高等计算机体系结构
第十五讲: 能耗/并行与多处理

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
2021-06-04

1

提醒：作业

- 作业6
  - 已发布，6月18日截止
  - 预取和并行

2

2

提醒：实验2-5

- 7月16日截止

3

3

阅读: 多处理

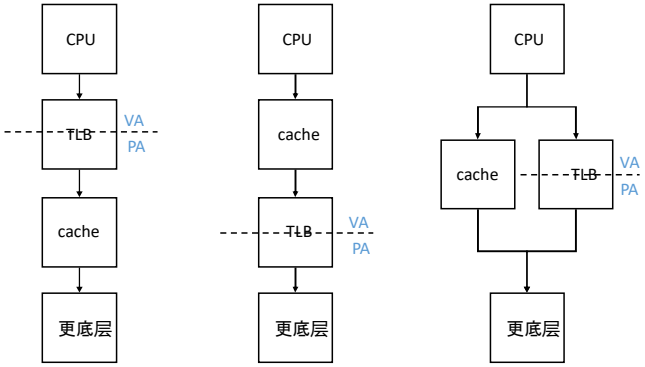
- 必读
  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.
  - Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979
  - Patterson & Hennessy's Computer Organization and Design: The Hardware/Software Interface （计算机组成与设计：软硬件接口）第5.8节 (第四版)
- 推荐
  - Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966
  - Hill, Jouppi, Sohi, "Multiprocessors and Multicomputers," pp. 551-560 in Readings in Computer Architecture.
  - Hill, Jouppi, Sohi, "Dataflow and Multithreading," pp. 309-314 in Readings in Computer Architecture.
  - Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.

4

4

回顾：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?



5

5

回顾：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

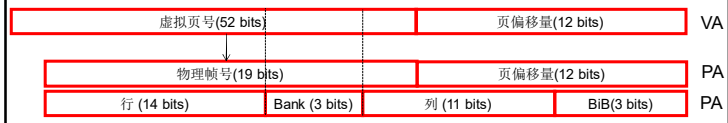
- 同音异义词：相同的声音不同的含义
  - 同一个虚拟地址可能映射到两个不同的物理地址
- 同义词：不同的声音相同的含义
  - 不同的虚拟地址可能映射到同一个物理地址

6

6

回顾：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

7

7

回顾：“性能”通常的定义

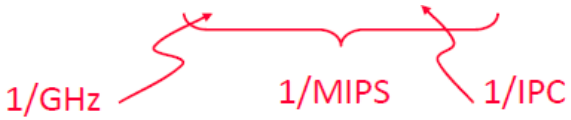
- 所有的一切几乎都和**时间**有关：性能∝ 1/时间
- 两种**截然不同**的性能！
  - 延迟=任务开始和完成之间的时间
  - 吞吐量=在给定时间单位内完成的任务数(速率度量)
- 不管怎样，时间越短，性能越高，但是.....
  - 吞吐量 ≠ 1/延迟，可以通过两者的权衡进行优化
  - 吞吐量 ≠ 吞吐量，当存在非经常启动开销时，吞吐量是N（任务量）的函数
  - 延迟 ≠ 延迟，延迟=实际操作时间+等待时间，“隐藏”延迟
- 除了性能，还有其它重要的指标：功率/能量、成本、风险、社会因素...
  - 如果不考虑它们之间的权衡，无法优化单个指标
  - 帕累托最优
  - 复合指标：定义标量函数来反映需求——整合维度及其关系

8

8

回顾：“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
- 墙钟时间=(时间/周期数)(周期数/指令数)(指令数/程序)



9

9

回顾：不仅与硬件相关

- 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
- 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)

10

10

“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

11

11

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

12

12

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%

13

13

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$_{加强之后}$

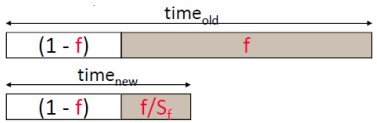
  = 时间$_Y$ / 时间$_X$

14

14

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”

15

15

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

16

16

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)

