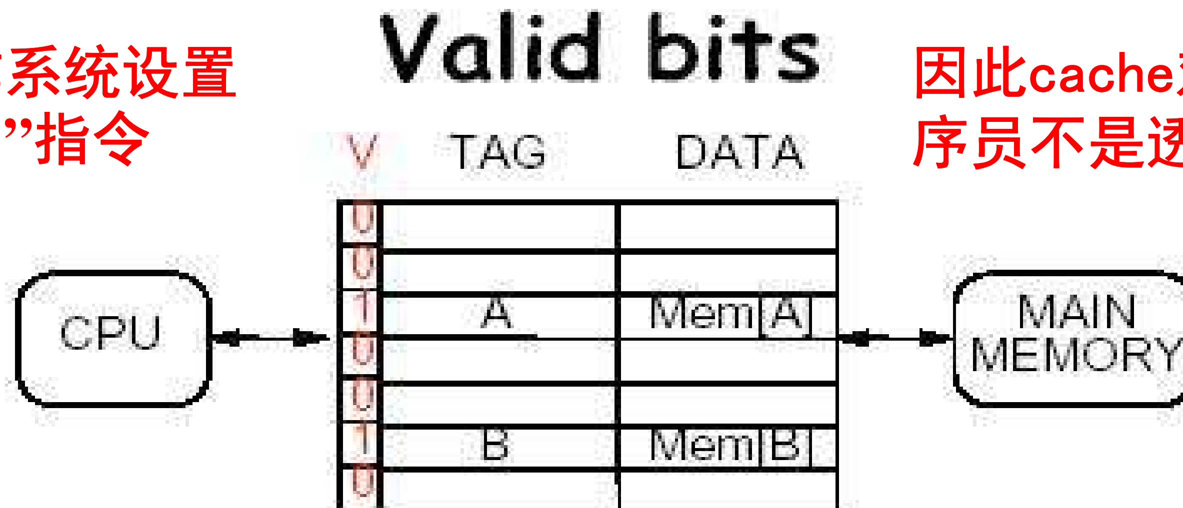
Lecture 12: Cache IV

有效位（Valid Bit）

通常为操作系统设置“cache冲刷”指令



因此cache对操作系统程序员不是透明的！

Problem:

Ignoring cache lines that don't contain anything of value... e.g., on

- start-up
- “Back door” changes to memory (eg loading program from disk)

Solution:

Extend each TAG with VALID bit.

- Valid bit must be set for cache line to HIT. 装入新块时 使V=1
- At power-up / reset : clear all valid bits 开机或复位时使V=0
- Set valid bit when cache line is first replaced. 第一次被替换时使V=1
- Cache Control Feature: *Flush* cache by clearing all valid bits, Under program/external control. 通过使V=0冲刷Cache

举例

- 假定计算机系统有一个容量为32KB的主存，且有一个4KB的4路组相联Cache，主存和Cache之间的数据交换块的大小为64B。假定Cache开始为空，处理器顺序地从存储单元0、1、…、4351中取一个Byte，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？

- 答：主存按字节（B）编址。

  主存：32KB=512块 x 64B / 块

  Cache：4KB=16组 x 4路 / 组 x 64 B / 路

  主存地址划分为：

标志位	组号	字节号
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 路	第1 路	第2 路	第3 路
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,对于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

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

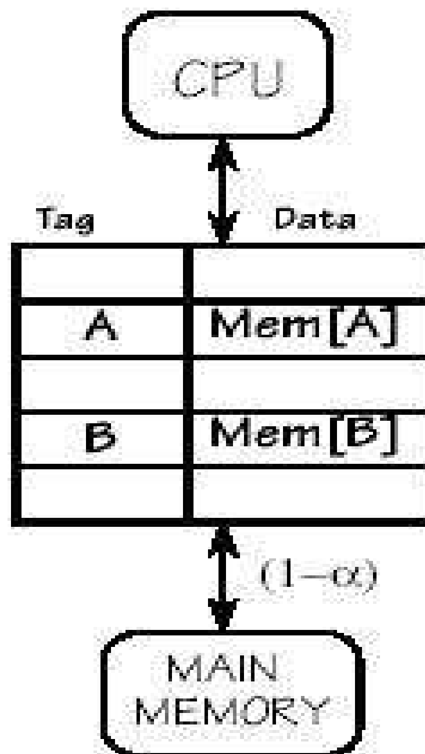
      例如：I/O设备可直接读写内存时，如果Cache中的内容被修改，则I/O设备读出的对应主存单元的内容无效；若I/O设备修改了主存单元的内容，则Cache中对应的内容无效。
    - 当多个CPU都带有各自的Cache而共享主存时

      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。
- 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

基本的Cache处理算法

问题：有什么问题？如果需频繁写，会怎么样？有没有其他方法？

'ithm



ON REFERENCE TO Mem[X]: Look for X among cache 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] (写分配方式)

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]

