高等计算机体系结构
第十一讲: 主存储器(II)、虚拟存储

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
2020-05-29

1

提醒：作业

- 作业 5
  - 5月8日发布，6月5日上课前截止提交
  - Cache和Memory
- 作业 6
  - 6月5日发布，6月19日截止
  - 预取和并行

2

2

实验2-5

- 5月10日发布，预计7月10日截止

3

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

4

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

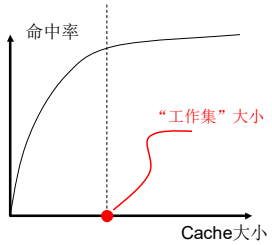
- C, 表示Cache的容量（多少byte）
  - 比如：16KByte(L1), 1MByte(L2)
- B = 2^b , Cache块的大小（多少byte）
  - 比如：16(L1), > 64(L2)
- a, Cache的相联度
  - 比如：1, 2, 4, 5(?), …… C/B

5

5

回顾：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



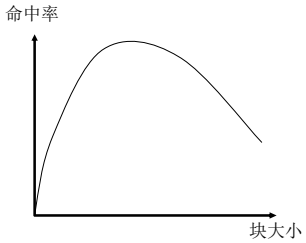
- 工作集: 执行应用时会引用的所有数据的集合
  - 在一个时间段之内

6

6

回顾：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



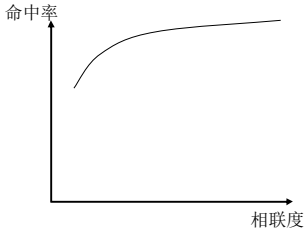
- 太小的块
  - 不能很好地利用空间局部性
  - 标签的开销更大
- 太大的块
  - 块的数量太少
    - 很可能导致无用的数据移动
    - 消耗额外的带宽/电能

7

7

回顾：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

8

8

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

9

9

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

10

10

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)

11

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

12

12

回顾：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

13

13

回顾：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

14

14

回顾：交叉存取

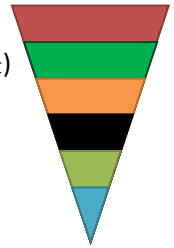
- 问题: 单片的存储阵列访问时间很长，并且无法并行执行多个访存
- 目标: 减小对存储阵列访问的延迟，并且能够并行执行多个访存
- 思路: 将存储阵列划分为多个可以(在同一个周期或连续的周期)独立访问的Bank
  - 每个 Bank 都比整个存储空间小
  - 可以重叠地访问不同的 Bank
- 需要解决的难题: 如何将数据映射到不同的 Bank? (如何在不同的Bank之间交叉存取数据?)

15

15

回顾：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

回顾：延迟组件

- 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)

17

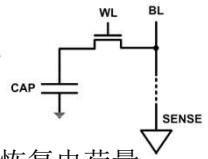
17

DRAM 刷新

18

18

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

19

19

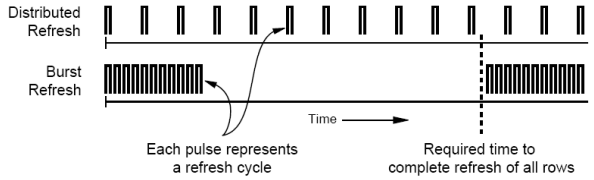
DRAM 刷新: 性能

- 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

20

20

分布式刷新

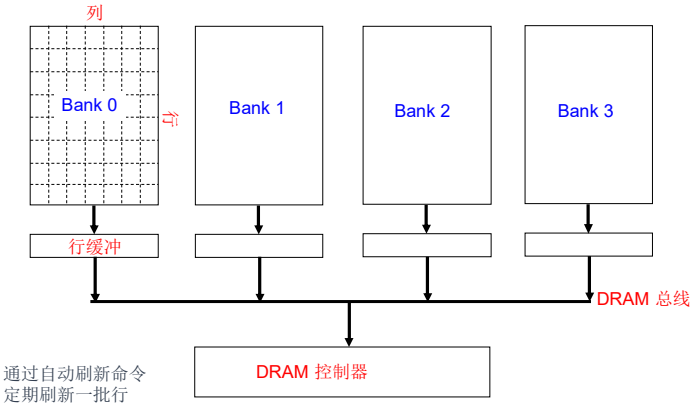


- 分布式刷新可以消除长的暂停时间
- 还能如何减少刷新对性能/QoS的影响?
- 分布式刷新能减少对能耗的影响吗?
- 是否能够减少刷新的数量?