17

17

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！！！

18

18

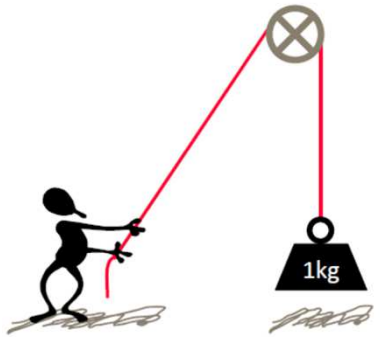
什么是能耗？

19

19

力：牛顿=千克·米/秒^2

- 9.8牛顿可以支持1千克质量的物体对抗重力
- 抓着绳子不消耗能量，不管重量有多重



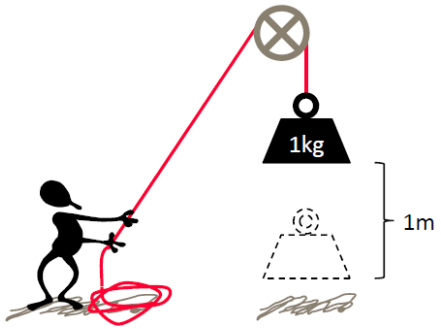
20

20

能量：焦耳=牛顿·米

- 9.8焦耳可以克服重力将1千克质量的物体提升1米

变化前后之间的静态概念



21

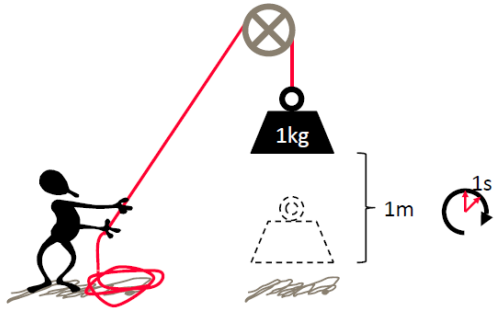
21

功率：瓦特=焦耳/秒

- 9.8瓦可以在1秒内克服重力将1千克质量物体提升1米

变化率的动态概念

- 9.8瓦可以是1千克/10米/10秒、10千克/1米/10秒等等



22

22

电子学中的能量与功率

- CMOS逻辑转换涉及到电容的充电和放电
- 当“电荷”从电源(VDD)流向地(GND)时，能量(焦耳)以阻抗产生的热耗散掉
  - 每次操作需要一定量的能量，例如，加法、寄存器读/写、对节点充放电
  - 能量∝计算量(功)

此外，只要保持通电就会有“漏”电流！！

- 功率(瓦特=焦耳/秒)是能量耗散率
  - 运算数/秒越高，焦耳/秒就越大
  - 功率∝性能

如果性能∝频率，那么一切将变得简单

功率=(½CV^2)·f

23

23

功和时间

- ***W***
  - 表示一个任务的“工作量”的标量（变量）
- ***T=W/C_{perf}***
  - 表示一个任务的执行时间
  - C_{perf} 是表示做功速率的标量（常量），即“单位时间做的功”

24

24

能量和功率

- $E_{switch}=c_{switch}W$
  - 与任务相关的“开关”能量
  - c_{switch}是表示做单位功产生的能量的标量（常量）
- $E_{static}=c_{static}T=c_{static}W/c_{perf}$
  - 保持给芯片供电产生的“泄漏”能量
  - c_{static}是所谓的“漏电功率”
- $E_{total}=E_{switch}+E_{static}=c_{switch}W+c_{static}W/c_{perf}=(c_{switch}+c_{static}/c_{perf})\cdot W$
  - 在高性能处理器中，静态功率可以接近50%
- $P_{total}=E_{total}/T=(c_{switch}W+c_{static}T)/T=c_{switch}c_{perf}+c_{static}$

25

25

简而言之，

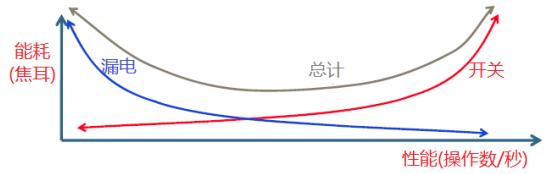
- 对于一个做功为W的任务
  - $T = W / c_{perf}$
    做功（工作量）越少的任务执行得越快
  - $E = E_{switch} + E_{static} = (c_{switch} + c_{static} / c_{perf})\cdot W$
    做功（工作量）越少的任务消耗的能量越少
  - $P = P_{switch} + P_{static} = c_{switch}c_{perf} + c_{static}$
    功率与任务不相关
- 现实是
  - W不是标量
  - c既不是标量也不是常数
  - ½CV²和½CV²f本身是非常“粗略”的近似值

