



Classes of Computers



计算机的分类

- **个人移动设备 (PMD)**
 - 智能手机、平板电脑
 - ARM-ISA兼容的通用处理器芯片 (SoC) 在市场上处于统治地位
 - SoC中包含大量的专用加速器 (radio, image, video, graphics, audio, motion, location, security, etc.)
 - 强调能效和实时性 (energy efficiency and real-time)
- **桌面计算 (Desktop Computing)**
 - 强调性价比 (price-performance)
- **服务器 (Servers)**
 - 强调可用性、可缩放性、吞吐率 (availability, scalability, throughput)



Cost of downtime

Application	Cost of downtime per hour	Annual losses with downtime of		
		1% (87.6 h/year)	0.5% (43.8 h/year)	0.1% (8.8 h/year)
Brokerage service	\$4,000,000	\$350,400,000	\$175,200,000	\$35,000,000
Energy	\$1,750,000	\$153,300,000	\$76,700,000	\$15,300,000
Telecom	\$1,250,000	\$109,500,000	\$54,800,000	\$11,000,000
Manufacturing	\$1,000,000	\$87,600,000	\$43,800,000	\$8,800,000
Retail	\$650,000	\$56,900,000	\$28,500,000	\$5,700,000
Health care	\$400,000	\$35,000,000	\$17,500,000	\$3,500,000
Media	\$50,000	\$4,400,000	\$2,200,000	\$400,000

Figure 1.3 Costs rounded to nearest \$100,000 of an unavailable system are shown by analyzing the cost of downtime (in terms of immediately lost revenue), assuming three different levels of availability, and that downtime is distributed uniformly. These data are from Landstrom (2014) and were collected and analyzed by Contingency Planning Research.

availability

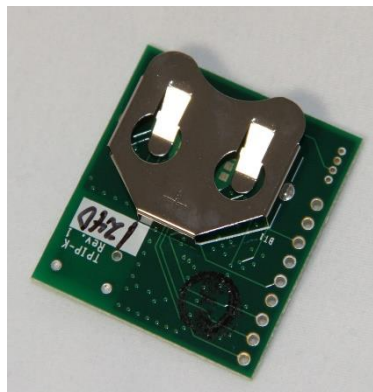
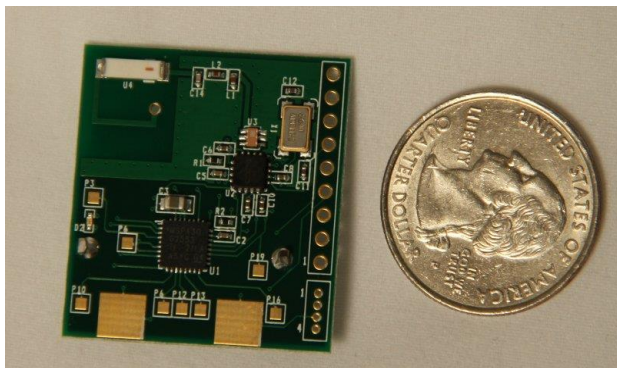


计算机的分类（续）

- **集群/仓储级计算机（Clusters / Warehouse Scale Computers）**
 - 每个数据中心包含10+万个处理器核
 - X86-ISA兼容的服务器芯片在市场上占统治地位
 - 专门的应用程序，加上虚拟机的云托管
 - 现在越来越多地使用gpu、fpga和定制硬件来加速工作负载
 - 其分支：Supercomputers：强调浮点运算性能和高速内部互联
 - 强调可用性和性价比(availability、price-performance、energy)

计算机的分类 (续)

