高等计算机体系结构
第十二讲: Cache的性能/主存储器

栾钟治
北京航空航天大学 计算机学院 中德联合软件研究所
2021-05-14

1

提醒：作业

- 作业 5
  - 已发布，5月28日上课前截止提交
  - Cache和Memory
- 作业 6
  - 5月28日发布，6月11日截止
  - 预取和并行

2

实验2-5

- 近期发布，预计7月11日截止

3

阅读材料

- 分层存储体系结构

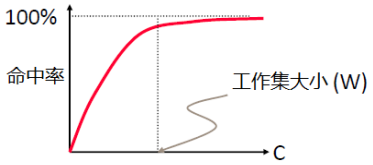
- Patterson & Hennessy's *Computer Organization and Design: The Hardware/Software Interface* （计算机组成与设计：软硬件接口）
  - 第五章：5.1-5.3, 5.4
- Maurice Wilkes早期关于cache的论文
  - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.
- 推荐阅读
  - Denning, P. J. *Virtual Memory*. ACM Computing Surveys. 1970
  - Jacob, B., & Mudge, T. *Virtual Memory in Contemporary Microprocessors*. IEEE Micro. 1998

4

回顾：直接映射的Cahce

- C字节存储分为C/B块
- 根据地址的块索引域将一块内存映射到一个特定的Cache块
- 所有具有相同块索引域的地址映射到相同的Cache块
  - 2^t个这样的地址；一次只能缓存一个这样的块
  - 即使C>工作集大小，也可能产生冲突
  - 给定2个随机的地址，冲突几率为1/(C/B)

    注意，冲突的可能性随着Cache块数量的增加而降低

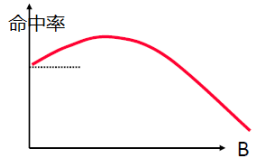


5

5

回顾：块的大小和缺失率m_i

- 共享一个公共标签tag的字节是作为一个整体处理的
- 一次加载多个字的块具有基于空间局部性预取的效果
  - 缺失时每块仅接受一次惩罚
  - 在指令Cache中尤其有效
  - 有效性受到空间局部性极限的限制
- 但是，增加块大小(同时保持C不变)
  - 会减少块数
  - 增加冲突的可能性

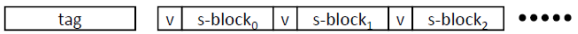


6

6

回顾：块的大小和T_{i+1}

- 加载大的块可以增加T_{i+1}
  - 如果需要块上的最后一个字，必须等待整个块被加载
- 解决方案1：关键词优先重装
  - L_{i+1}首先返回请求的字，然后再完成整个块的其他部分
  - 尽快向流水线提供请求的字
- 解决方案2：划分子块
  - 每一个子块有独立的有效位
  - 仅按需加载请求的子块
  - 注意：所有子块共享公共标签tag



7

7

回顾：“a”路组相联的Cache

- C字节的存储分成a个直接映射的bank，每个组都有C/a/B块
  - 地址被映射到每个bank中特定的块，存在a个这样的bank
    - a个可能的位置加在一起就是“组(set)”
    - 直接映射是它的特例(a=1)
  - 额外的开销：需要a个比较器和a选1的多路选择器
- 块索引域相同的地址都映射到同一“组”cache块上
  - 2^t个这样的地址；同时可以cache a个这样的块
  - 如果C>工作集大小

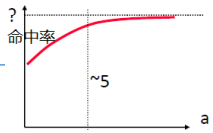
    相联度越高→冲突越少
  - 如果C<工作集大小

    ？？？

8

8

回顾：全相联的Cache：



- “内容可寻址”存储器
  - 不是常规的SRAM
  - 给定标签，返回与该标签匹配的块，否则就是一次缺失
  - 查找中不使用索引位
- 任何地址可能在C/B个块中的任何一块里
  - 如果C>工作集大小，则无冲突
- 每个Cache块需要一个比较器、一个巨大的多路选择器和许多长导线
  - 考虑L1延迟，数十个块会带来非常昂贵的开销和复杂的处理

幸运的是，没有理由采用非常大的全相联Cache

任何足够大且合理的C/B，a=4~5与a=C/B，对于典型的程序而言没有太大的区别（一样好）

9

9

回顾：Cache基本参数

ISA

- M=2^m，表示地址空间的大小（多少byte）
  - 比如：2^{32}，2^{64}
- G=2^g，表示Cache访问的粒度大小（多少byte）
  - 比如：4，8

实现

- C，表示Cache的容量（多少byte）
  - 比如：16KByte(L1), 1MByte(L2)
- B = 2^b，Cache块的大小（多少byte）
  - 比如：16(L1), > 64(L2)
- a，Cache的相联度
  - 比如：1, 2, 4, 5(?), …… C/B

10

10

回顾：替换策略

- 哪一块在cache缺失时被替换?
  - 首先是任何无效的块
  - 如果所有块都有效，替换策略
    - 随机
    - FIFO
    - 最近最少使用LRU (如何实现?)
    - 非最近使用Not MRU
    - 最不经常使用
    - 重取成本最低
      - 为什么内存的访问会有不同的开销?
    - 混合替换策略
    - 最优替换策略

11

11

回顾：流水线设计中的多层cache

- 第一层cache (指令和数据)
  - 决策受时钟周期影响很大
  - 容量小, 较低的相联度
  - 标签存储和数据存储并行访问
- 第二层cache
  - 决策需要平衡命中率和访问延迟
  - 通常比较大而且有较高的相联度; 延迟并不是最重要的因素
  - 标签存储和数据存储串行访问
- 层次间的串行vs. 并行访问

12

12

回顾：处理“写”(Store)

- 写直达: 当写的动作发生时把cache中修改过的数据写到下一级
  + 更简单
  + 所有层都是最新的
    一致性: 更简单的cache一致性，因为无需检查低层次的cache
  -- 更高的带宽需求; 无法进行写合并
- 写回: 当cache块被换出时把cache中修改过的数据写到下一级
  + 可以在换出之前把对同一个块的多个写合并
    - 节省不同级cache之间的带宽，并且节省能耗
  --需要在标签存储中使用1位标记某块“被修改”

- 写缺失时分配
  + 可以合并写而不是每次单独写下一层cache
  + 更简单，因为写缺失可以和读缺失同样对待
  -- 需要移动整个cache块
- 无分配
  + 如果写的局部性比较低能够节约cache空间 (隐含有更好的cache命中率)

13

回顾：Cache缺失的种类

- 强制(Compulsory)缺失
  - 第一次引用某个地址(块)总是导致一个缺失
  - 后续的引用将会命中，除非cache块因为某些原因被替换掉
  - 当局部性很差的时候会成为主要的缺失类型

- 容量(Capacity)缺失
  - Cache太小不足以保持需要的每一个数据
  - 相同容量情况下，在全相联cache (采用最优替换策略)中也可能发生

- 冲突(Conflict)缺失
  - 不属于强制缺失和容量缺失的任何其它缺失情况

14

14

Cache的性能

15

Cache的参数vs. 缺失率

- Cache大小

- 块大小

- 相联度

- 替换策略

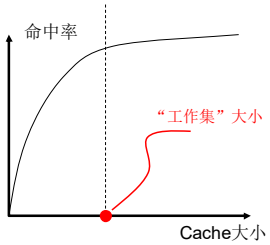
- 插入/放置策略

16

16

Cache 大小

- Cache大小: 总的数据(不包含标签等)容量
  - 越大越能够更好地利用时间局部性
  - 越大**并不总是**越好
- 太大的cache对命中和缺失延迟都会有不利影响
  - 越小越快=> 越大越慢
  - 访问时间可以缩短关键路径
- 太小的cache
  - 不能很好地利用时间局部性
  - 有用的数据也会经常替换



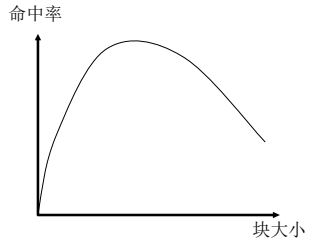
- 工作集: 执行应用时会引用的所有数据的集合
  - 在一个时间段之内

17

17

块大小

- 块大小是一个与地址标签关联的数据
  - 不一定是层次结构中层次之间移动的数据单元
    - 子块: 块被细分为多个片段 (每个片段都带有效位)
      - 可以提升“写”性能
- 太小的块
  - 不能很好地利用空间局部性
  - 标签的开销更大
- 太大的块
  - 块的数量太少
    - 很可能导致无用的数据移动
    - 消耗额外的带宽/电能



18

18

大的块: 关键字与子块

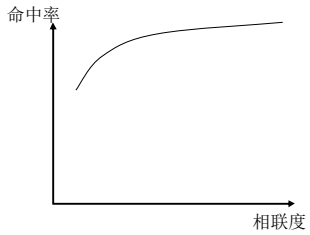
- 大的cache块需要更长的时间去填入cache
  - 填写cache line时“关键字”优先
  - 完全填入之前可以重启cache的访问
- 大cache块会浪费总线带宽
  - 块细分为子块
  - 每个子块有独立的有效位
  - 什么时候有用?

