高速缓存

Cache Memories

本章内容

Topic

- 高速缓存的结构和工作原理

  Cache memory organization and operation

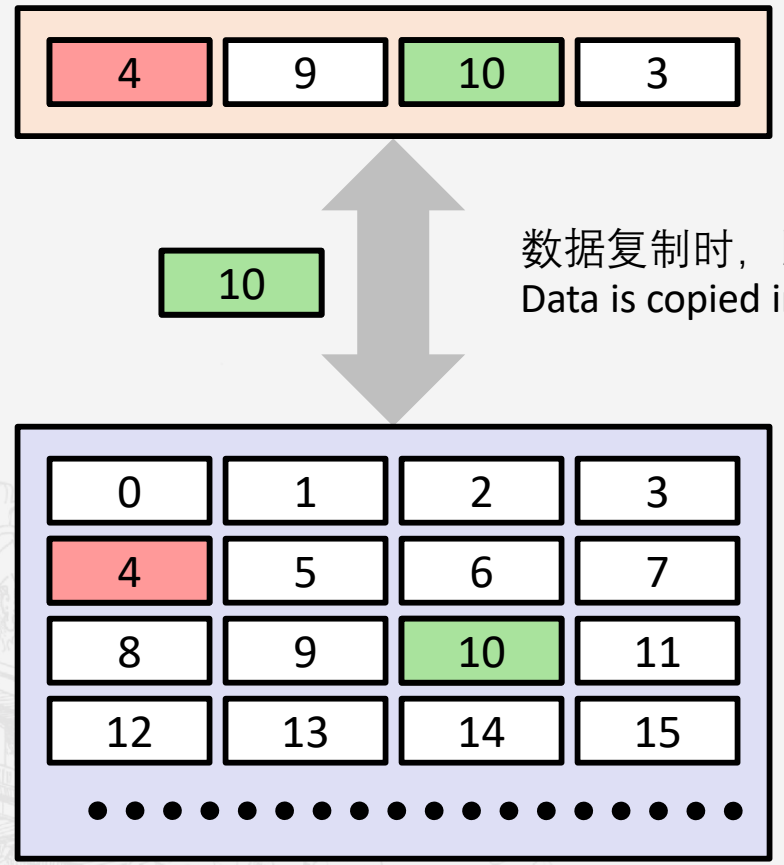
- 高速缓存对软件性能的影响

  Performance impact of caches

高速缓存的结构和工作原理

Cache memory organization and operation

缓存的基本概念 （回顾）
General Cache Concepts (Reminder)



更小，更快，成本更高，缓存内存数据块的子集
Smaller, faster, more expensive
memory caches a subset of the blocks

数据复制时，以“块”作为基本传输单元
Data is copied in block-sized transfer units

更大，更慢，成本更低的内存，在逻辑上内存可以被划分为多个数据块
Larger, slower, cheaper memory
viewed as partitioned into “blocks”

高速缓存的结构和工作原理

Cache memory organization and operation

几个缓存设计原则
Many common design issues

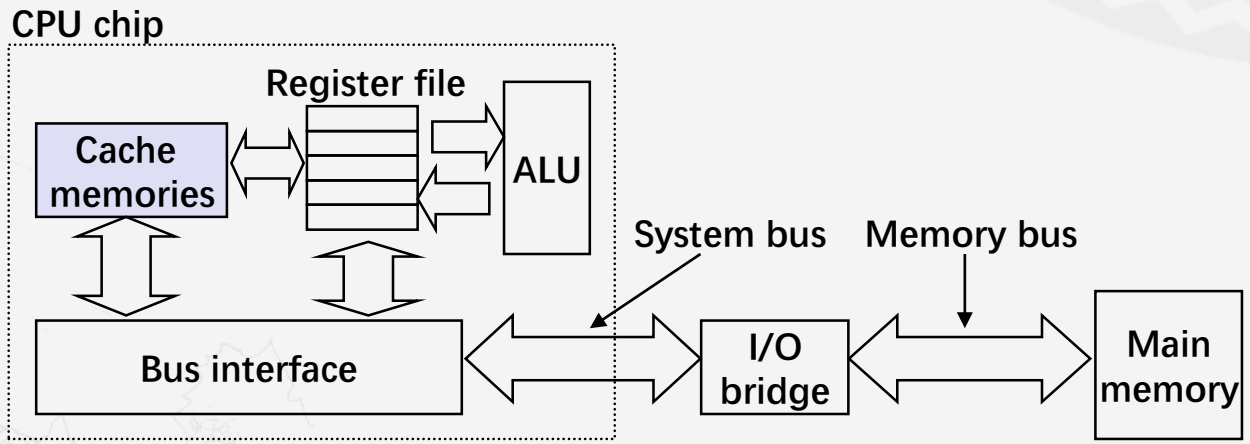
- 每一个缓存项都要包含“标签”（ID）和“内容”
  Each cached item has a “tag” (an ID) plus contents
- 需要一种快速有效的机制用来判断给定的数据是否被缓存
  Need a mechanism to efficiently determine whether given item is cached
  - 通过对有效位置建立索引和约束规则
    Combinations of indices and constraints on valid locations
- 未命中时，通常要选择一个现有缓存项，并被新缓存项所替代
  On a miss, usually need to pick something to replace with the new item
  - 称之为“替换策略”
    called a “replacement policy”
- 在写入时，需要传播更改的内容，或将该缓存项标记为“脏”
  On writes, need to either propagate change or mark item as “dirty”
  - 直写与回写
    write-through vs. write-back
- 不同的缓存场景采用不同的解决方案。我们以CPU中的高速缓存为例进行说明……
  Different solutions for different caches. Lets talk about CPU caches as a concrete example…

高速缓存的结构和工作原理

Cache memory organization and operation

CPU高速缓存
CPU Cache Memories

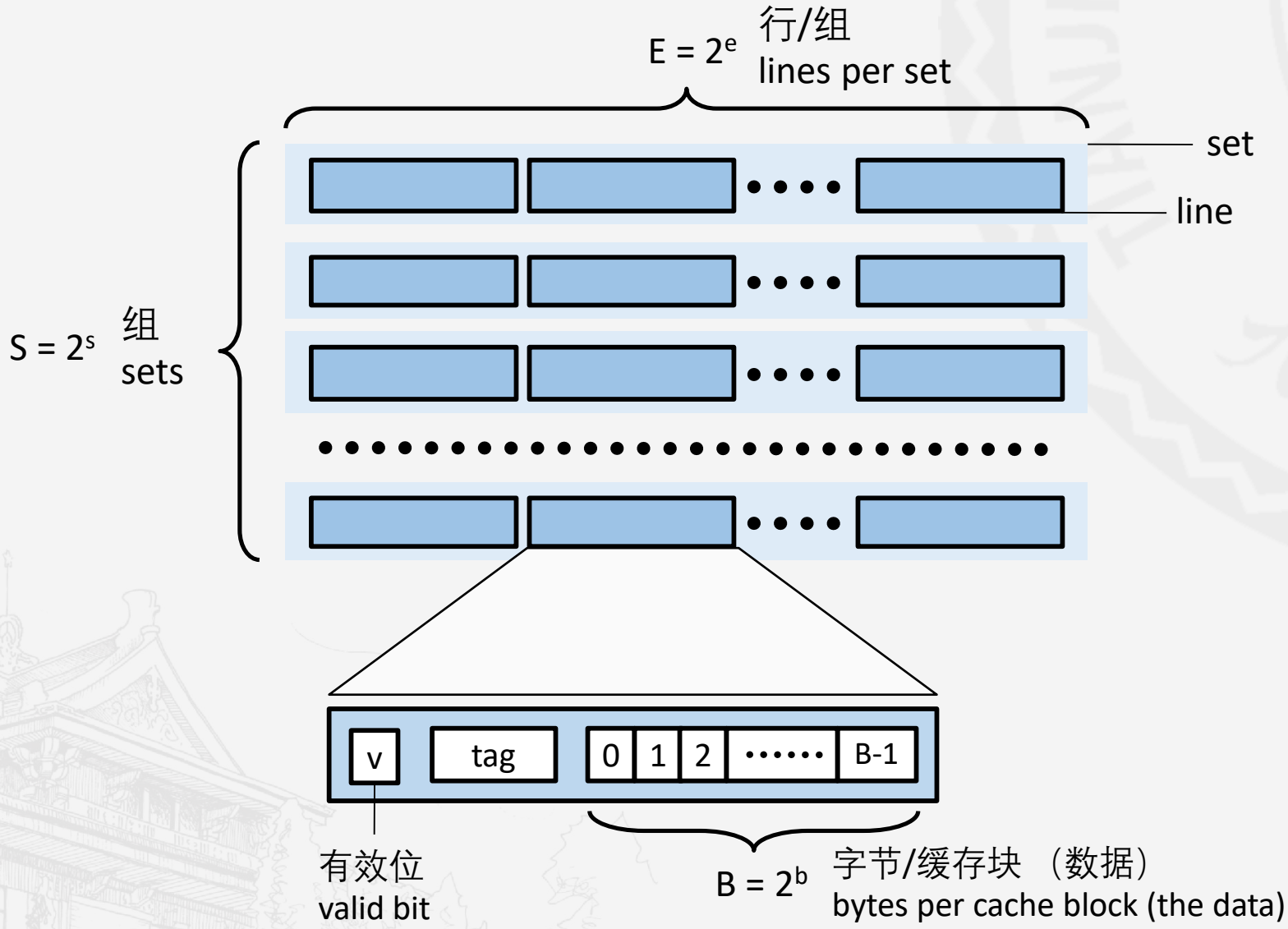
- CPU高速缓存是一个小型的，快速的，基于SRAM技术的存储器，由硬件自动控制（对用户透明）
  CPU Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - 用于保存频繁访问的主存数据块
    Hold frequently accessed blocks of main memory
- CPU首先在缓存（例如，L1、L2和L3)中查找数据，然后才会在主存中查找数据
  CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory
- 典型的系统结构:
  Typical system structure:



