

高等计算机体系结构

第十四讲: 互连和性能能耗

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
2020-06-19

1

提醒：作业

- 作业6
 - 已截止
 - 预取和并行

2

2

提醒：实验2-5

- 7月10日截止

3

3

期末考试

- 6月26日上午9:50-12:20
 - 试题提前10分钟通过课程中心作业区发布
 - 自备白色答题纸答题，写清题号不用抄题
 - 答题完毕后拍照上传课程中心作业区，最迟提交时间为12:30
 - 有任何提交问题请及时沟通联系
 - 同时开通qq群课堂，所有同学进入课堂，不用开视频，交卷后方可退出群课堂

4

4

互连网络基础

5

回顾：Amdahl定律

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

$$\text{加速比}_{p \rightarrow \infty} = \frac{1}{1-\alpha}$$

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）

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

6

回顾：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

7

回顾：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

8

回顾：顺序一致性

- Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

- 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

9

9

Tradeoff: 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

10

10

回顾：Cache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

11

11

回顾：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)

12

12

回顾：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)** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
 - 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)** [Censier & Feautrier, IEEE ToC 1978]
 - 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

13

13

回顾：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

14

14

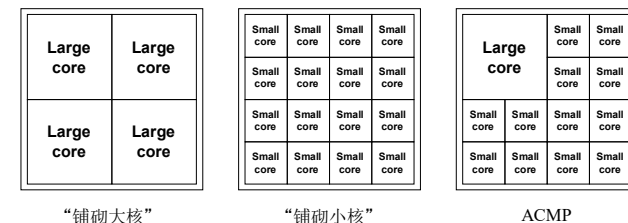
回顾：串行化的代码段

- 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

15

15

回顾：非对称片上多处理器(ACMP)



- 提供一个大核和多个小核
- + 利用大核加速串行部分
- + 在小核和大核上执行并行部分以获得高的吞吐率

16

16

回顾：利用不对称

- 串行部分的执行时间必须短
- 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
 - 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
- 对程序员来说缩短这些串行段相当困难
- 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

17

17

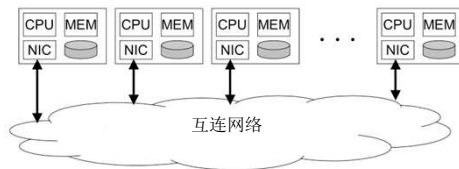
互连网络

18

18

哪里需要互连网络?

- 有组件需要互连
- 很多例子
 - 处理器和处理器
 - 处理器和内存(bank)
 - 处理器和cache (bank)
 - Cache和cache
 - I/O 设备



19

19

为什么这个很重要?

- 影响系统的可扩展性
 - 可以构建一个多大的系统?
 - 增加更多的处理器有多容易?
- 影响性能和能效
 - 处理器、cache和内存之间通信有多快?
 - 访存延迟有多大?
 - 通信消耗多少能量?

20

20

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

21

21

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
- ...

22

22

评价互连网络的指标

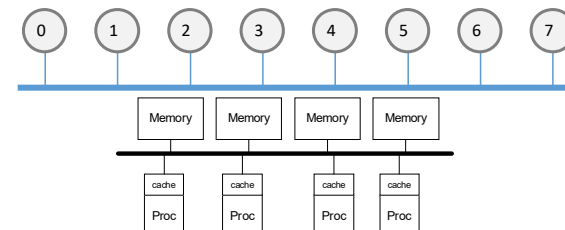
- 成本
- 延迟 (按跳, 单位纳秒)
- 竞争度
- 其它需要考虑的指标
 - 能耗
 - 带宽
 - 系统整体性能

23

23

总线

- + 简单
- + 节点数量少时成本效率高
- + 很容易实现一致性 (监听和顺序)
- 节点数量大时没有可扩展性 (受限的带宽, 电负载 → 频率降低)
- 高竞争度 → 快饱和

