存储层次

100076202： 计算机系统导论

任课教师：

宿红毅　　张艳　　黎有琦　　颜珂

原作者：

Randal E. **Bryant and** David R. O'Hallaron

提纲

- **Section 1**
  - 内存抽象 The memory abstraction
  - 随机访问存储器：主存构建块 RAM : main memory building block
  - 引用的局部性 Locality of reference
  - 存储器层次结构 The memory hierarchy
  - 存储技术和趋势 Storage technologies and trends
- **Section 2**
  - Cache存储器结构和操作 Cache memory organization and operation
  - Cache对性能的影响 Performance impact of caches
    - 存储器性能山丘 The memory mountain
    - 循环变换提升空间局部性 Rearranging loops to improve spatial locality
    - 使用blocking提升时间局部性 Using blocking to improve temporal locality

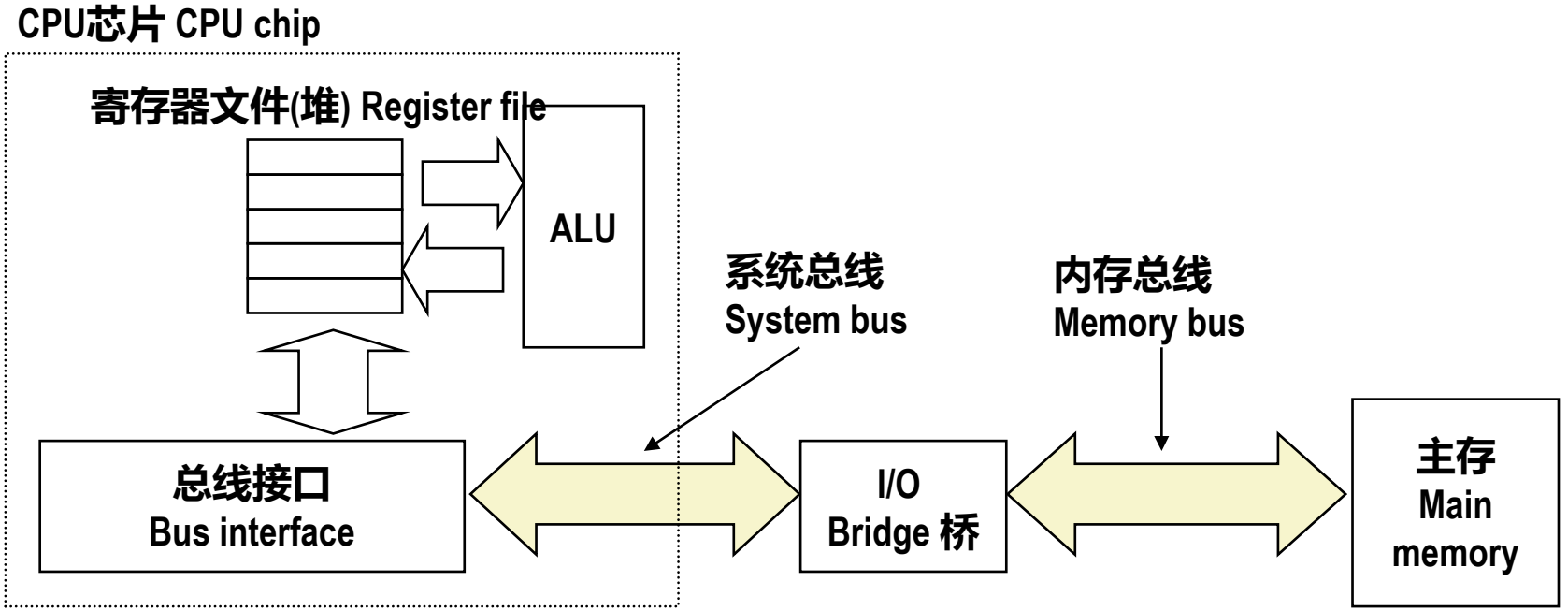
写和读内存 Writing & Reading Memory

- **写 Write**
  - 从CPU向内存传送数据 Transfer data from CPU to memory
    `movq %rax, 8(%rsp)`
  - “存储”操作 “Store” operation

- **读 Read**
  - 从内存向CPU传送数据 Transfer data from memory to CPU
    `movq 8(%rsp), %rax`
  - “装载”操作 “Load” operation

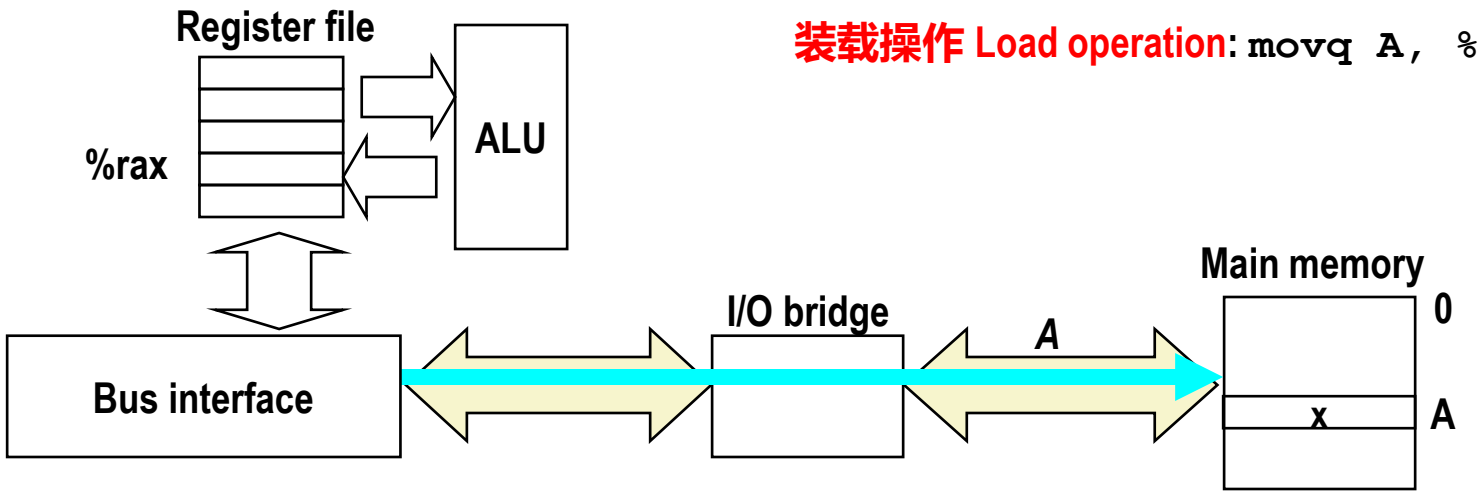
传统（2008之前）CPU和内存之间的互连总线结构 Traditional (pre-2008) Bus Structure Connecting CPU and Memory

- 总线是一组并行的用于传输地址、数据和控制信号的导线 A bus is a collection of parallel wires that carry address, data, and control signals.
- 总线通常是多个设备共享的 Buses are typically shared by multiple devices.



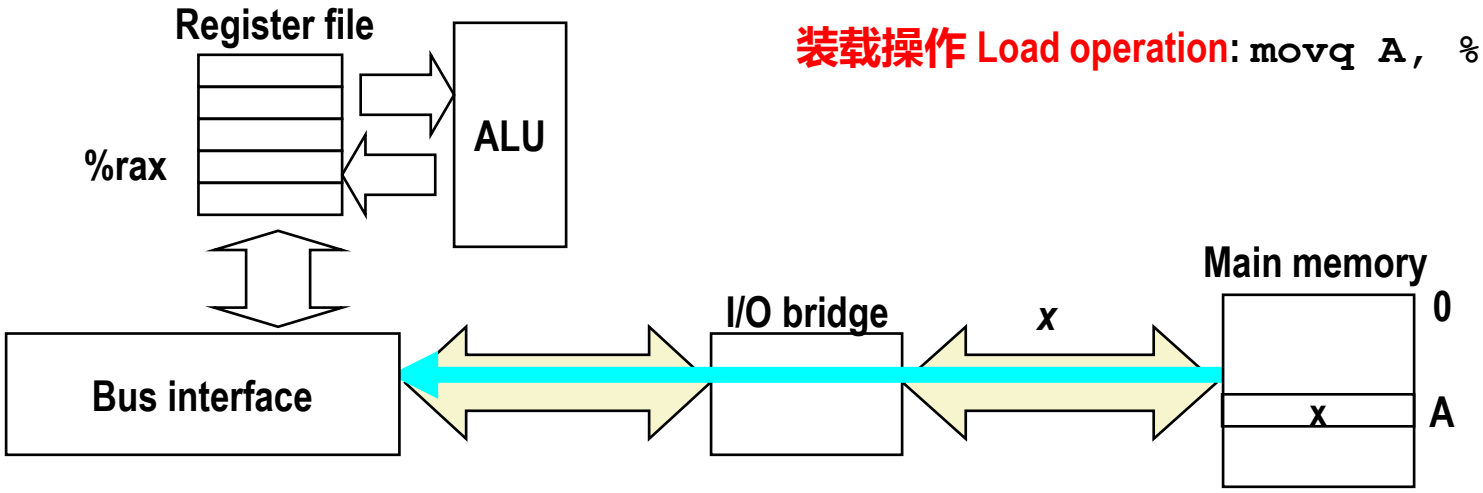
内存读事务（1）
Memory Read Transaction (1)

- CPU将地址A放到内存总线上 CPU places address A on the memory bus.



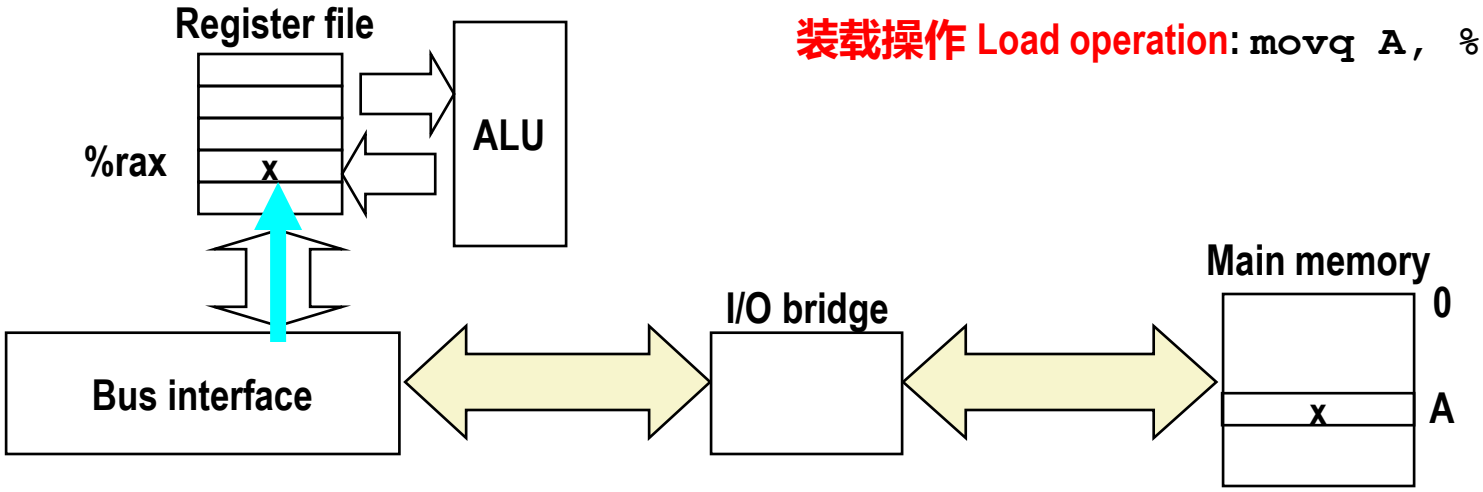
内存读事务（2）
Memory Read Transaction (2)

- 主存从内存总线获得地址A，读取对应的字x，并将其放置到总线上 Main memory reads A from the memory bus, retrieves word x, and places it on the bus.



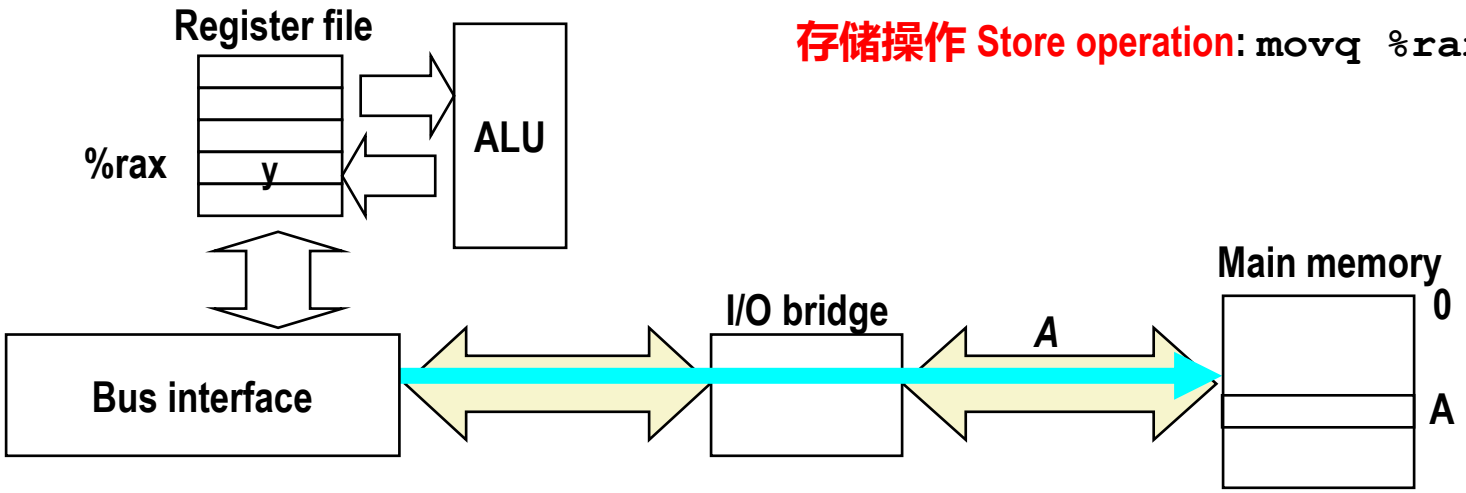
内存读事务（3）
Memory Read Transaction (3)

- CPU从总线读取x并拷贝到寄存器%rax中 CPU read word x from the bus and copies it into register %rax.



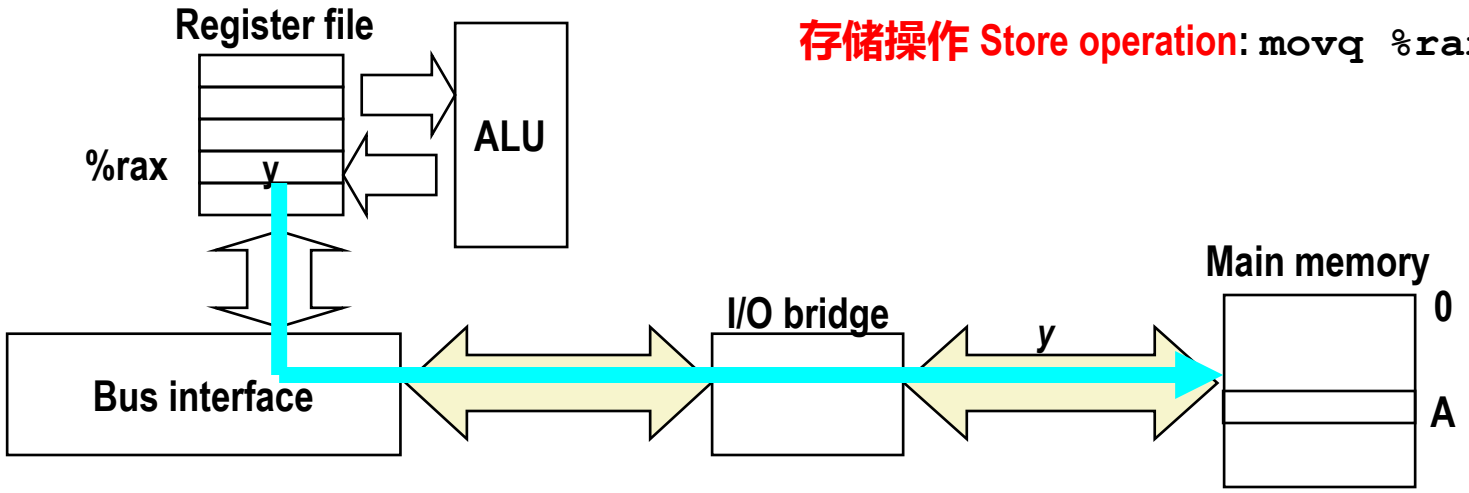
内存写事务（1）
Memory Write Transaction (1)

- CPU将地址A放到总线上，主存读取地址并等待数据字到来 CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



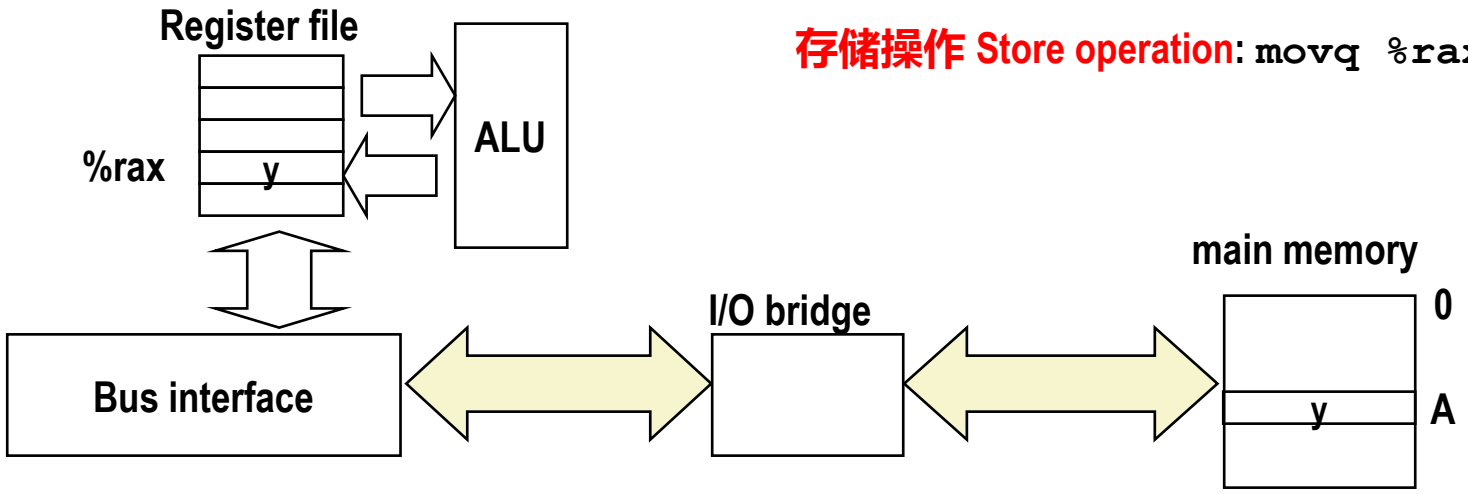
内存写事务（2）
Memory Write Transaction (2)

- CPU将数据字y放到总线上 CPU places data word y on the bus.



内存写事务（3）
Memory Write Transaction (3)

- 主存从总线读取数据字y并将其写入地址A Main memory reads data word y from the bus and stores it at address A.



提纲

- 内存抽象 The memory abstraction
- **随机访问存储器：主存构建块 RAM : main memory building block**
- 引用的局部性 Locality of reference
- 存储器层次结构 The memory hierarchy
- 存储技术和趋势 Storage technologies and trends

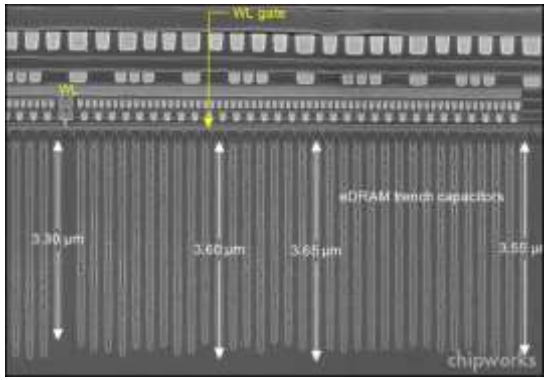
随机访问内存(RAM)
Random-Access Memory (RAM)

- **主要特点 Key features**
  - 传统上封装为一个芯片 RAM is traditionally packaged as a chip.
    - 或者嵌入到处理器芯片中 or embedded as part of processor chip
  - 基本存储单元正常是一个cell（每个cell是一个bit）Basic storage unit is normally a cell (one bit per cell).
  - 多个RAM芯片构成一个内存 Multiple RAM chips form a memory.

- **RAM有两种类型 RAM comes in two varieties:**
  - SRAM(静态RAM) SRAM (Static RAM)
  - DRAM(动态RAM) DRAM (Dynamic RAM)

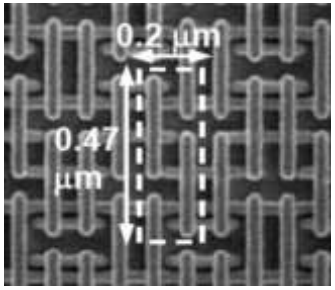
RAM技术 RAM Technologies

- DRAM



- 每个比特需要一个晶体管+一个电容 1 Transistor + 1 capacitor / bit
  - 电容器垂直定向 Capacitor oriented vertically
- 必须周期性刷新状态 Must refresh state periodically

- SRAM



- 每个比特需要六个晶体管 6 transistors / bit
- 永久保持状态 Holds state indefinitely

SRAM vs DRAM Summary 小结

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	6 or 8	1x	No	Maybe	1000x	Cache memories
DRAM	1	10x	Yes	Yes	1x	Main memories, frame buffers

EDC：差错检测和纠正　EDC: Error detection and correction

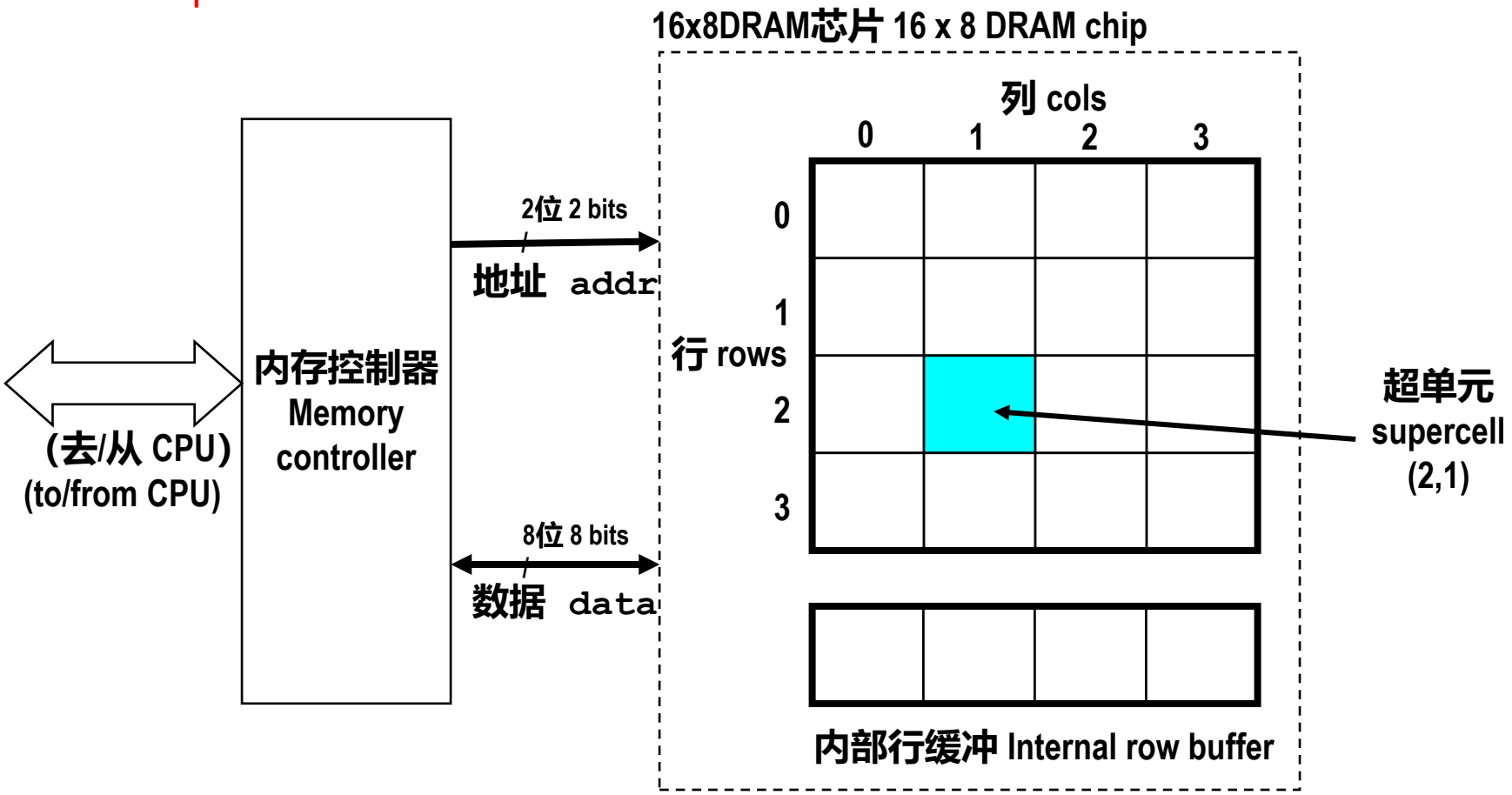
- **趋势 Trends**
  - SRAM随着半导体技术进展 SRAM scales with semiconductor technology
    - 达到其极限 Reaching its limits
  - DRAM进展受最小电容需求限制 DRAM scaling limited by need for minimum capacitance
    - 纵横比限制了电容器的深度 Aspect ratio limits how deep can make capacitor
    - 也达到了其极限 Also reaching its limits

增强的DRAM
Enhanced DRAMs

- **DRAM单元的操作从其发明开始没有变化 Operation of DRAM cell has not changed since its invention**
  - 1970年由Intel商业化 Commercialized by Intel in 1970.
- **具有更好接口逻辑和更快I/O的DRAM核心 DRAM cores with better interface logic and faster I/O :**
  - 同步DRAM（SDRAM） Synchronous DRAM (SDRAM)
    - 使用传统时钟信号代替异步控制 Uses a conventional clock signal instead of asynchronous control
  - 双倍数据率同步DRAM（DDR SDRAM） Double data-rate synchronous DRAM (DDR SDRAM)
    - 双边沿时钟每引脚每周期发送两位 Double edge clocking sends two bits per cycle per pin
    - 根据小型预取缓冲区的大小区分不同类型 Different types distinguished by size of small prefetch buffer:
      - DDR (2 bits), DDR2 (4 bits), DDR3 (8 bits), DDR4 (16 bits)
    - 到2010年，大多数服务器和桌面系统的标准配置 By 2010, standard for most server and desktop systems
    - Intel Core i7支持DDR3和DDR4 SDRAM Intel Core i7 supports DDR3 and DDR4 SDRAM

传统DRAM结构
Conventional DRAM Organization

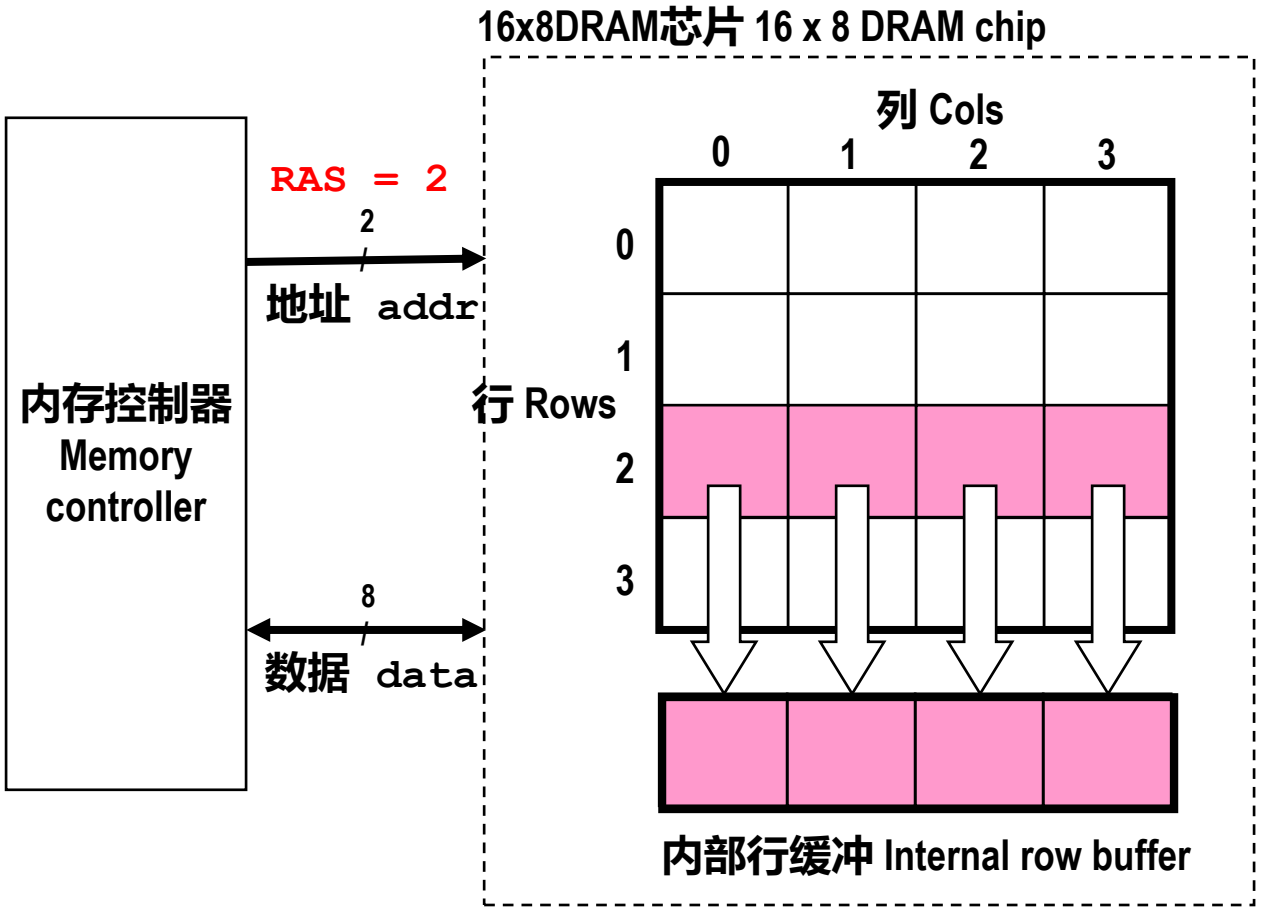
- **d x w DRAM:**
  - 总计d·w位组织成d个w位的超单元 d·w total bits organized as d supercells of size w bits



读取DRAM超单元
Reading DRAM Supercell (2,1)

步骤1a：行访问选通（RAS）选择第2行 Step 1(a): Row access strobe (RAS) selects row 2.

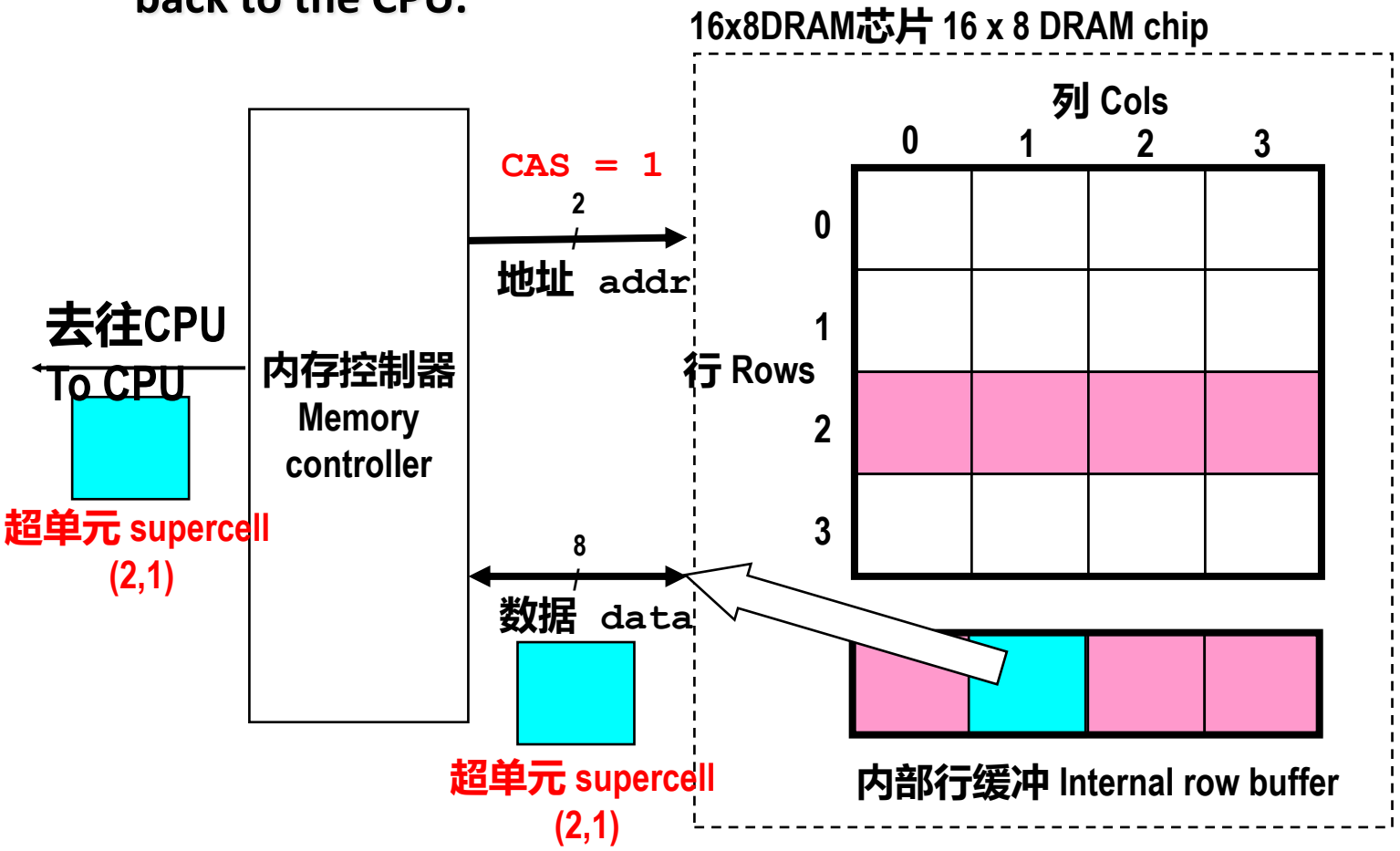
步骤1b：第2行从DRAM矩阵复制到行缓冲 Step 1(b): Row 2 copied from DRAM array to row buffer.