26

26

关于静态功率的特别说明

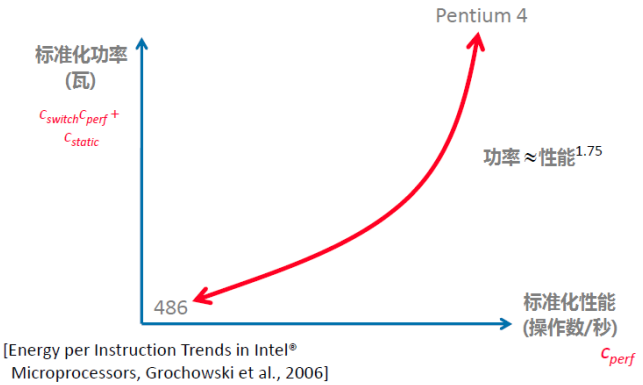
- $E_{total} = (c_{switch} + c_{static} / c_{perf})\cdot W = \underbrace{c_{switch}\cdot W}_{\text{开关}} + \underbrace{c_{static}\cdot W / c_{perf}}_{\text{静态}}$
- 执行得更慢(调慢时钟)会降低功率消耗，但会由于漏电的存在而增加能量消耗
- 执行得更快(必须改进设计)需要让c_{switch}和c_{static}有超线性的增长



27

27

注意：c_{switch},c_{static} ,c_{perf}互相依赖



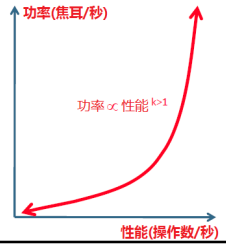
[Energy per Instruction Trends in Intel® Microprocessors, Grochowski et al., 2006]

28

28

暗示：功率和性能是密不可分的

- 如果不关心性能，很容易将功耗降至最低
- 可以预期功率超线性增加将提高性能
  - 较慢的设计更简单
  - 较低的频率需要的电压较低
  - “低挂果”优先
- 推论：较低的性能会产生较低的焦耳/操作
- 总之，越慢越节能



29

29

为什么能耗对今天的计算机体系结构如此重要？

30

30

技术缩放的入门知识

- 预计的缩放发生在离散的“工艺节点”中，每个节点线性缩放为大约先前的0.7x

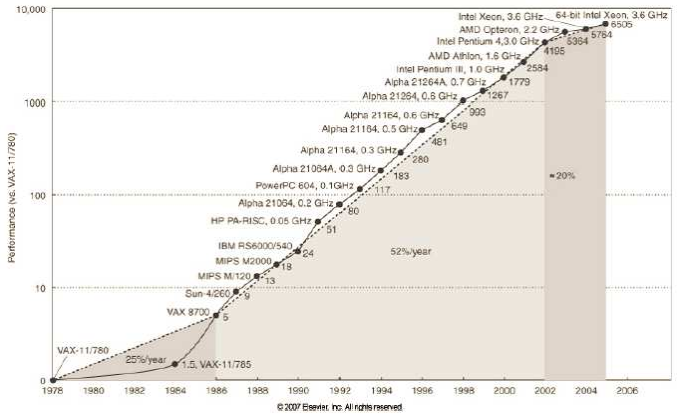
  90nm, 65nm, 45nm, 32nm, 22nm, 15nm, 7nm, . . .

- 如果设计不变，尺寸线性减小0.7x(也称为“门收缩”)理想情况下会导致
  - 芯片面积= 0.5x
  - 延迟= 0.7x，频率=1.43x
  - 电容= 0.7倍
  - Vdd=0.7x(恒定磁场)或Vdd=1x(恒定电压)
  - 功率= C×V²×f = 0.5x(恒定磁场)
  - 功率= 1x(恒定电压)
- 如果面积不变
  - 晶体管数量= 2x
  - 功率= 1x(恒定磁场)，功率= 2x(恒定电压)

31

31

摩尔定律的一种表现形式



32

32

摩尔定律→性能

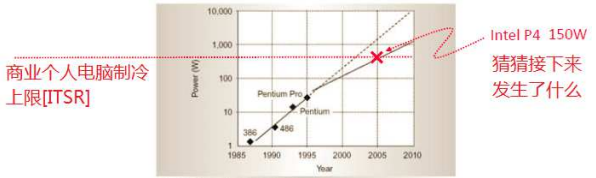
- 根据缩放理论的说法，我们应该得到
  @恒定复杂性：以1x晶体管数获得1.43x频率
  → 1.43x性能，0.5x功耗
  @最大复杂度：以2x晶体管数获得1.43x频率
  → 恒定功率下2.8x性能
- 实际上，我们得到了(高性能CPU)
  - 2x晶体管数
  - 2x频率(注意：比缩放理论快)
  - 总的来说，我们以约2x功率获得约2x性能

