第5周 Cache替换算法和写策略

第1讲 Cache替换算法

第2讲 Cache写策略（一致性问题）

第3讲 Cache实现的几个因素

第4讲 Cache实现举例

第5讲 Cache综合计算举例

替换(Replacement)算法

- 问题举例：

  组相联映射时，假定第0组的两行分别被主存第0和8块占满，此时若需调入主存第16块，根据映射关系，它只能放到Cache第0组，因此，第0组中必须调出一块，那么调出哪一块呢？

  这就是淘汰策略问题，也称替换算法。

- 常用替换算法有：
  - 先进先出FIFO（first-in-first-out）
  - 最近最少用LRU（ least-recently used）
  - 最不经常用LFU（ least-frequently used）
  - 随机替换算法（Random）

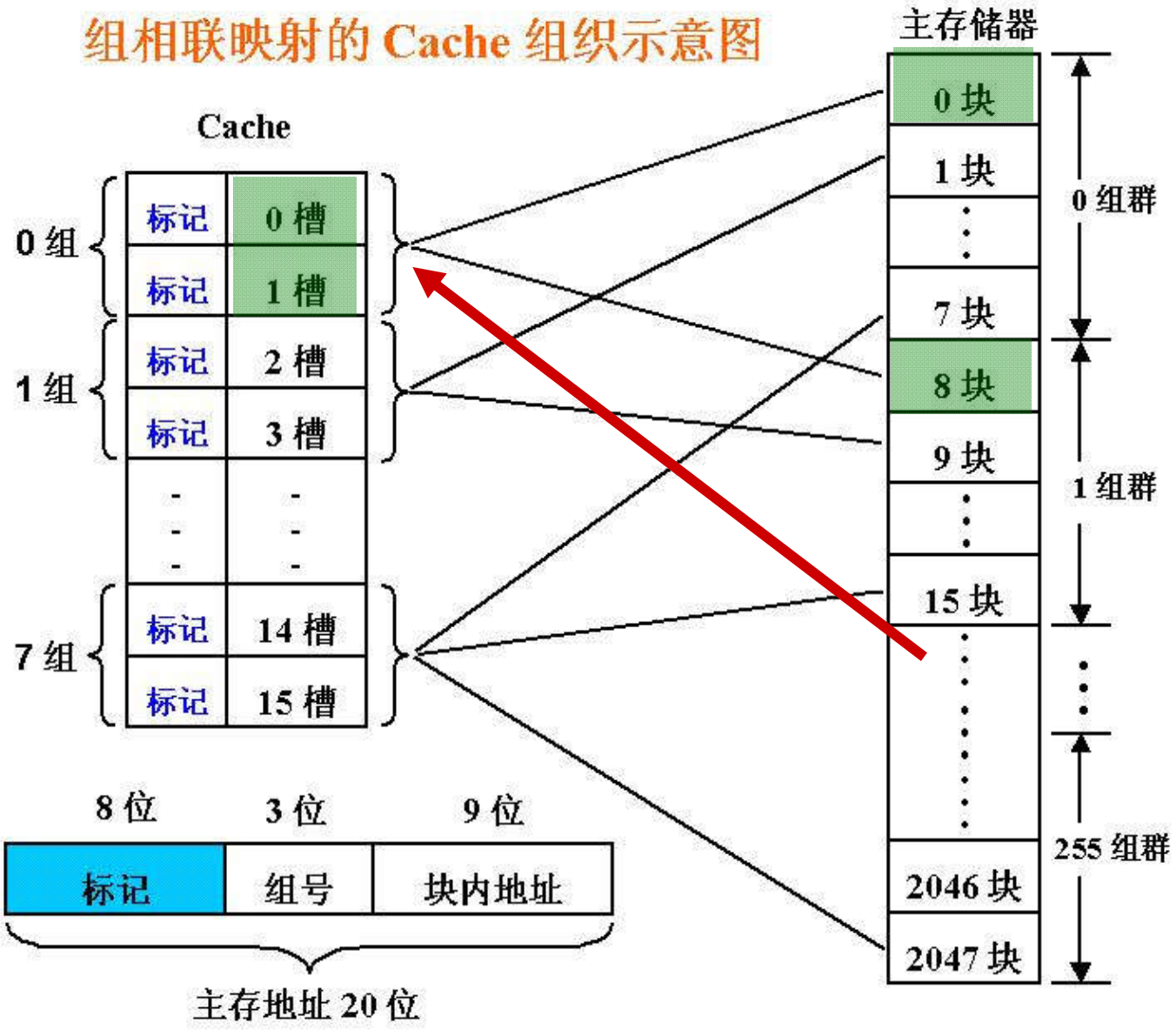
  等等

SKIP

这里的替换策略和后面的虚拟存储器所用的替换策略类似，将是以后操作系统课程的重要内容，本课程只做简单介绍。有兴趣的同学可以自学。

假定第0组的两行分别被主存第0和8块占满，此时若需调入主存第16块
该怎么办？

第0组中必须调出一块，那么，调出哪一块呢？



BACK

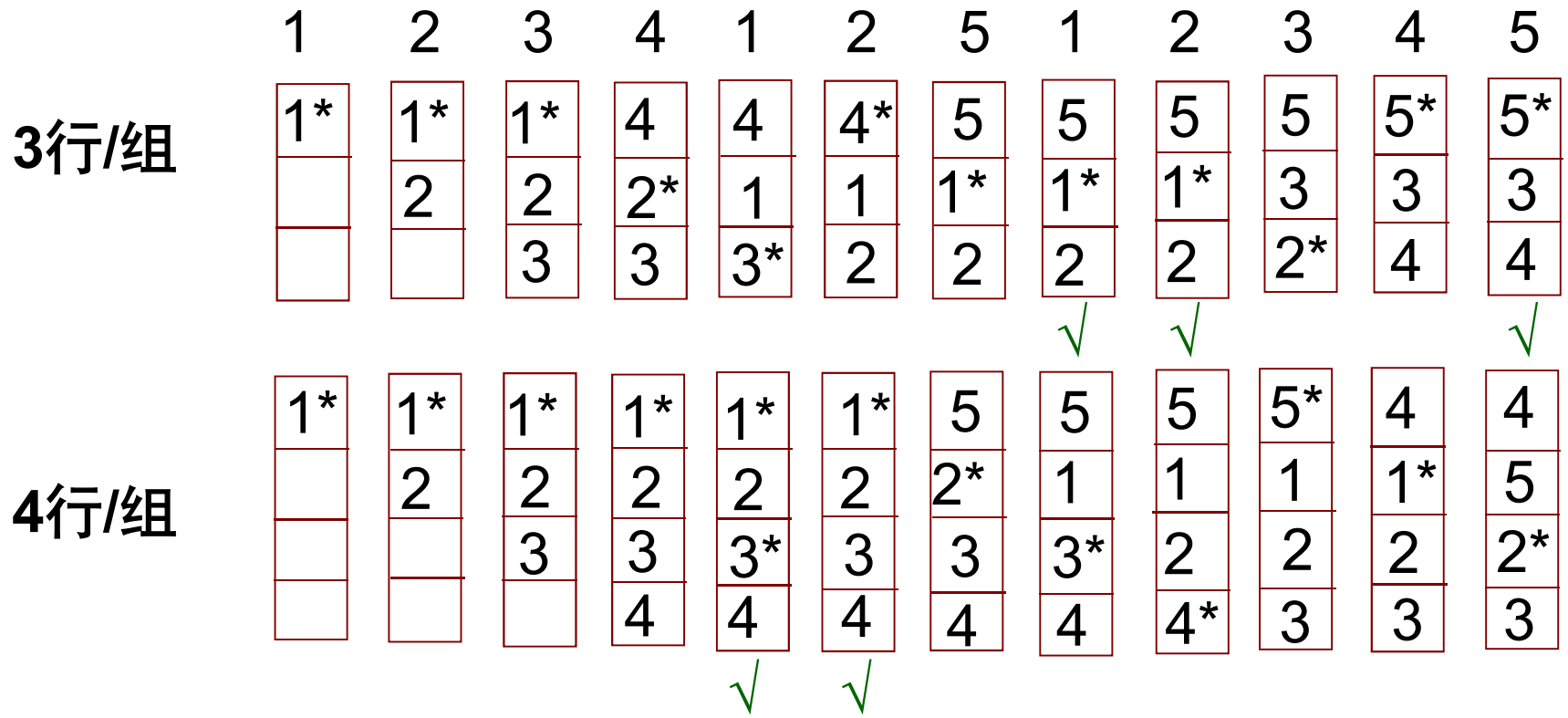
替换算法-先进先出（FIFO）

- 总是把最先进入的那一块淘汰掉。

总是把最先从图书馆搬来的书还回去！

例：假定主存中的5块{1,2,3,4,5}同时映射到Cache同一组中，对于同一地址流，考察3行/组、 4行/组的情况。

注：通常一组中含有2^k行，这里3行/组主要为了简化问题而假设



由此可见，FIFO不是一种栈算法，即命中率并不随组的增大而提高。

替换算法-最近最少用(LRU)

总是把最长时间不看的书还回去！

- 总是把最近最少用的那一块淘汰掉。

例：假定主存中的5块{1,2,3,4,5}同时映射到Cache同一组中，对于同一地址流，考察3行/组、 4行/组、 5行/组的情况。

