高等计算机体系结构
第十四讲: 虚拟存储(续)/性能与能耗

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
2021-05-28

1

提醒：作业

- 作业 5
  - 今天截止
  - Cache和Memory
- 作业 6
  - 今晚发布，6月11日截止
  - 预取和并行

2

2

实验2-5

- 已发布，预计7月16日截止

3

3

阅读材料

- 虚拟存储
- 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：5.4 ,5.8
- 推荐阅读
  - Denning, P. J. *Virtual Memory*. ACM Computing Surveys. 1970
  - Jacob, B., & Mudge, T. *Virtual Memory in Contemporary Microprocessors*. IEEE Micro. 1998

4

4

回顾：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)

5

5

回顾：相变存储器: 优点和缺点

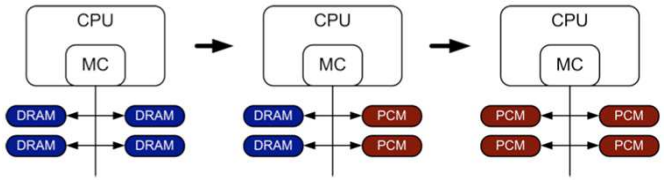
- 优于DRAM之处
  - 更好的工艺规模(容量和成本)
  - 非易失
  - 空闲时功率低 (无需刷新)
- 缺点
  - 延迟更高: ~4-15x DRAM (尤其是写入时)
  - 活跃状态能耗更高: ~2-50x DRAM (尤其是写入时)
  - 重复使用寿命较低 (位元寿命~10^8 次写入)
- 用PCM替换或者协助DRAM组成主存储器的挑战:
  - 减小PCM缺陷的影响
  - 找到合适的方式将PCM引入系统

6

6

回顾：基于PCM的主存储器

- 基于PCM的(主)存储器如何组织?



- 混合PCM+DRAM [Qureshi+ ISCA'09, Dhiman+ DAC'09]:
  - 在PCM和DRAM之间如何分区/迁移数据
- 纯 PCM的主存储器 [Lee et al., ISCA'09, Top Picks'10]:
  - 如何重新设计整个层次结构(包括核)以克服PCM的缺点

7

7

回顾：基于PCM的存储系统：研究上的挑战

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

8

8

回顾：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

9

9

回顾：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

10

10

回顾：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)

11

11

回顾：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.....

12

12

回顾：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

13

13

回顾：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

14

14

回顾：如何预取不规则访问模式?

- 规则模式: 跨度, 流预取器
- 不规则访问模式
  - 间接数组访问
  - 链式数据结构
  - 多跨度(1,2,3,1,2,3,1,2,3,…)
  - 随机模式?
  - 针对所有模式的通用预取器?
- 基于相关性的预取器
- 基于内容的预取器
- 基于预计算或预执行的预取器

15

15

回顾：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

16

16

回顾：基和界

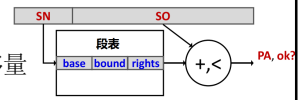
- 一个进程的私有存储区域被定义为
  - 基：该区域的首地址
  - 界：该区域的大小
- 基和界寄存器
  - 翻译和保护机制针对每一次用户的内存访问检查硬件
  - 每次切换用户进程，操作系统设置基和界寄存器
  - 用户进程不能自行修改基和界寄存器
- 一组“基和界”是保护机制起作用的基本单元
  →给用户多个内存“段”
  - 每个段是连续的存储区域
  - 每个段由一对基和界定义

17

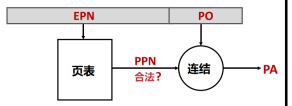
17

回顾：分段和分页

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



分成几个大段有利于隔离管理



分成很多个页有利于分配管理

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

18

18

碎片

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

段和页扮演着不同的角色

19

19

请求页面调度

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

20

20

请求页面调度——就像Cache一样

- M字节的内存空间，其中最常用的C字节保存在DRAM C<<M
- 和以前一样的基本问题
  - (1)在DRAM的什么位置“缓存”页面？
  - (2)如何在DRAM中找到一个页面？
  - (3)什么时候把一页放进DRAM？
  - (4)将哪个页面从DRAM置换到磁盘，释放DRAM用于新页面？
- 关键概念差异：交换vs.缓存
  - DRAM不保存磁盘上的内容
  - 一页要么在DRAM中要么在磁盘上
  - 地址不会始终绑定到一个位置