21

21

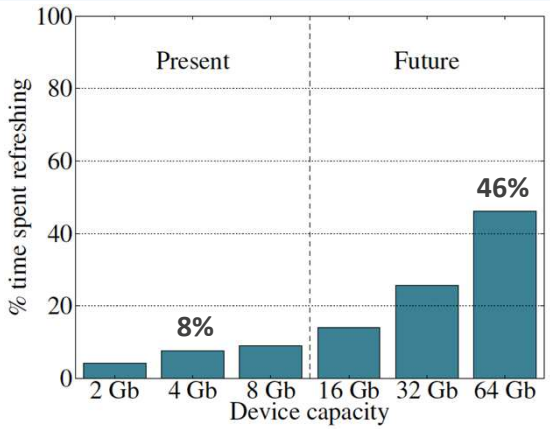
现在的刷新技术: 自动刷新



22

22

刷新的开销: 性能

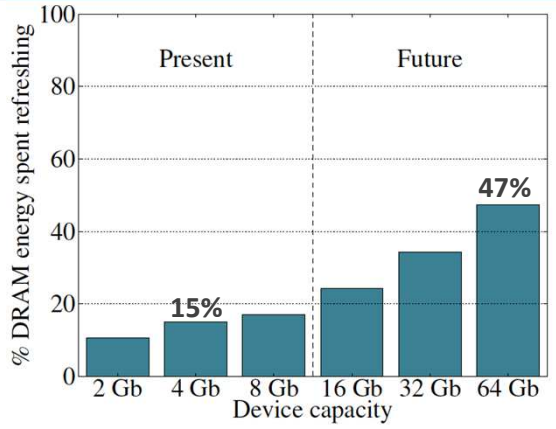


Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

23

23

刷新的开销: 能耗



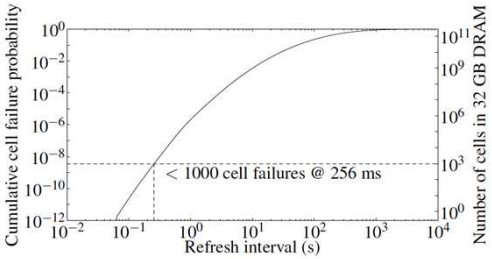
Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

24

24

传统刷新方法的问题

- 每一行以同样的频率刷新



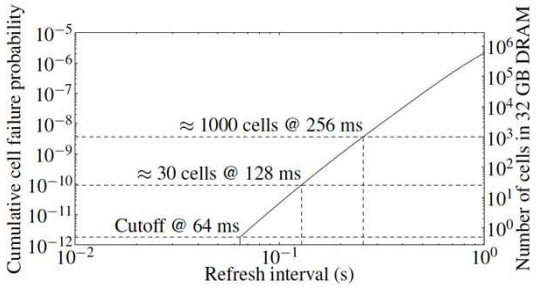
- 观察: 大多数行可以通过不那么频繁的刷新就能保证不丢失数据 [Kim+, EDL'09]
- 问题: DRAM中并不支持为每一行设置不同的刷新率

25

25

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?

26

26

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.

27

27

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

28

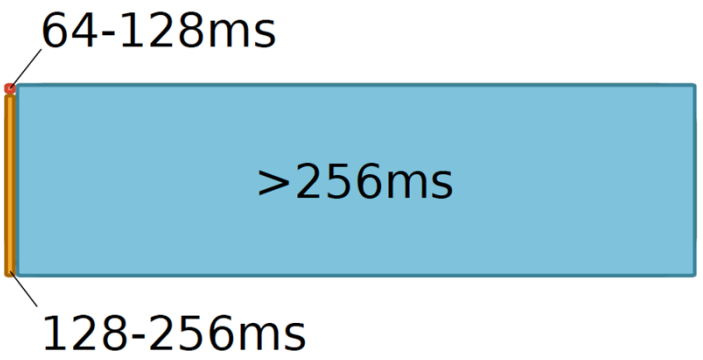
28

RAIDR: 机制

29

29

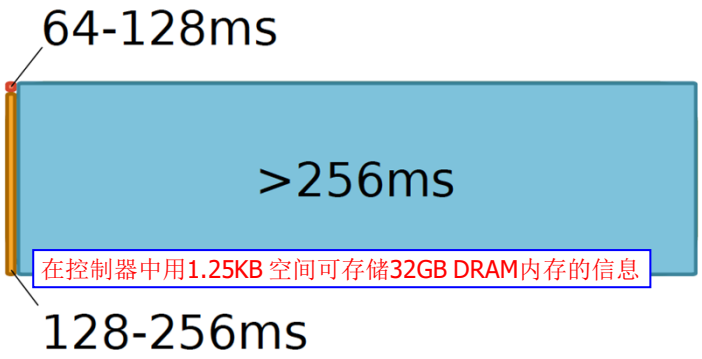
RAIDR: 机制



30

30

RAIDR: 机制



31

31

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

32

32

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

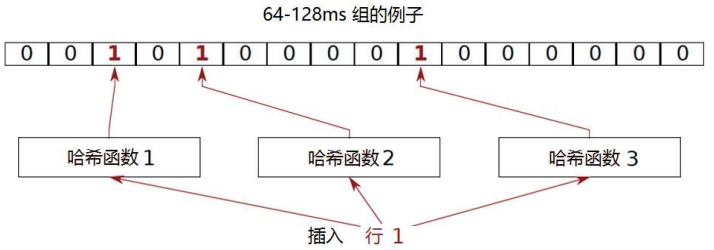
	行1	行2	行3
初始值	11111111...	11111111...	11111111...
64ms之后	11111111...	11111111...	11111111...
128ms之后	11011111... (64–128ms)	11111111...	11111111...
256ms之后		11111011... (128–256ms)	11111111... (>256ms)

33

33

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.

