

第六章 存储器层次结构

第二部分

高速缓冲器Cache

要点

- 高速缓存存储器组织结构和操作
- 高速缓存对程序性能的影响
 - 存储器山
 - 重新排列循环以提高空间局部性
 - 使用分块来提高时间局部性

局部性举例

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

■ 对数据的引用

- 顺序访问数组元素
(步长为1的引用模式)
- 变量**sum**在每次循环迭代中被引用一次

空间局部性

时间局部性

■ 对指令的引用

- 顺序读取指令
- 重复循环执行for循环体

空间局部性

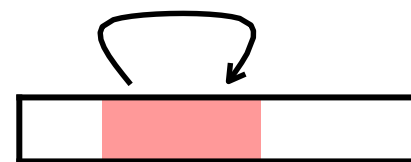
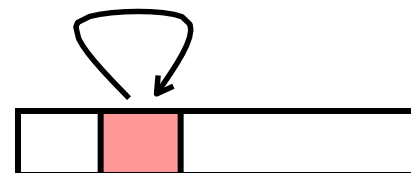
时间局部性

局部性

- **局部性原理:** 程序倾向于使用与最近使用过数据的地址接近或是相同的的数据和指令

- **时间局部性:**

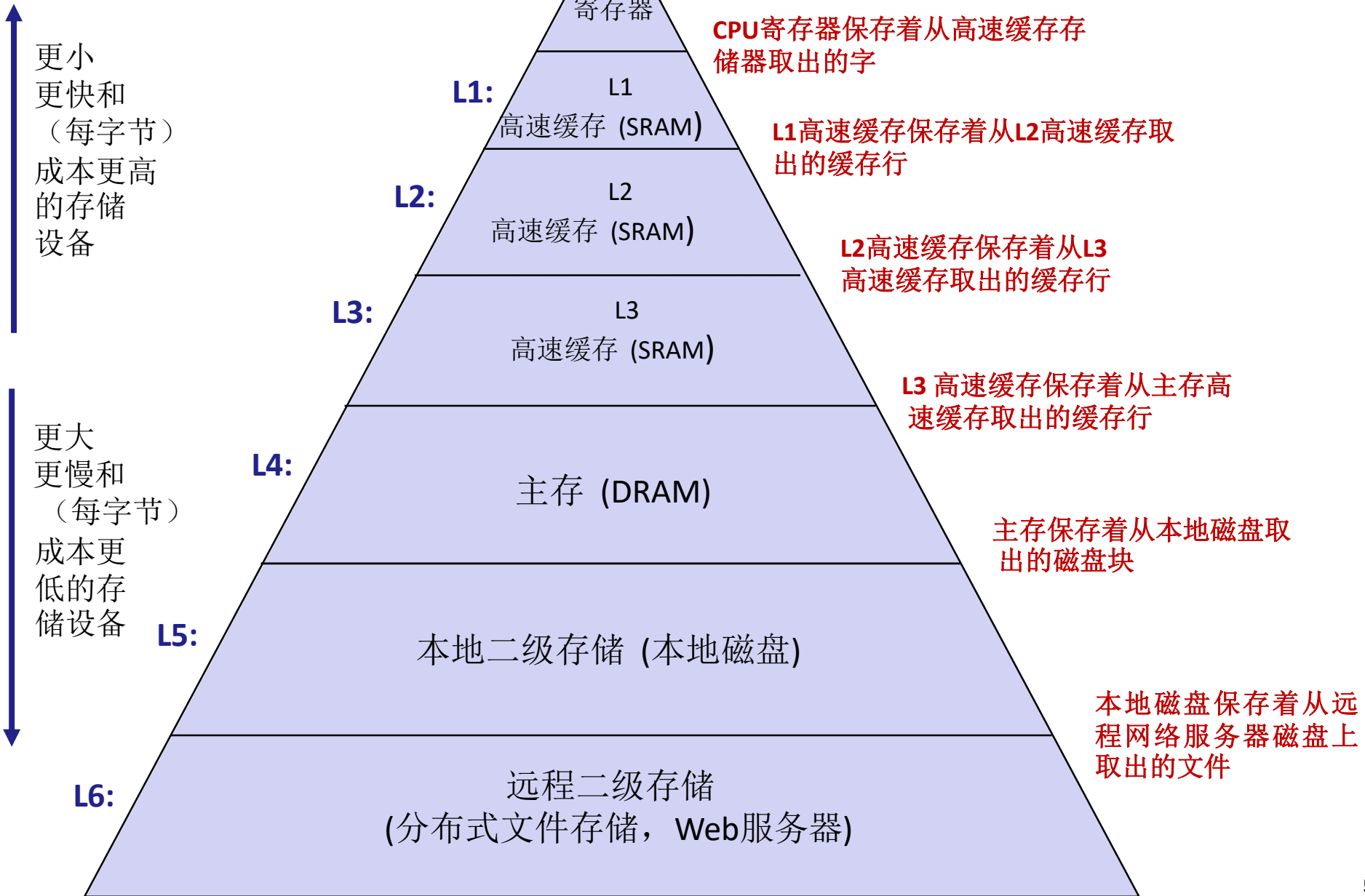
- 最近引用的项很可能在不久的将来再次被引用



- **空间局部性:**

- 与被引用项相邻的项有可能在不久的将来再次被引用

存储器层次结构



存储器层次结构中的缓存

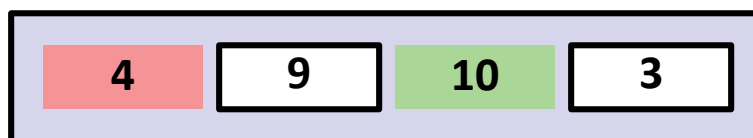
缓存类型	缓存什么	被缓存在何处	延迟(周期数)	由谁管理
寄存器	4-8 字节字	CPU 核心	0	编译器
TLB	地址译码	片上 TLB	0	硬件 MMU
L1 高速缓存	64字节块	片上 L1	4	硬件
L2 高速缓存	64字节块	片上 L2	10	硬件
虚拟内存	4KB 页	主存	100	硬件 + OS
缓冲区缓存	部分文件	主存	100	OS
磁盘缓存	磁盘扇区	磁盘控制器	100,000	磁盘固件
网络缓冲区缓存	部分文件	本地磁盘	10,000,000	NFS 客户
浏览器缓存	Web页	本地磁盘	10,000,000	Web浏览器
Web缓存	Web 页	远程服务器磁盘	1,000,000,000	Web 代理 服务器

高速缓存

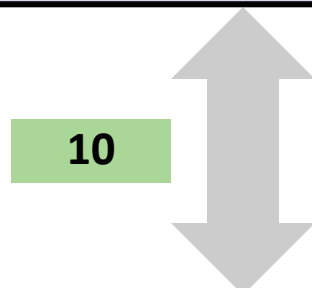
- **高速缓存(Cache):** 一种更小、速度更快的存储设备。作为更大、更慢存储设备的缓存区。
- 存储器层次结构的基本思想:
 - 对于每个 k ，位于 k 层的更快更小存储设备作为位于 $k+1$ 层的更大更慢存储设备的缓存。
- 为什么存储器层次结构行的通?
 - 因为局部性原理，程序访问第 k 层的数据比访问第 $k+1$ 层的数据要频繁
 - 因此，第 $k+1$ 层的存储设备更慢、容量更大、价格更便宜
- **妙策:** 存储器层次结构构建了一个大容量的存储池，像底层存储器一样廉价，而又可以达到顶层存储器的速度。

