

高等计算机体系结构

第九讲: 分层存储体系结构和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
2020-05-08

1

提醒: 作业

- 作业 4
 - 今天截止
 - 流水线2
- 作业 5
 - 今晚发布, 5月22日截止
 - Cache和Memory

2

实验2-5

- 今晚发布, 预计7月10日截止

3

阅读材料

- 分层存储体系结构
- Patterson & Hennessy's *Computer Organization and Design: The Hardware/Software Interface* (计算机组成与设计: 软硬件接口)
 - 第五章: 5.1-5.3
- Maurice Wilkes早期关于cache的论文
 - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

4

回顾：精确异常的解决方案

- 重排序缓冲
 - 思路: 乱序执行指令, 产生体系结构状态可见的结果之前重排序
 - 好处
 - 用很简单的概念来支持精确异常
 - 可以消除“虚假的”相关
 - 坏处
 - 有可能需要访问ROB以获得尚未写入寄存器堆的结果
- 历史缓冲
- 未来寄存器堆
- 检查点

5

5

回顾：精确异常的解决方案

- 重排序缓冲
- 历史缓冲
 - 思路: 指令执行完成后更新寄存器堆, 但是当有异常发生时撤销那些更新 (UNDO)
 - 好处:
 - 寄存器堆中保有最新的值, HB的访问不在关键路径上
 - 坏处:
 - 需要读目的寄存器的旧值
 - 在异常时需要回滚HB→ 增加异常/中断的处理时延
- 未来寄存器堆
- 检查点

6

6

回顾：精确异常的解决方案

- 重排序缓冲
- 历史缓冲
- 未来寄存器堆
 - 思路: 维护两个寄存器堆 (投机的和体系结构的)
 - 体系结构的寄存器堆: 按程序序更新以获得精确异常, 后端寄存器堆
 - 使用ROB来保证按序的更新
 - 未来的寄存器堆: 一条指令执行完毕后立即更新(如果这条指令是最新的一条与寄存器堆的指令), 前端寄存器堆
 - 好处
 - 不需要从ROB中读取值 (不需要CAM或者间接寻址)
 - 坏处
 - 多个寄存器堆
 - 发生异常时需要从一个堆向另一个堆复制数据
- 检查点

7

7

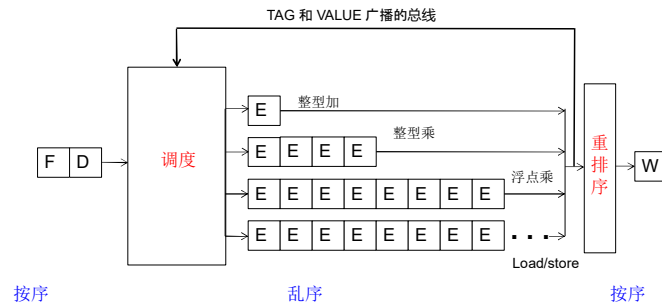
回顾：精确异常的解决方案

- 重排序缓冲
- 历史缓冲
- 未来寄存器堆
- 检查点
 - 目标: 恢复前端状态 (未来寄存器堆), 这样可以使分支后正确的下一条指令能够在分支预测错误被解决后立即执行
 - 思路: 当分支取指时对前端寄存器状态设立检查点, 同时对分支旧的指令结果保持状态更新
 - 好处?
 - 坏处?

8

8

回顾：现代流水线的两个“驼峰”



- 驼峰 1: 保留站(调度窗口)
 - 驼峰 2: 重排序(ROB, 又叫指令窗口或者动态窗口)
- 带精确异常的乱序执行

9

9

回顾：乱序执行

- 寄存器重命名消除错误的相关, 建立了生产者和消费者的联系
- 缓冲使得流水线可以执行独立的操作以保持流水
- 标签广播使得指令之间能够交互生产出的值
- 唤醒和选择保证了乱序的分发

10

10

乱序执行: 受限的数据流

- 乱序引擎动态的构建一个程序块的数据流图
 - 哪个程序块?
- 数据流受限指令窗口
 - 指令窗口: 所有已经译码但是尚未提交(回收)的指令
- 能对整个程序乱序执行吗?
- 为什么最好能?
- 换句话说, 如何能有一个大的指令窗口?
- 用Tomasulo算法就能高效地处理乱序执行吗?