34

34

Bloom Filter

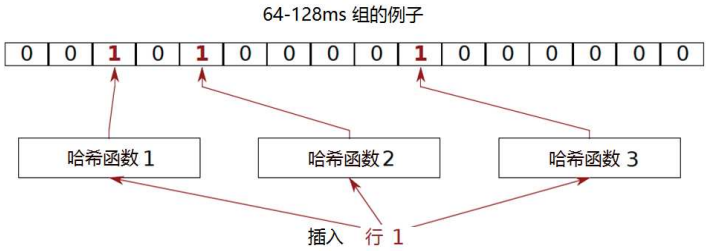
- [Bloom, CACM 1970]
- 一种能够简洁描述集合成员关系(元素在/不在集合内)的概率数据结构
- 非近似集合成员: 每个元素用1bit表示该元素存在/不在一个由N个元素组成的元素空间中
- 近似集合成员: 使用比元素个数少得多的bit的子集表示每个元素的成员关系(存在/不在)
  - 一些元素映射到的bit也会被其他元素映射到
- 操作: 1) 插入, 2) 测试, 3) 移除所有元素

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

35

35

Bloom Filter 操作示例

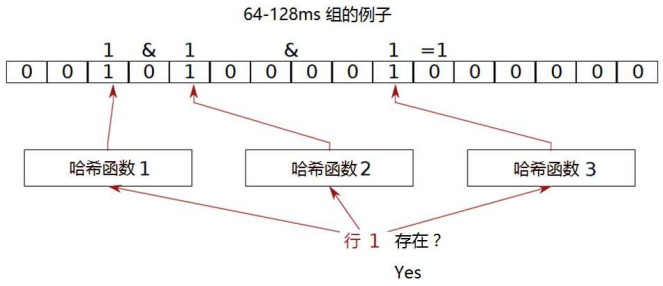


Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", CACM 1970.

36

36

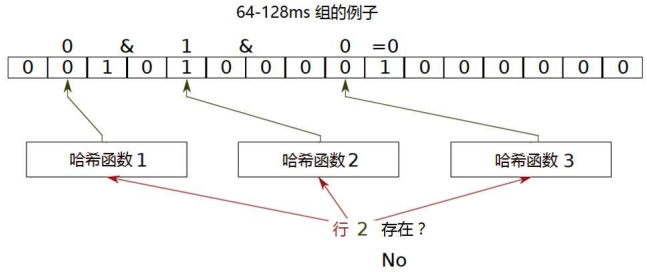
Bloom Filter 操作示例



37

37

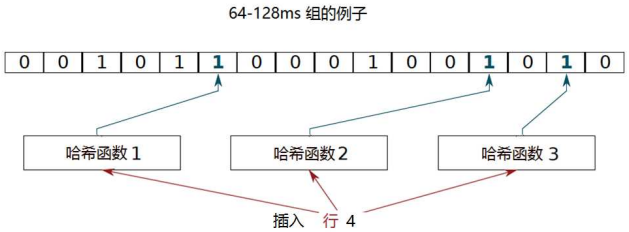
Bloom Filter 操作示例



38

38

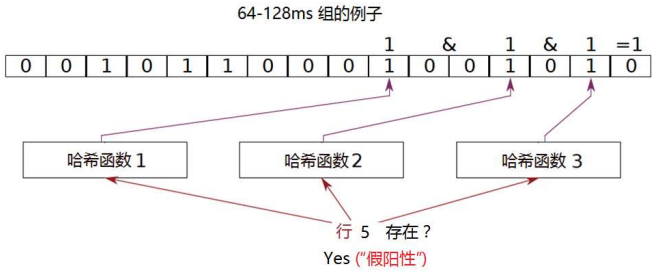
Bloom Filter 操作示例



39

39

Bloom Filter 操作示例



40

40

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

41

41

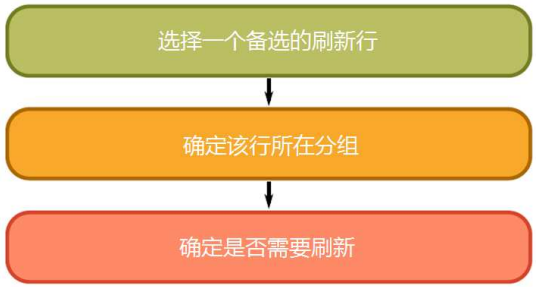
在硬件中使用Bloom Filter

- 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.

42

42

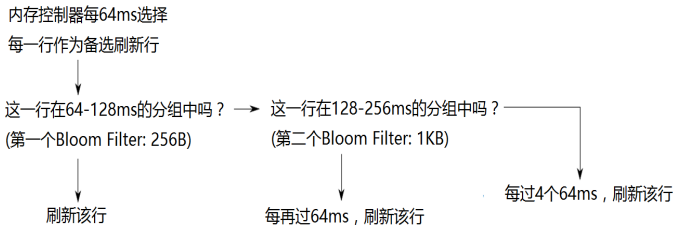
刷新(RAIDR 刷新控制器)



43

43

刷新(RAIDR 刷新控制器)

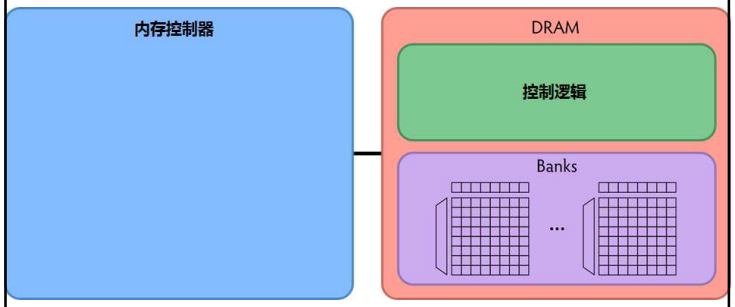


Liu et al., “RAIDR: Retention-Aware Intelligent DRAM Refresh,” ISCA 2012.

