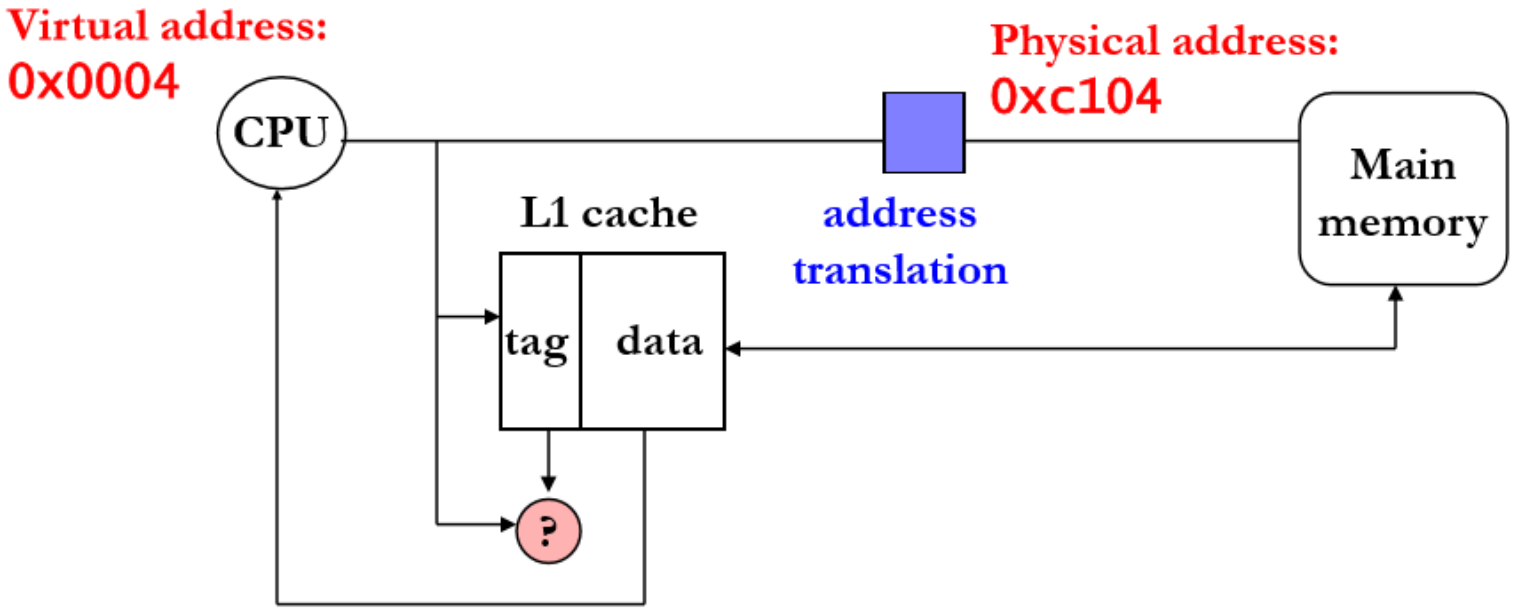
- Hit time is increased to accommodate translation ☹