	1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	1	2	3	4	5
		1	2	3	4	1	2	5	1	2	3	4
			1	2	3	4	1	2	5	1	2	3
				1	2	3	4	4	4	5	1	2
							3	3	3	4	5	1
3行/组								√	√			
4行/组					√	√		√	√			
5行/组					√	√		√	√	√	√	√

替换算法-最近最少用

- LRU是一种栈算法，它的命中率随组的增大而提高。
- 当分块局部化范围(即：某段时间集中访问的存储区)超过了Cache存储容量时，命中率变得很低。极端情况下，假设地址流是1,2,3,4,1 2,3,4,1,……，而Cache每组只有3行，那么，不管是FIFO，还是LRU算法，其命中率都为0。这种现象称为颠簸(Thrashing / PingPong)。
- LRU具体实现时，并不是通过移动块来实现的，而是通过给每个cache行设定一个计数器，根据计数值来记录这些主存块的使用情况。这个计数值称为**LRU位**。

具体实现

替换算法-最近最少用

通过计数值来确定cache行中主存块的使用情况

即：计数值为0的行中的主存块最常被访问，计数值为3的行中的主存块最不经常被访问，先被淘汰！

- 计数器变化规则：
  - 每组4行时，计数器有2位。计数值越小则说明越被常用。
  - 命中时，被访问行的计数器置0，比其低的计数器加1，其余不变。
  - 未命中且该组未满时，新行计数器置为0，其余全加1。
  - 未命中且该组已满时，计数值为3的那一行中的主存块被淘汰，新行计数器置为0，其余加1。

1	2	3	4	1	2	5	1	2	3	4	5
0	1	1	1	2	1	3	1	0	1	1	1	2	1	0	1	1	1	2	1	3	1	0	5
		0	2	1	2	2	2	3	2	0	2	1	2	2	2	0	2	1	2	2	2	3	4
				0	3	1	3	2	3	3	3	0	5	1	5	2	5	3	5	0	4	2	3
						0	4	1	4	2	4	3	4	3	4	3	4	0	3	1	3	1	2

The Need to Replace! (何时需要替换？)

