第5章

大容量和高速度：开发存储器层次结构

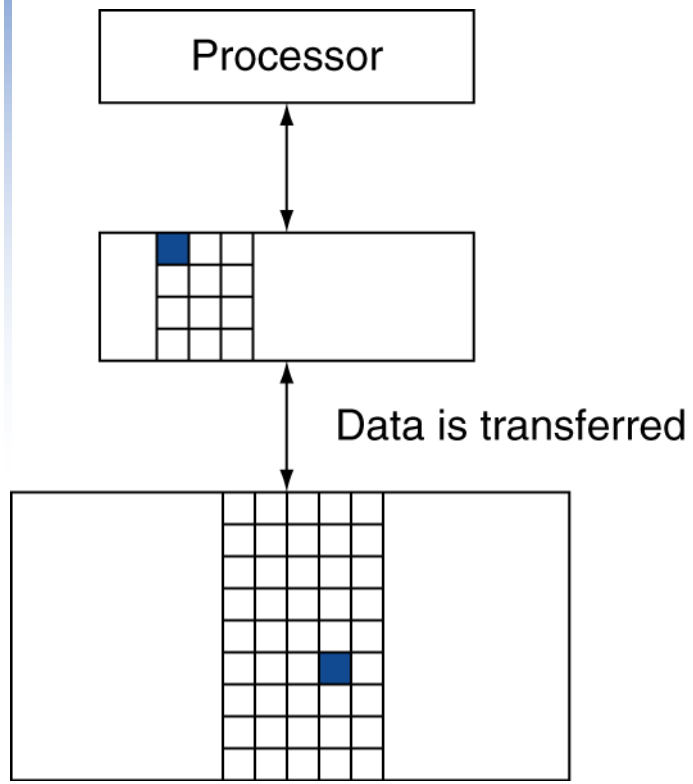
局部性原理

- 在任何时候，程序仅访问地址空间中一小块
- 时间局部性
  - 最近被访问的项可能马上又被访问到
  - 例如，循环体中的指令，归纳变量(induction variables)
- 空间局部性
  - 与最近被访问项相邻的项可能马上被访问到
  - 例如，顺序的指令访问，数组数据

利用局部性

- 存储器层次结构
- 把所有东西存在磁盘上
- 将近期被访问（及其附近）的项从磁盘复制到相对小的DRAM存储器
  - 主存储器
- 将更近期被访问（及其附近）的项从DRAM复制到更小的SRAM存储器
  - 附属于CPU的cache存储器

存储器层次结构的各层级



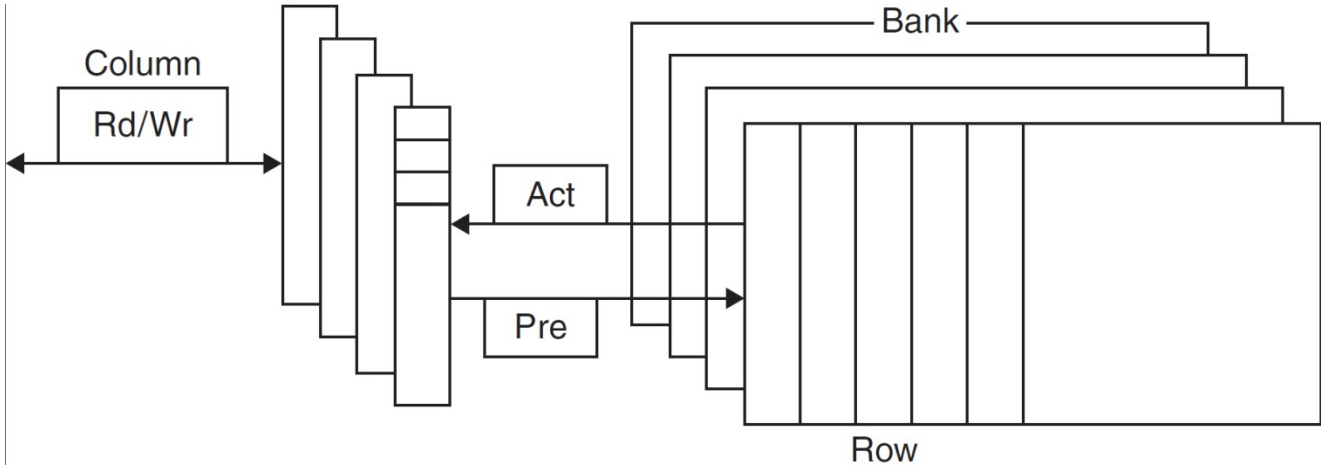
- 块（亦称行）：复制单位
  - 可能是多个字
- 如果要访问的数据在高层中
  - 命中：可从高层访问
    - 命中率：命中次数/访问次数
- 如果要访问的数据不在高层中
  - 缺失：从低层复制一块数据
    - 所用时间：缺失代价
    - 缺失率：缺失次数/访问次数
      = 1 – 命中率
  - 然后访问高层提供的数据

存储器技术

- 静态RAM (SRAM)
  - 0.5ns – 2.5ns，每GB $2000 – $5000
- 动态RAM (DRAM)
  - 50ns – 70ns，每GB $20 – $75
- 磁盘
  - 5ms – 20ms，每GB $0.20 – $2
- 理想存储器
  - 访问时间与SRAM相同
  - 容量和每GB的成本与磁盘相同

DRAM技术

- 数据以电容中电荷的形式存储
  - 用单个晶体管访问电荷
  - 必须定期刷新
    - 读取内容再写回
    - 作用于DRAM的一“行”

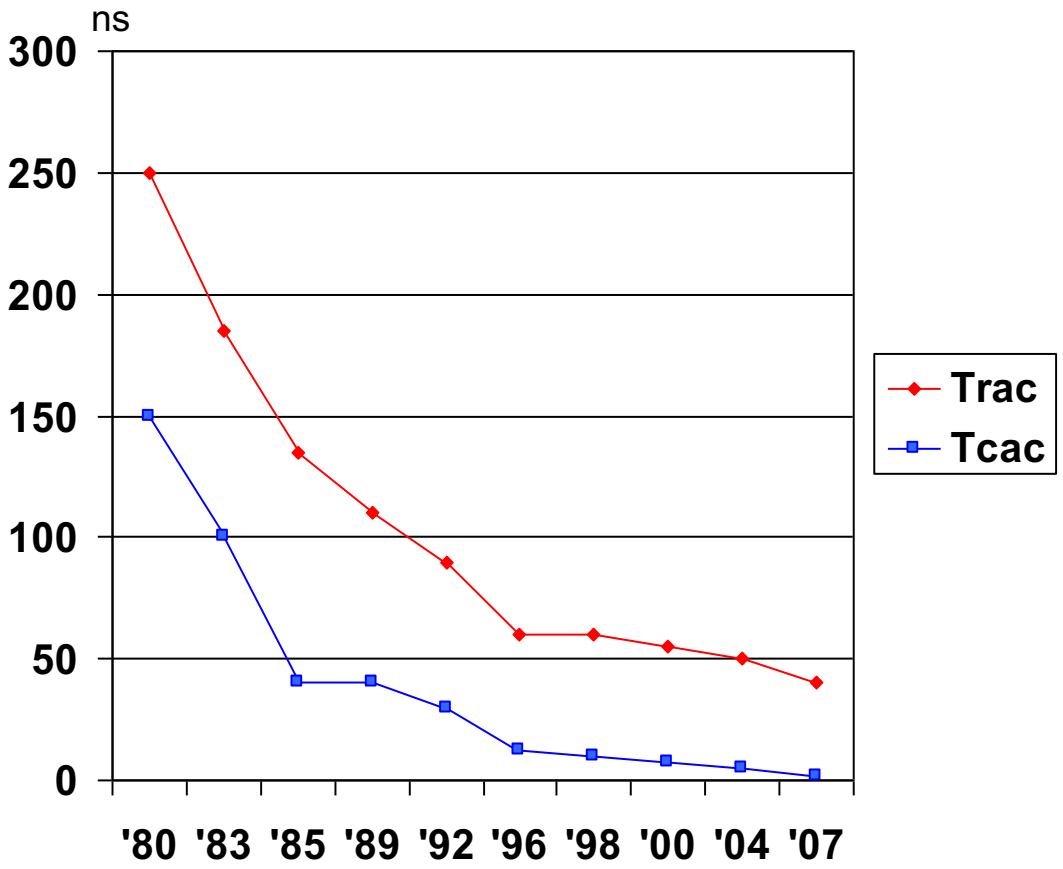


先进的DRAM组织方式

- DRAM中的位组织成矩形阵列
  - DRAM访问一整行
  - 突发模式：用更短的延时提供一行中连续的字
- 双倍数据速率(DDR) DRAM
  - 在时钟的上升沿和下降沿都传输
- 四倍数据速率(QDR) DRAM
  - 将DDR的输入和输出分离

各代DRAM

年份	容量	价格/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



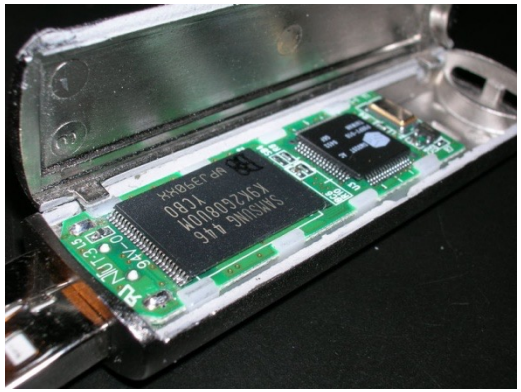
Trac：从新访问一行数据的时间
Tcac：访问已缓存数据的时间

DRAM性能要素

- 行缓冲器
  - 允许并行地读取或刷新多个字
- 同步DRAM
  - 允许以突发方式连续访问而无需发送每个地址
  - 提高带宽
- 多体DRAM
  - 允许同时访问多个DRAM
  - 提高带宽

闪存

- 非易失性半导体存储器
  - 比磁盘快100到1000倍
  - 更小、功耗更低、更坚固
  - 但每GB成本更高（介于磁盘和DRAM之间）

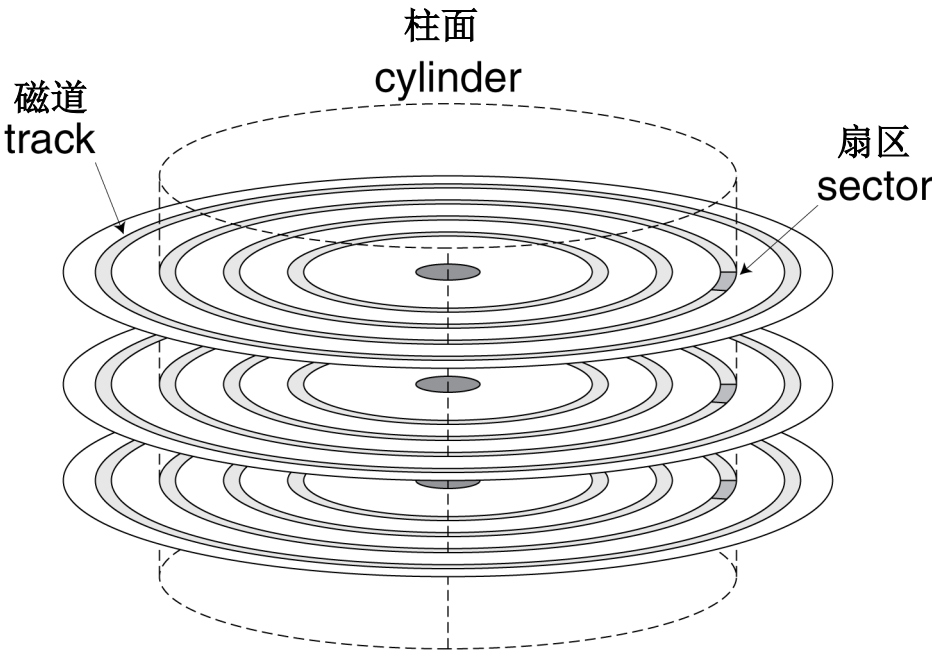


闪存的种类

- NOR闪存：位单元类似或非门
  - 随机读/写访问
  - 在嵌入式系统中用作指令存储器
- NAND闪存：位单元类似与非门
  - 密度更大（单位面积的位数），但每次只能访问一个块
  - 每GB更便宜
  - 用于U盘、媒体存储器等等
- 闪存位在上千次访问后损坏
  - 不适于直接代替RAM或磁盘
  - 损耗均衡：将数据重新映射到较少使用的块中

磁盘存储器

- 非易失、转动的磁介质存储器



磁盘扇区及访问

- 每个扇区记录有
  - 扇区ID
  - 数据（标准规定了512字节、4096字节）
  - 纠错码(ECC)
    - 用于隐藏缺陷与记录差错
  - 同步字段和间隙
- 访问一个扇区涉及
  - 如果其他访问正在等待，则有排队延时
  - 寻道：移动磁头
  - 旋转延时
  - 数据传输
  - 控制器开销