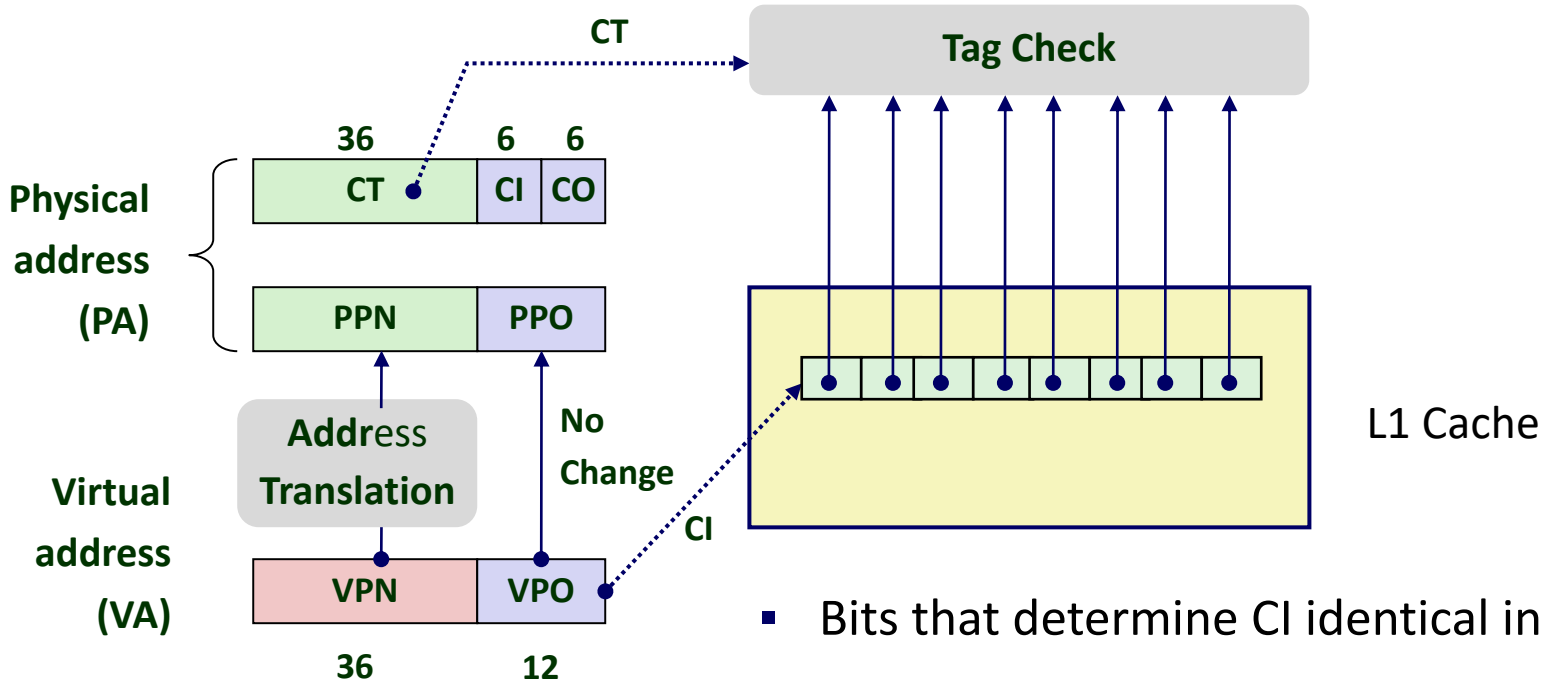
Address a cache in a virtual-memory system

Option 2: virtually-addressed caches → perform address translation after cache access if miss