Write Policy: Write Through versus Write Back

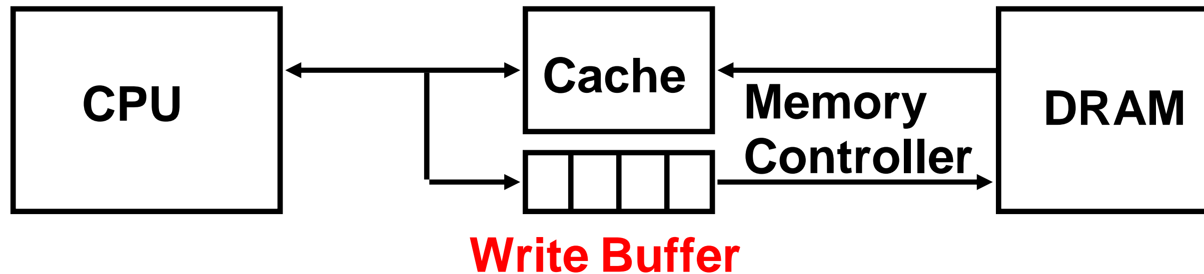
- 处理Cache读比Cache写更容易，故指令Cache比数据Cache容易设计
- 对于写命中，有两种处理方式
  - Write Through (通过式写、写直达、直写)
    - 同时写Cache和主存单元
    - What!!! How can this be? Memory is too slow(>100Cycles)?

      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)
    - 在缺失时一次写回Cache块，每块有个修改位（“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，不装入主存块到Cache

直写Cache可用非写分配或写分配
写回Cache通常用写分配　为什么？

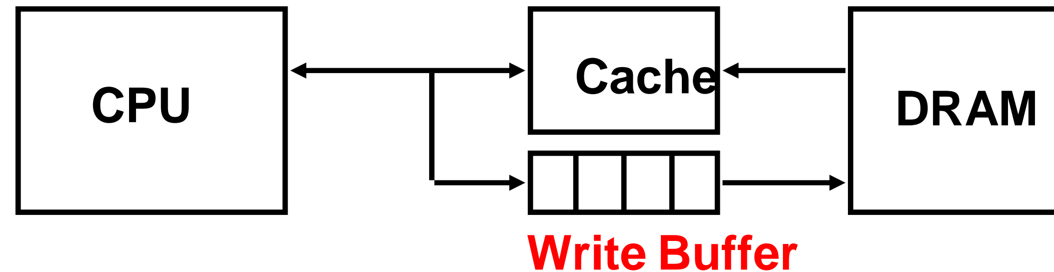
SKIP

Write Through中的Write Buffer

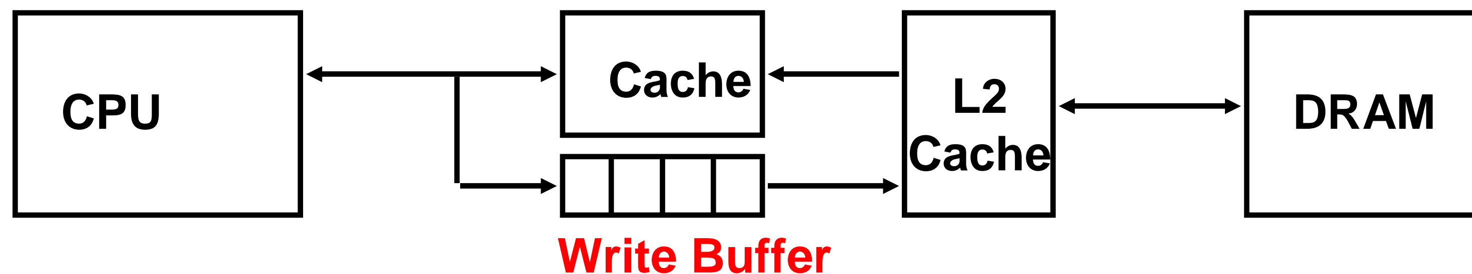


- 在 Cache 和 Memory之间加一个Write Buffer
  - CPU: 同时写数据到Cache和Write Buffer
  - Memory controller（存控）：将缓冲内容写主存
- Write buffer (写缓冲) 是一个FIFO队列
  - 一般有4项
  - 在存数频率<<DRAM写（周期）频率情况下，效果好
- 最棘手的问题
  - Store frequency > 1 / DRAM write cycle(频繁写)时，使Write buffer 饱和(溢出)，会发生阻塞

Write Buffer Saturation（写缓冲饱和）



- 发生写缓冲饱和的可能性
  - CPU时钟周期< DRAM写周期 (客观上如此)
  - 存数频率 >> 1/ DRAM写周期 (发生频繁写)
- 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]]

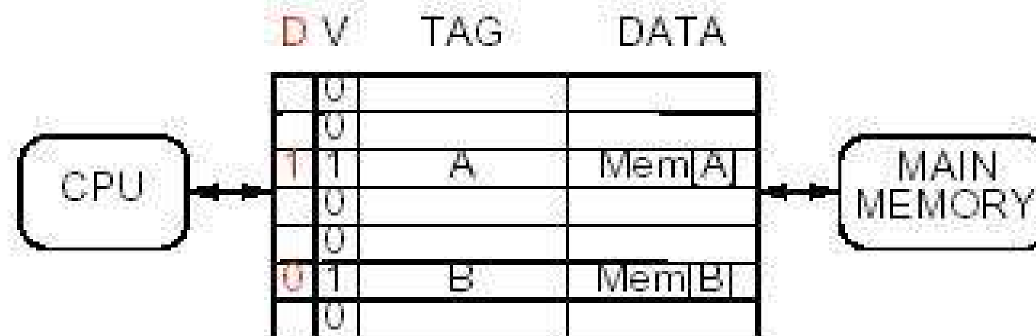
READ: Read Mem[X]
Set TAG[k] = X, DATA[k] = Mem[X]

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]