访问磁盘的例子

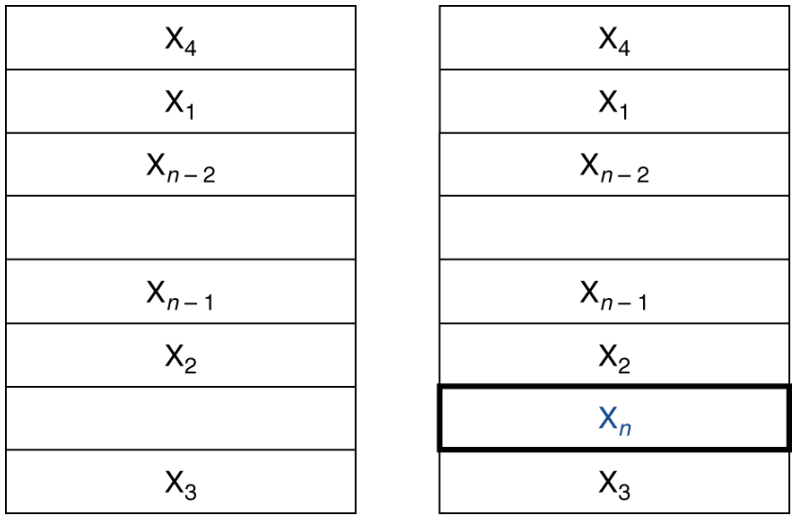
- 已知
  - 扇区512字节，每分钟15000转，平均寻道时间4ms，传输速率100MB/s，控制器开销0.2ms，理想磁盘
- 平均读取时间
  - 4ms寻道时间
+ ½ / (15000/60) = 2ms 旋转延时
+ 512 / 100MB/s = 0.005ms 传输时间
+ 0.2ms 控制器延时
= 6.2ms
- 如果实际平均寻道时间为1ms
  - 平均读取时间 = 3.2ms

磁盘性能的几个问题

- 制造商标示平均寻道时间
  - 基于所有寻道可能
  - 局部性及操作系统调度使得平均寻道时间更短
- 智能磁盘控制器为磁盘分配物理扇区
  - 为主机提供逻辑扇区接口
  - SCSI、ATA、SATA
- 磁盘驱动器内含cache
  - 预判访问位置，预取扇区
  - 避免寻道和旋转延时

cache存储器

- cache存储器
  - 最靠近CPU的存储器层级
- 给定访问序列$X_1, \ldots, X_{n-1}, X_n$

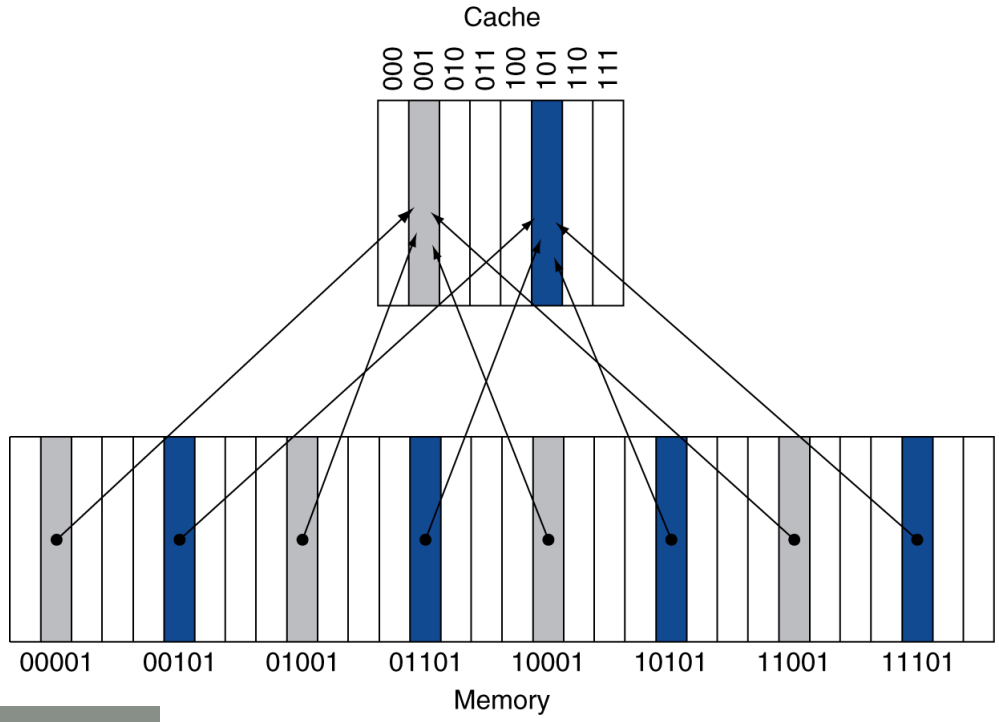


a. Before the reference to X_n　　b. After the reference to X_n

- 我们如何知道数据是否在cache中？
- 我们该去哪里找数据？

直接映射cache

- 位置取决于地址
- 直接映射：只有一个选择
  - (块地址) % (cache中的块数) // %为求余运算



- 块数是2的幂
- 使用地址低位

标记和有效位

- 我们如何知道cache中的某个位置存储的是哪个块？
  - 既保存数据，也保存块地址
  - 实际仅需保存高位地址
  - 称为标记(tag)
- 如果一个位置中没有数据怎么办？
  - 有效位：1 = 有数据，0 = 没数据
  - 初值为0

cache的例子

- 8块，每块1字，直接映射
- 初始状态

索引	V	标记	数据
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

cache的例子

字地址	二进制地址	命中/缺失	cache块
22	10 110	缺失	110

索引	V	标记	数据
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

cache的例子

字地址	二进制地址	命中/缺失	cache块
26	11 010	缺失	010

索引	V	标记	数据
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

cache的例子

字地址	二进制地址	命中/缺失	cache块
22	10 110	命中	110
26	11 010	命中	010

索引	V	标记	数据
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

cache的例子

字地址	二进制地址	命中/缺失	cache块
16	10 000	缺失	000
3	00 011	缺失	011
16	10 000	命中	000

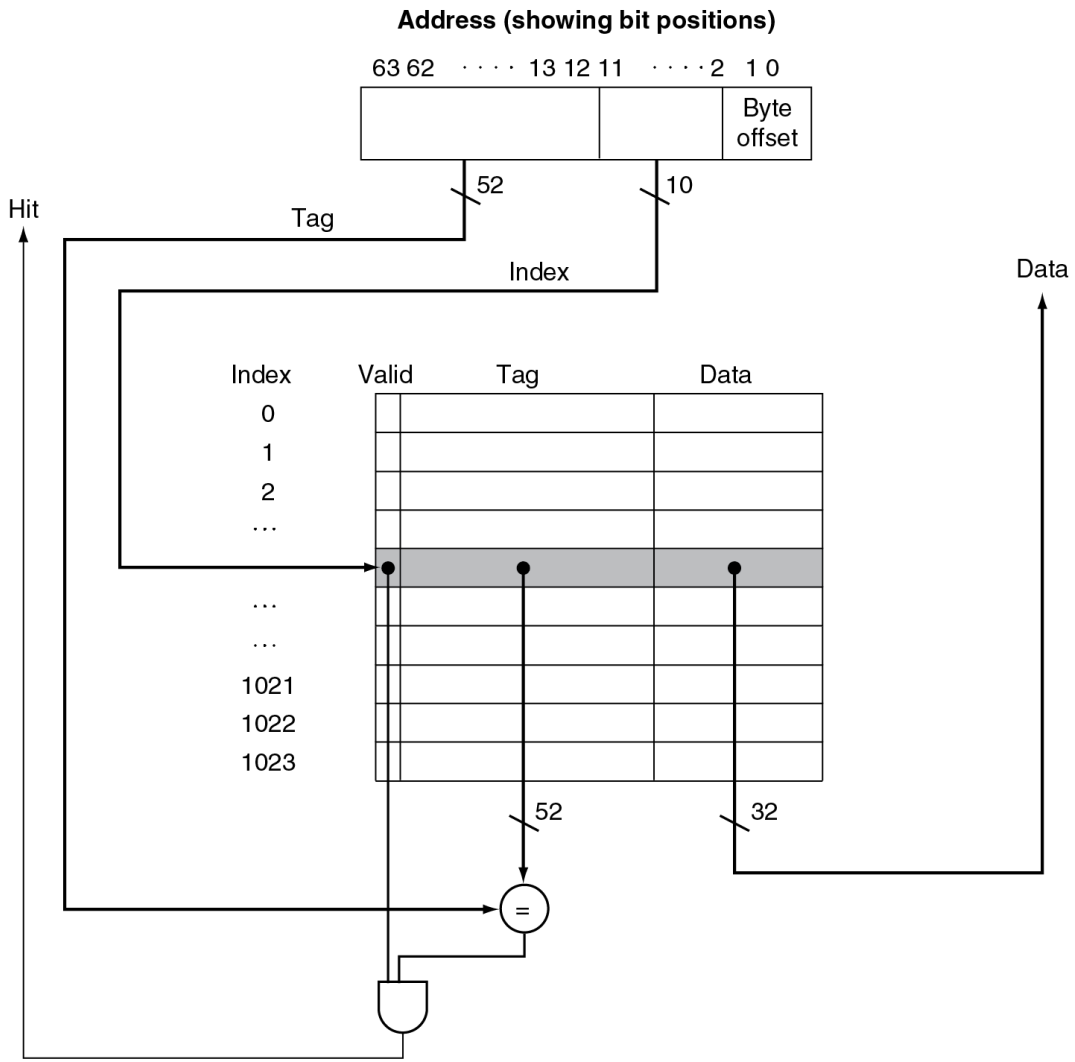
索引	V	标记	数据
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

cache的例子

字地址	二进制地址	命中/缺失	cache块
18	10 010	缺失	010

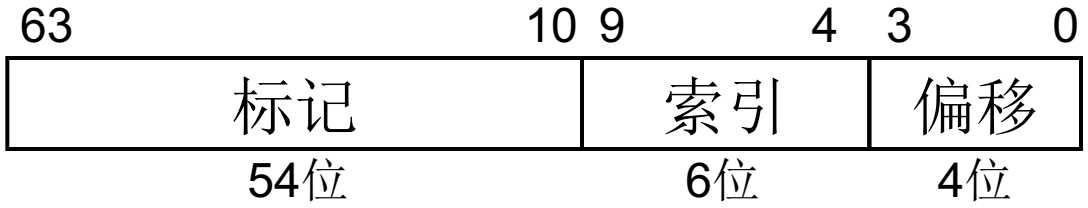
索引	V	标记	数据
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

地址分解


Address (showing bit positions)
63 62 · · · · 13 12 11 · · · · 2 1 0
Byte offset
52
10
Hit
Tag
Index
Data
Index
Valid
Tag
Data
0
1
2
· · ·
· · ·
· · ·
1021
1022
1023
52
32
=

例：更大的块

- 64个块，每块16字节
  - 地址1200映射到几号块？
- 块地址 = ⌊1200/16⌋ = 75
- 块号 = 75 % 64 = 11



块大小的考量

- 用更大的块可以降低缺失率
  - 由于空间局部性
- 不过对于一个大小固定的cache
  - 块越大 ⇒ 块数量越小
    - 竞争更多 ⇒ 缺失率增加
  - 更大的块 ⇒ 内容污染
- 缺失代价更大
  - 会超过缺失率降低带来的好处
  - 使用早期重启和关键字先传技术会有所帮助

cache缺失

- 当cache命中时，CPU正常处理
- 当cache缺失时
  - 阻塞CPU流水线
  - 从下一层级取块
  - 指令cache缺失
    - 重新取指
  - 数据cache缺失
    - 完成数据访问

写直达（write-through）

- 当写数据命中时，可以只更新cache中的块
  - 但cache和存储器会不一致
- 写直达：同时也更新存储器
- 但使写耗时更长
  - 例如，如果基本CPI = 1，10%的指令是存数，写存储器花费100个周期
    - 实际CPI = 1 + 10% ×100 = 11
- 解决办法：写缓冲(write buffer)
  - 保持等待写入存储器的数据
  - CPU立即继续工作
    - 仅当写缓冲满时写操作才会引起阻塞

写回（write-back）

- 替代方案：写数据命中时，仅更新cache中的块
  - 跟踪每个块是否变脏（内容改变）
- 当一个脏块被替换时
  - 将其写回存储器
  - 可用写缓冲使被替换的块能先被读取