- Hit time does not include translation ☺
- Aliases ☹



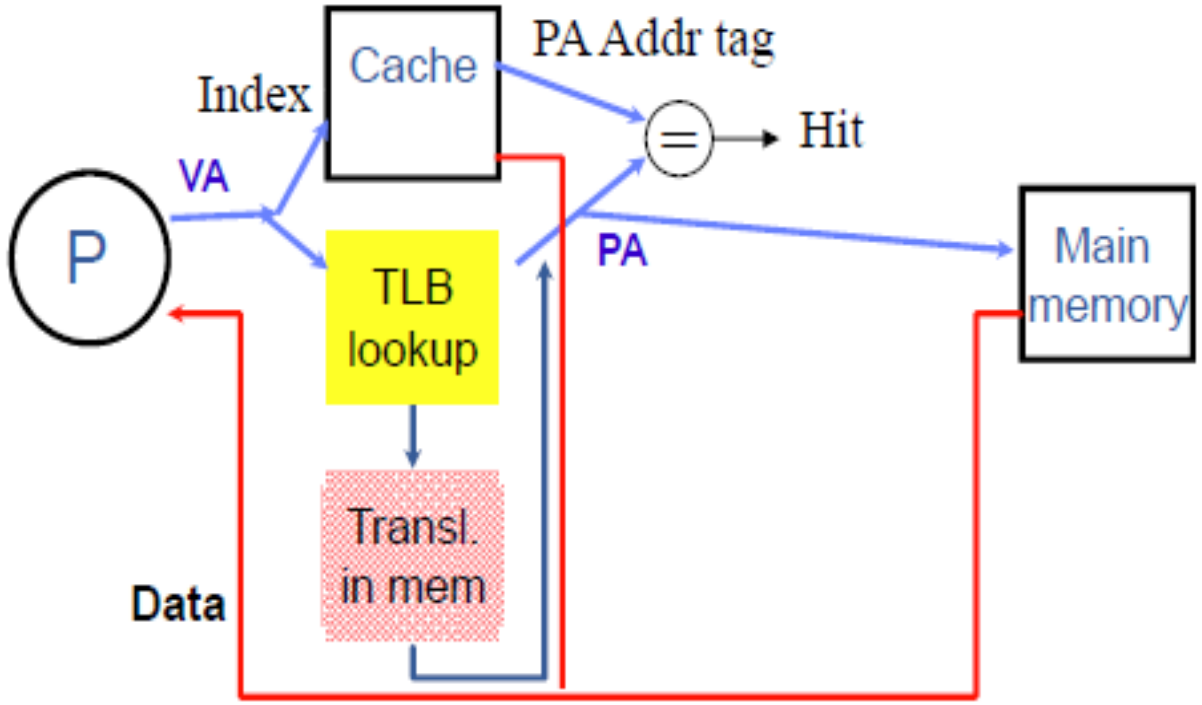
Cute Trick for Speeding Up L1 Access



- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available next
- “Virtually indexed, physically tagged”
- Cache carefully sized to make this possible

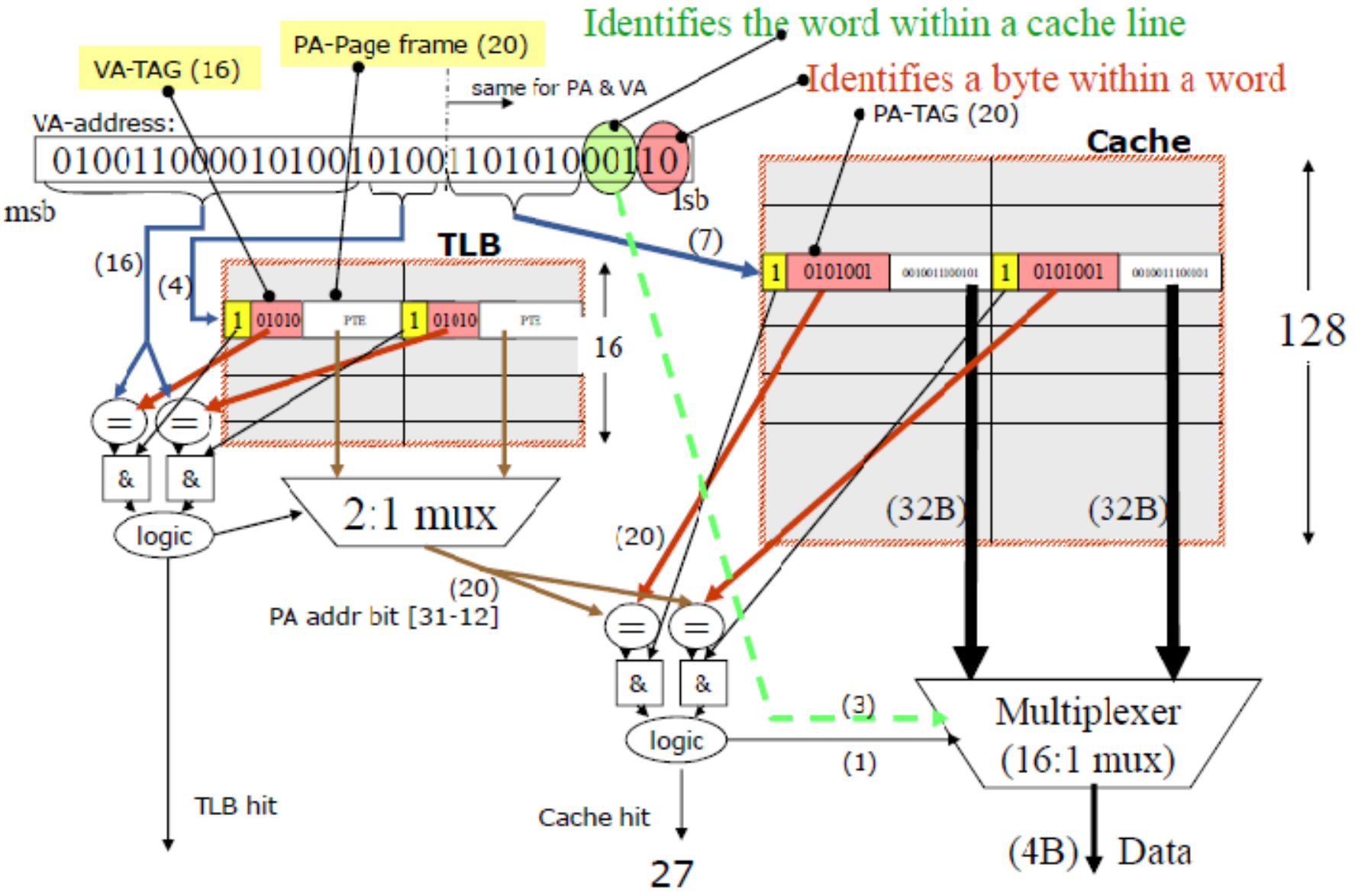
Virtually Indexed Physically Tagged: VIPT