高速缓存的结构和工作原理

Cache memory organization and operation

通用的高速缓存组织结构 (S, E, B)
General Cache Organization (S, E, B)



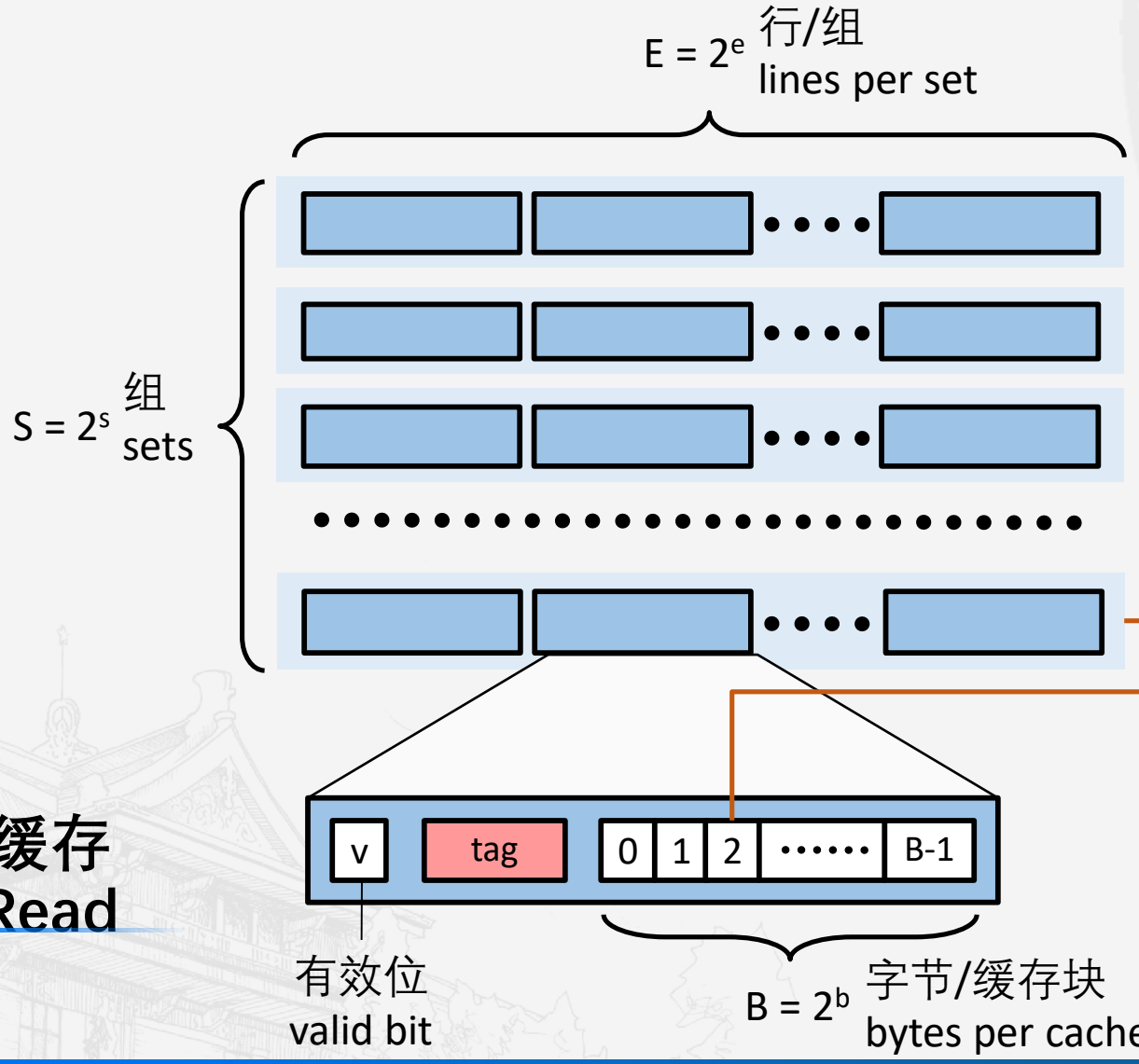
缓存容量

Cache size:

C = S x E x B bytes

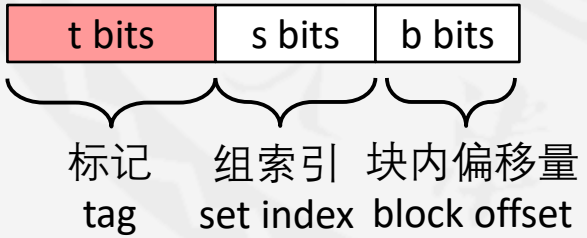
高速缓存的结构和工作原理

Cache memory organization and operation



- 确定组的位置
  Locate set
- 检查是否有组中的行能够匹配标记
  Check if any line in set has matching tag
- 匹配 且 行的有效位有效： 命中
  Yes + line valid: hit
- 根据偏移量定位数据在行中的起始位置
  Locate data starting at offset

字地址
Address of word:



数据起始于这个偏移量
data begins at this offset

读高速缓存
Cache Read

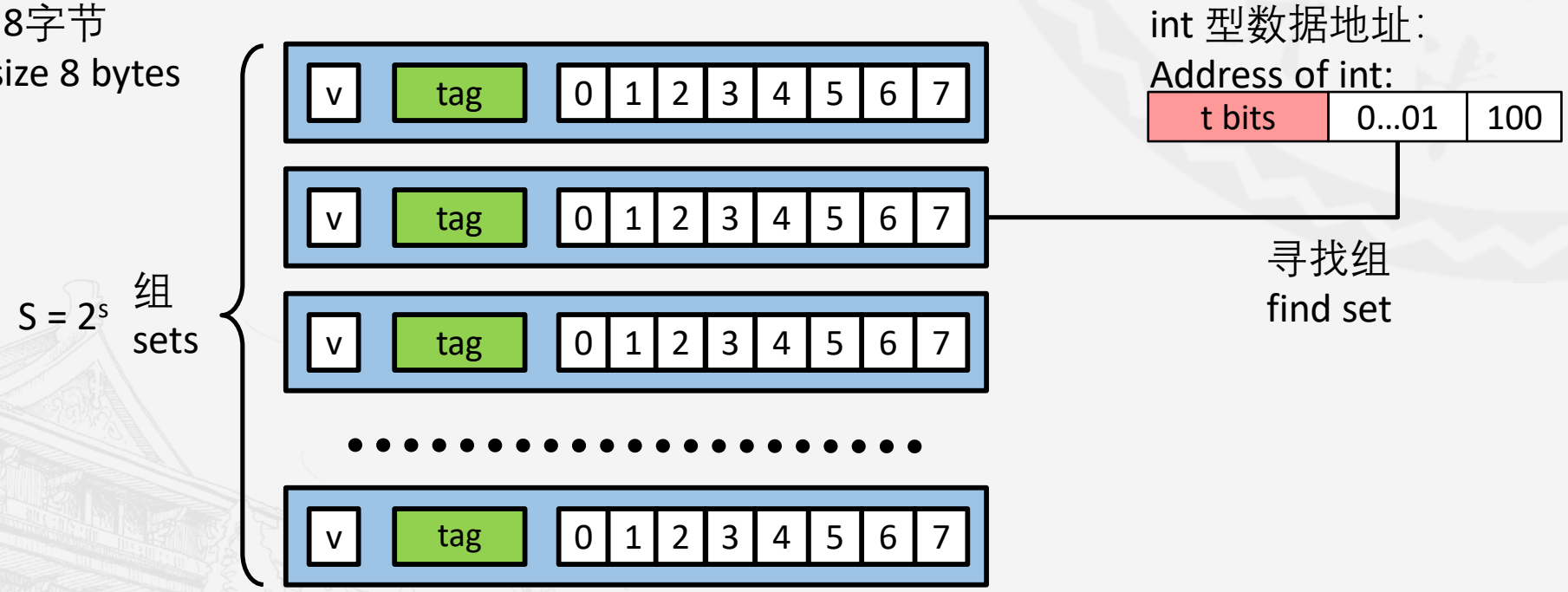
高速缓存的结构和工作原理

Cache memory organization and operation

示例：直接映射高速缓存 （E=1）
Example: Direct Mapped Cache (E = 1)

