高等计算机体系结构
第八讲: 流水线设计中的指令调度

栾钟治
北京航空航天大学 计算机学院 中德联合软件研究所

1

阅读材料

- Patterson & Hennessy's *Computer Organization and Design: The Hardware/Software Interface* （计算机组成与设计：软硬件接口）
  - 第四章 (4.9-4.11)
- 选读
  - Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
    - 更高级的流水线
    - 中断和异常处理
    - 乱序和超标量执行的概念

2

回顾：解决方案

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

3

回顾：解决方案

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

4

回顾：解决方案

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

5

5

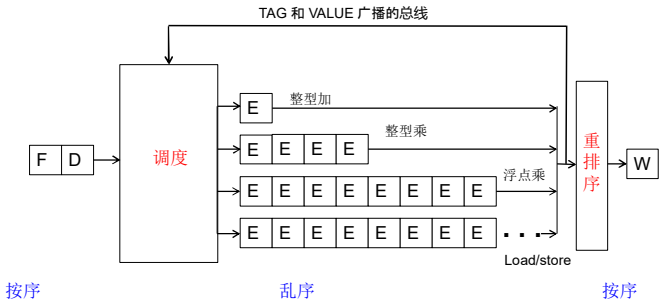
回顾：解决方案

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

6

6

回顾：现代流水线的两个“驼峰”

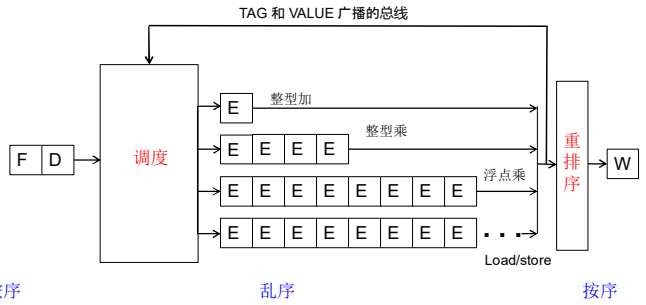


- 驼峰 1: 保留站(调度窗口)
- 驼峰 2: 重排序(ROB, 又叫指令窗口或者动态窗口)

7

7

回顾：带精确异常的乱序执行



- 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)

8

8

回顾：乱序执行概念小结

- 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

9

9

回顾：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”

10

10

思考

- 为什么乱序执行是有益的?
  - 如果所有的操作都占用1个时钟周期会怎么样?
  - 延迟容忍: 乱序执行能够通过并发执行独立的操作容忍多周期操作的延迟
- 如果一条指令要花费500个周期会怎么样?
  - 需要多大的指令窗口才能保证持续的译码?
  - 乱序执行能够容忍多少个周期的延迟?
  - 是什么限制了Tomasulo算法的延迟容忍的可扩展性?
    - 动态/指令窗口大小: 由寄存器堆、调度窗口、重排序缓冲等决定

11

11

回顾：寄存器 vs. 存储器

- 到目前为止, 我们考虑的都是指令之间基于寄存器的值的通信
- 存储器会怎么样?
- 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)

12

12

处理存储相关性(I)

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

处理存储相关性(II)

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

处理Store-Load相关性

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

存储违例消解(I)

- 选项 1: 假设 load 与所有前序 store 相关

  + 不需要恢复

  -- 太保守: 对独立的 load 施加了不必要的延迟

- 选项 2: 假设 load 与所有前序 store 不相关

  + 简单并且可能是常见的情况: 独立的 load 没有延迟

  --预测错误需要恢复/重新执行

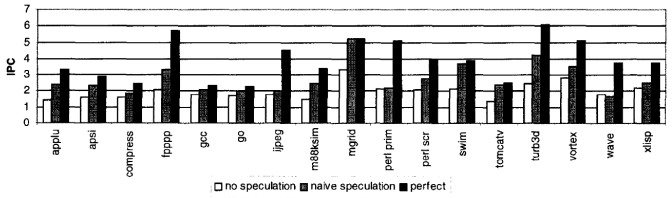
- 选项 3: 预测 load 与未完成的store 的相关性

  + 更准确

  -- 预测错误还是需要恢复/重新执行
  - Alpha 21264 ：先假设 load 是独立的，当发现相关之后延迟 load
  - Moshovos et al., "Dynamic speculation and synchronization of data dependences," ISCA 1997.
  - Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.