- Direct Mapped Cache:
  - 映射唯一，毫无选择，无需考虑替换
- N-way Set Associative Cache:
  - 每个主存数据有N个Cache行可选择，需考虑替换
- Fully Associative Cache:
  - 每个主存数据可存放到Cache任意行中，需考虑替换

结论：若Cache miss in a N-way Set Associative or Fully Associative Cache，则可能需要替换。其过程为：

- 从主存取出一个新块
- 选择一个有映射关系的空Cache行
- 若对应Cache行被占满时又需调入新主存块，则必须考虑从Cache行中替换出一个主存块

举例

- 假定计算机系统主存空间大小为32Kx16位，且有一个4K字的4路组相联Cache，主存和Cache之间的数据交换块的大小为64字。假定Cache开始为空，处理器顺序地从存储单元0、1、…、4351中取数，一共重复10次。设Cache比主存快10倍。采用LRU算法。试分析Cache的结构和主存地址的划分。说明采用Cache后速度提高了多少？采用MRU算法后呢？
- 答：假定主存按字编址。每字16位。

  主存：32K字=512块 x 64字 / 块

  Cache：4K字=16组 x 4行 / 组 x 64 字 / 行

  主存地址划分为：

标志位	组号	字号
5	4	6

4352/64=68，所以访问过程实际上是对前68块连续访问10次。

举例

	第0 行	第1 行	第2 行	第3 行
第0组	0/64/48	16/0/64	32/16	48/32
第1组	1/65/49	17/1/65	33/17	49/33
第2组	2/66/50	18/2/66	34/18	50/34
第3组	3/67/51	19/3/67	35/19	51/35
第4组	4	20	36	52
……	……	……	……	……
……	……	……	……	……
第15组	15	31	47	63

LRU算法：第一次循环,每一块只有第一字未命中,其余都命中;
以后9次循环,有20块的第一字未命中,其余都命中.
所以,命中率p为 (43520-68-9x20)/43520=99.43%
速度提高：tm/ta=tm/(tc+(1-p)tm)=10/(1+10x(1-p))=9.5倍

举例

	第0 行	第1 行	第2 行	第3 行
第0组	0/16/32/48	16/32/48/64	32/48/64/0	48/64/0/16
第1组	1/17/33/49	17/33/49/65	33/49/65/1	49/65/1/17
第2组	2/18/34/50	18/34/50/66	34/50/66/2	50/66/2/18
第3组	3/19/35/52	19/35/51/67	35/51/67/3	51/67/3/19
第4组	4	20	36	52
……	…	…	…	…
……	…	…	…	…
第15组	15组	31	47	63

MRU算法：第一次68字未命中；第2,3,4,6,7,8,10次各有4字未命中；第5,9次各有8字未命中；其余都命中.

所以,命中率p为 (43520-68-7x4-2 x8)/43520=99.74%

速度提高：tm/ta=tm/(tc+(1-p)tm)=10/(1+10x(1-p))=9.77倍

写策略（Cache一致性问题）

- 为何要保持在Cache和主存中数据的一致？
  - 因为Cache中的内容是主存块副本，当对Cache中的内容进行更新时，就存在Cache和主存如何保持一致的问题。
  - 以下情况也会出现“Cache一致性问题”
    - 当多个设备都允许访问主存时

      例如：I/O设备可直接读写内存时，如果Cache中的内容被修改，则I/O设备读出的对应主存单元的内容无效；若I/O设备修改了主存单元的内容，则Cache中对应的内容无效。
    - 当多个CPU都带有各自的Cache而共享主存时

      某个CPU修改了自身Cache中的内容，则对应的主存单元和其他CPU中对应的内容都变为无效。
- 写操作有两种情况
  - 写命中（Write Hit）：要写的单元已经在Cache中
  - 写不命中（Write Miss）：要写的单元不在Cache中

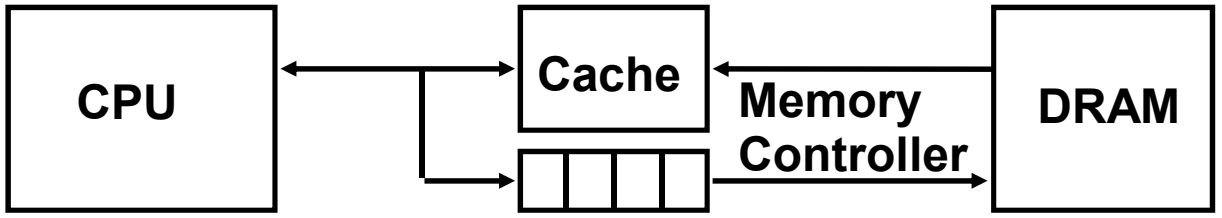
写策略（Cache一致性问题）

- 处理Cache读比Cache写更容易，故指令Cache比数据Cache容易设计
- 对于写命中，有两种处理方式
  - Write Through (通过式写、写直达、直写)
    - 同时写Cache和主存单元
    - What!!! How can this be? Memory is too slow(>100Cycles)?
      10%的存储指令使CPI增加到：1.0+100x10%=11
    - 使用写缓冲（Write Buffer）
  - Write Back (一次性写、写回、回写)
    - 只写cache不写主存，缺失时一次写回，每行有个修改位（“dirty bit-脏位”），大大降低主存带宽需求，控制可能很复杂
- 对于写不命中，有两种处理方式
  - Write Allocate (写分配)
    - 将主存块装入Cache，然后更新相应单元
    - 试图利用空间局部性，但每次都要从主存读一个块
  - Not Write Allocate (非写分配)
    - 直接写主存单元，不把主存块装入到Cache

直写Cache可用非写分配或写分配
回写Cache通常用写分配 为什么？

SKIP

Write Through中的Write Buffer



- 在 Cache 和 Memory之间加一个Write Buffer
  - CPU同时写数据到Cache和Write Buffer
  - Memory controller（存控）将缓冲内容写主存
- Write buffer (写缓冲) 是一个FIFO队列
  - 一般有4项
  - 在存数频率不 高时效果好
- 最棘手的问题
  - 当频繁写时，易使写缓存饱和，发生阻塞
- 如何解决写缓冲饱和？
  - 加一个二级Cache
  - 使用Write Back方式的Cache

BACK

写策略（Cache一致性问题）

问题1：以下描述的是哪种写策略？

Write Through 、Write Allocate !

