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

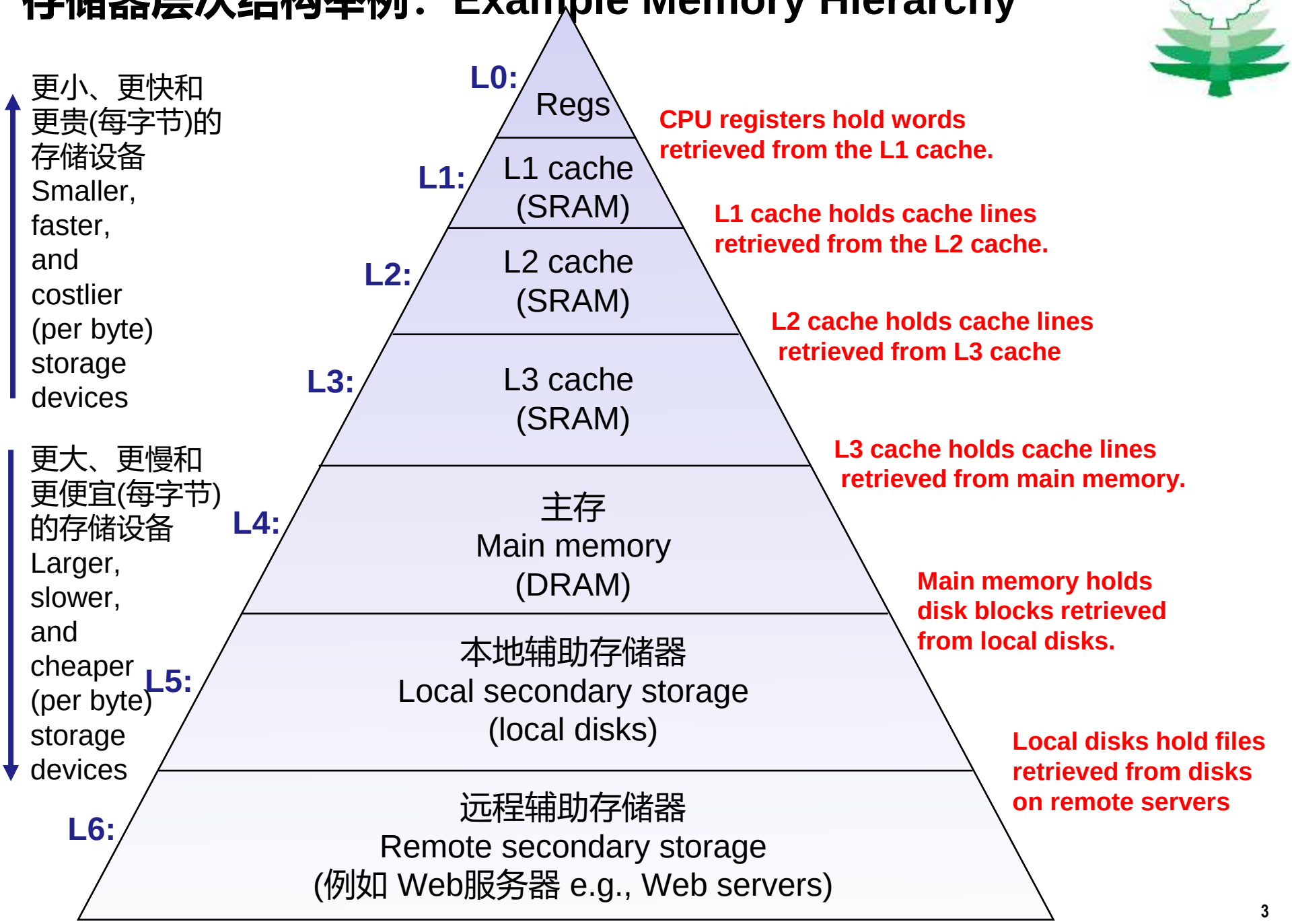
原作者：

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

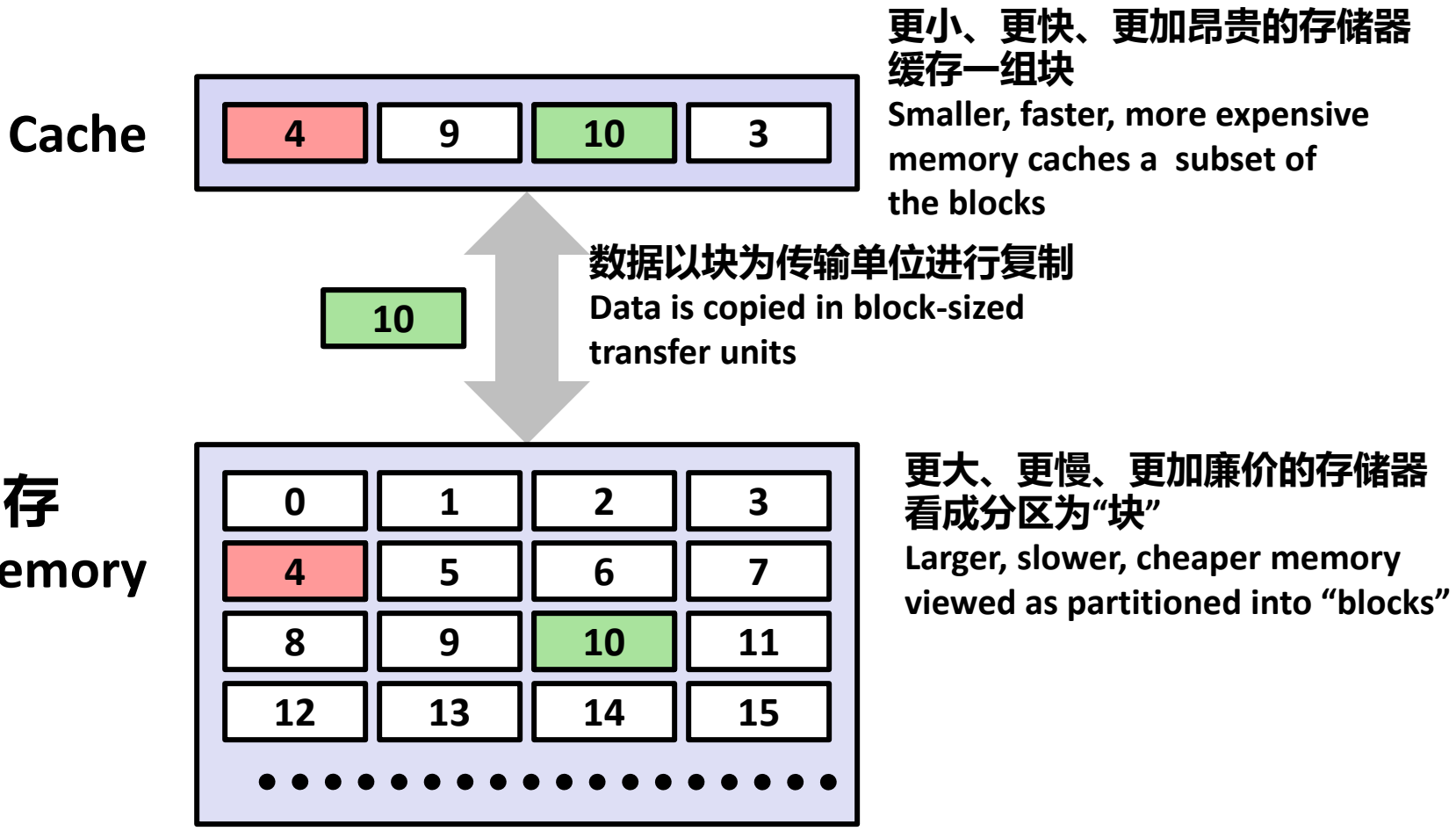
主要内容

- **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

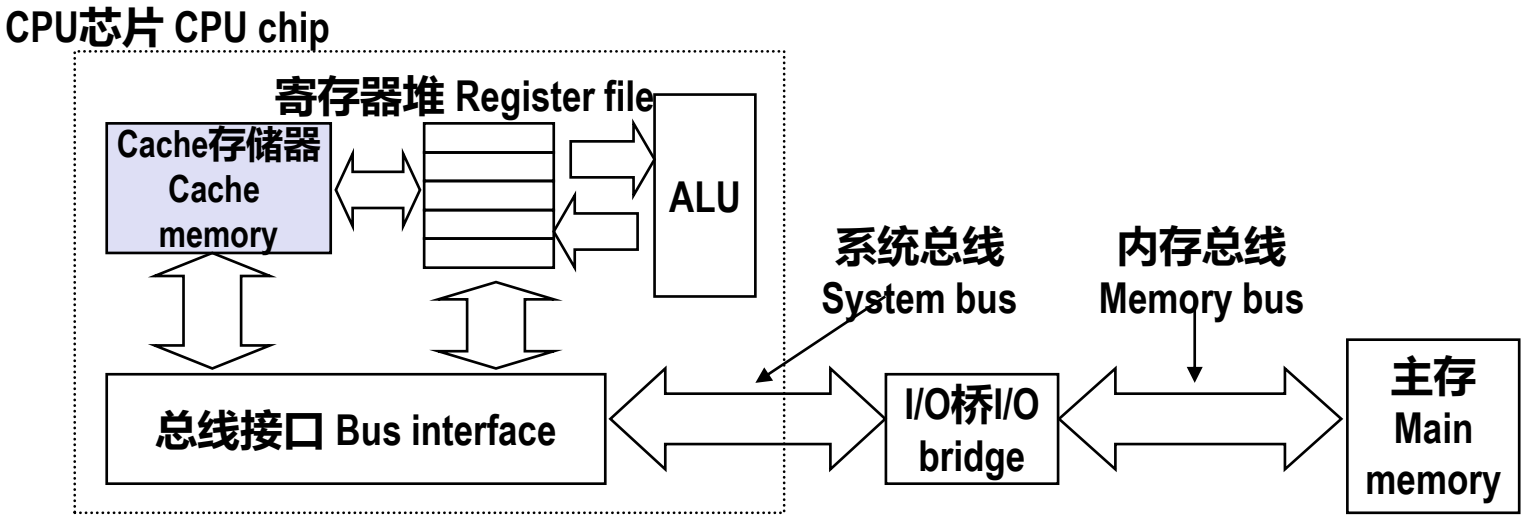


Cache基本概念 General Cache Concept

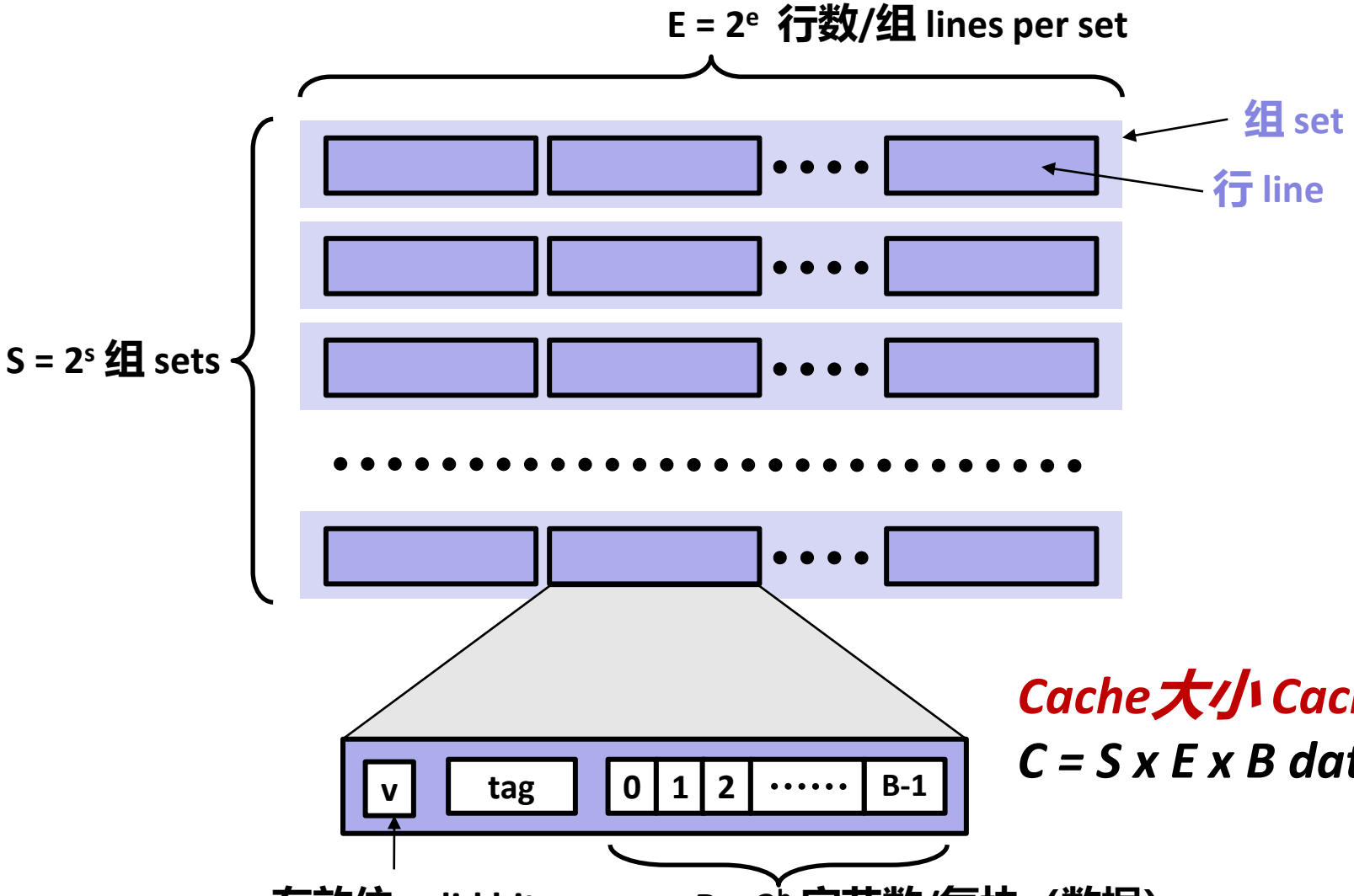


Cache存储器 Cache Memories

- **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)