21

21

请求页面调度:一点也不像Cache

- 规模和时间尺度的差异导致完全不同的实现选择

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

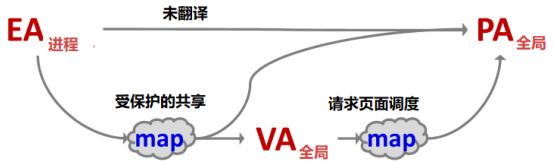
命中时间、缺失开销、缺失率都不是互相独立的指标

22

22

不要用“虚存”来表示一切

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



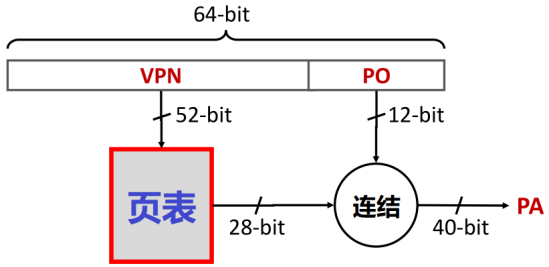
23

23

页表有多大？

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

  2^{52} x ~4 字节≈16x10^{15} 字节



这只是一个进程的页表

24

24

页表应该有多大？

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

  今天主要有两种使用模式:

  分层页表和哈希页表

25

快表-TLB

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

  标签：地址tag(来自VA), ASID

  页表项(PTE)：PPN和保护位

  其它：有效位、脏位等

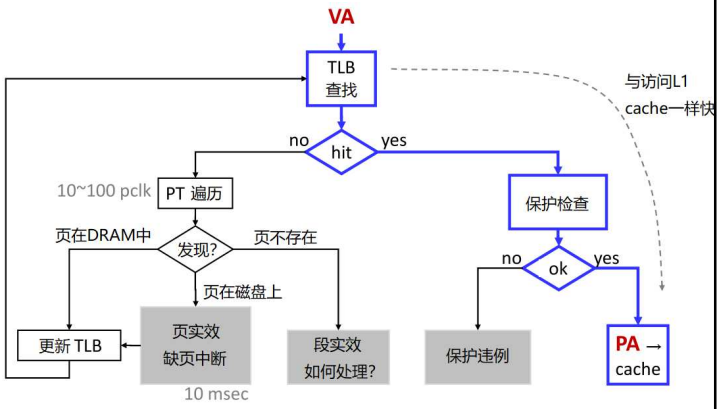
26

TLB设计

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

27

VA→PA翻译流程



28

虚存和cache交互

29

地址翻译与cache

- 什么时候需要做地址翻译?
  - 访问L1 cache之前还是之后?
- 换句话说, cache是虚拟编址还是物理编址?
  - 虚拟 vs 物理 cache
- 即使有TLB，翻译也需要时间
- 简单地讲，最佳情况下的内存访问时间是
  TLB命中时间+ cache命中时间
- 为什么不使用虚拟地址访问cache；只在cache miss 要访问DRAM时才做地址翻译
  如果TLB命中时间>> cache命中时间就有意义
- 大约在1990年，SUN SPARC 指令集的虚拟cache
  - 处理器速度足够快，片外SRAM访问需要多个周期
  - 芯片尺寸足够大可以设计片内L1 cache
  - MMU和TLB在不同的芯片上

