高等计算机体系结构
第十一讲: 主存储器

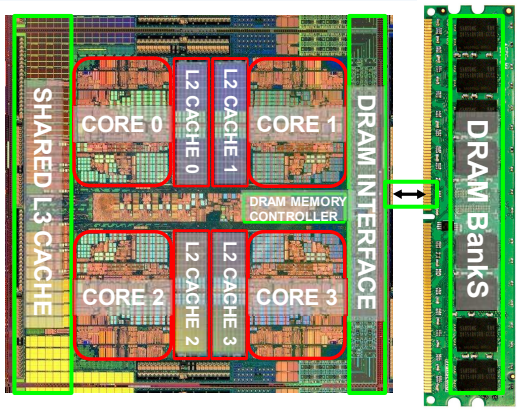
栾钟治
北京航空航天大学 计算机学院 中德联合软件研究所

1

主存储器

2

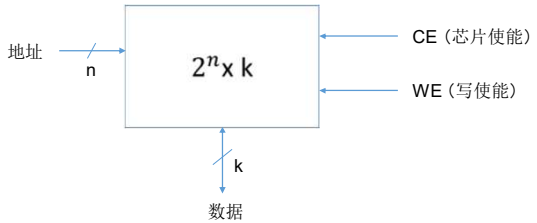
系统中的主存储器



3

3

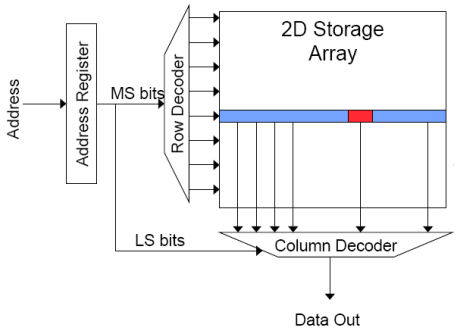
存储芯片/系统的抽象



4

4

回顾：存储器Bank的组织和操作



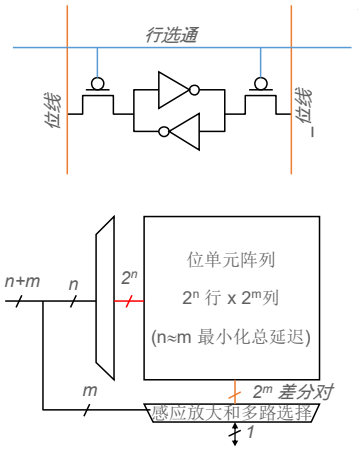
- 读访问过程:

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

5

5

回顾：SRAM



读过程

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. 差分感应并选通列
   (数据准备好)
5. 预充电所有位线
   (为下一次读或者写)

第2、3两步是产生访存延迟的主要阶段

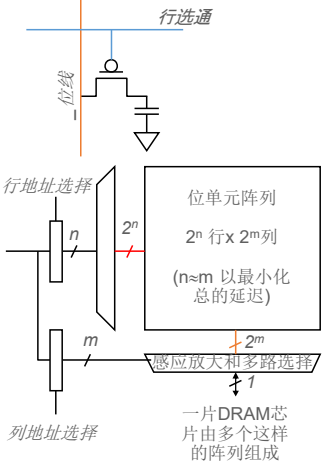
第2、3和5步是整个循环中最耗时的

- 第2步与2^m成正比
- 第3和5步与2^n成正比

6

6

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

7

7

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

8

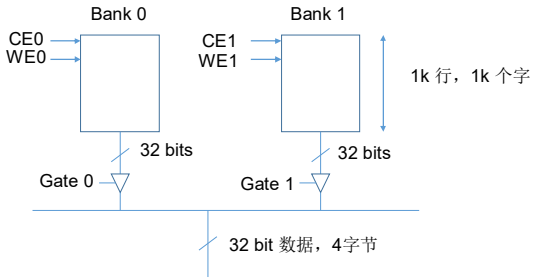
8

基本概念(II)

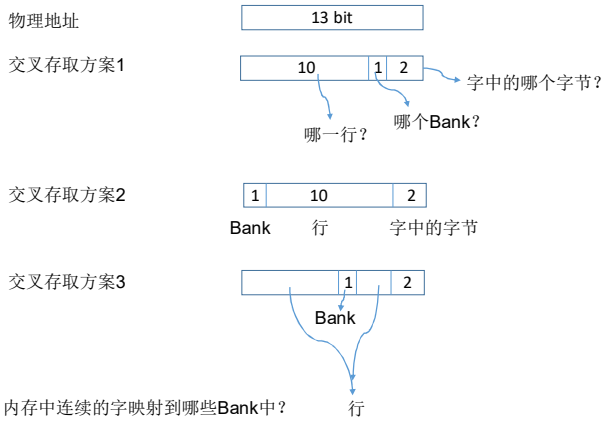
- 交叉存取 (堆叠)
  - 问题: 单片的存储阵列访问时间很长，并且无法并行执行多个访存
  - 目标: 减小对存储阵列访问的延迟，并且能够并行执行多个访存
  - 思路: 将存储阵列划分为多个可以(在同一个周期或连续的周期)独立访问的Bank
    - 每个 Bank 都比整个存储空间小
    - 可以重叠地访问不同的 Bank
  - 需要解决的难题: 如何将数据映射到不同的Bank? (如何在不同的Bank之间交叉存取数据?)

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？



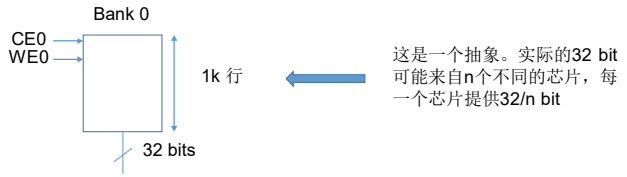
交叉存取方案



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

Bank的抽象

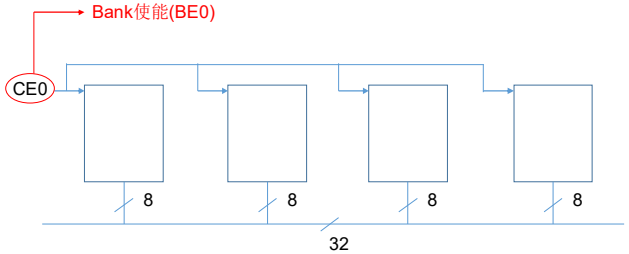


这是一个抽象。实际的32 bit可能来自n个不同的芯片，每一个芯片提供32/n bit

13

13

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

14

14

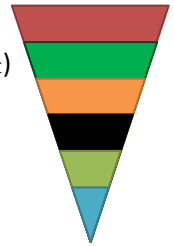
DRAM 子系统

15

15

DRAM 子系统的组织

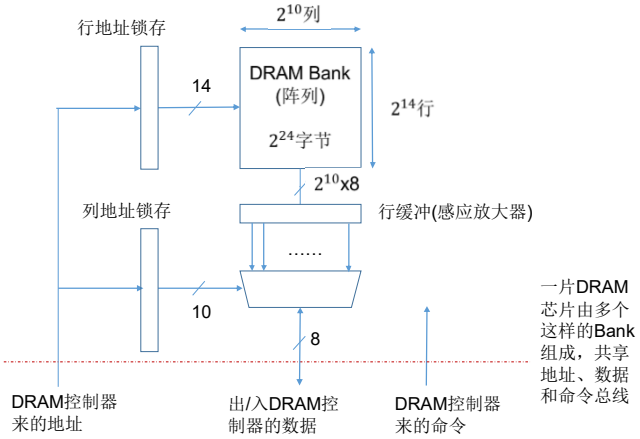
- 通道
- DIMM(双列直插式存储模块)
- Rank
- 芯片
- Bank
- 行/列



16

16

DRAM Bank 的结构



17

17

页模式DRAM

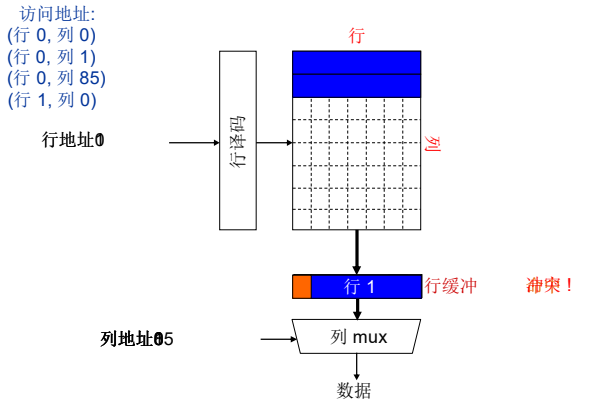
- DRAM Bank 是由位元组成的二维阵列: 行 x 列
- “DRAM 行”也被称作“DRAM 页”
- “感应放大器”也被称作“行缓冲”

- 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

18

18

DRAM Bank 操作



19

19

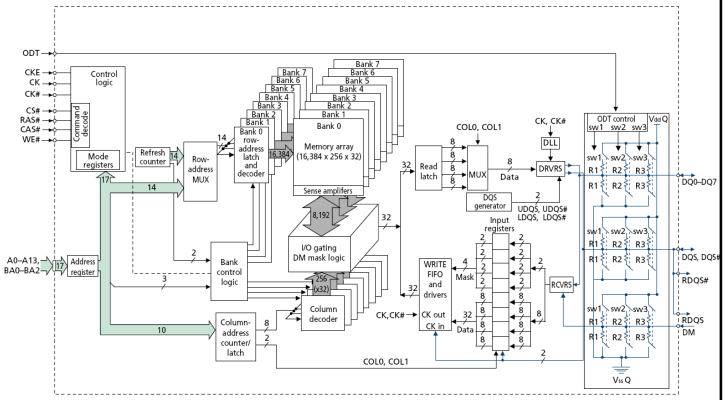
DRAM 芯片

- 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)