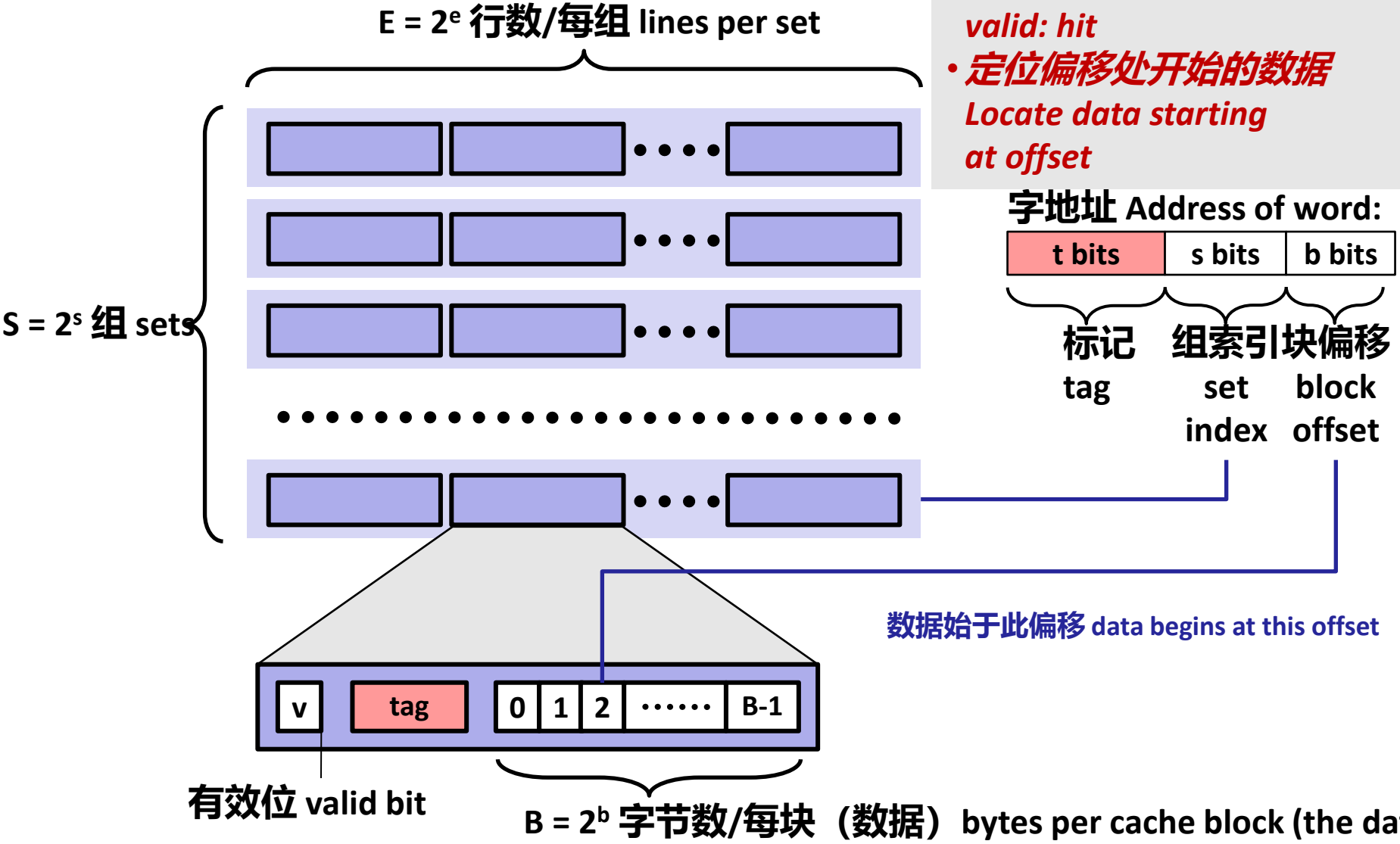


Cache大小 Cache size:

C = S x E x B data bytes

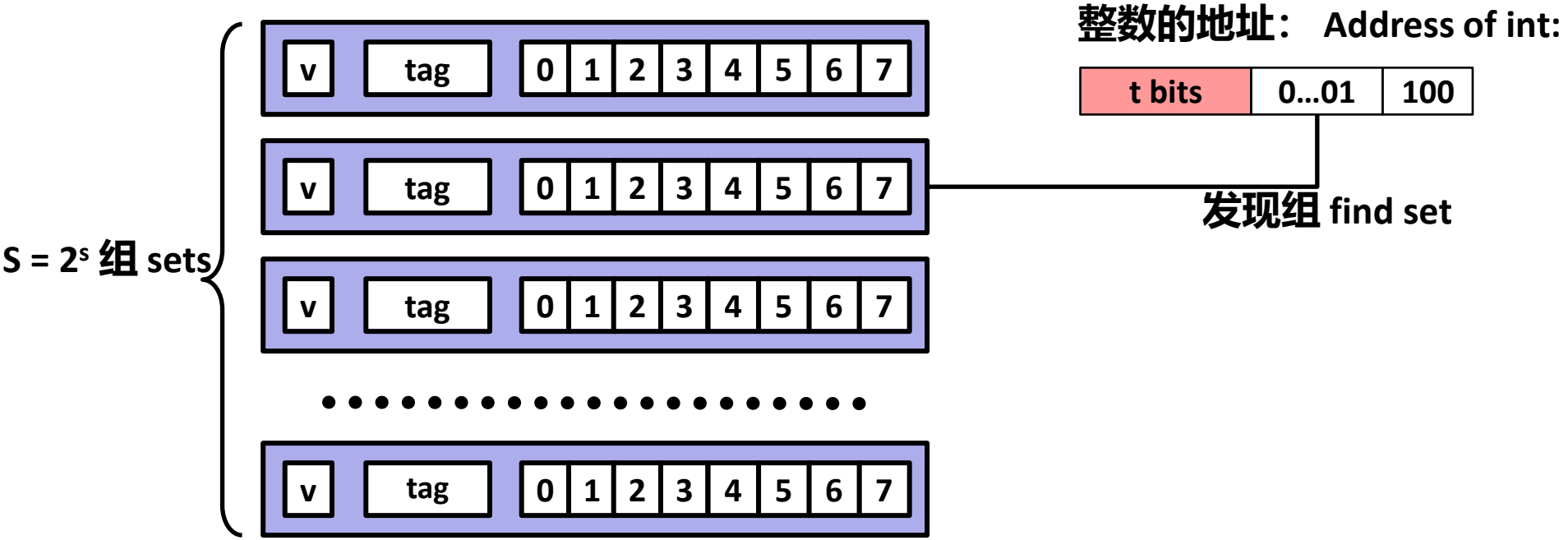
Cache读操作 Cache Read

- *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*



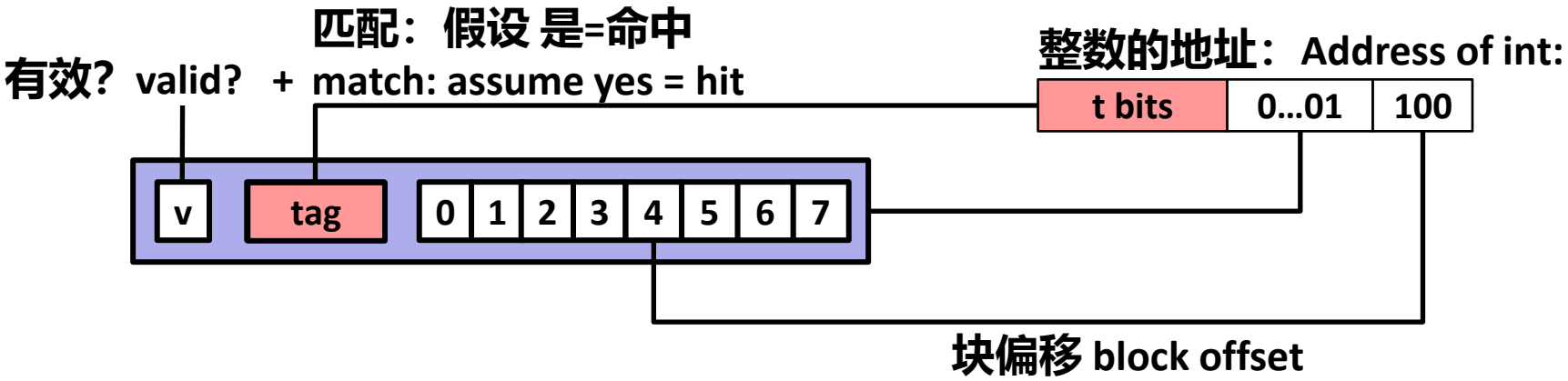
示例：直接映射Cache（E=1）
Example: Direct Mapped Cache (E = 1)

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



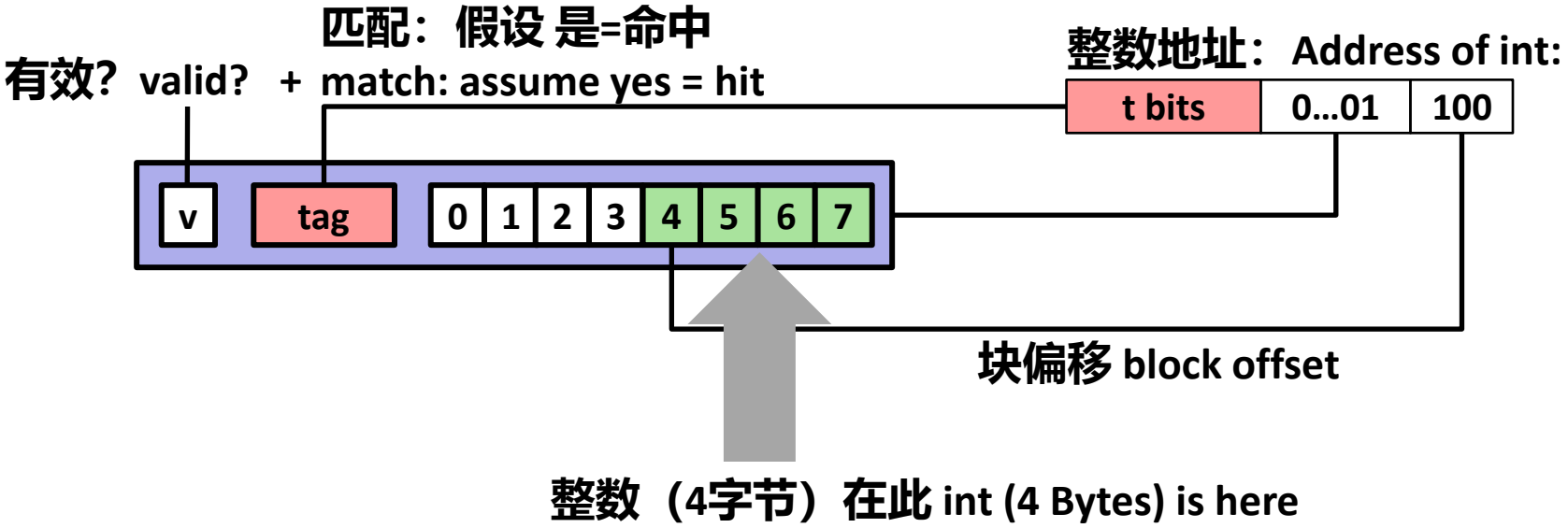
示例：直相联映射Cache（E=1）
Example: Direct Mapped Cache (E = 1)

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）

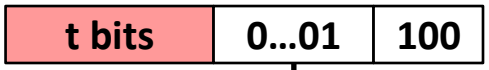
Cache E-way Set Associative Cache (Here: E = 2)

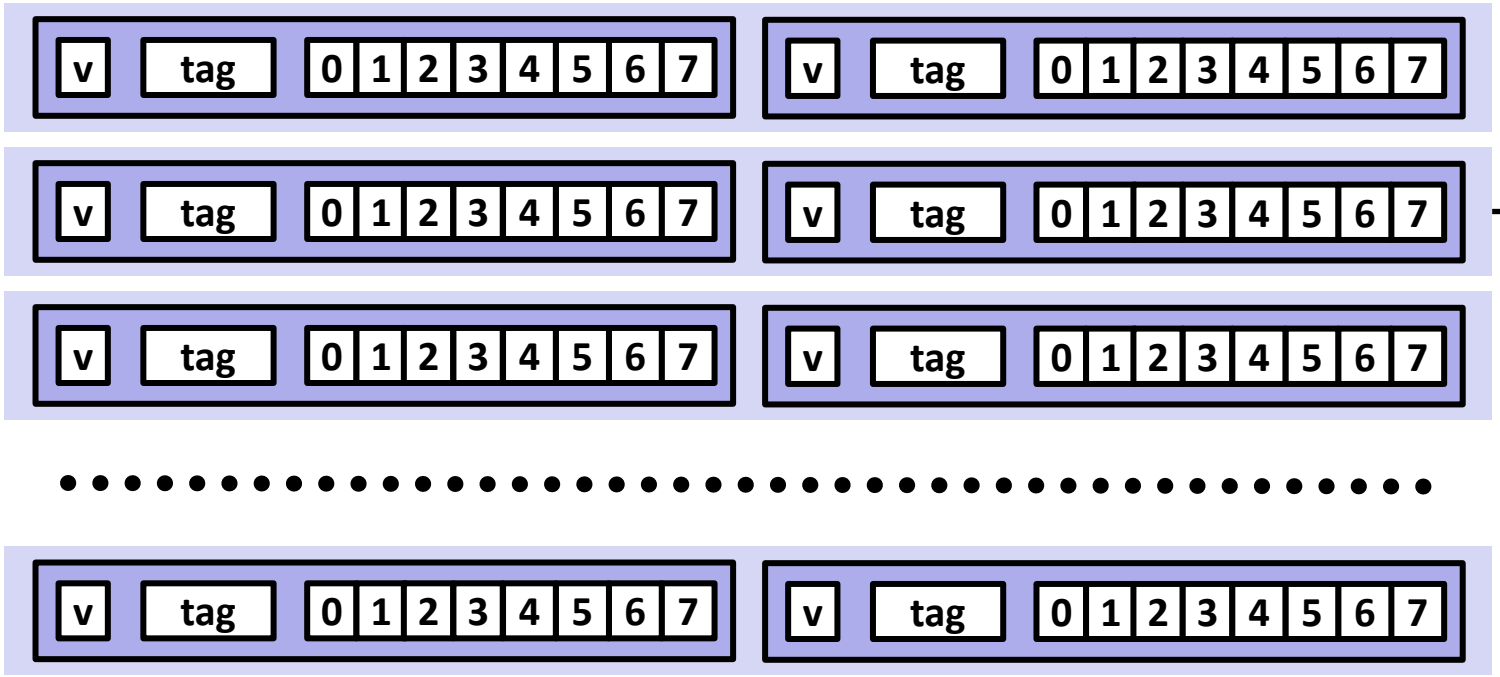
E=2：每组2行 E = 2: Two lines per set

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:

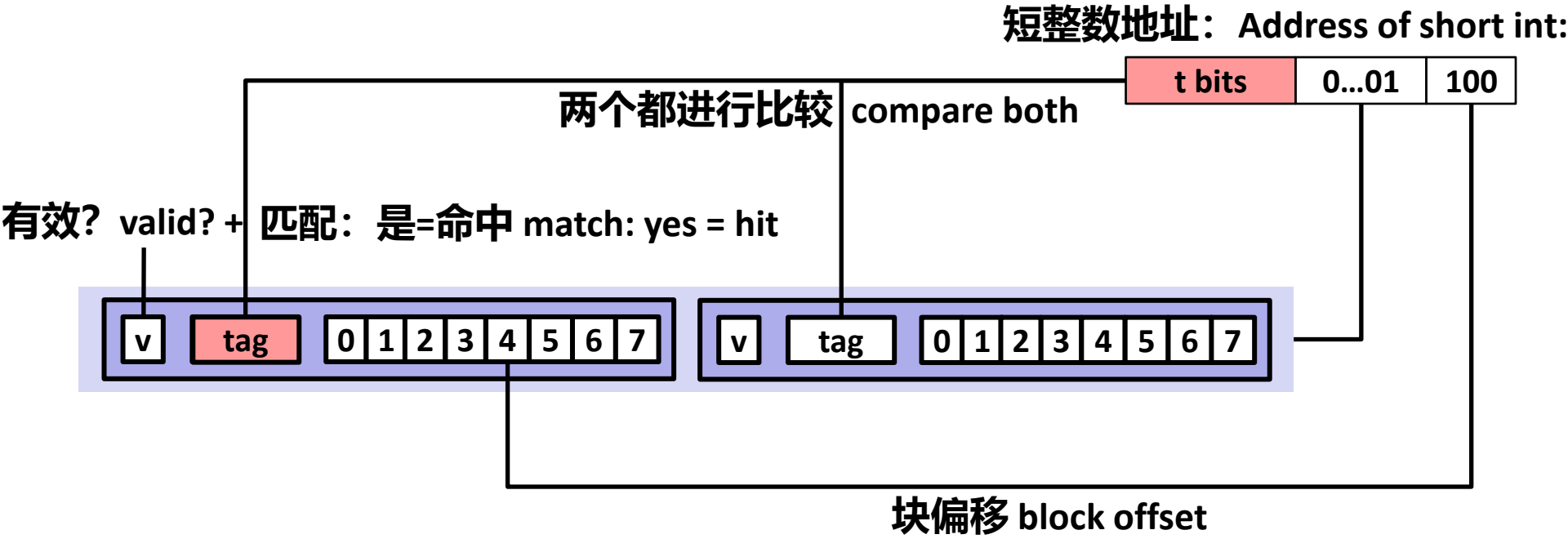




E路组相联Cache（E=2）
E-way Set Associative Cache (Here: E = 2)