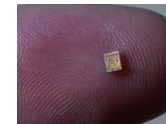
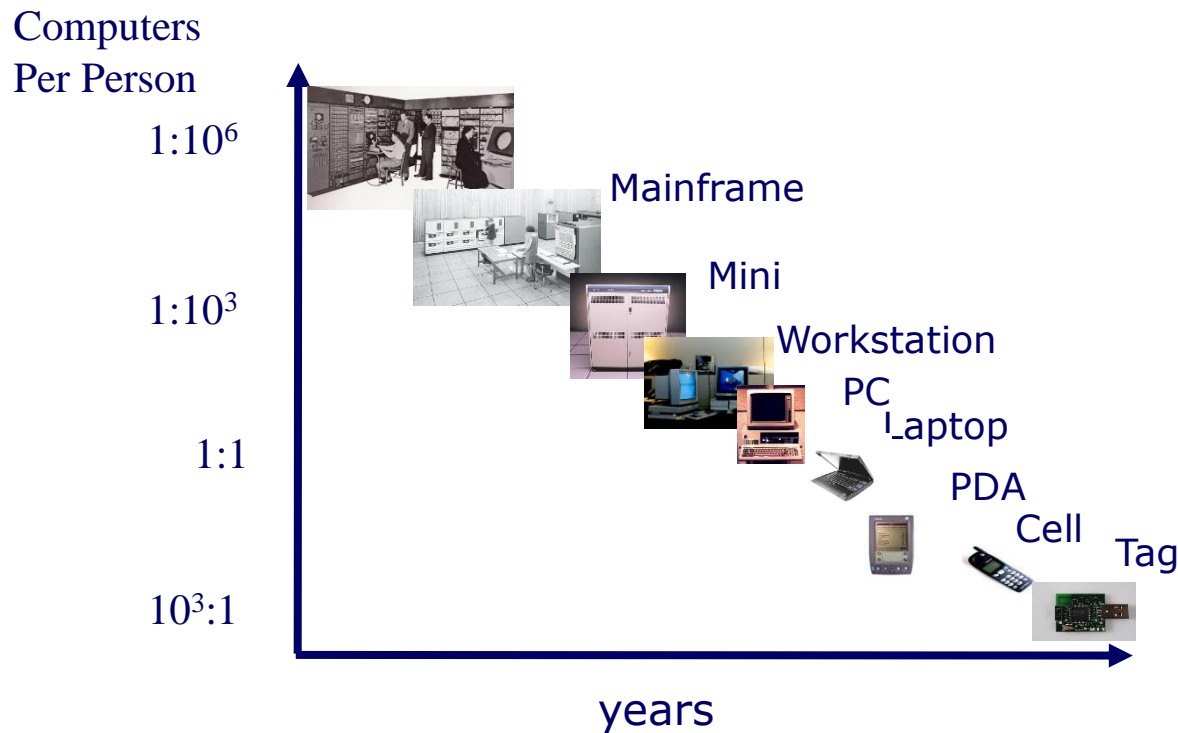
- **嵌入式计算机/物联网 (Embedded Computers / Internet of Things)**
 - 有线/无线网络基础设施, 打印机
 - 消费类产品
(TV/Music/Games/Automotive/Camera/MP3)
 - **物联网** (Internet of Things!)
 - 强调价格、能耗及面向特定应用的性能(Emphasis: price、energy、application-specific performance)





The Opportunity

Bell's Law: a new computer class emerges every 10 years





五类主流计算系统特点

Feature	Personal mobile device (PMD)	Desktop	Server	Clusters/warehouse-scale computer	Internet of things/embedded
Price of system	\$100–\$1000	\$300–\$2500	\$5000–\$10,000,000	\$100,000–\$200,000,000	\$10–\$100,000
Price of microprocessor	\$10–\$100	\$50–\$500	\$200–\$2000	\$50–\$250	\$0.01–\$100
Critical system design issues	Cost, energy, media performance, responsiveness	Price-performance, energy, graphics performance	Throughput, availability, scalability, energy	Price-performance, throughput, energy proportionality	Price, energy, application-specific performance

Figure 1.2 A summary of the five mainstream computing classes and their system characteristics. Sales in 2015 included about 1.6 billion PMDs (90% cell phones), 275 million desktop PCs, and 15 million servers. The total number of embedded processors sold was nearly 19 billion. In total, 14.8 billion ARM-technology-based chips were shipped in 2015. Note the wide range in system price for servers and embedded systems, which go from USB keys to network routers. For servers, this range arises from the need for very large-scale multiprocessor systems for high-end transaction processing.