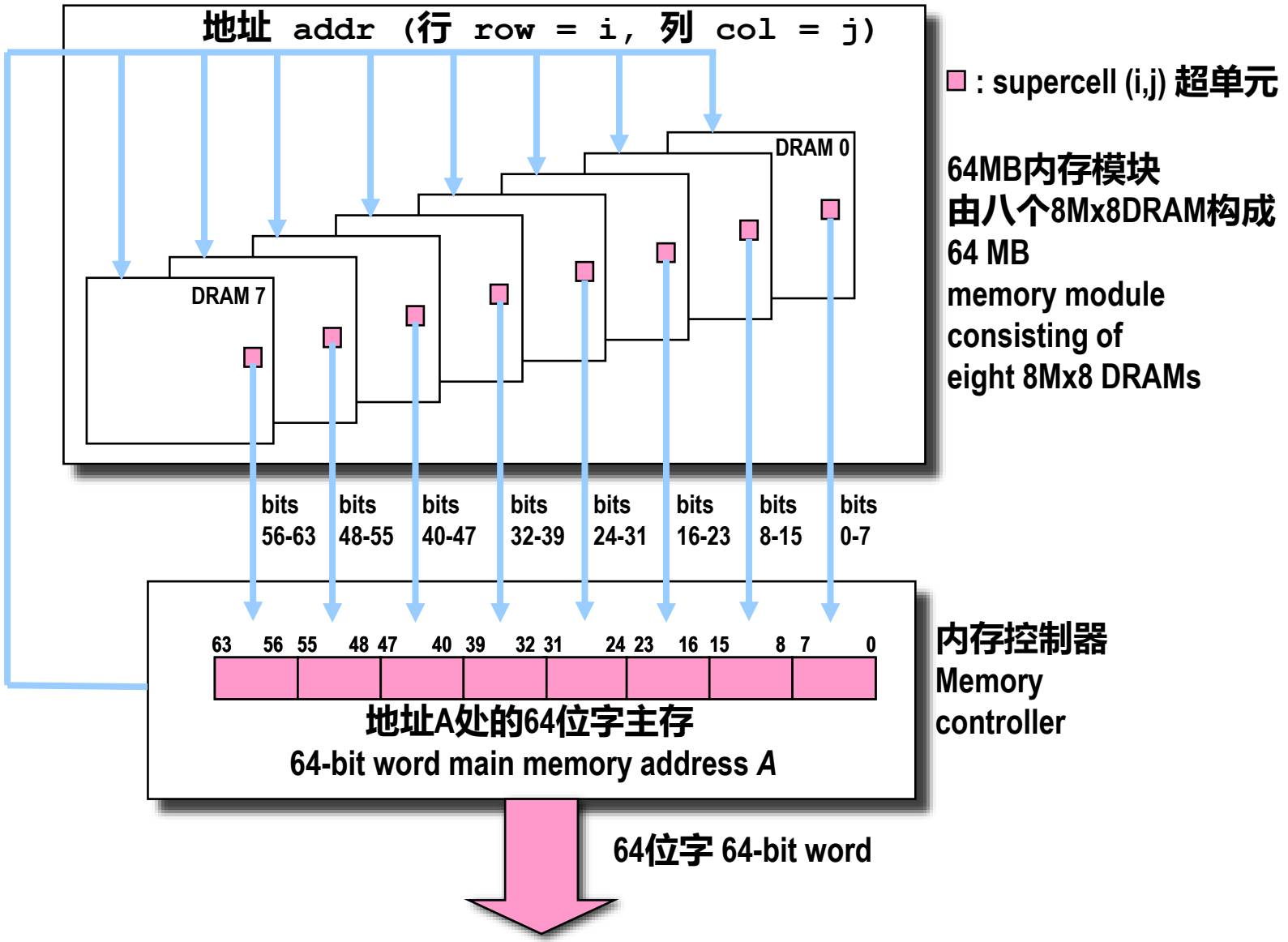
读取DRAM超单元
Reading DRAM Supercell (2,1)

步骤2a：列访问选通（CAS）选择第1列 Step 2(a): Column access strobe (CAS) selects column 1.

步骤2b：超单元（2，1）从缓冲区复制到数据线，最终回到CPU Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.



内存模块
Memory Modules



: supercell (i,j) 超单元

64MB内存模块
由八个8Mx8DRAM构成
64 MB
memory module
consisting of
eight 8Mx8 DRAMs

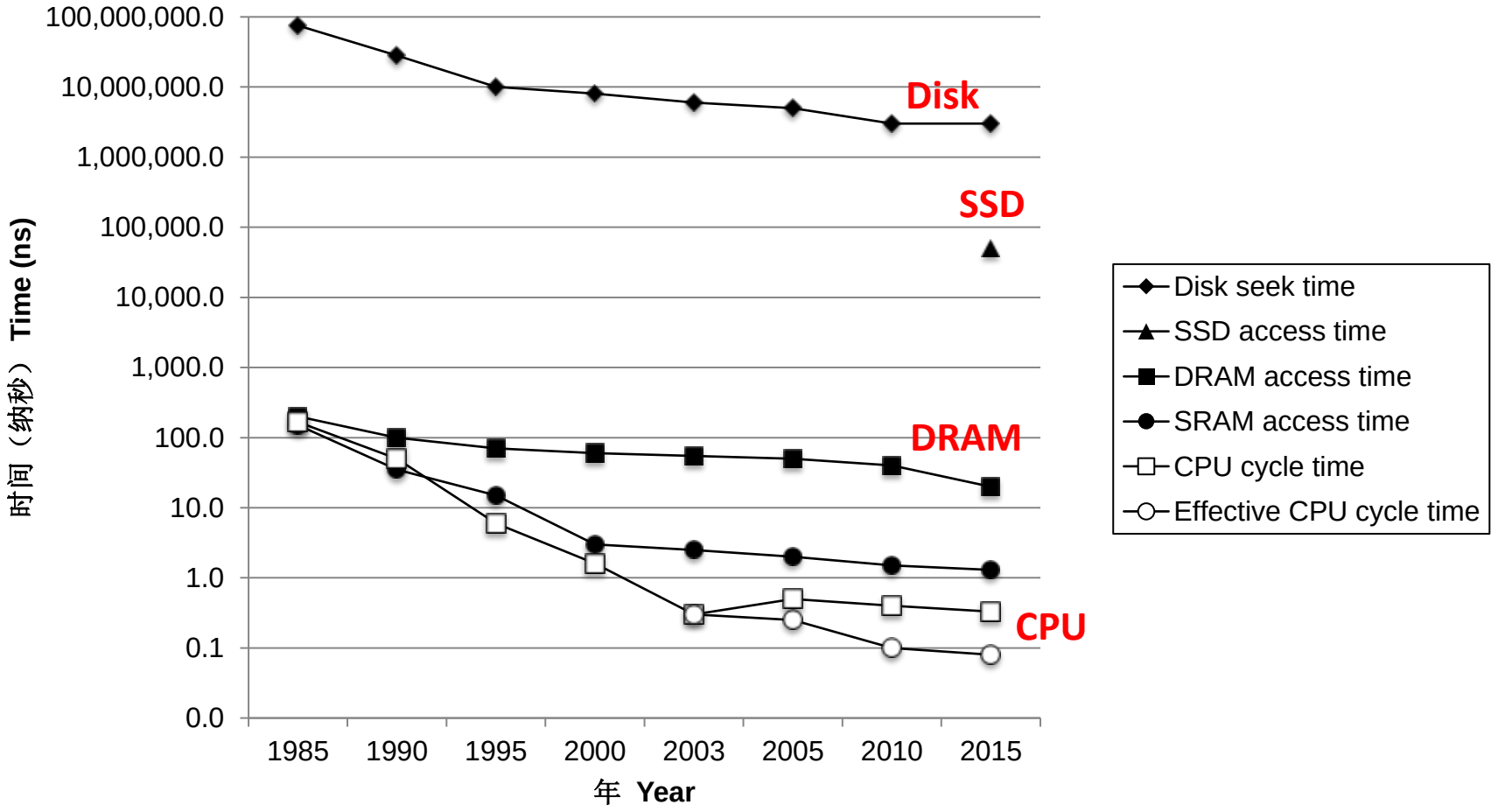
内存控制器
Memory
controller

提纲

- 内存抽象 The memory abstraction
- 随机访问存储器：主存构建块 RAM : main memory building block
- **引用的局部性 Locality of reference**
- 存储器层次结构 The memory hierarchy
- 存储技术和趋势 Storage technologies and trends

CPU-内存差距 The CPU-Memory Gap

DRAM、磁盘和CPU之间速度的差距加大 The gap widens between DRAM, disk, and CPU speeds.



局部性能够弥补这个差距
Locality to the Rescue!

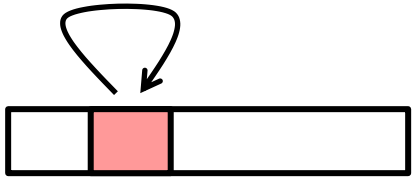
弥补CPU-内存之间速度差距的关键是计算机程序的一个基本属性，即局部性

The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality

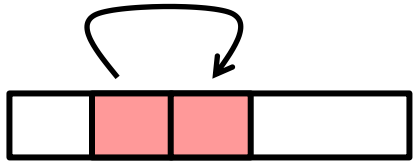
局部性 Locality

- **局部性原理：程序倾向于使用地址接近或等于其最近使用的地址的数据和指令 Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

- **时间局部性 Temporal locality:**
  - 最近引用的项目可能在不久的将来再次引用
  - Recently referenced items are likely to be referenced again in the near future



- **空间局部性 Spatial locality:**
  - 具有附近地址的项目往往在时间上被频繁地引用
  - Items with nearby addresses tend to be referenced close together in time



局部性举例 Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **数据访存 Data references**
  - 按序引用数组元素（步长-1的引用模式）Reference array elements in succession (stride-1 reference pattern).
  - 每次迭代引用变量sum Reference variable `sum` each iteration.
- **指令访存 Instruction references**
  - 顺序引用指令 Reference instructions in sequence.
  - 通过循环周期性重复 Cycle through loop repeatedly.

空间局部性
Spatial locality

时间局部性
Temporal locality

空间局部性
Spatial locality

时间局部性
Temporal locality

局部性量化评估
Qualitative Estimates of Locality

- **声明：对于专业程序员来说，能够查看代码并对局部性进行量化评估是一项关键技能。** **Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.
- **问题：这个函数对于数组a有良好的局部性吗?**
  Question: Does this function have good locality with respect to array a?

提示：数组布局采用行优先顺序
Hint: array layout is row-major order

答案：是 Answer: yes

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

a [0] [0]	. . .	a [0] [N-1]	a [1] [0]	. . .	a [1] [N-1]	. . .	a [M-1] [0]	. . .	a [M-1] [N-1]

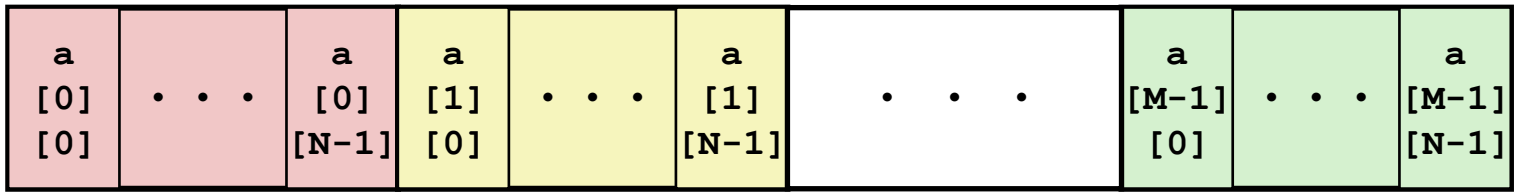
局部性示例 Locality Example

- **问题：这个函数对于数组a有良好的局部性吗?**
  Question: Does this function have good locality with respect to array a?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

答案：否，除非。。。
Answer: no, unless...

M非常小 M is very small



局部性示例 Locality Example

- **问题：您能否排列循环，以便该函数使用步长-1引用模式扫描三维数组a（从而具有良好的空间局部性）？**
  Question: Can you permute the loops so that the function scans the 3-d array a with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```

答案：让j成为最内循环 Answer: make j the inner loop

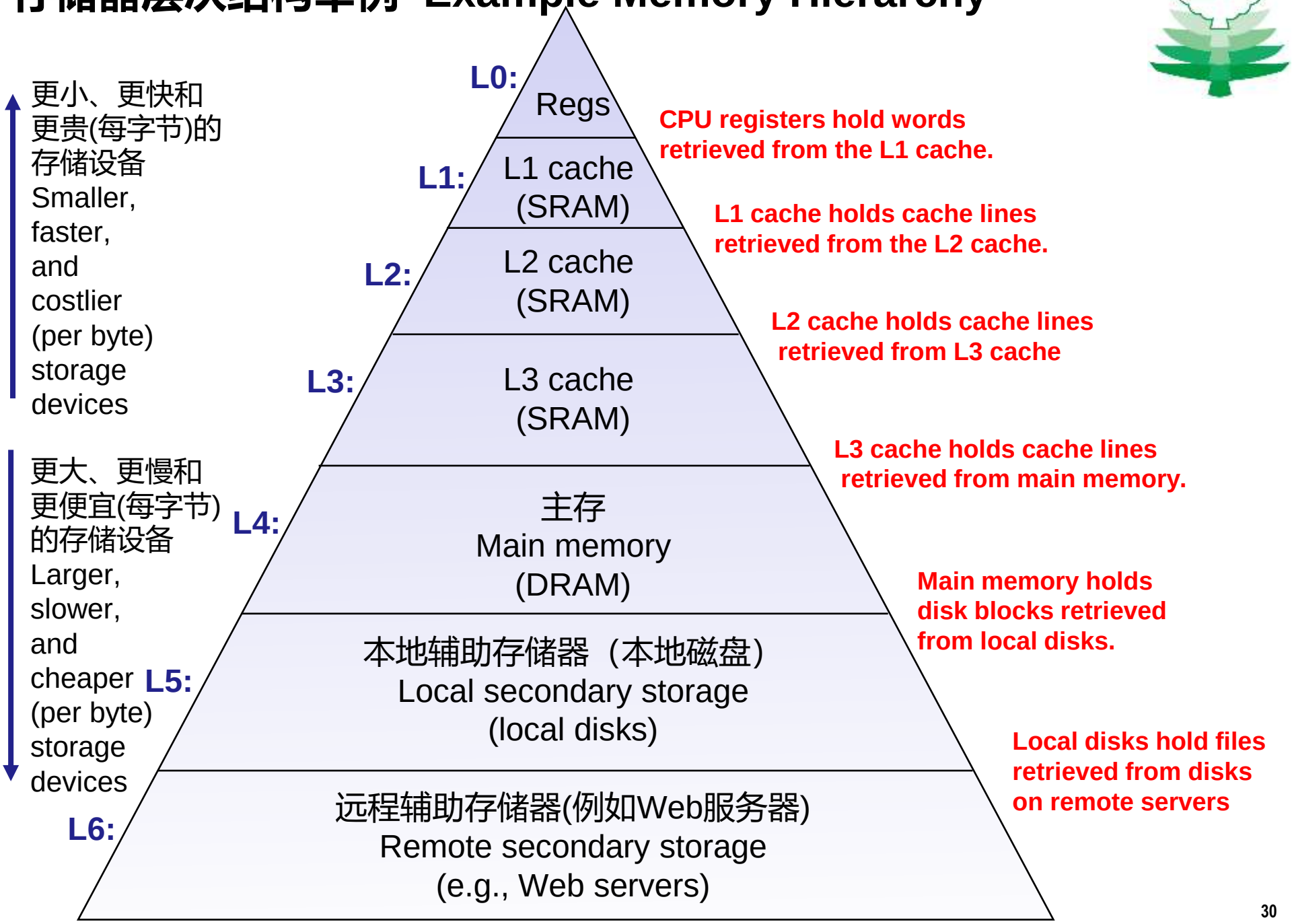
提纲

- 内存抽象 The memory abstraction
- 随机访问存储器：主存构建块 RAM : main memory building block
- 引用的局部性 Locality of reference
- **存储器层次结构 The memory hierarchy**
- 存储技术和趋势 Storage technologies and trends

存储层次结构 Memory Hierarchies

- **硬件和软件的一些基本和持久特性： Some fundamental and enduring properties of hardware and software:**
  - 快速存储技术每字节成本更高，容量更小，并且需要更大功率（发热！） Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - CPU和主存之间的速度差距正在扩大 The gap between CPU and main memory speed is widening.
  - 写得好的程序往往具有良好的局部性 Well-written programs tend to exhibit good locality.
- **对于许多类型的程序，这些属性可以很好地相互补充。 These properties complement each other well for many types of programs.**
- **他们提出了一种组织内存和存储系统的方法，称为存储器层次结构 They suggest an approach for organizing memory and storage systems known as a memory hierarchy.**

存储器层次结构举例 Example Memory Hierarchy



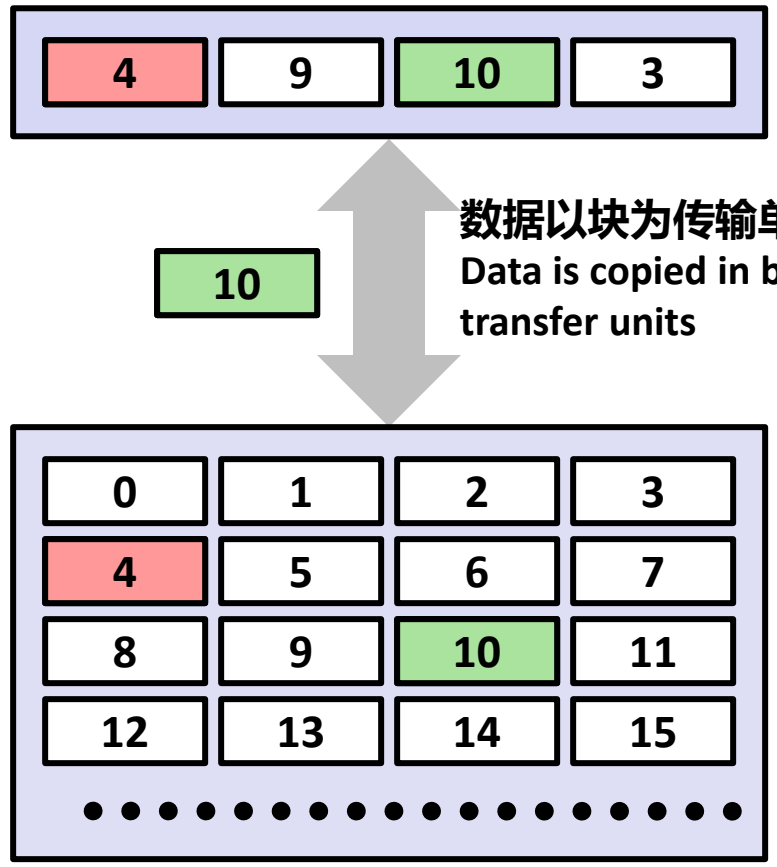
高速缓存 Caches

- ***缓存:*** **一种较小、较快的存储设备，用作较大、较慢设备中数据子集的暂存区域。** ***Cache:*** **A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.**
- **存储器层次结构的基本思想： Fundamental idea of a memory hierarchy:**
  - 对于每个k，级别为k的更快、更小的设备充当级别为k+1的更大、更慢设备的缓存 For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- **为什么存储器层次结构有效？ Why do memory hierarchies work?**
  - 由于局部性的原因：程序倾向于访问k级的数据，而不是访问k+1级的数据 Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - 因此，k+1级的存储速度可能较慢，因此存储量更大、每比特的成本更低。 Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.

高速缓存 Caches

- 大思想：存储器层次结构创建了一个大的存储池，其成本与底部的廉价存储一样多，但它以接近顶部的快速存储速率向程序提供数据。
- *Big Idea:* The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Cache一般概念 General Cache Concepts

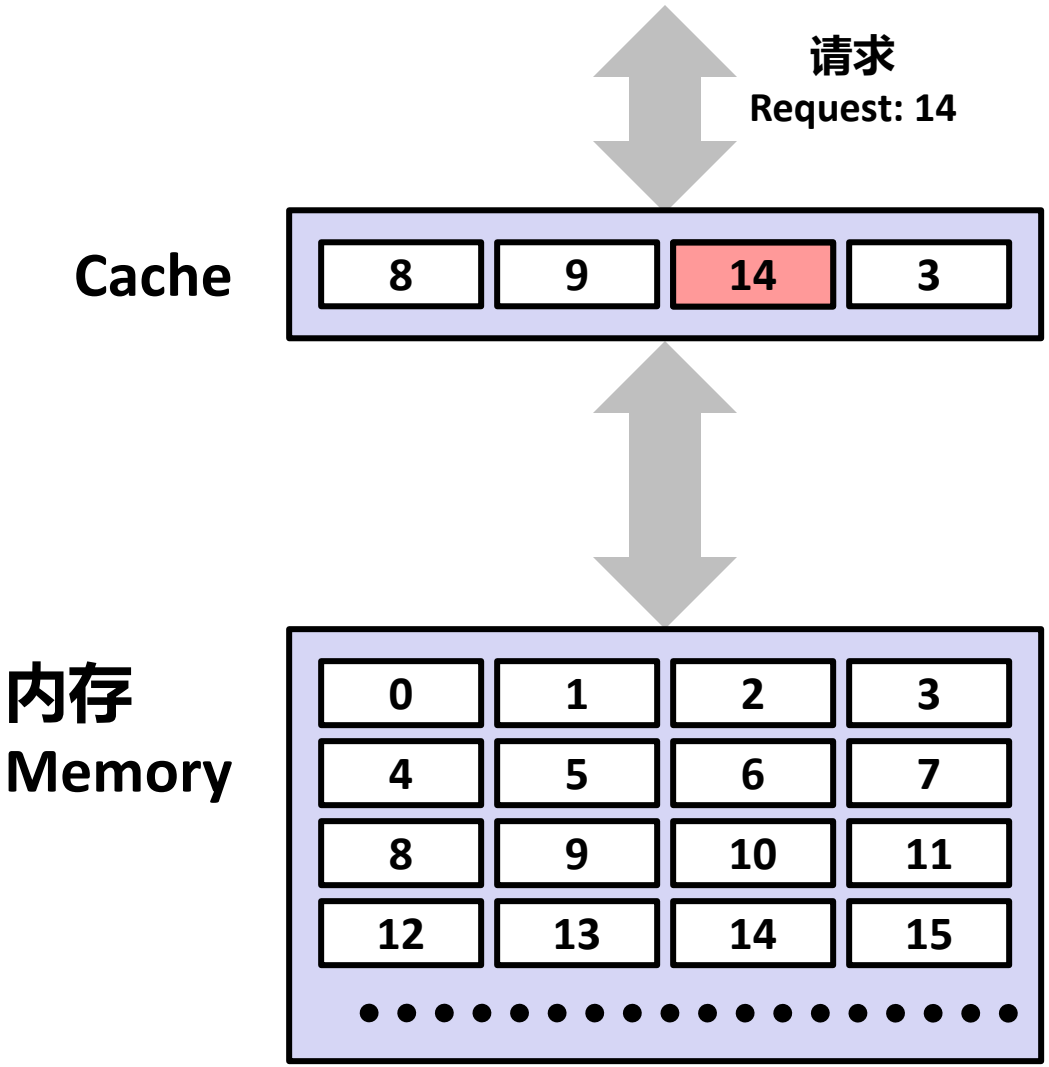


更小、更快和更贵的存储器缓存一组块
Smaller, faster, more expensive memory caches a subset of the blocks

数据以块为传输单位进行复制
Data is copied in block-sized transfer units

更大、更慢和更便宜的内存看成分区的“块”
Larger, slower, cheaper memory viewed as partitioned into “blocks”

Cache一般概念：命中
General Cache Concepts: Hit



需要数据块 b

Data in block b is needed

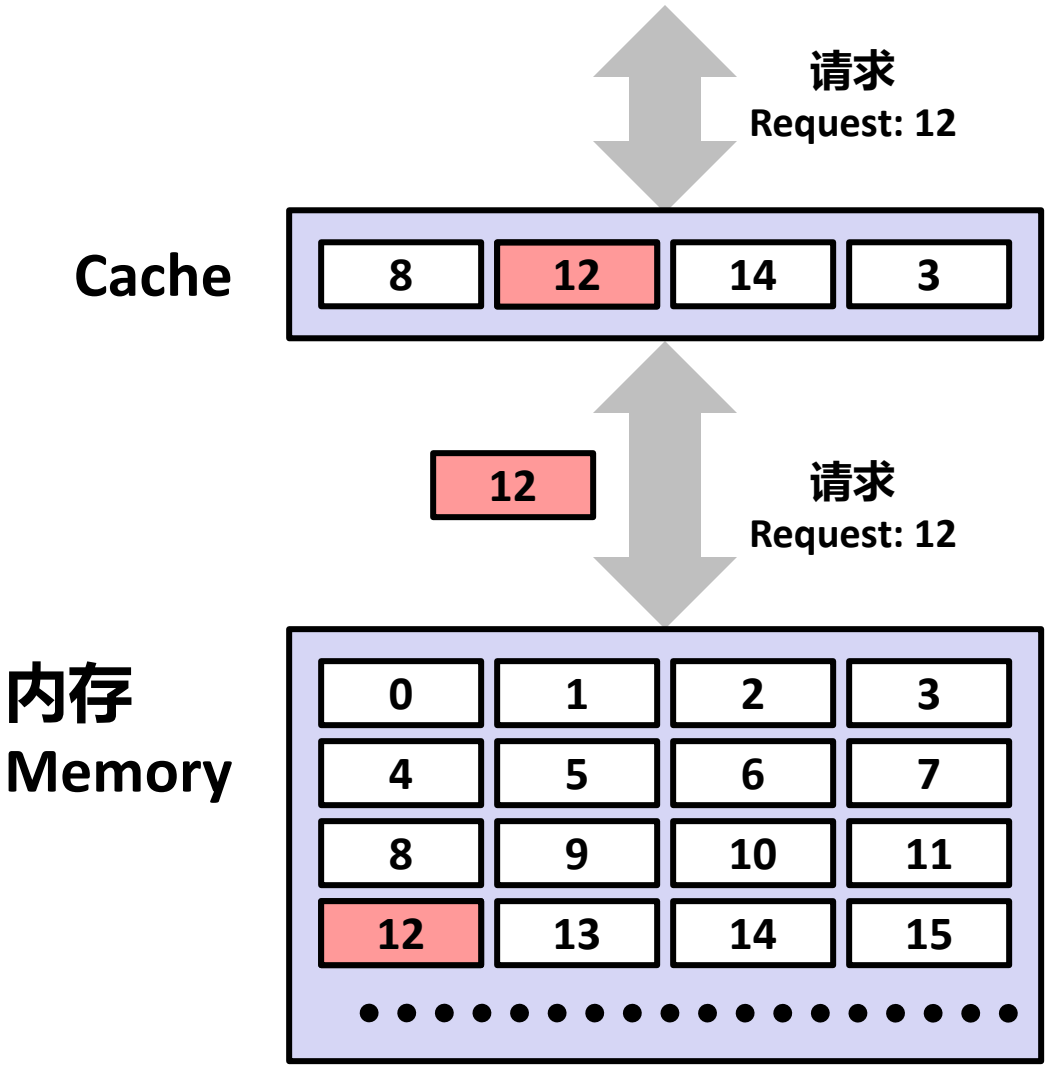
块 b 在 cache 中：

命中!

Block b is in cache:

Hit!

Cache一般概念：不命中
General Cache Concepts: Miss



需要数据块 b

Data in block b is needed

块 b 没在 cache 中：不命中！

Block b is not in cache: Miss!

从内存取块 b

Block b is fetched from memory

Cache 中存储的块 b

Block b is stored in cache

- 放置策略：确定b放在哪
  Placement policy:
  determines where b goes
- 替换策略：确定驱逐哪个块（牺牲）Replacement policy:
  determines which block gets evicted (victim)

Cache一般概念：Cache不命中类型
General Caching Concepts: Types of Cache Misses

- **冷（强制）不命中 Cold (compulsory) miss**
  - 因为cache为空而发生冷不命中 Cold misses occur because the cache is empty.
- **冲突不命中 Conflict miss**
  - 大多数缓存将k+1级的块限制为k级块位置的一个小子集（有时是单个位置）Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - 例如k+1层的块i必须放置在k层的块（i mod 4）中 E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - 当k级缓存足够大，但多个数据对象都映射到同一个k级块时，就会发生冲突不命中 Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - 例如引用块0、8、0、8，0、8…每次都会不命中 E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- **容量不命中 Capacity miss**
  - 当活动缓存块集（工作集）大于缓存时发生 Occurs when the set of active cache blocks (working set) is larger than the cache.