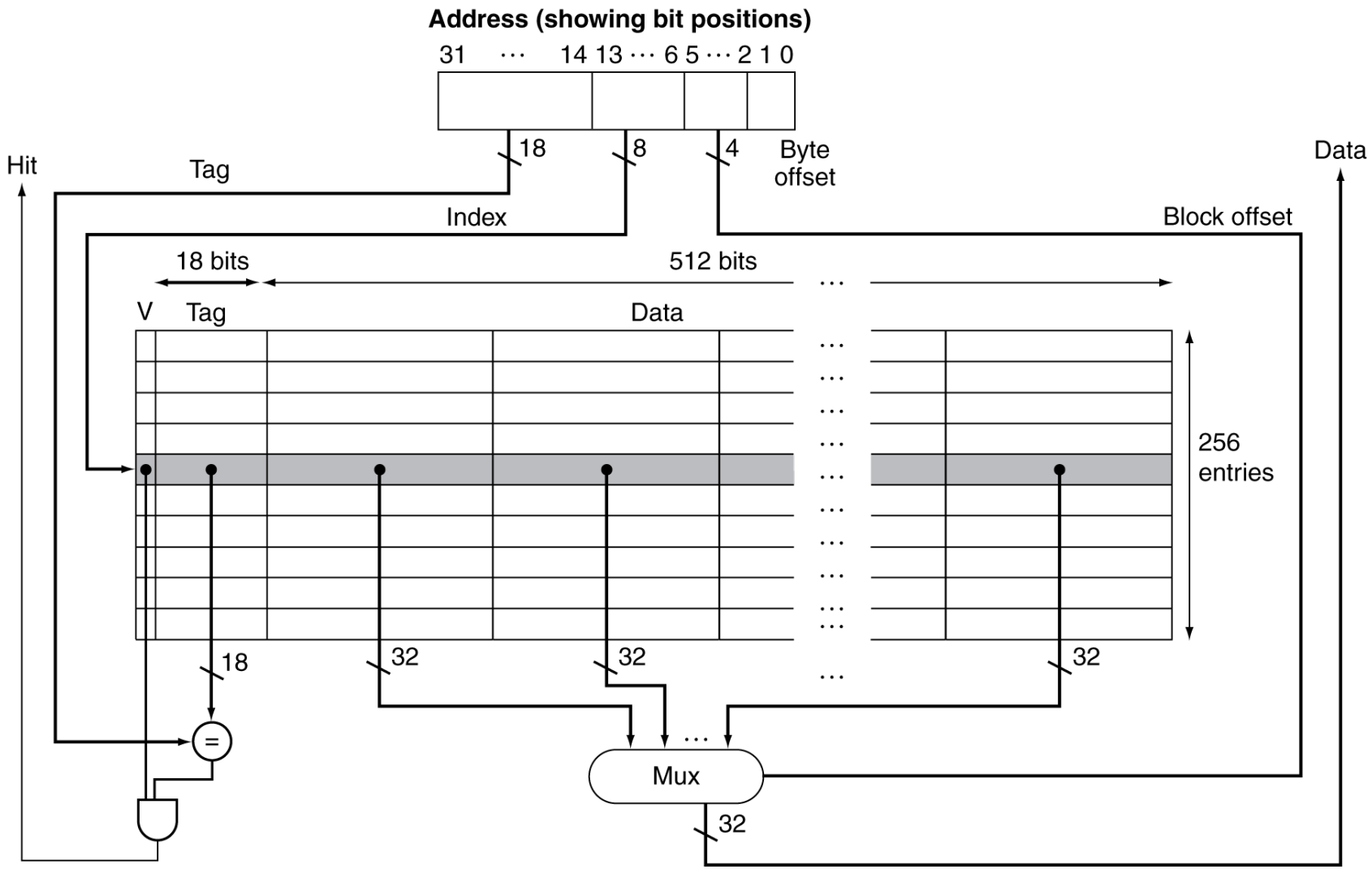
写分配（write allocation）

- 发生写缺失时该怎么办？
- 对于写直达，有两种方法
  - 缺失时分配：取这个块到cache中
  - 绕过cache：只写主存，不写到cache中
    - 因为程序经常在读某个块之前先写这整个块（例如，初始化）
- 对于写回
  - 通常取这个块

例子：Intrinsity FastMATH

- 嵌入式MIPS处理器
  - 12级流水线
  - 每周期进行指令和数据访问
- 分离的cache：独立的指令cache和数据cache
  - 各为16KB：256 块 × 16 字/块
  - 数据cache：写直达或写回
- SPEC2000缺失率
  - 指令cache：0.4%
  - 数据cache：11.4%
  - 加权平均：3.2%

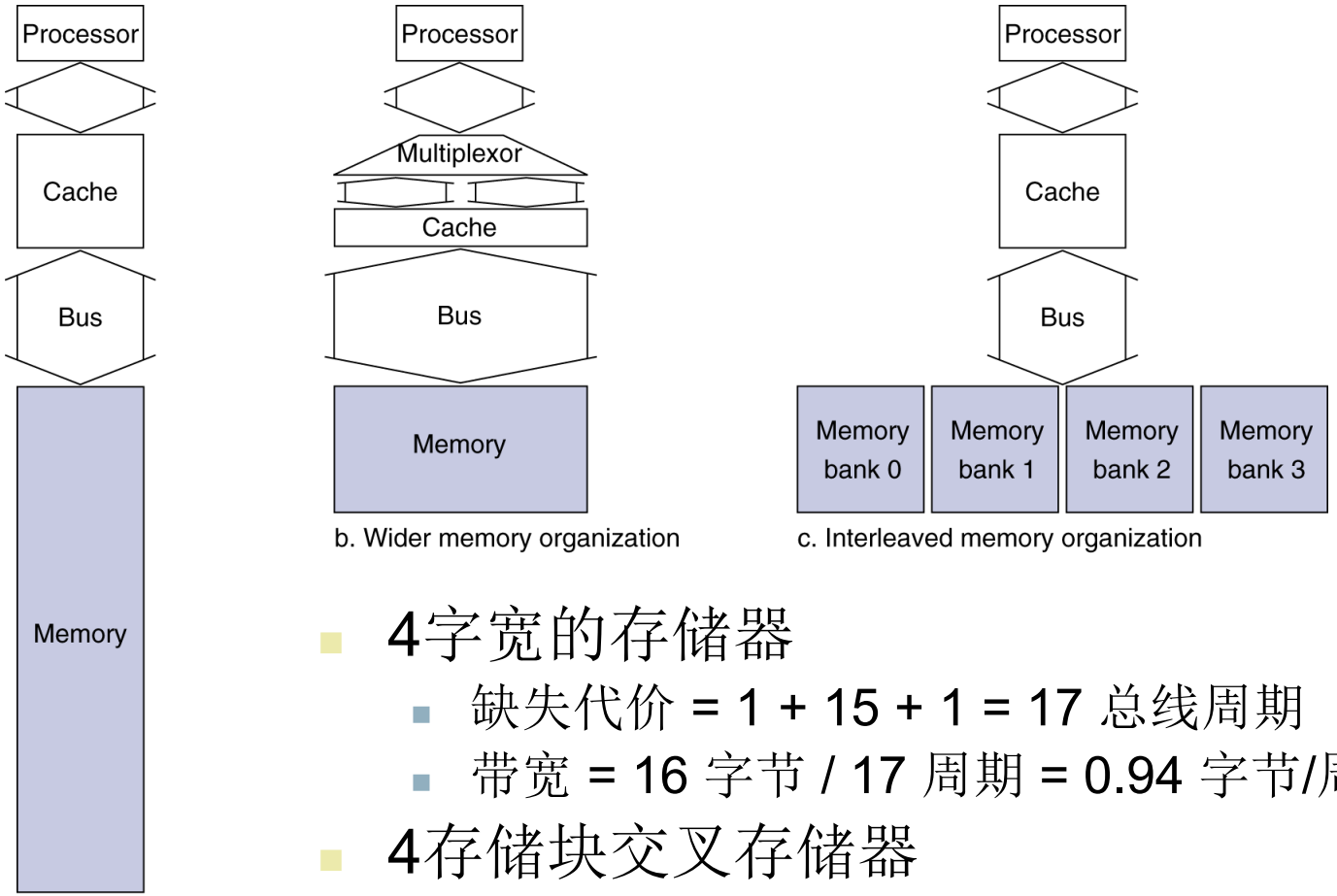
例子：Intrinsity FastMATH



主存储器支持cache

- 用DRAM做主存储器
  - 固定宽度（例如，1个字）
  - 由固定宽度的带时钟总线连接
    - 总线时钟通常比CPU时钟慢
- 例：读cache块
  - 地址传输用1个总线周期
  - 每次DRAM访问用15个总线周期
  - 每次数据传送用1个总线周期
- 对于4字的块、1字宽的DRAM
  - 缺失代价 = 1 + 4×15 + 4×1 = 65 总线周期
  - 带宽 = 16 字节 / 65 周期 = 0.25 字节/周期

提高存储器带宽



- 4字宽的存储器
  - 缺失代价 = 1 + 15 + 1 = 17 总线周期
  - 带宽 = 16 字节 / 17 周期 = 0.94 字节/周期
- 4存储块交叉存储器
  - 缺失代价 = 1 + 15 + 4×1 = 20 总线周期
  - 带宽 = 16 字节 / 20 周期 = 0.8 字节/周期

衡量cache性能

- CPU时间的组成
  - 程序执行周期数
    - 包括cache命中时间
  - 存储器阻塞周期数
    - 主要来自cache缺失
- 简化假设：

$$\text{存储器阻塞周期数} = \frac{\text{存储器访问次数}}{\text{程序}} \times \text{缺失率} \times \text{缺失代价}$$

$$= \frac{\text{指令数}}{\text{程序}} \times \frac{\text{缺失数}}{\text{指令}} \times \text{缺失代价}$$

cache性能的例子

- 已知
  - 指令cache缺失率 = 2%
  - 数据cache缺失率 = 4%
  - 缺失代价 = 100周期
  - 基本CPI（理想cache）= 2
  - 取数&存数指令占所有指令的36%
- 每条指令的缺失周期数
  - 指令cache：2% × 100 = 2
  - 数据cache：36% × 4% × 100 = 1.44
- 实际 CPI = 2 + 2 + 1.44 = 5.44
  - 理想CPU是它的5.44 / 2 = 2.72倍快

平均访问时间

- 命中时间对性能也很重要
- 平均存储器访问时间(AMAT)
  - AMAT = 命中时间 + 缺失率 × 缺失代价
- 例
  - 时钟为1ns的CPU，命中时间 = 1周期，
    缺失代价 = 20 周期，指令cache缺失率= 5%
  - AMAT = 1 + 5% × 20 = 2ns
    - 每条指令2周期

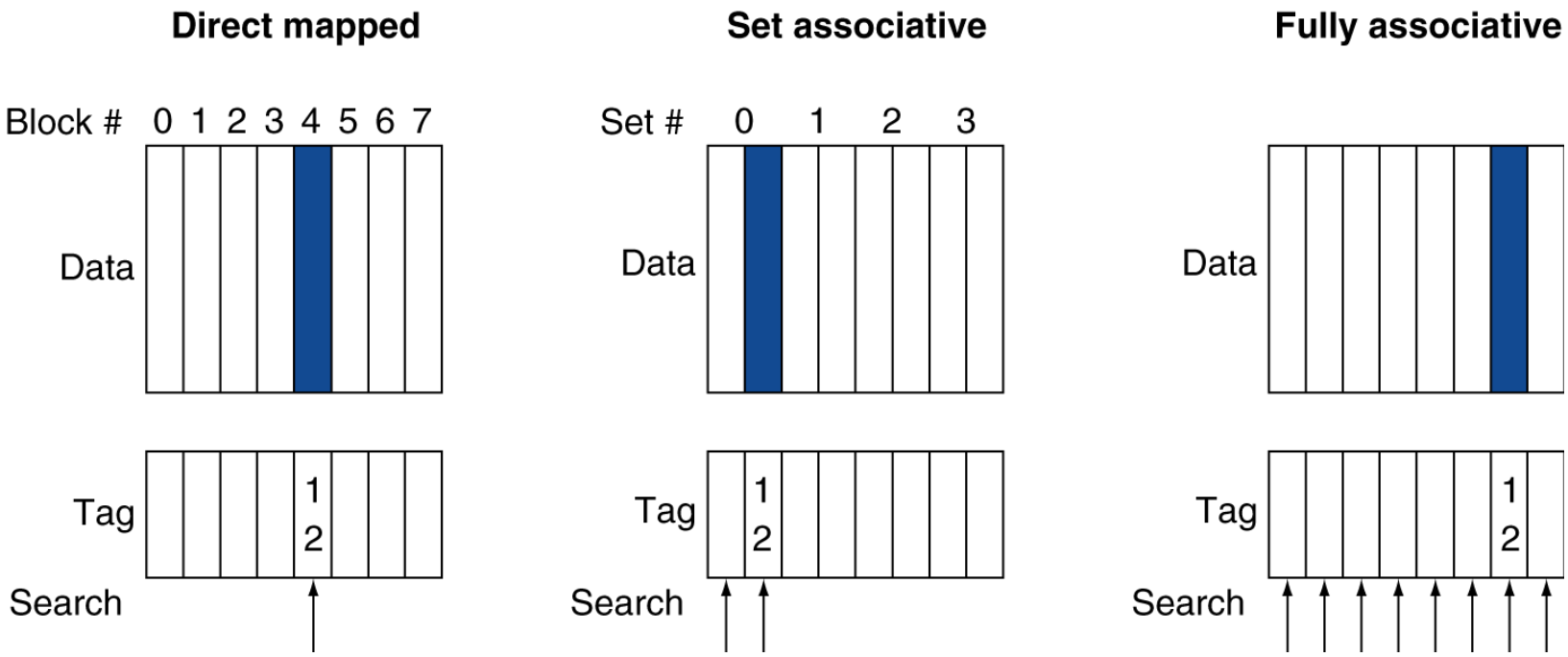
性能小结

- 当CPU性能提升时
  - 缺失代价更为显著
- 减小基本CPI
  - 内存阻塞耗时的比例更大
- 增加时钟速率
  - 存储器阻塞占用更多的CPU周期
- 在评估系统性能时不能忽视cache的行为

cache相联

- 全相联
  - 允许一个给定的块放入cache中的任意项
  - 需要一次搜索所有的项
  - 每项用一个比较器（昂贵）
- *n*路组相联
  - 每组包含*n*项
  - 在哪一组取决于块号
    - (块号) % (cache的组数)
  - 一次搜索给定组的所有项
  - *n*个比较器（不那么贵）

cache相联的例子


Direct mapped
Block # 0 1 2 3 4 5 6 7
Data
Tag
1
2
Search
Set associative
Set # 0 1 2 3
Data
Tag
1
2
Search
Fully associative
Data
Tag
1
2
Search

