高速缓存

2022年秋

内容提要

- Cache的地址映射
  - 全相联映射
  - 直接映射
  - 多路组相联
- Cache写策略
- 提高Cache性能的途径
  - 组织结构
  - Cache参数（大小、块大小、替换策略）

层次存储器系统

- 使用高速缓冲存储器Cache来提高CPU对存储器的平均访问速度。
- 时间局部性：最近被访问的信息很可能还要被访问。
  - 将最近被访问的信息项装入到Cache中。
- 空间局部性：最近被访问的信息临近的信息也可能被访问。
  - 将最近被访问的信息项临近的信息一起装入到Cache中。

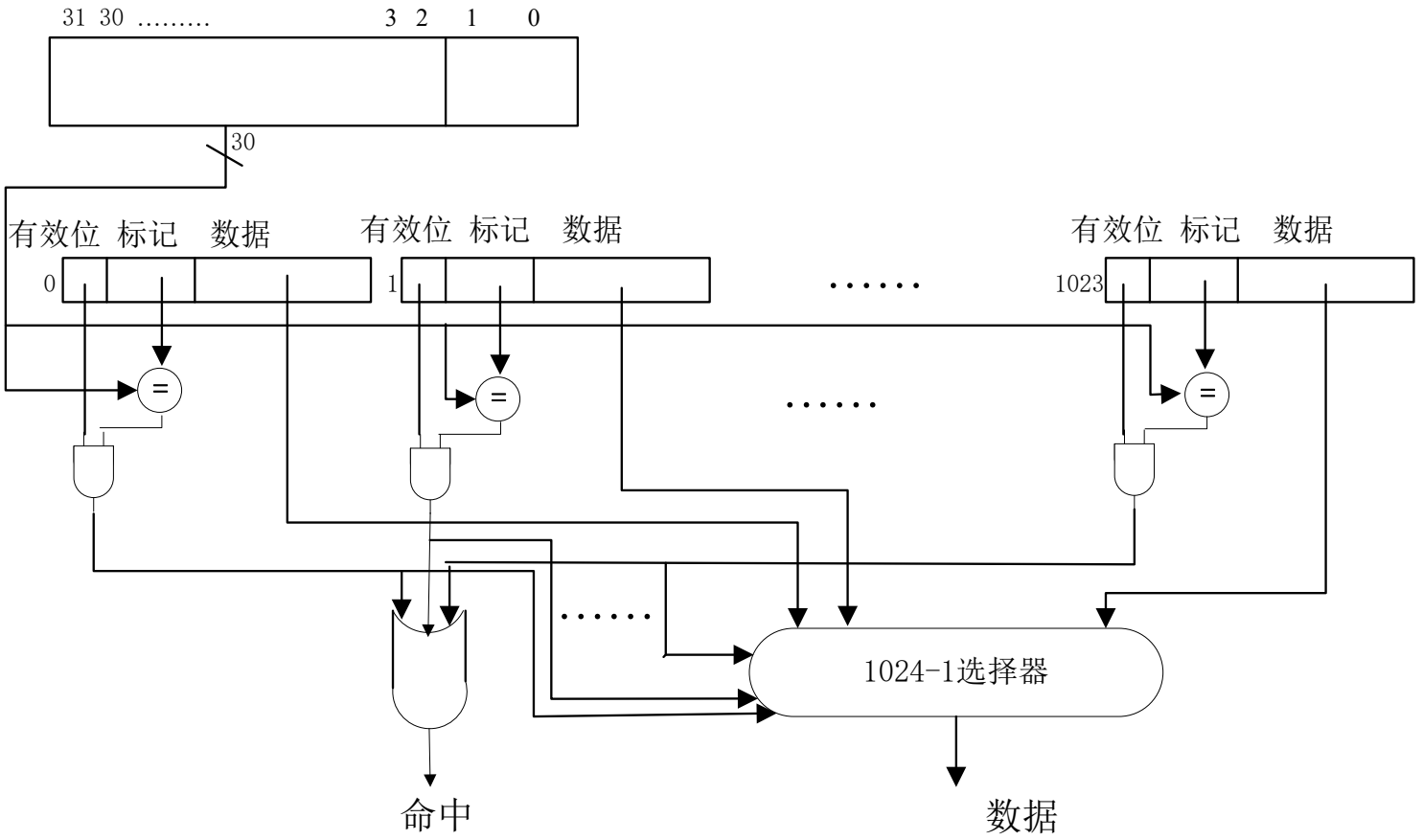
高速缓冲存储器Cache

- 基于程序的局部性原理
  - 时间局部性
  - 空间局部性
- 利用静态存储器的高速特性
- 设置于主存储器与CPU之间
- 缓存CPU频繁访问的信息
- 提高CPU访问存储器的整体性能

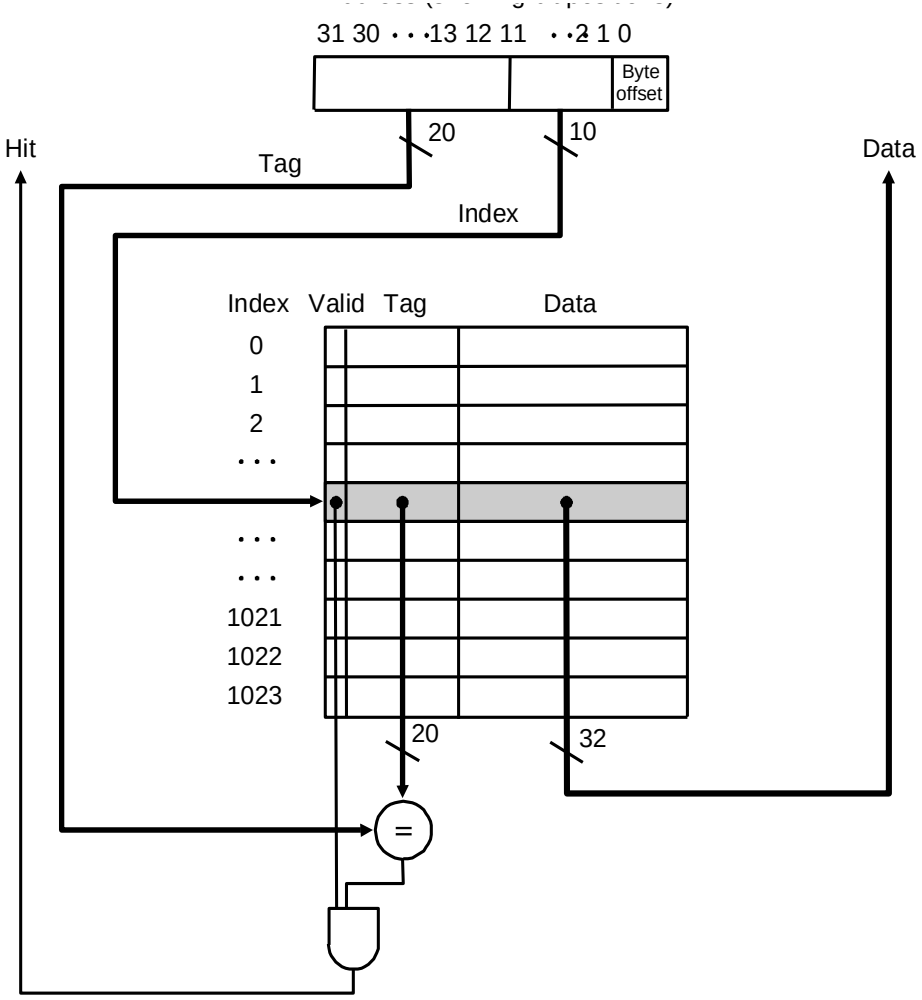
需要解决的问题

- 如何通过主存地址去访问Cache？
  - 全相联
  - 直接映射
  - 多路组相联
- 如何保证层次间一致性？
  - 有效位、写策略
- Cache参数对性能的影响
  - Cache的组织：块大小
  - 替换策略
  - 接入方式