19

19

相联度

- 多少块可以映射到同一个索引(或者set)?
- 更大的相联度
  - 更小的缺失率, 程序之间的差异性较小
  - 边际收益递减, 更高的命中延迟
- 更小的相联度
  - 更小的开销
  - 更低的命中延迟
    - 对 L1 cache尤其重要



20

20

如何减少各种缺失

- 强制缺失
  - 高速缓存机制本身起不到效果
  - 预取
- 冲突缺失
  - 更高的相联度
  - 用不通过cache相联的其他方法获得更高的相联度
    - 牺牲者cache
    - 哈希
    - 软件?
- 容量缺失
  - 更好地利用cache空间: 保持将会被引用的块
  - 软件管理: 将工作集切分为多个“段”以适配cache的容量

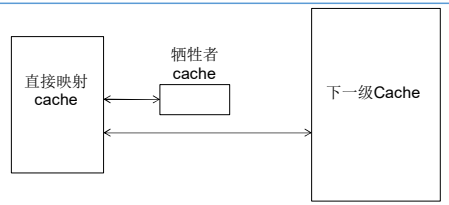
改善 Cache “性能”

- 回忆
  - 平均存储访问时间 (AMAT)
    = (命中率 * 命中延迟) + (缺失率 * 缺失延迟)
- 降低缺失率
  - 提示:当有较多的有重取开销的块被替换掉时，降低缺失率可能降低性能
- 降低缺失延迟/开销
- 降低命中延迟

改善Cache性能

- 降低缺失率
  - 更高的相联度
  - 相联的替代/增强
    - 牺牲者cache, 哈希, 伪相联, 偏斜相联
  - 更好的替换/插入策略
  - 软件的方法
- 降低缺失延迟/开销
  - 多层cache
  - 关键字优先
  - 子块/分区
  - 更好的替换/插入策略
  - 非阻塞cache (多个cache缺失并发)
  - 每周期多次访存
  - 软件的方法

牺牲者Cache: 降低冲突缺失



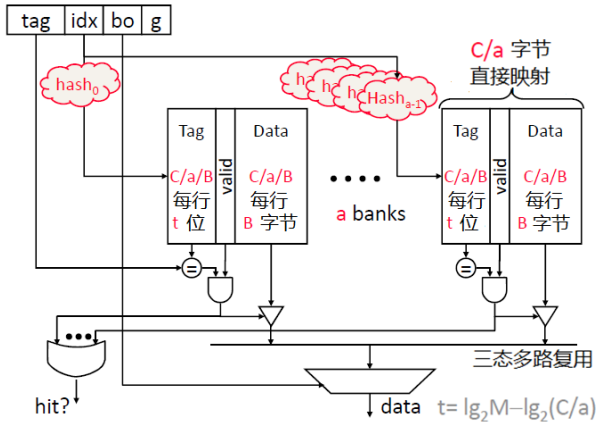
- Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,” ISCA 1990.
- 思路: 用一个小的全相联缓冲 (牺牲者 cache)存储被换出的块
  + 可以避免cache块交替映射到相同的set (两个cache块连续在相邻的时间被互相冲突的访问)
  -- 如果是串行访问L2 cache，会增加缺失延迟

哈希和伪相联

- 哈希: 更好的“随机化”索引函数
  - + 能够减少冲突缺失
    - 更均匀地分布内存块到set
  - -- 实现更复杂: 可能加长关键路径
- 伪相联 (“穷人相联” cache)
  - 串行查找: 缺失时, 采用不同的索引函数再次访问cache
  - 对K个cache块直接映射

25

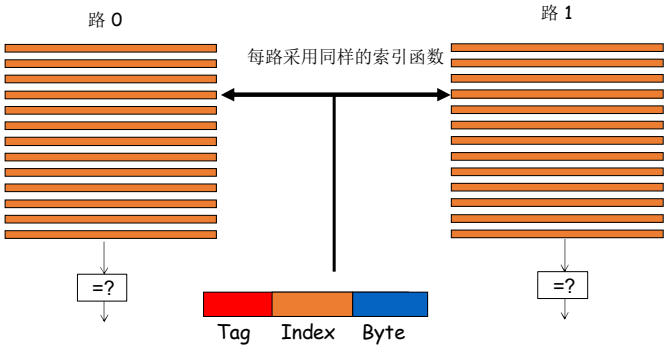
偏斜组相联



26

偏斜相联Cache (I)

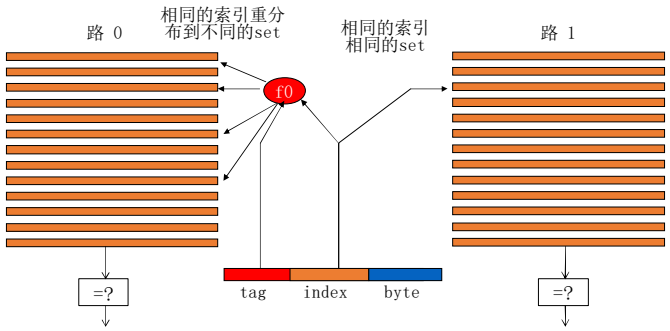
- 基本的2路相联 cache 结构



27

偏斜相联Cache (II)

- 偏斜相联cache
  - 每个 bank 有不同的索引函数



28

偏斜相联Caches (III)

- 思路: cache的每1路采用不同的索引函数以减少冲突缺失
- 收益: 索引被随机化了
  - 两个块拥有相同索引的可能很小
    - 减小冲突缺失
  - 也许能够减少相联度
- 开销: 引入哈希函数的额外延迟
- Seznec, "A Case for Two-Way Skewed-Associative Caches," ISCA 1993.

29

29

通过软件改善命中率 (I)

- 重新组织数据的布局
- 例如: 采用列优先
  - 在内存中，x[i+1,j] 紧跟在 x[i,j] 后面
  - x[i,j+1] 则离 x[i,j] 很远

糟糕的代码

```
for i = 1, rows
    for j = 1, columns
        sum = sum + x[i,j]
```

较好的代码

```
for j = 1, columns
    for i = 1, rows
        sum = sum + x[i,j]
```

- 这叫做循环交换
- 其他的优化也可以提升命中率
  - 循环融合, 数组合并, …
- 多数组? 编译时数组大小未知?

30

30

数据结构的布局

```
struct Node {
    struct Node* node;
    int key;
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access other fields of node
    }
    node = node→next;
}
```

- 基于指针的遍历(比如, 链表)
- 假设一个巨大的链表(1百万个节点，对应的键值)
- 为什么左边的代码命中率超低?
  - "Other fields"占据了绝大多数的cache line，即使它们极少被访问!

31

31

如何改进?

```
struct Node {
    struct Node* node;
    int key;
    struct Node-data* node-data;
}

struct Node-data {
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access node→node-data
    }
    node = node→next;
}
```

- 思路: 将数据结构中经常使用的域分离出来，"装进"一个独立的数据结构中
- 这件事应该谁来做?
  - 程序员
  - 编译器
    - 静态 vs. 动态
  - 硬件?
  - 谁能决定什么是经常使用的?

32

32

通过软件改善命中率(II)

- 分块
  - 把对数组的循环操作切分成块，每个块可以在cache中保有自己的数据
  - 避免不同块之间的冲突
  - 本质上: 切分工作集使得每一个子集适合cache的访问
- 仍然存在冲突的可能性
  1. 不同数组之间可能有冲突
  2. 编译/编码时不一定能够确定数组的大小

33

33

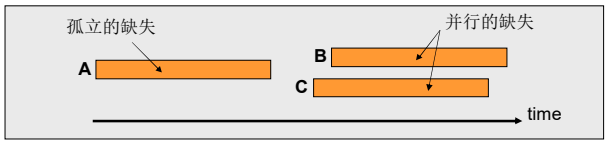
改善Cache性能

- 降低缺失率
  - 更高的相联度
  - 相联的替代/增强
    - 牺牲者caches, 哈希, 伪相联, 偏斜相联
  - 更好的替换/插入策略
  - 软件的方法
- 降低缺失延迟/开销
  - 多层cache
  - 关键字优先
  - 子块/分区
  - 更好的替换/插入策略
  - 非阻塞cache (多个cache缺失并发)
  - 每周期多次访存
  - 软件的方法

34

34

存储级并行(MLP)



- 存储级并行(MLP) 意味着并行地生成和维护多个访存行为[Glew' 98]
- 很多技术可以用来提升MLP (比如, 乱序执行)
- MLP有各种不同的情况，有些缺失是孤立的有些是并行的

这些会如何影响cache的替换?