33

33

性能效率低下

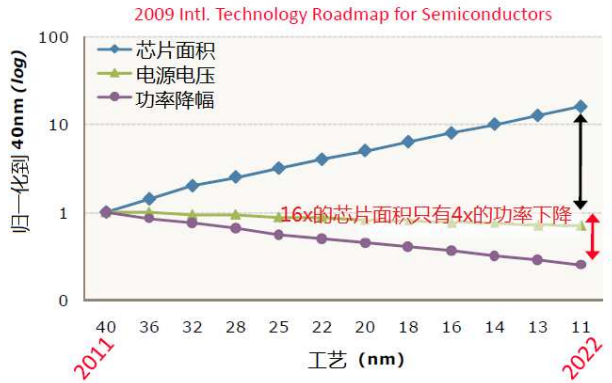
- 为了在单线程微处理器上达到“预期”的性能目标
  - 通过增加流水线深度越来越难提高频率
  - 使用2x晶体管构建更复杂的微体系结构(cache、分支预测、超标量、乱序执行)，以使更快/更深的流水线不会停顿
- 性能效率低下的后果是



34

34

登纳德缩放定律率先失效



必须以更少的焦耳/秒完成更多的操作数/秒

35

35

频率和电压缩放

- 每次转换的开关能量为½CV^2(寄生电容建模)
- 每秒f次转换的开关功率为½CV^2f
- 降低功率，就能降低时钟
- 如果时钟变慢了，可以通过降低电源电压来降低晶体管的速度
  - V→V‘，因此½CV^2 → ½CV'^2

漏电流/功率也由于更低的电压V’而超线性降低!!!

36

36

频率缩放

- 若$W/c_{perf} < T_{bound}$，可以通过一个因素的周期性缩放来降低性能

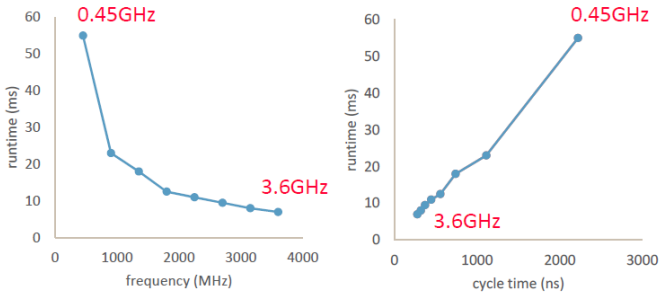
  $(W/c_{perf})/T_{bound} < s_{freq} < 1$

  当满足 $c_{perf}' = c_{perf} s_{freq}$
- $T' = W/(c_{perf} s_{freq})$
  - $1/s_{freq}$ 使执行时间变长
- $E' = (c_{switch} + c_{static}/(c_{perf} s_{freq})) \cdot W$
  - 更长的执行时间导致更高的(泄漏)能量
- $P' = c_{switch} c_{perf} s_{freq} + c_{static}$
  - 更长的执行时间导致更低的开关功耗

37

37

Intel P4 660 频率缩放: FFT_{64K}

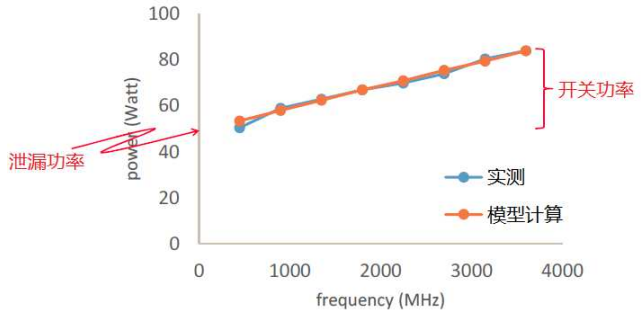


circa 2005, 90nm

38

38

Intel P4 660 频率缩放: FFT_{64K}



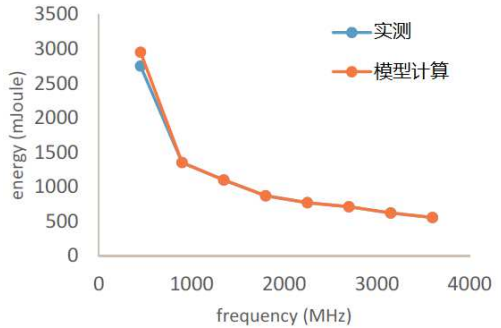
c_{perf} = 145操作数/秒; c_{switch}=0.24焦耳/操作; c_{static}=49.4瓦
(归一化到W=1)