- Have to guarantee that all aliases have the same index
- L1_cache_size < (page-size * associativity)



Putting it all together: VIPT

Cache: 8kB, 2-way, CL=32B, word=4B, page =4kB， TLB: 32 entries, 2-way



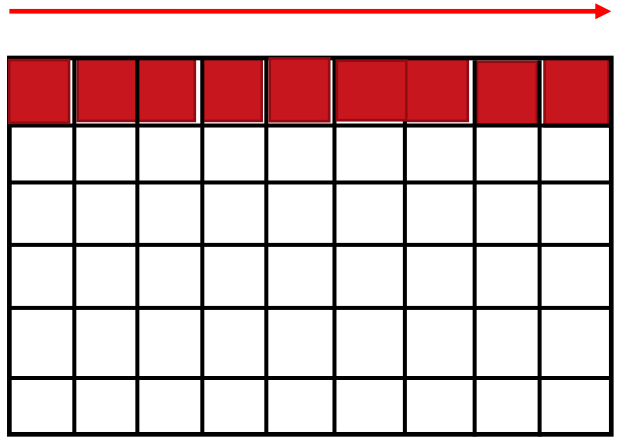
Summary: How to address a cache?

- PI-PT : Physically indexed, physically tagged
  - Translation first; then cache access
  - Con: Translation occurs in sequence with L1-$ access → high latency
- VI-VT : Virtually indexed, virtually tagged
  - L1-$ indexed with virtual address, tag contains virtual address
  - Con: Cannot distinguish synonyms/homonyms in cache
  - Pro: Only perform TLB lookup on L1-$ miss
- VI-PT : Virtually indexed, physically tagged
  - L1-$ indexed with virtual address, or often just the un-translated bits
  - Translation must take place before tag can be checked
  - Con: Translation must take place on every L1-$ access
  - Pro: No synonyms/homonyms in the cache
- PI-VT : Physically indexed, virtually tagged
  - Not interesting

PART 03

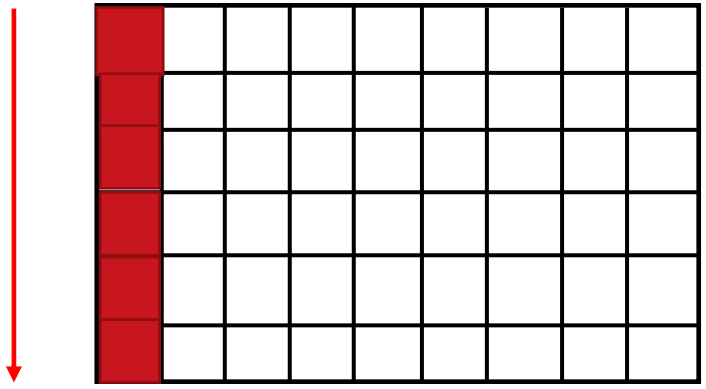
cache friendly code

什么是cache 友好的代码?