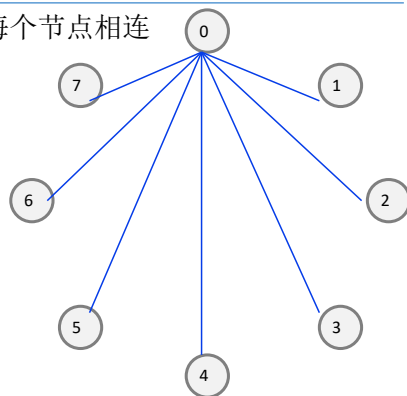


24

24

点到点

每个节点都与其它的每个节点相连

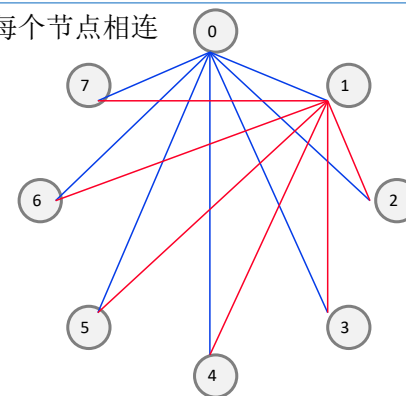


25

25

点到点

每个节点都与其它的每个节点相连

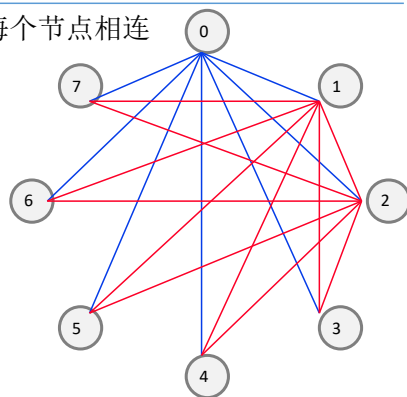


26

26

点到点

每个节点都与其它的每个节点相连

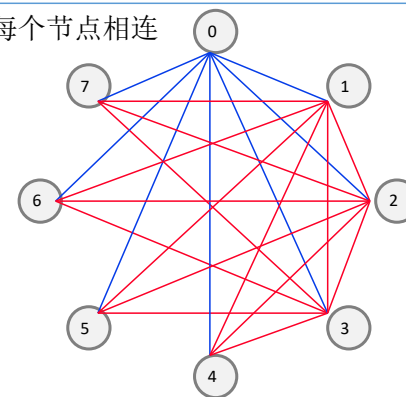


27

27

点到点

每个节点都与其它的每个节点相连

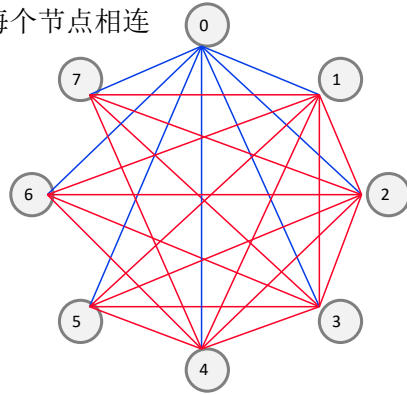


28

28

点到点

每个节点都与其它的所有节点相连

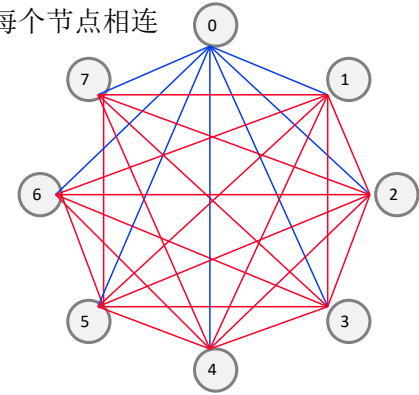


29

29

点到点

每个节点都与其它的所有节点相连

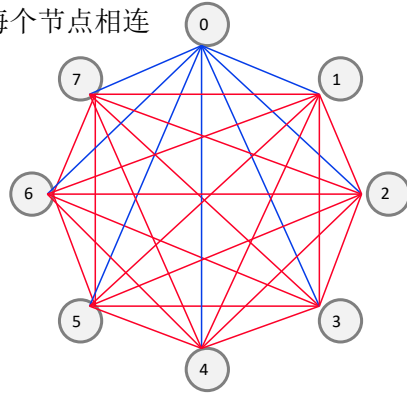


30

30

点到点

每个节点都与其它的所有节点相连



31

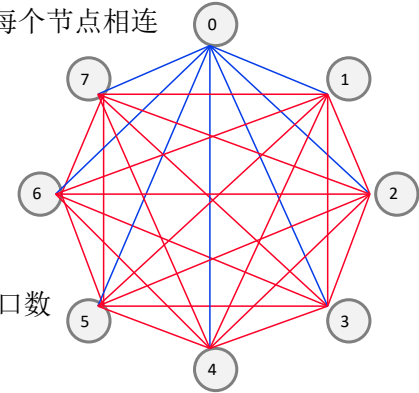
31

点到点

每个节点都与其它的所有节点相连

- + 竞争度最低
- + 潜在的最低延迟
- + 理想(如果不差钱)

- 成本最高
- $O(N)$ 连接/每节点端口数
- $O(N^2)$ 链路
- 没有扩展性
- 在芯片上如何布局?



32

32

交叉开关(Crossbar)

- 每个节点可连接到任何其它节点 (无阻塞)，不同的是任一节点可随时使用连接
- 允许同时向无冲突的目的节点发送
- 适用于节点数目较小的情况

+ 低延迟、高吞吐

- 昂贵

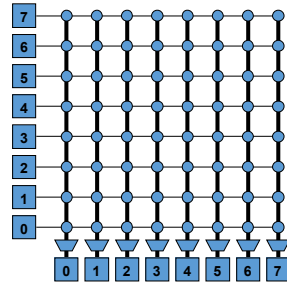
- 无可扩展性 $\rightarrow O(N^2)$ 成本

- 随着N的增加仲裁越来越困难

在核到cache到bank中使用

- IBM POWER5

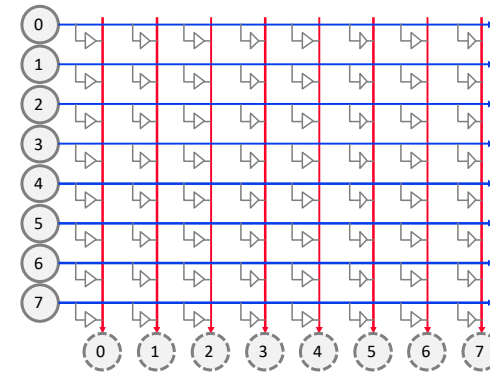
- Sun Niagara I/II



33

33

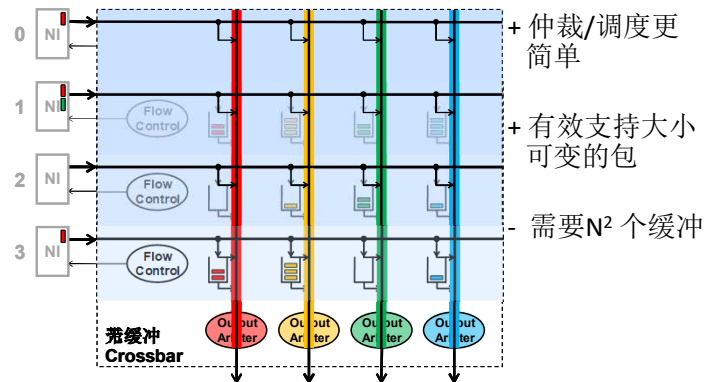
另一种 Crossbar 设计



34

34

带缓冲的 Crossbar



35

35

能比Crossbar成本更低吗?