circa 2005, 90nm

39

39

Intel P4 660 频率缩放: FFT_{64K}



c_{perf} = 145操作数/秒; c_{switch}=0.24焦耳/操作; c_{static}=49.4瓦
(归一化到W=1)

circa 2005, 90nm

40

40

频率和电压缩放

- 频率按s_{freq}缩放使得电压可以按相应的因子$s_{voltage}$缩放
- E∝V²，因此
  - $c_{switch}'' = c_{switch}\, s_{voltage}^2$
  - $c_{static}'' = c_{static}\, s_{voltage}\, e^{2.7(s\_voltage-1)} \approx ?$

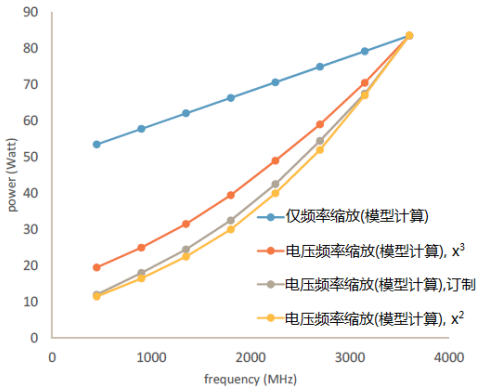
  简单起见，$c_{static} s_{voltage}^3$
- $T'' = W / (c_{perf}\, s_{freq})$
  - 执行时间增加1/ s_{freq}
- $E'' = (c_{switch} s_{voltage}^2 + c_{static} s_{voltage}^3 / c_{perf} s_{freq}) \cdot W$
  - 降低开关和静态能耗
- $P'' = c_{switch}\, s_{voltage}^2\, c_{perf}\, s_{freq} + c_{static} s_{voltage}^3$
  - 开关功率按立方减小，静态功率按平方减小

架构师们通常进行粗略的简化

41

41

Intel P4 660 电压频率缩放: FFT_{64K}

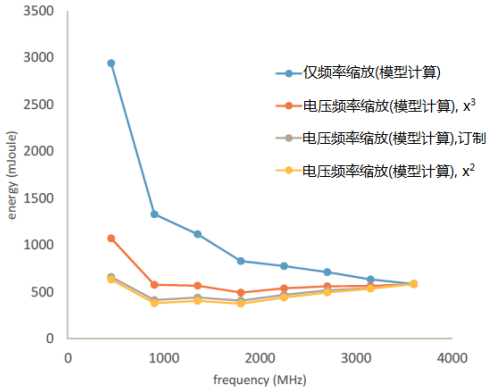


circa 2005, 90nm

42

42

Intel P4 660 电压频率缩放: FFT_{64K}

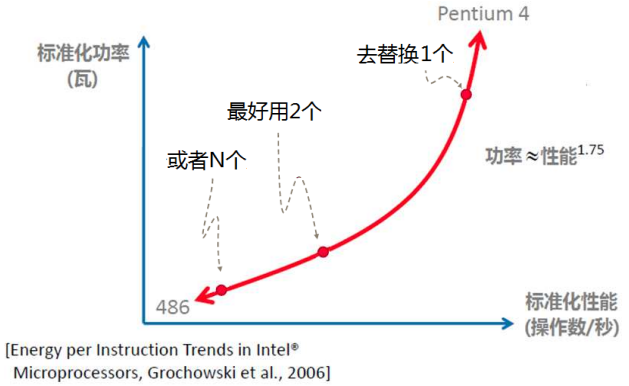


circa 2005, 90nm

43

43

并行化



[Energy per Instruction Trends in Intel® Microprocessors, Grochowski et al., 2006]

44

44

并行化

- 使用N个处理器时理想的并行化
  - $T = W / (c_{perf} N)$
  - $E = (c_{switch} + c_{static} / c_{perf}) \cdot W$

