计算机组织结构

9 高速缓冲存储器（Cache）

任桐炜

2021年10月21日

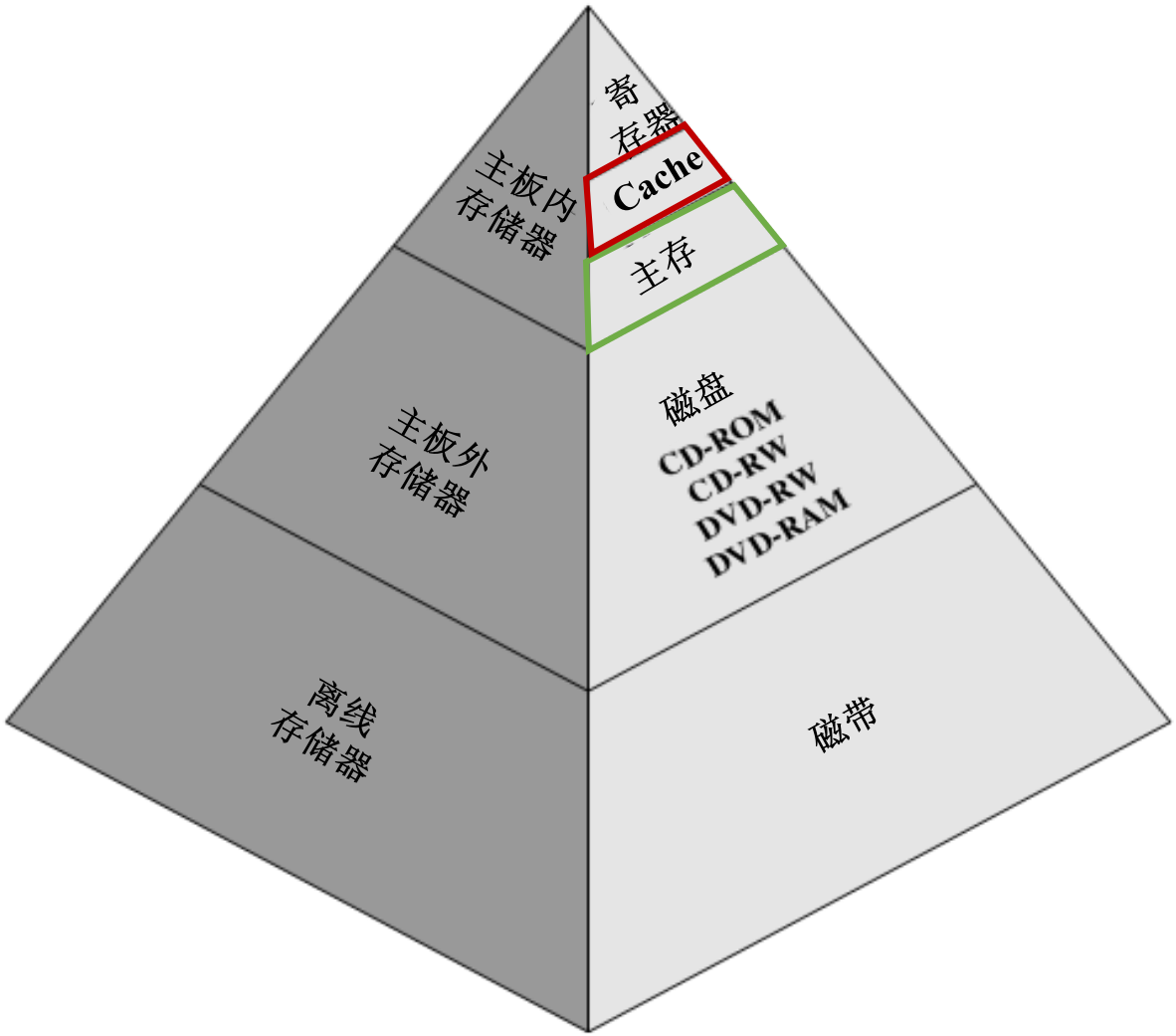
南京大学 NANJING UNIVERSITY

教材对应章节

第7章 存储器分层体系结构

第4章 cache存储器

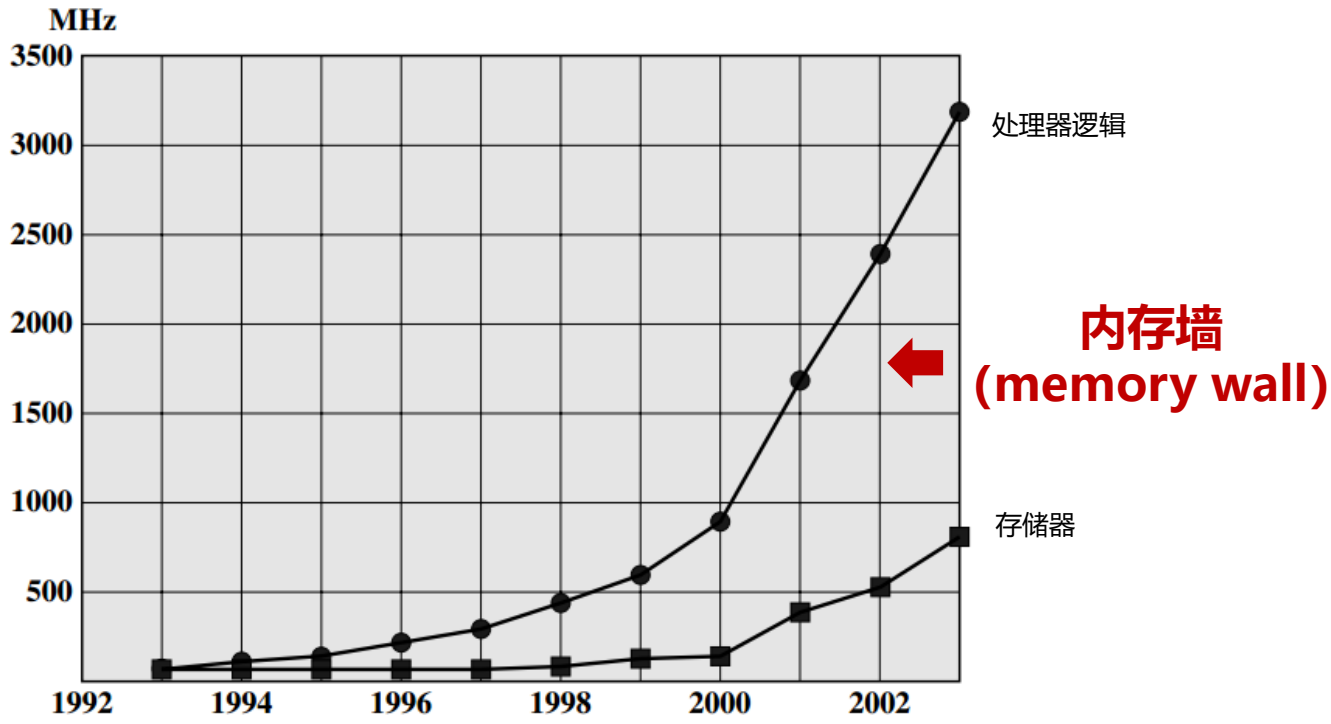
存储器层次结构


寄存器
Cache
主存
磁盘
CD-ROM
CD-RW
DVD-RW
DVD-RAM
磁带
主板内存储器
主板外存储器
离线存储器

回顾：内存墙

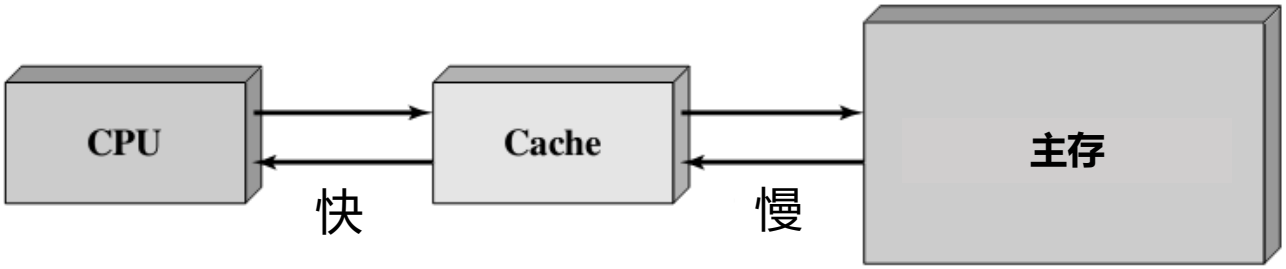
内存墙

- 问题：CPU的速度比内存的速度快，且两者差距不断扩大



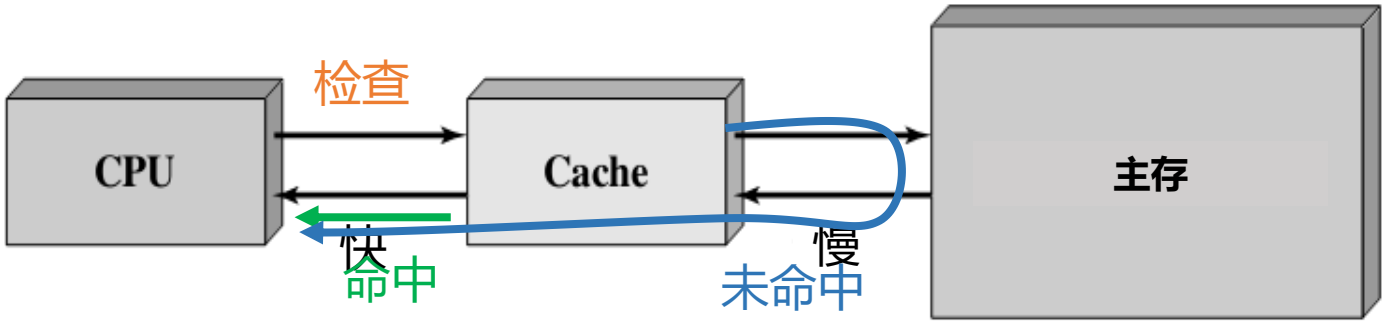
Cache的基本思路

- 解决内存墙带来的CPU和主存协作问题
  - 在使用主存（相对大而慢）之余，添加一块小而快的cache
  - Cache位于CPU和主存之间，可以集成在CPU内部或作为主板上的一个模块
  - Cache中存放了主存中的部分信息的副本



Cache的工作流程

- **检查（Check）**：当CPU试图访问主存中的某个字时，首先检查这个字是否在cache中
- 检查后分两种情况处理：
  - **命中（Hit）**：如果在cache中，则把这个字传送给CPU
  - **未命中（Miss）**：如果不在cache中，则将主存中包含这个字固定大小的块（block）读入cache中，然后再从cache传送该字给CPU



问题

- 如何判断是命中还是未命中?
- 如果未命中，为什么不直接把所需要的字从内存传送到CPU?
- 如果未命中，为什么从内存中读入一个块而不只读入一个字?
- 使用Cache后需要更多的操作，为什么还可以节省时间?

命中和未命中的判断

- 冯·诺伊曼体系的设计
  - CPU通过**位置**对主存中的内容进行寻址，不关心存储在其中的内容
- Cache通过**标记（tags）**来标识其内容在主存中的对应**位置**

问题

- 如何判断是命中还是未命中?
- 如果未命中，为什么不直接把所需要的字从内存传送到CPU?
- 如果未命中，为什么从内存中读入一个块而不只读入一个字?
- 使用Cache后需要更多的操作，为什么还可以节省时间?