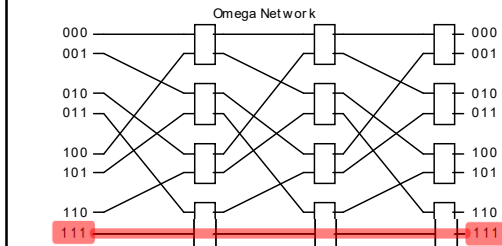
- 仍能拥有低竞争度?
- 思路: 多阶段网络

36

36

多级对数网络

- 思路: 在终端/节点之间通过多层交换实现间接组网
- 成本: $O(N \log N)$, 延迟: $O(\log N)$
- 很多变种(Omega, 蝴蝶, Benes, Banyan, ...)
- Omega 网络:



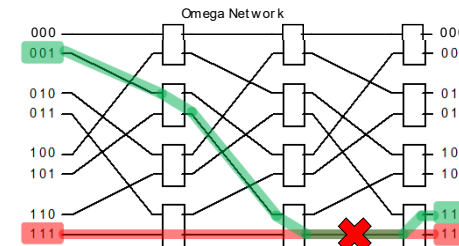
Gottlieb et al. "The NYU Ultracomputer-designing a MIMD, shared-memory parallel machine," ISCA 1982.

37

37

多级对数网络

- 思路: 在终端/节点之间通过多层交换实现间接组网
- 成本: $O(N \log N)$, 延迟: $O(\log N)$
- 很多变种(Omega, 蝴蝶, Benes, Banyan, ...)
- Omega 网络:



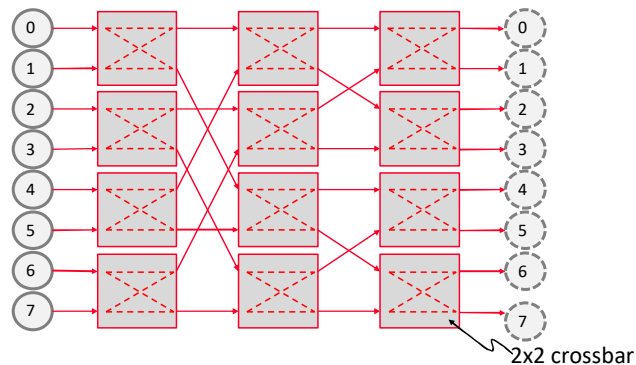
冲突

Gottlieb et al. "The NYU Ultracomputer-designing a MIMD, shared-memory parallel machine," ISCA 1982.

38

38

多级电路交换

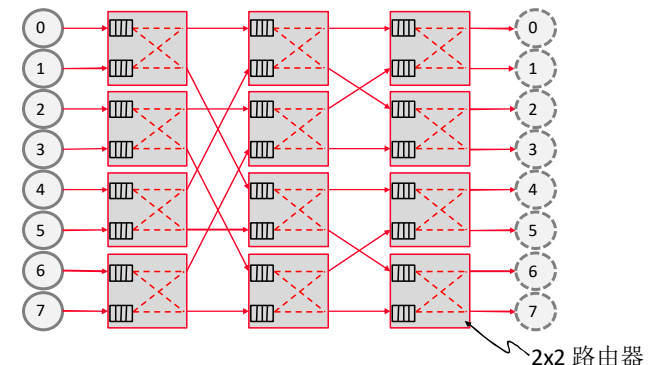


- 对并发更严格
- 但是相对于crossbar的成本来说具有更好的可扩展性

39

39

多级包交换



- 包在路由器之间逐“跳”传递, 等待下一跳交换机和缓冲的可用性

40

40

交换 vs. 拓扑

- 电路/包交换的选择不依赖于拓扑
- 消息如何传送至目的地依靠高层协议
- 当然, 某些拓扑确实可能更适用于电路或者包交换

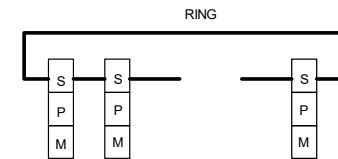
41

41

环

- + 便宜: $O(N)$ 成本
- 高延迟: $O(N)$
- 不易扩展
- 对分带宽是常数

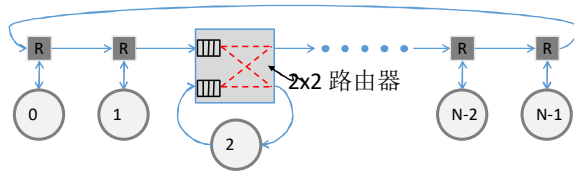
在Intel Haswell, Intel Larrabee, IBM Cell等很多现代商业化系统中使用



42

42

单向环



- 拓扑及实现简单
 - 如果 N 和带宽及延迟要求都相对较低, 性能比较合理
 - $O(N)$ 成本
 - $N/2$ 平均跳数; 延迟依赖于利用率

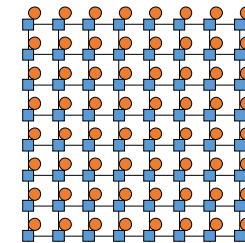
43

43

网状网(Mesh)

- $O(N)$ 成本
- 平均延迟: $O(\sqrt{N})$
- 容易在芯片上布局: 规则并且等长的连接
- 路径多样性: 从一点到另一点有很多条路

- Tiler 100核芯片中使用
- 是许多片上网络的原型

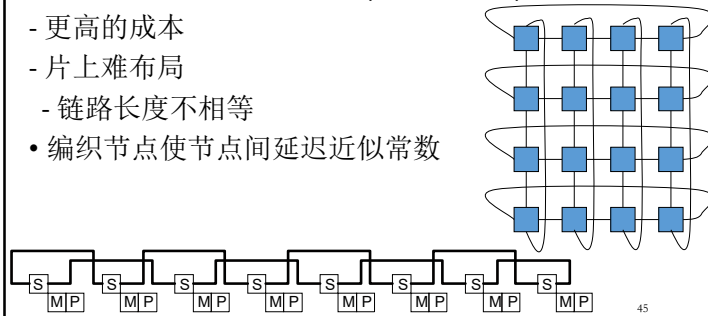


44

44

Torus

- Mesh 在边缘部分不对称: 在边缘放置任务时性能会非常敏感
- Torus 避免了这个问题
- + 比mesh更高的路径多样性(和对分带宽)
- 更高的成本
- 片上难布局
- 链路长度不相等
- 编织节点使节点间延迟近似常数



45

树

平面、分层拓扑结构

延迟: $O(\log N)$

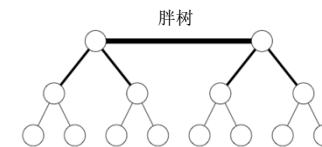
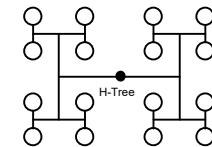
对局部流量很有效

