

高等计算机体系结构

第十讲 Cache

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

阅读材料

- 分层存储体系结构和Cache
- 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
- Maurice Wilkes早期关于cache的论文
 - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

2

2

回顾: 为什么要有分层存储体系结构?

- 我们想要既快又大
- 但是我们无法仅靠一层存储达到目的
- 思路: 采用多层的存储 (越大并且越慢的离处理器越远) 并且确保处理器需要的大多数数据在更快的层中

3

3

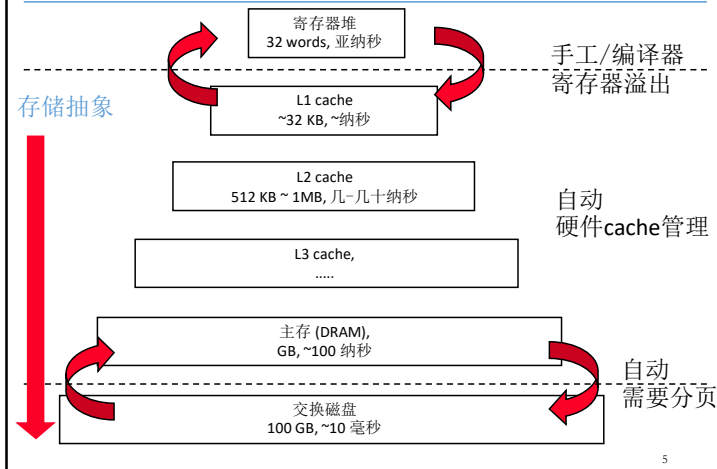
回顾: 存储局部性

- 一个“典型”的程序在引用存储器方面有很多的局部性
 - 比如, 很多典型的程序是由“循环”组成的
- 时间局部性: 一个程序往往会在一个小的时间窗口内多次引用相同的存储位置
- 空间局部性: 一个程序倾向于一次引用一串存储位置
 - 最引人关注的例子:
 - 1. 指令对存储的引用
 - 2. 数组或类似数据结构的引用

4

4

回顾：现代的分层存储体系结构



5

回顾：层次设计注意事项

- 递归的延迟方程

$$T_i = t_i + m_i \cdot T_{i+1}$$

- 目标: 在可以接受的开销范围内获得满意的 T_1
- $T_i \approx t_i$ 将是令人满意的
- 保持低的缺失率 m_i
 - 增加容量 C_i 以降低缺失率 m_i , 但是要注意会增加 t_i
 - 通过更好的管理降低缺失率 m_i (替换::预测你不需要什么, 预取::预测你需要什么)
- 保持低的 T_{i+1}
 - 让更低的层次更快, 但是要注意会增加成本
 - 引入中间层做折衷

6

回顾：cache基础

Block (line): cache中的存储单元

- 内存存在逻辑上按照cache block划分并映射到cache的相应位置

当数据被引用

- 命中HIT: 如果在cache中, 使用被缓存的数据, 不再访存
- 缺失MISS: 如果不在cache中, 将相应的block调入cache
 - 可能不得不将某些别的block踢出cache

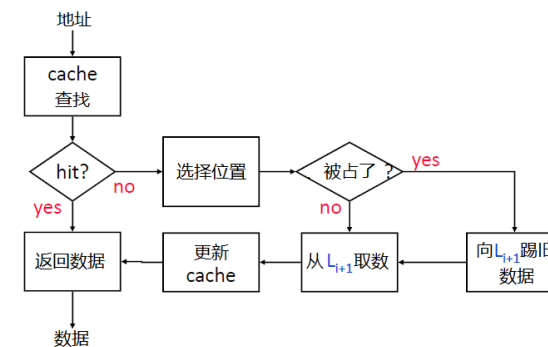
一些重要的cache设计决策

- 放置: 在哪儿以及如何如何在cache中放置/寻找一个block?
- 替换: cache中哪些数据应该被移除?
- 管理的粒度: 大的, 小的还是统一的block?
- 写策略: 写cache的时候应该怎么做?
- 指令/数据: 应该分别对待吗?

7

7

基本操作

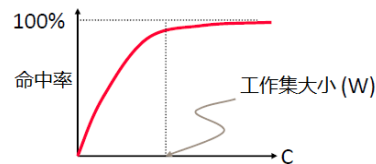


8

8

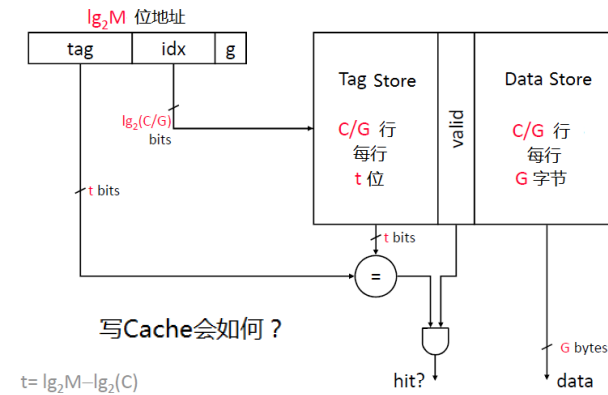
Cache基本参数

- 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)



9

直接映射的Cache (I)