存储器层次结构中的Cache举例
Examples of Caching in the Mem. Hierarchy

Cache类型 Cache Type	缓存什么? What is Cached?	缓存到哪儿? Where is it Cached?	延迟（周期） Latency (cycles)	管理 Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

提纲

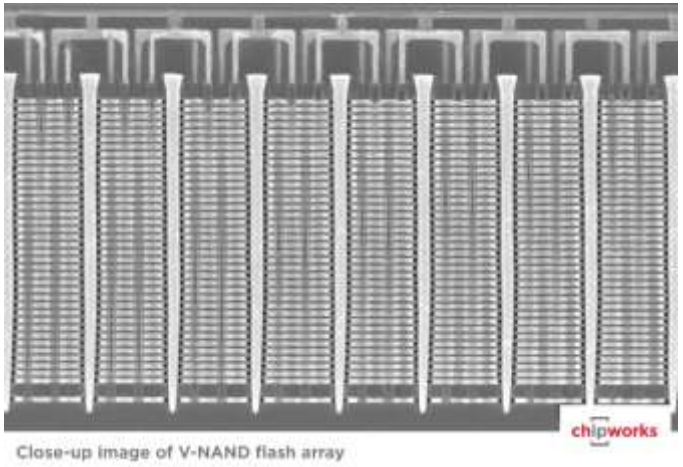
- 内存抽象 The memory abstraction
- 随机访问存储器：主存构建块 RAM : main memory building block
- 引用局部性 Locality of reference
- 存储器层次结构 The memory hierarchy
- **存储技术和趋势 Storage technologies and trends**

存储技术 Storage Technologies

- 磁盘 Magnetic Disks

- 存储在磁介质上 Store on magnetic medium
- 电子机械访问 Electromechanical access

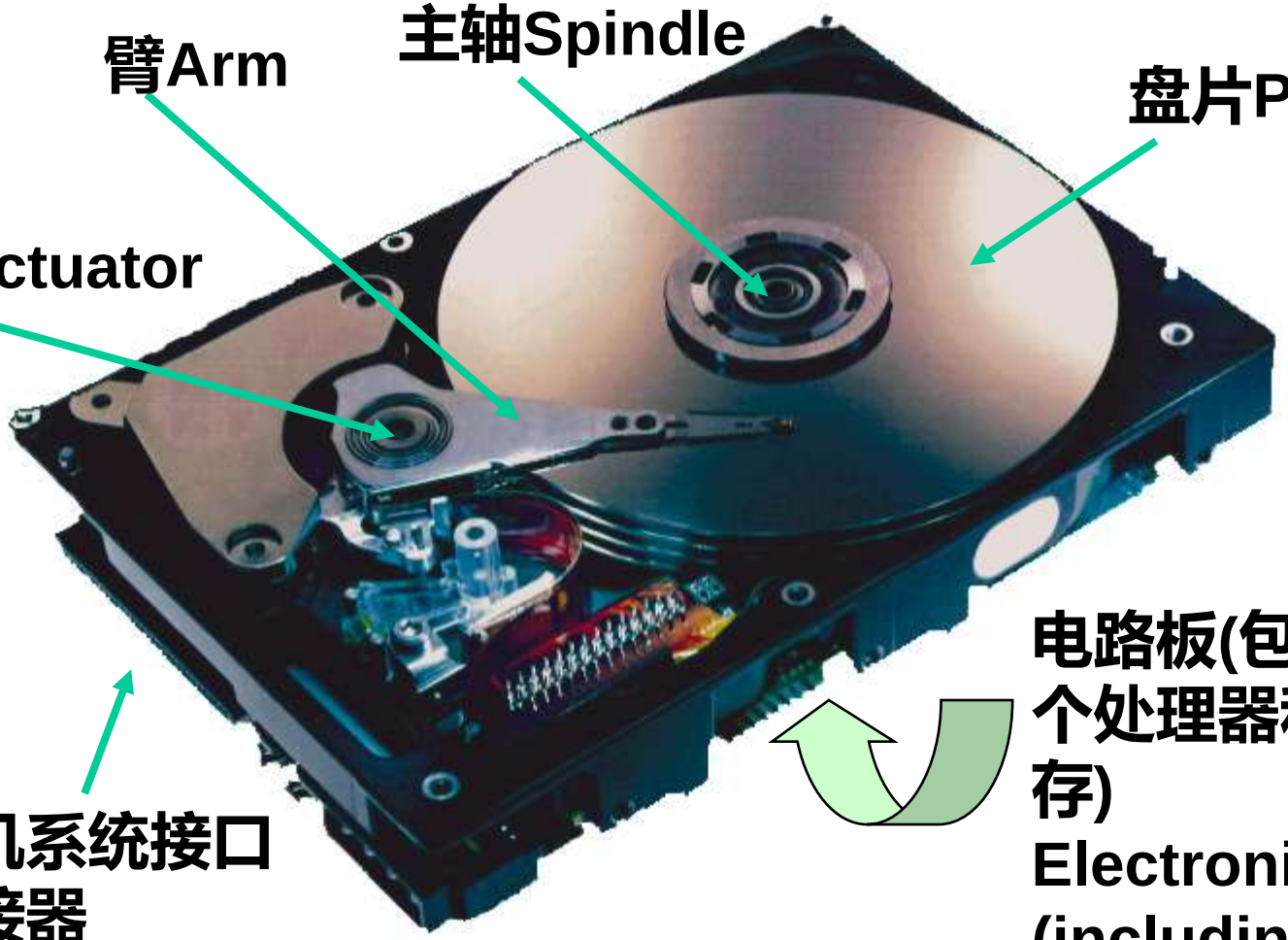
- 非易失（闪存）存储器 Nonvolatile (Flash) Memory



Close-up image of V-NAND flash array

- 永久性存储 Store as persistent charge
- 用三维结构实现 Implemented with 3-D structure
  - 100以上的单元级别 100+ levels of cells
  - 每个单元3-4位数据 3-4 bits data per cell

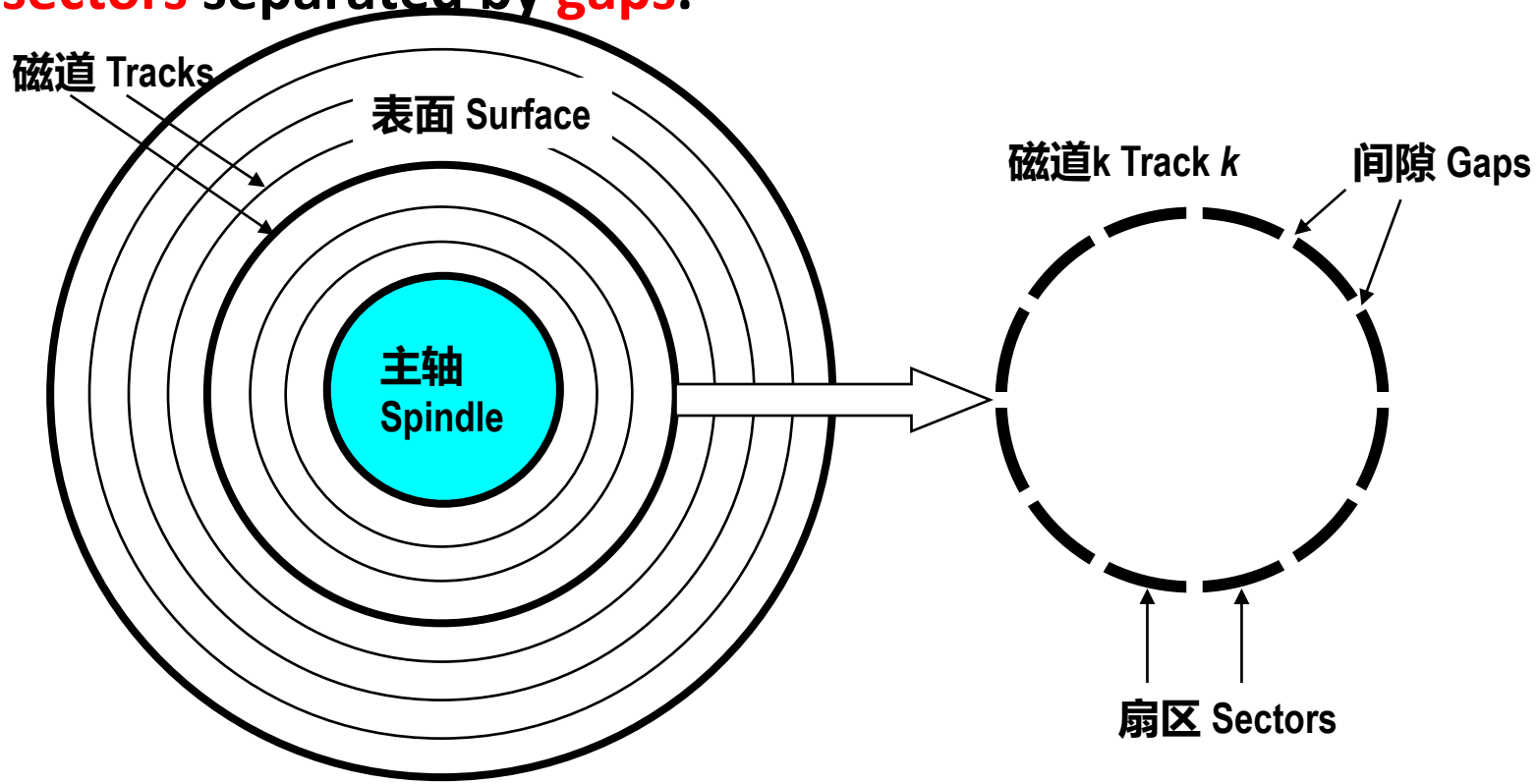
硬盘驱动器内部有什么？What's Inside A Disk Drive?



图片由希捷科技提供
Image courtesy of Seagate Technology

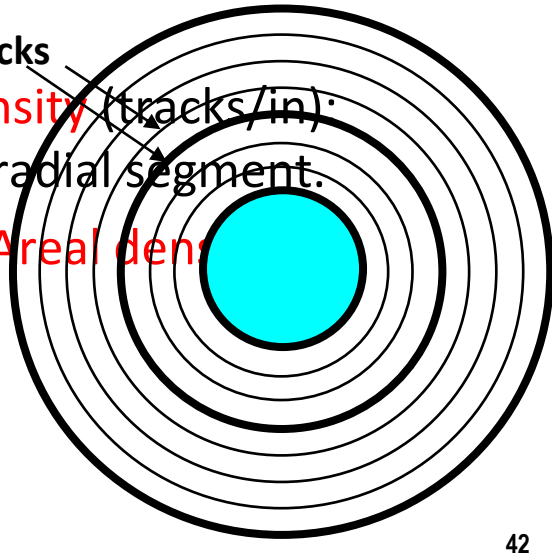
磁盘结构 Disk Geometry

- 磁盘由盘片构成，每个有两个表面 Disks consist of platters, each with two surfaces.
- 每个表面由同心圆环构成，称为磁道 Each surface consists of concentric rings called tracks.
- 每个磁道由扇区构成，中间用间隙分隔 Each track consists of sectors separated by gaps.



磁盘容量 Disk Capacity

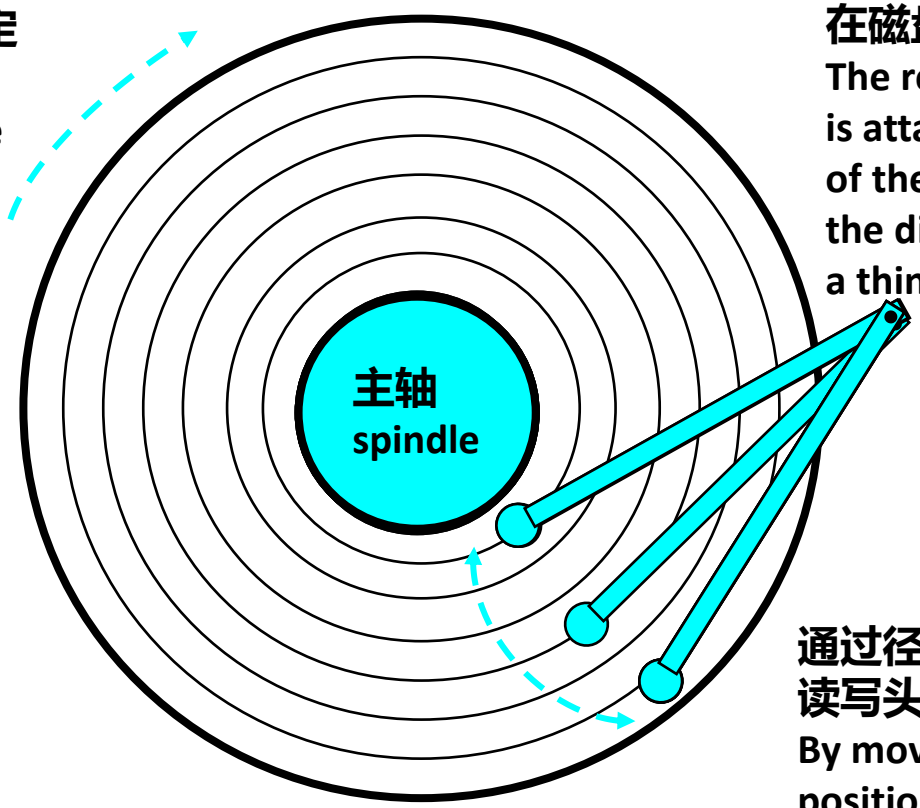
- **容量：可以存储的最大位数 Capacity: maximum number of bits that can be stored.**
  - 供应商表示容量以GB为单位，其中1 GB = 10^9 字节 Vendors express capacity in units of gigabytes (GB), where 1 GB = 10^9 Bytes.
- **容量由这些技术因素确定 Capacity is determined by these technology factors:**
  - 记录密度(位/英寸)：一个磁道中1英寸段存储的位数 Recording density (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
  - 道密度(道/英寸)：1英寸半径段的磁道数 Track density (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
  - 面密度(位/平方英寸)：记录密度和道密度的乘积 Areal density (bits/in2): product of recording and track density.



磁盘操作（单盘片视图）
Disk Operation (Single-Platter View)

磁盘表面以固定旋转速率旋转
The disk surface spins at a fixed rotational rate

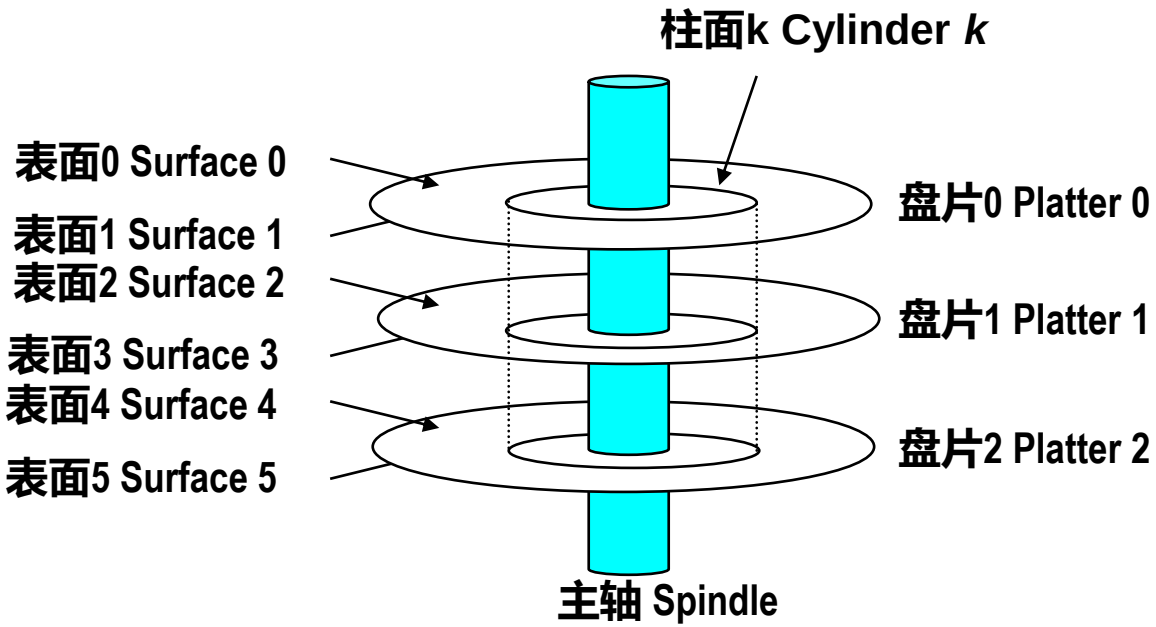
读/写头连接到传动臂末端，悬浮在磁盘表面上飞过
The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.



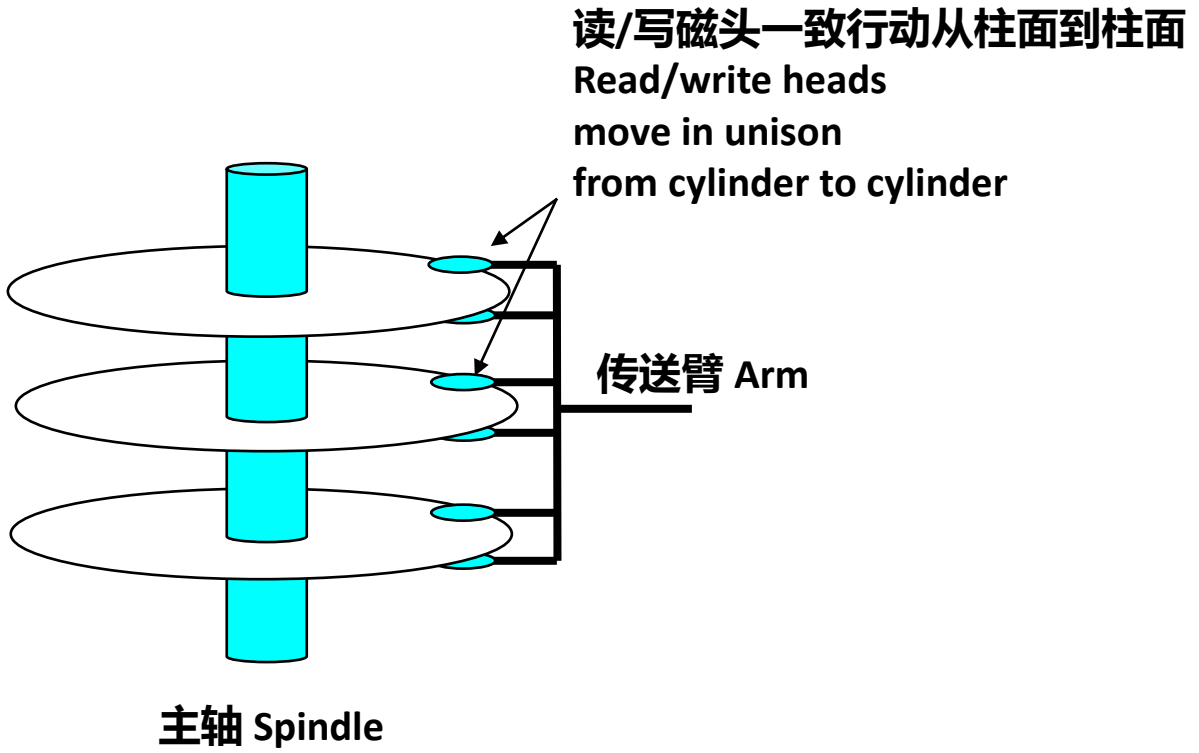
通过径向移动，传动臂可以将读写头放置在任何磁道上。
By moving radially, the arm can position the read/write head over any track.

磁盘操作（多盘片视图）
Disk Operation (Muliple-Platter View)

- 同心磁道形成一个柱面 Aligned tracks form a cylinder.

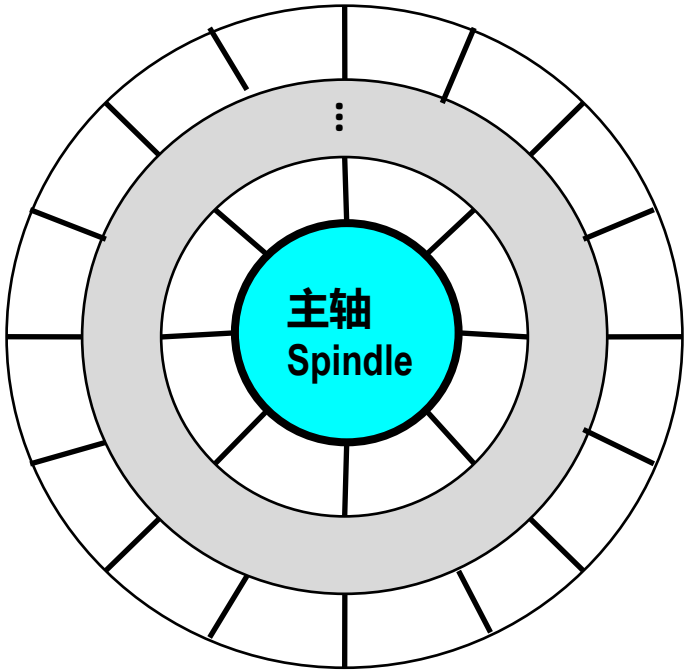


磁盘操作（多盘片视图）
Disk Operation (Multi-Platter View)



记录区 Recording zones

- **现代磁盘把磁道分区成不相交的子集，称为记录区 Modern disks partition tracks into disjoint subsets called recording zones**
  - 在区内每个磁道有同样的扇区数，由最内磁道的圆周确定 Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
  - 每个区有不同每道扇区数，外侧区比内侧区有更多的每道扇区数 Each zone has a different number of sectors/track, outer zones have more sectors/track than inner zones.
  - 因此当计算容量时我们使用平均每道扇区数 So we use **average** number of sectors/track when computing capacity.



计算磁盘容量 Computing Disk Capacity

容量 =字节数/扇区 x 平均扇区数/磁道 x
磁道数/面 x 表面数/盘片 x 盘片数/磁盘

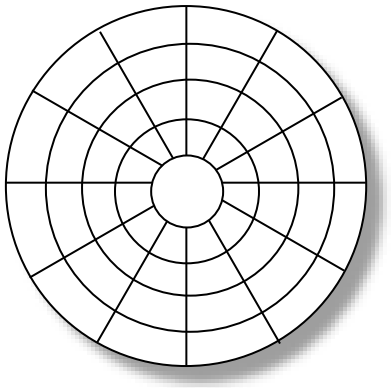
Capacity = (# bytes/sector) x (avg. # sectors/track) x
(# tracks/surface) x (# surfaces/platter) x
(# platters/disk)

例如： Example:

- 512 bytes/sector 字节数/扇区
- 300 sectors/track (on average) 扇区数/磁道（平均）
- 20,000 tracks/surface 磁道数/面
- 2 surfaces/platter 表面数/盘片
- 5 platters/disk 盘片数/磁盘

容量 Capacity = 512 x 300 x 20000 x 2 x 5
= 30,720,000,000
= 30.72 GB

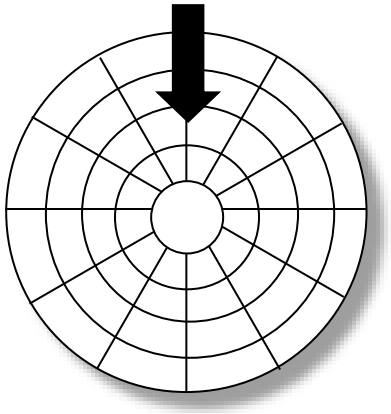
磁盘结构-单个盘片的顶层视图
Disk Structure - top view of single platter



表面组织成磁道
Surface organized into tracks

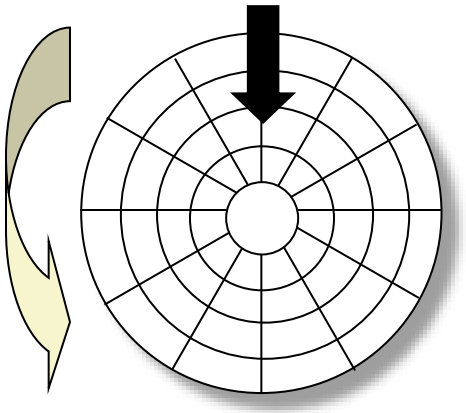
磁道分成扇区
Tracks divided into sectors

磁盘访问 Disk Access



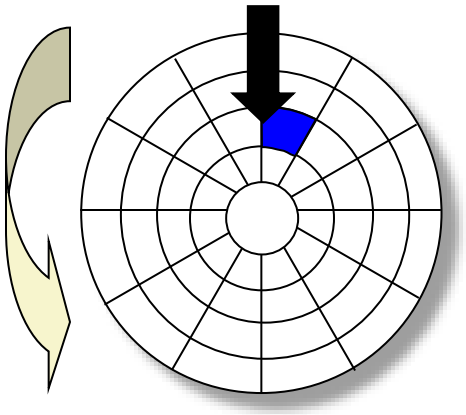
磁头在磁道上某个位置
Head in position above a track

磁盘访问 Disk Access



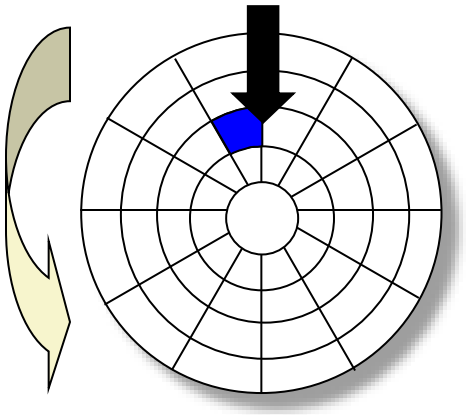
逆时针旋转
Rotation is counter-clockwise

磁盘访问-读 Disk Access – Read



准备读取蓝色扇区
About to read blue sector

磁盘访问-读 Disk Access – Read



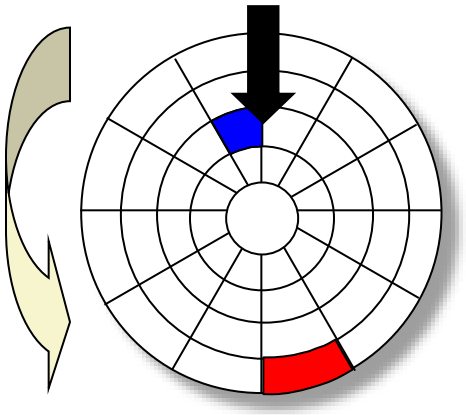
读蓝色扇区后

After BLUE read

读取蓝色扇区之后

After reading blue sector

磁盘访问-读 Disk Access – Read



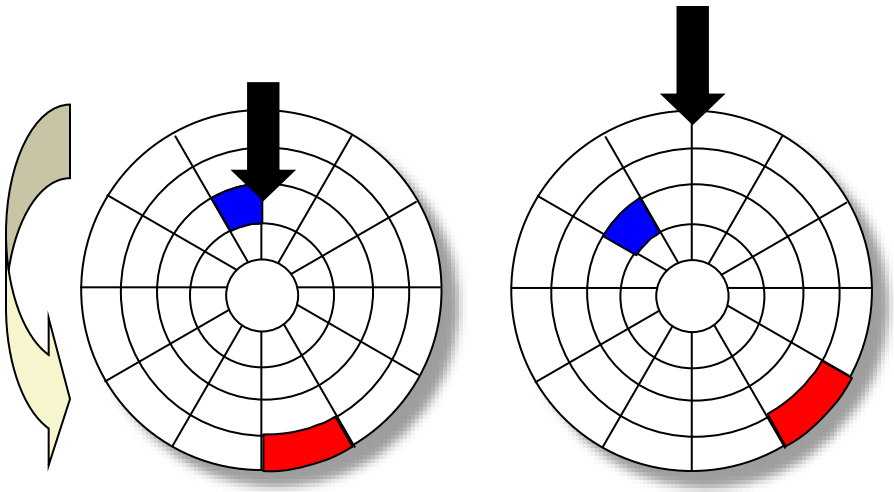
读蓝色扇区后

After BLUE read

下次请求调度红色扇区

Red request scheduled next

磁盘访问-寻道 Disk Access – Seek



读蓝色扇区后

After BLUE read

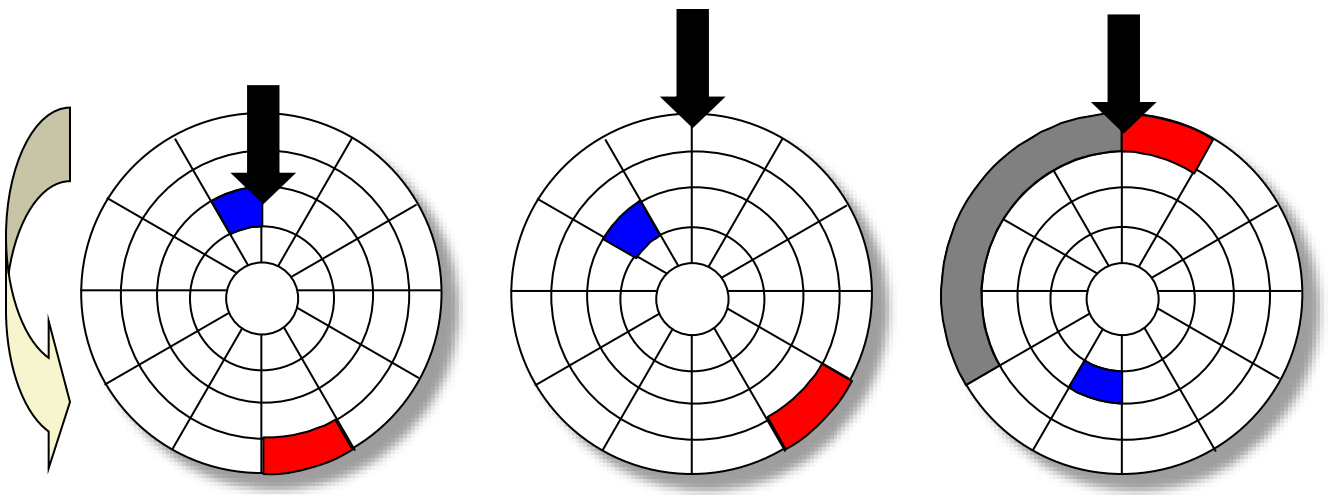
寻找红色磁道

Seek for RED

寻找红色磁道

Seek to red's track

磁盘访问-旋转延迟 Disk Access – Rotational Latency



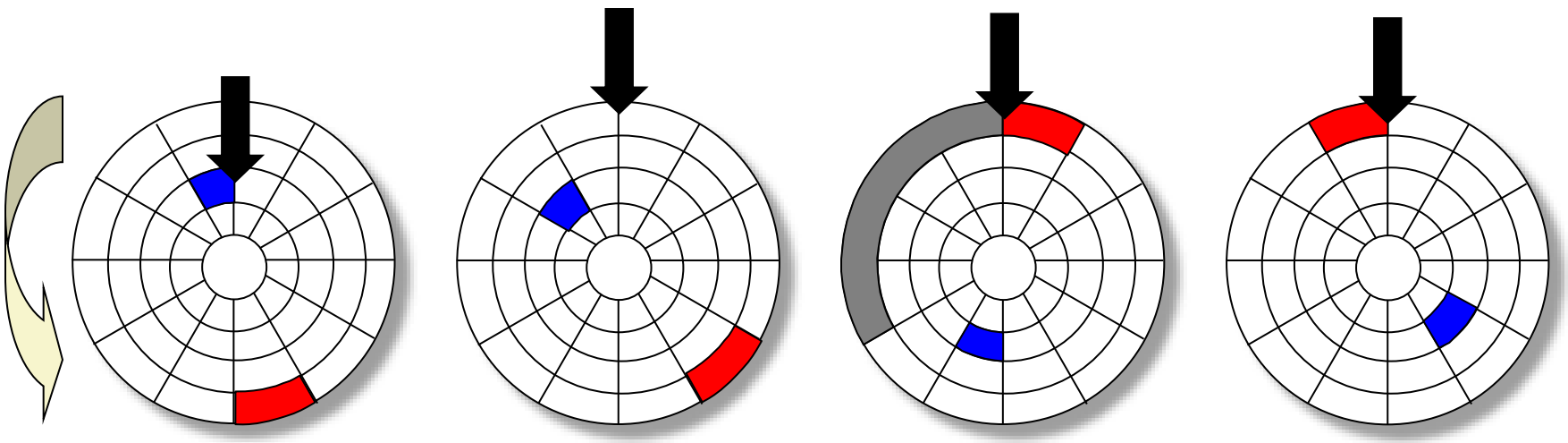
读蓝色后 寻找红色磁道 旋转延迟

After BLUE read Seek for RED Rotational latency

等待旋转到红色扇区

Wait for red sector to rotate around

磁盘访问-读 Disk Access – Read



读蓝色扇区后
After BLUE read

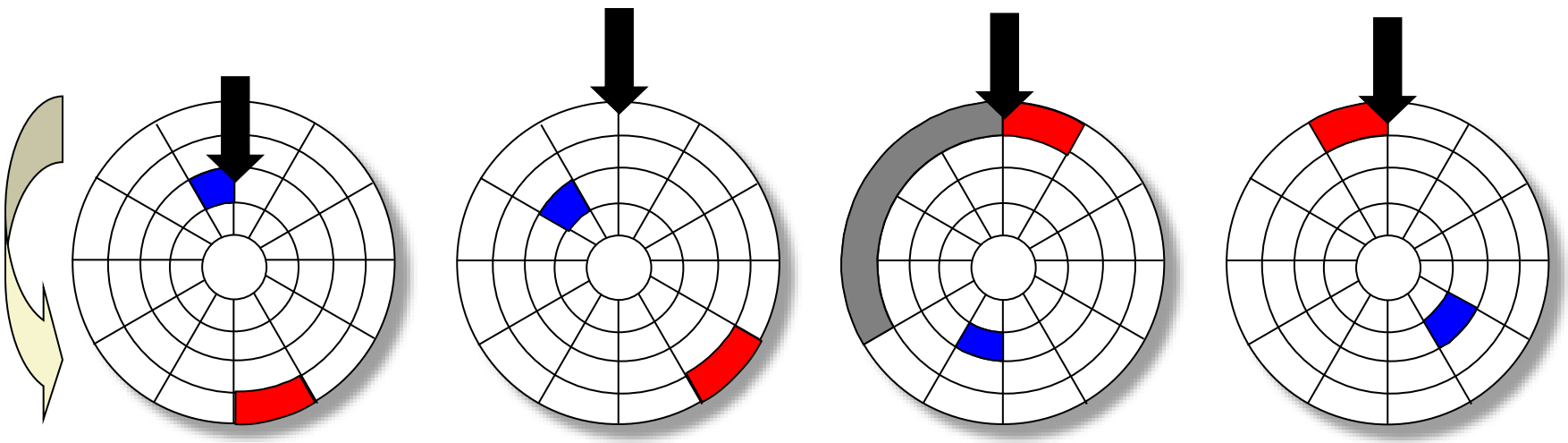
寻找红色磁道
Seek for RED

旋转延迟
Rotational latency

读取红色扇区后
After RED read

完成红色扇区读取
Complete read of red

磁盘访问-服务时间构成
Disk Access – Service Time Components



数据传输
Data transfer

寻找磁道
Seek

旋转延迟
Rotational latency

数据传输
Data transfer

磁盘访问时间 Disk Access Time

- **访问某个目标扇区平均时间约为： Average time to access some target sector approximated by :**
  - Taccess = Tavg seek + Tavg rotation + Tavg transfer
- **寻道时间 Seek time (Tavg seek)**
  - 磁头定位到包含目标扇区的柱面所需时间 Time to position heads over cylinder containing target sector.
  - 典型平均寻道时间为3-9ms Typical Tavg seek is 3—9 ms
- **旋转时间 Rotational latency (Tavg rotation)**
  - 等待目标扇区第一位通过读/写磁头下面的时间 Time waiting for first bit of target sector to pass under r/w head.
  - 平均旋转时间=1/2 x 1/每分钟转数 x 60秒/分钟 Tavg rotation = 1/2 x 1/RPMs x 60 sec/1 min
  - 典型平均旋转时间为每分钟7200转 Typical Tavg rotation = 7200 RPMs
- **传输时间 Transfer time (Tavg transfer)**
  - 读取目标扇区位的时间 Time to read the bits in the target sector.
  - 平均传输时间=1/每分钟转数 x 1/(平均每道扇区数) x 60秒/分钟 Tavg transfer = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min.