E=2 每组2行： E = 2: Two lines per set

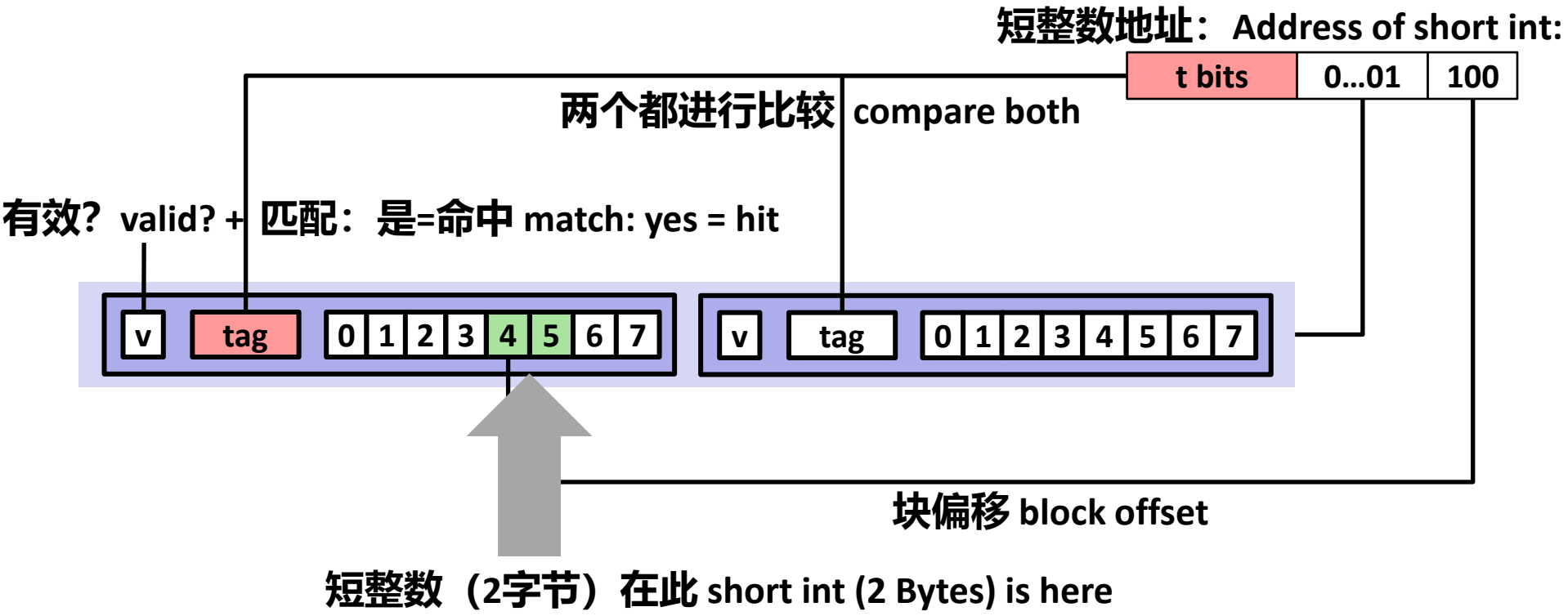
假设： Cache块大小为8字节 Assume: cache block size 8 bytes



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):

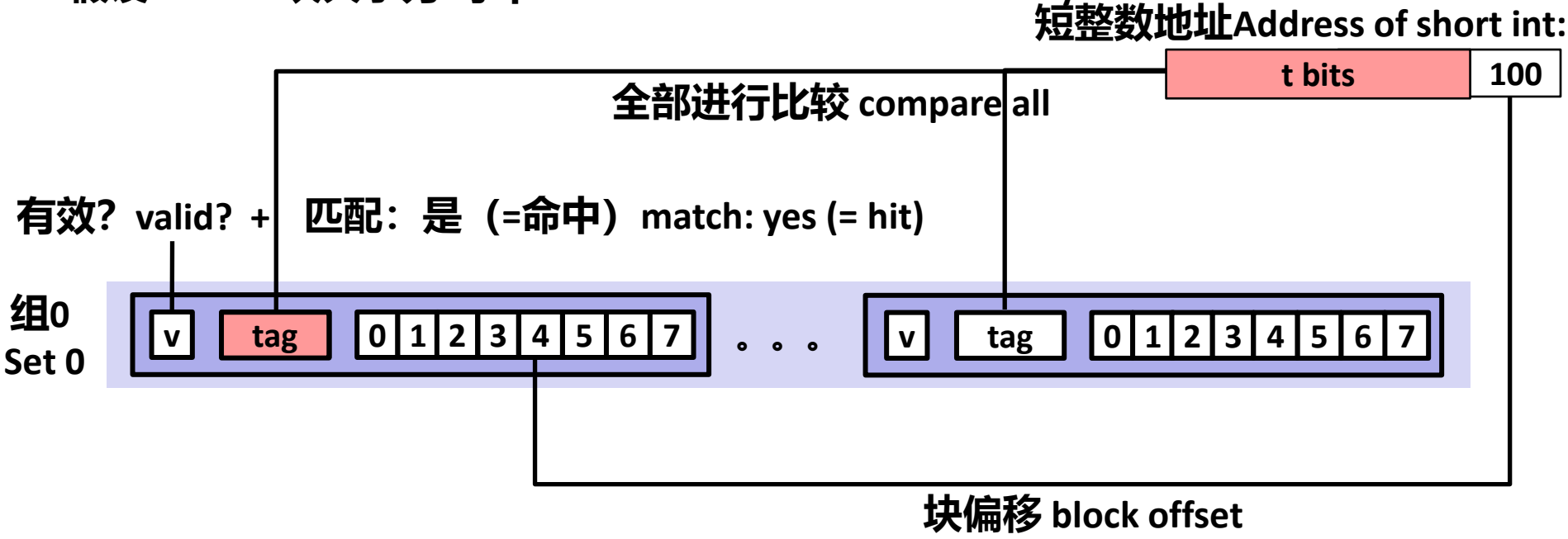
0	**[0000$_2$],**	不命中 miss
1	**[0001$_2$],**	命中 hit
7	**[0111$_2$],**	不命中 miss
8	**[1000$_2$],**	不命中 miss
0	**[0000$_2$]**	命中 hit

	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

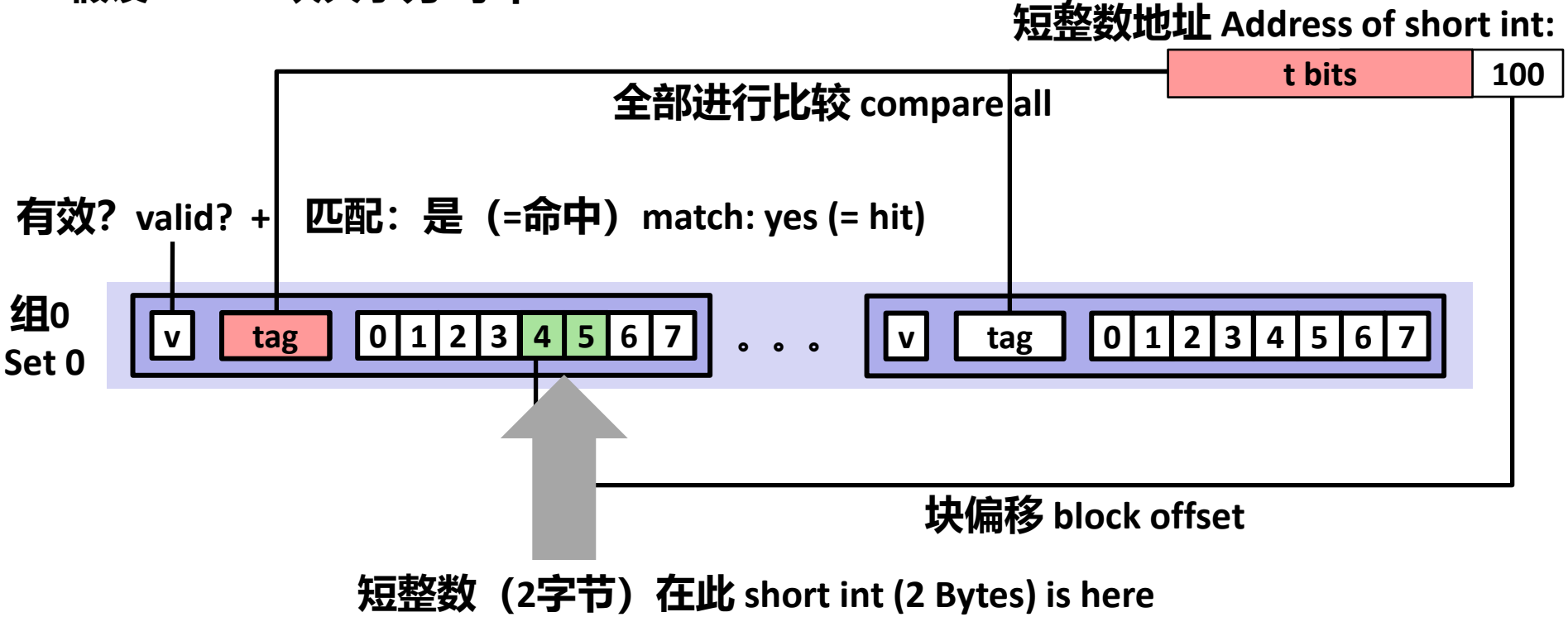
假设：cache块大小为8字节 Assume: cache block size B=8 bytes



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

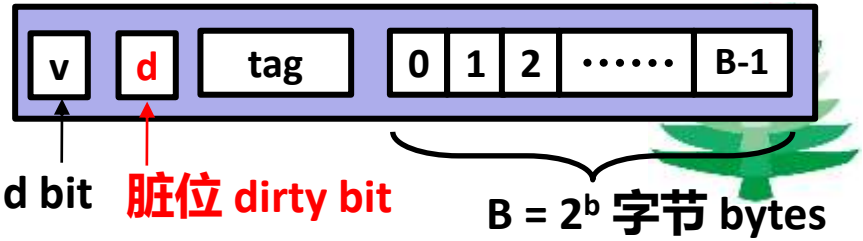
假设： cache块大小为8字节 Assume: cache block size B=8 bytes



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

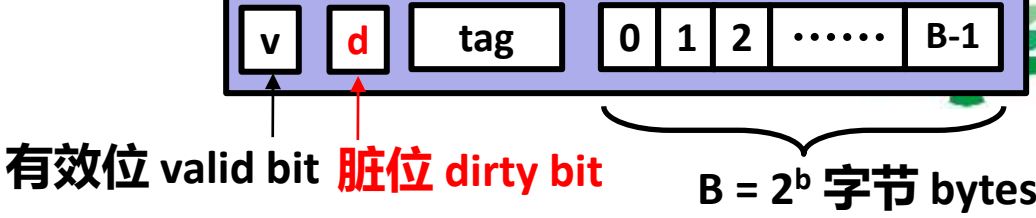
- 组0中的一行进行驱逐替换 One line in set 0 is for eviction and replacement
- 替换策略：随机，最近最少使用（LRU）。。。 Replacement policies: random, least recently used (LRU), …

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**

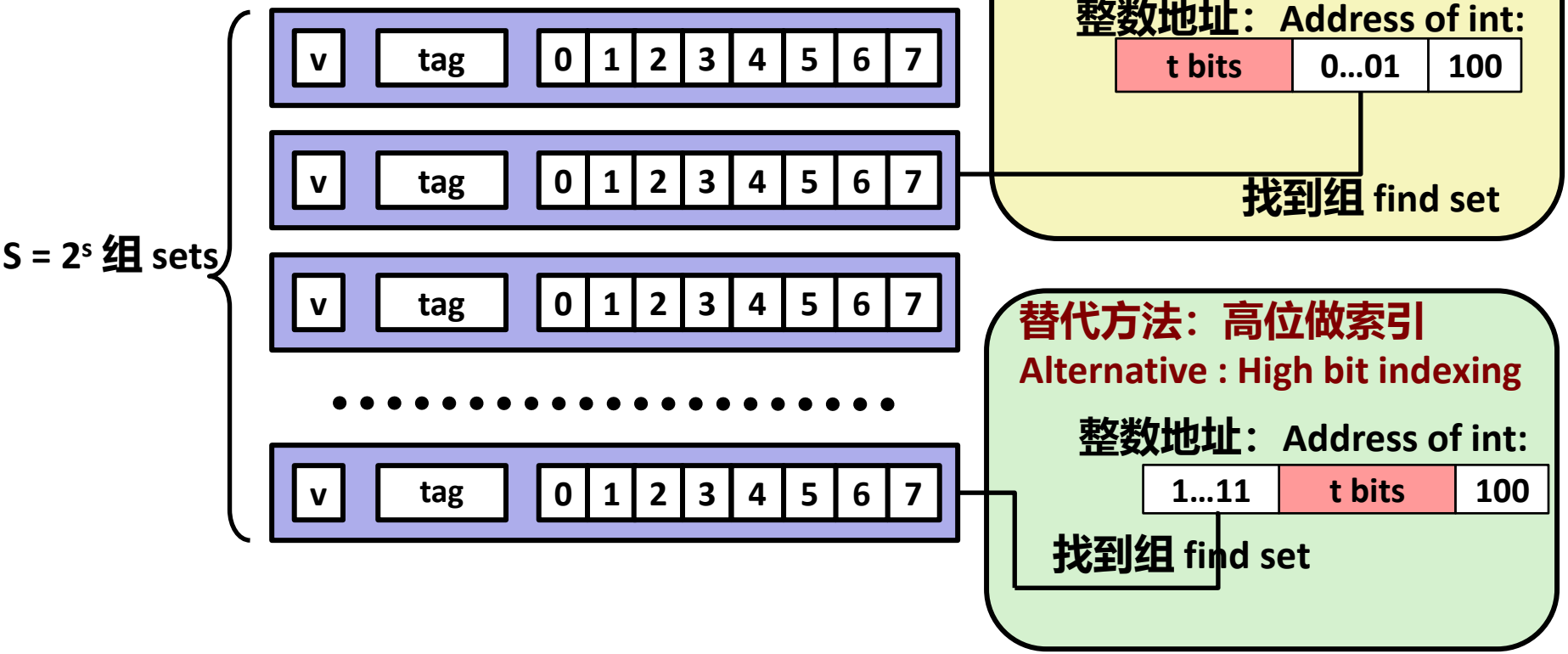
实践写回+写分配
Practical Write-back Write-allocate



- **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

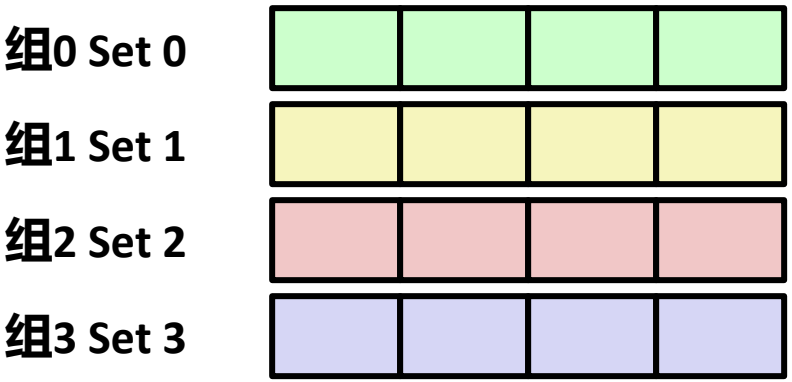
为何使用中间位做索引?
Why Index Using Middle Bits?

