高等计算机体系结构
第十七讲: 总复习/习题参考答案

栾钟治
北京航空航天大学 计算机学院 中德联合软件研究所
2021-06-18

1

提醒：作业

- 作业6
  - 已截止
  - 预取和并行

2

提醒：实验2-5

- 7月16日截止

3

考试相关

- 6月25日 10:00-12:00, B202

- 卷面100分
- 简答题60分
  - 10题
- 分析题和计算题40分
  - 4题

4

总复习

5

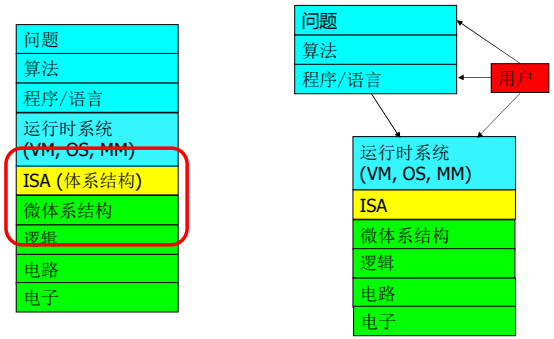
5

什么是计算机体系结构?

6

6

系统层次结构



7

7

什么是计算机体系结构?

- **通过硬件组件的设计、选择、互连以及软硬件接口的设计来创造计算系统的科学与艺术，它使得创造出的计算系统能够满足功能、性能、能耗、成本以及其他特定的目标。**

- 更好的系统
- 更新的应用
- 更好的解决问题
- 理解计算机为什么会这样工作

8

8

ISA vs. 微体系结构

冯诺依曼结构/模型

- 也叫*存储程序计算机*(指令在内存中)，两个关键的属性：
- 存储程序
  - 指令存储在一个线性的存储阵列中
  - 内存统一的存储指令和数据
    - 依靠控制信号实现对存储的值的解释
- 顺序的指令处理
  - 一次处理一条指令 (取指、执行)
  - 程序计数器(指令指针) 标识“当前”指令
  - 程序计数器按顺序推进， 除了控制转移指令

数据流模型(计算机)

- 冯诺依曼模型: 指令的获取和执行按照控制流的顺序
  - 由指令指针来指定
  - 顺序推进除非遇到明确的控制转移指令
- 数据流模型:指令的获取和执行按照数据流的顺序
  - 当操作数准备好
  - 没有指令指针
  - 指令的顺序依赖数据流来确定
    - 每条指令指定结果的接收者
    - 一条指令在获得所有操作数后就可以执行
  - 意味着多条指令可能同时执行
    - 本质上具备更高的并行性

ISA 和 微体系结构的折衷

- 在ISA层面需要做出是数据流还是控制流的抉择，在微体系结构层面也要做出类似的折衷
- ISA: 程序员视角看指令如何执行
  - 程序员看到一个顺序的、控制流驱动的执行序

    vs.
  - 程序员看到一个数据流驱动的执行序
- 微体系结构: 底层实现如何执行指令
  - 微体系结构可以按照任意的序来执行指令，只要它能够按照ISA确定的语义将指令结果呈献给软件即可
    - 程序员应该看到的是ISA确定的序

ISA vs. 微体系结构

- ISA
  - 约定软硬件之间的接口
  - 软件开发者需要了解以便编写及调试系统或用户程序
- 微体系结构
  - 某种ISA的一个特定实现
  - 对软件不可见
- 微处理器
  - **ISA, 微架构**, 电路
  - “Architecture” = ISA + microarchitecture

微体系结构: ISA 在具体设计约束和目标之下的具体实现

13

13

设计要点（Design Point）

- 一组设计时需要考虑的重要问题
  - 将导致包括ISA和微架构方面的tradeoff
- 关注
  - 成本
  - 性能
  - 最大功耗限制
  - 能耗(电池寿命)
  - 可用性
  - 可靠性和正确性
  - 上市时间

- 设计要点由“问题”空间 (应用)或者面向的用户/市场决定

14

14

Tradeoff: 计算机体系结构的灵魂

- ISA层面的折衷
- 微体系结构层面的折衷
- 系统和任务层面的折衷
  - 如何分配软件和硬件应该承担的工作？
- *计算机体系结构是为满足设计点要求做出合适折衷的科学和艺术*

15

15

ISA Tradeoffs

- 复杂指令与简单指令: semantic gap
- 利用“翻译”的方法改变tradeoff策略
- 固定长度与可变长度，统一与非统一译码
- 寄存器个数
- 执行指令之前把复杂指令翻译成“简单”指令有什么好处?
  - 硬件 (Intel, AMD)?
  - 软件 (Transmeta)?
- 哪一种 ISA 更容易扩展: 固定长度 还是 可变长度?
- 如何拥有可变长度、统一译码的 ISA?

16

16

其它有关ISA的折衷

- 有 vs. 无状态码
- VLIW vs. 单指令
- 精确 vs. 非精确异常
- 有vs. 无虚拟存储
- 对齐 vs. 非对齐访问
- 硬件互锁 vs. 软件保证的互锁
- 软件 vs. 硬件管理的页失效处理
- Cache 一致性 (硬件 vs. 软件)
- …

17

17

单周期和多周期微体系结构

18

18

机器如何处理指令

- 处理指令是什么意思?
- 冯诺依曼模型/结构

A = 指令执行之前程序员可见的体系结构状态

处理指令

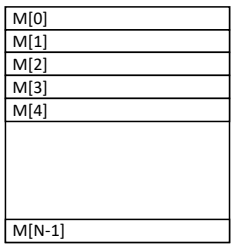
A'= 指令执行之后程序员可见的体系结构状态

- 处理指令: 根据ISA的指令规范将A变换成A'

19

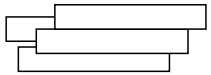
19

程序员可见的(体系结构)状态



内存

用地址索引的存储位置的阵列



寄存器

- 在ISA中会给寄存器命名 (相当于地址)
- 通用和专用寄存器

程序计数器

当前指令的内存地址

指令和程序指定如何转换程序员可见的状态值

20

20

指令处理“周期”

- 指令在“控制单元”的指示下一步一步地处理
- 指令周期：指令处理的步骤序列
- 从根本上说，指令处理大约分为6个阶段：
  - 取指令
  - 译码
  - 计算地址
  - 取操作数
  - 执行
  - 存结果
- 不是所有的指令都需要所有6个阶段

- 指令处理“周期”vs. 机器时钟周期

21

21

观察指令处理的另一个视角

- 指令将数据(AS)转换成数据 (AS')
- 由功能单元完成转换
  - “操作”数据的单元
- 需要有人告诉这些单元对数据做什么操作

- 一个指令处理的引擎由两部分组件构成
  - 数据通路：由处理和转换数据信号的硬件部件组成
    - 操作数据的功能单元
    - 存储数据的存储单元（比如寄存器）
    - 使数据流能够流入功能单元和寄存器的硬件结构（比如连线和多路选择器）
  - 控制逻辑：由决定控制信号的硬件部件组成，这些控制信号决定了数据通路上的部件会如何操作数据
- 有很多方法可以用来设计数据通路和控制逻辑
- 控制信号和结构依赖于数据通路的设计

22

22

“处理指令”的步骤

- ISA 抽象地说明给定一条指令和A，A’应该是什么
  - 定义一个抽象的有限态机
    - 状态 = 程序员可见的状态
    - 次态逻辑 = 指令执行的规范
  - 从 ISA 的视角，指令执行的过程中A和A’之间没有“中间状态”
    - 每条指令对应一个状态转换

- 微体系结构实现 A 向 A’的转换
  - 有很多种实现方式的选择
  - 我们可以加入程序员不可见的状态来优化指令执行的速度：每条指令有多个状态转换
    - 选择 1：A → A’（在一个时钟周期内完成 A 到 A’的转换）
    - 选择 2：A → A+MS1 → A+MS2 → A+MS3 → A’(使用多个时钟周期完成 A 到 A’的转换)

23

23

最基本的指令处理引擎

- 每条指令花费一个时钟周期来执行
- 只用组合逻辑来实现指令的执行
  - *没有中间的、程序员不可见的状态更新*

A = 时钟周期开始时的体系结构状态（程序员可见）

在一个时钟周期内处理指令

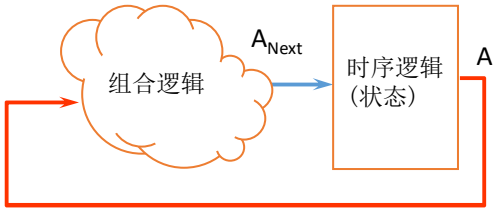
A’ = 时钟周期结束时的体系结构状态(程序员可见)

24

24

最基本的指令处理引擎

- 单周期机器



- 时钟周期长度由谁来决定?
- 组合逻辑中的关键路径由谁来决定?

25

25

多周期微体系结构

- 目标：使每一条指令的执行只（大致）花费它该花费的时间

- 思路
  - 时钟周期的决定独立于指令处理时间
  - 每条指令需要花费多少时钟周期
    - 一条指令执行过程中会有多次状态转换
    - 每条指令的状态变换是不同的

26

26

多周期微体系结构

AS = 指令执行之前程序员可见的体系结构状态

↓

第1步：在一个时钟周期内处理一部分指令

↓

第2步：在下一个时钟周期内处理一部分指令

↓

...

↓

AS' =指令执行之后程序员可见的体系结构状态

27

27

性能分析

- 指令执行时间
  - {CPI} x {clock cycle time}
- 程序执行时间
  - 所有指令的[{CPI} x {clock cycle time}]之和
  - {指令数} x {平均 CPI} x {clock cycle time}

- 单周期微体系结构的性能
  - CPI = 1
  - Clock cycle time 长
- 多周期微体系结构的性能
  - CPI = 每条指令不同
    - 平均 CPI → 希望能很小
  - Clock cycle time 短

有两个独立的自由度可以优化

28

28

基本的多周期微体系结构

- 指令执行周期被划分为多个“状态”
  - 指令执行周期的每个阶段可以拥有多个状态
- 多周期微体系结构通过状态到状态的序列处理指令
  - 某个状态下机器的行为由该状态下的控制信号决定
- 整个处理器的行为可以被定义成一个有限状态机
- 在某个状态(时钟周期)中，控制信号控制
  - 数据通路如何处理数据
  - 如何为下一个时钟周期生成控制信号

29

29

是否可以更好?

- 在多周期设计中你看到哪些局限?
- 有限的并发
  - 在指令处理周期的不同阶段，一些硬件资源会闲置
  - 例如，当指令在“译码”或“执行”阶段，“取指”逻辑会闲置
  - 当发生访存时绝大多数数据通路闲置

30

30

流水线微体系结构

31

31

流水线的基本思想

- 系统性更强
  - 多条指令流水线执行
  - 类比：指令的“装配线处理”
- 思路
  - 指令处理周期切分为不同的处理“阶段”
  - 保证有足够的硬件资源在每个阶段处理指令
  - 每个阶段处理不同的指令
    - 指令在连续的阶段中按照程序序连续地处理
- 好处：提升了指令处理的吞吐量（1/CPI）
- 坏处？

32

32

理想的流水线

- 目标：增加少量成本(指令处理的硬件开销)提升吞吐量
- 重复相同的操作
  - 对大量不同的输入执行同样的操作
- 重复独立的操作
  - 重复的操作之间没有相关性
- 统一划分子操作
  - 处理可以被平均地划分成相同延时的子操作(不共享资源)

33

33

并非理想的流水线

■相同的操作... 不是!

⇒ 不同的指令不一定需要所有的阶段

- 迫使不同的指令流经相同的多段流水线

→ 外部碎片 (对于某些指令会有某些流水段闲置)

■统一的子操作 ... 不是!

⇒ 很难平衡不同的流水段

- 不是所有流水段都完成同样的工作量

→ 内部碎片 (有些流水段完成的太快但仍旧需要占用同样的时钟周期时间)

■独立的操作... 不是!

⇒ 指令之间互相不是独立的

- 需要检测和解决指令之间的相关性以确保流水线操作的正确性

→ 流水不是永远流动的 (它会停顿)

34

34

流水线设计中的问题

- 流水段的平衡
  - 需要多少段以及每一段完成什么任务
- 有影响流水的事件时，保持流水线正确、顺畅、满负荷
  - 处理相关性（冒险）
    - 数据
    - 控制
  - 处理资源争用
  - 处理长时延（多个周期）操作
- 处理异常、中断
- 更高的要求：提高流水线的吞吐
  - 使停顿最少

35

35

产生流水线*停顿*的原因

- 资源争用
- 相关性（指令之间）
  - 数据
  - 控制
- 长时延（多个周期）操作

36

36

相关和相关的类型

- 也叫“依赖”或者“冒险”
- 相关性表明了指令之间关于“序”的需求
- 两种类型
  - 数据相关
  - 控制相关
- 资源争用有时也叫资源相关
  - 但是,这种“相关”不是由程序语义表明的基本类型，所以我们把它和上面两种相关区别对待

37

37

处理资源争用

- 当处于两个流水段的指令需要同一个资源时会发生争用
- 解决方案 1：消除争用的起因
  - 复制资源或者提高资源的吞吐能力
    - 例如，将指令存储器（Cache）和数据存储器（Cache）分开
    - 例如，为存储结构设计多个端口
- 解决方案 2：检测资源争用，使其中一个争用流水段停顿
  - 让哪一个流水段停顿?
  - 例如：如果你的寄存器堆分别只有一个读和写端口会怎么样?

38

38

如何处理数据相关

- 反相关和输出相关更容易处理
- 五种处理流相关的基本方法
  - 检测并等待直到值在寄存器堆中可以访问
  - 检测并转发/旁路数据给相关的指令
  - 检测并消除相关性（在软件层面）
    - 不需要硬件检测相关性
  - 预测需要的值，“投机”执行，并且验证
  - 其它(细粒度多线程)
    - 不需要检测

39

39

硬件vs.软件互锁的问题

- 硬件和软件在数据相关处理中各扮演什么角色?
  - 基于软件的互锁
  - 基于硬件的互锁
  - 谁插入/管理流水线气泡?
  - 谁找到独立的指令填充“空闲”的流水线时隙(槽)?
  - 两种方法的优缺点各是什么?

40

40

硬件vs.软件互锁

- 硬件和软件在指令在流水线中执行的过程中发挥了什么作用?
  - 基于软件的互锁→ 静态调度
  - 基于硬件的互锁→ 动态调度
- 基于软件的指令调度→静态调度
  - 编译器对指令排序，硬件按这个序执行
  - 与之形成对比的是动态调度(硬件不按编译器给定的序执行指令)
  - 编译器怎么知道每条指令的延迟?
- 编译器不知道哪些信息使得静态调度很困难?
  - 答案: 所有在运行时（run time）决定的东西
    - 可变的操作延迟, 内存的地址, 分支的方向
- 编译器如何缓解这些困难(如何估计这些未知量)?
  - 答案: Profiling

41

41

控制相关

- 问题: 下一个周期从PC里取出来的是什么?
- 答案: 下一条指令的地址
  - 所有的指令都和他们之前的指令存在控制相关。为什么？
- 如果取到的指令不是一个控制指令:
  - 下一次取的PC是下一条顺序执行的指令
  - 只要我们知道取到的指令尺寸就行了
- 如果取到的指令是控制指令:
  - 我们如何决定下一个要取的PC?
- 实际上，我们怎么知道取的指令是不是一个控制指令?

42

42

分支的类型

类型	取指阶段能判断的分支方向	下一个可能地址的数量?	何时能够解析出下一个取指的地址?
条件分支	不知道	2	执行 (寄存器相关)
无条件分支	总是发生转跳	1	译码 (PC + offset)
调用	总是发生转跳	1	译码 (PC + offset)
返回	总是发生转跳	多	执行 (寄存器相关)
间接分支	总是发生转跳	多	执行 (寄存器相关)

不同类型的分支处理方式不同

43

43

如何处理控制相关

- 关键在于使流水线保持充满正确的动态指令序列
- 当指令是控制指令时可能的解决方案有:
  - 停顿流水线直到得到下一条指令的取指地址
  - 猜测下一条指令的取指地址（分支预测）
  - 采用延迟分支（分支延迟槽/时隙）
  - 其它(细粒度多线程)
  - 消除控制指令（推断执行）
  - 从所有可能的方向取指（如果知道的话）（多路径执行）

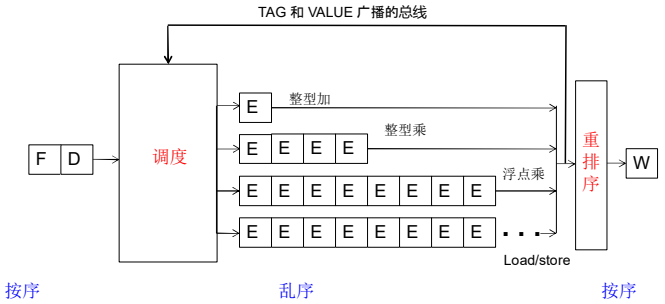
44

44

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
- 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）
- 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写寄存器堆的指令)，前端寄存器堆
- 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 ，同时对比分支旧的指令结果保持状态更新

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

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

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”

寄存器 vs. 存储器

- 寄存器和存储器之间有什么根本的不同?
  - 寄存器的相关是静态可知的 – 存储器的相关是动态决定的
  - 寄存器的状态空间小 – 存储器的状态空间大
  - 寄存器状态对其它线程/处理器不可见 – 存储器状态在线程/处理器之间是共享的 (共享存储多处理器)

49

49

处理存储相关性

- 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

50

50

分层存储体系结构

51

51

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

52

52

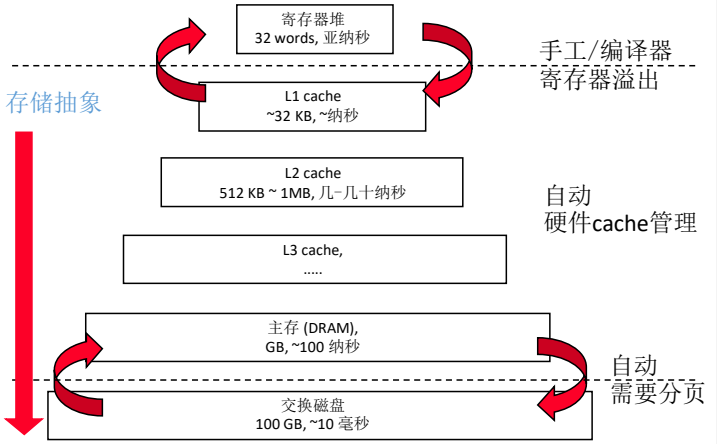
存储局部性

- 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

53

53

现代的分层存储体系结构



54

54

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

55

55

分层存储体系结构
—— Cache

56

56

cache基础

- Block (line): cache中的存储单元
- 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

- 一些重要的cache设计决策
  - 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?