- 对同一行的元素依次访问:
  - `for (i = 0; i < N; i++)`

    `sum += a[0][i];`



- 对同一列的元素依次访问:
  - `for (i = 0; i < n; i++)`

    `sum += a[i][0];`

Observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor “cache friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)

Case Study： Memory Mountain MicroBenchmark

```
for (times = 0; times < Max; times++) /* many times*/

  for (i=0; i < ArraySize; i = i + Stride)
    dummy = A[i]; /* touch an item in the array */
```

Array Size = 16, Stride=4

Array Size = 16, Stride=8…

Array Size = 32, Stride=4…

Array Size = 32, Stride=8…

Memory Mountain Test Function

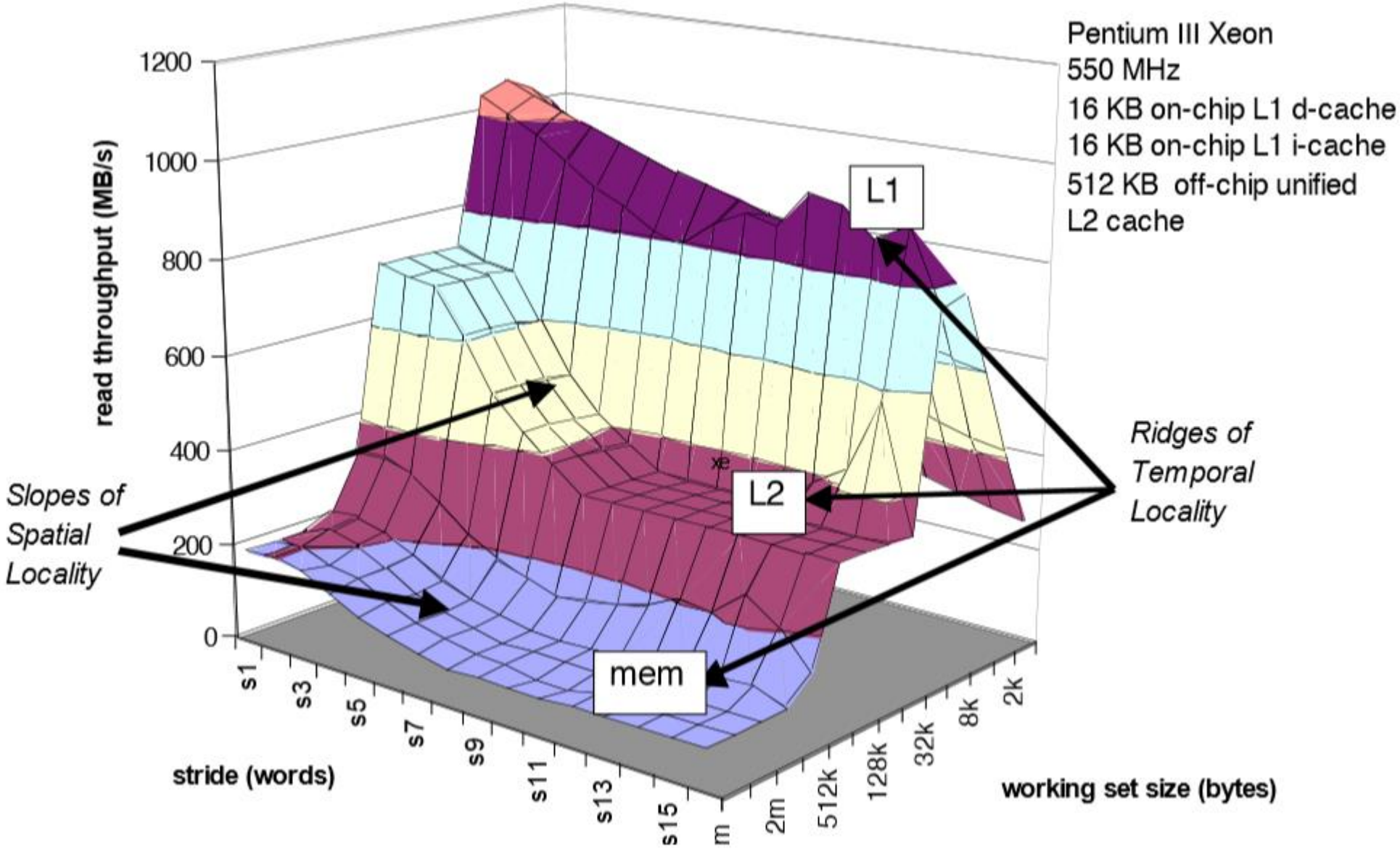
```
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}
```

Call `test()` with many combinations of `elems` and `stride`.