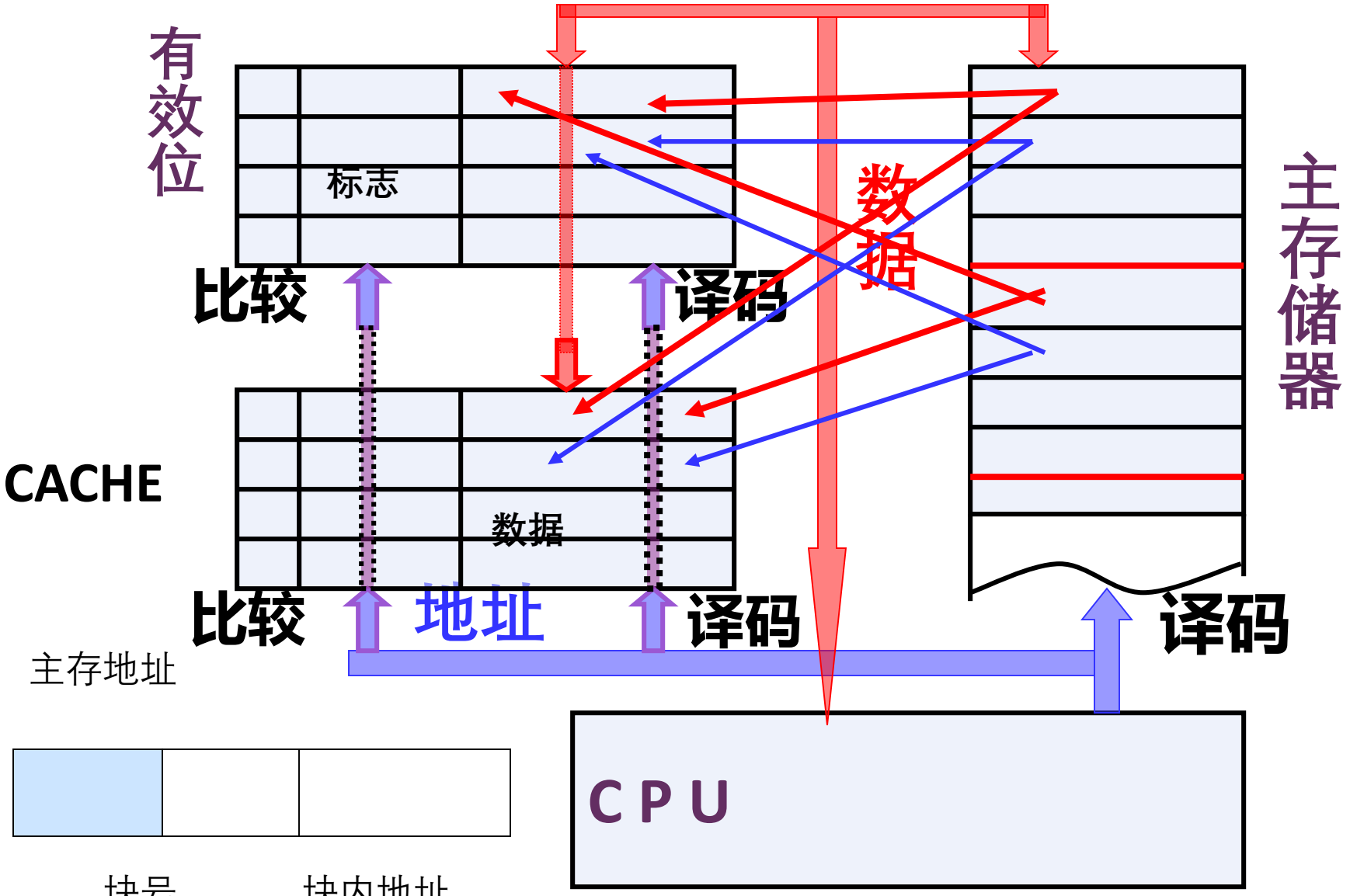
全相联映射硬件实现



直接映射Cache硬件实现



两路组相联方式



两路组相联方式的地址映射

特点

1. 前两种方式的折衷方案。组内为全相联，路内为直接映射。

2. 集中了两个方式的优点。成本也不太高。

是常用的方式

组相联Cache访问举例

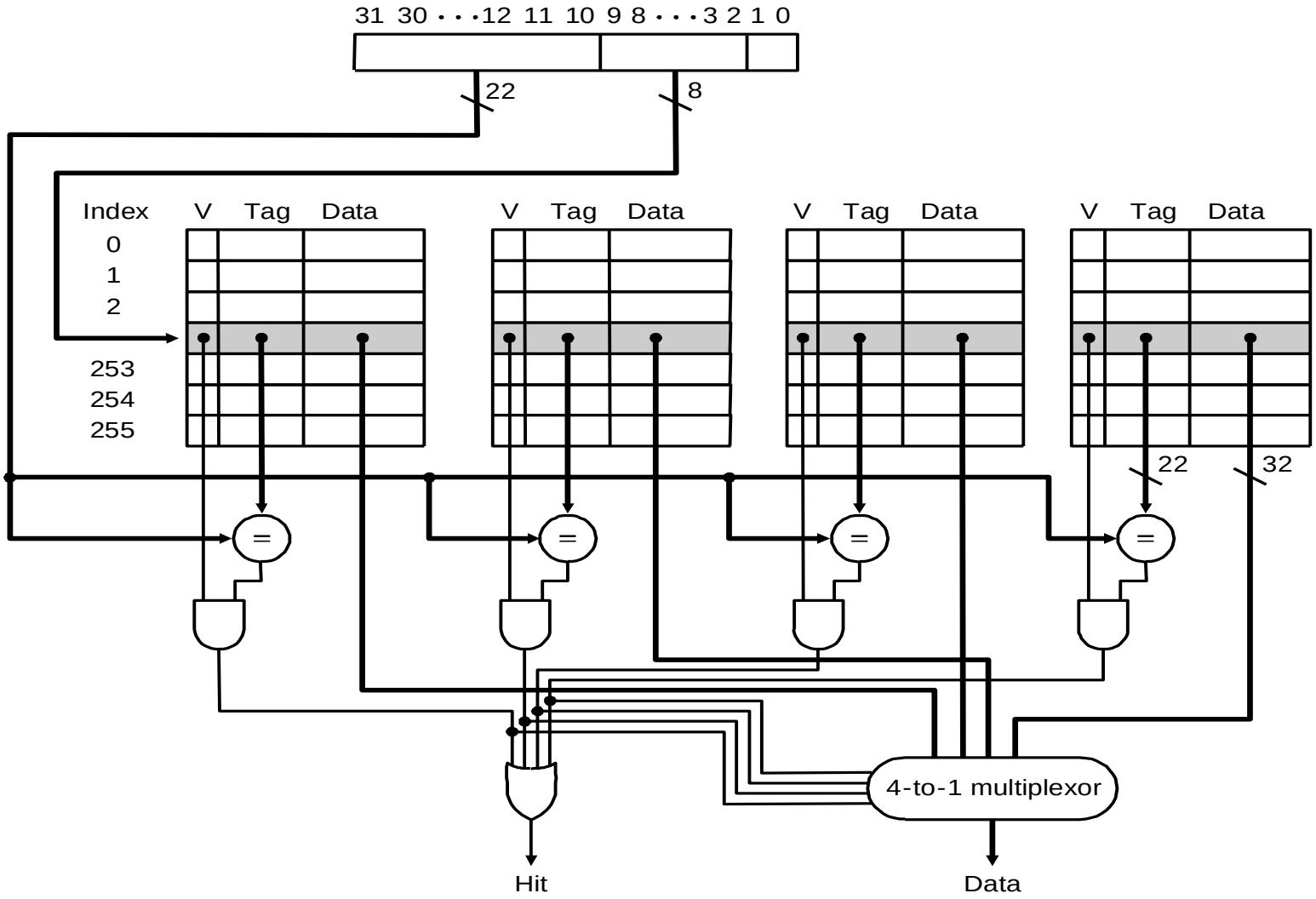
假设有下列访问主存顺序:

Read location 0: Miss
Read location 16: Miss
Read location 32: Miss
Read location 4: Hit
Read location 8: Hit
Read location 36: Hit
Read location 32: Hit
Read location 128: Miss
Read location 148: Miss
Read location 0: Hit
Read location 128: Hit
Read location 4: Hit
Read location 132: Hit

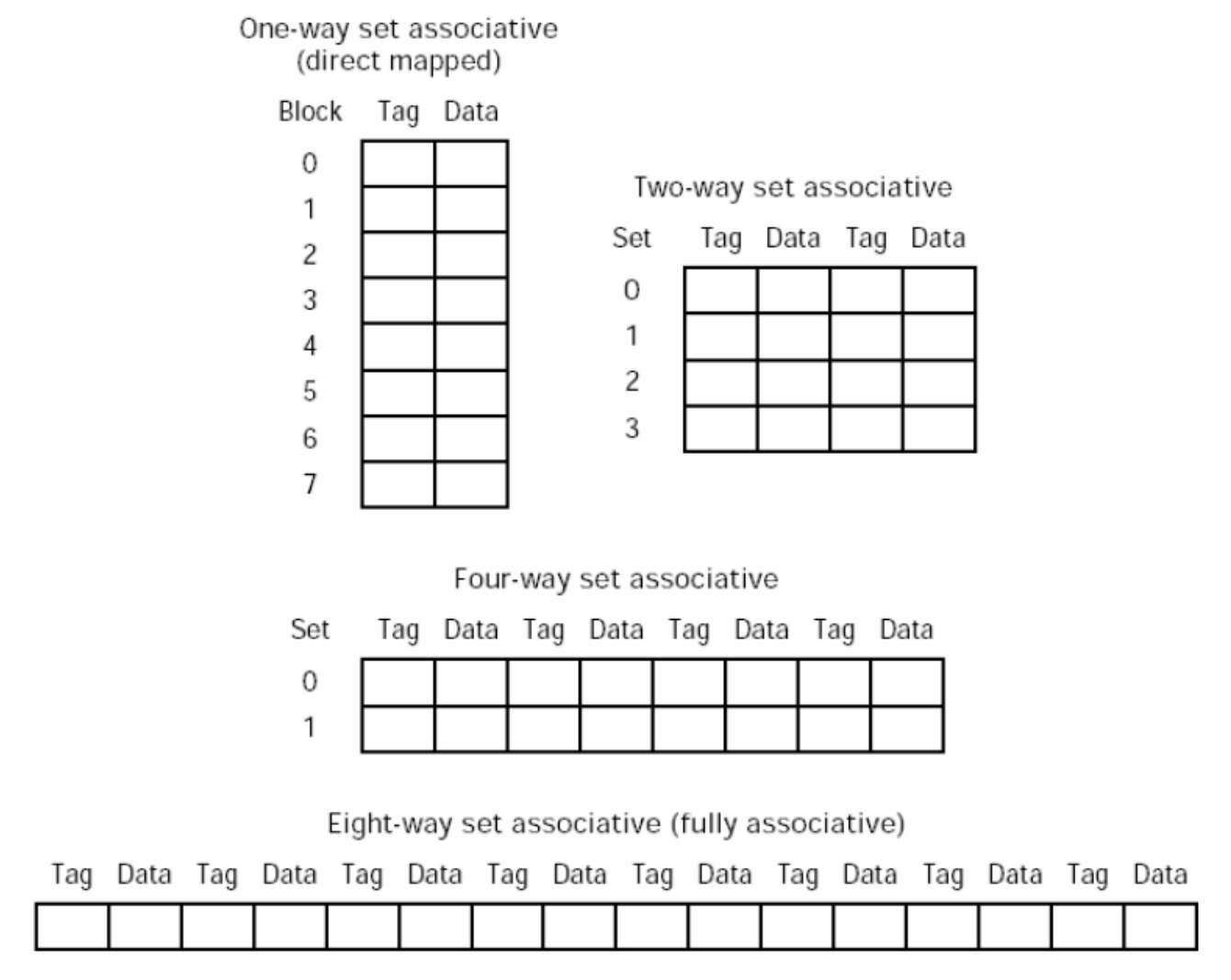
0-15
64-79
…

0-15	128-143
16-31	144-159
32-47	

四路组相联的Cache实现方式



直接映射到全相联


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

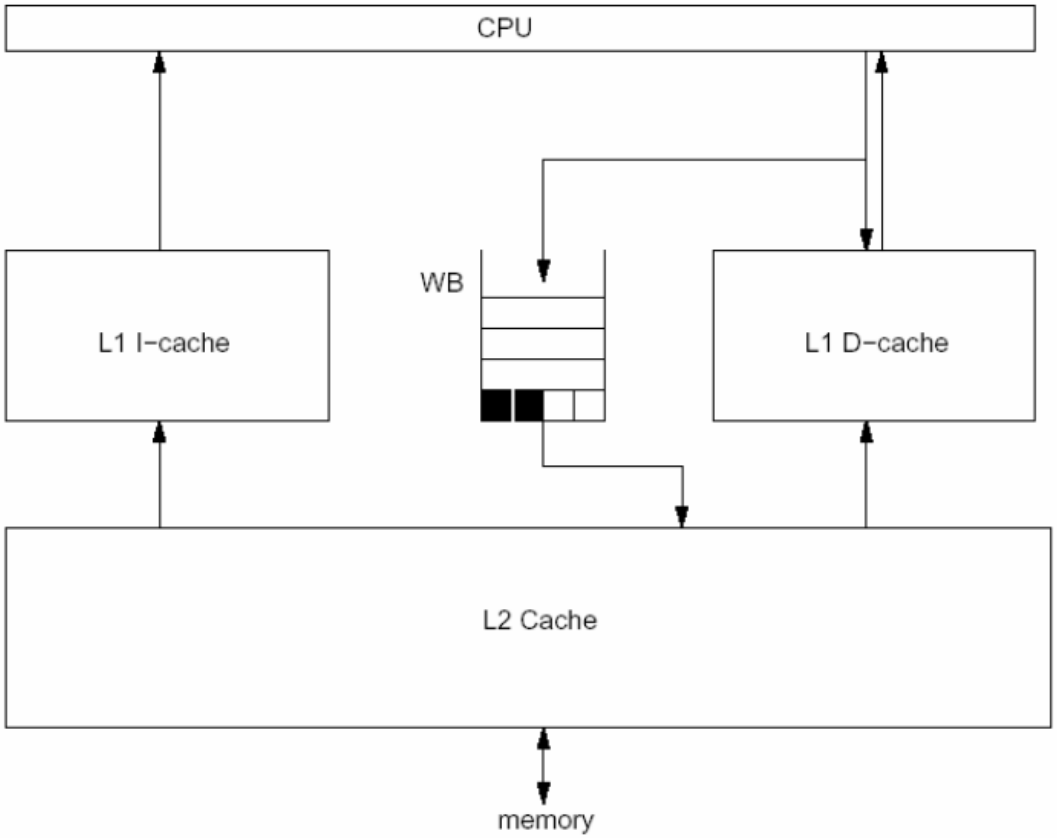
三种映射方式比较

- 直接映射
  - 主存中的一块只能映射到Cache中唯一的一个位置
  - 定位时，不需要判断，只需替换
- 全相联映射
  - 主存中的一块可以映射到Cache中任何一个位置
- N路组相联映射
  - 主存中的一块可以选择映射到Cache中N个位置
- 全相联映射和N路组相联映射的失效处理
  - 从主存中取出新块
  - 为了腾出Cache空间，需要替换出一个Cache行
  - 不唯一，则需要判断应替出哪行