磁盘访问时间举例 Disk Access Time Example

- **假定： Given:**
  - 旋转速率 Rotational rate = 7,200 RPM
  - 平均寻道时间 Average seek time = 9 ms.
  - 每道平均扇区数 Avg # sectors/track = 400.
- **推导： Derived:**
  - 平均旋转时间 Tavg rotation = 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms.
  - 平均传输时间 Tavg transfer = 60/7200 RPM x 1/400 secs/track x 1000 ms/sec = 0.02 ms
  - 访问时间 Taccess = 9 ms + 4 ms + 0.02 ms

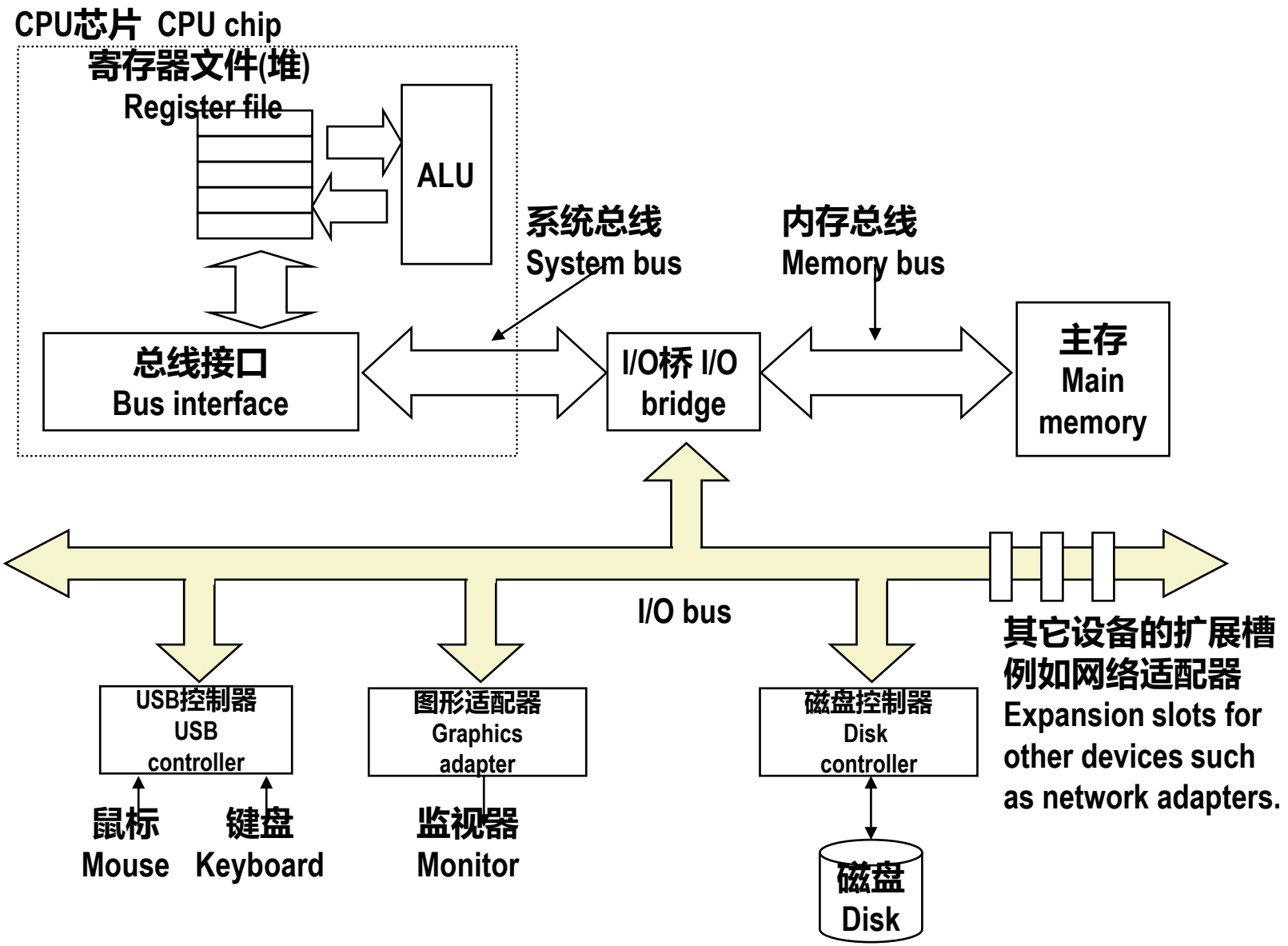
磁盘访问时间举例 Disk Access Time Example

- **重要点：Important points:**
  - 访问时间中大部分为寻道时间和旋转延迟 Access time dominated by seek time and rotational latency.
  - 访问扇区中第一位时间所需最长，扇区中其它位访问时间很短 First bit in a sector is the most expensive, the rest are free.
  - 静态RAM访问时间大约双字为4ns，动态RAM大约为60ns SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    - 磁盘比静态RAM慢约4万倍Disk is about 40,000 times slower than SRAM,
    - 比动态RAM慢约2千5百倍 2,500 times slower then DRAM.

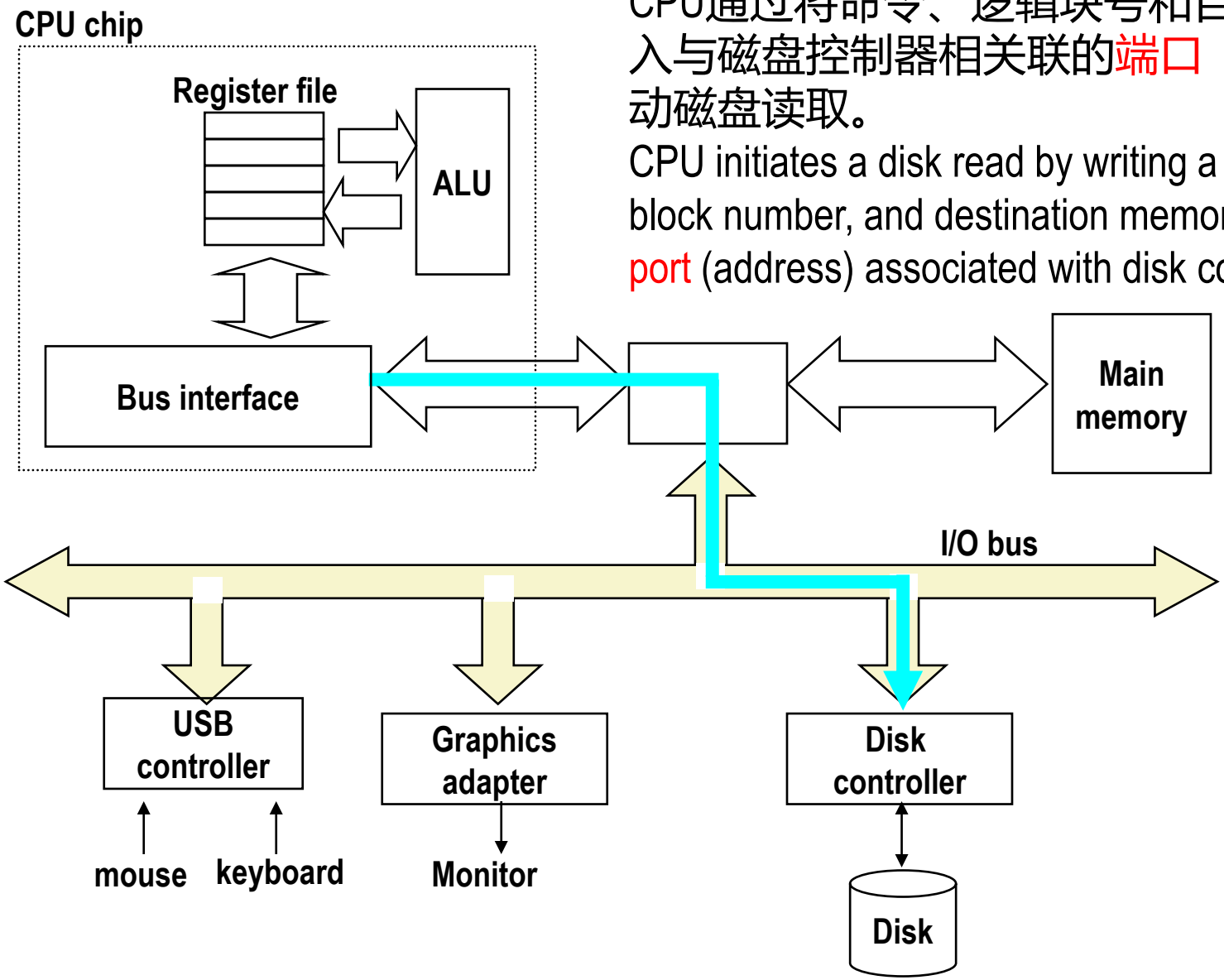
逻辑磁盘块 Logical Disk Blocks

- **现代磁盘提供了复杂扇区结构的更简单抽象视图： Modern disks present a simpler abstract view of the complex sector geometry:**
  - 可用扇区集建模为一系列b大小的逻辑块（0、1、2…） The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, ...)
- **逻辑块和实际（物理）扇区之间的映射 Mapping between logical blocks and actual (physical) sectors**
  - 由称为磁盘控制器的硬件/固件设备维护。 Maintained by hardware/firmware device called disk controller.
  - 将逻辑块请求转换为（表面、磁道、扇区）三元组。 Converts requests for logical blocks into (surface,track,sector) triples.
- **允许控制器为每个区域留出备用柱面 Allows controller to set aside spare cylinders for each zone.**
  - 这也是 “格式化容量”和“最大容量”之间存在差异的原因 Accounts for the difference in “formatted capacity” and “maximum capacity”.

I/O总线 I/O Bus


CPU芯片 CPU chip
寄存器文件(堆)
Register file
ALU
系统总线
System bus
内存总线
Memory bus
总线接口
Bus interface
I/O桥 I/O bridge
主存
Main memory
I/O bus
其它设备的扩展槽
例如网络适配器
Expansion slots for other devices such as network adapters.
USB控制器
USB controller
图形适配器
Graphics adapter
磁盘控制器
Disk controller
鼠标
Mouse
键盘
Keyboard
监视器
Monitor
磁盘
Disk

读取磁盘扇区（1）
Reading a Disk Sector (1)



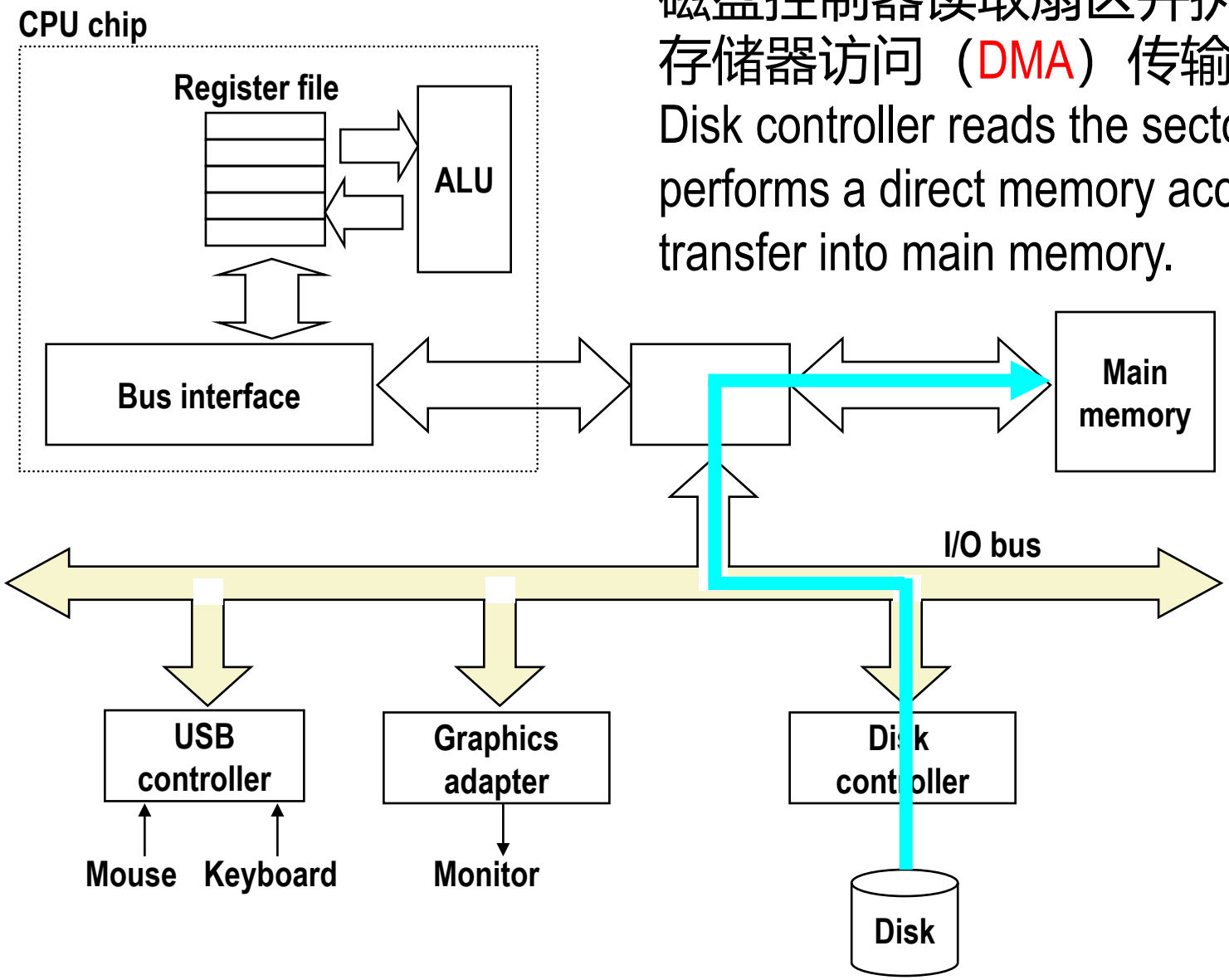
CPU通过将命令、逻辑块号和目标内存地址写入与磁盘控制器相关联的端口（地址）来启动磁盘读取。

CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

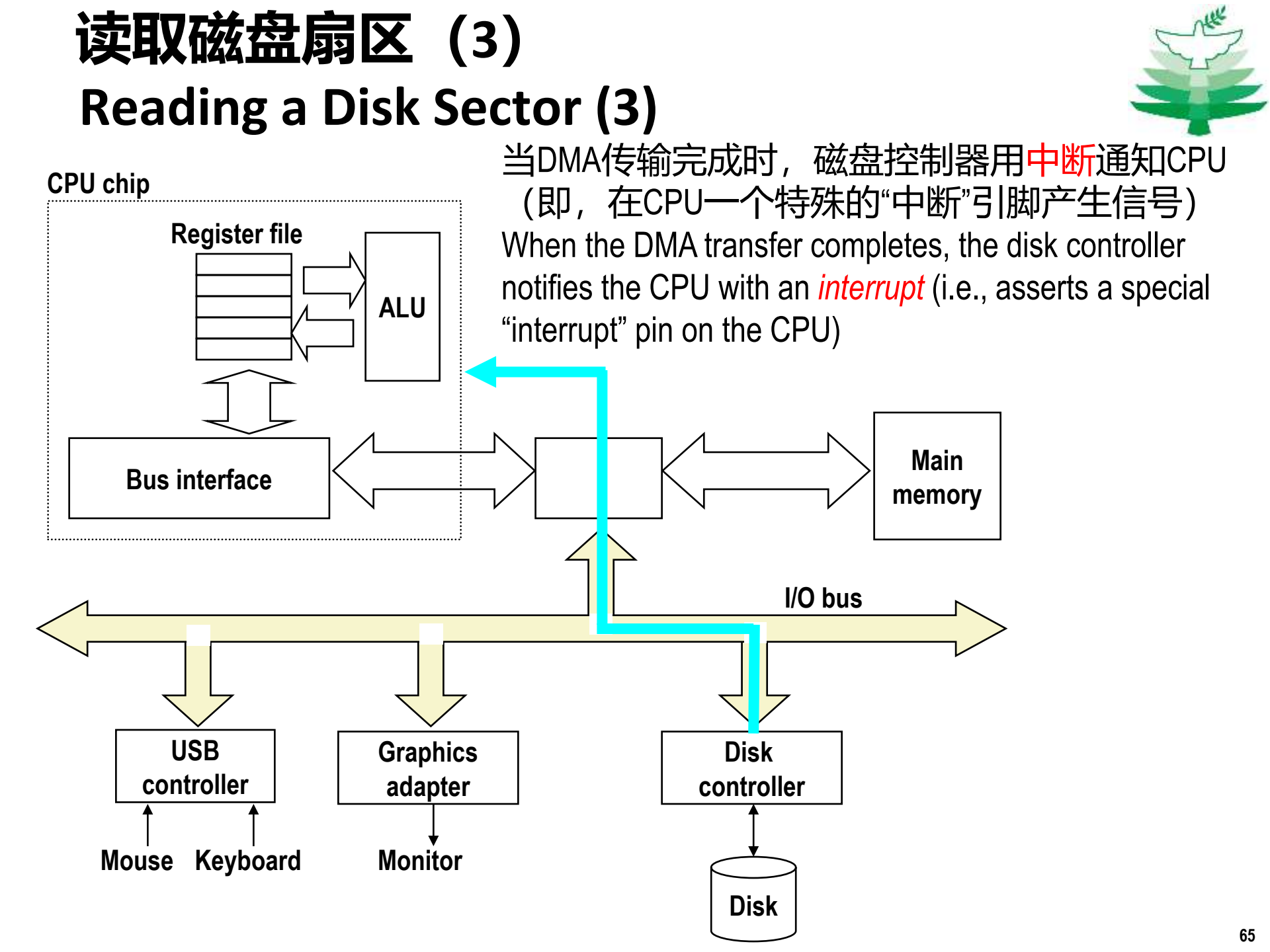
读取磁盘扇区（2）
Reading a Disk Sector (2)

磁盘控制器读取扇区并执行直接存储器访问（DMA）传输到主存

Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.



读取磁盘扇区（3）
Reading a Disk Sector (3)



当DMA传输完成时，磁盘控制器用中断通知CPU（即，在CPU一个特殊的“中断”引脚产生信号）

When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special “interrupt” pin on the CPU)

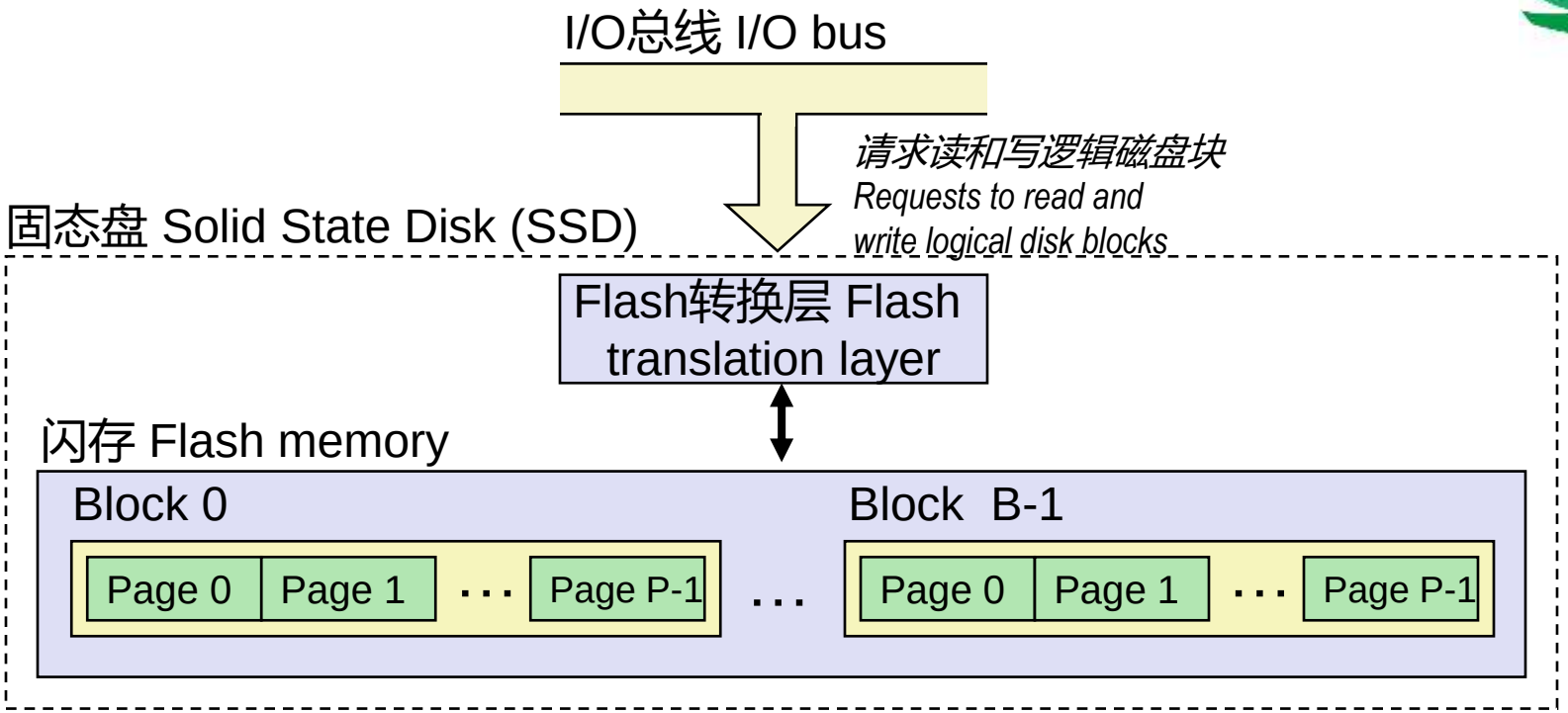
非易失性存储器 Nonvolatile Memories

- **DRAM和SRAM是易失性存储器 DRAM and SRAM are volatile memories**
  - 断电时会丢失信息 Lose information if powered off.
- **即使断电，非易失性存储器仍能保持存储的值 Nonvolatile memories retain value even if powered off**
  - 只读存储器（ROM）：在生产过程中编程 Read-only memory (ROM): programmed during production
  - 电可擦除PROM（EEPROM）：电子擦除功能 Electrically eraseable PROM (EEPROM): electronic erase capability
  - 闪存：EEPROM，具有部分（块级）擦除功能 Flash memory: EEPROMs, with partial (block-level) erase capability
    - 大约100000次擦除后会磨损 Wears out after about 100,000 erasings
  - 3D XPoint（Intel Optane）和新兴NVM 3D XPoint (Intel Optane) & emerging NVMs
    - 新材料 New materials

非易失性存储器 Nonvolatile Memories

- **非易失性存储器用途 Uses for Nonvolatile Memories**
  - 在ROM中的固件程序（BIOS、磁盘控制器、网卡、图形加速器、安全子系统…） Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,…)
  - 固态磁盘（替代旋转磁盘） Solid state disks (replacing rotating disks)
  - 磁盘缓存 Disk caches

固态盘（SSD） Solid State Disks (SSDs)



- **页大小：512B至4KB，块大小：32至128页 Pages: 512B to 4KB, Blocks: 32 to 128 pages**
- **以页面为单位读取/写入数据 Data read/written in units of pages.**
- **只有擦除页面块后才能写入页面 Page can be written only after its block has been erased**
- **在大约100000次重复写入之后，一个块会磨损 A block wears out after about 100,000 repeated writes.**

SSD性能特点 SSD Performance Characteristics

- **三星产品测试 Benchmark of Samsung 940 EVO Plus**

  https://ssd.userbenchmark.com/SpeedTest/711305/Samsung-SSD-970-EVO-Plus-250GB

顺序读吞吐量	**Sequential read throughput**	**2,126 MB/s**
顺序写吞吐量	**Sequential write tput**	**1,880 MB/s**
随机读吞吐量	**Random read throughput**	**140 MB/s**
随机写吞吐量	**Random write tput**	**59 MB/s**

- **顺序访问比随机访问更快 Sequential access faster than random access**
  - 存储器层次结构中的公共主题 Common theme in the memory hierarchy
- **随机写入有些慢 Random writes are somewhat slower**
  - 擦除块需要很长时间（~1毫秒） Erasing a block takes a long time (~1 ms).
  - 修改块页面需要将所有其他页面复制到新块 Modifying a block page requires all other pages to be copied to new block.
  - 闪存转换层允许在执行块写入之前积累一系列小写入 Flash translation layer allows accumulating series of small writes before doing block write.

SSD与旋转磁盘的权衡
SSD Tradeoffs vs Rotating Disks

- **优势 Advantages**
  - 无移动部件 → 速度更快、功耗更低、更坚固 No moving parts → faster, less power, more rugged

- **缺点 Disadvantages**
  - 有可能磨损 Have the potential to wear out
    - 通过闪存转换层中的“损耗均衡逻辑”缓解 Mitigated by “wear leveling logic” in flash translation layer
    - 例如三星940 EVO Plus保证在磨损前每字节可以写入600次 E.g. Samsung 940 EVO Plus guarantees 600 writes/byte of writes before they wear out
    - 控制器迁移数据以最小化磨损程度 Controller migrates data to minimize wear level
  - 2019年，每字节的成本大约高出4倍 In 2019, about 4 times more expensive per byte
    - 而且，相对成本将继续下降 And, relative cost will keep dropping

SSD与旋转磁盘的权衡
SSD Tradeoffs vs Rotating Disks

- **应用 Applications**
  - MP3播放器、智能手机、笔记本电脑 MP3 players, smart phones, laptops
  - 在台式机和服务器中越来越常见 Increasingly common in desktops and servers

小结 Summary

- **CPU、内存和大容量存储之间的速度差距继续扩大 The speed gap between CPU, memory and mass storage continues to widen.**
- **编写良好的程序具有一种称为*局部性*的特性 Well-written programs exhibit a property called *locality*.**
- **基于缓存的内存层次结构通过利用局部性缩小了差距 Memory hierarchies based on *caching* close the gap by exploiting locality.**
- **闪存的进步超过了所有其他内存和存储技术（DRAM、SRAM、磁盘） Flash memory progress outpacing all other memory and storage technologies (DRAM, SRAM, magnetic disk)**
  - 能够三维堆叠单元 Able to stack cells in three dimensions

补充幻灯片 Supplemental slides

存储发展趋势 Storage Trends

SRAM

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/MB	2,900	320	256	100	75	60	*25*	*116*
access (ns)	150	35	15	3	2	1.5	1.3	*115*

DRAM

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/MB	880	100	30	1	0.1	0.06	0.02	*44,000*
access (ns)	200	100	70	60	50	40	20	*10*
typical size (MB)	0.256	4	16	64	2,000	8,000	16.000	*62,500*

Disk

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/GB	100,000	8,000	300	10	5	0.3	0.03	*3,333,333*
access (ms)	75	28	10	8	*5*	*3*	*3*	*25*
typical size (GB)	0.01	0.16	1	20	160	1,500	3,000	*300,000*

CPU时钟频率
CPU Clock Rates

计算机历史中的拐点，当设计师撞上“功率墙”时

Inflection point in computer history when designers hit the “Power Wall”

	1985	1990	1995	2003	2005	2010	2015	*2015:1985*
CPU	80286	80386	Pentium	P-4	Core 2	Core i7(n)	Core i7(h)	
Clock rate (MHz)	6	20	150	3,300	2,000	2,500	3,000	500
Cycle time (ns)	166	50	6	0.30	0.50	0.4	0.33	500
Cores	1	1	1	1	2	4	4	4
Effective cycle time (ns)	166	50	6	0.30	0.25	0.10	0.08	2,075

(n) Nehalem processor
(h) Haswell processor

Cache存储器

100076202： 计算机系统导论

任课教师：

宿红毅　张艳　黎有琦　颜珂

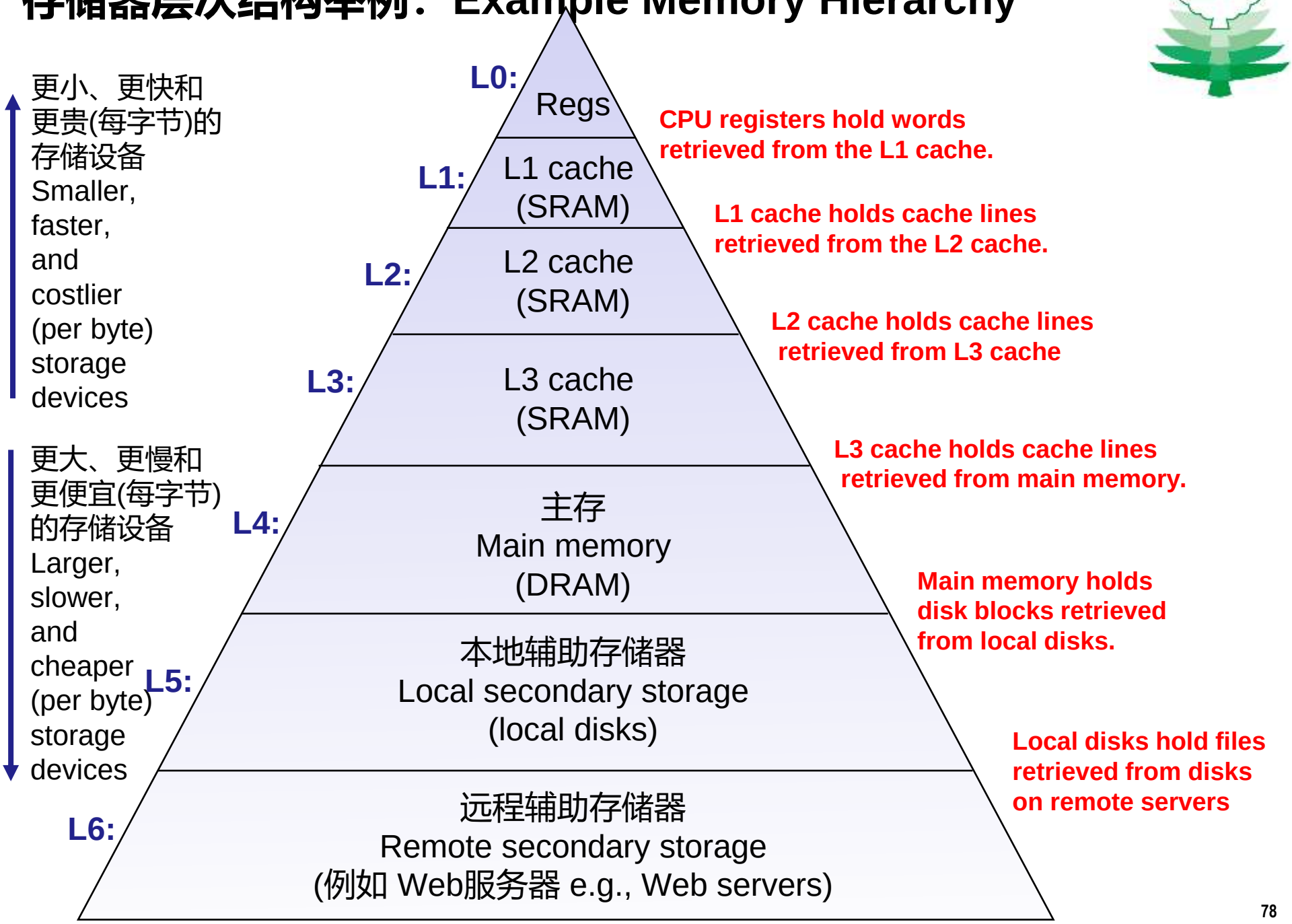
原作者：

Randal E. **Bryant and** David R. O'Hallaron

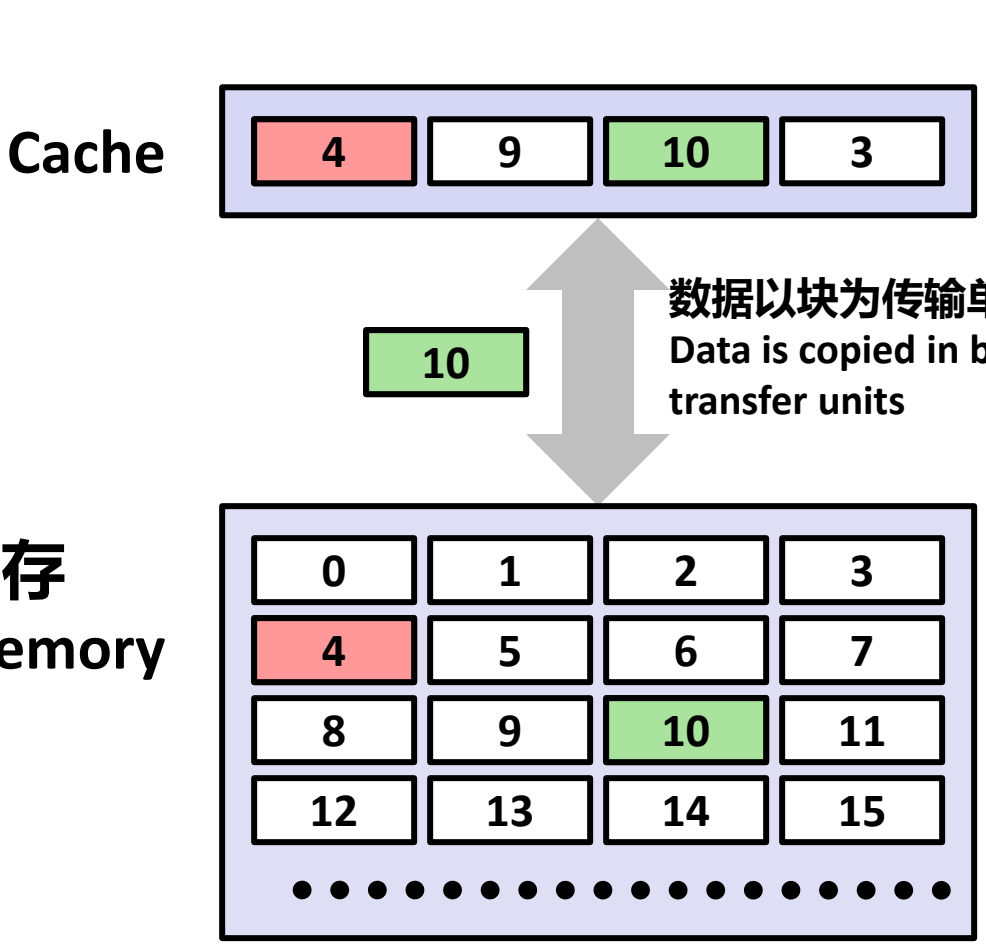
主要内容

- **Cache存储器结构和操作 Cache memory organization and operation**
- **Cache对性能的影响 Performance impact of caches**
  - 存储器性能山丘 The memory mountain
  - 循环变换提升空间局部性 Rearranging loops to improve spatial locality
  - 使用blocking提升时间局部性 Using blocking to improve temporal locality