    注意：4x的静态功率，4x的执行时间提升
  - $P = N (c_{switch} c_{perf} + c_{static})$

    “我们曾经这样想”
- 或者，假设$s_{voltage} \approx s_{freq}$，我们可以基于$s_{freq} = 1/N$用加速比$N$来换取功率和能量的下降
  - $T = W / c_{perf}$
  - $E'' = (c_{switch} / N^2 + c_{static} / (c_{perf} N)) \cdot W$
  - $P'' = c_{switch} c_{perf} / N^2 + c_{static} / N$

    “我们现在这样想”

45

45

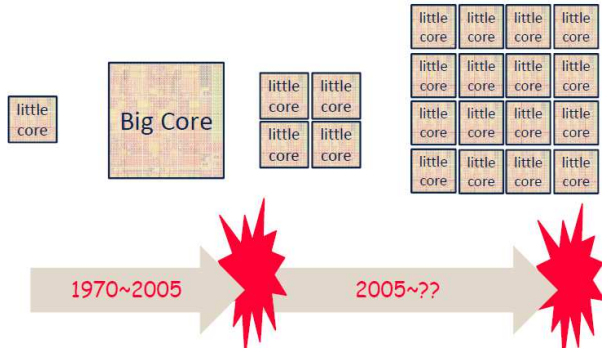
所以问题在哪儿？

- 我们知道如何在芯片上封装更多的核，从而在“聚合”或“吞吐”性能方面保持摩尔定律
- 如何使用它们？
  - 如果N个任务单元是N个独立的程序，生活是美好的⇒只需要运行它们就好了
  - 如果N个任务单元是同一程序的N个操作，该怎么办？⇒重写一个并行程序……就好了……
  - 如果N个任务单元是同一程序的N个串行相关的操作，该怎么办？⇒？？
- 能有效地使用多少核？

46

46

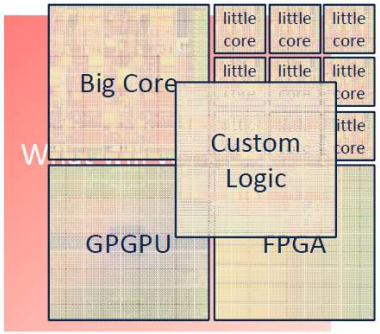
摩尔定律通过核数的增加继续扩展



47

47

记住：性能/瓦特和操作数/焦耳



48

48

并行

并行结构

- 为什么采用并行结构
  - 绝对性能
  - 功耗和能耗
  - 复杂度
  - 性能价格比
- 关键的使能因素
  - 半导体和互连网络技术的发展
  - 软件技术的发展

并行机

- 定义：并行机就是一系列能够通过协同和通信来快速解决大规模问题的处理单元的集合 --- Almasi and Gottlieb
- “…处理单元的集合…”
  - 多少个? 每个处理单元的能力有多强? 可扩展性如何?
- “…能够通信…”
  - 如何通信? (shared-memory 还是 message passing)
  - 互连网络的结构如何? (bus, crossbar, multistage…)
  - 评价指标: cost, latency, throughput, scalability, fault tolerance

并行机（续）

- “… 能够协同…”
  - 重要的问题有：synchronization, granularity, and autonomy
  - 同步允许按顺序操作保证正确性
    - 各种原语 (test&set, fetch&add..)
  - 粒度要与程序中的子任务大小相对应
    - 减小粒度→ Parallelism, Communication, Overhead increase
    - 粒度 vs 指令数
      - Program level: 10^6 instr
      - Task level: 10^3 - 10^6 instr
      - Loop level: 10 - 1000 instr
      - Stmt/instr level: 1 - 10 instr
  - 自治性: tradeoff

并行机（续）

- “… 快速解决大规模问题…”
  - 通用 vs 专用?
    - 任何机器都可以很好的解决某一类问题
  - 什么样的应用适合并行处理?
    - 高并行应用
      - 很多科学计算利用数据的并行性 (MonteCarlo)
    - 中并行应用
      - 很多工程性的应用(有限元, VLSI-CAD)
    - 不太并行的应用
  - 需要考虑应用和机器的规模

53

53

并行机的分类

- Michael Flynn的分类
- 基于模型的分类

54

54

Flynn分类法

- Mike Flynn, “Very High-Speed Computing Systems,” Proc. of IEEE, 1966
- 基于指令流、数据流的概念
- 根据硬件对指令流和数据流的支持不同来区分并行的组织
  - SISD: Single Instruction and Single Data Stream (uniprocessor)单指令操作单个数据元素
  - SIMD: Single I and Multiple D Streams (GPUs)单指令操作多个数据元素
    - 阵列处理机
    - 向量处理器
  - MISD: Multiple I and Single D Streams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: Multiple I and Multiple D Streams (multicores)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

55

55

基于模型的分类

- 共享内存（Shared-memory）
- 消息传递（Message-passing）
- 数据流（Dataflow）
- 脉动阵列（systolic）
- 数据并行（Data parallel）
- ……

56

56

多处理器和多处理中的难题

57

为什么要有并行机?