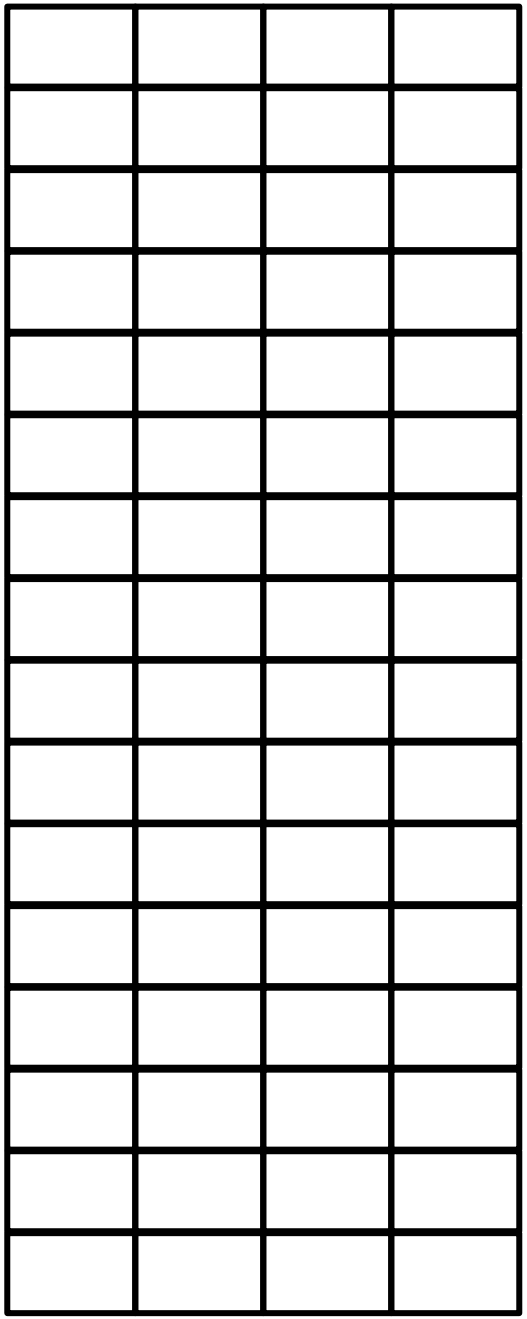
直接映射：每组一行 Direct mapped: One line per set
假设：cache块大小8字节 Assume: cache block size 8



索引方法说明 Illustration of Indexing Approaches

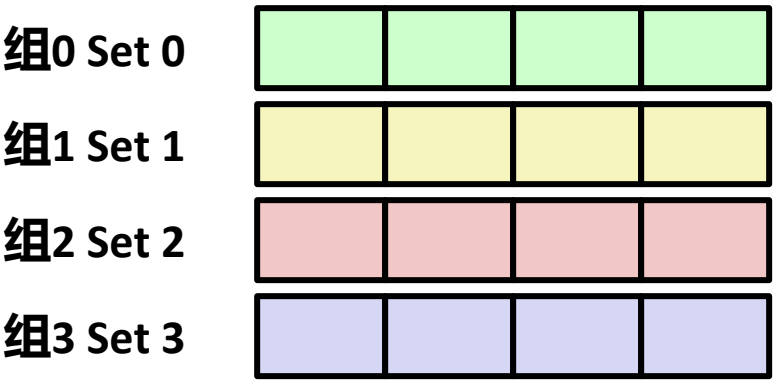
- 64字节内存 64-byte memory
  - 6位地址 6-bit addresses
- 16字节直接映射cache 16 byte, direct-mapped cache
- 块大小为4（因此分成4组）Block size = 4. (Thus, 4 sets; why?)
- 2位标记、2位索引、2位偏移 2 bits tag, 2 bits index, 2 bits offset

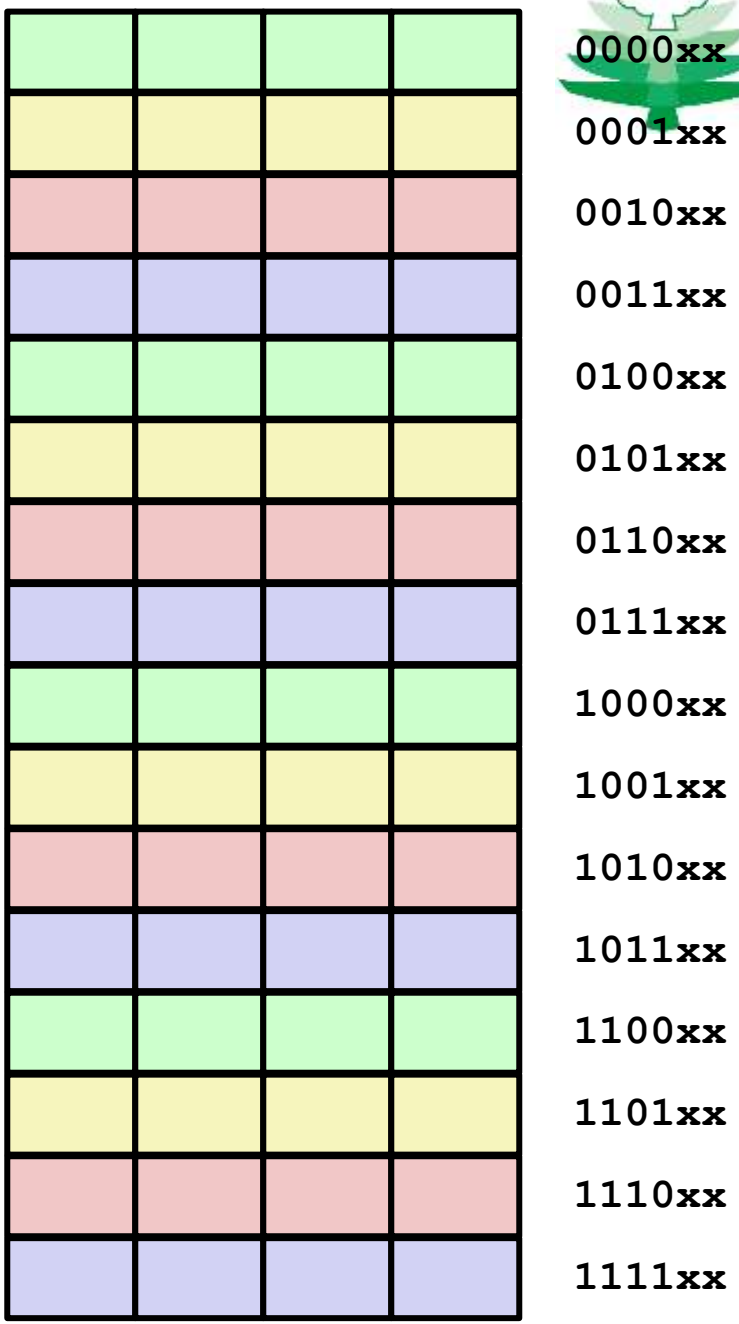




中间位做索引 Middle Bit Indexing

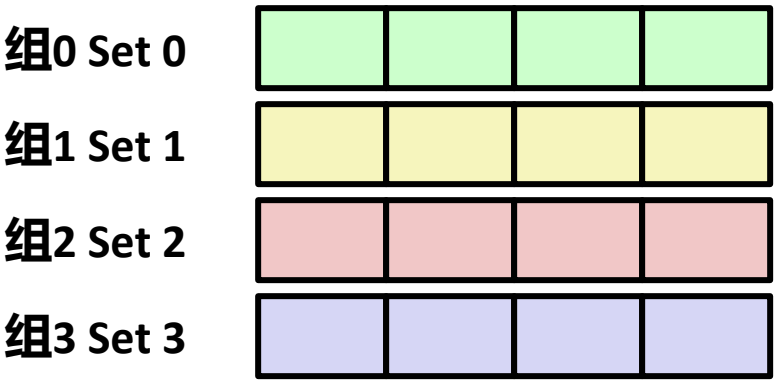
- 地址形式为 TTSSBB Addresses of form TTSSBB
  - TT 标记位 Tag bits
  - SS 组索引位 Set index bits
  - BB 偏移位 Offset bits
- 很好地利用了空间局部性 Makes good use of spatial locality

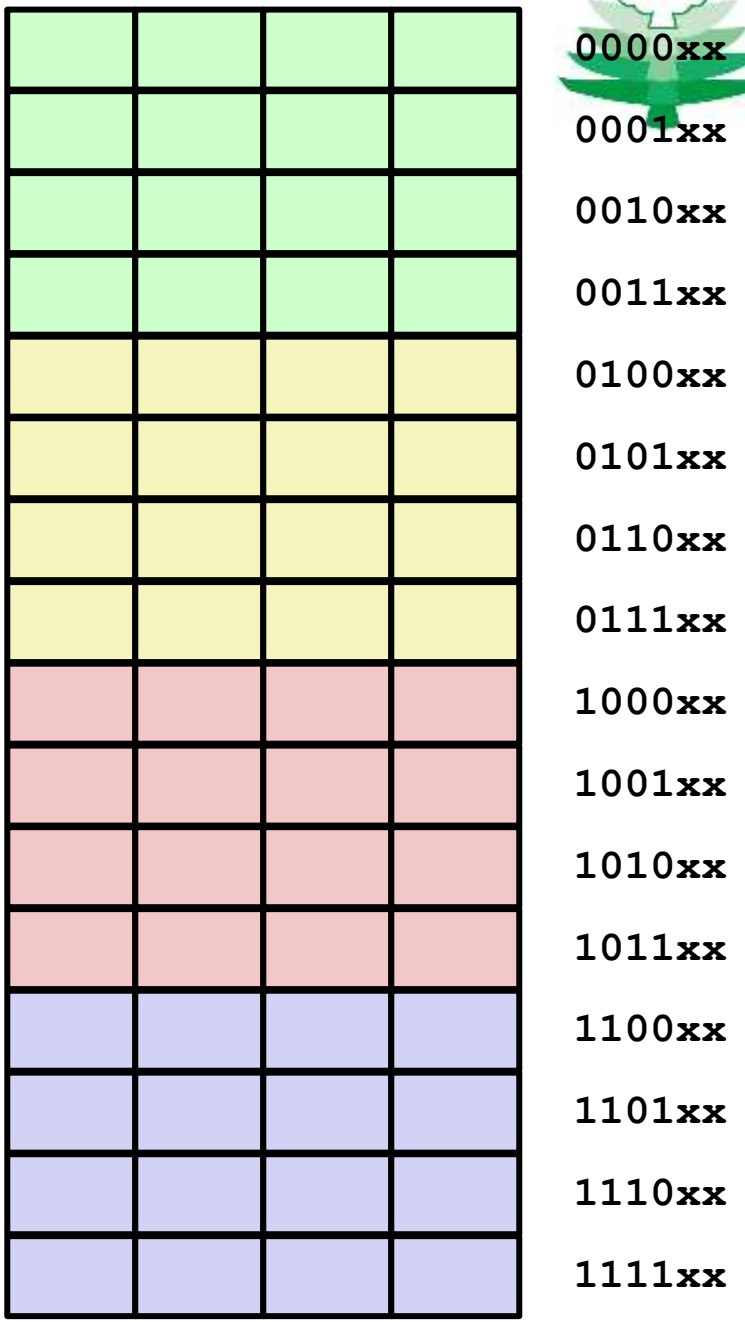




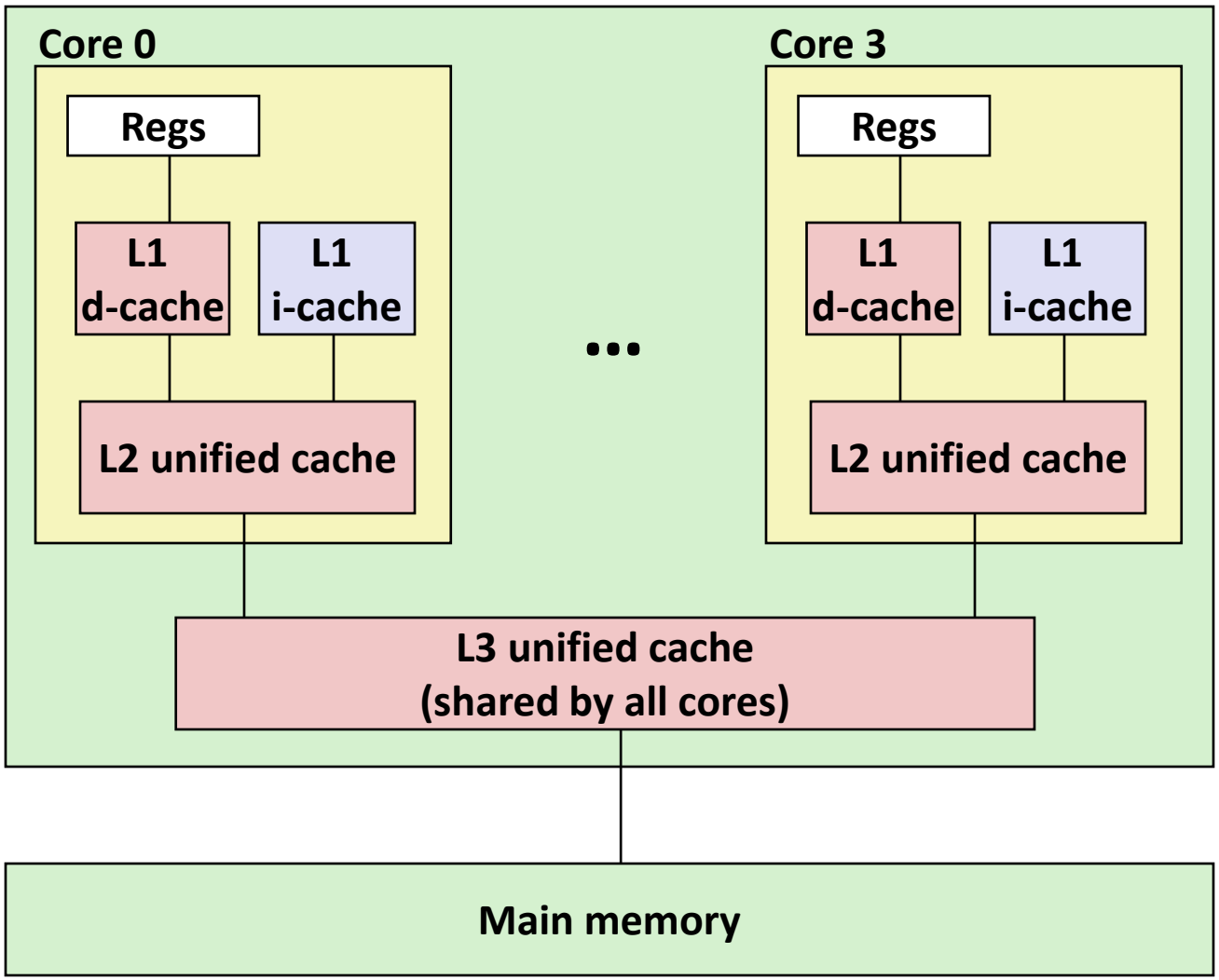
高位做索引 High Bit Indexing

- 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`.