57

57

Cache基本参数

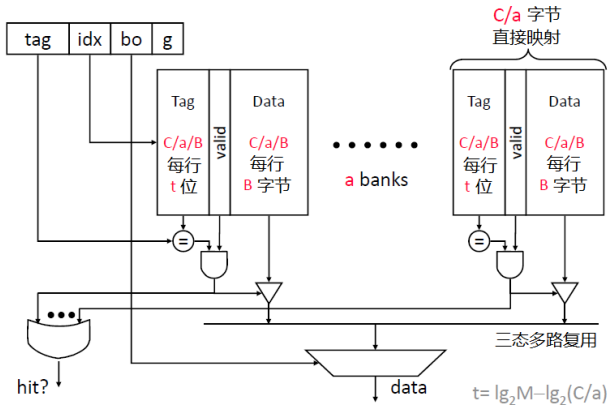
ISA
- M=2^m , 表示地址空间的大小（多少byte）
  - 比如：2^{32} , 2^{64}
- G=2^g , 表示Cache访问的粒度大小（多少byte）
  - 比如：4，8

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

58

58

“a”路组相联——更通用的方案



59

59

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

60

60

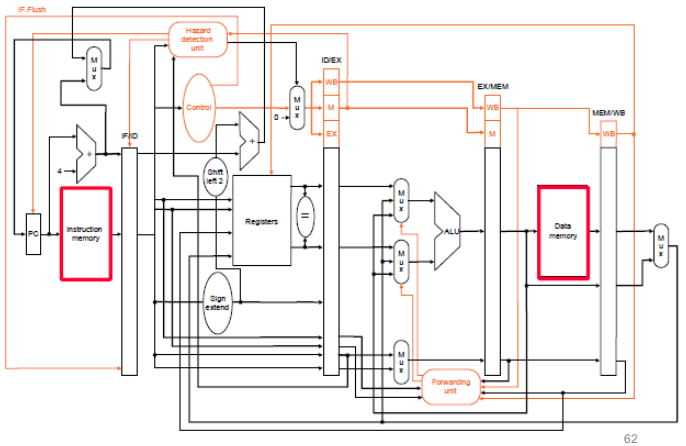
Cache缺失的种类

- 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

61

61

流水线中的Cache



62

62

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

63

63

处理“写”(Store)

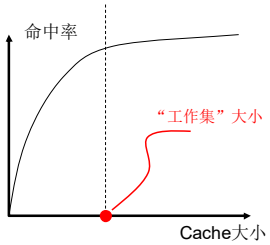
- 写直达: 当写的动作发生时把cache中修改过的数据写到下一级
  - \+ 更简单
  - \+ 所有层都是最新的
    - 一致性: 更简单的cache一致性，因为无需检查低层次的cache
  - -- 更高的带宽需求; 无法进行写合并
- 写回: 当cache块被换出时把cache中修改过的数据写到下一级
  - \+ 可以在换出之前把对同一个块的多个写合并
    - 节省不同级cache之间的带宽，并且节省能耗
  - --需要在标签存储中使用1位标记某块“被修改”
- 写缺失时分配
  - \+ 可以合并写而不是每次单独写下一层cache
  - \+ 更简单，因为写缺失可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)

64

64

Cache参数的影响：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



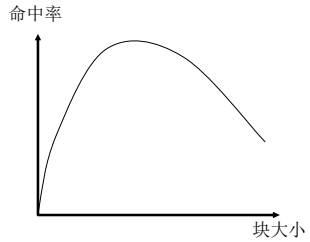
- 工作集: 执行应用时会引用的所有数据的集合
  - 在一个时间段之内

65

65

Cache参数的影响： 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

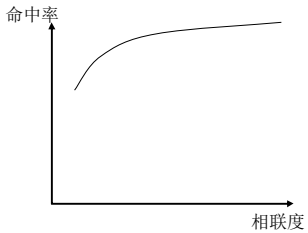


66

66

Cache参数的影响：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



67

67

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

68

68

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

69

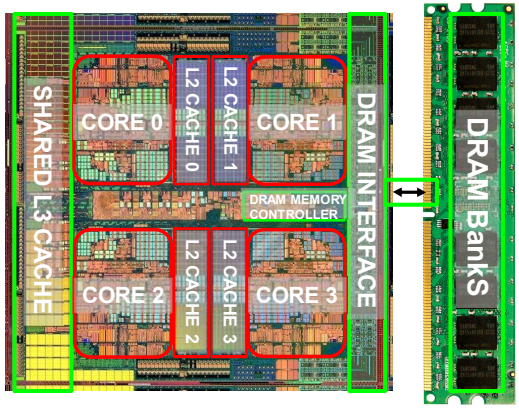
69

分层存储体系结构
—— 主存

70

70

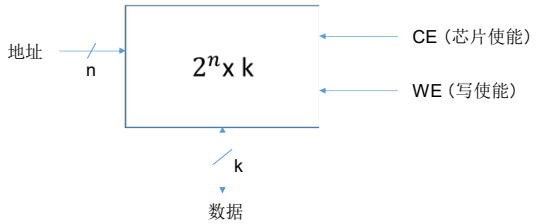
系统中的主存储器



71

71

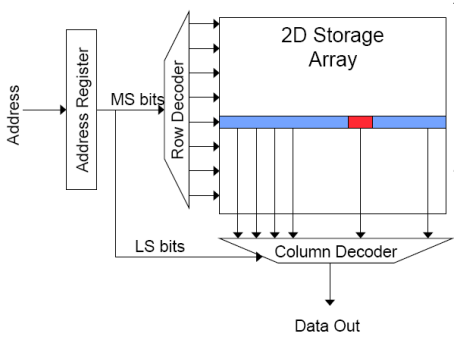
存储芯片/系统的抽象



72

72

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

73

73

交叉存取

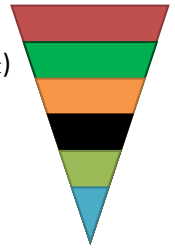
- 问题: 单片的存储阵列访问时间很长，并且无法并行执行多个访存
- 目标: 减小对存储阵列访问的延迟，并且能够并行执行多个访存
- 思路: 将存储阵列划分为多个可以(在同一个周期或连续的周期)独立访问的Bank
  - 每个 Bank 都比整个存储空间小
  - 可以重叠地访问不同的 Bank
- 需要解决的难题: 如何将数据映射到不同的 Bank? (如何在不同的Bank之间交叉存取数据?)

74

74

DRAM 子系统的组织

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



75

75

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)

76

76

DRAM 刷新

- 刷新对性能的影响
  - 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
- 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

77

77

DRAM 控制器

- 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

78

78

DRAM 调度策略

- 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

  调度策略实际上是优先级的序

79

79

多核系统中的内存干扰和调度

- FR-FCFS (行缓冲优先)
  - 最大化DRAM吞吐量
  - 确保向前推进
- 多核环境下进程间干扰不受控导致性能不可预测
  - 感知QoS的内存控制
- 其它处理干扰的方法
  - 目标: 减少/控制干扰
    - 优先级或请求调度
    - 数据映射到Bank/通道/Rank
    - 核/源调节
    - 应用/线程调度

80

80

新型的非易失性存储技术

- 问题: 非易失存储器件一直以来都比DRAM慢很多
- 机遇: 一些新兴的存储技术，非易失而且相对比较快
- 提问: 是否可以采用这些新兴技术来实现主存储器？

- 新兴的电阻式存储器技术
  - PCM(相变存储器)
  - STT-MRAM(自旋转矩磁随机存取存储器)
  - Memristor(忆阻器)

- 基于PCM的主存储器
  - 如何组织：纯，混合
  - 设计上的问题

81

81

访存延迟容忍

- 由长延迟指令导致的停顿
  - 满窗口停顿
  - L2 cache的缺失是导致满窗口停顿的最主要原因

- 如何容忍内存导致的停顿?
  - 两种主要方法
    - 减少/消除停顿
    - 当停顿发生时容忍它的影响
  - 四种基本技术
    - 高速缓存
    - 预取
    - 多线程
    - 乱序执行

有很多技术使这四种基本技术在容忍存储延迟时更加有效

82

82

预取

- 思路: 在程序需要使用之前取数据
- 包括预测哪个地址会是未来需要的
  - 预取时的错误预测不会影响正确性

- 现代系统中, 预取通常在cache块的粒度上实现
  - 预取技术可以减小
    - 缺失率
    - 缺失延迟
  - 预取可以在以下层面实现
    - 硬件
    - 编译器
    - 程序员

83

83

预取的四个问题(I)

- What
  - 预取什么地址
  - 预取精度 = 有用的预取/ 发出的预取
  - 基于过去访问模式的预测
  - 利用编译器关于数据结构的知识
  - 预取算法决定预取什么
- When
  - 何时发起预取请求
  - 数据被预取的时机影响一个预取器的及时性指标
  - 更加的激进: 尽量保持领先处理器访问流的幅度 (硬件)
  - 在代码中更早的发起预取指令 (软件)

84

84

预取的四个问题(II)

- Where
  - 预取的数据放到哪儿
    - 放到cache里
    - 放到独立的预取缓冲中
    - 很多现代系统将预取数据放入cache
  - 预取到哪个级别的cache?
  - 在cache里把预取的数据放到哪儿?
  - 需要调整替换策略以使它能够更优待按需取的块吗?
  - 硬件预取器应该放在分层存储结构的什么位置?
    - 预取器应该看到什么样的访问模式?
- How
  - 软件、硬件、基于执行.....

85

85

预取的四个问题(III)

- 软件预取
  - ISA 提供预取指令
  - 程序员或编译器插入预取指令
  - 通常只对"常规的访问模式"有效
- 硬件预取
  - 硬件监控处理器的存取
  - 记录或者发现模式
  - 自动生成预取地址
- 基于执行的预取器
  - 执行一个"线程"为主程序预取数据
  - 可以通过软件/程序员或者硬件生成

86

86

预取器性能

- 精度 (有用的预取 / 发出的预取)
- 覆盖率 (预取的缺失 / 所有的缺失)
- 及时性 (准时的预取 / 有用的预取)
- 带宽消耗
  - 有/没有预取器时，存储带宽的消耗
  - 好消息: 可以利用空闲时的总线带宽
- Cache污染
  - 由于预取放在cache中导致的额外的按需访问缺失
  - 很难量化，但是会影响性能

87

87

分层存储体系结构
—— 虚拟存储

88

88

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）
- 通过地址翻译
  - 实现大的、私有的、统一的抽象
  - 控制进程可以引用哪些物理位置，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

89

89

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
- 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
  - 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
- 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

90

90

访问保护

- 表项中的保护位表征访问权限
- 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）

91

91

请求页面调度（缺页中断）

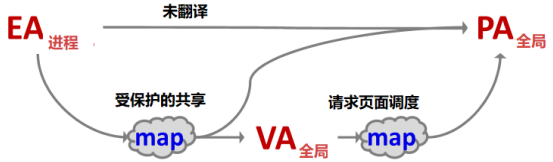
- 使用主存和“交换”磁盘作为*自动管理*的内存层级
  类似于缓存和主存
- 和Cache一样的基本问题
  (1)在DRAM的什么位置“缓存”页面?
  (2)如何在DRAM中找到一个页面?
  (3)什么时候把一页放进DRAM?
  (4)将哪个页面从DRAM置换到磁盘，释放DRAM用于新页面?
- 关键概念差异：交换vs.缓存
  - DRAM不保存磁盘上内容的副本
  - 一页既在DRAM中也在磁盘上
  - 地址始终没有绑定到一个位置
- 规模和时间尺度的差异导致完全不同的实现选择

92

92

虚拟存储和“虚存”

- 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)



93

93

页表的大小

- 页表保存着从虚拟页号到物理页号的映射
- 每个进程有一个页表，页表可能很大

- 不需要跟踪整个虚拟地址空间
- 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

今天主要有两种使用模式: 分层页表和哈希页表

94

94

快表-TLB

- 用户的每次访存都需要翻译
- 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
    - C：L1 指令TLB应覆盖与L1 cache相同的空间
    - B：访问一页后，访问下一页的可能性有多大？
    - a：相联度最小化冲突？
  - TLB的表项：

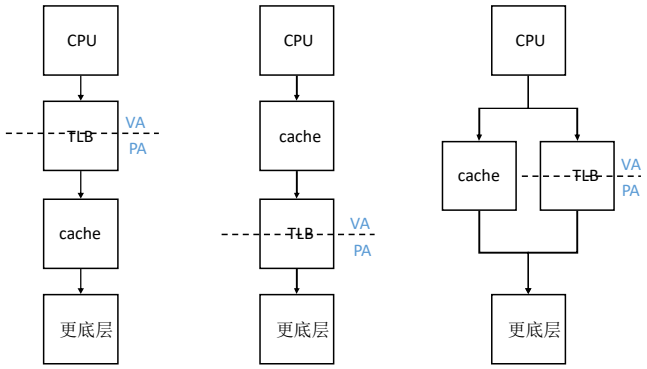
标签：地址tag(来自VA), ASID
页表项(PTE)：PPN和保护位
其它：有效位、脏位等

95

95

虚拟存储与cache的交互

- 什么时候需要做地址翻译?
  - 换句话说, cache是虚拟编址还是物理编址?



96

96

虚拟-索引,物理-标签

- 如果 Cache ≤ (页大小 × 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
- 如果 Cache > (页大小 × 相联度), cache索引位将包含VPN ⇒ “同义词”会引发问题

- 同音异义词：相同的声音不同的含义
  - 同一个虚拟地址可能映射到两个不同的物理地址
- 同义词：不同的声音相同的含义
  - 不同的虚拟地址可能映射到同一个物理地址

97

97

解决“同义词”问题

- 限制cache大小 (页大小x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

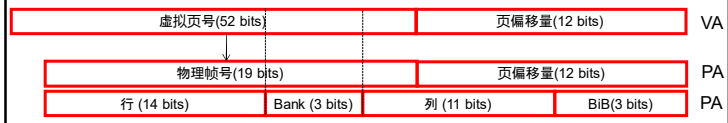
- 在操作系统中限制页的放置
  - 确保虚拟地址的索引 = 物理地址的索引
  - 称为页着色
  - SPARC

98

98

虚拟存储与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

99

99

虚拟存储和DMA的交互

- VA中连续的块
  - 在PA中不保证连续
  - 可能根本不在内存中
- 软件解决方案
  - 在DMA之前，内核从用户缓冲区复制到固定的连续缓冲区，或者
  - 用户为零拷贝DMA分配特殊的固定连续页
- 更智能的DMA引擎可依照一个命令“链表”移动不连续块
- 虚拟编址I/O总线(带I/O MMU)

100

100

并行和多核
——并行基础

101

101

Flynn的分类

- SISD: 单指令操作单个数据元素
- SIMD: 单指令操作多个数据元素
  - 阵列处理机
  - 向量处理器
- MISD: 多指令操作单个数据元素
  - 最接近的形式: 脉动阵列处理器, 流处理器
- MIMD: 多指令操作多个数据元素 (多指令流)
  - 多处理器
  - 多线程处理器

102

102

Amdahl定律

$$加速比_{P个处理器} = \frac{\tau_1}{\tau_p} = \frac{1}{\frac{\alpha}{p} + (1-\alpha)}$$

$$加速比_{P\to\infty} = \frac{1}{1-\alpha}$$

并行加速比的瓶颈

- 最大化加速比受限于串行部分：串行瓶颈
- 并行部分通常也不是完美的并行
  - 同步开销（比如，更新共享的数据）
  - 负载不均衡开销（并行化不完美）
  - 资源共享开销（N个处理器之间的竞争）

103

103

并行部分的瓶颈

- 同步: 对共享数据的操作不能并行
  - 锁, 同步互斥, barrier同步
  - 通信: 任务之间可能需要互相的数据
  - 竞争共享数据时会造成线程串行
- 负载不均衡: 并行的任务可能有不同的长度
  - 由于并行化不理想或者微体系结构的影响
  - 在并行部分降低加速比
- 资源竞争: 并行任务会共享硬件资源, 互相延迟
  - 为所有资源设计冗余 (比如内存) 成本太高
  - 每个任务单独运行时并没有额外的延迟产生

104

104

紧耦合多处理器的主要难点

- 共享存储同步
  - 锁, 原子操作
- Cache 一致性
- 访存操作的序
  - 程序员希望硬件提供什么?
- 资源共享, 竞争和分区
- 通信: 互连网络
- 负载不均衡

105

105

MIMD 处理器中访存的序

- 每个处理器的访存操作按照在该处理器上运行的"线程"的顺序执行序(假设每个处理器遵循冯诺依曼模型)
- 多处理器并发执行访存操作
- 存储器看到来自所有处理器的访问的序是什么样的?
  - 换句话说, 跨不同处理器的操作的序是什么样的?

106

106

保护共享数据

- 线程不能够并发地更新共享数据
  - 为了正确性
- 对共享数据的访问被封装在*临界区*或者通过*同步机制(锁, 信号量, 条件变量)*
- 只能有一个线程在某个确定的时间执行临界区
  - 同步互斥原则
- 多处理器需要提供同步原语的*正确*执行以确保程序员能够保护共享数据

107

107

顺序一致性

- 一个多处理器系统是顺序一致的，如果:
- 所有操作的结果都是相同的，就好像所有处理器的操作都按照某种顺序序执行

并且

- 每个处理器操作所呈现出的序是按照程序所指定的序
- 这是一个访存执行序模型，或者说是一个内存模型
- 由ISA指明

108

108

更弱的内存一致性

- 优点
  - 不需要保证访存操作非常严格的序
    → 使得一些性能提升技术的硬件实现更简单
    → 可以比严格的序性能更高
- 缺点
  - 程序员(或者软件)的负担更重(需要保证“barrier” 正确)
- 程序员-微架构折衷的又一个例子

109

109

Cache 一致性解决方案

- 完全不依靠硬件的一致性
  - 保持cache一致性是软件的责任
  - \+ 让微架构更简单
  - -- 使程序员的工作更困难
    - 需要考虑硬件cache以保持程序的正确性?
  - -- 软件中保证一致性会带来额外开销
- 所有的处理器共享所有的cache
  - \+ 不需要一致性
  - -- 共享的cache成为瓶颈
  - -- 这种方案下极难设计即具备低延迟cache又有可扩展性的系统

110

110

保持一致性

- 需要保证所有处理器看到相同存储位置的值的一致性(一致更新)
- 一致性需要提供:
  - **写的传播**: 保证更新被传播出去
  - **写的序列化**: 为所有处理器提供一致的全局序
- 需要一个全局的序列化点对写排序
- 硬件 Cache 一致性的基本思路
  - 一个处理器/cache向所有其它处理器广播它对某个内存位置的写/更新
  - 另一个拥有这个位置的数据的cache要么更新要么置无效它的本地拷贝

111

111

更新 vs. 置无效的Tradeoffs

- 目的是什么?
  - 写的频率和共享行为都是至关重要的
- **更新**
  - \+ 如果共享者集合是常数并且更新操作不频繁，可以避免被置无效数据重新获取的开销(广播更新模式)
  - – 如果其它核重写的数据并没有被读取，更新就是无用的
  - – 写直达cache策略 → 总线成为瓶颈
- **置无效**
  - \+ 置无效广播后, 处理器核对数据有独占访问权
  - \+ 只有在每个写之后会持续读的核会保有一份拷贝
  - – **如果写竞争度很高**, 会导致ping-pong 效应(快速的互相置无效/重取)

112

112

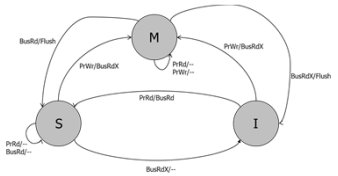
两种cache一致性方法

- 如何确保合适的cache被更新?
- **监听总线(Snoopy Bus)**
  - 基于总线, 所有请求在单点序列化
  - 处理器观察其它处理器的动作
    - 比如: P1 在总线上发出对A的 “排他读”请求, P0 看到后将自己的A的拷贝置为无效
- **目录(Directory)**
  - *每个块*单点序列化, 序列化点分布在各节点
  - 处理器生成对块的显式请求
  - 目录追踪每个块的所有权 (共享者集合)
  - 目录协调置无效操作
    - 比如: P1 向目录请求排他的拷贝, 目录要求 P0 置无效相关数据, 等待应答, 然后向 P1 响应请求

113

基于监听的cache一致性协议

- 简单的监听cache一致性协议
  - 两个状态：有效、无效
- 更复杂一些的一致性协议MSI
  - **M**(odified): cache行是唯一拷贝并且被修改过(dirty)，写缺失导致一个*排他读*请求, 状态迁移到**M**
  - **S**(hared): cache行是一系列拷贝中的一个，读缺失导致总线上一个*读*请求, 状态迁移到**S**
  - **I**(nvalid): 无效，当一个处理器监听到从另一个处理器发出的*排他读*请求, 它必须置无效自己的拷贝(如果有的话)



114

MSI协议的问题

- 一开始任意一个块都不在cache中
- 问题: 当读一个块时, 这个块立即变为S状态，即使它可能只是cache中仅有的一个拷贝
- 为什么这是一个问题?
  - 设想一下，读这个块的cache要在某个时刻写它
  - 即使它拥有的是唯一拷贝，也需要向总线广播“置无效”!
  - 如果cache知道系统中只有它有这份拷贝, 它在写块的时候无需通知其它cache → 省去不必要的广播

115

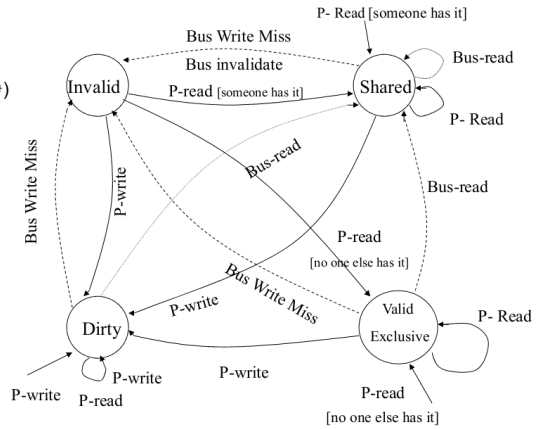
解决方案: MESI

- 思路: 增加一个状态表示cache中仅有的拷贝，并且是干净的
  - *独占(Exclusive)*状态
- 如果总线读(*BusRd*)时，没有其它cache有拷贝，块就会被置为*独占(E)*状态
- 写块是可能会导致状态从*独占(E)*→ *修改(M)* 的变换!
- MESI 也称为 *Illinois 协议(模式)* [Papamarcos and Patel, ISCA 1984]

116

MESI(Illinois 模式)

四种状态：
M(独占拷贝，脏)
E(独占拷贝，干净)
S(共享，干净)
I (无效)



117

117

MESI的问题

- 共享(S)状态需要数据是干净的
  - 即, 所有拥有这个块的cache必须都拥有最新的拷贝, 并且跟内存中的一致
- 问题: 当有总线读时, 如果块处于修改(M)状态需要将块写回内存
- 为什么这是个问题?
  - 内存可能做不必要的更新 → 其它处理器可能会在cache之后写块

118

118

改进MESI

- 思路 1: 总线读时不做M→S的状态转换, 将拷贝置为无效, 直接把修改过的块发给请求的处理器而不更新内存
- 思路 2: 做 M→S的转换, 但是指定一个cache作为拥有者 (O), 它负责在块被移除的时候写回
  - 这时候 “共享(S)” 意味着 “共享的并且可能是脏的”
  - 这是MOESI 协议的一个版本