- 并行性: 同时做多件"事情"
- "事情": 指令, 操作, 任务
- 主要目标
  - 改善性能 (执行时间或任务吞吐量)
    - 程序的执行时间由Amdahl定律控制
- 其他目标
  - 降低功耗
  - 提高成本效率和可扩展性, 降低复杂性
  - 提高可靠性: 在空间上冗余执行

58

58

并行的类型

- 指令级并行
  - 一个指令流中的不同指令可以并行执行
  - 流水线, 乱序执行, 投机执行, VLIW
  - 数据流
- 数据并行
  - 数据的不同片段可以被并行的操作
  - SIMD: 向量处理, 阵列处理
  - 脉动阵列, 流处理器
- 任务级并行
  - 不同的"任务/线程"可以被并行执行
  - 多线程
  - 多处理 (多核)

59

59

任务级并行: 生成任务

- 将一个问题分割成多个相关的任务(线程)
  - 显式: 并行编程
    - 当问题中的任务能够很自然地划分时
      - Web/数据库请求
    - 当任务的边界不那么清晰时会比较困难
  - 透明/隐式: 线程级投机
    - 投机地分割单个线程
- 同时运行多个独立的任务(进程)
  - 当有多个进程时
    - 批处理的仿真, 不同用户的进程, 云计算的工作负载
  - 不能提升单个任务的性能

60

60

多处理基础

61

61

多处理器的类型

- 松耦合多处理器
  - 没有共享的全局存储地址空间
  - 多机网络
    - 基于网络的多处理器
  - 通常通过消息传递编程
    - 通过显式调用 (send, receive) 通信
- 紧耦合多处理器
  - 共享全局存储地址空间
  - 传统的多处理: 对称多处理器(SMP)
    - 现在的多核处理器, 多线程处理器
  - 编程模型与单处理器类似(多任务单处理器)，除了
    - 操作共享数据需要同步

62

62

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

63

63

基于硬件的多线程

- 粗粒度多线程
  - 基于定量
  - 基于事件
- 细粒度多线程
  - 每周期
  - Thornton, “CDC 6600: Design of a Computer,” 1970.
  - Burton Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978.
- 同时多线程
  - 能够同时由多个线程分发指令
  - 有效提升执行单元的利用率

64

64

并行加速比

- $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$
- 假设每个操作占用1个周期, 没有通信开销, 每个操作可以在不同的处理器上执行
- 用单个处理器执行有多快?
  - 假设没有流水线或者对指令的并发执行
- 用3个处理器有多快?
  - 加速比(speedup)

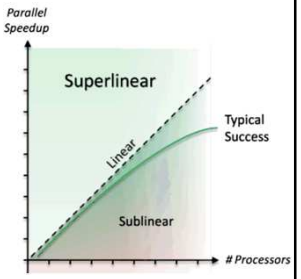
Horner, "A new method of solving numerical equations of all orders, by continuous approximation," Philosophical Transactions of the Royal Society, 1819.

65

65

超线性加速比

- 当使用P个处理单元时能否获得大于P的加速比?
  - Cache 的影响
  - 工作集的影响
- 两种情况下:
  - 对比不公平
  - 访存的影响



66

66

利用率, 冗余度和效率

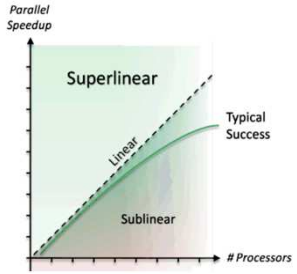
- 常用的指标
  - 假设所有P个处理器都满负荷参与并行计算
- 利用率: 有多少处理能力被使用
  - U = (并行操作的数量) / (处理器 x 时间)
- 冗余度: 并行处理时做了多少额外的工作
  - R = (并行操作的数量) / (用最佳的单处理器算法执行的操作数)
  - R总是≥1
- 效率: 用了多少资源占能够获得多少资源的比例
  - E = (使用1个处理器花费的时间) / (处理器 x 使用P个处理器花费的时间)
  - E = U/R