一致性保证

- 写直达（Write through）
  - 强一致性保证，效率低
  - 在Cache中命中
    - 同时修改Cache和对应的主存内容
  - 没有在Cache中命中
    - 写分配（Write allocate）
    - 非写分配（not Write allocate ）
- 拖后写（Write back）
  - 弱一致性保证，替换时再写主存
    - 主动替换
    - 被动替换
  - 通过监听总线上的访问操作来实现
  - 实现复杂，效率比较高

Cache写（命中）



Cache（不命中）写策略

	Write through				Write back	
	Write allocate		No write allocate		Write allocate	
Steps	fetch on miss	no fetch on miss	write around	write invalidate Hit	fetch on miss	no fetch on miss
1	pick replacement	pick replacement			pick re-placement	pick re-placement
2				invalidate tag	[write back]	[write back]
3	fetch block				fetch block	
4	write cache	write partial cache			write cache	write partial cache
5	write memory	write memory	write memory	write memory		

提高存储访问的性能

- 平均访问时间=
- 命中时间x 命中率+ 缺失损失x 缺失率

- 提高命中率
- 缩短缺失时的访问时间
- 提高Cache本身的速度

Cache缺失的四类原因

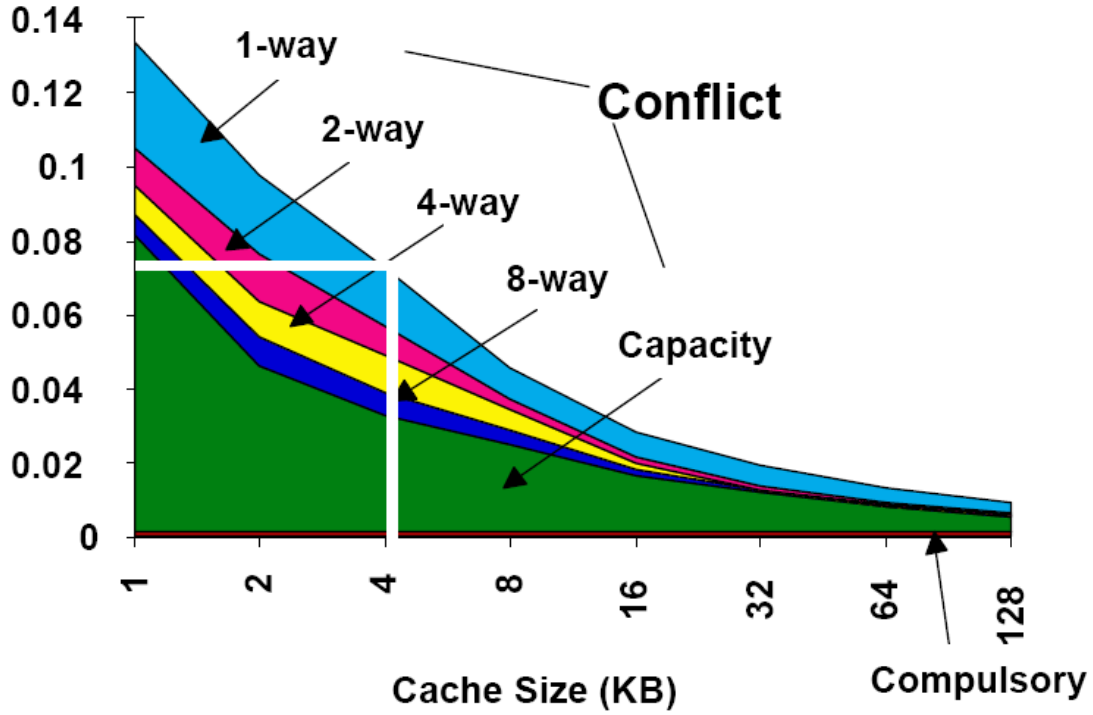
- 必然缺失（Compulsory Miss）
  - 开机或者是进程切换
  - 首次访问数据块
- 容量缺失（Capacity Miss）
  - 活动数据集超出了Cache的大小
- 冲突缺失（Conflict Miss）
  - 多个内存块映射到同一Cache块
  - 某一Cache组块已满，但空闲的Cache块在其他组
- 无效缺失
  - 其他进程修改了主存数据

对策

- 必然缺失
  - 世事总有缺憾
  - 如果程序访问存储器的次数足够多，也就可以忽略了
  - 策略
    - 预取
- 容量缺失
  - 出现在Cache容量太小的时候
  - 增加Cache容量，可缓解缺失现象
- 冲突缺失
  - 两块不同的内存块映射到相同的Cache块
  - 对直接映射的Cache，这个问题尤其突出
    - 增加Cache容量有助于缓解冲突
    - 增加相联的路数有助于缓解冲突

影响Cache缺失率的因素

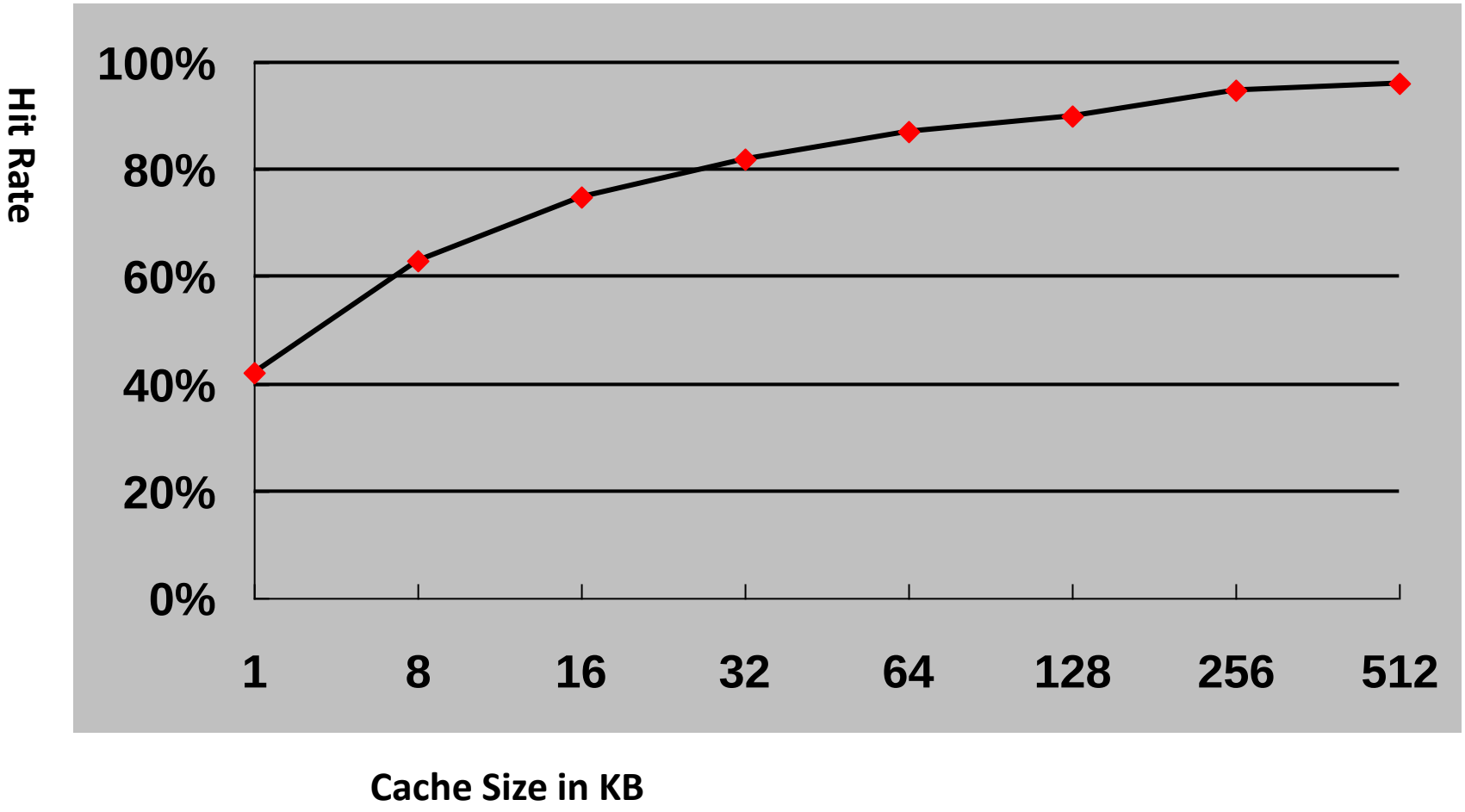
- 经验总结：容量为N、采用直接映射方式Cache的缺失率和容量为N/2、采用2路组相联映射方式Cache的缺失率相当