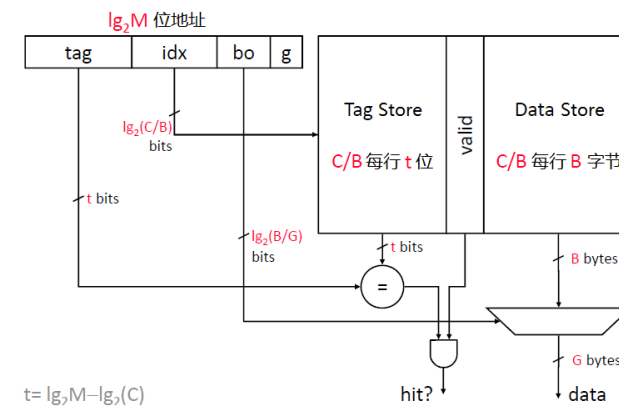
10

存储开销

- 对于每个Cache块 (G字节), 还必须存储额外的“t+1”位, $t = \lg_2 M - \lg_2 (C)$
 - 如果 $M=2^{32}$, $G=4$, $C=16K=2^{14}$
 - 每个4字节的块需要t=18位
 - 60%的存储开销
 - 16KB的cache需要25.5KB的SRAM
- 解决方案: 让多个块 (G字节) 共享一个标签tag
 - 每个B字节的块包含B/G个子块
 - 如果 $M=2^{32}$, $B=16$, $G=4$, $C=16K$
 - 每个16字节的块需要t=18位
 - 15%的存储开销
 - 16KB的cache需要18.4KB的SRAM
 - 16KB的15%够小, 而1MB的15%是152KB
 - 较低/较大的层次, 需要更大的块

11

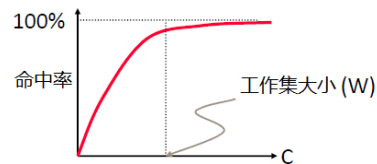
直接映射的Cache (II)



12

直接映射的Cache

- 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^k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

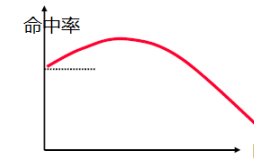


13

13

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

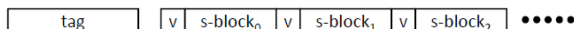


14

14

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

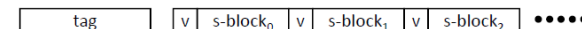


15

15

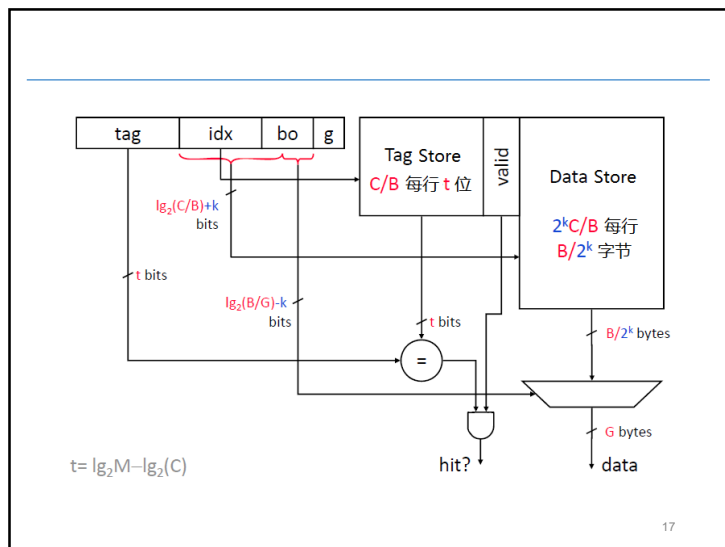
分区 Cache

- 将一个块分成多个子块(或者叫“扇区”)
 - 每个扇区有独立的有效位和脏位
 - 什么时候有用?(思考cache写...)
 - 读的时候会移动多少子块?
- ++ 不需要移动整个cache块
(写的时候只需要验证和更新一个子块即可)
- ++ 可以更自由地移动子块到cache中(一个cache块不需要全部都在cache里)
- 更复杂的设计
- 读的时候可能不能完全利用空间局部性

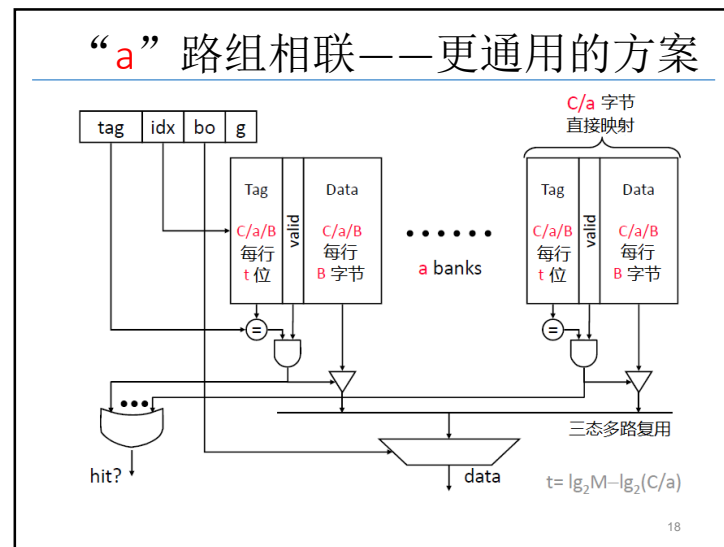


16

16



17



18

“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)”
 - * 直接映射只是特例($a=1$)
 - 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 \rightarrow 冲突越少
 - 如果 $C <$ 工作集大小
 - ???

19

19

组相联 Cache

- 通过提高相联度获得更好的命中率存在边际效益递减
- 更高的相联度使得访问时间更长
- 组内的哪一块在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

20

20

替换策略

- 替换策略只具有二级效应
 - 如果在一组中使用的块少于 **a**，任何敏感的替换策略都会很快收敛
 - 如果在一组中使用的块大于 **a**，所有的替换策略都没法解决问题

21

21

实现 LRU

- 思路: 换出最近最少访问的块
- 需要记录块被访问的序
- Q: 2路组相联cache
 - 需要什么来实现 LRU?
- Q: 4路组相联cache
 - 一个组中有4块时, 有多少种可能的序?
 - 编码一个块的LRU序需要多少位?
 - 需要什么逻辑来确定LRU策略中被替换的块(victim牺牲者)?