Closer Look at Processor Technology

Closer Look at Processor Technology

- **特征尺寸不断减小 (feature size) , 芯片集成度不断提高**

什么是特征尺寸? (x, y) 维度上最小的尺寸

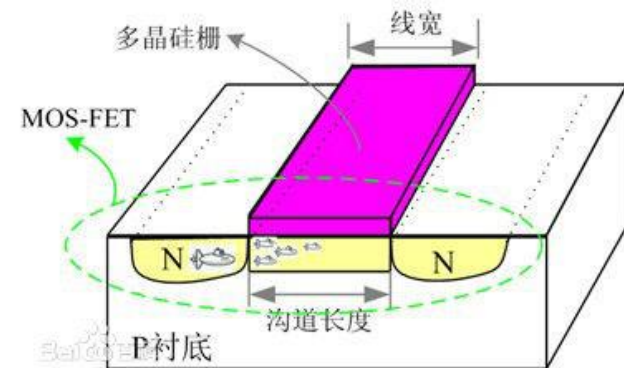
- 电路的速度在不断提高
- 芯片的面积在不断增加 (10%-20%)
- 主频在提高 (功耗成为问题) $C \cdot V^2 \cdot A \cdot f$
- 单位面积的晶体管数量在增加

- **性能每10年提高100倍**

- 时钟频率提高了10倍
- DRAM 的密度每3年提高4倍

- **如何使用这些晶体管?**

- Parallelism in processing: 有更多的功能部件
 - 每个时钟周期并行多个操作降低了CPI
- Locality in data access: 更大的Cache
 - 降低了访存的延时从而降低了CPI, 提高了处理器的利用率



