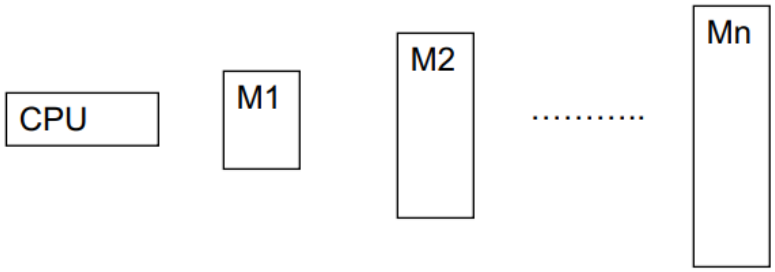
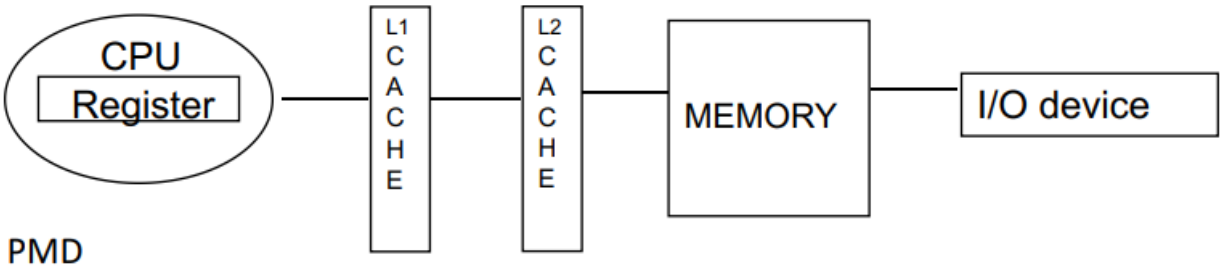
CA Cache和存储器

1 Cache基本概念

- **多级分层结构**



  - M1 速度最快，容量最小，每位价格最高
  - Mn速度最慢，容量最大，每位价格最低
- **存储系统接近M1的速度，容量和价格接近Mn**
- **并行**



500ps	2ns	10-20ns	50-100ns	25-50μs
500B	64KB	256K	256-512MB	4-8GB

存储层次结构涉及的基本概念

- **Block**
  - Block：不同层次的Block大小可能不同
  - 命中和命中率
  - 失效和失效率
- **镜像和一致性问题**
  - 高层存储器是较低层存储器的一个镜像
  - 高层存储器内容的修改必须反映到低层存储器中
    - 数据一致性问题
- **寻址：不管如何组织，我们必须知道如何访问数据**
- **要求：不同层次上块大小可以不同**
  - **以块为单位进行数据交换**
  - 不同级Cache块的大小可能不同
  - 页面（块）大小通常大于Cache块大小
  - 存在地址映射问题
  - **地址格式** Block Frame Address + Block Offset

- **存储层次的平均每位价格C**
  - C=(C1*S1+C2*S2)/(S1+S2)
- **命中(Hit): 访问的块在存储系统的较高层次上**
  - 若一组程序对存储器的访问，其中N1次在M1中找到所需数据，N2次在M2中找到数据 则
  - Hit Rate（命中率）：存储器访问在较高层命中的比例 H= N1/(N1+N2)
  - Hit Time（命中时间）:访问较高层的时间，TA1
- **失效(Miss):访问的块不在存储系统的较高层次上**
  - Miss Rate （失效率） = 1 - (Hit Rate) = 1 – H = N2/(N1+N2)
  - 当在M1中没有命中时：一般必须从M2中将所访问的数据所在块搬到M1中，然后CPU才能在M1中访问。
  - 设传送一个块的时间为TB,即不命中时的访问时间为：TA2+TB+TA1 = TA1+TM
  - TM 通常称为失效开销
- **平均访存时间:**
  - 平均访存时间 TA = HTA1+(1-H)(TA1+TM) = TA1+(1-H)TM
  - 即：**平均访存时间 = 命中时间+失效率×失效开销**

Q1：映象规则

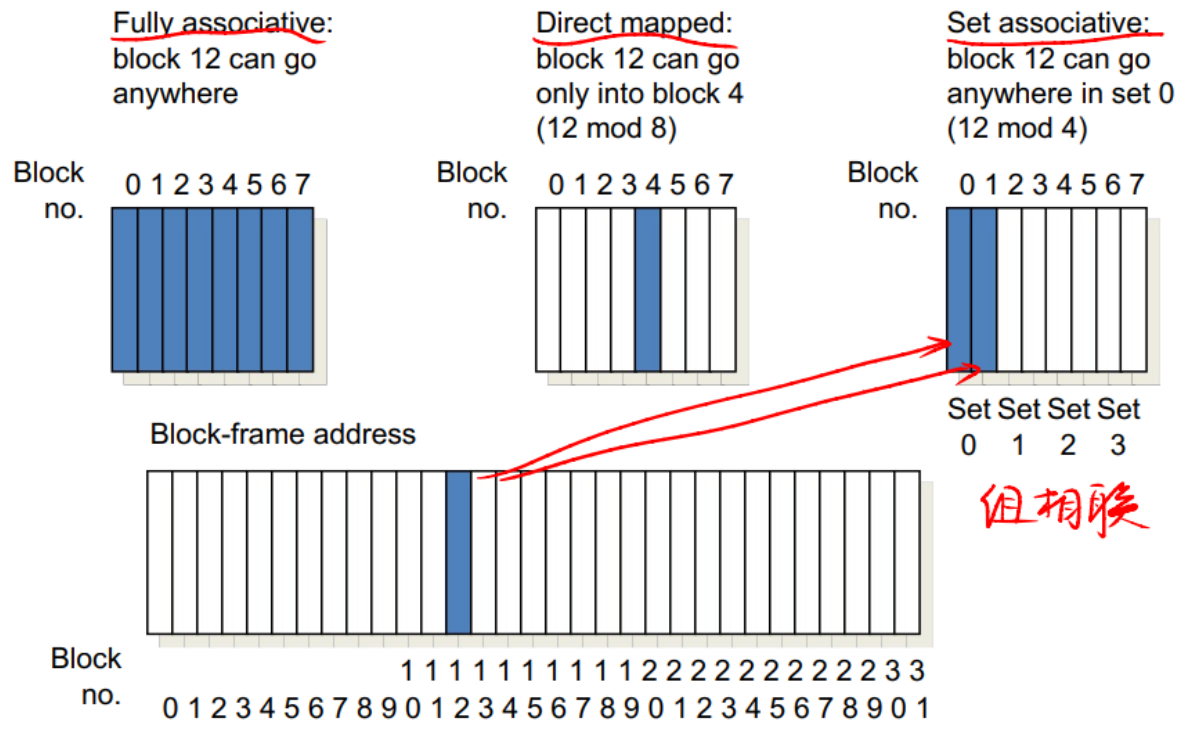
- **解决的问题：当要把一个块从主存调入Cache时的放置位置**
- **三种方式**
  - 全相联方式：即所调入的块可以放在cache中的任何位置
  - 直接映象方式：主存中每一块只能存放在cache中的唯一位置

    一般，主存块地址i 与cache中块地址j 的关系为：

    j = i mod （M），M为cache中的块数

  - 组相联映象：主存中每块可以被放置在Cache中唯一的一组中的任意一位置，组由若干块构成，若一组由n块构成，我们称N路组相联
    - 组间直接映象
    - 组内全相联
    - 若cache中有G组，则主存中的第i 块的组号K
    - K = i mod (G),

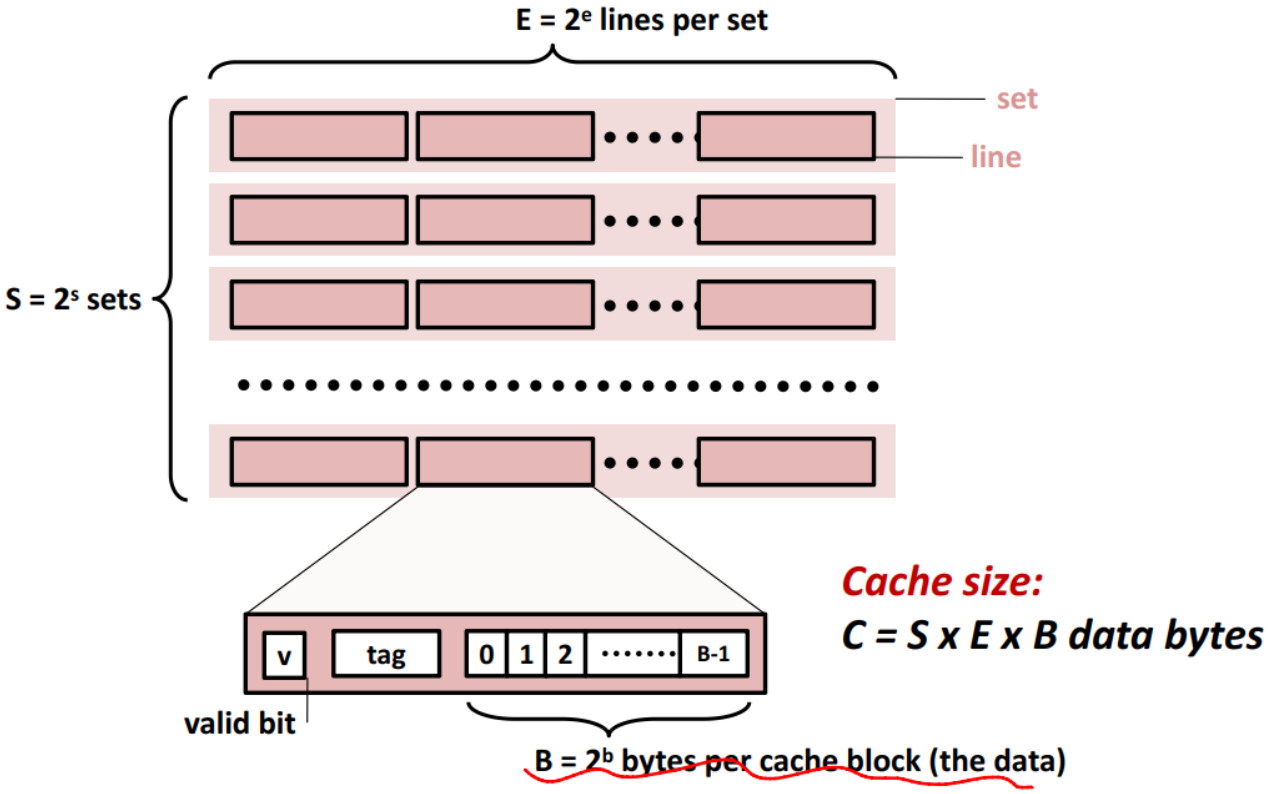
- **Block 12 placed in 8 block cache:**
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets



- **N-Way组相联：如果每组由N个块构成，cache的块数为M，则cache的组数G为M/N**
- **不同相联度下的路数和组数**

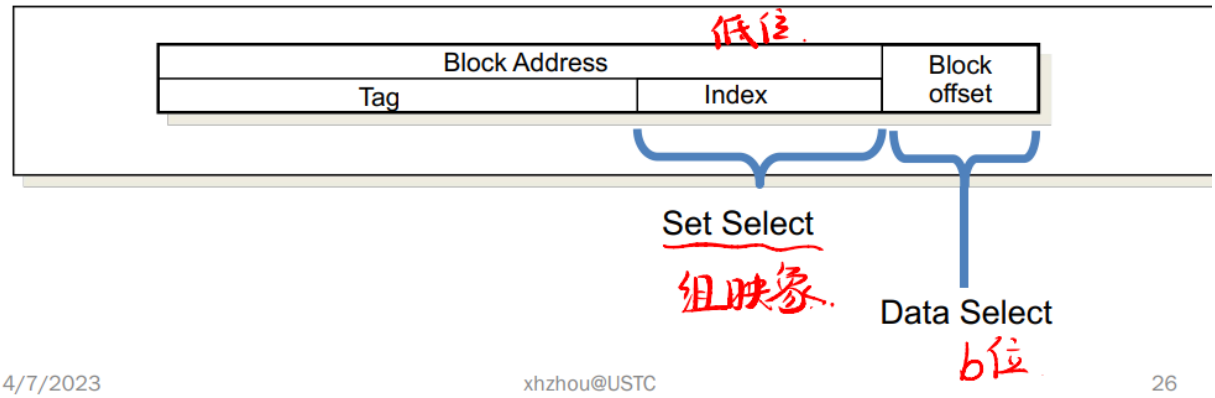
	路数	组数
全相联	M	1
直接相联	1	M
其他组相联	1 < N <M	1 < G < M

  - 相联度越高，cache空间利用率就越高，块冲突概率就越小，失效率就越低
  - N值越大，失效率就越低，但Cache的实现就越复杂，代价越大
  - 现代大多数计算机都采用直接映象，两路或四路组相联。



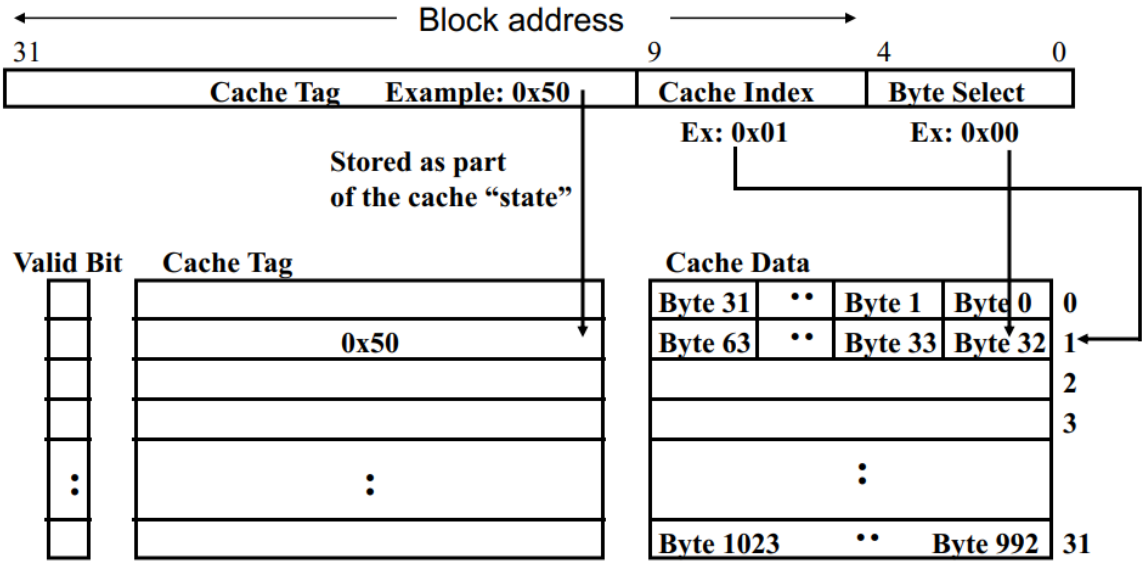
Q2(1/2): 查找方法

- **在CACHE中每一block都带有tag域（标记域），标记分为两类**
  - Address Tags：标记所访问的单元在哪一块中，这样物理地址就分为三部分： Address Tags ## Block index## block Offset
  - 全相联映象时，没有Block Index
  - 显然 Address tag越短，查找所需代价就越小
  - Status Tags：标记该块的状态，如Valid, Dirty等



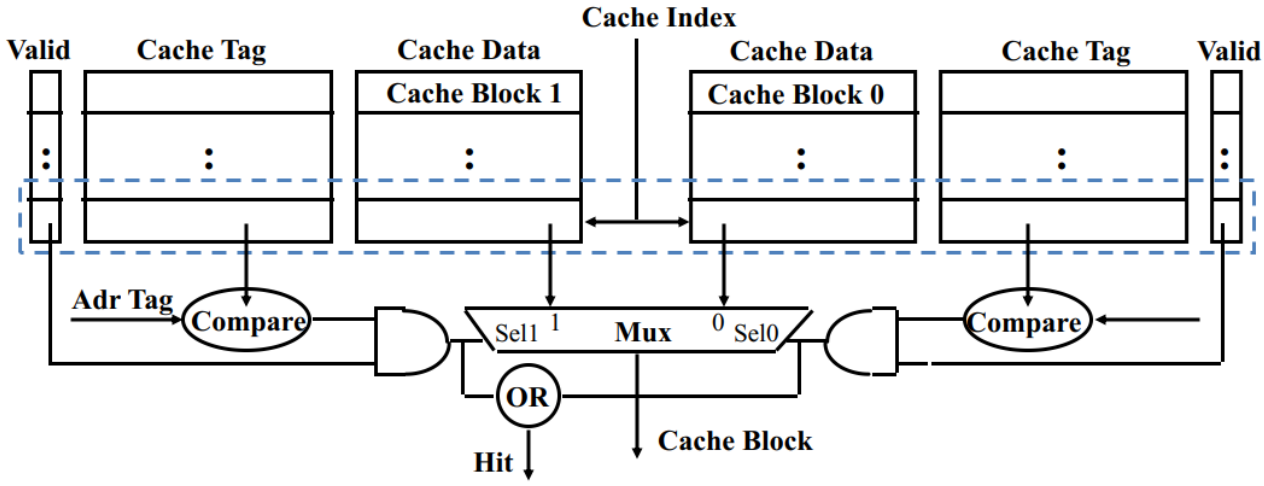
Example: 1 KB Direct Mapped Cache with 32 B Blocks

- **对于容量为 2^N 字节的cache:**
  - 最高(32-N)位部分 为 Cache Tag
  - 最低M位为字节选择位 (Block Size = 2^M)



Example: Set Associative Cache

- **N-way set associative: 每一个cache索引对应N个cache entries**
  - 这N个cache项并行操作
- **Example: Two-way set associative cache**
  - Cache index 选择cache中的一组
  - 这一组中的两块对应的Tags与输入的地址同时比较
  - 根据比较结果选择数据



Q3：替换算法

- **主存中块数一般比cache中的块多，可能出现该块所对应的一组或一个Cache块已全部被占用的情况，这时需强制腾出其中的某一块，以接纳新调入的块，替换哪一块，这是替换算法要解决的问题。**
  - 直接映象，因为只有一块，别无选择
  - 组相联和全相联有多种选择
- **替换方法**
  - 随机法（Random），随机选择一块替换
    - 优点：简单，易于实现
    - 缺点：没有考虑Cache块的使用历史，反映程序的局部性较差，失效率较高
  - FIFO - 选择最早调入的块
    - 优点：简单
    - 虽然利用了同一组中各块进入Cache的顺序，但还是反映程序局部性不够，因为最先进入的块，很可能是经常使用的块
  - 最近最少使用法（LRU) (Least Recently Used)
    - 优点：较好的利用了程序的局部性，失效率较低
    - 缺点：比较复杂，硬件实现较困难

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

- **观察结果（失效率）**
  - 相联度高，失效率较低。
  - Cache容量较大，失效率较低。
  - LRU 在Cache容量较小时，失效率较低
  - 随着Cache容量的加大，Random的失效率在降低

- **写直达法（Write through）**
  - 优点：易于实现，容易保持不同层次间的一致性
  - 缺点：速度较慢
- **写回法**
  - 优点：速度快，减少访存次数
  - 缺点：一致性问题
- **当发生写失效时的两种策略**
  - 按写分配法(Write allocate)：写失效时，先把所写单元所在块调入Cache，然后再进行写入，也称写时取（Fetch on Write)方法
  - 不按写分配法（no-write allocate): 写失效时，直接写入下一级存储器，而不将相应块调入Cache，也称绕写法（Write around）
  - 原则上以上两种方法都可以应用于写直达法和写回法，一般情况下
    - Write Back 用Write allocate
    - Write through 用no-write allocate

保持 Cache 和 Memory 一致性.

4/7/2023 xhzhou@USTC 40

1.1 写直达与写回策略

写直达（Write-Through）：

在这个策略里，每一次数据都要写入到主内存里面。在写直达的策略里面，写入前，我们会先去判断数据是否已经在 Cache 里面了。如果数据已经在 Cache 里面了，我们先把数据写入更新到 Cache 里面，再写入到主内存里面；如果数据不在 Cache 里，我们就只更新主内存。写直达的这个策略很直观，但是问题也很明显，那就是这个策略很慢。无论数据是不是在 Cache 里面，我们都需要始终把数据同步到主内存里面。

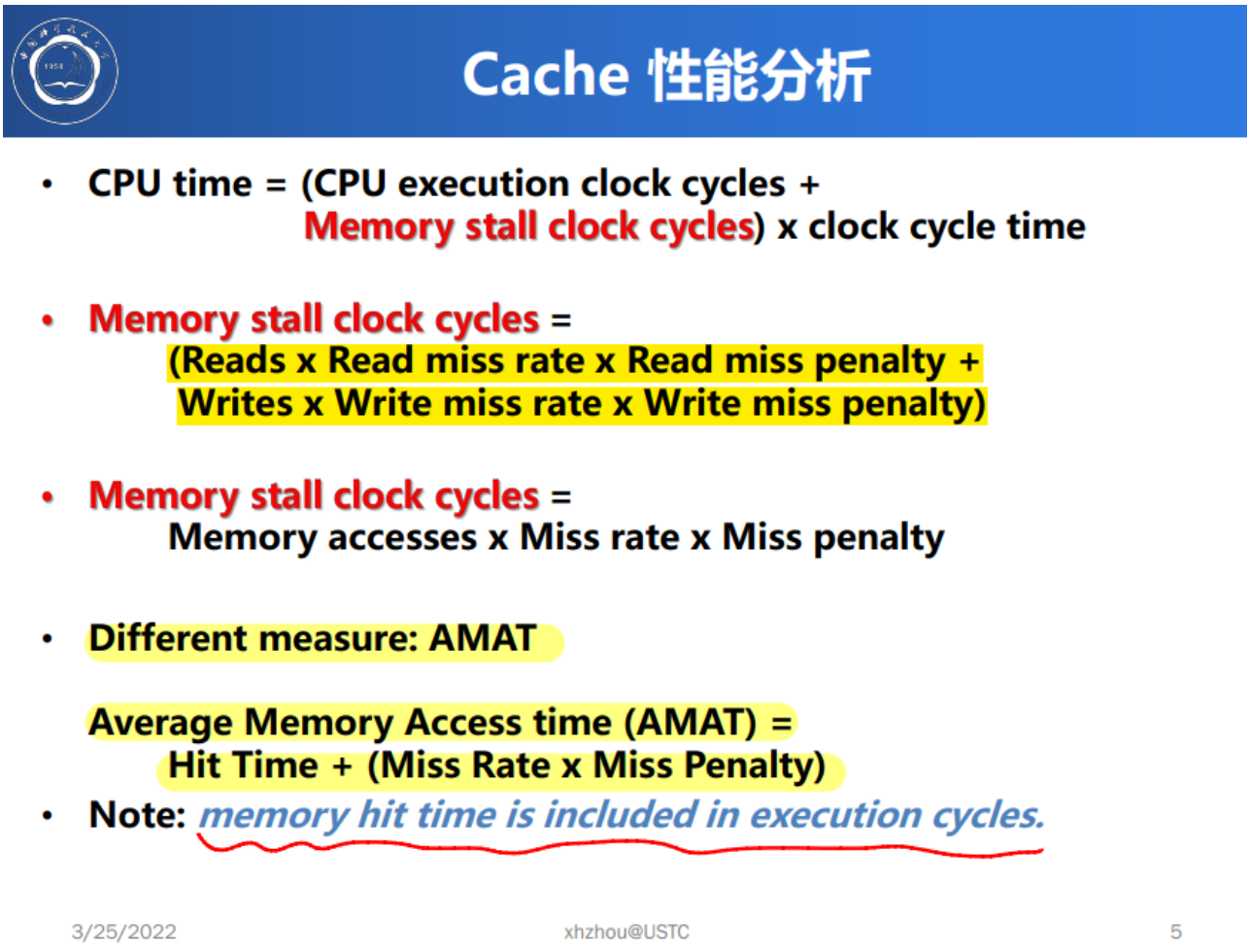
写回策略（Write back）

如果发现我们要写入的数据，**就在 CPU Cache 里面，那么我们就只是更新 CPU Cache 里面的数据**。同时，我们会标记 CPU Cache 里的这个 Block 是脏（Dirty）的。所谓**脏的，就是指这个时候，我们的 CPU Cache 里面的这个 Block 的数据，和主内存是不一致的**。