高速缓存的基本概念

高速缓存

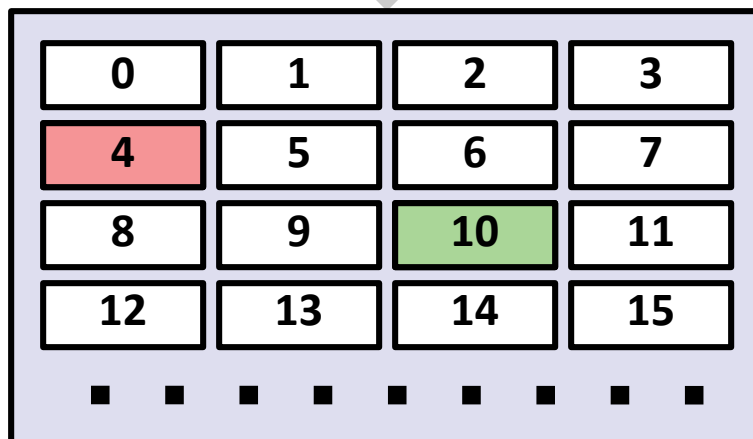


第K层更小、更快、更昂贵的设备缓存着第K+1层块的一个子集



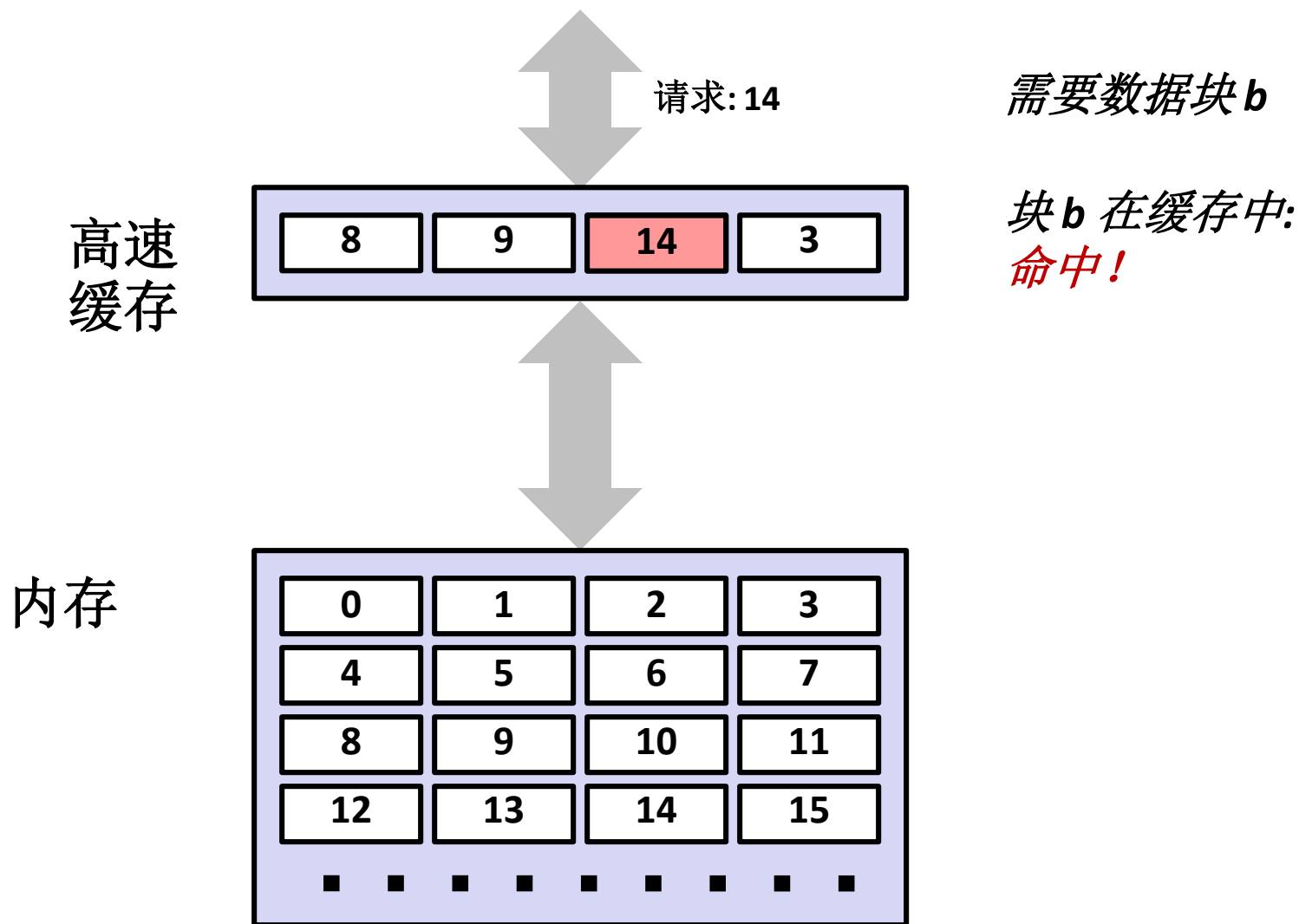
数据以块为传输单元
在层与层之间复制

存储器

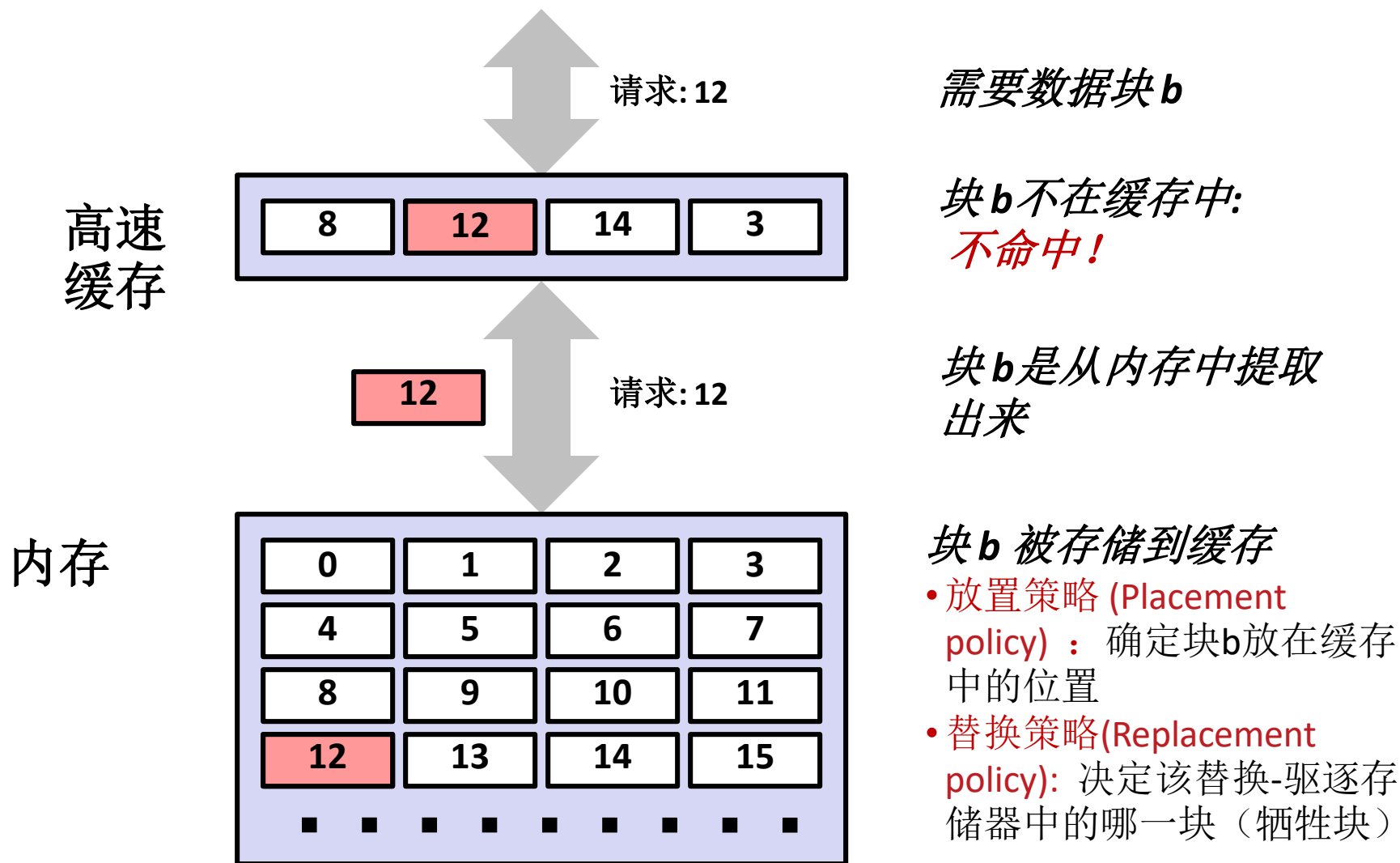


第K+1层更大、更慢、更便宜的设备被划分成块

高速缓存概念：缓存命中



高速缓存概念：缓存不命中



高速缓存概念：

缓存不命中的种类

■ 冷 (强制性) 不命中

- 当缓存为空时，对任何数据的请求都会不命中，此类不命中称为冷不命中

■ 冲突不命中

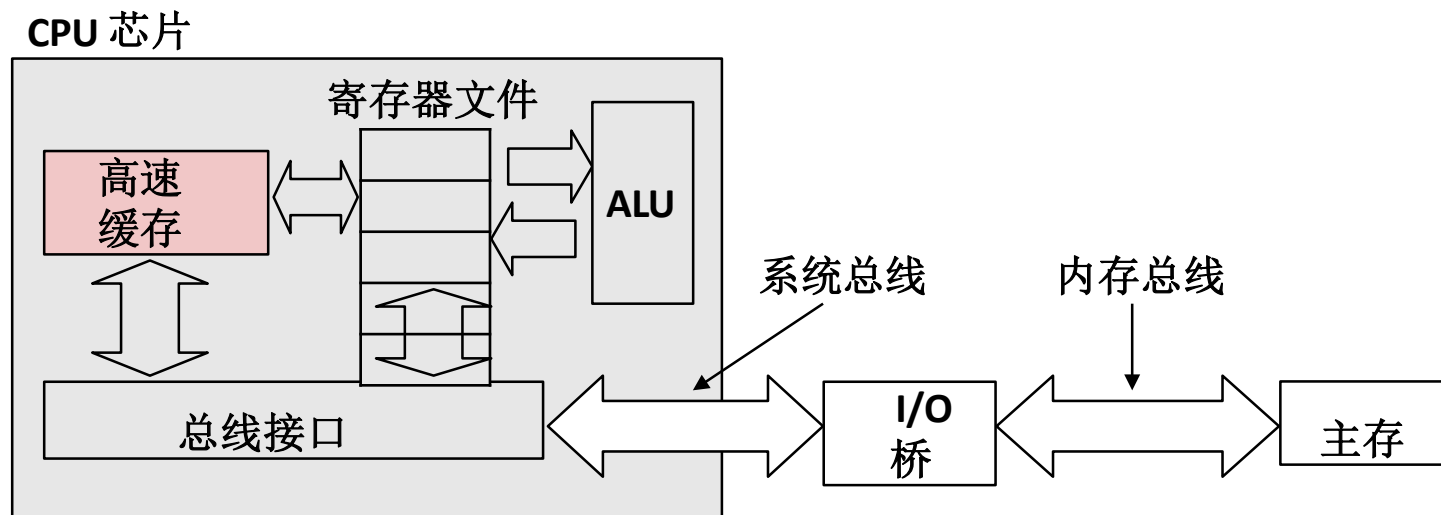
- 大多数缓存将第 $k+1$ 层的某个块限制放置在第 k 层块的一个小的子集中（有时只是一个块）。
 - 例如.第 $k+1$ 层的块 i 必须放置在第 k 层的块 $(i \bmod 4)$ 中.
- 冲突不命中发生在缓存足够大，但是这些多个数据对象会映射到同一个缓存块.
 - 例如.如果程序请求块0,8,0,8,0,8,...这样每次引用都会不命中.

■ 容量不命中

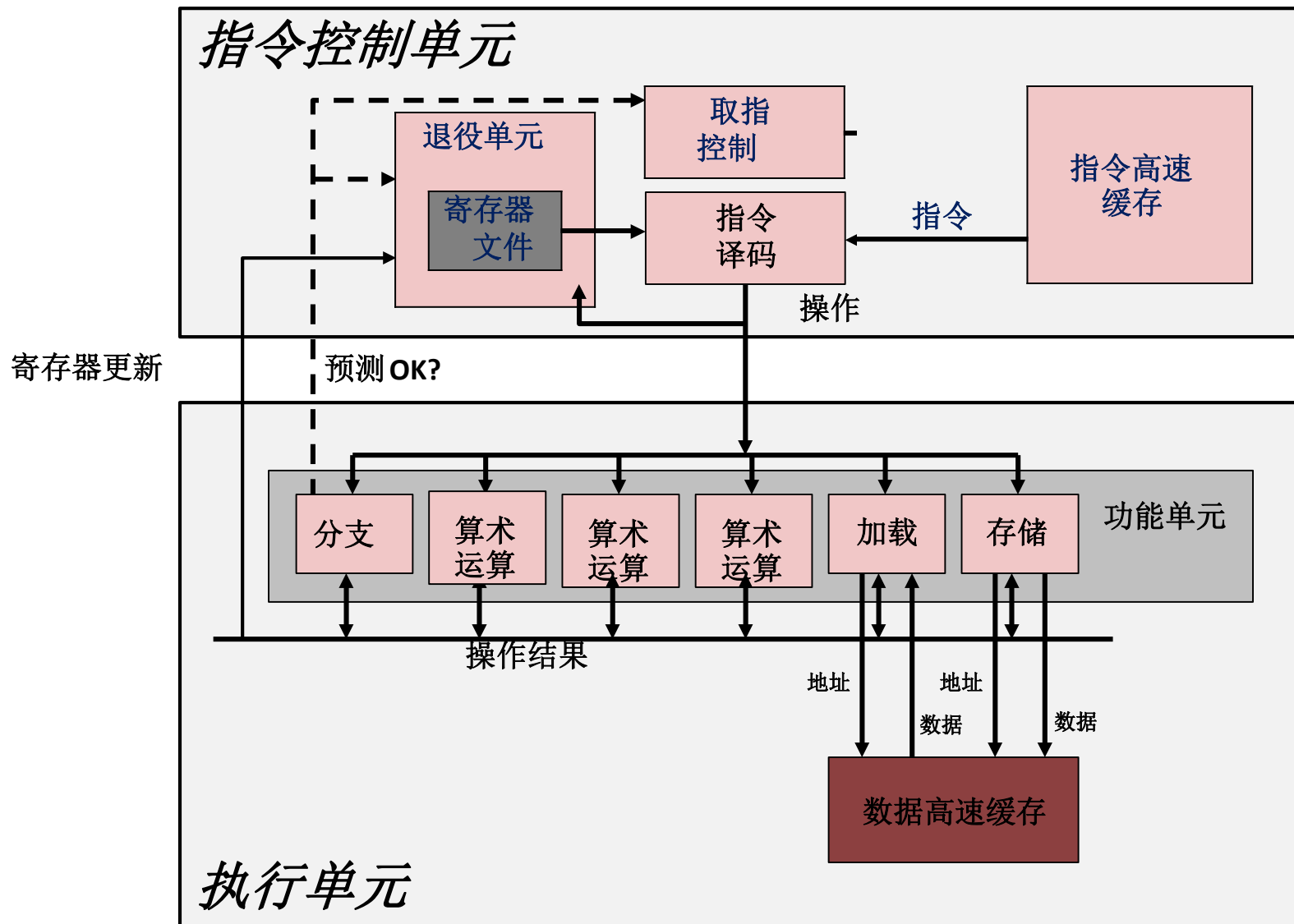
- 发生在当活跃块集合（工作集 **working set**）的大小比缓存大.

高速缓存存储器

- **高速缓存存储器**是小型的、快速的基于**SRAM**的存储器，是在硬件中自动管理的（非用户程序访问的）
 - 保持经常访问主存的块
- **CPU** 首先查找缓存中的数据
- 典型的系统结构:



现代 CPU 设计



实例

台式机 PC

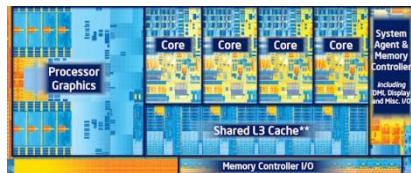


Source: Dell

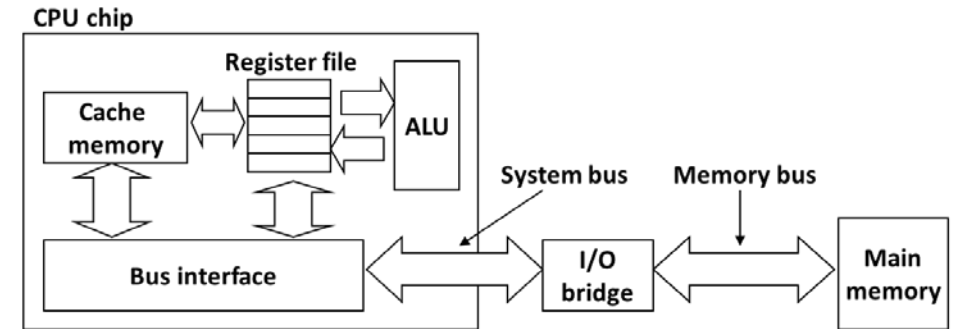
CPU (Intel Core i7)



Source: PC Magazine



Source: techreport.com



主板



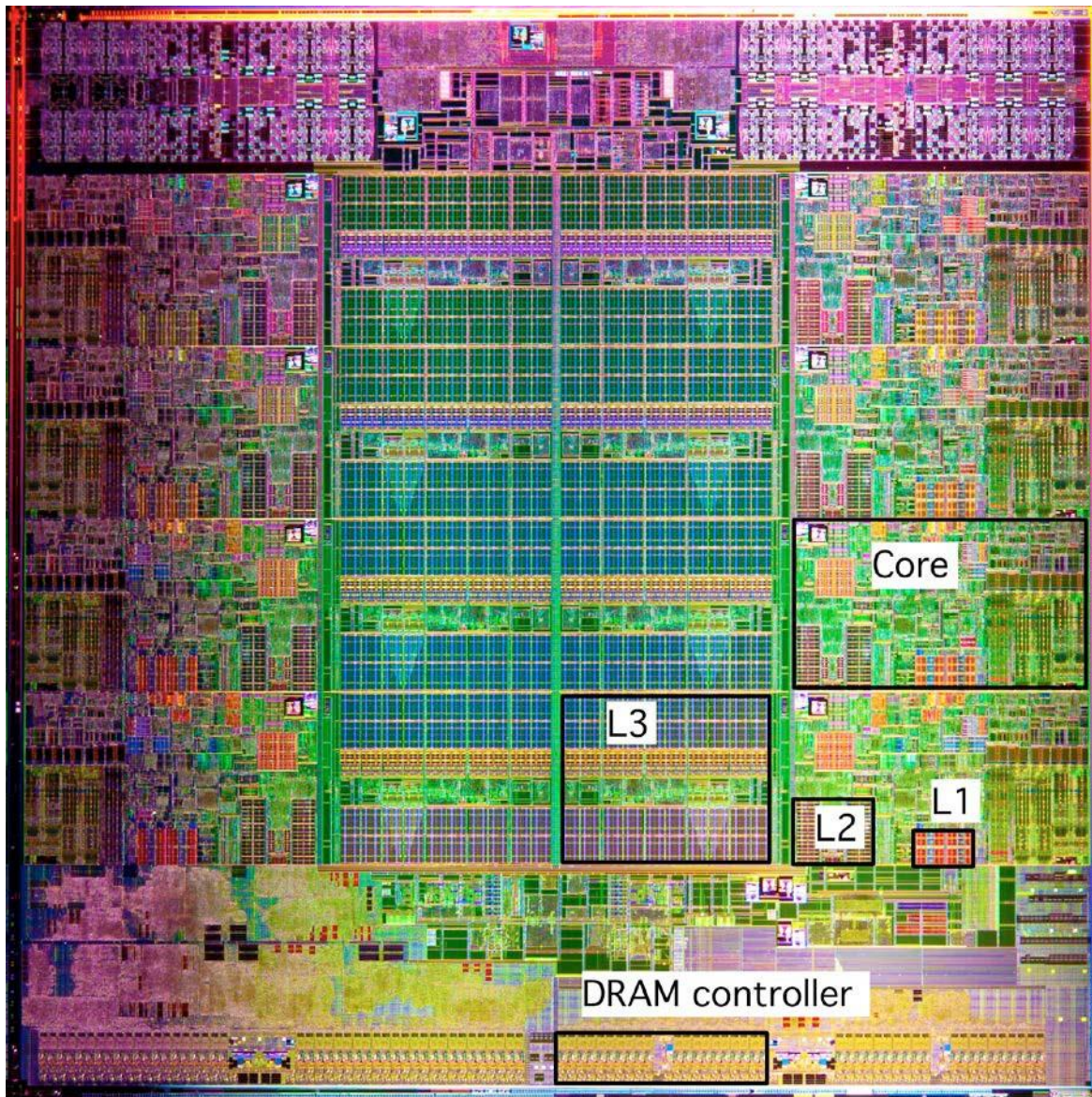
Source: Dell

主存 (DRAM)



Source: Dell

实例(Cont.)