20

20

128M x 8-bit DRAM 芯片



21

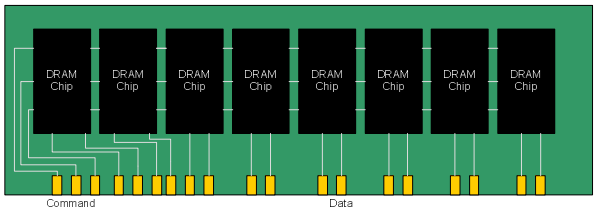
DRAM Rank 和模块

- 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

22

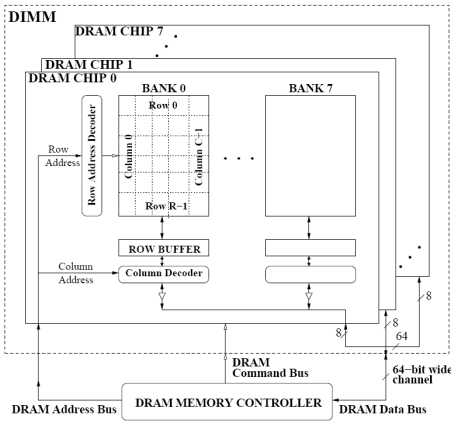
22

64位宽DIMM (1个Rank)



23

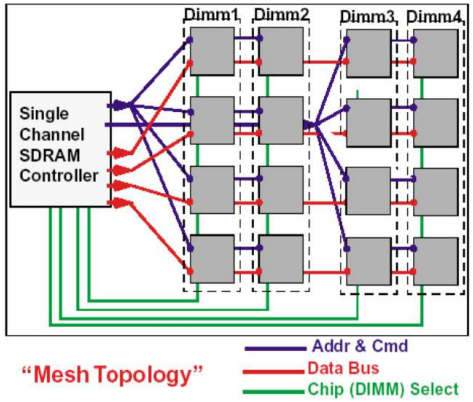
64位宽DIMM (1个Rank)



- 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

24

多个 DIMM

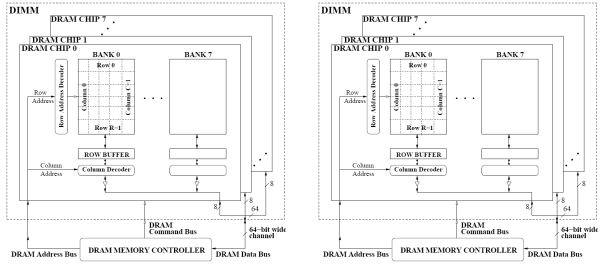


- 优点:
  - 允许更大的容量
- 缺点:
  - 互连的复杂性和能耗都比较高

25

25

DRAM 通道

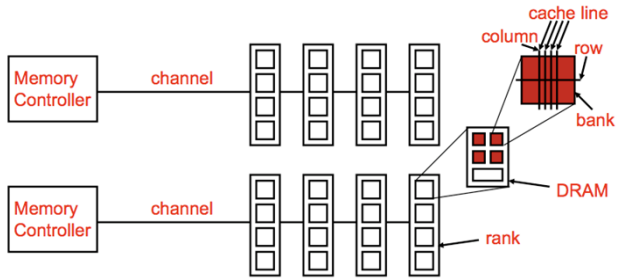


- 2 个独立通道: 2个内存控制器
- 2 个非独立/同步通道: 1个有宽接口的内存控制器

26

26

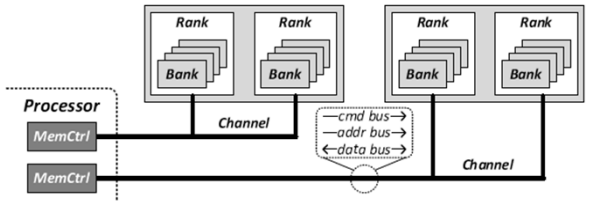
通用(广义的)内存结构



27

27

通用(广义的)内存结构



28

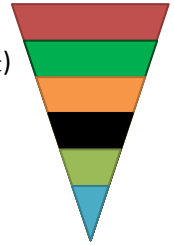
28

DRAM 子系统
自顶向下的视角

29

DRAM 子系统的组织

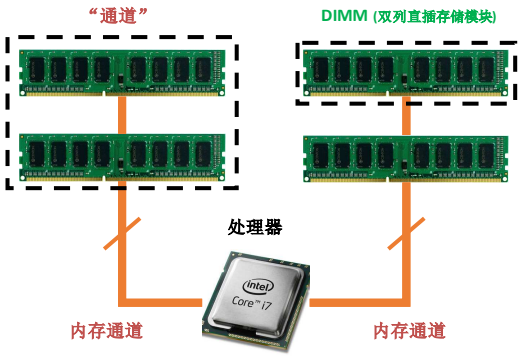
- 通道
- DIMM(双列直插式存储模块)
- Rank
- 芯片
- Bank
- 行/列