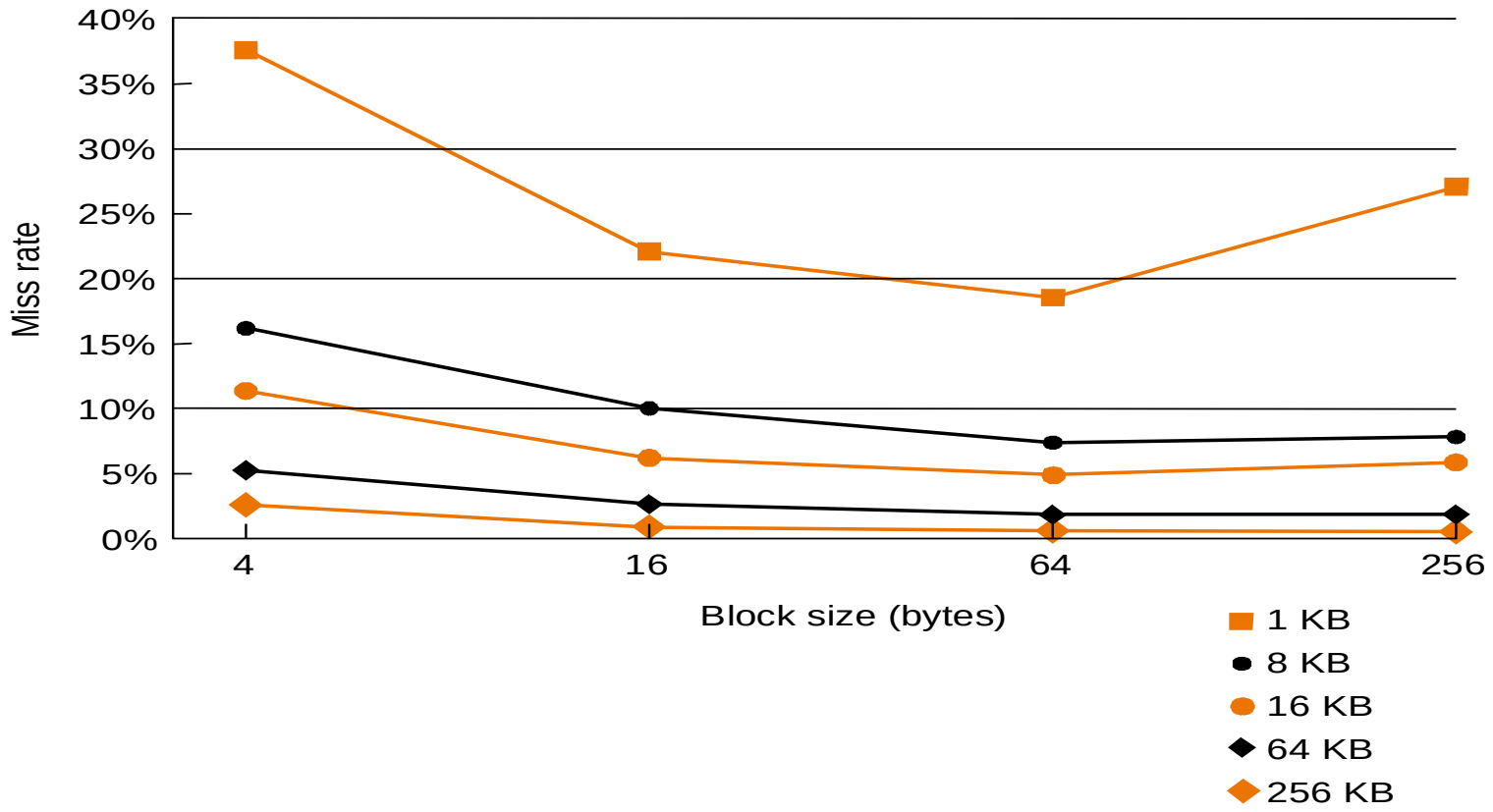
影响Cache命中率的因素

- Cache容量
  - 大容量可以提高命中率，但是……
- Cache块大小
  - 选择多大的行，还真是个问题
- 地址映射方式
  - 多路组相联，但到底多少路呢？
- 替换算法
  - 替换哪行出去呢？
- 多级Cache
  - 给用户更多的选择

