# 计算机组成原理
# CH5_层次化存储

卢建良
lujl@ustc.edu.cn
2022年春季学期

# 提纲

- 5.1 引言
- 5.2 存储技术
- 5.3 cache基础
- 5.4 cache的性能评估和改进
- 5.5 可靠的存储器层次
- 5.6 虚拟机
- 5.7 虚拟存储
- 5.8 存储层次结构的一般框架
- 5.9 使用有限状态机控制简单的cache
- 5.10 并行和存储层次结构：cache一致性

# 提纲

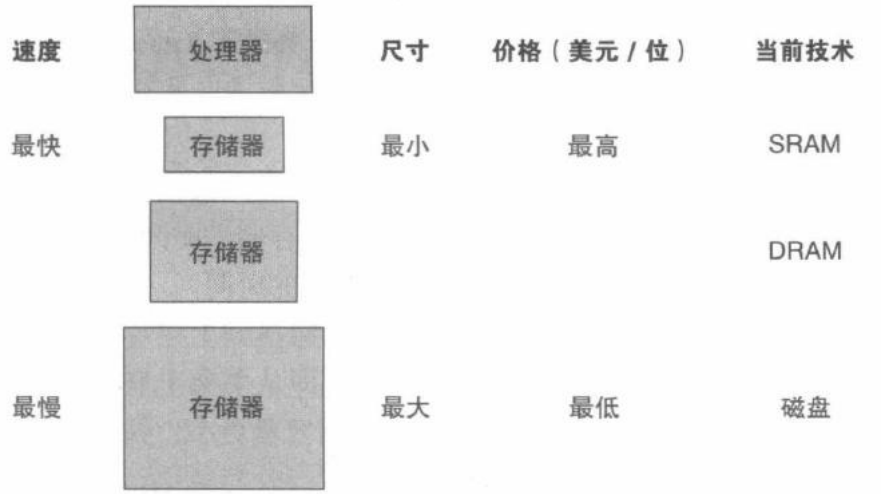
- 5.11 并行与存储层次结构：RAID
- 5.12 高级专题：实现缓存控制器
- 5.13 实例：Cortex-A53和Core i7的存储层次结构
- 5.14 实例：RISC-V系统的其它部分和特殊指令
- 5.15 加速：cache分块和矩阵乘法
- 5.16 谬误与陷阱
- 5.17 本章小结

# 5.1 引言

- 局部性原理
  - 程序在任何时候都会访问其地址空间中较小的一小部分
  - 时间局部性、空间局部性
  - 类比：图书馆中借书的例子
- 时间局部性
  - 如果某个数据项被访问，那么在不久的将来可能再次被访问
  - 指令：循环指令
  - 数据：循环指令相关的变量
- 空间局部性
  - 如果某个数据项被访问，与它地址相邻的数据项可能很快也将被访问
  - 指令：顺序执行的指令、循环指令
  - 数据：数组

# 5.1 引言

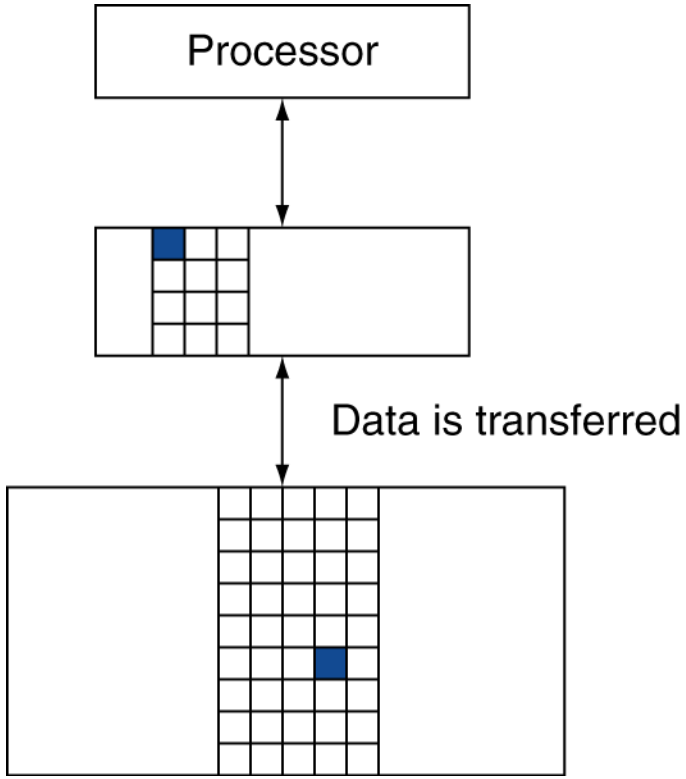
- 局部性原理的应用
  - 层次化的存储器
    - 多级存储采用的结构，与处理器距离越远，存储的容量越大，访问速度越慢
  - 所有信息存放在硬盘上
  - 将近期访问（及其临近）的数据项从硬盘拷贝到DRAM
    - 从硬盘加载到主存/内存
    - 如：操作系统程序、系统服务程序、正在运行的用户程序等
  - 将更加近期访问的数据项从DRAM拷贝到一个更小的SRAM
    - 从内存/主存拷贝到缓存/cache
    - cache离CPU更近
    - 速度更快，但容量更小
    - cache也可能有多个层次



# 5.1 引言

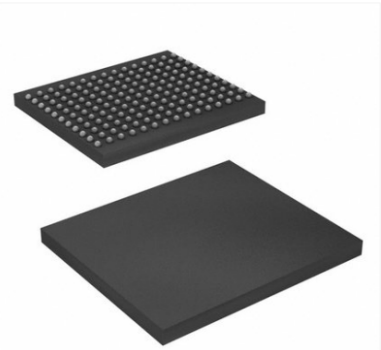
- 存储器层次结构

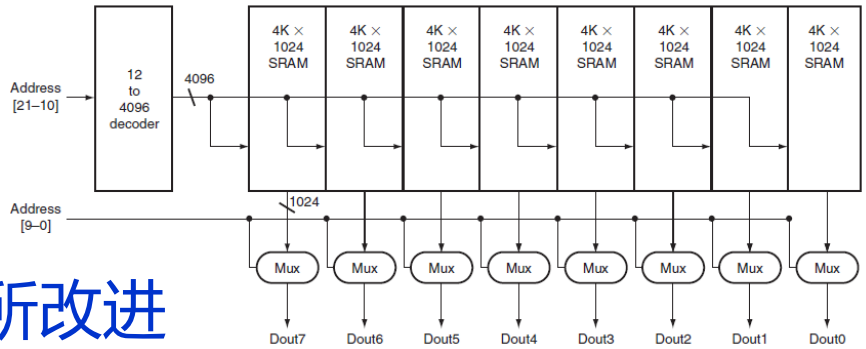
- 块/行（block/line）
  - 在相邻层次结构间信息交换的最小单元
  - 也是缓存中存储信息的最小单位
  - 一般会包含多个字：16~256字节
- 命中（hit）
  - 待访问的数据在上层存储单元中
  - 命中率：命中次数/访问次数
- 缺失（miss）
  - 待访问的数据不在上层存储单元中
  - 缺失率：缺失次数/访问次数 = 1-命中率
  - 数据缺失时会从下一层次获取数据

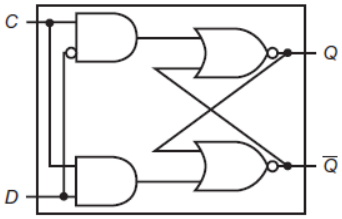


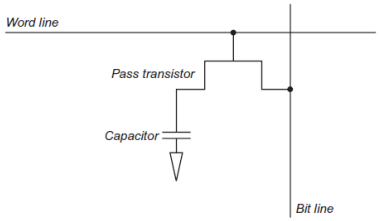
# 5.2 存储技术

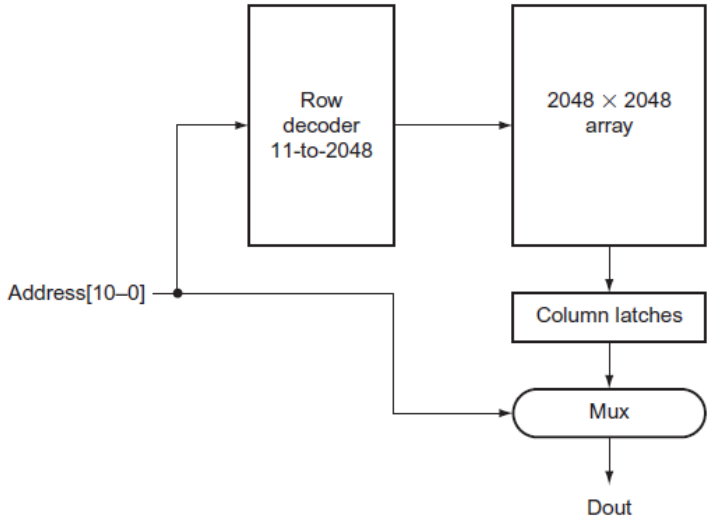
- 静态RAM（SRAM）
  - 0.5ns~2.5ns， $2000 – $5000/GB
- 动态RAM（DRAM）
  - 50ns~70ns， $20 – $75/GB
- 磁盘
  - 5ms~20ms， $0.20 – $2/GB
- 随着工艺的进步，上述参数会有所改进
- 理想的存储器
  - 访问时间与SRAM相当
  - 容量和每GB成本与磁盘相当



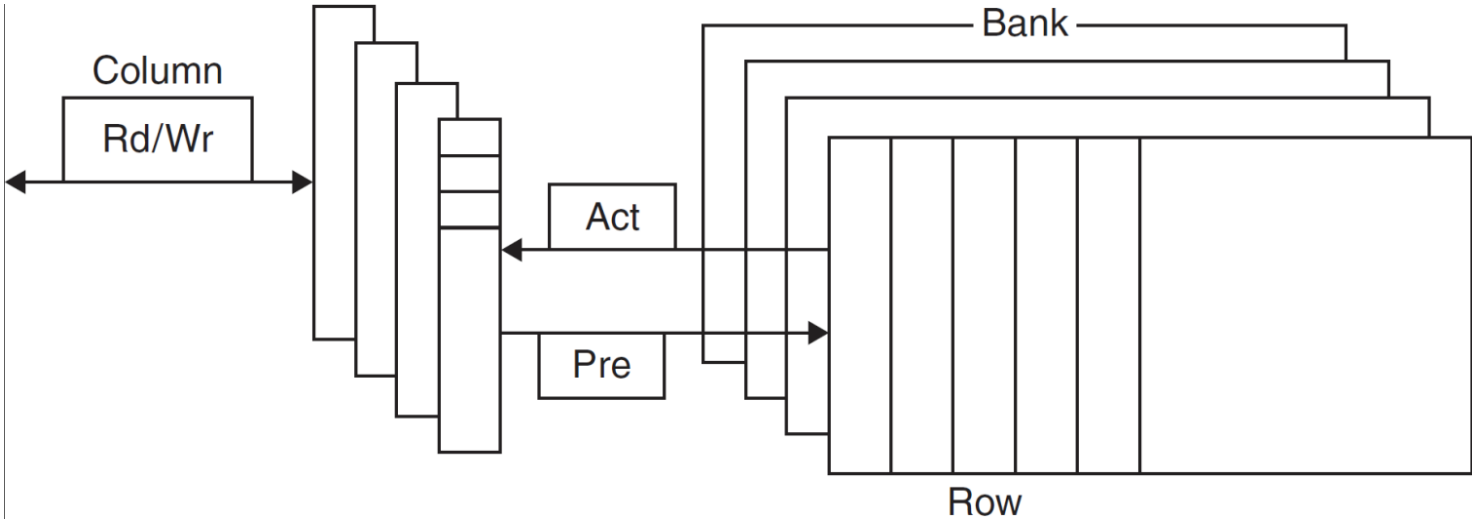



# 5.2 存储技术

- DRAM技术
  - 以电容充放电的方式存放数据
  - 每bit数据单元使用一个晶体管控制
  - 必须进行周期性的刷新
    - 刷新操作：读取数据内容，然后重新写回
    - 刷新时，以DRAM的“行”作为操作单元



# 5.2 存储技术

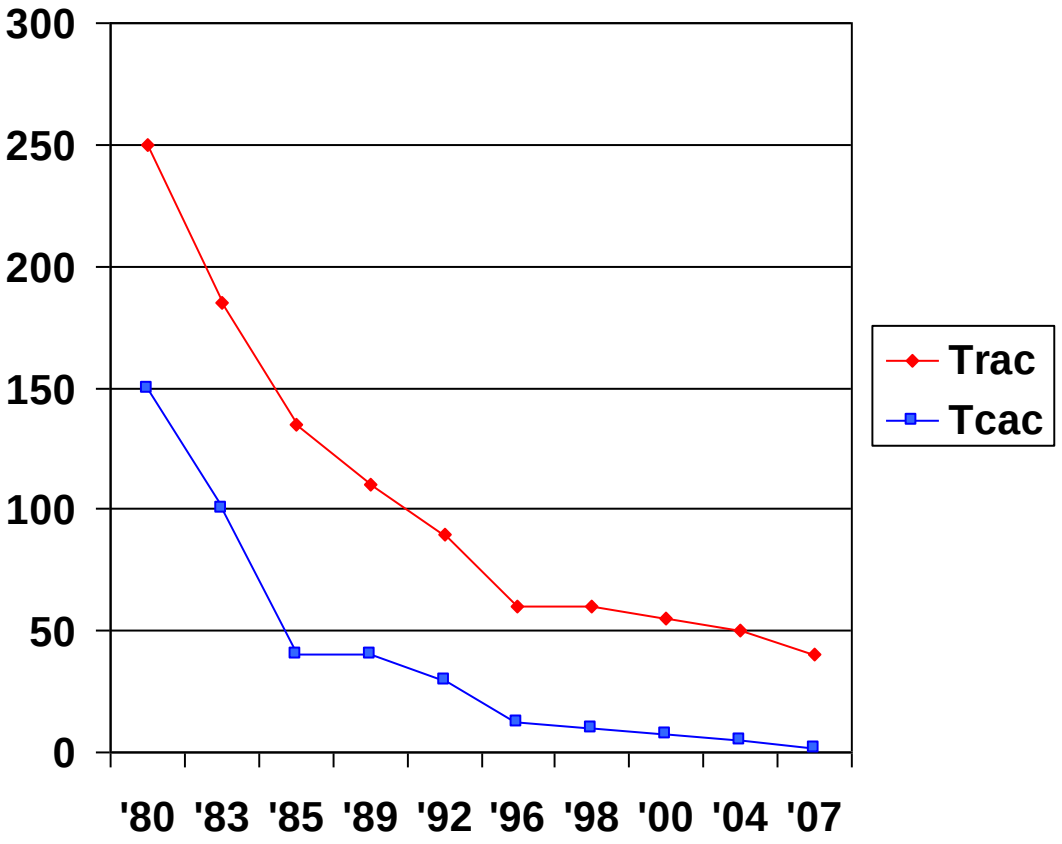
- DRAM的组织结构

- DRAM中的位被组织成矩形阵列
  - DRAM访问整行
  - 突发模式：连续提供一行中的所有字，减少延迟
- 双倍数据速率（DDR）DRAM
  - 在上升和下降时钟边缘传输
- 四倍数据速率（QDR）DRAM
  - 独立的DDR输入和输出

# 5.2 存储技术

- DRAM技术演化

| Year | Capacity | $/GB |
|---|---|---|
| 1980 | 64Kbit | $1500000 |
| 1983 | 256Kbit | $500000 |
| 1985 | 1Mbit | $200000 |
| 1989 | 4Mbit | $50000 |
| 1992 | 16Mbit | $15000 |
| 1996 | 64Mbit | $10000 |
| 1998 | 128Mbit | $4000 |
| 2000 | 256Mbit | $1000 |
| 2004 | 512Mbit | $250 |
| 2007 | 1Gbit | $50 |



# 5.2 存储技术

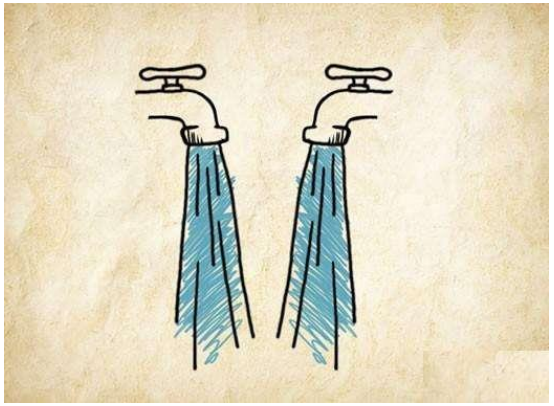
- 影响DRAM性能的相关因素

- 行缓冲区（Row buffer）
  - 允许同时读取和刷新多个字
- 同步动态随机存储器
  - 允许以突发方式连续访问，无需发送每个地址
  - 提高带宽
- DRAM banking
  - 允许同时访问多个DRAM
  - 提高带宽