11

11

前面的例子：周期7结束时RAT和RS的状态

周期7结束时:

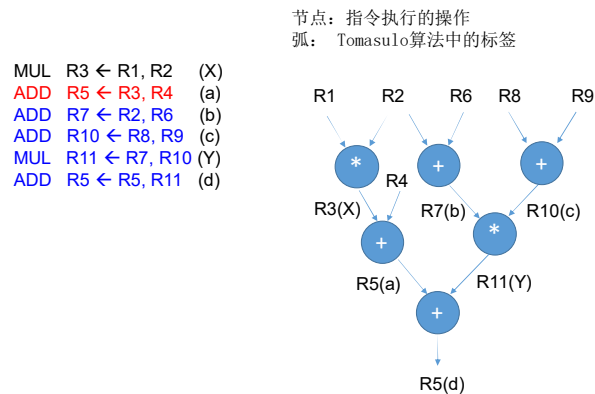
	V	Tag	Value	V	Tag	Value	
R1	1	~	1	a	0	X	1
R2	1	~	2	b	1	~	2
R3	0	X	~	c	1	8	1
R4	1	~	4	d	0	a	~
R5	0	d	~				
R6	1	~	6				
R7	0	b	~				
R8	1	~	8				
R9	1	~	9				
R10	0	c	~				
R11	0	y	~				



12

12

数据流图



13

受限的数据流

- 乱序执行的机器是“受限的数据流”机
 - 基于数据流的执行被局限在微体系结构层
 - ISA 仍然是基于冯诺依曼模型的(顺序执行)
- 回顾数据流模型(ISA 层):
 - 数据流模型: 指令的取指和执行按照数据流的序
 - 操作数准备好
 - 没有指令指针(程序计数器)
 - 指令的序由数据流的相关性决定
 - 每条指令指明“谁”是结果的接收者
 - 当所有操作数准备好, 指令就可以“发射”

14

处理存储相关性(I)

- 乱序执行的机器中需要遵从存储的相关性
 - 需要在提供高性能的同时做到这一点
- 观察和问题: 存储的地址直到load/store的执行阶段才能够获得
- 结果 1: 重命名存储地址很困难
- 结果 2: 决定load/store的相关或者独立需要在它们执行之后处理
- 结果 3: 当一个load/store的地址已经准备好, 可能同时会有新的/旧的load/store尚未确定地址

15

处理存储相关性(II)

- 什么时候可以在乱序执行引擎中调度一条load指令?
 - 问题: 一条新的load指令的地址比一条旧的store指令的地址先准备好
 - 被称为存储违例消解问题或未知地址问题
- 方法
 - 保守: 停顿 load 直到所有之前的 store 计算出它们的地址(或者甚至提交)
 - 积极: 假设 load 独立于地址未知的 store, 立即调度 load
 - 智能: 预测 (使用更复杂的预测器) load 是否与未知地址的 store 相关

16

处理Store-Load相关性

- 在所有前序的store地址可用之前，load的相关性状态是未知的
- 乱序执行引擎如何检测出一条load指令与它之前的store指令相关？
 - 选项 1: 等待，直到所有前序store提交(无需检测)
 - 选项 2: 在store缓冲中维护一个等待的store列表，检查load地址是否与前序store地址匹配
- 乱序执行引擎如何基于前序store来处理load指令调度？
 - 选项 1: 假设 load 与所有前序store相关
 - 选项 2: 假设 load 与所有前序store不相关
 - 选项 3: 预测 load 与一条未完成的store的相关性

17

17

存储违例消解(I)

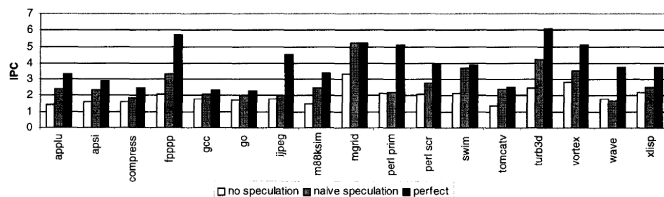
- 选项 1: 假设 load 与所有前序 store 相关
 - + 不需要恢复
 - 太保守: 对独立的 load 施加了不必要的延迟
- 选项 2: 假设 load 与所有前序 store 不相关
 - + 简单并且可能是常见的情况: 独立的 load 没有延迟
 - 预测错误需要恢复/重新执行
- 选项 3: 预测 load 与未完成的store的相关性
 - + 更准确
 - 预测错误还是需要恢复/重新执行
 - Alpha 21264 : 先假设 load 是独立的，当发现相关之后延迟 load
 - Moshovos et al., "Dynamic speculation and synchronization of data dependences," ISCA 1997.
 - Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.

18

18

存储违例消解(II)

- Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.



- 预测 store-load 的相关性对性能非常重要
- 简单的预测器(基于过去历史)能够获得大部分的潜在性能

19

19

思考

- 许多其它的设计选择
- 保留站应该是集中式的还是分布式的？
 - 有什么样的tradeoff?
- 是应该由保留站和ROB存储值还是应该有一个集中式的物理的寄存器堆保存所有的值？
 - 有什么样的tradeoff?
- 到底什么时候一条指令会广播它的标签？
- ...

20

20

思考

- 如何在乱序执行的机器中实现分支预测?
 - 考虑分支历史寄存器和PHT的更新
 - 考虑预测错误的恢复
 - 如何能够快速地完成?
- 如何结合乱序执行和超标量执行?
 - 不同的概念
 - 指令重命名的并发
 - 标签广播的并发
- 如何结合超标量、乱序和分支预测?