44

44

RAIDR: 基准设计



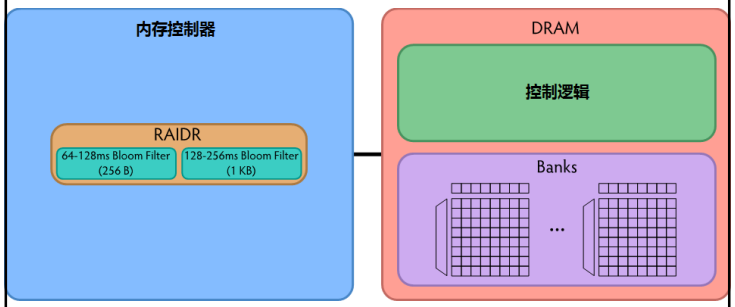
今天的自动刷新系统中，在DRAM里有刷新控制

RAIDR 可以在内存控制器中也可以在DRAM中实现

45

45

RAIDR 在内存控制器中: 选择 1

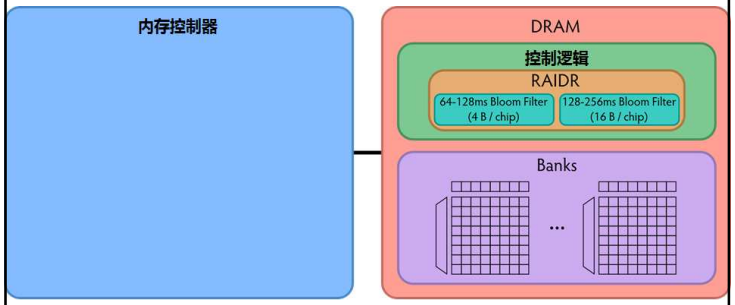


RAIDR在内存控制器中的开销:
1.25 KB Bloom Filter, 3个计数器, 为每行刷新而额外发射的命令
(都是评估中要考虑的因素)

46

46

RAIDR 在 DRAM 芯片中: 选择 2



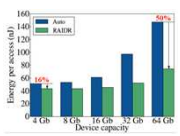
RAIDR 在DRAM芯片中的开销:
每芯片开销: 20B Bloom Filter, 1个计数器 (4 Gbit 芯片)
总开销: 1.25KB Bloom Filter, 64个计数器 (32 GB DRAM)

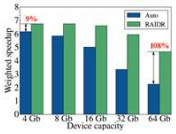
47

47

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





48

48

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.

内存控制器

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

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

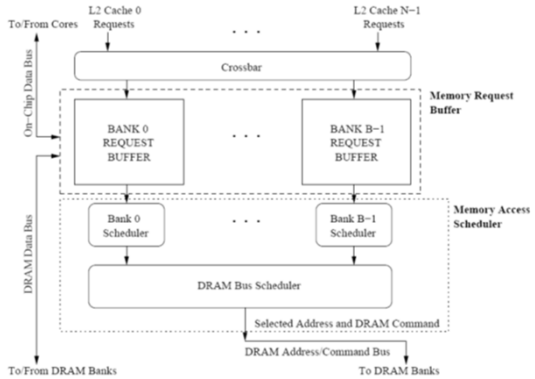
DRAM 控制器: 在什么地方？

- 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

53

53

现代DRAM控制器



54

54

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

55

55

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?

56

56

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

57

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

58

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?

59

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.

60

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.

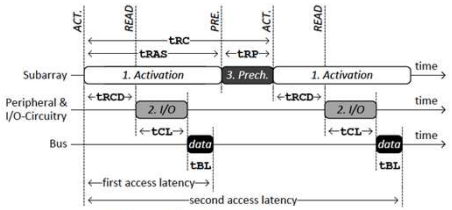


Figure 5. Three Phases of DRAM Access

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

61

61

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

62

62

多核系统中的内存干扰和调度

63

63

单核系统的调度策略

- 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

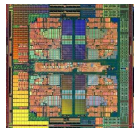
- 在多核系统中这还是个好的策略吗?

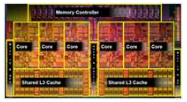
64

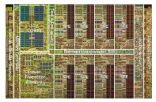
64

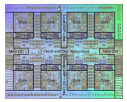
趋势: 片上众核(Many Core)

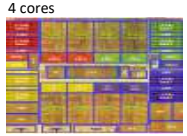
- 比一个大核更简单，功率更低
- 片上大规模并行

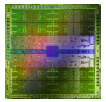

AMD Barcelona
4 cores


Intel Core i7
8 cores

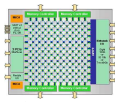

IBM Cell BE
8+1 cores


IBM POWER7
8 cores


Sun Niagara II
8 cores


Nvidia Fermi
448 "cores"