22

22

LRU的近似性

- 大多数现代处理器在高相联度的cache中都没有实现“真正的LRU”
- 为什么?
 - 真的LRU很复杂
 - LRU只是对局部性的近似预测 (不是可能的最佳替换策略)
- “伪”LRU的例子:
 - 非 MRU (非最近使用)
 - 分层 LRU: 将4路组(set)分为2路“组(group)”, 记录MRU“组”和每一“组”中的MRU块
 - 牺牲者-下一个牺牲者替换: 仅保持记录牺牲者和下一个牺牲者

23

23

分层 LRU (非MRU)

- 将一个set划分为多个group
- 记录MRU group
- 记录每个group中的MRU块
- 替换时, 这样选择牺牲者:
 - 非MRU group中的某个非MRU块

24

24

分层 LRU (非MRU) 的问题

- 8路cache
- 2个4路group
- 什么样的访问模式会比真LRU表现的还差?
- 什么样的访问模式会比真LRU表现的要好?

25

25

牺牲者/下一个牺牲者策略

- 每一个set中只有2个块的状态被记录
 - 牺牲者 (V), 下一个牺牲者(NV)
 - 所有其它的块被标记为(O) – 普通块
- 当cache不命中时
 - 替换 V
 - 将 NV 变为 V
 - 随机选择一个O 变为 NV
- 当cache命中V
 - 将 NV 变为 V
 - 随机选择一个O 变为 NV
 - 将 V 变成 O

26

26

牺牲者/下一个牺牲者策略 (II)

- 当cache命中NV
 - 随机选择一个O 变为 NV
 - 将 NV 变成 O
- 当cache命中O
 - 什么也不做

27

27

替换策略

- LRU vs. 随机
 - Set 抖动: 当“工作集”大于组相联度时可能发生
 - 4路: 循环引用A, B, C, D, E
 - 使用LRU策略命中率为0%
 - 随机替换策略在抖动发生时效果更好
- 实际当中:
 - 取决于工作负载
 - LRU和随机的平均命中率差不多
- LRU和随机的混合
 - 如何在两者中选择? Set采样
 - See Qureshi et al., “A Case for MLP-Aware Cache Replacement,” ISCA 2006.

28

28

最优替换策略?

- Belady选择
 - 替换程序会在最远的将来引用的块
 - Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, 1966.
 - 如何实现? 模拟?
- 这对最小化缺失率是最优的吗?
- 这对最小化执行时间是最优的吗?
 - 不是的. Cache的缺失延迟/开销在不同的块之间是不一样的!
 - 两个原因: 远程 vs. 本地 cache, 缺失的重叠
 - Qureshi et al. "A Case for MLP-Aware Cache Replacement," ISCA 2006.

29

29

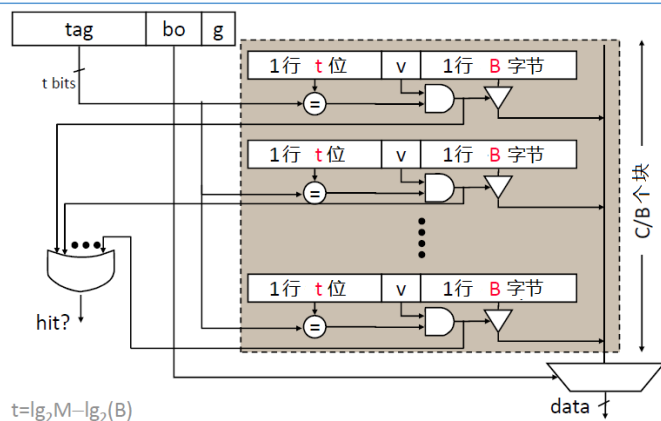
Cache替换和页替换

- 物理内存(DRAM)是磁盘的cache
 - 通常由系统软件通过虚拟存储子系统来管理
- 页替换与cache替换类似
- 页表是物理内存数据存储的“标签存储”
- 有什么不同?
 - 硬件vs软件
 - Cache中块的编号 vs 物理内存的编号
 - 可以容忍的找到替换内容的时间长短

30

30

全相联的Cache: $a=C/B$

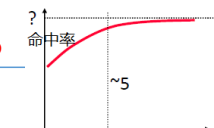


31

31

全相联的Cache: $a=C/B$

- “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 , $a=4 \sim 5$ 时与 $a=C/B$, 对于典型的程序而言没有太大的区别 (一样好)



32

32

Cache缺失的种类

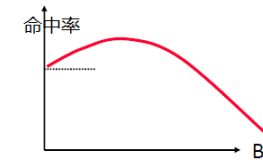
- 强制(Compulsory)缺失
 - 第一次引用某个地址(块)总是导致一个缺失
 - 后续的引用将会命中，除非cache块因为某些原因被替换掉
 - 当局部性很差的时候会成为主要的缺失类型
- 容量(Capacity)缺失
 - Cache太小不足以保持需要的每一个数据
 - 相同容量情况下，在全相联cache(采用最优替换策略)中也可能发生
- 冲突(Conflict)缺失
 - 不属于强制缺失和容量缺失的任何其它缺失情况

33

33

强制缺失

- 对Cache块的第一次引用总是不命中
- 当局部性较差时，在几种缺失中占主导地位
 - 例如，在“流式”数据访问模式中，访问了许多地址，但每个地址都被恰好访问一次→很少重复使用来平摊成本
- 主要设计因素：**B**和“预取”

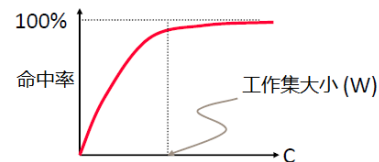


34

34

容量缺失

- Cache太小，无法容纳需要的所有内容
- 使用最佳(Belady)替换策略的全相联Cache中可能发生的缺失
- 当 $C < W$ 时，占主导地位
 - 例如，由于对周期时间的权衡，L1 Cache永远做不到足够大
- 主要设计因素：**C**

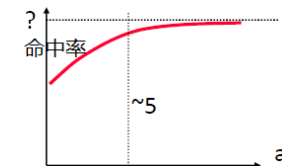


35

35

冲突缺失

- 直接映射或组相联时，由于冲突而替换先前访问的Cache块导致的缺失
- 既非强制也非容量导致的缺失
- 当 $C \approx W$ 或 C/B 较小时，占主导地位
- 主要设计因素：**a**