21

21

推荐阅读

- Kessler, “The Alpha 21264 Microprocessor,” IEEE Micro, March-April 1999.
- Boggs et al., “The Microarchitecture of the Pentium 4 Processor,” Intel Technology Journal, 2001.
- Yeager, “The MIPS R10000 Superscalar Microprocessor,” IEEE Micro, April 1996
- Tendler et al., “POWER4 system microarchitecture,” IBM Journal of Research and Development, January 2002.

22

22

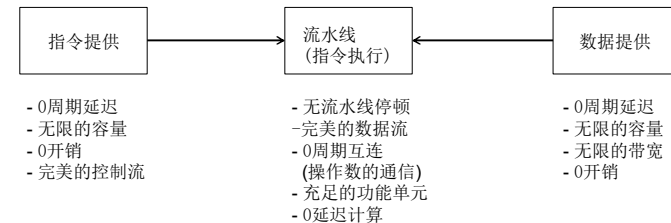
分层的存储体系结构

23

理想化

到目前为止，我们想象

- 程序看到连续的4GB内存
- 在一个处理周期中可以访问内存的任意位置



4.1. Ideally one would desire an indefinitely large memory capacity such that any particular aggregate of 40 binary digits, word (cf. 2.3), would be immediately available—i.e. in a tin

--- Burks, Goldstein, von Neumann, 1946

24

24

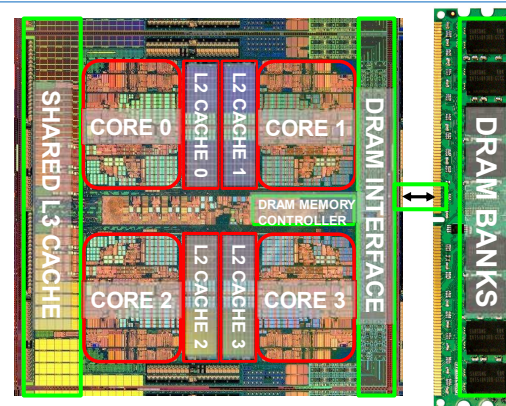
真实的世界

- 无法负担也不需要像用户地址空间那样大的内存(想想64位的ISAs)
- 大多数机器在若干程序之间是“多任务”执行的
- 找不到既能支持千兆字节（GB），又能在千兆赫（GHz）主频下使用的存储技术
 - “魔法”内存
- “魔法”内存仍然是非常接近现实的“可用”抽象，因为：
 - 分层存储:又大又快
 - 虚拟内存:连续且私有

25

25

现代系统中的存储



26

26

理想存储器

- 0访问时间(延迟)
- 无限的容量
- 0开销
- 无限的带宽 (支持多路并行访存)

27

27

问题

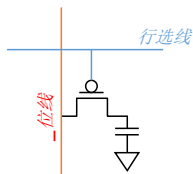
- 理想存储器的需求之间互相制约
- 越大则越慢
 - 越大 → 确定位置花费的时间越长
- 越快则越贵
 - 存储器技术: SRAM vs. DRAM
- 带宽越高越贵
 - 需要更多的bank, 更多的端口, 更高的频率, 或更快的技术

28

28

存储器技术: DRAM

- 动态随机存取存储器(Dynamic random access memory)
- 电容器充电状态表示了存储的值
 - 电容器充电或者未充电代表存储1或0
 - 1 个电容器
 - 1 个存取晶体管
- 电容器向行选线方向漏电
 - DRAM单元随时间的推移损失电荷
 - DRAM单元需要刷新

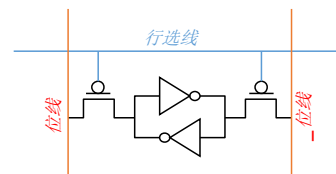


29

29

存储器技术: SRAM

- 静态随机存取存储器(Static random access memory)
- 两个交叉耦合的反相器存储1bit
 - 反馈路径使被存储的值保持在SRAM单元中
 - 4个晶体管用于存储
 - 2个晶体管用于存取



30

30

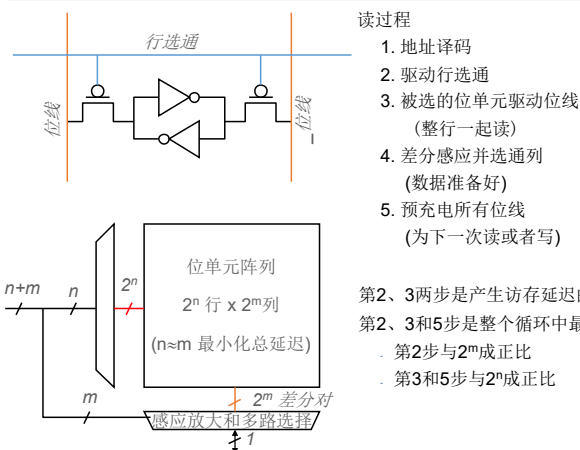
存储器Bank的组织 and 操作

-
- 读访问过程:
 1. 译码行地址并驱动字线
 - 读整行
 2. 选择位驱动位线
 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