119

119

复杂Cache一致性协议中的tradeoff

■协议可以通过更多的状态和预测机制优化
+ 减少不必要的置无效和块的传输

■当然, 更多的状态和优化
-- 设计和验证更困难(导致更多需要考虑的情况和条件)
-- 收益递减

120

120

并行和多核
——互连网络

121

121

互连网络基本概念

- 拓扑
  - 指明组件连接的方式
  - 影响路由、可靠性、吞吐量、延迟
- 路由 (算法)
  - 消息如何从源到目的
  - 静态还是自适应
- 缓冲和流量控制
  - 在互连网络中存储些什么?
    - 完整的包, 部分包, 其它?
  - 如何在过载时进行调节?
  - 与路由策略紧耦合

122

122

拓扑

- 总线 (最简单)
- 点到点互连 (理想方式、成本最高)
- 交叉开关 (Crossbar)
- 环
- 树
- Omega
- 超立方
- 网状网(Mesh)
- Torus
- 蝶形
- …

123

123

路由算法

- 类型
  - **确定的**: 总是为一个源-目的对之间的通信选择同样的路径
  - **健忘的**: 选择不同的路径, 不考虑网络状态
  - **适应的**: 可以选择不同的路径, 适应网络的状态
- 如何适应
  - 局部/全局反馈
  - 最小或非最小路径

124

124

确定性路由

- 相同(源, 目的)对的包走同样的路径

- 维度序路由
  - 比如, XY 路由(Cray T3D, 其它很多片上网络)
  - 先遍历维度 X, 再遍历维度 Y

+ 简单
+ 无死锁(资源分配不需要时钟周期)
- 可能导致高度竞争
- 没有利用路径多样性

125

125

处理死锁

- 避免路由中的环
  - 维度序路由
    - 不会产生循环依赖
  - 限制每个包的“轮次”

- 通过增加缓冲避免死锁(逃生路径)

- 检测并突破死锁
  - 抢占缓冲区

126

126

健忘性路由: 勇士算法

- 健忘性算法的例子
- 目标: 均衡网络负载
- 思路: 随机选择一个中间目的节点, 首先路由到该节点, 接着从该节点路由到最终目的
  - 源-中间节点和中间节点-目的, 可以使用维度序路由

+ 随机的/均衡网络负载
- 非最小(包延迟可能增加)

- 优化:
  - 在高负载时使用
  - 限制中间节点

127

127

适应性路由

- 最小适应性
  - 路由器根据网络状态(比如, 下游缓冲区的占用情况)来选择高效输出端口发送包
  - 高效输出端口: 能使包离目的更近的端口
  + 能感知局部拥塞
  - 追求最小性限制了高连接利用率的获得(负载均衡)

- 非最小(完全) 适应性
  - 根据网络状态将包“错误路由”到非高效输出端口
  + 能够获得更好的网络利用率和负载均衡
  - 需要保证避免活锁

128

128

并行和多核
——片上多核

129

129

对于片上多核

- 我们想要:
  - 当我们在N个核上并行化一个应用，我们能获得N倍在单个核上的性能
- 我们能够得到:
  - Amdahl定律 (串行瓶颈)
  - 并行部分的瓶颈

130

130

问题: 串行化的代码段

- 很多并行程序无法完全并行化
- 串行化代码段的产生
  - 连续的部分(Amdahl的“串行部分”)
  - 临界区
  - 栅障
  - 流水化程序中的受限阶段
- 串行化的代码段
  - 降低性能
  - 限制扩展性
  - 浪费能源

131

131

不同代码段的需求

- 我们想要:
- 串行代码段→ 一个强有力的“大” 核
- 并行代码段→ 许多弱的“小”核
- 这两者互相冲突:
  - 如果你有一个强有力的核，就不可能同时拥有很多核
  - 一个小核在能耗和面积消耗方面都远比一个大核更高效

132

132

两全其美?

- 铺砌大核
  - +单线程、串行代码段时可获得高性能
  - - 并行程序段时吞吐率低
- 铺砌小核
  - + 并行部分吞吐率高
  - - 串行部分、单线程性能低, 相比现有单线程处理器性能还差
- 思路: 在一个芯片上同时集成大核和小核 → 性能不对称

133

133

利用不对称

- 串行部分的执行时间必须短
- 对程序员来说缩短这些串行段相当困难
  - 领域知识不够
  - 硬件平台的多样性
  - 受限的资源
- 目标: 一种不需要程序员参与的缩小串行瓶颈的机制
- 思路: 在非对称多核平台上通过将串行代码段迁移到强有力的核上来加速串行部分的执行

134

134

加速并行瓶颈

- 并行部分中的序列化或不均衡的执行同样可以得益于大核
- 例子:
  - 临界区竞争
  - 比别的阶段执行时间更长的并行阶段
- 思路: 动态判别会导致序列化执行的代码段并将它们放到大核上执行
  - 加速临界区
  - 瓶颈识别和调度

135

135

多线程应用中的瓶颈

一种定义: 任何会被线程竞争的代码段

- Amdahl的串行段
  - 只有一个线程在执行→ 在关键路径上
- 临界区
  - 保证互斥 → 如果有竞争很可能就在关键路径上
- 栅障
  - 再继续推进之前保证所有线程到达该点 → 最后到达的线程在关键路径上
- 流水线阶段
  - 一个循环迭代的不同阶段可能在不同线程上执行, 最慢的阶段会使其它阶段等待 → 在关键路径上

136

136

瓶颈识别与调度

- 主要观点:
  - 线程等待损害并行性并且可能降低性能
  - 代码是导致大多数线程等待的原因 → 可能的关键路径
- 主要思路:
  - 动态识别导致大多数线程等待的瓶颈
  - 加速它们(使用ACMP中的强有力的核)

137

作业1——指令集体系结构（ISA）和折衷参考答案

138

1 指令集体系结构 (ISA)

比较五种不同风格的指令集代码序列的内存效率。不同的体系结构类型有:
1.零地址的机器是一种基于栈的机器，它的所有操作都通过存储在操作数栈上的值进行。允许以下操作: •PUSH M •POP M •OP
2.单地址的机器使用一个累加器来执行计算。允许以下操作: •LOAD M •STORE M •OP M
3.双地址的机器有两个操作数来源，对这两个来源的操作数执行操作并将结果存回其中一个源。允许以下操作：•OP M1, M2
4.三地址的机器，通常有两个操作数来源，执行操作后的结果存回不同于两个操作数来源的第三个目的地址。
(a) 对于一台操作数和结果目的地址都是内存地址的三地址机器，允许如下操作：• OP M3, M1, M2
(b) 对于一台源和目的都是寄存器的三地址机器，使用内存操作将值载入寄存器 (MIPS就是这种机器的例子)。允许如下操作：•OP R3, R1, R2 •LD R1, M •ST R2, M

对以上5种不同类型的指令集，假设
每条指令都有一个操作码和一组操作数，
•所有的操作码均为 1 字节 (8 bits)
•所有的寄存器操作数均为 1字节(8 bits)
•所有的内存地址均为 2字节(16 bits)
•所有的数据操作数均为 4字节(32 bits)
•所有指令的长度均为字节的整数倍
内存带宽消耗=传输的代码量+ 传输的数据量
传输的数据量=涉及的数据数量x 4 Bytes
内存访问没有其他优化，变量初始值都在内存中

（a）将下边的高级语言片段翻译成前述5种结构适用的代码序列
A= B + C;
B= A + C;
D = A - B;
（b）请计算这5种结构对应的指令序列在执行时的取指令字节数和内存数据访问（读和写）字节数。
（c）从代码尺寸的角度哪一种结构最高效?
（d）从内存总带宽的需求（代码+数据）角度哪一种结构最高效?

139

参考答案:

代码的大小：每条指令都有一个操作码和一组操作数，
•所有的操作码均为 1 字节 (8 bits)
•所有的寄存器操作数均为 1字节(8 bits)
•所有的内存地址均为 2字节(16 bits)
•所有的数据操作数均为 4字节(32 bits)
•所有指令的长度均为字节的整数倍
内存带宽：
内存带宽消耗=传输的代码量（代码大小）+ 传输的数据量
传输的数据量=涉及的数据数量x 4 Bytes

代码片段：
A= B + C;
B= A + C;
D = A - B;

以下我们用I-Bytes表示传输的代码量，用D-Bytes表示传输的数据量

(a)和(b)

指令集体系结构	操作码	操作数	I-Bytes	D-Bytes	总字节数
零地址	PUSH	B	3	4	
	PUSH	C	3	4	
基于栈的机器，它的所有操作都通过存储在操作数栈上的值进行。	ADD		1		
	POP	A	3	4	
	PUSH	A	3	4	
	PUSH	C	3	4	
• PUSH M – 将位于内存地址 M 处的值压入操作数栈	ADD		1		
	POP	B	3	4	
• POP M – 弹出操作数栈并将值存入内存地址 M 处	PUSH	A	3	4	
	PUSH	B	3	4	
• OP – 从操作数栈中弹出两个值，对这两个值执行二进制操作 OP，结果压回到操作数栈	SUB		1		
	POP	D	3	4	
			30	36	66

140

参考答案（续）：

代码的大小：每条指令都有一个操作码和一组操作数，
- 所有的操作码均为 1 字节 (8 bits)
- 所有的寄存器操作数均为 1字节(8 bits)
- 所有的内存地址均为 2字节(16 bits)
- 所有的数据操作数均为 4字节(32 bits)
- 所有指令的长度均为字节的整数倍

内存带宽：
内存带宽消耗=传输的代码量（代码大小）+ 传输的数据量
传输的数据量=涉及的数据数量x 4 Bytes

代码片段：
A= B + C;
B= A + C;
D = A - B;

以下我们用I-Bytes表示传输的代码量，用D-Bytes表示传输的数据量

(a)
和
(b)

指令集体系结构	操作码	操作数	I-Bytes	D-Bytes	总字节数
单地址 使用一个累加器来执行计算。 • LOAD M – 将存储在内存地址为 M 处的值载入累加器 • STORE M – 将累加器中的值存入内存地址为 M 处 • OP M – 对内存地址为 M 处存储的值和当前在累加器中的值执行二进制操作 OP，结果存进累加器	LOAD ADD STORE ADD STORE LOAD SUB STORE	B C A C B A B D	3 3 3 3 3 3 3 3	4 4 4 4 4 4 4 4	
			24	32	56

141

参考答案（续）：

代码的大小：每条指令都有一个操作码和一组操作数，
- 所有的操作码均为 1 字节 (8 bits)
- 所有的寄存器操作数均为 1字节(8 bits)
- 所有的内存地址均为 2字节(16 bits)
- 所有的数据操作数均为 4字节(32 bits)
- 所有指令的长度均为字节的整数倍

内存带宽：
内存带宽消耗=传输的代码量（代码大小）+ 传输的数据量
传输的数据量=涉及的数据数量x 4 Bytes

代码片段：
A= B + C;
B= A + C;
D = A - B;

以下我们用I-Bytes表示传输的代码量，用D-Bytes表示传输的数据量

(a)
和
(b)

指令集体系结构	操作码	操作数	I-Bytes	D-Bytes	总字节数
双地址 有两个操作数来源，对这两个来源的操作数执行操作并将结果存回其中一个源。 • OP M1, M2 – 对存储在内存地址为 M1 和 M2 的值进行二进制操作 OP，将结果存回内存地址 M1 处	SUB ADD ADD SUB ADD ADD SUB ADD SUB	A, A A, B A, C B, B B, A B, C D, D D, A D, B	5 5 5 5 5 5 5 5 5	12 12 12 12 12 12 12 12 12	
			45	108	153

142

参考答案（续）：

代码的大小：每条指令都有一个操作码和一组操作数，
- 所有的操作码均为 1 字节 (8 bits)
- 所有的寄存器操作数均为 1字节(8 bits)
- 所有的内存地址均为 2字节(16 bits)
- 所有的数据操作数均为 4字节(32 bits)
- 所有指令的长度均为字节的整数倍

内存带宽：
内存带宽消耗=传输的代码量（代码大小）+ 传输的数据量
传输的数据量=涉及的数据数量x 4 Bytes

代码片段：
A= B + C;
B= A + C;
D = A - B;

以下我们用I-Bytes表示传输的代码量，用D-Bytes表示传输的数据量

(a)
和
(b)

指令集体系结构	操作码	操作数	I-Bytes	D-Bytes	总字节数
三地址 Memory-Memory 一台操作数和结果目的地址都是内存地址的三地址机器 • OP M3, M1, M2 – 对存储在内存地址为M1 和 M2 处的值执行二进制操作 OP，结果存回内存地址为 M3 处	ADD ADD SUB	A, B, C B, A, C D, A, B	7 7 7	12 12 12	
			21	36	57

143

参考答案（续）：

代码的大小：每条指令都有一个操作码和一组操作数，
- 所有的操作码均为 1 字节 (8 bits)
- 所有的寄存器操作数均为 1字节(8 bits)
- 所有的内存地址均为 2字节(16 bits)
- 所有的数据操作数均为 4字节(32 bits)
- 所有指令的长度均为字节的整数倍

内存带宽：
内存带宽消耗=传输的代码量（代码大小）+ 传输的数据量
传输的数据量=涉及的数据数量x 4 Bytes

代码片段：
A= B + C;
B= A + C;
D = A - B;

以下我们用I-Bytes表示传输的代码量，用D-Bytes表示传输的数据量

(a)
和
(b)

指令集体系结构	操作码	操作数	I-Bytes	D-Bytes	总字节数
三地址 Load-Store 一台源和目的都是寄存器的三地址机器，使用内存操作将值载入寄存器 (MIPS 就是这种机器的例子)。 • OP R3, R1, R2 – 对寄存器 R1 和 R2 中的值执行 OP 操作，将结果存回寄存器 R3 • LD R1, M – 将内存地址为 M 处的值取出存入寄存器 R1 • ST R2, M – 将寄存器 R2 中的值存入内存地址 M 处	LD LD ADD ST ADD ST SUB ST	R1, B R2, C R1, R1, R2 R1, A R3, R1, R2 R3, B R3, R1, R3 R3, D	4 4 4 4 4 4 4 4	4 4 4 4 4	
			32	20	52

144

参考答案（续）：

代码的大小：每条指令都有一个操作码和一组操作数，
- 所有的操作码均为 1 字节 (8 bits)
- 所有的寄存器操作数均为 1字节(8 bits)
- 所有的内存地址均为 2字节(16 bits)
- 所有的数据操作数均为 4字节(32 bits)
- 所有指令的长度均为字节的整数倍

内存带宽：
内存带宽消耗=传输的代码量（代码大小）+ 传输的数据量
传输的数据量=涉及的数据数量x 4 Bytes

代码片段：
A= B + C;
B= A + C;
D = A - B;

(c) 从代码尺寸角度看，三地址Memory-Memory的机器最高效，21Bytes的代码

(d) 从内存总带宽的消耗角度，三地址Load-Store的机器最高效，52Bytes的总带宽消耗

145

2 性能指标

请简要回答以下问题。

• 如果在具有更高主频的处理器上运行给定程序，是否意味着相比主频较低的处理器而言它总是能够在单位时间（比如1秒钟）内执行更多的指令？

参考答案：否。主频较低的处理器可能具有更高的 IPC (每周期指令数)

注: 一个主频较低的处理器可能能够在一个周期中执行多条指令，而一个主频较高的处理器可能在一个周期中只能执行一条指令。

•如果一个处理器执行给定程序时每秒钟能够执行更多的指令，是否意味着相比每秒执行指令数较少的处理器而言它总是能够更快地执行完这个程序。

参考答案：否。因为每秒执行指令数较多的处理器可能需要执行更多的指令

注: 运行完一个程序需要执行的指令总数，不同的处理器可能会不同。

146

3 性能评价

评价两个实现不同指令集体系结构的处理器的潜在性能。
评价是基于运行特定基准程序（benchmark）时的性能而做出的。在实现ISA A的处理器上，最优的编译代码执行benchmark的性能是10 IPC，这一款处理器主频是500MHz。在实现ISA B的处理器上，同样最优的编译代码的执行性能是2 IPC，处理器主频是600MHz。

• 请问实现ISA A的处理器每秒可以执行多少百万条指令(MIPS)?

参考答案： ISA A: $10\frac{\text{instructions}}{\text{cycle}} \times 500{,}000{,}000\frac{\text{cycle}}{\text{second}} = 5000$ MIPS

• 请问实现ISA B的处理器每秒可以执行多少百万条指令(MIPS)?

参考答案： ISA B: $2\frac{\text{instructions}}{\text{cycle}} \times 600{,}000{,}000\frac{\text{cycle}}{\text{second}} = 1200$ MIPS

• 哪一个是更高性能的处理器: A? B? 不知道?

参考答案：不知道。对每个处理器来说，最佳的编译代码所具有的指令数量可能是不一样的。

147

4 固定长度和可变长度ISA

考虑以下两种Load/Store结构的ISA

1. 第一种是固定长度的ISA，它使用如下的指令编码。

操作码	操作数 1（目的）	操作数 2（源 1）Reg/Imm	操作数 3（源 2）Reg/Imm

其中：操作码 1 byte，每个操作数均为1 byte；所有寄存器/寄存器以及寄存器/立即数的操作需要1个时钟周期，所有Load和Store操作需要4个时钟周期

2. 第二种是可变长度的ISA，它使用如下的指令编码。

操作码	操作数 1（目的）	操作数 2（源 1）Reg/Imm	操作数 3（源 2）Reg/Imm（可选）

其中：操作码 1 byte，每个操作数均为1 byte，**操作数3是可选的，由操作码隐式说明。**如果指令不需要第3个操作数，这个字段就不会使用。

可变长使得第二种ISA的译码变得复杂，所以，它的所有指令执行时间都比第一种固定长度ISA的指令多2个时钟周期。也就是说，所有寄存器/寄存器以及寄存器/立即数的操作需要3个时钟周期，所有Load和Store操作需要6个时钟周期。

148

操作码	操作数 1（目的）	操作数 2（源 1）Reg/Imm	操作数 3（源 2）Reg/Imm

操作码	操作数 1（目的）	操作数 2（源 1）Reg/Imm	操作数 3（源 2）Reg/Imm（可选）

(a) 对以上两种ISA，这段汇编代码的尺寸（字节数）分别是多少？

```
ADD  r3, r1, r2 // r3 = r1+r2
SLL  r3, 0x2 // r3 = r3 << 2
MOV  r5, 0xa // r5 = 0x0a
STW  r3, (r5) // MEMORY[r5] = r3
```

参考答案：

固定长度ISA： 4x4 = 16 字节

可变长度ISA： 1x4 (ADD指令) +3x3 (其它指令) = 13 字节

(b) 对以上两种ISA，执行这一段代码序列分别需要多少时钟周期？

参考答案：

固定长度 ISA： 3x1 (其它指令) +1x4 (STW 指令) = 7 时钟周期

可变长度 ISA： 3x3 (其它指令) +1x6 (STW 指令) = 15 时钟周期

149

操作码	操作数 1（目的）	操作数 2（源 1）Reg/Imm	操作数 3（源 2）Reg/Imm

操作码	操作数 1（目的）	操作数 2（源 1）Reg/Imm	操作数 3（源 2）Reg/Imm（可选）

(c) 哪种ISA的代码尺寸更小？为什么？

参考答案：

可变长度ISA的代码尺寸更小，因为它使用更少的字节对给定代码段的指令进行编码。

(d) 哪种ISA的执行时间更短？为什么？

参考答案：

固定长度ISA执行时间更短， 因为译码复杂性低，所以译码指令所需的时钟周期更少。

150

5 可寻址性

假如我们有64MB的内存，请计算要获得以下寻址能力所需地址的长度：

(i) 位寻址 ISA

参考答案：29 bits

(ii) 字节寻址 ISA

参考答案：26 bits

(iii) 8字节寻址 ISA

参考答案：23 bits

(iv) 32字节寻址 ISA

参考答案：21 bits

151

6 微体系结构与ISA

(a) 简要叙述微体系结构和ISA之间的区别。编译器需要知道微体系结构的什么信息才能正确的编译程序？

参考答案：

ISA层是一台计算机向软件暴露的接口，而微体系结构则是这台计算机实际的底层实现。因此，微体系结构本身及其各种改变对编译器和程序员是透明的（除了性能方面的影响），而ISA及其改变将影响编译器和程序员。

编译器不需要知道微体系结构的任何信息就能够正确的编译程序。

152

(b) 判别一台机器的以下性质是微体系结构的属性还是ISA的属性:

(i) 这台机器没有减法指令.
(ii) 这台机器的ALU没有减法单元.
(iii) 这台机器没有状态码.
(iv) 在加法指令中可以指明一个5位的立即数.
(v)执行一条加法指令需要n个时钟周期.
(vi) 有8个通用寄存器.
(vii) ALU的一个输入需要一个2选1多路选择器.
(viii) 寄存器堆有1个输入端口和2个输出端口.