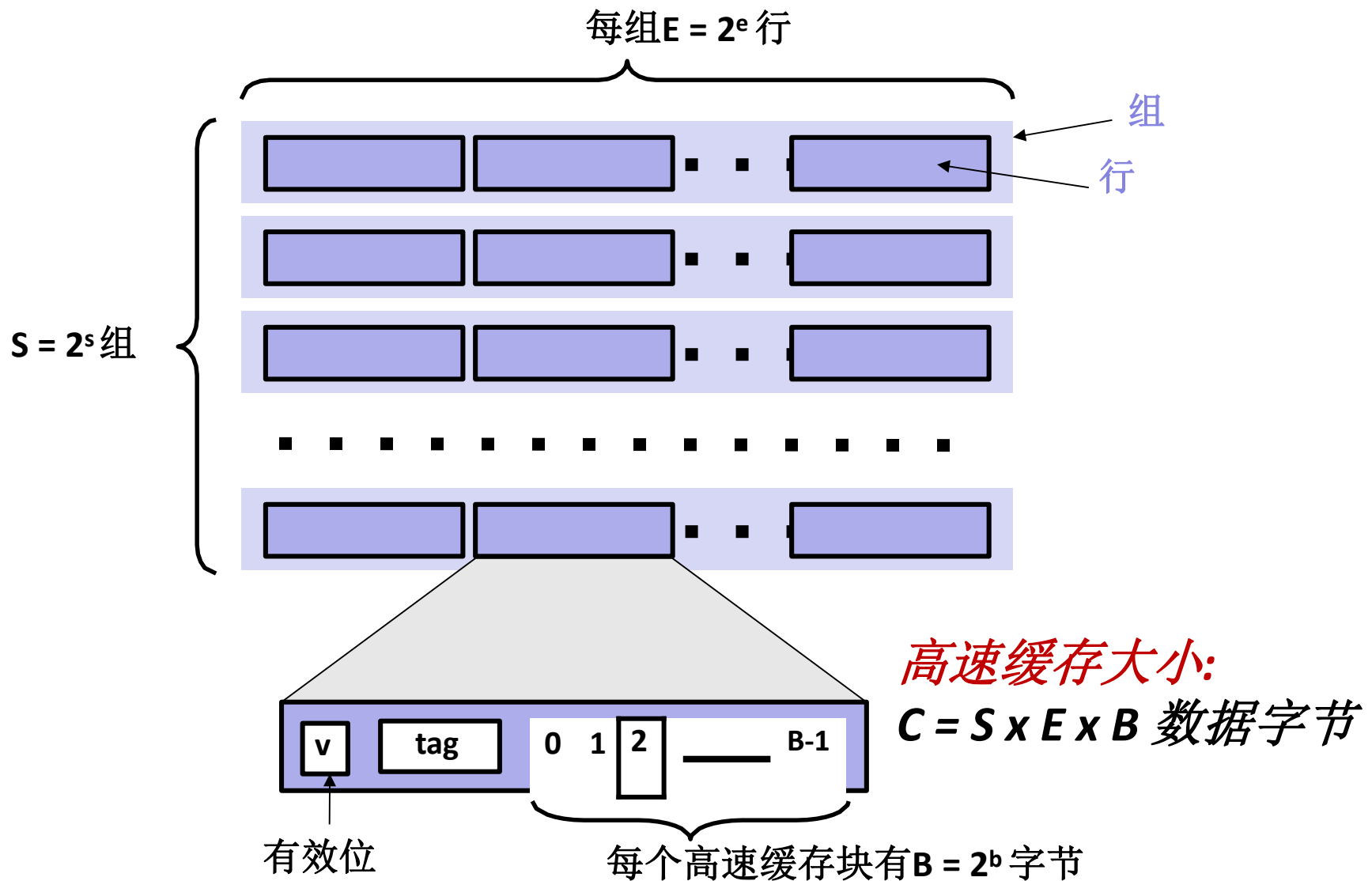
Intel Sandy Bridge
Processor Die

L1: 32KB 指令 + 32KB 数据
L2: 256KB
L3: 3–20MB

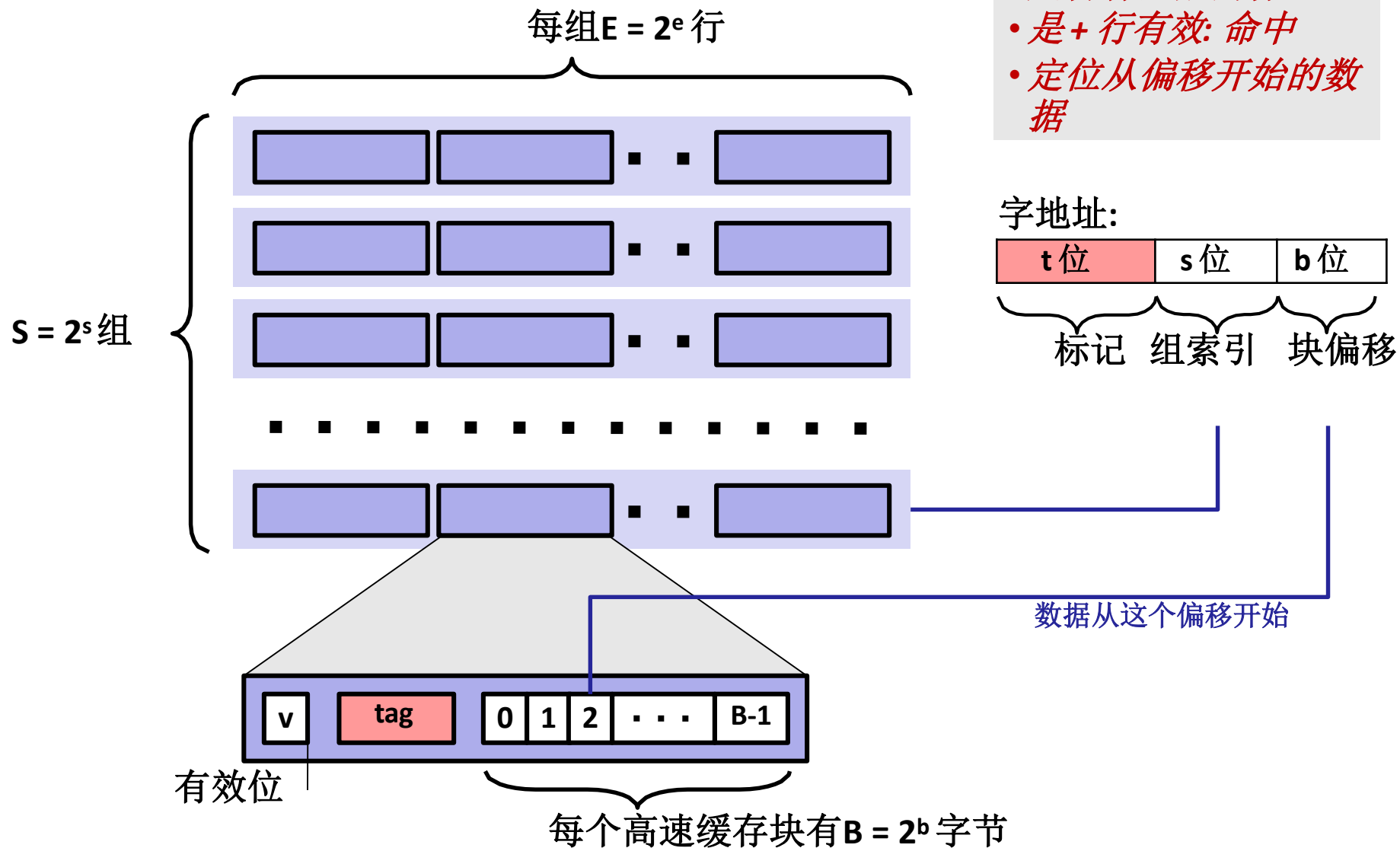
回顾下缓存到主存的三种映射关系

- 组相联映射
- 直接映射
- 全相联映射

高速缓存通用组织(S, E, B)



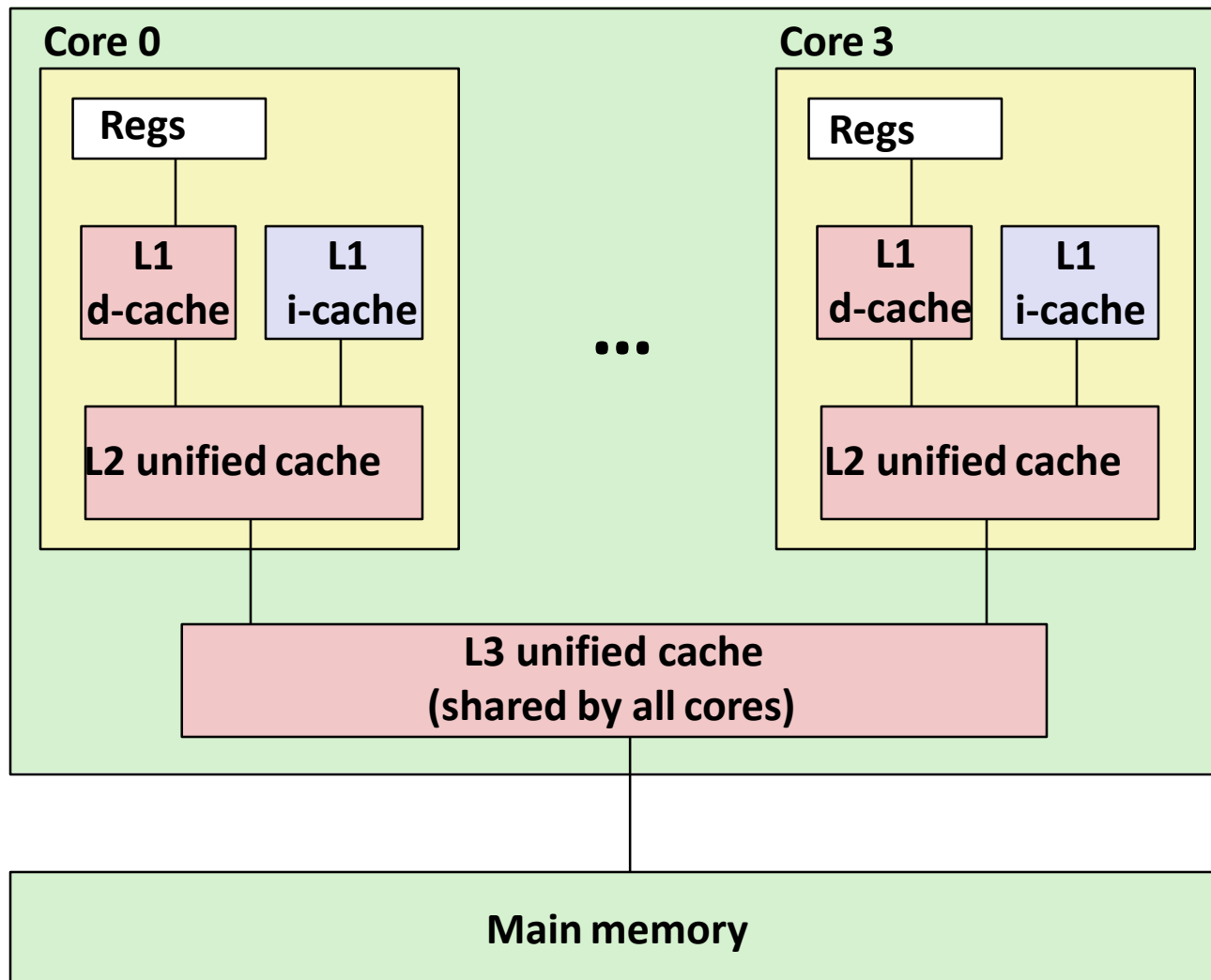
高速缓存读



- 定位组
- 检查集合中的任何行是否有匹配的标记
- 是+ 行有效: 命中
- 定位从偏移开始的数据

Intel Core i7高速缓存层次结构

处理器封装



L1 指令高速缓存 和 数据高速缓存:

32 KB, 8-way,
访问时间: 4周期

L2 统一的高速缓存:

256 KB, 8-way,
访问时间: 10 周期

L3 统一的高速缓存:

8 MB, 16-way,
访问时间: 40-75 周期

块大小: 所有缓存都是64 字节

例子: Core i7 L1 数据缓存

32 kB 的8路组相联

64 字节/块

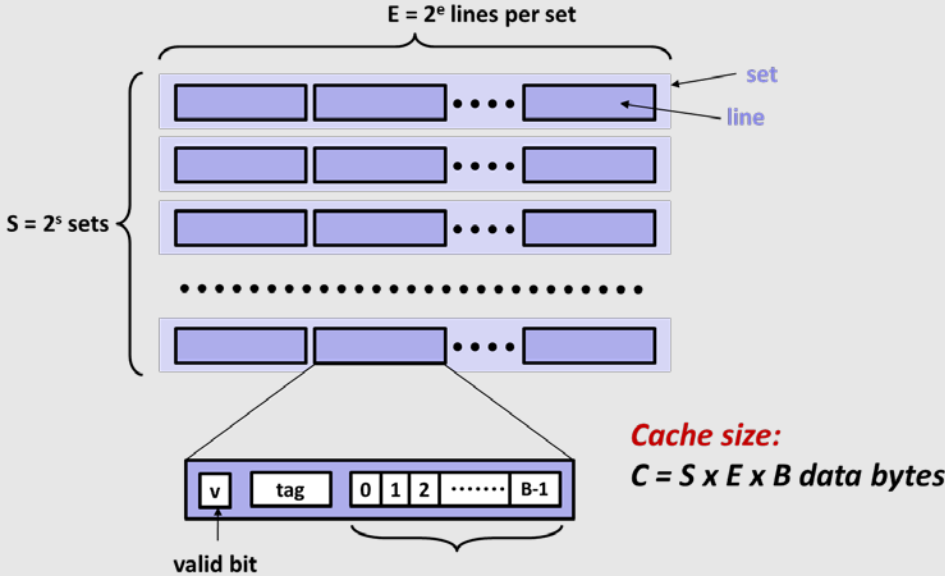
47 位地址范围

B =

S = , s =

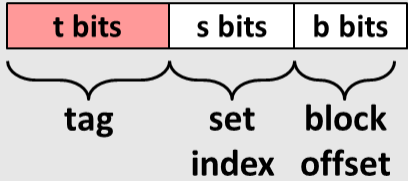
E = , e =

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:



块偏移: . 位
组索引: . 位
标记: . 位

栈地址: 0x00007f7262a1e010

块偏移: 0x??
组索引: 0x??
标记: 0x??