31

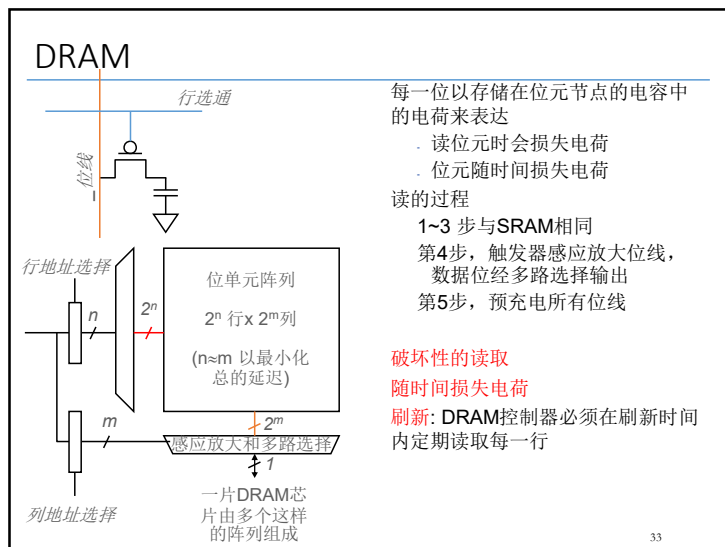
31

SRAM



32

32



33

DRAM vs. SRAM

- DRAM
 - 存取更慢 (电容)
 - 密度更高 (每单元1个晶体管、1个电容)
 - 成本更低
 - 需要刷新 (功耗, 性能, 电路)
 - 需要把电容器和逻辑电路加工到一起
- SRAM
 - 存取更快 (无需电容)
 - 密度更低 (每单元6个晶体管)
 - 成本更高
 - 不需要刷新
 - 与加工逻辑电路过程一致 (没有电容)

34

问题

- 越大越慢
 - SRAM, 512 B, 亚纳秒
 - SRAM, KB~MB, ~纳秒
 - DRAM, GB, ~50 纳秒
 - 硬盘, TB, ~10 毫秒
- 越快越贵(\$ 以及芯片面积)
 - SRAM, < 10\$ 每MB
 - DRAM, < 1\$ 每MB
 - 硬盘, < 1\$ 每GB
 - 这些数据随时间变化很快
- 其它技术也在发展
 - 闪存, 相变存储器 (技术还没有成熟)

35

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

36

存储器分层

向这里移动需要用的数据

利用良好的局部性，
存储系统看起来似乎
又大又快

在这里备份
所有的数据

大但是慢

37

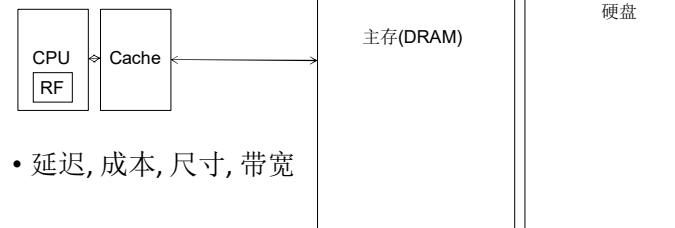
37

存储器分层

• 基本的tradeoff

- 快存储: 小
- 大存储: 慢

• 思路: 存储器分层



• 延迟, 成本, 尺寸, 带宽

38

38

解决方案背后的基本原则——#1

局部性原则

- 最近的过去是不久将来的最佳预测器
- 时间局部性(Temporal Locality): 如果你刚刚做了某事，很可能你很快会再次做同样的事情
 - 你今天教室里，很有可能今后你会不断地定期来教室
 - 反之亦然
- 空间局部性(Spatial Locality): 如果你刚刚做了某事，很可能你会做类似/相关的事情
 - 每次在这个教室里，你可能都坐在同一个座位上(或附近)
 - 你可能坐在同一个人旁边

程序往往比人更容易预测

39

39

解决方案背后的基本原则——#1

内存局部性

- “典型”程序在内存引用中具有很强的局部性
 - 典型的程序通常由“循环”导致
- 时间:
 - 程序倾向于在很短的时间内多次引用(读和写)相同的存储位置
- 空间:
 - 程序倾向于引用一组邻近的内存位置
 - 最值得注意的例子: 1, 指令存储器的引用; 2, 数组/数据结构引用)
- 推论: 一个程序可能在其生命周期中引用大量不同的内存位置，但不是在同一时间