+ 便宜: $O(N)$ 成本

+ 容易布局

- 根会成为瓶颈

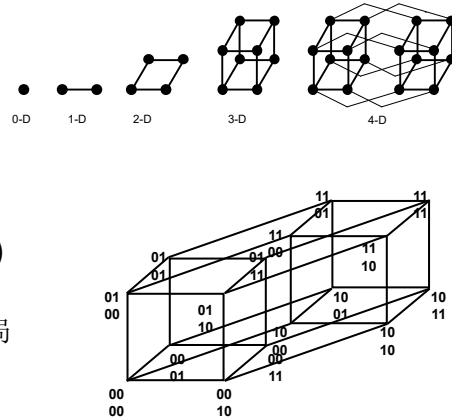
胖树可以避免这一问题(CM-5)



46

超立方

- 延迟: $O(\log N)$
- 基数: $O(\log N)$
- 连接数: $O(N \log N)$
- + 低延迟
- 在2D/3D中难布局

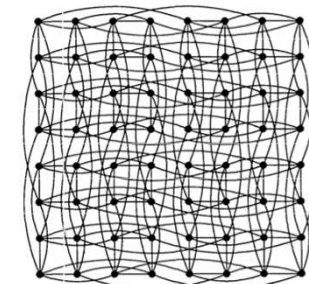
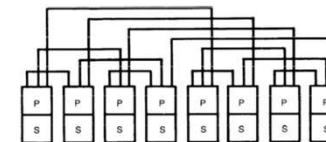


47

加州理工的宇宙立方

- 64节点的消息传递机器

- Seitz, "The Cosmic Cube," CACM 1985.

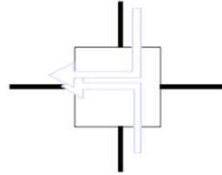


A hypercube connects $N = 2^n$ small computers, called nodes, through point-to-point communication channels in the Cosmic Cube. Shown here is a two-dimensional projection of a six-dimensional hypercube, or binary 6-cube, which corresponds to a 64-node machine.

FIGURE 1. A Hypercube (also known as a binary cube or a Boolean n -cube)

48

处理竞争



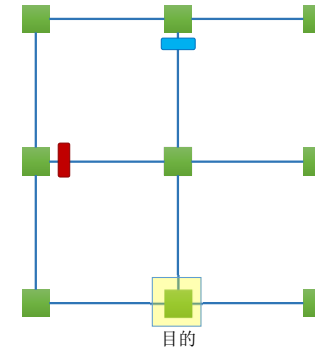
- 两个包试图同时使用同一个连接
- 如何处理?
 - 缓冲一个
 - 丢弃一个
 - 偏转 (错误路由) 一个
- Tradeoff?

49

49

无缓冲偏转路由

- 核心思路: 包在网络中绝不缓冲, 当两个包争用同一连接时, **偏转** 其中一个

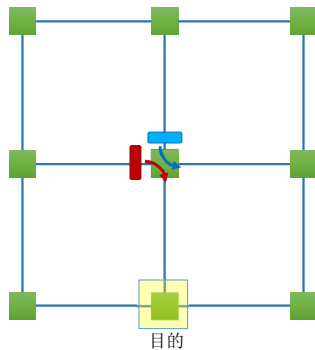


Baran, "On Distributed Communication Networks." RAND Tech. Report., 1962 / IEEE Trans.Comm., 1964.
50

50

无缓冲偏转路由

- 核心思路: 包在网络中绝不缓冲, 当两个包争用同一连接时, **偏转** 其中一个

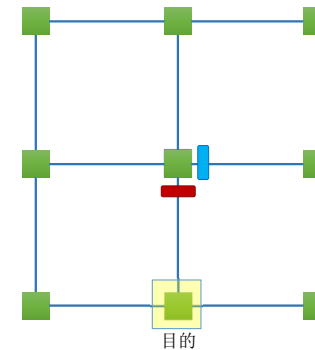


Baran, "On Distributed Communication Networks." RAND Tech. Report., 1962 / IEEE Trans.Comm., 1964.
51

51

无缓冲偏转路由

- 核心思路: 包在网络中绝不缓冲, 当两个包争用同一连接时, **偏转** 其中一个



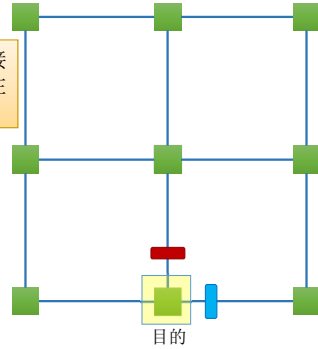
Baran, "On Distributed Communication Networks." RAND Tech. Report., 1962 / IEEE Trans.Comm., 1964.
52

52

无缓冲偏转路由

- 核心思路: 包在网络中绝不缓冲, 当两个包争用同一连接时, **偏转**其中一个

每当有输出连接可用, 就可以注入新的流量



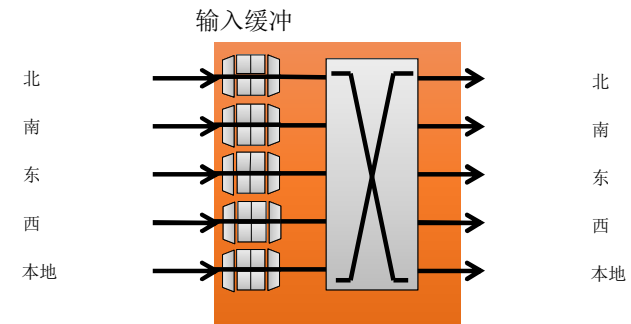
Baran, "On Distributed Communication Networks." RAND Tech. Report., 1962 / IEEE Trans.Comm., 1964.