如果写入的数据**所对应的 Cache Block 里，放的是别的内存地址的数据**，那么我们就要看一看，那个 Cache Block **里面的数据有没有被标记成脏的**。如果是脏的话，我们**要先把这个 Cache Block 里面的数据，写入到主内存里面。然后，再把当前要写入的数据，写入到 Cache 里，同时把 Cache Block 标记成脏的**。

如果 Block 里面的数据没有被标记成脏的，那么我们直接把数据写入到 Cache 里面，然后再把 Cache Block 标记成脏的就好了。

2 Cache性能计算



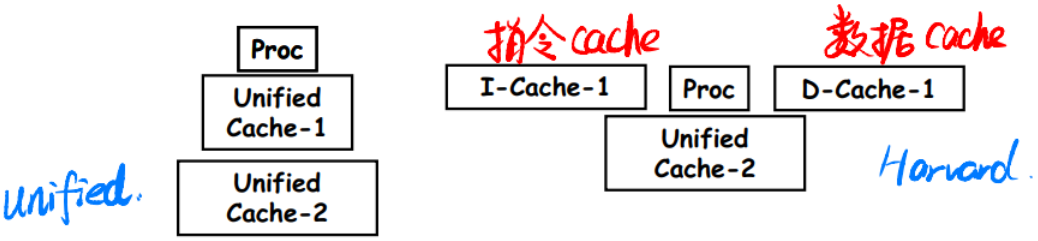
性能分析举例

- Suppose a processor executes at
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- Miss Behavior:
  - 10% of memory operations get 50 cycle miss penalty
  - 1% of instructions get same miss penalty 每个指令取指有1%失效
- CPI = ideal CPI + average stalls per instruction

  = 1.1(cycles/ins) +
  [0.30 (DataMops/ins)
  x 0.10 (miss/DataMop) x 50 (cycle/miss)] +
  [1 (InstMop/ins)
  x 0.01 (miss/InstMop) x 50 (cycle/miss)]
  = (1.1 + 1.5 + .5) cycle/ins = 3.1
- 65% (2/3.1) of the time the proc is stalled waiting for memory!
- AMAT=(1/1.3)x[1+0.01x50]+(0.3/1.3)x[1+0.1x50]=2.54

Example: Harvard Architecture

- Unified vs Separate I&D (Harvard)



- Statistics (given in H&P):
  - 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
  - 32KB unified: Aggregate miss rate=1.99%
- Which is better (ignore L2 cache)?
  - Assume 33% data ops ⇒ 75% accesses from instructions (1.0/1.33)
  - hit time=1, miss time=50
  - Note that data hit has 1 stall for unified cache (only one port)
- AMATHarvard=75%x(1+0.64%x50)+25%x(1+6.47%x50) = 2.05
- AMATUnified =75%x(1+1.99%x50)+25%x(1+1+1.99%x50)= 2.24

以顺序执行的计算机 UltraSPARC III为例。假设Cache失效开销为100 clock cycles，所有指令忽略存储器停顿需要1个cycle, Cache失效可以用两种方式给出

（1）假设平均失效率为2%，平均每条指令访存1.5次

（2）假设每1000条指令cache失效次数为30次

3% 失效一次

分别基于上述两种条件计算处理器的性能

- 结论：
- CPUtime = IC * (1 + 2%*1.5*100) * T = IC * 4 * T

(1) 理想CPI越低，固定周期数的Cache失效开销的相对影响就越大

(2) 在计算CPI时，失效开销的单位是时钟周期数。因此，即使两台计算机的存储层次完全相同，时钟频率较高的CPU的失效开销会较大，其CPI中存储器停顿部分也就较大。

Cache失效开销与存储器性能、CPU频率有关

Cache对于低CPI，高时钟频率的CPU来说更加重要

考虑不同组织结构的Cache对性能的影响:

B-19例题：直接映像Cache 和两路组相联Cache，试问他们对CPU性能的影响？先求平均访存时间，然后再计算CPU性能。分析时请用以下假设：

（1）理想Cache(命中率为100%）情况下CPI 为1.0，时钟周期为0.35ns，平均每条指令访存1.4次

（2）两种Cache容量均为128KB，块大小都是64B

（3）采用组相联时，由于多路选择器的存在，时钟周期增加到原来的1.35倍

（4）两种结构的失效开销都是70ns

（5）命中时间为1个cycle，128KB直接映像Cache的失效率为2.1%, 相同容量的两路组相联Cache 的失效率为1.9%

直接映射Cache的CPU时间=指令数×(1.0×0.35+1.4×2.1%×65ns)
两路组相联Cache的CPU时间=指令数×(1.0×0.35×1.35+1.4×1.9%×65ns)

3 基本优化

Cache失效的原因 可分为三类 3C （+ Coherence）

- **强制性失效 (Compulsory)**
  - 第一次访问某一块，只能从下一级Load，也称为冷启动或首次访问失效
- **容量失效（Capacity)**
  - 如果程序执行时，所需块由于容量不足，不能全部调入Cache，则当某些块被替换后，若又重新被访问，就会发生失效。
  - 可能会发生“抖动”现象
- **冲突失效（Conflict (collision))**
  - 组相联和直接相联的副作用
  - 若太多的块映像到同一组（块）中，则会出现该组中某个块被别的块替换（即使别的组或块有空闲位置），然后又被重新访问的情况，这就属于冲突失效

从统计规律中得到的一些结果

- **失效与Cache结构的关系**
  - 相联度越高，冲突失效就越小
  - 强制性失效和容量失效不受相联度的影响
  - 强制性失效不受Cache容量的影响
  - 容量失效随着容量的增加而减少
- **符合2:1Cache经验规则**
  - 即大小为N的直接映象Cache的失效率约等于大小为N/2的两路组相联的Cache失效率。

减少3C的方法

从统计规律可知：

- **增大Cache容量**
  - 对冲突和容量失效的减少有利
- **增大块**
  - 减缓强制性失效 换入换出传送data开销大
  - 可能会增加冲突失效（为什么？）
- **通过预取可帮助减少强制性失效**
  - 必须小心不要把你需要的东西换出去
  - 需要预测比较准确（对数据较困难，对指令相对容易）

ML.

块大小、容量的权衡

- **从统计数据可得到如下结论**
  - 对于给定Cache容量，块大小增加时，失效率开始是下降，但后来反而上升
  - Cache容量越大，使失效率达到最低的块大小就越大
- **分析**
  - 块大小增加，可使强制性失效减少（空间局部性原理）
  - 块大小增加，可使冲突失效增加（Cache中块数量减少）
  - 失效开销增大（上下层间移动，数据传输时间变大）
- **设计原则：选取块大小时，不能仅看失效率**
  - 原因：平均访存时间 = 命中时间 + 失效率×失效开销

Victim Cache(1/2)

- **在Cache和Memory之间增加一个小的全相联Cache**

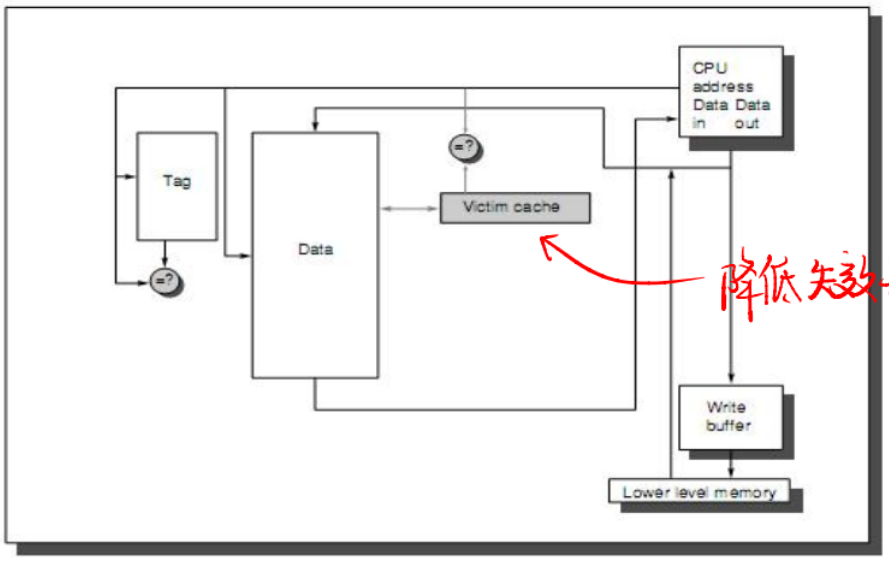


FIGURE 5.13 Placement of victim cache in the memory hierarchy. Although it reduces miss penalty, the victim cache is aimed at reducing the damage done by conflict misses, described in the next section. Jouppi [1990] found the four-entry victim cache could reduce the miss penalty for 20% to 95% of conflict misses.