40

40

解决方案背后的基本原则——#2

避免重复运算

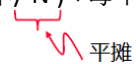
- 如果计算某个东西计算代价很高，那么可以记住一会儿答案，以防不久之后又需要它
- 需要局部性才能有效
- 缺乏局部性
 - 存储大量不同的答案(其中许多答案从未重复使用)
 - 从大量存储的答案中查找答案可能比重计算更贵
- 有局部性
 - 少量答案会一直重复使用!
 - 存储少量频繁用到的答案可以避免大多数重计算

41

41

解决方案背后的基本原则——#3

成本平摊

- 间接成本:构建某些东西的一次性成本
- 每单元成本:操作的每个单元的成本
- 总成本=间接费用+每单元成本 $\times N$
- 平均成本=总成本/ $N = (\text{间接成本}/N) + \text{每单元成本}$

- 如果成本可以平摊到大量的单元上，那么高昂的间接成本通常是可以接受的
⇒ 降低了平均成本

42

42

存储局部性

- 最近的过去是预测不久的将来的最好参考
- 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

43

43

cache的基本要素:利用时间局部性

- 思路: 将最近访问过的数据保存在自动管理的快速存储中 (称为 cache)
- 预期: 这些数据将会很快被再次访问
- 时间局部性原理
 - 最近访问的数据将会在不久的将来被再次访问
 - Maurice Wilkes:
 - Wilkes, “Slave Memories and Dynamic Storage Allocation,” IEEE Trans. On Electronic Computers, 1965.
 - “The use is discussed of a fast core memory of, say 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.”

44

44

cache的基本要素:利用空间局部性

- 思路: 将与最近访问过的数据地址相邻的数据保存到自动管理的快速存储中
 - 逻辑上将存储器划分为大小相等的块
 - 以整块访问的方式向 cache 取数据
- 期待: 附近的数据将很快被访问
- 空间局部性原理
 - 存储器中相邻的数据将会在不久的将来被访问
 - 比如, 顺序的指令存取, 数组的遍历
 - IBM 360/85的实现
 - 16 KB 的 cache, 64B 的数据块
 - Liptay, "Structural aspects of the System/360 Model 85 II: the cache," IBM Systems Journal, 1968.

45

45

以书架类比

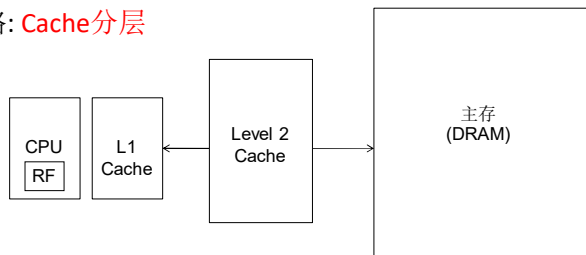
- 手里的书
- 书桌
- 书架
- 家里的储物盒
- 仓库里的储物盒
- 最近使用的书更可能被放在桌上
 - 计算机体系结构相关的书
 - 直到书桌被堆满
- 书架上相邻的书在同一时间段可能都会是需要的
 - 如果书架整理的很好的话

46

46

流水线设计中的cache

- Cache需要和流水线紧密的集成
 - 理想情况下, 只用1个周期存取, 可以使相关的操作不用停顿
- 高频流水线 → 不能让cache太大
 - 但是, 我们又想要一个大的cache 并且 是流水线设计
- 思路: Cache分层



47

47

手动 vs. 自动管理

- 手动: 程序员管理跨层的数据迁移
 - 对于大的程序而言, 程序员会非常痛苦
 - 上世纪50年代的“磁芯” vs “磁鼓” 存储器
 - 仍在某些嵌入式处理器中使用
- 自动: 硬件管理跨层的数据迁移, 对程序员透明
 - ++ 程序员的人生更美好
 - 简单的启发式方法: 将最近使用的内容保存在cache中
 - 一般的程序员不需要了解这一点
 - 不需要知道cache有多大, 也不需要了解它是如何工作的, 就能够写出“正确”的程序! (如果你想要一个“快”的程序呢?)

48

48

分层存储结构中的自动管理

- Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

Slave Memories and Dynamic Storage Allocation

M. V. WILKES

SUMMARY

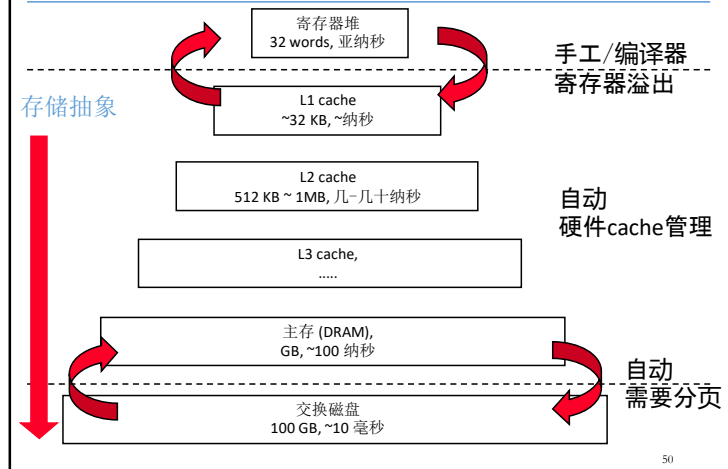
The use is discussed of a fast core memory of, say, 32 000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.