36

36

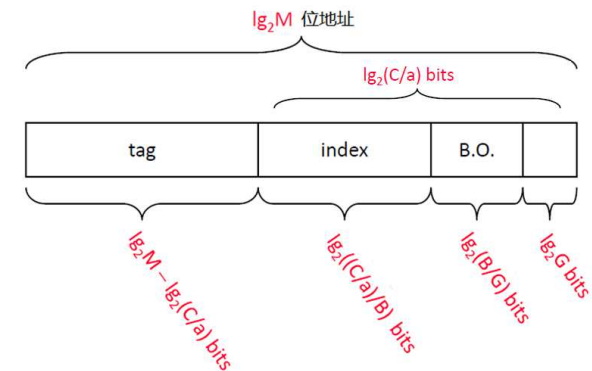
回顾：Cache基本参数

- 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

37

37

地址域的逻辑分配

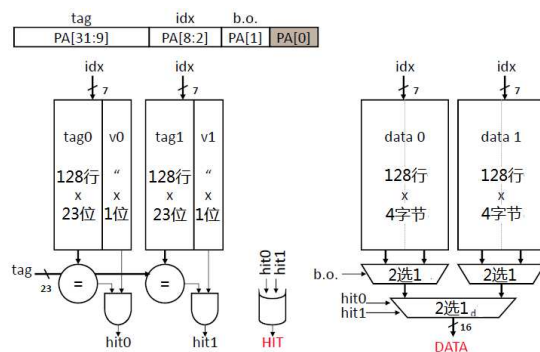


38

38

$$M=2^{32}, a=2, C=1k, B=4, G=2$$

基本方案

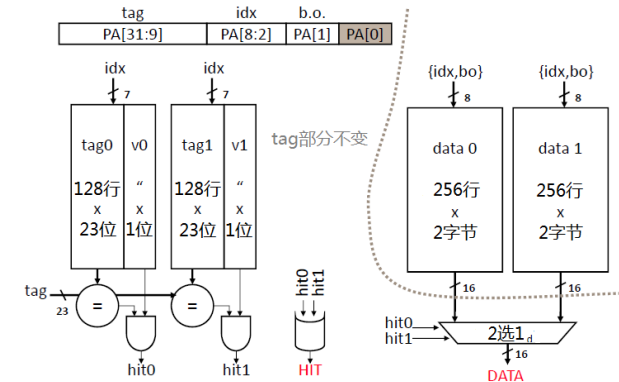


39

39

$$M=2^{32}, a=2, C=1k, B=4, G=2$$

更“瘦”的Data Store

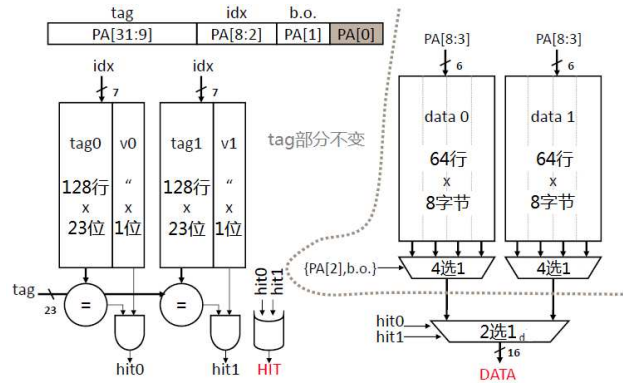


40

40

$$M=2^{32}, a=2, C=1k, B=4, G=2$$

更“胖”的Data Store

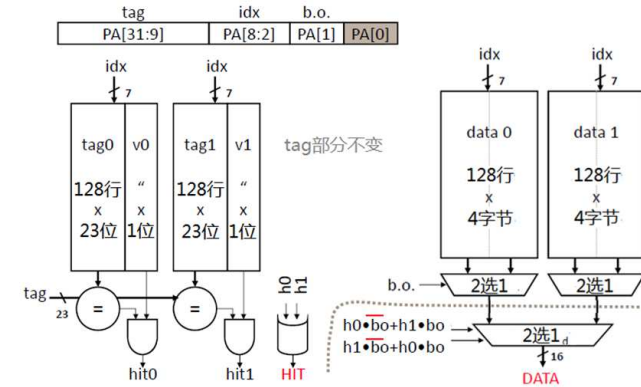


41

41

$$M=2^{32}, a=2, C=1k, B=4, G=2$$

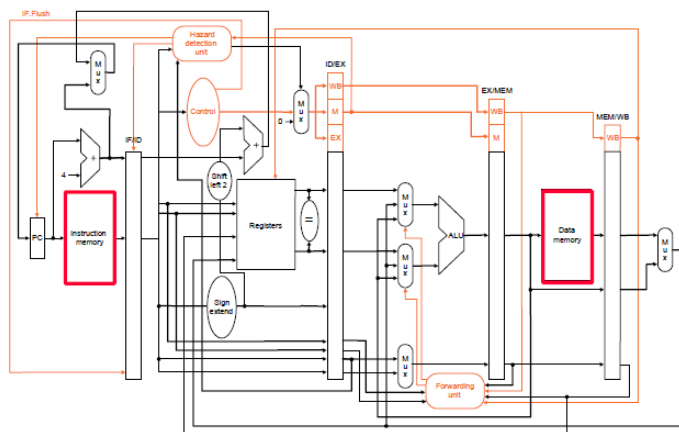
每个块在SRAM的2个bank上交叉存取



42

42

流水线中的Cache



43

43

在按序执行流水线中加入Cache

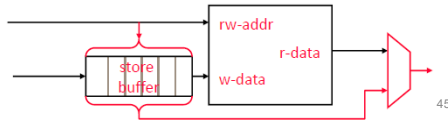
- 取指令和LW时，假设SRAM查找只需要1个周期
 - 如果命中，“魔法内存”
 - 如果不命中，暂停流水线，直到Cache准备好
- SW时，假设SRAM查找只需要1个周期
 - 如果不命中，暂停流水线，直到Cache准备好
 - 流水线必须等待吗？
 - 如果命中，???