参考答案:

(i) ISA
(ii) 微体系结构
(iii) ISA
(iv) ISA
(v) 微体系结构
(vi) ISA
(vii) 微体系结构
(viii) 微体系结构

153

作业2——单周期vs. 多周期微体系结构 参考答案

154

1 MIPS单周期微体系结构分析

1.1

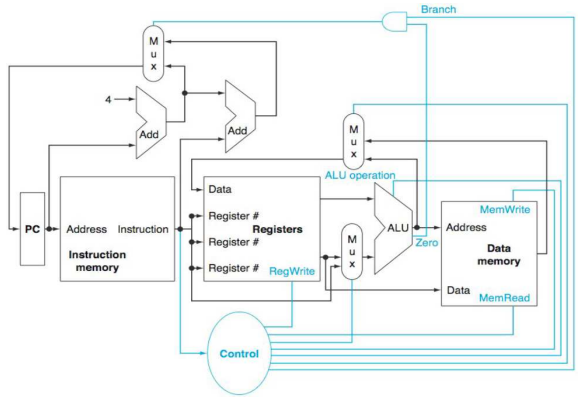
- 不同单元有不同的延迟时间。在图1中有七种主要单元。
- 对一条指令而言，关键路径(产生最长延迟的那条路径)上各个单元的延迟时间决定了该指令的最小延迟。
- 假设个单元的延迟时间如下表所示，回答下列3个问题。

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	控制
400ps	100ps	30ps	120ps	200ps	350ps	100ps

(a) 对一条MIPS的与指令(AND)而言，关键路径是什么？
(b) 对一条MIPS的装载指令(LW)而言，关键路径是什么？
(c) 对一条MIPS的相等则分支指令(BEQ)而言，关键路径是什么？

155

- 图1



156

参考答案

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	控制
400ps	100ps	30ps	120ps	200ps	350ps	100ps

(a) 对一条MIPS的与指令(AND)而言，关键路径是什么？

- 关键路径为：I-Mem、Regs(Read)、Mux、ALU、Mux、Regs(Write)
- 解析：对于AND指令(and rd, rs, rt)，存在这样一条长路径：读指令、读寄存器堆、通过ALUMux多选器、进行ALU运算、通过RegMux多选器、写寄存器堆(即I-Mem、Regs(Read)、Mux、ALU、Mux、Regs(Write))。另外一条长路径与之类似，但是这条路径是在寄存器堆进行读操作时通过控制器的，即：I-Mem、Control、Mux、ALU、Mux、Regs(Write)，但由于控制器的速度快于寄存器堆，因而前者为关键路径，其它的路径都短于这两条路径。

157

参考答案（续）

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	控制
400ps	100ps	30ps	120ps	200ps	350ps	100ps

(b) 对一条MIPS的装载指令(LW)而言，关键路径是什么？

- 关键路径为：I-Mem、Regs(Read)、Mux、ALU、D-Mem(Read)、Mux、Regs(Write)
- 解析：对于LW指令，存在这样一条长路径：读指令、读寄存器堆获得基址、使用多选器选择立即数作为ALU的输入、使用ALU计算地址、访问数据存储器、使用多选器选择存储器输出作为寄存器的数据输入、写寄存器堆，故有路径I-Mem、Regs(Read)、Mux、ALU、D-Mem(Read)、Mux、Regs(Write)。还有一条与之类似的长路径，但是这条路径是通过控制器而不是寄存器堆的(用于生成ALUMux的控制信号)，由于控制器的速度快于寄存器堆，于是前者为关键路径，除这两条之外的路径都是比较短的路径。

158

参考答案（续）

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	控制
400ps	100ps	30ps	120ps	200ps	350ps	100ps

(c) 对一条MIPS的相等则分支指令(BEQ)而言，关键路径是什么？

- 由于控制器的速度快于寄存器堆，故关键路径为：I-Mem、Regs(Read)、Mux、ALU、Mux
- 解析：这条指令有两种长路径——决定分支条件以及计算新PC值。对于决定分支条件的路径，需要读指令、读寄存器堆或使用控制单元、使用ALUMux、使用ALU比较两个值、使用ALU的零输出端来控制选择新PC值的多选器。对于计算新PC值的路径，其中一条是PC值加4(Add)、加偏移量offset(Add)、选择这个值作为新的PC值(Mux)；另一条是读指令(为了取得偏移量)、使用分支加法单元和相应的多选器。但是这两条计算PC值中的路径都比决定分支条件的路径要短，这是因为从表中可以看到指令存储器的速度要慢于执行PC+4的加法器、ALU的速度要慢于分支加法器。

159

1.2

- 图1中基本的单周期MIPS实现仅能实现某些指令。
- 可以在这个指令集中加入新的指令，但决定是否加入取决于给处理器的数据通路和数据通路增加的复杂度。
- 对于下表中的新指令而言，试回答下列3个问题。

指令	解释
add3 Rd,Rs,Rt,Rx	Reg[Rd]=Reg[Rs]+Reg[Rt]+Reg[Rx]

(a) 对上述指令而言，哪些已有的单元还可以被使用？

(b) 对上述指令而言，还需要增加哪些功能单元？

(c) 为了支持这些指令，需要在控制单元增加哪些信号？

160

❖ 当设计者考虑改进处理器数据通路时，往往要考虑性能与成本的折中。假设我们从图1的数据通路出发，其中指令存储器(Instruction Memory)、加法器(Add)、多选器(Mux)、ALU、寄存器堆(Registers)、数据寄存器(Data Memory)和控制单元(Control)的延迟分别为400ps、100ps、30ps、120ps、200ps、350ps和100ps，相应的成本分别为1000、30、10、100、200、2000和500。试根据表中的改进分别回答下列问题。

	改进	延迟	成本	优势
a.	更快的加法器	加法单元-20ps	每个加法单元+20	把已有的加法器用更快的加法器替代
b.	更大的寄存器堆	寄存器堆+100ps	寄存器堆+200	需要更少的load和store指令。这将导致指令数减少5%

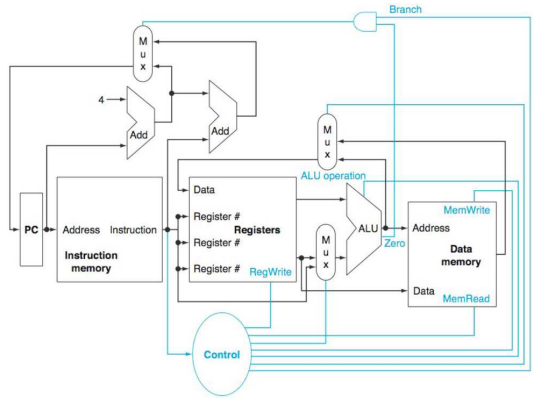
(d) 改进前后的时钟周期分别是多少？

(e) 改进后将获得多大的加速比？

(f) 比较改进前后的性能/价格比，进行这样的改进是否有意义？

161

❖图1



162

参考答案

指令	解释
add3 Rd,Rs,Rt,Rx	Reg[Rd]=Reg[Rs]+Reg[Rt]+Reg[Rx]

(a) 对上述指令而言，哪些已有的单元还可以被使用？

- 除分支加法器、数据存储器之外的所有单元

163

参考答案（续）

指令	解释
add3 Rd,Rs,Rt,Rx	Reg[Rd]=Reg[Rs]+Reg[Rt]+Reg[Rx]

(b) 对上述指令而言，还需要增加哪些功能单元？

- 寄存器堆增加一个输出端 和一个相应的读地址输入端，以读出Rx；增加一个加法器(或为现有的ALU增加一个输入端)，加法器的一个输入端连接至寄存器堆新增的输出端，另一个输入端连接至ALU的输出端。如果采用增加一个加法器的方案，则还需增加寄存器堆数据输入选通器的一个输入端，并连接至加法器的输出端

164

参考答案（续）

指令	解释
add3 Rd,Rs,Rt,Rx	Reg[Rd]=Reg[Rs]+Reg[Rt]+Reg[Rx]

(c) 为了支持这些指令，需要在控制单元增加哪些信号？

- 如果增加一个加法器，则需增加寄存器堆数据输入选通器的控制信号，以实现数据通道3选1；如果为现有的ALU增加一个输入端，则需增加针对三输入端的ALU的功能控制信号定义，使其可控制新增的ADD3操作

165

参考答案（续）

	改进	延迟	成本	优势
a.	更快的加法器	加法单元-20ps	每个加法单元+20	把已有的加法器用更快的加法器替代
b.	更大的寄存器堆	寄存器堆+100ps	寄存器堆+200	需要更少的load和store指令。这将导致指令数减少5%

(d) 改进前后的时钟周期分别是多少？

- a. 由于加法单元不在关键路径上，因此对加法器的改进不影响时钟周期。
- b. 寄存器堆位于关键路径上，因而，使用更大的寄存器堆后，时钟周期变为1330ps + 2 × 100ps = 1530ps。
- 解析：时钟周期是由关键路径决定的，这里的关键路径为：I-Mem(读指令)、Regs(Read)(由于寄存器堆的延迟大于控制器，因而寄存器堆位于关键路径上)、Mux(选择ALU的输入)、ALU、Data Memory(Read)、Mux(选择存储器写入到寄存器堆中的数据)、Regs(Write)(数据写入寄存器堆)，该路径的延迟为400ps + 200ps + 30ps + 120ps + 350ps + 30ps + 200ps = 1330ps。

166

参考答案（续）

	改进	延迟	成本	优势
a.	更快的加法器	加法单元-20ps	每个加法单元+20	把已有的加法器用更快的加法器替代
b.	更大的寄存器堆	寄存器堆+100ps	寄存器堆+200	需要更少的load和store指令。这将导致指令数减少5%

(e) 改进后将获得多大的加速比？

- a.加速比由时钟周期本身的变化以及需要执行的时钟周期数目共同决定，对加法器的改进不影响时钟周期，并且，需要执行的时钟周期数目也不变，因此加速比为1.000
- b.需要的指令数减少5%，需要的时钟周期数目也相应减少5%，同时，时钟周期由1330ps增加为1530ps，因而加速比为 (1/0.95)×(1330/1530)= 0.915

167

参考答案（续）

	改进	延迟	成本	优势
a.	更快的加法器	加法单元-20ps	每个加法单元+20	把已有的加法器用更快的加法器替代
b.	更大的寄存器堆	寄存器堆+100ps	寄存器堆+200	需要更少的load和store指令。这将导致指令数减少5%

(f) 比较改进前后的性能/价格比，进行这样的改进是否有意义？

- a. 原来的处理器的总成本为1000(I-Mem) + 200(Regs) + 500(Control) + 100(ALU) + 2000(D-Mem) + 2×30(2个加法单元) + 3×10(3个多选器) = 3890，更换加法器之后的总成本为3890 + 2 × 20 = 3930，相对成本为3930 / 3890 = 1.010，性能/价格比为1.000 / 1.010 = 0.990，成本增加但性能没有提升。
- b. 使用更大的寄存器堆的成本为3890 + 200 = 4090，相对成本为4090 / 3890 = 1.051，性能/价格比为 0.915 / 1.051 = 0.871，说明用更大的投入反而换来了性能的下降。

168

1.3

❖ 下表给出了实现处理器数据通路的逻辑单元延迟。试根据下表的两种情况分别回答下列问题。

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
400ps	100ps	30ps	120ps	200ps	350ps	20ps	2ps

(a) 如果处理器只需做连续取指这一件事(见图2)，那么时钟周期是多少？

(b) 考虑一个与图3类似的数据通路，但是假设处理器只需处理无条件相对跳转指令，那么时钟周期是多少？

(c) 同样考虑一个与图3类似的数据通路，但这次假设只需处理有条件相对跳转指令，那么时钟周期是多少？(请注意图3中ALU的零输出端不是与数据存储器连接，该输出与选择PC值来源的多选器的控制有关)

➢ 提示：图3中靠右侧的加法器延迟应当按照ALU来计算

169

❖ 根据下表的两种数据通路的逻辑单元，分别回答下列问题。

	单元
a.	执行加4的加法器(对PC)
b.	数据存储器

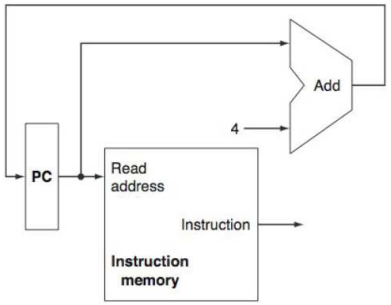
(d) 哪些类型的指令需要该单元？

(e) 对哪些类型的指令而言，该单元位于关键路径上？

(f) 假设仅需支持beq指令和add指令，讨论该单元的延迟变化对处理器时钟周期的影响。假设其他单元的延迟不变。

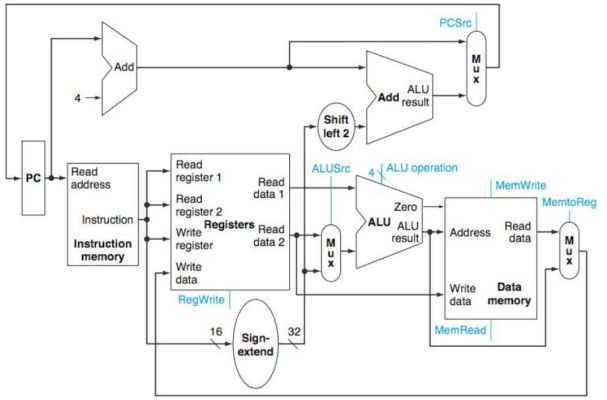
170

❖图2



171

❖图3



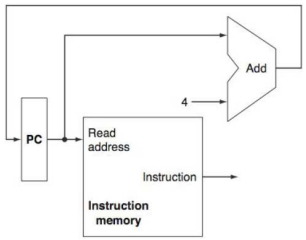
172

参考答案

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
400ps	100ps	30ps	120ps	200ps	350ps	20ps	2ps

(a) 如果处理器只需做连续取指这一件事(见图2)，那么时钟周期是多少？

- 由于指令存储器慢于加法器，因此，时钟周期决定于指令存储器的延迟，时钟周期为400ps。



173

参考答案（续）

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
400ps	100ps	30ps	120ps	200ps	350ps	20ps	2ps

(b) 考虑一个与图3类似的数据通路，但是假设处理器只需处理无条件相对跳转指令，那么时钟周期是多少？

- 关键路径为I-Mem、Sign-extend、Shift-left-2、Add(ALU)、Mux，因此，时钟周期为400ps + 20ps + 2ps + 120ps + 30ps = 572ps。

174

参考答案（续）

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
400ps	100ps	30ps	120ps	200ps	350ps	20ps	2ps

(c) 同样考虑一个与图3类似的数据通路，但这次假设只需处理有条件相对跳转指令，那么时钟周期是多少？(请注意图3中ALU的零输出端不是与数据存储器连接，该输出与选择PC值来源的多选器的控制有关)

- 根据题目中给出的延迟，后者为关键路径，因此时钟周期为400ps + 200ps + 30ps + 120ps + 30ps = 780ps。
- 解析：对于有条件相对跳转指令，除存在长路径I-Mem、Sign-extend、Shift-left-2、Add(ALU)、Mux外，还存在长路径I-Mem、Registers(Read)、Mux、ALU、Mux，关键路径为这两条路径中较长的一个。

175

参考答案（续）

	单元
a.	执行加4的加法器(对PC)
b.	数据存储器

(d) 哪些类型的指令需要该单元？

- a.所有指令。
- b.与存取有关的指令，如：Lw, Sw等。

176

参考答案（续）

	单元
a.	执行加4的加法器(对PC)
b.	数据存储器

(e) 对哪些类型的指令而言，该单元位于关键路径上？

- a.所有指令的关键路径都不会包含这个加法器，因为指令存储器的速度慢于加法器，而所有的指令都必须执行读指令这一操作。
- b.与存取有关的指令，因为只有与存取有关的指令会用到该单元。

177

参考答案（续）

	单元
a.	执行加4的加法器(对PC)
b.	数据存储器

(f) 假设仅需支持beq指令和add指令，讨论该单元的延迟变化对处理器时钟周期的影响。假设其他单元的延迟不变。

- a.beq指令的关键路径为I-Mem、Regs(Read)、Mux、ALU、Mux，延迟为780ps，add指令的关键路径为I-Mem、Regs(Read)、Mux、ALU、Mux、Regs(Write)，延迟为980ps。这里只需讨论该单元延迟变化相比于较长的关键路径的影响，较长的关键路径为add指令，其延迟为980ps，而执行加4的加法器所在的路径为Add、Add(ALU)、Mux，其延迟为100ps + 120ps + 30ps = 250ps，若要该单元延迟变化而导致该路径成为关键路径，从而影响时钟周期，则执行加4的加法器延迟要大于980ps – 150ps = 830ps，才会影响时钟周期。
- b.数据存储器未被beq或add指令使用，因此，它的延迟不影响时钟周期。

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
400ps	100ps	30ps	120ps	200ps	350ps	20ps	2ps

178

1.4

❖ 本题讨论数据通路中不同的单元延迟对整个数据通路时钟周期的影响，以及指令如何利用不同的数据通路单元。根据下面的延迟情况，分别回答下列问题。

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps

(a) 如果仅需支持ALU类指令(如add、and等)，处理器的时钟周期是多少？

(b) 如果仅需支持lw类指令，时钟周期是多少？

(c) 如果必须支持add、beq、lw和sw指令，时钟周期是多少？

179

❖ 假设各类型指令所占比例如下表所示，试根据下表的两种情况分别回答下列问题。

add	addi	not	beq	lw	sw
30%	15%	5%	20%	20%	10%

(d) 数据存储器平均用了多少时钟周期？

(e) 符号扩展电路的输入平均用了多少时钟周期？在未用到该输入的其他时间，符号扩展电路在做什么？

(f) 如果可以将数据通路上某个单元的延迟减少10%，应该减少哪个单元的延迟？改进后整个处理器的加速比是多少？

180

参考答案

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps

(a) 如果仅需支持ALU类指令(如add、and等)，处理器的时钟周期是多少？

- 时钟周期为500ps + 220ps + 100ps + 180ps + 100ps + 220ps = 1320ps
- 解析：关键路径为I-Mem、Registers(Read)、Mux(选择ALU输入)、ALU、Mux(选择寄存器写入端)、Registers(Write)

181

参考答案（续）

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps

(b) 如果仅需支持lw类指令，时钟周期是多少？

- 时钟周期为500ps + 220ps + 100ps + 180ps + 1000ps + 100ps + 220ps = 2320ps
- 解析：关键路径为I-Mem、Registers(Read)、Mux(选择ALU输入)、ALU、D-Mem(Read)、Mux(选择写入寄存器堆的数据)、Registers(Write)

182

参考答案（续）

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位
500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps

(c) 如果必须支持add、beq、lw和sw指令，时钟周期是多少？

- 时钟周期为2320ps
- 解析：lw指令的关键路径最长，相比较而言，sw指令少使用了一个多选器并且不用向寄存器堆写数据，add和beq少使用了数据存储器

183

参考答案（续）

add	addi	not	beq	lw	sw
30%	15%	5%	20%	20%	10%

(d) 数据存储器平均用了多少时钟周期？

- 平均有 20% + 10% = 30% 的时钟周期里，会用到数据存储器
- 解析：只有lw和sw指令会用到数据存储器

184

参考答案（续）

add	addi	not	beq	lw	sw
30%	15%	5%	20%	20%	10%

(e) 符号扩展电路的输入平均用了多少时钟周期？在未用到该输入的其他时间，符号扩展电路在做什么？

- 符号扩展电路实际上在每个周期都有计算结果，但是它的输出在add和not指令中被忽略了，符号扩展电路的输入只在addi指令(提供ALU需要的立即数)、beq指令(提供计算PC需要的偏移量)、lw指令和sw指令(提供寻址过程中需要的偏移量)中是需要的
- 结果为 15% + 20% + 20% + 10% = 65%

185

参考答案（续）

add	addi	not	beq	lw	sw
30%	15%	5%	20%	20%	10%

(f) 如果可以将数据通路上某个单元的延迟减少10%，应该减少哪个单元的延迟？改进后整个处理器的加速比是多少？

- lw指令有最长的关键路径为：I-Mem、Registers(Read)、Mux、D-Mem(Read)、ALU、Mux、Registers(Write)，决定着时钟周期的长度。
- 在路径中，由于指令存储器的延迟值最大，因此，如将它的延迟从400ps减小到360ps(即减少10%)，则时钟周期由1330ps减小到1290ps，加速比为1330/1290=1.031 。

186

1.5

❖ 本题讨论处理器时钟周期与控制单元设计之间的相互影响。根据下表的两种数据通路单元延迟情况分别回答下列问题。

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位	ALU控制
500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps	55ps

(a) 为了避免增加图4中数据通路的关键路径长度，留给控制单元产生MemWrite信号的时间有多少？

(b) 图4中哪个控制信号最不关键，控制单元需要在多长时间内产生该信号以避免其成为关键路径？

(c) 图4中哪个控制信号最关键，控制单元需要在多长时间内产生该信号以避免其成为关键路径？

187

❖ 假设控制单元产生控制信号的时间如下表所示，试根据表中的两种情况回答下列问题(各部件的延迟与前面相同)。

RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
1600ps	1600ps	1400ps	500ps	1400ps	400ps	1500ps	400ps	1700ps