程序访问的局部性原理

- 定义：
  - 处理器频繁访问主存中相同位置或者相邻存储位置的现象（维基百科）
- 类型
  - **时间局部性**：在相对较短的时间周期内，重复访问特定的信息（也就是访问相同位置的信息）
  - **空间局部性**：在相对较短的时间周期内，访问相邻存储位置的数据
    - **顺序局部性**：当数据被线性排列和访问时，出现的空间局部性的一种特殊情况
      - 例如：遍历一维数组中的元素

局部性原理的示例

- 时间局部性

```
int factorial = 1;
for (int i = 2; i <= n; i++) {
    factorial = factorial * i;
}
```

- 空间局部性

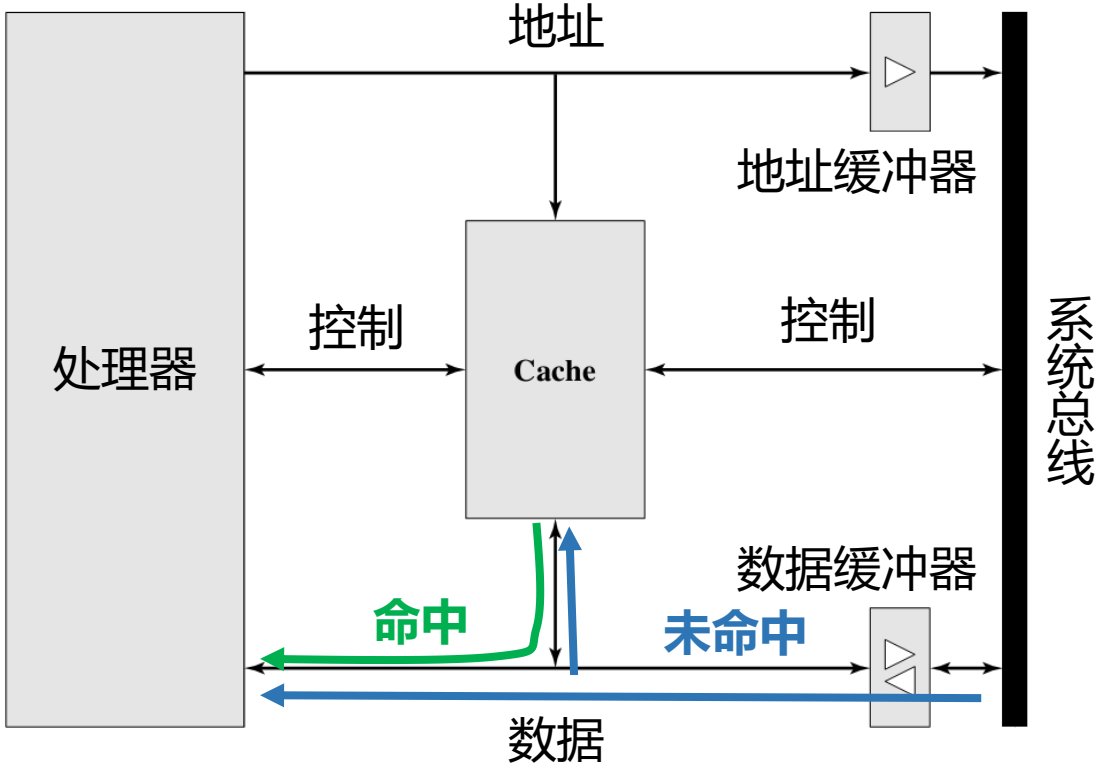
```
for (int i = 0; i < num; i++) {
    score[i] = final[i] * 0.4 + midterm[i] * 0.3 + assign[i] * 0.2 + activity[i] * 0.1;
}
```