存储器层次结构举例：Example Memory Hierarchy



Cache基本概念 General Cache Concept



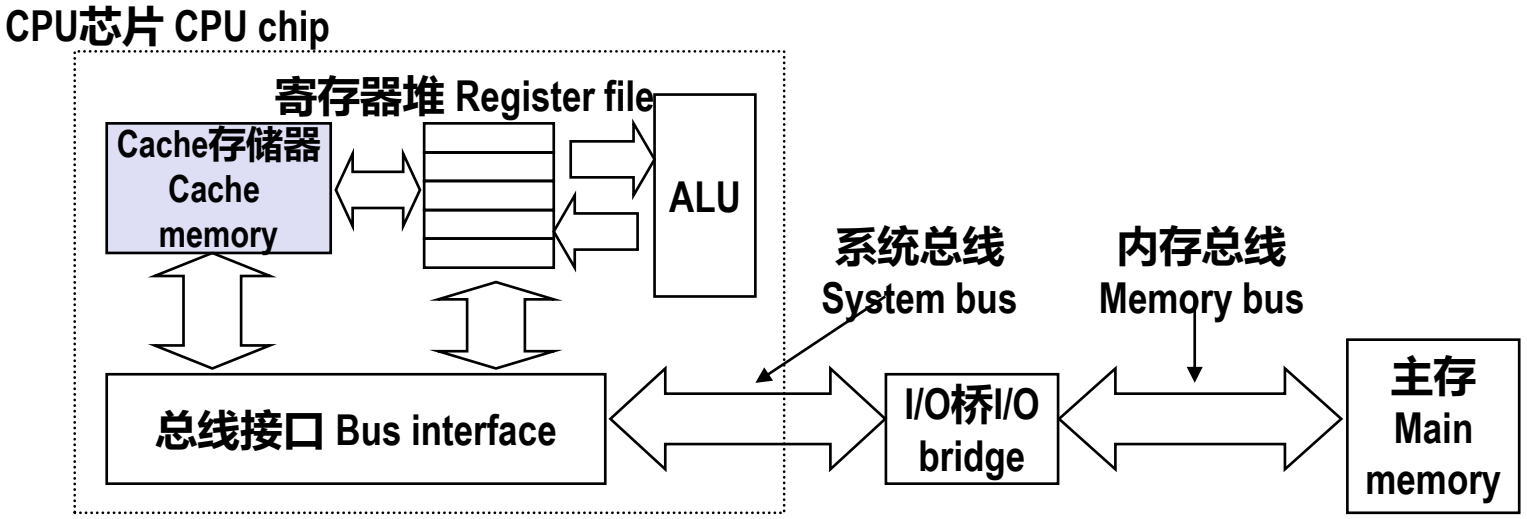
更小、更快、更加昂贵的存储器缓存一组块
Smaller, faster, more expensive memory caches a subset of the blocks

数据以块为传输单位进行复制
Data is copied in block-sized transfer units

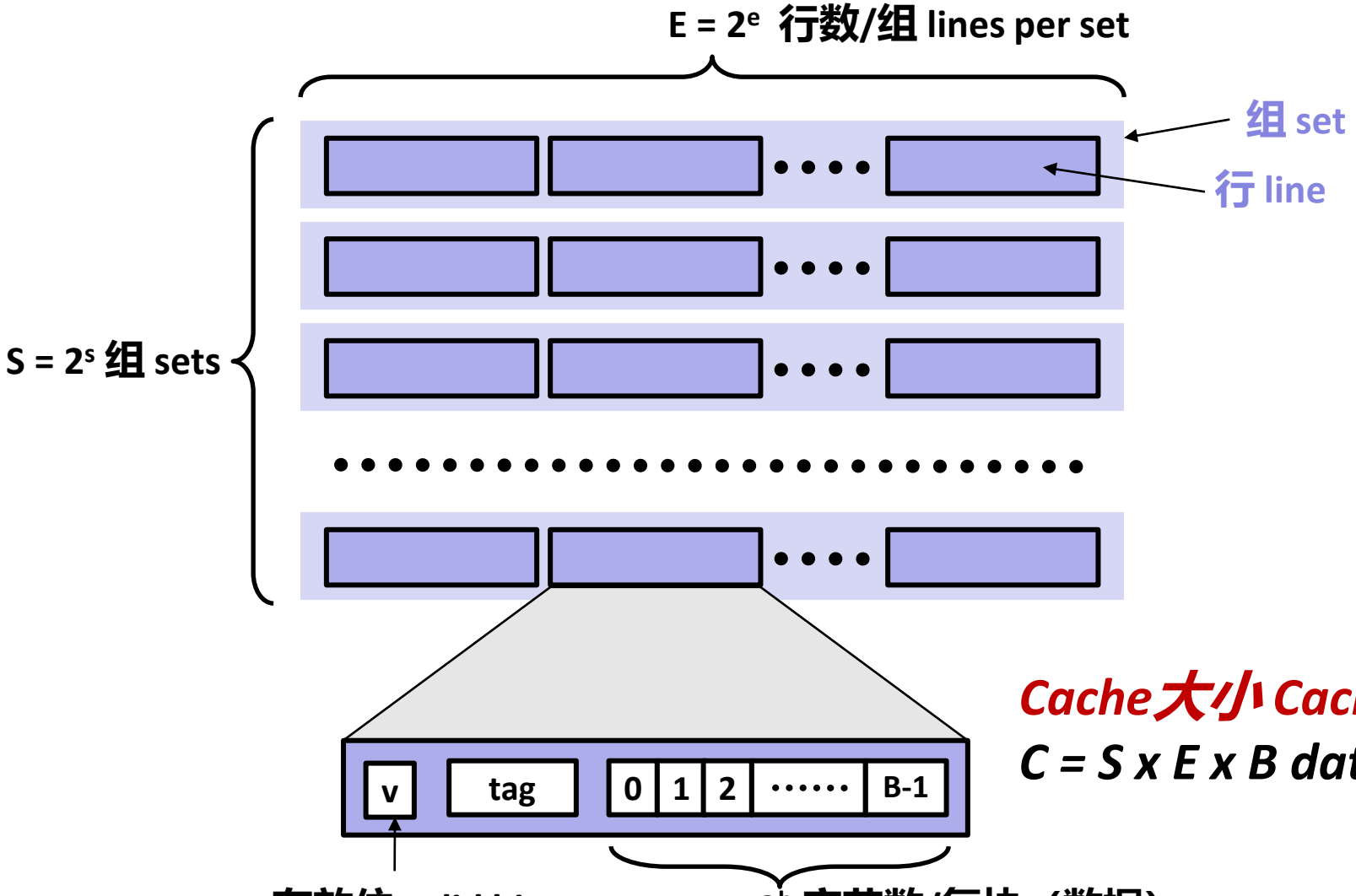
更大、更慢、更加廉价的存储器看成分区为“块”
Larger, slower, cheaper memory viewed as partitioned into “blocks”

Cache存储器 Cache Memories

- **Cache存储器是由硬件自动管理的容量较小速度较快的SRAM存储器**
  Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - 存储被频繁访问的主存块 Hold frequently accessed blocks of main memory
- **CPU首先在Cache中查找数据 CPU looks first for data in cache**
- **典型系统结构 Typical system structure:**



Cache一般组织结构（S，E，B）
General Cache Organization (S, E, B)

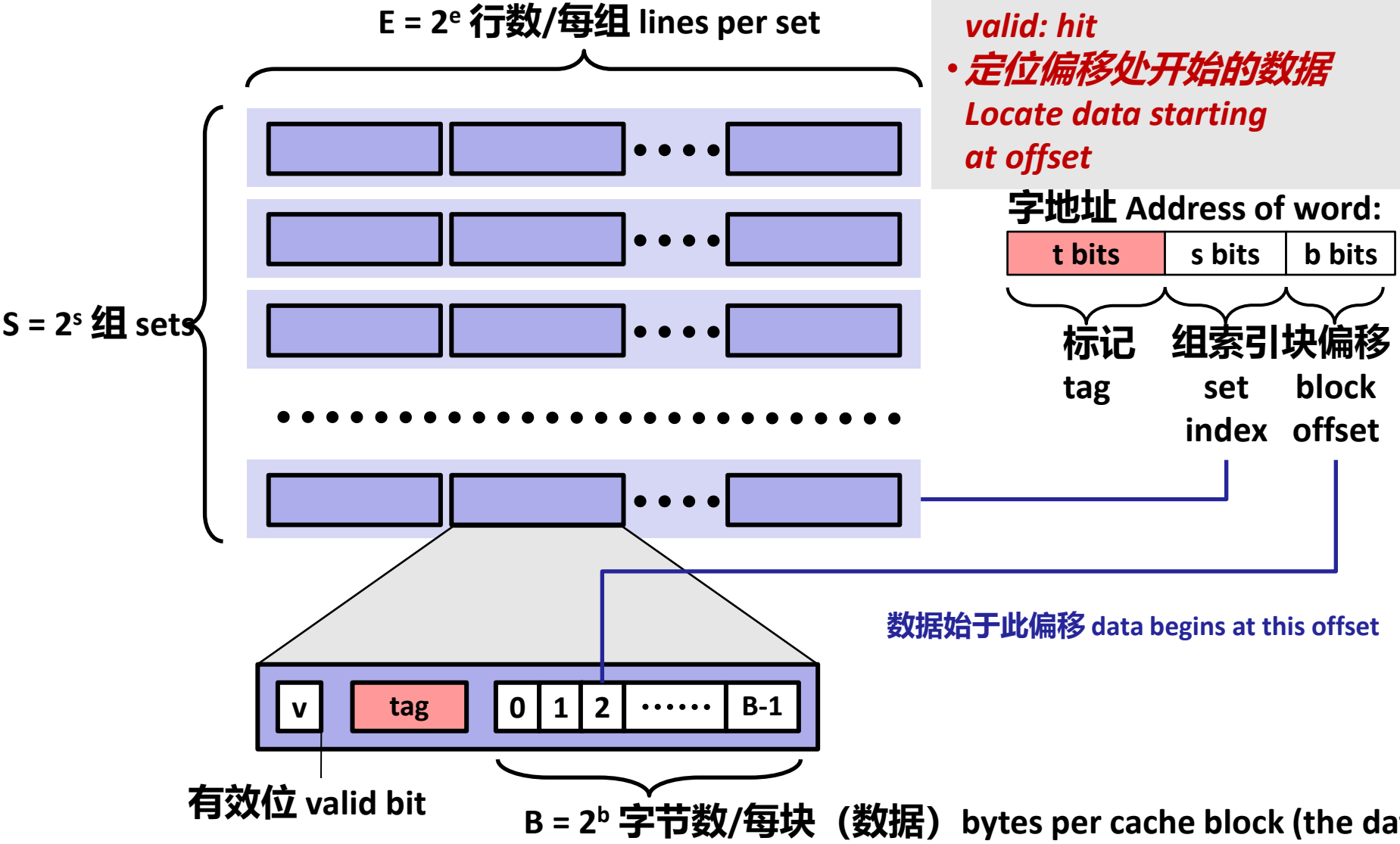


Cache大小 Cache size:

C = S x E x B data bytes

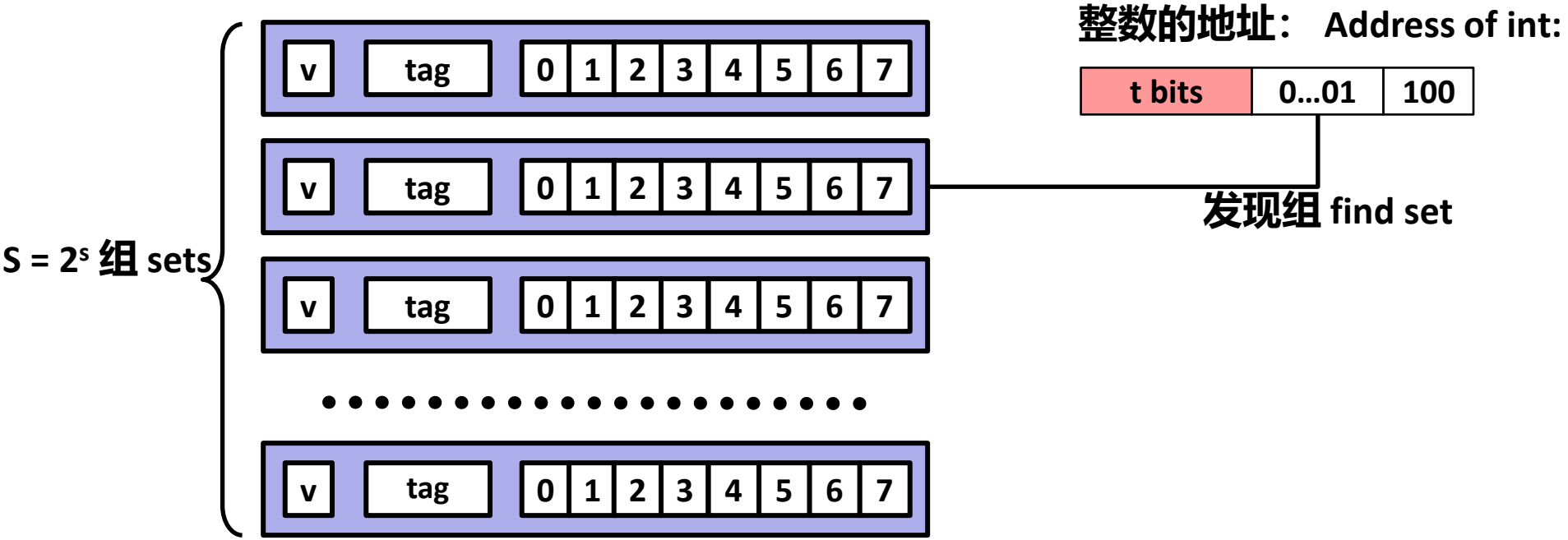
Cache读操作 Cache Read

- *定位组 Locate set*
- *检查组内是否有匹配标记的行 Check if any line in set has matching tag*
- *是+行有效：命中 Yes + line valid: hit*
- *定位偏移处开始的数据 Locate data starting at offset*



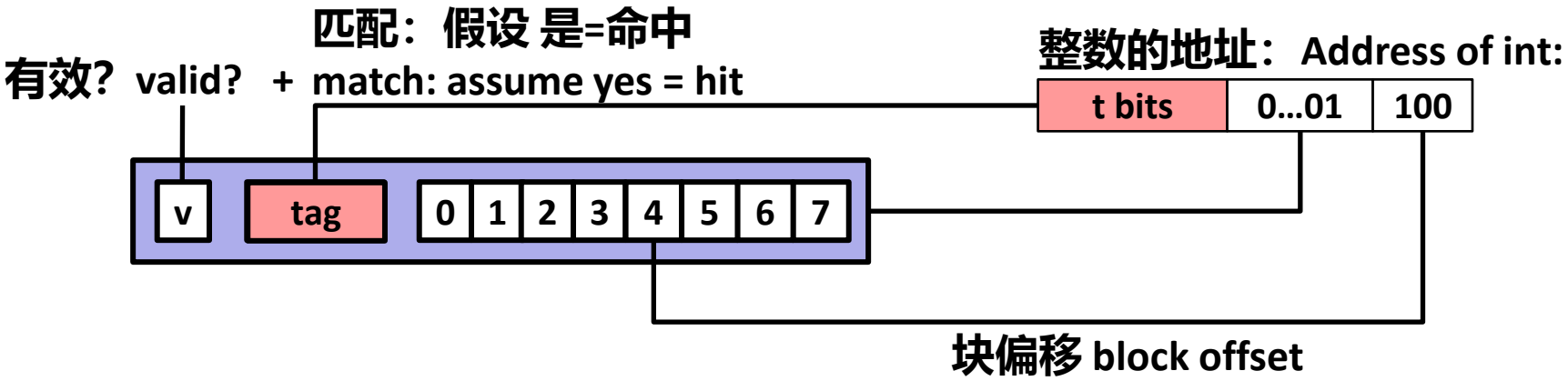
示例：直接映射Cache（E=1）
Example: Direct Mapped Cache (E = 1)

直接映射：每组一行 Direct mapped: One line per set
假设：每个Cache块大小为8个字节 Assume: cache block size 8 bytes



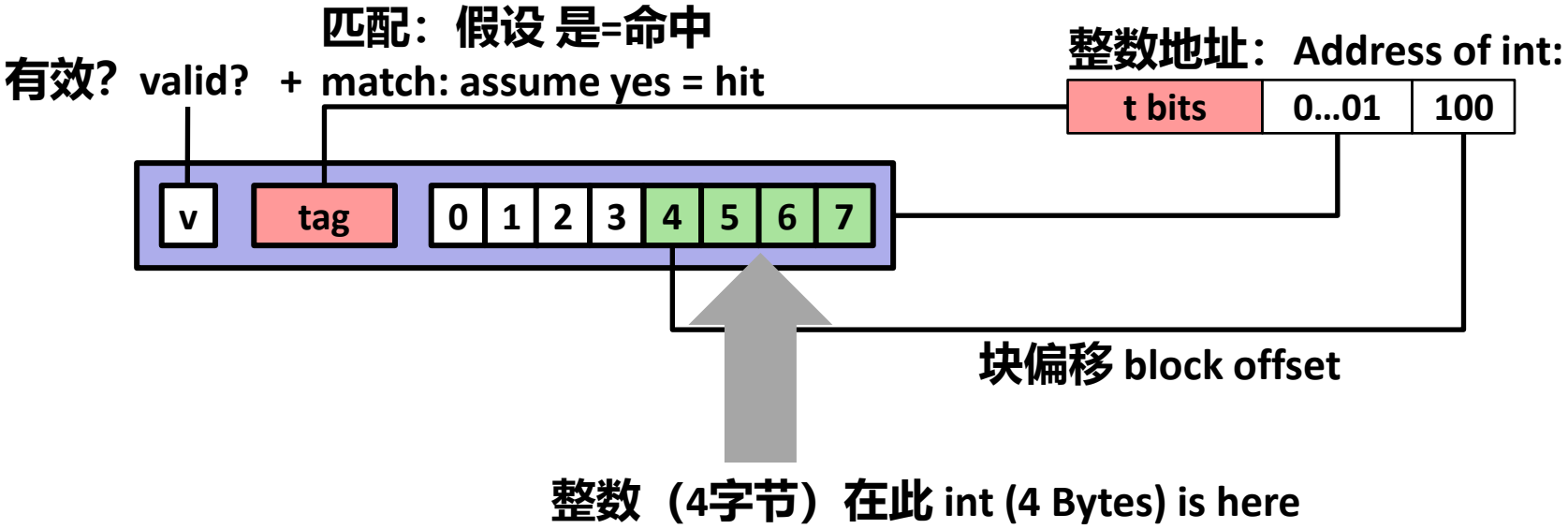
示例：直相联映射Cache（E=1）
Example: Direct Mapped Cache (E = 1)

直接映射：每组一行 Direct mapped: One line per set
假设：每个Cache块大小为8个字节 Assume: cache block size 8 bytes



示例：直接映射Cache（E=1）
Example: Direct Mapped Cache (E = 1)

直接映射：每组一行 Direct mapped: One line per set
假设：每个Cache块大小为8个字节 Assume: cache block size 8 bytes



如果标记不匹配，则驱逐旧的行并进行替换
If tag doesn't match: old line is evicted and replaced

直接映射Cache模拟 Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 字节（4位地址） M=16 bytes (4-bit addresses),
B=2 字节/块 B=2 bytes/block,
S=4 组 S=4 sets,
E=1 块/组 E=1 Blocks/set

地址轨迹（读，每次读一个字节）
Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	不命中 miss
1	**[0$\underline{00}$1$_2$],**	命中 hit
7	**[0$\underline{11}$1$_2$],**	不命中 miss
8	**[1$\underline{00}$0$_2$],**	不命中 miss
0	**[0$\underline{00}$0$_2$]**	不命中 miss

	有效 v	标记 Tag	块 Block
组0 Set 0	1	0	M[0-1]
组1 Set 1			
组2 Set 2			
组3 Set 3	1	0	M[6-7]

E路组相联Cache（E=2）

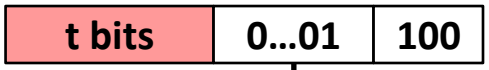
Cache E-way Set Associative Cache (Here: E = 2)

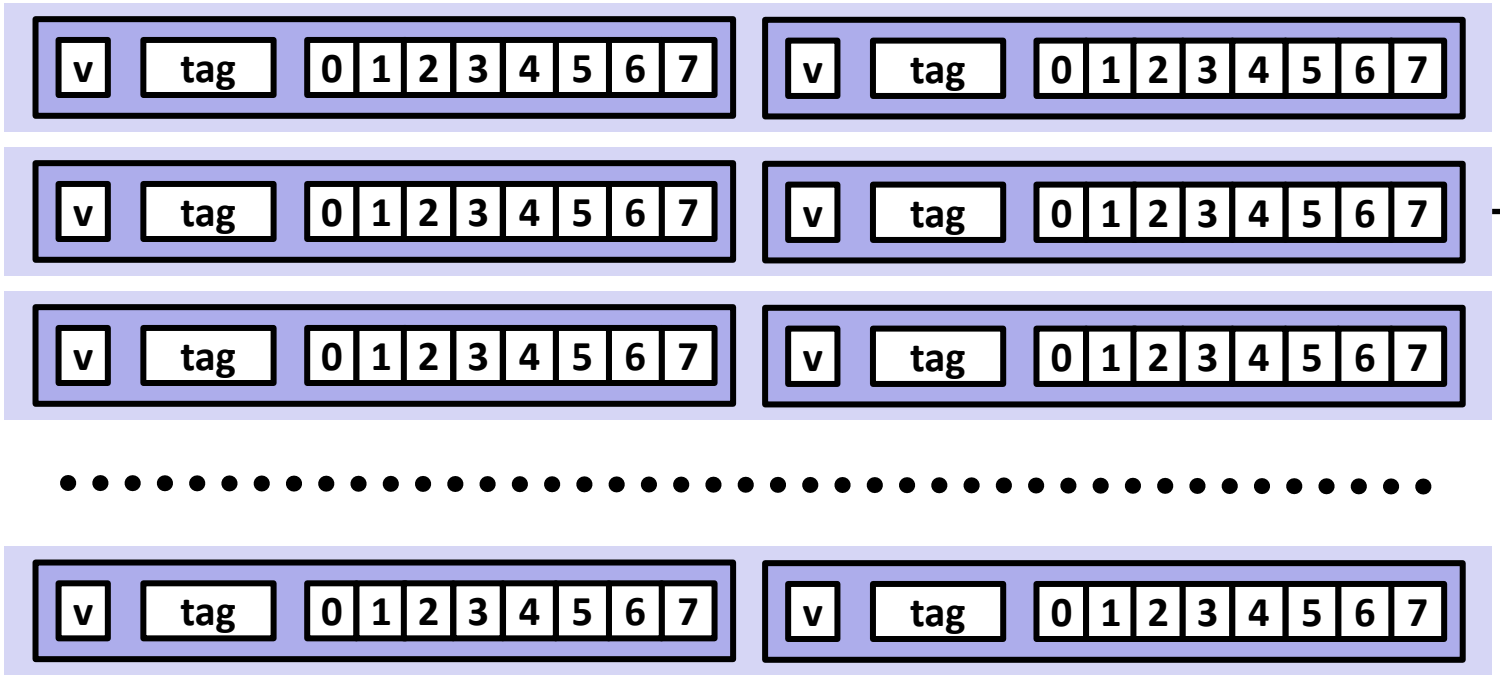
E=2：每组2行 E = 2: Two lines per set

假设：Cache块大小为8字节 Assume: cache block size 8 bytes

短整数地址：

Address of short int:

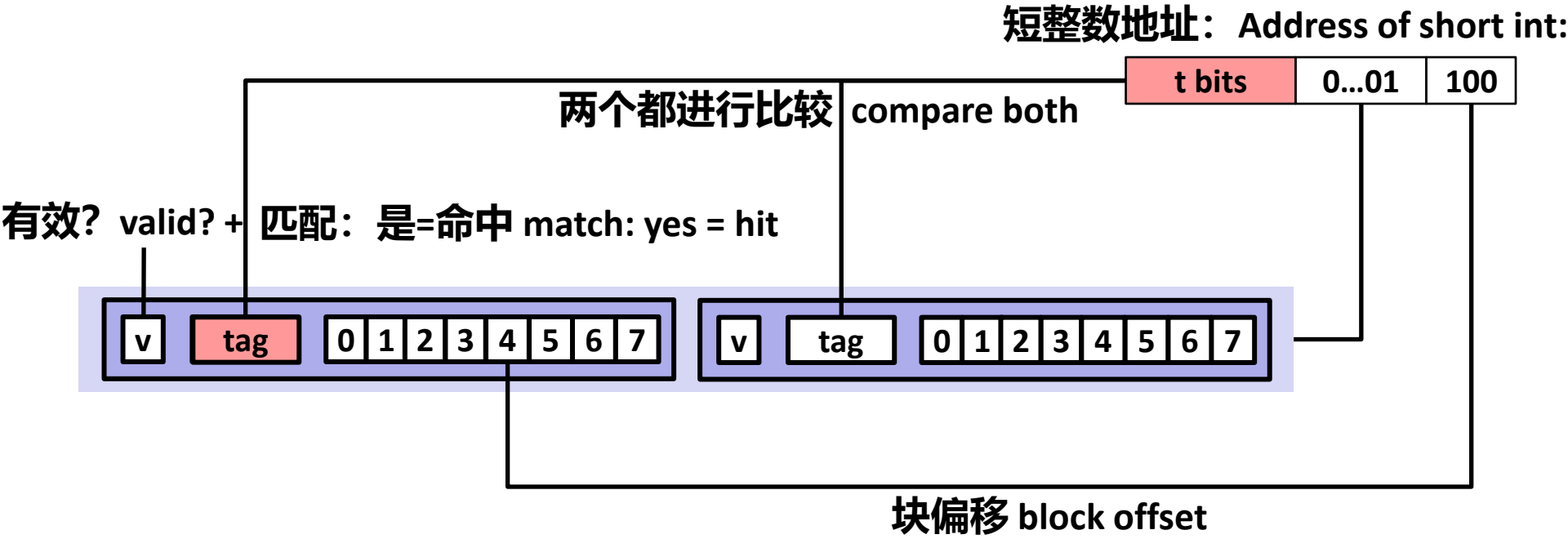




E路组相联Cache（E=2）
E-way Set Associative Cache (Here: E = 2)

E=2 每组2行： E = 2: Two lines per set

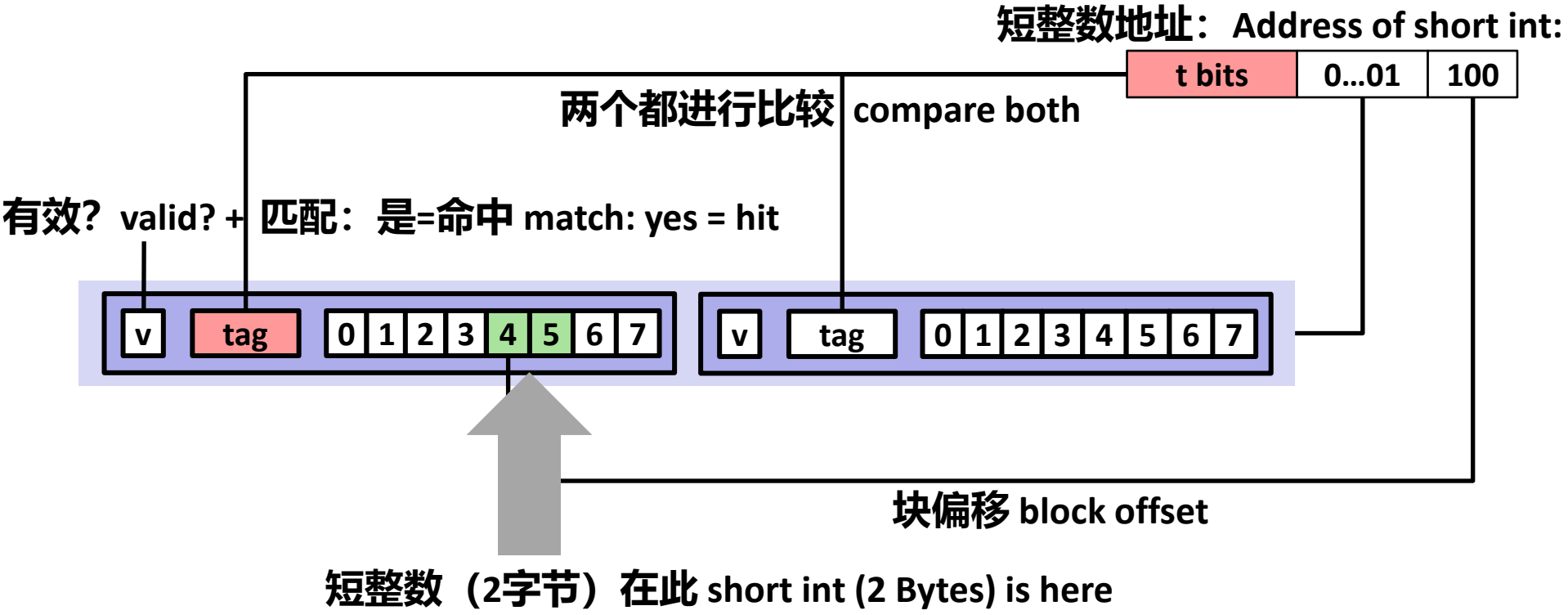
假设： Cache块大小为8字节 Assume: cache block size 8 bytes



E路组相联Cache（E=2）
E-way Set Associative Cache (Here: E = 2)

E=2：每组2行： E = 2: Two lines per set

假设： Cache块大小为8字节 Assume: cache block size 8 bytes



没有匹配 No match:

- 选中相应组中的一行进行驱逐替换 One line in set is selected for eviction and replacement
- 替换策略：随机，最近最少使用（LRU）。。。 Replacement policies: random, least recently used (LRU), …

2路组相联Cache模拟
2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 字节地址 M=16 byte addresses,
B=2 字节/块 B=2 bytes/block,
S=2 组 S=2 sets,
E=2 块/组 E=2 blocks/set
地址序列（读，每次读一个字节）：
Address trace (reads, one byte per read):