例子: Core i7 L1 数据缓存

32 kB 的8路组相联

64 字节/块

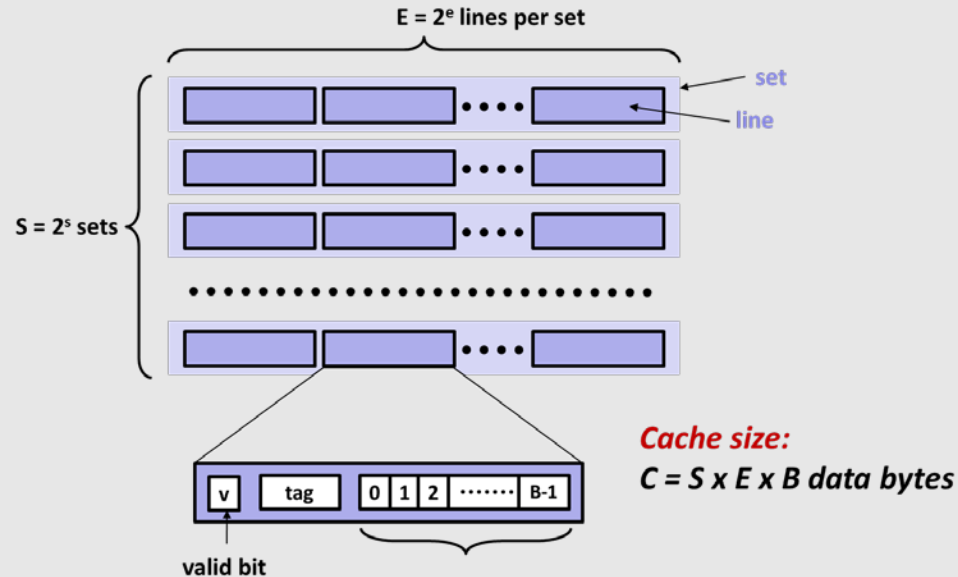
47 位地址范围

$B = 64$

$S = 64, s = 6$

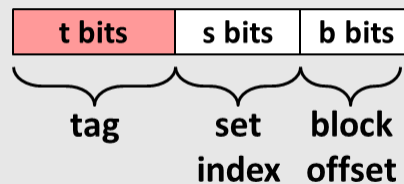
$E = 8, e = 3$

$C = 64 \times 64 \times 8 = 32,768$



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:



块偏移: 6 bits

组索引: 6 bits

标记: 35 bits

栈地址:

0x00007f7262a1e010

0000 0001 0000

块偏移:

0x10

组索引:

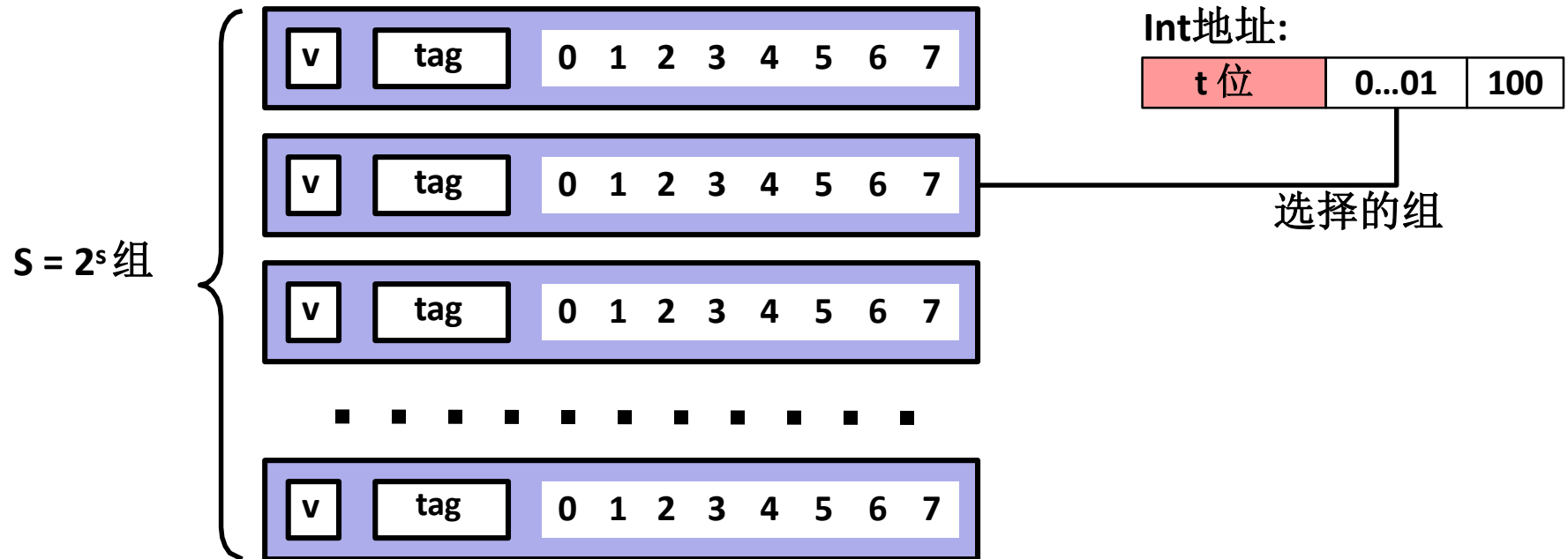
0x0

标记:

0x7f7262a1e

示例：直接映射高速缓存 ($E = 1$)

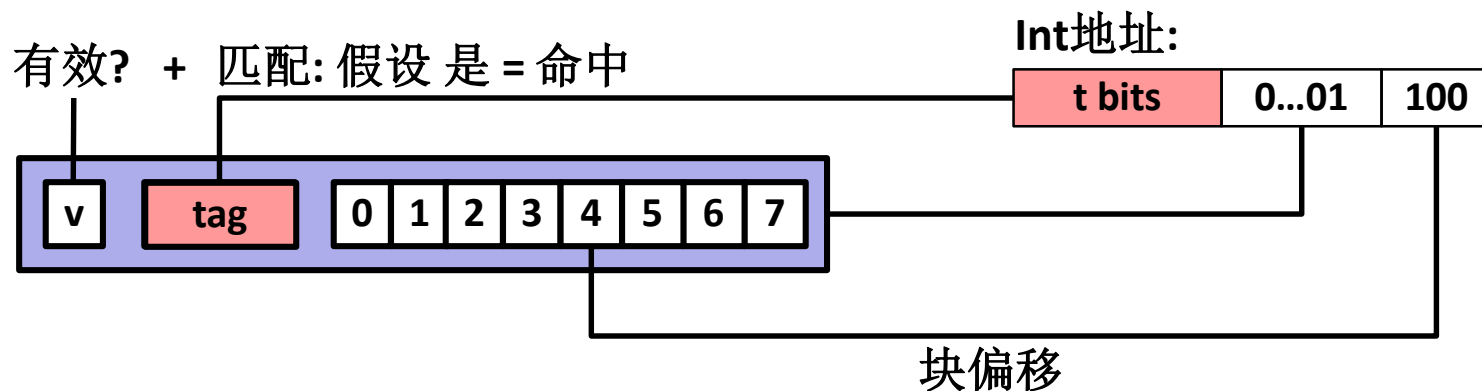
直接映射: 每一组只有一行
假设: 缓存块大小为8字节



示例：直接映射高速缓存 ($E = 1$)

直接映射：每一组只有一行

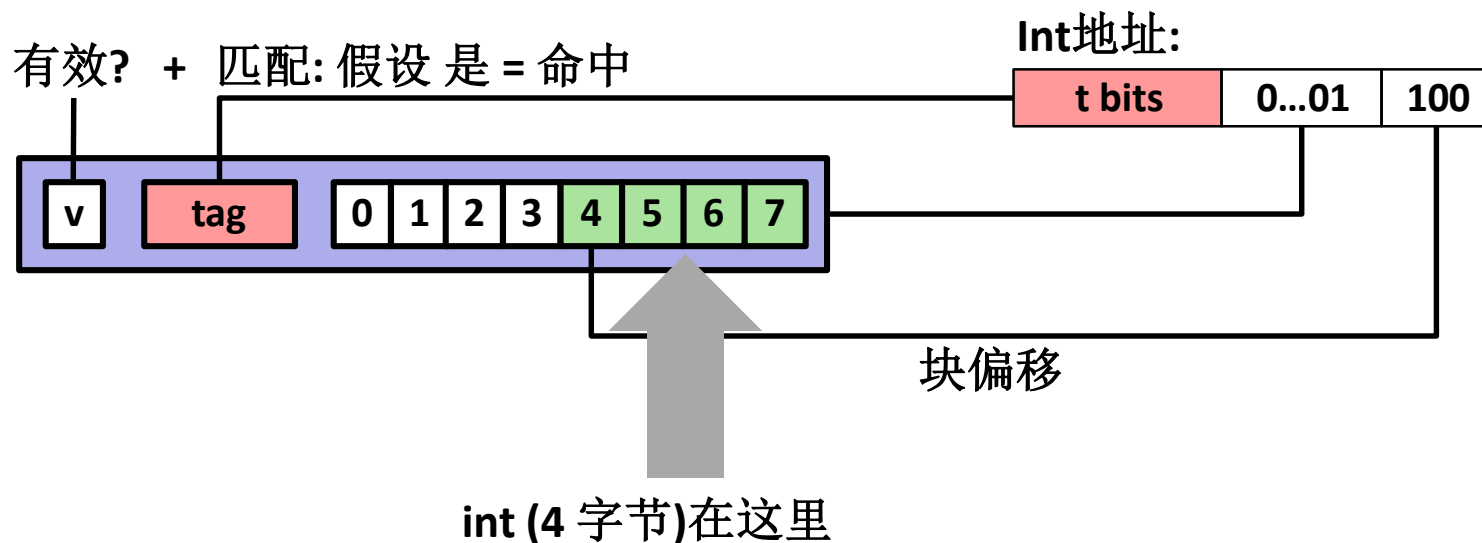
假设：缓存块大小为8字节



示例：直接映射高速缓存 ($E = 1$)

直接映射: 每一组只有一行

假设: 缓存块大小为8字节



如果标记不匹配: 旧的行被驱逐和替换

直接映射高速缓存模拟

M=16 字节 (4-位 地址), B=2 字节/块,
S=4 组, E=1 块/组

t=1	s=2	b=1
X	XX	X

地址跟踪(读, 每读一个字节):

0	[0 <u>00</u> 0] ₂ ,	miss
1	[0 <u>00</u> 1] ₂ ,	hit
7	[0 <u>11</u> 1] ₂ ,	miss
8	[<u>1</u> 000] ₂ ,	miss
0	[0 <u>00</u> 0] ₂	miss

抖动：频繁间距访问的数据映射到同一个Cache组。

优化：组相联。

	V	Tag	Block
组 0	1	0	M[0-1]
组 1			
组 2			
组 3	1	0	M[6-7]

E-路组相联高速缓存 (E = 2)