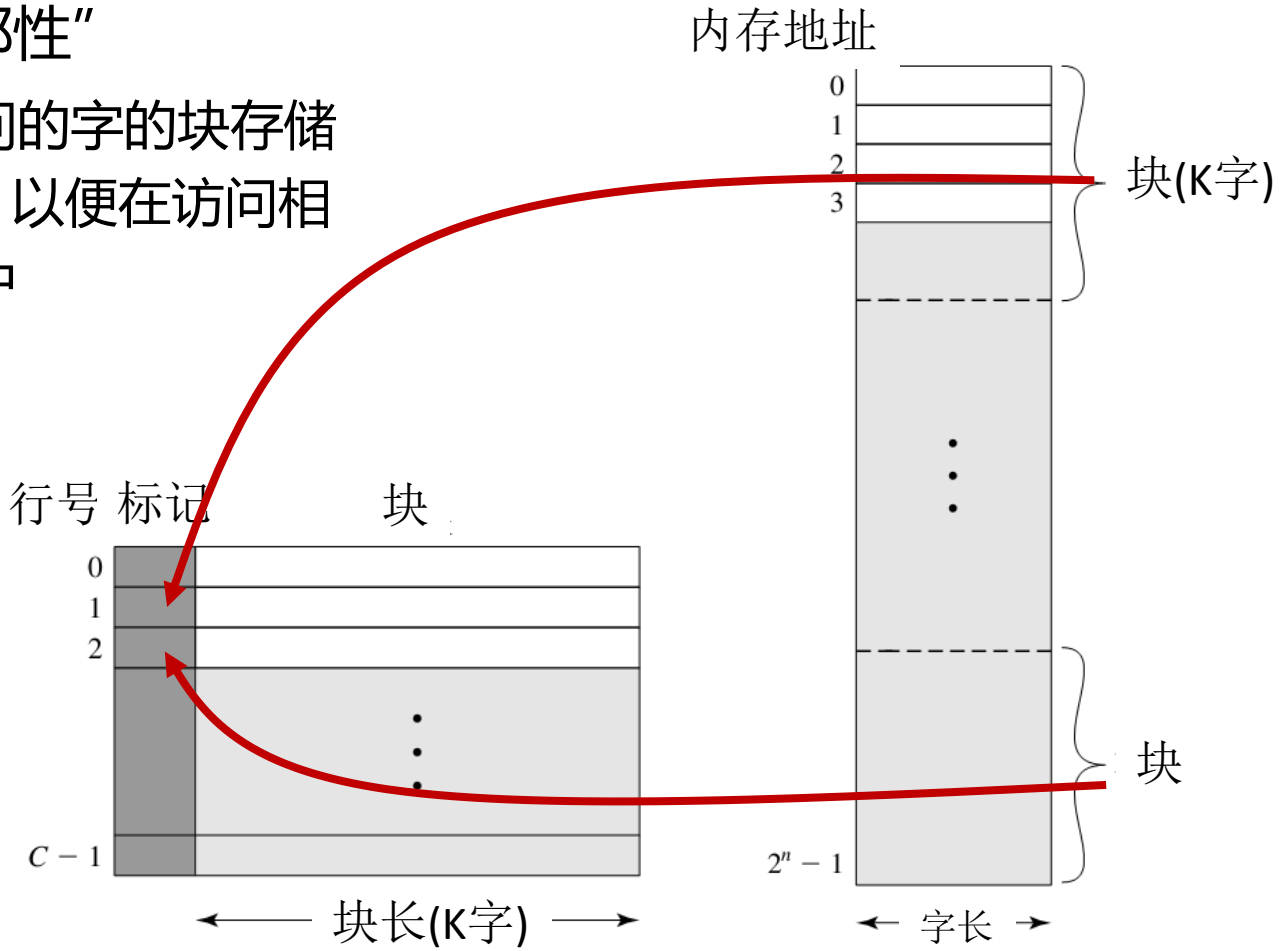
向Cache传送内容

- 利用“时间局部性”
  - 将未命中的数据在返回给CPU的同时存放在Cache中，以便再次访问时命中



传送块而不是传送字

- 利用“空间局部性”
  - 将包含所访问的字的块存储到Cache中，以便在访问相邻数据时命中



问题

- 如何判断是命中还是未命中?
- 如果未命中，为什么不直接把所需要的字从内存传送到CPU?
- 如果未命中，为什么从内存中读入一个块而不只读入一个字?
- 使用Cache后需要更多的操作，为什么还可以节省时间?

平均访问时间

- 假设p是**命中率**， T_C 是cache的访问时间， T_M 是主存的访问时间，使用cache时的平均访问时间为

$$T_A = p \times T_C + (1-p) \times (T_C + T_M)$$

$$= T_C + (1-p) \times T_M$$

- 命中率p越大， T_C越小， 效果越好
- 如果想要 $T_A < T_M$, 必须要求

$$p > T_C \,/\, T_M$$

- 难点：cache的容量远远小于主存的容量

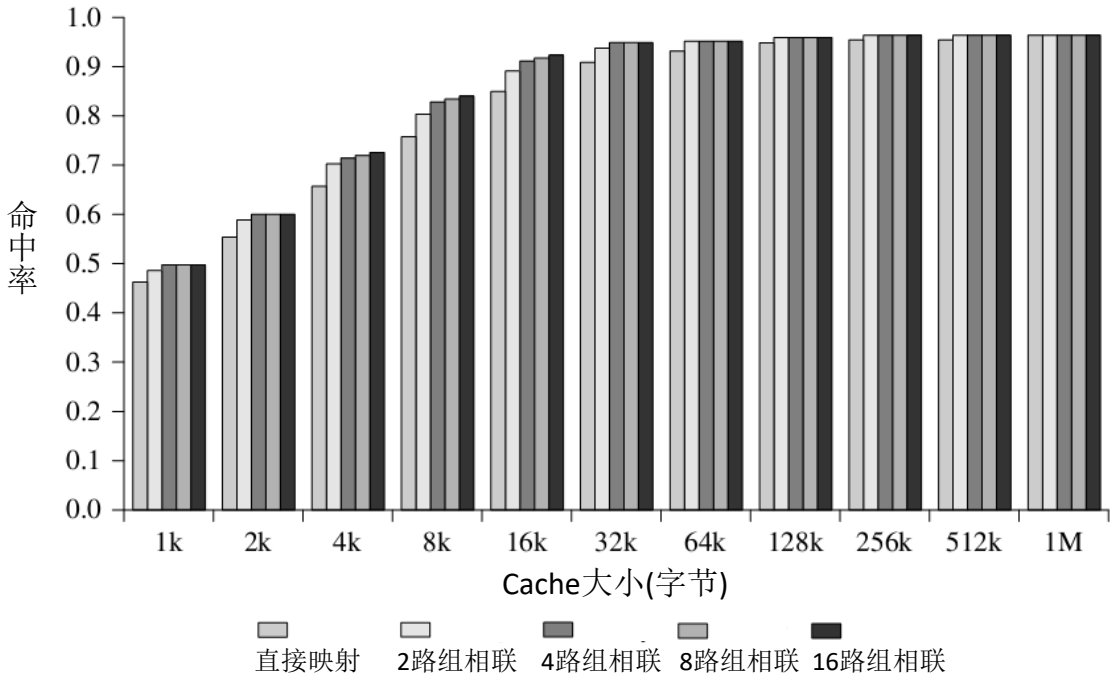
Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- 行大小
- Cache数目




Cache容量

- 扩大cache容量带来的结果：
  - 增大了命中率p
  - 增加了cache的开销和访问时间 T_C



Cache容量

- 一些处理器的cache容量

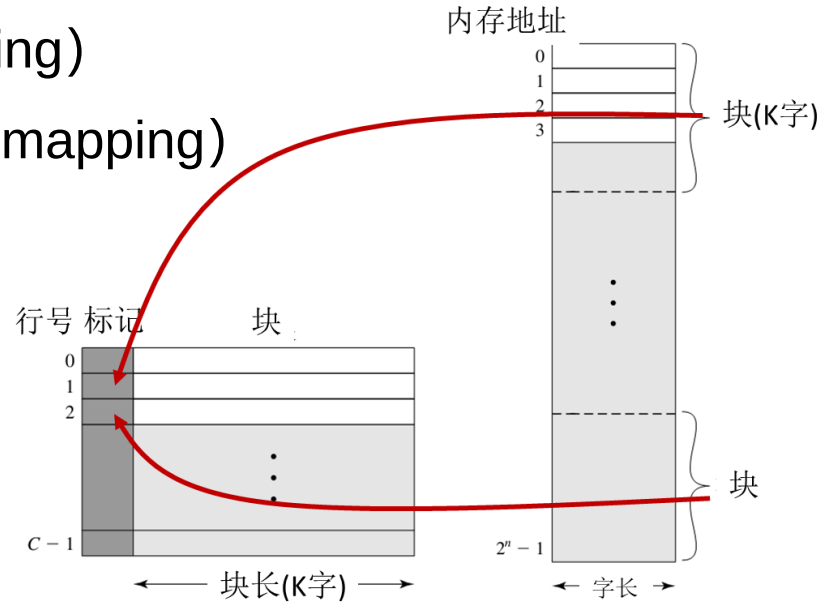
处理器	类型	推出年份	L1 cache①	L2 cache	L3 cache
IBM 360/85	大型机	1968	16～32kB	—	—
PDP-11/70	小型机	1975	1kB	—	—
VAX 11/780	小型机	1978	16kB	—	—
IBM 3033	大型机	1978	64kB	—	—
IBM 3090	大型机	1985	128～256kB	—	—
Intel 80486	PC	1989	8kB	—	—
Pentium	PC	1993	8kB/8kB	256～512KB	—
PowerPC 601	PC	1993	32kB	—	—
PowerPC 620	PC	1996	32kB/32kB	—	—
PowerPC G4	PC/服务器	1999	32kB/32kB	256KB～1MB	2MB
IBM S/390 G4	大型机	1997	32kB	256KB	2MB
IBM S/390 G6	大型机	1999	256kB	8MB	—
Pentium 4	PC/服务器	2000	8kB/8kB	256KB	—
IBM SP	高端服务器/超级计算机	2000	64kB/32kB	8MB	—
CRAY MTA②	超级计算机	2000	8kB	2MB	—
Itanium	PC/服务器	2001	16kB/16kB	96KB	4MB

Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- 行大小
- Cache数目

映射功能（Mapping Function）

- 实现主存块到cache行的映射
- 块号，块内地址
- 映射方式的选择会影响cache的组织结构
  - 直接映射（Direct mapping）
  - 关联映射（Associative mapping）
  - 组关联映射（Set associative mapping）



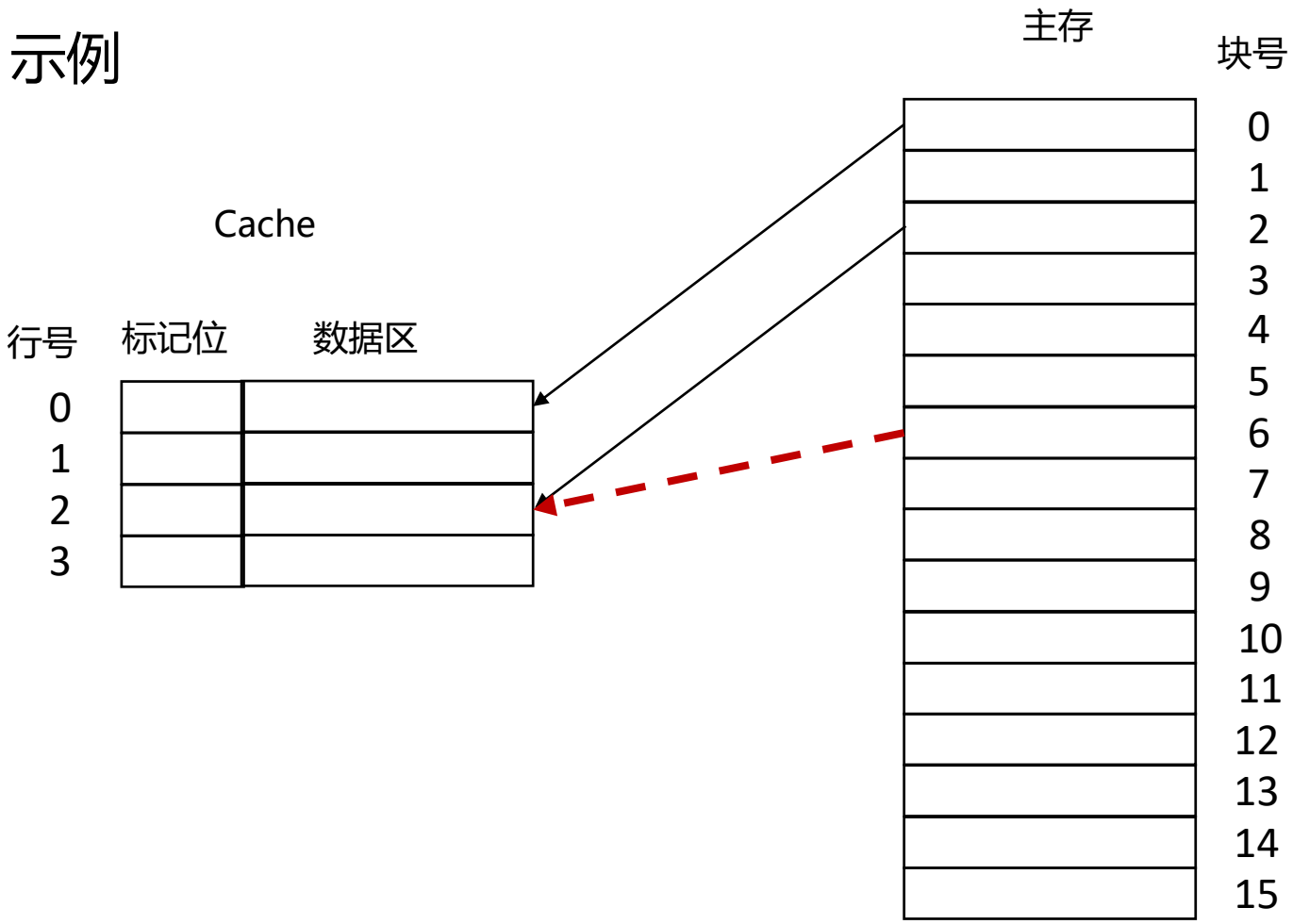
直接映射

- 将主存中的每个块映射到**一个固定可用的**cache行中
- 假设i是cache行号，j是主存储器的块号，C是cache的行数

$$i = j \ mod \ C$$

直接映射

- 示例



直接映射

- 标记