For each elems and stride:
1. Call test() once to warm up the caches.
2. Call test() again and measure the read throughput(MB/s)

The Memory Mountain， CSAPP 2e



```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *      array “data” with stride of "stride", using
 *      using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

Memory Mountain Test Function CSAPP 3e

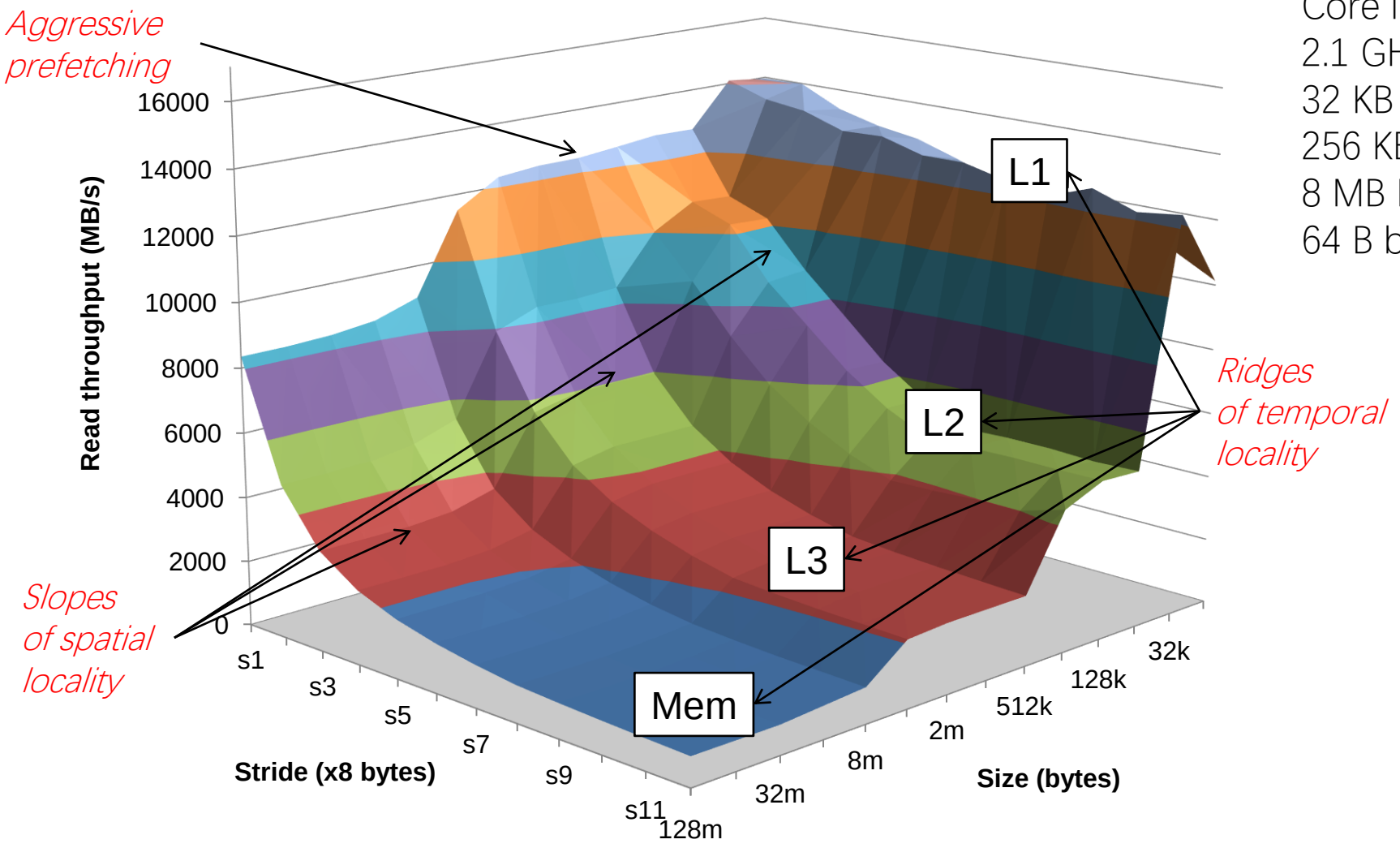
Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

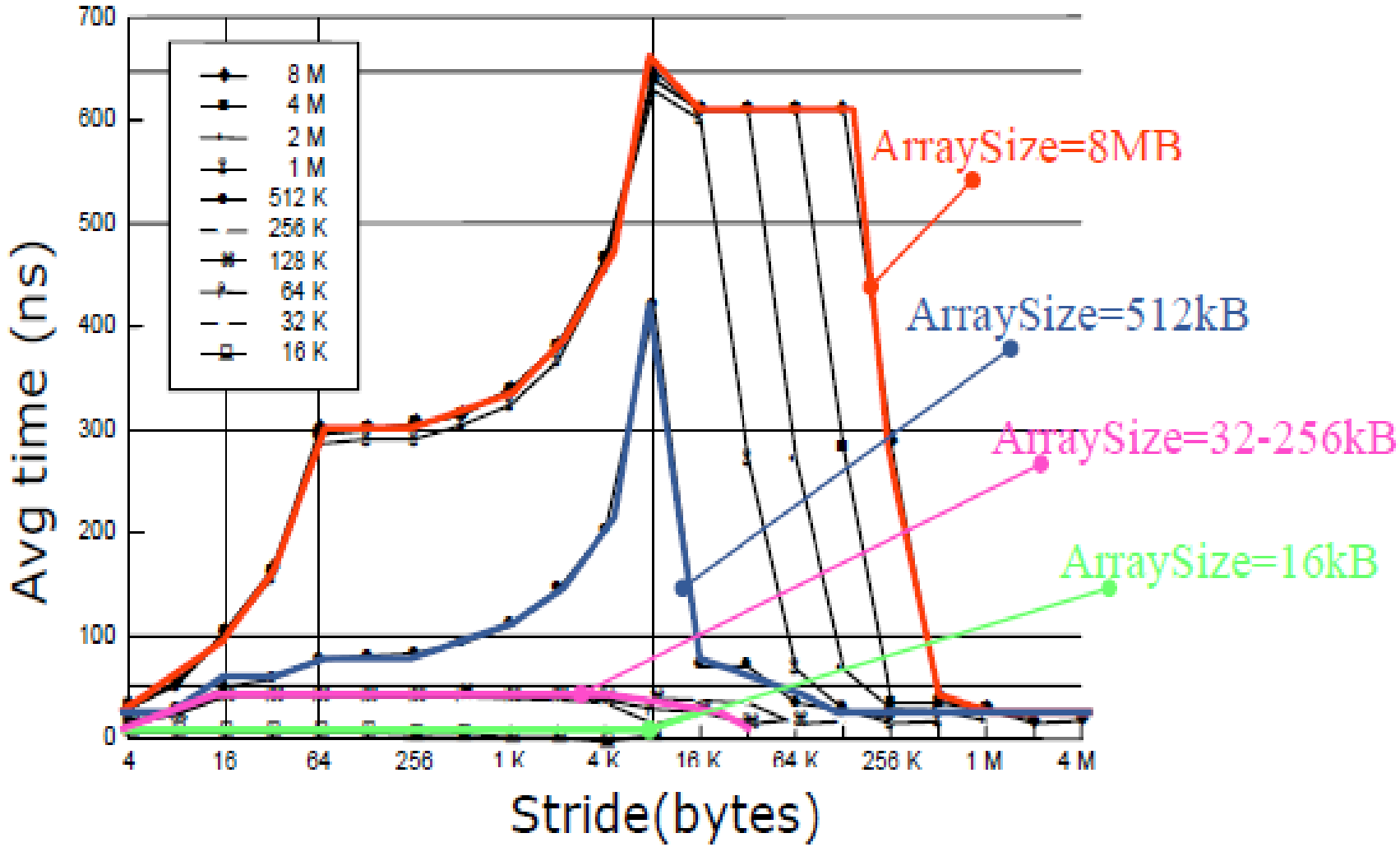