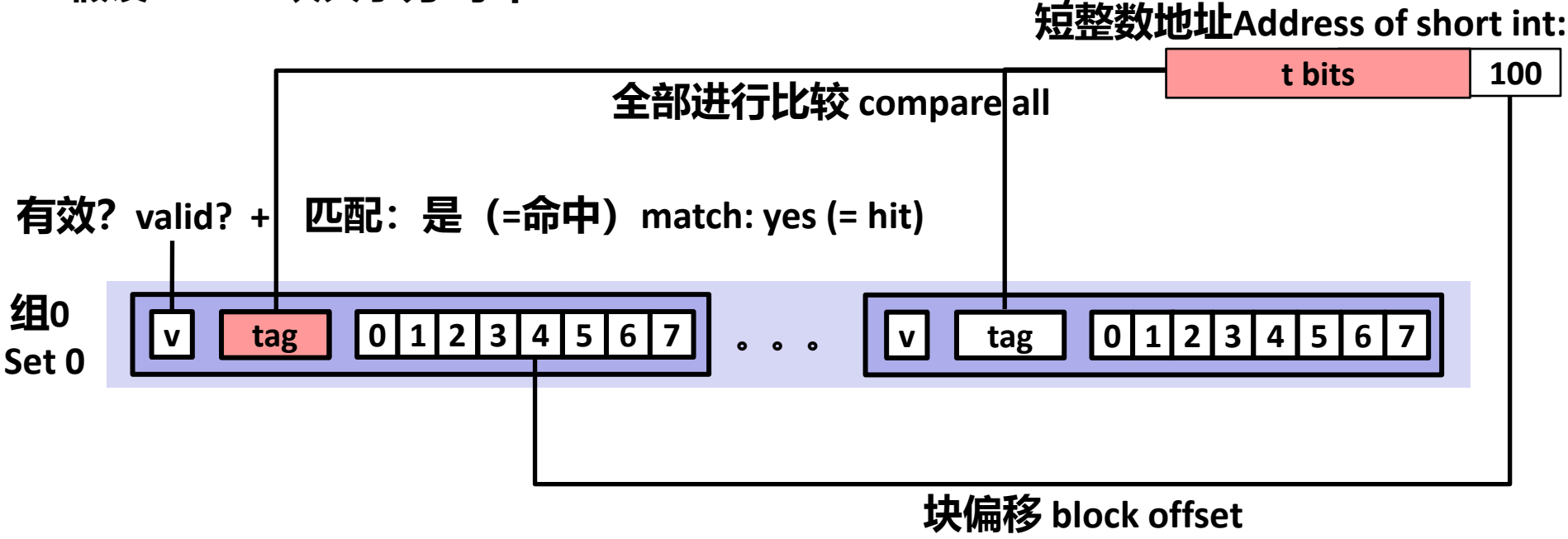
0	**[0000$_2$],**	不命中 miss
1	**[0001$_2$],**	命中 hit
7	**[0111$_2$],**	不命中 miss
8	**[1000$_2$],**	不命中 miss
0	**[0000$_2$]**	命中 hit

	有效 v	标记 Tag	块 Block
组0 Set 0	1	00	M[0-1]
	1	10	M[8-9]
组1 Set 1	1	01	M[6-7]
	0		

全相联Cache（S=1只有一组）
Fully Associative Cache (S = 1)

S = 1: 只有一个组0 S = 1: only set 0

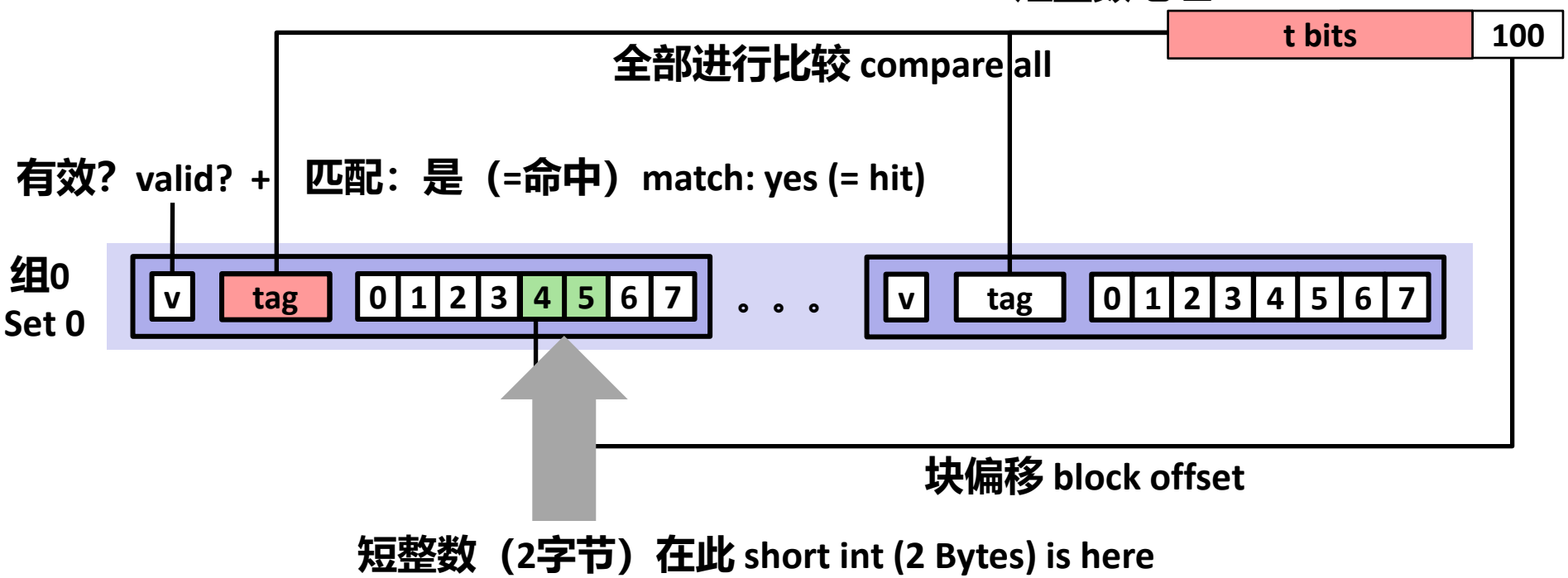
假设： cache块大小为8字节 Assume: cache block size B=8 bytes



全相联Cache（S=1只有一组）
Fully Associative Cache (S = 1)

S = 1： 只有一个组0 S = 1: Only set 0

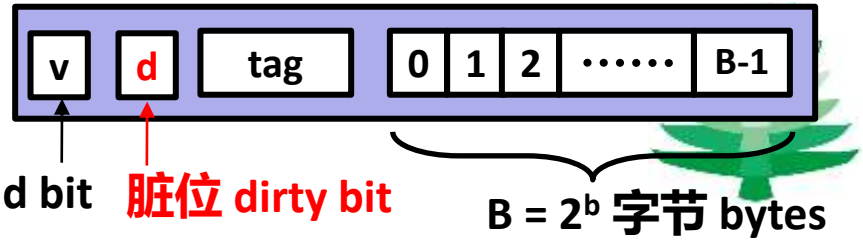
假设： cache块大小为8字节 Assume: cache block size B=8 bytes



没有匹配或无效（不命中） No match or not valid (= miss):

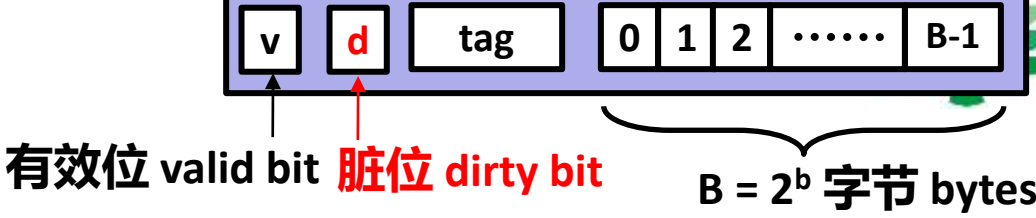
- 组0中的一行进行驱逐替换 One line in set 0 is for eviction and replacement
- 替换策略：随机，最近最少使用（LRU）。。。 Replacement policies: random, least recently used (LRU), …

Cache写操作
What about writes?



- **多数据副本 Multiple copies of data exist:**
  - L1、L2、L3、主存、磁盘 L1, L2, L3, Main Memory, Disk
- **写命中时如何处理？ What to do on a write-hit?**
  - 写直达（立即写入内存）Write-through (write immediately to memory)
  - 写回（推迟到行替换时写到内存）Write-back (defer write to memory until replacement of line)
    - 需要脏比特位标识（行与内存是否相同） Need a dirty bit (line different from memory or not)
- **写不命中时如何处理？ What to do on a write-miss?**
  - 写分配（装载进Cache后进行更新）Write-allocate (load into cache, update line in cache)
    - 如果后续位置还有写时比较好 Good if more writes to the location follow
  - 非写分配（直接写入内存，不装载进cache）No-write-allocate (writes straight to memory, does not load into cache)
- **通常策略 Typical**
  - 写直达 + 非写分配 Write-through + No-write-allocate
  - **写回 + 写分配 Write-back + Write-allocate**

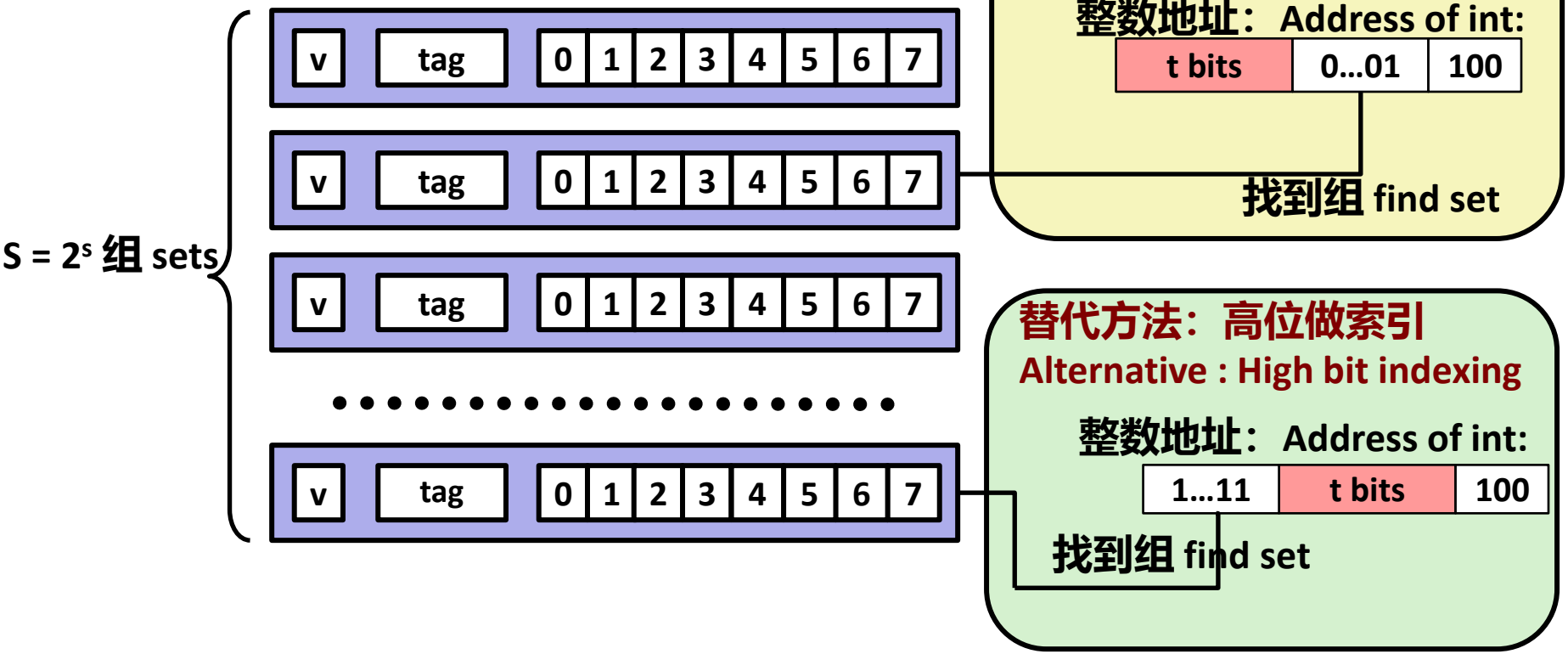
实践写回+写分配
Practical Write-back Write-allocate



- **发出对地址X的写操作 A write to address X is issued**
- **如果命中 If it is a hit**
  - 更新块的内容 Update the contents of block
  - 设置脏位为1（该位是指示而且仅在驱逐该块时清除该位） Set dirty bit to 1 (bit is sticky and only cleared on eviction)
- **如果不命中 If it is a miss**
  - 从内存取块（每次读不命中） Fetch block from memory (per a read miss)
  - 执行写操作(每次写命中) The perform the write operations (per a write hit)
- **如果驱逐一行而且脏位设置为1 If a line is evicted and dirty bit is set to 1**
  - 整个2^b 字节的块都写回到内存 The entire block of 2^b bytes are written back to memory
  - 脏位清除（设置为0） Dirty bit is cleared (set to 0)
  - 行由新的内容替换 Line is replaced by new contents

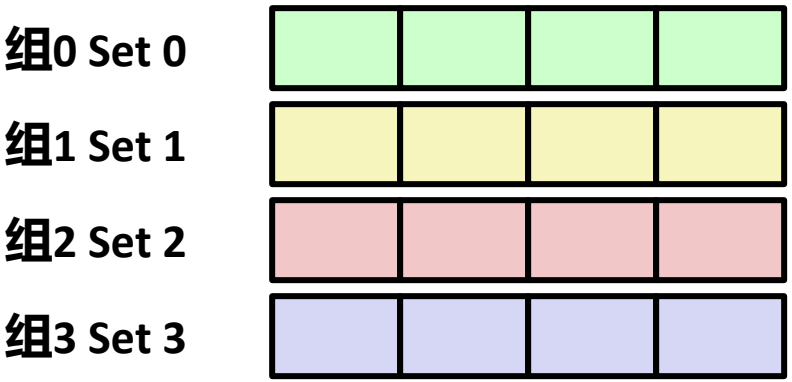
为何使用中间位做索引?
Why Index Using Middle Bits?

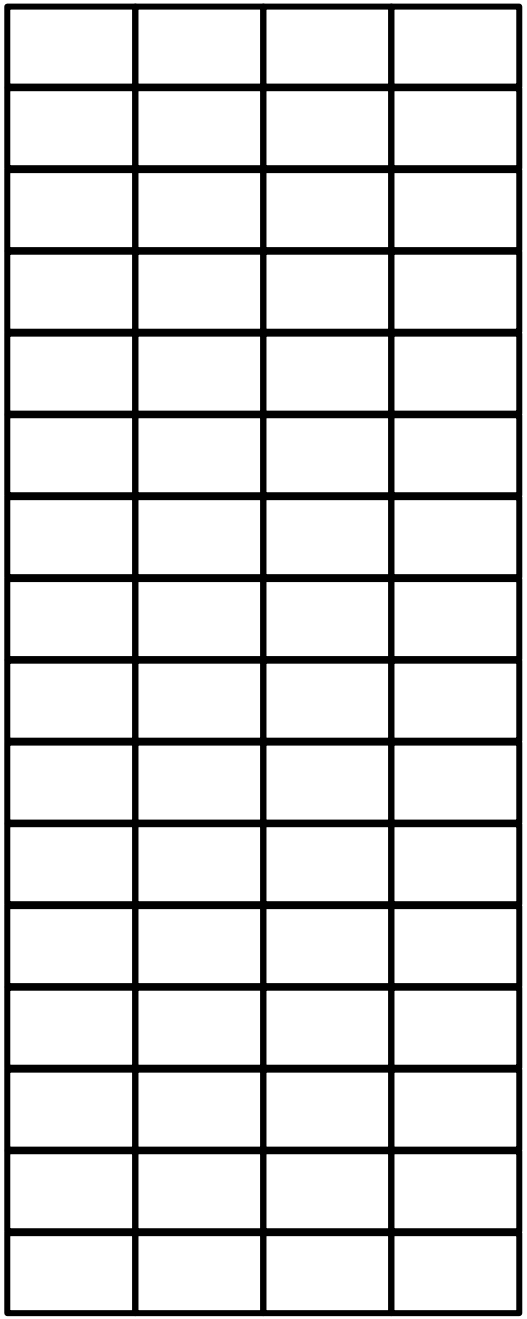
直接映射：每组一行 Direct mapped: One line per set
假设：cache块大小8字节 Assume: cache block size 8



索引方法说明 Illustration of Indexing Approaches

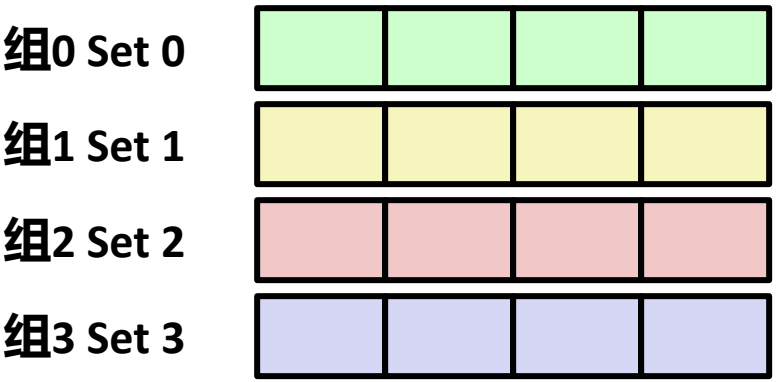
- 64字节内存 64-byte memory
  - 6位地址 6-bit addresses
- 16字节直接映射cache 16 byte, direct-mapped cache
- 块大小为4（因此分成4组）Block size = 4. (Thus, 4 sets; why?)
- 2位标记、2位索引、2位偏移 2 bits tag, 2 bits index, 2 bits offset

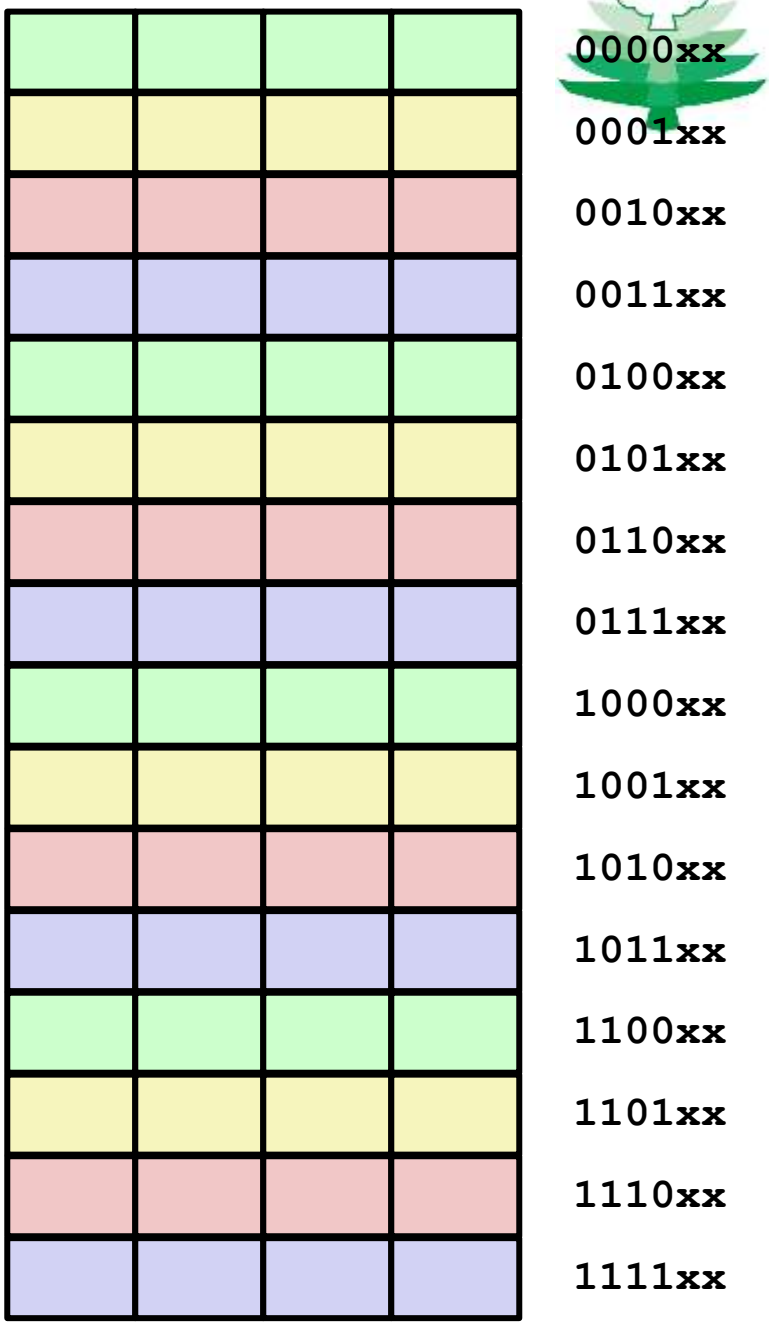




中间位做索引 Middle Bit Indexing

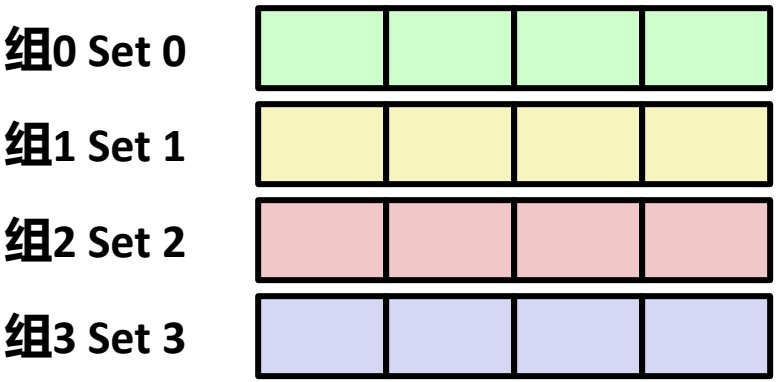
- 地址形式为 TTSSBB Addresses of form TTSSBB
  - TT 标记位 Tag bits
  - SS 组索引位 Set index bits
  - BB 偏移位 Offset bits
- 很好地利用了空间局部性 Makes good use of spatial locality

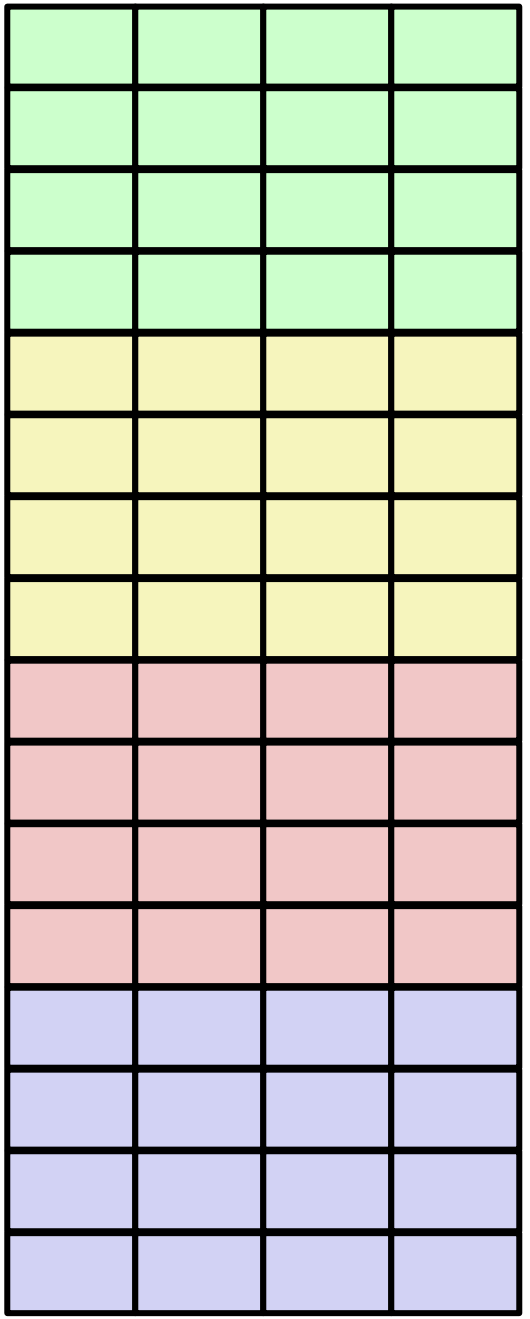




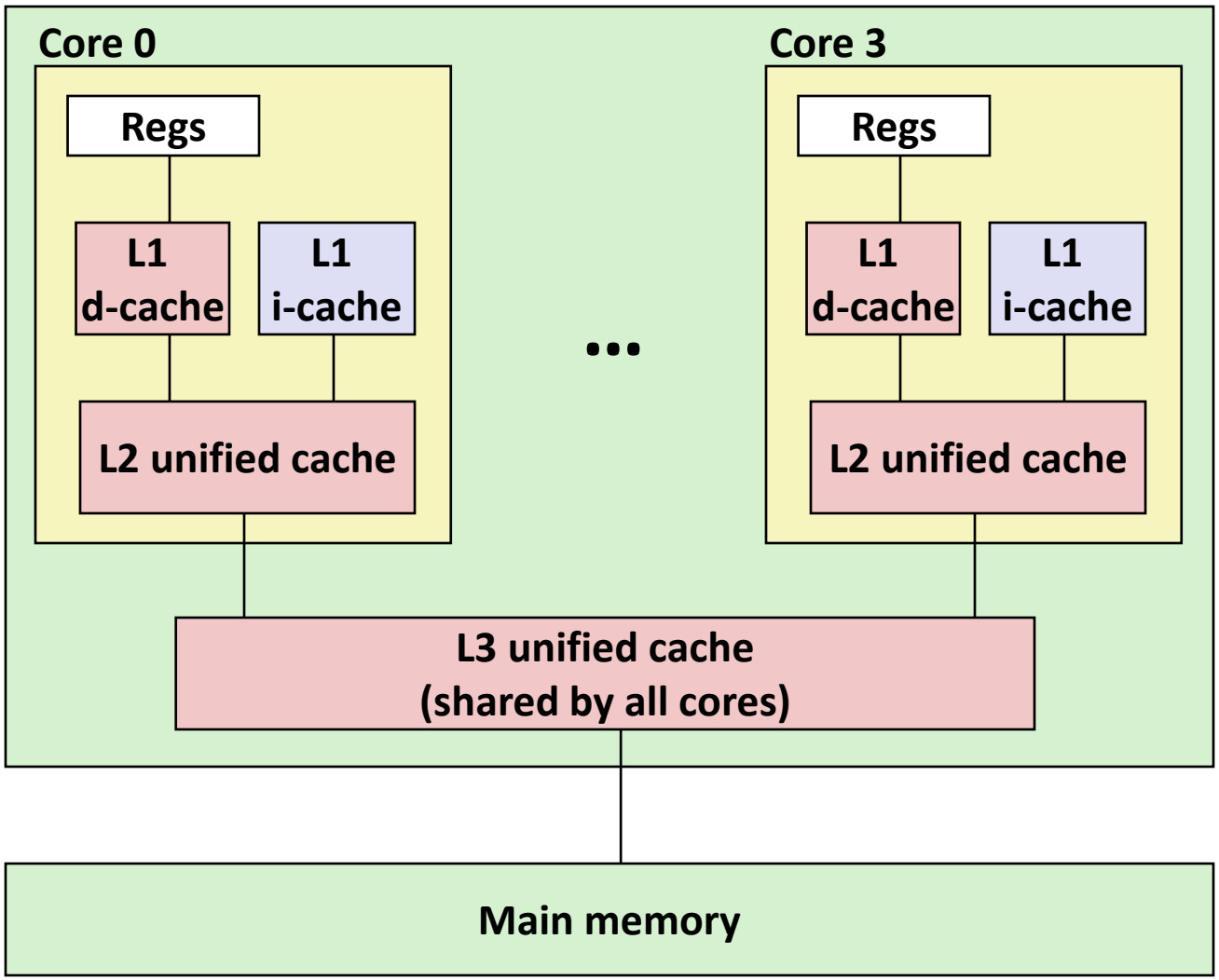
高位做索引 High Bit Indexing

- 地址形式为SSTTBB Addresses of form SSTTBB
  - SS 组索引位 Set index bits
  - TT 标记位 Tag bits
  - BB 偏移位 Offset bits
- 程序如果有很高的空间局部性会产生很多冲突 Program with high spatial locality would generate lots of conflicts





Intel Core i7 Cache层次结构
Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
指令和数据cache
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
统一cache
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
统一cache
8 MB, 16-way,
Access: 40-75 cycles

块大小：所有cache均为64字节 Block size: 64 bytes for all caches.

Cache性能评价 Cache Performance Metrics

- **不命中率 Miss Rate**
  - 内存引用没有在Cache中找到的比率（不命中次数/访问次数） Fraction of memory references not found in cache (misses / accesses)
    等于1-命中率 = 1 – hit rate
  - 通常的Cache不命中率 （百分比）Typical numbers (in percentages):
    - L1 cache为3-10% 3-10% for L1
    - L2也可能很小（例如小于1%），依赖Cache大小等 can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **命中时间 Hit Time**
  - 传递Cache中一行内容到处理器的时间 Time to deliver a line in the cache to the processor
    - 包括判断Cache是否命中的时间 includes time to determine whether the line is in the cache
  - 通常的时间 Typical numbers:
    - L1 Cache 4个时钟周期 4 clock cycle for L1
    - L2 Cache 10个时钟周期 10 clock cycles for L2
- **不命中开销 Miss Penalty**
  - 不命中需要额外的时间 Additional time required because of a miss
    - 典型情况下主存的访问周期50~200（趋势：增加！） typically 50-200 cycles for main memory (Trend: increasing!)

让我们来分析这些数字
Let's think about those numbers

- **命中和不命中之间的差距较大 Huge difference between a hit and a miss**
  - 如果只有L1 cache和主存，则会差100倍 Could be 100x, if just L1 and main memory
- **你能相信吗？99%的命中率的性能是97%的两倍 Would you believe 99% hits is twice as good as 97%?**
  - 假设 Consider:
    Cache命中需要1个周期 cache hit time of 1 cycle
    Cache不命中需要100个周期 miss penalty of 100 cycles
  - 平均访问时间 Average access time:
    97%命中：97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**
    99%命中：99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**
- **这就是为什么要用"不命中率"代替"命中率" This is why "miss rate" is used instead of "hit rate"**

编写Cache友好的代码 Writing Cache Friendly Code

- **让最常见的情况最快 Make the common case go fast**
  - 关注核心函数的内层循环 Focus on the inner loops of the core functions

- **减少内层循环的不命中率 Minimize the misses in the inner loops**
  - 重复访问（时间局部性）Repeated references to variables are good (temporal locality)
  - 步长为1的连续访问模式（空间局部性）Stride-1 reference patterns are good (spatial locality)

关键思想：我们对局部性的定性概念是通过我们对高速缓冲存储器的理解来量化的

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

主要内容

- Cache结构和操作 Cache memory organization and operation
- **Cache对性能的影响 Performance impact of caches**
  - 存储器性能山丘 The memory mountain
  - 循环变换提升空间局部性 Rearranging loops to improve spatial locality
  - 使用分块提升时间局部性 Using blocking to improve temporal locality

内存性能山丘 The Memory Mountain

- **读吞吐率Read throughput (读带宽 read bandwidth)**
  - 每秒从内存读取的字节数（MB/s） Number of bytes read from memory per second (MB/s)

- **存储山丘：读吞吐率测量为空间和时间局部性的函数**
  Memory mountain: Measured read throughput as a function of spatial and temporal locality.
  - 刻画内存系统性能的简单方法 Compact way to characterize memory system performance.