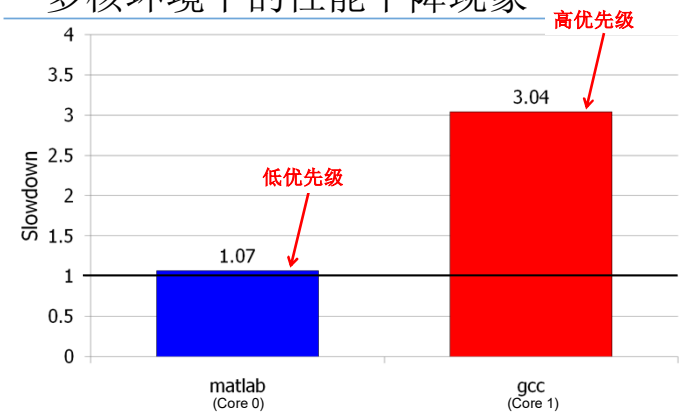

Intel SCC
48 cores, networked


Tilera TILE Gx
100 cores, networked

片上众核

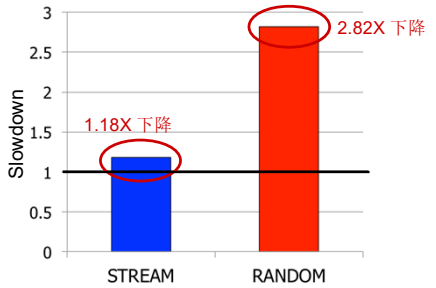
- 我们想要:
  - 用N倍的核获得N倍的系统性能
- 我们得到了什么?

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.

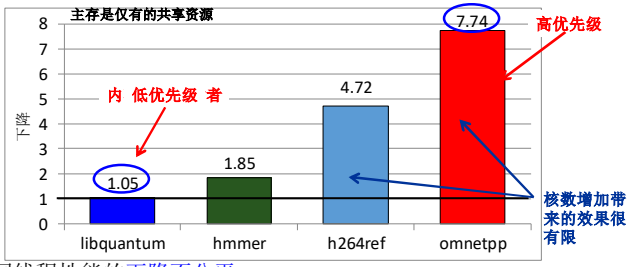
内存性能抢占现象的影响



Intel Pentium D,Windows XP
(Intel Core Duo, AMD Turion, Fedora Linux, 结果类似)

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

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

69

69

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

70

70

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

71

71

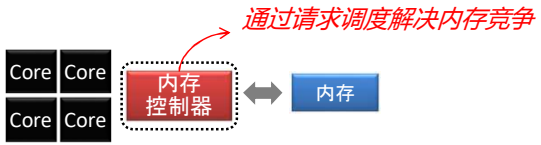
问题: 无法感知QoS的内存控制

- 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)

72

72

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

73

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

74

其它处理干扰的方法

75

基本的干扰控制技术

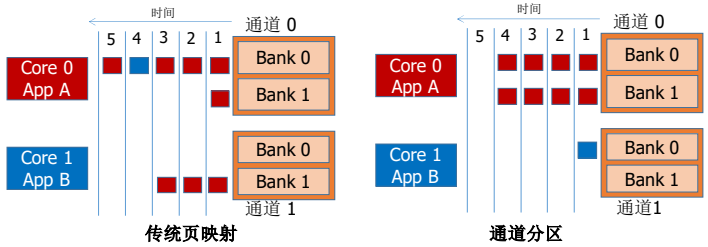
- 目标: 减少/控制干扰

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

76

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.

77

77

内存通道分区(MCP)机制

硬件

1. 分析应用
2. 将应用分类放入不同的组
3. 在不同的应用组之间将通道分区
4. 为每一个应用分配一个优先的通道
5. 分配应用的页到优先通道

软件系统

78

78

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

79

79

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.

80

80

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

81

81

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.

82

82

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

83

83

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

84

84

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

85

85

新型的非易失性存储技术

86

86

非易失存储器

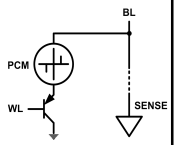
- 如果存储器是非易失的...
  - 不需要刷新...
  - 不会在掉电时丢失数据...
- 问题: 非易失存储器件一直以来都比DRAM慢很多
  - 比如硬盘... 甚至闪存...
- 机遇: 一些新兴的存储技术，非易失而且相对比较快
  - 同时，比DRAM可扩展性更好
- 提问: 是否可以采用这些新兴技术来实现主存储器？

87

87

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?

88

88

电阻式存储器技术

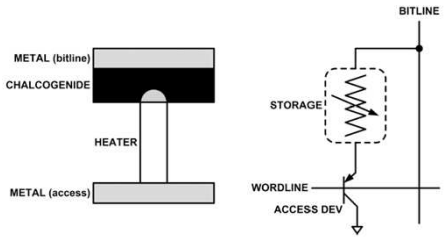
- PCM(相变存储器)
  - 通过注入电流使材料发生相变
  - 相变决定阻抗的不同
- STT-MRAM(自旋转矩磁随机存取存储器)
  - 通过注入电流改变磁极
  - 极性改变决定阻抗的不同
- Memristor(忆阻器)
  - 通过注入电流改变原子结构
  - 原子间的距离决定阻抗

89

89

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



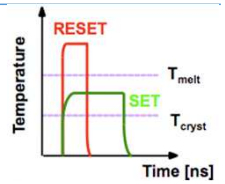
PCM是电阻式存储器: 高阻态 (0)，低阻态 (1)
PCM的位元可以在不同状态之间可靠、快速地切换