# 5.2 存储技术

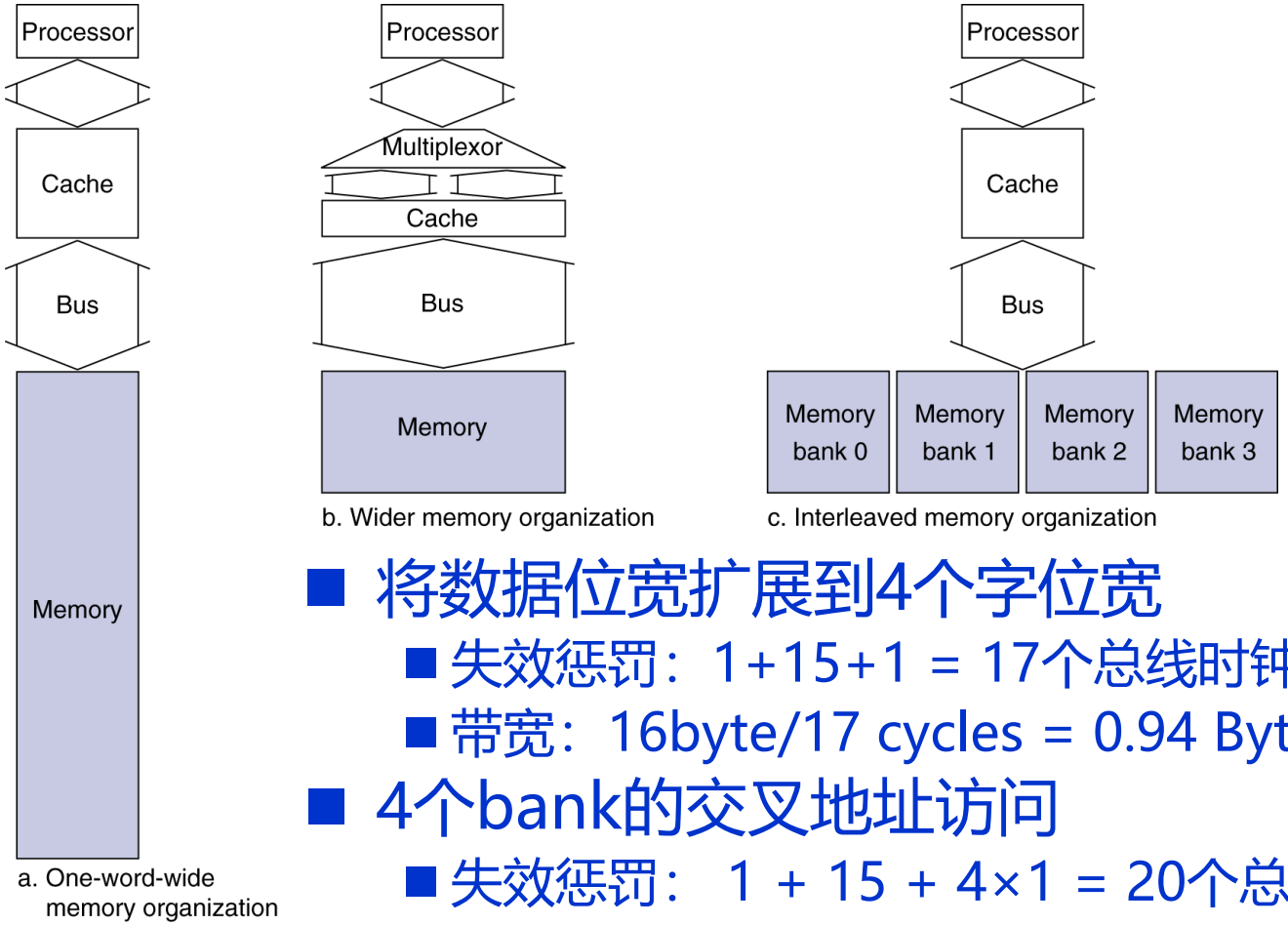
- 从DRAM取数据 vs 从水龙头取水

- DRAM有访问延时，打开水龙头需要耗时
- CPU消耗速度大于DRAM带宽，水的消耗速率大于排水量
- 数据存在局部性（重复利用），水可重复利用
- 提高效率的途径
  - 增加带宽
  - 增加重复使用的效率



# 5.2 存储技术

- 增加存储器带宽




a. One-word-wide memory organization

b. Wider memory organization

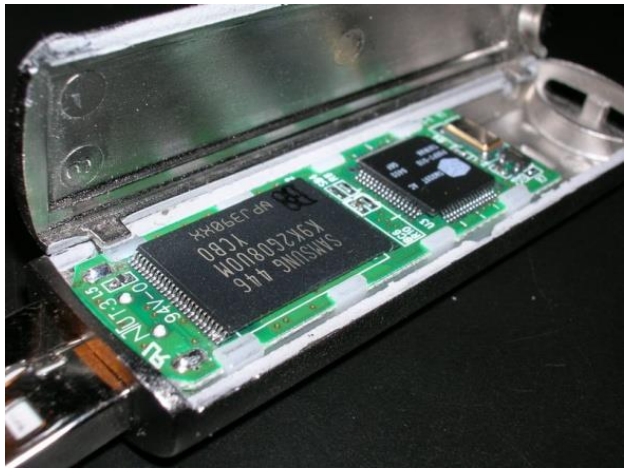
c. Interleaved memory organization

- 将数据位宽扩展到4个字位宽
  - 失效惩罚：1+15+1 = 17个总线时钟周期
  - 带宽：16byte/17 cycles = 0.94 Byte/cycle
- 4个bank的交叉地址访问
  - 失效惩罚： 1 + 15 + 4×1 = 20个总线时钟周期
  - 带宽： 16 bytes / 20 cycles = 0.8 B/cycle

# 5.2 存储技术

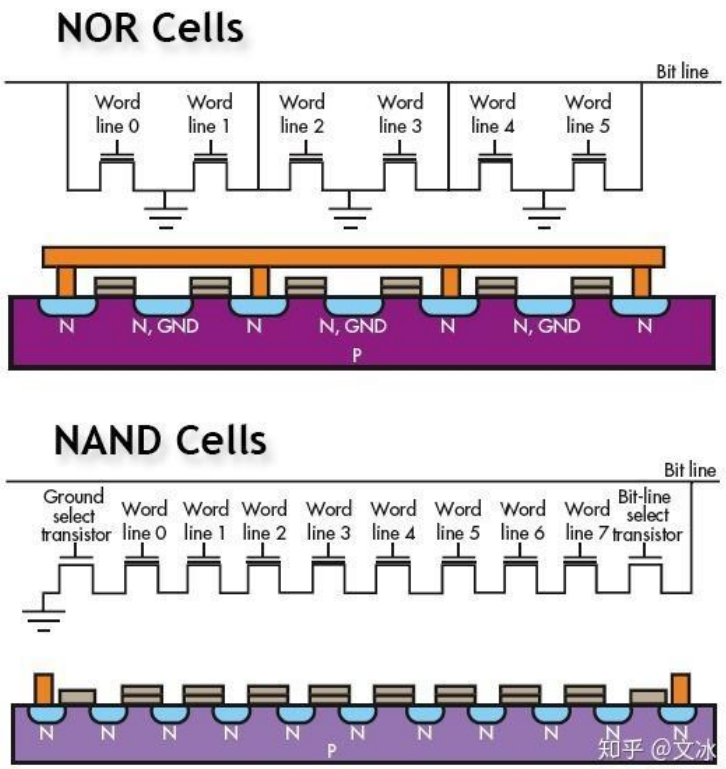
- 闪存(Flash)
  - 闪存是一种新型的EEPROM（电可擦除只读存储器）

- 闪存是一种非易失性的半导体存储器件
- 比磁盘快100~1000倍
- 体积更小、功耗更低、性能更稳定
- 价格介于磁盘与DRAM之间

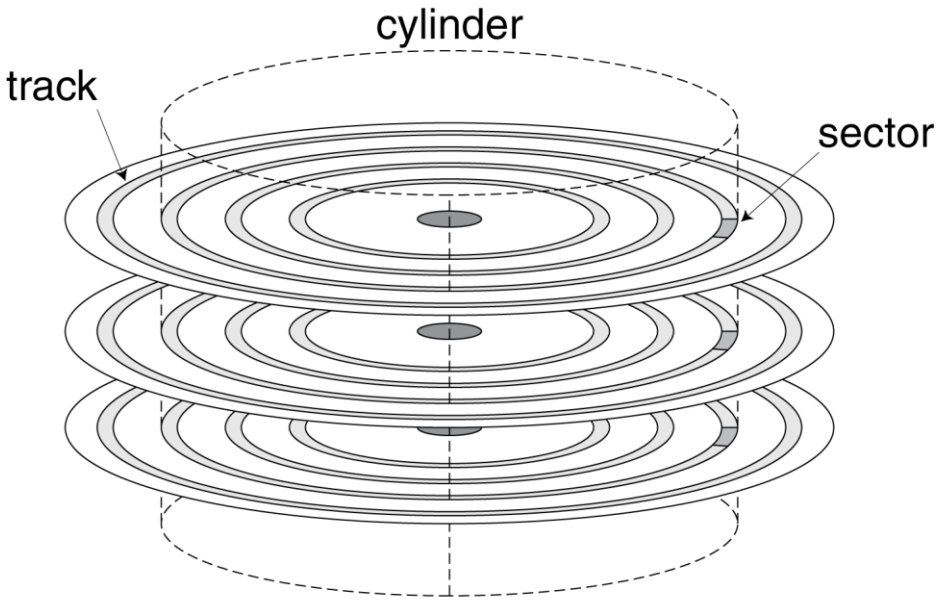
# 5.2 存储技术

- 闪存种类
  - NOR flash、NAND flash
- NOR flash：bit单元类似NOR门
  - 随机读/写访问
  - 用于嵌入式系统中的指令存储器
- NAND flash：bit单元类似NAND门
  - 更密集（位/面积），但一次阻塞访问
  - 每GB更便宜
  - 用于USB密钥、媒体存储...
- 经过1000次访问后，闪存位会磨损
  - ~~不适合直接替代RAM或磁盘~~（现在已可以）
  - 磨损均衡：将数据重新映射到较少使用的块
  - 可以类比快递柜的使用原理




# 5.2 存储技术

- 磁盘存储
  - 非易失性，旋转磁存储器



# 5.2 存储技术

- 磁盘扇区和访问
- 每个磁盘扇区记录的信息
  - 扇区ID
  - 数据（512字节，建议4096字节）
  - 纠错码（ECC）
    - 用于隐藏缺陷和记录错误
  - 同步字段和间隙
- 影响访问磁盘扇区数据的因素
  - 等待其他访问时的排队延迟
  - 寻道：移动头部
  - 旋转潜伏期
  - 数据传输
  - 控制器开销

# 5.2 存储技术

- 磁盘访问示例
  - 512B扇区，15000rpm，4ms平均寻道时间，100MB/s传输速率，0.2ms控制器开销，空闲磁盘
- 平均读取时间
  - 4ms寻道时间
  - +½/（15000/60）=2ms旋转延迟
  - +512/100MB/s=0.005ms传输时间
  - +0.2ms控制器延迟
  - =6.2ms
- 如果实际平均寻道时间为1ms
  - 平均读取时间=3.2ms

# 5.2 存储技术

- 磁盘性能问题
- 制造商引用平均寻道时间
  - 基于所有可能的追求
  - 局部性和操作系统调度导致实际平均寻道时间缩短
- 智能磁盘控制器在磁盘上分配物理扇区
  - 向主机提供逻辑扇区接口
  - SCSI、ATA、SATA
- 磁盘驱动器包括缓存
  - 预取扇区以预期访问
  - 避免寻道和旋转延迟

# 5.3 Cache基础

- Cache实验
- 存储器：512Byte
- Cache：128Byte
- 块大小：16Byte

- 右侧为地址序列

- 9′ b0_0000_0000
- 9′ b0_0000_0100
- 9′ b0_0000_1000
- 9′ b0_0000_1100
- 9′ b0_0001_0000
- 9′ b0_0001_0100
- 9′ b0_0001_1000
- 9′ b0_1000_0100
- 9′ b0_1000_1000
- 9′ b1_0000_0000

**CPU**

| 编号 | 地址 | 有效 | Tag | 块（word3~word0） |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |

| 编号 | 地址 | Word 3 | Word 2 | Word 1 | Word 0 |
|---|---|---|---|---|---|
| 0 | 0_0000_0000 | 3 | 2 | 1 | 0 |
| 1 | 0_0001_0000 | 7 | 6 | 5 | 4 |
| 2 | 0_0010_0000 | 11 | 10 | 9 | 8 |
| 3 | 0_0011_0000 | 15 | 14 | 13 | 12 |
| 4 | 0_0100_0000 | 19 | 18 | 17 | 16 |
| 5 | 0_0101_0000 | 23 | 22 | 21 | 20 |
| 6 | 0_0110_0000 | 27 | 26 | 25 | 24 |
| 7 | 0_0111_0000 | 31 | 30 | 29 | 28 |
| 8 | 0_1000_0000 | 35 | 34 | 33 | 32 |
| 9 | 0_1001_0000 | 39 | 38 | 37 | 36 |
| 10 | 0_1010_0000 | … | … | … | … |
| 11 | 0_1011_0000 | … | … | … | … |
| 12 | 0_1100_0000 | … | … | … | … |
| 13 | 0_1101_0000 | … | … | … | … |
| 14 | 0_1110_0000 | … | … | … | … |
| 15 | 0_1111_0000 | … | … | … | … |
| 16 | 1_0000_0000 | … | … | … | … |
| 17 | 1_0001_0000 | … | … | … | … |
| 18 | 1_0010_0000 | … | … | … | … |
| 19 | 1_0011_0000 | … | … | … | … |
| 20 | 1_0100_0000 | … | … | … | … |
| 21 | 1_0101_0000 | … | … | … | … |
| 22 | 1_0110_0000 | … | … | … | … |
| … | … | … | … | … | … |

# 5.3 Cache基础

- Cache实验
- 存储器：512Byte
- Cache：128Byte
- 块大小：16Byte

- 右侧为指令序列

- 9′ b0_0000_0000
- 9′ b0_0001_0000
- 9′ b0_0010_0000
- 9′ b0_0011_0000
- 9′ b0_0100_0000
- 9′ b0_0101_0000
- 9′ b0_1000_0100

CPU

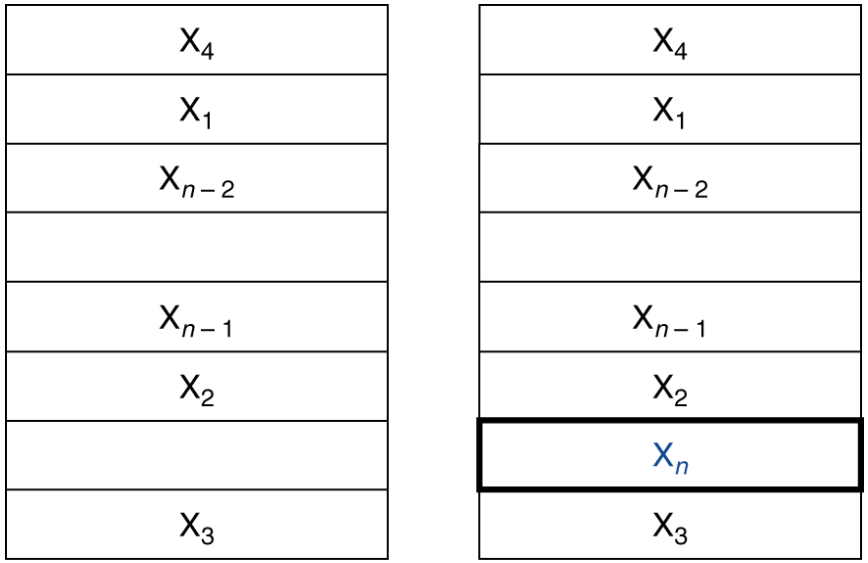
| 编号 | 地址 | 有效 | tag | 块1 | 有效 | tag | 块0 |
|---|---|---|---|---|---|---|---|
| 0 | | | | | | | |
| 1 | | | | | | | |
| 2 | | | | | | | |
| 3 | | | | | | | |

| 编号 | 地址 | Word 3 | Word 2 | Word 1 | Word 0 |
|---|---|---|---|---|---|
| 0 | 0_0000_0000 | 3 | 2 | 1 | 0 |
| 1 | 0_0001_0000 | 7 | 6 | 5 | 4 |
| 2 | 0_0010_0000 | 11 | 10 | 9 | 8 |
| 3 | 0_0011_0000 | 15 | 14 | 13 | 12 |
| 4 | 0_0100_0000 | 19 | 18 | 17 | 16 |
| 5 | 0_0101_0000 | 23 | 22 | 21 | 20 |
| 6 | 0_0110_0000 | 27 | 26 | 25 | 24 |
| 7 | 0_0111_0000 | 31 | 30 | 29 | 28 |
| 8 | 0_1000_0000 | 35 | 34 | 33 | 32 |
| 9 | 0_1001_0000 | 39 | 38 | 37 | 36 |
| 10 | 0_1010_0000 | ... | ... | ... | ... |
| 11 | 0_1011_0000 | ... | ... | ... | ... |
| 12 | 0_1100_0000 | ... | ... | ... | ... |
| 13 | 0_1101_0000 | ... | ... | ... | ... |
| 14 | 0_1110_0000 | ... | ... | ... | ... |
| 15 | 0_1111_0000 | ... | ... | ... | ... |
| 16 | 1_0000_0000 | ... | ... | ... | ... |
| 17 | 1_0001_0000 | ... | ... | ... | ... |
| 18 | 1_0010_0000 | ... | ... | ... | ... |
| 19 | 1_0011_0000 | ... | ... | ... | ... |
| 20 | 1_0100_0000 | ... | ... | ... | ... |
| 21 | 1_0101_0000 | ... | ... | ... | ... |
| 22 | 1_0110_0000 | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... |

# 5.3 Cache基础

- Cache存储器
  - 一个隐藏或者存储信息的安全场所
  - 在存储器层次结构中离CPU最近的一层

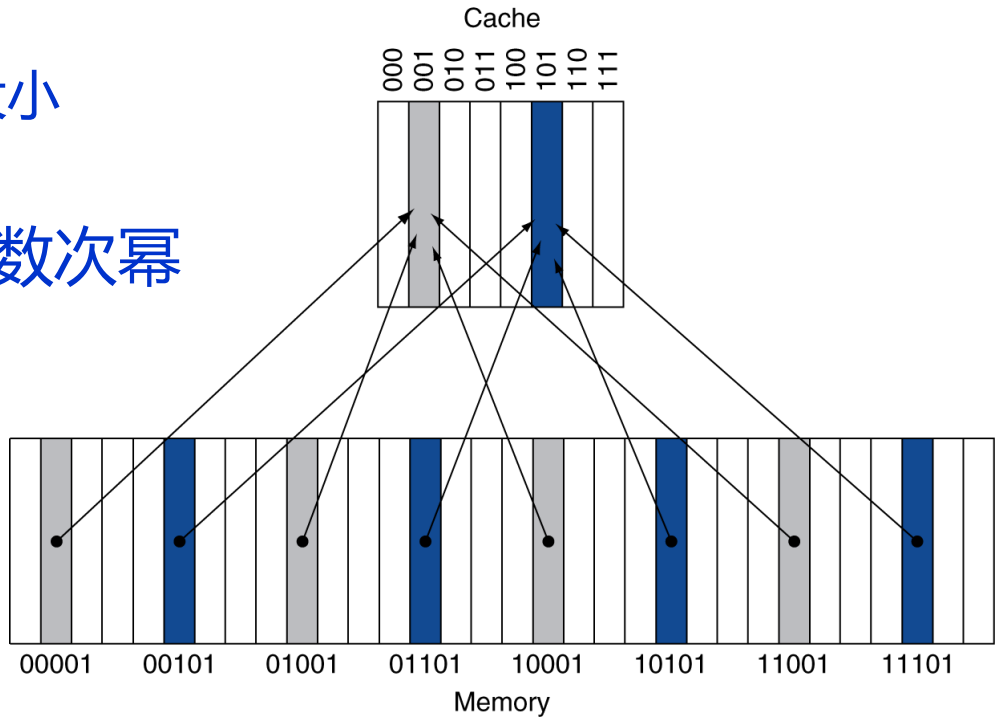
- 示例：给定访问序列$X_1, \ldots, X_{n-1}, X_n$
  - 如何知道访问的数据是否存在
  - 应该到哪里获取数据?