金属氧化物半导体场效应晶体管



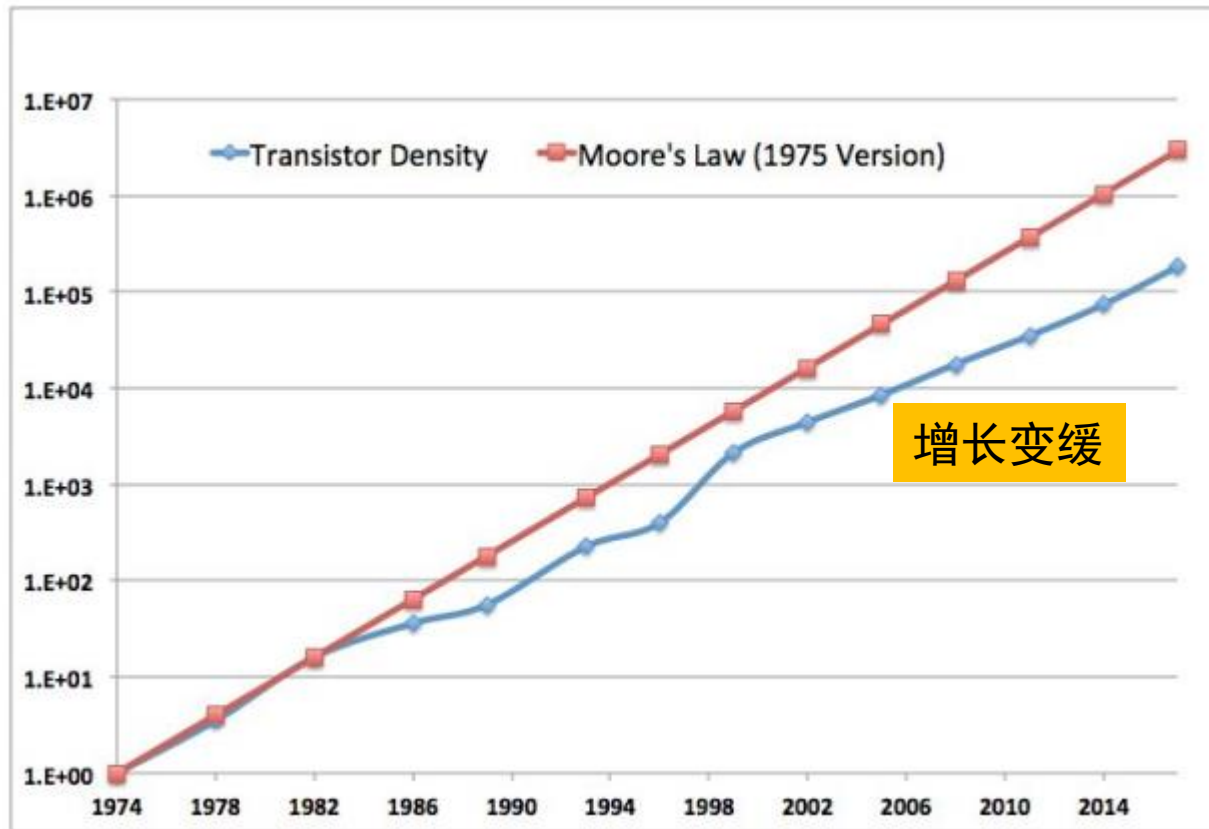
Architecture: Increase in Parallelism

- **1985年以前位级(4-8-16位)并行 (Bit level parallelism)**
 - 32-bit 处理器后性能提升变慢
 - 90年代采用64-bit、128-bit 及更高位 通过向量处理提高性能
 - 重大的拐点: 32位处理器和缓存适合一个芯片
- **80年代中到90年代后期主要发展指令级并行 (Instruction Level Parallelism)**
 - 流水线、RISC、编译技术的进步 → 挖掘指令级并行
 - 片上cache及功能部件的增加 → 超标量执行(superscalar)
 - 更精巧的技术: 乱序执行(out of order execution)和硬件投机执行(speculative execution)
- **现状: 线程级并行和片上多处理器 (thread level parallelism and chip multiprocessors)**
 - 线程级并行超越了指令集并行
 - 在单个芯片中部署多个处理器及互联结构
 - 在处理器芯片内多个线程并行执行