30

30

虚拟cache中的同义词和同音异义词

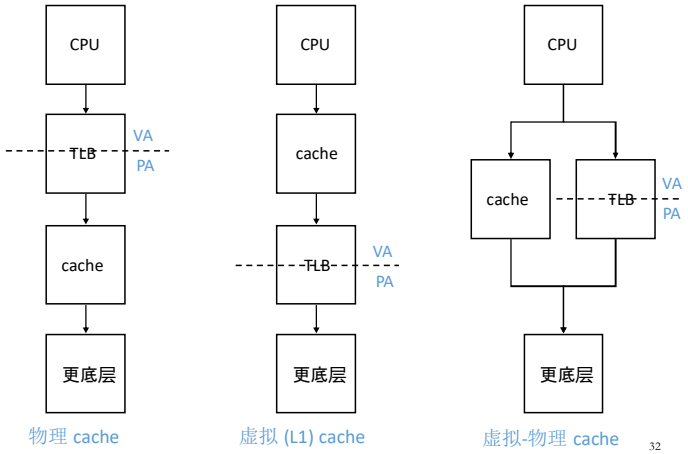
- 同音异义：相同的声音不同的含义
  - 相同的EA(不同进程中)→不同的PA（同一个虚拟地址可能映射到两个不同的物理地址）
    - 虚拟地址可能在不同的进程中
  - 在上下文之间刷虚拟cache，或者在cache标签中加入ASID
- 同义词：不同的声音相同的含义（不同的虚拟地址可能映射到同一个物理地址）
  - 不同的EA(相同或不同的进程)→相同的PA
    - 不同的页可能共享同一进程内或者跨进程的物理帧
    - 原因: 共享库, 共享数据, 同一个进程内的写时拷贝页, …
  - PA可以由不同的EA cache两次
  - 写一个cache的副本不会反映在另一个cache的副本中

  要确保一次仅在cache中有一个这样的EA

31

31

Cache-VM 如何交互

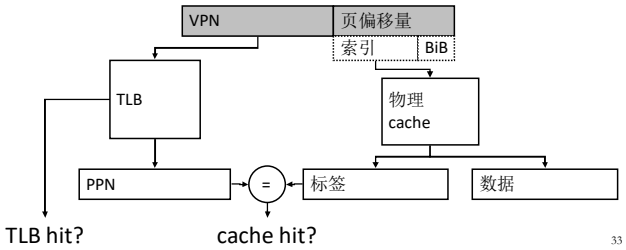


32

32

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

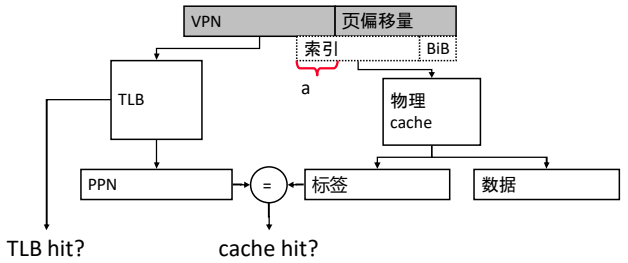


33

33

虚拟-索引,物理-标签

- 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
  - 同一个物理地址可能存在于两个不同的位置
- 如何解决?



34

34

解决“同义词”问题的一些方法

- 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

35

35

深入思考的问题

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

36

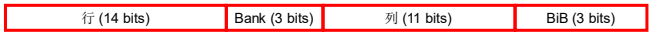
36

虚存和内存交互

37

内存地址映射(单通道)

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



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

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

38

38

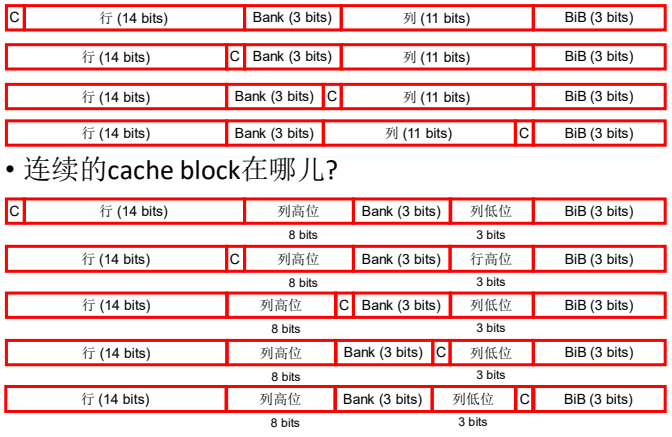
Bank 随机映射

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

39

39

内存地址映射(多通道)



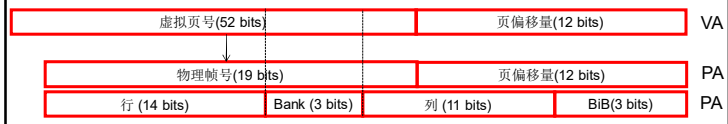
- 连续的cache block在哪儿?