a. Before the reference to $X_n$　　b. After the reference to $X_n$

# 5.3 Cache基础

- 直接映射Cache
  - 一种cache结构，其中每个存储地址都映射到cache中的确定位置
- 映射位置由地址决定
- 直接映射，只有一个选项
  - 块地址 % Cache中的块数量
  - 块地址：内存物理地址 / 块大小
- 块（block）大小是2的整数次幂
- 块数量是2的整数次幂
- 低位地址用于在块内寻址



# 5.3 Cache基础

- 标签和有效位
- 我们如何知道哪个特定块存储在缓存位置?
  - 存储块地址和数据
  - 实际上，只需要地址的高位字段（高多少位?）
  - 该字段称为标签（Tag）
- 如果某个位置没有数据怎么办?
  - 有效位：1=存在，0=不存在
  - 初始值为0

# 5.3 Cache基础

- Cache示例
  - 8个block，每个block含1个字，直接映射
  - 初始状态

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | N | | |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | N | | |
| 111 | N | | |

# 5.3 Cache基础

- Cache示例
  - 8个block，每个block含1个字，直接映射

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 22 | 10 110 | Miss | 110 |

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | N | | |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| **110** | **Y** | **10** | **Mem[10110]** |
| 111 | N | | |

# 5.3 Cache基础

- Cache示例
  - 8个block，每个block含1个字，直接映射

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 26 | 11 010 | Miss | 010 |

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| **010** | **Y** | **11** | **Mem[11010]** |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

# 5.3 Cache基础

- Cache示例

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 22 | 10 110 | Hit | 110 |
| 26 | 11 010 | Hit | 010 |

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | Y | 11 | Mem[11010] |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

# 5.3 Cache基础

- Cache示例

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 16 | 10 000 | Miss | 000 |
| 3 | 00 011 | Miss | 011 |
| 16 | 10 000 | Hit | 000 |

| Index | V | Tag | Data |
|---|---|---|---|
| **000** | **Y** | **10** | **Mem[10000]** |
| 001 | N | | |
| 010 | Y | 11 | Mem[11010] |
| **011** | **Y** | **00** | **Mem[00011]** |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

# 5.3 Cache基础

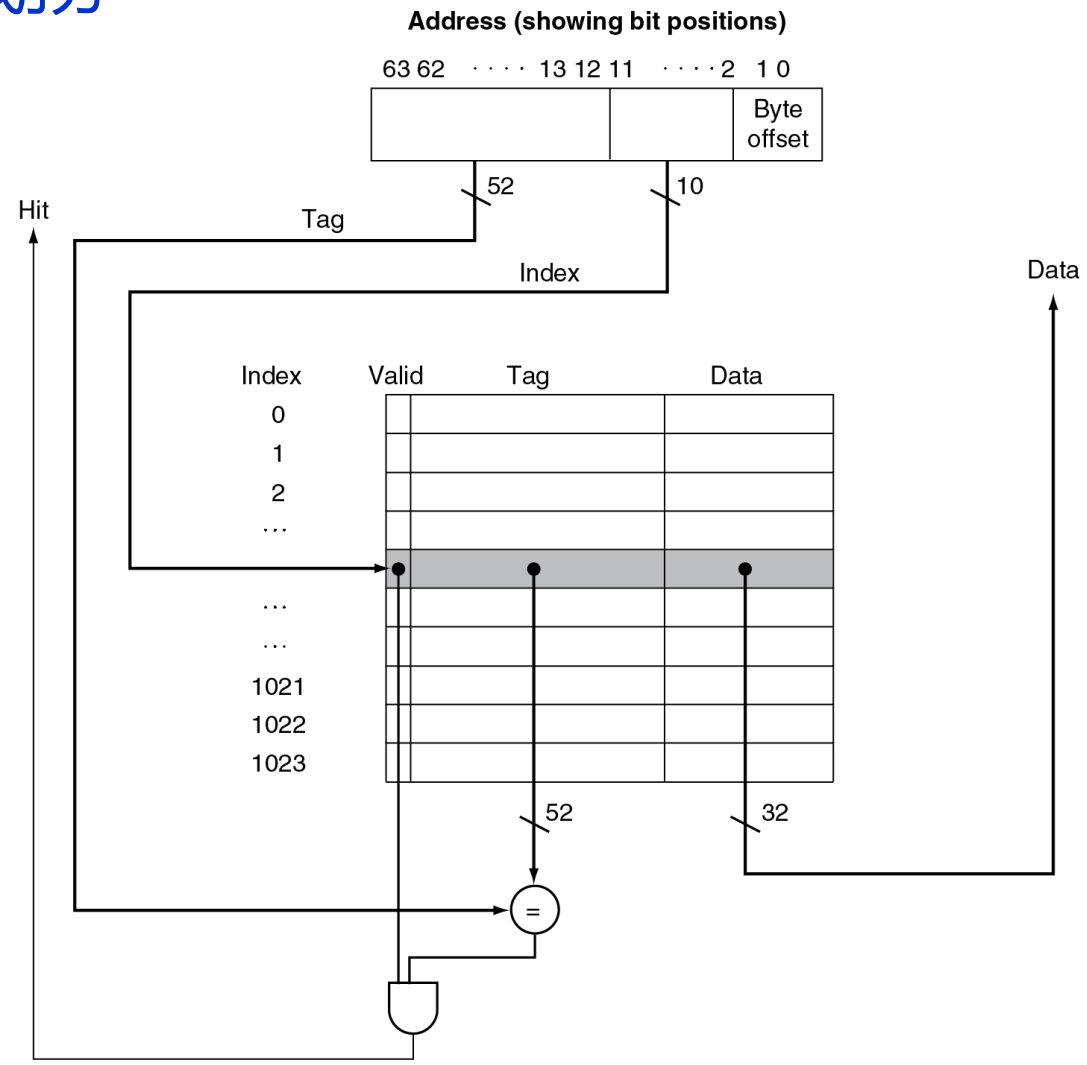
- Cache示例

| Word addr | Binary addr | Hit/miss | Cache block |
|---|---|---|---|
| 18 | 10 010 | Miss | 010 |

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | Y | 10 | Mem[10000] |
| 001 | N | | |
| **010** | **Y** | **10** | **Mem[10010]** |
| 011 | Y | 00 | Mem[00011] |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

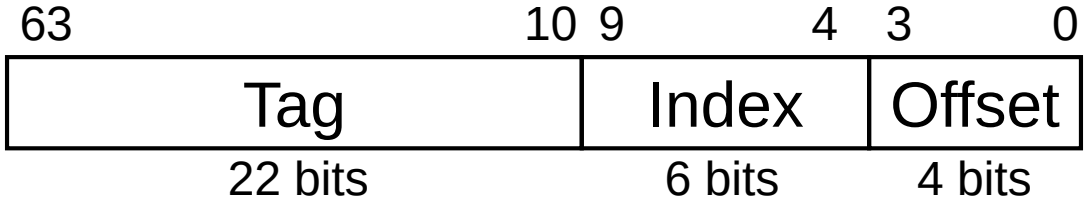
# 5.3 Cache基础

- 地址字段划分



# 5.3 Cache基础

- 示例：增加块（block）大小
  - 64个块，16字节/块
    - 地址1200对应于哪个块?
  - [主存的]块地址 = 物理地址 / 块大小　1200/16 = 75
  - [缓存的]块地址 = 主存块地址 % 缓存块数量　75 % 64 = 11

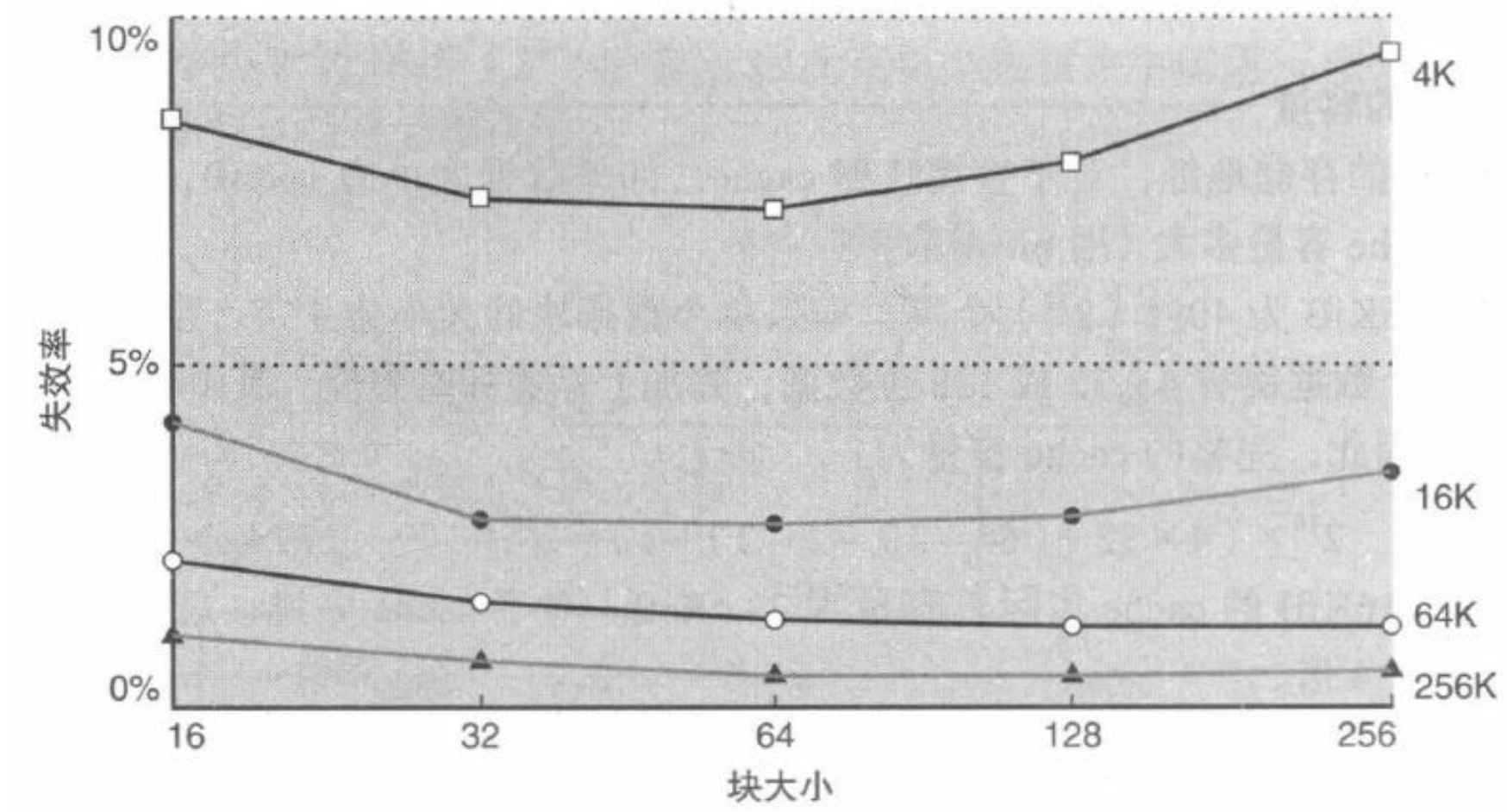


# 5.3 Cache基础

- 块大小设置的相关考虑
  - 由于空间局部性原理，增加block大小可以减少失效率
  - 对于固定大小的缓存，增加block大小，将导致block数量减少
  - block数量减少，将导致更多竞争，从而增加失效率
- 失效惩罚的相关考虑
  - block越大，数据失效所造成的延时越大
  - 较大的延时将会抵消掉降低失效率所带来的收益
- 对于大数据块所造成延时的改进办法
  - 提早重启技术
    - 只要所需数据返回来就继续执行，无须等数据块中的所有数据都完成传输
  - 关键字先行技术
    - 重新组织存储，让数据块中所需的数据首先从内存传输到cache中
  - 会带来新的问题

# 5.3 Cache基础

- 块大小与失效率间的关系



# 5.3 Cache基础

- 缓存失效（Cache Miss）

- Cache命中时，CPU可正常执行
- Cache失效时
  - 阻塞CPU流水线
  - 从下一级存储器（如主存）获取一个数据块
  - 对于指令cache失效
    - 重新启动取指操作
  - 对于数据cache失效
    - 完成数据访问

# 5.3 Cache基础

- 在数据写入命中时，可以只更新缓存中的块
  - 但这样一来，缓存和内存就会不一致
- 写直达（write-through）
  - 又称写穿透，一种写策略，写操作总是同时更新cache和下一级存储，保证两者之间的数据一致
  - 写直达：更新缓存的同时也更新内存
- 但这会让写操作耗时更长
  - eg：如果基本CPI=1，10%的指令是存储，写入内存需要100个周期
    - 有效CPI=1+0.1×100=11
- 解决方案：写入缓冲区
  - 使用缓冲区保存需要写入内存的数据
  - CPU立即继续
    - 仅当写入缓冲区已满时写入暂停，阻塞CPU

# 5.3 Cache基础

- 写返回（Write-Back）
  - 一种写策略，处理写操作时，只更新cache中对应数据块的数值，当该数据块被替换时，再将更新后的数据块写入下一级存储
- 写返回策略：在数据写入命中时，只需更新缓存中的块
  - 跟踪每个数据块是否是脏块
  - 什么是脏块？...
- 当更换脏块时
  - 写回存储器
  - 可以使用写缓冲区，将要替换的块存在到缓冲区中

# 5.3 Cache基础

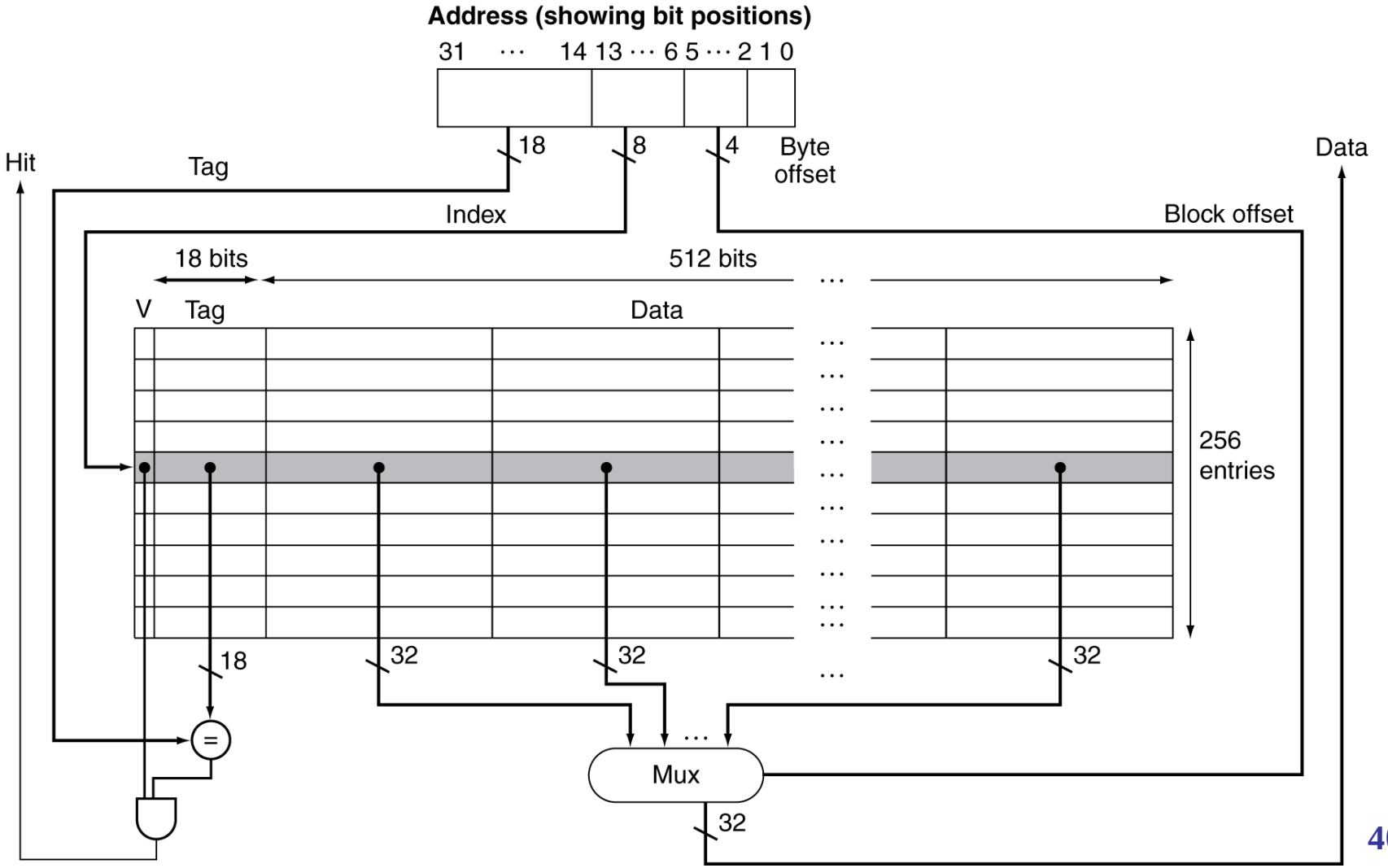
- 写分配（write allocate）
  - 如果发生写失效（write miss），该怎么办?
  - 写穿透策略中，如果要写入的块不在cache中，则在cache中为其分配一个数据块，称为写分配
  - 与其相对应的是写不分配策略（no write allocate），多用于内存的初始化操作
- 对于写直达策略
  - 选项1——未命中时再分配：先读取块
  - 选项2——直接写：不要读取数据块
    - 因为程序通常会在读取前写入整个块（例如初始化）
- 对于写返回策略
  - 通常会读取整个数据块