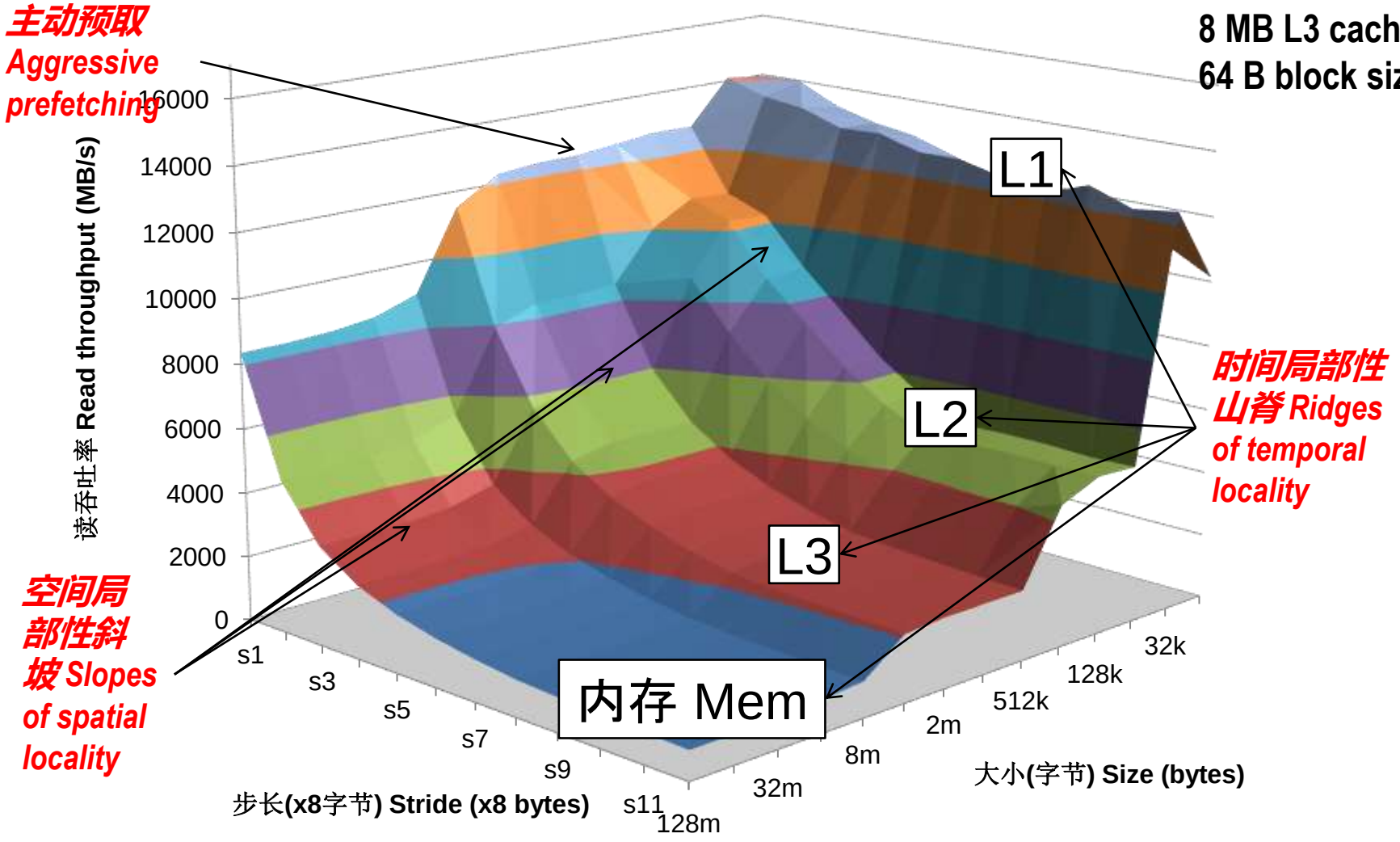
对于每个`elems`和`stride`参数:
For each `elems` and `stride`:

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

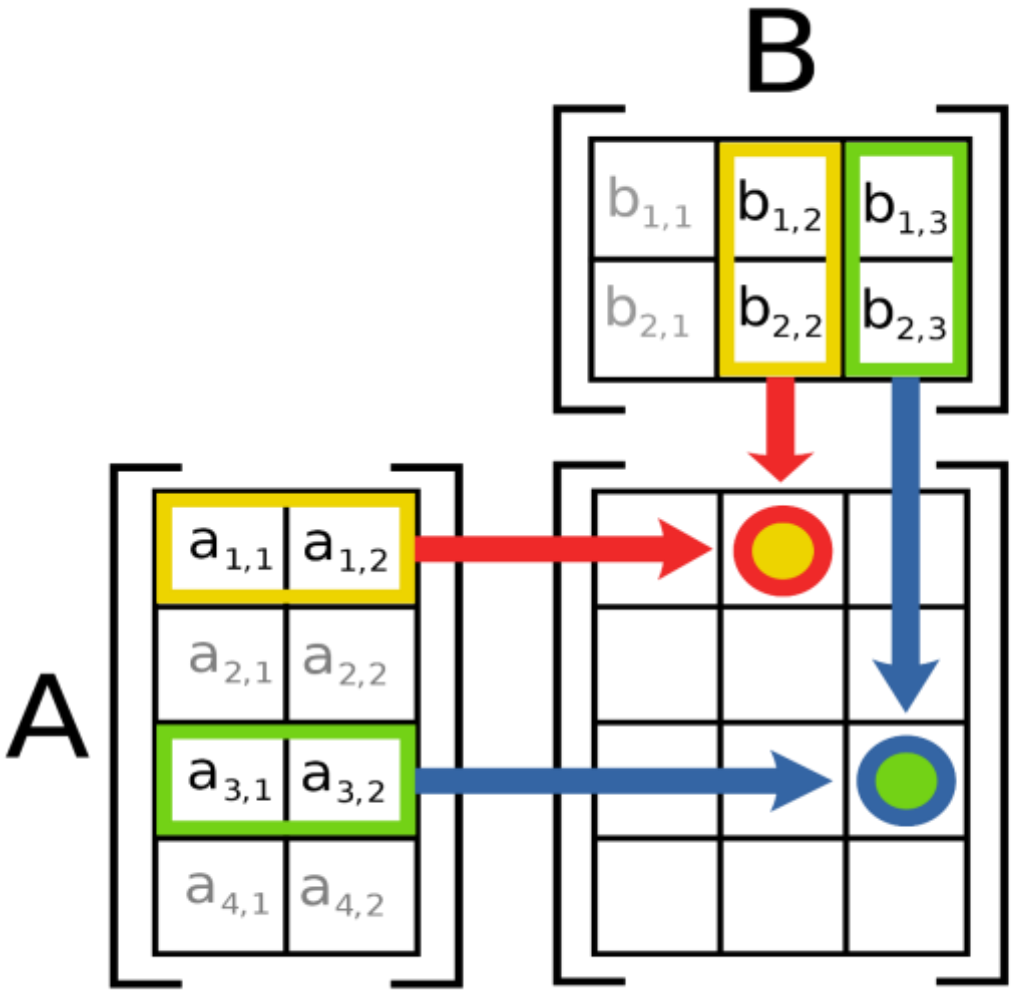
Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



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

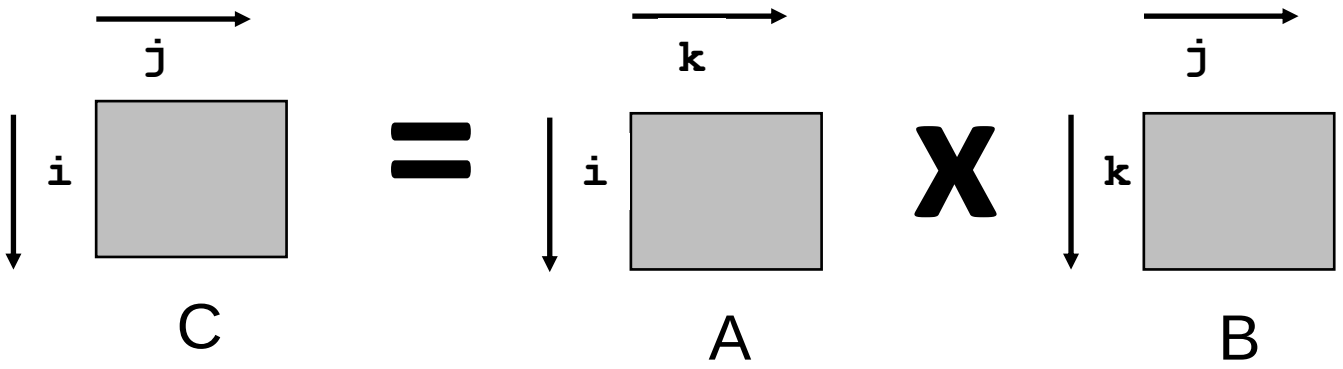
变量sum存储在寄存器中
Variable sum held in register

```
/* 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)

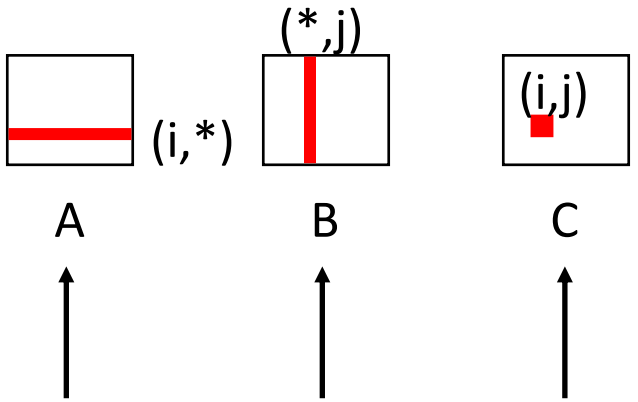
- **C语言数组按照行存储 C arrays allocated in row-major order**
  - 每行连续存储 each row in contiguous memory locations
- **访问每行的每列元素 Stepping through columns in one row:**
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - 访问连续元素 accesses successive elements
  - 如果块大小大于元素的字节数，则可以利用空间局部性 if block size (B) > sizeof(a_{ij}) 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:



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

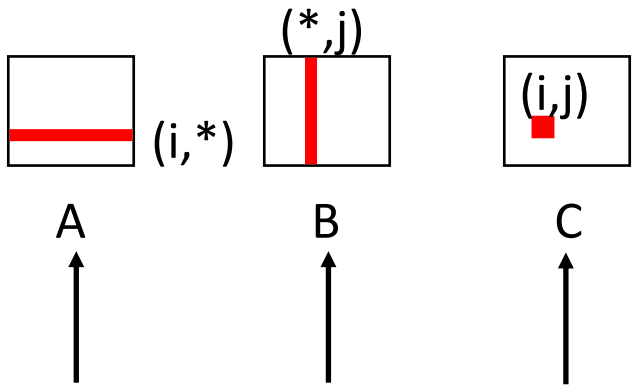
块大小=32字节（四个双精度浮点数） Block size = 32B (four doubles)

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:

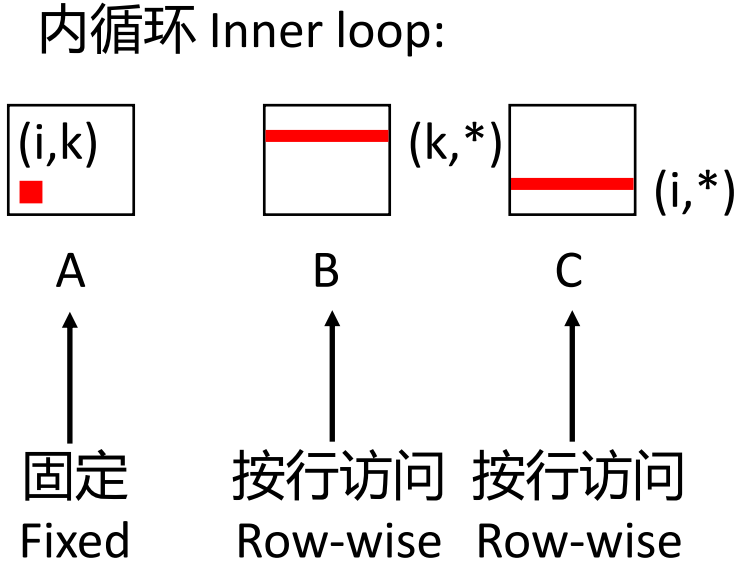
A	B	C
0.25	1.0	0.0

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



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

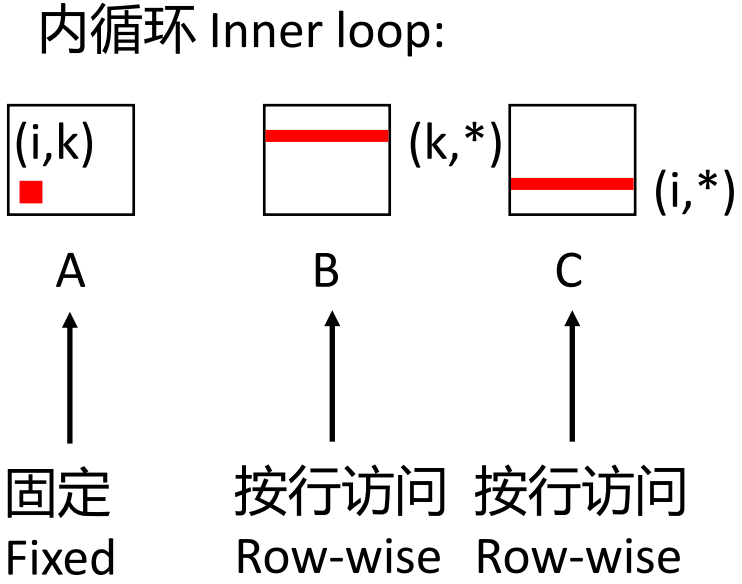
A	B	C
0.0	0.25	0.25

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

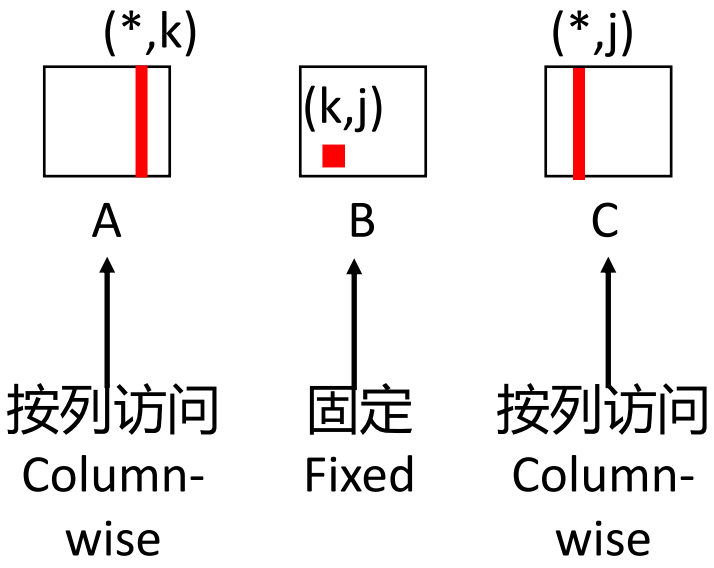
块大小=32字节（四个双精度浮点数）Block size = 32B (four doubles)

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

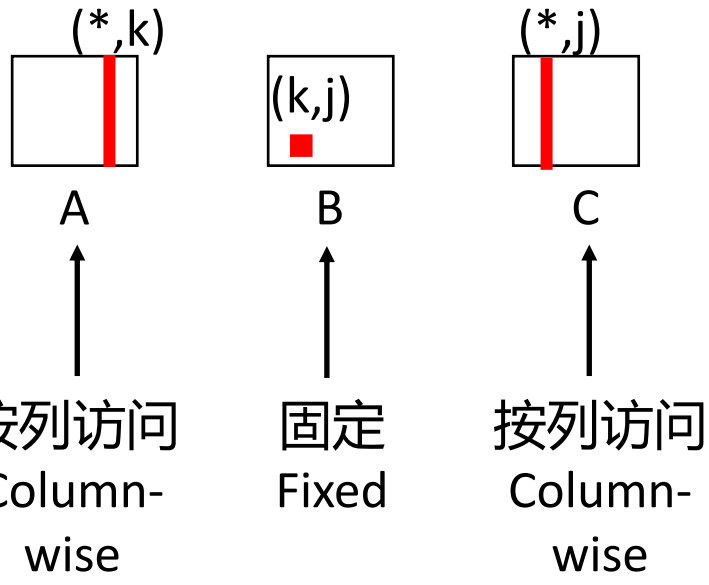
块大小=32字节（四个双精度浮点数）Block size = 32B (four doubles)

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):

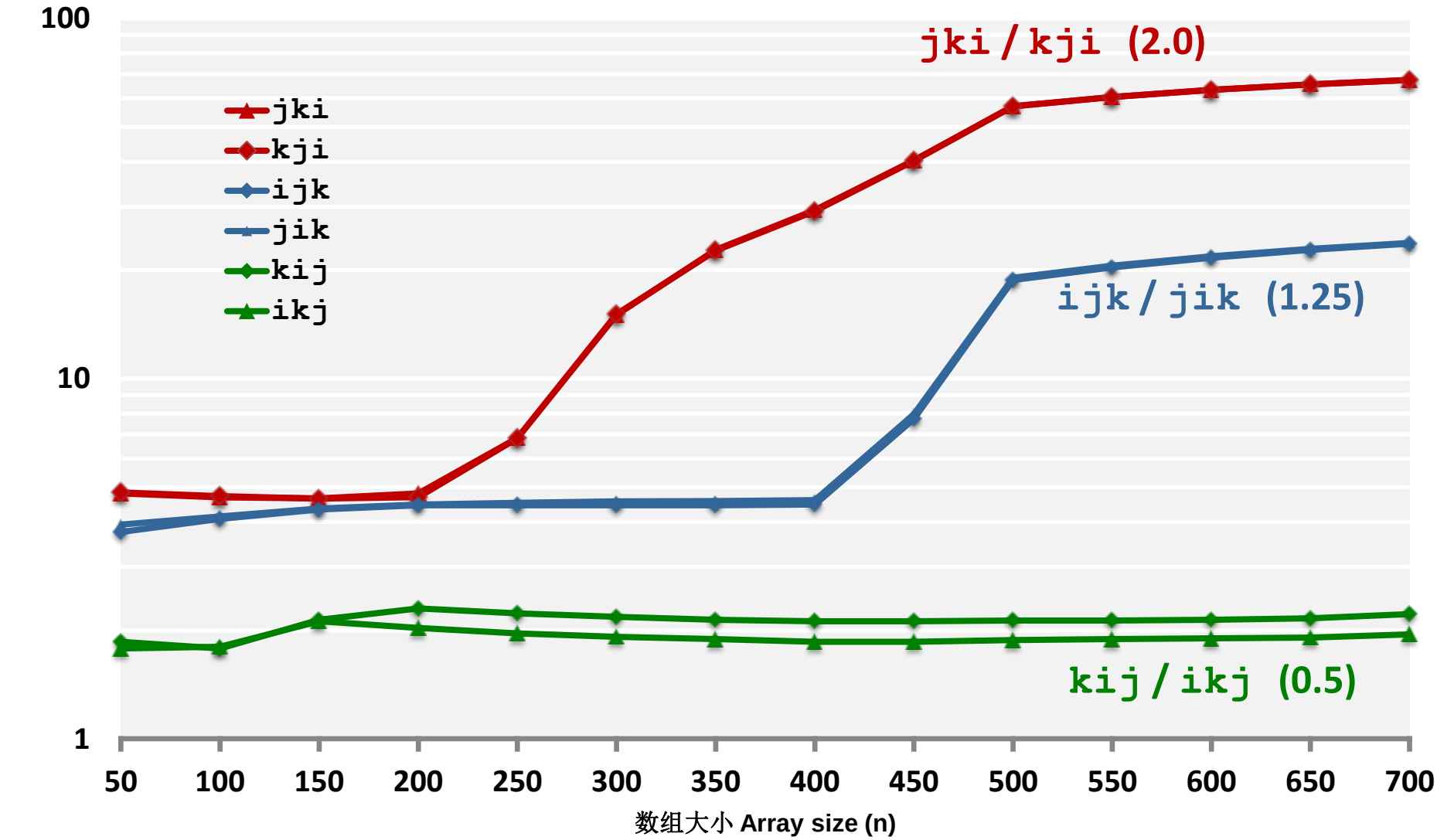
- **2条装载，1条存储** 2 loads, 1 store
- 不命中/迭代 misses/iter = **0.5**