存储违例消解(II)

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



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

17

17

思考

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

18

18

思考

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

19

19

推荐阅读

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

20

20

获得(指令级)并行的(其它)方法

- 流水线执行
- 乱序执行
- 数据流 (ISA层面)
- SIMD处理
- VLIW
- 脉动阵列
- 解耦访问/执行

21

21

获得(指令级)并发的(其它)方法

- 流水线执行
- 乱序执行
- 数据流 (ISA层面)
  - 利用不规则的并行
- SIMD处理
- VLIW
- 脉动阵列
- 解耦访问/执行

22

22

获得(指令级)并发的(其它)方法

- 流水线执行
- 乱序执行
- 数据流 (ISA层面)
- SIMD处理
  - 利用规则的(数据)并行
  - 向量处理
  - SIMT/GPU
- VLIW
- 脉动阵列
- 解耦访问/执行

23

23

获得(指令级)并发的(其它)方法

- 流水线执行
- 乱序执行
- 数据流 (ISA层面)
- SIMD处理
- VLIW
  - 超长指令字
  - 编译器发现指令级并行，硬件尽可能简单
  - 静态调度
- 脉动阵列
- 解耦访问/执行

24

24

获得(指令级)并发的(其它)方法

- 流水线执行
- 乱序执行
- 数据流 (ISA层面)
- SIMD处理
- VLIW
- 脉动阵列
  - 数据流出入存储器的节奏
  - 处理单元的规则阵列
  - 不同于常规的流水线
- 解耦访问/执行

25

获得(指令级)并发的(其它)方法

- 流水线执行
- 乱序执行
- 数据流 (ISA层面)
- SIMD处理
- VLIW
- 脉动阵列
- 解耦访问/执行
  - DAE
  - 解决Tomasulo算法过于复杂的问题
  - 两个独立的指令流分别处理操作数的访问和执行
  - ISA可见的队列
  - 编译器
  - 指令流同步

26

静态调度

- 更好的静态调度: 更大的块
  - 推断执行
  - 循环展开
  - 基于Trace的调度
  - 基于Superblock的调度
  - 基于Hyperblock的调度
  - 块结构的 ISA

27

推荐阅读

- Gurd et al., "The Manchester prototype dataflow computer," CACM 1985.
- Hwu and Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," ISCA 1986.
- Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.
- Huck et al., "Introducing the IA-64 Architecture," IEEE Micro 2000.
- Russell, "The CRAY-1 computer system," CACM 1978.
- Rau and Fisher, "Instruction-level parallel processing: history, overview, and perspective," Journal of Supercomputing, 1993.
- Faraboschi et al., "Instruction Scheduling for Instruction Level Parallel Processors," Proc. IEEE, Nov. 2001.

28

系统性能分析：概念和基本方法

29

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

30

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

31

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）。

32

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

33

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

34

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

35

流水线性能分析

- 获得什么收益？
- 付出什么代价？

36

36

性能

——所有的一切几乎都和**时间**有关

37

37

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

38

38

“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……

39

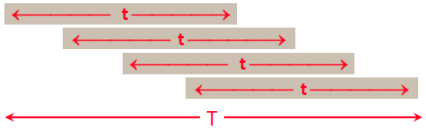
39

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

40

40

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$_{有效} \approx 1/t_1$

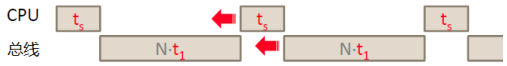
  在后一种情况下，我们说t_s被“平摊”了
- 例子：总线上的DMA传输
  - 10^{-6}秒对DMA引擎进行初始化
  - 总线吞吐量$_{原始}$ = 1G字节/秒= 1字节/ (10^{-9}秒)
  - 那么传输1B，1KB、1MB、1GB的吞吐量$_{有效}$分别是多少？