问题2：如果用非写分配，则如何修改算法？

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X == TAG(i) , for some cache line i*

READ: return DATA[I]
WRITE: change DATA[I]; Start Write to Mem[X]

MISS: *X not found in TAG of any cache line*

REPLACEMENT SELECTION:
Select some line k to hold Mem[X]

READ: Read Mem[X]
Set TAG[k] = X, DATA[k] = Mem[X]

WRITE: Start Write to Mem[X]
~~Set TAG[k] = X, DATA[k] = new Mem[X]~~

BACK

写策略2：Write Back算法

问题：以下算法描述的是哪种写策略？

Write Back 、Write Allocate !

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X = TAG(i) , for some cache line I*

READ: return DATA(i)
WRITE: change DATA(i); ~~Start Write to Mem[X]~~

MISS: *X not found in TAG of any cache line*

REPLACEMENT SELECTION:
Select some line k to hold Mem[X]
Write Back: Write Data(k) to Mem[Tag[k]]

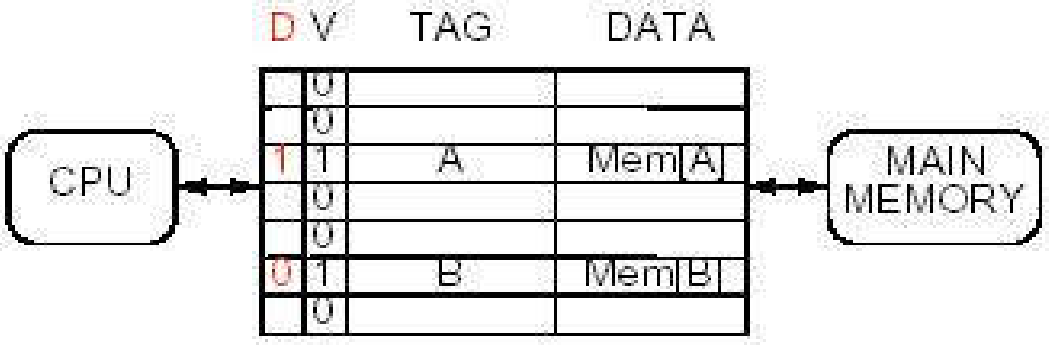
READ: Read Mem[X]
Set TAG[k] = X, DATA[k] = Mem[X]

WRITE: ~~Start Write to Mem[X]~~
Set TAG[k] = X, DATA[k] = new Mem[X]

Is write-back worth the trouble? Depends on (1) cost of write; (2) consistency issues.

写策略2：Write Back中的修改（"脏"）位

Write-back w/ "Dirty" bits



BACK

ON REFERENCE TO Mem[X]: Look for X among tags...

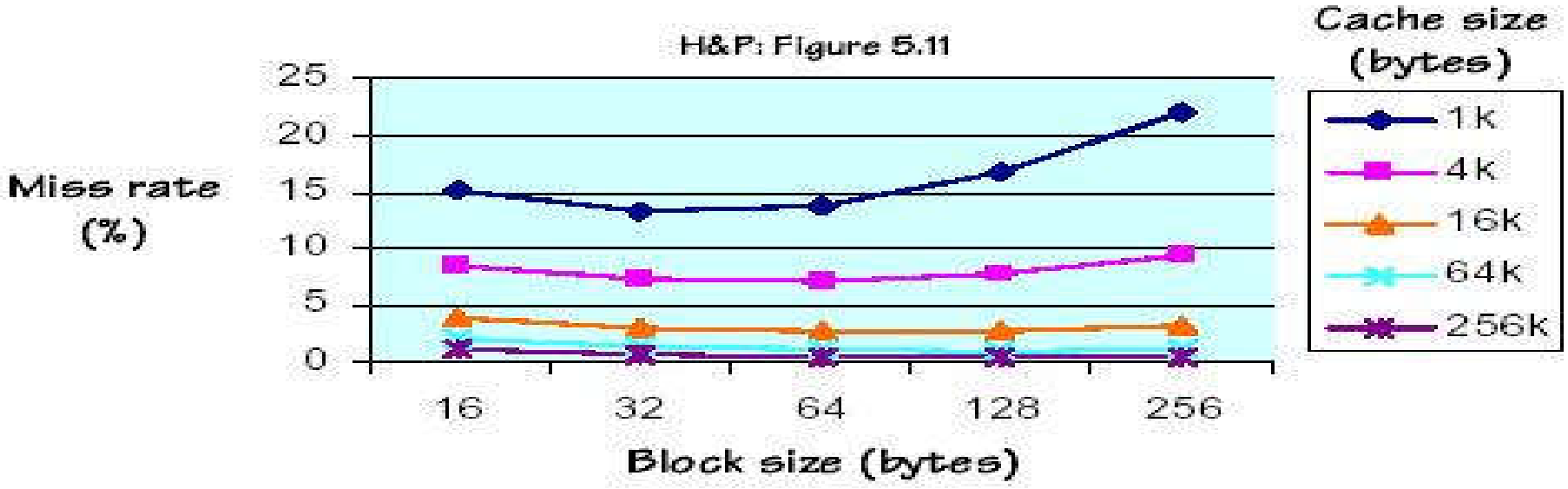
HIT: *X = TAG(i) , for some cache line I*
READ: return DATA(i)
WRITE: change DATA(i); ~~Start Write to Mem[X]~~ D[i]=1