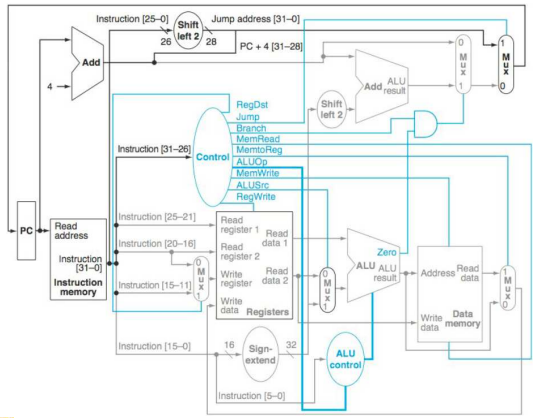
(d) 处理器的时钟周期为多少？

(e) 如果你可以加速控制信号的产生，但加快一个控制信号5ps的代价是处理器成本增加1元。那么为了最大化性能你会加速哪些控制信号？这种性能改进的最小代价是多少？

(f) 如果一个处理器的成本已经很高，那么我们需要在维持处理器性能的同时降低其成本，而不是像第⑤问中所作的那样为提高它的性能而买单。如果你可以使用更慢的逻辑来实现对信号的控制，并且单个控制信号每减慢5ps，处理其成本就可以节省1元，那么在保持处理器性能的同时，你会减慢哪些控制信号，并且减慢多少来降低成本？

188

❖图4



189

参考答案

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位	ALU控制
500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps	55ps

(a) 为了避免增加图4中数据通路的关键路径长度，留给控制单元产生MemWrite信号的时间有多少？

- 关键路径延迟为500ps + 220ps + 100ps + 180ps + 1000ps + 100ps + 220ps = 2320ps，留给控制单元产生MemWrite信号的最长时间为2320ps – 500ps – 1000ps = 820ps
- 解析：假设控制单元的延迟为0，则lw具有最长的关键路径，于是得到关键路径I-Mem、Regs(Read)、Mux、ALU、D-Mem(Read)、Mux、Regs(Write)。控制单元在读取完指令存储器之后才可以产生MemWrite控制信号，并且必须在时钟周期结束之前产生MemWrite信号，但是，由于MemWrite信号是数据存储器的写使能信号，因此在产生该信号之后还应当至少留出写存储器的时间D-Mem(Write)。

190

参考答案（续）

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位	ALU控制
500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps	55ps

(b) 图4中哪个控制信号最不关键，控制单元需要在多长时间内产生该信号以避免其成为关键路径？

- Jump信号具有最长的松弛时间，为2320ps – 500ps – 100ps = 1720ps
- 解析：所有的控制信号都必须在指令读取之后生成，同时一个信号最晚必须在时钟周期结束之前到来，对于MemWrite、RegWrite和Jump信号，由于它们都只在时钟周期的最后才会用到，因而相比于其它控制信号会拥有更长的松弛时间，由于两种情况下均是数据存储器的延迟>寄存器堆>多选器，因而Jump具有最长的松弛时间。
- 这个题目里面没有考虑PC的延迟。

191

参考答案（续）

指令存储器	加法器	多选器	ALU	寄存器堆	数据存储器	符号扩展	左移两位	ALU控制
500ps	150ps	100ps	180ps	220ps	1000ps	90ps	20ps	55ps

(c) 图4中哪个控制信号最关键，控制单元需要在多长时间内产生该信号以避免其成为关键路径？

最关键信号	产生该信号可用的时间
ALUSrc (100ps > 55ps)	220ps

- 解析：为了不影响关键路径，控制信号必须在数据到达之前产生以便不影响数据的通过，最早出现在关键路径上的控制信号是ALUOp和ALUSrc，ALUSrc的松弛时间为Regs(Read)，ALUOp的松弛时间为Regs(Read) + Mux – ALU Ctrl，拥有较小松弛时间的信号更为关键，可见二者对于ALU计算的影响取决于ALU Ctrl与Mux的延迟大小。

192

参考答案（续）

RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
1600ps	1600ps	1400ps	500ps	1400ps	400ps	1500ps	400ps	1700ps

(d) 处理器的时钟周期为多少？

对时钟周期影响最大的控制信号	理想的时钟周期(由第(1)问得来)	实际的时钟周期
MemWrite (+680ps)	2320ps	3000ps

- 解析：为了便于后面的计算，我们首先计算各个控制信号的松弛时间，如下表所示：

RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
1500ps	1720ps	1620ps	500ps	1500ps	265ps	820ps	220ps	1600ps

- 若实际产生信号的时间小于松弛时间，则该信号的产生不会影响时钟周期，反之，该信号会影响时钟周期，并且显然是会使时钟周期变大，由此可以计算出新的时钟周期(下表中的数据为实际时间减去松弛时间)。

RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
100ps	-120ps	-220ps	0ps	-100ps	135ps	680ps	180ps	100ps

193

参考答案（续）

RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
1600ps	1600ps	1400ps	500ps	1400ps	400ps	1500ps	400ps	1700ps

(e) 如果你可以加速控制信号的产生，但加快一个控制信号5ps的代价是处理器成本增加1元。那么为了最大化性能你会加速哪些控制信号？这种性能改进的最小代价是多少？

对时钟周期有影响的信号	代价
RegDst (+100ps)	1195/5=239
ALUOp (+135ps)	
MemWrite (+680ps)	
ALUSrc (+180ps)	
RegWrite (+100ps)	

- 解析：应当只加速影响了时钟周期的控制信号，目标是将这些控制信号对时钟周期的影响至少变为0。

194

参考答案（续）

RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
1600ps	1600ps	1400ps	500ps	1400ps	400ps	1500ps	400ps	1700ps

(f) 如果一个处理器的成本已经很高，那么我们需要在维持处理器性能的同时降低其成本，而不是像第⑤问中所作的那样为提高它的性能而买单。如果你可以使用更慢的逻辑来实现对信号的控制，并且单个控制信号每减慢5ps，处理其成本就可以节省1元，那么在保持处理器性能的同时，你会减慢哪些控制信号，并且减慢多少来降低成本？

RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
100ps	-120ps	-220ps	0ps	-100ps	135ps	680ps	180ps	100ps

- 上表中具有最大正值的信号决定了周期，则在保证性能的前提之下，可以将所有的信号都减慢到具有跟该信号相同的值，于是最小化成本的方法是按照下表减慢响应信号的速度：

RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
580ps	800ps	900ps	680ps	780ps	545ps	0ps	500ps	580ps

195

2 MIPS多周期微体系结构分析

2.1

- 假设在多周期MIPS处理器的下列控制信号中存在固定为0缺陷，那么哪些指令将会失效？为什么？(数据通路参考图1，其中不包含j指令)

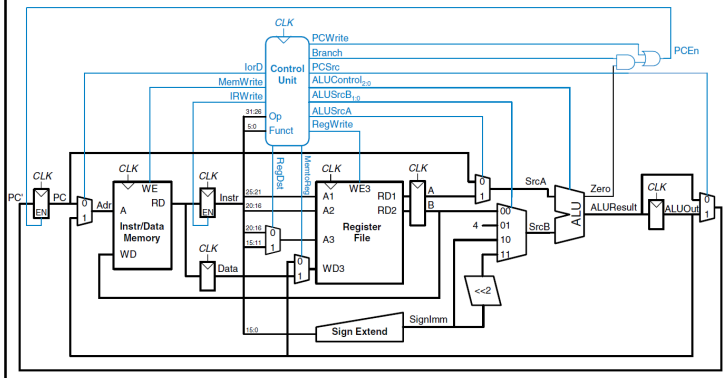
(a) MemtoReg

(b) ALUOp0

(c) PCSrc

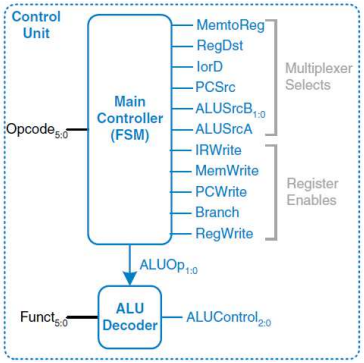
196

❖ 图5



197

❖ 图6. 图5中控制器的内部结构 ($ALUOp_{1:0}$信号)



198

参考答案

(a) MemtoReg

- lw指令将会失效，MemtoReg固定为0将导致无法选择DR作为寄存器堆写数据的来源，而lw指令需要将DR中的数据写入寄存器堆。

(b) ALUOp0

- beq指令将会失效，ALUOp0固定为0将导致ALU无法进行beq需要的减法运算(相应的ALUOp为01)。

(c) PCSrc

- beq指令将会失效，PCSrc固定为0将导致无法选择ALUOut的输出作为PC值的来源，而beq指令需要保存在ALUOut中的ALU的运算结果来修改PC的值。

199

2.2

❖ 假设多周期MIPS处理器各个部件的延迟如下表所示(假设存储器和寄存器堆的写速度与读速度相等，数据通路参考图1)：

Element	Parameter	Delay(ps)
Register clk-to-Q	t_{pcq}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

(a) 通过提高哪个部件的速度(即减小该部件的延迟)可以对整个处理器的速度有最大的提升？

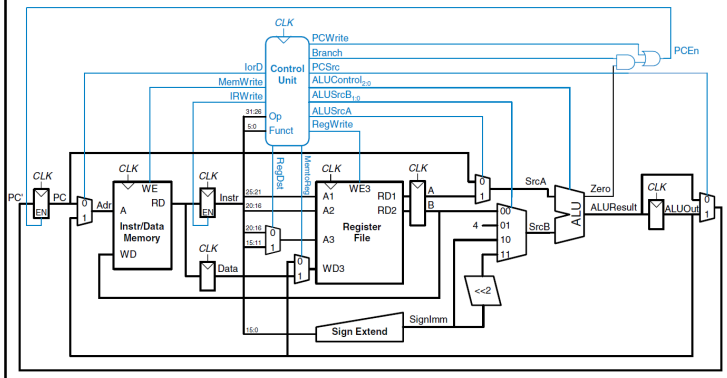
(b) 在避免不必要浪费的前提之下，该部件的延迟应减小到多少？

(c) 提升之后的处理器周期是多少？

(d) 有一种寄存器堆，它比现有的寄存器堆功耗低40%，但是延迟是现有寄存器堆的两倍，请分析一下使用这种寄存器堆是否有意义。

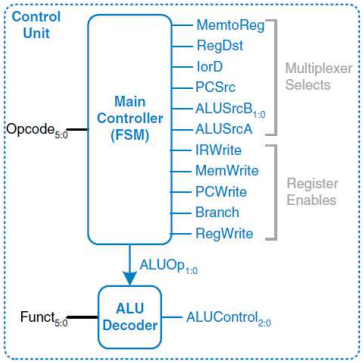
200

❖ 图5



201

❖ 图6 图5中控制器的内部结构



202

参考答案（续）

Element	Parameter	Delay(ps)
Register clk-to-Q	t_{pcq}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

(a) 通过提高哪个部件的速度(即减小该部件的延迟)可以对整个处理器的速度有最大的提升？

- 由于寄存器堆的速度快于存储器($t_{RFread} < t_{mem}$)且$t_{setup} = t_{RFsetup}$，于是有$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$，由表中数据可得$t_{ALU} + t_{mux} = 225ps < t_{mem}$，于是为了减小时钟周期，应当提高存储器的速度。

203

参考答案（续）

Element	Parameter	Delay(ps)
Register clk-to-Q	t_{pcq}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

(b) 在避免不必要浪费的前提之下，该部件的延迟应减小到多少？

- 应将存储器的延迟减小到225ps。

204

参考答案（续）

Element	Parameter	Delay(ps)
Register clk-to-Q	t_{pcq}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

(c) 提升之后的处理器周期是多少？

- 原来处理器的时钟周期为30ps + 25ps + 250ps + 20ps = 325ps，提升存储器速度之后处理器的时钟周期为30ps + 25ps + 225ps + 20ps = 300ps。

参考答案（续）

Element	Parameter	Delay(ps)
Register clk-to-Q	t_{pcq}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

(d) 有一种寄存器堆，它比现有的寄存器堆功耗低40%，但是延迟是现有寄存器堆的两倍，请分析一下使用这种寄存器堆是否有意义。

- 寄存器堆的延迟变为$t_{RF\,read}$=300ps
- 考虑与寄存器堆有关的两个时间，一个是读寄存器堆操作，时间为30ps + 300ps + 20ps = 350ps；另一个是写寄存器堆操作，时间为30ps + 25ps + 300ps + 20ps = 375ps。而原来的时钟周期为325ps，于是处理器的时钟周期将变为375ps，计算性能/功耗比为(325/375)/0.6 = 1.44，说明功耗的下降幅度大于性能的下降幅度，使用这种寄存器堆是有一定意义的。

2.3

❖ 在多周期MIPS处理器上运行下面的程序需要多少个周期？这个程序的CPI是多少？

```
        addi    $s0, $0, 5          # sum = 5
while:
        beq     $s0, $0, done       # if result > 0, execute the while block
        addi    $s0, $s0, -1        # while block: result = result - 1
        j       while
done:
```


参考答案

```
        addi    $s0, $0, 5          # sum = 5
while:
        beq     $s0, $0, done       # if result > 0, execute the while block
        addi    $s0, $s0, -1        # while block: result = result - 1
        j       while
done:
```

- addi和beq都是I型指令，j是跳转指令，addi需要4个时钟周期，beq需要3个时钟周期，j需要3个时钟周期。对程序进行分析可知，第一个addi指令执行了1遍，beq指令执行了6遍(跳出循环时判断条件不成立也执行了一遍)，第二个addi指令执行了5遍，j指令执行了5遍，于是共需要4 + 6×3 + 5×4 + 5×3 = 57个周期。
- CPI为 57 / (1+6+5+5) = 3.35。

作业3、4——流水线
参考答案

209

1 流水线

分别计算该程序在以下的机器上执行时花费的时钟周期数:
(a) 非流水线机器
(b) 采用计分板（scoreboarding）的流水线机器，有5个加法器、5个乘法器，没有数据转发逻辑
(c) 采用计分板（scoreboarding）的流水线机器，有5个加法器、5个乘法器，带数据转发逻辑
(d) 采用计分板（scoreboarding）的流水线机器，有1个加法器、1个乘法器，没有数据转发逻辑
(e) 采用计分板（scoreboarding）的流水线机器，有1个加法器、1个乘法器，带数据转发逻辑

观察以下程序：

```
MUL R3   R1，R2
ADD R5   R4，R3
ADD R6   R4，R1
MUL R7   R8，R9
ADD R4   R3，R7
MUL R10  R5，R6
```

对于所有上述的机器模型，采用以下4阶段的基本指令周期:
1）取指 (1 个时钟周期)
2）译码 (1 个时钟周期)
3）执行
MUL(5个时钟周期)、ADD(2 个时钟周期)，乘法器和加法器内部不是流水线的
4） 写回 (2 个时钟周期)
请列出为了计算对流水线的所有假设（例如如何在流水段之间做数据转发）

210

参考答案：

(a) 非流水线机器
9 + 6 + 6 + 9 + 6 + 9 = 45 时钟周期

(b) 采用计分板（scoreboarding）的流水线机器，有5个加法器、5个乘法器，没有数据转发逻辑

```
Cycles          1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28
MUL R3, R1, R2  F|D|E|E|E|E|E|W|W
ADD R5, R4, R3    F|D|-|-|-|-|-|-|- |E |E |W |W
ADD R6, R4, R1      F|-|-|-|-|-|-|- |D |E |E |W |W
MUL R7, R8, R9                      F |D |E |E |E |E |E |W |W
ADD R4, R3, R7                        F |D |- |- |- |- |- |- |- |E |E |W |W
MUL R10, R5, R6                         F |- |- |- |- |- |- |- |D |E |E |E |E |E |W |W
```

28 时钟周期(或 26时钟周期，利用寄存器堆内部的数据旁路)

211

参考答案（续）：

(c) 采用计分板（scoreboarding）的流水线机器，有5个加法器、5个乘法器，带数据转发逻辑

```
Cycles          1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22
MUL R3, R1, R2  F|D|E|E|E|E|E|W|W
ADD R5, R4, R3    F|D|-|-|-|-|E|E|W |W
ADD R6, R4, R1      F|-|-|-|-|D|E|E |W |W
MUL R7, R8, R9                F|D|E |E |E |E |E |W |W
ADD R4, R3, R7                  F|D |- |- |- |- |E |E |W |W
MUL R10, R5, R6                   F |- |- |- |- |D |E |E |E |E |E |W |W
```

22时钟周期

(d) 采用计分板（scoreboarding）的流水线机器，有1个加法器、1个乘法器，没有数据转发逻辑

```
Cycles          1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29
MUL R3, R1, R2  F|D|E|E|E|E|E|W|W
ADD R5, R4, R3    F|D|-|-|-|-|-|-|- |E |E |W |W
ADD R6, R4, R1      F|-|-|-|-|-|-|- |D |- |E |E |W |W
MUL R7, R8, R9                      F |- |D |E |E |E |E |E |W |W
ADD R4, R3, R7                        F |D |- |- |- |- |- |- |- |E |E |W |W
MUL R10, R5, R6                         F |.- |- |- |- |- |- |- |D |E |E |E |E |E |W |W
```

29时钟周期(或 27时钟周期，利用寄存器堆内部的数据旁路)

212

参考答案（续）：

(e) 采用计分板（scoreboarding）的流水线机器，有1个加法器、1个乘法器，带数据转发逻辑

```
Cycles          1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23
MUL R3, R1, R2  F|D|E|E|E|E|E|W|W
ADD R5, R4, R3    F|D|-|-|-|-|E|E|W |W
ADD R6, R4, R1      F|-|-|-|-|D|-|E |E |W |W
MUL R7, R8, R9              F|-|D |E |E |E |E |E |W |W
ADD R4, R3, R7                  F |D |- |- |- |- |E |E |W |W
MUL R10, R5, R6                   F |- |- |- |- |D |E |E |E |E |E |W |W
```

23时钟周期

213

2 延迟槽

一台五阶段流水线的机器，五个流水段分别是：取指、译码、执行、访存和写回。该机器采用延迟槽技术处理控制相关。无条件分支和有条件分支都在执行阶段获得分支的结果。

(a) 需要多少个延迟槽才能够确保正确的操作?

参考答案：2个

(b) 按照你在(a)中设计的延迟槽数量，下列汇编指令序列中，哪（些）条指令可以放入延迟槽？请使用合适的延迟槽填充方案重写下边的汇编指令代码。

(i)
```
ADD R5  R4, R3
OR   R3  R1, R2
SUB  R7  R5, R6
J       X
延迟槽
LW   R10  (R7)
ADD R6  R1, R2
X:
```

参考答案：
```
ADD R5  R4, R3
J      X
OR   R3  R1, R2
SUB R7  R5, R6
LW   R10  (R7)
ADD R6  R1, R2
X:
```


214

2 延迟槽

一台五阶段流水线的机器，五个流水段分别是：取指、译码、执行、访存和写回。该机器采用延迟槽技术处理控制相关。无条件分支和有条件分支都在执行阶段获得分支的结果。

(a) 需要多少个延迟槽才能够确保正确的操作?

参考答案：2个

(b) 按照你在(a)中设计的延迟槽数量，下列汇编指令序列中，哪（些）条指令可以放入延迟槽？请使用合适的延迟槽填充方案重写下边的汇编指令代码。

(ii)
```
ADD R5  R4, R3
OR   R3  R1, R2
SUB  R7  R5, R6
BEQ  R5  R7, X
  延迟槽
LW   R10  (R7)
ADD  R6  R1, R2
X:
```

参考答案：
```
ADD R5  R4, R3
SUB R7  R5, R6
BEQ R5  R7, X
OR   R3  R1, R2
NOP
LW   R10  (R7)
ADD R6  R1, R2
X:
```


215

2 延迟槽

一台五阶段流水线的机器，五个流水段分别是：取指、译码、执行、访存和写回。该机器采用延迟槽技术处理控制相关。无条件分支和有条件分支都在执行阶段获得分支的结果。

(a) 需要多少个延迟槽才能够确保正确的操作?

参考答案：2个

(b) 按照你在(a)中设计的延迟槽数量，下列汇编指令序列中，哪（些）条指令可以放入延迟槽？请使用合适的延迟槽填充方案重写下边的汇编指令代码。

(iii)
```
ADD R2  R4, R3
OR   R5  R1, R2
SUB R7  R5, R6
BEQ R5  R7, X
  延迟槽
LW   R10  (R7)
ADD R6  R1, R2
X:
```

参考答案：
```
ADD R2  R4, R3
OR   R5  R1, R2
SUB R7  R5, R6
BEQ R5  R7, X
NOP
NOP
LW   R10  (R7)
ADD R6  R1, R2
X:
```


216

2 延迟槽

一台五阶段流水线的机器，五个流水段分别是：取指、译码、执行、访存和写回。该机器采用延迟槽技术处理控制相关。无条件分支和有条件分支都在执行阶段获得分支的结果。

(c) 你能修改流水线减少延迟槽的数量吗(不使用分支预测的技术)? 请清楚地说明你的方法并解释为什么这样能减少延迟槽。

参考答案：

将对jump和branch的目标的解析放到译码阶段。

Jump和 branch可以提前一个时钟周期得到解析，因此一个延迟槽足够保证正确的操作。

217

3 分支预测

考察以下高级语言代码段:

```
int array[1000] = { /* random values */ };
int sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;
for (i = 0; i < 1000; i ++) // 分支 1: 循环分支
{
if (i % 4 == 0) // 分支 2: If 条件分支 1
sum1 += array[i]; // 发生分支的路径
else
sum2 += array[i]; // 不发生分支的路径
if (i % 2 == 0) // 分支 3: If 条件分支 2
sum3 += array[i]; // 发生分支的路径
else
sum4 += array[i]; // 不发生分支的路径
}
```

参考答案：

分支 1:

998/1000 = 99.8%.