41

41

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



42

42

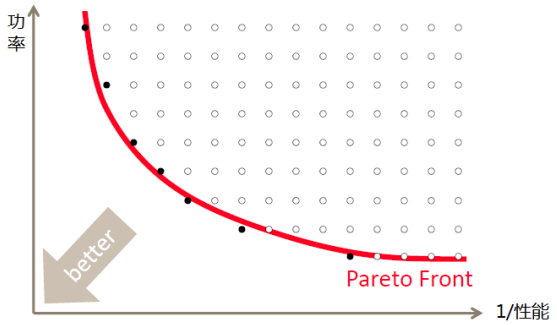
什么情况下不仅仅与时间有关？

- 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

43

43

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？

44

44

复合指标

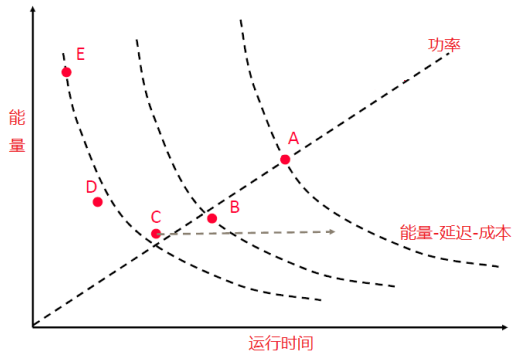
- 定义标量函数来反映需求——整合维度及其关系
- 例子，能量-延迟-成本
  - 越小越好
  - 不能最小化一个忽略另一个
  - 不需要有物理意义
- 地板和天花板
  - 现实生活中的设计往往是足够好，但不是最佳
  - 例如，满足功率(成本)上限下的性能基本要求

45

45

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



46

46

“伪”性能

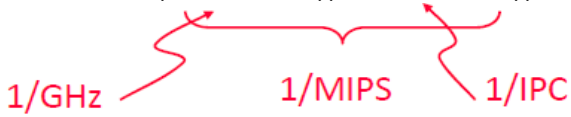
- 最有可能在宣传中看到的指标
  - IPC(每周期执行指令数)
  - MIPS(每秒百万条指令数)
  - GHz(每秒周期数)
- “听起来”像是性能，但不完整而且可能有误导
  - MIPS和IPC是平均值(取决于指令组合)
  - GHz、MIPS或IPC可以在牺牲彼此和实际性能的情况下得到改进

47

47

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

48

48

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，$2N^3 \approx 2\times10^9$ vs. $5N\log_2(N) \approx 5\times10^4$
- 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)

49

49

“伪”FLOPS

- 科学计算领域经常使用FLOPS作为性能度量

$$\frac{\text{浮点运算的“标称”数量}}{\text{程序运行时间}}$$

  - 例如，抽样点数为N的快速傅立叶变换名义上有$5N\log_2(N)$次浮点运算
- 这是一个好的、公平的衡量标准吗

  硬件+语言+编译器+算法组合？
  - 并非所有快速傅立叶变换算法都具有相同的浮点运算数
  - 并非所有浮点运算都是相等的(FADD vs. FMULT vs. FDIV)

计算同样问题时，FLOPS与1/时间成比例

50

50

相对性能

- 性能= 1/时间
  - 更小的延迟 → 更高的性能
  - 更高的吞吐（任务数/时间）→更高的性能
- 问题：如果X比Y慢50%，$time_X$= 1.0s，$time_Y$ = ？
  - 第一种情况：$time_Y$ = 0.5s，因为$time_Y / time_X$= 0.5
  - 第二种情况：$time_Y$ = 0.66666s，因为$time_X / time_Y$ = 1.5