命中率和容量的关系

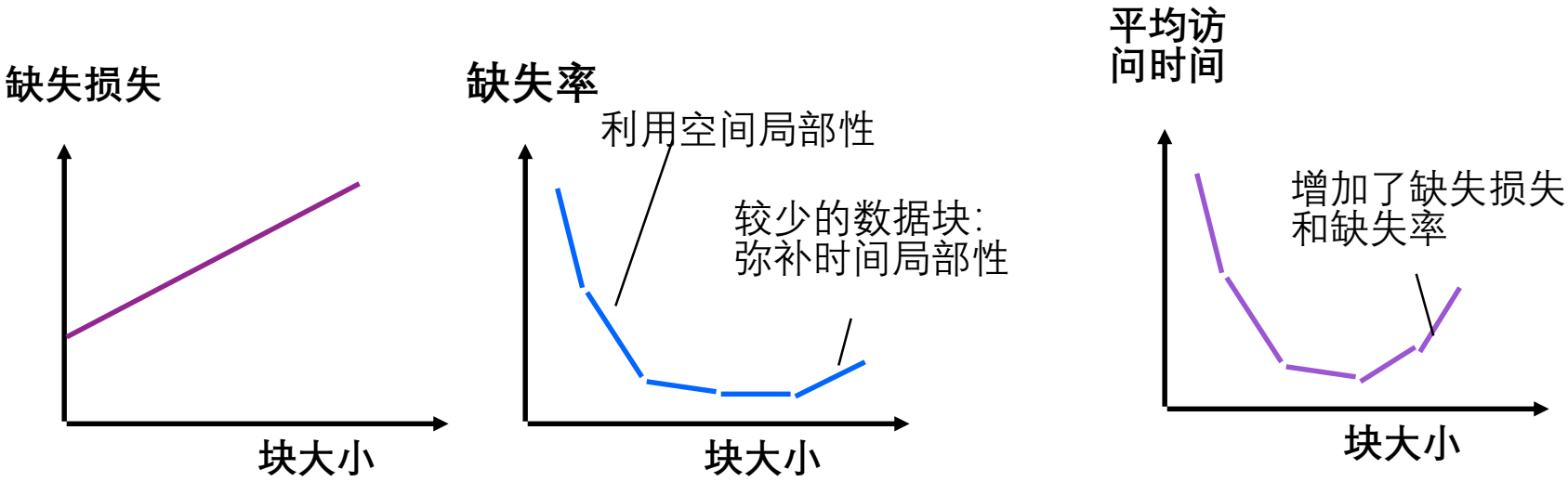


块大小和缺失率的关系


40%
35%
30%
25%
20%
15%
10%
5%
0%
Miss rate
4
16
64
256
Block size (bytes)
1 KB
8 KB
16 KB
64 KB
256 KB

块大小的权衡

- 一般来说，数据块较大可以更好地利用空间局部性，但是：
  - 数据块大意味着缺失损失的增大：
    - 需要花费更长的时间来装入数据块
  - 若块大小相对Cache总容量来说太大的话，命中率将降低
    - Cache块数太少
- 一般来说，平均访问时间 = 命中时间 x 命中率 + 失效损失 x 缺失率



块替换策略

- 直接映射
  - 主存中的一块只能映射到Cache中唯一的一个位置
  - 定位时，不需要选择，只需替换
- 全相联映射
  - 主存中的一块可以映射到Cache中任何一个位置
- N路组相联映射
  - 主存中的一块可以选择映射到Cache中N个位置
- 全相联映射和N路组相联映射的失效处理
  - 从主存中取出新块
  - 为了腾出Cache空间，需要替换出一个Cache块
  - 不唯一，则需要选择应替出哪块

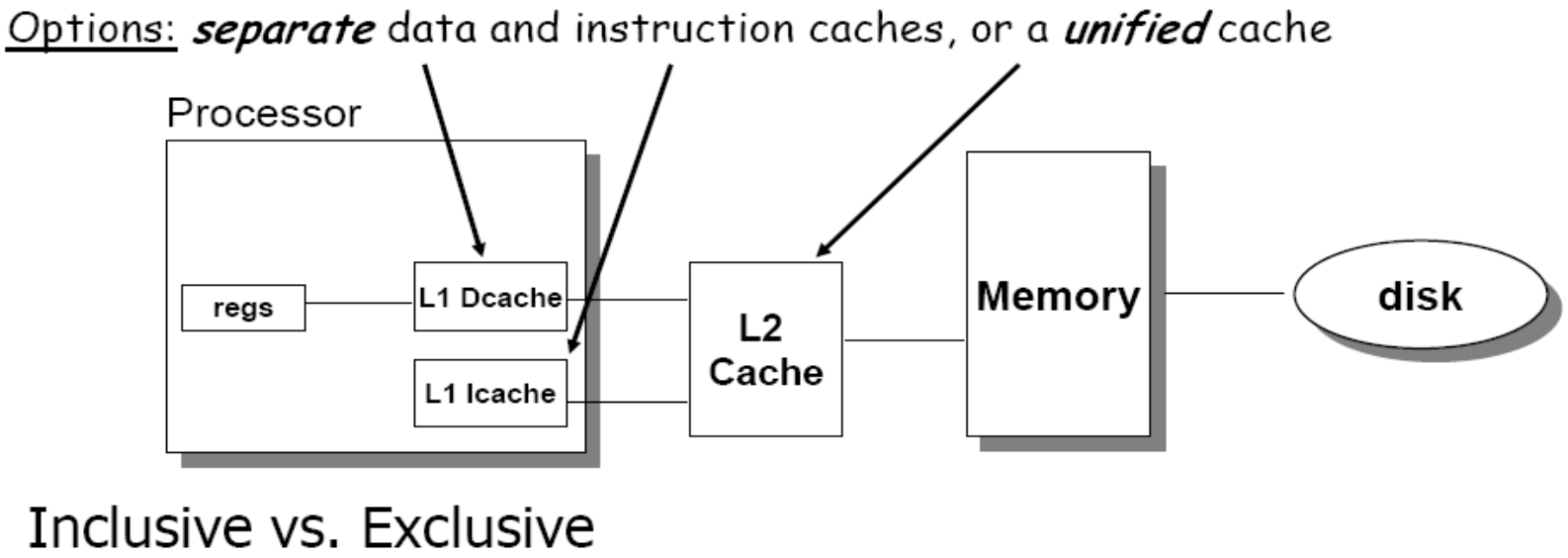
替换策略

- 最近最少使用LRU
  - 满足程序局部性要求
  - 有较高命中率
  - 硬件实现复杂
- 先进先出FIFO
  - 满足时间局部性
  - 实现比较简单
- 随机替换RAND
  - 实现简单
  - 命中率也不太低

多级Cache

- 采用两级或更多级cache来提高命中率
  - 增加Cache层次
  - 增加了用户的选择
- 将Cache分解为指令Cache和数据Cache
  - 指令流水的现实要求
  - 根据具体情况，选用不同的组织方式、容量