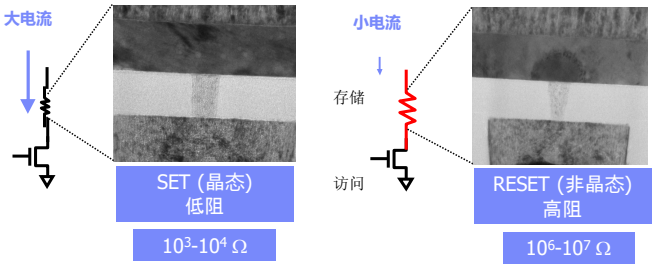
90

90

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





91

91

相变存储器: 优点和缺点

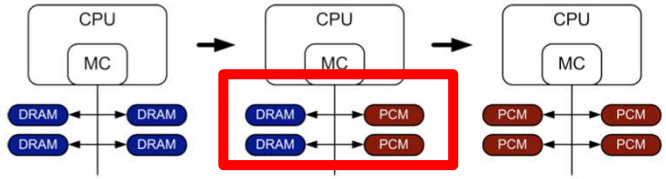
- 优于DRAM之处
  - 更好的工艺规模(容量和成本)
  - 非易失
  - 空闲时功率低 (无需刷新)
- 缺点
  - 延迟更高: ~4-15x DRAM (尤其是写入时)
  - 活跃状态能耗更高: ~2-50x DRAM (尤其是写入时)
  - 重复使用寿命较低 (位元寿命~10^8 次写入)
- 用PCM替换或者协助DRAM组成主存储器的挑战:
  - 减小PCM缺陷的影响
  - 找到合适的方式将PCM引入系统

92

92

基于PCM的主存储器 (I)

- 基于PCM的(主)存储器如何组织?



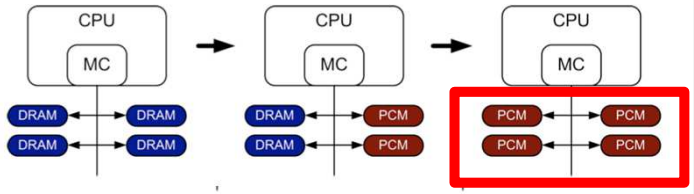
- 混合PCM+DRAM [Qureshi+ ISCA'09, Dhiman+ DAC'09]:
  - 在PCM和DRAM之间如何分区/迁移数据

93

93

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

94

94

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

95

95

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

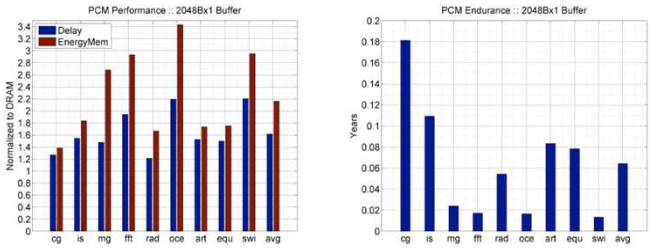
Energy
- 40μA Rd, 150μA Wr
- 2×, 43× DRAM

96

96

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

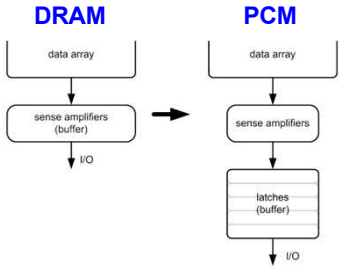


- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.

97

设计PCM架构以减小缺陷的影响

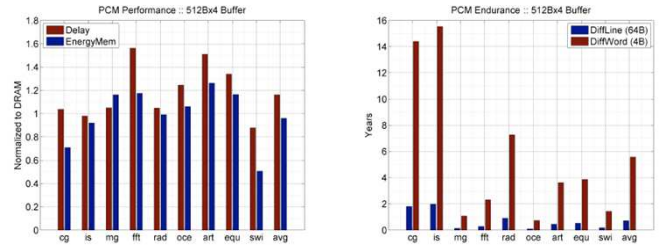
- 思路1: 在每个PCM芯片中使用多个窄行缓冲
  → 减少阵列的读/写 → 更好的耐久性、延迟和能耗表现
- 思路2: 写入阵列时按cache line或字的粒度写
  → 减少不必要的损耗



98

结果: 用PCM构建主存储器

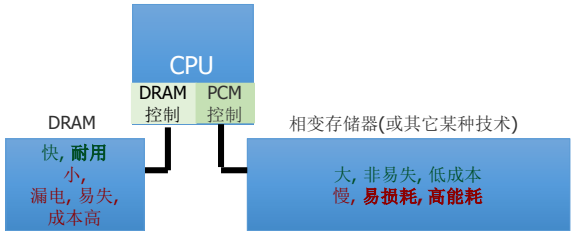
- 1.2x延迟, 1.0x能耗, 5.6-year平均寿命
- 调整规模可以改善能耗、耐久性和密度等指标



- 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?

99

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.