图解相联度

- 对于8项的cache

One-way set associative
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

相联度的例子

- 比较容量为4个块的几种cache
  - 直接映射、2路组相联、全相联
  - 访问块的次序为0、8、0、6、8
- 直接映射

块地址	cache 索引	命中/缺失	访问后cache中的内容			
			0	1	2	3
0	0	缺失	主存[0]			
8	0	缺失	主存[8]			
0	0	缺失	主存[0]			
6	2	缺失	主存[0]		主存[6]	
8	0	缺失	主存[8]		主存[6]	

相联度的例子

- 2路组相联

块地址	cache索引	命中/缺失	访问后cache中的内容			
			组0		组1	
0	0	缺失	主存[0]			
8	0	缺失	主存[0]	主存[8]		
0	0	命中	**主存[0]**	主存[8]		
6	0	缺失	主存[0]	**主存[6]**		
8	0	缺失	**主存[8]**	主存[6]		

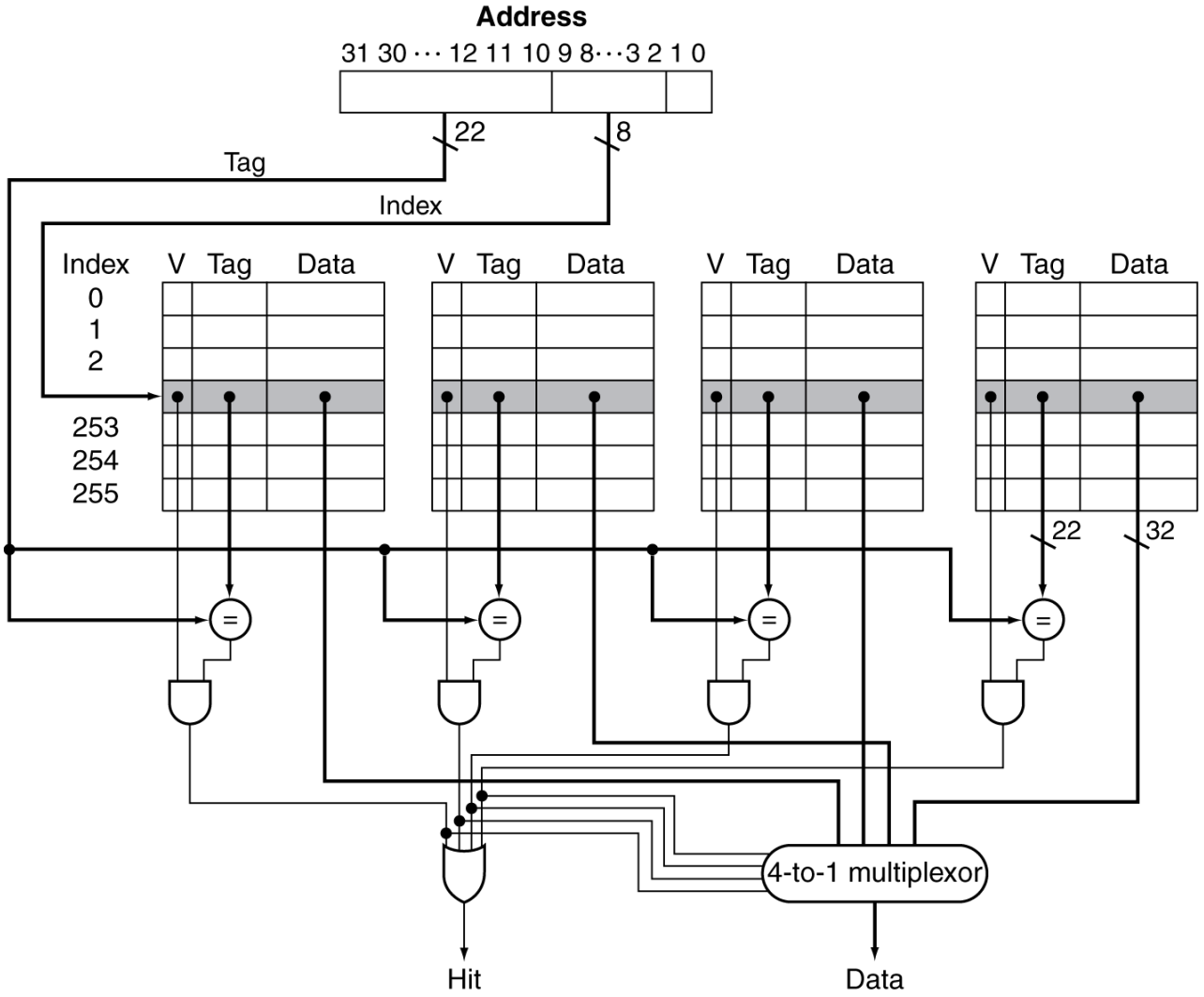
- 全相联

块地址		命中/缺失	访问后cache中的内容			
0		缺失	主存[0]			
8		缺失	主存[0]	主存[8]		
0		命中	**主存[0]**	主存[8]		
6		缺失	主存[0]	主存8]	主存[6]	
8		命中	主存[0]	**主存[8]**	主存[6]	

用多高的相联度

- 提高相联度可降低缺失率
  - 但回报是递减的
- 仿真：一个数据cache为64KB、块大小为16字的系统，跑SPEC2000
  - 1路：10.3%
  - 2路：8.6%
  - 4路：8.3%
  - 8路：8.1%

组相联cache的组织



替换规则

- 直接映射：无需选择
- 组相联
  - 如果有无效项，就选无效项
  - 否则，从这一组的项中选择
- 最近最少使用(LRU)
  - 选择最久没有使用的那一块
    - 对2路容易实现，4路亦可操作，但再多路就困难了
- 随机
  - 高相联度下与LRU算法的性能大致相同

多级cache

- 连接CPU的一级cache
  - 小，但快
- 二级cache处理一级cache缺失
  - 更大，更慢，但仍然比主存储器快
- 主存储器处理二级cache缺失
- 一些高端系统有三级cache

多级cache的例子

- 已知
  - CPU基本CPI = 1，时钟频率 = 4GHz
  - 每条指令的缺失率 = 2%
  - 主存储器访问时间 = 100ns
- 仅有一级cache
  - 缺失代价 = 100ns / 0.25ns = 400周期
  - 实际CPI = 1 + 2% × 400 = 9

例子（续）

- 现在加入二级cache
  - 访问时间 = 5ns
  - 到主存储器的全局缺失率 = 0.5%
- 一级缺失，二级命中
  - 代价 = 5ns / 0.25ns = 20周期
- 一级缺失，二级缺失
  - 额外代价 = 400周期
- CPI = 1 + 2% × 20 + 0.5% × 400 = 3.4
- 性能比 = 9 / 3.4 = 2.6

多级cache的考虑

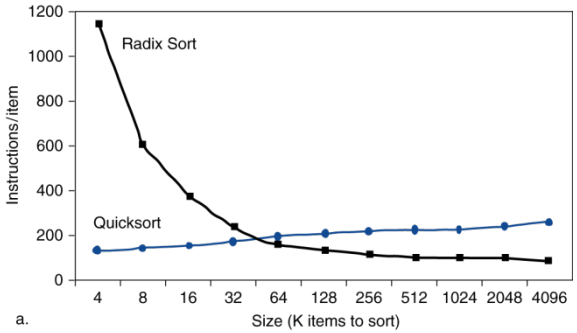
- 一级cache
  - 侧重于最小的命中时间
- 二级cache
  - 侧重于低缺失率，以避免访问主存储器
  - 命中时间对总体影响较小
- 结果
  - 一级cache通常比单一cache小
  - 一级cache的块比二级的小

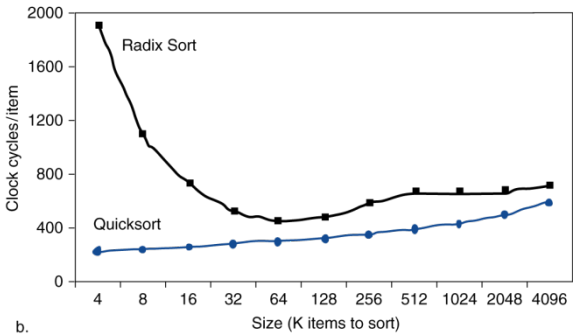
与先进CPU的交互作用

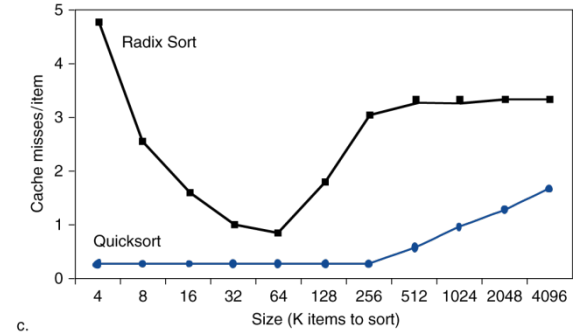
- 乱序执行CPU能在cache缺失期间执行指令
  - 未完成的存数操作停留在取数/存数单元中
  - 有依赖性的指令在保留站中等待
    - 无依赖性的指令继续执行
- 缺失造成的影响取决于程序的数据流动
  - 分析起来困难得多
  - 进行系统仿真

与软件的交互

- 缺失取决于存储器访问模式
  - 算法行为
  - 编译器对存储器访问的优化







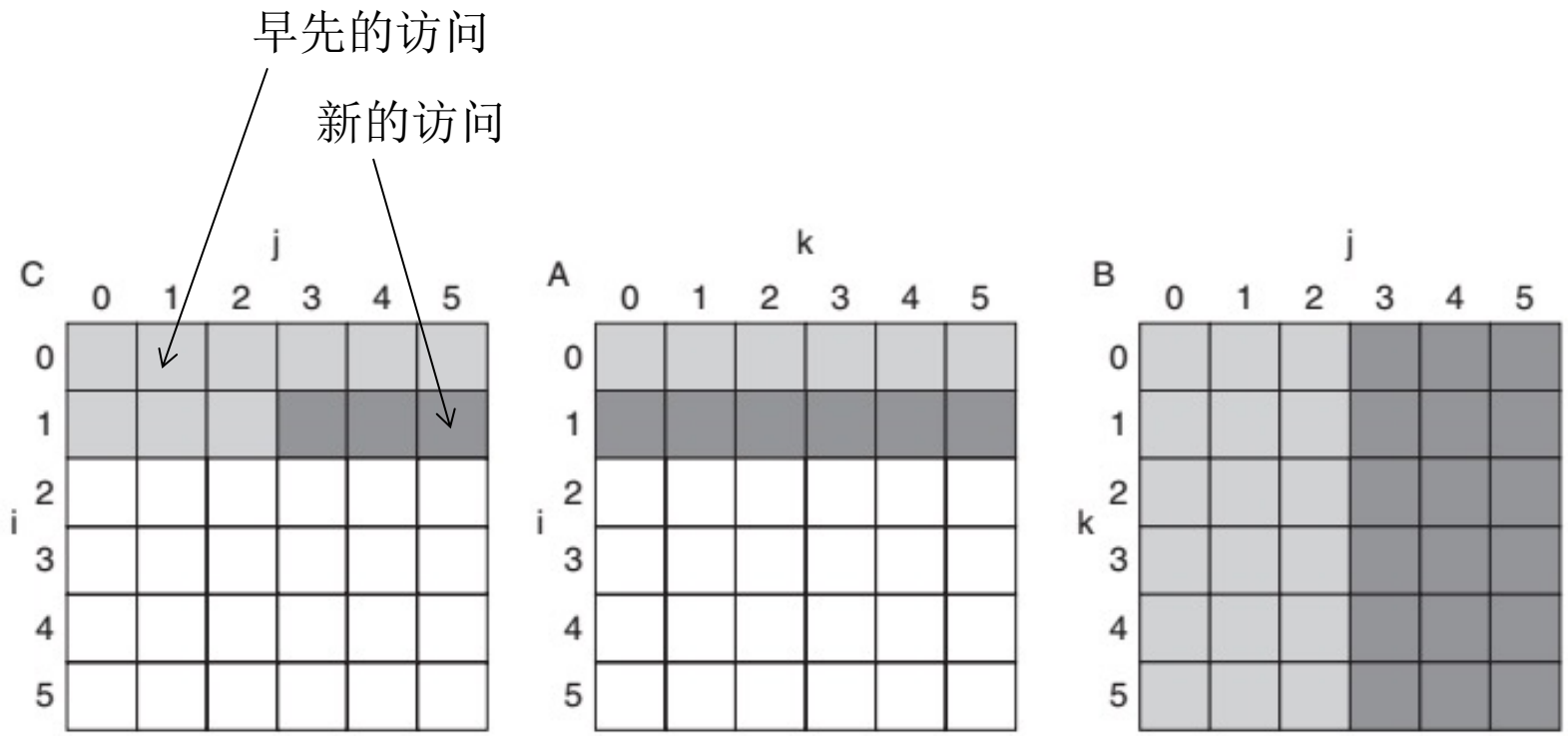
通过分块做软件优化

- 目标：在数据被替换之前尽量多用到
- 考虑DGEMM（double型通用矩阵乘法）中的内循环：

```
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;
}
```