  - 地址中最高n位, $n = log_2 M - log_2 C$

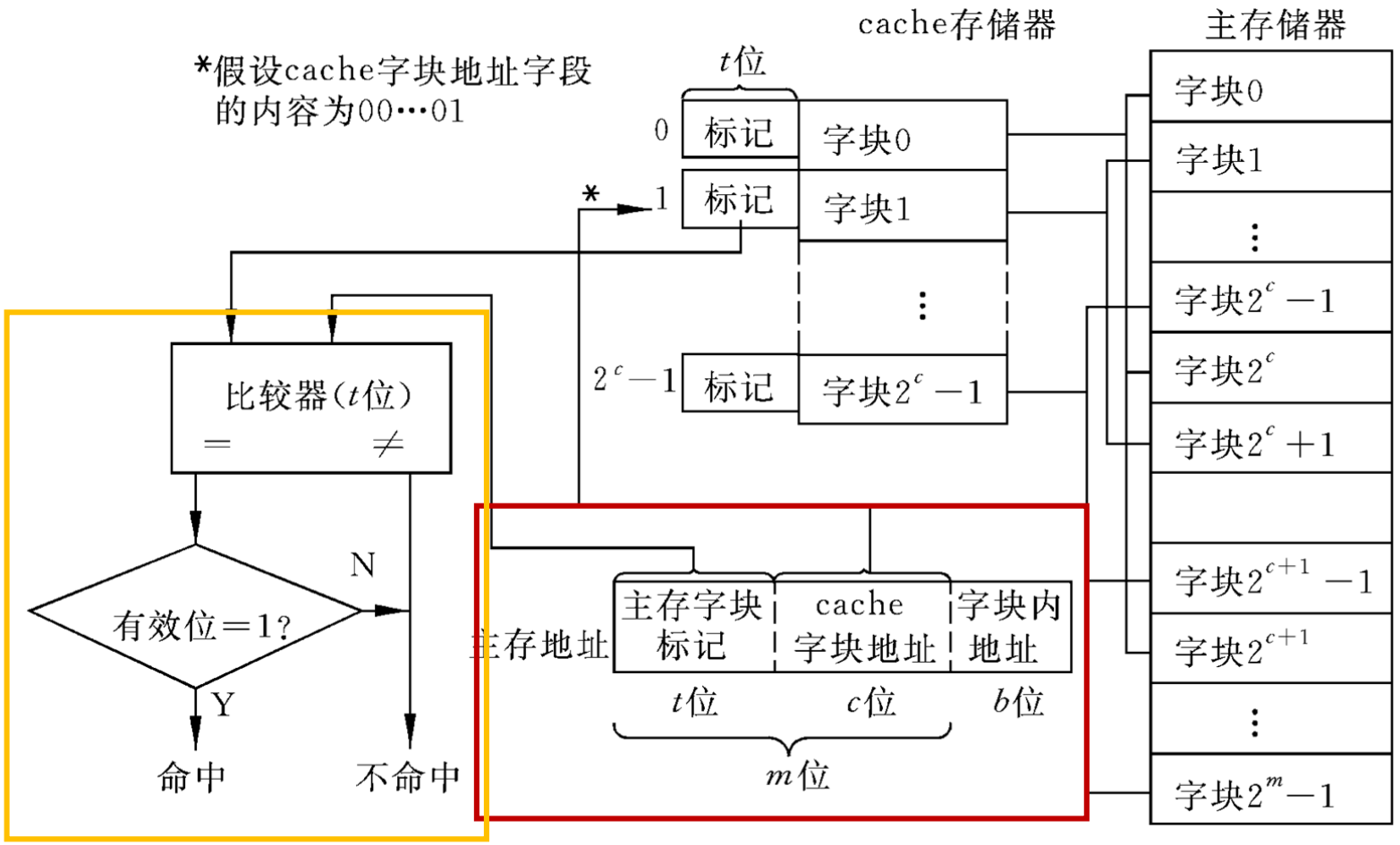
主存地址	标记	Cache行号	块内地址

- 例
  - 假设cache有4行，每行包含8个字；主存中包含128个字。访问主存的地址长度为7位，则：
    - 最低的3位：块内地址
    - 中间的2位：映射时所对应的Cache行号
    - 最高的2位：区分映射到同一行的不同块，记录为Cache标记

直接映射

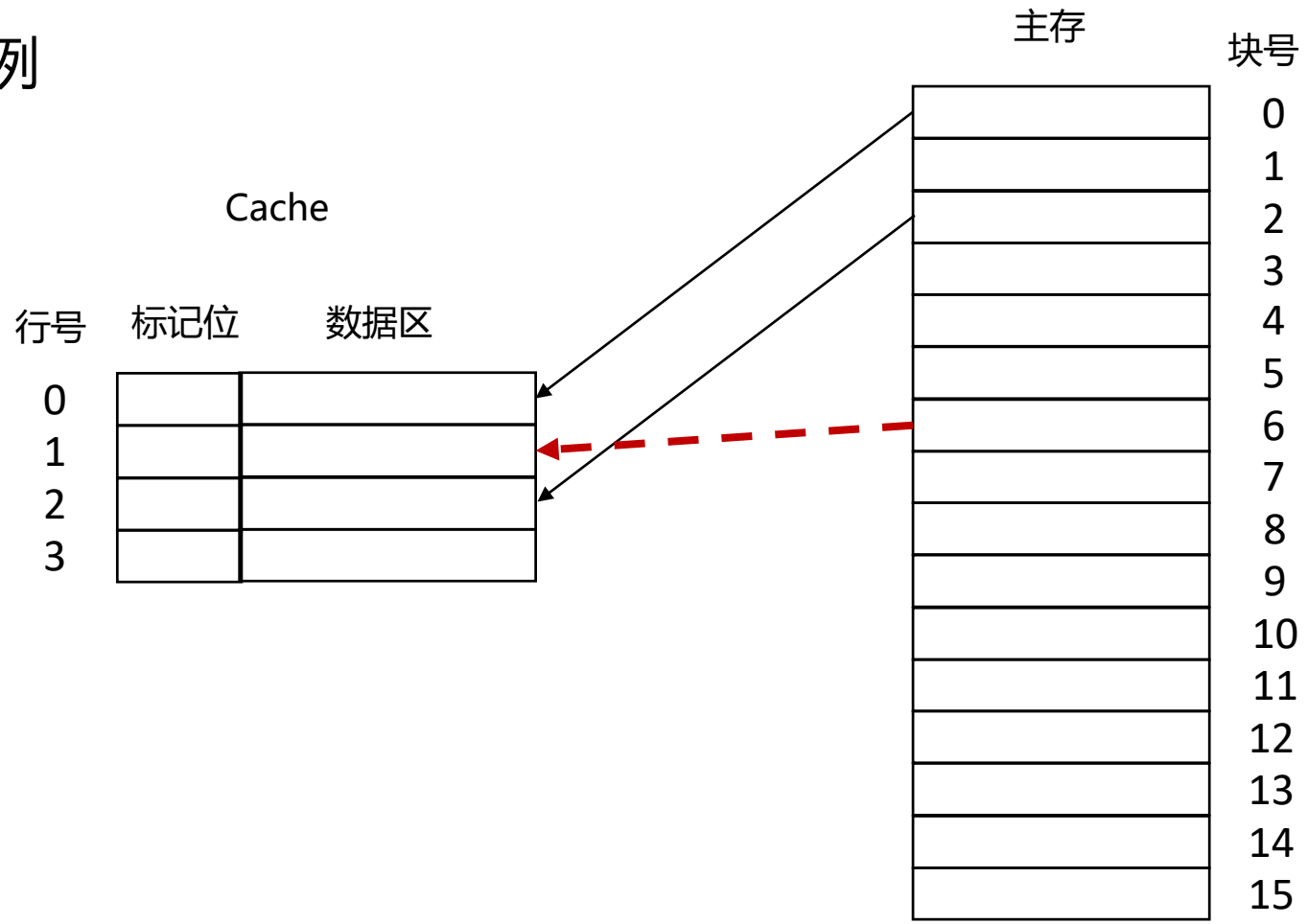
- 优点
  - 简单
  - 快速映射
  - 快速检查
- 缺点
  - 抖动现象（Thrashing）：如果一个程序重复访问两个需要映射到同一行中且来自不同块的字，则这两个块不断地被交换到cache中，cache的命中率将会降低
- 适合大容量的cache