51

51

相对性能

- “X比Y快n倍”表示

  n = 性能$_X$ / 性能$_Y$

  = 吞吐量$_X$ / 吞吐量$_Y$

  = 时间$_X$ / 时间$_Y$
- “X比Y快m%”表示

  1 + m/100 =性能$_X$ / 性能$_Y$
- 为了避免混淆，使用“比……快”这种描述
  - 对于前面第一种情况：Y比X快100%
  - 对于前面第二种情况：Y比X快50%

52

52

加速比

- 如果X是Y的加强版，这种加强的程度叫“加速比”（speedup）

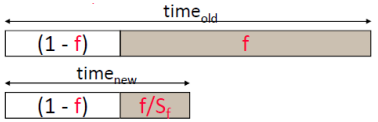
S = 时间$_{未加强之前}$ / 时间$_{加强之后}$
= 时间$_{Y}$ / 时间$_{X}$

53

53

加速比的Amdahl定律

- 假设通过优化将一个任务的f部分加速到原来的1/S_f



$time_{new} = time_{old} \cdot ((1-f) + f/S_f)$
$S_{overall} = 1 / ((1-f) + f/S_f)$

- 优化最“普遍”的情况
  - $S_{overall}$ 永远也不可能超过 1/(1-f)
  - f应该是对运行时间起支配作用的“普遍”情况
    (不要与“频繁”情况混淆)
  - f改善后，不“普遍”的情况会变得更加“普遍”

54

54

标准基准程序（Benchmark）

- 为什么要有标准基准程序？
  - 每个人都关心不同的应用(性能的不同方面)
  - 应用程序可能不适用于要研究的机器
- 例如：SPEC 基准程序 (www.spec.org)
  - Standard Performance Evaluation Corporation
  - 由一个多行业委员会选择的一组“实际可行的”、通用目的、公共领域的应用程序
  - 每隔几年更新一次，以反映使用和技术的变化
  - 反映客观性和预测能力

大家都知道并不完美，但大家都遵守同样的规则

55

55

SPEC CPU基准程序包

- CINT2006
  - perlbench (编程语言)，bzip2 (压缩)，gcc(编译)，mcf(优化)，gobmk(围棋)，hmmer(基因序列搜索)，sjeng(国际象棋)，libquantum(物理模拟)，h264ref(视频压缩)，omnetpp(C++,离散事件模拟)，astar(C++, 路径搜索)，xalancbmk(C++, XML)
- CFP2006
  - bwaves(CFD)，gamess(量子化学)，milc(C, QCD)，zeusmp(CFD)，gromacs (C+Fortran, 分子动力学)，cactusADM(C+Fortran, 相对论)，leslie3d(CFD)，namd (C++, 分子动力学)，dealII (C++, 有限元)，soplex (C++, 线性规划)，povray (C++, 光线轨迹)，calculix (C+Fortran, 有限元)，GemsFDTD (E&M)，tonto (量子化学)，lbm(C, CFD)，wrf(C+Fortran, 天气)，sphinx3(C, 语音识别)

56

56

性能小结

- 当比较两台计算机X和Y时，它们的相对性能很大程度上取决于要求X和Y做什么
  - 对于应用程序A，X可能比Y快m%
  - 对于应用程序B， X可能比Y快n%(m!=n)
  - 对于应用程序C，Y可能比X快k%
- 哪台计算机更快，速度提高了多少？
  - 取决于你关心的应用程序
  - 如果你关心不止一个应用程序，也取决于它们的相对重要性
- 很多方法可以将性能比较转化成单一的量化指标
  - 有些可能对你的目的有意义
  - 但是你必须知道什么时候做什么
- 没有一刀切的方法
  - 确保理解你想要衡量什么
  - 确保理解你测量了什么
  - 确保报告的内容准确且有代表性
  - 准备好公开原始数据
- 反正，没人相信你的数字……
  - 解释你试图衡量的效果
  - 解释你实际测量的内容和方式
  - 解释性能是如何总结和表示的

如果真的很重要，别人会想亲自检验一下

最重要的是要诚实！！！

57

57