```
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

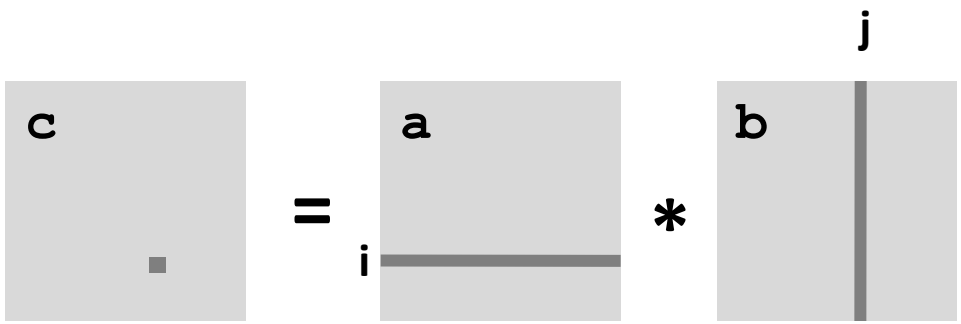


主要内容

- 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

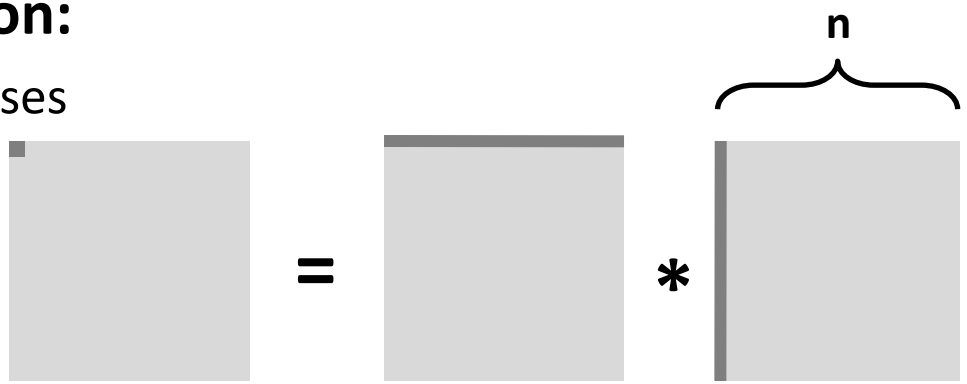
```
c = (double *) calloc(sizeof(double), n*n);

/* 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

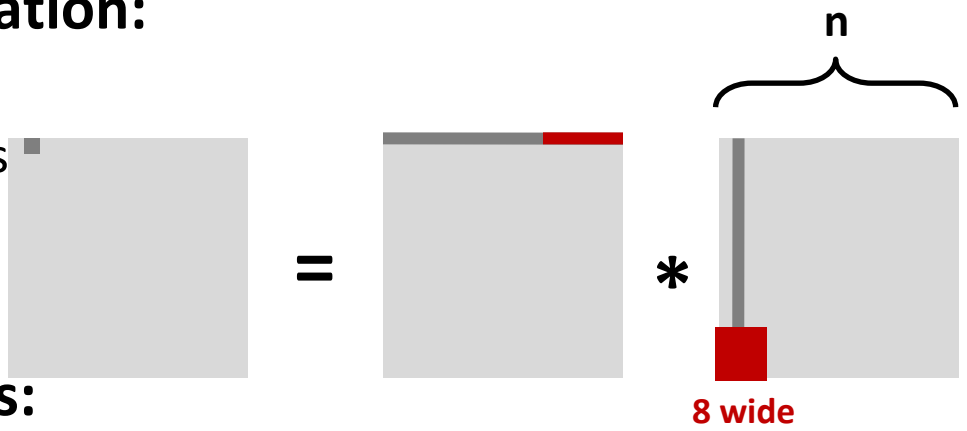


  - 后来缓存中： Afterwards in cache:
    (简图 schematic)

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

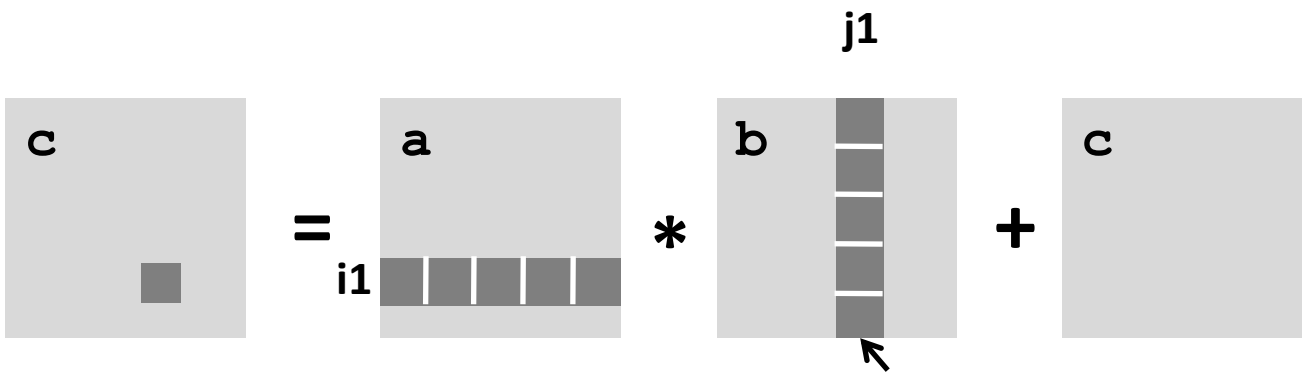


- **总计不命中 Total misses:**
  - $9n/8 * n^2 = (9/8) * n^3$

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];
}
```

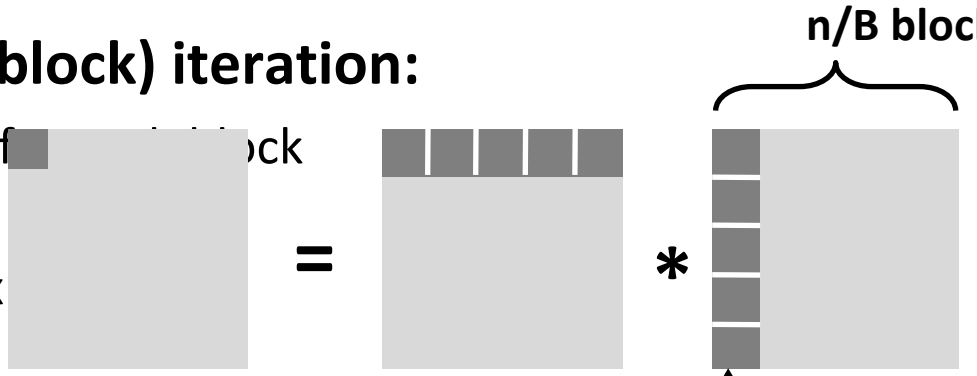
matmult/bmm.c



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

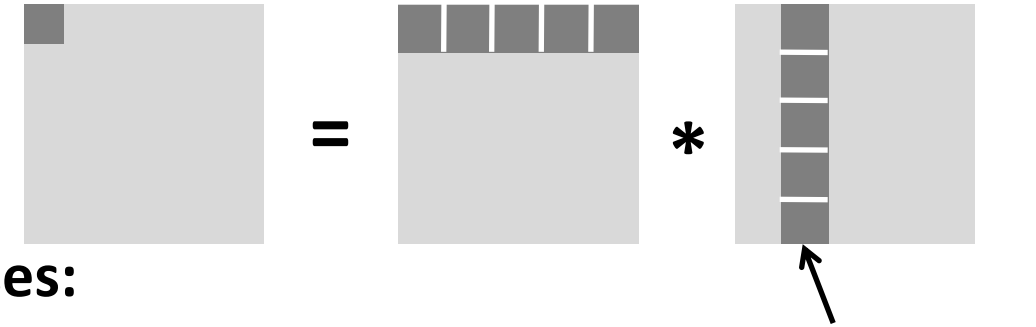


  - 后来缓存中 Afterwards in c[illegible]
    (简图 schematic)