DGEMM访问模式

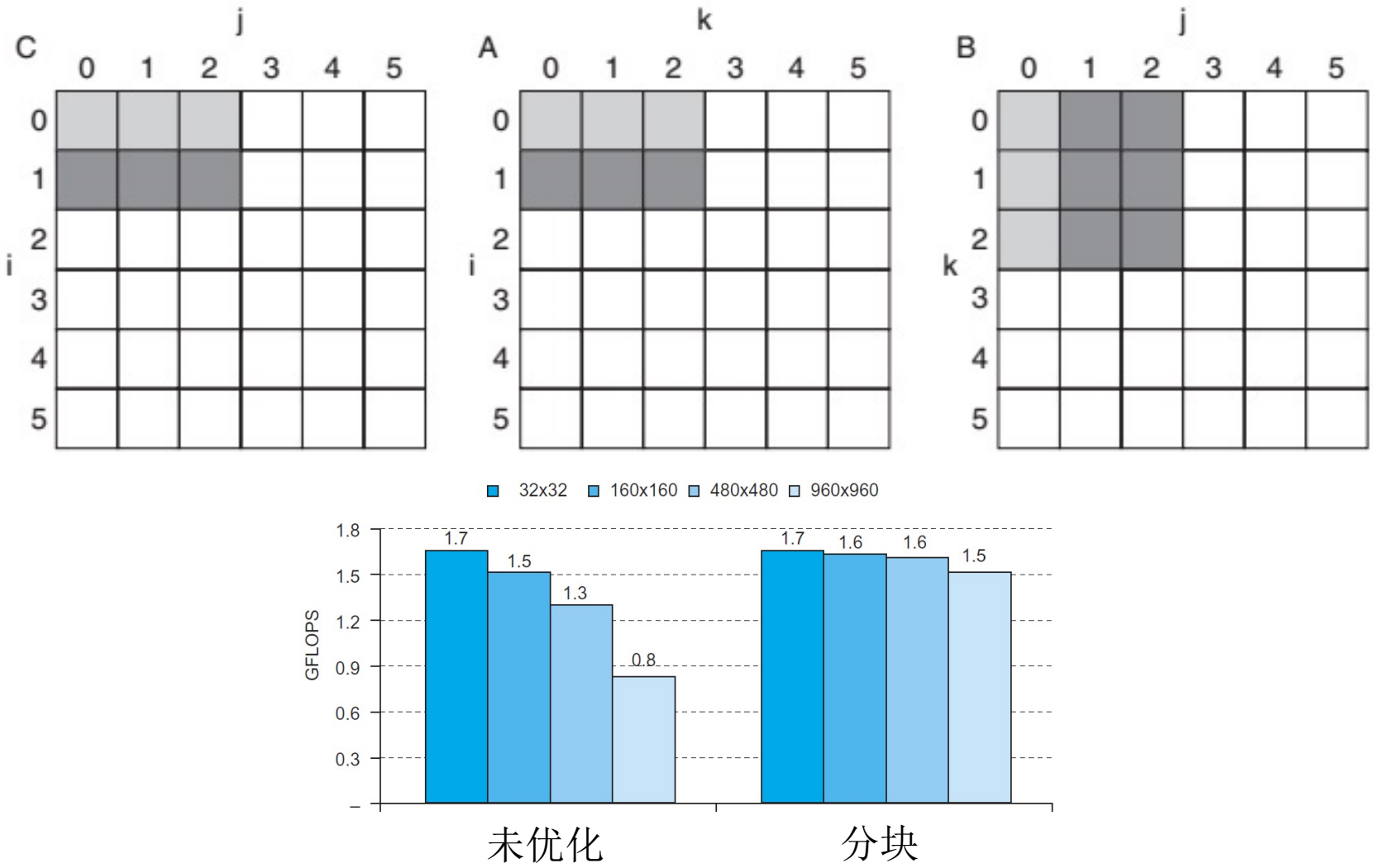
- 数组C、A和B



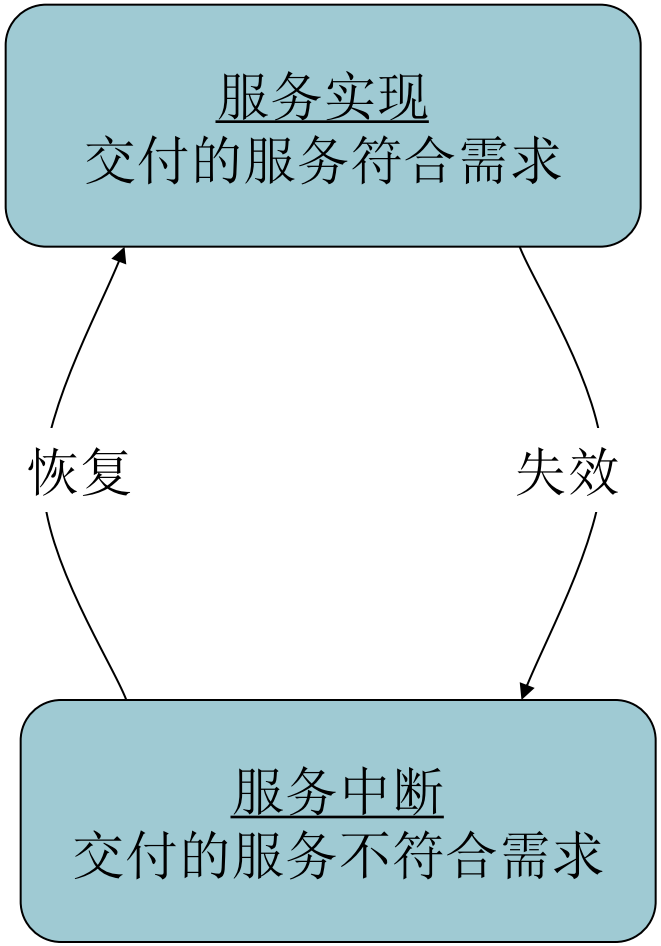
cache分块的DGEMM

```
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 }
```

分块DGEMM的访问模式



可信性(Dependability)



- 故障：组件的失效
  - 会/不会导致系统失效

可信性的度量

- 可靠性(reliability)：平均无故障时间(MTTF, mean time to failure)
- 服务中断：维修平均时间(MTTR, mean time to repair)
- 平均失效间隔时间(mean time between failures)
  - MTBF = MTTF + MTTR
- 可用性 = MTTF / (MTTF + MTTR)
- 提高可用性
  - 增大MTTF：故障避免技术、故障容忍技术、故障预报技术
  - 减小MTTR：改进用于诊断和修复的工具与流程

汉明SEC(1位纠错)码

- 汉明距离
  - 两个二进制数的相异位数
- 若最小距离为2，则可检测1位错
  - 例如奇偶校验码
- 若最小距离为3，则可纠正1位错、检测2位错

SEC的编码

- 计算汉明码：
  - 从左侧由1开始逐位编号
  - 所有位于2的幂的数位是奇偶校验位
  - 每个奇偶校验位用于校验特定的数据位

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

SEC的解码

- 奇偶校验位的值指出了出错的位
  - 沿用编码过程采用的编号
  - 例如
    - 奇偶校验位 = 0000，表示没有错误
    - 奇偶校验位 = 1010，表示10号位反转了

SEC/DED编码

- 为整个字附加一个奇偶校验位p_n
- 使汉明距离为4
- 解码：
  - 令H = SEC奇偶校验位
    - H为偶，p_n为偶，没有错误
    - H为奇，p_n为奇，有一位可纠正错误
    - H为偶，p_n为奇，p_n 位出错
    - H为奇，p_n为偶，有两位错误
- 注：ECC DRAM使用的SEC/DED码用8位来保护64位数据

虚拟机(Virtual Machine)

- 主机模拟客户操作系统和机器资源
  - 更好地隔离多客户
  - 避免安全性和可靠性问题
  - 有助于资源共享
- 虚拟化对性能有一定影响
  - 在现代高性能计算机上是可行的
- 例
  - IBM VM/370（1970年代的科技！）
  - VMWare
  - Microsoft Virtual PC

虚拟机监视器

- 将虚拟资源映射到物理资源
  - 存储器、I/O设备、CPUs
- 客户代码运行在用户模式下的本地机器上
  - 若执行特权指令或访问受保护资源，则向VMM发出trap中断
- 客户OS可与主机OS不同
- VMM操控真实的I/O设备
  - 为用户模拟出通用虚拟I/O设备

例子：定时器虚拟化

- 发生定时器中断时，在本地机器上
  - OS挂起当前进程，处理中断，选择并恢复下一进程
- 对于虚拟机监视器
  - VMM挂起当前VM，处理中断，选择并恢复下一个VM
- 如果一个VM需要定时器中断
  - VMM模拟一个虚拟定时器
  - 当物理定时器中断发生时，为VM模拟中断

指令集支持

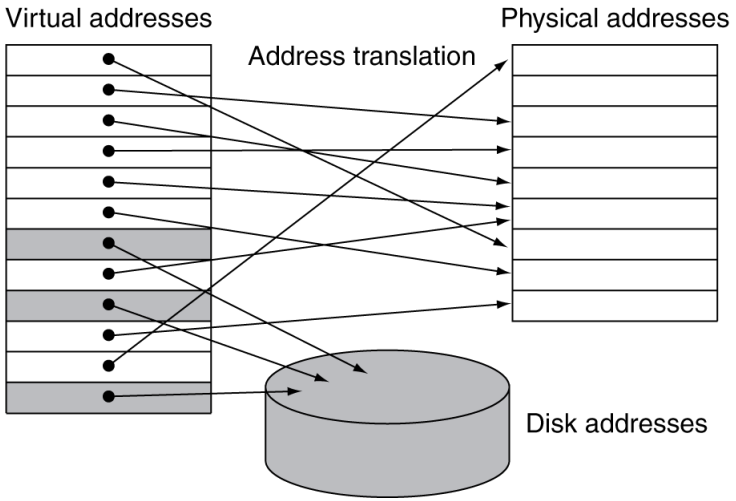
- 用户模式和系统模式
- 特权指令仅在系统模式下可用
  - 如果在用户模式下执行，就向系统发送trap
- 所有物理资源只允许用特权指令访问
  - 包括页表、中断控制、I/O寄存器
- 虚拟化支持的复兴
  - 现今的ISAs（例如x86）适应虚拟化

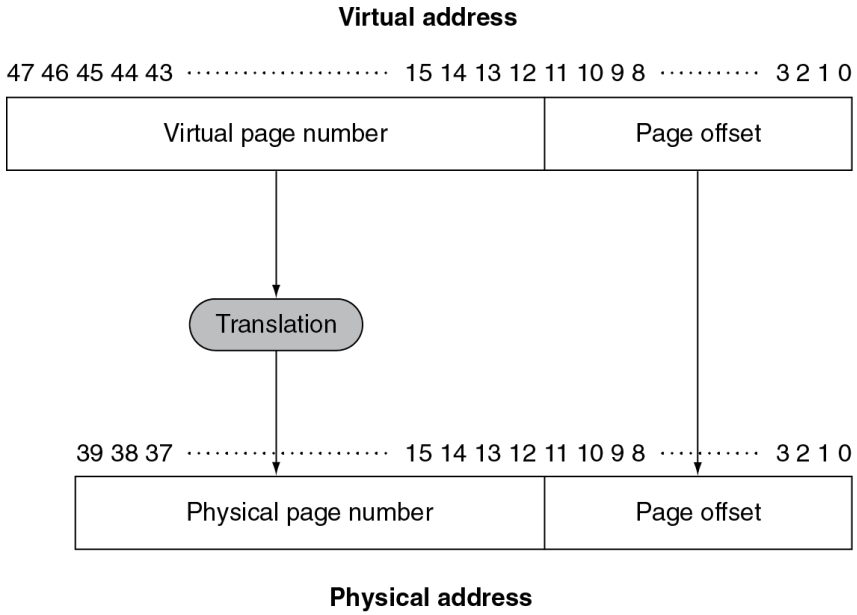
虚拟存储器(Virtual Memory)