内存山测试函数 Memory Mountain Test Function

```
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

用多种elems和stride的组合调用test函数
Call `test()` with many combinations of `elems` and `stride`.

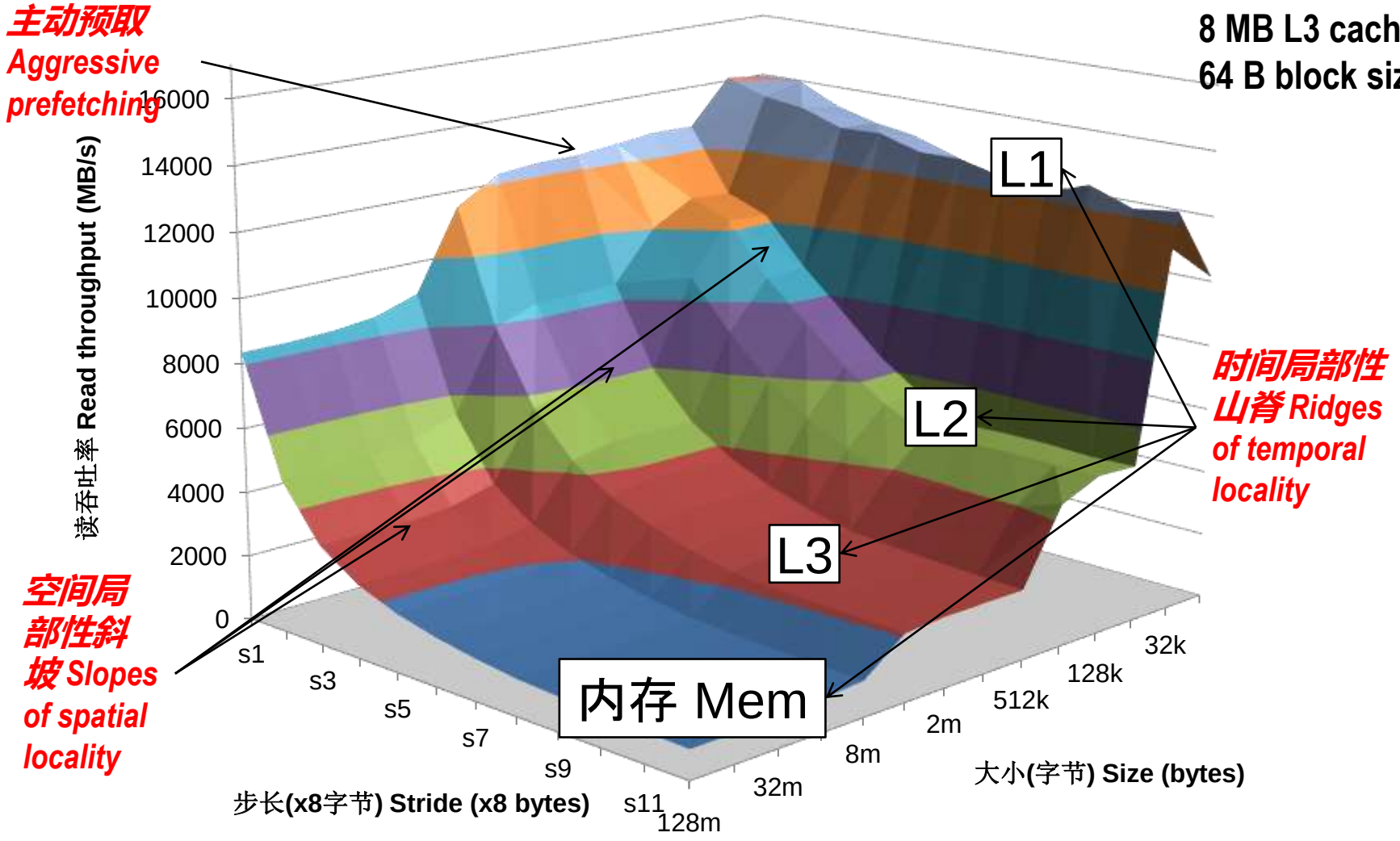
对于每个`elems`和`stride`参数:
For each `elems` and `stride`:

1.调用`test`一次来避免cache冷不命中
1. Call `test()` once to warm up the caches.

2.再次调用`test`函数并测量读吞吐率（MB/s）
2. Call `test()` again and measure the read throughput(MB/s)

内存性能山丘

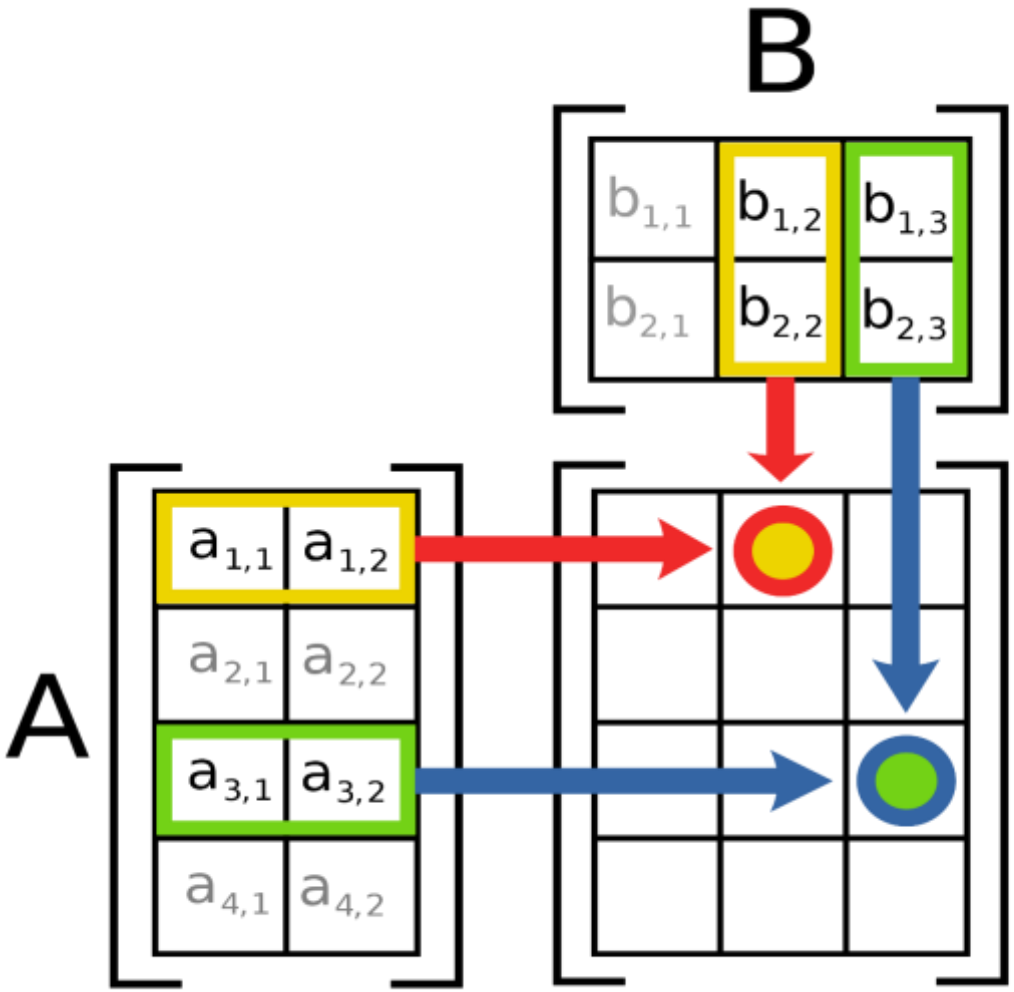
Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



主要内容

- Cache结构和操作 Cache memory organization and operation
- Cache对性能的影响 Performance impact of caches
  - 存储器性能山丘 The memory mountain
  - 循环变换提升空间局部性 Rearranging loops to improve spatial locality
  - 使用分块提升时间局部性 Using blocking to improve temporal locality

回忆矩阵乘法计算方法
Remember matrix multiplication




```
Out[i, j] =
   dot product(A[i, ..], B[..,j])
 = sum (
            a[i, 0] * b[0, j],
             a[i, 1] * b[1, j]
                    )
        //点积 dot product
```

矩阵乘法示例
Matrix Multiplication Example

- **描述 Description:**
  - N x N矩阵乘法 Multiply N x N matrices
  - 矩阵元素为双精度浮点数(8字节) Matrix elements are doubles (8 bytes)
  - 总操作时间复杂度为 $O(N^3)$ total operations
  - 每个源元素需要N次读 N reads per source element
  - 每个目的进行N个值求和 N values summed per destination
    - 但是可能会存储在寄存器中 but may be able to hold in

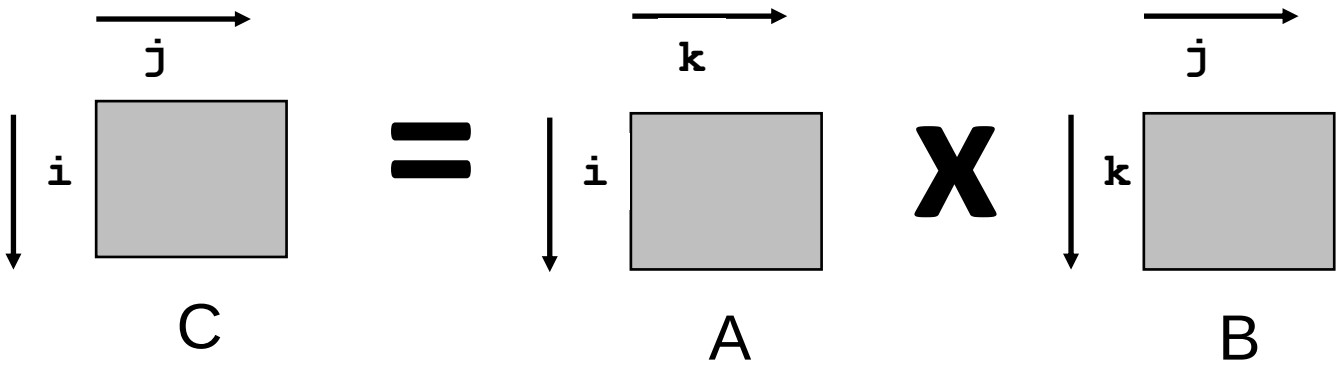
变量sum存储在寄存器中
Variable sum held in register

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c

矩阵乘法的Cache不命中率分析
Miss Rate Analysis for Matrix Multiply

- **假设 Assume:**
  - 块大小为32字节(大到装下4个双精度浮点数) Block size = 32B (big enough for four doubles)
  - 矩阵维度N非常大 Matrix dimension (N) is very large
    - 大约1/N为零 Approximate 1/N as 0.0
  - Cache不足以容纳多行数据 Cache is not even big enough to hold multiple rows
- **分析方法 Analysis Method:**
  - 内层循环的访问模式 Look at access pattern of inner loop



C语言数组的内存布局（回忆）
Layout of C Arrays in Memory (review)

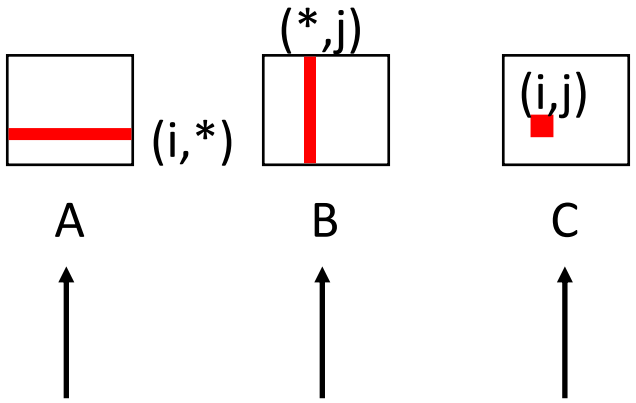
- **C语言数组按照行存储 C arrays allocated in row-major order**
  - 每行连续存储 each row in contiguous memory locations
- **访问每行的每列元素 Stepping through columns in one row:**
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - 访问连续元素 accesses successive elements
  - 如果块大小大于元素的字节数，则可以利用空间局部性 if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - 不命中率为： miss rate = sizeof(a_{ij}) / B
- **访问一列的每行元素 Stepping through rows in one column:**
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - 访问不连续元素 accesses distant elements
  - 无空间局部性 no spatial locality!
    - 不命中率为1（即100%） miss rate = 1 (i.e. 100%)

矩阵乘法（ijk）
Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c

内循环 Inner loop:



按行访问	按列访问	固定
Row-wise	Column-wise	Fixed

每次内循环迭代的不命中率 Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

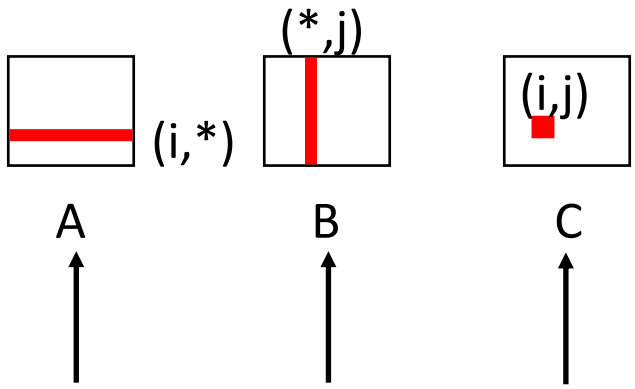
块大小=32字节（四个双精度浮点数） Block size = 32B (four doubles)

矩阵乘法（jik）
Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

matmult/mm.c

内循环 Inner loop:



每次内循环迭代的不命中率 Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

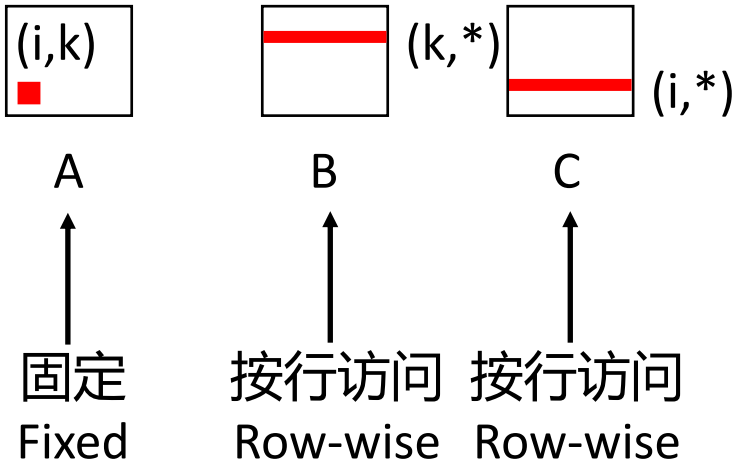
块大小=32字节（四个双精度浮点数） Block size = 32B (four doubles)

矩阵乘法（kij）
Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c

内循环 Inner loop:



每次内循环迭代的不命中率 Misses per inner loop iteration:

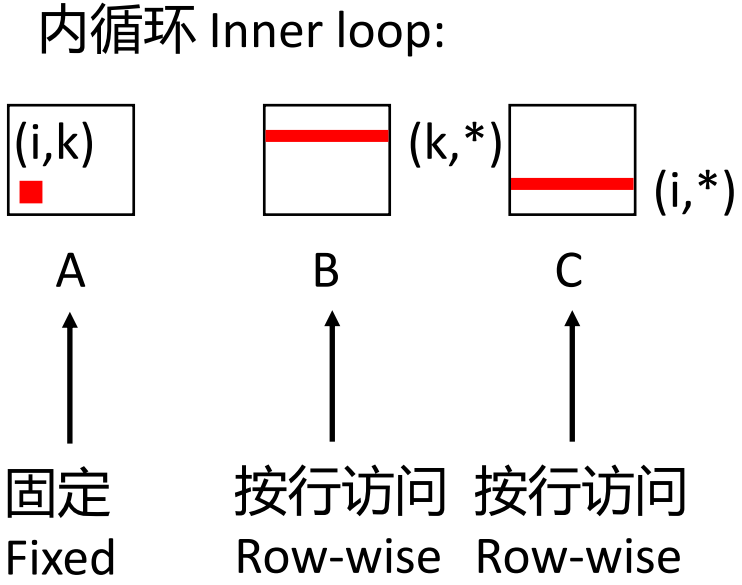
A	B	C
0.0	0.25	0.25

块大小=32字节（四个双精度浮点数） Block size = 32B (four doubles)

矩阵乘法（ikj）
Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c



每次内循环迭代的不命中率 Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

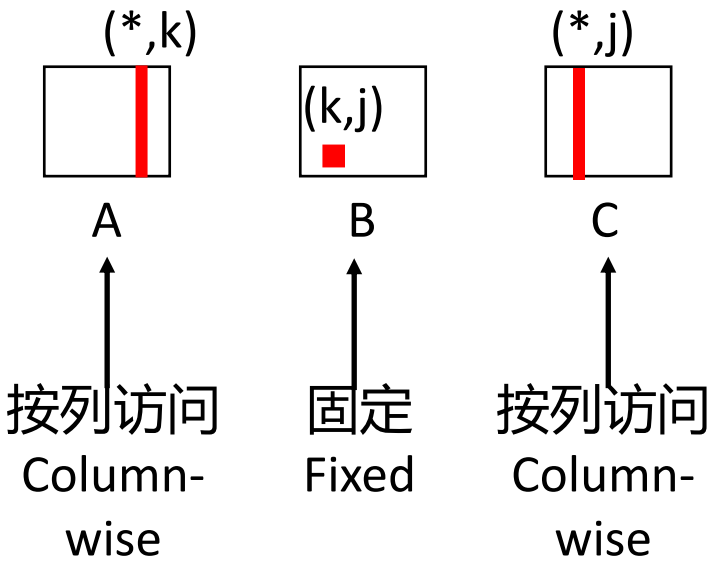
块大小=32字节（四个双精度浮点数）Block size = 32B (four doubles)

矩阵乘法（jki）
Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c

内循环 Inner loop:



每次内循环迭代的不命中率 Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

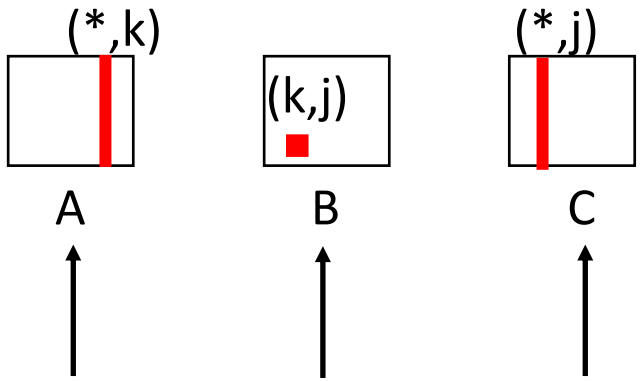
块大小=32字节（四个双精度浮点数） Block size = 32B (four doubles)

矩阵乘法（kji）
Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c

内循环 Inner loop:



每次内循环迭代的不命中率 Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

块大小=32字节（四个双精度浮点数） Block size = 32B (four doubles)

矩阵乘法总结
Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

ijk (& jik):
- **2条装载，0条存储** 2 loads, 0 stores
- 不命中/迭代 misses/iter = **1.25**

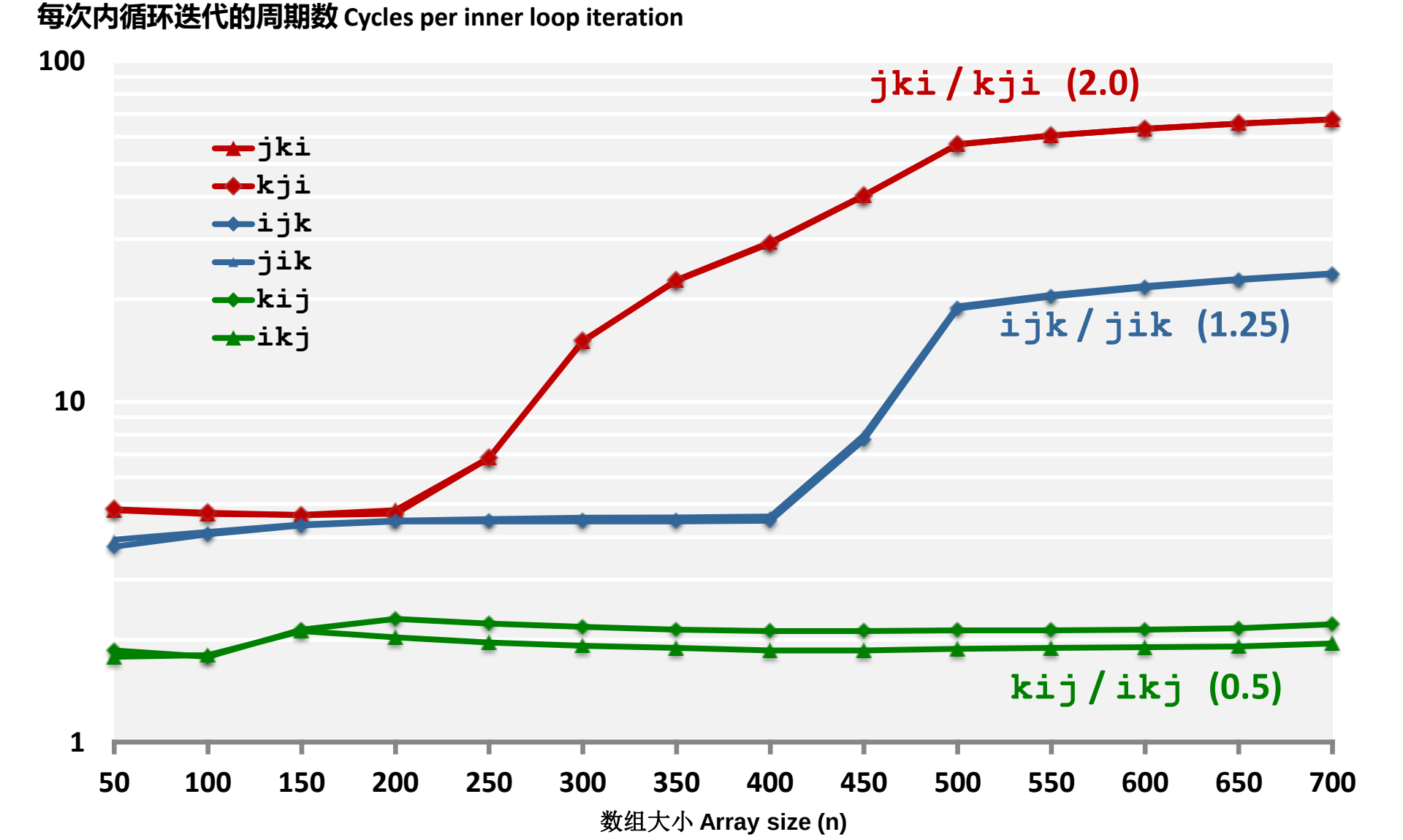
```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

kij (& ikj):
- **2条装载，1条存储** 2 loads, 1 store
- 不命中/迭代 misses/iter = **0.5**

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

jki (& kji):
- **2条装载，1条存储** 2 loads, 1 store
- 不命中/迭代 misses/iter = **2.0**

Core i7矩阵乘法性能
Core i7 Matrix Multiply Performance

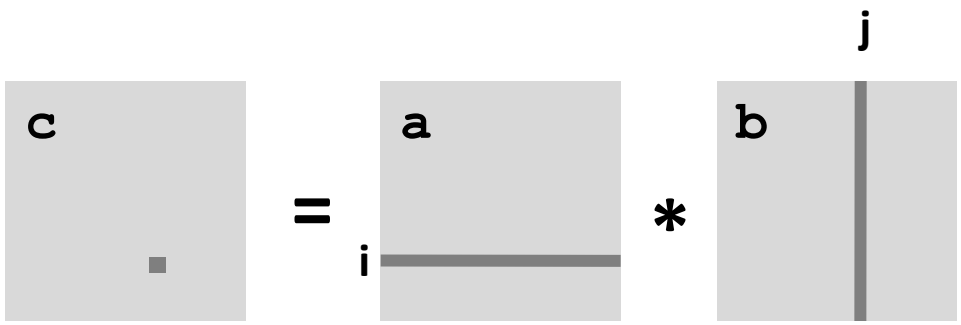


主要内容

- Cache结构和操作 Cache memory organization and operation
- Cache对性能的影响 Performance impact of caches
  - 存储性能山丘 The memory mountain
  - 循环变换提升空间局部性Rearranging loops to improve spatial locality
  - 使用分块提升时间局部性/Using blocking to improve temporal locality