35

35

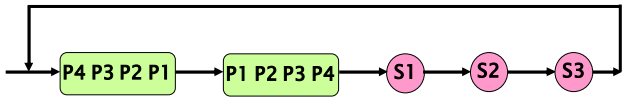
传统的 Cache 替换策略

- 传统的 Cache 替换策略主要目标是减少缺失的数量
- 隐含的假设: 减少缺失数会减少与存储相关的停顿时间
- 不同开销的缺失以及MLP使这个假设不再成立!
- 消除孤立的缺失比消除并行的缺失对性能帮助更大
- 消除更高延迟的缺失比消除较低延迟的缺失对性能帮助更大

36

36

一个例子



块 P1, P2, P3, P4 的缺失可以并行
块 S1, S2, S3 的缺失是孤立的

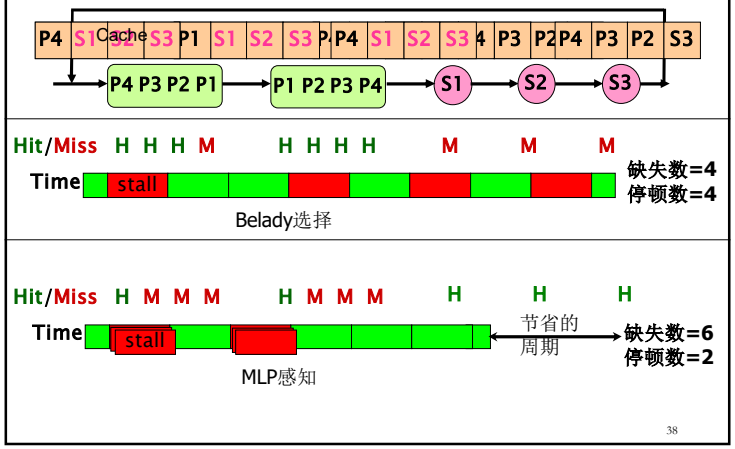
两种替换算法:
1. 最小化缺失数 (Belady选择)
2. 减少孤立缺失 (MLP感知)

Cache有4行，全相联

37

37

最少的缺失≠最佳的性能



38

38

MLP感知的 Cache 替换

- 该如何在制定替换策略时把MLP考虑进去?
- Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

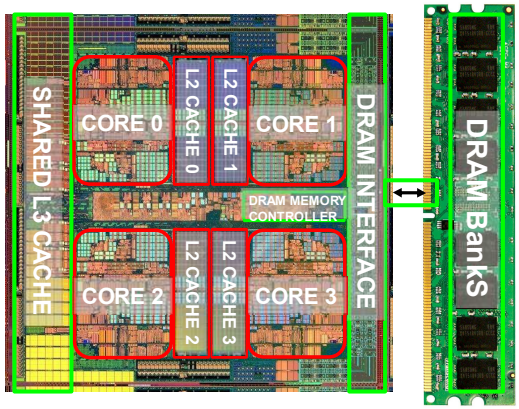
39

39

主存储器

40

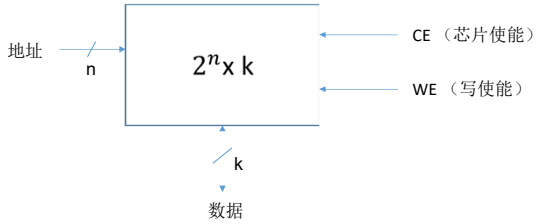
系统中的主存储器



41

41

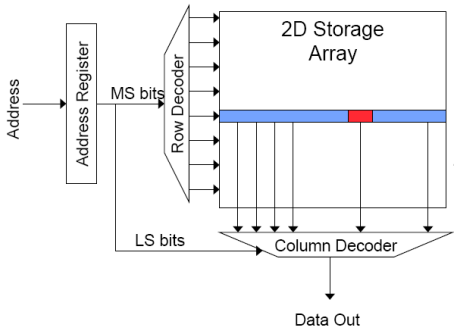
存储芯片/系统的抽象



42

42

回顾：存储器Bank的组织和操作



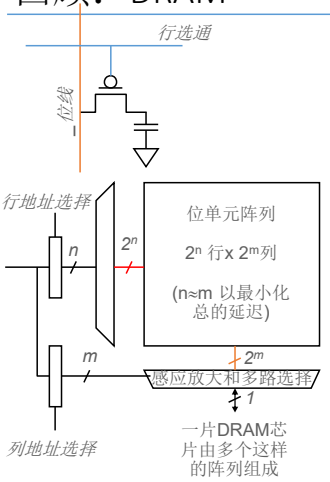
- 读访问过程:

1. 译码行地址并驱动字线
2. 选择位驱动位线
   - 读整行
3. 放大行数据
4. 译码列地址并选择行的子集
   - 发送至输出
5. 预充电位线
   - 为下次访问做准备

43

43

回顾：DRAM



每一位以存储在位元节点的电容中的电荷来表达

- 读位元时会损失电荷
- 位元随时间损失电荷

读的过程

1. 地址译码
2. 驱动行选通
3. 被选的位单元驱动位线 (整行一起读)
4. 触发器感应放大位线，数据位经多路选择输出
5. 预充电所有位线

破坏性的读取

随时间损失电荷

刷新: DRAM控制器必须在刷新时间内定期读取每一行

44

44

基本概念(I)

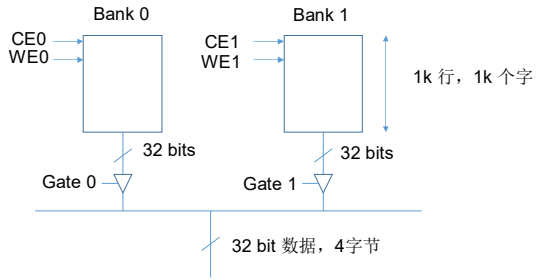
- 物理地址空间
  - 主存的最大尺寸: 可被唯一标识的位置总数
- 物理寻址能力
  - 主存中数据可被寻址的最小尺寸
  - 字节寻址, 字寻址, 64-bit-寻址
  - 可寻址能力依赖的是实现的抽象层次
- 对齐
  - 硬件能否对软件透明的支持非对齐的访问?
- 交叉存取

基本概念(II)

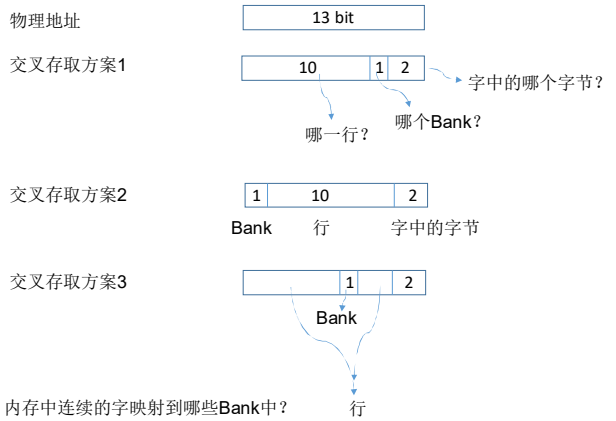
- 交叉存取 (堆叠)
  - 问题: 单片的存储阵列访问时间很长，并且无法并行执行多个访存
  - 目标: 减小对存储阵列访问的延迟，并且能够并行执行多个访存
  - 思路: 将存储阵列划分为多个可以(在同一个周期或连续的周期)独立访问的Bank
    - 每个 Bank 都比整个存储空间小
    - 可以重叠地访问不同的 Bank
  - 需要解决的难题: 如何将数据映射到不同的Bank? (如何在不同的Bank之间交叉存取数据?)

交叉存取—示例

假设每个Bank一次存取1个字
内存中连续的字映射到哪些Bank中？ ⇐ 如何在Bank之间交叉存取这些字？



交叉存取方案



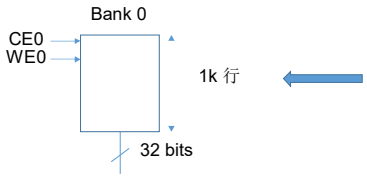
几个问题/概念

- CRAY-1 有 16 个Bank
  - 11 个时钟周期的 Bank 延迟
  - 主存中连续的字放在连续的 Bank 中 (字交叉存取)
  - 每个时钟周期可以开始(和结束)一次访存
- Bank 可以被完全并行的操作吗?
  - 每个周期开始多次访存?
- 这样做的开销是什么?
- 现代超标量处理器的L1数据cache有多个完全独立的Bank