直接映射


*假设cache字块地址字段的内容为00…01
cache存储器
主存储器
t位
0
标记
字块0
1
标记
字块1
2^c−1
标记
字块2^c−1
*
比较器(t位)
=
≠
有效位=1?
N
Y
命中
不命中
主存地址
主存字块标记
cache字块地址
字块内地址
t位
c位
b位
m位
字块0
字块1
字块2^c−1
字块2^c
字块2^c+1
字块2^c+1−1
字块2^c+1
字块2^m−1

关联映射

- 一个主存块可以装入cache**任意一行**
- 示例



关联映射

- 标记
  - 地址中最高n位, $n = log_2 M$

- 示例
  - 假设cache有4行，每行包含8个字；主存中包含128个字。访问主存的地址长度为7位，则：
    - 最低的3位：块内地址
    - 最高的4位：块号，记录为Cache标记

关联映射

- 优点
  - 避免抖动
- 缺点
  - 实现起来比较复杂
  - Cache搜索代价很大，即在检查的时候需要去访问cache的每一行
- 适合容量较小的cache

组关联映射

- Cache分为若干组，每一组包含相同数量的行，每个主存块被映射到**固定组的任意一行**
- 假设s是cache组号，j是主存块号，S是组数

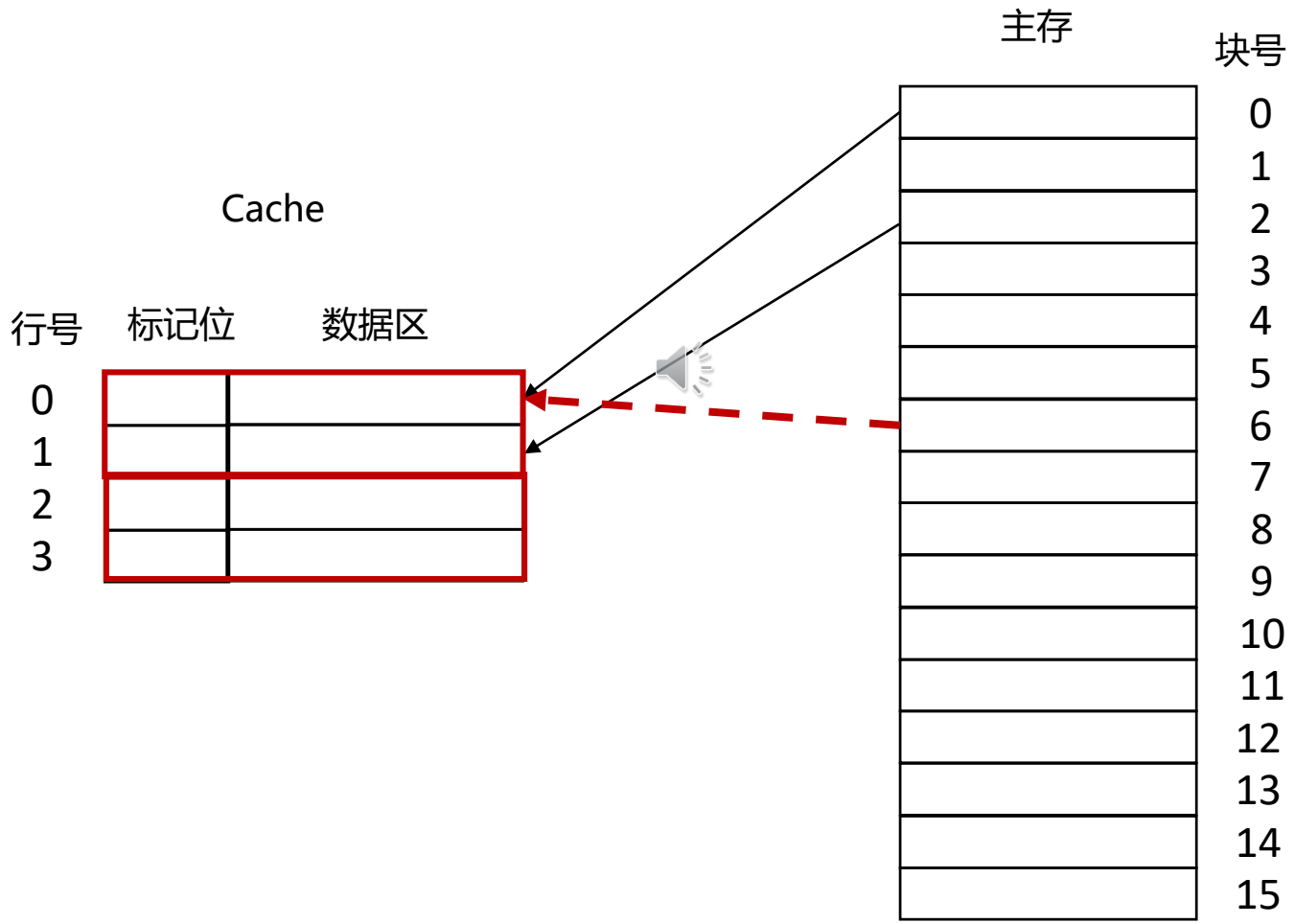
$$s = j \ mod \ S$$

- K-路组关联映射

$$K = C/S$$

组关联映射

- 示例



组关联映射

- 标记
  - 地址中最高n位，$n = log_2 M - log_2 S$

主存地址	标记	Cache组号	块内地址

- 示例
  - 假设cache有4行，每行包含8个字，分成2个组；主存中包含128个字。访问主存的地址长度为7位，则：
    - 最低的3位：块内地址
    - 中间的1位：映射时所对应的Cache中的组
    - 最高的3位：区分映射到同一组的不同块，记录为Cache标记