E = 2: 每组两行

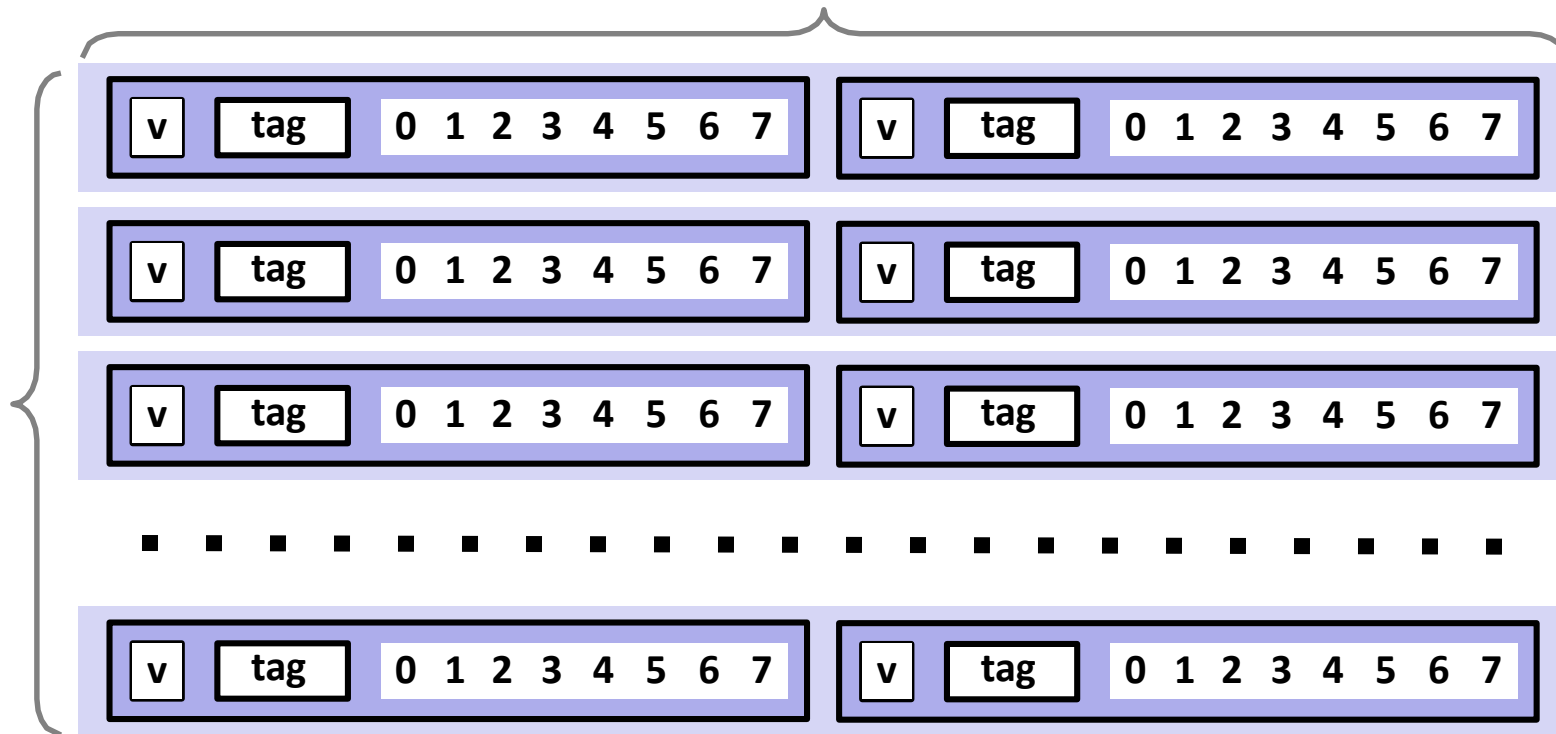
假设:缓存块大小为8字节

每组两行

short int地址:

t bits	0...01	100
--------	--------	-----

1.选择组

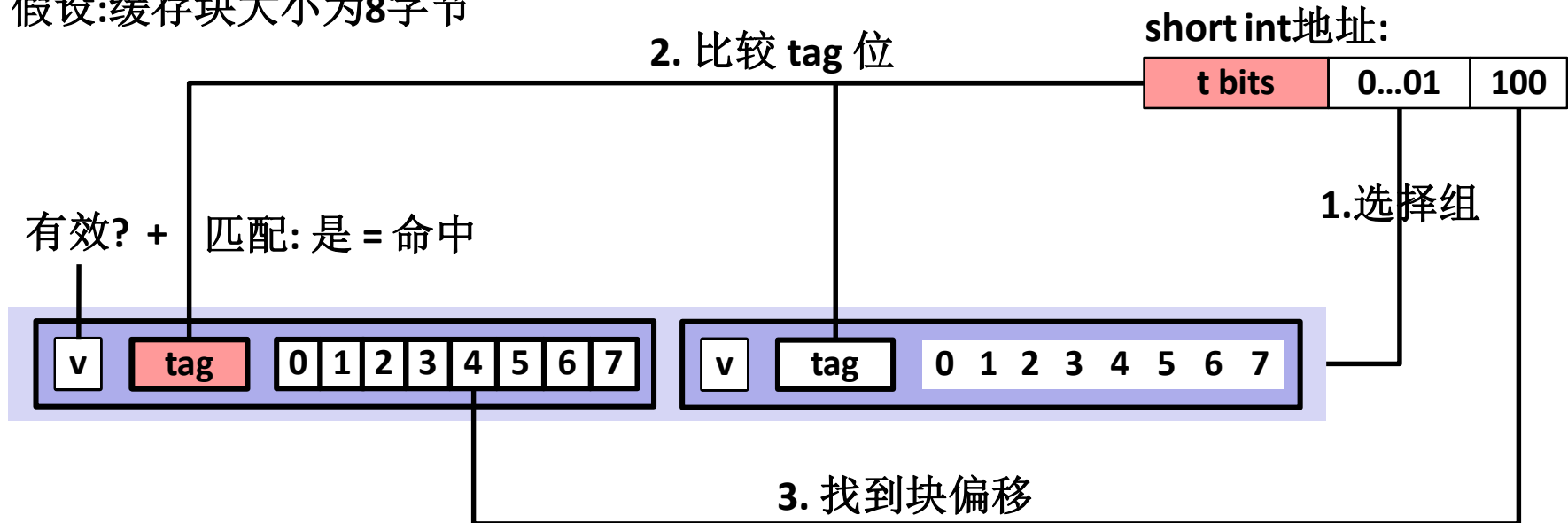


s组

E-路组相联高速缓存(E = 2)

E = 2: 每组两行

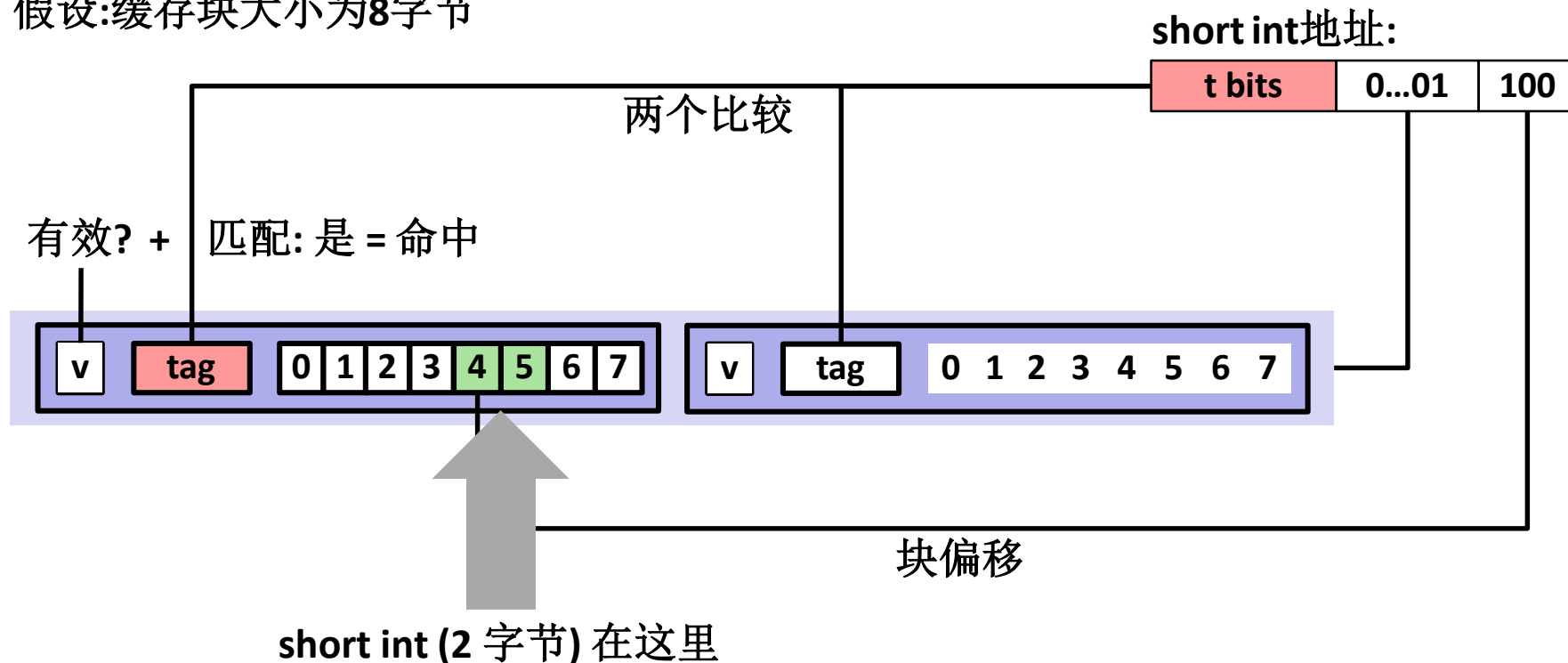
假设: 缓存块大小为8字节



E-路组相联高速缓存(E = 2)

E = 2: 每组两行

假设: 缓存块大小为8字节



不匹配:

- 在组中选择1行用于驱逐和替换
- 替换策略: 随机、最不常使用LFU、最近最少使用(LRU)、...

2-路 组相联缓存模拟

t=2 s=1 b=1

xx	x	x
----	---	---

M=16 字节地址, B=2 字节/块, S=2 组,
E=2 块/组

地址跟踪(读, 每读一个字节):