- **基本思想**
  - 通常Cache为直接映象时冲突失效率较大
  - Victim cache采用全相联 - 失效率较低
  - Victim cache存放由于（冲突）失效而被丢弃的那些块
  - 失效时，首先检查Victim cache是否有该块，如果有就将该块与Cache中相应块互换。

多级包容性(multilevel inclusive)

- **L1 cache 的块总是存在于L2 cache中**
  - 优点：数据一致性好（镜像）
  - 缺点：浪费了L2 cache 空间，L2 还应当有存放其他块的空间

- **L1中miss, 但在L2中命中，则从L2拷贝相应的块到L1**
- **在L1和L2中均miss, 则从更低级拷贝相应的块到L1和L2**

- **对L1写操作导致将数据同时写到L1和L2**
- **Write-through 策略用于L1到L2**
- **Write-back 策略可用于L2 到更低级存储器，以降低存储总线数据传输压力**

- **L2的替换动作（或无效)对L1可见**
  - 即L2的一块被替换出去，那么其在L1中对应的块也要被替换出去。
- **L1和L2的块大小可以相同也可以不同**
  - Pentium 4 had 64-byte blocks in L1 but 128-byte blocks in L2
  - Core i7 uses 64-byte blocks at all cache levels (simpler)

多级不包容（Multilevel Exclusive）

- **L1 cache 中的块不会在L2 cache中，以避免浪费空间**
- **在L1中miss, 但在L2中命中，将导致Cache间块的互换**
- **在L1和L2均miss, 将仅仅从更低层拷贝相应的块到L1**
- **L1的被替换的块移至L2**
  - L2 存储L1抛弃的块，以防后续L1还需要使用
- **L1到L2的写策略为 Write-Back**
- **L2到更低级cache的写策略为 Write-Back**
- **AMD Athlon： 64KB L1、 256KB L2**

多级cache的性能分析

- **局部失效率：该级Cache失效次数 / 到达该级Cache的访存次数**
  - Miss rateL1 for L1 cache
  - Miss rateL2 for L2 cache

$\frac{N_3}{N_2+N_3}$

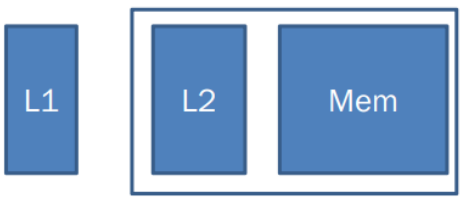
- **全局失效率：该级Cache失效次数/ CPU发出的访存总次数**
  - Miss rateL1 for L1 cache
  - Miss rateL1 × Miss rateL2 for L2 cache
  - 全局失效率是度量L2 cache性能的更好方法

$\frac{N_2+N_3}{N}$

全局：N3/N
局部：N3/N2+N3

- **性能参数**
  - AMAT = Hit TimeL1+Miss rateL1×Miss penaltyL1
  - Miss penaltyL1 = HitTimeL2 + Miss rateL2×Miss penaltyL2
  - AMAT = Hit TimeL1+Miss rateL1× (Hit TimeL2+Miss rateL2×Miss penaltyL2)

$N = N_1 + N_2 + N_3$
L1 L2 MEM



局部失效率：Miss rate L1，Miss rate L2

↙

到达该Cache访存次数

L1 cache：局部=全局

L2 cache
- miss rate 局局
- 全局 = miss rate L1 × L2

理想CPI=1：流水指令cycle数相同

CPI<1 有吗？ ex：branch可以只用3cycle

st 可以无MB

L1 cache 失效率

- **对于I-Cache和D-Cache分开的L1 Cache**

  Miss RateL1 = %inst × Miss RateI-Cache + %data × Miss RateD-Cache

  %inst = Percent of Instruction Accesses = 1 / (1 + %LS)

  %data = Percent of Data Accesses = %LS / (1 + %LS)

  %LS = Frequency of Load and Store instructions

- **每条指令的L1 失效次数:**

  Misses per InstructionL1 = Miss RateL1 × (1 + %LS)

  Misses per InstructionL1 = Miss RateI-Cache + %LS × Miss RateD-Cache

- **Problem: 计算AMAT**
  - I-Cache 失效率 = 1%， D-Cache失效率 = 10%
  - L2 Cache失效率 = 40%
  - L1 命中时间 = 1 cycle (I-Cache 和D-Cache相同)
  - L2 命中时间 = 8 cycles， L2 失效开销 = 100 cycles
  - Load + Store 指令频度 = 25%
- **Solution:**
  - 平均每条指令访存次数 = 1+25% = 1.25
  - 平均每条指令的失效次数 = 1%+25%× 10% = 0.035
  - L1的失效率= 0.035/1.25=0.028
  - L1的失效开销 = 8 + 0.4×100 = 48 cycles
  - AMAT = 1+0.028×48 = 2.344

- **Memory Stall Cycles per Instruction**

  = Memory Access per Instruction × Miss RateL1 × Miss PenaltyL1

  = (1 + %LS) × Miss RateL1 × Miss PenaltyL1

  = (1 + %LS) × Miss RateL1 × (Hit TimeL2 + Miss RateL2 × Miss PenaltyL2)

两级Cache的性能

- **Problem: 程序运行产生1000个存储器访问**
  - I-Cache misses = 5, D-Cache misses = 35, L2 Cache misses = 8
  - L1 Hit = 1 cycle, L2 Hit = 8 cycles, L2 Miss penalty = 80 cycles
  - Load + Store frequency = 25%, CPIexecution = 1.1 (perfect cache)
  - 计算memory stall cycles per instruction 和有效的CPI
  - 如果没有L2 cache, 有效的CPI是多少?
- **Solution:**
  - L1 Miss Rate = (5 + 35) / 1000 = 0.04 (or 4% per access)
  - L1 misses per Instruction =0.04 × (1 + 0.25) = 0.05
  - L2 misses per Instruction =(8 / 1000) × 1.25 = 0.01
  - Memory stall cycles per Instruction =0.05 × 8 + 0.01 × 80 = 1.2
  - CPIL1+L2 = 1.1 + 1.2 = 2.3, CPI/CPIexecution = 2.3/1.1 = 2.1x slower
  - CPIL1only =1.1 + 0.05 × 80 = 5.1 (worse)

两级Cache的一些研究结论

- **在L2比L1大得多得情况下，两级Cache全局失效率 和 容量与第二级Cache相同的单级Cache的失效率接近**
- **局部失效率不是衡量第二级Cache的好指标**
  - 它是第一级Cache失效率的函数
  - 不能全面反映两级Cache体系的性能
- **第二级Cache设计需考虑的问题**
  - 容量：一般较大，可能无容量失效，有强制性失效和冲突失效
    - 相联度对第二级Cache的作用
    - Cache可以较大，以减少失效次数
  - 多级包容性问题：第一级Cache中的数据是否总是同时存在于第二级Cache中。
    - 如果L1和L2的块大小不同，增加了多级包容性实现的复杂性

多级Cache举例

Given the data below, what is the impact of second-level cache associativity on its miss penalty?

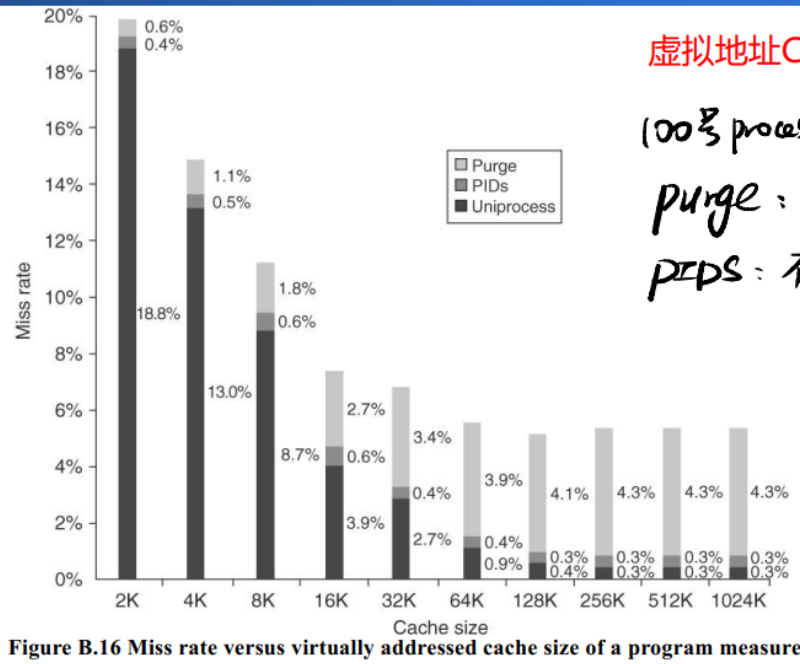
- Hit timeL2 for direct mapped = 10 clock cycles
- Two-way set associativity increases hit time by 0.1 clock cycles to 10.1clock cycles
- Local miss rateL2 for direct mapped = 25%
- Local miss rateL2 for two-way set associative = 20%
- Miss penaltyL2 = 100 clock cycles

- **结论：提高相联度，可减少第一级Cache的失效开销**
- **第二级Cache特点：容量大，高相联度，块较大，重点减少失效次数。**

让读失效优先于写

- **由于读操作为大概率事件，需要读失效优先，以提高性能**
- **Write-Through Cache ->Write Buffer (写缓冲)，特别对写直达法更有效**
  - Write Buffer: CPU不必等待写操作完成，即将要写的数据和地址送到Write Buffer后，CPU继续作其他操作。
  - 写缓冲导致对存储器访问的复杂化
    - **在读失效时写缓冲中可能保存有所读单元的最新值，还没有写回**
    - 例如，直接映射、写直达、512和1024映射到同一块。则
    - SW R3, 512(R0) //命中，直接修改Cache块，并将新的值 写入write Buffer
    - LW R1, 1024(R0) //失效，将512(R0)对应块换成1024(R0)单元所在块
    - LW R2, 512(R0) //失效，从主存中调入512(R0)单元所在块
  - 解决办法
    - 推迟对读失效的处理，直到写缓冲器清空，导致新的问题——读失效开销增大。
    - 在读失效时，检查写缓冲的内容，如果没有冲突，而且存储器可访问，就可以继续处理读失效
  - 写回法时，也可以利用写缓冲器来提高性能
    - 把脏块放入缓冲区，然后读存储器，最后写存储器
- **Write-Back Cache ->Victim Buffer**
  - 被替换的脏块放到了victim buffer
  - 问题：在脏块被写回前，需要处理读失效。 victim buffer 可能含有该读失效要读取的块
    - Solution: 查找victim buffer，如果命中直接将该块调入Cache

缩短命中时间



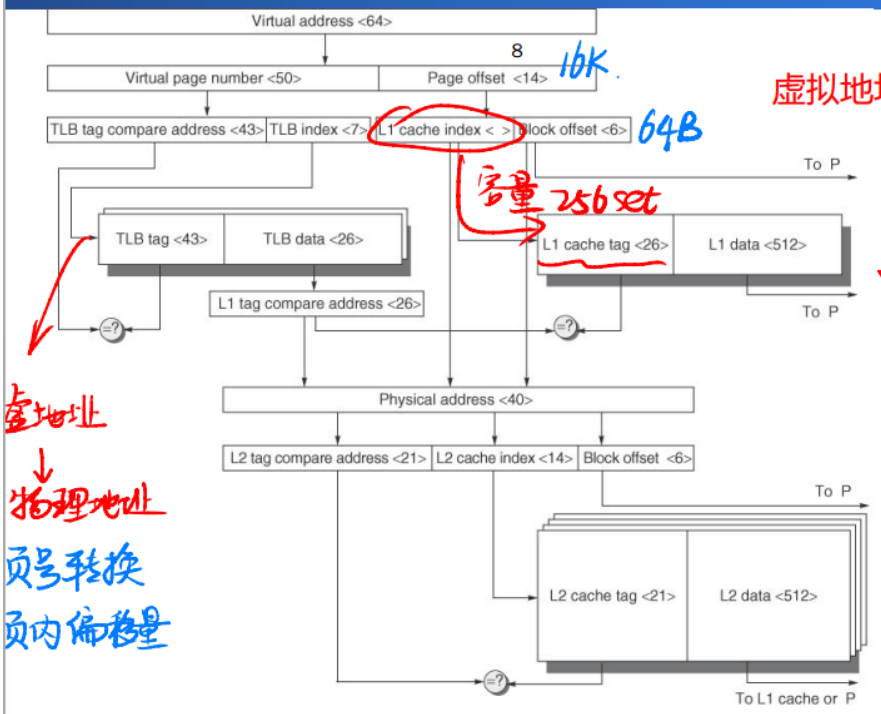
虚拟地址Cache VS. 物理地址Cache

100号process → 101号.

purge：清除旧process，switch

pids：在旧任务上更新.