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

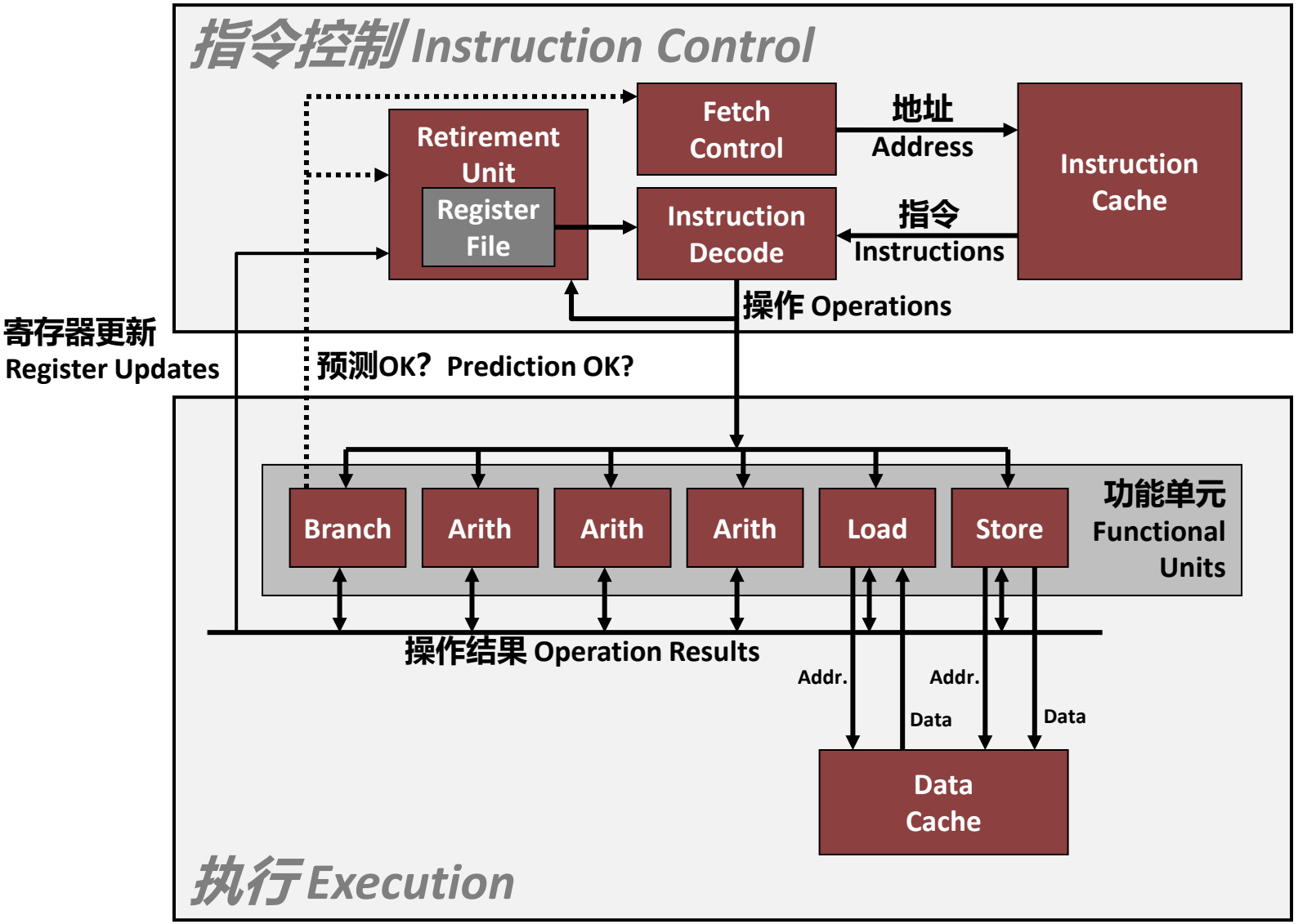
Cache总结 Cache Summary

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

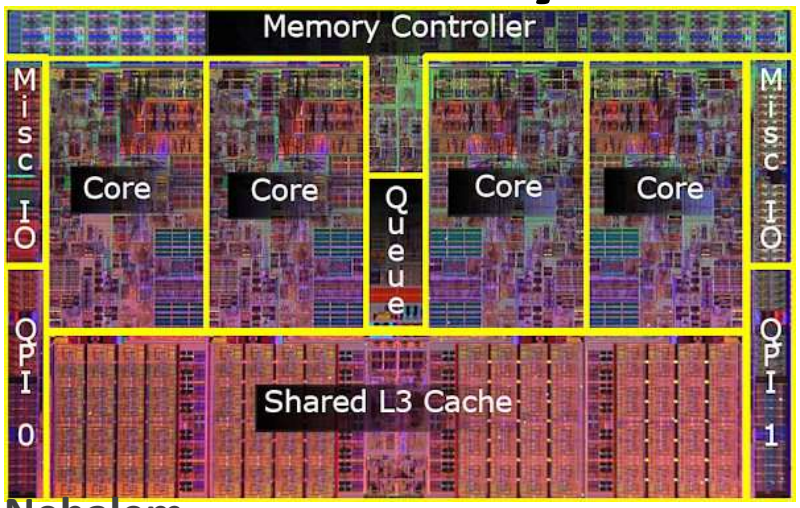
- **编写程序时需要充分利用 You can write your programs to exploit this!**
  - 关注内层循环，其中有大量计算和访存 Focus on the inner loops, where bulk of computations and memory accesses occur.
  - 充分利用空间局部性，按顺序连续读取数据对象 Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - 充分利用时间局部性，一次读入多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

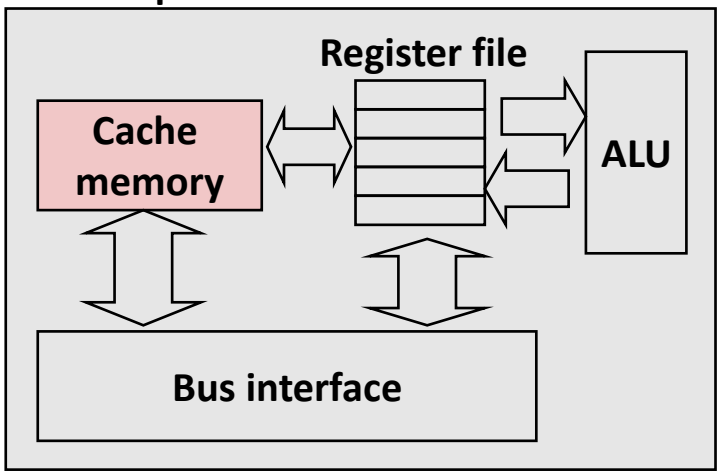


CPU内部真实是什么样?
What it Really Looks Like

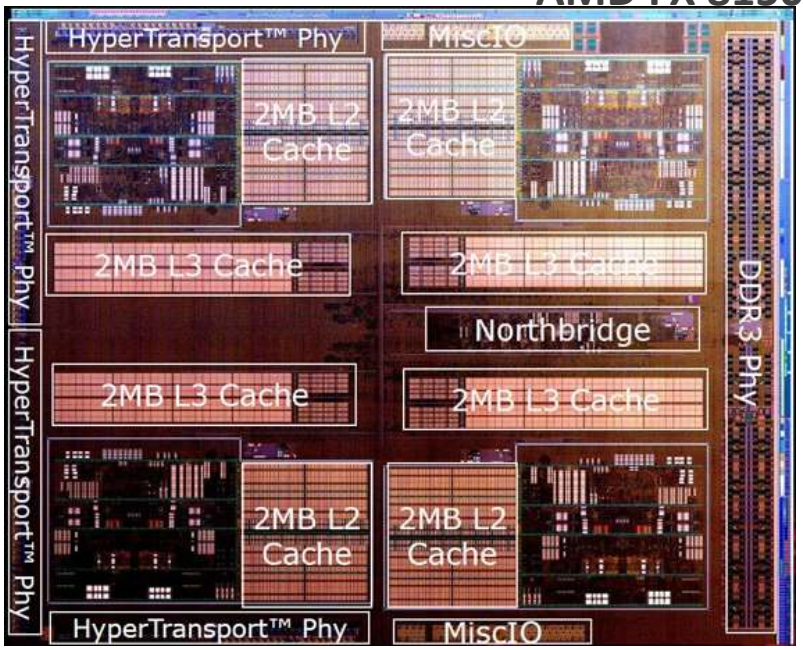


Nehalem

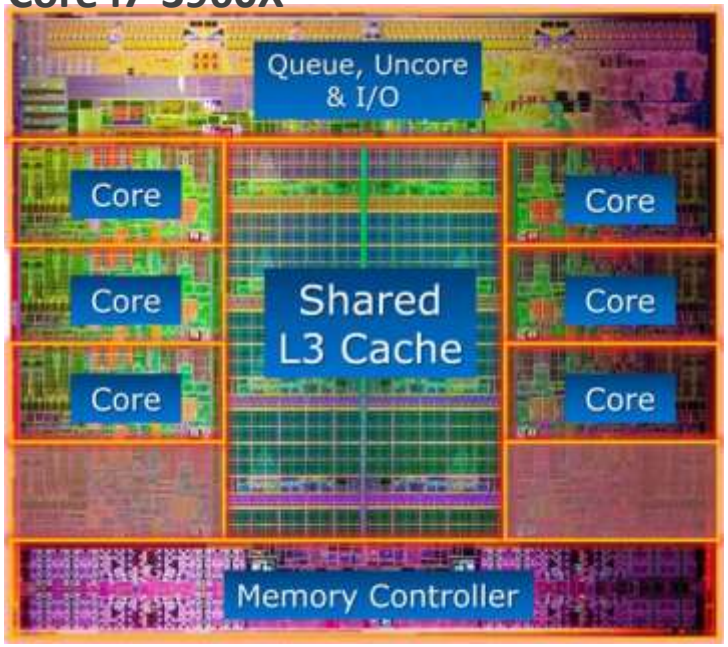
CPU chip



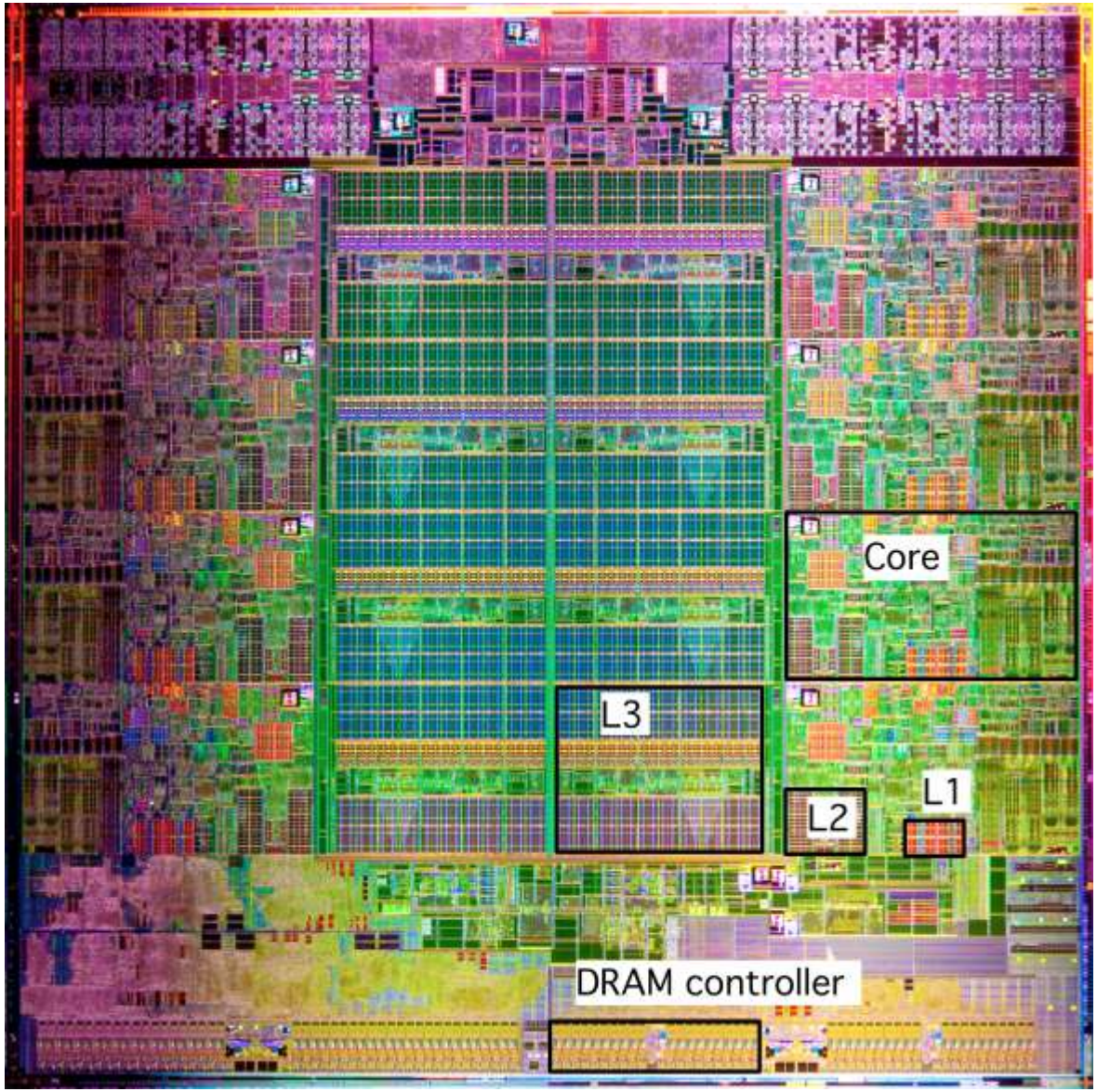
AMD FX 8150



Core i7-3960X



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



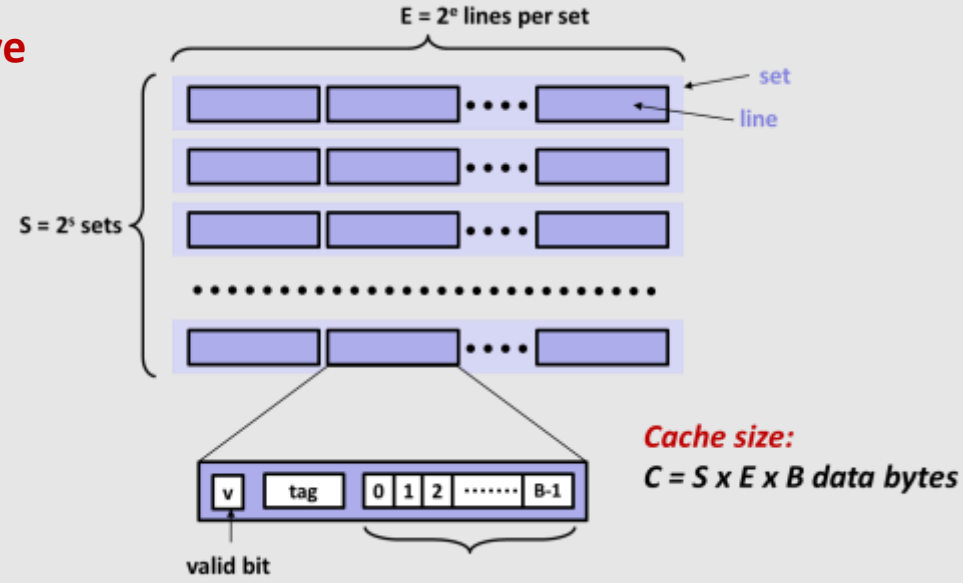
Intel Sandy Bridge
Processor Die

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

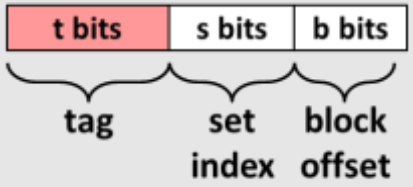
32 kB 8-way set associative
64 bytes/block
47 bit address range

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

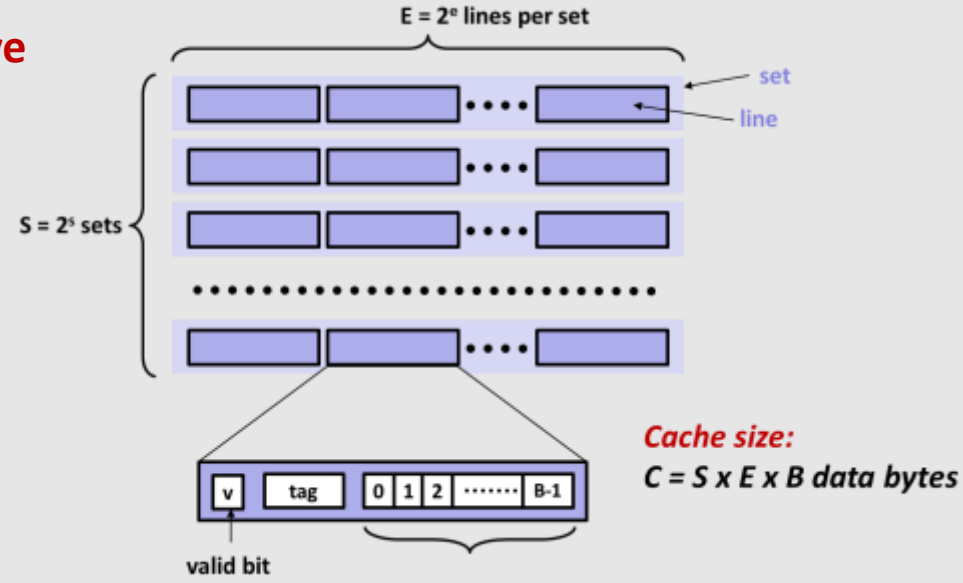
Stack Address:
0x00007f7262a1e010

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

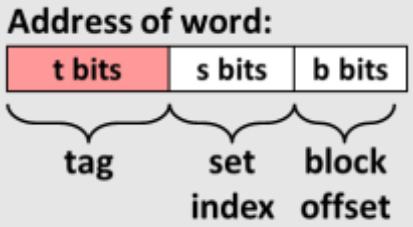
示例： Core i7 L1 数据高速缓存
Example: Core i7 L1 Data Cache