# 5.3 Cache基础

- 实例：Intrinsity FastMATH处理器
- 嵌入式MIPS处理器
  - 12级流水线
  - 指令和数据访问均占用一个时钟周期
- 分离的缓存：指令缓存和数据缓存
  - 每个缓存16KB：256个块×16个字/块
  - D-cache：直写或回写
- SPEC CPU2000测评程序失效率
  - 什么是SPEC CPU2000?
    - The Standard Performance Evaluation Corporation (SPEC)
  - I-cache:0.4%
  - D-cache:11.4%
  - 加权平均：3.2%
  - I-cache和D-cache哪种失效率对CPU性能影响更大？为什么？

# 5.3 Cache基础

- 实例：Intrinsity FastMATH处理器



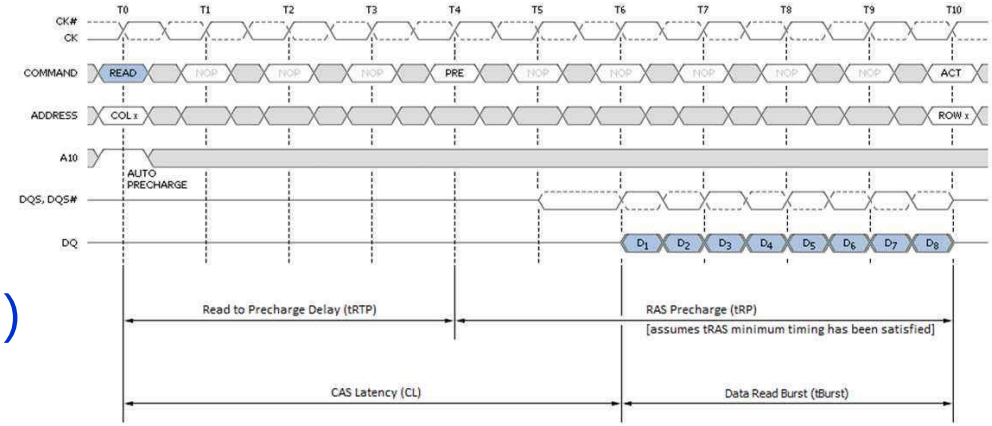
# 5.3 Cache基础

**自我检测** 存储系统的速度影响了设计者对于cache数据块容量的判断。针对cache设计者，下面哪一条准则通常是有效的?

1. 存储延迟越短，cache的数据块容量越小。
2. 存储延迟越短，cache的数据块容量越大。
3. 存储带宽越高，cache的数据块容量越小。
4. 存储带宽越高，cache的数据块容量越大。

# 5.4 Cache的性能评估和改进

- 支持缓存的主存




- 使用DRAM作为主存
  - 固定的数据位宽（例如：32bit）
  - 使用确定位宽的总线进行连接
    - 总线的时钟频率一般都比CPU频率低
- Cache数据块读取示例
  - 发送读地址：1个总线周期
  - DRAM访问：15个总线周期
  - 数据传输：1个总线周期/字
- 对于4word的数据块，1word位宽的DRAM
  - 失效惩罚：1 + 4 * 15 + 4 * 1 = 65个总线周期
  - 带宽：16byte/65总线周期 = 0.25byte/总线周期

# 5.4 Cache的性能评估和改进

- 计算Cache性能
- CPU时间的组成部分
  - 程序执行周期
    - 包括缓存命中时间
  - 内存暂停周期
    - 主要是缓存未命中
- 通过简化假设：

$$\text{Memory stall cycles}$$

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructio ns}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instructio n}} \times \text{Miss penalty}$$

# 5.4 Cache的性能评估和改进

- Cache性能示例
  - I-cache未命中率=2%
  - D-cache未命中率=4%
  - 未命中惩罚=100个时钟周期
  - 基本CPI（理想缓存）=2
  - 加载和存储占指令的36%
- 每个指令的未命中周期
  - I-cache:0.02×100=2
  - D-cache:0.36×0.04×100=1.44
- 实际CPI=2+2+1.44=5.44
  - 理想的CPU速度是实际速度的5.44/2=2.72倍

# 5.4 Cache的性能评估和改进

- 平均访问时间

- 命中时间对性能也很重要
- 平均内存访问时间（AMAT）
  - AMAT=命中时间+未命中率×未命中惩罚
- 实例
  - CPU具有1ns的时钟（1GHz），命中时间=1个周期，未命中惩罚=20个周期，I-cache未命中率=5%
  - AMAT=1+0.05×20=2ns
    - 每个指令2个周期

# 5.4 Cache的性能评估和改进

- 性能总结

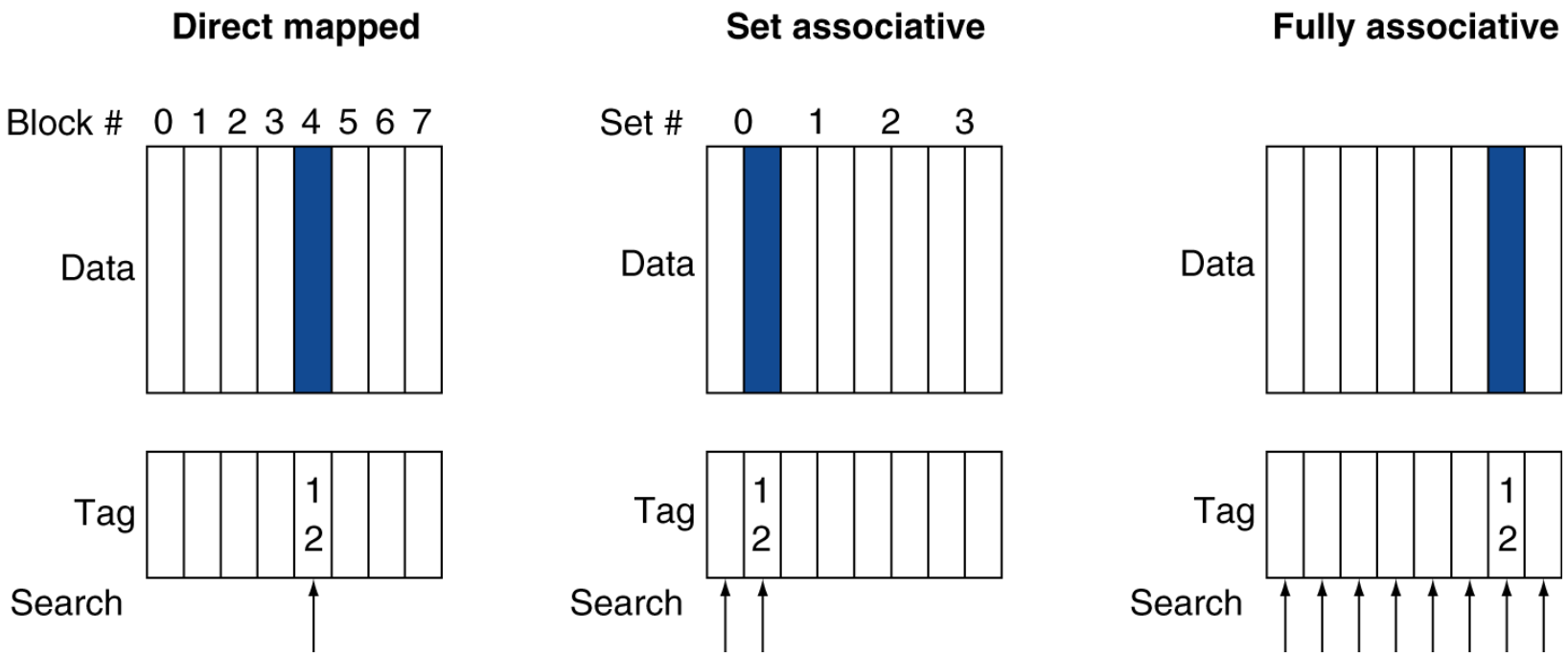
- 当CPU性能提高时
  - 失效惩罚所带来的影响变得更加显著
- 降低基础CPI
  - 花在内存阻塞上的时间比例更大
- 提高时钟频率
  - 内存阻塞会导致占用更多的CPU周期
- 在评估系统性能时不能忽略缓存行为

# 5.4 Cache的性能评估和改进

- 相连Cache
  - 全相连 Fully associative
  - N路组相连 *n*-way set associative
- 全相联
  - 允许给定的块进入任何缓存项
  - 要求立即搜索所有条目
  - 需要对每个条目使用比较器进行比较（昂贵）
- N路组相连
  - 每组包含n个条目
  - 块号决定了存入cache中的哪个集合
  - （块号）模（#缓存中的集合）
  - 只需搜索给定集合中的所有条目
  - n个比较器（更便宜）

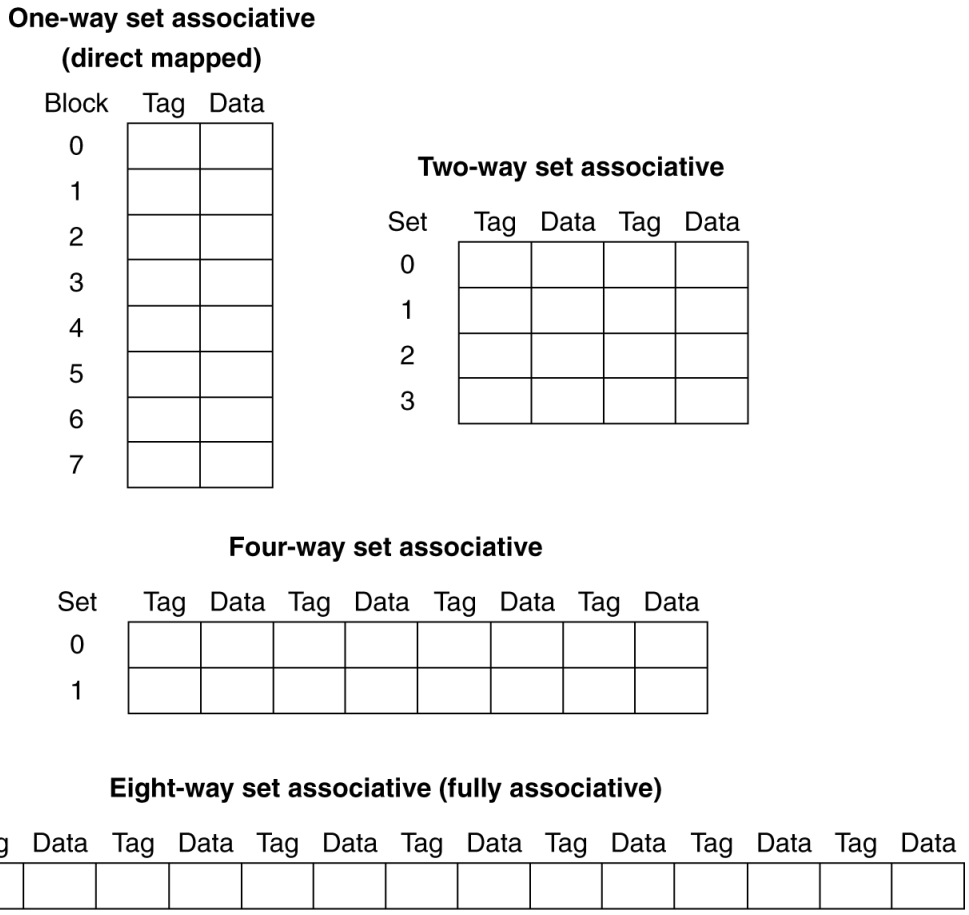
# 5.4 Cache的性能评估和改进

- 相联Cache示例
  - 直接映射
  - 组相联
  - 全相联



# 5.4 Cache的性能评估和改进

- 相联Cache的结构
- 以包含8个条目的Cache为例



# 5.4 Cache的性能评估和改进

- 相连示例

- 比较4块缓存
  - 直接映射，2路组相联，全相联
- 块访问顺序：0,8,0,6,8
- 直接映射

| Block address | Cache index | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 3 |
| 0 | 0 | miss | Mem[0] | | | |
| 8 | 0 | miss | Mem[8] | | | |
| 0 | 0 | miss | Mem[0] | | | |
| 6 | 2 | miss | Mem[0] | | Mem[6] | |
| 8 | 0 | miss | Mem[8] | | Mem[6] | |

# 5.4 Cache的性能评估和改进

- 2路组相联

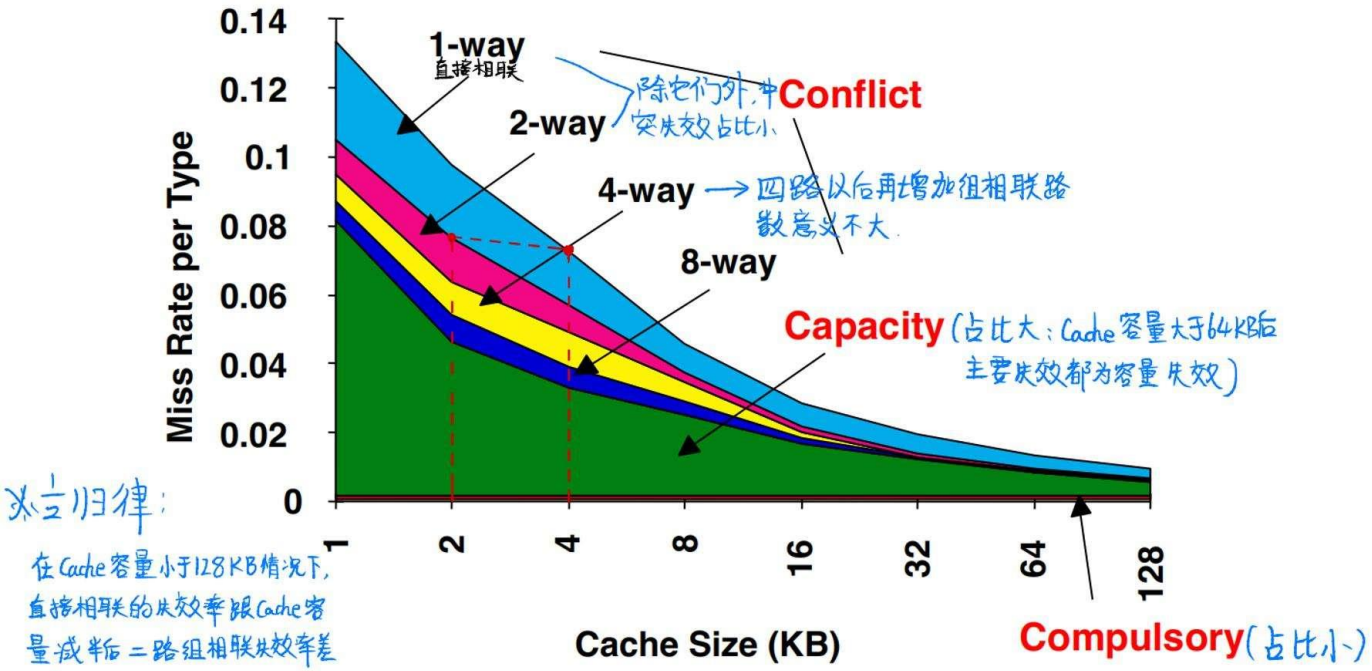
| Block address | Cache index | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| | | | Set 0 | | Set 1 | |
| 0 | 0 | miss | Mem[0] | | | |
| 8 | 0 | miss | Mem[0] | Mem[8] | | |
| 0 | 0 | hit | **Mem[0]** | Mem[8] | | |
| 6 | 0 | miss | Mem[0] | **Mem[6]** | | |
| 8 | 0 | miss | **Mem[8]** | Mem[6] | | |

- 全相联

| Block address | | Hit/miss | Cache content after access | | | |
|---|---|---|---|---|---|---|
| 0 | | miss | Mem[0] | | | |
| 8 | | miss | Mem[0] | Mem[8] | | |
| 0 | | hit | **Mem[0]** | Mem[8] | | |
| 6 | | miss | Mem[0] | Mem[8] | Mem[6] | |
| 8 | | hit | Mem[0] | **Mem[8]** | Mem[6] | |

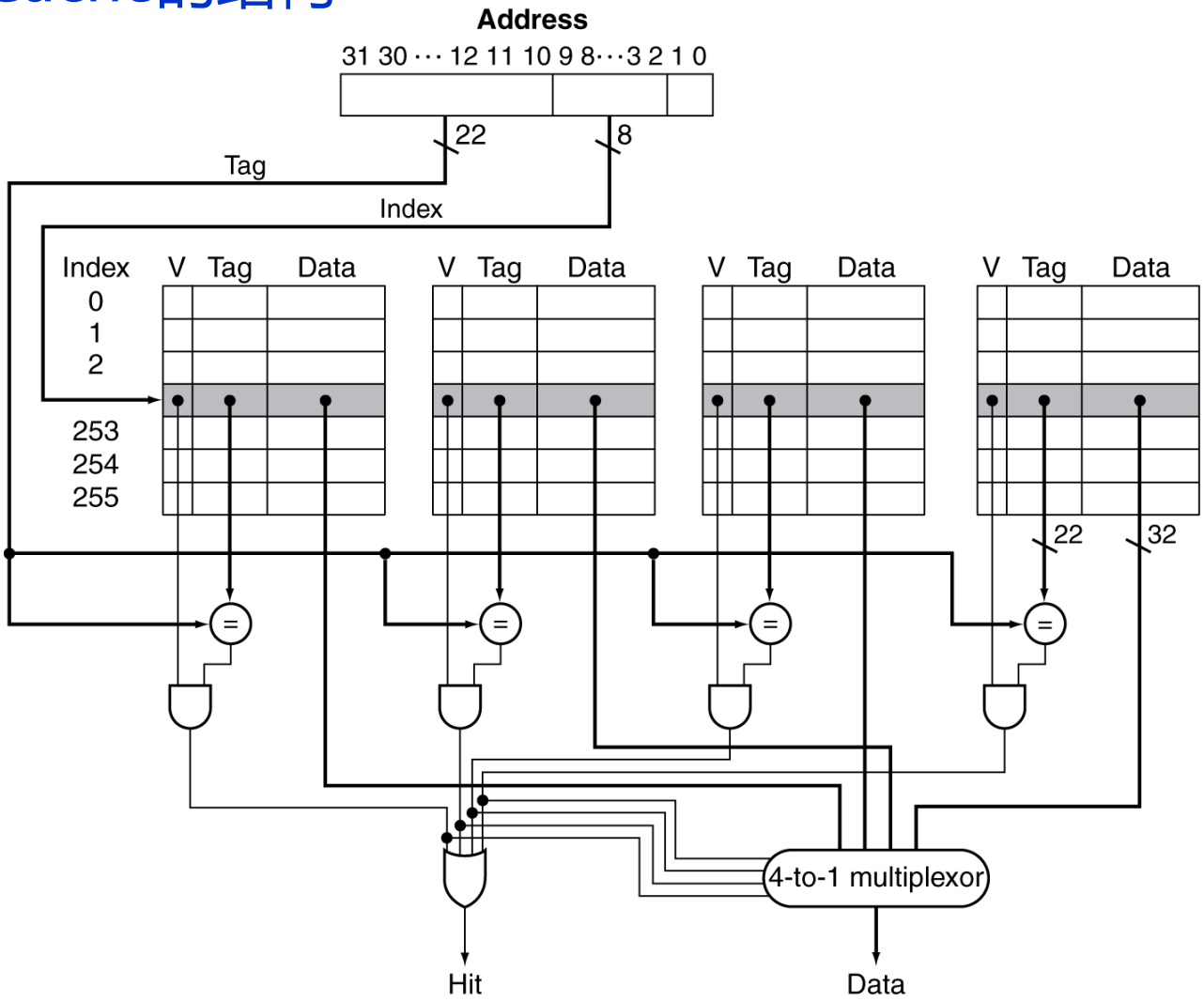
# 5.4 Cache的性能评估和改进

- 如何确定相联程度
- 增加关联性可以降低失效率
  - 但回报却在递减
- 系统模拟：64KB数据缓存、16字/块、SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%
  - 右图仅供参考