49

49

Bank的抽象

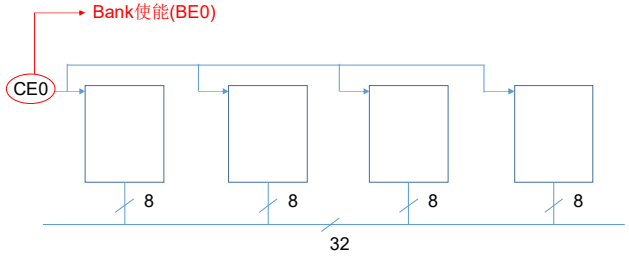


这是一个抽象。实际的32 bit可能来自n个不同的芯片，每一个芯片提供32/n bit

50

50

Rank



这种组织形式构成了"Rank" (上图只画出了Bank 0 的情况)
Rank: 同时响应同一命令对同一地址进行访问的一组芯片，每个芯片提供请求数据的不同片段

为什么？ 生产 8bit 输出的芯片比生产32bit 输出的芯片便宜

思路： 生产8bit 的芯片，但是以Rank的形式控制/操作，可以一次读取32bit 数据

51

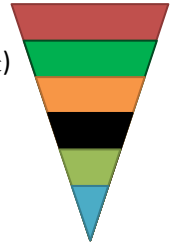
51

DRAM 子系统

52

DRAM 子系统的组织

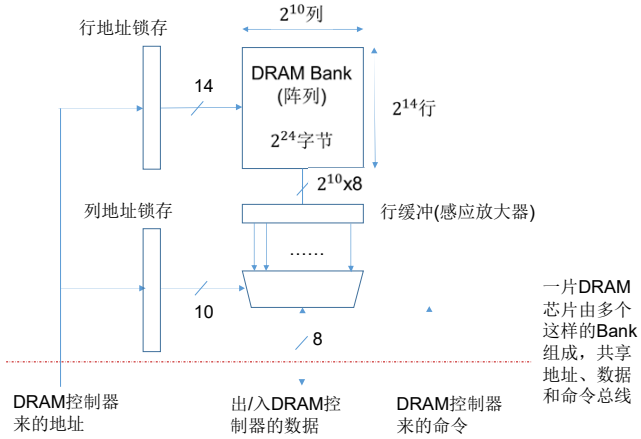
- 通道
- DIMM(双列直插式存储模块)
- Rank
- 芯片
- Bank
- 行/列



53

53

DRAM Bank 的结构



54

54

页模式DRAM

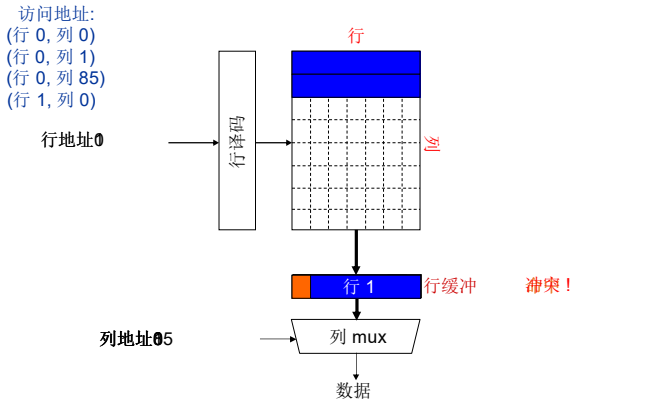
- DRAM Bank 是由位元组成的二维阵列: 行 x 列
- “DRAM 行”也被称作“DRAM 页”
- “感应放大器”也被称作“行缓冲”

- 每个地址是一个<行,列> 对
- 访问一个“关闭的行”，需要执行
  - 激活命令打开行 (放入行缓冲)
  - 读/写命令读/写行缓冲中的列
  - 预充电命令关闭行，同时为下一次访问Bank做准备
- 访问一个“打开的行”
  - 不需要执行激活命令

55

55

DRAM Bank 操作



56

56

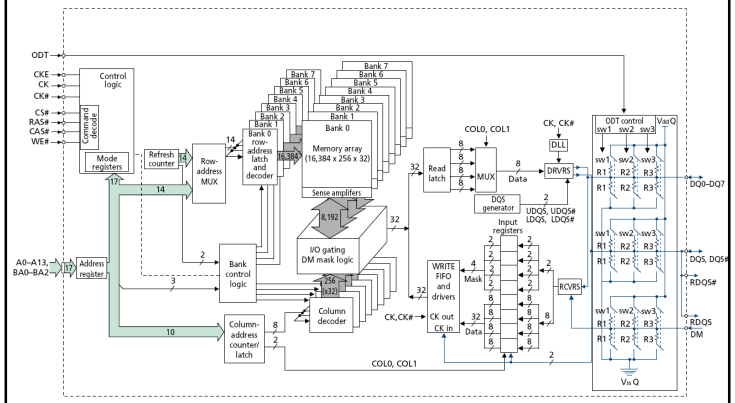
DRAM 芯片

- 由多个Bank组成 (SDRAM中通常有2-16个)
- Bank之间共享命令/地址/数据总线
- 芯片本身有一个窄接口 (每次读4-16 位)

57

57

128M x 8-bit DRAM 芯片



58

58

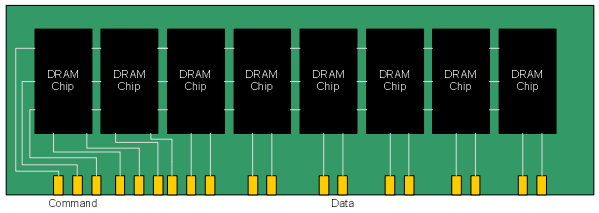
DRAM Rank 和模块

- Rank: 多个芯片一起操作构成宽接口
- 组成Rank的所有芯片同时被控制执行操作
  - 响应一个命令
  - 共享地址和命令总线, 但提供不同的数据
- DRAM 模块由1个或多个Rank构成
  - 比如, DIMM (双列直插存储模块)
- 如果内存条的芯片有8位接口, 一次访存要读取8个字节, DIMM需要8个芯片

59

59

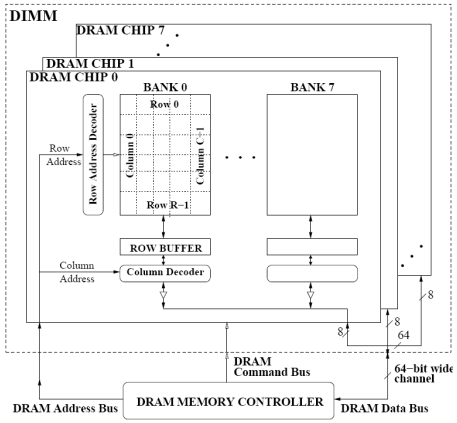
64位宽DIMM (1个Rank)



60

60

64位宽DIMM (1个Rank)

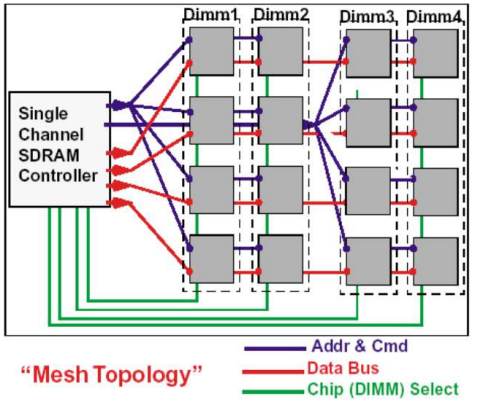


- 优点:
  - 看起来就像一个有宽接口的大容量 DRAM 芯片
  - 灵活性: 内存控制器不需要控制单个芯片
- 缺点:
  - 粒度: 访问不能小于接口宽度

61

61

多个 DIMM

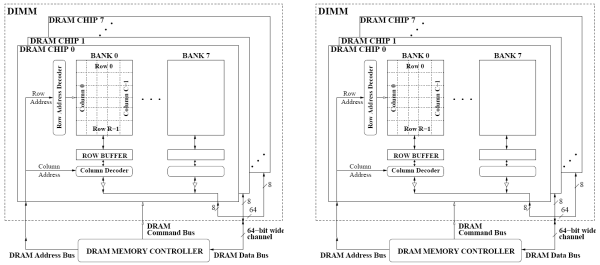


- 优点:
  - 允许更大的容量
- 缺点:
  - 互连的复杂性和能耗都比较高

62

62

DRAM 通道

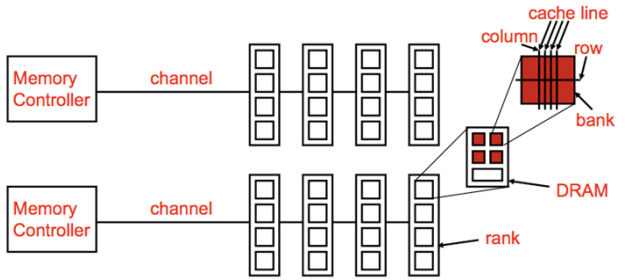


- 2 个独立通道: 2个内存控制器
- 2 个非独立/同步通道: 1个有宽接口的内存控制器

63

63

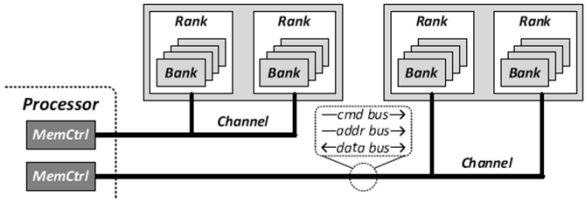
通用(广义的)内存结构



64

64

通用(广义的)内存结构



65

65

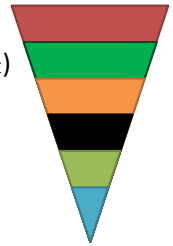
DRAM 子系统
自顶向下的视角

66

66

DRAM 子系统的组织

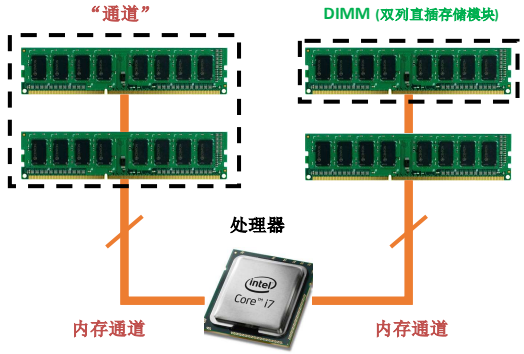
- 通道
- DIMM(双列直插式存储模块)
- Rank
- 芯片
- Bank
- 行/列