MISS: *X not found in TAG of any cache line*
REPLACEMENT SELECTION:
Select some line k to hold Mem[X]
If D[k] == 1 (Write Back) Write Data(k) to Mem[Tag[k]]
READ: Read Mem[X]; Set TAG[k] = X, DATA[k] = Mem[X], D[k]=0
WRITE: ~~Start Write to Mem[X]~~ D[k]=1
Set TAG[k] = X, DATA[k] = new Mem[X]

Cache大小、Block大小和缺失率的关系

Cache性能由缺失率确定

而缺失率与Cache大小、Block大小等有关



- spatial locality: larger blocks → reduce miss rate
- fixed cache size: larger blocks
  - → fewer lines in cache
  - → higher miss rate, especially in small caches

Cache大小：Cache越大，Miss率越低，但成本越高！

Block大小：Block大小与Cache大小有关，且不能太大，也不能太小！

系统中的Cache数目

- 刚引入Cache时只有一个Cache。近年来多Cache系统成为主流
- 多Cache系统中，需考虑两个方面：

  [1] 单级/多级？

  外部(Off-chip)Cache:不做在CPU内而是独立设置一个Cache

  片内(On-chip)Cache: 将Cache和CPU作在一个芯片上

  单级Cache：只用一个片内Cache

  多级Cache：同时使用L1 Cache和L2 Cache，甚至有L3 Cache，L1 Cache更靠近CPU，其速度比L2快，其容量比L2小

  [2] 联合/分立？

  分立：指数据和指令分开存放在各自的数据和指令Cache中

  一般L1 Cache都是分立Cache，为什么？

  L1 Cache的命中时间比命中率更重要！为什么？

  联合：指数据和指令都放在一个Cache中

  一般L2 Cache都是联合Cache，为什么？

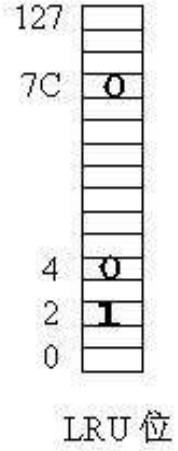
  L2 Cache的命中率比命中时间更重要！为什么？