硬件线程+ 软件线程

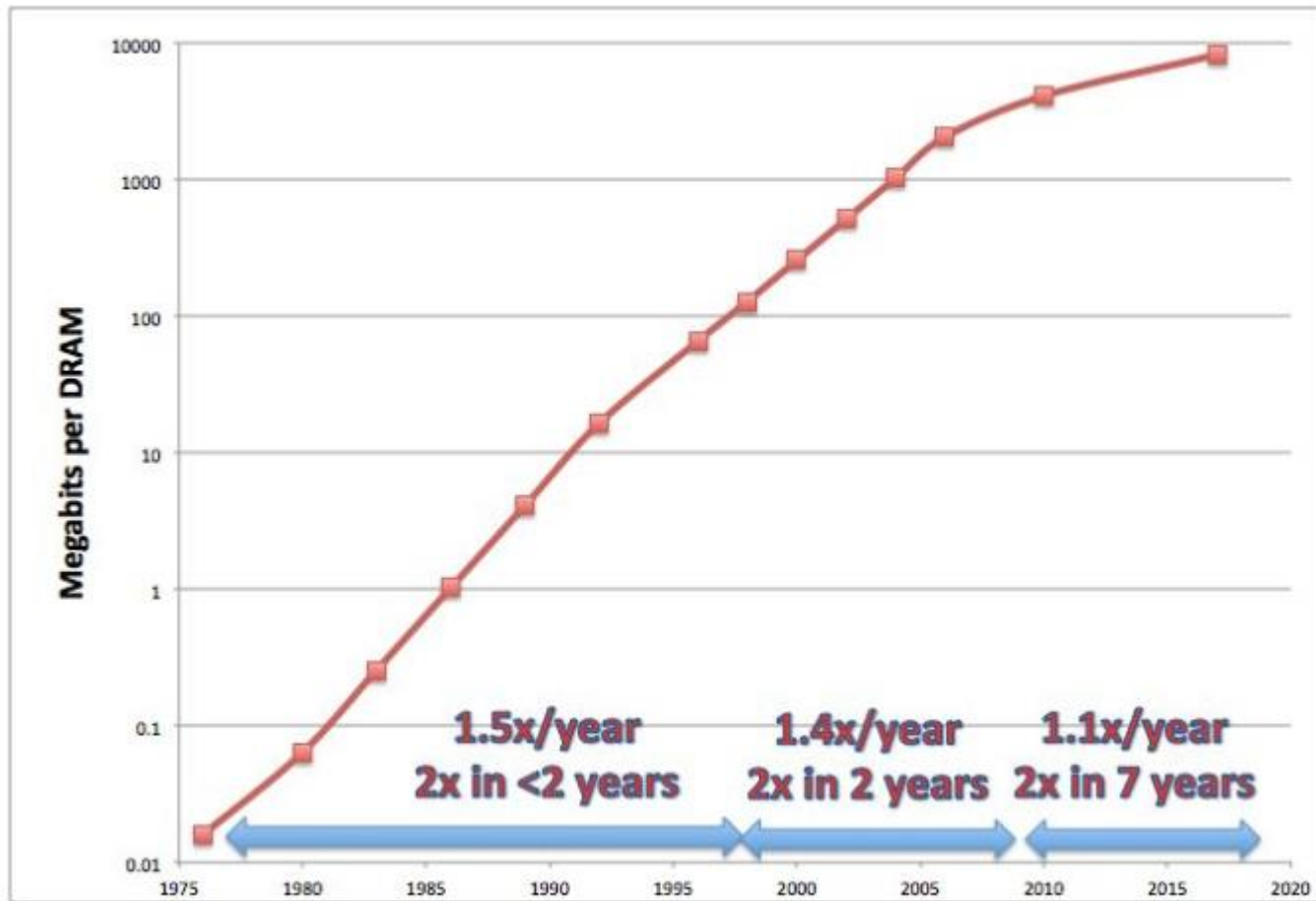


Moore's Law Slowdown in Intel Processors



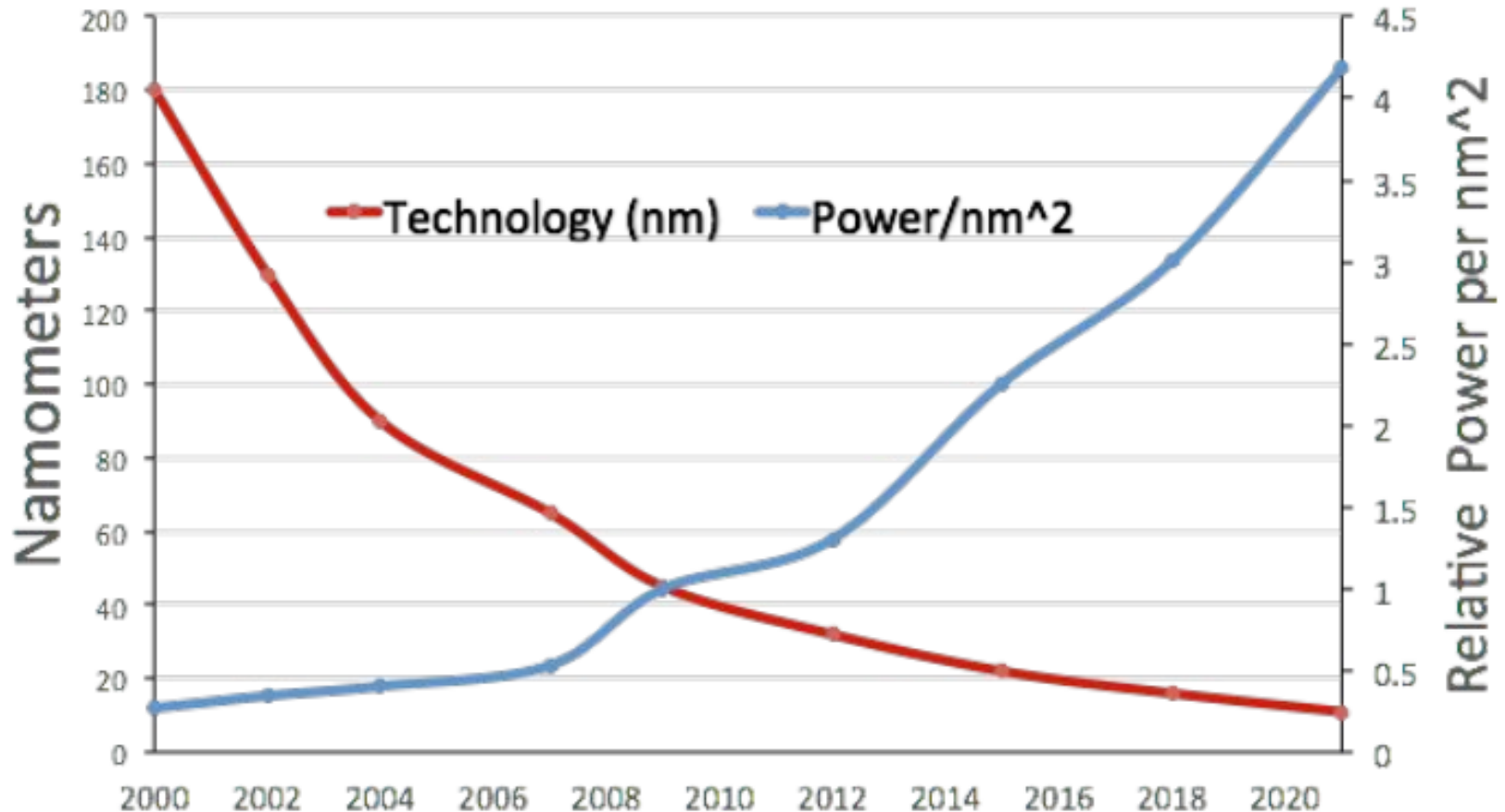


Moore's Law in DRAMs





Technology & Power: Dennard Scaling



Energy scaling for fixed task is better, since more and faster transistors



Why is low-power Important?

- **Power must be brought in and distributed around the chip**
- **Power is dissipated as heat and must be removed**
- **Three things to consider**
 - Peak power (电源能够提供的功率 < 处理器需要的最大功率 => 电压下降 => 无法正常工作)
 - Thermal design power (TDP)热设计功耗, 决定了电路的冷却需求 (often TDP > 平均功率)
 - Energy能耗 or power功耗? Which one is a better metric?

能耗！和完成一件任务的时间有关 $E = pt$



Power in Integrated Circuits

- **功耗问题是当今计算机设计面临的最大挑战之一**
 - 芯片内部和外围电路都存在功耗问题
 - 功耗会产生热，须解决散热问题
- **热设计功率 (Thermal Design Power (TDP))**
 - Intel: 热设计功率，一种基于最坏情况应用的功率耗散目标
 - 表达了设备的功耗特征
 - 主要用于**电源和冷却系统**的设计
 - 通常它低于峰值功耗，但高于平均功耗
- **TDP和功耗的关系?**