Figure B.16 Miss rate versus virtually addressed cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PID), and with process switches but without PIDs (purge). PIDs increase the uniprocess absolute miss rate by 0.3% to 0.6% and save 0.6% to 4.3% over purging. Agarwal [1987] collected these statistics for the Ultrix operating system running on a VAX, assuming direct-mapped caches with a block size of 16 bytes. Note that the miss rate goes up from 128K to 256K. Such nonintuitive behavior can occur in caches because changing size changes the mapping of memory blocks onto cache blocks, which can change the conflict miss rate.



虚拟地址转换与Cache定位 并行

虚地址 → 物理地址

页号转换

页内偏移量

Cache定位

把直接映像方式 → 组相联.

容量↑ 时间↑

trade-off

Figure B.17 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 16 KB. The TLB is two-way set associative with 256 entries. The L1 cache is a direct-mapped 16 KB, and the L2 cache is a four-way set associative with a total of 4 MB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 40 bits.

4 高级优化

高级Cache优化方法

- **缩短命中时间**
  - 1、小而简单的第一级Cache
  - 2、路预测方法
- **增加Cache带宽**
  - 3、Cache访问流水化
  - 4、无阻塞Cache
  - 5、多体Cache
- **减小失效开销**
  - 6、关键字优先和提前重启
  - 7、合并写
- **降低失效率**
  - 8、编译优化
- **通过并行降低失效开销或失效率**
  - 9、硬件预取
  - 10、编译器控制的预取

1、Small and simple first level caches

- **Small and simple first level caches**
  - 容量小，一般命中时间短，有可能做在片内
  - 另一方案，保持Tag在片内，块数据在片外，如DEC Alpha
  - 第一级Cache应选择容量小且结构简单的设计方案
- **Critical timing path:**
  - 1) 定位组, 确定tag的位置
  - 2) 比较tags,
  - 3) 选择正确的块
- **Direct-mapped caches can overlap tag compare and transmission of data**
  - 数据传输和tag 比较并行
- **Lower associativity reduces power because fewer cache lines are accessed**
  - 简单的Cache结构、可有效减少tag比较的次数，进而降低功耗

2、Way Prediction

- **为改进命中时间，预测被选中的路(way)**
  - 预测错误会导致更长的命中时间
  - 预测的准确性
    - 90%+ for two-way
    - 80%+ for four-way
    - I-cache比D-cache具有更好的准确性
  - 90年代中期第一次用于MIPS R10000
  - 用于ARM Cortex-A8
- **Way Prediction的扩展方法也可降低功耗：将直接映像方式查找与Way Prediction选择结合**
  - 也称“路选择”“Way selection”
  - 可有效降低功耗，但一旦预测错误会有更长的命中时间

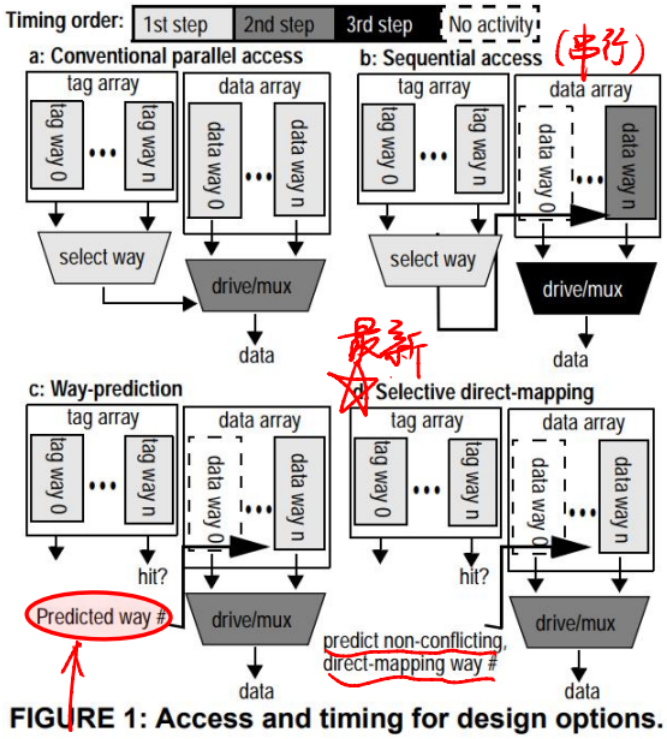


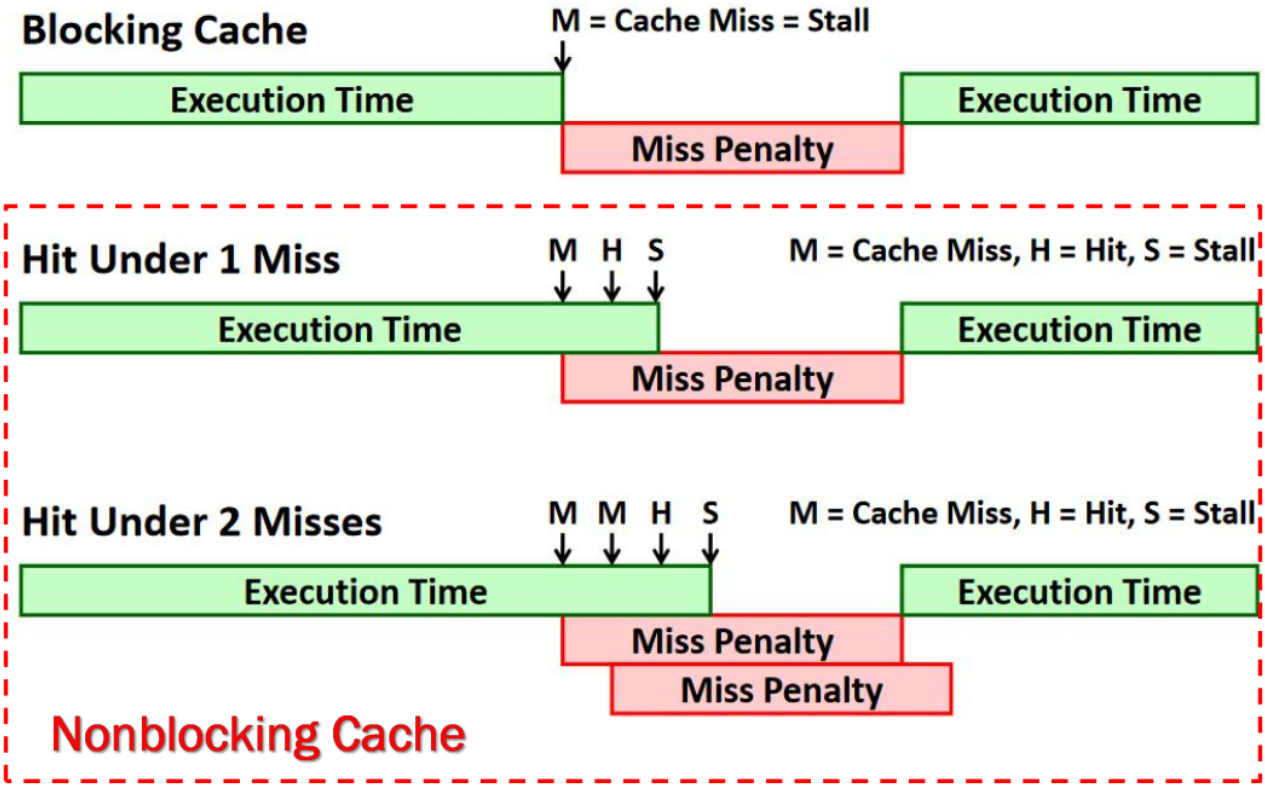
FIGURE 1: Access and timing for design options.

3、Pipelining Cache

- **实现Cache访问的流水化**
  - 提高Cache的带宽，有利于采用高相联度的缓存
  - L1 cache的访问由多个时钟周期构成
    - Pentium: 1 cycle
    - Pentium Pro – Pentium III: 2 cycles
    - Pentium 4 – Core i7: 4 cycles
    - IBM Power7: 3 cycles
- **缺点：增加流水线的段数**
  - 增加了分支预测错误造成的额外开销
  - 增加了Load指令与要使用其结果的指令间的latency
  - 增加了I-Cache和D-Cache的延时

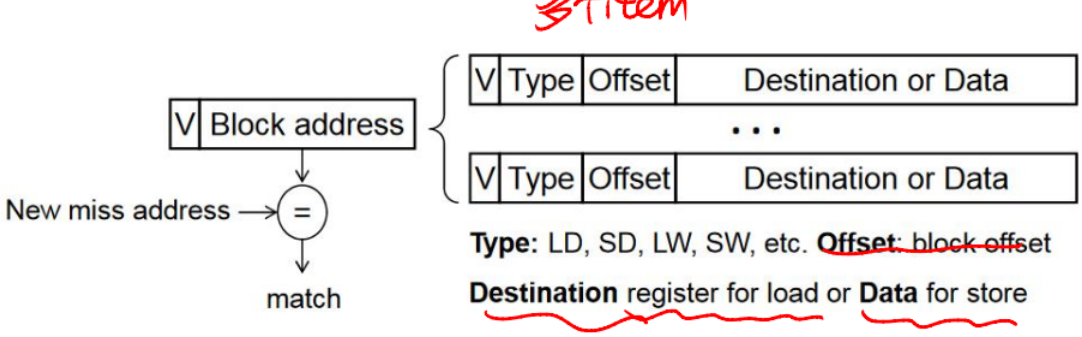
4、Nonblocking Caches

- **允许在Cache失效下继续命中** 非阻塞
  - 在Cache失效时，CPU无需stall
  - 主要用于乱序执行和多线程处理器
- **Hit under a Miss**
  - 减少有效的失效开销
  - 增加Cache的带宽
- **Hit under Multiple Misses**
  - 针对多个未解决的Cache失效
  - 可能会更多地减少有效的失效开销
  - 增加了Cache控制器的复杂性
  - 存储系统可以支持多个失效时的存储服务



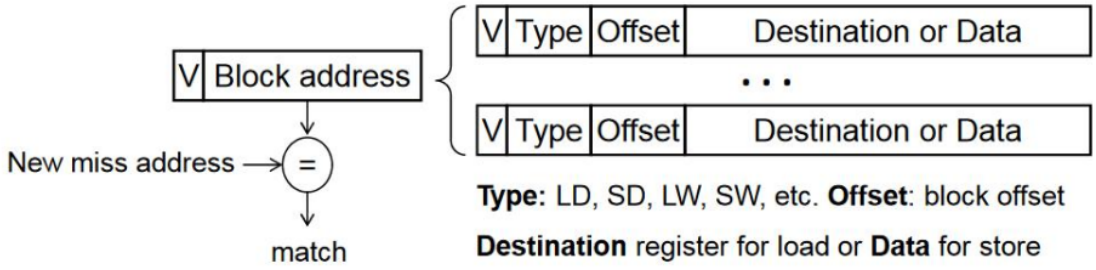
Miss Status Holding Register (MSHR)

- **MSHR 包含正在等待处理的失效**
  - 相同的块可以包含多个未解决的Load/Store 失效
  - 可以有多个未解决的块地址
- **失效可以分为**
  - Primary: 第一次发起存取请求时的失效块
  - Secondary: 在后续过程中的失效
  - Structural Stall miss： MSHR 硬件资源耗尽