40

40

虚拟存储– DRAM 的交互

- 操作系统会影响DRAM中的地址映射



- 操作系统能够控制虚页映射到哪一个bank/channel/rank

- 可以通过页着色最小化bank的冲突
- 或者最小化应用之间的干扰

41

41

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)

42

42

性能分析

43

43

例子：单周期性能分析

MIPS不同类型指令的指令周期

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

数据通路各部分以及各类指令的执行时间

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

44

例子：单周期性能分析

- 指令执行时间计算
  1. 方式一：采用单周期，即所有指令周期固定为单一时钟周期
     - 时钟周期有最长的指令决定（LW指令），为 600ps
     - 指令平均周期 = 600ps
  2. 方式二：不同类型指令采用不同指令周期（可变时钟周期）
     - 假设指令在程序中出现的频率
       - **lw指令 ：25%**
       - **sw指令 ：10%**
       - **R类型指令：45%**
       - **beq指令 ：15%**
       - **j指令 ：5%**
     - 平均指令执行时间

600*25%+550*10%+400*45%+350*15%+200*5% = 447.5ps

—若采用可变时钟周期，时间性能比单周期更高；
—但控制比单周期要复杂、困难，得不偿失。
—改进方法：改变每种指令类型所用的时钟数，即采用多周期实现

45

例子：多周期设计

- 为什么不使用单周期实现方式
  - 单周期设计中，时钟周期对所有指令等长。而时钟周期由计算机中可能的最长路径决定，一般为取数指令。但某些指令类型本来可以在更短时间内完成。
- 多周期方案
  - 将指令执行分解为多个步骤，每一步骤一个时钟周期，则指令执行周期为多个时钟周期，不同指令的指令周期包含时钟周期数不一样。
  - 优点：
    - 提高性能：不同指令的执行占用不同的时钟周期数；
    - 降低成本：一个功能单元可以在一条指令执行过程中使用多次，只要是在不同周期中（这种共享可减少所需的硬件数量）。

46

例子：多周期性能分析

- 假设主要功能单元的操作时间
  - 存储器：200ps
  - ALU：100ps
  - 寄存器堆：50ps
  - 多路复用器、控制单元、PC、符号扩展单元、线路没有延迟

各类指令执行时间

步骤	R型指令	Lw指令	Sw指令	Beq指令	J指令	执行时间
取指令	IR ← M[PC]，PC ← PC + 4					200ps
读寄存器/译码	A ← R[IR[25:21]]，B ← R[IR[20:16]] ALUOut ← PC + Signext[IR[15:0]]<<2					100ps
计算	ALUOut ← A op B	ALUOut ← A + Signext(IR[15:0])		If (A-B==0) then PC ← ALUout	PC ← PC[31:28] || IR[25:0]<<2	100ps
R型完成/访问内存	R[IR[15:11]] ← ALUOut	DR ← M[ALUOut]	M[ALUOut] ← B			200ps
写寄存器		R[IR[20:16]] ← DR				50ps

47

例子：多周期性能分析

- 时钟周期
  - 时钟周期取各步骤中最长的时间，200ps

各类指令执行时间

时钟周期	R型指令	Lw指令	Sw指令	Beq指令	J指令	周期时间
TC1	IR ← M[PC]，PC ← PC + 4					200ps
TC2	A ← R[IR[25:21]]，B ← R[IR[20:16]] ALUOut ← PC + Signext[IR[15:0]]<<2					200ps
TC3	ALUOut ← A op B	ALUOut ← A + Signext(IR[15:0])		If (A-B==0) then PC ← ALUout	PC ← PC[31:28] || IR[25:0]<<2	200ps
TC4	R[IR[15:11]] ← ALUOut	DR ← M[ALUOut]	M[ALUOut] ← B			200ps
TC5		R[IR[20:16]] ← DR				200ps