67

67

真实的加速比

- 为什么真实的加速比是这样的?



$$\tau_p = \alpha \cdot \frac{\tau_1}{p} + (1-\alpha) \cdot \tau_1$$

单处理器程序中可以并行化的部分　　不可以并行化的部分

68

68

Amdahl定律

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

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

并行加速比的瓶颈

Amdahl定律的内涵：
1、当α<1时，增加越来越多的处理器，得到的收益(加速比)越来越少；
2、收益(加速比)不大，除非α≈1

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

69

69

Amdahl定律的启示

- Amdahl定律的另一种表示
  - f: 程序可并行化的比例
  - N: 处理器数量

$$\text{加速比} = \frac{1}{1-f + \frac{f}{N}}$$

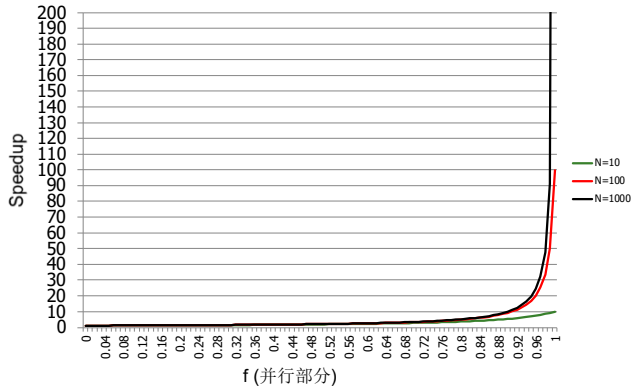
  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

- 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)

70

70

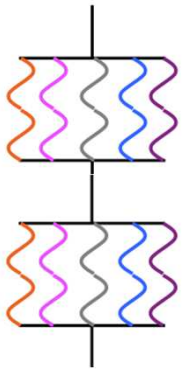
串行瓶颈



71

71

为什么串行是瓶颈?



- 并行的机器有串行瓶颈
- 主要原因: 有不能并行化的数据操作(比如, 不能并行化的循环)

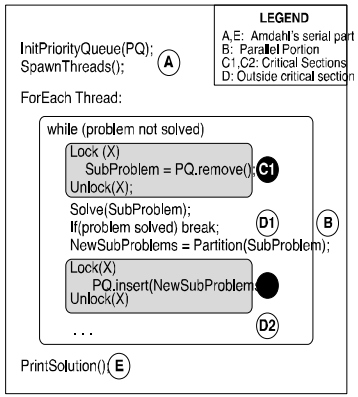
```
for ( i = 0 ; i < N; i++)
    A[i] = (A[i] + A[i-1]) / 2
```

- 数据准备是单线程的，而任务生成是并行的(通常任务本身又是串行的)

72

72

串行瓶颈的例子



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

并行编程的困难

- 如果存在天然的并行性不就太难
  - “高度并行”的应用
  - 多媒体, 物理模拟, 图形图像处理
  - 大型 web 服务器, 数据库?
- 困难在于
  - 让并行程序正确运行
  - 存在瓶颈时优化性能
- **并行计算机体系结构**主要是关于
  - 如何设计计算机以克服串行和并行瓶颈，获得高性能和高效率
  - 使程序员更容易开发正确并且高性能的并行程序

多处理中访存的序

操作的序

- 操作: A, B, C, D
  - 硬件该按照什么顺序执行(报告结果)这些操作?
- 程序员和微架构之间的约定
  - 由ISA确定
- 维护一个“期望的” 序能够使程序员的人生更加美好
  - 调试; 状态恢复和处理异常
- 维护一个“期望的”序通常使硬件设计者的人生更加……
  - 尤其是需要设计高性能处理器: 乱序执行中的load-store队列

77

77

单处理器中访存的序

- 由冯诺依曼模型指定
- 顺序序
  - 硬件执行load和store操作，按照程序串行执行所指定的序
- 乱序执行不改变语义
  - 硬件提交 (向软件报告结果) load和store操作，按照程序串行执行所指定的序
- 优点: 1) 执行过程中的体系结构状态是精确的 2) 程序每次运行的体系结构状态是一致的 → 容易调试程序
- 缺点: 保持序会增加开销, 降低性能

78

78