44

44

写缓冲 (Store Buffer)

- 对于SW，在提交写入Data Store之前，需要先检查Tag Store以确定命中
 - Data Store写入发生在下一个周期
 - 如果SW后面紧跟着LW → 可能由结构冒险导致停顿
- 能不能更好？
 - 检查Tag Store是否命中后，缓冲SW的数据，直到下一个空闲的Data Store周期
 - 必须确保在此之前Cache行不被替换
- 内存转发
 - 后续的LW必须检查写缓冲中未执行完的SW的地址，检测RAW相关



45

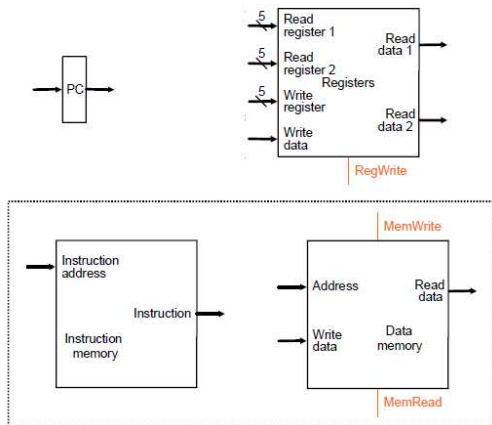
CPU是否必须等待miss?

- 严格的按序执行流水线在LW缺失时必须停顿
- SW缺失是“无阻塞”的
 - 其他指令可以继续执行，包括LW和SW
 - 未完成的前序SW缺失可能会延迟后续的LW
 - 通过停顿或转发来解决RAW冒险
 - 可以跟踪多个缺失的SW（相同和不同的地址）
- 现代的乱序执行处理器允许在LW和SW缺失都是无阻塞的
 - 在检测和解决所有的内存数据依赖(RAW/WAW/WAR)时增加了极大的复杂性
 - 在高频率“超标量”处理器中必不可少，否则将会在缺失发生时付出高昂的代价

46

46

程序可见的状态（体系结构状态）



47

47

指令和数据，统一还是分离？

- 回顾历史
 - “哈佛”结构来源于霍华德艾肯在哈佛建造的Mark I，有独立的指令和数据存储器
 - “普林斯顿”结构来源于冯·诺依曼的指令和数据统一存储
- 今天用来描述统一或分离的“Cache”
- 高性能处理器对指令和数据使用分离和不对称L1缓存
 - 指令和数据内存空间不相交
 - 取指令通常占用空间较小，空间局部性较高，是只读的
 - 分离的L1 Cache提供双倍带宽、无交叉污染和独立设计定制
- L2和L3是统一的(为什么?)

48

48

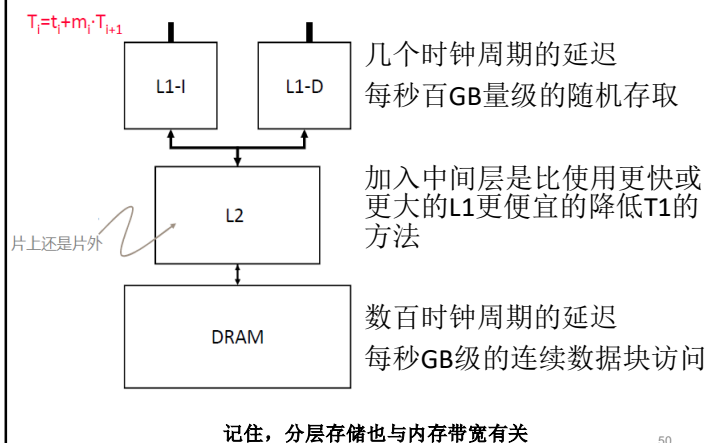
指令 vs. 数据 Cache

- 统一的cache:
 - + 动态共享cache空间: 不会出现静态分区中可能出现的过度供应问题 (将指令和数据cache分开)
 - 指令和数据可能互相竞争 (对任何一方都没有空间的保证)
 - 指令和数据的访问发生在流水线的不同位置, 统一的cache应该放在什么位置才能保证快速的访问?
- 第一层cache(FLC)几乎总是分开的
 - 主要是因为上面的最后一个原因
- 第二层和更高层的cache几乎总是统一的

49

49

多层Cache



50

50

多层Cache设计

- 高层
 - **C**小: 受SRAM访问时间上限的约束
 - **B**小: 受C/B影响和细粒度空间局部性收益上限的约束
 - **a**: 需要考虑C/B的影响
- 底层
 - **C**大: 芯片面积上限的约束(或愿意支付多少片外开销)
 - **B**大: 减少标签存储开销并利用粗粒度空间局部性
 - **a**: 复杂性上限的约束(片外实现)

51

51

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
 - 通常比较大而且具有较高的相联度; 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
 - 串行: 只在第一层cache缺失时访问第二层cache
 - 第二层与第一层看到的访问行为不一样
 - 第一层更像一个过滤器

52

52

处理“写”(Store)

■ 什么时候把cache中修改过的数据写到下一级?

- 写直达: 当写的动作发生时
- 写回: 当cache块被换出时

• 写回

- + 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”

• 写直达

- + 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

53

53

处理“写”(Store)

• 当发生写缺失时是否分配cache块?

- Yes
- No

• 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)

54

54

写直达Cache

• 在 L_i 中写命中时, 是否应该更新 L_{i+1} (无论是Cache还是DRAM)?

• 写直达

- 简单的策略, 直接访问内存的I/O设备总是看到与Cache一致的值
- 对于当今的高性能微处理器来说, 这不是一个可行的选择
3.0GHz, IPC=2, 10%的SW, ~8byte/SW \rightarrow ~5GB/秒

L1写直达L2仍然有用

• 采用写直达策略时, 在不命中时是否应该在 L_i 中分配Cache块(也称为写分配)?