直接映射：每组只有一行
Direct mapped: One line per set

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



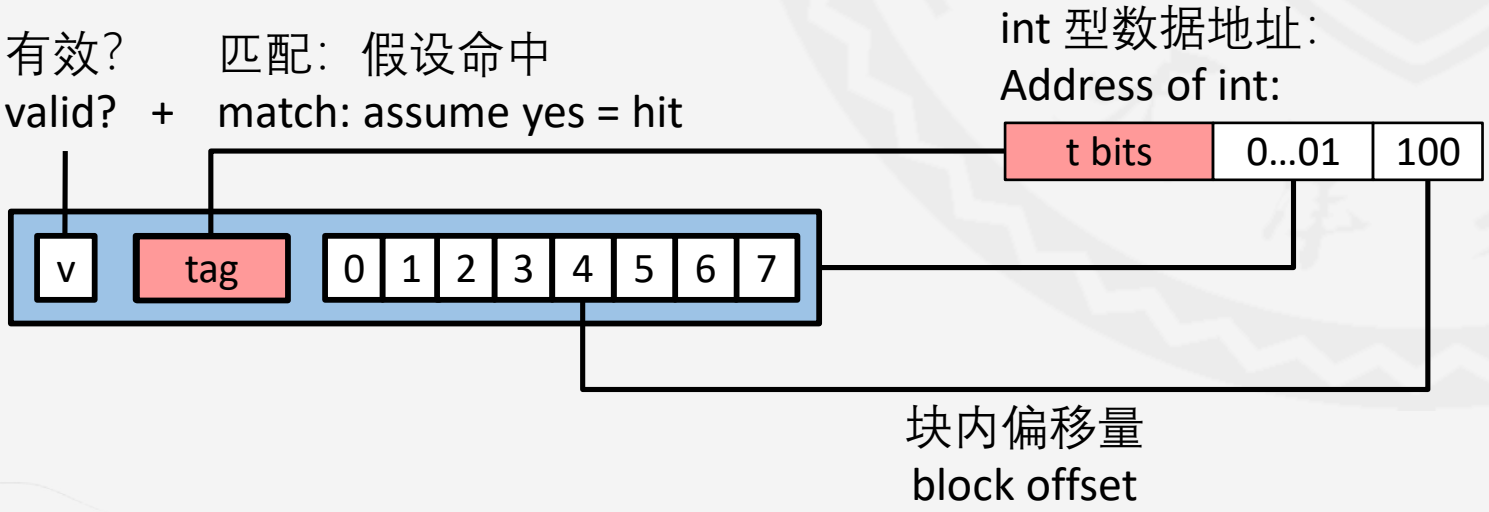
高速缓存的结构和工作原理

Cache memory organization and operation

示例：直接映射高速缓存 （E=1）
Example: Direct Mapped Cache (E = 1)

直接映射：每组只有一行
Direct mapped: One line per set

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



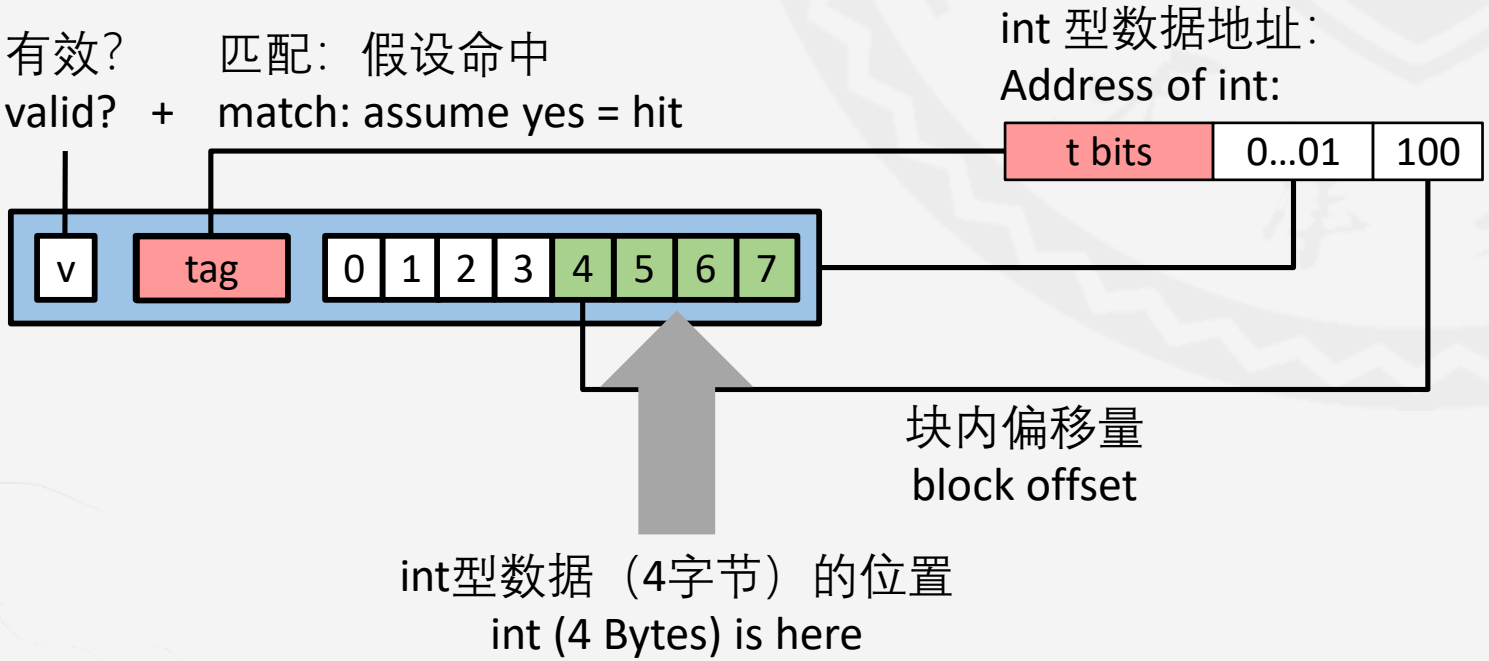
高速缓存的结构和工作原理

Cache memory organization and operation

示例：直接映射高速缓存 （E=1）
Example: Direct Mapped Cache (E = 1)

直接映射：每组只有一行
Direct mapped: One line per set

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



如果标记未匹配：原有的行将会被回收并替换
If tag doesn’t match: old line is evicted and replaced

高速缓存的结构和工作原理

Cache memory organization and operation

直接映射高速缓存模拟
Direct Mapped Cache Simulation

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

Assume（假设）：

- 4-bit addresses
- M=16 bytes
- B=2 bytes/block,
- S=4 sets
- E=1 Blocks/set

地址跟踪（读操作，每次读1字节）

Address trace (reads, one byte per read):

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

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

高速缓存的结构和工作原理

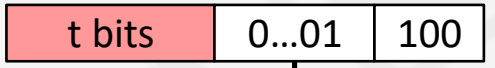
Cache memory organization and operation

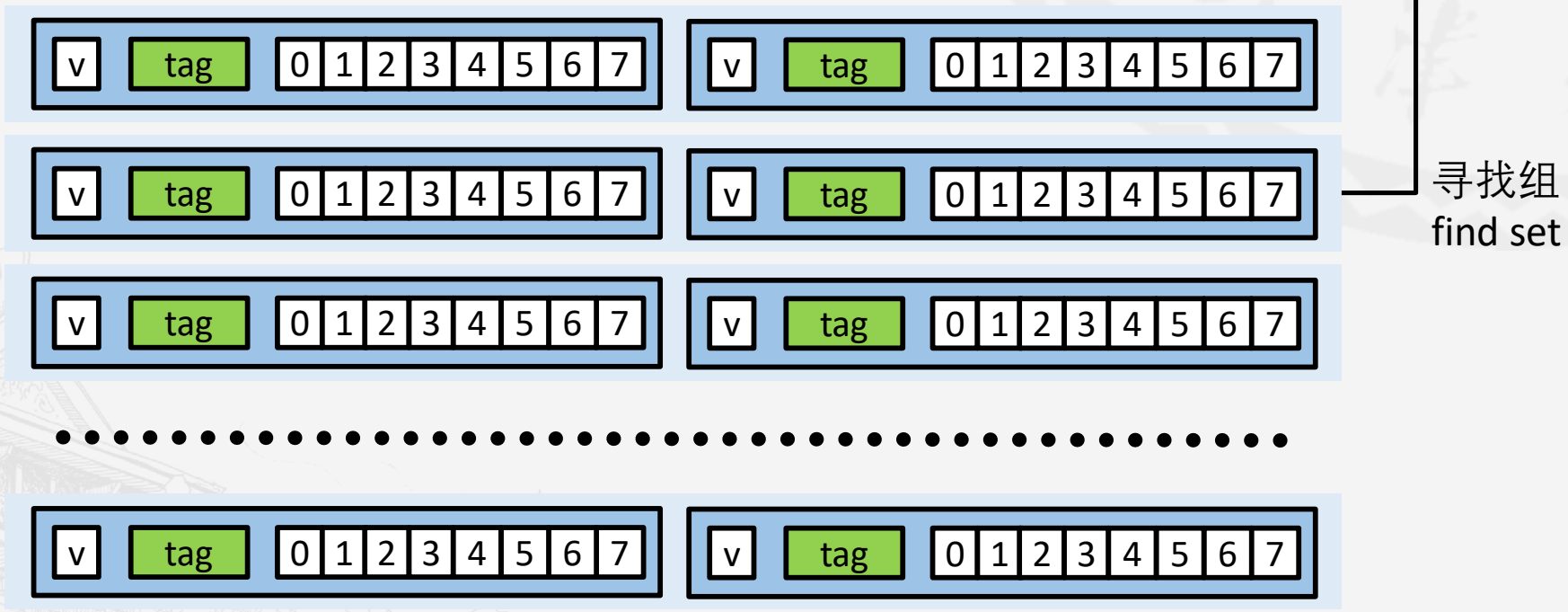
示例：E路组相联高速缓存 （假设：E=2）
Example: E-way Set Associative Cache (Assume: E=2)

E = 2: 每组两行
E = 2: Two lines per set

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

short int型数据地址
Address of short int:





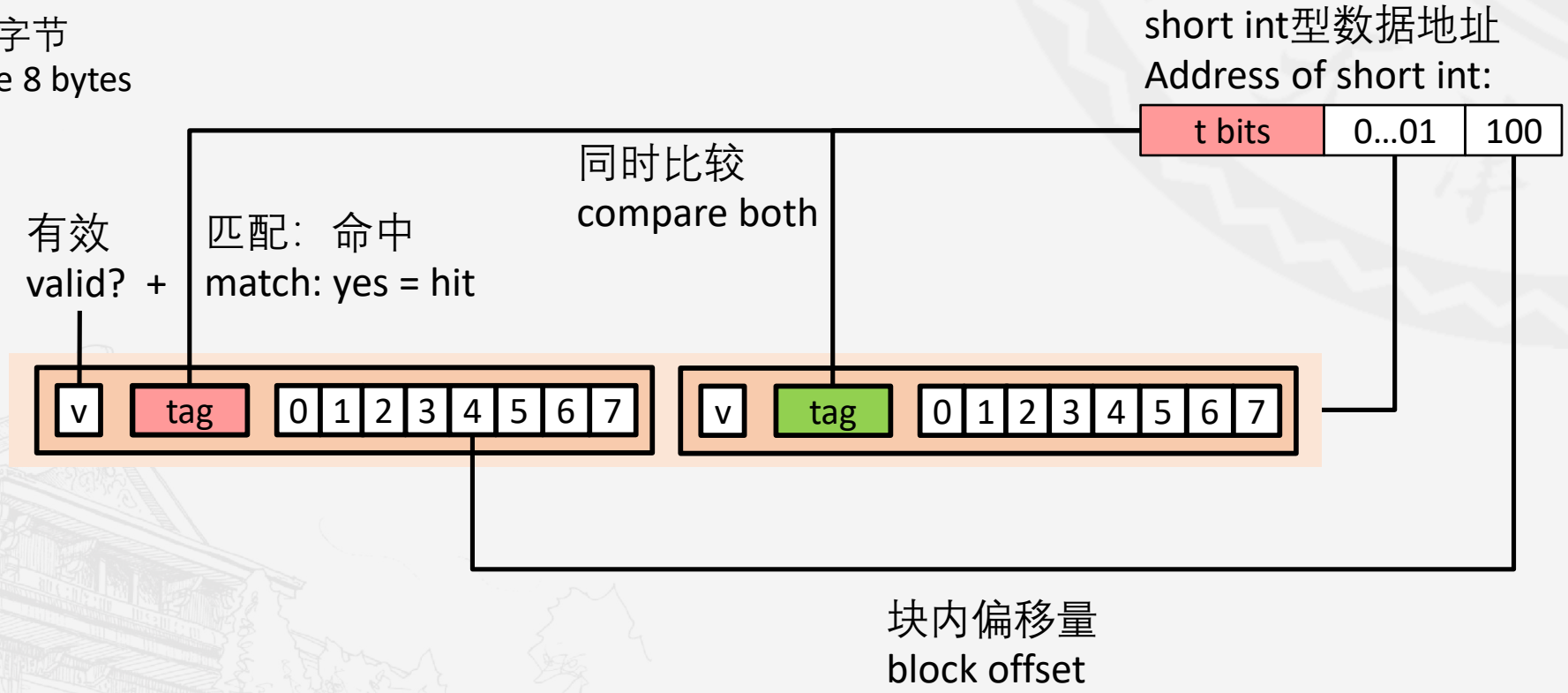
高速缓存的结构和工作原理

Cache memory organization and operation

示例：E路组相联高速缓存 （假设：E=2）
Example: E-way Set Associative Cache (Assume: E=2)

E = 2: 每组两行
E = 2: Two lines per set

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



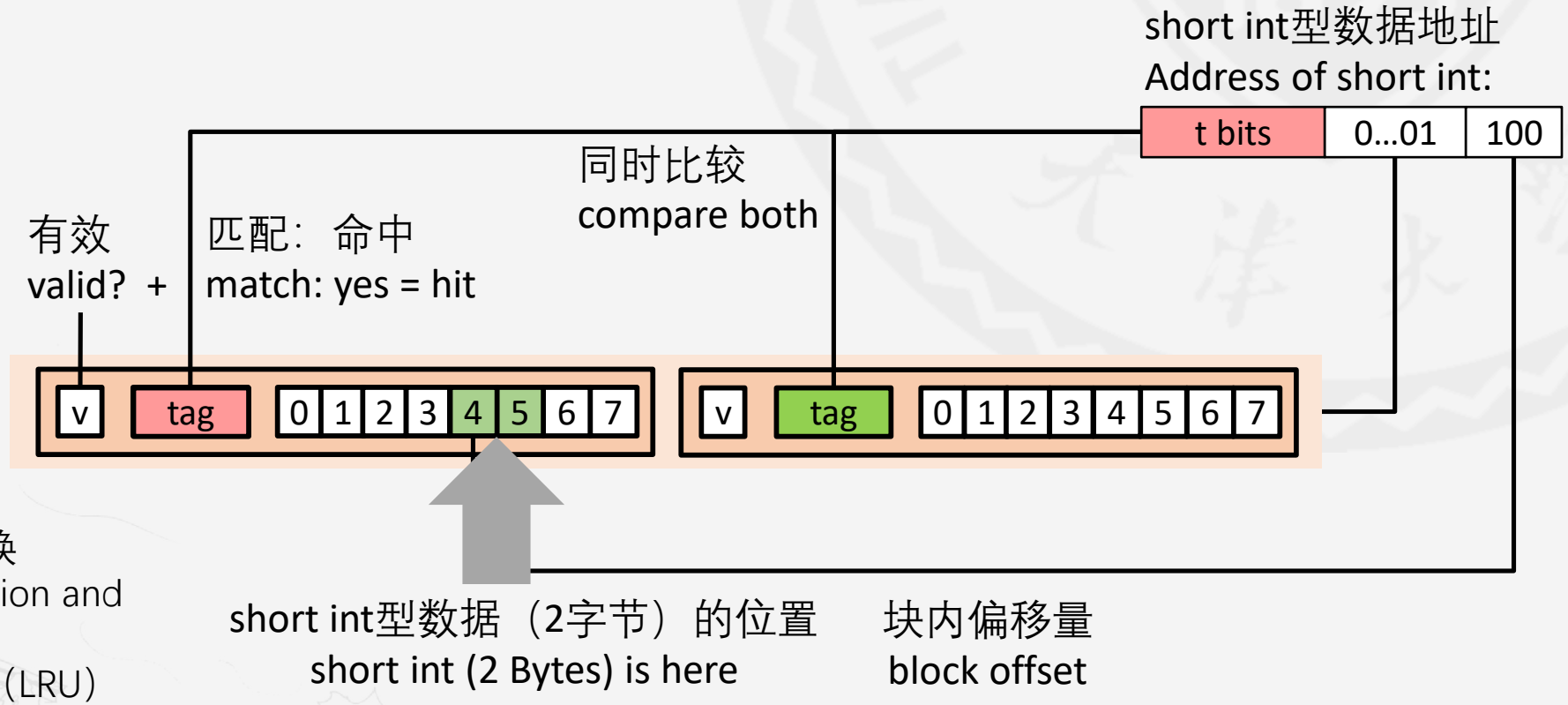
高速缓存的结构和工作原理

Cache memory organization and operation

示例：E路组相联高速缓存 （假设：E=2）
Example: E-way Set Associative Cache (Assume: E=2)

E = 2: 每组两行
E = 2: Two lines per set

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



没有成功匹配：
No match:

- 选择当前组中的一行被回收和替换
  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), …

高速缓存的结构和工作原理

Cache memory organization and operation

2路组相联高速缓存模拟
2-Way Set Associative Cache Simulation

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

Assume（假设）：

- 4-bit addresses
- M=16 bytes
- B=2 bytes/block,
- S=2 sets
- E=2 Blocks/set

地址跟踪（读操作，每次读1字节）

Address trace (reads, one byte per read):

0	[00$\underline{0}$0$_2$],	miss
1	[00$\underline{0}$1$_2$],	hit
7	[01$\underline{1}$1$_2$],	miss
8	[10$\underline{0}$0$_2$],	miss
0	[00$\underline{0}$0$_2$]	hit

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

高速缓存的结构和工作原理

Cache memory organization and operation

- 数据在存储系统中存在多份副本:
  Multiple copies of data exist:
  - 一级缓存，二级缓存，主存和磁盘
    L1, L2, Main Memory and Disk
- 怎么处理写命中的情况?
  What to do on a write-hit?
  - 直写 （立即写入主存）
    Write-through (write immediately to memory)
  - 回写 （至行被替换时才写入主存）
    Write-back (defer write to memory until replacement of line)
- 怎么处理写未命中的情况?
  What to do on a write-miss?
  - 写分配 （加载至缓存，然后更新缓存中的行）
    Write-allocate (load into cache, update line in cache)
    - 如果随后有更多在这个位置附近的写操作，会显著提升性能
      Good if more writes to the location follow
  - 非写分配 （直接写入内存，不加载数据块至缓存）
    No-write-allocate (writes straight to memory, does not load into cache)