多级Cache

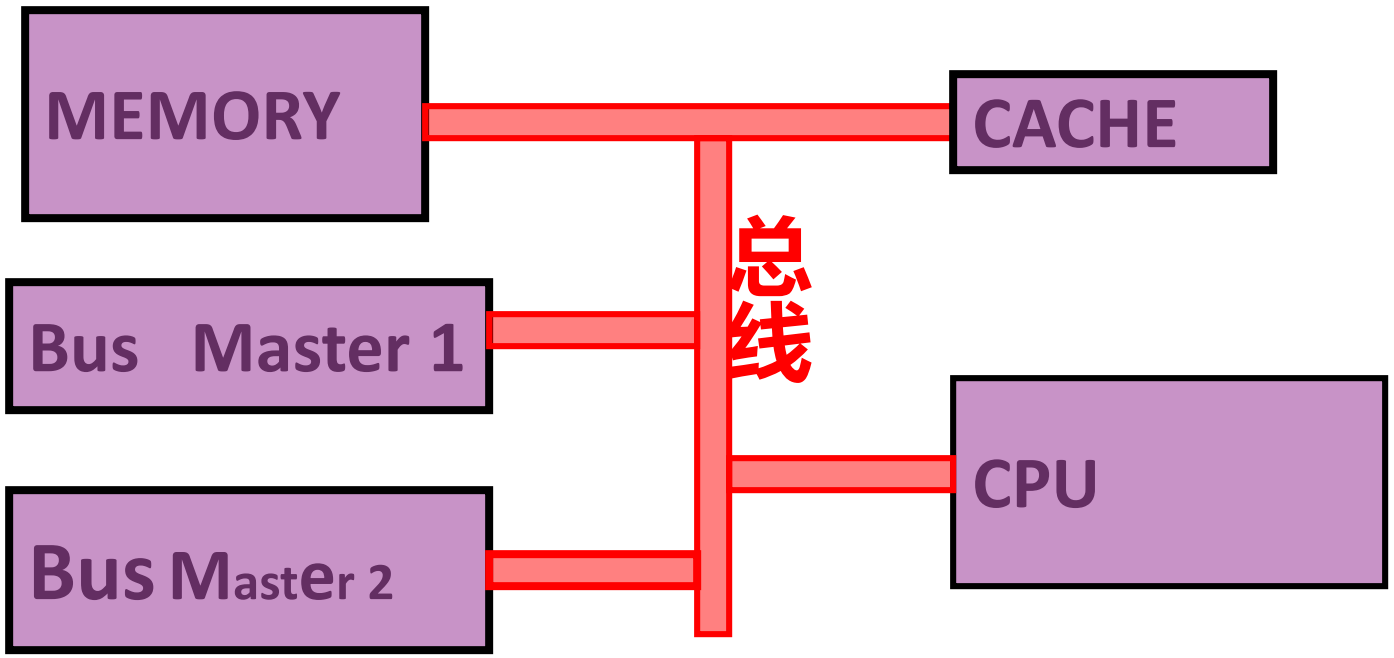


Inclusive vs. Exclusive

- 在处理器中设置独立的数据Cache和指令Cache
- 在处理器外设置第二级Cache，甚至是第三级Cache

Cache接入系统的体系结构

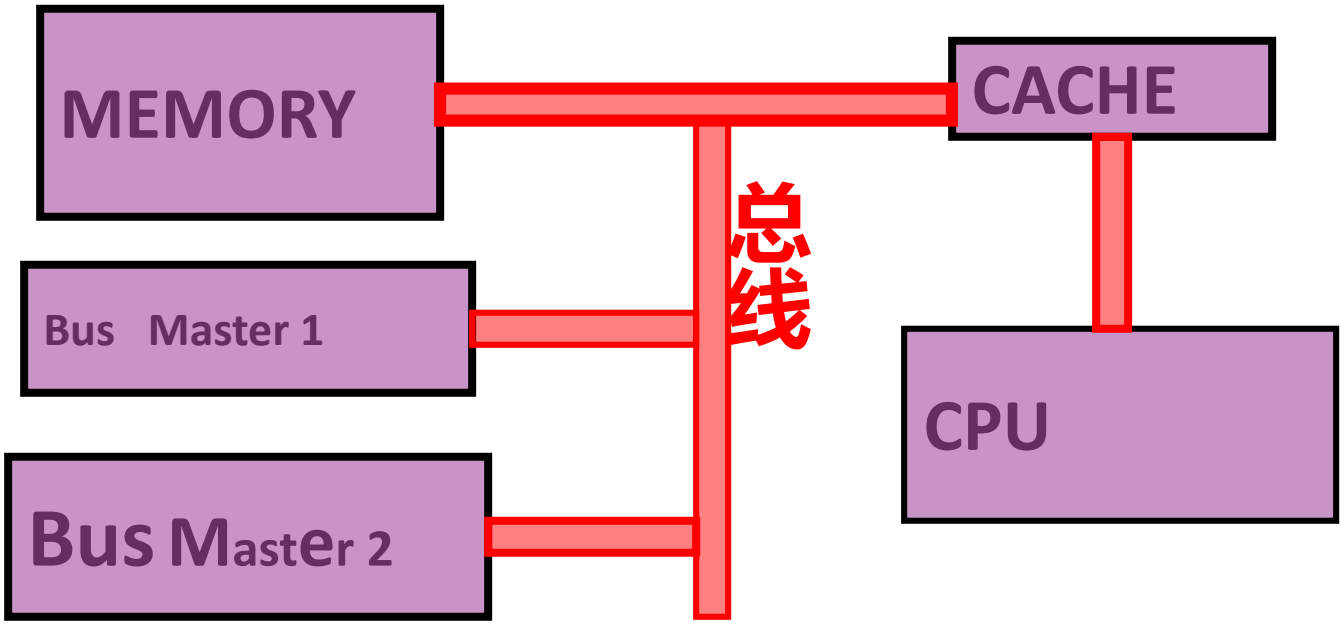
1. 侧接法：像入出设备似的连接到总线上，优点是结构简单，成本低，缺点是不利于降低总线占用率。



CACHE 接入系统的体系结构

2. 隔断法：把原来的总线打断为两段，

使 CACHE 处在两段之间，优点是有利于提高总线利用率，支持总线并发操作，缺点是结构复杂，成本较高。

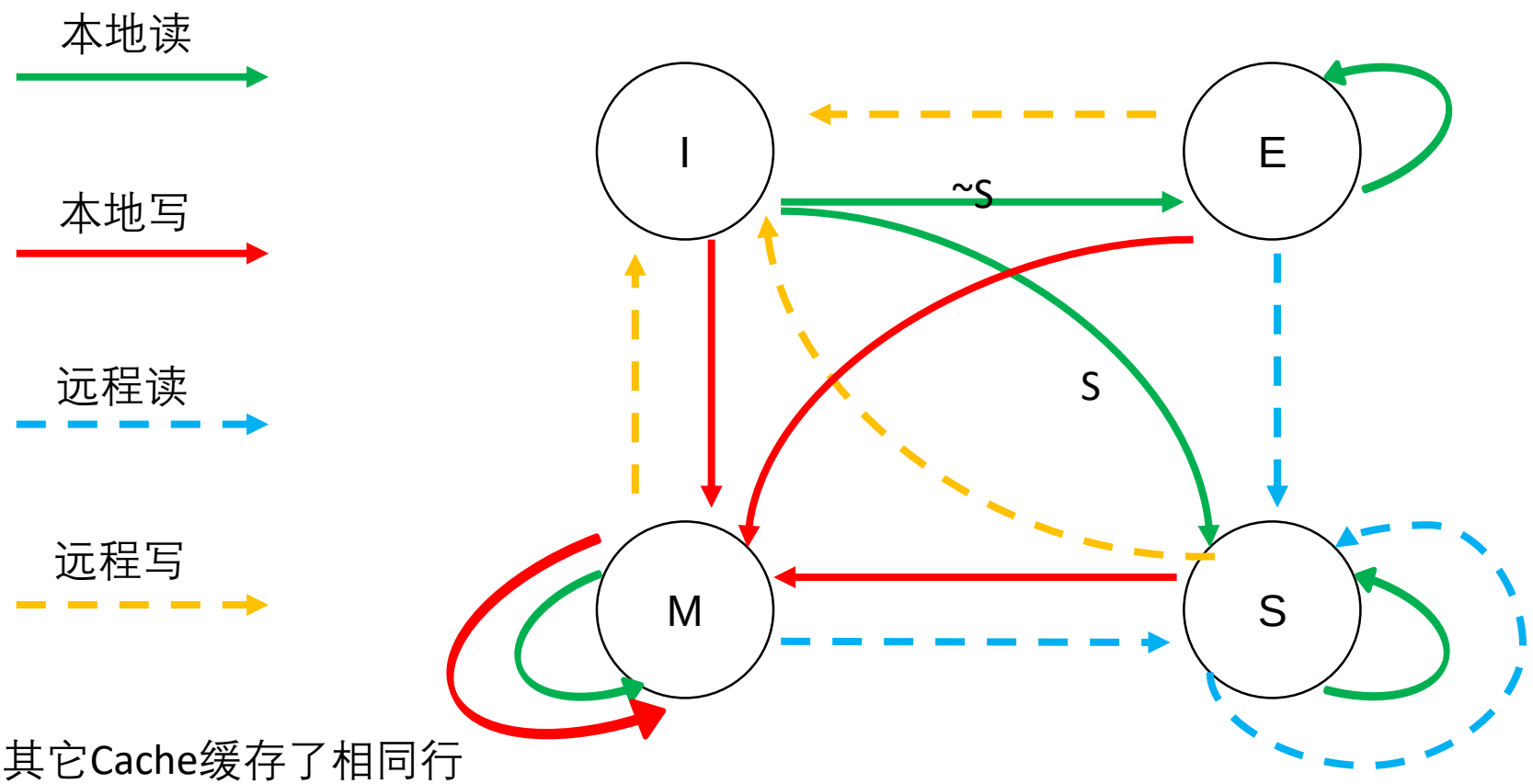


一致性保证策略（MESI）

- 要保证本地cache的数据，其它核cache的数据，内存的数据有一个一致的视图

- 修改态（M）：处于这个状态的cache块中的数据已经被修改过，和主存中对应的数据已不同，只能从cache中读到正确的数据
- 独占态（E）：处于本状态的cache块的数据和主存中对应的数据块内容相同，而且在其它cache中没有副本
- 共享态（S）：处于本状态的cache块的数据和主存中对应的数据块内容相同，而且可能在其它cache中有该块的副本
- 无效态（I）：处于本状态的cache块中尚未装入数据