67

67

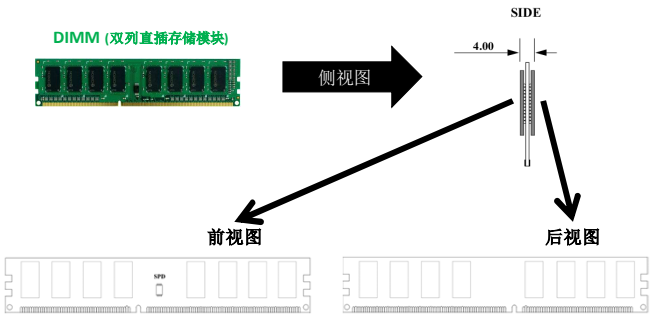
DRAM 子系统



68

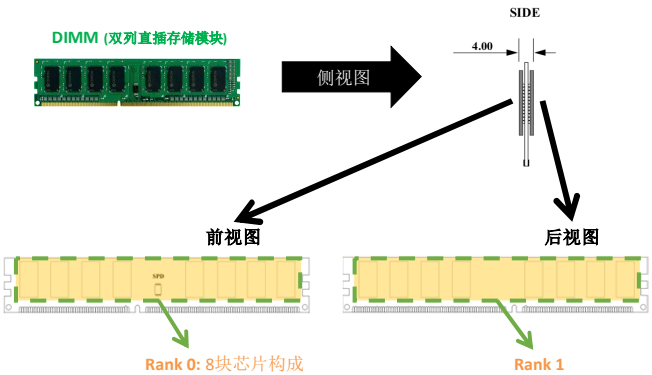
68

分解 DIMM



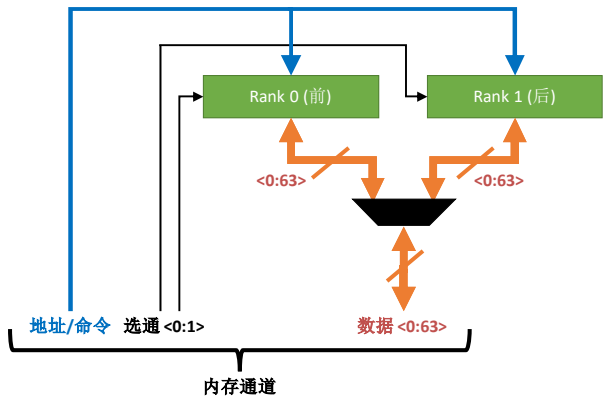
69

分解 DIMM



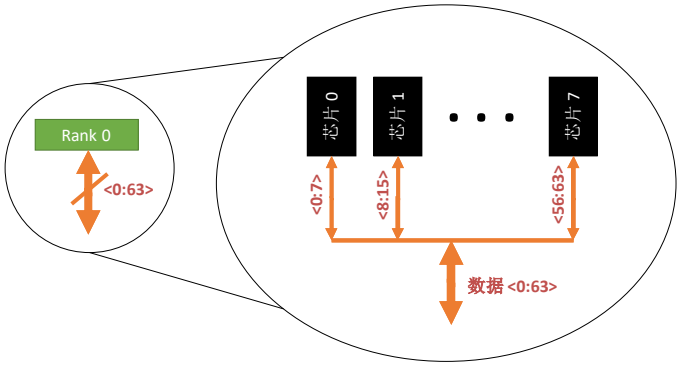
70

Rank



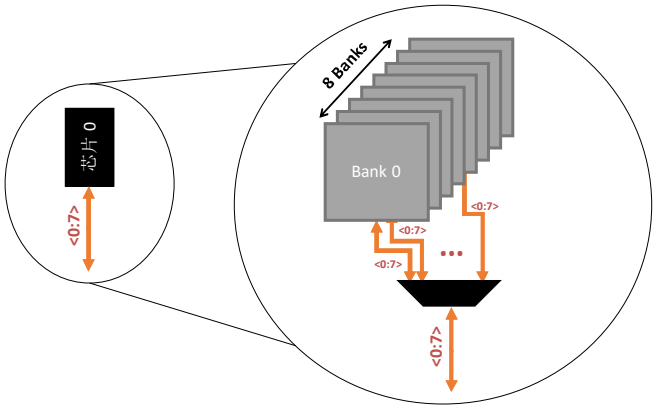
71

分解 Rank



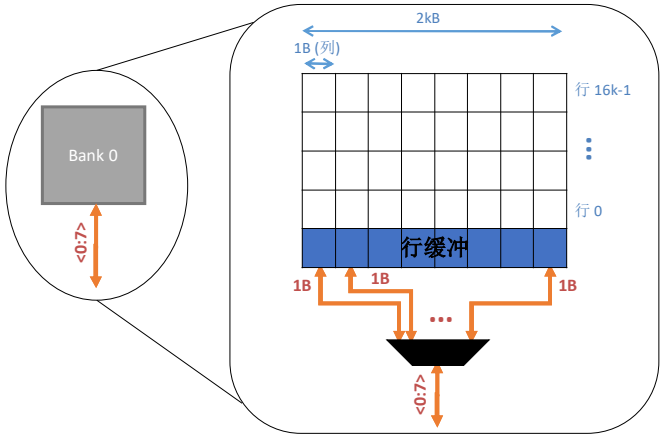
72

分解芯片



73

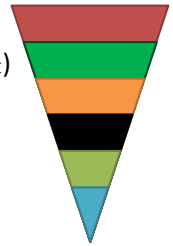
分解Bank



74

DRAM 子系统的组织

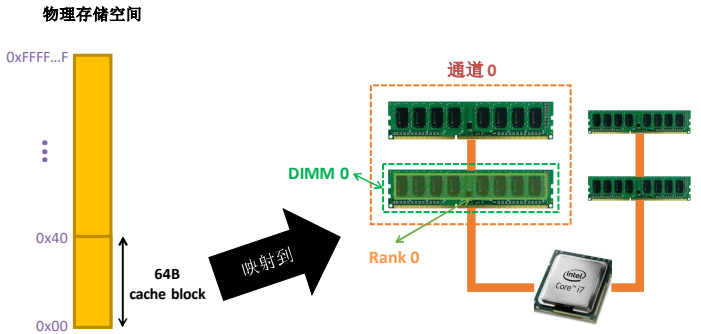
- 通道
- DIMM(双列直插式存储模块)
- Rank
- 芯片
- Bank
- 行/列