# 5.4 Cache的性能评估和改进

- 组相联Cache的结构



# 5.4 Cache的性能评估和改进

- 替换策略

- 直接映射：别无选择
- 组相联
  - 如果有无效条目，请选择无效条目
  - 否则，请在集合中的条目中进行选择
- 最近最少使用（LRU）
  - 选择最长时间未使用的一个
    - 2路简单，4路可控，8路及以上非常困难
- 随机的
  - 在高关联性方面提供与LRU大致相同的性能

# 5.4 Cache的性能评估和改进

- 多级缓存

- 主缓存（L1 cache）连接到CPU
  - 虽小，但速度很快
- L1 cache miss时访问L2 cache
  - 比L1更大、速度更慢，但仍然很快
- 主存储器服务L-2缓存未命中
- 一些高端系统包括L-3缓存

# 5.4 Cache的性能评估和改进

- 多级缓存示例
  - CPU基本CPI=1，时钟频率=4GHz（0.25ns/clock）
  - 未命中率/指令=2%
  - 主存储器访问时间=100ns
- 只有主缓存（L-1）
  - 未命中惩罚=100ns/0.25ns=400个时钟周期
  - 有效CPI=1+0.02×400=9
- 现在添加L-2缓存
  - 访问时间=5ns（20个clock）
  - 主存储器的全局未命中率=0.5%
- L-1失效，L-2命中
  - 惩罚=5ns/0.25ns=20个时钟周期
- L-1失效，L-2失效
  - 额外惩罚=400个时钟周期
- CPI=1+0.02×20+0.005×400=3.4
- 性能比=9/3.4=2.6

# 5.4 Cache的性能评估和改进

- 对于多级缓存的考虑

- 主缓存（L-1）
  - 关注最短的命中时间
- L-2缓存
  - 重点关注低未命中率，以避免主内存访问
  - 命中时间对整体影响较小
- 结果
  - L-1缓存通常比单个缓存小
  - L-1区块大小小于L-2区块大小

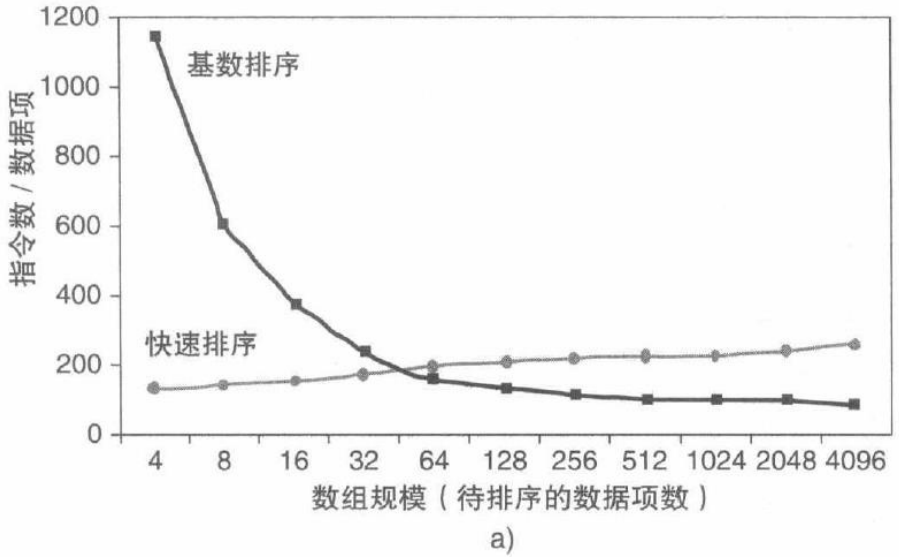
# 5.4 Cache的性能评估和改进

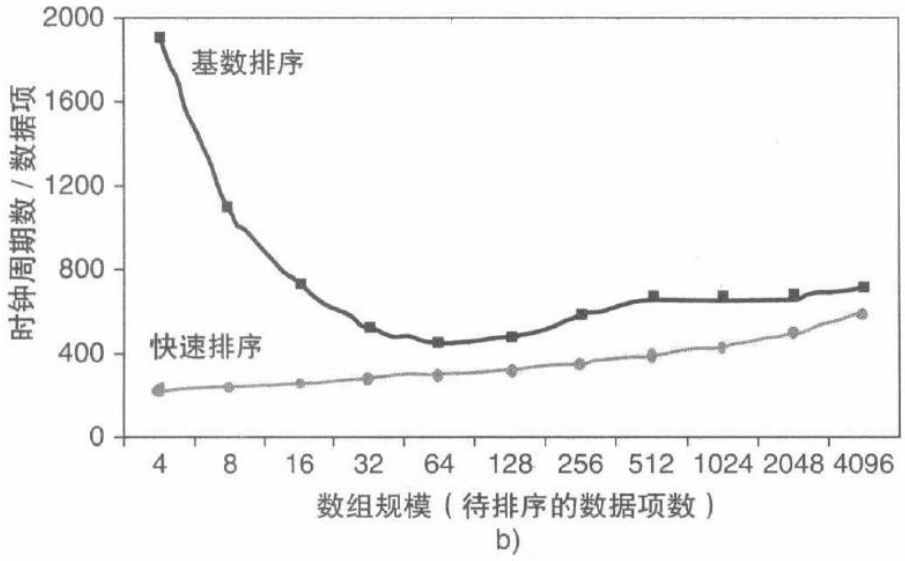
- 高级CPU中对于Cache失效的处理

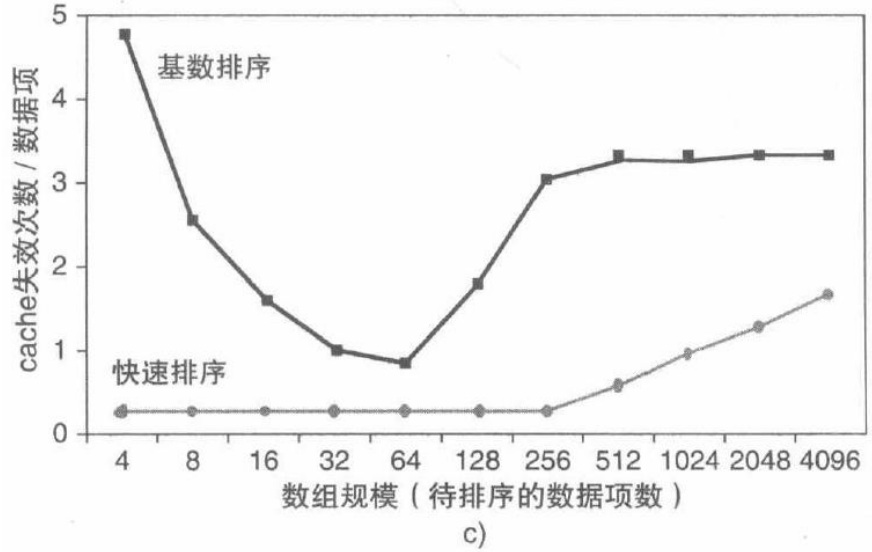
- 支持乱序执行的CPU可以在cache miss时继续执行其它指令
  - 将相关的访存指令保持在load/store单元
  - 与访存相关的指令继续在保留站等待
    - 不相关指令可继续执行
- 未命中的影响取决于程序数据流
  - 更难分析
  - 使用系统模拟

# 5.4 Cache的性能评估和改进

- 对软件的影响
- 未命中取决于内存访问模式
  - 算法行为
  - 内存访问的编译器优化



# 5.4 Cache的性能评估和改进

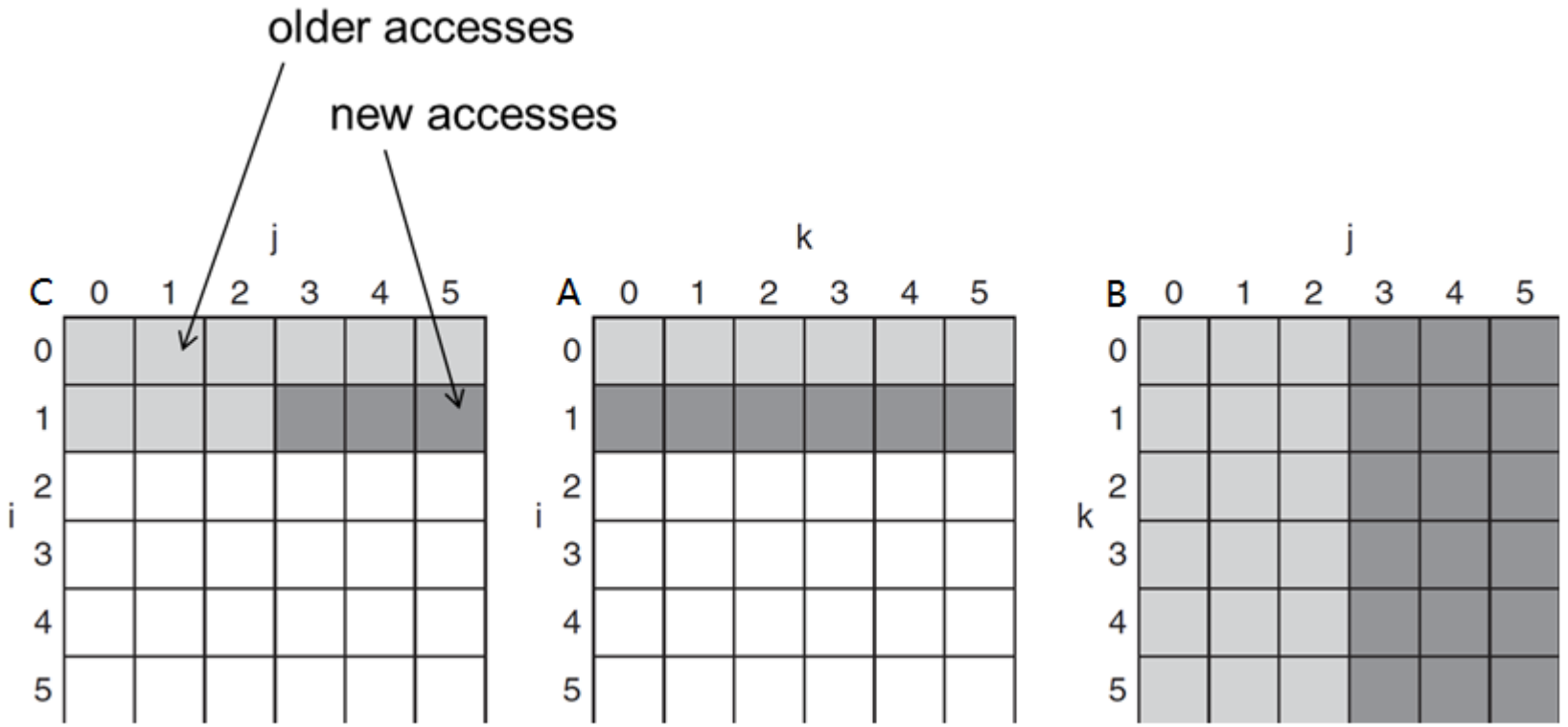
- 通过分块进行软件优化

- 目标：在数据被替换之前最大限度地访问数据
- 考虑DGEMM的内部循环：

```
for (int j = 0; j < n; ++j)
  {
    double cij = C[i+j*n];
    for( int k = 0; k < n; k++ )
      cij += A[i+k*n] * B[k+j*n];
    C[i+j*n] = cij;
  }
```

# 5.4 Cache的性能评估和改进

- DGEMM访存模式
  - 数组C、A、B的缓存模式



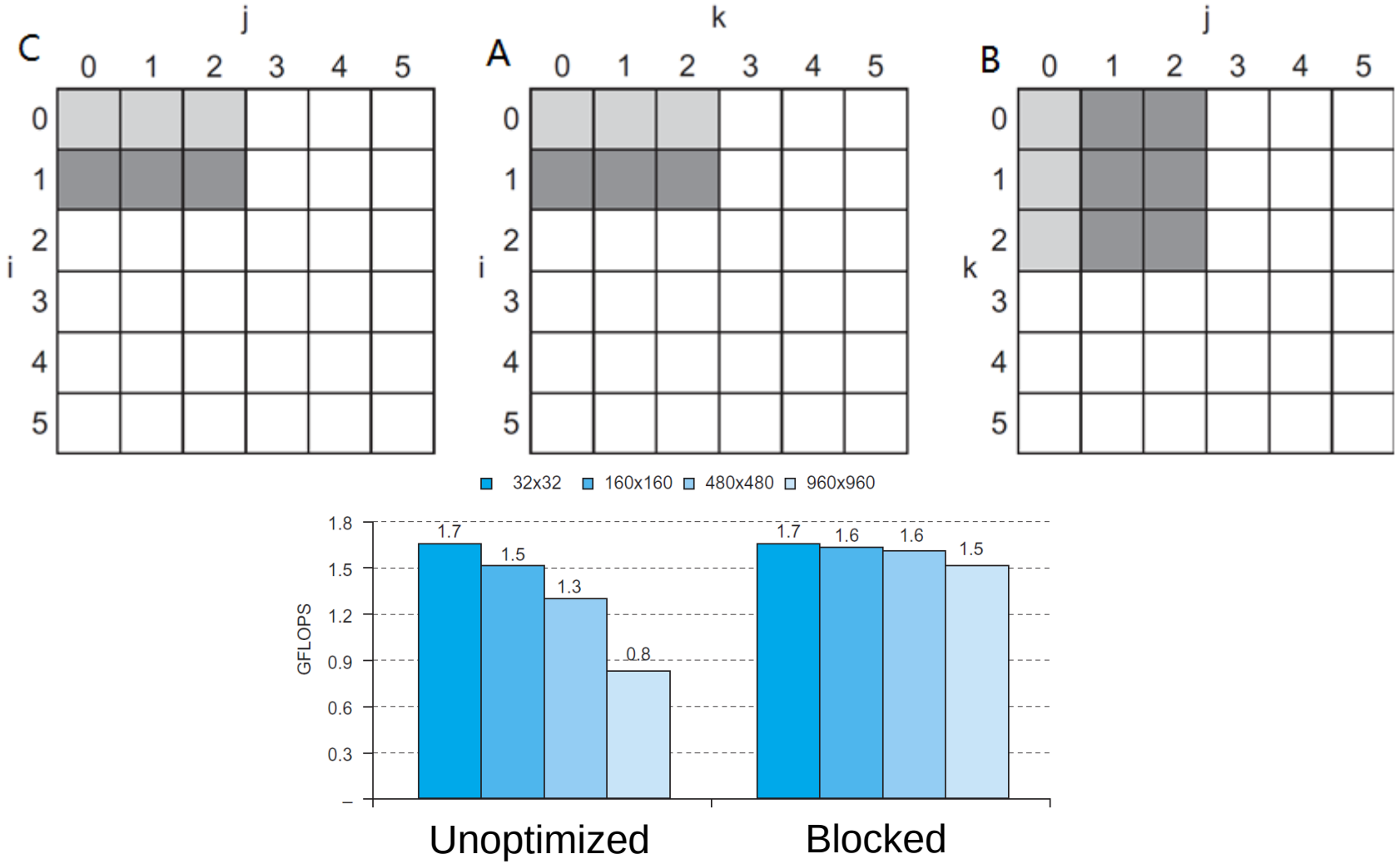
# 5.4 Cache的性能评估和改进

- DGEMM的cache分块版本

```
1 #define BLOCKSIZE 32
2 void do_block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
5  for (int i = si; i < si+BLOCKSIZE; ++i)
6   for (int j = sj; j < sj+BLOCKSIZE; ++j)
7   {
8    double cij = C[i+j*n];/* cij = C[i][j] */
9    for( int k = sk; k < sk+BLOCKSIZE; k++ )
10    cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11   C[i+j*n] = cij;/* C[i][j] = cij */
12  }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16  for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17   for ( int si = 0; si < n; si += BLOCKSIZE )
18    for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19     do_block(n, si, sj, sk, A, B, C);
20 }
```