- 将主存储器作为二级存储器（磁盘）的“cache”使用
  - 由CPU硬件和操作系统(OS)共同管理
- 程序共享主存储器
  - 每个程序获得私有虚拟地址空间，用以存放它频繁使用的代码和数据
  - 受到保护不被其他程序访问
- CPU和OS将虚拟地址转换为物理地址
  - VM“块”称为页
  - VM转换“缺失”称为缺页(page fault)

地址转换

- 页大小固定（例如4KB）





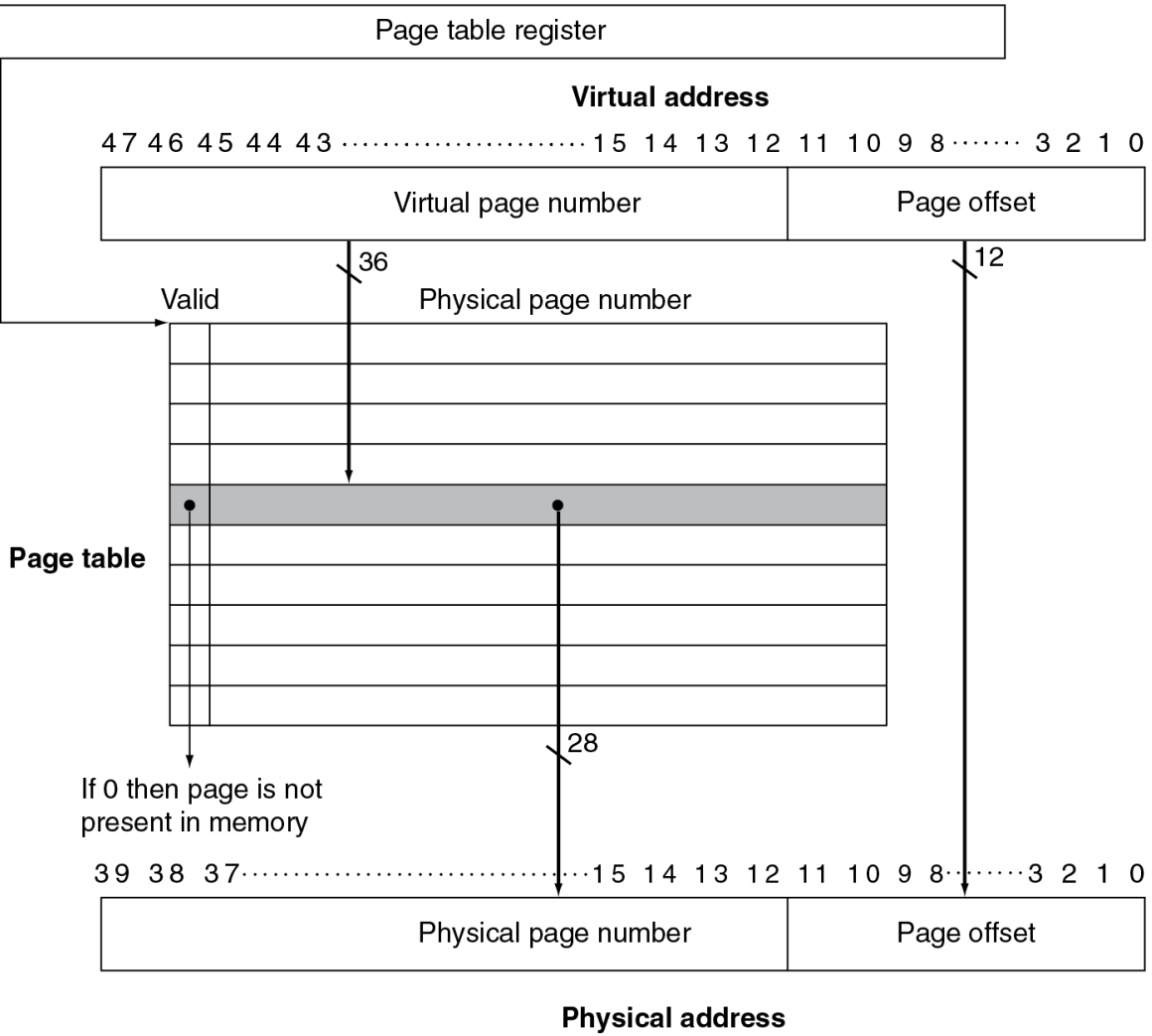
缺页代价

- 缺页时，必须从磁盘取这一页
  - 花费数百万时钟周期
  - 由OS代码处理
- 尽量将缺页率降到最低
  - 以全相联方式放置
  - 智能的替换算法

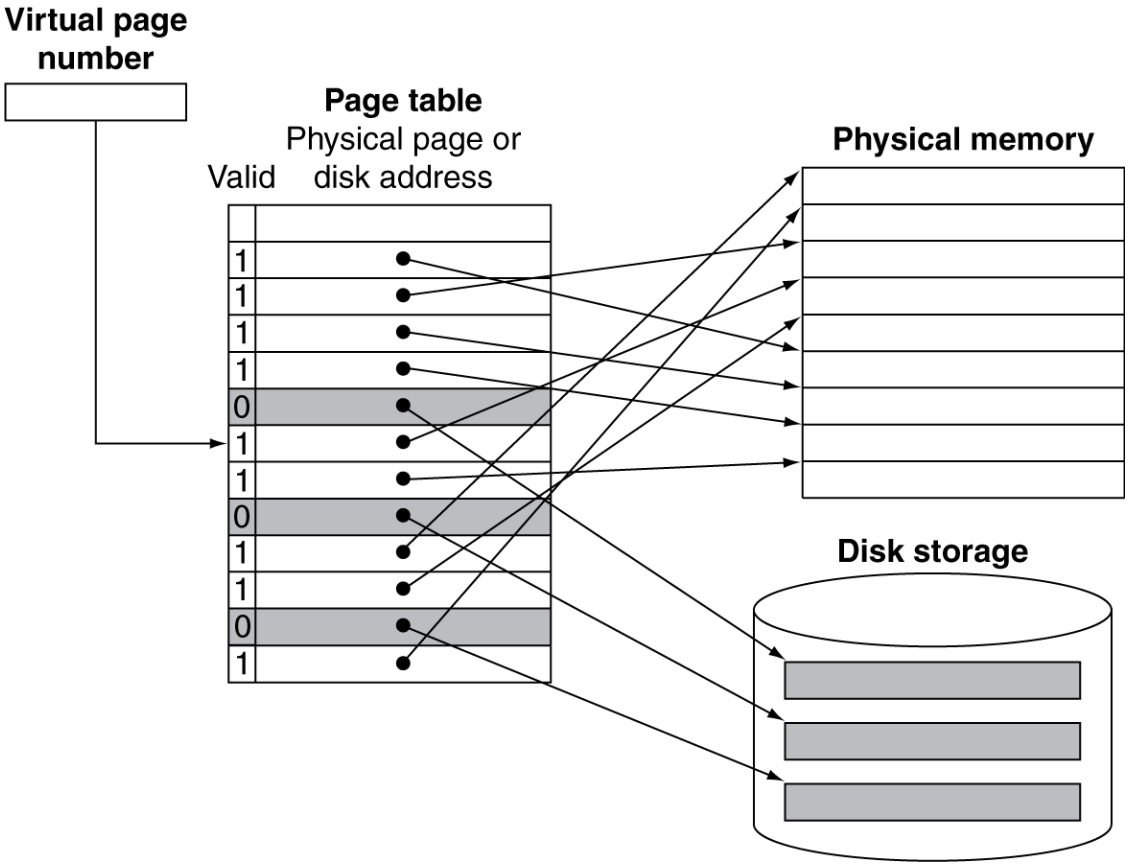
页表

- 存储位置信息
  - 页表项(PTE)构成的数组，按虚页号索引
  - CPU的页表寄存器指向物理存储器中的页表
- 如果页在存储器中
  - PTE存储物理页号
  - 加上其他状态位（引用位、脏位等）
- 如果该页不存在
  - PTE能指向它在磁盘交换区中的位置

使用页表进行转换


Page table register
Virtual address
47 46 45 44 43 ····· 15 14 13 12 11 10 9 8 ····· 3 2 1 0
Virtual page number
Page offset
36
12
Valid
Physical page number
Page table
28
If 0 then page is not present in memory
39 38 37 ····· 15 14 13 12 11 10 9 8 ····· 3 2 1 0
Physical page number
Page offset
Physical address

将页映射到存储器


Virtual page number
Page table
Physical page or disk address
Valid
1
1
1
1
0
1
1
0
1
1
0
1
Physical memory
Disk storage

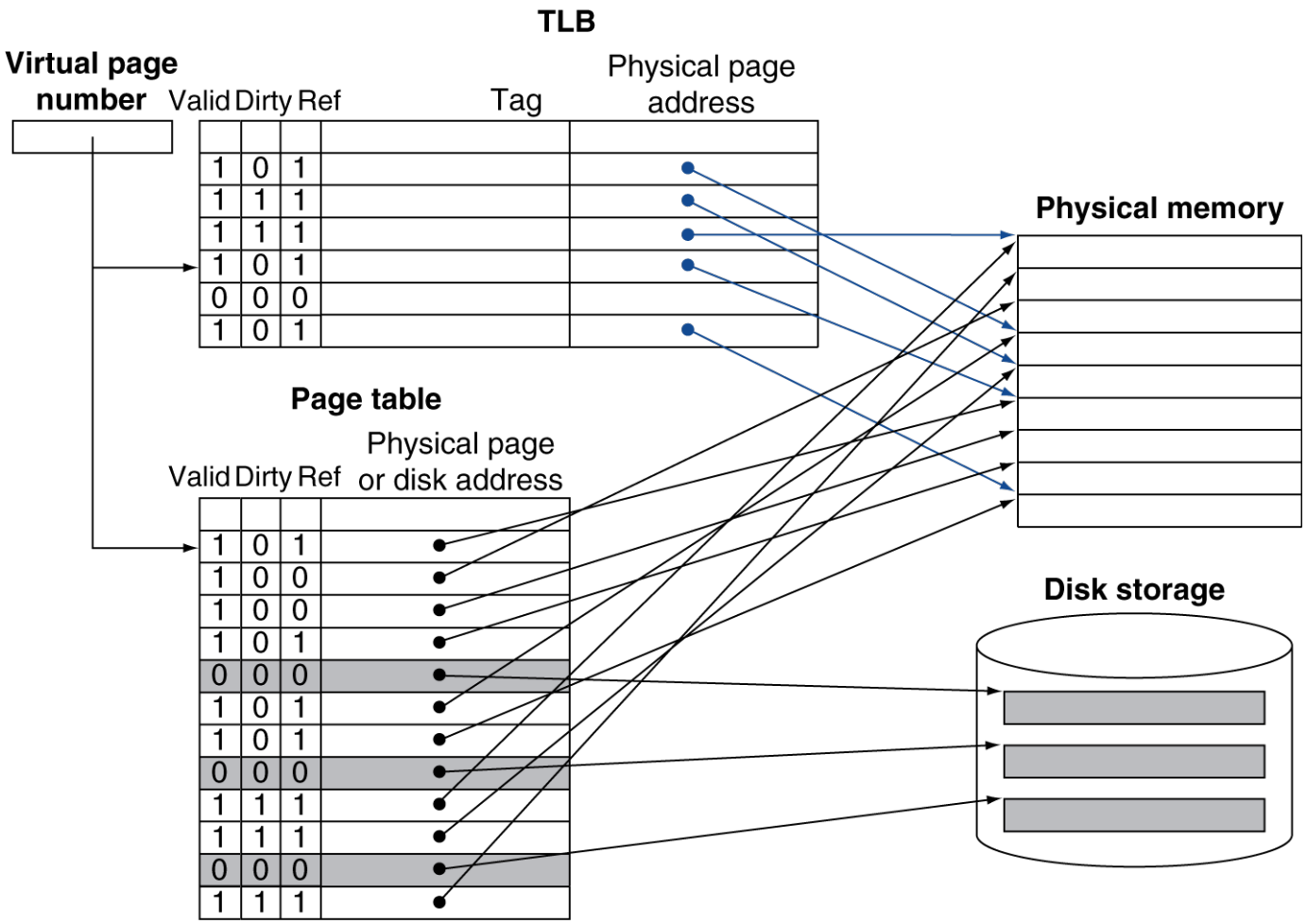
替换和写

- 为了降低缺页率，一般采用最近最少使用(LRU)替换策略
  - 访问页时将PTE中的引用位（亦称使用位）置1
  - 由OS周期性地清0
  - 引用位为0的页最近未被使用
- 写磁盘花费数百万时钟周期
  - 每次写一个块，而不是单一的位置
  - 写直达是不可行的
  - 使用写回
  - 当页被写时，将PTE中的脏位置位

使用TLB进行快速转换

- 看起来地址转换需要额外的存储器访问
  - 一次是访问PTE
  - 然后是访问实际的存储器
- 但访问页表有着很好的局部性
  - 因此使用CPU中PTE的cache
  - 称为快表(Translation Look-aside Buffer, TLB)
  - 典型值：16–512个PTE项，0.5–1个周期的命中时间，10–100个周期的缺失时间，0.01%–1%的缺失率
  - 可以由硬件或软件来处理缺失