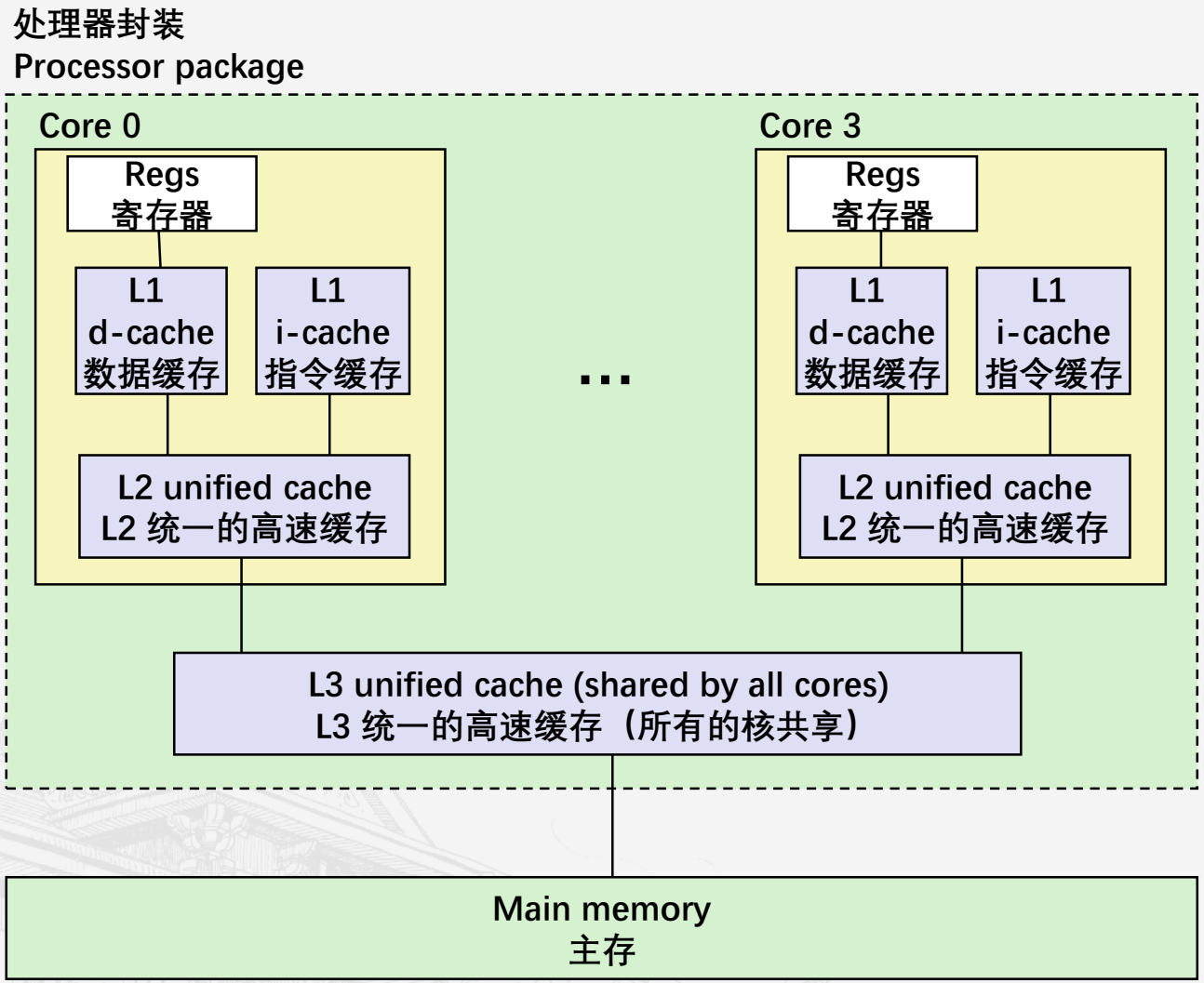
怎么处理写操作?
What about writes?

典型的策略组合:
Typical policy combination:

	写命中 Write-hit	写未命中 Write-miss
效率优先 Efficiency first	回写 Write-back	写分配 Write-allocate
可靠性优先 Reliability first	直写 Write-through	非写分配 No-write-allocate

高速缓存的结构和工作原理

Cache memory organization and operation



Intel 酷睿 i7处理器高速缓存层次结构
Intel Core i7 Cache Hierarchy

L1 i-cache and d-cache:
32 KB, 8-way（8 lines per set）,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

高速缓存的结构和工作原理

Cache memory organization and operation

高速缓存的性能指标
Cache Performance Metrics

未命中率
Miss Rate

- 在缓存中找不到所引用的内存数据的百分比（未命中次数/总访问次数）= 1 – 命中率
  Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
- 典型值（百分数）：
  Typical numbers (in percentages):
  - 一级缓存：3% ~ 10%
    3-10% for L1
  - 在二级缓存中这个值非常小（例如：<1%），取决于缓存容量等因素
    can be quite small (e.g., < 1%) for L2, depending on size, etc.

命中时间
Hit Time

- 将缓存中的行数据发送到处理器所需的时间
  Time to deliver a line in the cache to the processor
  - 包括判定该行是否在缓存中的时间
    includes time to determine whether the line is in the cache
- 典型值：
  Typical numbers:
  - 一级缓存：1-2个周期
    1-2 clock cycle for L1
  - 二级缓存：5-20个周期
    1-2 clock cycle for L1

高速缓存的结构和工作原理

Cache memory organization and operation

高速缓存的性能指标
Cache Performance Metrics

- **未命中惩罚**
  Miss Penalty
  - 由于缓存未命中所需要的额外数据访问时间
    Additional time required because of a miss
  - 典型值：
    Typical numbers:
    - 主存：50-200个时钟周期
      50-200 cycles for main memory
    - 趋势还在扩大!
      Trend: increasing!

这就是为什么用“未命中率”而不使用“命中率”作为性能指标的原因了
This is why “miss rate” is used instead of “hit rate”

- **命中与未命中的巨大性能差异**
  Huge difference between a hit and a miss
  - 如果仅比较一级缓存和内存的话，可能相差达到100倍
    Could be 100x, if just L1 and main memory
- **你相信吗？99%命中率的缓存系统效率，两倍于97%命中率的缓存系统**
  Would you believe 99% hits is twice as good as 97%?
  - 考虑下面的情况：命中时间1个周期，未命中惩罚100个周期
    Consider: cache hit time of 1 cycle, miss penalty of 100 cycles
  - 平均访问时间：
    Average access time

97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**
99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**

高速缓存的结构和工作原理

Cache memory organization and operation

编写缓存友好的代码
Writing Cache Friendly Code

- 可以让程序在常见的场景下更快的运行
  Make the common case go fast
  - 关注核心函数的内部循环
    Focus on the inner loops of the core functions
- 减少内部循环中的缓存未命中
  Minimize the misses in the inner loops
  - 对变量更好的重复利用（时间局部性）
    Repeated references to variables are good (temporal locality)
  - 尽量使用步长为1的模式访问存储器（空间局部性）
    Stride-1 （步长为1）reference patterns are good (spatial locality)
- 核心思想：通过理解高速缓存的工作机制，量化我们对局部性原理的定性认知
  Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

本章内容

Topic

- 高速缓存的结构和工作原理
  Cache memory organization and operation
- 高速缓存对软件性能的影响
  Performance impact of caches
  - 通过循环重排提高空间局部性特征
    Rearranging loops to improve spatial locality
  - 通过分块提高时间局部性特征
    Using blocking to improve temporal locality
  - 存储器山
    The memory mountain

高速缓存对软件性能的影响

Performance impact of caches

矩阵乘法的未命中率分析
Miss Rate Analysis for Matrix Multiply

- 假设：

  Assume:

  - 每个矩阵是一个n×n的数组，数据类型为double，sizeof(double)==8

    Each matrix is an n×n array of double, with sizeof(double) == 8
  - 缓存块大小为32字节（可以容纳4个64位数据）

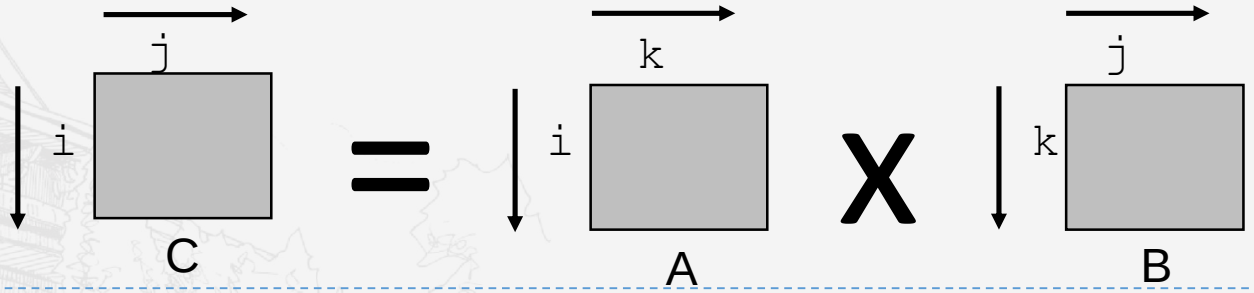
    Cache block size = 32Bytes (big enough for four 64-bit words)
  - 矩阵的维度(n)是一个非常大的值：1/N趋近于0

    Matrix dimension (N) is very large: Approximate 1/N as 0.0
  - 缓存的容量较小，不足以同时缓存矩阵中的多行数据