30

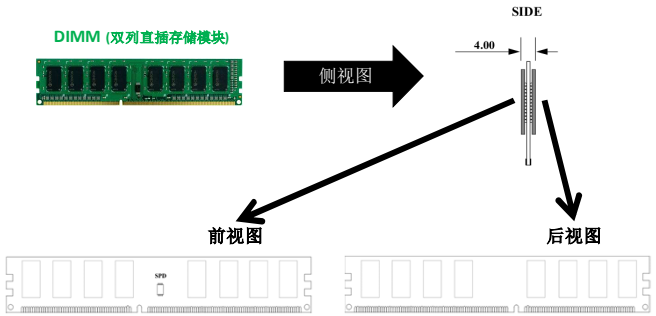
30

DRAM 子系统



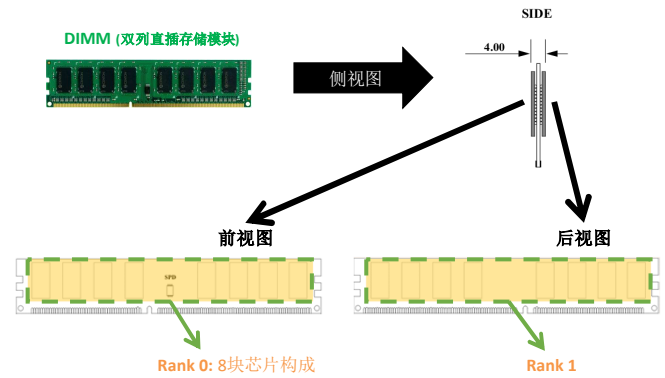
31

分解 DIMM



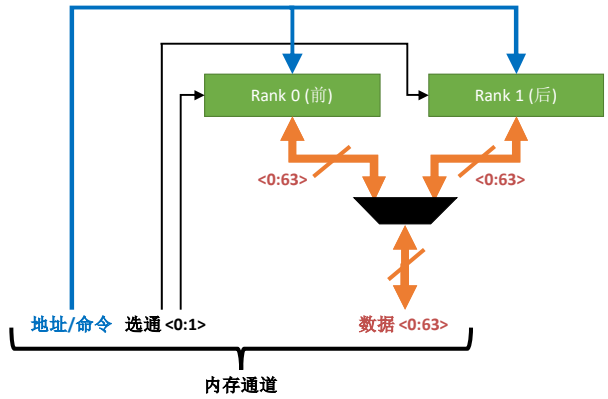
32

分解 DIMM



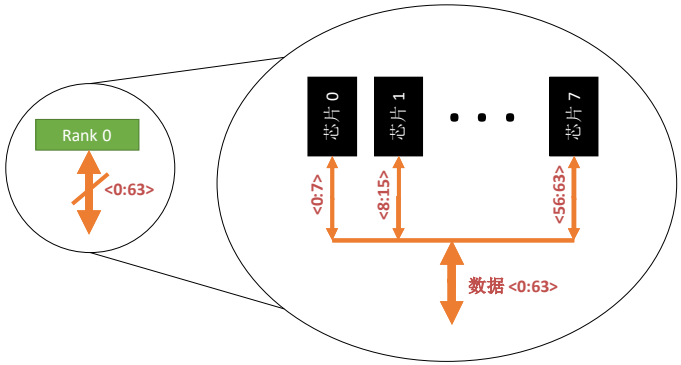
33

Rank



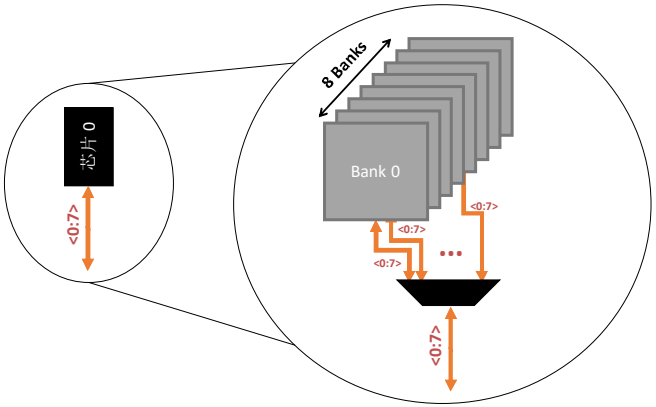
34

分解 Rank



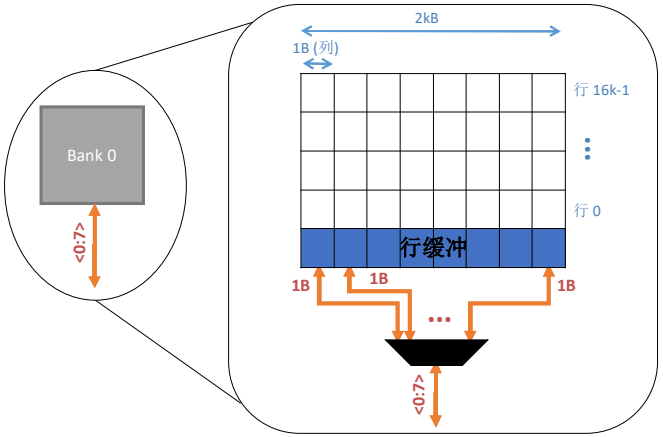
35

分解芯片



36

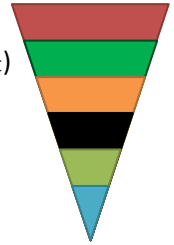
分解Bank



37

DRAM 子系统的组织

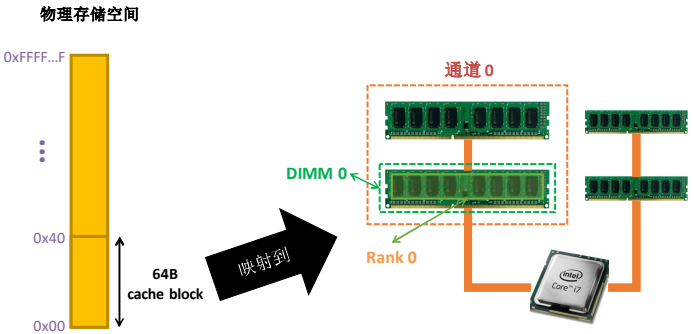
- 通道
- DIMM(双列直插式存储模块)
- Rank
- 芯片
- Bank
- 行/列



38

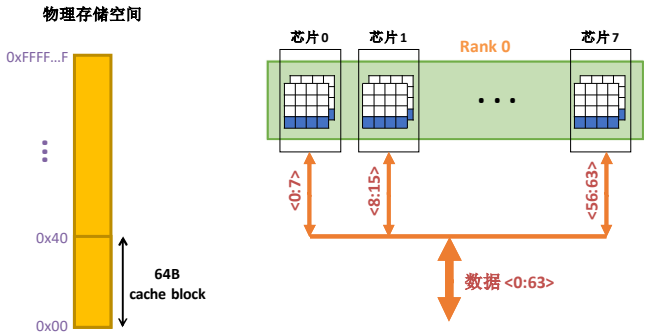
38

例子: 移动一个 cache block



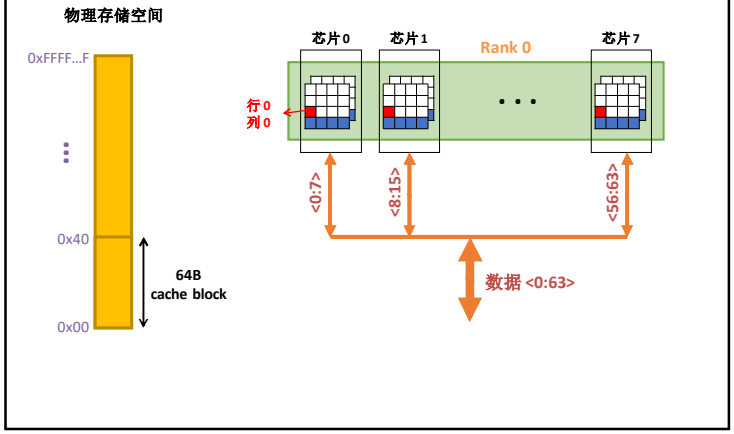
39

例子: 移动一个 cache block



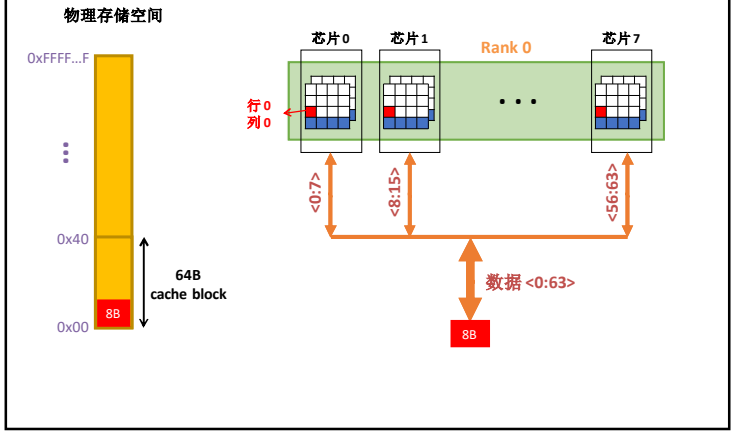
40

例子: 移动一个 cache block



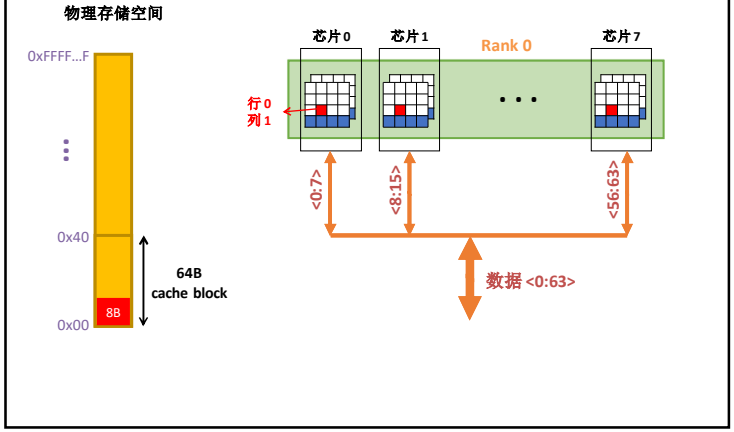
41

例子: 移动一个 cache block



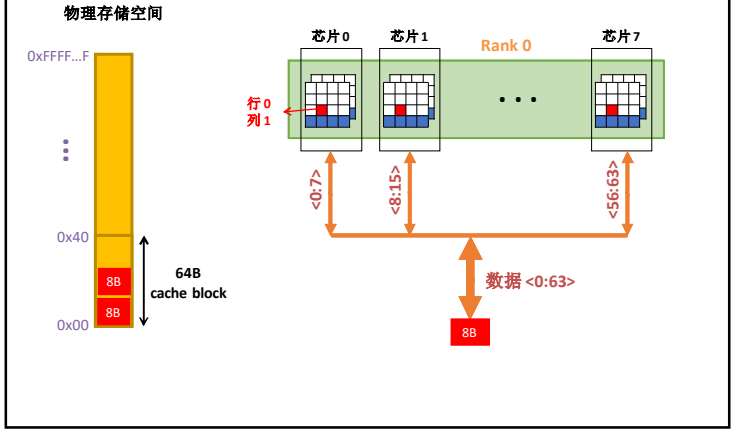
42

例子: 移动一个 cache block



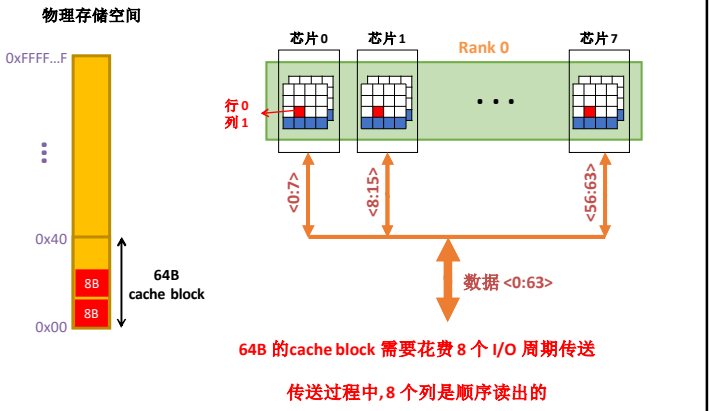
43

例子: 移动一个 cache block



44

例子: 移动一个 cache block



64B 的cache block 需要花费 8 个 I/O 周期传送

传送过程中, 8 个列是顺序读出的

45

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)

46

46

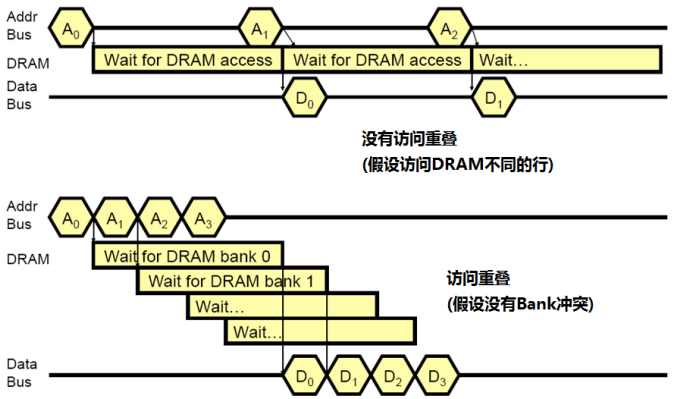
多Bank(交叉存取) 和多通道

- 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)

47

47

多Bank/通道的好处



48

48

多通道

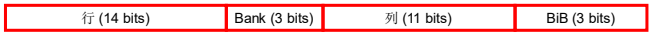
- 优点
  - 增加带宽
  - 并发访问 (如果通道独立的话)
- 缺点
  - 比单通道的成本高
    - 板上的线更多
    - 更多的管脚 (如果是片上内存控制器)

49

49

地址映射(单通道)

- 8字节内存总线的单通道系统
  - 2GB 内存, 8个 Bank, 每个 Bank 有 16K 行 x 2K 列
- 行交叉存取
  - 内存中连续的行在连续的Bank中



- Cache block交叉存取
  - 连续的cache block地址在连续的Bank中
  - 64字节的cache blocks

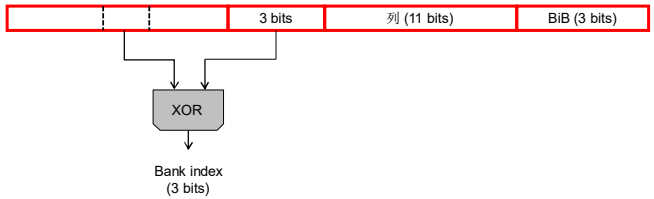
  - 访问连续的 cache block 可以并行
  - 随机访问? 跷步访问?