53

53

无缓冲偏转路由

- 可以免除输入缓冲: 传输的包会通过**流水线锁存器**和**网络连接**“缓冲”

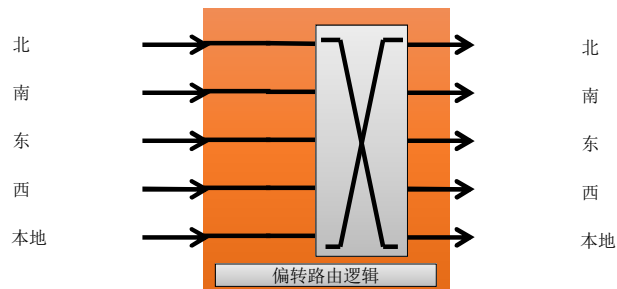


54

54

无缓冲偏转路由

- 可以免除输入缓冲: 传输的包会通过**流水线锁存器**和**网络连接**“缓冲”



55

55

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

56

56

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)

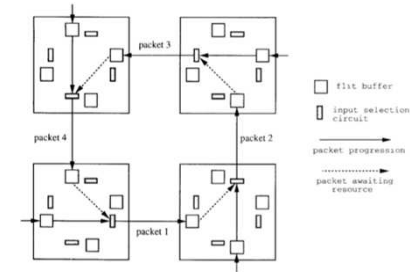
- 可能导致高度竞争
- 没有利用路径多样性

57

57

死锁

- 没有进程可以向前推进
- 由对资源的循环依赖引发
- 每个包等待被下游包占据的缓冲区



58

58

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

59

59

避免死锁的转向模型

- 思路
 - 分析一个网络中包可能的转移方向
 - 确定这些转向可以构成的环
 - 禁止某些转向以打破可能的环
- Glass and Ni, “The Turn Model for Adaptive Routing,” ISCA 1992.

FIG. 2. The possible turns and simple cycles in a two-dimensional mesh.

FIG. 3. The four turns allowed by the xy routing algorithm.

FIG. 4. Six turns that complete the cycles and allow deadlock.

60

60

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

61

61

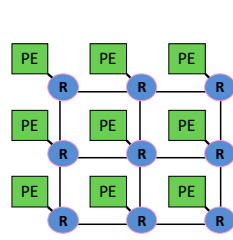
适应性路由

- 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)来选择高效输出口发送包
 - 高效输出口: 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

62

62

片上网络

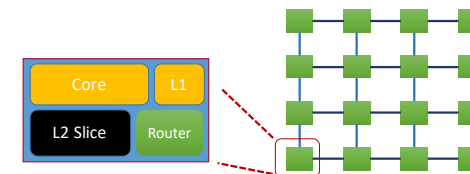
- 
- 连接核、cache、内存控制器等
 - 总线和交叉开关不具有可扩展性
 - 包交换
 - 2D mesh: 最常用的拓扑
 - 主要用来应对cache缺失和访存请求
- R 路由器
- PE 处理单元
(核, L2 Bank, 内存控制器, 等等)

63

63

高效互连的动机

- 在众核芯片中, 片上互连(NoC)消耗了巨大的能量
- | 芯片 | 功耗 |
|-----------------|-----------|
| Intel Terascale | ~28% 芯片功耗 |
| Intel SCC | ~10% |
| MIT RAW | ~36% |



- 最近的一些工作利用无缓冲偏转路由来减小功耗和芯片尺寸

64

64

性能分析

65

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

66

例子：单周期性能分析

指令执行时间计算

- 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**
- 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

$$600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\% = 447.5ps$$

- 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

67

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）。

68

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

69

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

70

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

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

71

流水线性能分析

- 获得什么收益？
- 付出什么代价？

72

72

性能

——所有的一切几乎都和**时间**有关

73

73

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

74

74

“性能”通常的定义

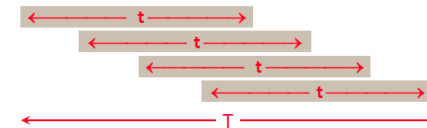
- 首先，性能 $\propto 1/\text{时间}$
- 两种**截然不同**的性能！
 - 延迟=任务开始和完成之间的时间
 - 吞吐量=在给定时间单位内完成的任务数(速率度量)
 - 不要混淆
- 不管怎样，时间越短，性能越高，但是.....

75

75

吞吐量 $\neq 1/\text{延迟}$!

- 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 $\neq 1/\text{延迟}$



- 可以通过两者的权衡进行优化

76

76

吞吐量 ≠ 吞吐率

- 当存在非经常启动开销时，吞吐率是 N （任务量）的函数
- 若启动时间= t_s ，吞吐率_{原始}= $1/t_1$
 - 吞吐率_{有效}= $N / (t_s + N \cdot t_1)$
 - 若 $t_s \gg N \cdot t_1$ ，吞吐率_{有效} $\approx N/t_s$
 - 若 $t_s \ll N \cdot t_1$ ，吞吐率_{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
 - 总线吞吐率_{原始} = 1G字节/秒 = 1字节/ $(10^{-9}$ 秒)
 - 那么传输1B, 1KB、1MB、1GB的吞吐率_{有效}分别是多少？

77

77

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



78

78

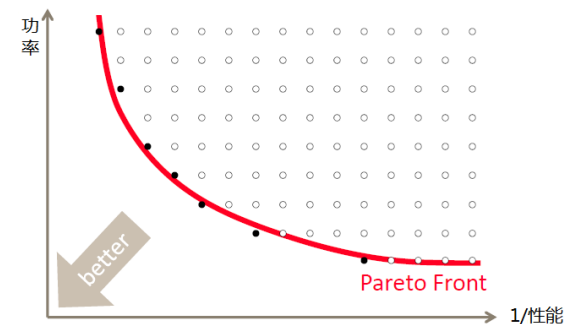
什么情况下不仅仅与时间有关？

- 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

79

79

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？

80

80

复合指标

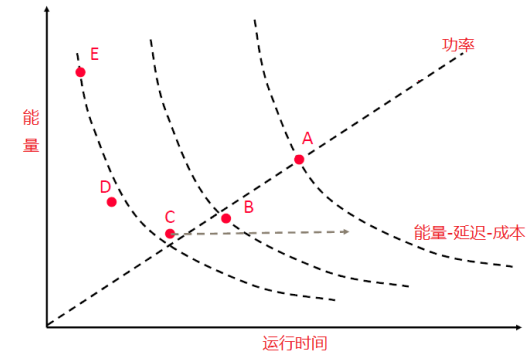
- 定义标量函数来反映需求——整合维度及其关系
- 例子，能量-延迟-成本
 - 越小越好
 - 不能最小化一个忽略另一个
 - 不需要有物理意义
- 地板和天花板
 - 现实生活中的设计往往是足够好，但不是最佳
 - 例如，满足功率(成本)上限下的性能基本要求