使用TLB进行快速转换



TLB缺失

- 如果页在内存中
  - 从内存读PTE，然后重试
  - 可以由硬件处理
    - 对于更复杂的页表结构，硬件也变复杂
  - 或用软件处理
    - 产生一次特殊的异常，调用优化的处理程序
- 如果页不在内存中（缺页）
  - OS进行取页并更新页表
  - 然后重启引起缺页的指令

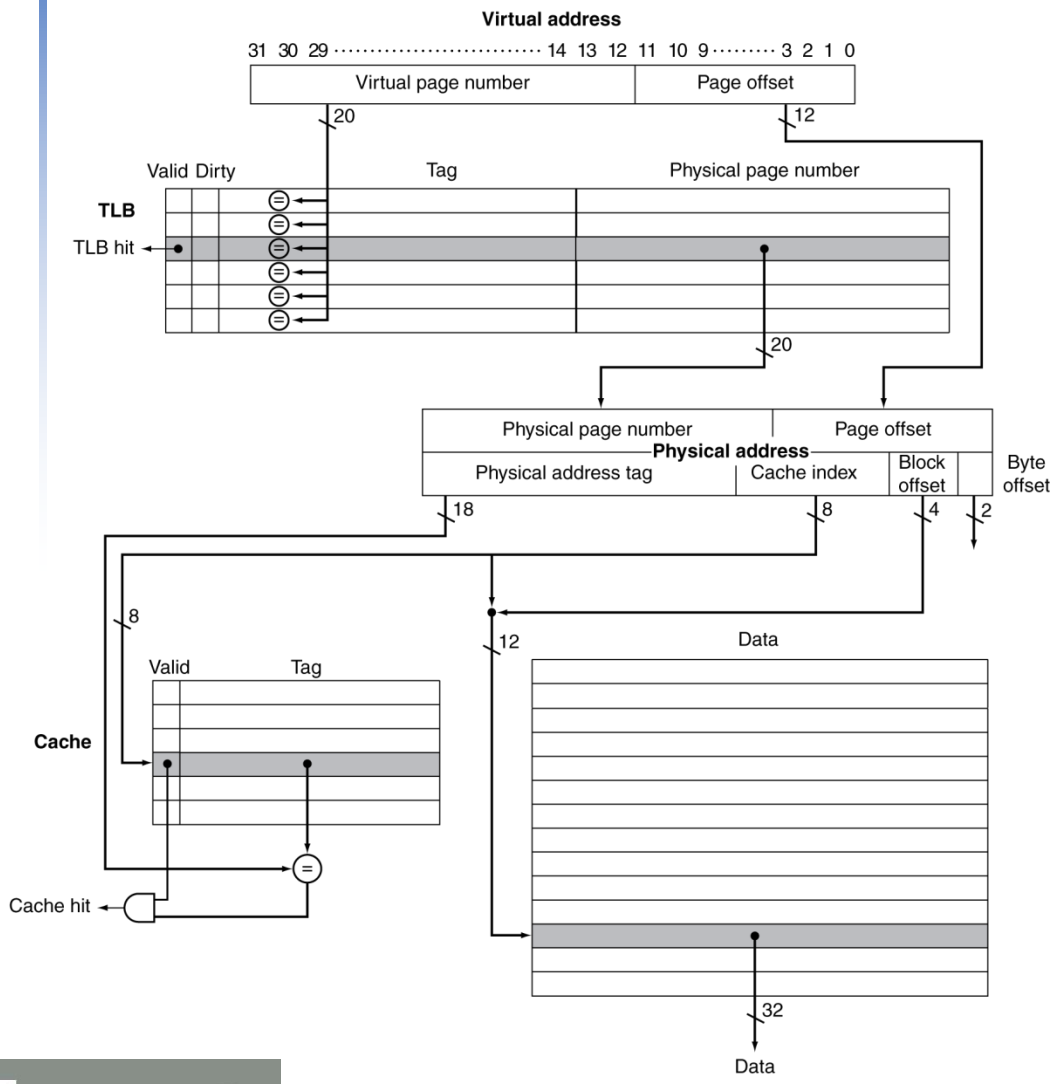
TLB缺失处理程序

- TLB缺失意味着
  - 页是存在的，但PTE不在TLB中
  - 页不存在
- 必须在目标寄存器被覆盖前识别出TLB缺失
  - 产生异常
- 处理程序将PTE从内存复制到TLB
  - 然后重启指令
  - 如果该页不存在，将发生缺页

缺页处理程序

- 使用缺失的虚拟地址来查找PTE
- 在磁盘上定位页
- 选一页进行替换
  - 如果是脏页，先把它写到磁盘
- 将页读入存储器并更新页表
- 使进程可再次运行
  - 从引起缺页的指令重启

TLB与cache的交互



- 如果cache的标记使用物理地址
  - 需要在查找cache之前进行转换
- 或者使用虚拟地址标记
  - 重名会引起复杂情况
    - 共享的物理地址有不同的虚拟地址

存储器保护

- 不同任务可以共享其部分虚拟地址空间
  - 但需要保护，防止不当访问
  - 需要OS协助
- 硬件对OS保护的支持
  - 特权超级用户模式（亦称内核模式）
  - 特权指令
  - 页表和其他状态信息只在超级用户模式下可访问
  - 系统调用异常（例如RISC-V中的ecall）

存储器层次结构

重点

- 应用于存储器层次结构所有层级的普遍原则
  - 基于缓存的概念
- 在层次结构的每个层级中
  - 块的放置
  - 查找一个块
  - 缺失时的替换
  - 写的策略

块的放置

- 取决于相联度
  - 直接映射（1路相联）
    - 1个放置选择
  - *n*路组相联
    - 1组中有*n*种选择
  - 全相联
    - 任意位置
- 提升相联度可以降低缺失率
  - 但增加了复杂度、成本和访问时间

查找一个块

相联度	定位方法	比较标记的次数
直接映射	索引	1
*n*路组相联	组索引，然后查找组内的项	*n*
全相联	查找所有的项	项数
	整个查找表	0

- 硬件cache
  - 减少比较的次数以降低成本
- 虚拟存储器
  - 整表查找使得全相联可行
  - 有利于降低缺失率

替换

- 发生缺失时如何选择替换哪一项
  - 最近最少使用(LRU)
    - 对于高相联度，硬件复杂而且成本高
  - 随机
    - 接近LRU，容易实现
- 虚拟存储器
  - 硬件支持近似LRU的算法

写的策略

- 写直达
  - 高低两层同时更新
  - 简化了替换，但可能需要写缓冲
- 写回
  - 只更新高层
  - 块被替换时才更新低层
  - 需要保存更多状态信息
- 虚拟存储器
  - 鉴于写磁盘的延迟，只有写回是可行的

缺失的来源

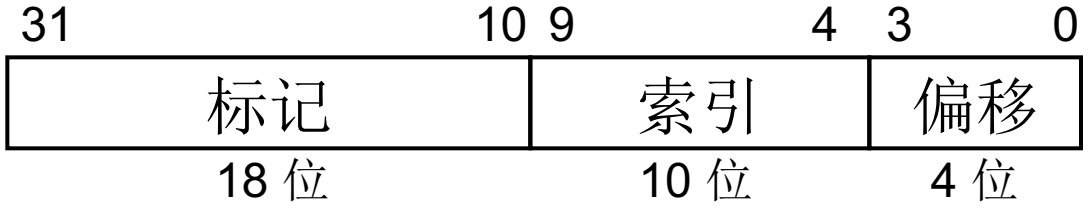
- 强制缺失（亦称冷启动缺失）
  - 首次访问一个块时
- 容量缺失
  - 由于cache的大小有限
  - 被替换掉的块随后又被访问到
- 冲突缺失（亦称碰撞缺失）
  - 在非全相联的cache中
  - 由于竞争同一组中的项
  - 不会发生在同样大小的全相联cache中

cache设计的取舍

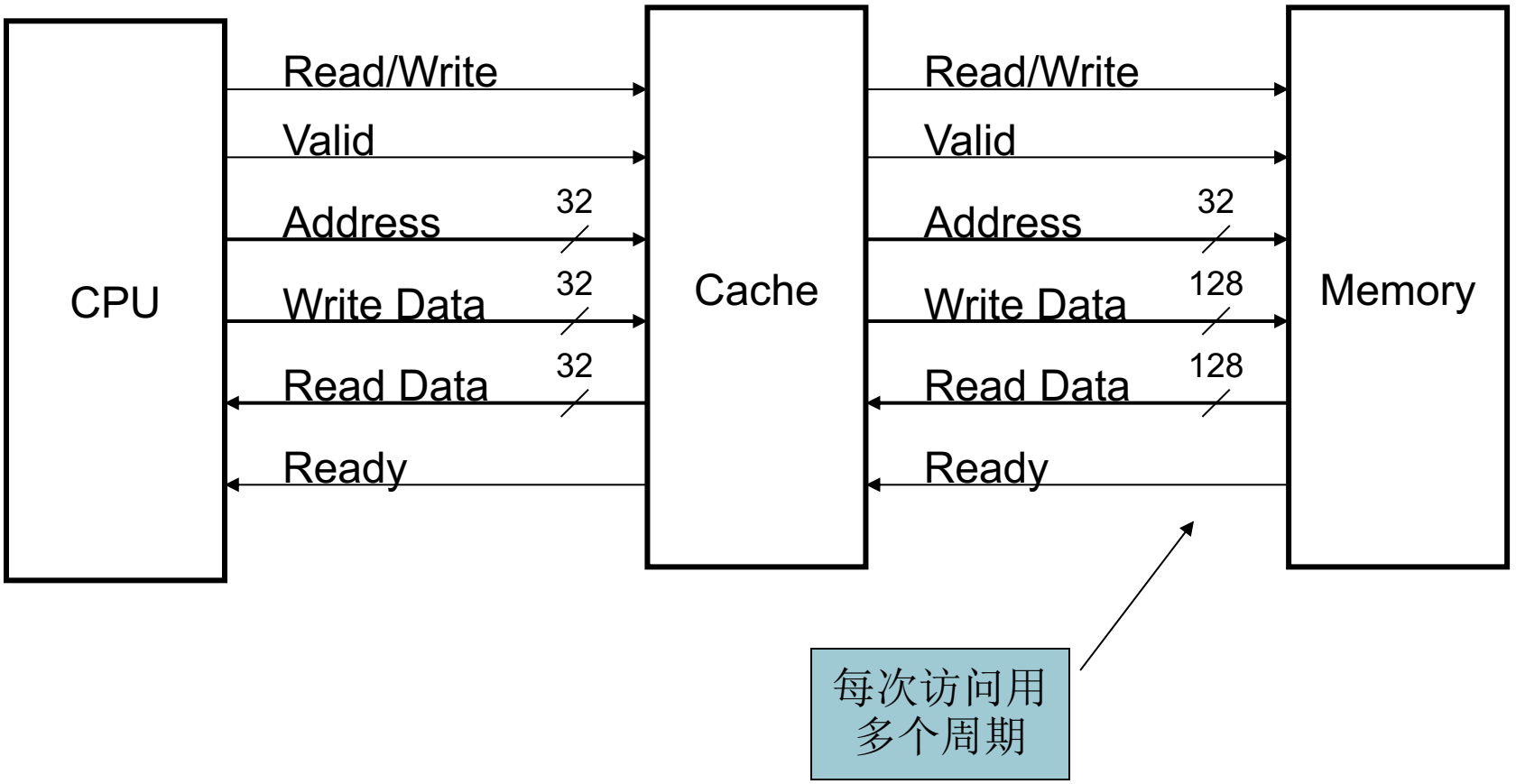
设计变化	对缺失率的影响	对性能的负面影响
增加cache容量	减少了容量缺失	可能增加访问时间
提高相联度	减少了冲突缺失	可能增加访问时间
增加块大小	减少了强制缺失	增加缺失代价。 对非常大的块来说，可能因污染而增加缺失率。

cache的控制

- cache特性示例
  - 直接映射，写回，写分配
  - 块的大小：4字（16字节）
  - cache的大小：16 KB（1024个块）
  - 32位字节地址
  - 每个块包含有效位和脏位
  - 阻塞式cache
    - CPU等待直到访问完成