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



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

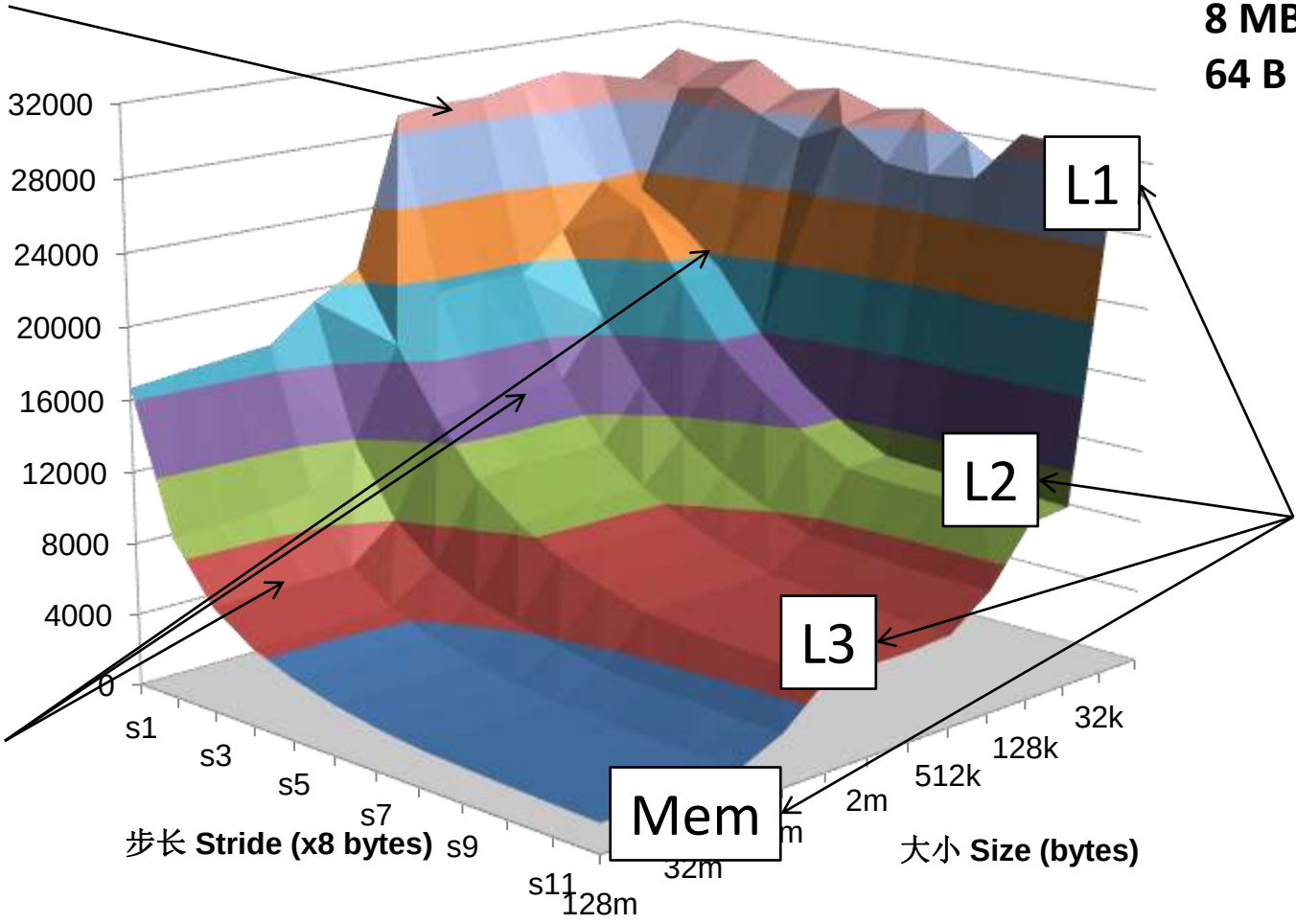
Stack Address:
0x00007f7262a1e010

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

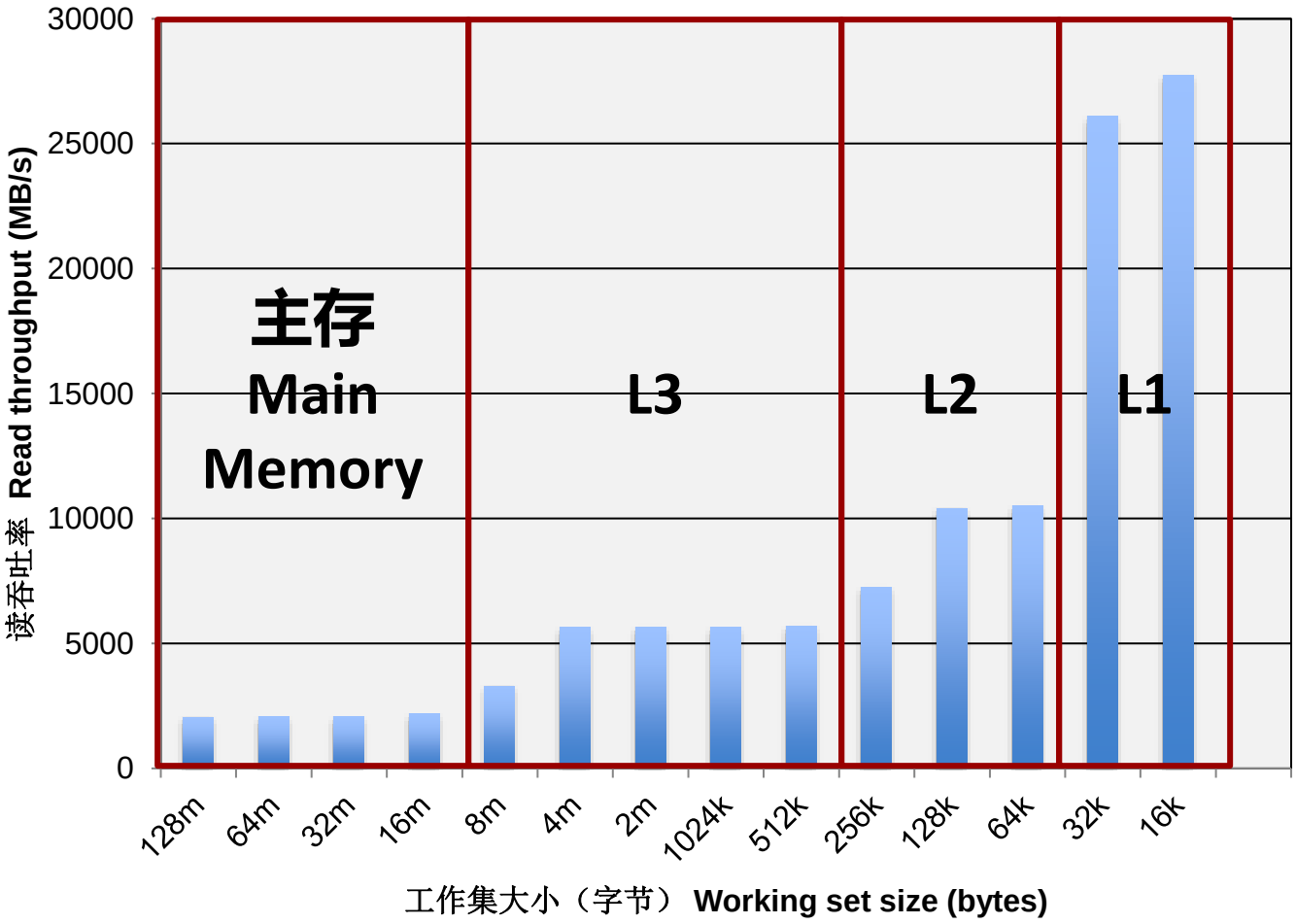


主动预取
Aggressive prefetching

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



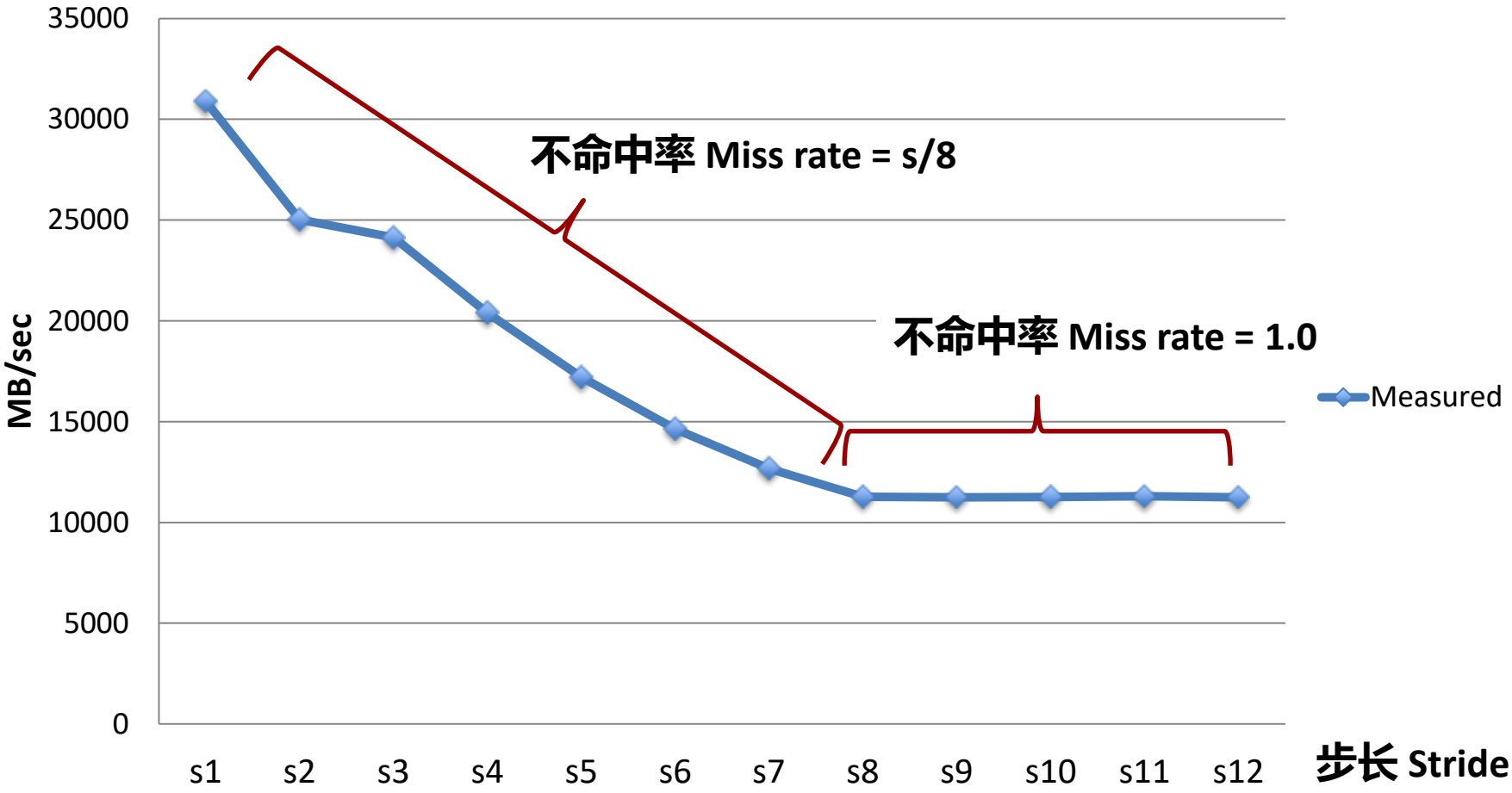
Core i7 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

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

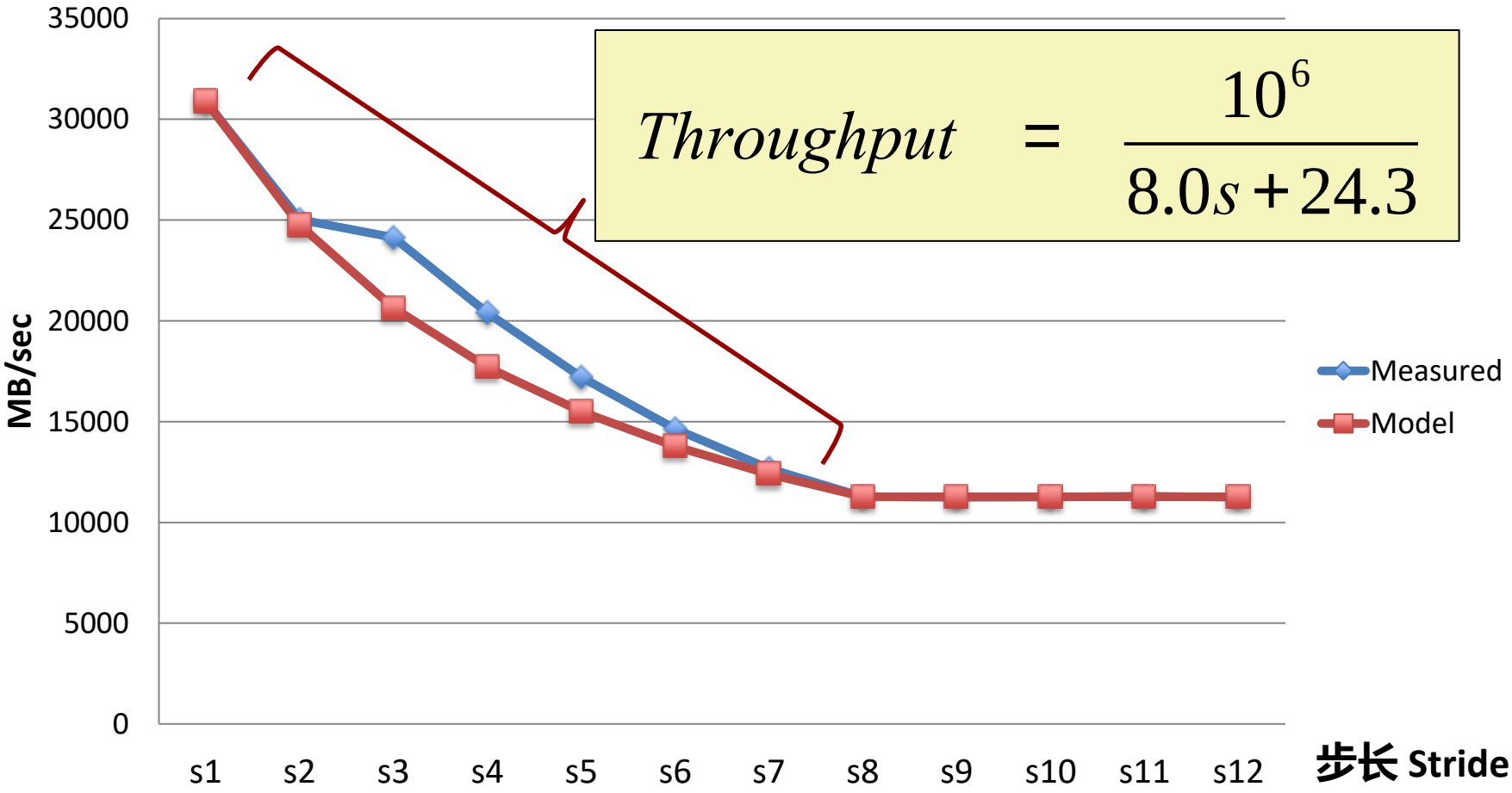
大小为128K时的吞吐量 Throughput for size = 128K



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