2. Call `test()` again and measure the read throughput(MB/s)

The Memory Mountain

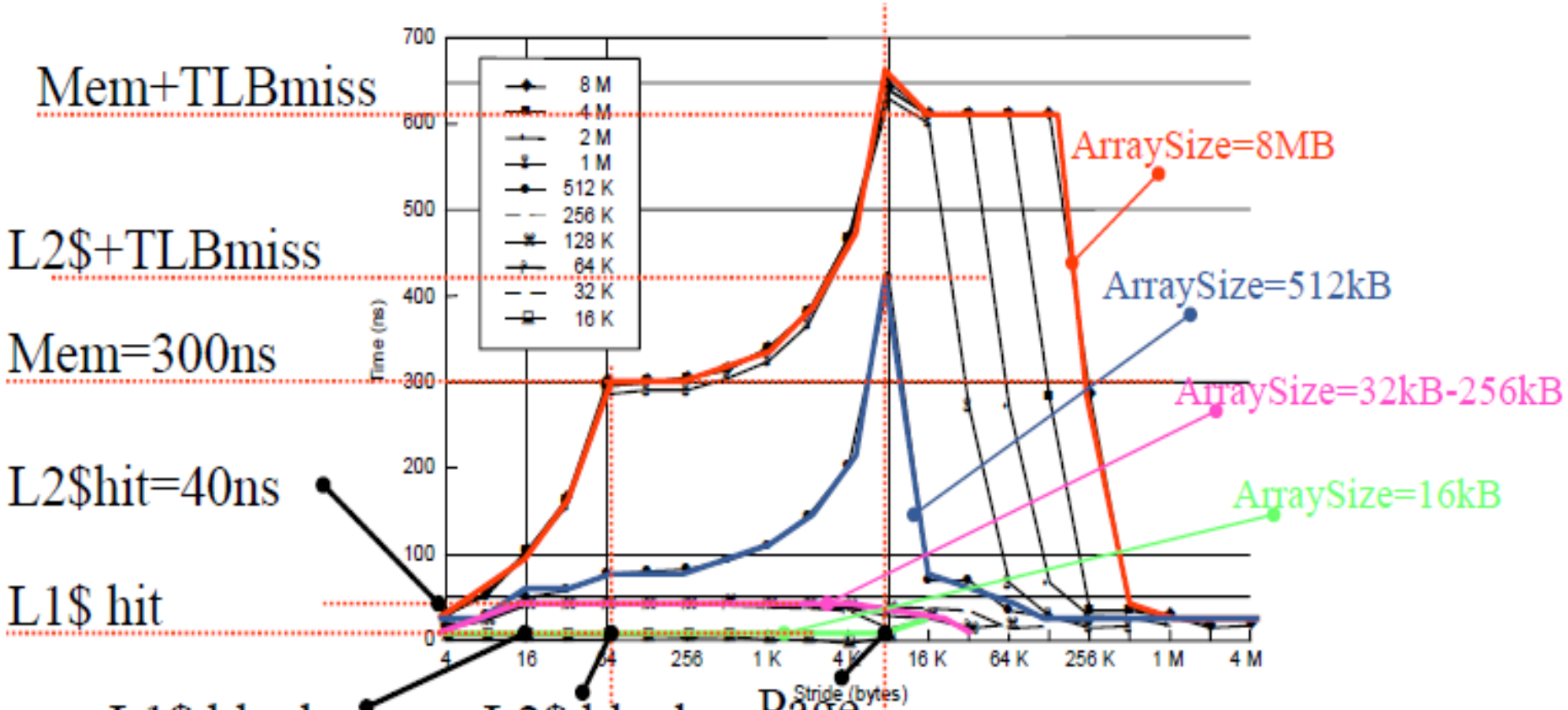


Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Micro Benchmark Signature



Putting It All Together: 分析各参数的值



下一节

- 周四8：00
- 指令系统
- 请做好准备

再见