81

81

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



82

82

“伪”性能

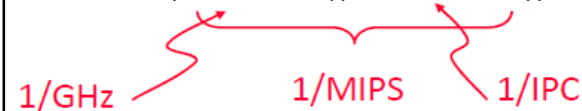
- 最有可能在宣传中看到的指标
 - IPC(每周期执行指令数)
 - MIPS(每秒百万条指令数)
 - GHz(每秒周期数)
- “听起来”像是性能，但不完整而且可能有误导
 - MIPS和IPC是平均值(取决于指令组合)
 - GHz、MIPS或IPC可以在牺牲彼此和实际性能的情况下得到改进

83

83

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

84

84

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 $N=1024$, $2N^3 \approx 2 \times 10^9$ vs. $5N\log_2(N) \approx 5 \times 10^4$
- 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)

85

85

“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

86

86

相对性能

- 性能= 1/时间
 - 更小的延迟 \rightarrow 更高的性能
 - 更高的吞吐 (任务数/时间) \rightarrow 更高的性能
- 问题: 如果 X 比 Y 慢50%, $\text{time}_X = 1.0s$, $\text{time}_Y = ?$
 - 第一种情况: $\text{time}_Y = 0.5s$, 因为 $\text{time}_Y / \text{time}_X = 0.5$
 - 第二种情况: $\text{time}_Y = 0.66666s$, 因为 $\text{time}_X / \text{time}_Y = 1.5$

87

87

相对性能

- “ X 比 Y 快 n 倍”表示
 - $n = \text{性能}_X / \text{性能}_Y$
 - $= \text{吞吐量}_X / \text{吞吐量}_Y$
 - $= \text{时间}_X / \text{时间}_Y$
- “ X 比 Y 快 $m\%$ ”表示
 - $1 + m/100 = \text{性能}_X / \text{性能}_Y$
- 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%

88

88

加速比

- 如果X是Y的加强版，这种加强的程度叫“加速比”(speedup)

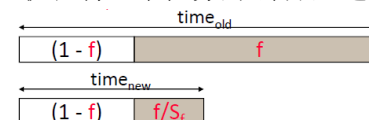
$$S = \text{时间}_{\text{未加强之前}} / \text{时间}_{\text{加强之后}} \\ = \text{时间}_Y / \text{时间}_X$$

89

89

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”

90

90

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

91

91

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)

92

92

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!!!

93

93

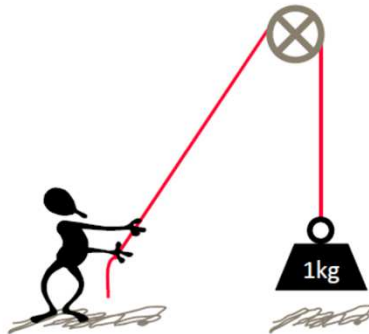
什么是能耗？

94

94

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

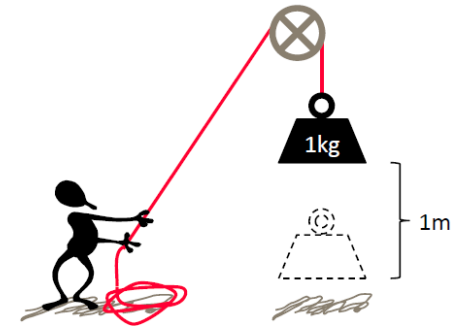


95

95

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

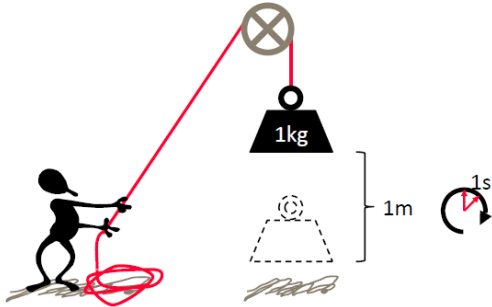


96

96

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



97

97

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 \propto 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 \propto 性能

如果性能 \propto 频率，那么一切将变得简单
功率 $= (\frac{1}{2} CV^2) \cdot f$

98

98

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”

99

99

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%
- $P_{total} = E_{total} / T = (C_{switch} W + C_{static} T) / T = C_{switch} C_{perf} + C_{static}$

100

100

简而言之，

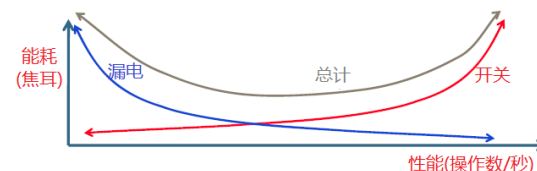
- 对于一个做功为 W 的任务
 - $T = W / c_{perf}$
做功（工作量）越少的任务执行得越快
 - $E = E_{switch} + E_{static} = (c_{switch} + c_{static} / c_{perf}) \cdot W$
做功（工作量）越少的任务消耗的能量越少
 - $P = P_{switch} + P_{static} = c_{switch} c_{perf} + c_{static}$
功率与任务不相关
- 现实是
 - W 不是标量
 - c 既不是标量也不是常数
 - $\frac{1}{2}CV^2$ 和 $\frac{1}{2}CV^2f$ 本身是非常“粗略”的近似值