    Cache is not even big enough to hold multiple rows

- 分析方法：观察内部循环中的内存访问模式

  Analysis Method: Look at access pattern of inner loop



高速缓存对软件性能的影响

Performance impact of caches

举例：矩阵乘法
Matrix Multiplication Example

- n x n 矩阵相乘
  Multiply N x N matrices
- 共$O(N^3)$ 次操作（时间复杂度）
  $O(N^3)$ total operations
- 每个元素都需要进行N次读操作
  N reads per source element
- 每个目标元素都需要进行N次累加
  N values summed per destination
  - 这些累加可能会在寄存器中进行
    but may be able to hold 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;
  }
}
```

sum变量保存在寄存器中
Variable sum held in register

高速缓存对软件性能的影响

Performance impact of caches

C语言数组在内存中的布局（回顾）
Layout of C Arrays in Memory (review)

- C语言数组以“行优先”方式分配内存
  C arrays allocated in row-major order
  - 每一行的元素在内存中的位置都是连续的
    Each row in contiguous memory locations

- 在一行中，逐列进行访问
  Stepping through columns in one row:

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

  - 访问连续的元素
    accesses successive elements
  - 如果块大小B大于8字节，则可利用空间局部性
    if block size (B) > 8 bytes, exploit spatial locality
  - 未命中率 = 8/B
    miss rate = 8 bytes / B

- 在一列中，逐行进行访问
  Stepping through rows in one column:

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

  - 依次访问元素之间距离很远
    accesses distant elements
  - 没有空间局部性!
    no spatial locality!
  - 未命中率 = 1 （即100%）
    miss rate = 1 （i.e. 100%）

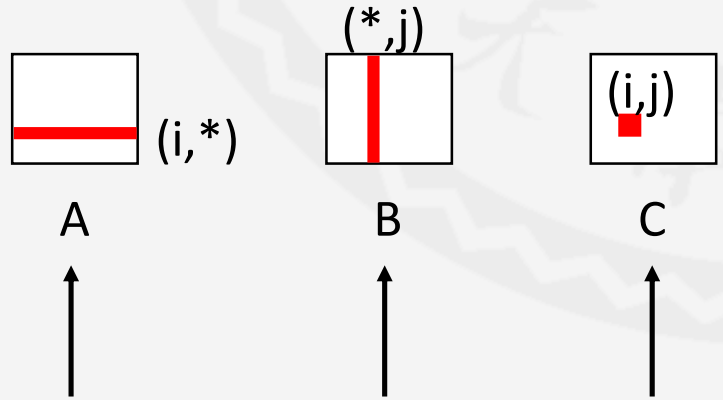
高速缓存对软件性能的影响

Performance impact of caches

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;
  }
}
```

内循环:
Inner loop:



每次内循环迭代的未命中次数
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

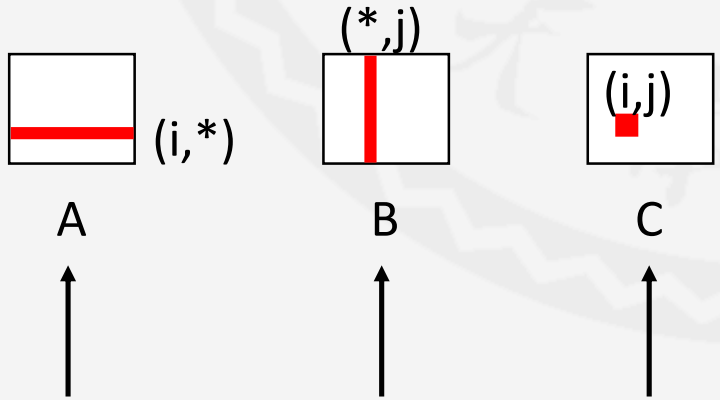
高速缓存对软件性能的影响

Performance impact of caches

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
  }
}
```

内循环:
Inner loop:



每次内循环迭代的未命中次数
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

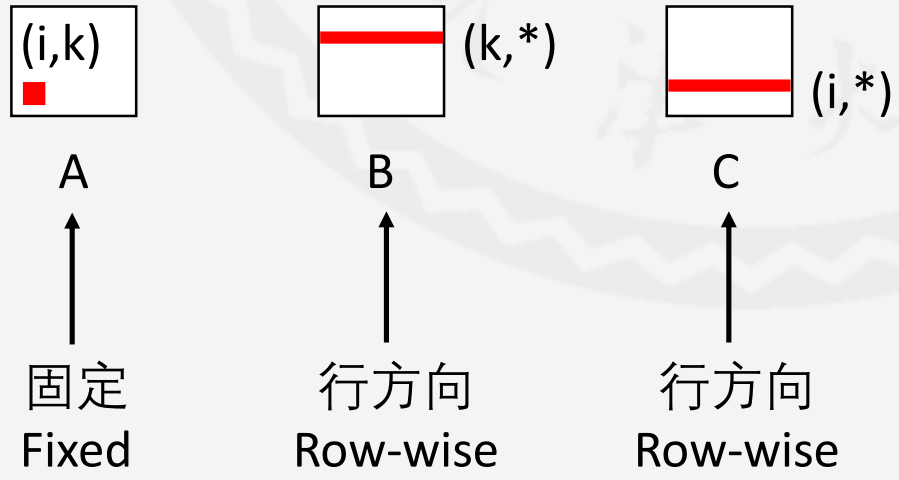
高速缓存对软件性能的影响

Performance impact of caches

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];
  }
}
```

内循环:
Inner loop:



每次内循环迭代的未命中次数
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

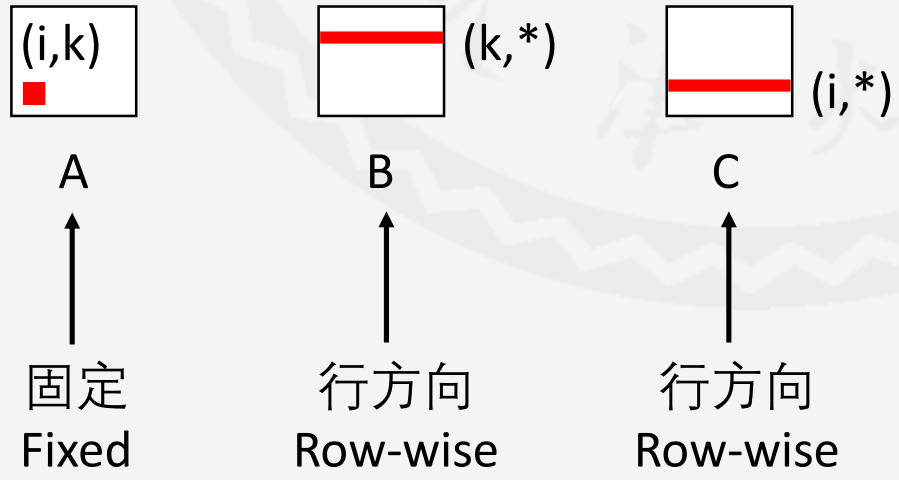
高速缓存对软件性能的影响

Performance impact of caches

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];
  }
}
```

内循环:
Inner loop:



每次内循环迭代的未命中次数
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

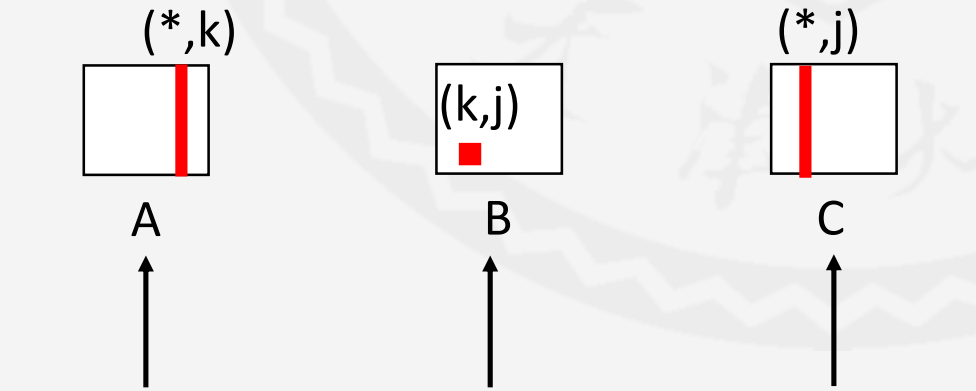
高速缓存对软件性能的影响

Performance impact of caches

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;
  }
}
```

内循环:
Inner loop:



每次内循环迭代的未命中次数
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

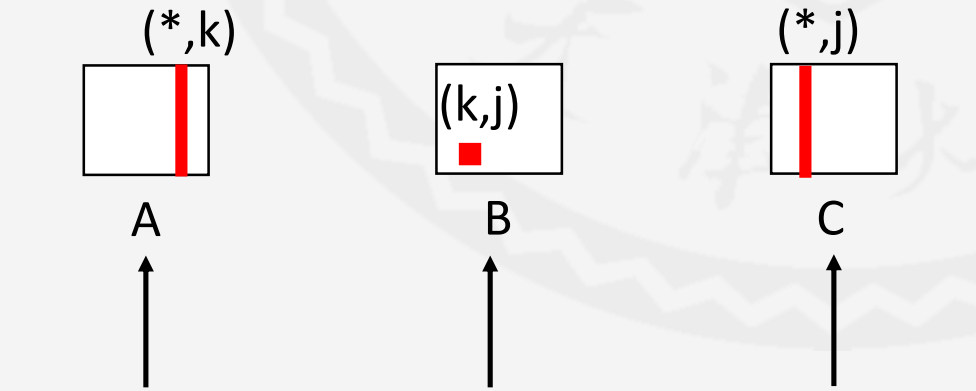
高速缓存对软件性能的影响

Performance impact of caches

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;
  }
}
```

内循环:
Inner loop:



每次内循环迭代的未命中次数
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

高速缓存对软件性能的影响

Performance impact of caches

矩阵乘法小结
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次内存加载，0次存储
  2 loads, 0 stores
- 未命中次数/循环 = 1.25
  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次内存加载，1次存储
  2 loads, 1 store
- 未命中次数/循环 = 0.5
  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;
  }
}
```