- "By a slave memory I mean one which **automatically accumulates to itself words** that come from a slower main memory, and keeps them available for subsequent use without it being necessary for the penalty of main memory access to be incurred again."

49

49

现代的分层存储体系结构



50

50

分层的延迟分析

- 对于给定的存储层次 i , 它有一个技术上固有的访问时间 t_i , 我们感知到的访问时间 T_i 比 t_i 要长
- 除了最外层, 每当要寻找一个给定地址时
 - 有一定的“命中”机会(命中率 h_i), 访问时间是 t_i
 - 有一定的“缺失”机会(缺失率 m_i), 访问时间是 $t_i + T_{i+1}$
 - $h_i + m_i = 1$
- 因此

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

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

记住, 这里的 h_i 和 m_i 定义的命中率和缺失率针对的是在 L_{i-1} 缺失的引用

51

51

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

52

52

层次设计注意事项

- DRAM
 - 针对容量/美元进行优化
 - 无论容量如何, T_{DRAM} 基本相同
- SRAM
 - 首先针对容量/延迟优化, 再针对容量/美元优化
 - 容量和延迟之间存在不同的折衷可能
$$t_i = O(\sqrt{C_i})$$
- 分层结构弥合了CPU速度和DRAM速度之间的差异
 - $T_{\text{pclk}} \approx T_{\text{DRAM}} \Rightarrow$ 无需分层结构
 - $T_{\text{pclk}} \ll T_{\text{DRAM}} \Rightarrow$ 通过一级或多级SRAM, 在保持成本可控的情况下最小化 T_1

53

53

Intel Pentium 4

- 90nm P4, 3.6 GHz
- L1 D-cache
 - if $m_1=0.1, m_2=0.1$
 $T_1=7.6, T_2=36$
 - $C_1 = 16\text{KB}$
 - $t_1 = 4$ 周期 整型 / 9 周期 浮点 if $m_1=0.01, m_2=0.01$
 $T_1=4.2, T_2=19.8$
- L2 D-cache
 - if $m_1=0.05, m_2=0.01$
 $T_1=5.00, T_2=19.8$
 - $C_2 = 1024\text{ KB}$
 - $t_2 = 18$ 周期 整型 / 18 周期 浮点
- 主存
 - if $m_1=0.01, m_2=0.50$
 $T_1=5.08, T_2=108$
 - $t_3 = \sim 50\text{ns}$ 或 180 周期
- 注意
 - 最好情况的延迟不再是 1 个时钟周期
 - 最坏情况的访问延迟视情况不同可达到 300+ 时钟周期

54

为什么DRAM慢?

- DRAM的制造是超大规模集成电路技术的前沿, 在容量和成本上与摩尔定律同步扩展, 但不是速度
- 1980 ~ 2004年间的DRAM
 - 64K bit \rightarrow 1024M bit (约55%/年指数级)
 - 250ns \rightarrow 50ns (线性)
 - 这是一个非常慎重的选择
 - 如果需要, 我们可以“设计”更快的DRAM
- 内存容量需要随着CPU速度线性增长, 以保持系统平衡——Amdahl的另一定律
- DRAM/处理器速度的差异通过内存分层结构进行协调(L1, L2, L3,)
 - L2在1990年代开始普及
 - L3在2000年初开始普及

55

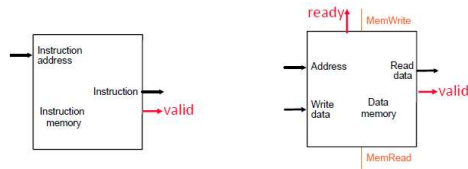
55

Cache 基础和操作

56

高速缓存(Cache)

- 通常, 任何可以通过“记忆”频繁操作的结果, 以避免从头重复执行长延迟操作的结构, 都可以叫做cache, 比如web cache
- 最常见的是, 能够自动管理的基于SRAM的分层存储
 - 将DRAM存储中被最频繁访问的内容记忆在SRAM中以避免重复出现DRAM的访问延迟



57

57

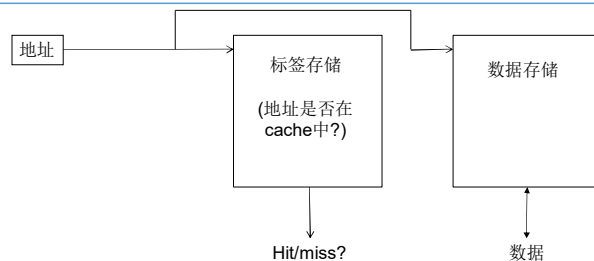
cache基础

- Block (line):** cache中的存储单元
 - 内存在逻辑上按照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 ($C \ll M$)
 - 放置: 在哪儿以及如何在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?