50

50

Bank 随机映射

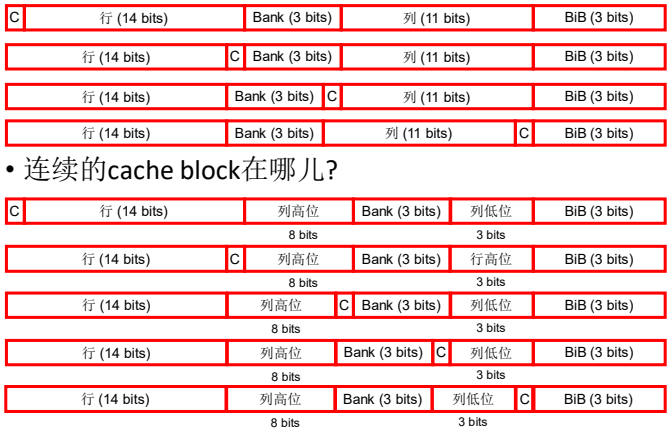
- DRAM控制器可以随机映射地址到Bank，这样就不太可能出现Bank冲突了



51

51

地址映射(多通道)



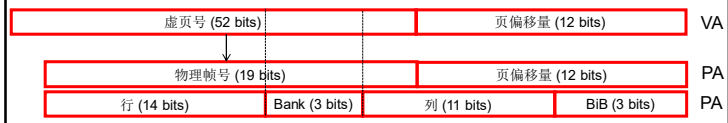
- 连续的cache block在哪儿?

52

52

虚拟→物理映射

- 操作系统会影响地址映射到 DRAM 中的什么位置



- 操作系统能够控制一个虚页会映射到哪个Bank/通道/Rank

- 用页着色的方法最小化Bank冲突
- 或者最小化应用间的干扰

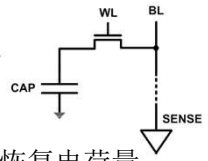
53

53

DRAM 刷新

54

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/预测性的影响: 刷新时暂停时间长
  - -- 刷新率限制了DRAM容量的扩展能力

55

55

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

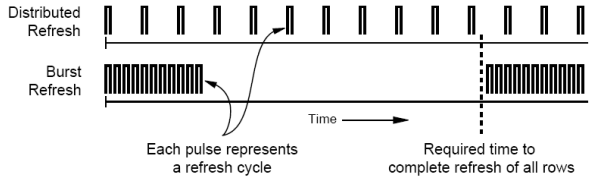
- 集中式刷新: 所有行在前一行刷新完成后立即刷新

- 分布式刷新: 每一行按照固定的间隔在不同时间刷新

56

56

分布式刷新

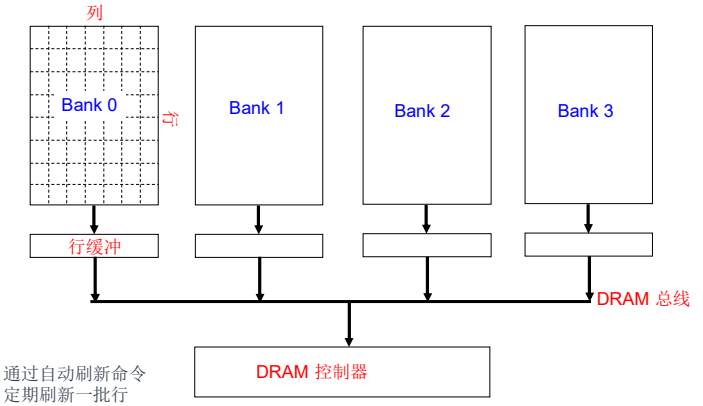


- 分布式刷新可以消除长的暂停时间
- 还能如何减少刷新对性能/QoS的影响?
- 分布式刷新能减少对能耗的影响吗?
- 是否能够减少刷新的数量?

57

57

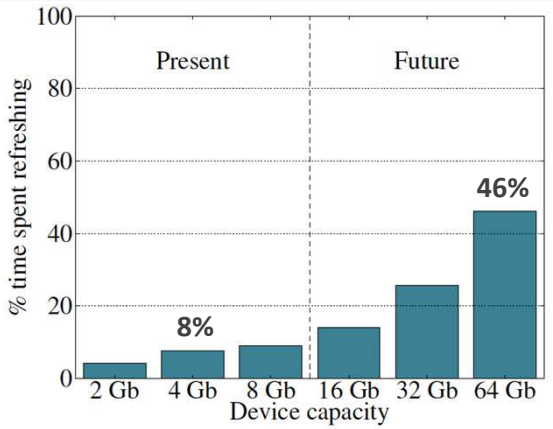
现在的刷新技术: 自动刷新



58

58

刷新的开销: 性能

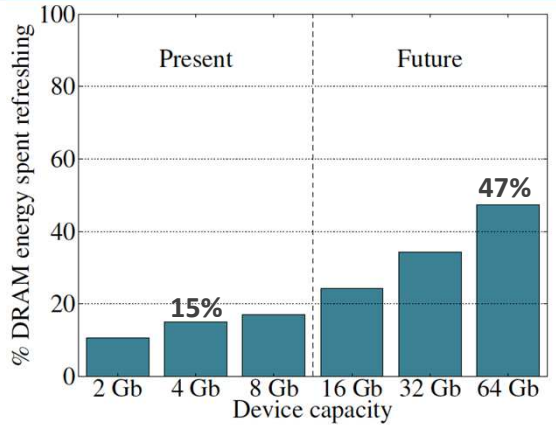


Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

59

59

刷新的开销: 能耗



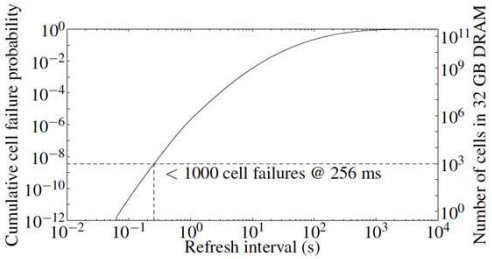
Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

60

60

传统刷新方法的问题

- 每一行以同样的频率刷新



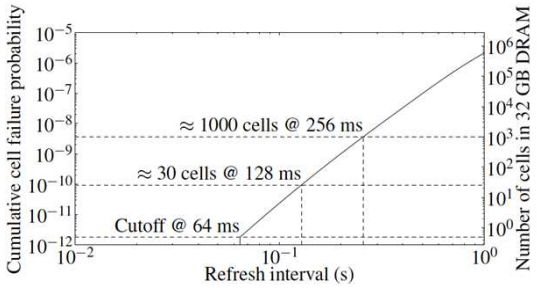
- 观察: 大多数行可以通过不那么频繁的刷新就能保证不丢失数据 [Kim+, EDL'09]
- 问题: DRAM中并不支持为每一行设置不同的刷新率

61

61

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?

62

62

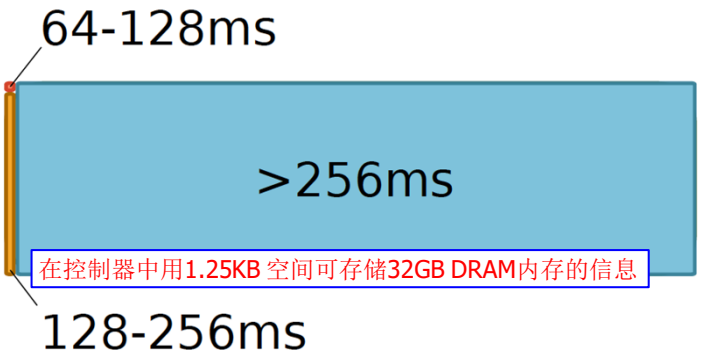
减少刷新操作

- 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.

63

63

RAIDR: 机制



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

64

64

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

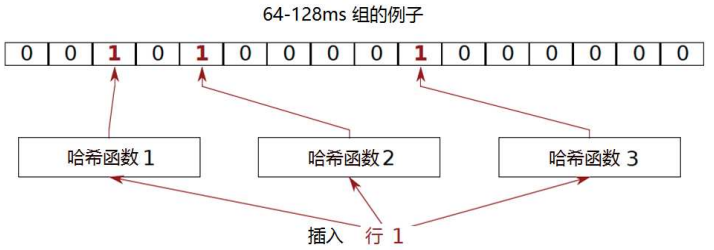
	行1	行2	行3
初始值	11111111...	11111111...	11111111...
64ms之后	11111111...	11111111...	11111111...
128ms之后	11011111... (64–128ms)	11111111...	11111111...
256ms之后		11111011... (128–256ms)	11111111... (>256ms)

65

65

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.