你相信LW和SW会对同一个地址有局部性吗?

55

55

写回Cache

• 对Cache的写可以在 L_1 中缓存, 直到该块被替换到 L_{1+1}

- 写缺失时, 整个数据块会被引入, 被写的部分可以更新
- LW和SW在替换之前会命中
- 替换时, 必须将Cache的副本写入内存替换可能的过时副本
降低了较低层次所需的带宽

• 脏位

- 每个数据块保留一个状态位, 以记录一个数据块自引入 L_i 后是否有过修改
- 如果不脏, 更换时无需写回

• 如果I/O设备想读取在Cache中有副本的内存位置, 该怎么办?

56

56

标签存储表项中有什么?

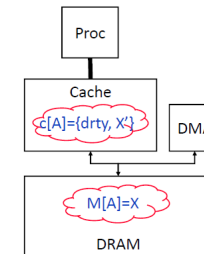
- 有效位
- 标签
- 替换策略位
- 脏位?
 - 写回vs.写直达cache

57

57

写回Cache和DMA

- 写回策略时的内存不总是最新的
- DMA应该可以看到当前值(也就是Cache一致性)
- 选项1: 软件在对DMA编程之前清洗整个Cache
- 选项2: Cache监控总线对Cache的外部请求(DMA和其他Cache/进程), 并“以某种方式纠正它”



58

58

包含原则

- 传统上, L_i 的内容总是 L_{i+1} 的子集
 - 如果一个内存位置必须在 L_i 中, 则应该也必须在 L_{i+1} 中
 - 外部的代理(I/O、其他CPU)只需检查底层, 就可以知道内存位置是否已Cache, 不需要消耗L1带宽
- 当 L_{i+1} 具有较低的关联度时, 保持不重要
- 例如, 一个 L_i 的缺失可能触发多个 L_i 的替换
 - 假设 L_i 的 $a < 1$, L_{i+1} 的 $a = 1$
 - 假设 x 、 y 、 z 具有相同的 L_i 索引
 - 假设 y 、 z 具有相同的 L_{i+1} 索引, 但不同于 x 的索引
 - 假设最初 x 和 y 都Cache在 L_i 中(因此也在 L_{i+1})
 - 根据LRU, 假设对 z 的缺失将使 x 从 L_i 中被踢出
 - 由于冲突, z 必须将 y 从 L_{i+1} 中踢出
 - y 也必须被踢出 L_i 以保持包含关系

59

59

软件辅助内存层次结构

- 分离“时间”和“非时间”层次
 - 暴露在ISA中(如Intel IA64)
 - load和store指令包含关于Cache缺失时Cache位置的提示
 - 只是“提示”而已, 实现时可以支持层次结构的子集或不支持任何层次结构

60

60

你机器里的Cache是什么样的？

- 如何确定计算机中的Cache配置
 - 容量(C)、关联度(a)和块大小(B)
 - 层数
- 软件的功能行为应该检测不到Cache是否存在
- 但是你可以通过测量执行时间来推断Cache未命中的数量

61

61

容量测试

- 假设C是2的幂
- 增加R值（R是2的幂）
 - 分配大小为R的缓冲区
 - 依次读取R中的每个存储位置，并重复
- 对于 $R \leq C$ ，我们预期命中Cache
- 对于 $R > C$ ，我们预期会有Cache缺失，并且出现显著的访存时间增加
- 通过继续增大R，当缓冲区大小超出下一个Cache层次时，可以感觉到访存时间的再次显著增加
 - 注意：这个测试独立于B和a

62

62

块大小测试（已知C的情况下）

- 分配一个大小为R（C的整数倍）的缓冲区
- 增加S，只读取缓冲区中第S个内存位置，然后重复
- 因为 $R > C$ ，我们预计大约每次第一次访问R块时都会miss
- 我们预计平均访存时间会随着S的增大逐渐变得糟糕，直到 $S \geq B$

如何检测较低层次Cache的块大小？

63

63

相联度测试（已知C的情况下）

- 增加R值（R是C的整数倍）
 - 分配大小为R的缓冲区
 - 依次读取第C个内存位置，并重复
- 所有 R/C 的引用地址映射到同一个组
- 当 $a \geq R/C$ 时，我们预期访存命中，因为所有引用的地址都在一组中
- 当 $a < R/C$ 时，我们预期至少会有一些miss，因为不是所有引用的地址都能同时匹配
 - 如果使用LRU，我们预期会出现100%的Cache miss

如何检测较低层次Cache的相联度？

64

64

还可以测什么？

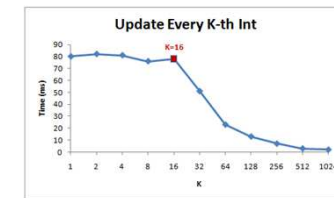
- 写直达还是写回
- 写分配
- 统一设计还是分离设计
- 指令Cache的C, B, a
- T_i
- 相联Cache的替换策略
-
- 基于我们对Cache的简单理解，实验可能无法准确预测具有虚拟内存、复杂层次结构和预取器的现代CPU的行为，但它们仍然可以告诉你很多。试试看！

65

65

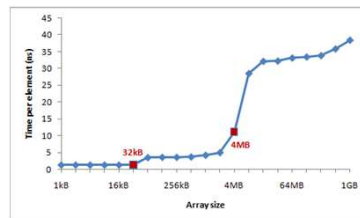
```
int[] arr = new int[64 * 1024 * 1024];
// Loop 1
for (int i = 0; i < arr.Length; i++) arr[i] *= 3;
// Loop 2
for (int i = 0; i < arr.Length; i += 16) arr[i] *= 3;

for (int i = 0; i < arr.Length; i += K) arr[i] *= 3;
```