  因为缺失时需从主存取数，并要送L1和L2cache，损失大！

实例：奔腾机的Cache组织

主存：4GB=2^{20}x 2^{7}块x 2^{5}B/块

Cache：8KB=128组x2行/组



替换算法：

LRU，每组一位LRU位

0：下次淘汰第0路

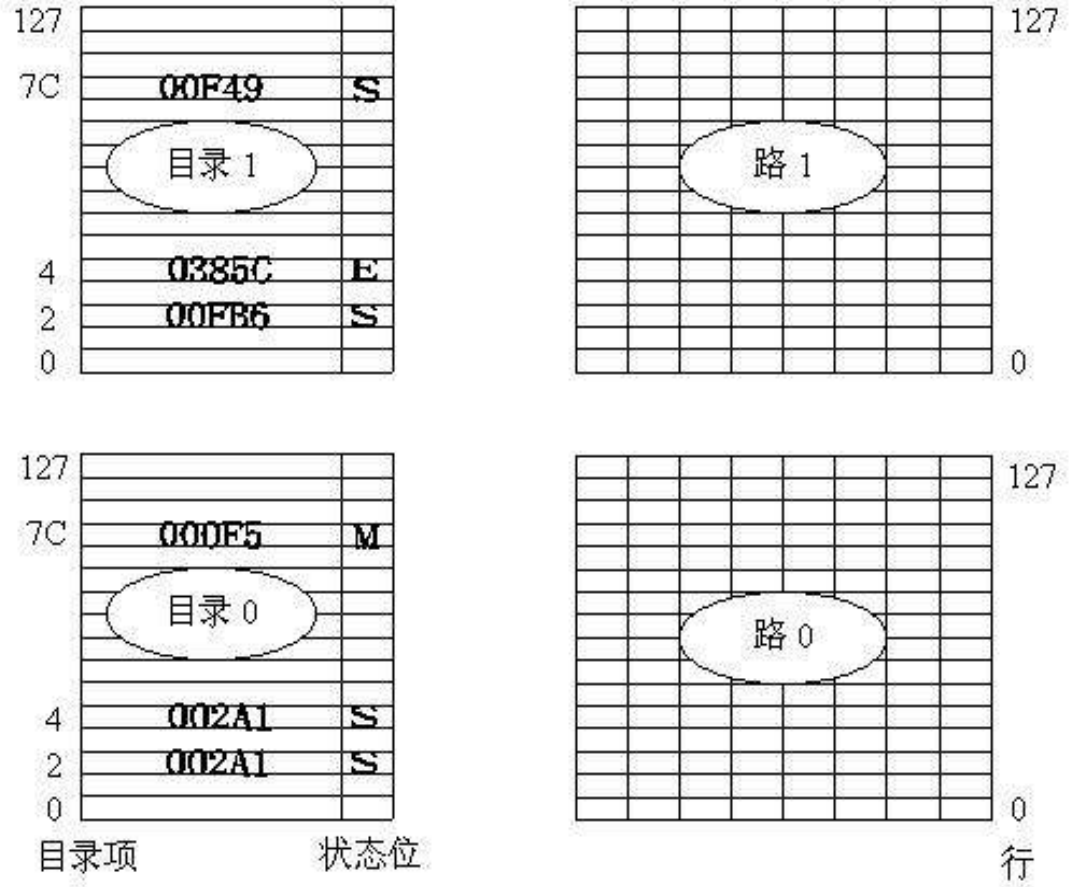
1：下次淘汰第1路

写策略：

默认为Write Back，可动态设置为Write Through。

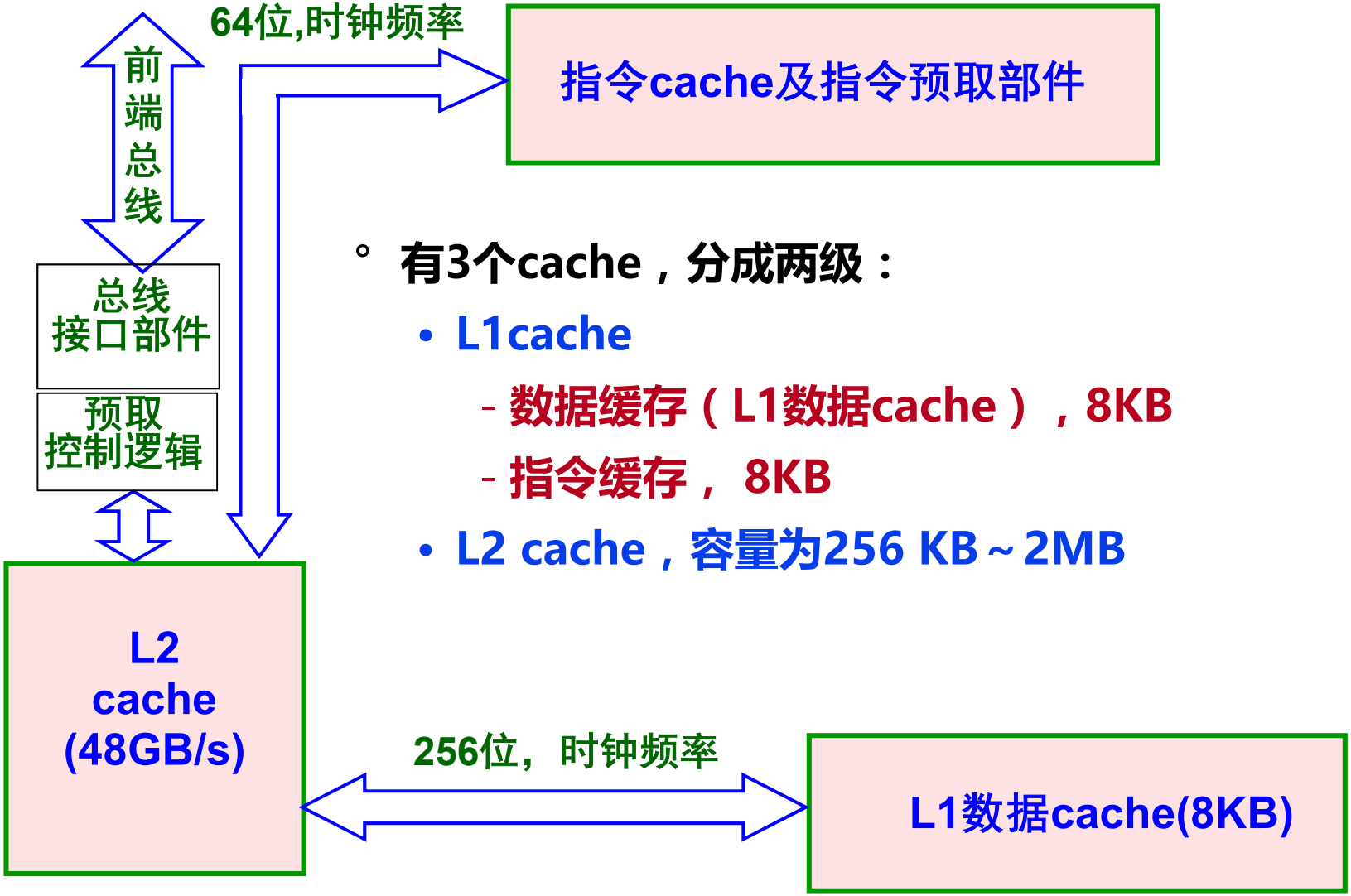
Cache一致性：

支持MESI协议



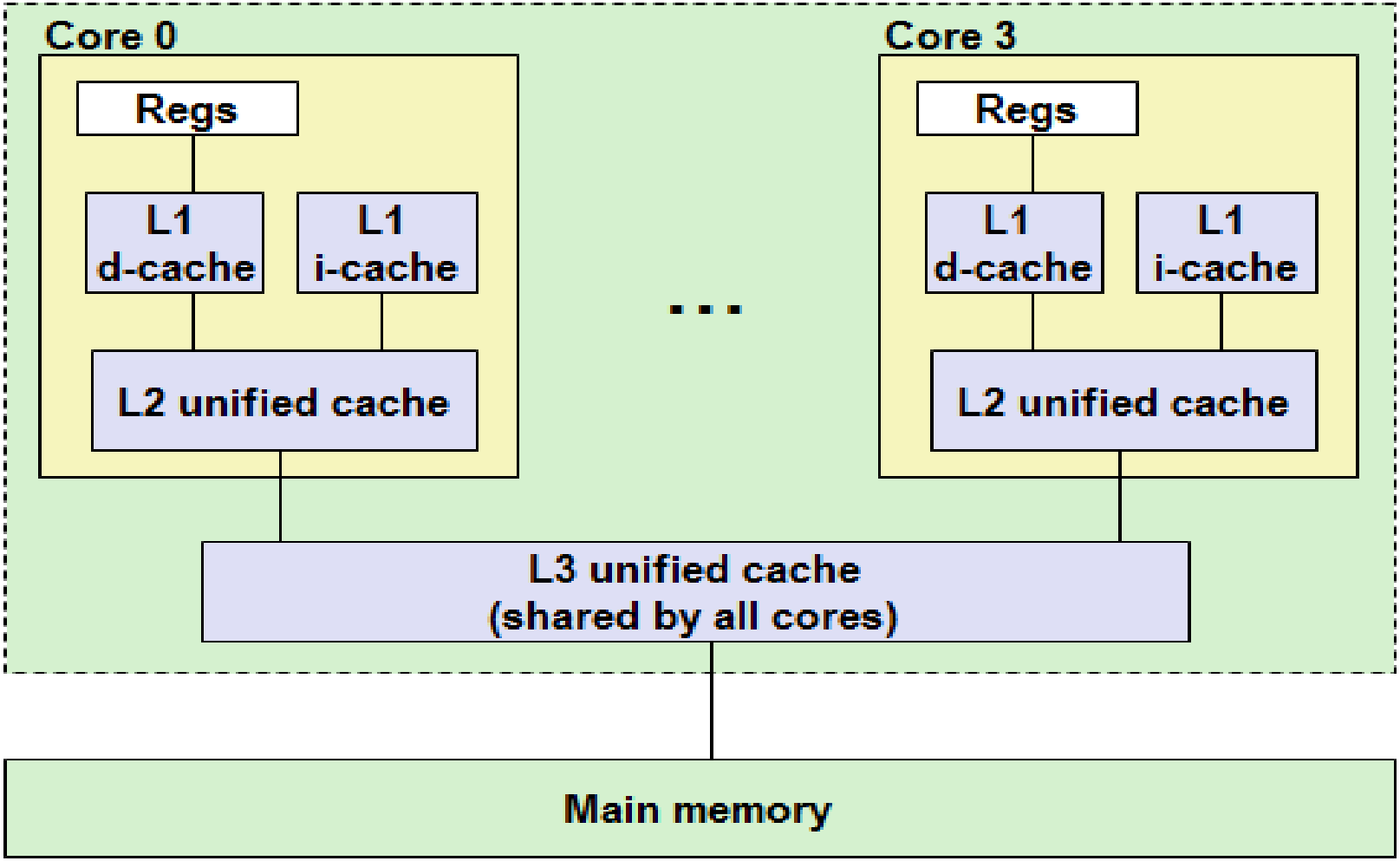
Pentium 内部数据 Cache 的结构

实例：Pentium 4的cache存储器



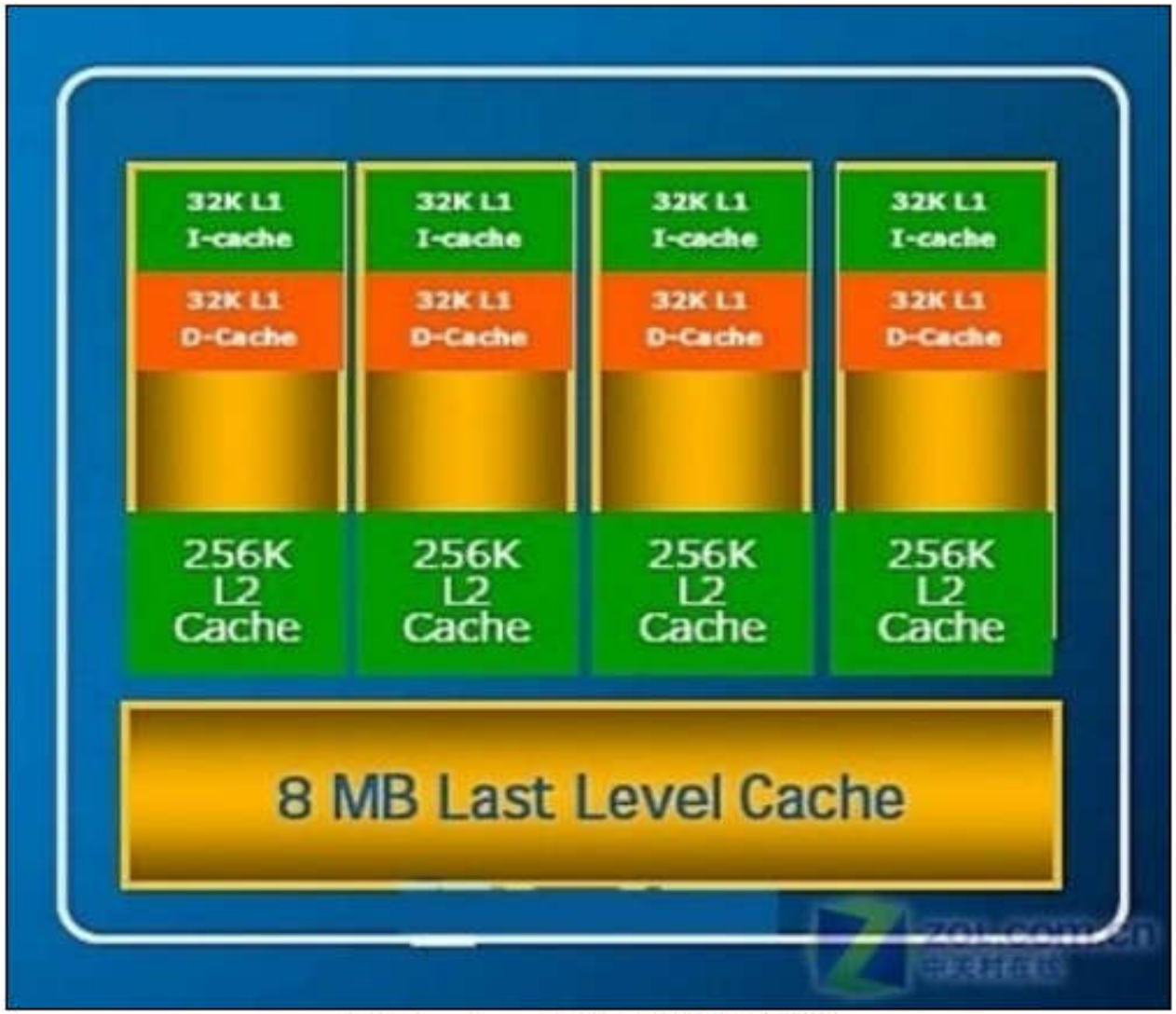
- 有3个cache，分成两级：
  - L1cache
    - 数据缓存（L1数据cache），8KB
    - 指令缓存， 8KB
  - L2 cache，容量为256 KB～2MB