58

58

Cache的抽象和指标



- Cache命中率: $\text{Cache hit rate} = (\# \text{命中}) / (\# \text{命中} + \# \text{缺失}) = (\# \text{命中}) / (\# \text{访问})$
- 平均内存访问时间: Average memory access time (AMAT)

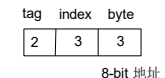
$$= (\text{命中率} * \text{命中延迟}) + (\text{缺失率} * \text{缺失延迟})$$
 减小AMAT对性能有什么影响?

59

59

Cache的块和寻址

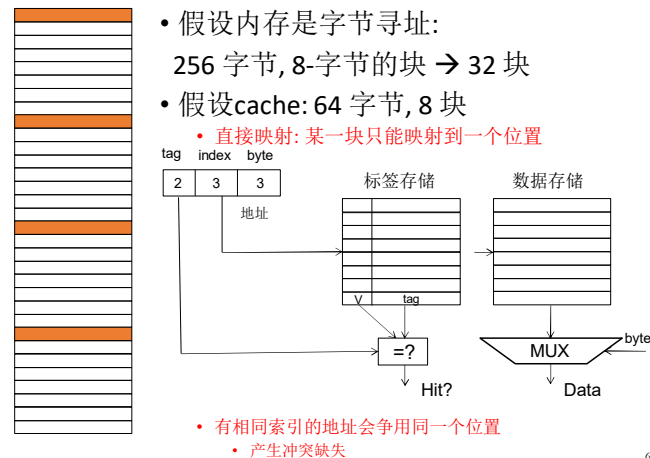
- 内存逻辑上按照cache块划分
- 每一块映射到cache中的某个位置, 由地址中的索引(index)位决定
 - 用来索引标签和数据存储
- Cache的访问: 基于地址中的索引位索引到标签和数据存储, 检查标签存储中的有效位, 比较地址中的标签位和标签存储中存储的标签
- 如果一个块在cache中 (cache命中), 标签存储中就应该有相应块的标签



60

60

直接映射 Cache: 放置和访问



61

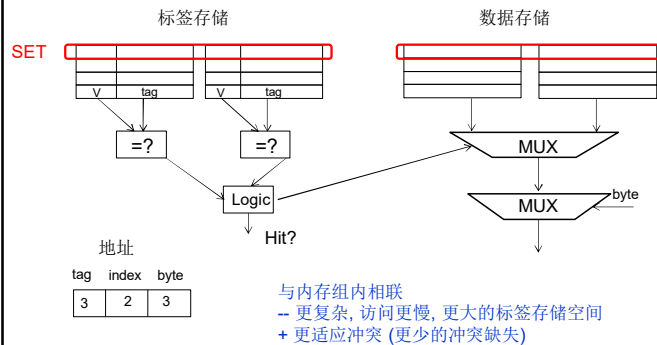
直接映射 Cache

- 直接映射 cache: 当内存中的两个不同块映射到cache中相同的索引上时，这两块不能同时出现在cache中
 - 一个索引 → 一个表项
- 当多个映射到相同索引上的块交替被访问时可能导致0%的命中率
 - 假设地址A和B有相同的索引位和不同的标签位
 - A, B, A, B, A, B, A, B, ... → cache索引冲突
 - 所有的访问都发生冲突缺失(conflict miss)

62

组(set)相联

- 直接映射的cache中，地址0和8总是产生冲突(刚才的例子中)
- 采用两组各4块代替1组8块



63

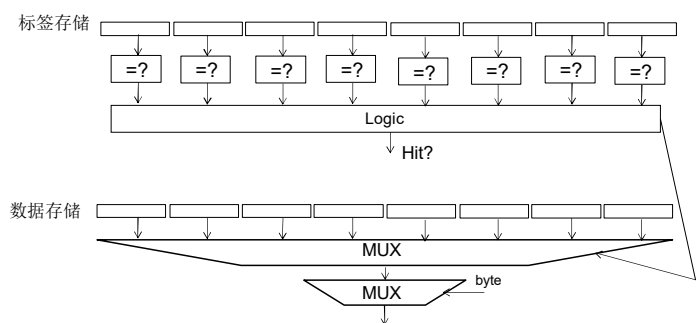
更高的相联度

- 4路:
 - Tag storage (tag, index, byte) and Data storage (MUX).
 - Access flow: Address (tag, index, byte) → Tag storage → Hit? (tag comparison) → Data storage (MUX) → Data.
 - Conflict: Addresses with the same index compete for the same position, leading to conflict misses.
 - 更多的标签比较器, 更大的数据多路选择器, 更大的标签存储
 - 冲突缺失的可能性更低