100

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]

101

101

虚拟存储

102

102

现代虚拟存储的两个部分

- 在多任务系统中，虚拟内存为每个进程提供了一个大的、私有的、统一的内存空间幻象
- 命名和保护
  - 每个进程都看到一个大的、连续的地址空间（为了方便）
  - 每个进程的内存都是私有的，即受保护不被其他进程访问（为了共享）
- 需求分页（针对层次结构）
  - 辅助存储容量(磁盘上的交换空间)
  - 主存储速度（DRAM）

103

103

共同的基准：地址翻译

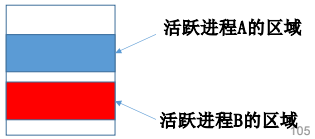
- 通过地址翻译实现大的、私有的、统一的抽象
  - 用户进程基于有效地址（EA）运行
  - 在每次内存引用时硬件将有效地址翻译成物理地址
- 通过地址翻译
  - 控制进程可以引用哪些物理位置(DRAM和/或交换磁盘)
  - 允许动态分配和重新定位物理存储(在DRAM和/或交换磁盘中)
  - 地址转换的硬件和策略由操作系统控制，受用户保护

104

104

内存保护的演化

- 早期不需要保护或翻译
  - 单一进程，每次单一用户
  - 直接使用物理地址(PA)访问所有位置
- 多任务
  - 每个进程被限制在非重叠、连续的物理存储区域
    (存储空间不是从0 地址开始的。。。)
  - 所有的内容都必须放在这个区域
  - 如何防止一个进程读取或破坏另一个进程的代码和数据？

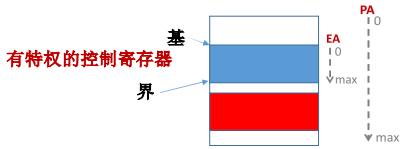


105

105

基和界

- 一个进程的私有存储区域被定义为
  - 基：该区域的首地址
  - 界：该区域的大小
- 用户进程发出从0到界的“有效”地址（EA，私有且统一）



106

106

基和界寄存器

- 翻译和保护机制针对每一次用户的内存访问检查硬件
  - 物理地址(PA)=有效地址(EA)+基
  - 如果有效地址>=界则产生冲突
- 每次切换用户进程，操作系统设置基和界寄存器
- 用户进程不能自行修改基和界寄存器

操作系统需要至少2个有受保护指令和状态的特权级

107

107

分段的地址空间

- 单由基和界确定存储区域的局限性
  - 可分配空间变得支离破碎，很难找到大的连续空间
  - 两个进程可以共享一些存储区域，但不能共享其他存储区域吗？
- 一组“基和界”是保护机制起作用的基本单元
  →给用户多个内存“段”
  - 每个段是连续的存储区域
  - 每个段由一对基和界定义
- 在最开始，代码段和数据段分离
  - 代码与数据使用2组基和界
  - 子进程可以共享代码段

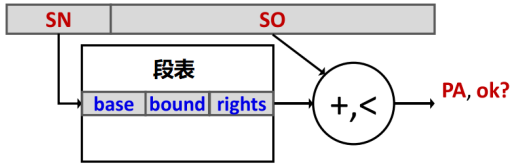
  后来变得复杂:代码、数据、堆栈等

108

108

分段的地址翻译

- 有效地址被划分为段号和段偏移量
  - 段的最大尺寸受段偏移量限制
  - 界动态的设置段的大小
- 每进程一张段翻译表
  - 将段号映射到相应的基和界
  - 每个进程独立映射
  - 用于实施保护的特别结构



分成几个大段有利于隔离管理

109

109

访问保护

- 段表项中的保护位表征每一段的访问权限
- 通常的选项有
  - 可读(R)?
  - 可写(W)?
  - 可执行(E)?
  - ……（可能是各种混杂的选项，比如可高速缓存）
- 比如
  - 正常的数据段→RW(!E)
  - 静态共享数据段→R(!W)(!E)
  - 代码段→R(!W)E
  - 非法段→(!R)(!W)(!E)

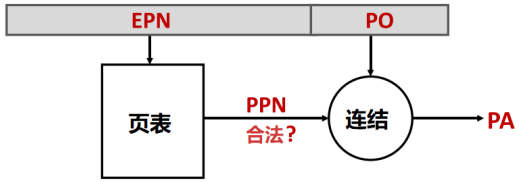
  访问冲突异常会引发操作系统介入

110

110

分页的地址空间

- 将物理地址和有效地址空间分成大小相等且尺寸固定的片段，称为页（页帧，最经典的大小是4KB）
- 物理地址和有效地址都可以解释为页号+偏移量
  - 页表将有效页号翻译成物理页号，偏移量是相同的
  - 物理地址=物理页号+页内偏移量



分成很多个页有利于分配管理

111

111

碎片

- 按段划分导致的外部碎片
  - 有足够大的连续区域，但是存在大量未分配的DRAM空间
  - 内存分页则消除了外部碎片
- 页的内部碎片
  - 分配整个页，页内未使用的字节会被浪费掉
  - 较小的页尺寸能减少内部碎片
  - 现代ISA正在向更大的页尺寸变化（MB）

段和页扮演着不同的角色

112

112

请求页面调度

- 使用主存和“交换”磁盘作为*自动管理*的内存层级
  类似于缓存和主存
- 早期的尝试
  - 冯·诺依曼描述过手动的存储层次
  - Brookner的解释编码（1960年）
  程序解释器管理40KB主存和640KB磁鼓之间的分页
  - Atlas （1962年）
  硬件管理32页磁芯主存和192页磁鼓(512字/页)之间的分页