75

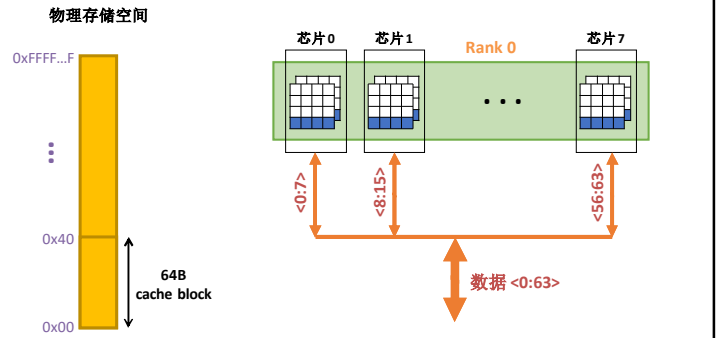
75

例子: 移动一个 cache block



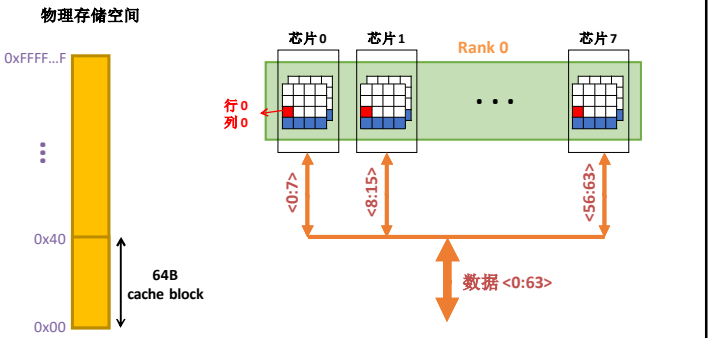
76

例子: 移动一个 cache block



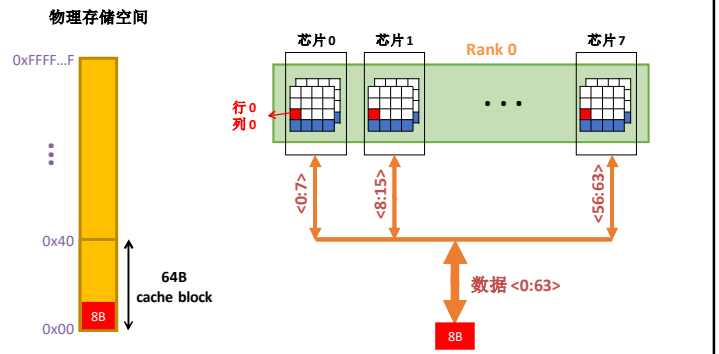
77

例子: 移动一个 cache block



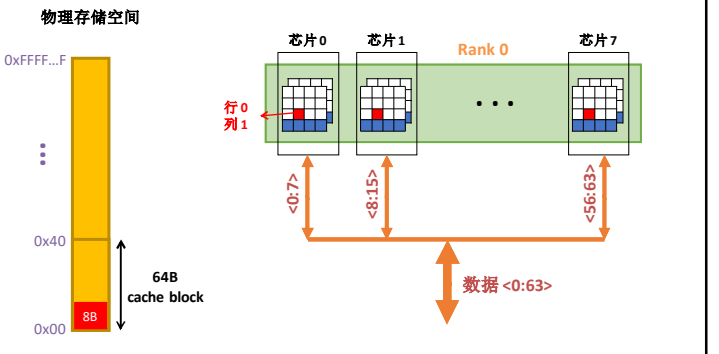
78

例子: 移动一个 cache block



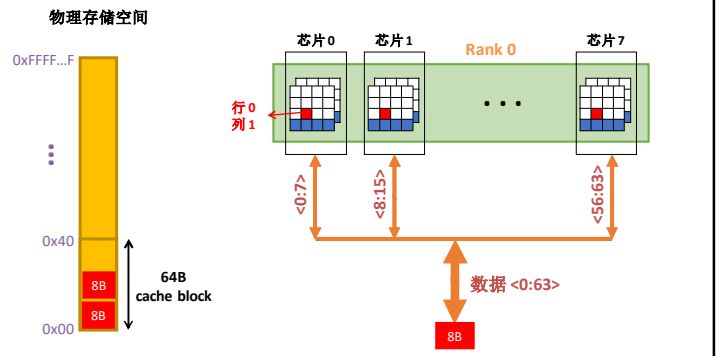
79

例子: 移动一个 cache block



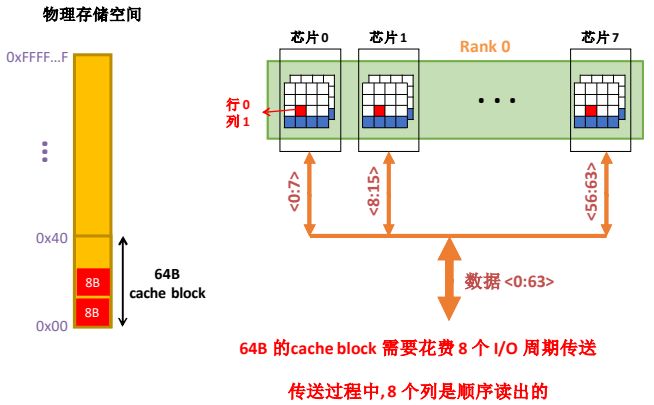
80

例子: 移动一个 cache block



81

例子: 移动一个 cache block



82

延迟组件: 基本的DRAM操作

- CPU → 控制器的传输时间
- 控制器延迟
  - 控制器中排队和调度的延迟
  - 访存被转换为基本命令
- 控制器 → DRAM的传输时间
- DRAM Bank 延迟
  - 如果行已经打开，则简单的列选通，或者
  - 如果阵列已经预充电则行选通 + 列选通，或者
  - 预充电 + 行选通 + 列选通 (最坏的情况)
- DRAM → CPU 的传输时间 (通过控制器)

83

83

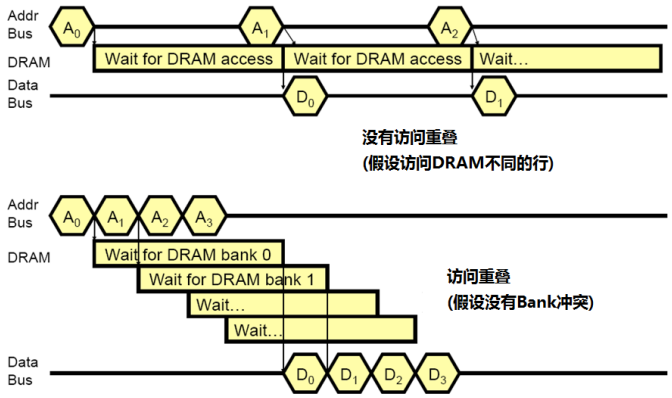
多Bank(交叉存取) 和多通道

- 多个Bank
  - 能够被并发访问
  - 地址中的某些位决定了哪一个Bank才是这个地址驻留的位置
- 多个独立的通道服务于同一个目的
  - 这样会更好，因为这些通道有独立的数据总线
  - 增加总线带宽
- 要获得更多的并发性需要减少
  - Bank的冲突
  - 通道的冲突
- 如何在地址中选择/随机分配 Bank/通道的编号?
  - 低位具有更高的熵值
  - 随机哈希函数 (不同地址位异或)

84

84

多Bank/通道的好处



85

85

多通道

- 优点
  - 增加带宽
  - 并发访问 (如果通道独立的话)
- 缺点
  - 比单通道的成本高
    - 板上的线更多
    - 更多的管脚 (如果是片上内存控制器)

86

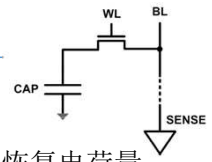
86

DRAM 刷新

87

DRAM 刷新

- DRAM的电容会随着时间推移漏电



- 内存控制器需要阶段性地刷新DRAM行恢复电荷量
  --每N个ms读并且关闭每行一次
  --典型的N = 64 ms
- 刷新的缺陷
  -- 能耗: 每次刷新消耗能量
    -- 性能下降: DRAM Rank/Bank 在刷新时不可用
    -- QoS的影响: 刷新时暂停时间长
    -- 刷新率限制了DRAM容量的扩展能力

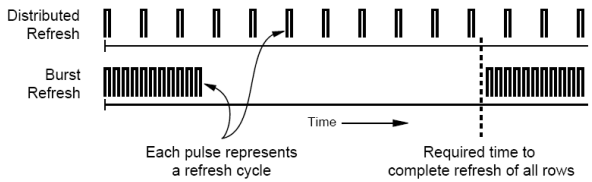
88

88

DRAM 刷新: 性能

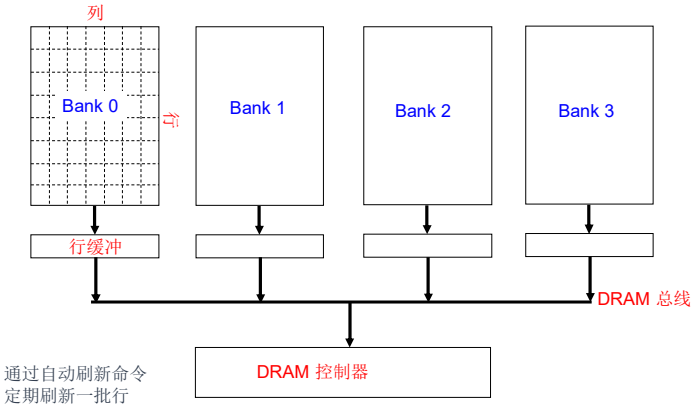
- 刷新对性能的影响
  - -- DRAM Bank在刷新时不可用
  - -- 长的暂停时间: 如果集中刷新所有行，那么意味着每个 64ms DRAM 就会不可用一段时间
- 集中式刷新: 所有行在前一行刷新完成后立即刷新
- 分散式刷新: 每存储周期刷新一行
- 分布式刷新: 每一行按照固定的间隔在不同时间刷新

分布式刷新

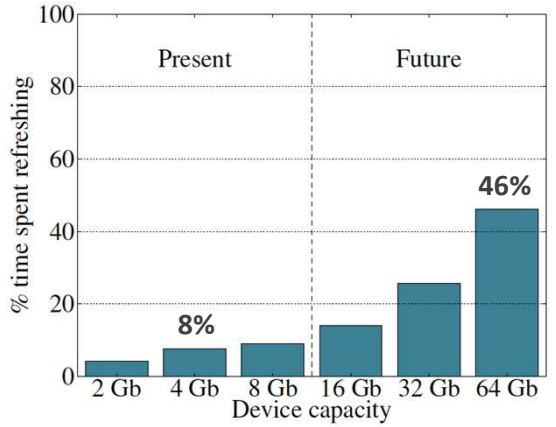


- 分布式刷新可以消除长的暂停时间
- 还能如何减少刷新对性能/QoS的影响?
- 分布式刷新能减少对能耗的影响吗?
- 是否能够减少刷新的数量?