 - 答案是没有关系。很多测试表明，CPU的最大真实功耗小于TDP值，但是在加压超频后峰值功耗有可能超过TDP。
- **降低频率可导致功耗下降**



Power versus Energy

- **功耗(Power) 指单位时间的能耗: $1 \text{ Watt} = 1 \text{ Joule} / \text{Second}$**
- **一个任务执行的能耗 (energy) = Average Power \times Execution Time**
- **Power or Energy? 哪个指标更合适?**
 - 针对给定的任务, 能耗是一种更合适的度量指标 (joules)
 - 针对电池供电的设备, 我们需要关注能效
- **Example: which processor is more energy efficient?**
 - Processor A consumes 20% more power than B on a given task
 - However, A requires only 70% of the execution time needed by B
- **Answer: Energy consumption of A = $1.2 \times 0.7 = 0.84$ of B**
 - Processor A consumes less energy than B (more energy-efficient)



Power & Energy

- **Dynamic Energy动态能耗** \propto
Capacitive Load \times Voltage²
 - 从0-1-0 或 1-0-1逻辑跃迁的脉冲能量
 - Capacitive Load = 输出晶体管和导线的电容负载
 - 20年来晶体管供电电压已经从5V降到1V
- **Dynamic Power** \propto
Capacitive Load \times Voltage² \times Frequency Switched
时钟周期