66

66

Bloom Filter

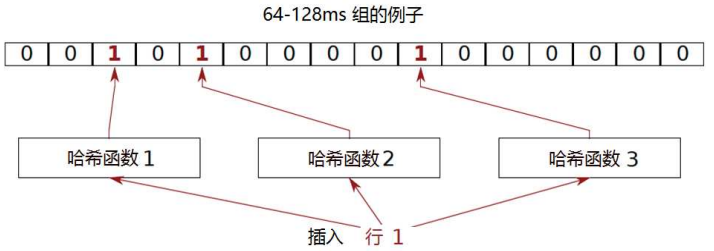
- [Bloom, CACM 1970]
- 一种能够简洁描述集合成员关系(元素在/不在集合内)的概率数据结构
- 非近似集合成员: 每个元素用1bit表示该元素存在/不在一个由N个元素组成的元素空间中
- 近似集合成员: 使用比元素个数少得多的bit的子集表示每个元素的成员关系(存在/不在)
  - 一些元素映射到的bit也会被其他元素映射到
- 操作: 1) 插入, 2) 测试, 3) 移除所有元素

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

67

67

Bloom Filter 操作示例

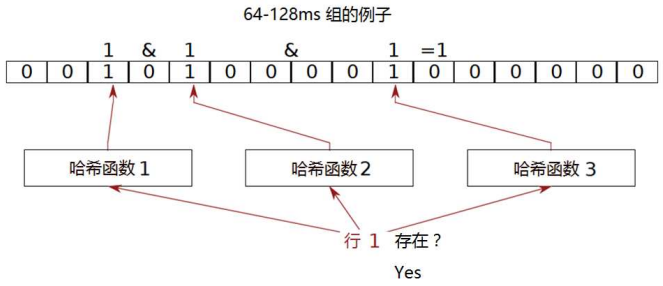


Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", CACM 1970.

68

68

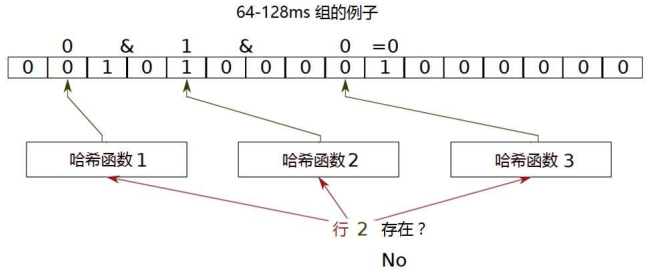
Bloom Filter 操作示例



69

69

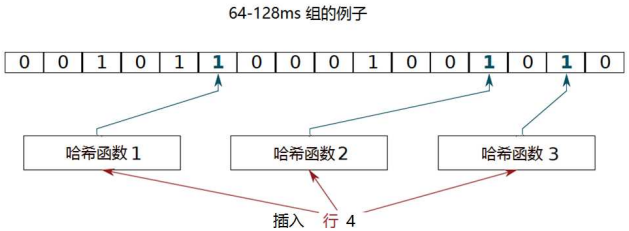
Bloom Filter 操作示例



70

70

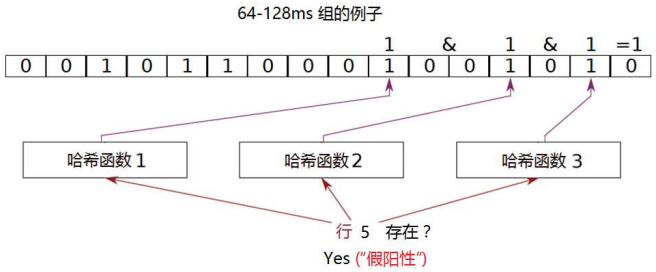
Bloom Filter 操作示例



71

71

Bloom Filter 操作示例



72

72

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

73

73

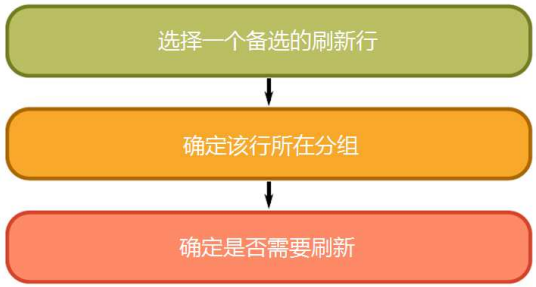
在硬件中使用Bloom Filter

- 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.

74

74

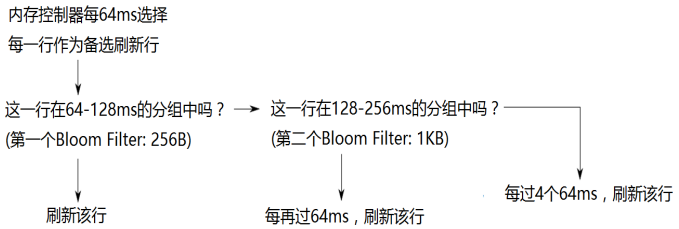
刷新(RAIDR 刷新控制器)



75

75

刷新(RAIDR 刷新控制器)

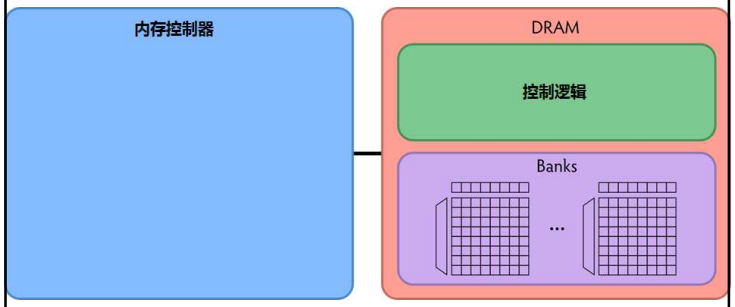


Liu et al., “RAIDR: Retention-Aware Intelligent DRAM Refresh,” ISCA 2012.

76

76

RAIDR: 基准设计



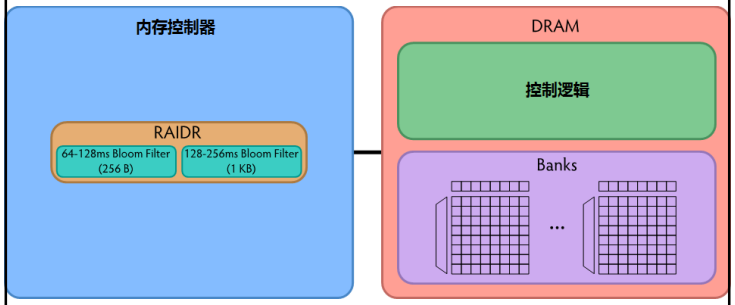
今天的自动刷新系统中，在DRAM里有刷新控制

RAIDR 可以在内存控制器中也可以在DRAM中实现

77

77

RAIDR 在内存控制器中: 选择 1

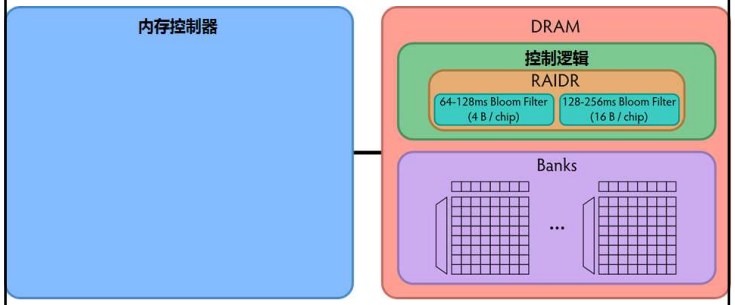


RAIDR在内存控制器中的开销:
1.25 KB Bloom Filter, 3个计数器, 为每行刷新而额外发射的命令
(都是评估中要考虑的因素)

78

78

RAIDR 在 DRAM 芯片中: 选择 2



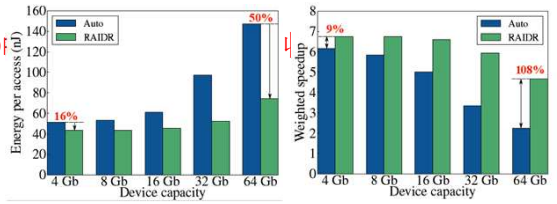
RAIDR 在DRAM芯片中的开销:
每芯片开销: 20B Bloom Filter, 1个计数器 (4 Gbit 芯片)
总开销: 1.25KB Bloom Filter, 64个计数器 (32 GB DRAM)

79

79

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D



80

80

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.

81

内存控制器

82

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

83

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

84

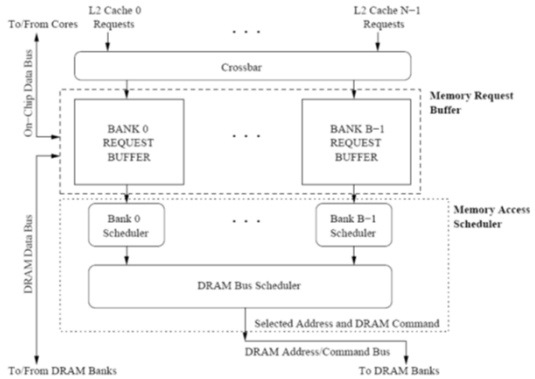
DRAM 控制器: 在什么地方？

- 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

85

85

现代DRAM控制器



86

86

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

87

87

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?