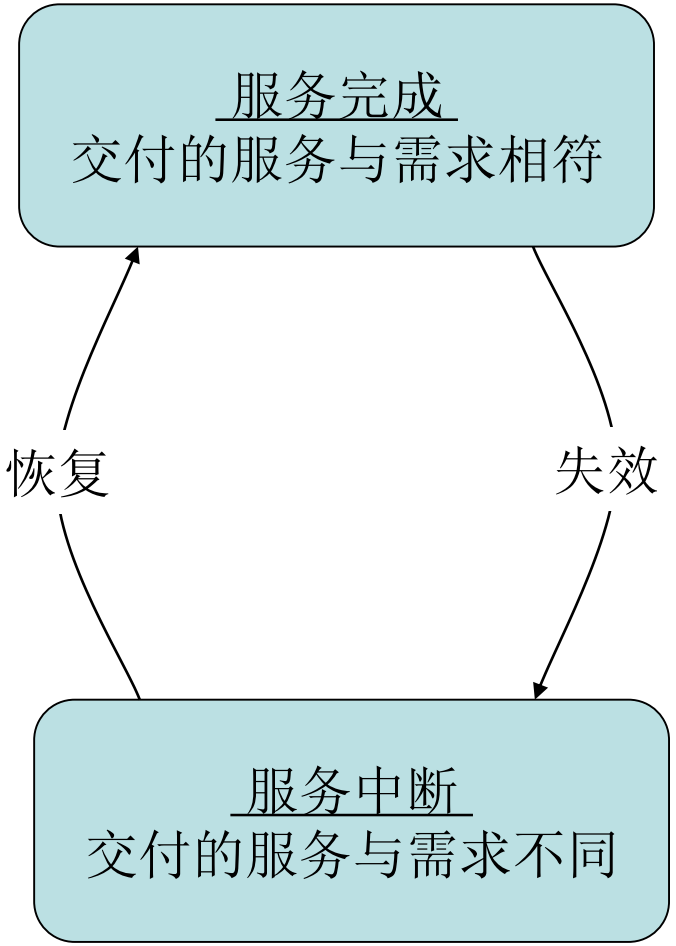
# 5.4 Cache的性能评估和改进

- DGEMM的cache分块访存模式



# 5.5 可靠的存储器层次

- 可靠性
  - 是一个系统能够持续提供用户需求的服务的度量，即从参考时刻到失效的时间间隔
  - 度量参数为平均无故障时间，MTTF，Mean Time To Failures



# 5.5 可靠的存储器层次

- 可靠性度量

- 可靠性：平均失效时间（MTTF）
- 服务中断：平均维修时间（MTTR）
- 平均失效间隔
  - MTBF=MTTF+MTTR
- 可用性=MTTF/（MTTF+MTTR）
- 提高可用性
  - 提高MTTF：故障避免、容错、故障预测
  - 减少MTTR：改进诊断和维修工具和流程

# 5.5 可靠的存储器层次

- 汉明码
  - SEC码：Single Error Correcting Code
  - DED码：Double Error Detecting Code
- 汉明距离
  - 两个位模式之间不同的位数
- 最小距离等于2时可以提供单位错误检测
  - 例如奇偶校验码
- 最小距离等于3时提供单错误校正，和2位错误检测

# 5.5 可靠的存储器层次

- 汉明SEC编码

- 为了计算汉明编码
  - 从bit1开始，从左向右进行编号
  - 所有2的整数次幂的位置用来放置校验位
  - 每个校验位对特定的数据位进行校验

| Bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded date bits | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 |
| Parity bit coverate | p1 | X | | X | | X | | X | | X | | X | |
| | p2 | | X | X | | | X | X | | | X | X | |
| | p4 | | | | X | X | X | X | | | | | X |
| | p8 | | | | | | | | X | X | X | X | X |

# 5.5 可靠的存储器层次

- 汉明SEC解码
  - 所有校验位所构成的数据的数值代表了出错的bit位
  - 使用编码过程中的编号
  - 例如：
    - 校验值=0000，表明没有出错
    - 校验值=1010，表明bit10发生了翻转

# 5.5 可靠的存储器层次

- SEC/DEC编码
- 为整个字添加额外的奇偶校验位-$p_n$
- 将汉明距离设为4
- 解码：
  - 设H=SEC校验位数
    - H偶数，$p_n$偶数，无错误
    - H奇数、pn奇数、可纠正的单bit错误
    - H偶数，pn奇数，pn位错误
    - 出现H奇数、pn偶数、双错误
- 注：ECC DRAM使用SEC/DEC，64位数据，8位校验

# 5.6 虚拟机

- 主机模拟访客操作系统和机器资源
  - 改进了对多个访客账户的隔离
  - 避免安全性和可靠性问题
  - 目标在于共享资源
- 虚拟化对性能有一定影响
  - 适用于现代高性能计算机
- 例子
  - IBM VM/370（1970年代的技术！）
  - VMWare
  - 微软虚拟机

# 5.6 虚拟机

- 虚拟机监视器（VMM）

- 将虚拟资源映射到物理资源
  - 内存、I/O设备、CPU
- 访客代码以用户模式在本机上运行
  - 根据特权指令和对受保护资源的访问设置VMM陷阱
- 访客操作系统可能不同于主机操作系统
- VMM处理真实的I/O设备
  - 为访客模拟通用虚拟I/O设备

# 5.6 虚拟机

- 示例：定时器虚拟化

- 在本地机器中，定时中断发生时
  - 操作系统暂停当前进程，处理中断，选择并恢复下一个进程
- 使用虚拟机监视器
  - VMM挂起当前VM，处理中断，选择并恢复下一个VM
- 如果虚拟机需要定时器中断
  - VMM模拟虚拟计时器
  - 当物理计时器中断发生时，模拟VM的中断

# 5.6 虚拟机

- ISA支持

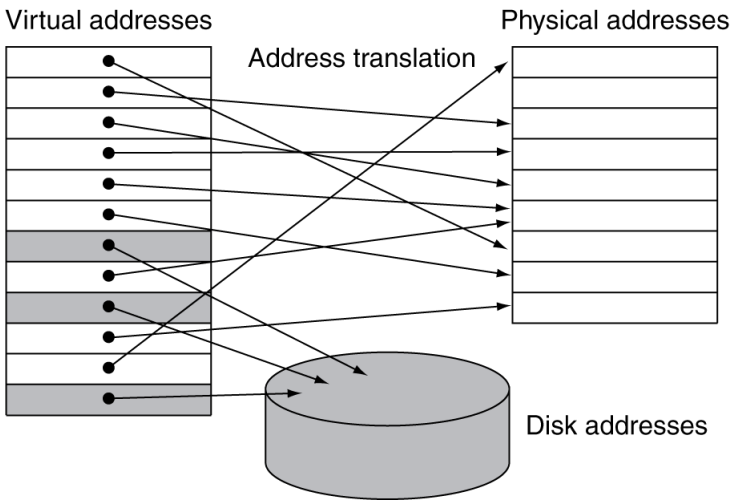
- 支持用户模式和系统模式
- 特权指令仅在系统模式下可用
  - 如果在用户模式下执行，将陷入到系统模式
- 所有物理资源只能使用特权指令访问
  - 包括页表、中断控制、I/O寄存器
- 虚拟化支持的复兴
  - 当前的ISA（如x86）正在调整

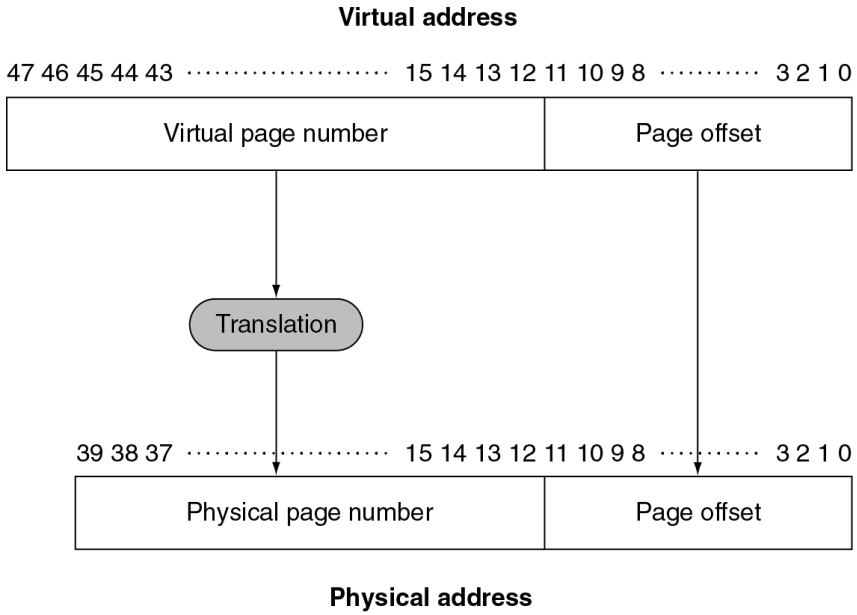
# 5.7 虚拟存储

- 虚拟存储
  - 一种将主存看作辅助存储的cache的技术
  - 由CPU硬件和操作系统（OS）共同管理
  - VM中的“块”称为页
  - VM中将“未命中”称为缺页失效
- 程序共享主存
  - 每个程序都有一个私有的虚拟地址空间，其中包含其常用的代码和数据
  - 不受其他程序的影响
- CPU和操作系统将虚拟地址转换为物理地址
  - 虚拟地址：虚拟空间的地址，当访问内存时，需要通过地址映射转换为物理地址
  - 物理地址：主存的地址
  - 地址转换：也称地址映射，将虚拟地址转换为物理地址的过程

# 5.7 虚拟存储

- 地址转换
  - 页大小固定，一般为4K



# 5.7 虚拟存储

- 缺页失效代价

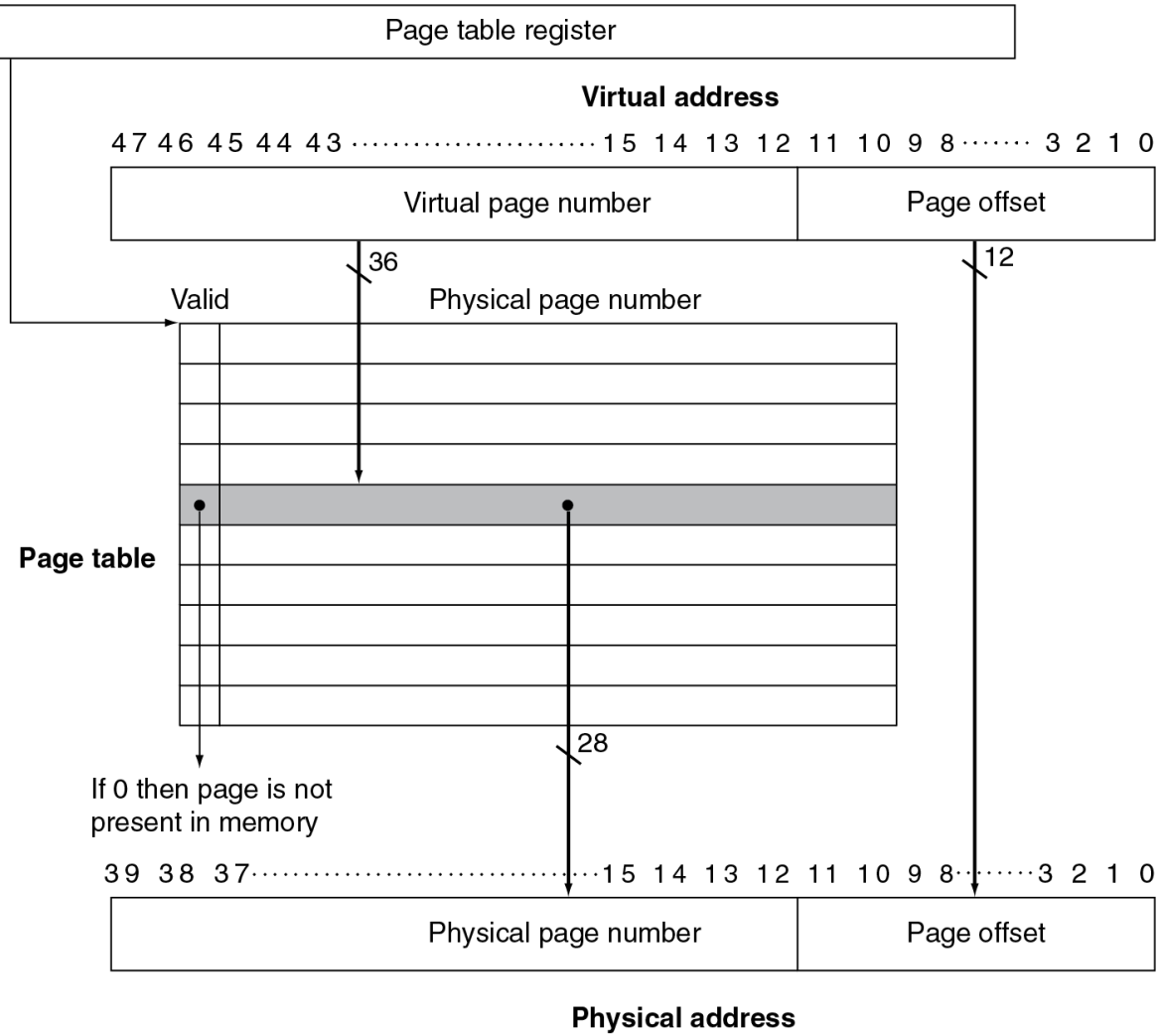
- 缺页失效时，必须从磁盘中提取页面
  - （对于机械硬盘来说）需要数百万个时钟周期
  - 由操作系统代码处理
- 尽量减少缺页失效的比率
  - 全相联的映射方式
  - 智能替换算法

# 5.7 虚拟存储

- 页表
  - 在虚拟存储系统中，保存着虚拟地址和物理地址之间转换关系的表
  - 页表保存在内存中，通常使用虚拟页号来索引
  - 如果这个页在内存中，页表中的对应项包含该页对应的物理页号
- 页表用来存储映射信息
  - 按虚拟页码索引的页表项数组
  - CPU中的页表寄存器指向物理内存中的页表
- 如果页在内存中
  - PTE存储物理页码
    - page table entry页表条目
  - 加上其他状态位（引用、脏等）
- 如果页不在内存中
  - 页表条目可以指磁盘上交换空间中的位置