0 [0000₂], miss
1 [0001₂], hit
7 [0111₂], miss
8 [1000₂], miss
0 [0000₂] hit

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

全相联映射Cache组织示意图

假定数据在主存和Cache间的传送单位为512字。

Cache大小： 2^{13} 字=8K字=16行 x 512字/行

主存大小： 2^{20} 字=1024K字=2048块 x 512字/块

Cache标记 (tag) 指出对应行取自哪个主存块

主存tag指出对应地址位于哪个主存块

如何对01E0CH单元进行访问？

0000 0001 1110 0000 1100B
是第15块中的第12个单元！

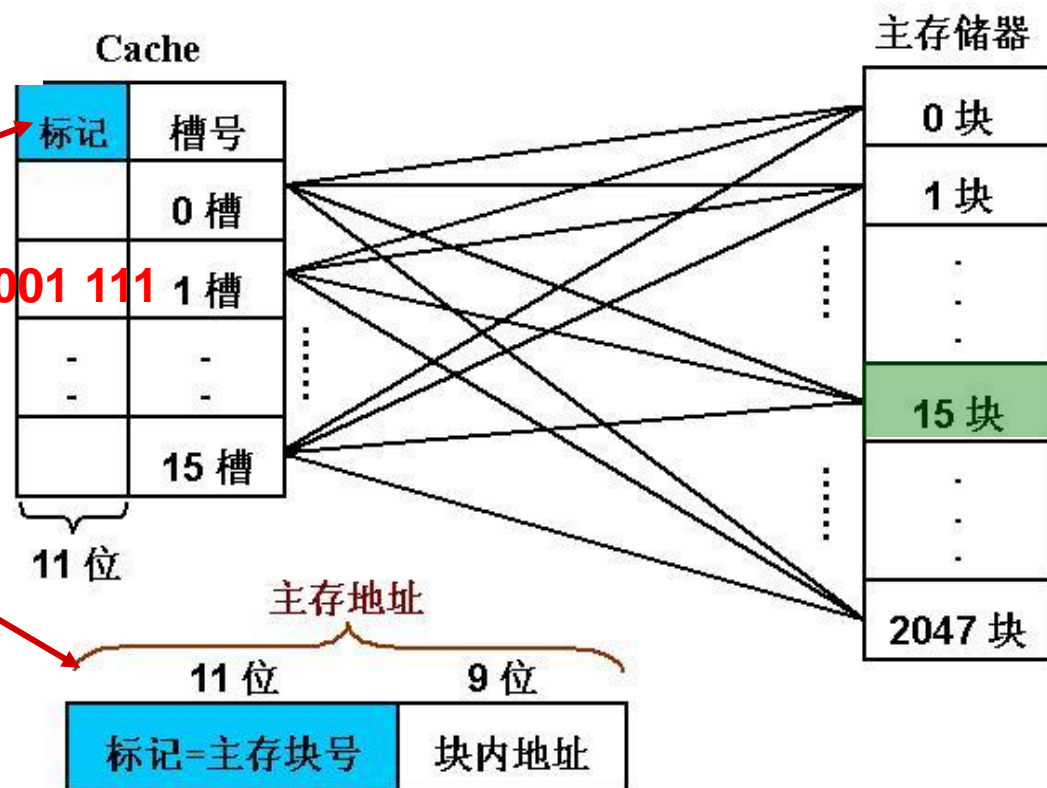
按内容访问，是相联存取方式！

如何实现按内容访问？

直接比较！

每个主存块可装到Cache任一行中。

全相联映射的 Cache 组织示意图



为何地址中没有cache索引字段？
因为可映射到任意一个cache行中！

举例：全相联

全相联 Cache

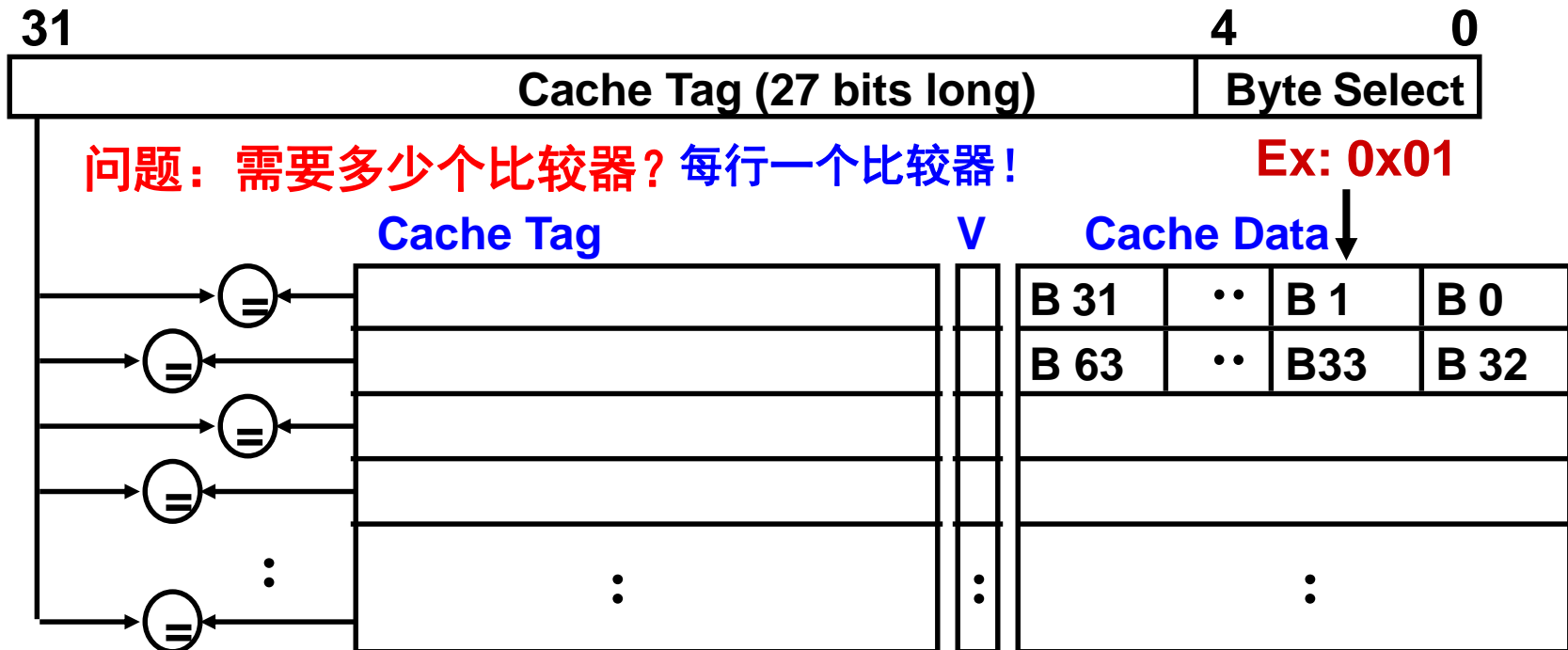
无需Cache索引，为什么？ 因为同时比较所有Cache项的标志

根据定义：冲突不命中 = 0

(没有冲突缺失，因为只要有空闲Cache块，都不会发生冲突)

Example: 32bits 内存地址, 32 B 块大小. 比较器位数多长？

需要 27-bit 大小的比较器



Cache的组相联映射和直接映射的对比

组相联映射

cache利用率较高
cache冲突率较低(减少了冲突不命中)
淘汰算法较复杂
应用场合：
 小容量cache

直接映射

cache利用率很低
cache冲突率很高
淘汰算法很简单
应用场合：
 大容量cache

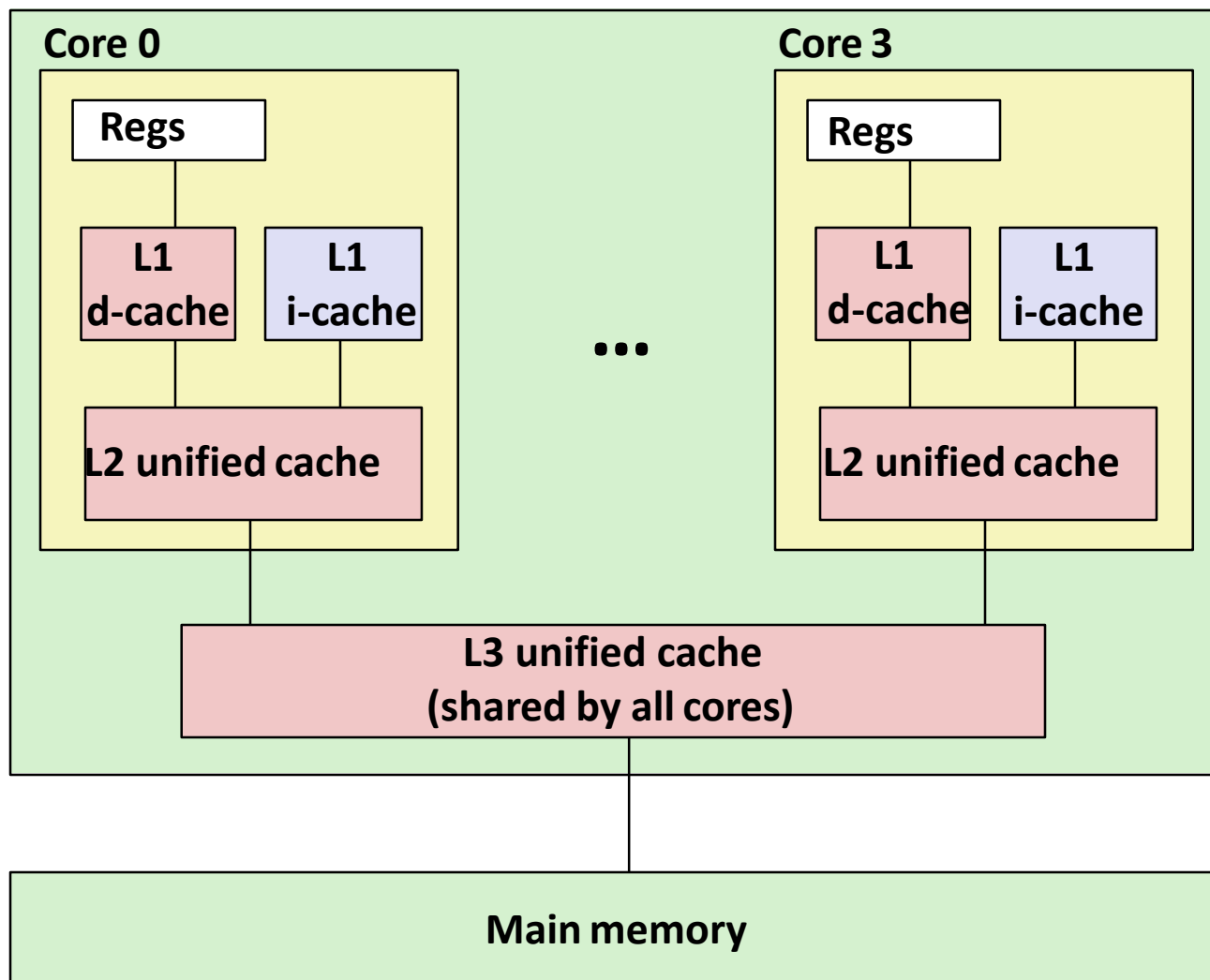
全相联映射(特殊的组相联映射)

cache利用率很高
cache冲突率为零(无冲突不命中)
淘汰算法较复杂
应用场合：小容量cache

看似最好：全相联映射。
但是全相联映射也有缺点：硬件结构很复杂，实现难度和价格都很高，因此实际应用中往往采取**组相联映射**。

Intel Core i7高速缓存层次结构（回顾）

处理器封装



L1 指令高速缓存 和 数据高速缓存:

32 KB, 8-way,
访问时间: 4周期

L2 统一的高速缓存:

256 KB, 8-way,
访问时间: 10 周期

L3 统一的高速缓存:

8 MB, 16-way,
访问时间: 40-75 周期

块大小: 所有缓存的块都是64 字节

高速缓存通用组织(S, E, B) 例题

行数而不是大小

若主存地址 32 位，高速缓存总大小为 2K 行，块大小 16 字节，采用 2 路组相联，则标记位的总位数是 36K 位。

每组 2 行，共 1k 组，标记占 $32-4-10=18$ 位，总位数占 $2k \times 18 = 36k$ 位

关于怎么写?

■ 存在多个数据副本:

- L1, L2, L3, 主存, 磁盘

■ 在写命中时要做什么?

- **直写** (立即写入存储器), 将V=1
- **写回** (推迟写入内存直到行要替换)
 - 需要一个修改位 (和内存相同或不同的行)

■ 写不命中时要做什么?

- **写分配** (加载到缓存, 更新这个缓存行)
 - 好处是更多的写遵循局部性
- **非写分配** (直接写到主存中, 不加载到缓存中)

■ 典型的

- 直写 + 非写分配 --- 高层-成本低
- 写回 + 写分配-----建议 --- 低层如虚拟内存 --- 高层也用 (集成度提高)

高速缓存性能指标

■ 不命中率

- 一部分内存引用在缓存中没有找到 (不命中 / 访问)
= $1 - \text{命中率}$
- 典型的数 (百分比):
 - 3-10% L1
 - 可以相当小(e.g., < 1%) 根据大小, 等等.

■ 命中时间

- 从高速缓存向处理器发送一行的时间
 - 时间包括行是否在缓存中
- 典型的数:
 - L1 4个时钟周期
 - L2 10个时钟周期

■ 不命中处罚

- 由于不命中需要额外的时间
 - 通常主存为50-200周期(趋势: 增加!)

让我们想想那些数字

- 在命中和不命中之间差距巨大
 - 如果只有L1 和 主存，那么可以差100倍
- 你会相信99%命中率要比97%好两倍？
 - 思考：
缓存命中时间为1个周期
不命中处罚要100个周期
 - 平均访问时间：
97% 命中率: $1 \text{ 周期} + 0.03 \times 100 \text{ 周期} = 4 \text{ 周期}$
99% 命中率: $1 \text{ 周期} + 0.01 \times 100 \text{ 周期} = 2 \text{ 周期}$
- 这就是为什么用“不命中率”而不是“命中率”

编写高速缓存友好的代码

- 让通用或共享的功能或函数—最常见情况运行得快
 - 专注在核心函数和内循环上
- 尽量减少每个循环内部的缓存不命中数量
 - 反复引用变量是好的(时间局部性)—寄存器-编译器
 - 步长为1的参考模式是好的(空间局部性)---缓存是连续块

关键思想：通过我们对高速缓冲器的理解来量化我们对原有局部性的定性概念

- 高速缓存的组织结构和运算
- 高速缓存对程序性能的影响
 - 存储器山
 - 重新排列以提升空间局部性
 - 使用块来提高时间局部性

存储器山

■ 读吞吐量 (读带宽)

- 每秒从存储系统中读取的字节数 (MB/s)

■ 存储器山：测量读取吞吐量作为空间和时间局部性的函数

- 紧凑方式去描述存储系统性能

存储器山测试函数

```
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

test() 函数有许多
elems and stride组
合.

对于每个 elems
and stride:

1. test()函数开始预
热缓存.

2. Call test() 函数之后
测量读吞吐量(MB/s)

存储器山测试函数

```
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.  
 */
```

```
/* run - Run test(elems, stride) and return read throughput (MB/s).
```

```
 *      "size" is in bytes, "stride" is in array elements, and Mhz is  
 *      CPU clock frequency in Mhz.  
 */
```

```
double run(int size, int stride, double Mhz)
```

```
{
```

```
    double cycles;
```

```
    int elems = size / sizeof(double);
```

```
    test(elems, stride); /* Warm up the cache */
```

```
    cycles = fcyc2(test, elems, stride, 0); /* Call test(elems, stride) */
```

```
    return (size / stride) / (cycles / Mhz); /* Convert cycles to MB/s */
```

```
}
```

```
    return ((acc0 + acc1) + (acc2 + acc3));
```

```
}
```

mountain/mountain.c

`test()` 函数有许多 `elems` and `stride` 组合。

对于每个 `elems` and `stride`:

1. `test()` 函数开始预热缓存。

2. Call `test()` 函数之后测量读吞吐量(MB/s): 即为 `fcyc2` 函数。

存储器山

Core i7 Haswell

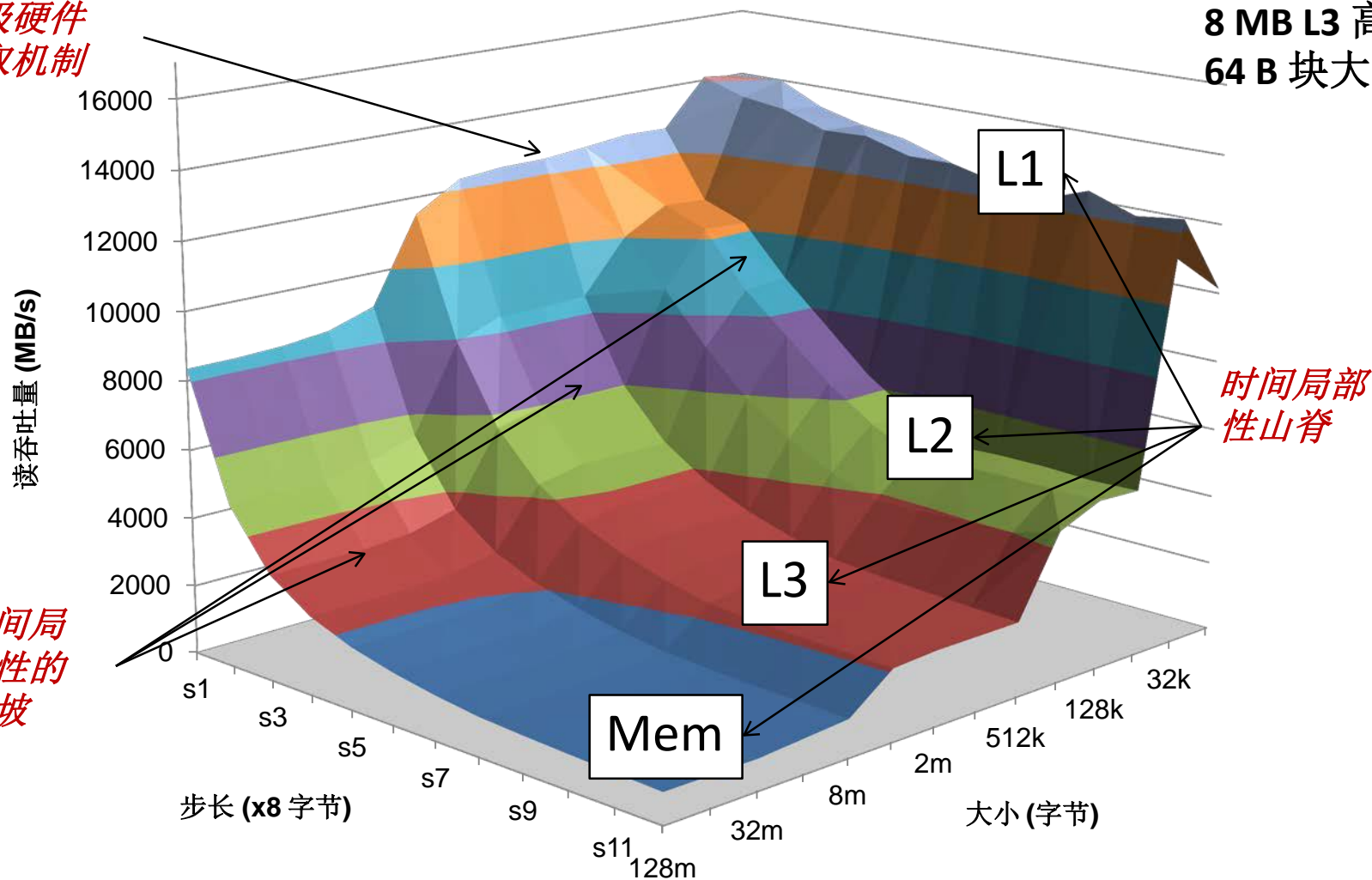
2.1 GHz

32 KB L1 高速缓存

256 KB L2 高速缓存

8 MB L3 高速缓存

64 B 块大小

积极硬件
预取机制

- 高速缓存的组织结构和运算
- 高速缓存对程序性能的影响
 - 存储器山
 - 重新排列以提升空间局部性
 - 使用块来提高时间局部性

矩阵乘法的例子

■ 描述:

- $N \times N$ 矩阵相乘
- 矩阵元素类型是 doubles (8 字节)
- 总共 $O(N^3)$ 个操作
- 每个元素都要读 N 次
- 每个目标中都要对 N 个值求和
 - 但也可以保持在寄存器中