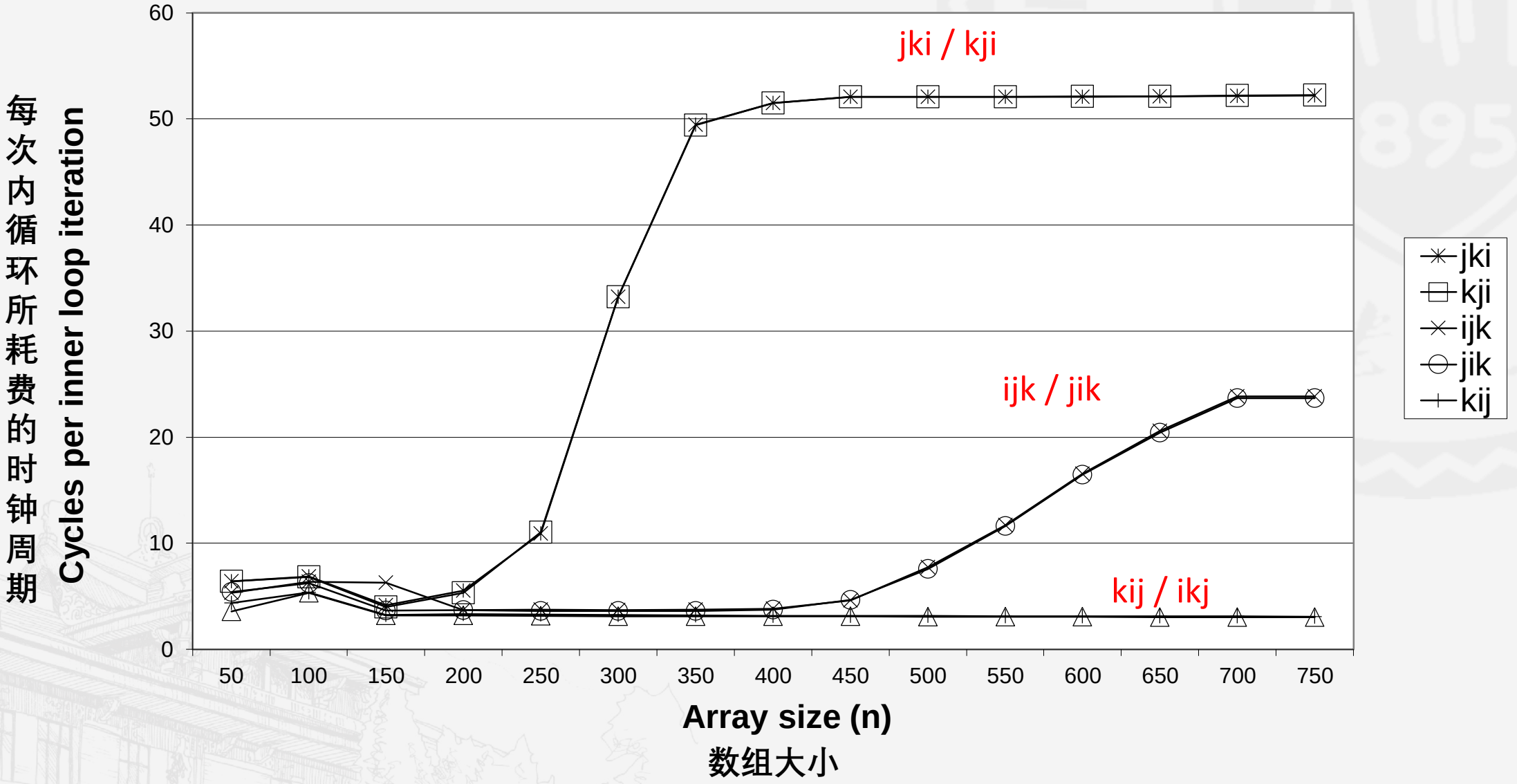
jki (& kji):

- 2次内存加载，1次存储
  2 loads, 1 store
- 未命中次数/循环 = 2.0
  misses/iter = 2.0

高速缓存对软件性能的影响

Performance impact of caches

酷睿i7处理器矩阵乘法性能
Core i7 Matrix Multiply Performance



本章内容

Topic

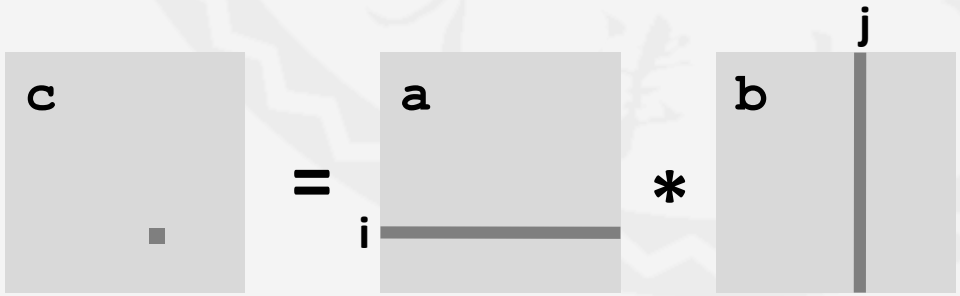
- 高速缓存的结构和工作原理
  Cache memory organization and operation
- 高速缓存对软件性能的影响
  Performance impact of caches
  - 通过循环重排提高空间局部性特征
    Rearranging loops to improve spatial locality
  - 通过分块提高时间局部性特征
    Using blocking to improve temporal locality
  - 存储器山
    The memory mountain

高速缓存对软件性能的影响

Performance impact of caches

矩阵乘法
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];
}
```



高速缓存对软件性能的影响

Performance impact of caches

缓存未命中分析
Cache Miss Analysis

- 假设:
  Assume:
  - 矩阵中的元素是double类型
    Matrix elements are doubles
  - 缓存块容量是8个double类型数据（64字节）
    Cache block = 8 doubles
  - 缓存总的容量 C << n (远远小于n)
    Cache size C << n (much smaller than n)

- 第一次迭代
  First iteration:

  - n/8 + n = 9n/8 misses （未命中）
  - 随后在缓存中
    Afterwards in cache:

高速缓存对软件性能的影响

Performance impact of caches

缓存未命中分析
Cache Miss Analysis

- 假设:
  Assume:
  - 矩阵中的元素是double类型
    Matrix elements are doubles
  - 缓存块容量是8个double类型数据（64字节）
    Cache block = 8 doubles
  - 缓存总的容量 C << n (远远小于n)
    Cache size C << n (much smaller than n)

- 第二次迭代
  First iteration:

  - n/8 + n = 9n/8 misses （未命中）

- 总的未命中次数:
  Total misses:

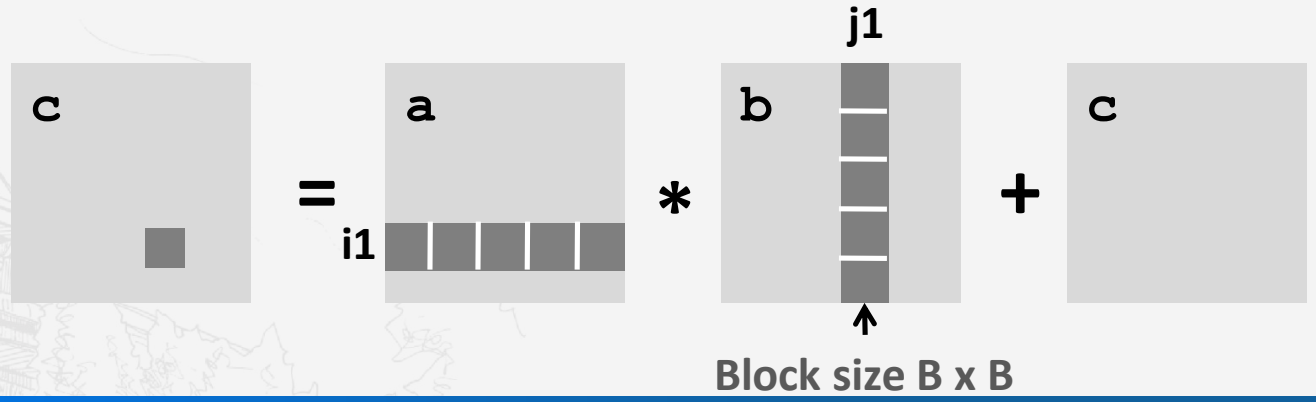
$$9n/8 * n^2 = (9/8) * n^3$$

高速缓存对软件性能的影响

Performance impact of caches

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];
}
```



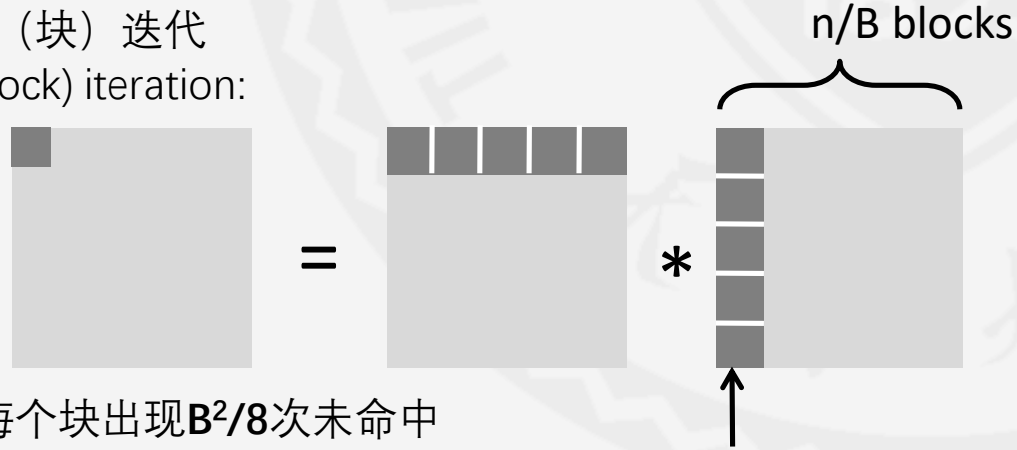
高速缓存对软件性能的影响

Performance impact of caches

缓存未命中分析
Cache Miss Analysis

- 假设：
  Assume:
  - 矩阵中的元素是double类型
    Matrix elements are doubles
  - 缓存块容量是8个double类型数据（64字节）
    Cache block = 8 doubles
  - 缓存能够容纳三个“块”： $3B^2<C$
    Three blocks fit into cache: $3B^2 < C$