48

例子：多周期性能分析

- 各型指令所需的时钟周期数和时间
  - R型指令：800ps
  - lw指令 ：1000ps
  - sw指令 ：800ps
  - beq指令 ：600ps
  - j指令 ：600ps
- 假设指令在程序中出现的频率
  - lw指令 ：25%
  - sw指令 ：10%
  - R型指令：45%
  - beq指令：15%
  - j指令 ：5%

  则一条指令的平均CPI
  - 5*25%+4*10%+4*45%+3*15%+3*5% = 4.05
- 一条指令的平均执行时间：
  - 1000*25%+800*10%+800*45%+600*15%+600*5% = 810ps

49

流水线性能分析

- 获得什么收益？
- 付出什么代价？

50

性能

——所有的一切几乎都和**时间**有关

51

不是所有的时间都是生来平等的

- 例子：UNIX系统的程序运行时间
  - 用户CPU时间：运行代码所花费的时间
  - 系统CPU时间：为运行用户代码而运行其它代码所花费的时间
  - 运行时间：墙钟时间
  - 运行时间–用户CPU时间–系统CPU时间=运行其它无关代码的时间

    注意：实际系统测量值有差异
- 经验法则
  - 不要欺骗自己：搞清楚要衡量什么和实际测量的是什么
  - 不要愚弄他人：要准确说明衡量了什么以及是如何测量的
  - 最佳选择：在没有其它负载的系统上多次测量实际完成工作负载的墙钟时间

52

“性能”通常的定义

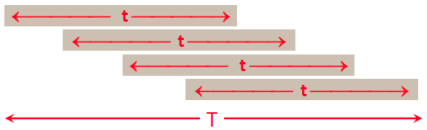
- 首先，性能∝ 1/时间
- 两种**截然不同**的性能！
  - 延迟=任务开始和完成之间的时间
  - 吞吐量=在给定时间单位内完成的任务数(速率度量)
  - 不要混淆
- 不管怎样，时间越短，性能越高，但是……

53

53

吞吐量 ≠ 1/延迟！

- 如果执行N个任务需要花费T秒，吞吐量= N/T；
  延迟=T/N吗？
- 如果完成一项任务需要t秒，延迟= t；
  吞吐量=1/t吗？
- 当有并发时，吞吐量 ≠ 1/延迟



- 可以通过两者的权衡进行优化

54

54

吞吐量 ≠ 吞吐量

- 当存在非经常启动开销时，吞吐量是N（任务量）的函数
- 若启动时间=t_s，吞吐量$_{原始}$=$1/t_1$
  - 吞吐量$_{有效}$= $N / (t_s + N \cdot t_1)$
  - 若$t_s >> N \cdot t_1$，吞吐量$_{有效} \approx N/t_s$
  - 若$t_s << N \cdot t_1$，吞吐量$_{有效} \approx 1/t_1$

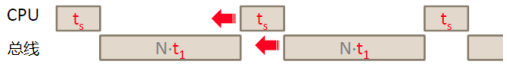
  在后一种情况下，我们说t_s被“平摊”了
- 例子：总线上的DMA传输
  - 10^{-6}秒对DMA引擎进行初始化
  - 总线吞吐量$_{原始}$ = 1G字节/秒= 1字节/ (10^{-9}秒)
  - 那么传输1B，1KB、1MB、1GB的吞吐量$_{有效}$分别是多少？

55

55

延迟 ≠ 延迟

- 延迟的时候“你”在做什么？
- 延迟=实际操作时间+等待时间
- DMA的例子中
  - CPU消耗t_s对DMA引擎进行初始化
  - CPU必须消耗$N \cdot t_1$等待DMA完成
  - CPU在$N \cdot t_1$期间可以做些其它事情来“隐藏”延迟