```
/* 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;  
    }  
}
```

变量和保持在寄存器中

matmult/mm.c

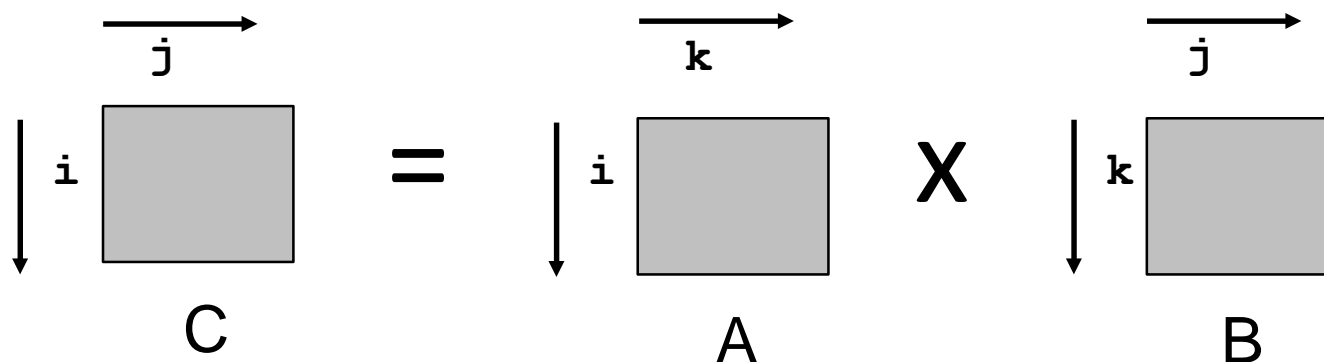
矩阵相乘不命中率分析

■ 假设:

- 块大小 = 32B (足够大4倍)
- 矩阵的维数(N)是非常大的
 - 大约 $1/N$ 为 0.0
- 缓存不是大到足够容纳多行

■ 分析方法:

- 看内循环的访问模式



内存中C数组的布局(回顾)

■ C 数组分配按行顺序

- 每行在连续的内存位置

■ 按行扫描:

- `for (i = 0; i < N; i++)`
 `sum += a[0][i];`
- 访问连续的元素
- 如果块大小 $(X) > \text{sizeof}(a_{ij})$ 字节, 利用空间局部性
 - 不命中率 = $\text{sizeof}(a_{ij}) / X$

■ 按列扫描:

- `for (i = 0; i < n; i++)`
 `sum += a[i][0];`
- 访问远隔的元素
- 没有空间局部性!
 - 不命中率 = 1 (i.e. 100%)

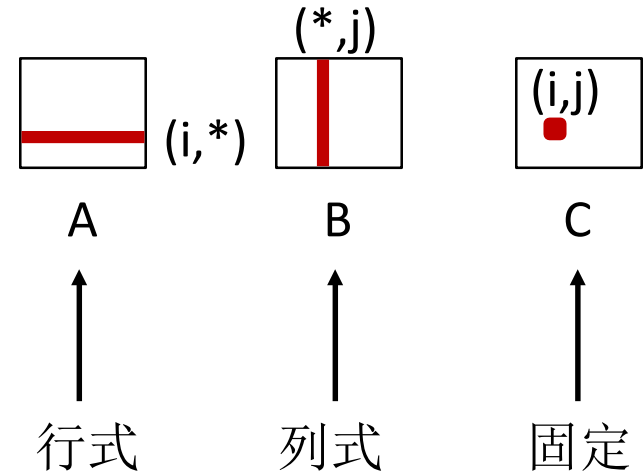
矩阵乘法(ijk)

```

/* 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

```

内部循环:



每个内部循环迭代不命中:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

块大小 = 32B (4 doubles)

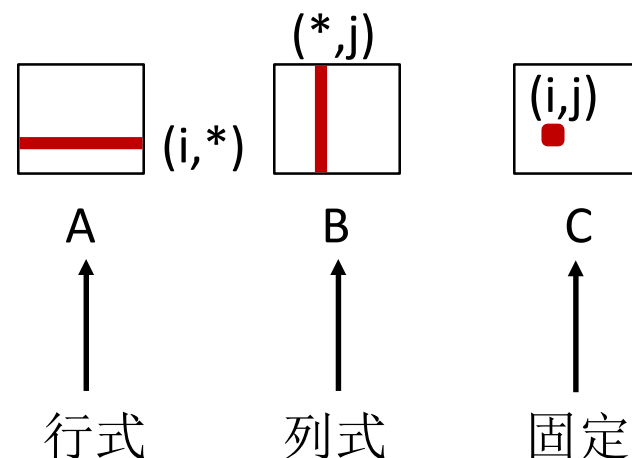
矩阵乘法(jik)

```

/* 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

```

内部循环:



每个内部循环迭代不命中:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

块大小 = 32B (4 doubles)

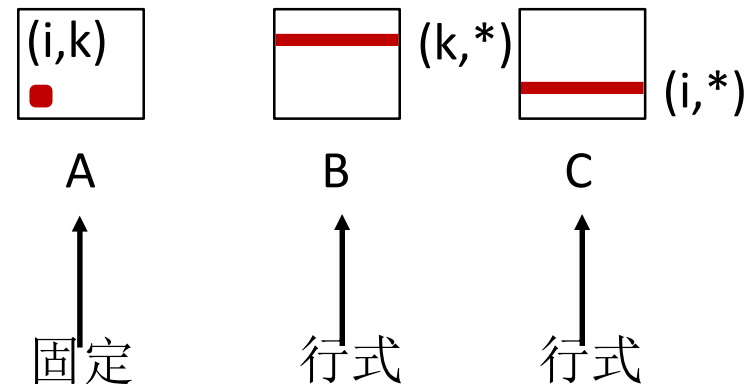
矩阵乘法(kij)

```

/* 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

```

内部循环:



每个内部循环迭代不命中:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

块大小 = 32B (4 doubles)

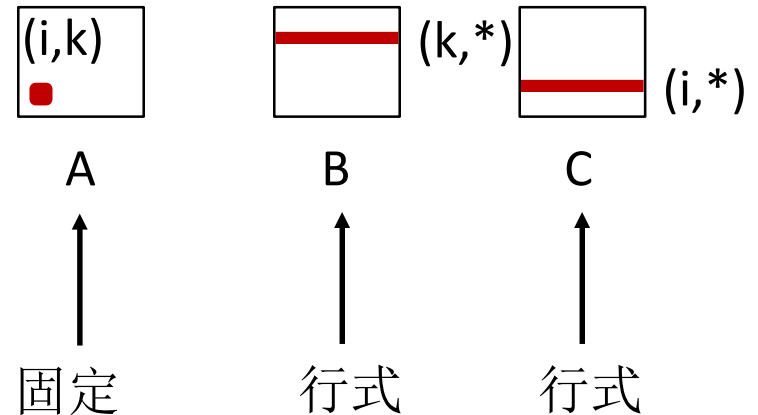
矩阵乘法 (ikj)

```

/* 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

```

内部循环:



每个内部循环迭代不命中:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

块大小 = 32B (4 doubles)

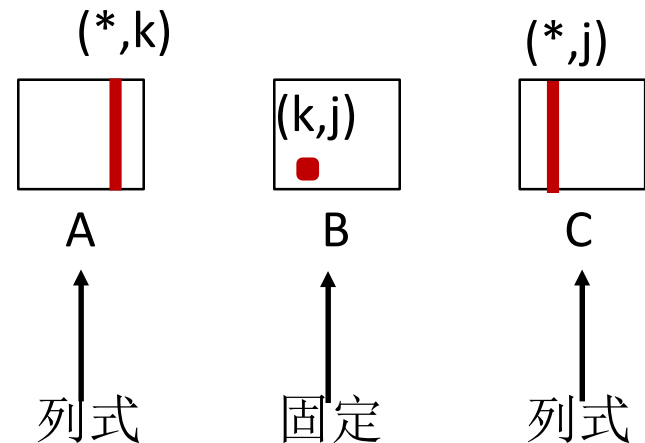
矩阵乘法 (jki)

```

/* 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

```

内部循环:



每个内部循环迭代不命中:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

块大小 = 32B (4 doubles)

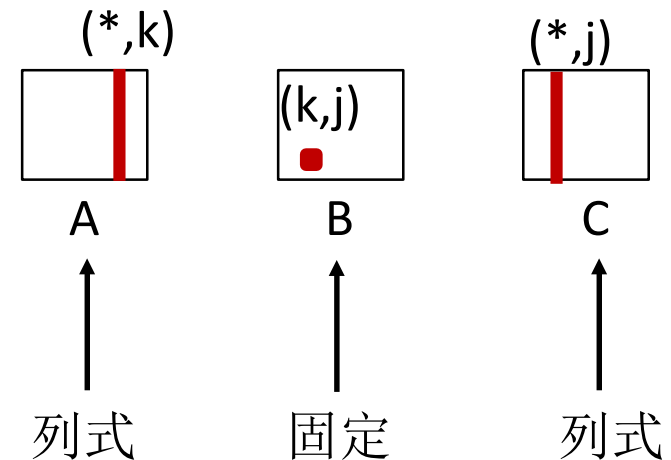
矩阵乘法 (kji)

```

/* 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

```

内部循环:



每个内部循环迭代不命中:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

块大小 = 32B (4 doubles)

矩阵乘法总结

```
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;
    }
}
```

```
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];
    }
}
```

```
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;
    }
}
```

ijk (& jik):

- 2 加载, 0 存储
- 不命中率/迭代次数 = **1.25**

前面求的不命中率直接相加就行

kij (& ikj):

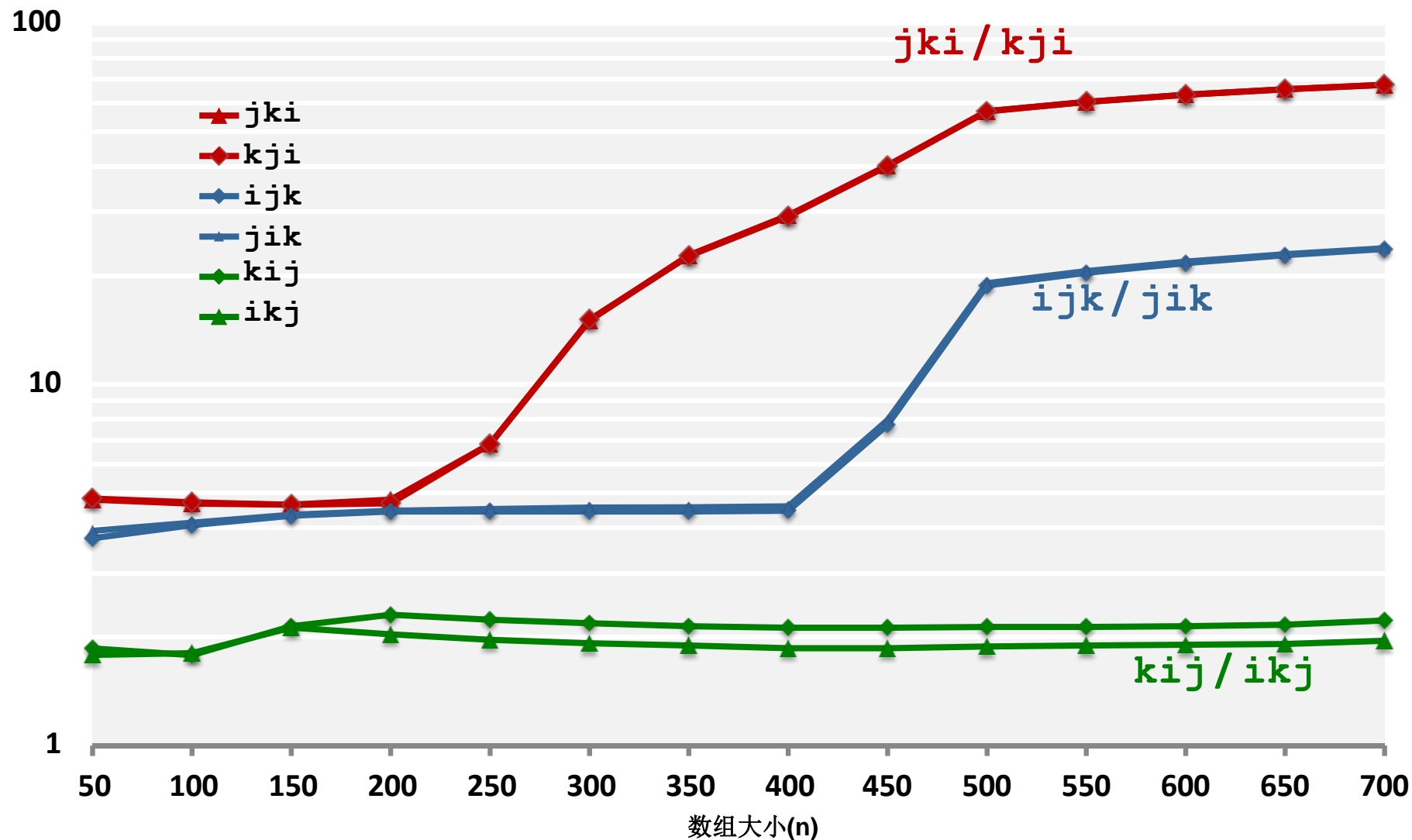
- 2加载, 1存储
- 不命中率/迭代次数 = **0.5**

jki (& kji):

- 2加载, 1存储
- 不命中率/迭代次数 = **2.0**

Core i7矩阵乘法性能

每个内部循环迭代周期



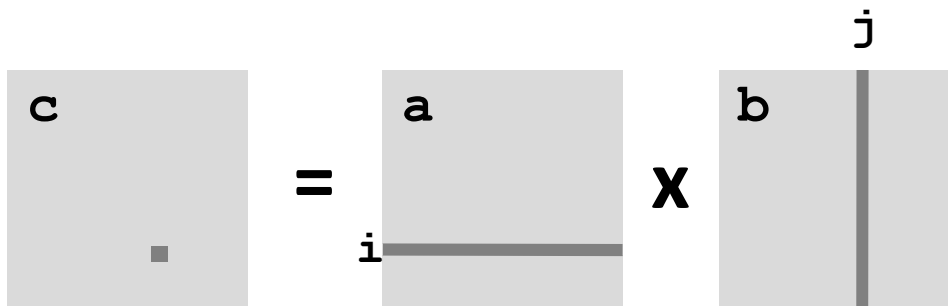
今天

- 高速缓存的组织结构和运算
- 高速缓存对程序性能的影响
 - 存储器山
 - 重新排列以提升空间局部性
 - 使用块来提高时间局部性

例子:矩阵乘法