现在的刷新技术: 自动刷新

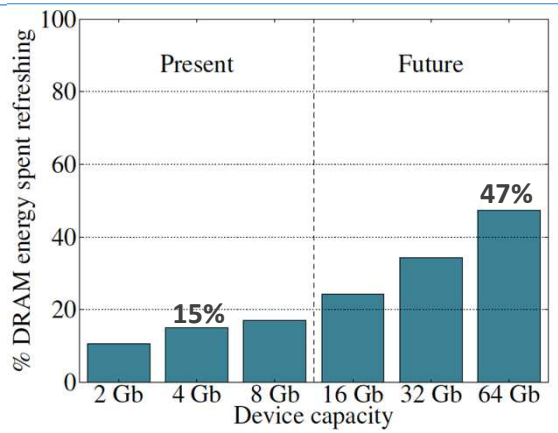


刷新的开销: 性能



Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

刷新的开销: 能耗



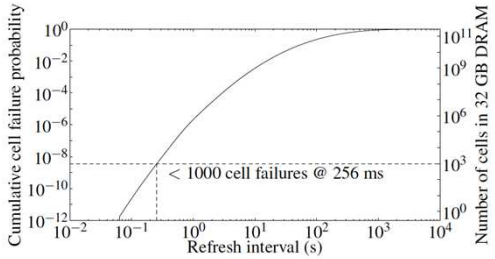
Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

93

93

传统刷新方法的问题

- 每一行以同样的频率刷新



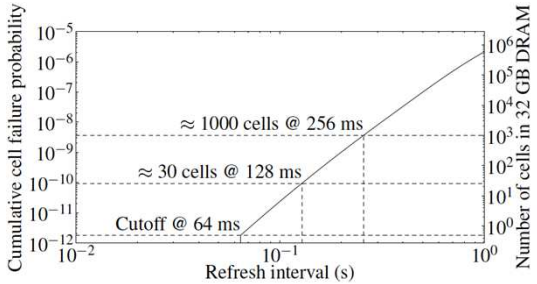
- 观察: 大多数行可以通过不那么频繁的刷新就能保证不丢失数据 [Kim+, EDL'09]
- 问题: DRAM中并不支持为每一行设置不同的刷新率

94

94

DRAM 行的保持时间

- 观察: 只有很少几行需要按照最坏情况的频率刷新



- 可以利用这一点在低成本基础上减少刷新操作吗?

95

95

减少刷新操作

- 思路: 确定不同行的保持时间，根据每行需要的刷新频率刷新每一行
- (注重成本的) 思路: 根据最小保持时间把行分组，再按照每组特定的刷新频率对组内行进行刷新
  - 比如, 64-128ms刷新的组, 128-256ms刷新的组, …
- 观察: 只有很少的行需要很高刷新频率 [64-128ms] → 只有很少的几组→ 用低的硬件开销实现刷新操作的大幅度减小
- Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

96

96

RAIDR: 机制

1. 分析: 分析DRAM所有行的保持时间
   → 可以在DRAM设计的时候，也可以动态分析

2. 分组: 根据保持时间分组“存储”行
   → 采用 Bloom Filter 实现高效和可扩展的存储

97

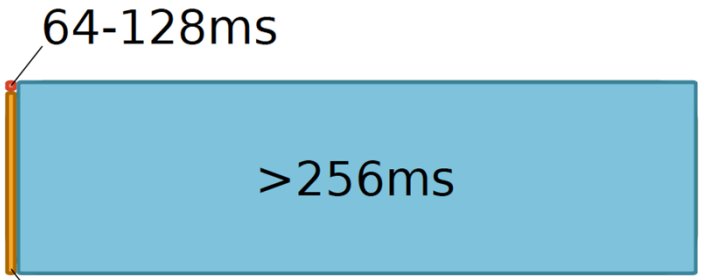
97

RAIDR: 机制

98

98

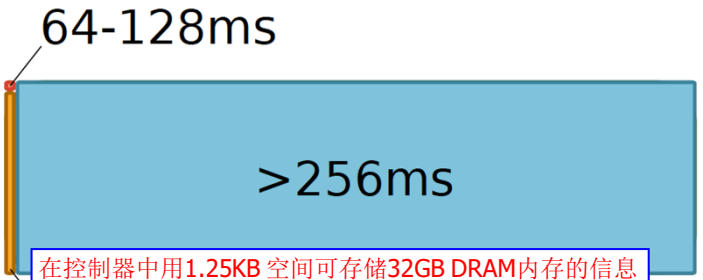
RAIDR: 机制



99

99

RAIDR: 机制



100

100

RAIDR: 机制

1. 分析: 分析DRAM所有行的保持时间
   → 可以在DRAM设计的时候，也可以动态分析

2. 分组: 根据保持时间分组“存储”行
   → 采用 Bloom Filter 实现高效和可扩展的存储

3. 刷新: 内存控制器以不同的频率刷新不同分组内的行
   → 通过探测Bloom Filter确定一行的刷新频率

101

101

分析

- 分析一行
  - 写一行数据
  - 保持行不被刷新
  - 测量数据损坏需要的时间

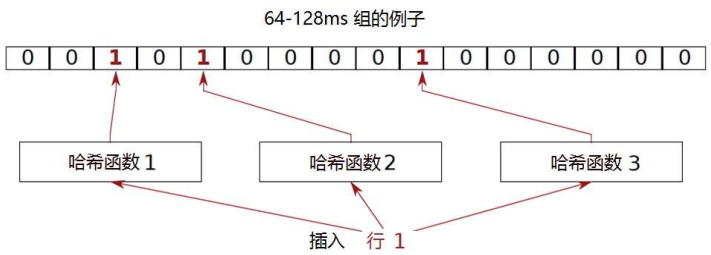
	行1	行2	行3
初始值	11111111…	11111111…	11111111…
64ms之后	11111111…	11111111…	11111111…
128ms之后	11011111… (64–128ms)	11111111…	11111111…
256ms之后		11111011… (128–256ms)	11111111… (>256ms)

102

102

分组

- 如何高效并且可扩展地将多个行按照保持时间分组?
  - 采用硬件Bloom Filters [Bloom, CACM 1970]



Bloom, “Space/Time Trade-offs in Hash Coding with Allowable Errors”, CACM 1970.

103

103

Bloom Filter

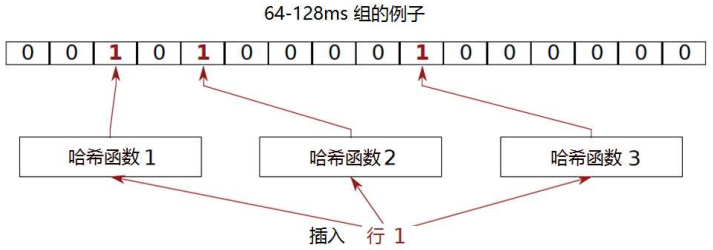
- [Bloom, CACM 1970]
- 一种能够简洁描述集合成员关系(元素在/不在集合内)的概率数据结构
- 非近似集合成员: 每个元素用1bit表示该元素存在/不在一个由N个元素组成的元素空间中
- 近似集合成员: 使用比元素个数少得多的bit的子集表示每个元素的成员关系(存在/不在)
  - 一些元素映射到的bit也会被其他元素映射到
- 操作: 1) 插入, 2) 测试, 3) 移除所有元素

Bloom, “Space/Time Trade-offs in Hash Coding with Allowable Errors”, CACM 1970.

104

104

Bloom Filter 操作示例

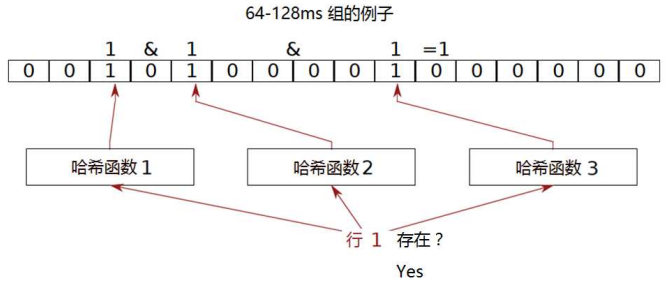


Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", CACM 1970.

105

105

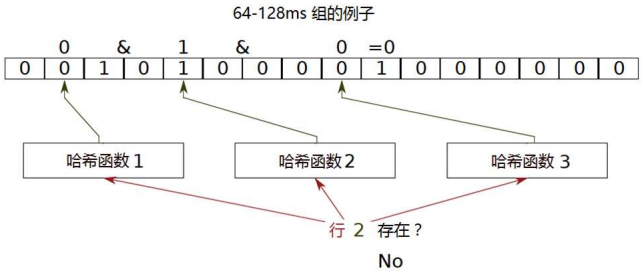
Bloom Filter 操作示例



106

106

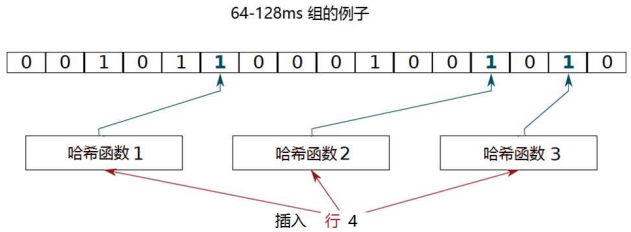
Bloom Filter 操作示例



107

107

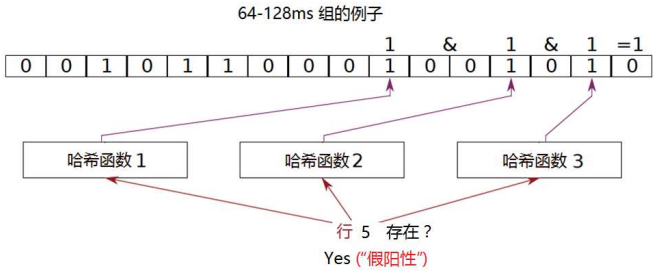
Bloom Filter 操作示例



108

108

Bloom Filter 操作示例



109

109

用Bloom Filter分组的好处

- "假阳性":虽然某一行可能从来没有被插入过，但是它仍然可能被确认存在于Bloom filter中
- 不是问题: 对某些行的刷新频率可能比需要的高
- 没有"假阴性": 行的刷新频率永远不会低于必须的频率 (没有正确性问题)
- 可扩展性: Bloom filter 永远不会溢出 (不像固定尺寸的表)
- 效率: 不需要逐行存储信息; 硬件简单 → 1.25 KB的2个filter用于32 GB DRAM 系统