实例：Intel Core i7处理器的cache结构



i-cache和d-cache都是32KB、8路、4个时钟周期；L2 cache：256KB、8路、11个时钟周期。所有核共享的L3 cache：8MB、16路、30~40个时钟周期。Core i7中所有cache的块大小都是64B

多核处理器中的多级Cache



Nehalem Core i7处理器缓存结构图

Per core:

-32KB, 4-way L1 $I

-32KB, 8-way L1 $D

-256KB, 8-way L2

Shared

- 8 MB, 16-way L3

缓存在现代计算机中无处不在

Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte word	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware
L1 cache	32-byte block	On-chip L1 cache	1	Hardware
L2 cache	32-byte block	Off-chip L2 cache	10	Hardware
Virtual memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/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

Cache和程序性能举例

- 举例：某32位机器主存地址空间大小为256 MB，按字节编址。指令cache和数据cache均有8行，主存块为64B，数据cache采用直接映射。假定编译时i, j, sum均分配在寄存器中，数组a按行优先方式存放，其首址为320。

程序 A：

```
int a[256][256];
……
int sum_array1 ( )
{
        int i, j, sum = 0;
        for ( i = 0; i < 256; i++)
                for (j = 0; j < 256; j++)
                        sum += a[i] [j];
        return sum;
}
```

程序 B：

```
int a[256][256];
……
int sum_array2 ( )
{
        int i, j, sum = 0;
        for ( j = 0; j < 256; j++)
                for ( i = 0; i < 256; i++)
                        sum += a[i] [j];
        return sum;
 }
```

（1）不考虑用于一致性和替换的控制位，数据cache的总容量为多少？

（2）a[0][31]和a[1][1]各自所在主存块对应的cache行号分别是多少？

（3）程序A和B的数据访问命中率各是多少？哪个程序的执行时间更短？

Cache和程序性能举例

- 举例：某32位机器主存地址空间大小为256 MB，按字节编址。指令cache和数据cache均有8行，主存块为64B，数据cache采用直接映射。假定编译时i, j, sum均分配在寄存器中，数组a按行优先方式存放，其首址为320。

（1）主存地址空间大小为256MB，因而主存地址为28位，其中6位为块内地址，3位为cache行号（行索引），标志信息有28-6-3=19位。在不考虑用于cache一致性维护和替换算法的控制位的情况下，数据cache的总容量为：8×(19+1+64×8)=4256位=532字节 。

（2）a[0][31]的地址为320+4×31=444，[444/64]=6（取整），因此a[0][31]对应的主存块号为6。6 mod 8=6，对应cache行号为6。

或: 444=0000 0000 0000 0000 000 110 111100B，中间3位110为行号（行索引），因此，对应的cache行号为6。 a[1][1]对应的cache行号为：[(320+4×(1×256+1))/64] mod 8=5。

（3）A中数组访问顺序与存放顺序相同，共访问64K次，占4K个主存块；首地址位于一个主存块开始，故每个主存块总是第一个元素缺失，其他都命中，共缺失4K次，命中率为1-4K/64K=93.75%。

方法二：每个主存块的命中情况一样。对于一个主存块，包含16个元素，需访存16次，其中第一次不命中，因而命中率为15/16=93.75%。

B中访问顺序与存放顺序不同，依次访问的元素分布在相隔256×4=1024的单元处，它们都不在同一个主存块中,cache共8行，一次内循环访问16块，故再次访问同一块时，已被调出cache，因而每次都缺失，命中率为0。

Cache和程序性能举例

a[0][0]所在主存块号为：320/64=5

一个主存块占64B/4B=16个元素

总访问次数为:256x256=64K

总块数(缺失次数)为64Kx4B/64B=4K

缺失率为：4K/64K=1/16

命中率为：1-4K/64K=15/16

° 程序A对数组元素的访问过程：

直接映射

5#: a[0][0], a[0][1], ..., a[0][15] → → → 第5行

6#: a[0][16], a[0][17], ..., a[0][31] → → → 第6行

7#: a[0][32], a[0][33], ..., a[0][47] → → → 第7行

8#: a[0][48], a[0][49], ..., a[0][63] → → → 第0行

.....

每块第1次都不命中

....., a[255][255]

每块都一样，因此，可以仅考虑一个主存块的情况：

第1次不命中，以后15次都命中，故命中率为15/16

Cache和程序性能举例

a[0][0]所在主存块号为：320/64=5

一个主存块占64B/4B=16个元素

每行数组元素占256x4B=1024B
即1024B/64B=16块

a[i][0]和a[i+1][0]之间相差1024B，即16块，因为16 mod 8=0
因此，被映射到cache同一行中！

° 程序B对数组元素的访问过程：

直接映射

5#: a[0][0], a[0][1], ..., a[0][15] → → → 第5行

.....

21#: a[1][0], a[1][1], ..., a[1][15] → → → 第5行

.....

37#: a[2][0], a[2][1], ..., a[2][15] → → → 第5行

.....

....., a[255][255]

访问后面数组元素时，总是把上一次装入到cache中的主存块覆盖掉！

每次都不命中，故命中率为0