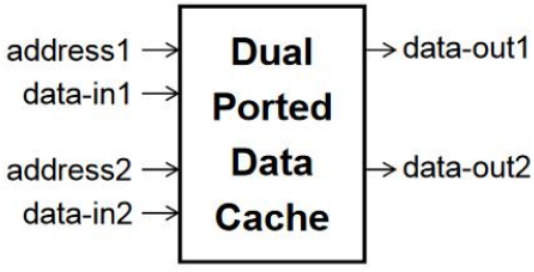
NonBlocking Cache Operation

- **当Cache失效时，检查MSHR(Miss-status Handling Registers)是否有匹配的块地址**
  - 如果有，为该地址分配新的load/store 表项 → 第2次miss
  - 如果没有，分配新的MSHR 和 load/store表项 → 第1次miss
  - 如果所有MSHR资源都分配完，则Stall（结构相关）
- **当从底层传输Cache块时**
  - 处理该块中的Load和Store指令引起的失效
  - Load：根据block offset从该块中装载数据到寄存器
  - Store：根据block offset将数据写入该块指定位置
  - 完成该块所有的失效的Load/Store后，释放MSHR中的对应表项



5、Multibanked Caches（1/2）

- **Multi-Ported Cache**
  - True Multi-ported Cache Design
    - 所有的控制和数据通路在Cache中是多份的
      - Address Decoder, way multiplexor, tag comparator, aligners
      - Tag Array, Data Array
    - 显著地增加了Cache的面积和访问时间
    - 没有商业处理器采用这种设计
  - Multi-Banked Cache
    - 将cache组织成多个banks
    - 每个bank是一个端口
    - 可以并行访问不同的Bank



- **根据块号进行顺序多体交叉编址**

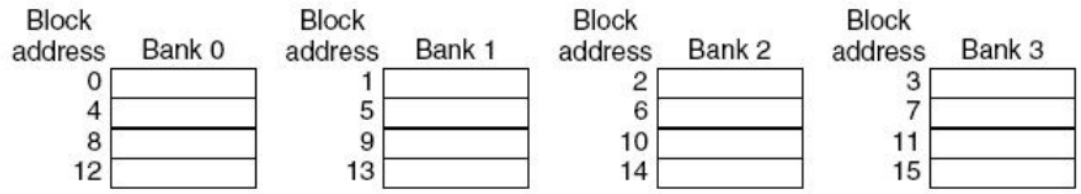


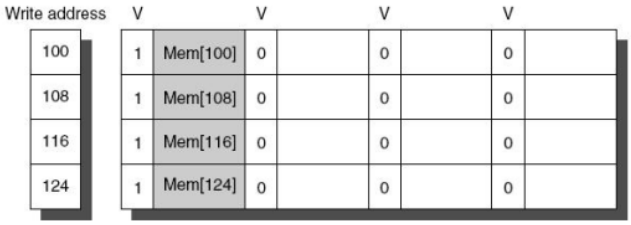
Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

6、Critical Word First, Early Restart

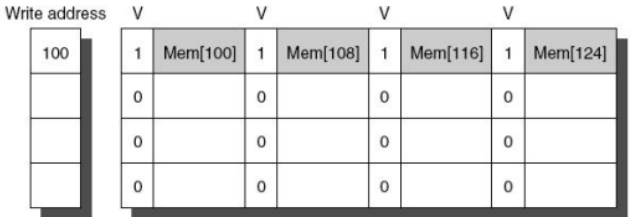
- **关键字优先**
  - 首先请求CPU所需要的字
  - 请求字一到达就发给CPU，使其继续执行，同时从存储器中调入其他字
- **提前重启**
  - 请求字的顺序不变
  - 请求字一到达就发给CPU，使其继续执行，同时从存储器中调入其他字
- **通常在块比较大时，这些技术才有效**

7、Merging Write Buffer

- **在向写缓冲器写入地址和数据时，如果写缓冲器中存在被修改过的块，就检查其地址，看看本次写入数据的地址是否与写缓冲器内的某个有效块地址匹配，如果匹配，就把新数据与该块合并，称为“合并写”**
- **可以缓解由于写缓冲满而造成的CPU停顿**
- **不适用于I/O地址空间？？** → I/O需要读状态，等等



No write buffering



Write buffering (merging)

8、编译器优化

- **无需对硬件做任何改动，通过软件优化降低失效率**
- **研究从两方面展开：减少指令失效 和 减少数据失效**
- **减少指令失效，重新组织程序（指令调度）而不影响程序的正确性**
  - 研究结果：通过使用profiling信息来判断指令组间可能发生的冲突，并将指令重新排序以减少失效。
  - 研究表明：
    - 容量为2KB, 块大小为 4Bytes的直接映象Icache，通过使用指令调度可以使失效率降低50%。容量增大到 8KB, 失效率可降低75%
    - 在有些情况下，当能够使某些指令不进入ICache时，可以得到最佳性能。即使不这样做，优化后（指令调度）的程序在直接映象Cache中的失效率也低于未优化程序在同样大小的8路组相联Cache中的失效率。
- **减少数据失效，主要通过优化来改善数据的空间局部性和时间局部性，基本方法为：**
  - 数据合并
  - 内外循环交换，循环融合
  - 分块

编译器优化方法举例之一：循环合并

```
// Original Code
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
for (i = 0; i < N; i++)
    d[i] = a[i] + b[i] * c[i];
Blocks are replaced in first loop then accessed in second
```

数据传送，计算，写回 ⇒ pipline 3
数据复用

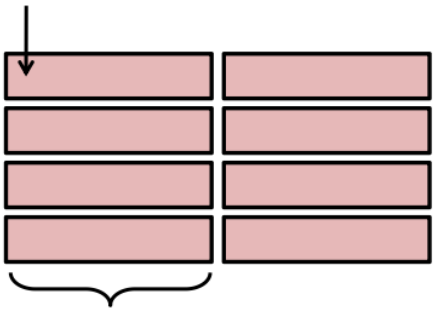
```
// After Loop Fusion
for (i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
    d[i] = a[i] + b[i] * c[i];
}
Revised version takes advantage of temporal locality
```

编译器优化方法举例之二：内外循环交换

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i< 16; i++)
            sum += a[i][j];
    return sum;
}
```

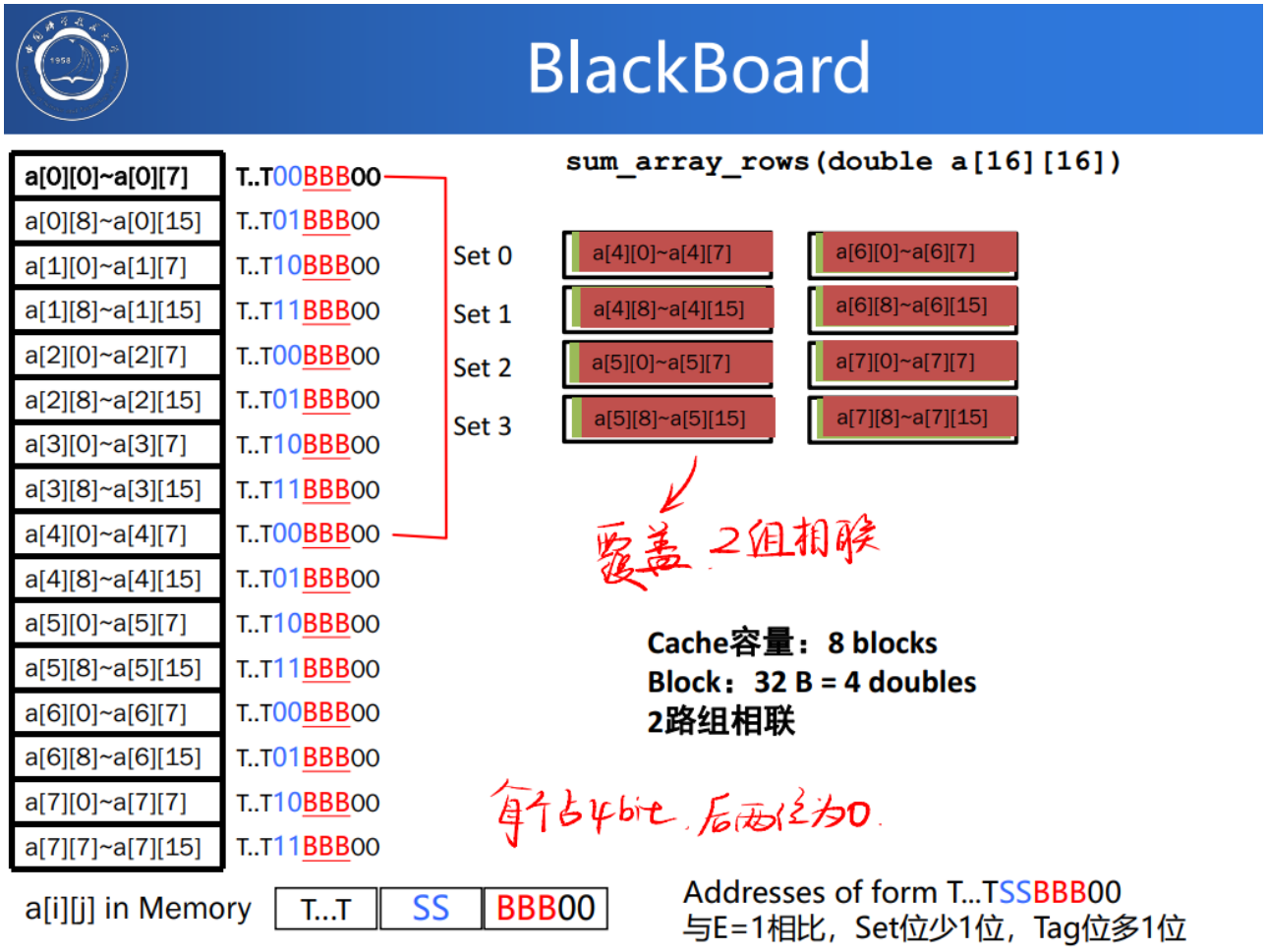
assume: cold (empty) cache,
a[0][0] goes here



32 B = 4 doubles

Ignore the variables sum, i, j

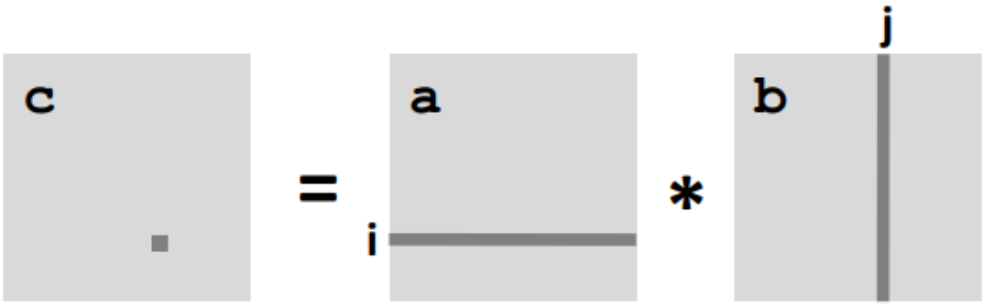
blackboard


BlackBoard
sum_array_rows(double a[16][16])
a[0][0]~a[0][7]
a[0][8]~a[0][15]
a[1][0]~a[1][7]
a[1][8]~a[1][15]
a[2][0]~a[2][7]
a[2][8]~a[2][15]
a[3][0]~a[3][7]
a[3][8]~a[3][15]
a[4][0]~a[4][7]
a[4][8]~a[4][15]
a[5][0]~a[5][7]
a[5][8]~a[5][15]
a[6][0]~a[6][7]
a[6][8]~a[6][15]
a[7][0]~a[7][7]
a[7][7]~a[7][15]
T..T00BBB00
T..T01BBB00
T..T10BBB00
T..T11BBB00
T..T00BBB00
T..T01BBB00
T..T10BBB00
T..T11BBB00
T..T00BBB00
T..T01BBB00
T..T10BBB00
T..T11BBB00
T..T00BBB00
T..T01BBB00
T..T10BBB00
T..T11BBB00
Set 0
Set 1
Set 2
Set 3
a[4][0]~a[4][7]
a[4][8]~a[4][15]
a[5][0]~a[5][7]
a[5][8]~a[5][15]
a[6][0]~a[6][7]
a[6][8]~a[6][15]
a[7][0]~a[7][7]
a[7][8]~a[7][15]
覆盖 2组相联
Cache容量：8 blocks
Block：32 B = 4 doubles
2路组相联
每个占4bit，后两位为0
a[i][j] in Memory
T...T
SS
BBB00
Addresses of form T...TSSBBB00
与E=1相比，Set位少1位，Tag位多1位

4.1 分块

基本的矩阵相乘

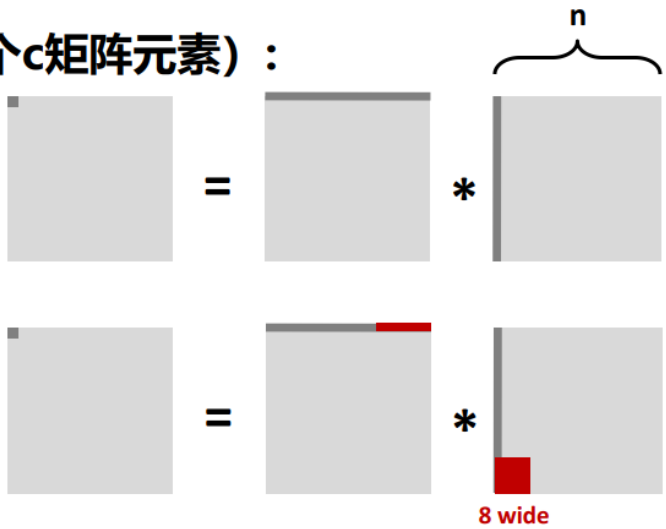
```
c = (double *) calloc(sizeof(double), n*n);