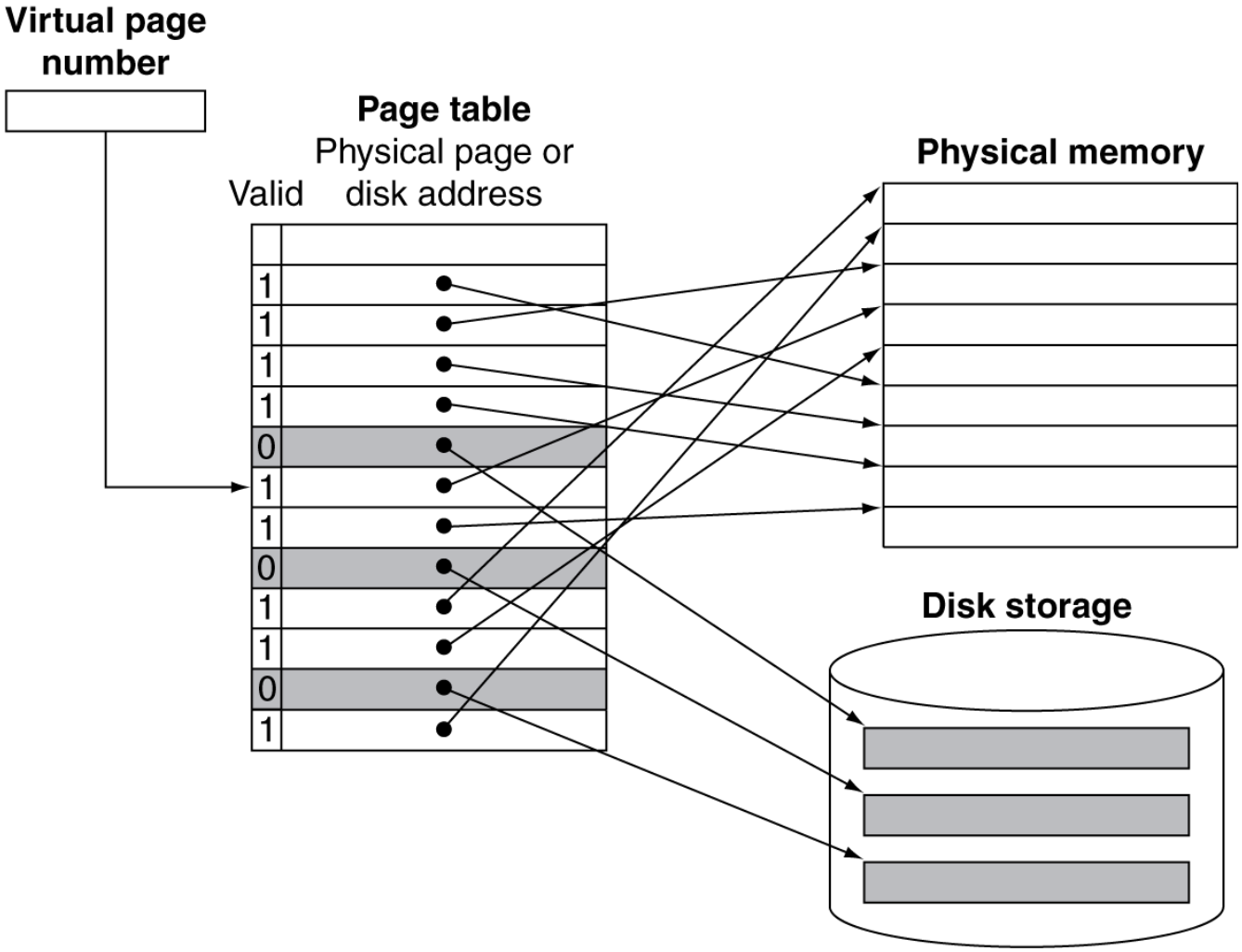
# 5.7 虚拟存储

- 使用页表进行地址转换



# 5.7 虚拟存储

- 将页表映射到磁盘



# 5.7 虚拟存储

- 替换和写操作

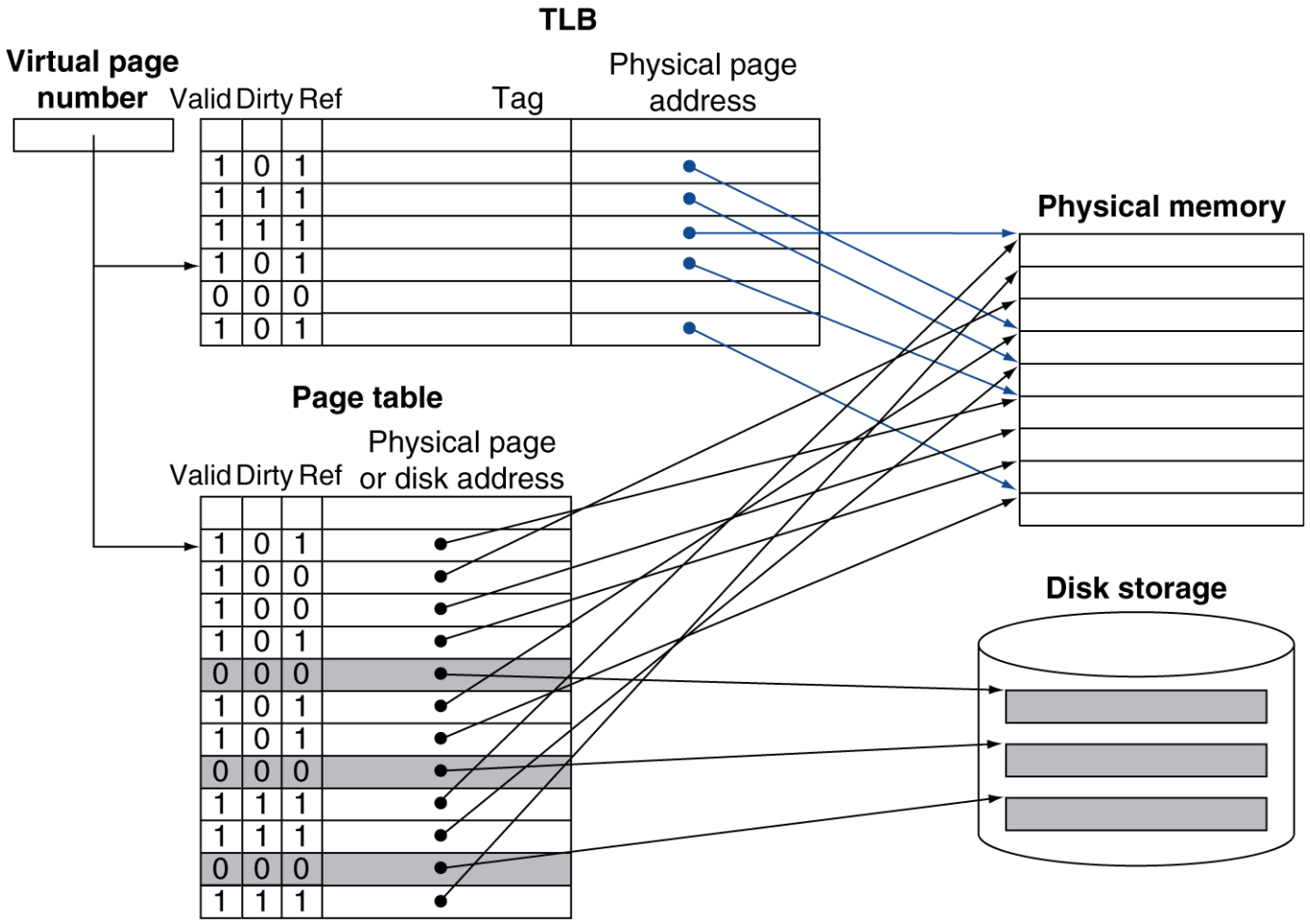
- 要降低缺页失效比率，需要选择最近最少使用（LRU）的页进行替换
  - 访问页时，PTE中的参考位（也称为使用位）设置为1
  - 操作系统定期将其清除为0
  - 最近未使用指的是引用位为0的页面
- 磁盘写入需要数百万个周期
  - 因此需要对整块进行操作，而不是每次访问都进行定位
  - 写直达（或称写穿透）是不现实的
  - 需要使用回写策略
  - 页内发生写操作时，设置PTE中的脏位为1

# 5.7 虚拟存储

- 使用快表（TLB）进行快速转换
  - TLB： Translation-Lookaside Buffer，用于记录最近使用地址的映射信息的cache，从而避免每次都要访问页表
  - 为什么需要引入快表?
- 地址转换需要额外的内存访问
  - 第一次访问PTE，获取物理地址
  - 第二次才是实际的内存访问
- 但对页表的访问具有良好的局部性
  - 因此，在CPU中使用快速缓存PTE
  - 典型：16-512次PTE，命中时耗时0.5-1个周期，未命中时耗时10-100个周期，未命中率为0.01%-1%
  - 未命中时可以通过硬件或软件处理

# 5.7 虚拟存储

- 使用快表（TLB）进行快速转换



# 5.7 虚拟存储

- TLB miss
  - 页在内存中
  - 页不在内存中
- 如果页面在内存中
  - 从内存中加载PTE，然后重试
  - 可以用硬件处理
    - 对于更复杂的页表结构，可能会变得复杂
  - 或者通过软件处理
    - 使用优化的处理程序，引发特殊异常
- 如果页面不在内存中（缺页失效）
  - 操作系统负责获取页面和更新页面表
  - 然后重新启动缺失指令

# 5.7 虚拟存储

- TLB miss 处理函数

- TLB miss表明
  - 页面存在，但PTE不在TLB中
  - 页面不存在
- 必须在重新目标寄存器之前识别TLB未命中
  - 引发异常
- 处理程序将PTE从内存复制到TLB
  - 然后重新启动指令
  - 如果页面不存在，则会出现缺页失效异常

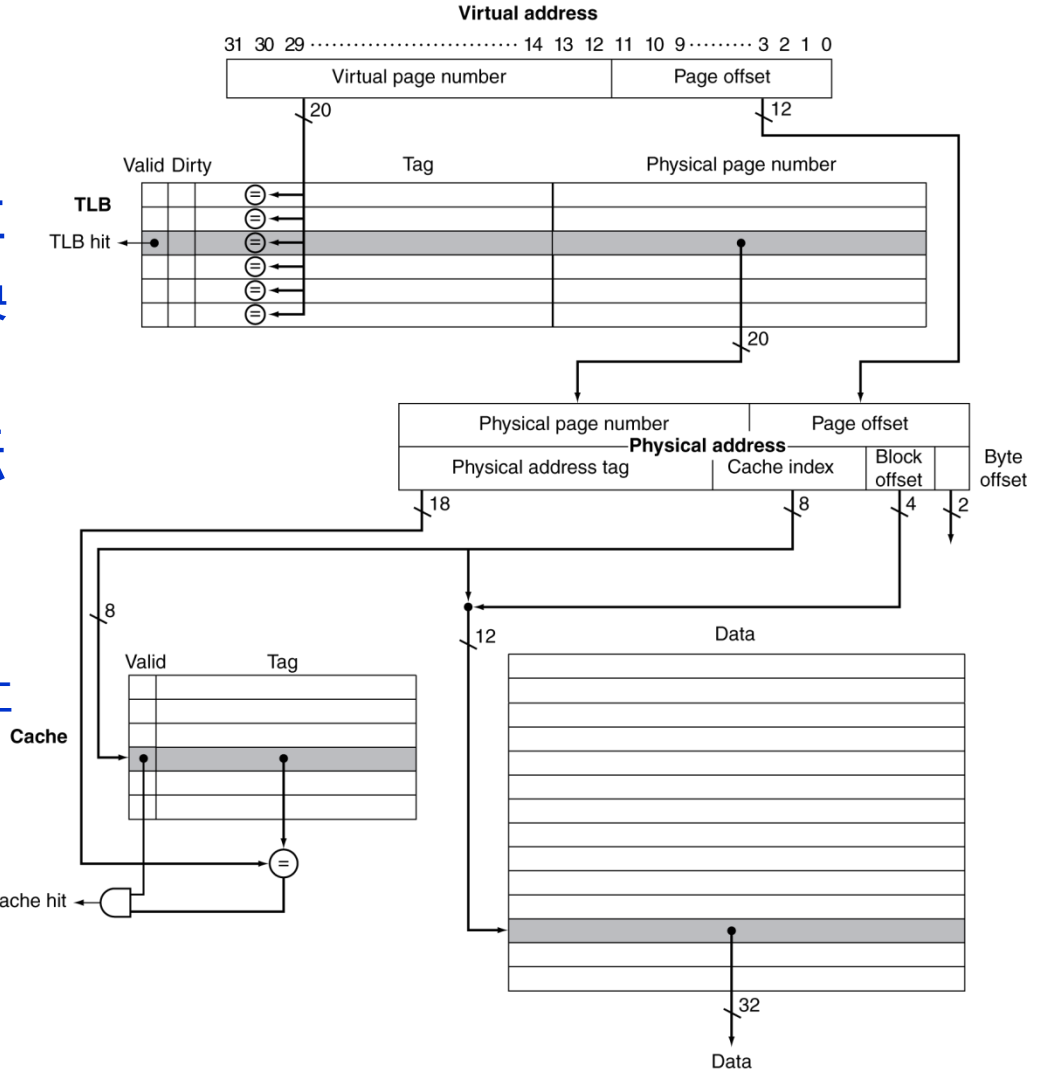
# 5.7 虚拟存储

- TLB miss 处理函数

- 使用缺失的虚拟地址查找PTE
- 在磁盘上找到页面
- 选择要替换的页面
  - 如果脏了，首先写入磁盘
- 将页面读入内存并更新页面表
- 使进程再次运行
  - 从缺失指令重新启动

# 5.7 虚拟存储

- TLB 和 Cache 交互
- 如果缓存标记使用物理地址
  - 需要在缓存查找之前进行转换
- 备选方案：使用虚拟地址标签
  - 别名引起的并发症
    - 共享物理地址的不同虚拟地址



# 5.7 虚拟存储

- 虚拟存储中的保护
- 不同的任务可以共享部分虚拟地址空间
  - 但需要防止错误访问
  - 需要操作系统协助
- 对操作系统保护的硬件支持
  - 特权管理模式（又称内核模式）
  - 特权指令
  - 页面表格和其他状态信息只能在管理器模式下访问
  - 系统调用异常（例如RISC-V中的ecall）
- 管理模式
  - 也称内核模式，是一种运行操作系统进程的模式
- 系统调用
  - 将控制权从用户模式转换到管理模式的特殊指令，触发进程中的一个例外机制

# 5.8 存储器层次结构的一般框架

- 通用原则适用于内存层次结构的所有级别
  - 基于缓存的概念
- 在层次结构中的每个级别都需要考虑如下问题
  - 块放置
  - 找到一个block
  - 未命中时的替换策略
  - 写策略

# 5.8 存储器层次结构的一般框架

- 块放置
  - 主要取决于相联性
- 直接映射（1路组相联）
  - 每个数据块只有一个对应位置可以放置
- N路组相联
  - 每个数据块有N个对应位置可以放置
  - N个为一组（set）
- 全相联
  - 数据块可放置在任意位置

- 增加相联度可以提高命中率
  - 相应的，会增加复杂性、成本和访问时间

# 5.8 存储器层次结构的一般框架

- 硬件缓存
  - 一般使用N路组相联方式
  - 减少比较器，从而降低成本和复杂度
- 虚拟存储
  - 一般采用全相联方式
  - 条目少，因此全相联方式可行
  - 可以提高命中率

| 相联方式 | 定位方式 | 标签比较器数量 |
| --- | --- | --- |
| 直接映射 | 索引 | 1 |
| N路组相联 | 索引组,查找组中元素 | N |
| 全相联 | 查找所有的cache表项 | 等于所有条目的数量 |
| | 独立的查找表 | 0 |

# 5.8 存储器层次结构的一般框架

- 替换策略
- 在未命中时选择要替换的条目
  - 最近最少使用（LRU）
    - 复杂且昂贵的硬件可实现高关联性
    - 一般实现的是LRU的近似形式
  - 随机选择
    - 性能接近LRU，更容易实施
- 对于虚拟存储
  - 失效代价很高
  - 发生频率较低
  - 经常使用软件实现近似的LRU算法

# 5.8 存储器层次结构的一般框架

- 写策略

- 写直达策略
  - 更新上层和下层
  - 简化了替换流程，但可能需要写缓冲区
- 写返回策略
  - 仅更新上层
  - 替换块时更新较低级别
  - 需要保存更多的状态

- 对于虚拟存储
  - 考虑到磁盘写入延迟，只有回写是可行的

# 5.8 存储器层次结构的一般框架

- 失效分类
  - 3C模型：将所有的cache失效都归为三种类型的cache模型，分别为强制失效（Compulsory misses）、容量失效（Capacity misses）、冲突失效（Conflict misses）
  - 强制失效，也称为冷启动失效
    - 对没有在cache中出现过的块进行第一次访问是产生的失效
  - 容量失效
    - 由于cache在全相联时都不可能容纳所有请求的块而导致的失效
    - 例如，一个被替换掉的块又被再次访问
  - 冲突失效，也称为碰撞失效
    - 在组相连或者直接映射cache中，很多块为了竞争同一个组导致的失效，这种失效在使用相同大小的全相联cache中是不存在的。

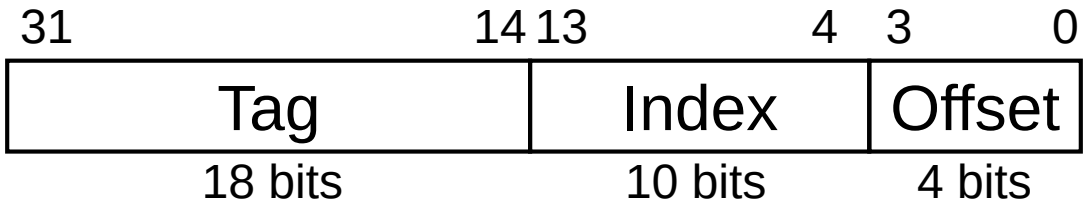
# 5.8 存储器层次结构的一般框架

- cache设计中的权衡

| 设计变化 | 对失效率的影响 | 可能对性能产生的负面影响 |
| --- | --- | --- |
| 增加cache容量 | 降低失效率 | 可能延长访问时间 |
| 增加相联度 | 由于减少了冲突失效，降低了失效率 | 可能延长访问时间 |
| 增加块容量 | 由于空间局部性，对很宽范围内变化的块大小，降低了失效率 | 增加失效损失，块太大还会增大失效率 |

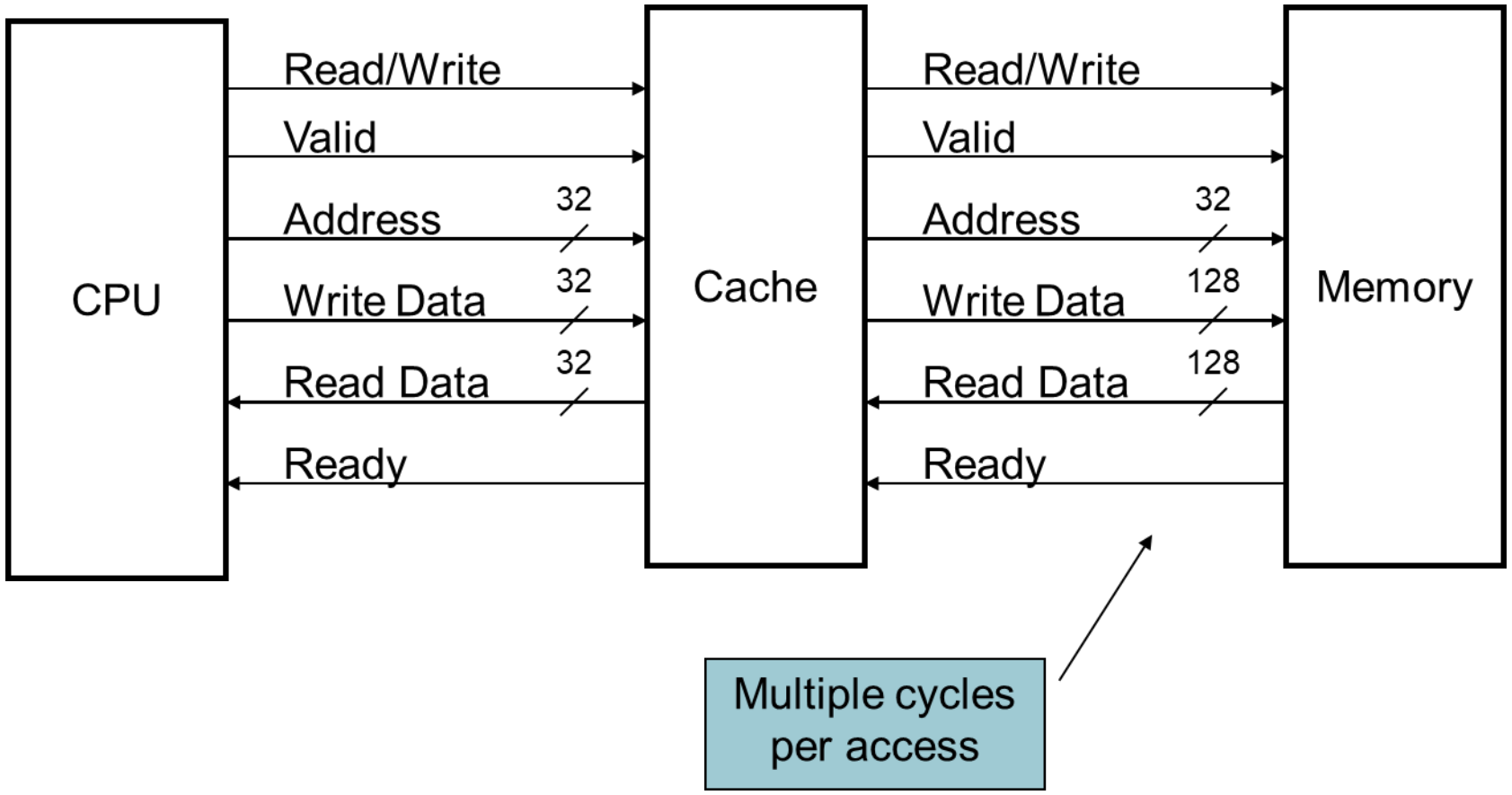
# 5.9 使用有限状态机控制简单的cache

- 缓存特性
  - 直接映射、写回、写分配
  - 块大小：4个字（16字节）
  - 缓存大小：16 KB（1024个块）
  - 32位字节地址
  - 每个块有效位和脏位各1bit
  - 阻塞缓存
    - CPU等待访问完成
- 计算相关字段位宽
  - cache块索引：10bit
  - 块内偏移：4bit
  - 标签位宽：32-10-4 = 18bit