101

101

关于静态功率的特别说明

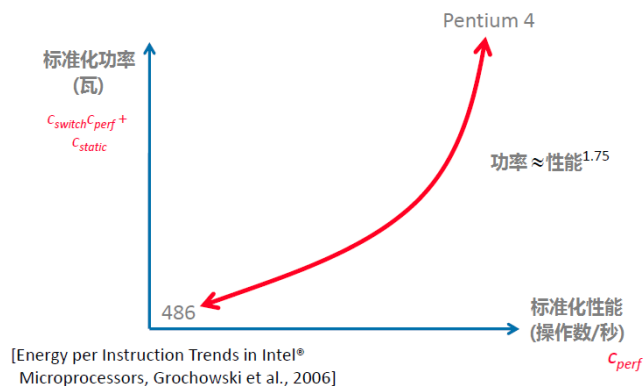
- $E_{total} = (c_{switch} + c_{static} / c_{perf}) \cdot W = \underbrace{c_{switch} \cdot W}_{\text{开关}} + \underbrace{c_{static} \cdot W / c_{perf}}_{\text{静态}}$
- 执行得更慢(调慢时钟)会降低功率消耗，但会由于漏电的存在而增加能量消耗
- 执行得更快(必须改进设计)需要让 c_{switch} 和 c_{static} 有超线性的增长



102

102

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

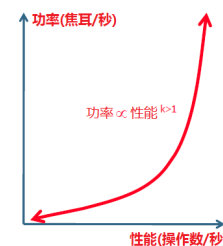


103

103

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
 - 较低的频率需要较低的电压
 - “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



104

104

为什么能耗对今天的计算机体系结构如此重要？

105

105

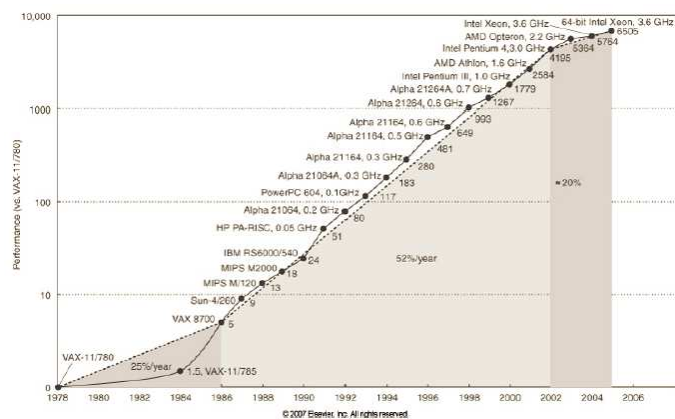
技术缩放的入门知识

- 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
 - $V_{dd}=0.7x$ (恒定磁场)或 $V_{dd}=1x$ (恒定电压)
 - 功率= $C \times V^2 \times f = 0.5x$ (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)

106

106

摩尔定律的一种表现形式



107

107

摩尔定律→性能

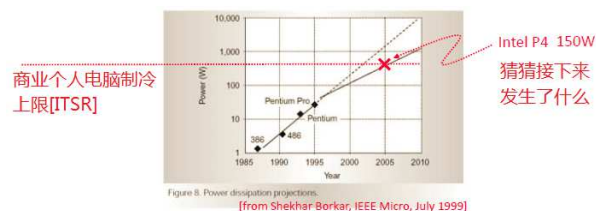
- 根据缩放理论的说法，我们应该得到
 - @恒定复杂性：以1x晶体管数获得1.43x频率
→ 1.43x性能, 0.5x功耗
 - @最大复杂度：以2x晶体管数获得1.43x频率
→ 恒定功率下2.8x性能
- 实际上，我们得到了(高性能CPU)
 - 2x晶体管数
 - 2x频率(注意：比缩放理论快)
 - 总的来说，我们以约2x功率获得约2x性能

108

108

性能效率低下

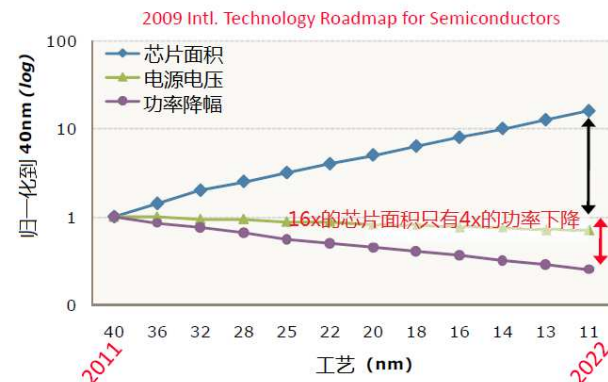
- 为了在单线程微处理器上达到“预期”的性能目标
 - 通过增加流水线深度越来越难提高频率
 - 使用2x晶体管构建更复杂的微体系结构(cache、分支预测、超标量、乱序执行), 以使更快/更深的流水线不会停顿
- 性能效率低下的后果是



109

109

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

110

110

频率和电压缩放

- 每次转换的开关能量为 $\frac{1}{2}CV^2$ (寄生电容建模)
- 每秒 f 次转换的开关功率为 $\frac{1}{2}CV^2f$
- 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
 - $V \rightarrow V'$, 因此 $\frac{1}{2}CV^2 \rightarrow \frac{1}{2}CV'^2$

漏电流/功率也由于更低的电压 V' 而超线性降低!!!