第一次和最后一次执行时预测错误

分支 2:

500/1000 = 50%:

分支 3:

0%. 每次执行时分支都改变方向.

(a) 当使用last-time预测器时三个分支的预测准确率分别是多少？(假设每一个分支的last-time计数器起始状态是'不发生') 请写出你的计算过程和依据。

218

3 分支预测

考察以下高级语言代码段:

```
int array[1000] = { /* random values */ };
int sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;
for (i = 0; i < 1000; i ++) // 分支 1: 循环分支
{
if (i % 4 == 0) // 分支 2: If 条件分支 1
sum1 += array[i]; // 发生分支的路径
else
sum2 += array[i]; // 不发生分支的路径
if (i % 2 == 0) // 分支 3: If 条件分支 2
sum3 += array[i]; // 发生分支的路径
else
sum4 += array[i]; // 不发生分支的路径
}
```

参考答案：

分支 1:

997/1000 = 99.7%. 分支头两次执行和最后一次执行（退出循环时）时预测错误

分支 2:

750/1000 = 75%. 分支重复 T N N N T N N N模式，饱和计数器在“强不发生”和“弱不发生”之间转换 (每四次预测发生一次，在分支真正发生之后)，预测结果永远是不发生。

分支 3:

500/1000 = 50%。 分支重复 T N T N模式，每次预测饱和计数器在“强不发生”和“弱不发生”之间转换，每次预测结果是不发生，只有一半是正确的。

(b) 当使用基于2-bit饱和计数器的预测器时三个分支的预测准确率分别是多少？(假设每一个分支的2-bit计数器起始状态是‘强不发生’) 请写出你的计算过程和依据。

219

3 分支预测

考察以下高级语言代码段:

```
int array[1000] = { /* random values */ };
int sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;
for (i = 0; i < 1000; i ++) // 分支 1: 循环分支
{
if (i % 4 == 0) // 分支 2: If 条件分支 1
sum1 += array[i]; // 发生分支的路径
else
sum2 += array[i]; // 不发生分支的路径
if (i % 2 == 0) // 分支 3: If 条件分支 2
sum3 += array[i]; // 发生分支的路径
else
sum4 += array[i]; // 不发生分支的路径
}
```

参考答案：

(i) 弱不发生

分支 2:

749/1000 = 74.9%

分支 3:

0%. 预测器在“弱不发生”和“弱发生”之间振荡

(ii) 弱发生

分支 2:

749/1000 = 74.9%. 分支的模式是 T N N N T N N N ，头四次迭代，预测器的计数器开始于“弱发生”，转换到“强发生”，再转换到“弱发生”，最后是“弱不发生”，因此它的预测是 T T T N；接下来是每组四个分支的循环，计数器状态变化为“强不发生”“弱不发生”“强不发生”“强不发生”，响应的预测为 N N N N 。因此一共有 249x3 + 2 = 749次正确的预测。

分支 3:

500/1000 = 50%. 分支的模式是 T N T N，第一个分支计数器处于“弱发生”状态，转换为“强发生”，接下来是“弱发生”…，因此预测结果是 T T T T ...所以预测器准确率为50%。

注意：对于分支 3, 预测准确率强烈地依赖于分支预测器的初始状态。

(c) 当分支2和分支3的2-bit计数器起始状态分别是 (i)‘弱不发生’；(ii)‘弱发生’ 时，预测准确率分别是多少？请写出你的计算过程和依据。

220

3 分支预测

考察以下高级语言代码段:

```
int array[1000] = { /* random values */ };
int sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;
for (i = 0; i < 1000; i ++) // 分支 1: 循环分支
{
if (i % 4 == 0) // 分支 2: If 条件分支 1
sum1 += array[i]; // 发生分支的路径
else
sum2 += array[i]; // 不发生分支的路径
if (i % 2 == 0) // 分支 3: If 条件分支 2
sum3 += array[i]; // 发生分支的路径
else
sum4 += array[i]; // 不发生分支的路径
}
```

(d) 当使用两层全局历史分支预测器（2bit全局历史寄存器+每分支一张独立的模式历史表，模式历史表的每个表项是一个2-bit饱和计数器）时，三个分支的预测准确率分别是多少？假设全局历史寄存器每一位的初始状态都是‘不发生’，模式历史表中的2-bit饱和计数器初始状态都是‘强不发生’，计算预测准确率时，忽略前500次循环迭代。

221

3 分支预测

参考答案：

分支 1:

499/500 = 99.8% 在500次循环迭代之后，预测器全局历史都进入“强发生”状态，它总是预测发生，只有最后一次循环迭代 (循环分支不发生) 的结果预测错误。

分支 2:

75%. 分析分支2和分支3的相关性很有帮助。全局分支历史包括分支3的最后一次结果以及总是发生的循环分支结果，分别为偶数次和奇数次循环迭代设立独立的饱和计数器很有效。当最后一个分支3 的分支不发生，分支2的模式是 N T N T … 因为永远不会有连续两个 T出现所以饱和计数器会在强不发生和弱不发生之间振荡，导致全部预测都是不发生以及 50% 准确率。当最后一个分支3的分支发生，分支 2的模式是 N N N N … 将导致100%的精度，因此总的准确率为75%。

分支 3:

75%. 与上述情况类似，分支 2的分支历史与分支3相关，并且分支 3 使用两个饱和计数器，由IF条件分支2的结果决定用哪一个。当分支 2不发生，分支 3的模式是 N T N N T N N T N … (这些分支结果来自于第 1, 2, 3, 5, 6, 7, 9, 10, 11, …次迭代)，预测器将会在弱不发生和强不发生之间振荡，但总是预测 N；当分支 2发生，分支3 同样总是发生，所以预测器会100%准确。所以，每4次迭代只有一次预测错误，总预测精度 75%。

222

4 两层分支预测器

假设一个两层全局预测器由全局历史寄存器和**所有分支共享**的一张模式历史表组成(称为预测器 A)
1）我们把分支预测器中不同的分支映射到相同位置的情况称为"分支干扰"。预测器A的结构中，不同分支会在哪里发生这种干扰?

参考答案：
全局历史寄存器 (GHR), 模式历史表 (PHT)

2）另一个两层全局预测器由全局历史寄存器和**每个分支一张**模式历史表组成 (称为预测器 B"),
(a) 什么情况下预测器A的预测准确率低于预测器B? 请解释理由，并举例说明。可以通过代码来说明。

参考答案：
当映射到同一个PHT条目的两个分支转向相反方向时预测器A的预测精度会低。考虑一个分支B1，它对于给定的全局历史总是发生，如果B1有自己的PHT，它永远能预测正确。现在，考虑另一个分支B2，对于同样的历史总是不发生，如果B2有自己的PHT，它也总是能预测正确。但是，如果B1和B2共享一个PHT，它们映射到同一个PHT条目，因此互相影响并降低了彼此的预测精度。

223

4 两层分支预测器

假设一个两层全局预测器由全局历史寄存器和**所有分支共享**的一张模式历史表组成(称为预测器 A)
2）另一个两层全局预测器由全局历史寄存器和**每个分支一张**模式历史表组成 (称为预测器 B"),
(b) 预测器A能获得比预测器B更高的预测准确率吗? 请解释理由，并举例说明。可以通过代码来说明。

参考答案：
如果一个分支B1在有另一个分支B2共享同一个PHT条目时比独有自己的PHT预测结果更准确，就有这种可能。 考虑这种场景，分支B1对于给定的全局历史(当B1有自己的PHT)总是预测错误，原因是这段历史中它总是在发生和不发生之间振荡，现在有一个总是发生的分支B2映射到相同的PHT条目，这将会改进B1的预测精度，因为这个时候分支B1会由于B2总是发生而总是被预测发生。如果B2在相同的历史中比B1执行的更频繁的话，这也可能不会降低B2的预测精度。因此，总的预测精度将会得到改善。

224

4 两层分支预测器

假设一个两层全局预测器由全局历史寄存器和**所有分支共享**的一张模式历史表组成(称为预测器 A)
2）另一个两层全局预测器由全局历史寄存器和**每个分支一张**模式历史表组成 (称为预测器 B"),
(c) 分支干扰是否总是影响预测器的预测准确率？请解释理由，并举例说明。可以通过代码来说明。

参考答案：
如果映射到同一个PHT条目的分支转向相同，那么预测器A和B会获得同样的预测精度。在这种情况下，分支之间的干扰不会影响预测精度。考虑两个分支B1和B2在某段固定的全局历史中总是发生，那么不管是有自己的PHT还是共享PHT条目，预测精度是相同的。

5 分支预测和推断

考察两台具有15段流水线的机器A和B，流水段分布如下：
取指 (1个阶段)
译码 (8个阶段)
执行 (5个阶段)
写回 (1个阶段)
两台机器都会在发生流相关时采用数据转发。在译码的最后一个阶段检测是否有流相关，指令会在停顿在这个阶段等待检测结果。
机器A有一个预测准确率为P%的分支预测器，分支方向和目标在执行的最后一个阶段产生。
机器B采用推断执行，类似咱们在课堂上讲的方式。

6 分支预测和推断

1）考察以下在机器A上执行的代码段：

```
Add   r3  r1, r2
sub   r5  r6, r7
beq   r3, r5, X
addi  r10  r1, 5
add   r12  r7, r2
add   r1  r11, r9
X: addi  r15  r2, 10
.....
```

我们把这段代码转化为在机器B上执行的推断代码，大概是下面这个样子:

```
add       r3  r1, r2
sub       r5  r6, r7
cmp       r3, r5
addi.ne   r10  r1, 5
add.ne    r12  r7, r2
add.ne    r14  r11, r9
addi      r15  r2, 10
.....
```

(假设条件结果由'cmp'指令计算，推断由'.ne'指令根据条件结果执行。条件结果在执行的最后一个阶段计算并且可以像其他值一样被转发)
这一段代码会重复执行上百次，分支40%可能性发生，60%可能性不发生，平均而言，P取什么样的值会使机器A比机器B具有更高的指令吞吐量?

6 分支预测和推断

参考答案：

这个问题显示了在分支预测机器上预测错误的惩罚和在推断执行机器上执行无用指令浪费的时钟周期之间的折衷。
我们在这里假设:

- 机器A和B具有分离的（流水）分支/比较和加法执行单元，即，在分支/比较指令停顿时可以执行加法指令
- 按序写回
- 当推断执行的指令被发现是无用的(在比较指令结果之后)，它仍然会继续剩余的流水段，相当于插入空操作。

对于不同的假设，这个问题的解答会不同，都是可能的正确答案。
在机器 A，当 beq r3, r5, X **分支不发生并且预测正确**，执行的过程如下：

```
add r3 <- r1, r2     F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
sub r5 <- r6, r7        F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
beq r3, r5, X              F|D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |E1|E2|E3|E4|E5|WB|
addi r10 <- r1, 5             F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |WB|
add r12 <- r7, r2                F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |WB|
add r1 <- r11, r9                   F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |WB|
X: addi r15 <- r2, 10                  F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |WB|
.....
```

6 分支预测和推断

参考答案：

这个问题显示了在分支预测机器上预测错误的惩罚和在推断执行机器上执行无用指令浪费的时钟周期之间的折衷。
我们在这里假设:

- 机器A和B具有分离的（流水）分支/比较和加法执行单元，即，在分支/比较指令停顿时可以执行加法指令
- 按序写回
- 当推断执行的指令被发现是无用的(在比较指令结果之后)，它仍然会继续剩余的流水段，相当于插入空操作。

对于不同的假设，这个问题的解答会不同，都是可能的正确答案。
在机器 A，当 beq r3, r5, X **分支发生并且预测正确**，执行过程如下:

```
add r3 <- r1, r2    F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
sub r5 <- r6, r7      F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
beq r3, r5, X           F|D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |- |E1|E2|E3|E4|E5|WB|
X: addi r15 <- r2, 10     F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |- |WB|
.....
```


229

6 分支预测和推断

参考答案：

机器A因为分支预测错误(不管是发生还是未发生)受到17个时钟周期(8个译码+5个执行+4个停顿)的惩罚。
机器 B在分支不发生且预测正确时的执行跟机器A完全一样，但是，当分支发生(比较结果相等)时机器B浪费3个时钟周期，如下图所示。

```
add r3 <- r1, r2    F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
sub r5 <- r6, r7      F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
cmp r3, r5              F|D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |- |E1|E2|E3|E4|E5|WB|
addi.ne r10 <- r1, 5      F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |- |WB|
add.ne r12 <- r7, r2        F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |- |WB|
add.ne r14 <- r11, r9         F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |- |WB|
addi r15 <- r2, 10              F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |- |WB|
.....
```

所以，如果预测错误的代价低于执行无用指令浪费的周期，机器A比机器B的指令吞吐高。
(1- P) x 17 < 3 x 0.4
因此，P > 0.9294时，机器A 比机器B有更高的指令吞吐量。

230

6 分支预测和推断

2）考察在机器A上执行的另一段代码:

```
add    r3  r1, r2
sub    r5  r6, r7
beq    r3, r5, X
addi   r10  r1, 5
add    r12  r10, r2
add    r14  r12, r9
X: addi   r15  r14, 10
.....
```

我们把这段代码转化为在机器B上执行的推断代码，大概是下面这个样子:

```
add       r3  r1, r2
sub       r5  r6, r7
cmp       r3, r5
addi.ne   r10  r1, 5
add.ne    r12  r10, r2
add.ne    r14  r12, r9
addi      r15  r14, 10
.....
```

(假设条件结果由'cmp'指令计算，推断由'.ne'指令根据条件结果执行。条件结果在执行的最后一个阶段计算并且可以像其他值一样被转发)
这一段代码会重复执行上百次，分支40%可能性发生，60%可能性不发生，平均而言，P取什么样的值会使机器A比机器B具有更高的指令吞吐量?

231

6 分支预测和推断

参考答案：

在机器A上， 当 beq r3, r5, X **分支未发生且预测正确**，执行过程如下：

```
add r3 <- r1, r2  F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
sub r5 <- r6, r7    F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
beq r3, r5, X         F|D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |- |E1|E2|E3|E4|E5|WB|
addi r10 <- r1, 5       F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |- |WB|
add r12 <- r10, r2        F|D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |E1|E2|E3|E4|E5|WB|
add r14 <- r12, r9          F|D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |- |- |- |E1|E2|E3|E4|E5|WB|
X: addi r15 <- r14, 10        F|D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |- |- |- |- |- |- |E1|E2|E3|E4|E5|WB|
```

当**分支发生且预测正确**，执行过程如下：

```
add r3 <- r1, r2  F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
sub r5 <- r6, r7    F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
cmp r3, r5            F|D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |E1|E2|E3|E4|E5|WB|
addi r15 <- r14, 10     F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |WB|
.....
```


232

6 分支预测和推断

参考答案：

机器A因为分支预测错误(不管是发生还是未发生)受到17个时钟周期(8个译码+5个执行+4个停顿)的惩罚。

机器 B在分支不发生且预测正确时的执行跟机器A完全一样，但是，当分支发生(比较结果相等)时机器B浪费11个时钟周期，如下图所示。

```
add r3 <- r1, r2   F|D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
sub r5 <- r6, r7     F |D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|WB|
cmp r3, r5             F |D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |E1|E2|E3|E4|E5|WB|
addi.ne r10 <- r1, 5     F |D1|D2|D3|D4|D5|D6|D7|D8|E1|E2|E3|E4|E5|- |- |- |WB|
add.ne r12 <- r10, r2      F |D1|D2|D3|D4|D5|D6|D7|D8|- |- |- |E1|E2|E3|E4|E5|WB|
add.ne r14 <- r12, r9        F |D1|D2|D3|D4|D5|D6|D7|- |- |- |- |D8|- |- |- |- |E1|E2|E3|E4|E5|WB|
addi r15 <- r14, 10            F |D1|D2|D3|D4|D5|D6|- |- |- |- |D7|- |- |- |- |D8|- |- |- |- |E1|E2|E3|E4|E5|WB|
```

所以，如果预测错误的代价低于执行无用指令浪费的周期，机器A比机器B的指令吞吐高。

(1- P) x 17 < 11 x 0.4

因此，P > 0.7411时，机器A 比机器B有更高的指令吞吐量。

233

作业5——Cache和Memory 参考答案

234

1 Cache

下面给出了运行在带数据cache的处理器上的程序所生成的四种不同的地址序列，同时给出了每种序列的cache命中率。假设cache在每个序列开始时是空的，请回答该处理器数据cache的下述参数分别是多少：

(a) 相联度(1, 2 还是4路)

假设：所有的访存都是单字节的访问，所有的地址都是字节地址。

序列	地址序列	命中率
1	0, 2, 4, 8, 16, 32	0.33
2	0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0	0.33
3	0, 64, 128, 256, 512, 256, 128, 64, 0	0.33
4	0, 512, 1024, 0, 1536, 0, 2048, 512	0.25

参考答案：

4

对于序列 2, 块 0, 512, 1024 和1536是仅有的重用块，也就是会第二次访问并可能会导致cache命中的块，其中3块应该在第二次被访问时命中，因此命中率才会是 0.33 (3/9)。

块大小是 8字节(见下一问)，对于任何的 cache 大小 (256B 或 512B)，这些块都映射到set 0。因此，相联度是1或者2会造成四块中最多1或2块在第二次访问时在cache中，使得最大的可能命中率小于3/9，而这个序列的命中率是3/9，说明相联度只能是4。

235

1 Cache

下面给出了运行在带数据cache的处理器上的程序所生成的四种不同的地址序列，同时给出了每种序列的cache命中率。假设cache在每个序列开始时是空的，请回答该处理器数据cache的下述参数分别是多少：

(b) 块大小(1, 2, 4, 8, 16 还是32 字节)

假设：所有的访存都是单字节的访问，所有的地址都是字节地址。

序列	地址序列	命中率
1	0, 2, 4, 8, 16, 32	0.33
2	0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0	0.33
3	0, 64, 128, 256, 512, 256, 128, 64, 0	0.33
4	0, 512, 1024, 0, 1536, 0, 2048, 512	0.25

参考答案：

8 字节

对于序列 1， 6次访问中只有2次(地址2和 4) 能够cache命中，命中率是0.33。除了8字节外其他的块大小都不能满足命中率为0.33，要么大要么小。

236

1 Cache

下面给出了运行在带数据cache的处理器上的程序所生成的四种不同的地址序列，同时给出了每种序列的cache命中率。假设cache在每个序列开始时是空的，请回答该处理器数据cache的下述参数分别是多少：
(c) cache总容量(256还是 512 字节)

假设：所有的访存都是单字节的访问，所有的地址都是字节地址。

序列	地址序列	命中率
1	0, 2, 4, 8, 16, 32	0.33
2	0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0	0.33
3	0, 64, 128, 256, 512, 256, 128, 64, 0	0.33
4	0, 512, 1024, 0, 1536, 0, 2048, 512	0.25

参考答案：
256 字节
对于序列3，512字节的容量会使命中率达到4/9(4路组相联，8字节cache块，无论什么替换策略)，高于 0.33，所以 cache 总容量是 256 字节。

237

1 Cache

下面给出了运行在带数据cache的处理器上的程序所生成的四种不同的地址序列，同时给出了每种序列的cache命中率。假设cache在每个序列开始时是空的，请回答该处理器数据cache的下述参数分别是多少：
(d) 替换策略(LRU 还是 FIFO)

假设：所有的访存都是单字节的访问，所有的地址都是字节地址。

序列	地址序列	命中率
1	0, 2, 4, 8, 16, 32	0.33
2	0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0	0.33
3	0, 64, 128, 256, 512, 256, 128, 64, 0	0.33
4	0, 512, 1024, 0, 1536, 0, 2048, 512	0.25

参考答案：
LRU
对于上述的参数，所有序列4中的cache块被映射到组0，如果使用 FIFO替换策略，命中率是 3/8，而采用LRU 替换策略命中率是 1/4，所以替换策略是LRU。

238

2 内存的交叉存取

2.1 一台机器有4 KB的主存，由1个通道、1个rank和N(N>1)个bank构成。系统没有虚拟存储。
1) 数据采用cache块交叉存取策略，即连续的cache块对应到连续的bank上；
2) cache块大小为32字节，bank的1行有128字节；
3) 采用打开行策略，即行缓冲中的行在被访问后继续保持在行缓冲中，直到有别的行被访问；
4) 行缓冲命中指访问的行存在于行缓冲，行缓冲缺失指访问的行不在行缓冲。
(a) 某个程序在这台机器上执行，访问以下字节时(数字表示字节的位置，比如320表示第320个字节)发生片上cache缺失而需要访存：0, 32, 320, 480, 4, 36, 324, 484, 8, 40, 328, 488, 12, 44, 332, 492，若行缓冲命中率为0，即所有访问的行都不在行缓冲中，请问bank数N的最小值是多少?
参考答案：
2 个
Cache块大小是32字节，所以，对于给定的访存序列相应的cache块访问序列是 0, 1, 10, 15, 0, 1, 10 ,15, 0, 1, 10, 15, 0, 1, 10, 15。
当bank数是1时，所有cache块映射到同一个bank，块 0、1、10和 15 映射到行0、0、2和3。所以，当块1紧接着块0被访问时，会产生行缓冲命中，即行缓冲命中率不为0。
当 bank数是 2时，块 0、1、10和15映射到不同的行和bank (bank 0, 行 0；bank 1, 行 0；bank 0, 行 1；bank 1, 行 1)。这样，访问序列就是(bank 0, 行 0), (bank 1, 行 0), (bank 0, 行 1), (bank 1, 行 1) (重复四次)。
因此，每个bank上的行 0和 1被交替访问，导致行缓冲命中率为0。

239

2 内存的交叉存取

2.1 一台机器有4 KB的主存，由1个通道、1个rank和N(N>1)个bank构成。系统没有虚拟存储。
1) 数据采用cache块交叉存取策略，即连续的cache块对应到连续的bank上；
2) cache块大小为32字节，bank的1行有128字节；
3) 采用打开行策略，即行缓冲中的行在被访问后继续保持在行缓冲中，直到有别的行被访问；
4) 行缓冲命中指访问的行存在于行缓冲，行缓冲缺失指访问的行不在行缓冲。
(b) 如果对于同一个序列，行缓冲命中率是75%，请问bank数N的最小值是多少?