66

```
int steps = 64 * 1024 * 1024;
int lengthMod = arr.Length - 1;
for (int i = 0; i < steps; i++)
{
    arr[(i * 16) & lengthMod]++;
}
```

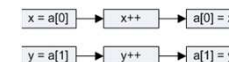


67

```
int steps = 256 * 1024 * 1024;
int[] a = new int[2];

// Loop 1
for (int i=0; i<steps; i++) { a[0]++; a[0]++; }

// Loop 2
for (int i=0; i<steps; i++) { a[0]++; a[1]++; }
```



68

Cache的性能

69

Cache的参数vs. 缺失率

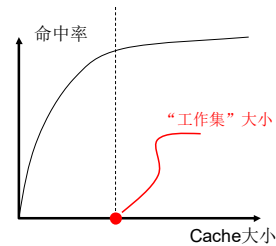
- Cache大小
- 块大小
- 相联度
- 替换策略
- 插入/放置策略

70

70

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

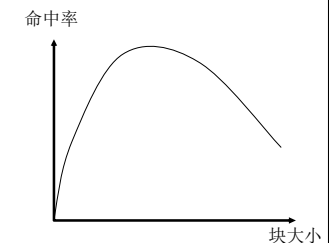


71

71

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



72

72

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?

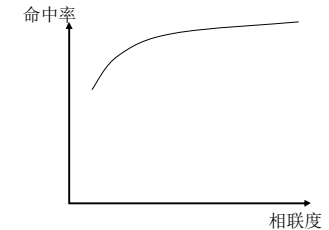


73

73

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



74

74

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

75

75

改善 Cache “性能”

- 回忆
 - 平均存储访问时间 (AMAT)
$$= (\text{命中率} * \text{命中延迟}) + (\text{缺失率} * \text{缺失延迟})$$
- 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

76

76

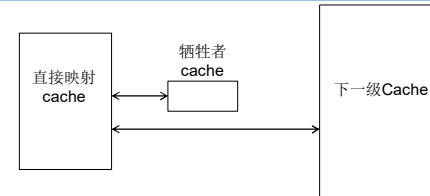
改善Cache性能

- 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)
 - 每周多次访问
 - 软件的方法

77

77

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

78

78

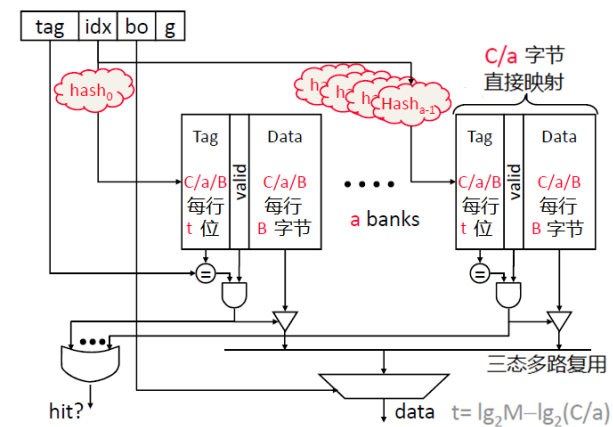
哈希和伪相联

- 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

79

79

偏斜组相联

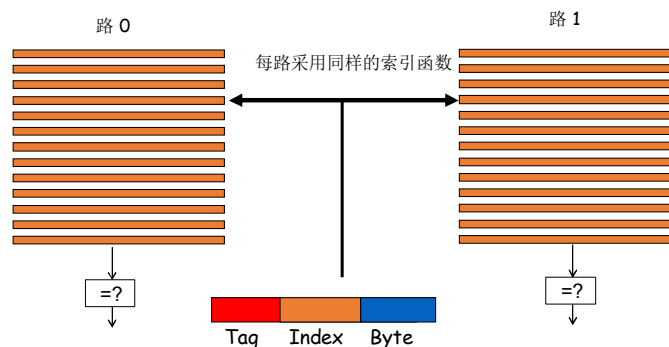


80

80

偏斜相联Cache (I)

- 基本的2路相联 cache 结构

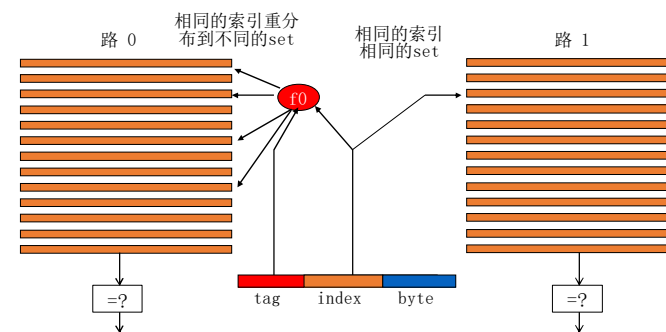


81

81

偏斜相联Cache (II)

- 偏斜相联cache
 - 每个 bank 有不同的索引函数



82

82

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.

83

83

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?

84

84

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!

85

85

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?

86

86

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

87

87

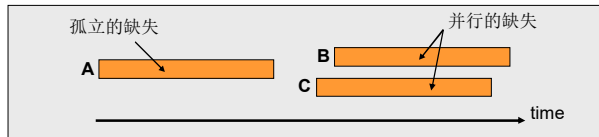
改善Cache性能

- 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)
 - 每周多次访问
 - 软件的方法

88

88

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?