64

全相联

- 全相联cache
 - 某一块可以放在cache的任何位置

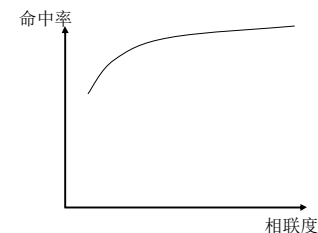


65

65

相联 (折衷)

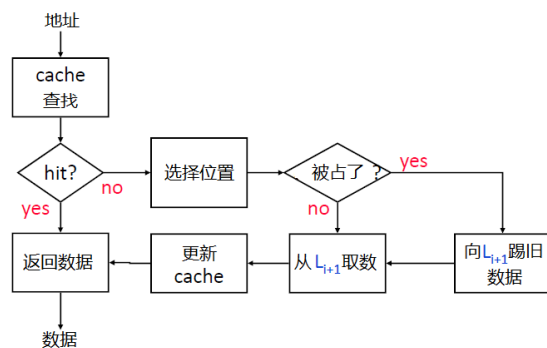
- 多少块可以映射到一个相同的索引(或组)?
- 更高的相联度
 - ++ 更高的命中率
 - 更长的cache访问时间(命中延迟和数据访问延迟)
 - 更昂贵的硬件(更多的比较器)
- 边际收益递减



66

66

基本操作

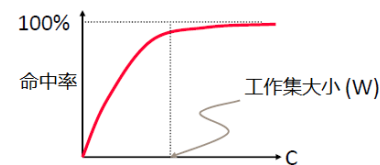


67

67

Cache基本参数

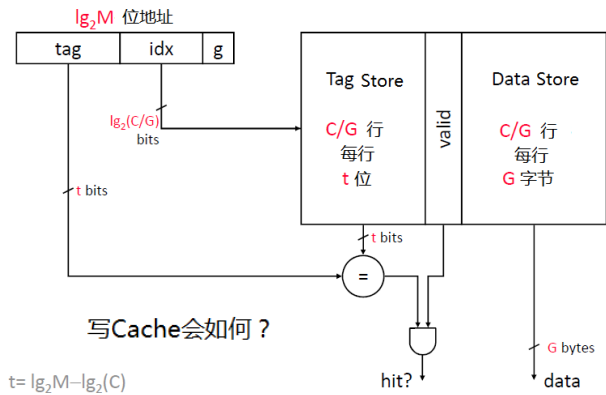
- $M=2^m$, 表示地址空间的大小 (多少byte)
 - 比如: 2^{32} , 2^{64}
- $G=2^g$, 表示Cache访问的粒度大小 (多少byte)
 - 比如: 4, 8
- C , 表示Cache的容量 (多少byte)
 - 比如: 16KByte(L1), 1MByte(L2)
- $B=2^b$, Cache块的大小 (多少byte)
 - 比如: 16(L1), > 64(L2)



68

68

直接映射的Cache (I)



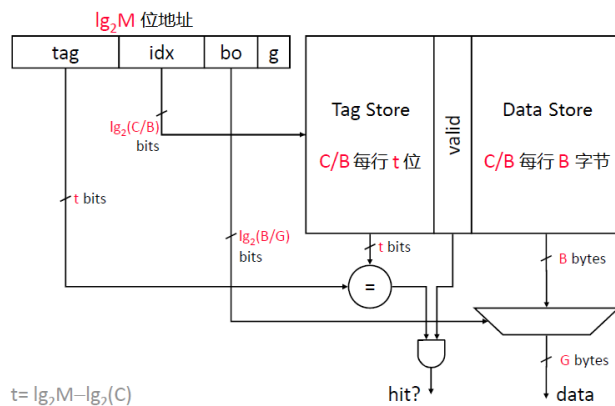
69

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

70

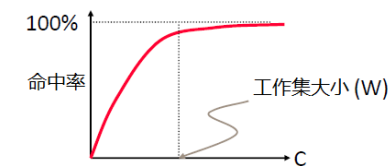
直接映射的Cache (II)



71

直接映射的Cache

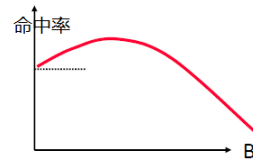
- C字节存储分为C/B块
- 根据地址的块索引域将一块内存映射到一个特定的Cache块
- 所有具有相同块索引域的地址映射到相同的Cache块
 - 2^t 个这样的地址; 一次只能缓存一个这样的块
 - 即使 $C >$ 工作集大小, 也可能产生冲突
 - 给定2个随机的地址, 冲突几率为 $1/(C/B)$
 - 注意, 冲突的可能性随着Cache块数量的增加而降低



72

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

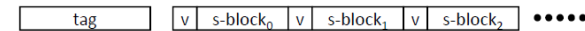


73

73

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

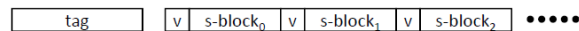


74

74

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



75

75