参考答案：
4 个
当 bank数是1时, cache 块0、1、10和 15 映射到行0、0、2和3 (与a中相同)。这时的行访问序列为 0, 0, 2, 3 (重复四次)，其中3次访问行缓冲不命中，命中率是 25%。
对于其它数量的 bank, 块0、1、10和15映射到不同的行。
假设有这四个cache块的行没有在行缓冲中打开，那么最大的命中率只能是75%，因为对每个块的第一次访问不命中(强制缺失)。这个最大命中率(75%)意味着每个块除了第一次访问后续访问不能有不命中，因此，含有四个块的行必须映射到不同的bank，即最少有4个bank。

240

2 内存的交叉存取

2.1 一台机器有4 KB的主存，由1个通道、1个rank和N(N>1)个bank构成。系统没有虚拟存储。
1) 数据采用cache块交叉存取策略，即连续的cache块对应到连续的bank上；
2) cache块大小为32字节，bank的1行有128字节；
3) 采用打开行策略，即行缓冲中的行在被访问后继续保持在行缓冲中，直到有别的行被访问；
4) 行缓冲命中指访问的行存在于行缓冲，行缓冲缺失指访问的行不在行缓冲。
(c) i) 对于同一序列，行缓冲的命中率能达到100%吗? 请解释原因

参考答案：
四个cache块映射到不同的行，因此行缓冲命中率达到100%的唯一可能就是包含每个块的行都已经在行缓冲中打开。

241

2 内存的交叉存取

2.1 一台机器有4 KB的主存，由1个通道、1个rank和N(N>1)个bank构成。系统没有虚拟存储。
1) 数据采用cache块交叉存取策略，即连续的cache块对应到连续的bank上；
2) cache块大小为32字节，bank的1行有128字节；
3) 采用打开行策略，即行缓冲中的行在被访问后继续保持在行缓冲中，直到有别的行被访问；
4) 行缓冲命中指访问的行存在于行缓冲，行缓冲缺失指访问的行不在行缓冲。
(c) ii) 如果能达到, 最少需要多少bank才能够获得100%的行缓冲命中率?

参考答案：
4 个bank就足以实现，只要4个分别包含4个cache块的行都已经打开(分别在4个bank)。

242

2 内存的交叉存取

2.2 一个DRAM主存储系统由1个通道、1个rank和N个bank构成。Bank一行256字节，一个cache块64字节。数据采用跨bank的行交叉存取方式组织，物理地址的分配方案如下：

行	Bank	列	BiB(Bytes in Bus)

采用打开行策略，即行缓冲中的行在被访问后继续保持在行缓冲中，直到有别的行被访问。初始时，所有bank的第1024行打开。
(a) 当有如下的cache块访问序列时，如果系统的行缓冲命中率为33.3% (即1/3)，请问系统中共有多少个bank:
0, 4, 8, 16, 32, 64, 128, 256, 128, 64, 32, 16, 8, 4, 0

参考答案：
2^5= 32 bank

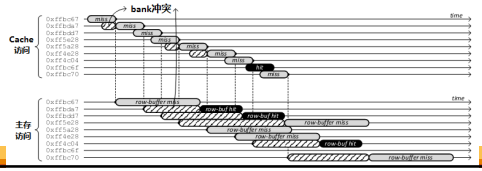
(b) 如果行缓冲命中率是7/15，请问系统中共有多少个bank?

参考答案：
2^7= 128 bank

243

3 Bank

一个处理器的分层存储结构由一个小的SRAM L1-cache和一个大的DRAM主存储器组成，SRAM和DRAM都被划分成bank。处理器有24位物理地址空间，并且不支持虚拟存储(即所有地址都是物理地址)。某个应用开始在这个处理器上运行，下图显示了在时间尺度上应用对存储系统引用的过程(包括在L1-cache和主存中)。
例如, 应用对存储第一次引用的字节地址是0xffbc67(假设所有的引用都是对按字节编址的内存地址的按字节读取)，但是这次引用在L1-cache中不命中 (假设L1-cache初始时为空)。紧接着，应用访问主存，这会经历一次行缓冲的不命中(初始时，假设主存的所有bank都打开一个永远不会被任何应用访问的行)。最后，包含字节地址0xffbc67的cache块从主存取到cache中。
随后的内存引用可能会经历L1-cache和/或主存的bank冲突(当某个特定的bank还在提供之前的某个引用时)。



244

3 Bank

下表用16进制和2进制分别给出了该应用对存储系统引用的地址序列。

16进制	2进制
ffbc67	1111 1111 1011 1100 0110 0111
ffbda7	1111 1111 1011 1101 1010 0111
ffbdd7	1111 1111 1011 1101 1101 0111
ff5e28	1111 1111 0101 1110 0010 1000
ff5a28	1111 1111 0101 1010 0010 1000
ff4e28	1111 1111 0100 1110 0010 1000
ff4c04	1111 1111 0100 1100 0000 0100
ffbc6f	1111 1111 1011 1100 0110 1111
ffbc70	1111 1111 1011 1100 0111 0000

注意：对于以下问题，假设所有的偏移量和索引来自连续的地址位

(a) L1-cache的块大小是多少字节？24-bit物理地址中哪些位是cache块偏移量？(物理地址的最低位为0位)

参考答案：
块大小: 16字节
块偏移量所在位置: 0-3位

请分析上面的图和表，回答下列有关处理器上cache和主存的组织相关的问题，以下是一些假设：
1) L1-cache的假设
块大小: ? (2的幂, 大于2)
相联度: ? (2的幂, 大于2)
数据存储的大小: ? (2的幂, 大于2)
Bank数: ? (2的幂, 大于2)
初始时为空
2) 主存的假设
通道数: 1
每通道rank数: 1
每rank的bank数: ? (2的幂, 大于2)
每bank的行数: ? (2的幂, 大于2)
每行的cache块数: ? (2的幂, 大于2)
包含应用的整个工作集
初始时，所有的bank打开第0行，应用永远不会访问该行

245

3 Bank

下表用16进制和2进制分别给出了该应用对存储系统引用的地址序列。

16进制	2进制
ffbc67	1111 1111 1011 1100 0110 0111
ffbda7	1111 1111 1011 1101 1010 0111
ffbdd7	1111 1111 1011 1101 1101 0111
ff5e28	1111 1111 0101 1110 0010 1000
ff5a28	1111 1111 0101 1010 0010 1000
ff4e28	1111 1111 0100 1110 0010 1000
ff4c04	1111 1111 0100 1100 0000 0100
ffbc6f	1111 1111 1011 1100 0110 1111
ffbc70	1111 1111 1011 1100 0111 0000

注意：对于以下问题，假设所有的偏移量和索引来自连续的地址位

(b) L1-cache有多少bank? 24-bit物理地址中哪些位是L1-cache的bank索引？(物理地址的最低位为0位)

参考答案：
L1-cache的 bank数: 4
L1-cache bank 索引位的位置: 4-5

请分析上面的图和表，回答下列有关处理器上cache和主存的组织相关的问题，以下是一些假设：
1) L1-cache的假设
块大小: ? (2的幂, 大于2)
相联度: ? (2的幂, 大于2)
数据存储的大小: ? (2的幂, 大于2)
Bank数: ? (2的幂, 大于2)
初始时为空
2) 主存的假设
通道数: 1
每通道rank数: 1
每rank的bank数: ? (2的幂, 大于2)
没bank的行数: ? (2的幂, 大于2)
每行的cache块数: ? (2的幂, 大于2)
包含应用的整个工作集
初始时，所有的bank打开第0行，应用永远不会访问该行

246

3 Bank

下表用16进制和2进制分别给出了该应用对存储系统引用的地址序列。

16进制	2进制
ffbc67	1111 1111 1011 1100 0110 0111
ffbda7	1111 1111 1011 1101 1010 0111
ffbdd7	1111 1111 1011 1101 1101 0111
ff5e28	1111 1111 0101 1110 0010 1000
ff5a28	1111 1111 0101 1010 0010 1000
ff4e28	1111 1111 0100 1110 0010 1000
ff4c04	1111 1111 0100 1100 0000 0100
ffbc6f	1111 1111 1011 1100 0110 1111
ffbc70	1111 1111 1011 1100 0111 0000

注意：对于以下问题，假设所有的偏移量和索引来自连续的地址位

(c) 主存中有多少bank? 24-bit物理地址中哪些位是主存的bank索引? (物理地址的最低位为0位)

参考答案：
主存bank数: 8
主存bank索引位的位置: 10-12

请分析上面的图和表，回答下列有关处理器上cache和主存的组织相关的问题，以下是一些假设：
1) L1-cache的假设
块大小: ? (2的幂, 大于2)
相联度: ? (2的幂, 大于2)
数据存储的大小: ? (2的幂, 大于2)
Bank数: ? (2的幂, 大于2)
初始时为空
2) 主存的假设
通道数: 1
每通道rank数: 1
每rank的bank数: ? (2的幂, 大于2)
没bank的行数: ? (2的幂, 大于2)
每行的cache块数: ? (2的幂, 大于2)
包含应用的整个工作集
初始时，所有的bank打开第0行，应用永远不会访问该行

247

3 Bank

下表用16进制和2进制分别给出了该应用对存储系统引用的地址序列。

16进制	2进制
ffbc67	1111 1111 1011 1100 0110 0111
ffbda7	1111 1111 1011 1101 1010 0111
ffbdd7	1111 1111 1011 1101 1101 0111
ff5e28	1111 1111 0101 1110 0010 1000
ff5a28	1111 1111 0101 1010 0010 1000
ff4e28	1111 1111 0100 1110 0010 1000
ff4c04	1111 1111 0100 1100 0000 0100
ffbc6f	1111 1111 1011 1100 0110 1111
ffbc70	1111 1111 1011 1100 0111 0000

注意：对于以下问题，假设所有的偏移量和索引来自连续的地址位

(d) 物理地址向主存映射时用了什么样的交叉存取方案?

参考答案：
行交叉存取

请分析上面的图和表，回答下列有关处理器上cache和主存的组织相关的问题，以下是一些假设：
1) L1-cache的假设
块大小: ? (2的幂, 大于2)
相联度: ? (2的幂, 大于2)
数据存储的大小: ? (2的幂, 大于2)
Bank数: ? (2的幂, 大于2)
初始时为空
2) 主存的假设
通道数: 1
每通道rank数: 1
每rank的bank数: ? (2的幂, 大于2)
没bank的行数: ? (2的幂, 大于2)
每行的cache块数: ? (2的幂, 大于2)
包含应用的整个工作集
初始时，所有的bank打开第0行，应用永远不会访问该行

248

3 Bank

下表用16进制和2进制分别给出了该应用对存储系统引用的地址序列。

16进制	2进制
ffbc67	1111 1111 1011 1100 0110 0111
ffbda7	1111 1111 1011 1101 1010 0111
ffbdd7	1111 1111 1011 1101 1101 0111
ff5e28	1111 1111 0101 1110 0010 1000
ff5a28	1111 1111 0101 1010 0010 1000
ff4e28	1111 1111 0100 1110 0010 1000
ff4c04	1111 1111 0100 1100 0000 0100
ffbc6f	1111 1111 1011 1100 0110 1111
ffbc70	1111 1111 1011 1100 0111 0000

注意：对于以下问题，假设所有的偏移量和索引来自连续的地址位

(e) 为了支持24-bit的物理地址空间，主存的每一个bank需要多少行？24-bit物理地址中哪些位是主存的行索引？(物理地址的最低位为0位)

参考答案：
每个主存bank的行数: 2048
行索引位的位置: 13-23

请分析上面的图和表，回答下列有关处理器上cache和主存的组织相关的问题，以下是一些假设：
1) L1-cache的假设
块大小: ? (2的幂, 大于2)
相联度: ? (2的幂, 大于2)
数据存储的大小: ? (2的幂, 大于2)
Bank数: ? (2的幂, 大于2)
初始时为空
2) 主存的假设
通道数: 1
每通道rank数: 1
每rank的bank数: ? (2的幂, 大于2)
没bank的行数: ? (2的幂, 大于2)
每行的cache块数: ? (2的幂, 大于2)
包含应用的整个工作集
初始时，所有的bank打开第0行，应用永远不会访问该行

249

3 Bank

下表用16进制和2进制分别给出了该应用对存储系统引用的地址序列。

16进制	2进制
ffbc67	1111 1111 1011 1100 0110 0111
ffbda7	1111 1111 1011 1101 1010 0111
ffbdd7	1111 1111 1011 1101 1101 0111
ff5e28	1111 1111 0101 1110 0010 1000
ff5a28	1111 1111 0101 1010 0010 1000
ff4e28	1111 1111 0100 1110 0010 1000
ff4c04	1111 1111 0100 1100 0000 0100
ffbc6f	1111 1111 1011 1100 0110 1111
ffbc70	1111 1111 1011 1100 0111 0000

注意：对于以下问题，假设所有的偏移量和索引来自连续的地址位

(f) 在一行中的每个cache块被称为列，一行中有多少列？24-bit物理地址中哪些位是主存的列索引？(物理地址的最低位为0位)

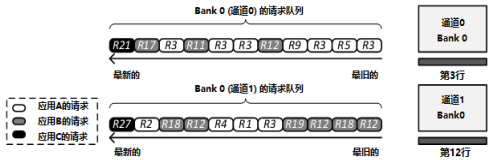
参考答案：
每行的列数: 64
列索引位的位置: 4

请分析上面的图和表，回答下列有关处理器上cache和主存的组织相关的问题，以下是一些假设：
1) L1-cache的假设
块大小: ? (2的幂, 大于2)
相联度: ? (2的幂, 大于2)
数据存储的大小: ? (2的幂, 大于2)
Bank数: ? (2的幂, 大于2)
初始时为空
2) 主存的假设
通道数: 1
每通道rank数: 1
每rank的bank数: ? (2的幂, 大于2)
没bank的行数: ? (2的幂, 大于2)
每行的cache块数: ? (2的幂, 大于2)
包含应用的整个工作集
初始时，所有的bank打开第0行，应用永远不会访问该行

250

4 内存调度

为了响应访存请求，内存控制器会发射1条或多条DRAM命令以从bank访问数据。有4种不同的DRAM命令。
1) 激活(ACTIVATE): 取被访问的行装入bank的行缓冲。 这一操作也被称为打开行(延迟: 15ns)
2) 预充电(PRECHARGE): 将bank的行缓冲中的内容存回行。这一操作也被称为关闭行 (延迟: 15ns)
3) 读/写: 从行缓冲中访问数据(延迟: 15ns)
下图显示了在时刻t0时内存控制器中的内存请求缓冲的快照。每一个请求按照颜色的不同代表了其所属的不同应用(假设所有的应用运行在独立的核上)。同时，每个请求标注了它要访问的行地址(或索引)，例如R3表示请求的是第3行。另外，假设所有的请求都是读请求。



251

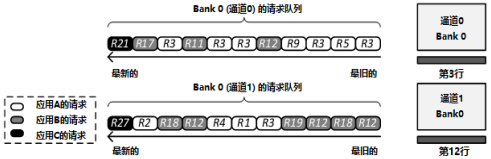
4 内存调度

访存请求在读命令完成后被响应(即读命令被发射15ns之后)，每个应用(A、B或C)停顿直到它所有访存请求被响应为止。
假设初始时(t0时)，每个bank的第3行和第12行分别被取出并存入行缓冲，没有任何其他应用的请求到达内存控制器。

4.1 非应用感知的调度策略

(a) 使用先来先服务调度策略(FCFS)，每个应用的停顿时间是多少？

参考答案：
H-命中的延迟，M-缺失的延迟
应用A: MAX (2H+7M, H+9M) = H+9M = 15+405 = 420ns
应用 B: MAX (2H+8M, H+8M) = 2H+8M = 30+360 = 390ns
应用 C: MAX (2H+9M, H+10M) = H+10M = 15+450 = 465ns



252

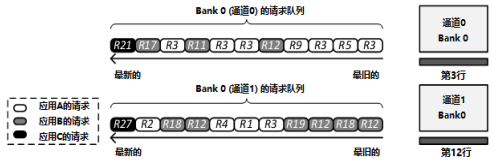
4 内存调度

访存请求在读命令完成后被响应(即读命令被发射15ns之后)，每个应用(A、B或C)停顿直到它所有访存请求被响应为止。
假设初始时(t0时)，每个bank的第3行和第12行分别被取出并存入行缓冲，没有任何其他应用的请求到达内存控制器。

4.1 非应用感知的调度策略

(b) 使用行缓冲优先加先来先服务的调度策略(FR-FCFS)，每个应用的停顿时间是多少?