- 第一次（块）迭代
  First (block) iteration:



  - 每个块出现$B^2/8$次未命中
    $B^2/8$ misses for each block
  - 有 $2n/B * B^2/8 = nB/4$ 次未命中（忽略矩阵C）
    Misses = $2n/B * B^2/8 = nB/4$ (omitting matrix c)

- 随后在缓存中
  Afterwards in cache:

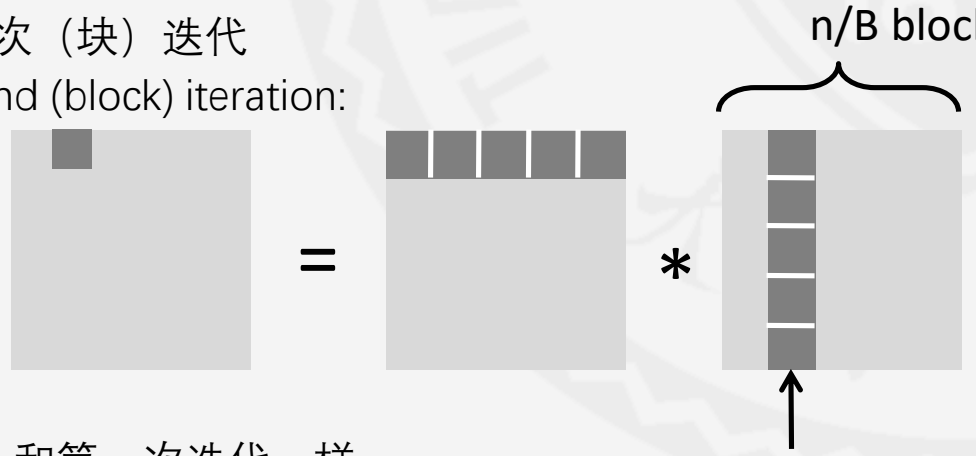
高速缓存对软件性能的影响

Performance impact of caches

缓存未命中分析
Cache Miss Analysis

- 假设：
  Assume:
  - 矩阵中的元素是double类型
    Matrix elements are doubles
  - 缓存块容量是8个double类型数据（64字节）
    Cache block = 8 doubles
  - 缓存能够容纳三个“块”：$3B^2<C$
    Three blocks fit into cache: $3B^2 < C$

- 第二次（块）迭代
  Second (block) iteration:



  - 和第一次迭代一样
    Same as first iteration
  - 有 **$2n/B * B^2/8 = nB/4$** 次未命中
    Misses = **$2n/B * B^2/8 = nB/4$**

- 总未命中次数：
  Total misses: **$nB/4 * (n/B)^2 = n^3/(4B)$**

高速缓存对软件性能的影响

Performance impact of caches

小结
Summary

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

- B的值越大，未命中次数越少。但要注意限制 $3B^2 < C$
  Suggest largest possible block size B, but limit $3B^2 < C$

- 导致这些显著差异的原因：
  Reason for dramatic difference:
  - 矩阵乘法中具有内在的时间局部性特征
    Matrix multiplication has inherent temporal locality:
    - 输入数据：$3n^2$，计算次数：$2n^3$
      Input data: 3n2, computation 2n3
    - 每个数组元素需要使用O(n)次
      Every array elements used O(n) times
  - 必须要恰当的编写程序，才能够实现这种局部性特征
    But program has to be written properly

本章内容

Topic

- 高速缓存的结构和工作原理
  Cache memory organization and operation
- 高速缓存对软件性能的影响
  Performance impact of caches
  - 通过循环重排提高空间局部性特征
    Rearranging loops to improve spatial locality
  - 通过分块提高时间局部性特征
    Using blocking to improve temporal locality
  - 存储器山
    The memory mountain

高速缓存对软件性能的影响

Performance impact of caches

存储器山
The Memory Mountain

- **读吞吐量**（读带宽）
  Read throughput (read bandwidth)
  - 从存储系统中读取数据的速率 （MB/S）
    Number of bytes read from memory per second (MB/s)

- **存储器山：一个读吞吐量关于时间和空间局部性的函数关系**
  Memory mountain: Measured read throughput as a function of spatial and temporal locality
  - 一种表征存储系统性能的简洁方法
    Compact way to characterize memory system performance

高速缓存对软件性能的影响

Performance impact of caches

测量存储器山的函数
Memory Mountain Test Function

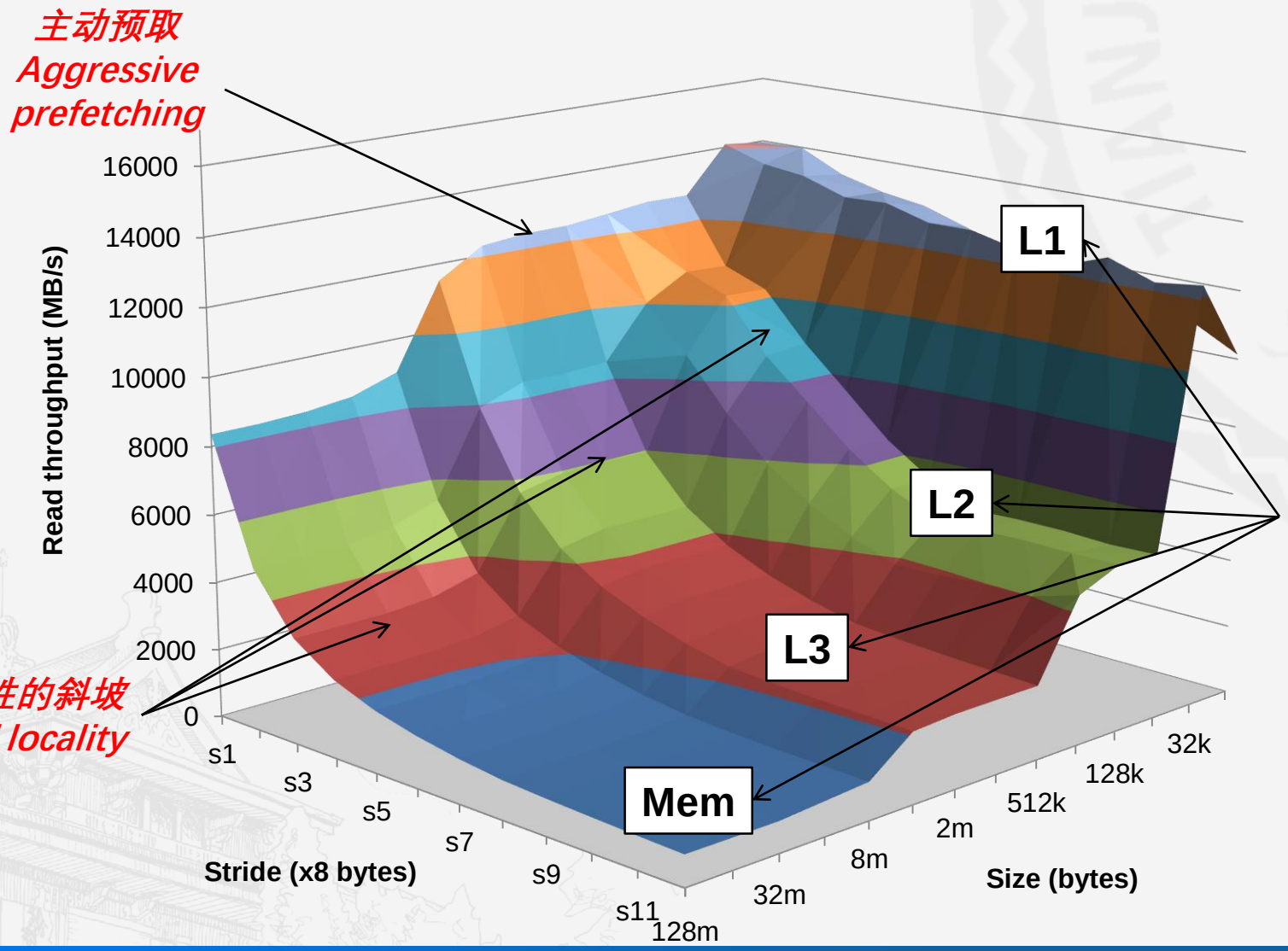
- 使用不同的elems和stride的参数组合，调用test()函数
  Call test() with many combinations of elems and stride
- 对于每一个 elems 和 stride 参数
  For each elems and stride:
  - 首先，调用一次test()，对缓存进行“预热”
    First, call test() once to warm up the caches:
  - 然后，再次调用test()，然后测量读吞吐量
    Then, call test() again and measure the read throughput

```
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+=stride) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

高速缓存对软件性能的影响

Performance impact of caches

存储器山
The Memory Mountain



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