111

111

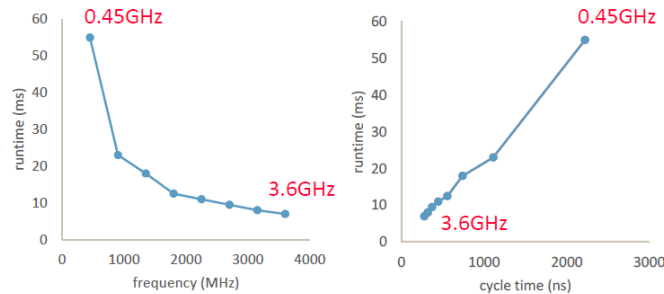
频率缩放

- 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

112

112

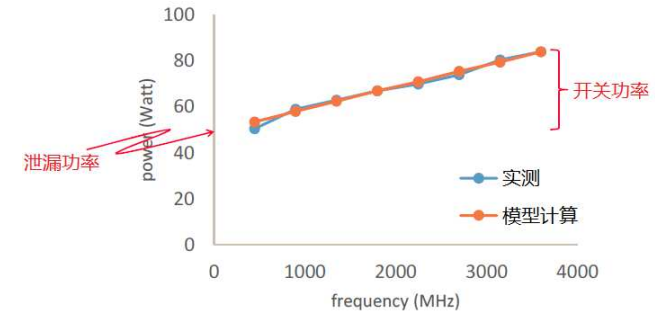
Intel P4 660 频率缩放: FFT_{64K}



circa 2005, 90nm

113

Intel P4 660 频率缩放: FFT_{64K}

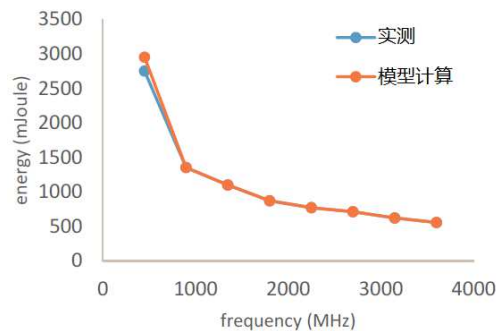


$C_{perf} = 145$ 操作数/秒; $C_{switch} = 0.24$ 焦耳/操作; $C_{static} = 49.4$ 瓦
(归一化到 $W=1$)

circa 2005, 90nm

114

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

115

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
- $E \propto V^2$, 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

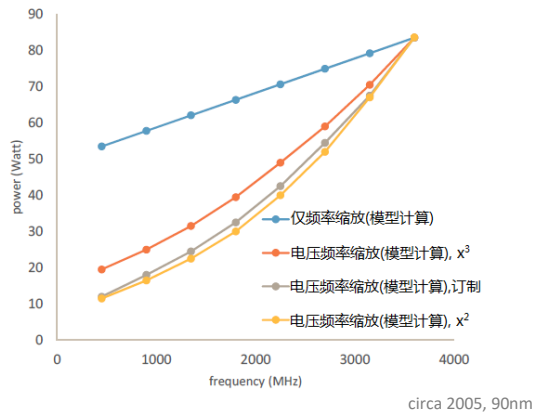
架构师们通常进行粗略的简化

116

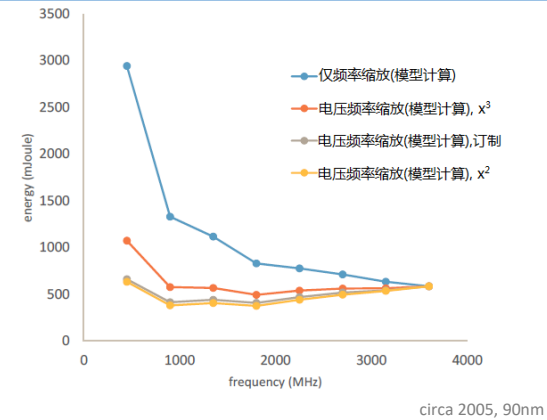
116

115

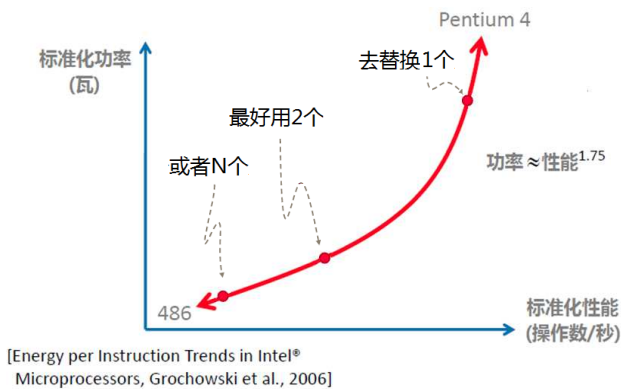
Intel P4 660 电压频率缩放: FFT_{64K}



Intel P4 660 电压频率缩放: FFT_{64K}



并行化



并行化

• 使用 N 个处理器时理想的并行化

$$T = W / (c_{\text{perf}} N)$$

$$E = (c_{\text{switch}} + c_{\text{static}} / c_{\text{perf}}) \cdot W$$

注意: $4x$ 的静态功率, $4x$ 的执行时间提升

$$P = N (c_{\text{switch}} c_{\text{perf}} + c_{\text{static}})$$

“我们曾经这样想”

• 或者, 假设 $s_{\text{voltage}} \approx s_{\text{freq}}$, 我们可以基于 $s_{\text{freq}} = 1/N$ 用加速比 N 来换取功率和能量的下降

$$T = W / c_{\text{perf}}$$

$$E'' = (c_{\text{switch}} / N^2 + c_{\text{static}} / (c_{\text{perf}} N)) \cdot W$$

$$P'' = c_{\text{switch}} c_{\text{perf}} / N^2 + c_{\text{static}} / N$$

“我们现在这样想”

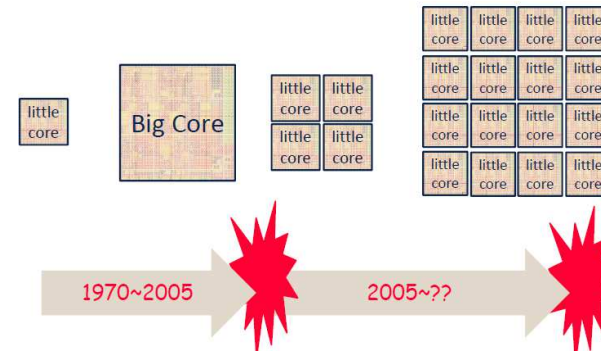
所以问题在哪儿？

- 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？

121

121

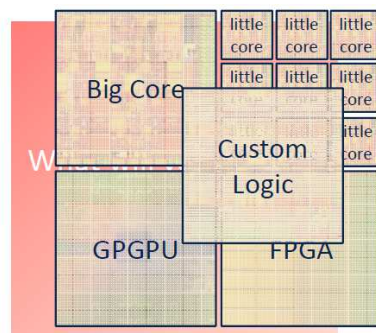
摩尔定律通过核数的增加继续扩展



122

122

记住：性能/瓦特和操作数/焦耳



123

123