/* 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];
}
```



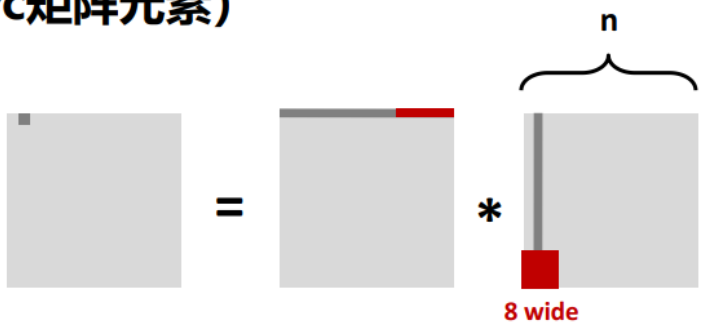
Cache 失效分析

- **假设:**
  - 矩阵元素为双精度浮点数（double型：8 bytes）
  - Cache 块大小 = 8 doubles
  - Cache 容量 C << n (much smaller than n)

- **第一次循环（计算第一个c矩阵元素）：**
  - n/8 + n = 9n/8 misses
  - Afterwards in cache: (schematic)



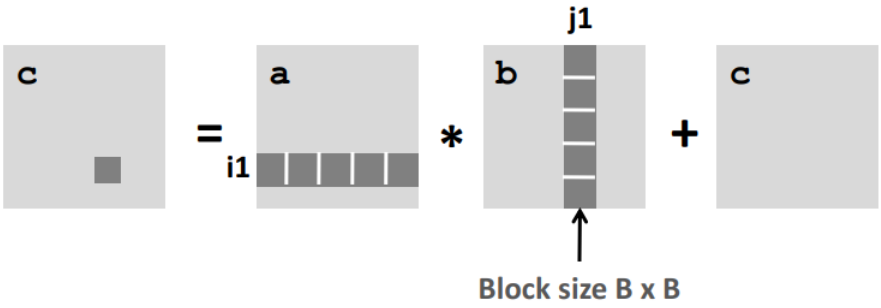
- **第2次循环（计算第2个c矩阵元素）**
  - Again:
    n/8 + n = 9n/8 misses



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

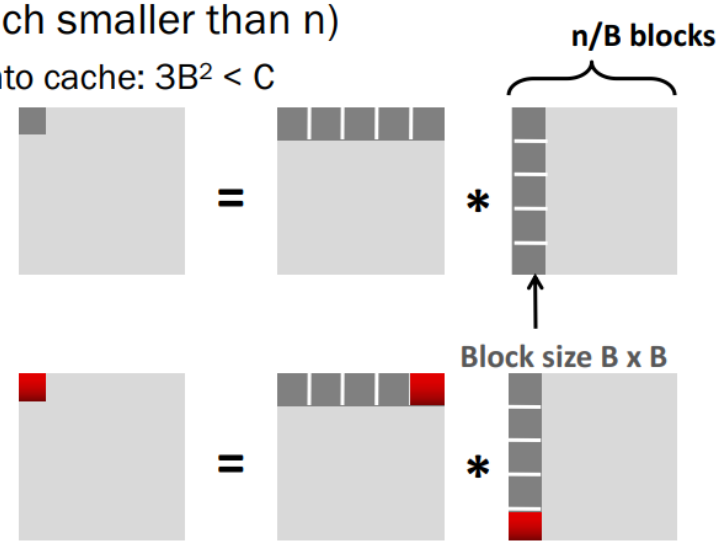
矩阵分块相乘