```
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];
}
```



缓存不命中分析

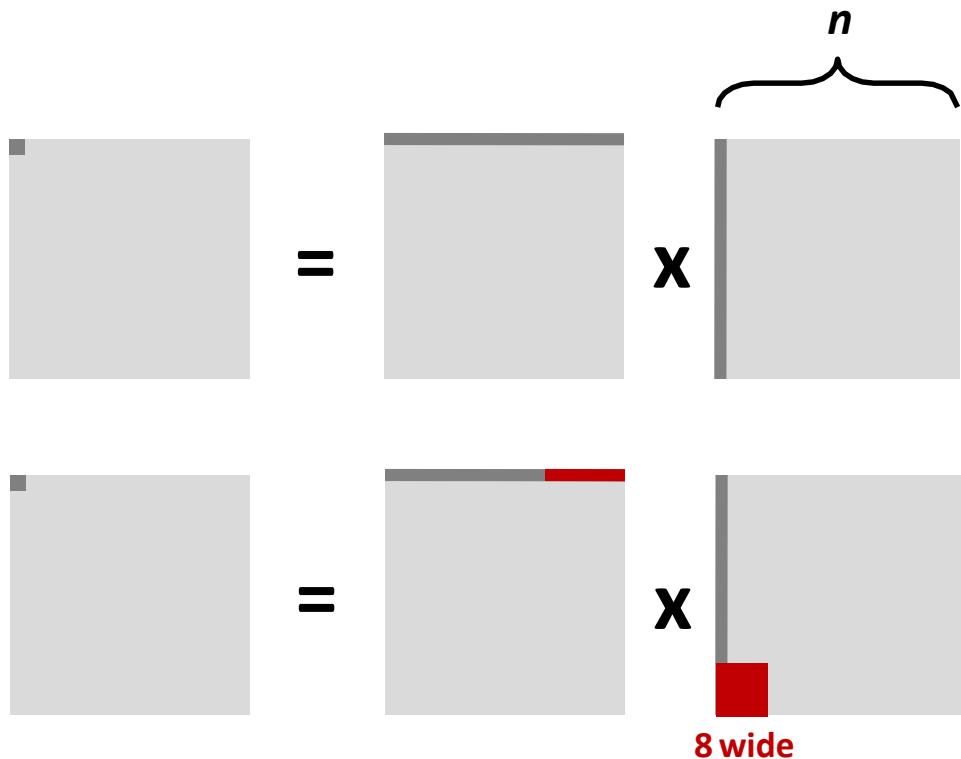
■ 假设:

- 矩阵元素类型是 double
- 缓存块 = 8 double
- 缓存大小 $C \ll n$ (比 n 小的得多)

■ 第一次迭代:

- $n/8 + n = 9n/8$ 不命中率

- 之后在缓存中:
(示意图)



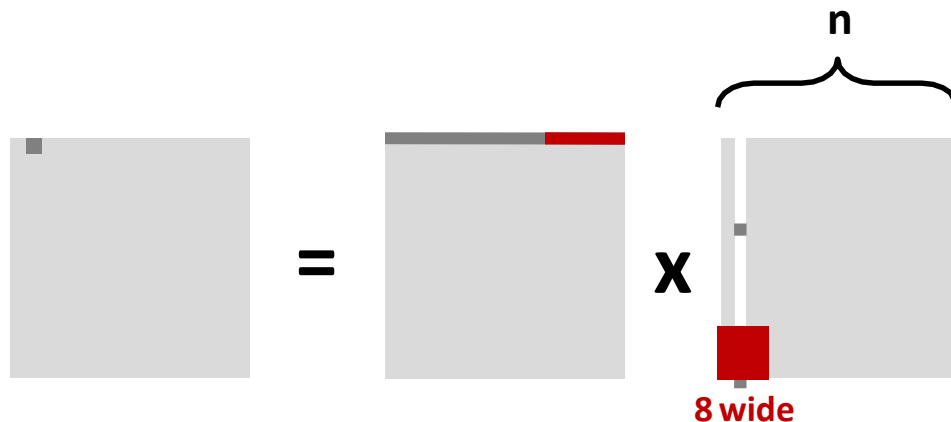
缓存不命中分析

■ 假设:

- 矩阵元素类型是double
- 缓存块 = 8 double
- 缓存大小 $C \ll n$ (比 n 小的得多)

■ 第二次迭代:

- 再次:
 $n/8 + n = 9n/8$ 不命中率



■ 总不命中率:

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

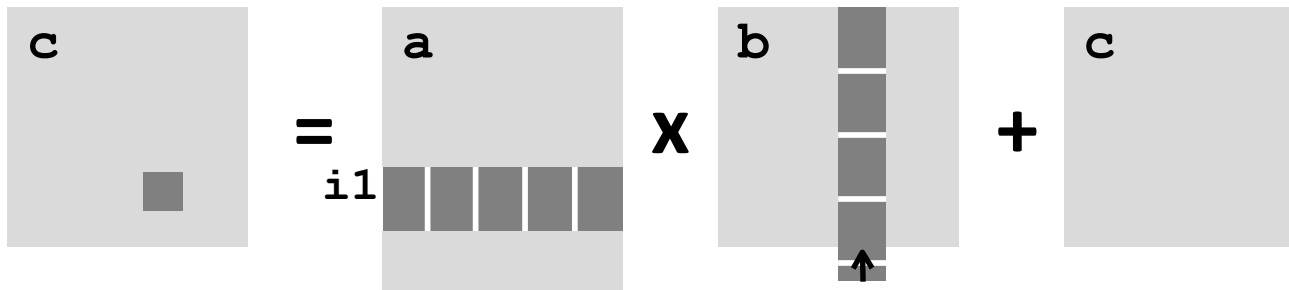
分块矩阵乘法

```
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; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

matmult/bmm.c

j1



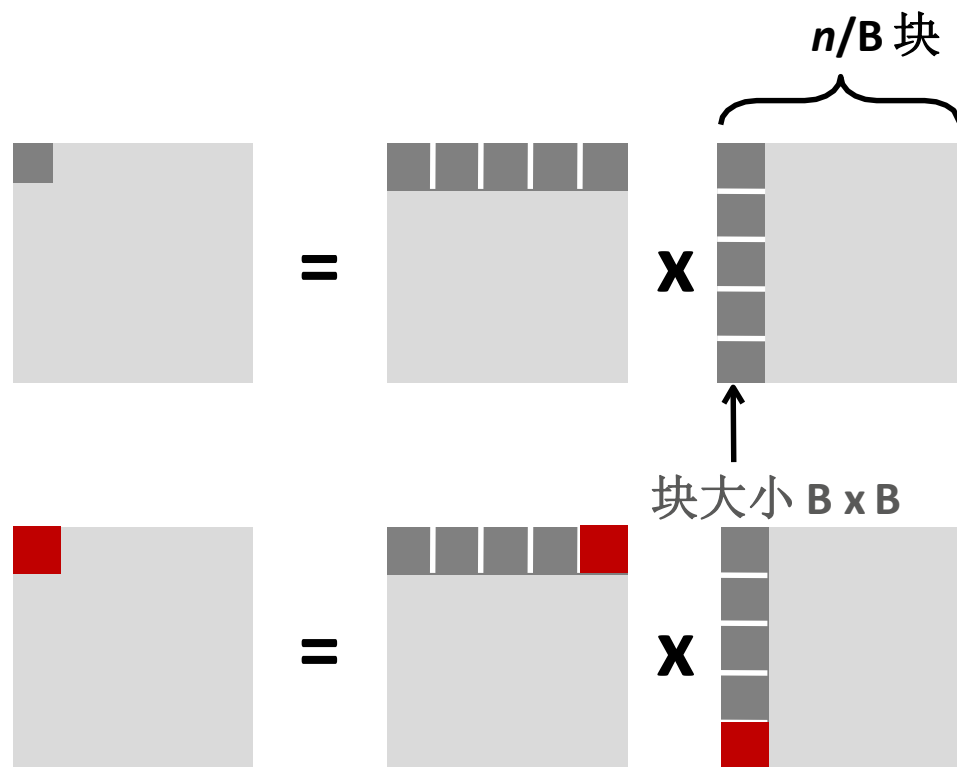
缓存不命中分析

■ 假设:

- 缓存块 = 8 double
- 缓存大小 $C \ll n$ (比 n 小的得多)
- 三块 放入缓存: $3B^2 < C$

■ 第一次 (块) 迭代:


- $B^2/8$ 每块不命中率
- $2n/B \times B^2/8 = nB/4$
(省略矩阵 c)



- 之后在缓存中:
- (示意图)

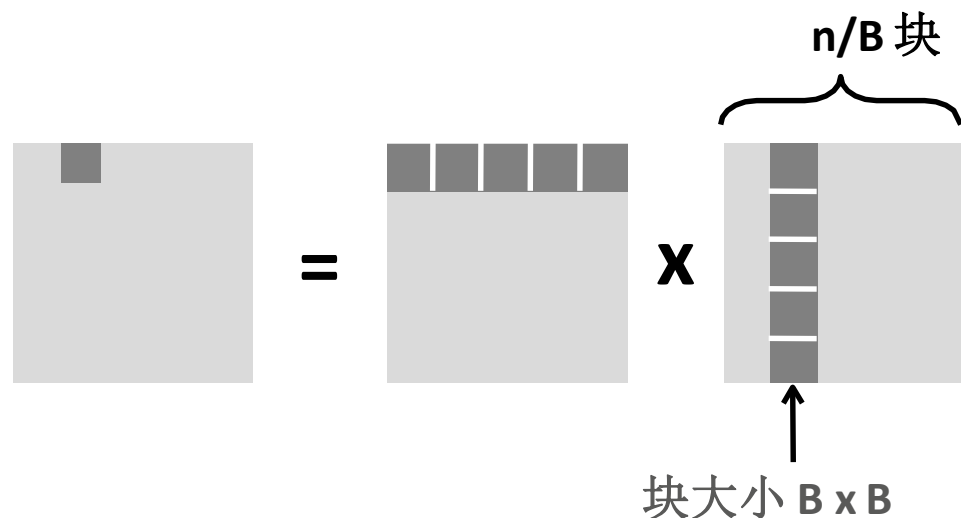
缓存不命中分析

■ 假设:

- 缓存块 = 8 double
- 缓存大小 $C \ll n$ (比 n 小的得多)
- 三块  放入缓存: $3B^2 < C$

■ 第二次 (块) 迭代:

- 同第一次迭代 n
- $2n/B \times B^2/8 = nB/4$



■ 总不命中率:

- $nB/4 * (n/B)^2 = n^3/(4B)$

分块总结

- 不分块: $(9/8) n^3$
- 分块: $1/(4B) n^3$
- 建议最大可能的块B大小, 限制为 $3B^2 < C!$
- 巨大差距的原因:
 - 矩阵乘法有天生的时间局部性:
 - 输入数据: $3n^2$, 计算 $2n^3$
 - 每个数据组元素使用 $O(n)$ 时间!
 - 但是程序必须被恰当的编写

高速缓存总结

- 对缓存存储器的性能有明显的影响
- 你可以在你的程序里用这个!
 - 集中在内部循环上，大部分计算和内存访问都发生在这里。
 - 尽量按照数据对象存储在内存中的顺序，以步长为1来读数据，使得程序的空间局部性最大。
 - 一旦从内存中读入一个数据对象，尽可能多的使用它，使得程序中的时间局部性最大。

*Hope you
enjoyed
the
CSAPP
course!*