89

89

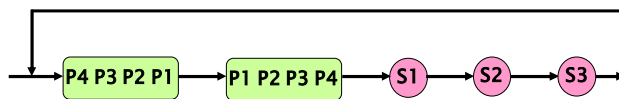
传统的 Cache 替换策略

- 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

90

90

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

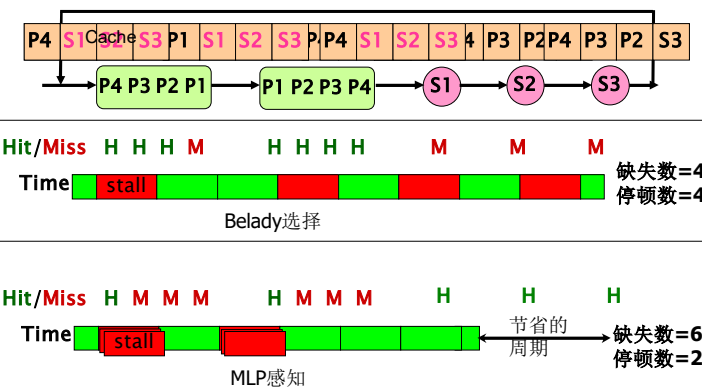
- 两种替换算法:
1. 最小化缺失数 (Belady选择)
 2. 减少孤立缺失 (MLP感知)

Cache有4行, 全相联

91

91

最少的缺失 ≠ 最佳的性能



92

92

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.

93

93

虚存和cache交互

94

地址翻译与cache

- 什么时候需要做地址翻译?
 - 访问L1 cache之前还是之后?
- 换句话说, cache是虚拟编址还是物理编址?
 - 虚拟 vs 物理 cache
- 虚拟编址的cache有什么问题要解决?
- “同义词”问题:
 - 两个不同的虚拟地址可能映射到相同的物理地址 → 相同的物理地址可能出现在cache中的多个位置 → 可能导致数据的不一致

95

95

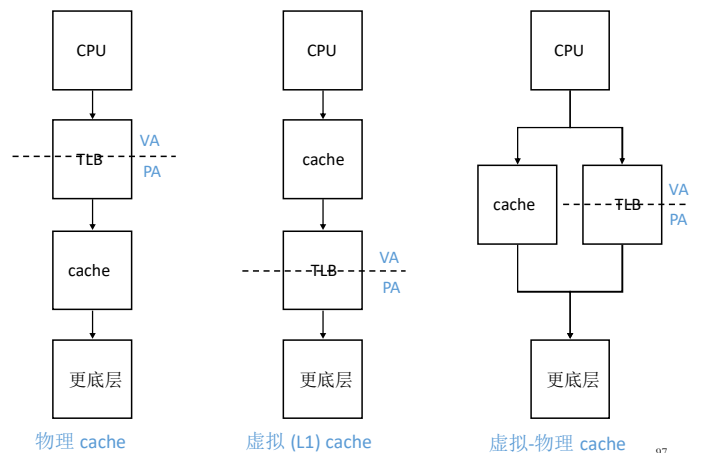
“同音异义词”和“同义词”

- “同音异义词”: 同一个虚拟地址可能映射到两个不同的物理地址
 - 为什么?
 - 虚拟地址可能在不同的进程中
- “同义词”: 不同的虚拟地址可能映射到同一个物理地址
 - 为什么?
 - 不同的页可能共享同一进程内或者跨进程的物理帧
 - 原因: 共享库, 共享数据, 同一个进程内的写时拷贝页, ...
- “同音异义词”和“同义词”会惹什么麻烦吗?
 - Cache是虚拟编址还是物理编址?

96

96

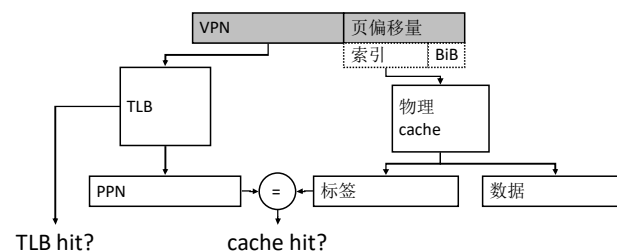
Cache-VM 交互



97

虚拟-索引,物理-标签

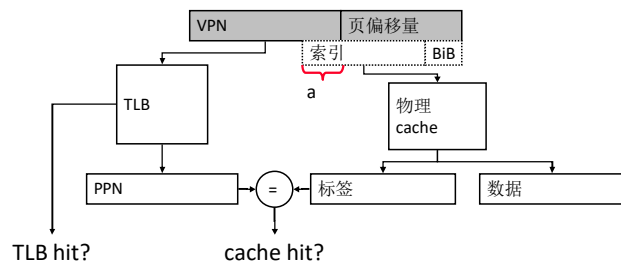
- 如果 $\text{Cache} \leq (\text{页大小} \times \text{相联度})$, cache索引位只来自页的偏移量部分 (在虚拟地址和物理地址中相同)
- 如果片内有cache和TLB
 - 用虚拟地址同时索引cache和TLB
 - cache检查标签(物理的)并比对TLB的输出给出结果



98

虚拟-索引,物理-标签

- 如果 $\text{Cache} > (\text{页大小} \times \text{相联度})$, cache索引位将包含 VPN \Rightarrow “同义词” 会引发问题
 - 同一个物理地址可能存在于两个不同的位置
- 如何解决?



99

解决“同义词”问题的一些方法

- 限制cache大小 (页大小 \times 相联度)
 - 只从页偏移量获得索引
- 写一个块时, 搜索所有可能包含相同物理块的标记, 更新/置为无效
 - Alpha 21264, MIPS R10K
- 在操作系统中限制页的放置
 - 确保虚拟地址的索引 = 物理地址的索引
 - 称为页着色
 - SPARC

100

深入思考的问题

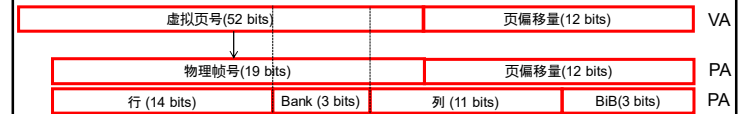
- 在哪一层cache我们需要考虑“同义词”和“同音异义词”的问题?
- 系统软件的页映射算法对分层存储结构的哪一层会产生影响?
- 页着色有哪些潜在的优点和缺点?

101

101

虚拟存储- DRAM 的交互

- 操作系统会影响DRAM中的地址映射



- 操作系统能够控制虚页映射到哪一个bank/channel/rank

- 可以通过页着色最小化bank的冲突
- 或者最小化应用之间的干扰

102

102