113

113

请求页面调度——就像Cache一样

- M字节的内存空间，其中最常用的C字节保存在Cache C<<M
- 和以前一样的基本问题
  (1)在DRAM的什么位置“缓存”页面？
  (2)如何在DRAM中找到一个页面？
  (3)什么时候把一页放进DRAM？
  (4)将哪个页面从DRAM置换到磁盘，释放DRAM用于新页面？
- 关键概念差异：交换vs.缓存
  - DRAM不保存磁盘上内容的副本
  - 一页既在DRAM中也在磁盘上–
  - 地址始终没有绑定到一个位置

114

114

请求页面调度:一点也不像Cache

- 规模和时间尺度的差异导致完全不同的实现选择

	L1Cache	L2Cache	分页调度
容量	10 ~ 100KB	MB	GB
块大小	~16B	~128B	4K~4MB
命中时间	几cyc	几十cyc	几百cyc
缺失惩罚	几十cyc	几百cyc	10毫秒
缺失率	0.1~10%	<<0.1%	0.00001~0.001%
命中处理	硬件	硬件	硬件
缺失处理	硬件	硬件	软件

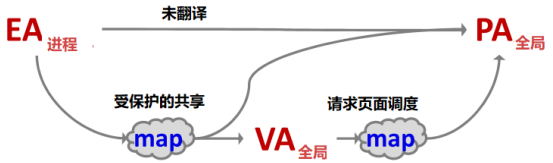
命中时间、缺失开销、缺失率都不是互相独立的指标

115

115

不要用“虚存”来表示一切

- 有效地址(EA)：由每个进程空间中的用户指令发出(保护)
- 物理地址(PA)：对应于DRAM或交换磁盘上的实际存储位置
- 虚拟地址(VA)：指系统范围内的大的线性地址空间中的位置；并非虚拟地址空间中的所有位置都有物理支持(按页调度)



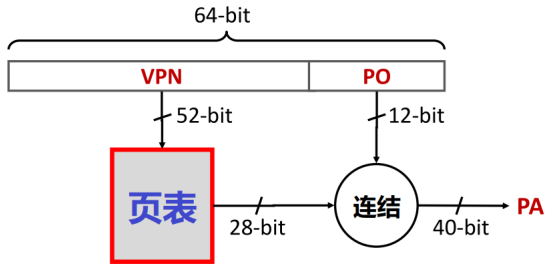
116

116

页表有多大？

- 页表保存着从虚拟页号到物理页号的映射
- 假设64bit虚拟地址和40bit物理地址，页表有多大？

  2^{52} x ~4 字节≈16x10^{15} 字节



这只是一个进程的页表

117

页表应该有多大？

- 不需要跟踪整个虚拟地址空间
  - 分配的虚拟地址总***空间***为2^{64}字节 x #个进程，但其中大部分没有***物理存储***的支持
  - 不能使用超过物理内存的内存位置(DRAM和交换磁盘)
- 好的页表设计应该随物理存储大小线性扩展而不是虚拟地址空间
- 表不能太复杂
  - 页表应该是可以由硬件有限状态机遍历的
  - 页表经常被访问

  今天主要有两种使用模式:

  分层页表和哈希页表

118

快表-TLB

- 用户的每次访存都需要翻译
  - 表的遍历本身也需要访存
  - 不能在每次访存时都遍历表
- 用“cache”保存最近使用过的翻译
- 与cache和BTB类似的“标签”查找结构
  - 相同的设计考虑：A/B/C、替换策略、拆分还是统一、L1/L2等
  - TLB的表项：

  标签：地址tag(来自VA), ASID

  页表项(PTE)：PPN和保护位

  其它：有效位、脏位等

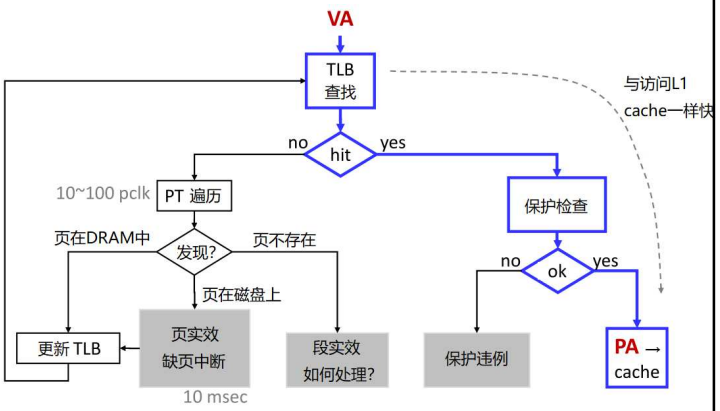
119

TLB设计

- C：L1 指令TLB应覆盖与L1 cache相同的空间，假如L1指令cache为64KB
  - L1指令TLB至少需要16页，但前提是工作集始终使用整页
  - 以前有32~64个表项；现在有几百个
- B：访问一页后，访问下一页的可能性有多大？(粗粒度空间局部性)
  - 通常一个TLB表项一个PTE
  - 例外：MIPS每个TLB表项存2个PTE
- a：相联度最小化冲突？
  - 过去的设计通常采用全相联
  - 现在，2~4路组相联更常见

120

VA→PA翻译流程



121