88

88

行缓冲管理策略

- 打开行
  - 一次访问之后保持该行打开
  + 下一次访问可能是同一行 → 行命中
  --下一次访问可能是不同的行 → 行冲突, 消耗能量
- 关闭行
  - 一次访问后关闭该行 (如果在请求缓冲中没有其它请求访问同一行)
  + 下一次访问可能是不同的行 → 避免一次行冲突
  -- 下一次访问可能是同一行 → 额外的激活延迟
- 自适应策略
  - 预测下一次对Bank的访问是否是同一行

打开 vs. 关闭行策略

策略	第一次访问	下一次访问	下一次访问需要的命令
打开行	行 0	行 0 (行命中)	读
打开行	行 0	行 1 (行冲突)	预充电 + 激活行 1 + 读
关闭行	行 0	行 0 – 在请求缓冲中 (行命中)	读
关闭行	行 0	行 0 – 不在请求缓冲中 (行关闭)	激活行 0 + 读 + 预充电
关闭行	行 0	行 1 (行关闭)	激活行1 + 读 + 预充电

为什么DRAM控制器很难设计?

- 需要遵从DRAM时序约束以保证正确性
  - DRAM中有很多时序约束 (50+)
  - tWTR: 写命令发射后发射读命令需要等待的最小周期数
  - tRC: 对于同一个Bank连续发射两条激活命令之间需要等待的最小周期数
  - ...
- 需要持续监视各种资源以防止冲突
  - 通道, Bank, Rank, 数据总线, 地址总线, 行缓冲
- 需要处理DRAM刷新
- 需要优化性能 (多约束条件下)
  - 重排序并不简单
  - 预测未来?

DRAM 时序约束

Latency	Symbol	DRAM cycles	Latency	Symbol	DRAM cycles
Precharge	^tRP	11	Activate to read/write	^tRCD	11
Read column address strobe	CL	11	Write column address strobe	CWL	8
Additive	AL	0	Activate to activate	^tRC	39
Activate to precharge	^tRAS	28	Read to precharge	^tRTP	6
Burst length	^tBL	4	Column address strobe to column address strobe	^tCCD	4
Activate to activate (different bank)	^tRRD	6	Four activate windows	^tFAW	24
Write to read	^tWTR	6	Write recovery	^tWR	12

Table 4. DDR3 1600 DRAM timing specifications

- From Lee et al., "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," HPS Technical Report, April 2010.

更多关于DRAM操作

- Kim et al., "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," ISCA 2012.
- Lee et al., "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," HPCA 2013.

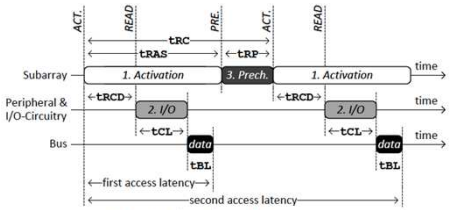


Figure 5. Three Phases of DRAM Access

Table 2. Timing Constraints (DDR3-1066) [43]

Phase	Commands	Name	Value
1	ACT → READ ACT → WRITE	tRCD	15ns
	ACT → PRE	tRAS	37.5ns
2	READ → *data* WRITE → *data*	tCL tCWL	15ns 11.25ns
	data burst	tBL	7.5ns
3	PRE → ACT	tRP	15ns
1 & 3	ACT → ACT	tRC (tRAS+tRP)	52.5ns

93

93

DRAM 电源管理

- DRAM芯片有多种电源管理模式
- 思路: 当某个芯片没有访问时进入节电模式

- 功率状态
  - 活跃 (最高功率)
  - 所有Bank空闲
  - 节电模式
  - 自刷新 (最低功率)

- 状态转换会产生延迟，在该延迟内芯片无法访问

94

94

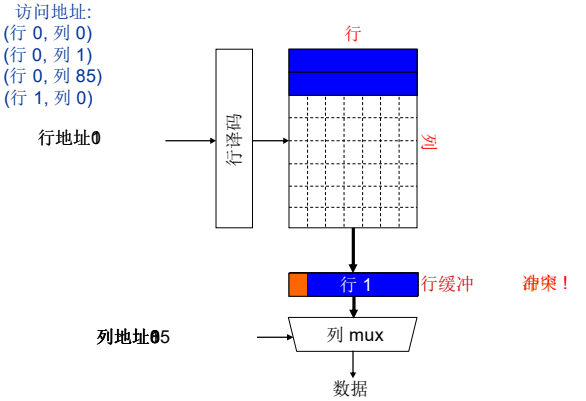
多核系统中的内存干扰和调度

95

95

DRAM Bank 操作

访问地址:
(行 0, 列 0)
(行 0, 列 1)
(行 0, 列 85)
(行 1, 列 0)



96

96

单核系统的调度策略

- 行不命中时的内存访问时间远远大于行命中时的访问时间
- 现在的控制器得益于行缓冲

- FR-FCFS (行缓冲优先)
  1. 行命中的优先
  2. 最旧的优先

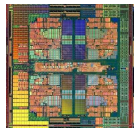
  目的1: 最大化行缓冲命中率 → 最大化DRAM吞吐量

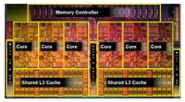
  目的2: 优先考虑旧的请求 → 确保向前推进

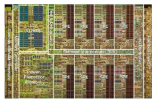
- 在多核系统中这还是个好的策略吗?

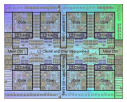
趋势: 片上众核(Many Core)

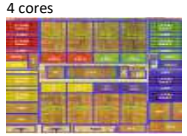
- 比一个大核更简单，功率更低
- 片上大规模并行

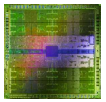

AMD Barcelona
4 cores


Intel Core i7
8 cores

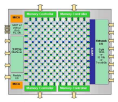

IBM Cell BE
8+1 cores


IBM POWER7
8 cores


Sun Niagara II
8 cores


Nvidia Fermi
448 "cores"

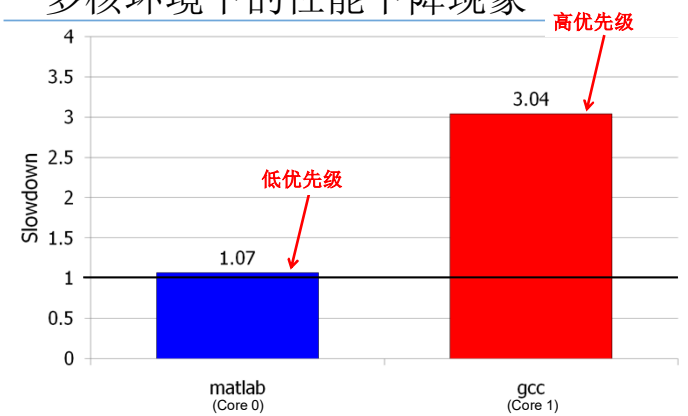

Intel SCC
48 cores, networked


Tilera TILE Gx
100 cores, networked

片上众核

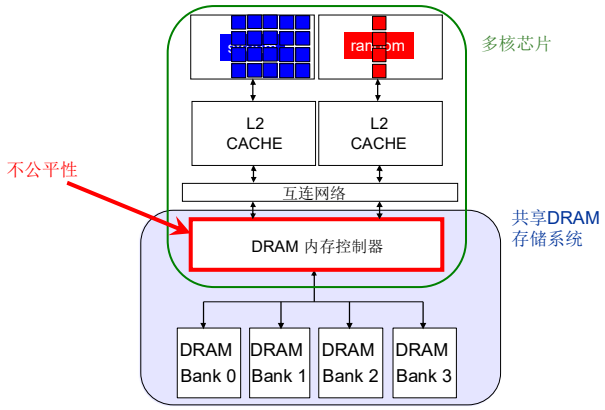
- 我们想要:
  - 用N倍的核获得N倍的系统性能
- 我们得到了什么?

多核环境下的性能下降现象



Moscibroda and Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," USENIX Security 2007.

干扰不受控: 例子



Moscibroda and Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," USENIX Security 2007.

101

101

```
// initialize large arrays A, B

for (j=0; j<N; j++) {
   index = j*linesize;   streaming
   A[index] = B[index];
   …
}
```

```
// initialize large arrays A, B

for (j=0; j<N; j++) {
   index = rand();   random
   A[index] = B[index];
   …
}
```

流

- 顺序的访存
- 极高的行缓冲局部性 (96% 命中率)
- 内存密集型

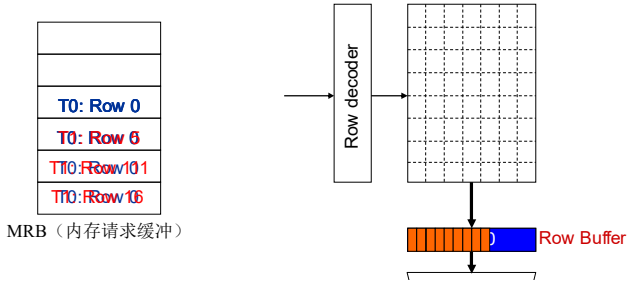
随机

- 随机访存
- 极低的行缓冲局部性(3% 命中率)
- 同样是内存密集型

Moscibroda and Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," USENIX Security 2007.