组关联映射

- 优点
  - 结合了直接映射和关联映射的优点
- 缺点
  - 结合了直接映射和关联映射的缺点
- 面向不同容量的cache做了折中

三种映射方式比较

- 三种方式的相关性
  - 如果 $K = 1$，组关联映射等同于直接映射
  - 如果 $K = C$，组关联映射等同于关联映射

三种映射方式比较

- 关联度（Correlation）：一个主存块映射到cache中可能存放的位置个数
  - 直接映射：1
  - 关联映射：C
  - 组关联映射：K
- 关联度越低，命中率越低
  - 直接映射的命中率最低，关联映射的命中率最高
- 关联度越低，判断是否命中的时间越短
  - 直接映射的命中时间最短，关联映射的命中时间最长
- 关联度越低，标记所占额外空间开销越小
  - 直接映射的标记最短，关联映射的标记最长

Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- 行大小
- Cache数目

替换算法

- 一旦cache行被占用，当新的数据块装入cache中时，原先存放的数据块将会被替换掉
- 对于直接映射，每个数据块都只有唯一对应的行可以放置，没有选择的机会
- 对于关联映射和组关联映射，每个数据块被允许在多个行中选择一个进行放置，就需要替换算法来决定替换哪一行中的数据块
  - 替换算法通过硬件来实现



常用的替换算法

- 最近最少使用算法（Least Recently Used, LRU）
- 先进先出算法（First In First Out, FIFO）
- 最不经常使用算法（Least Frequently Used, LFU）
- 随机替换算法（Random）

最近最少使用算法（LRU）

- 假设：最近使用过的数据块更有可能会被再次使用
- 策略：替换掉在cache中最长时间未被访问的数据块
- 实现：对于2路组关联映射
  - 每行包含一个USE位
  - 当同一组中的某行被访问时，将其USE位设为1，同时将另一行的USE位设为0
  - 当将新的数据块读入该组时，替换掉USE位为0的行中的数据块

先进先出算法（FIFO）

- 假设：最近由主存载入Cache的数据块更有可能被使用
- 策略：替换掉在Cache中停留时间最长的块
- 实现：时间片轮转法 或 环形缓冲技术
  - 每行包含一个标识位
  - 当同一组中的某行被替换时，将其标识位设为1，同时将其下一行的标识位设为0
    - 如果被替换的是该组中的最后一行，则将该组中的第一行的标识位设为0
  - 当将新的数据块读入该组时，替换掉标识位为0的行中的数据块



最不经常使用算法（LFU）

- 假设：访问越频繁的数据块越有可能被再次使用
- 策略：替换掉cache中被访问次数最少的数据块
- 实现：为每一行设置计数器

随机替换算法（Random）

- 假设：每个数据块被再次使用的可能性是相同的
- 策略：随机替换cache中的数据块
- 实现：随机替换
- 随机替换算法在性能上只稍逊于使用其它替换算法

Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- 行大小
- Cache数目



写策略

- 主存和cache的一致性
  - 当cache中的某个数据块被替换时，需要考虑该数据块是否被修改
- 两种情况
  - 如果没被修改，则该数据块可以直接被替换掉
  - 如果被修改，则在替换掉该数据块之前，必须将修改后的数据块写回到主存中对应位置
- 策略
  - 写直达（write through）
  - 写回法（write back）

写直达

- 所有写操作都同时对cache和主存进行
- 优点
  - 确保主存中的数据总是和cache中的数据一致，总是最新的
- 缺点
  - 产生大量的主存访问，减慢写操作



写回法

- 先更新cache中的数据，当cache中某个数据块被替换时，如果它被修改了，才被写回主存
- 利用一个脏位（dirty bit）或者使用位（use bit）来表示块是否被修改
- 优点
  - 减少了访问主存的次数
- 缺点
  - 部分主存数据可能不是最新的
    - I/O模块存取时可能无法获得最新的数据，为解决该问题会使得电路设计更加复杂且有可能带来性能瓶颈

Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- 行大小
- Cache数目

行大小

- 假设从行的大小为一个字开始，随着行大小的逐步增大，则Cache命中率会增加
  - 数据块中包含了更多周围的数据，每次会有更多的数据作为一个块装入cache中
  - 利用了空间局部性
- 当行大小变得较大之后，继续增加行大小，则Cache命中率会下降
  - 当Cache容量一定的前提下，较大的行会导致Cache中的行数变少，导致装入cache中的数据块数量减少，进而造成数据块被频繁替换
  - 每个数据块中包含的数据在主存中位置变远，被使用的可能性减小
- 行大小与命中率之间的关系较为复杂





Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- 行大小
- Cache数目



Cache数目：一级 vs. 多级

- 一级
  - 将cache与处理器置于同一芯片（片内cache）
  - 减少处理器在外部总线上的活动，从而减少了执行时间
- 多级
  - 当L1未命中时，减少处理器对总线上DRAM或ROM的访问
  - 使用单独的数据路径，代替系统总线在L2缓存和处理器之间传输数据，部分处理器将L2 cache结合到处理器芯片上

Cache数目：统一 vs. 分立

- 统一
  - 更高的命中率，在获取指令和数据的负载之间自动进行平衡
  - 只需要设计和实现一个cache
- 分立
  - 消除cache在指令的取值/译码单元和执行单元之间的竞争，在任何基于指令流水线的设计中都是重要的

总结

- Cache的目的、基本思路、工作流程
- Cache的若干问题
  - 命中 vs. 未命中
  - 未命中时将数据块传送到Cache中
  - 平均访问时间
- Cache的设计要素
  - Cache容量
  - 映射功能：直接映射，关联映射，组关联映射
  - 替换算法：LRU，FIFO，LFU，随机
  - 写策略：写直达，写回法
  - 行大小
  - Cache数目：一级vs.多级，统一vs.分立

谢谢

rentw@nju.edu.cn

南京大學
NANJING UNIVERSITY