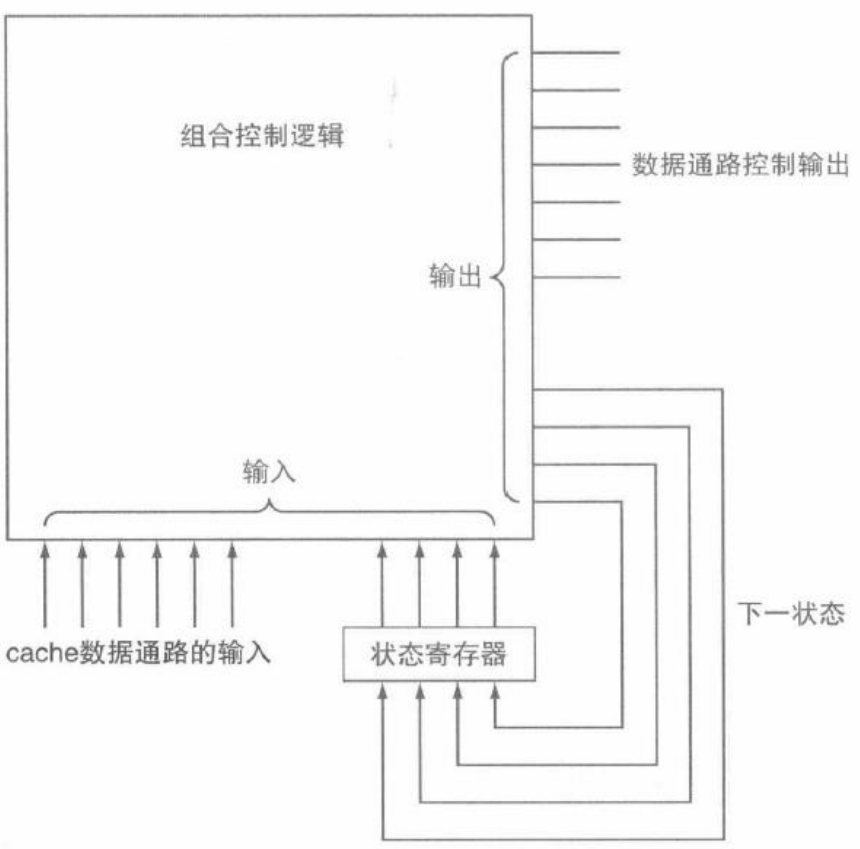
# 5.9 使用有限状态机控制简单的cache

- 接口信号



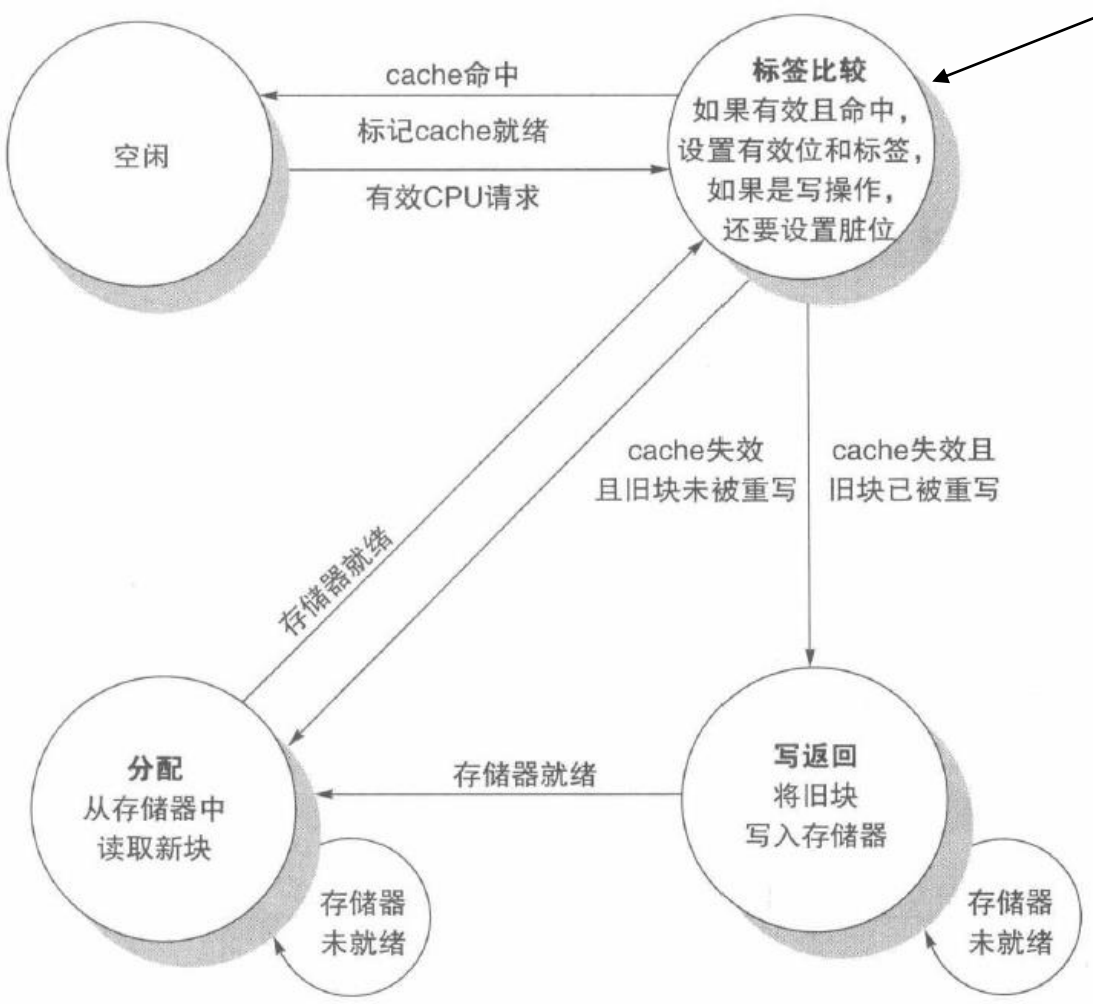
# 5.9 使用有限状态机控制简单的cache

- 使用FSM对控制步骤进行排序
- 状态集，每个时钟边缘上的转换
  - 状态值是二进制编码的
  - 存储在寄存器中的当前状态
  - 下一状态
    = fn（当前状态，当前输入）
- 控制输出信号
  = fo（当前状态）



# 5.9 使用有限状态机控制简单的cache

- cache控制器有限状态机




可以划分为不同的状态以减少时钟周期时间

# 5.10 并行和存储层次结构：cache一致性

- 缓存一致性问题
- 假设两个CPU核共享一个物理地址空间
  - 采用写直达策略

| 时间 | 事件 | CPU A cache内容 | CPU B cache内容 | 内存位置X的内容 |
|---|---|---|---|---|
| 0 | | | | 0 |
| 1 | CPU A读X | 0 | | 0 |
| 2 | CPU B读X | 0 | 0 | 0 |
| 3 | CPU A向X写入1 | 1 | 0 | 1 |

# 5.10 并行和存储层次结构：cache一致性

- 一致性定义
- 非正式地：读取返回最近写入的值
- 正式地：
  - P写X；P读取X（无中间写入）→ 读取时返回写入的值
  - P1写X；P2读X（足够晚） → 读取时返回写入的值
    - 例如在前面示例的步骤3之后，CPU B读取X
  - P1写X，P2写X　所有处理器都以相同的顺序看到写入操作
    - 最终读取到相同的X最终值

# 5.10 并行和存储层次结构：cache一致性

- cache一致性方案

- 由多处理器中的缓存执行的操作，以确保一致性
  - 将数据迁移到本地缓存
    - 减少共享内存的带宽
  - 读取共享数据的复制
    - 减少访问争用
- 监听协议
  - 每个缓存都监视总线读/写
- 基于目录的协议
  - 目录中块的缓存和内存记录共享状态

# 5.10 并行和存储层次结构：cache一致性

- 写无效协议
- 缓存在写入块时以独占方式访问该块
  - 在总线上广播失效消息
  - 另一个缓存中的后续读取未命中
    - 拥有缓存提供了更新的价值

| 处理器行为 | 总线行为 | CPU A cache内容 | CPU B cache内容 | 存储器位置X的内容 |
|---|---|---|---|---|
| | | | | 0 |
| CPU A读X | X在cache中失效 | 0 | | 0 |
| CPU B读X | X在cache中失效 | 0 | 0 | 0 |
| CPU A向X写入1 | 令X无效 | 1 | | 0 |
| CPU B读X | X在cache中失效 | 1 | 1 | 1 |

# 5.10 并行和存储层次结构：cache一致性

- 内存一致性
- 其他处理器何时可以看到写操作
  - “Seen”表示读取返回写入的值
  - 不可能是瞬间的
- 假设
  - 只有当所有处理器都看到写入时，写入才会完成
  - 处理器不会对其他访问的写入进行重新排序
- 结果
  - P写X，然后写Y ⇒ 所有看到新Y的处理器也会看到新X
  - 处理器可以对读取进行重新排序，但不能对写入进行重新排序

# 5.11 并行与存储层次结构：RAID

- 略

# 5.12 高级专题：实现缓存控制器

- 略

# 5.13 实例

- 多级片上cache

| Characteristic | ARM Cortex-A53 | Intel Core i7 |
|---|---|---|
| L1 cache organization | Split instruction and data caches | Split instruction and data caches |
| L1 cache size | Configurable 16 to 64 KiB each for instructions/data | 32 KiB each for instructions/data per core |
| L1 cache associativity | Two-way (I), four-way (D) set associative | Four-way (I), eight-way (D) set associative |
| L1 replacement | Random | Approximated LRU |
| L1 block size | 64 bytes | 64 bytes |
| L1 write policy | Write-back, variable allocation policies (default is Write-allocate) | Write-back, No-write-allocate |
| L1 hit time (load-use) | Two clock cycles | Four clock cycles, pipelined |
| L2 cache organization | Unified (instruction and data) | Unified (instruction and data) per core |
| L2 cache size | 128 KiB to 2 MiB | 256 KiB (0.25 MiB) |
| L2 cache associativity | 16-way set associative | 8-way set associative |
| L2 replacement | Approximated LRU | Approximated LRU |
| L2 block size | 64 bytes | 64 bytes |
| L2 write policy | Write-back, Write-allocate | Write-back, Write-allocate |
| L2 hit time | 12 clock cycles | 10 clock cycles |
| L3 cache organization | – | Unified (instruction and data) |
| L3 cache size | – | 8 MiB, shared |
| L3 cache associativity | – | 16-way set associative |
| L3 replacement | – | Approximated LRU |
| L3 block size | – | 64 bytes |
| L3 write policy | – | Write-back, Write-allocate |
| L3 hit time | – | 35 clock cycles |

# 5.13 实例

- 两级TLB结构

| Characteristic | ARM Cortex-A53 | Intel Core i7 |
|---|---|---|
| Virtual address | 48 bits | 48 bits |
| Physical address | 40 bits | 44 bits |
| Page size | Variable: 4, 16, 64 KiB, 1, 2 MiB, 1 GiB | Variable: 4 KiB, 2/4 MiB |
| TLB organization | 1 TLB for instructions and 1 TLB for data per core<br><br>Both micro TLBs are fully associative, with 10 entries, round robin replacement<br>64-entry, four-way set-associative TLBs<br><br>TLB misses handled in hardware | 1 TLB for instructions and 1 TLB for data per core<br><br>Both L1 TLBs are four-way set associative, LRU replacement<br><br>L1 I-TLB has 128 entries for small pages, seven per thread for large pages<br><br>L1 D-TLB has 64 entries for small pages, 32 for large pages<br><br>The L2 TLB is four-way set associative, LRU replacement<br><br>The L2 TLB has 512 entries<br><br>TLB misses handled in hardware |

# 5.13 实例

- Cortex-A53和Core i7对多发射的支持

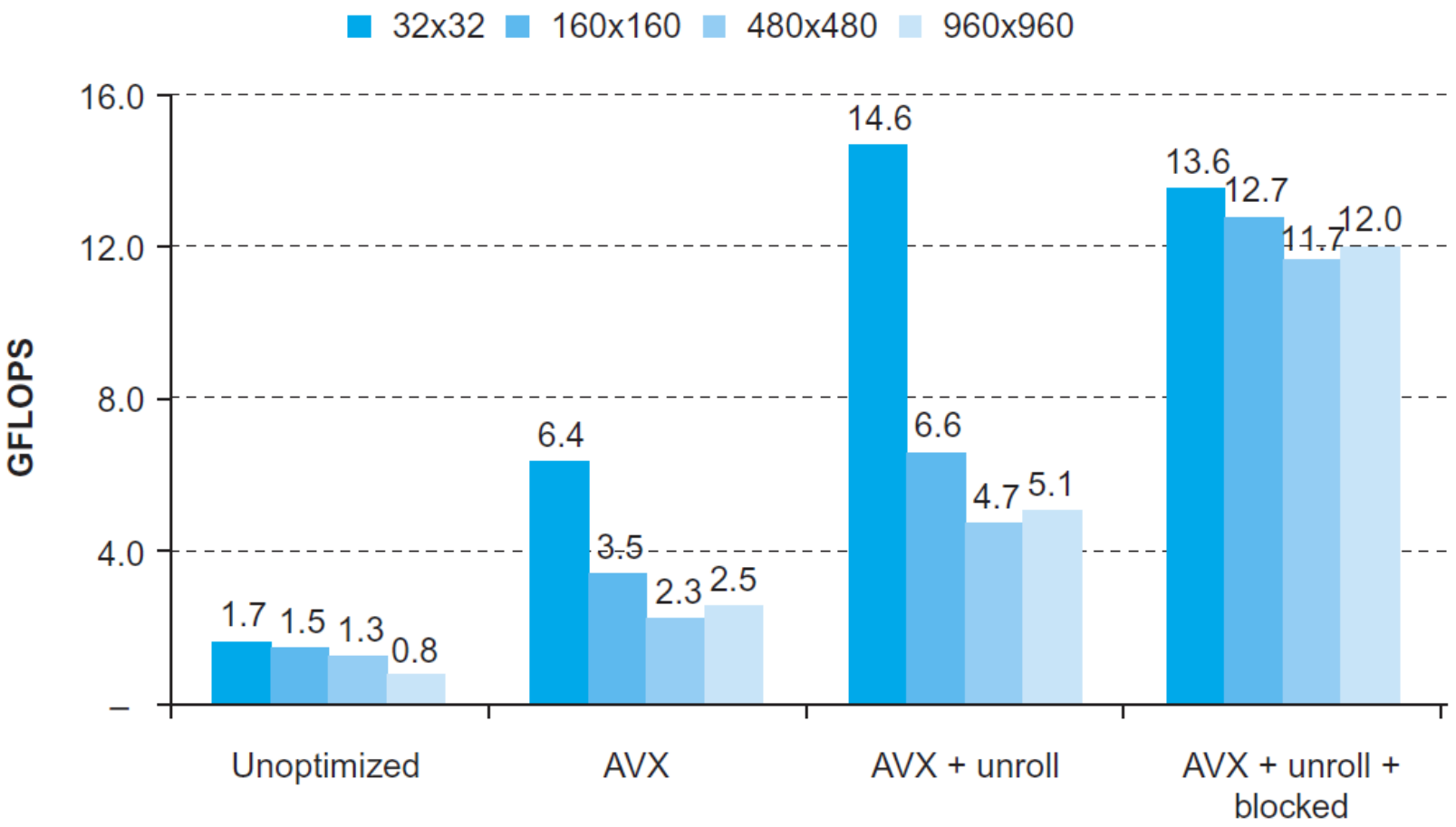
- 两者都有多银行缓存，在不存在银行冲突的情况下，允许每个周期进行多次访问
- 其他优化
  - 关键字优先：首先返回请求的数据
  - 非阻塞缓存
    - 缺失下命中：缺失期间有其它缓存命中
    - 缺失下缺失：允许发生多个未完成的缓存缺失
  - 数据预取

# 5.14 实例： RISC-V系统的其它部分和特殊指令

| Type | Mnemonic | Name |
|---|---|---|
| Mem ordering | fence.i | Instruction fence |
| | fence | Fence |
| | sfence.vm | Address translation fence |
| CSR access | csrrwi | CSR read/write immediate |
| | csrrsi | CSR read/set immediate |
| | csrrci | CSR read/clear immediate |
| | csrrw | CSR read/write |
| | csrrs | CSR read/set |
| | csrrc | CSR read/clear |
| System | ecall | Environment call |
| | ebreak | Environment breakpoint |
| | sret | Supervisor exception return |
| | wfi | Wait for interrupt |

# 5.15 加速：cache分块和矩阵乘法

- DGEMM
  - 结合子字并行子字并行、循环展开、缓存阻塞技术等优化技术



# 5.16 谬误与陷阱

- 字节编址 vs字编址
  - 示例：32byte直接映射的cache，4byte/block
    - 字节地址36，映射到block1
    - 字地址36，映射到block4
- 编写或生成代码时，忽略存储系统的影响
  - 示例：对数组行进行迭代 vs 对列进行迭代
  - 大范围跳转导致较差的局部性

# 5.16 谬误与陷阱

- 在具有共享二级或三级缓存的多处理器中
  - 关联性小于核心会导致冲突未命中
  - 更多核心　需要增加关联性
- 使用AMAT评估无序处理器的性能
  - 忽略非阻塞访问的影响
  - 相反，通过模拟来评估性能

# 5.16 谬误与陷阱

- 使用段扩展地址范围
  - 例如，英特尔80286
  - 但一个细分市场并不总是足够大
  - 使地址算法变得复杂
- 在非虚拟化设计的ISA上实现VMM
  - 例如，访问硬件资源的非特权指令
  - 要么扩展ISA，要么要求来宾操作系统不要使用有问题的指令

# 5.17 本章小结

- 快的记忆很小，大的记忆很慢
  - 我们真的想要快速、大容量的记忆
  - 缓存带来了这种错觉
- 局部性原则
  - 程序经常使用一小部分内存空间
- 内存层次结构
  - 一级缓存 ↔ 二级缓存 ↔ … ↔ DRAM存储器 ↔ 磁盘
- 内存系统设计对多处理器至关重要