102

102



假如 Row: 8KB, cache line: 64B

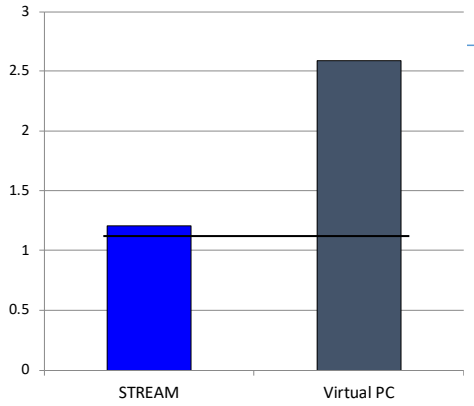
T0可能有128次请求优先于T1的请求被服务

Moscibroda and Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," USENIX Security 2007.

103

103

内



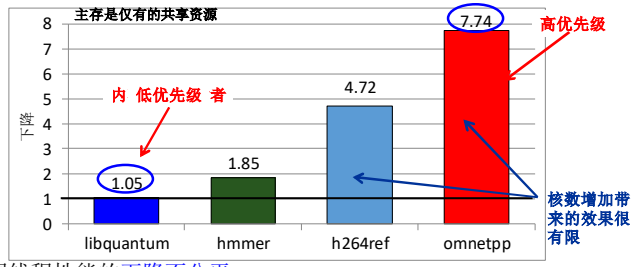
Intel Per
(Intel Co

Moscibroda and Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," USENIX Security 2007.

104

104

干扰不受控导致的问题

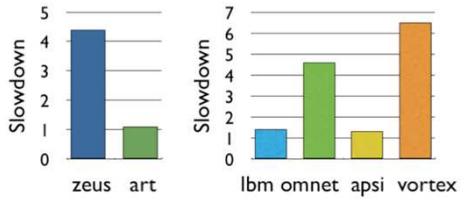


- 不同线程性能的下降不公平
- 系统性能低
- 拒绝服务的漏洞
- 优先级反转: 无法保证按优先级/SLA执行

105

105

干扰不受控导致的问题



- 不同线程性能的下降不公平
- 系统性能低
- 拒绝服务的漏洞
- 优先级反转: 无法保证按优先级/SLA执行
- 糟糕的性能可预测性 (无性能隔离)

无法控制、不可预料的系统

106

106

内存中的线程间干扰

- 内存控制器、管脚、内存Bank是共享的
- 管脚带宽的增加不像核数增加的那样快
  - 每核的带宽在减小
- 不同核上执行的不同线程在主存系统中会互相干扰
- 线程间由于资源竞争互相延迟:
  - Bank, 总线, 行缓冲 冲突 → 减小了 DRAM 吞吐量
- 线程还会破坏彼此的DRAM Bank访问的并行性

107

107

DRAM中线程间干扰的影响

- 排队/争用导致延迟
  - Bank 冲突, 总线冲突, 通道冲突, …
- 由于DRAM的约束导致额外的延迟
  - 称为 “协议开销”
  - 比如
    - 行冲突
    - 读-写和写-读延迟
- 线程内并行的丧失
  - 一个线程的并发请求被串行处理

108

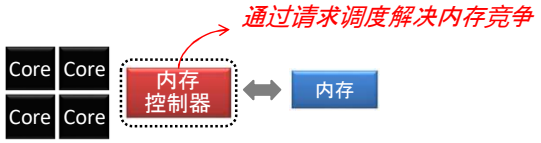
108

问题: 无法感知QoS的内存控制

- 现有的DRAM控制器无法感知DRAM系统中线程间的干扰
- 目标只是最大化DRAM的吞吐量
  - 不能感知线程，也存在线程间的不公平
  - 无法并行地响应线程的请求
  - FR-FCFS 策略: 1) 行命中优先, 2) 最旧优先
    - 行缓冲的高局部性导致线程优先级的不公平
    - 受益的线程往往是内存密集型的 (大量的访存)

109

解决方案: QoS感知的内存请求调度



- 如何通过请求调度获得
  - 高系统性能
  - 应用的高公平性
  - 系统软件的可配置性
- 内存控制器需要能够感知线程

110

一些内存调度方法

- O. Mutlu et.al., "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors" , 40th International Symposium on Microarchitecture (MICRO), pages 146-158, Chicago, IL, December 2007
- O. Mutlu et.al., "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems", 35th International Symposium on Computer Architecture (ISCA), pages 63-74, Beijing, China, June 2008
- Y. Kim et.al., "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior", 43rd International Symposium on Microarchitecture (MICRO), pages 65-76, Atlanta, GA, December 2010
- Y. Kim et.al., "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers", 16th International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, India, January 2010

111

其它处理干扰的方法

112

基本的干扰控制技术

- 目标: 减少/控制干扰

1. 优先级或请求调度

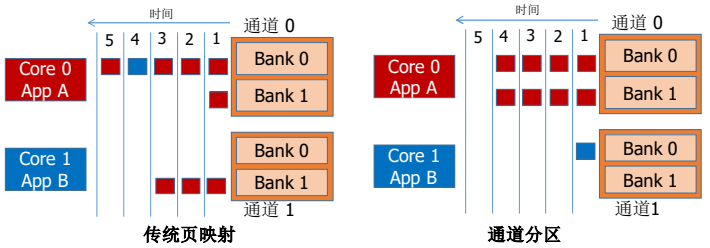
2. 数据映射到Bank/通道/Rank

3. 核/源调节

4. 应用/线程调度

内存通道分区

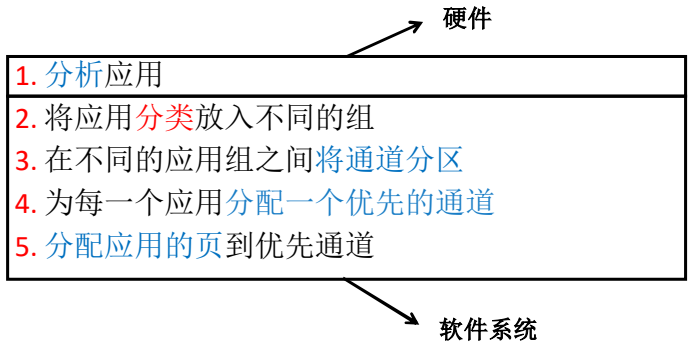
- 内存通道分区
  - 思路: 把干扰严重的应用的页映射到不同的通道[Muralidhara+, MICRO'11]



- 分离具有不同行局部性和内存密集程度的应用
- 对于减少具有“中等”和“重度”内存密集的线程的干扰尤其有效

Muralidhara et al., “Memory Channel Partitioning,” MICRO'11.

内存通道分区(MCP)机制



观察

- 内存密集程度非常低的应用很少访存 → 为它们分配通道会导致宝贵的内存带宽的浪费
- 它们非常可能使它们所在的核持续繁忙 → 真的应该给它们高的优先级
- 它们极少与其它应用发生干扰 → 给它们高优先级不会损害其它应用

集成的内存分区和调度(IMPS)

- 在内存调度时总是给内存密集程度很低的应用以高优先级
- 使用内存通道分区减少其它应用之间的干扰

Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

117

基本的干扰控制技术

- 目标: 减少/控制干扰

1. 优先级或请求调度
2. 数据映射到Bank/通道/Rank
3. 核/源调节
4. 应用/线程调度

118

另外一种方法: 源调节

- 在核(源)上管理线程间干扰, 而不是在共享资源上
- 在内存系统中动态估计公平性
- 将该信息反馈给控制器
- 相应地调节核的访存频率
  - 调节谁调节多少取决于性能目标(吞吐量, 公平性, 每个线程的QoS, 等等)
  - 比如, 如果不公平性 > 系统软件指定的目标则调低导致不公平的核的访存频率 & 调高被不公平对待的核的访存频率

- Ebrahimi et al., "Fairness via Source Throttling," ASPLOS'10, TOCS'12.

119

核 (源) 调节

- 思路: 估计由干扰造成的性能下降，调低"肇事"线程的访存量
  - Ebrahimi et al., "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," ASPLOS 2010.
- 优点
  + 核/请求的调节容易实现: 不需要改变内存调度算法
  + 是一种处理共享资源竞争的通用方法
- 缺点
  - 需要估计干扰/性能下降
  - 阈值优化会比较困难 → 吞吐量下降

120

基本的干扰控制技术

- 目标: 减少/控制干扰

1. 优先级或请求调度

2. 数据映射到Bank/通道/Rank

3. 核/源调节

4. 应用/线程调度

思路: 选择互相干扰不严重的线程一起调度到核上
共享内存系统

121

121

在并行应用中处理干扰

- 多线程应用中的线程是相互依赖的
- 由于同步的原因某些线程会在关键路径上执行，有些线程不会
- 如何调度相互依赖的线程的请求，才能最大化多线程应用的性能？

- 思路: 估计可能在关键路径上的线程，优先处理它们的请求; 调整非关键线程的优先级以减小它们之间的干扰 [Ebrahimi+, MICRO'11]
- 硬件/软件协同的关键线程估计

122

122

新型的非易失性存储技术

123

123

非易失存储器

- 如果存储器是非易失的...
  - 不需要刷新...
  - 不会在掉电时丢失数据...
- 问题: 非易失存储器件一直以来都比DRAM慢很多
  - 比如硬盘... 甚至闪存...
- 机遇: 一些新兴的存储技术，非易失而且相对比较快
  - 同时，比DRAM可扩展性更好
- 提问: 是否可以采用这些新兴技术来实现主存储器？

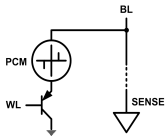
124

124

新兴的存储技术

- 一些新兴的电阻式存储技术似乎比DRAM具有更好的可扩展性 (并且它们是非易失的)

- 例如: 相变存储器
  - 通过材料的相变存储数据
  - 通过检测材料的阻抗读取数据
  - 预计尺寸可以达到9nm (2022 [ITRS])
  - 原型20nm (Raoux+, IBM JRD 2008)
  - 将比DRAM密度更高: 可存储多个bit/位元



- 当然, 新的技术会有一些缺陷
  - 它们能够代替DRAM吗?