缓存行的状态转换



S:其它Cache缓存了相同行
~S：其它Cache没有缓存

Cache

- 目标
  - 提高CPU访问存储器系统的平均速度
- 策略
  - 利用一容量较小（降低成本）的高速缓冲存储器
- 组织方式
  - 全相连、直接映射、组相连

Cache

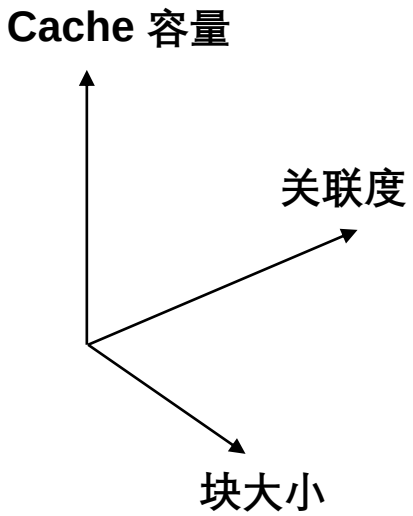
- 包含性保证
  - cache中的块和主存中的块进行映射
- 一致性保证
  - 写回主存策略
- 提高命中率
  - 容量
  - 关联方式
  - 块替换算法

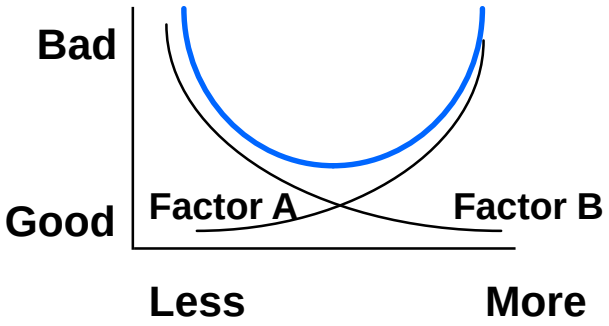
Cache

- 局部性原理：
  - 任何时候，程序需要访问的只是相对较小的一些地址空间。
    - 时间局部性
    - 空间局部性
- 三类主要的Cache缺失原因：
  - 无法避免的缺失：如：第一次装入
  - 块冲突：增大Cache容量、改进组织方式
    避免不断的块冲突
  - 容量冲突：增大Cache容量
- Cache设计
  - 总容量、块大小、组织方式
  - 替换算法
  - 写策略（命中时）：写直达、拖后写
  - 写策略（不命中时）：是否装入到Cache？

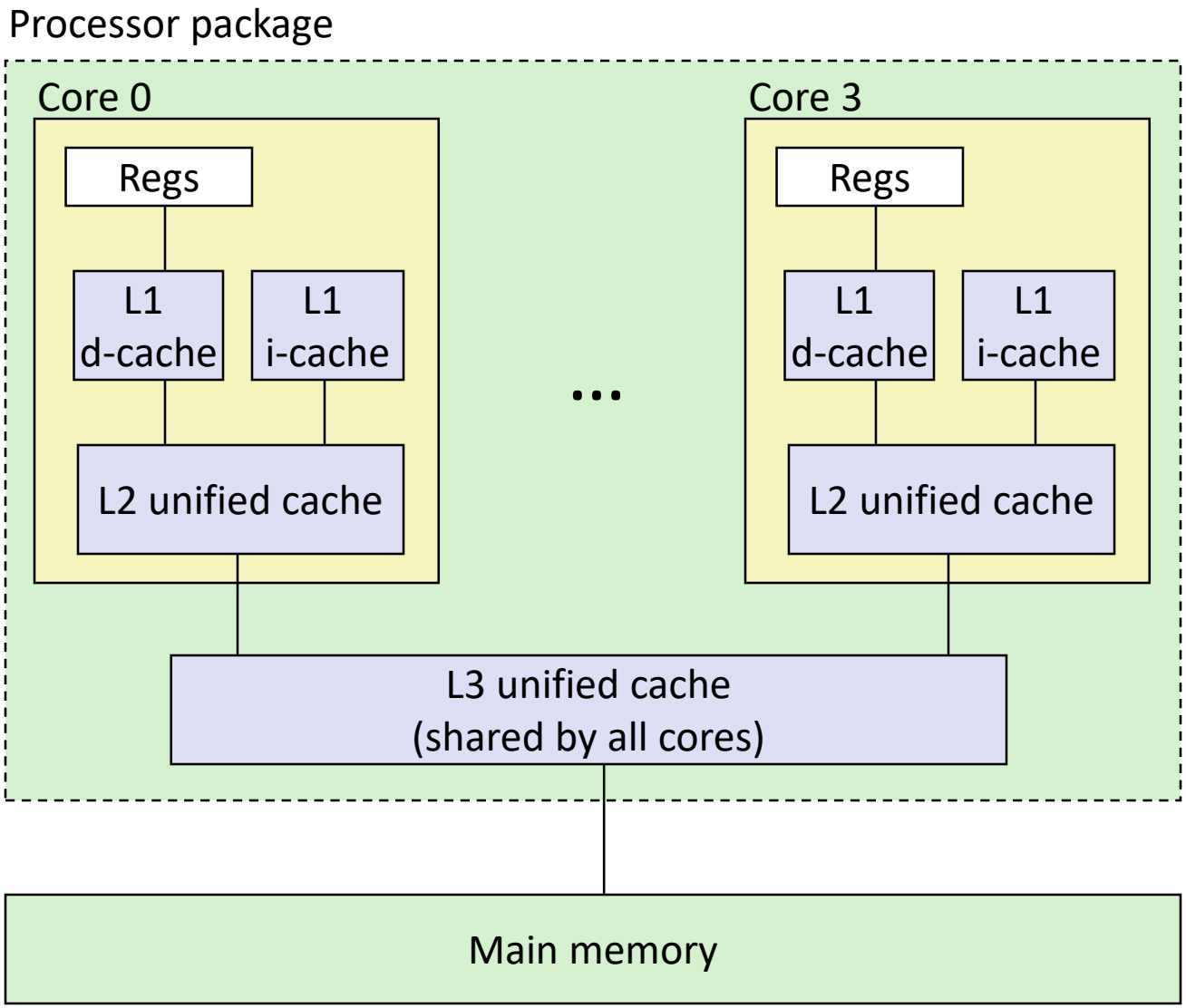
设计Cache

- 有关方案
  - cache 容量
  - 块大小
  - 组织方式
  - 替换算法
  - 写策略
- 方案优化
  - 根据用途选择
    - 海量数据处理
    - 指令数据平衡 (I-cache, D-cache)
  - 根据成本优化
- 简单化常常就是优化





Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

阅读和思考

- 阅读
  - 教材相关章节
  - 预习虚拟存储器
- 思考
  - Cache命中率和哪些因素有关？如何提高Cache的命中率？
  - Cache写有许多策略，试进行比较。

谢谢