56

56

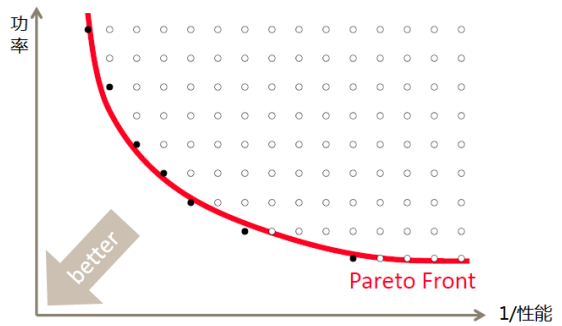
什么情况下不仅仅与时间有关？

- 除了性能，还有其它重要的指标：功率/能量、成本、风险、社会因素…
- 如果不考虑它们之间的权衡，无法优化单个指标
- 例如运行时间与能量
  - 可能愿意在每个任务上花费更多能量来加快运行速度
  - 相反，可能愿意让每个任务执行的更慢换取更低的能耗
  - 但是永远不要消耗更多的能量而跑得更慢

57

57

帕累托最优（Pareto Optimality）



所有在前沿的点都是最佳的(不能做得更好)
如何在它们之间进行选择？

58

58

复合指标

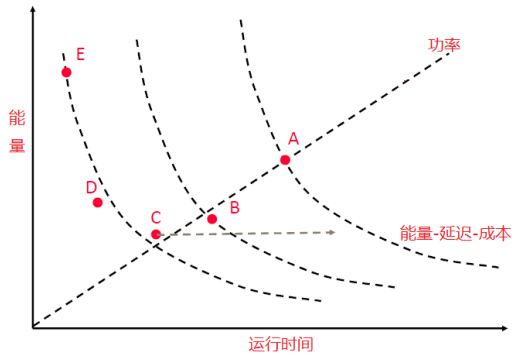
- 定义标量函数来反映需求——整合维度及其关系
- 例子，能量-延迟-成本
  - 越小越好
  - 不能最小化一个忽略另一个
  - 不需要有物理意义
- 地板和天花板
  - 现实生活中的设计往往是足够好，但不是最佳
  - 例如，满足功率(成本)上限下的性能基本要求

59

59

哪个设计点是最佳的？

考虑运行时间、功率、能量、能量-延迟-成本



60

60

“伪”性能

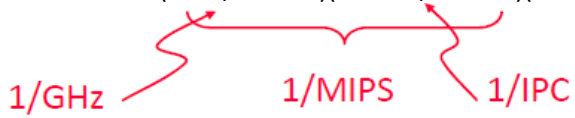
- 最有可能在宣传中看到的指标
  - IPC(每周期执行指令数)
  - MIPS(每秒百万条指令数)
  - GHz(每秒周期数)
- “听起来”像是性能，但不完整而且可能有误导
  - MIPS和IPC是平均值(取决于指令组合)
  - GHz、MIPS或IPC可以在牺牲彼此和实际性能的情况下得到改进

61

61

性能的铁律

- 墙钟时间=(时间/周期数)(周期数/指令数)(指令数/程序)



- 各影响因子
  (时间/周期数)受体系结构+实现影响
  (周期数/指令数)受体系结构+实现+工作负载影响
  (指令数/程序)受体系结构+工作负载影响
- 注：(周期数/指令数)是工作负载平均值
  由于指令类型和序列导致潜在的瞬时剧烈变化

62

62

不仅与硬件相关

- 算法通过(指令数/程序)对性能有直接影响，例如离散傅立叶变换
  - 矩阵乘法导致$2N^3$的浮点运算
  - 用快速算法只需要$5N\log_2(N)$的浮点运算

  如果N=1024，$2N^3 \approx 2x10^9$ vs. $5N\log_2(N) \approx 5x10^4$
- 更抽象的编程语言可以产生更高的(指令数/程序)
- 编译器优化的质量影响(指令数/程序)和(周期数/指令数)

63

63