125

电阻式存储器技术

- PCM(相变存储器)
  - 通过注入电流使材料发生相变
  - 相变决定阻抗的不同

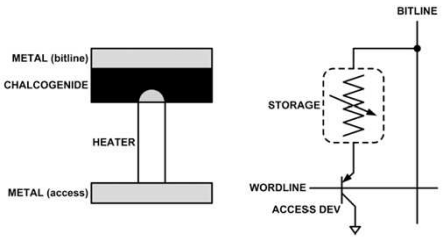
- STT-MRAM(自旋转矩磁随机存取存储器)
  - 通过注入电流改变磁极
  - 极性改变决定阻抗的不同

- Memristor(忆阻器)
  - 通过注入电流改变原子结构
  - 原子间的距离决定阻抗

126

什么是相变存储器?

- 相变材料(硫族化合物玻璃) 存在于两种状态中:
  - 非晶态: 低光反射率，高电阻率
  - 晶态: 高光反射率，低电阻率

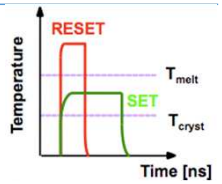


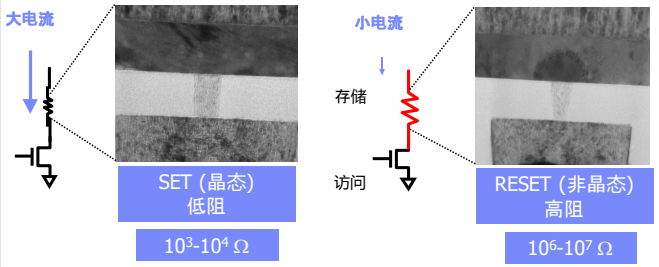
PCM是电阻式存储器: 高阻态 (0)，低阻态 (1)
PCM的位元可以在不同状态之间可靠、快速地切换

127

PCM 如何工作?

- 写: 通过电流注入改变物相
  - SET: 持续注入电流加热位元温度超过 T*cryst*
  - RESET: 位元加温超过T*melt* 并迅速冷却
- 读: 通过材料的阻抗判断物相
  - 非晶/晶态





128

相变存储器: 优点和缺点

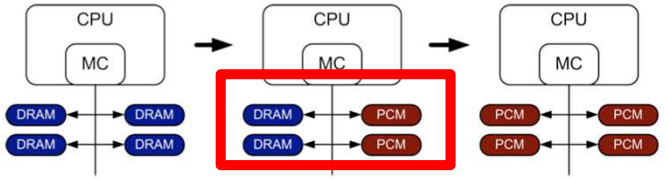
- 优于DRAM之处
  - 更好的工艺规模(容量和成本)
  - 非易失
  - 空闲时功率低 (无需刷新)
- 缺点
  - 延迟更高: ~4-15x DRAM (尤其是写入时)
  - 活跃状态能耗更高: ~2-50x DRAM (尤其是写入时)
  - 重复使用寿命较低 (位元寿命~10^8 次写入)
- 用PCM替换或者协助DRAM组成主存储器的挑战:
  - 减小PCM缺陷的影响
  - 找到合适的方式将PCM引入系统

129

129

基于PCM的主存储器 (I)

- 基于PCM的(主)存储器如何组织?



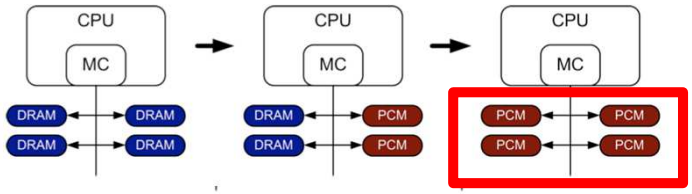
- 混合PCM+DRAM [Qureshi+ ISCA'09, Dhiman+ DAC'09]:
  - 在PCM和DRAM之间如何分区/迁移数据

130

130

基于PCM的主存储器(II)

- 基于PCM的(主)存储器如何组织?



- 纯 PCM的主存储器 [Lee et al., ISCA'09, Top Picks'10]:
  - 如何重新设计整个层次结构(包括核)以克服PCM的缺点

131

131

基于PCM的存储系统: 研究上的挑战

- 分区
  - 可以用DRAM做cache或者主存，或者灵活配置吗?
  - 各占多少比例? 需要多少控制器?
- 数据分配/移动 (能耗, 性能, 生命周期)
  - 谁来管理分配和移动?
  - 控制算法?
  - 如何预防因为材料老化导致的失效?
- Cache的层次设计, 内存控制器, OS
  - 消除PCM缺陷的影响, 利用PCM的优点
- PCM/DRAM芯片和模块设计
  - 重新考虑PCM/DRAM的新需求

132

132

初步的研究: 用PCM替换DRAM

- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.
  - 综述了2003-2008的原型系统 (比如 IEDM, VLSI, ISSCC)
  - 得出PCM"平均"F=90nm

Density
- 9 - 12F^2 using BJT
- 1.5× DRAM

Latency
- 50ns Rd, 150ns Wr
- 4×, 12× DRAM

Endurance
- 1E+08 writes
- 1E-08× DRAM

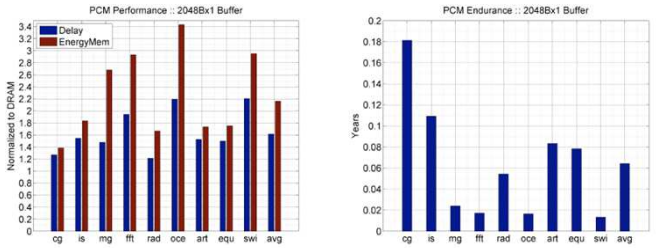
Energy
- 40μA Rd, 150μA Wr
- 2×, 43× DRAM

133

133

结果: 用PCM简单替换DRAM

- 在4核、4MB L2 cache的系统中用PCM替换DRAM
- PCM的组织与DRAM相同: 行缓冲, Bank…
- 1.6x延迟, 2.2x能耗, 500小时平均寿命



- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.

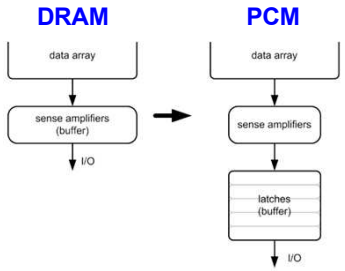
134

134

设计PCM架构以减小缺陷的影响

- 思路1: 在每个PCM芯片中使用多个窄行缓冲
  → 减少阵列的读/写 → 更好的耐久性、延迟和能耗表现

- 思路2: 写入阵列时按cache line或字的粒度写
  → 减少不必要的损耗

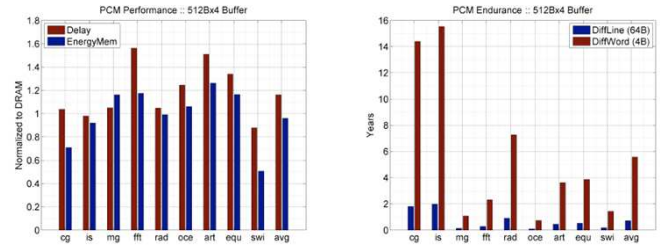


135

135

结果: 用PCM构建主存储器

- 1.2x延迟, 1.0x能耗, 5.6-year平均寿命
- 调整规模可以改善能耗、耐久性和密度等指标

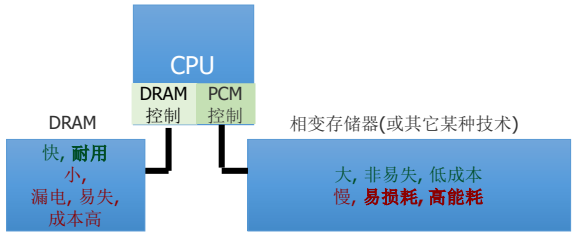


- 注1: 最坏情况下寿命很短 (无保障)
- 注2: 密集型应用的性能和能耗都会受到很大的影响
- 注3: 如何优化PCM的参数?

136

136

混合存储系统



硬件/软件管理数据分配和移动以获得多种技术的优势

Meza+, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.

Yoon, Meza et al., "Row Buffer Locality Aware Caching Policies for Hybrid Memories," ICCD 2012 Best Paper Award.

137

一种选择: DRAM作为PCM的cache

- PCM作主存; DRAM缓存主存的行/块
  - 好处: DRAM缓存命中时减小延迟; 写过滤
- 内存控制器硬件管理DRAM cache
  - 好处: 没有系统软件的开销

- 三个需要解决的问题:
  - 哪些数据应该放到DRAM中?
  - 数据移动的粒度该如何选择?
  - 如何设计低成本的硬件管理的DRAM cache?

- 两个思路:
  - 感知局部性的数据放置[Yoon+, ICCD 2012]
  - 低开销标签存储和动态粒度 [Meza+, IEEE CAL 2012]

138

138