```
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];
}
```

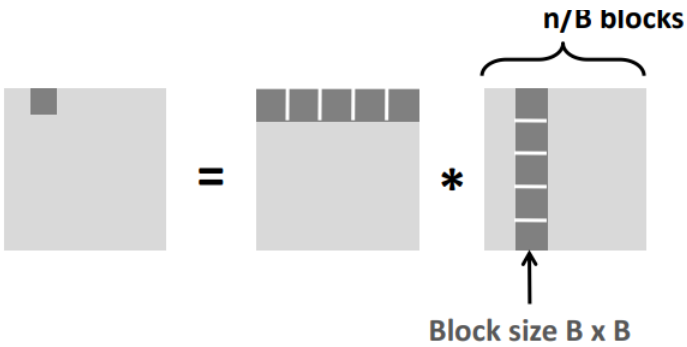


Cache 失效分析

- **假设:**
  - 矩阵元素为双精度浮点数（double型：8 bytes）
  - Cache 块大小 = 8 doubles
  - Cache 容量 C << n (much smaller than n)
  - 可以存放三块子矩阵： fit into cache: $3B^2 < C$



- **计算第1个子矩阵:**
  - $B^2/8$ misses for each block
    - **为什么?**
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)



- **计算第2个子矩阵:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$

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

- **矩阵不分块相乘失效次数: (9/8) * n^3**
- **矩阵分块相乘失效次数: 1/(4B) * n^3**

- **建议：尽可能分块大一些, 但需保证$3B^2 < C$!**

9、Hardware Prefetching

- **预取指令**
  - CPU在执行当前块代码时，硬件预取下一块代码
  - CPU可能马上就要执行这块代码，这样可以降低或消除Cache的访问失效
- **当块中有控制指令时，预取失效**
- **预取的指令可以放在Icache中，也可以放在其他地方（存取速度比Memory块的地方）**
- **AXP21064失效时，取2块指令块**
  - 目标块放在Icache，下一块放在ISB(指令流缓冲）中
  - 如果访问的块在ISB中，取消访存请求，直接从ISB中读，并发出对下一指令块的预取访存请求
- **研究结果：块大小为16字节，容量为4KB的直接映象Cache，1个块的指令流缓冲器，可以捕获15% - 25%的失效，4个块ISB可捕获50%的失效，16块ISB可捕获72%的失效**

- **举例：Alpha AXP21064采用指令预取技术，其实际失效率是多少？若不采用指令预取技术，Alpha AXP 21064的指令Cache必须为多大才能保持平均访存时间不变?**
  - 假设当指令不在指令Cache中，在预取缓冲区中找到时，需要多花1个时钟周期。
  - 假设预取命中率为25%，命中时间为1个时钟周期，失效开销为50个时钟周期
  - 8KB指令Cache的失效率为1.10%，16KB指令cache的失效率为0.64%

  - AMAT（预取）= 命中时间+失效率*预取命中率*1+失效率*（1-预取命中率）*失效开销

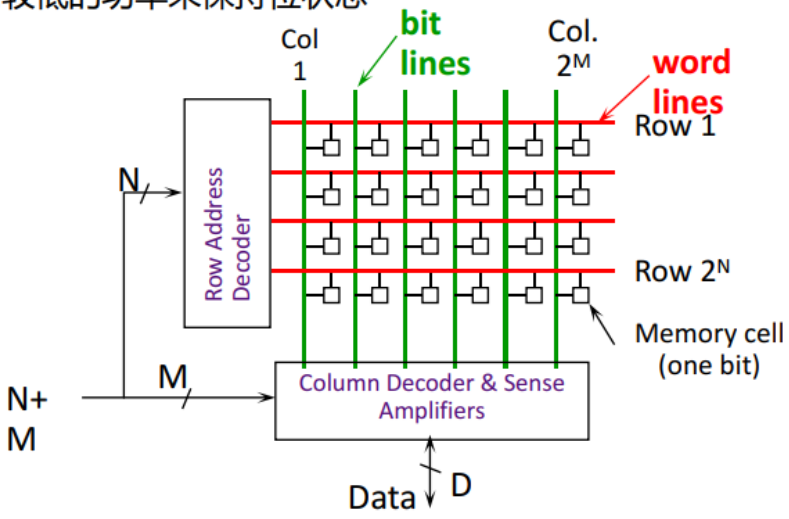
- **注意：预取是利用存储器的空闲带宽，而不是与正常的存储器操作竞争。**

10、Compiler Prefetching（1/2）

- **在ISA中增加预取指令，让编译器控制预取**
- **预取的种类**
  - 寄存器预取：把数据取到R中
  - Cache预取：只将数据取到Cache中，不放入寄存器
- **故障问题**
  - 两种类型的预取：故障性预取和非故障性预取
  - 所谓故障性预取：指在预取时若出现虚地址故障，或违反保护权限，就会有异常发生
  - 所谓非故障性预取：如导致异常就转化为空操作
- **只有在预取时，CPU可以继续执行的情况下，预取才有意义**
  - Cache在等待预取数据返回的同时，可以正常提供指令和数据，称为非阻塞Cache或非锁定Cache

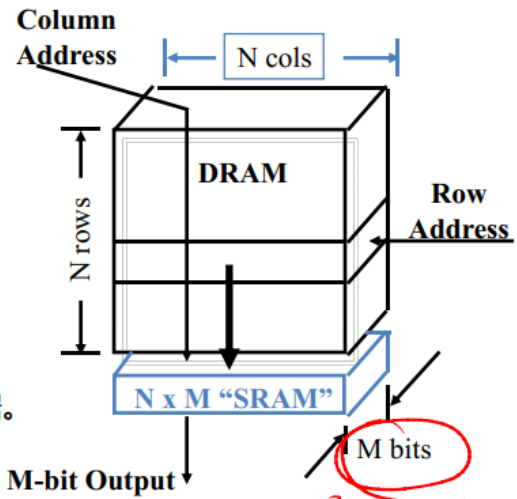
5 存储器

- **DRAM**
  - 破坏性读：读后需要重新写回，必须要周期性的刷新；每位1个 transistor
  - 地址线复用:
    - Lower half of address: column access strobe (CAS)
    - Upper half of address: row access strobe (RAS)
- **SRAM**
  - 每位6个transistors；只需较低的功率来保持位状态

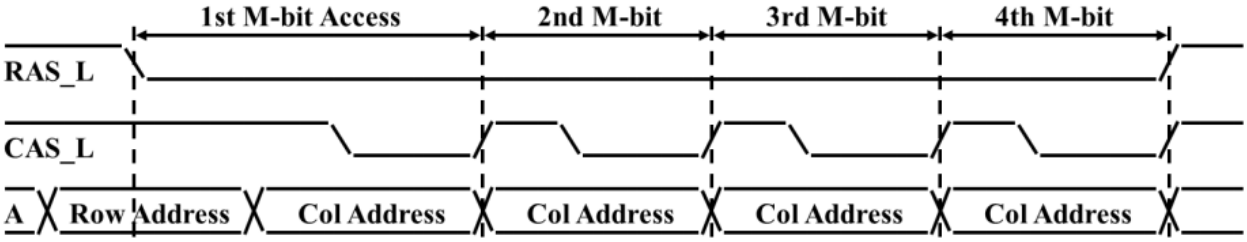


DRAM device性能优化

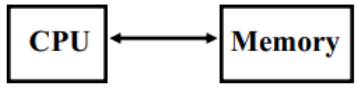
- Fast Page Mode Operation (异步)
  - Multiple accesses to same row
  - A bank include Multiple DRAM Array
- Synchronous DRAM (同步)
  - Added clock to DRAM interface
  - Burst mode with critical word first
  - Multiple banks on each DRAM device
  - Single Data Rate (SDR) - 单速率数据传输
    - 接受一个命令，一个时钟周期内传输一个数据。
  - Double data rate (DDR) - 双速率数据传输
    - 在不增加时钟速度的情况下传输两倍的数据

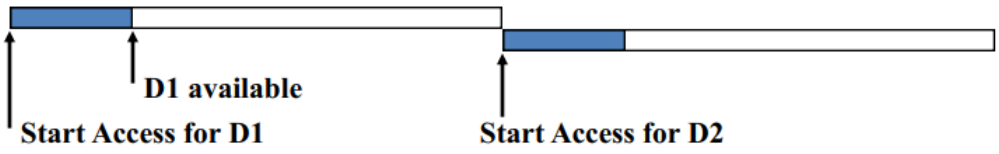


M个位平面叠加在一起

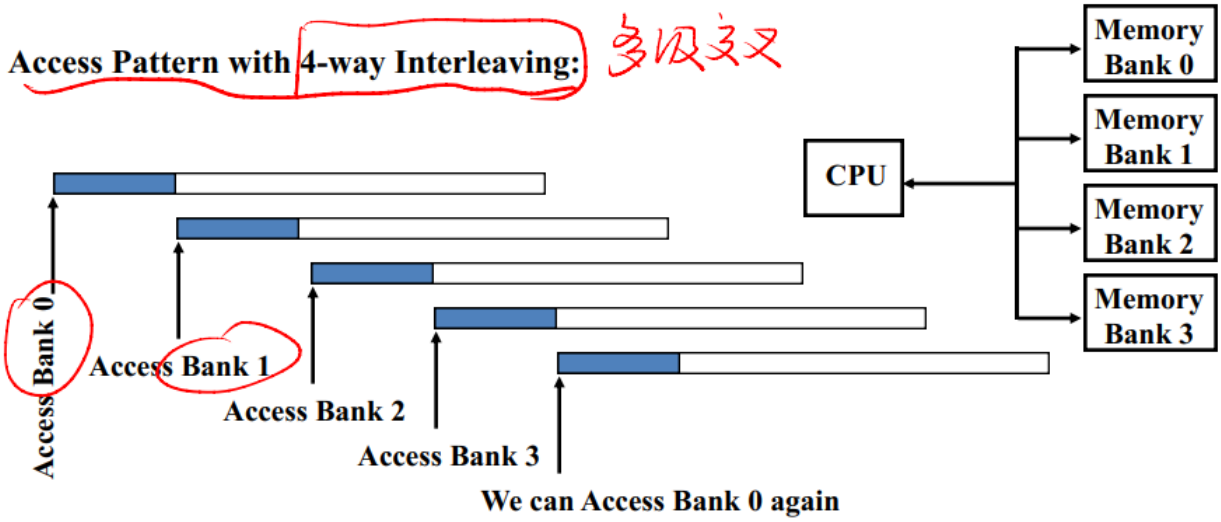


Access Pattern without Interleaving:

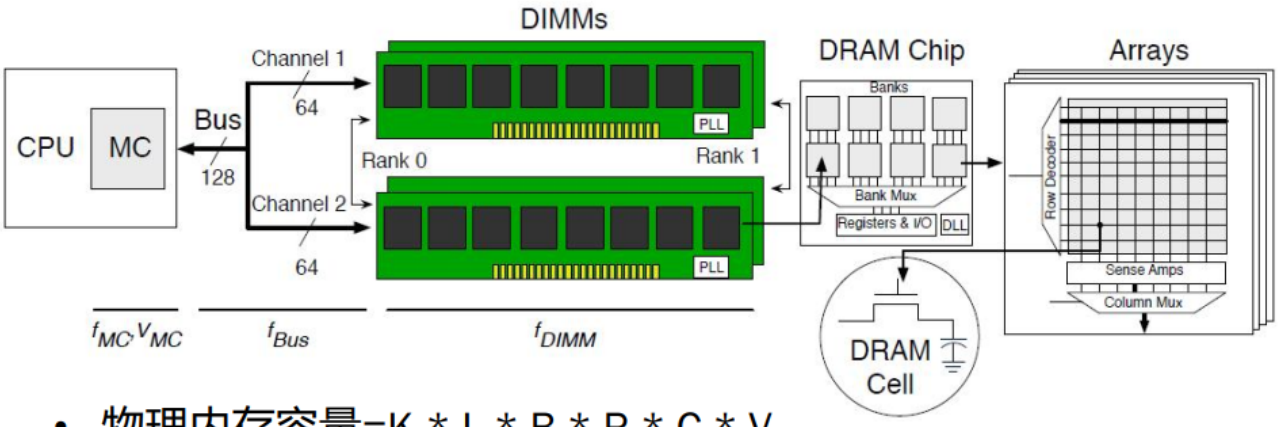




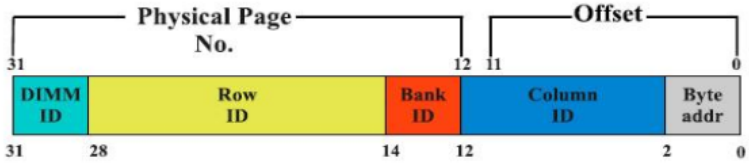
Access Pattern with 4-way Interleaving: 多级交叉

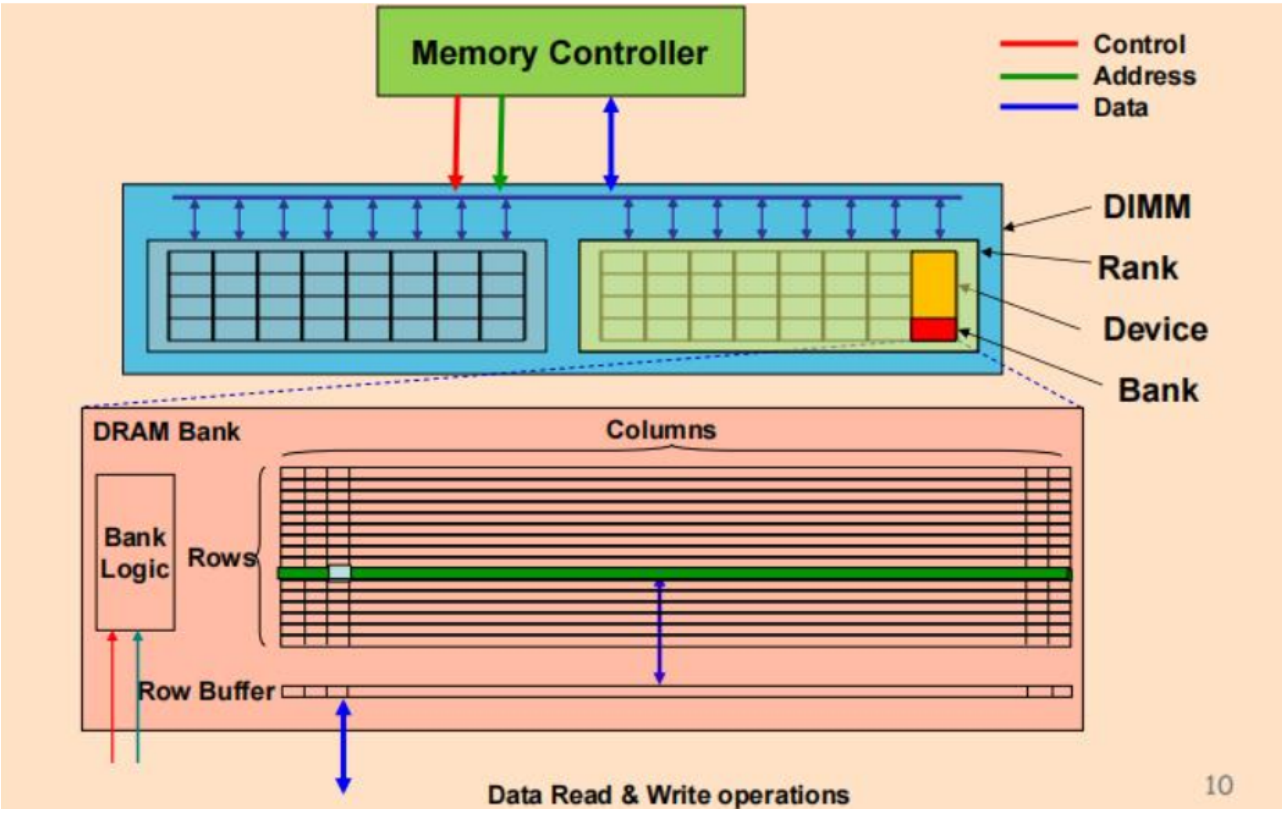


Channel：Multi-rank



- 物理内存容量=K * L * B * R * C * V.
  - K：channels(DIMM ID?)
  - L：ranks per channel
  - B：banks per rank
  - R：rows per bank
  - C：columns per row
  - V：bytes per column.





Cache与VM的区别

- **目的不同**
  - Cache是为了提高访存速度
  - VM是为了提高存储容量
- **替换的控制者不同**
  - Cache失效由硬件处理
  - VM的页失效通常由OS处理
    - 一般页失效开销很大，因此替换算法非常重要
- **地址空间**
  - VM空间由CPU的地址尺寸确定
  - Cache的大小与CPU地址尺寸无关
- **下一级存储器**
  - Cache下一级是主存
  - VM下一级是磁盘，大多数磁盘含有文件系统，文件系统寻址与主存不同，它通常在I/O空间中，VM的下一级通常称为SWAP空间

TLB之类，考的少……