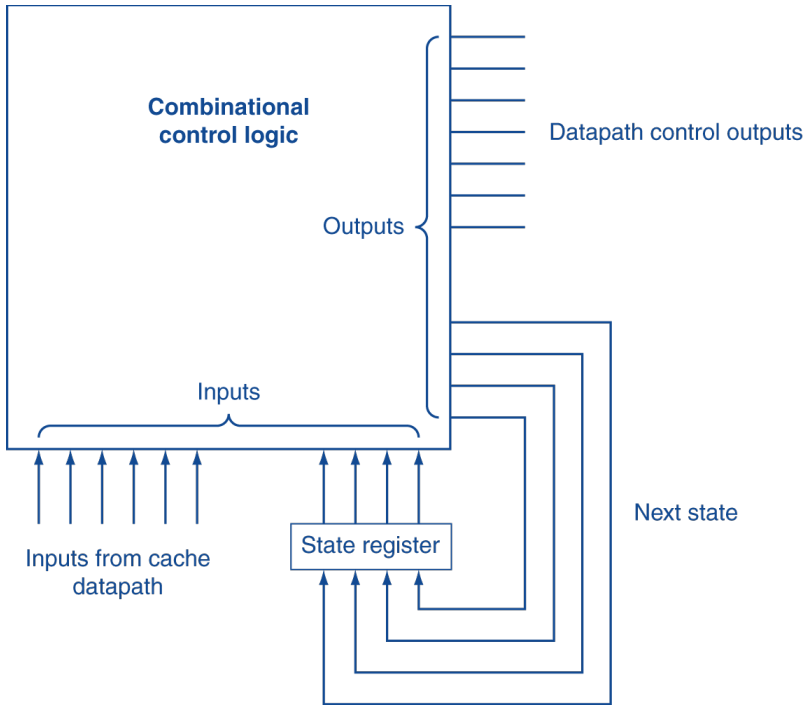


接口信号

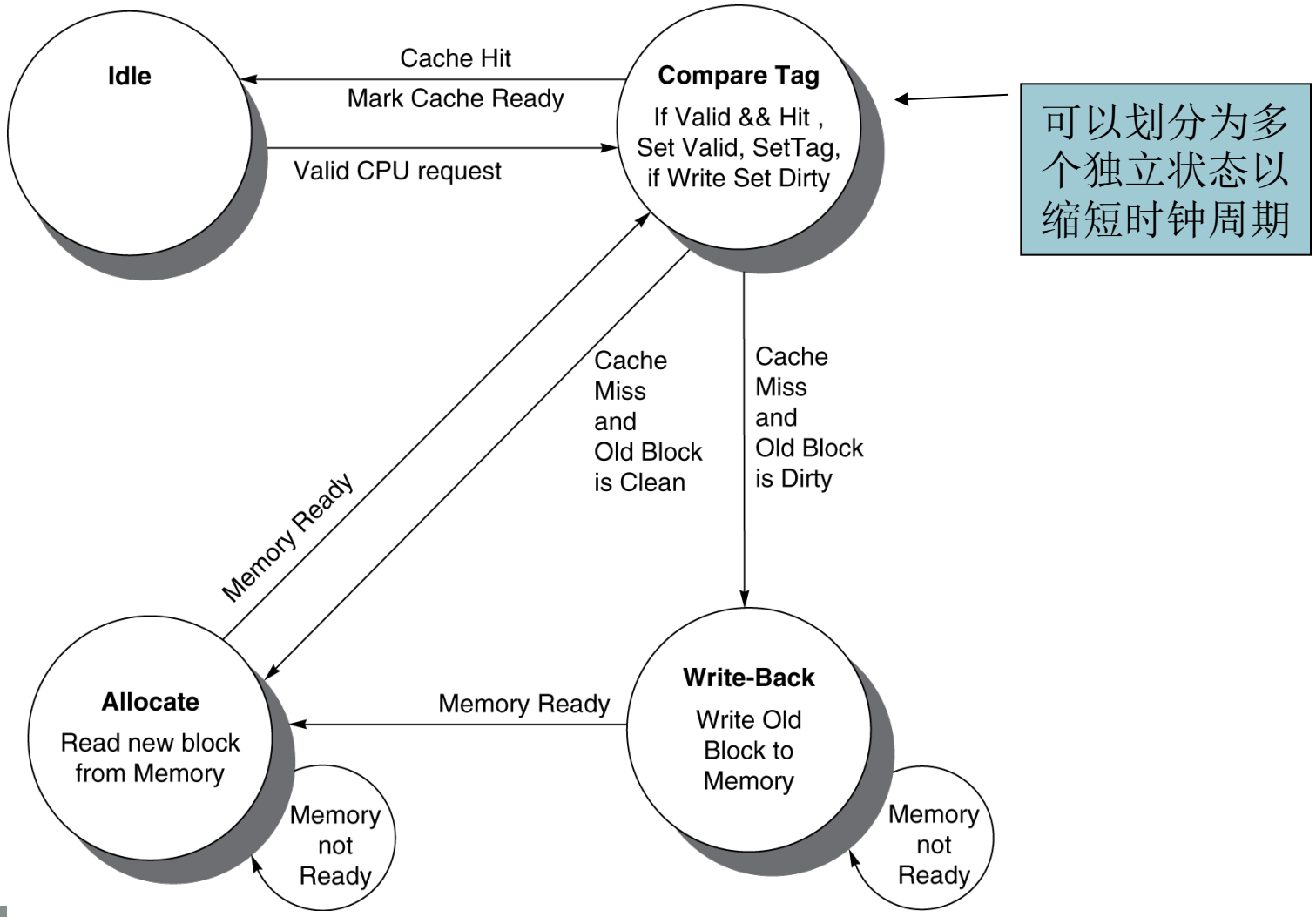

CPU
Read/Write
Valid
Address
32
Write Data
32
Read Data
32
Ready
Cache
Read/Write
Valid
Address
32
Write Data
128
Read Data
128
Ready
Memory
每次访问用
多个周期

有限状态机

- 用一个FSM排列各控制步骤
- 一组状态，在每个时钟沿改变
  - 状态值以二进制编码
  - 用一个寄存器来存储当前状态
  - 下一状态 = f_n (当前状态，当前输入)
- 控制输出信号 = f_o (当前状态)



cache控制器FSM



cache一致性问题

- 假定两个CPU核共享同一物理地址空间
  - 写直达cache

时间步数	事件	CPU A的cache	CPU B的cache	存储器
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

一致性的定义

- 非正式的定义：读操作返回最新写入的值
- 正式定义：
  - P写X；P读X（其间没有写操作）
    ⇒ 读操作返回写入的值
  - P_1写X；P_2读X （充分长时间后）
    ⇒ 读操作返回写入的值
    - 对比：前例中CPU B在第3步之后读X
  - P_1写X，P_2写X
    ⇒ 所有处理器看到的写操作顺序相同
    - 最后看到X的最终值相同

cache一致性协议

- 多处理器的cache为了保证一致性而执行的操作
  - 数据到本地cache的迁移
    - 减少对共享存储器带宽的需求
  - 读共享数据的复制
    - 减少由访问导致的竞争
- 监听协议
  - 每个cache监视总线的读/写
- 基于目录的协议
  - cache和存储器记录一个目录中块的共享状态

无效化监听协议

- cache要写一个块时，获得独占访问
  - 在总线上广播一条无效信息
  - 随后，对另一个cache的读操作发生缺失
    - 所有者cache提供更新后的值

CPU动作	总线动作	CPU A的cache	CPU B的cache	存储器
				0
CPU A 读 X	X在cache中缺失	0		0
CPU B 读 X	X在cache中缺失	0	0	0
CPU A 向 X 写 1	令X无效	1		0
CPU B 读 X	X在cache中缺失	1	1	1

存储器一致性

- 在什么时候写操作被其他处理器看见
  - “看见”指一次读操作返回了写入值
  - 无法是瞬时的
- 假定
  - 仅当所有处理器都看见一个写操作时，写操作才完成
  - 处理器不会重排写操作与其他访问操作的顺序
- 结果
  - P先写X，然后写Y
    ⇒ 所有看见新Y值的处理器也看见新X值
  - 处理器可以改变读的顺序，但不能改变写的顺序

片上多级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

二级TLB组织

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 Both micro TLBs are fully associative, with 10 entries, round robin replacement 64-entry, four-way set-associative TLBs TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, seven per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

支持多发射

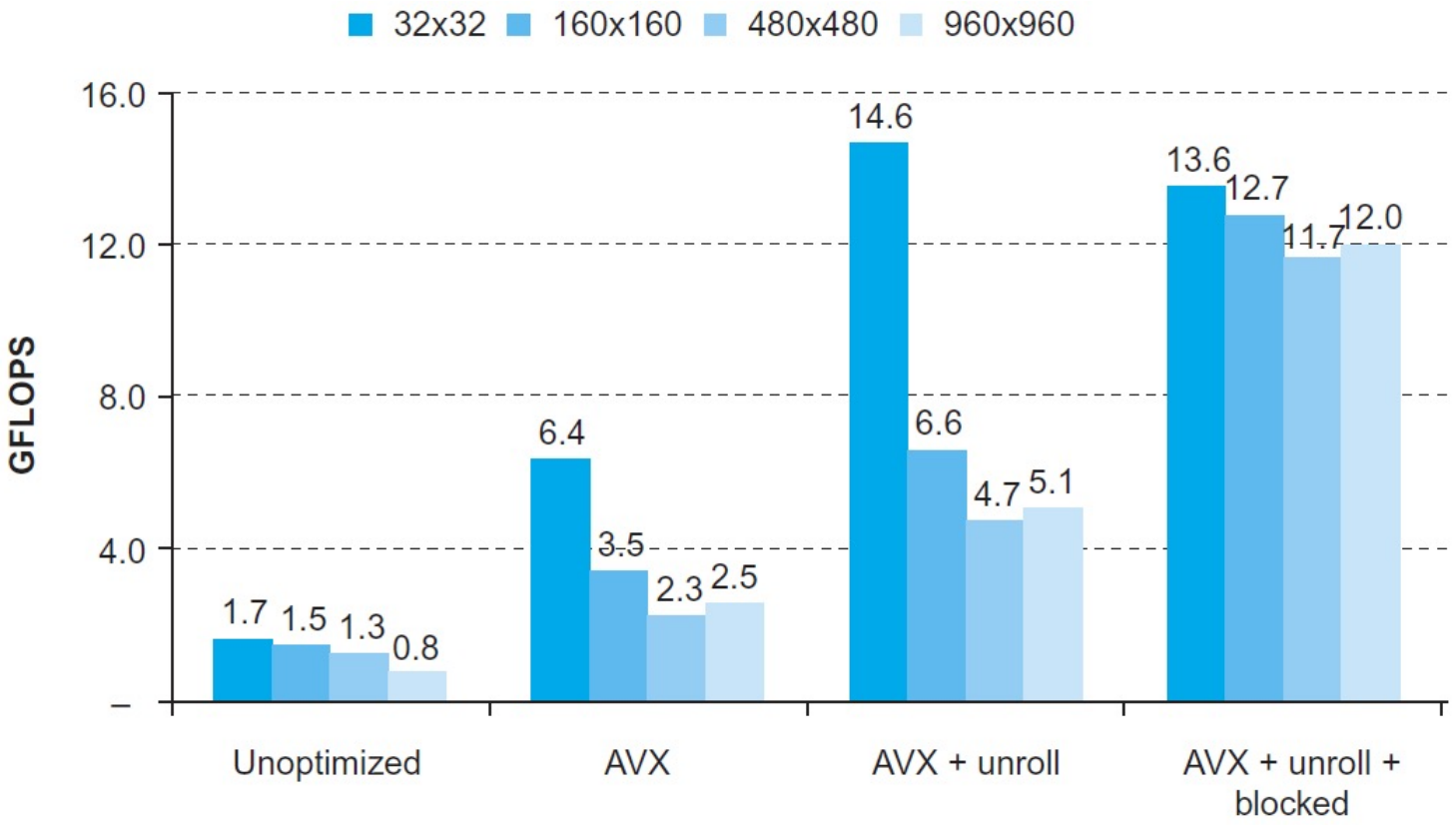
- 都具有多体cache，在bank不发生冲突时支持每个周期多个访问
- Core i7 cache优化
  - 首先返回请求字
  - 非阻塞cache
    - 缺失时命中(hit under miss)
    - 缺失时缺失(miss under miss)
  - 数据预取

RISC-V系统指令

类型	助记符	名称
内存访问排序	fence.i	Instruction fence
	fence	Fence
	sfence.vm	Address translation fence
控制状态寄存器访问	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
系统	ecall	Environment call
	ebreak	Environment breakpoint
	sret	Supervisor exception return
	wfi	Wait for interrupt

DGEMM

- 结合cache分块和子字并行



陷阱

- 字节编址与字编址
  - 例：32字节直接映射cache，4字节的块
    - 字节36映射到块1
    - 字36映射到块4
- 写程序或生成代码时忽略存储器系统的影响
  - 例：遍历数组是按行还是按列
  - 大间隔导致局部性变差

陷阱

- 对于共享二级或三级cache的多处理器
  - 相联度少于核数导致冲突缺失
  - 更多的核 ⇒ 需要增大相联度
- 使用AMAT评估乱序处理器的性能
  - 忽略了非阻塞访问的影响
  - 代之以仿真来评估性能

陷阱

- 用多段来扩展地址范围
  - 例如，Intel 80286
  - 但一个段有时不够大
  - 使寻址计算复杂化
- 在不为虚拟化设计的ISA上实现VMM
  - 例如，非特权指令访问硬件资源
  - 要么扩展ISA，要么要求客户OS不使用会产生问题的指令

本章小结

- 快速存储器容量小，大容量存储器速度慢
  - 我们真想要又大又快的存储器☹
  - 缓存能产生这种假象☺
- 局部性原理
  - 程序频繁使用其存储空间中的一小部分
- 存储器层次结构
  - 一级cache ↔ 二级cache ↔ … ↔ DRAM存储器 ↔ 磁盘
- 对多处理器来说，存储器系统的设计很关键