示例：矩阵乘法 Example: Matrix Multiplication

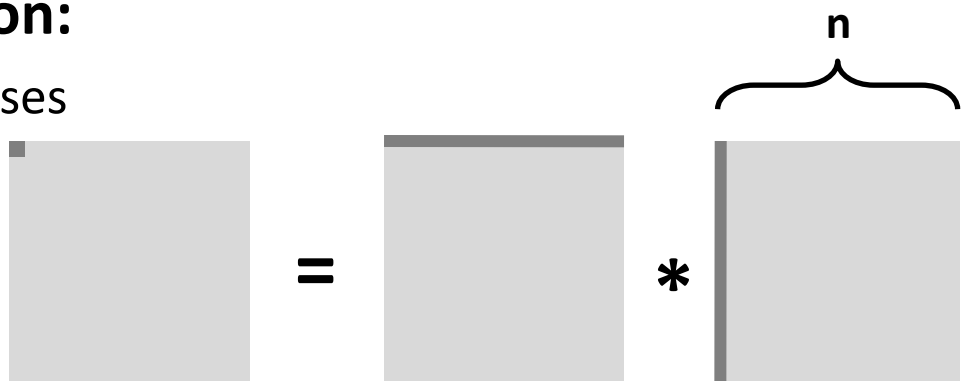
```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```



Cache不命中分析 Cache Miss Analysis

- **假设 Assume:**
  - 矩阵元素类型是双精度浮点 Matrix elements are doubles
  - Cache块大小为8个双精度浮点数 Cache block = 8 doubles
  - Cache大小C远小于n Cache size C << n (much smaller than n)

- **第一次迭代 First iteration:**
  - n/8 + n = 9n/8 不命中misses

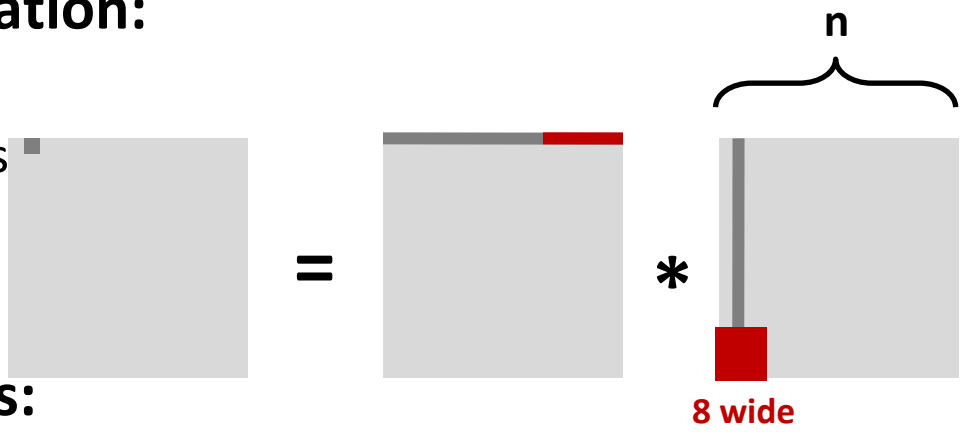


  - 后来缓存中： Afterwards in cache:
    (简图 schematic)

Cache不命中分析 Cache Miss Analysis

- **假设 Assume:**
  - 矩阵元素类型是双精度浮点 Matrix elements are doubles
  - Cache块大小为8个双精度浮点数 Cache block = 8 doubles
  - Cache大小C远小于n Cache size C << n (much smaller than n)

- **第二次迭代 Second iteration:**
  - 再次： Again:
    n/8 + n = 9n/8 不命中 mis

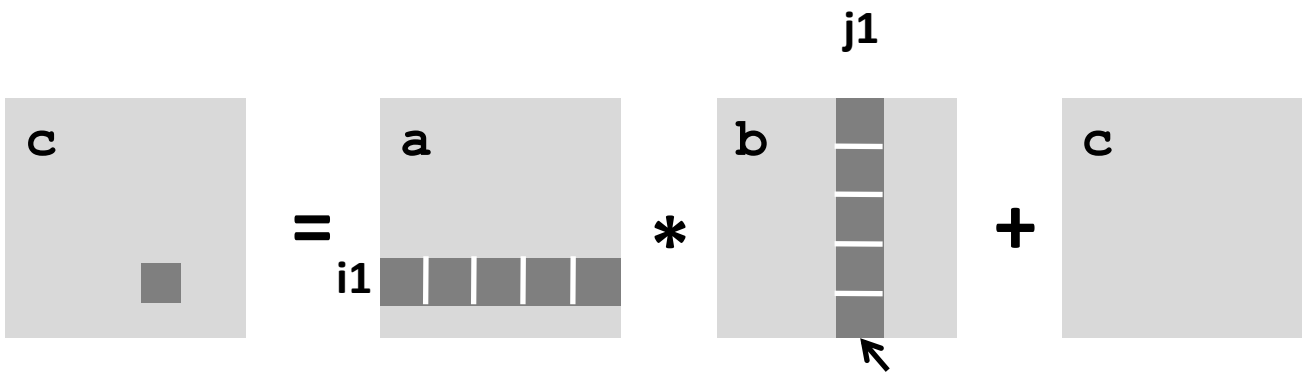


- **总计不命中 Total misses:**
  - $9n/8 * n^2 = (9/8) * n^3$

分块矩阵乘法
Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

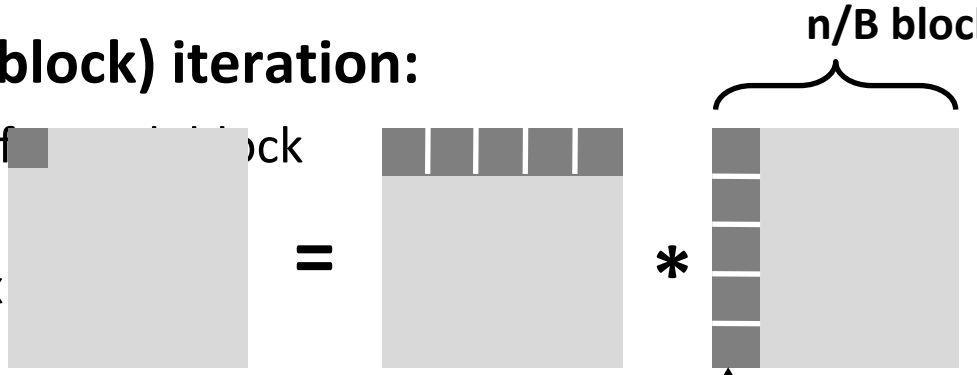
matmult/bmm.c



Cache不命中分析 Cache Miss Analysis

- **假设 Assume:**
  - Cache块大小为8个双精度浮点数 Cache block = 8 doubles
  - Cache大小C远小于n Cache size C << n (much smaller than n)
  - 三个块 ■ 适合cache Three blocks ■ fit into cache: $3B^2 < C$

- **第一次（块）迭代 First (block) iteration:**
  - $B^2/8$ 每个块不命中 misses f[illegible]ock
  - $2n/B * B^2/8 = nB/4$
    (省略矩阵c omitting matrix

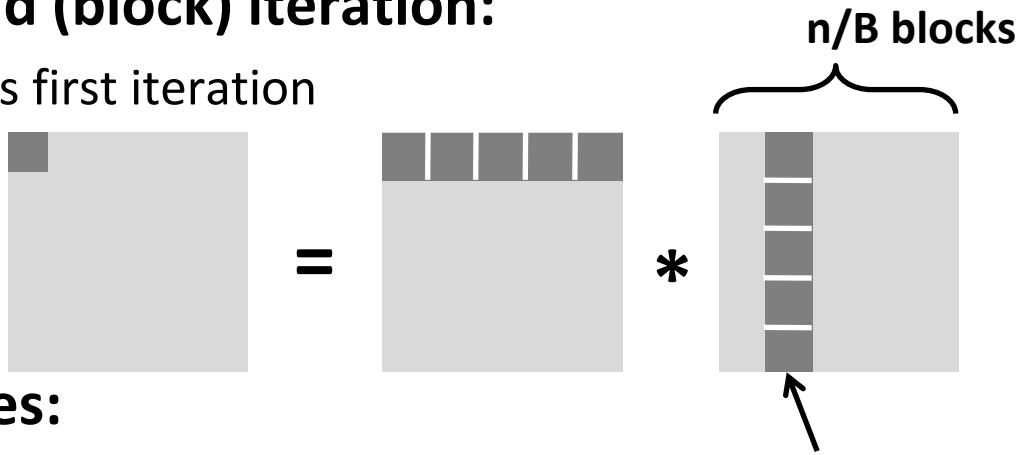


  - 后来缓存中 Afterwards in c[illegible]
    (简图 schematic)

Cache不命中分析 Cache Miss Analysis

- **假设 Assume:**
  - Cache块大小为8个双精度浮点数 Cache block = 8 doubles
  - Cache大小C远小于n Cache size C << n (much smaller than n)
  - 三个块 Three blocks　　适合大小为：fit into cache: $3B^2 < C$

- **第二次（块）迭代 Second (block) iteration:**
  - 与第一次迭代相同 Same as first iteration
  - $2n/B * B^2/8 = nB/4$



- **总计不命中： Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$

分块总结 Blocking Summary

- **无分块 No blocking: (9/8) * n^3**
- **分块 Blocking: 1/(4B) * n^3**

- **使用最大块大小B，以便满足$3B^2 < C$ Use largest block size B, such that B satisfies $3B^2 < C$**
  - 在cache中装3个块！两个输入，一个输出 Fit three blocks in cache! Two input, one output.

- **性能巨大差异原因分析 Reason for dramatic difference:**
  - 矩阵乘法有天然的时间局部性 Matrix multiplication has inherent temporal locality:
    - 输入数据规模 Input data: $3n^2$, 计算规模 computation $2n^3$
    - 每个数组元素使用次数 Every array elements used O(n) times!
  - 但程序必须正确编写 But program has to be written properly

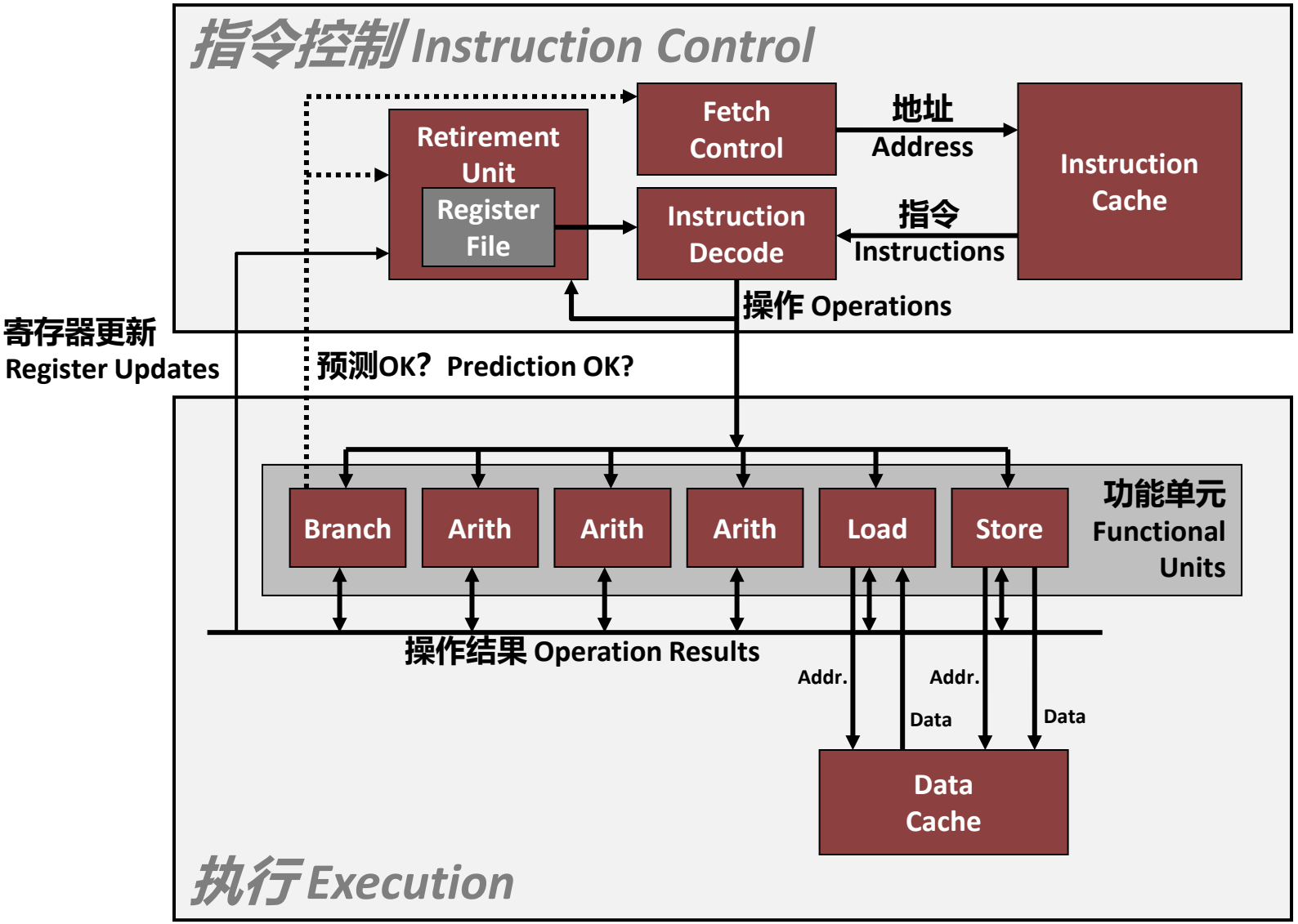
Cache总结 Cache Summary

- **Cache存储器对程序性能影响巨大 Cache memories can have significant performance impact**

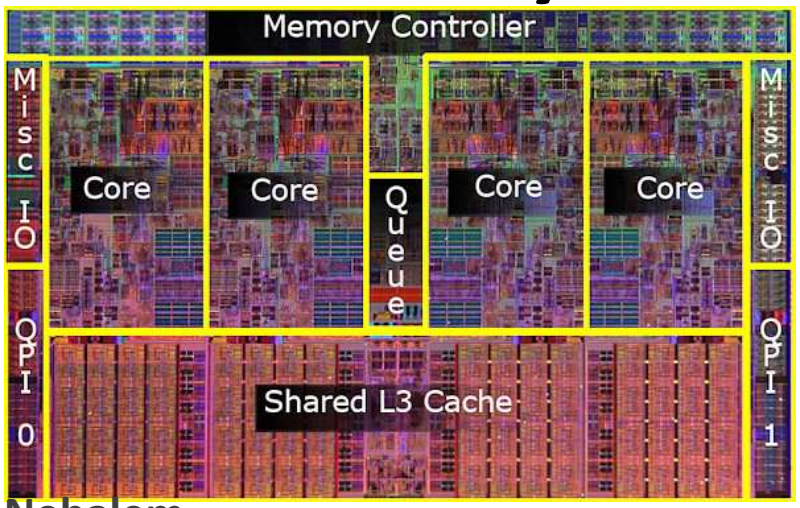
- **编写程序时需要充分利用 You can write your programs to exploit this!**
  - 关注内层循环，其中有大量计算和访存 Focus on the inner loops, where bulk of computations and memory accesses occur.
  - 充分利用空间局部性，按顺序连续读取数据对象 Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - 充分利用时间局部性，一次读入多次使用数据对象 Try to maximize temporal locality by using a data object as often as possible once it’s read from memory.

补充幻灯片 Supplemental slides

回忆：现代CPU设计
Recall: Modern CPU Design

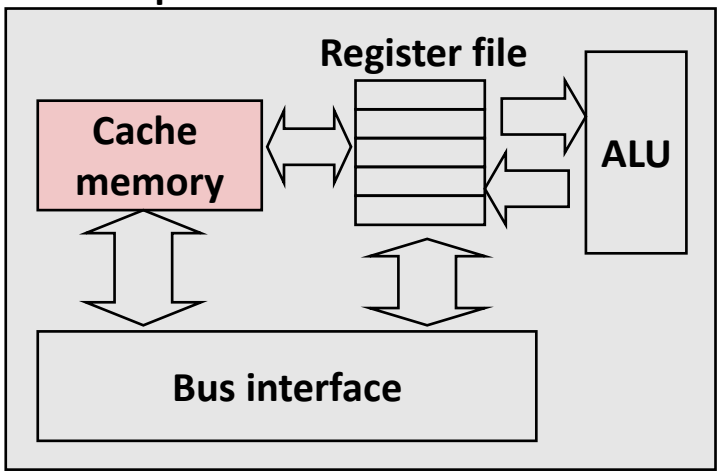


CPU内部真实是什么样?
What it Really Looks Like

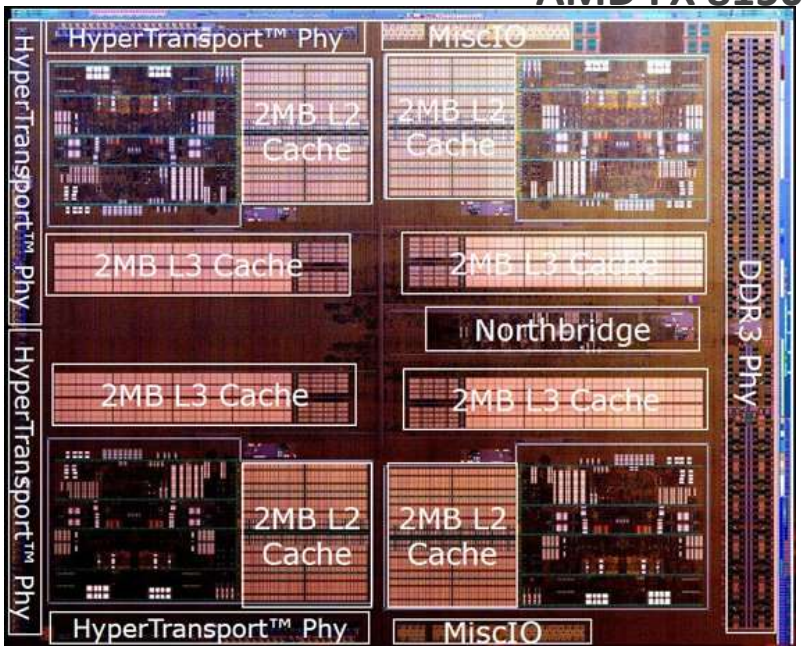


Nehalem

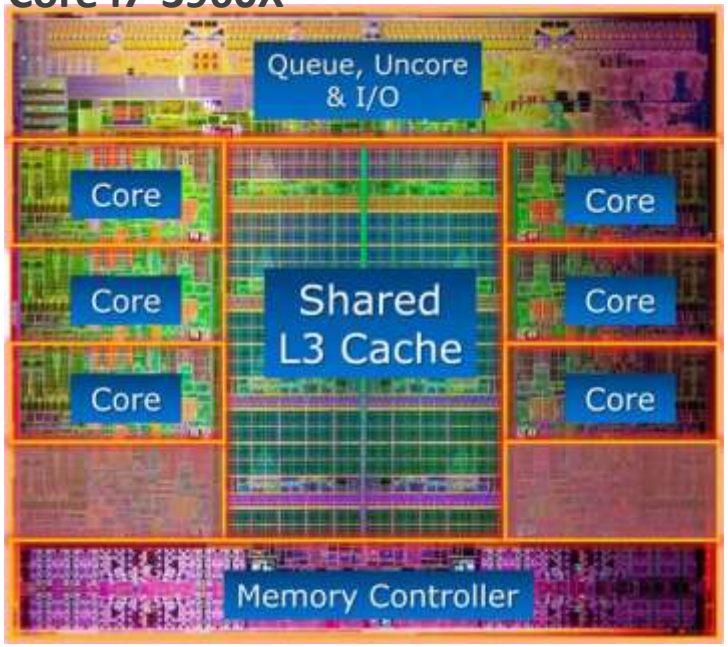
CPU chip



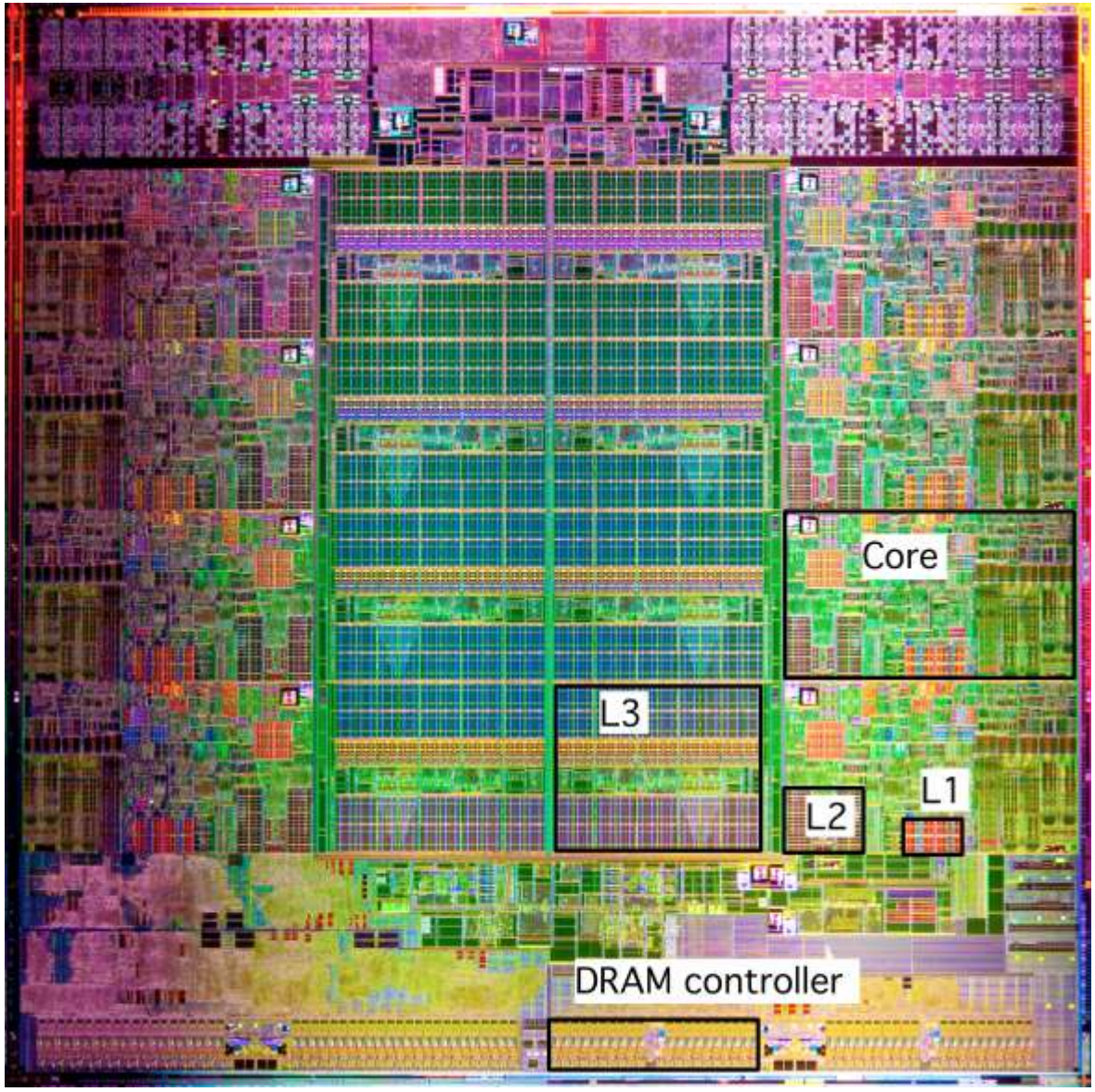
AMD FX 8150



Core i7-3960X



CPU内部真实是什么样？ （续）
What it Really Looks Like (Cont.)



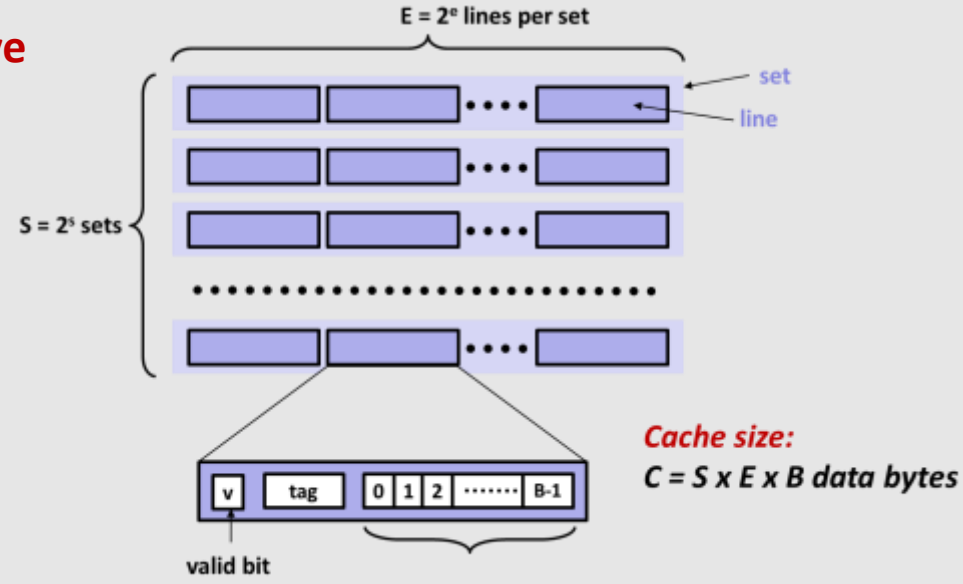
Intel Sandy Bridge
Processor Die

L1: 32KB Instruction + 32KB Data
L2: 256KB
L3: 3–20MB

示例： Core i7 L1 数据高速缓存
Example: Core i7 L1 Data Cache

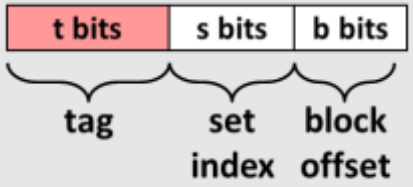
32 kB 8-way set associative
64 bytes/block
47 bit address range

B =
S = , s =
E = , e =
C =



Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:



Block offset: . bits
Set index: . bits
Tag: . bits

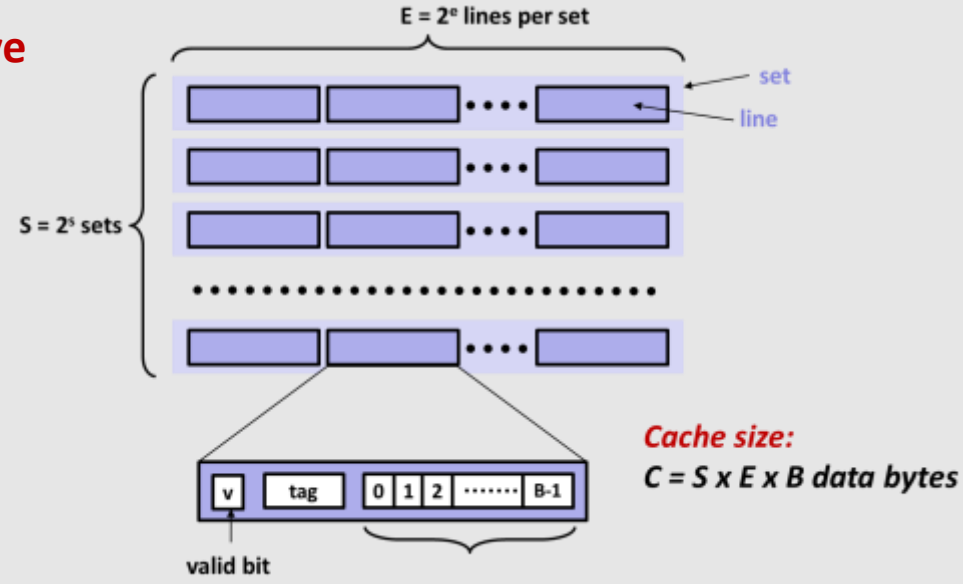
Stack Address:
0x00007f7262a1e010

Block offset: 0x??
Set index: 0x??
Tag: 0x??

示例： Core i7 L1 数据高速缓存
Example: Core i7 L1 Data Cache

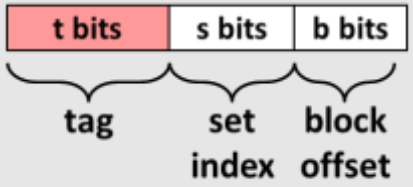
32 kB 8-way set associative
64 bytes/block
47 bit address range

B = 64
S = 64, s = 6
E = 8, e = 3
C = 64 x 64 x 8 = 32,768



Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:



Block offset: 6 bits
Set index: 6 bits
Tag: 35 bits

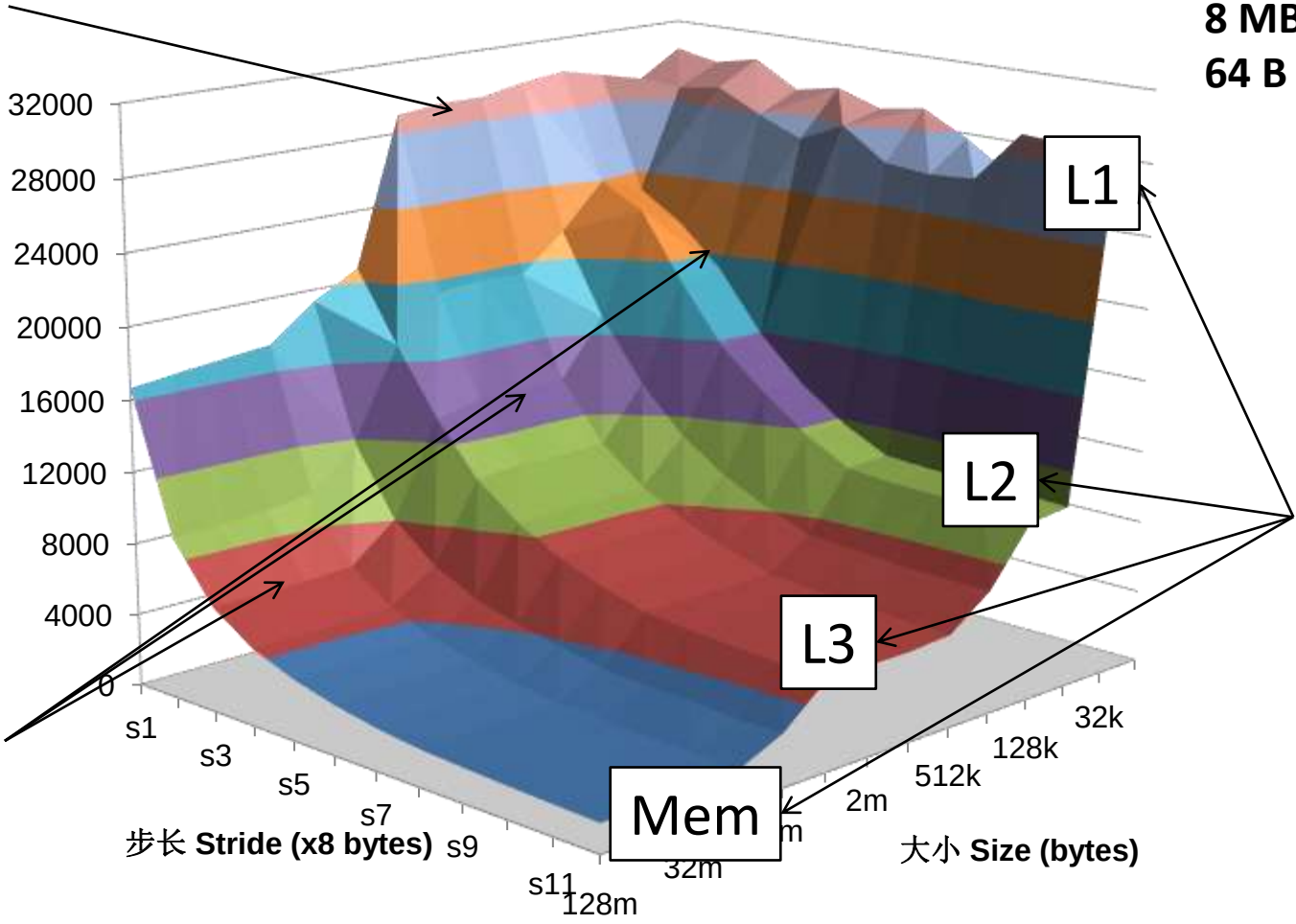
Stack Address:
0x00007f7262a1e010

0000 0001 0000

Block offset: 0x10
Set index: 0x0
Tag: 0x7f7262a1e

内存山丘
The Memory Mountain

Core i5 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

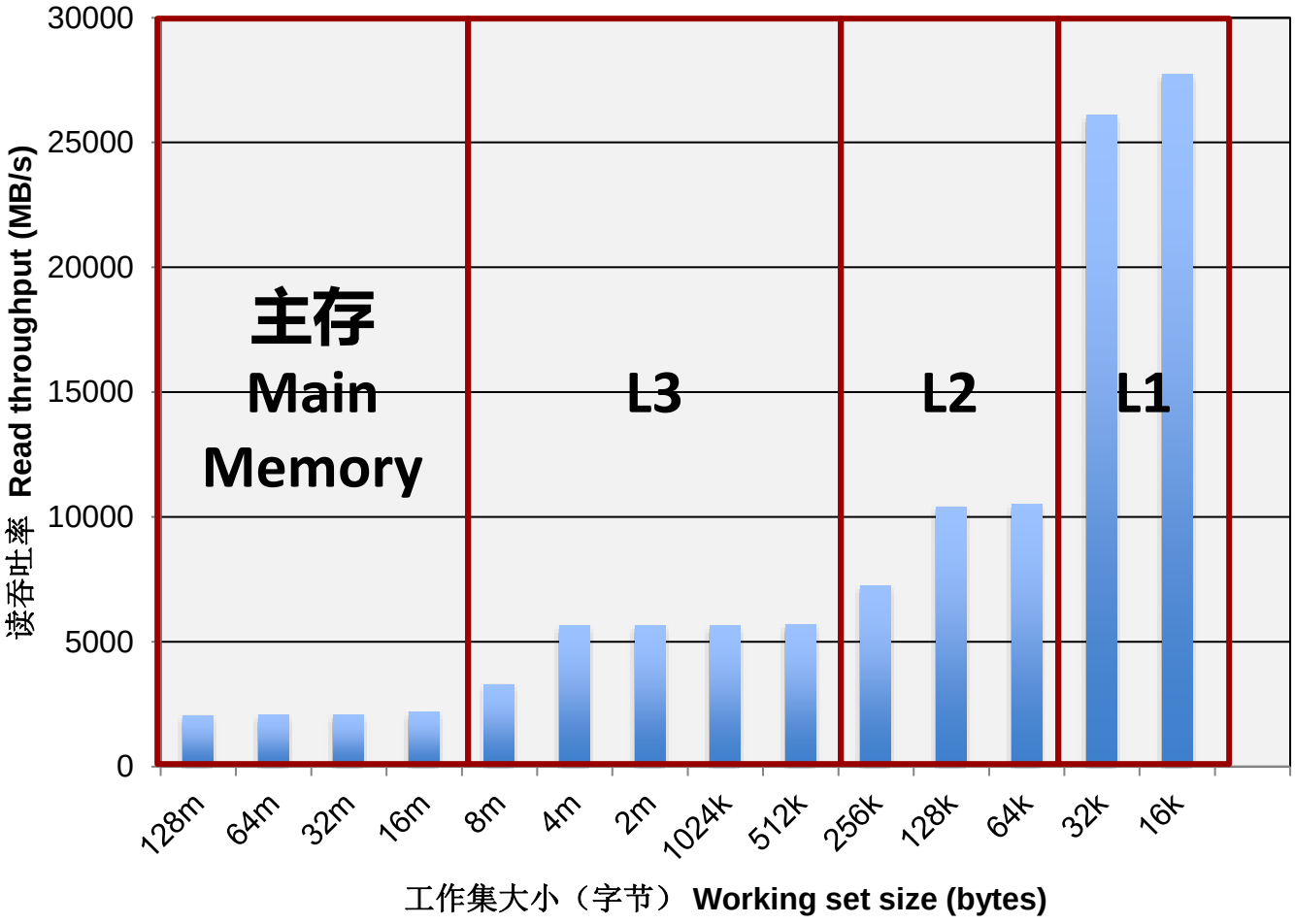


主动预取
Aggressive prefetching

空间局部性斜坡
Slopes of spatial locality

时间局部性山脊
Ridges of temporal locality

内存山丘对Cache容量的影响
Cache Capacity Effects from Memory Mountain



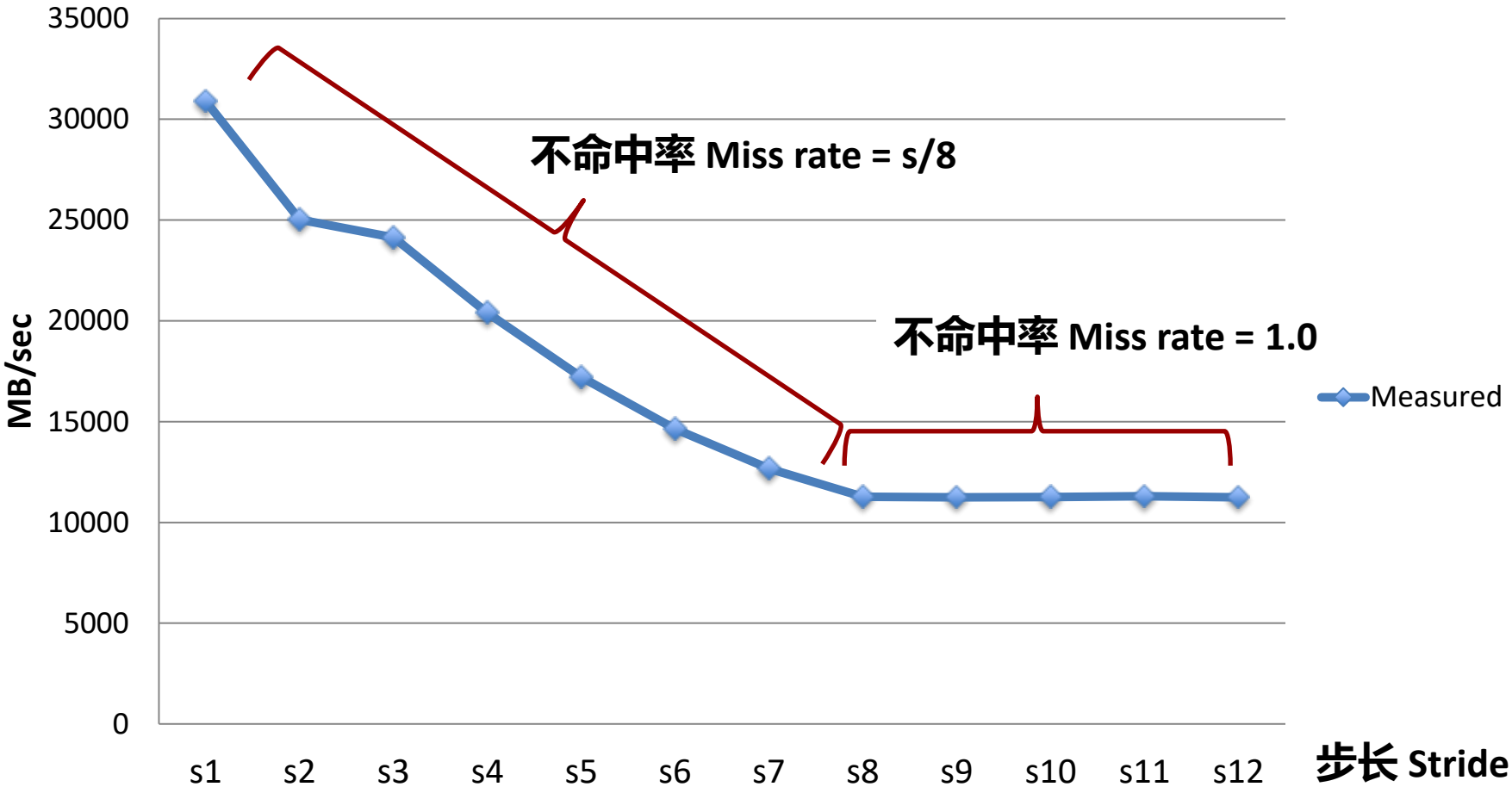
Core i7 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

用步长为8穿越内存山丘断面
Slice through memory mountain with stride=8

内存山丘对Cache块大小的影响
Cache Block Size Effects from Memory Mountain

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

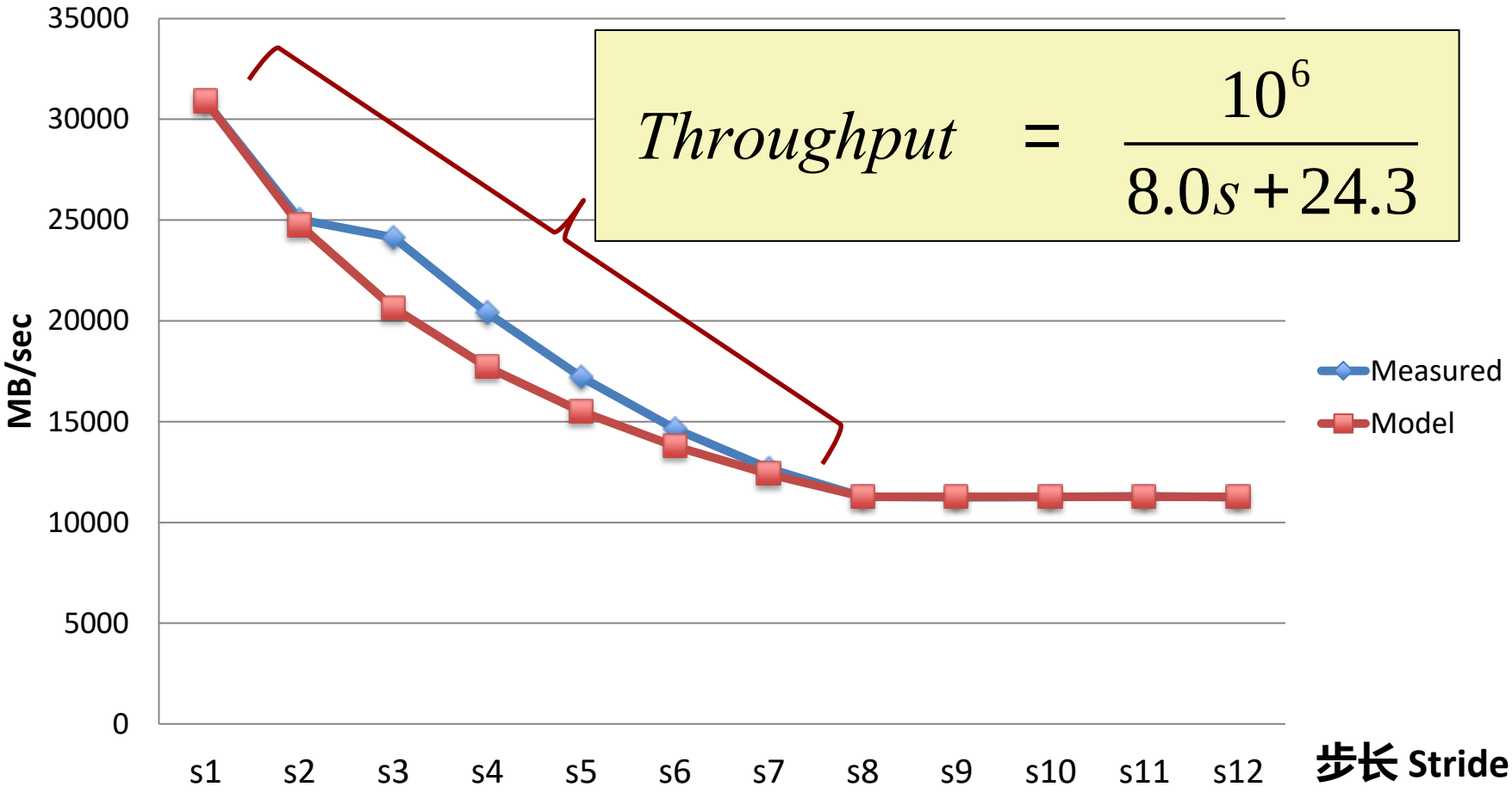
大小为128K时的吞吐量 Throughput for size = 128K



建模内存山丘的块大小效果
Modeling Block Size Effects from Memory Mountain

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

大小为128K时的吞吐量 Throughput for size = 128K



2008年的内存山丘
2008 Memory Mountain

Core 2 Duo
2.4 GHz
32 KB L1 d-cache
6MB L2 cache
64 B block size