参考答案：
应用 A: MAX (5H+2M, (4H+2M)+4M) = 4H+6M = 60+270 = 330ns
应用 B: MAX ((5H+2M)+3M, 4H+2M) = 5H+5M = 75+225 = 300ns
应用 C: MAX (((5H+2M)+3M)+M, ((4H+2M)+4M)+M) = 4H+7M = 60 + 315 = 375ns



253

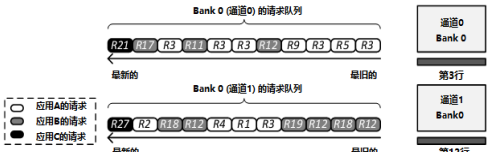
4 内存调度

访存请求在读命令完成后被响应(即读命令被发射15ns之后)，每个应用(A、B或C)停顿直到它所有访存请求被响应为止。
假设初始时(t0时)，每个bank的第3行和第12行分别被取出并存入行缓冲，没有任何其他应用的请求到达内存控制器。

4.1 非应用感知的调度策略

(c) FR-FCFS利用的是内存引用行为的什么特征? (6个字 ☺)

参考答案：
行缓冲局部性



254

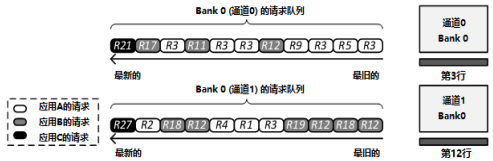
4 内存调度

访存请求在读命令完成后被响应(即读命令被发射15ns之后)，每个应用(A、B或C)停顿直到它所有访存请求被响应为止。
假设初始时(t0时)，每个bank的第3行和第12行分别被取出并存入行缓冲，没有任何其他应用的请求到达内存控制器。

4.1 非应用感知的调度策略

(d) 请简要描述可以最大化请求吞吐量的调度策略，请求吞吐量的意思是每单位时间响应的请求数。 (十个字左右☺)

参考答案：
行缓冲优先的先来先服务(FR-FCFS)



255

4 内存调度

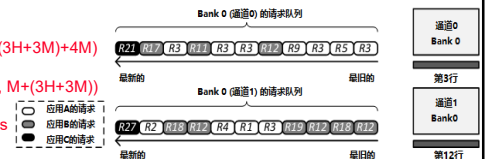
访存请求在读命令完成后被响应(即读命令被发射15ns之后)，每个应用(A、B或C)停顿直到它所有访存请求被响应为止。
假设初始时(t0时)，每个bank的第3行和第12行分别被取出并存入行缓冲，没有任何其他应用的请求到达内存控制器。

4.2 应用感知的调度策略

图中的3个应用，应用C是内存密集程度最低的(即有最少的请求数)。然而，它经历了最长的停顿时间，因为它的请求响应晚于其它多个被优先服务的应用的请求。为了保证应用C的停顿时间最短，可以为它的请求分配最高优先级，而给应用A和B的请求分配同样的低优先级。

(a) 调度策略 X: 当应用C分配高优先级并且应用A和B分配相同的低优先级，每个应用的停顿时间是多少? (对于相同优先级的请求，假设使用FR-FCFS策略)

参考答案：
应用 A: MAX (M+(4H+3M), M+(3H+3M)+4M)
= 3H+8M = 45+360 = 405ns
应用 B: MAX (M+(4H+3M)+3M, M+(3H+3M))
= 4H+7M = 60+315 = 375ns
应用 C: MAX (M, M) = M = 45ns



256

4 内存调度

访存请求在读命令完成后被响应(即读命令被发射15ns之后)，每个应用(A、B或C)停顿直到它所有访存请求被响应为止。
假设初始时(t0时)，每个bank的第3行和第12行分别被取出并存入行缓冲，没有任何其他应用的请求到达内存控制器。

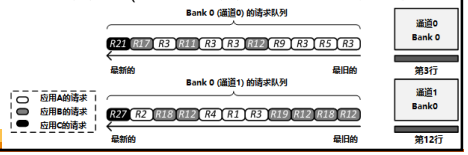
4.2 应用感知的调度策略

图中的3个应用，应用C是内存密集程度最低的(即有最少的请求数)。然而，它经历了最长的停顿时间，因为它的请求响应晚于其它多个被优先服务的应用的请求。为了保证应用C的停顿时间最短，可以为它的请求分配最高优先级，而给应用A和B的请求分配同样的低优先级。

你能否设计一个更好的调度策略？虽然应用C的停顿时间小了，但是应用A和B之间还是会互相影响。

(b) 为其它两个应用分配优先级，这样你可以最小化所有应用的平均停顿时间。请具体从大到小列出三个应用的优先级(对于相同优先级的请求，假设使用FR-FCFS策略)

参考答案：
C>B>A



4 内存调度

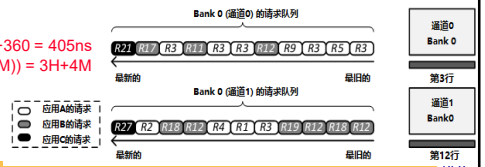
访存请求在读命令完成后被响应(即读命令被发射15ns之后)，每个应用(A、B或C)停顿直到它所有访存请求被响应为止。
假设初始时(t0时)，每个bank的第3行和第12行分别被取出并存入行缓冲，没有任何其他应用的请求到达内存控制器。

4.2 应用感知的调度策略

图中的3个应用，应用C是内存密集程度最低的(即有最少的请求数)。然而，它经历了最长的停顿时间，因为它的请求响应晚于其它多个被优先服务的应用的请求。为了保证应用C的停顿时间最短，可以为它的请求分配最高优先级，而给应用A和B的请求分配同样的低优先级。

(c) 调度策略Y: 使用你的新调度策略，每个应用的停顿时间分别是多少？(对于相同优先级的请求，假设使用FR-FCFS策略)

参考答案：
应用A: MAX (M+(3M)+(4H+3M), M+(3H+3M)+4M) = 3H+8M = 45+360 = 405ns
应用B: MAX (M+(3M), M+(3H+3M)) = 3H+4M = 45+180 = 225ns
应用C: MAX (M, M) = M = 45ns



4 内存调度

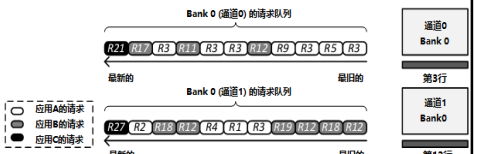
访存请求在读命令完成后被响应(即读命令被发射15ns之后)，每个应用(A、B或C)停顿直到它所有访存请求被响应为止。
假设初始时(t0时)，每个bank的第3行和第12行分别被取出并存入行缓冲，没有任何其他应用的请求到达内存控制器。

4.2 应用感知的调度策略

图中的3个应用，应用C是内存密集程度最低的(即有最少的请求数)。然而，它经历了最长的停顿时间，因为它的请求响应晚于其它多个被优先服务的应用的请求。为了保证应用C的停顿时间最短，可以为它的请求分配最高优先级，而给应用A和B的请求分配同样的低优先级。

(d) 请将四种调度策略 (FCFS, FR-FCFS, X, Y)的平均停顿时间从大到小排列

参考答案：
FCFS>FR-FCFS>X>Y



5 分层存储体系结构

假设你研究出了下一代的存储技术:“魔法RAM”。魔法RAM的位元是非易失性的；它的访问延迟是SRAM的2倍，与DRAM相同；读/写时的能耗和成本与DRAM相当；比DRAM的密度更高。然而，魔法RAM有一个缺点:每个位元在执行2000次写操作之后会停止运转。
(a) 相比DRAM，魔法RAM除了密度高之外，还有什么优势么？请解释。

参考答案：
是的。
魔法RAM不需要刷新，因为它是非易失性的。这可以降低动态功耗，总线的占用和bank的竞争。魔法RAM的非易失性还可能有利于新的使用模式或编程模型。

(b) 相比SRAM，魔法RAM有什么优势吗？请解释。

参考答案：
是的。
魔法RAM有更高的密度和更低的成本。

5 分层存储体系结构

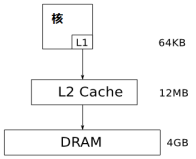
假设你研究出了下一代的存储技术:“魔法RAM”。魔法 RAM的位元是非易失性的；它的访问延迟是SRAM的2倍，与DRAM相同；读/写时的能耗和成本与DRAM相当；比DRAM的密度更高。然而，魔法RAM有一个缺点: 每个位元在执行2000次写操作之后会停止运转。

(c) 假设一个系统有64KB SRAM的L1 cache 、12MB SRAM的L2 cache和4GB DRAM的主存。

假设你可以利用这个分层存储结构，经过自由地设计和增加任何结构以克服魔法RAM的缺陷(除了修改魔法RAM本身)

(i) 可能将魔法RAM加入这个分层存储结构以减小它的缺陷吗?

参考答案:
是的。



261

5 分层存储体系结构

假设你研究出了下一代的存储技术:“魔法RAM”。魔法 RAM的位元是非易失性的；它的访问延迟是SRAM的2倍，与DRAM相同；读/写时的能耗和成本与DRAM相当；比DRAM的密度更高。然而，魔法RAM有一个缺点: 每个位元在执行2000次写操作之后会停止运转。

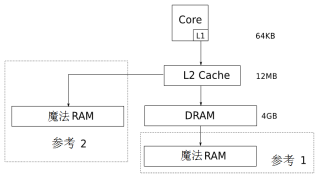
(c) 假设一个系统有64KB SRAM的L1 cache 、12MB SRAM的L2 cache和4GB DRAM的主存。

假设你可以利用这个分层存储结构，经过自由地设计和增加任何结构以克服魔法RAM的缺陷(除了修改魔法RAM本身)

(ii) 如果可能, 魔法RAM该放到哪里? 根据上面的图来说明，并说明为什么选择放在这个位置。

如果不可能，为什么? 请解释。

参考答案:
可能的正确答案不止一种。
其中一种: 在存储的层次结构中，将魔法RAM放置于DRAM之下，利用 DRAM作为魔法RAM的cache 。这样，由DRAM执行 更多的写操作，使魔法RAM不会过快的磨损。
另一种是把魔法RAM与 DRAM并排放置 (相同或不同的通道上)，利用魔法RAM显式地处理只读数据。



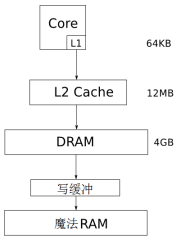
262

5 分层存储体系结构

假设你研究出了下一代的存储技术:“魔法RAM”。魔法 RAM的位元是非易失性的；它的访问延迟是SRAM的2倍，与DRAM相同；读/写时的能耗和成本与DRAM相当；比DRAM的密度更高。然而，魔法RAM有一个缺点: 每个位元在执行2000次写操作之后会停止运转。

(d) 请给出一种通过修改这个分层存储结构以减少或克服魔法RAM缺陷的方法。请简单清晰地说明你的方法，可以利用图示来说明问题。

参考答案:
采用上述参考1的方案，需要增加一个联合写缓冲的组件以减少写操作对魔法RAM的损耗。同时，存储层次结构中也应该提供一些损耗均衡的机制，或者预测哪些数据被修改的可能性低，将这些数据存入魔法RAM。



263

作业6——预取和并行
参考答案

264

1 预取 I

假如你是一位架构师，正在为你的机器设计预取引擎。你先在机器上使用跨度预取器执行了A和B两个应用。

应用A:
```
uint8_t  a[1000];
sum = 0;
for (i = 0; i < 1000; i += 4)
{
sum += a[i];
}
```

应用B:
```
uint8_t  a[1000];
sum = 0;
for (i = 1; i < 1000; i *= 4)
{
sum += a[i];
}
```

i 和 sum在寄存器中，数组a在内存中，一个cache块大小为4个字节。

(a) 使用跨度预取器，应用A和B的预取精度和覆盖率分别是多少？这个跨度预取器检测两次连续访存的跨度，从当前访问的cache块按照这个跨度预取下一个cache块。

参考答案：

应用 A的预取精度是 248/249，覆盖率是 248/250。

应用 A 访问 a[0], a[4], a[8], … a[996]，有1000/4 = 250次访存。前两次访问 a[0]和 a[4] 不命中，之后，预取器学习到跨度是4并开始预取a[8], a[12], a[16]等等直到 a[1000] (访问 a[996]时 a[1000] 被预取，虽然并没有被用到)。统计结果，249个 cache块被预取，248个被使用。

因此，预取精度为248/249，覆盖率为248/250。

应用 B的预取精度为0，覆盖率为0。

应用B访问 a[1], a[4], a[16], a[64]和a[256]，有5次访存。然而，由于这些访问的跨度不是常数，因为数组索引是4的倍数，而不是增或者减一个常数。因此，跨度预取器无法预取到被访问的cache块，使得预测精度和覆盖率均为0。

265

1 预取 I

假如你是一位架构师，正在为你的机器设计预取引擎。你先在机器上使用跨度预取器执行了A和B两个应用。

应用A:
```
uint8_t  a[1000];
sum = 0;
for (i = 0; i < 1000; i += 4)
{
sum += a[i];
}
```

应用B:
```
uint8_t  a[1000];
sum = 0;
for (i = 1; i < 1000; i *= 4)
{
sum += a[i];
}
```

i 和 sum在寄存器中，数组a在内存中，一个cache块大小为4个字节。

(b) 请分别为应用A和B建议能获得更好的精度和覆盖率的预取器

i) 应用A

参考答案：

下一块预取器总是预取下一个cache块，因此，有a[4] 的cache块也会被预取，则预取精度提高到 249/250并且覆盖率仍保持249/250。

ii) 应用B

参考答案：

大多数普通的预取器比如跨度、流、下一块等都无法提升应用B的预取精度，因为无法为这些预取器提供一个合适的访存模式。某些采用预执行方法的预取，比如双核执行可能能够改进应用B的预取精度。

266

2 预取II

你跟你的同学一起设计一个预取器，这台机器使用单核、 L1 和 L2 cache 以及 DRAM 内存。我们需要分析不同的预取器和可能的tradeoff。

在本题中，我们要计算预取器在达到稳定状态后的预取精度、覆盖率和带宽开销，所以，**所有计算都不包括最开始的6次请求，这6次请求作为预取器的训练集。**

(a)你首先设计一个跨度预取器，观察最后三次cache块请求，如果最后三次请求的跨度是常数，预取器将会使用这一跨度预取下一个cache块。

你执行了一个应用，它具有如下的访存模式 (这些是cache块地址):

A A+1 A+2 A+7 A+8 A+9 A+14 A+15 A+16 A+21 A+22 A+23 A+28 A+29 A+30...

假设这个模式持续了很长时间。

计算你的跨度预取器对于这个应用的精度和覆盖率

参考答案：

0%，0%。

每三个一组的请求之后，预取按照检测到的跨度触发，但是预取到的块总是无用的；需要的请求不会被这个预取器的预取覆盖。

267

2 预取II

你跟你的同学一起设计一个预取器，这台机器使用单核、 L1 和 L2 cache 以及 DRAM 内存。我们需要分析不同的预取器和可能的tradeoff。

在本题中，我们要计算预取器在达到稳定状态后的预取精度、覆盖率和带宽开销，所以，**所有计算都不包括最开始的6次请求，这6次请求作为预取器的训练集。**

(b) 你的同学设计了一个新的预取器，当有一个cache块访问时，预取紧接着的N个cache块

(i) 如果用他的预取器执行你刚刚执行过的应用，预取覆盖率和精度分别是66.67%和50%，N是多少?

参考答案：

N= 2。

比如在访问块14之后，预取器预取块15和16，访问15之后预取16 (与已经发射的预取整合) 和17，访问16之后预取17和 18。因此，每三个需要的访问中的两个被覆盖(66.7%)，并且预取的数据一半是有用的 (50%)。

(ii) 假如我们将带宽开销定义为：有预取器时所有cache块的请求数/没有预取器时所有cache块的请求数，那么你同学的预取器在执行刚才那个应用时的带宽开销是多少？

参考答案：

5/3。

对于每一组连续三个访问的cache块，有两个额外的块被预取。比如，取cache块14, 15和16，块17和18也会被预取。

268

2 预取II

你跟你的同学一起设计一个预取器，这台机器使用单核， L1 和 L2 cache 以及 DRAM 内存。我们需要分析不同的预取器和可能的tradeoff。
在本题中，我们要计算预取器在达到稳定状态后的预取精度、覆盖率和带宽开销，所以，**所有计算都不包括最开始的6次请求，这6次请求作为预取器的训练集。**
(c) 你的同学希望改进他的预取器对于刚才那个应用的覆盖率，他可以容忍带宽开销最多两倍。请问他能做到吗？为什么可以/不可以？

参考答案：
不可以。
要获得更好的覆盖率，预取器必须能够跨过上一组3个请求取到下一组中的请求，因为上一组的三个请求都已经在前一次预取到了。比如，访问A+14, A+15 和A+16，它们都已经被预取了，要想提高覆盖率，需要预取到A+21 (下一组三个跨度请求中的第一个)，但是，这需要预取 A+16和 A+21之间 的四个块(A+17, A+18, A+19, A+20)，增加的带宽开销超过两倍。

(d) 对于上面的应用，如果想获得100%的覆盖率，N最小得是多少？这个时候的带宽开销是多少?

参考答案：
N= 5 (这样， A+16 预取 A+21， A+21预取A+22, A+23等等)；
带宽开销是7/3。

269

3 Cache 一致性

(a) MESI cache一致性协议比MSI 协议好在哪里?

参考答案：
允许cache/处理器写一个位置(唯一的干净拷贝)而不需要通知其它的处理器/cache。

(b) 你想要利用MESI置无效协议设计一个基于目录的cache一致性系统，在特定的工作负载下，系统表现得很糟糕，经过仔细的分析，你发现有4个节点持续的发出对某个cache块的置无效请求，这什么情况?

参考答案：
4个节点间发生cache 块乒乓现象。

(c) 如何解决这一问题?

参考答案：
如果现象是由真共享导致的，重写代码减少共享或者使用同步原语减少通信。如果搞不定，可以考虑使用基于更新的一致性协议。
如果是由伪共享导致的，通过编译器或重写代码改变数据的布局，以消除伪共享。

270

4 一致性协议

假设有一个多处理器系统，系统有512个处理器，每个处理器有1MB的私有写回方式的cache，每个cache块64字节，主存大小为1GB。
(a)如果我们基于MESI cache 一致性协议设计了监听总线，需要多少状态位才能够实现这个一致性协议? 这些状态位放在哪?

参考答案：
2^{24}位。
总共有2^{23} 个cache块 (cache有2^{20} 字节，2^9个 cache，因此有2^{29}字节 在私有 cache中；用每个cache块2^6字节去除)，每块需要2位表示状态 (M, E, S或 I)，因此需要2^{24}位。
这些位存在于私有cache的标签存储中。

271

4 一致性协议

假设有一个多处理器系统，系统有512个处理器，每个处理器有1MB的私有写回方式的cache，每个cache块64字节，主存大小为1GB。
(b) 如果用基于目录的cache一致性协议（像我们课堂上讲的例子那样）替换，需要多少状态位? 这些状态位在哪?

参考答案：
2^{24} x 513 + 2^{24} 位
仍然需要在私有cache中的2^{24}个MESI状态位，然后，必须计算目录存储空间。共有2^{24} cache块在主存中，每个块需要对应每个处理器1位外加1个独占位(每个块513位)。
因此目录中一共需要2^{24} x 513位。
目录位: 存在于cache 目录；MESI状态位: 存在于私有cache。

272

4 一致性协议

假设有一个多处理器系统，系统有512个处理器，每个处理器有1MB的私有写回方式的cache，每个cache块64字节，主存大小为1GB。
(c) 对于这个系统，你会选择哪一个协议？为什么？

参考答案：
目录。
总线无法扩展到512个处理器，目录可以。
虽然基于总线的监听系统对于存储的需求要低很多，但是总线无法提供足够的带宽来维持512个处理器的需要。因此，基于目录的系统(用可扩展的互连网络构建)更合适。

273

5 并行加速比

假如你是一家公司的程序猿，你被要求并行化一个老程序以使它能够在现代多核处理器上跑得更快。
(a) 你并行化了这个程序，然后发现它对于单线程版本的加速比相比于处理器个数的增加而言相差很多。你发现在每个核的数据cache中有大量的cache无效存在，什么样的程序行为导致了这种现象？(请用10个字左右简要说明)
参考答案：
由数据共享导致的cache乒乓
(b)你修改了程序以解决这个性能问题，然后你发现 程序在每个并行计算之后的一个单个线程都会更新一个全局状态，因此导致性能的下降。你的程序有90%的工作是并行的(按照处理器个数x秒计算得出)，另有10%的工作是串行的，并行部分是完美的并行。如果多核处理器核数无限，程序的最大加速比可以到多少？
参考答案：
10。
根据 Amdahl定律: 对于n个处理器，加速比(n) = 1/(0.1 + 0.9/n)
由于n→∞， 加速比(n)→10。
(c) 如果要获得4倍的加速比，应该有多少处理器？
参考答案：
6。
由 加速比(n) = 4:
4 = 1/(0.1 + 0.9/n) → 0.25 = 0.1 + 0.9/n → 0.15 = 0.9/n， ∴ n= 6

274

5 并行加速比

(d) 为了使你改写的程序更高效，公司决定设计一款专门的异构处理器。这款处理器由一个大核(执行代码更快，但是占据的片上面积更大)和多个小核(执行代码更慢，但是消耗面积更小)共享处理器的片上空间。
你的程序并行部分的所有线程将只会在小核上执行；程序的串行部分将有一个线程执行在大核上。核的性能(执行速度)与它的面积的平方根成正比。
假设芯片面积有16个单元可用，一个小核至少占用1个单元，大核可以占用任意数量的单元。同时假设没有被大核使用的面积会被小核填满。
(i) 如果想让你的程序获得可能的最快执行速度，大核需要多大？
参考答案：
4个单元。
如果给定大核的尺寸是n^2，则大核在串行段的加速比是n，16-n^2个小核实现并行段的并行化。这样加速比= 1/(0.1/ n + 0.9/(16- n^2))。为了最大化加速比，需要最小化分母。对于n=1，分母是0.16；n=2，分母是0.125；n=3，分母是0.1619 。因此n=2是最佳的，所以大核占据n^2 = 4个单元。
(ii) 如果所有16个单元全部拿来用做小核，这个处理器就变成了同构的多核处理器，对于你的程序而言，它的加速比是多少？假设串行部分跑在一个小核上，并行部分跑在所有16个小核上。
参考答案：
6.4。。。加速比= 1/(0.10 + 0.90/16) = 6.4
(iii) 在串行部分是10%的情况下，使用异构多核(大小核)处理器是有意义的吗？为什么是/不是？
参考答案：
是。
因为串行部分足够大，使得大核在串行部分获得的加速比超过了由于大核带来的并行吞吐的下降

275

5 并行加速比

(e) 现在你继续优化了你的程序，使得串行部分仅占4%(剩下96%是并行部分)。
(i) 这个时候大核应该有多大(占多少单元)？
参考答案：
2 个单元
跟之前的题类似，加速比= 1/(0.04/ n + 0.96/(16- n^2))。最小化分母以最大化加速比。
n=√2时最大，因此大核占 2个单元。
(ii) 大核这么大的时候加速比是多少？
参考答案：
10。。。加速比= 1/(0.04/ √2 + 0.96/14) = 10.32
(iii) 假如此时我们采用16个小核的同构多核处理器，你的程序的加速比是多少(假设串行部分跑在一个小核上，并行部分跑在所有16个小核上)？
参考答案：
10。。。加速比= 1/(0.04/1 + 0.96/16) = 10
(iv) 在串行部分是4%的情况下，使用异构多核(大小核)处理器还是有意义的吗？为什么是/不是？
参考答案：
是 and 不是。
虽然仍能获得比同构系统略高的性能，但是异构系统如果无法提供比同构系统显著的性能收益，由于它比同构系统设计复杂得多，也将意义不大。

276