110

110

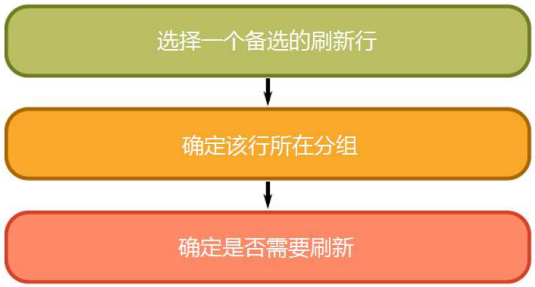
在硬件中使用Bloom Filter

- 当在集合成员测试中出现假阳性是可以容忍的时候，这种方法是很有用的
- 阅读相关的例子
  - Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.
  - Seshadri et al., "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," PACT 2012.

111

111

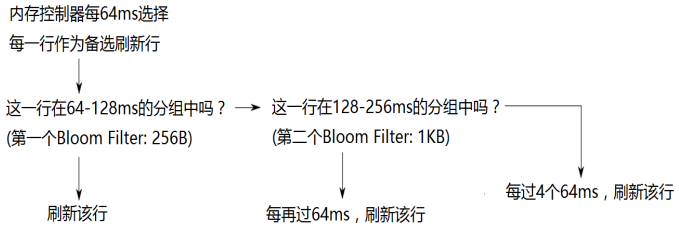
刷新(RAIDR 刷新控制器)



112

112

刷新(RAIDR 刷新控制器)

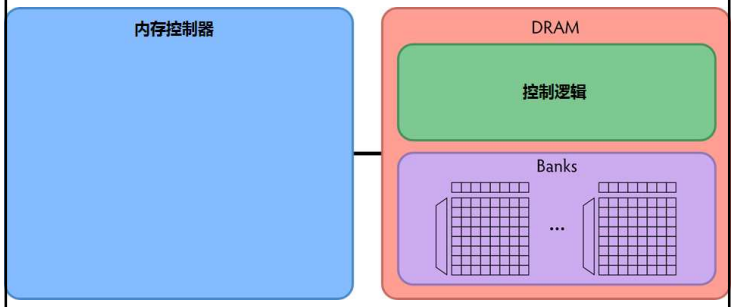


Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

113

113

RAIDR: 基准设计



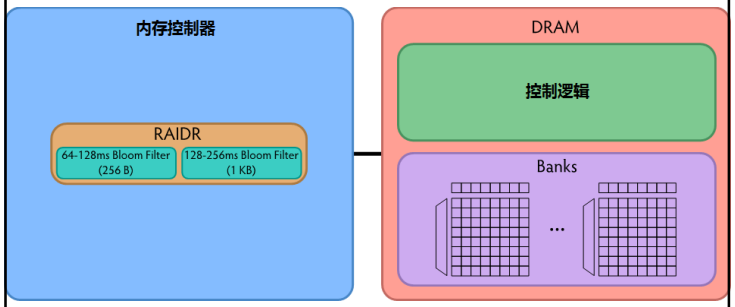
今天的自动刷新系统中，在DRAM里有刷新控制

RAIDR 可以在内存控制器中也可以在DRAM中实现

114

114

RAIDR 在内存控制器中: 选择 1

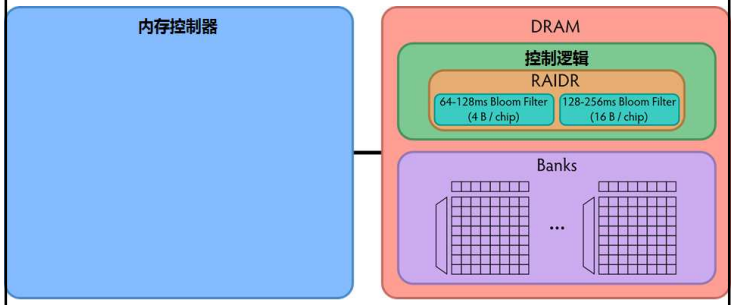


RAIDR在内存控制器中的开销:
1.25 KB Bloom Filter, 3个计数器, 为每行刷新而额外发射的命令
(都是评估中要考虑的因素)

115

115

RAIDR 在 DRAM 芯片中: 选择 2



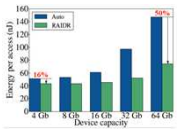
RAIDR 在DRAM芯片中的开销:
每芯片开销: 20B Bloom Filter, 1个计数器 (4 Gbit 芯片)
总开销: 1.25KB Bloom Filter, 64个计数器 (32 GB DRAM)

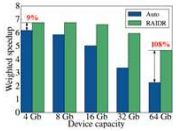
116

116

RAIDR: 一些结果

- 系统: 32GB DRAM, 8-core; SPEC, TPC-C, TPC-H 工作负载
- RAIDR 硬件成本: 1.25 kB (2个 Bloom filter)
- 减少刷新: 74.6%
- 动态减少DRAM能耗: 16%
- 空闲时降低DRAM功耗: 20%
- 性能提升: 9%
- 随着DRAM密度的增大，这些收益会增加





117

117

DRAM 刷新: 更多问题

- 还能做什么来减少刷新的影响?
- 如果知道行的保持时间，还能做些什么?
- 如何精确测量DRAM行的保持时间?
- 推荐阅读:
  - Liu et al., "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," ISCA 2013.

118

118

内存控制器

119

DRAM vs 其它类型的存储器

- 长延迟的存储器具有类似的特性，因此需要控制
- 下面的讨论以DRAM为例, 但是很多关键问题与其它类型存储器的控制器设计遇到的问题类似
  - 闪存
  - 其它新兴的存储器技术
    - 相变存储器
    - 自旋转矩磁存储器

120

120

DRAM 控制器: 功能

- 确保DRAM操作正确(刷新和时序)
- 在遵循DRAM芯片的时序约束下响应DRAM的请求
  - 约束: 资源冲突 (Bank, 总线, 通道), 最小的写-读延迟
  - 将请求翻译成DRAM命令序列
- 缓冲和调度请求以提升性能
  - 重排序, 行缓冲, Bank/Rank/总线管理
- 管理DRAM的功耗和发热
  - 开/关DRAM芯片, 管理功率模式

121

121

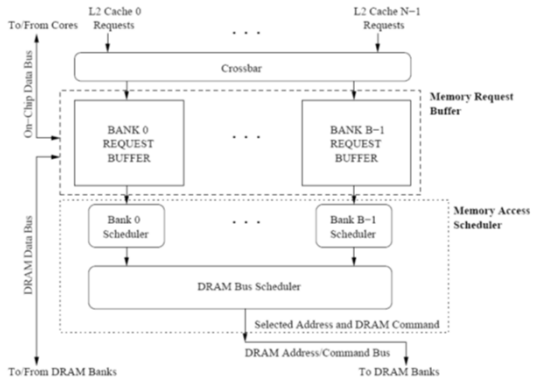
DRAM 控制器: 在什么地方?

- 在芯片组中
  - + 灵活性更高，系统可以插入不同类型的DRAM
  - + CPU芯片内的功率密度低
- 在CPU芯片上
  - + 减少主存访问延迟
  - + 核和控制器之间的带宽高
    - 可以有更多的信息交互

122

122

现代DRAM控制器



123

123

DRAM 调度策略(I)

- FCFS (先来先服务)
  - 最旧的请求最优先
- FR-FCFS (行缓冲优先)
  1. 行命中的优先
  2. 最旧的优先

  目的: 最大化行缓冲命中率 → 最大化DRAM吞吐量
  - 实际上, 调度是在命令层面完成的
    - 列命令 (读/写) 优先于行命令 (激活/预充电)
    - 在每一个组中, 旧的命令优先于新的命令

124

124

DRAM 调度策略(II)

- 调度策略实际上是优先级的序

- 优先级可以基于
  - 请求的新旧
  - 行缓冲命中与否的状态
  - 请求的类型(预取, 读, 写)
  - 请求者的类型 (load miss 还是 store miss)
  - 请求的边界条件
    - 核中最旧的miss?
    - 核中有多少指令依赖于该请求?

125