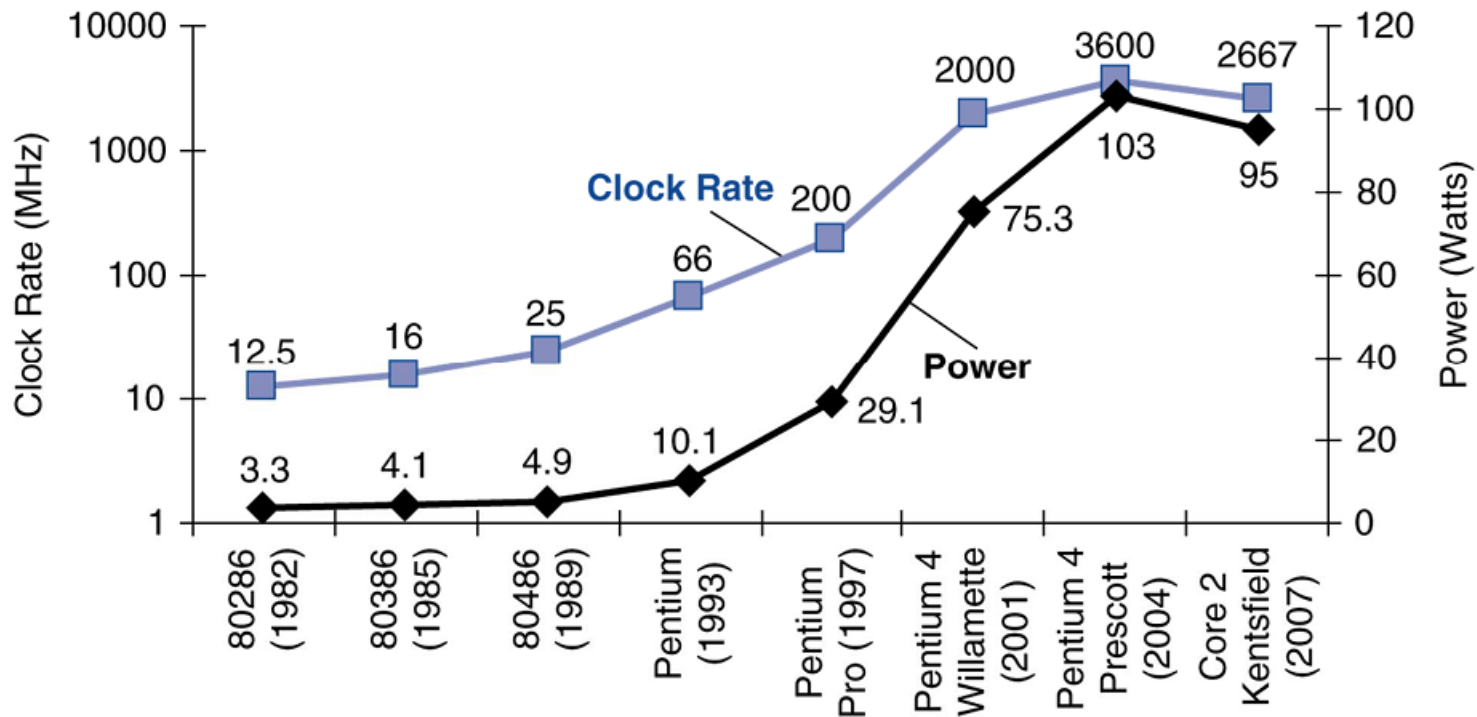


动态能耗和功耗

- 针对CMOS技术, 动态的能量消耗是由于晶体管的on和off状态的切换引起的
- **Dynamic Energy \propto Capacitive Load \times Voltage²**
 - the energy of pulse of the logic transition of 0-1-0 or 1-0-1
 - Capacitive Load = Capacitance of output transistors & wires
 - Voltage has dropped from 5V to below 1V in 20 years
- **Dynamic Power \propto Capacitive Load \times Voltage² \times Frequency Switched**
- 降低频率可以降低功耗
- 降低频率导致执行时间增加 -> 不能降低能耗
- 降低电压可有效降低功耗和能耗



Trends in Power & Clock Frequency



$\text{Power} \propto \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$

30X

5V → 1V

1000X



减少动态功耗的技术

- **关闭不活动模块或处理器核的时钟 (Do nothing well)**
 - Such as a floating-point unit when there are no FP instructions
- **Dynamic Voltage-Frequency Scaling (DVFS)**
 - 在有些场景不需要CPU全力运行
 - 降低电压和频率可降低功耗
- **针对典型场景特殊设计 (Design for the typical case)**
 - 电池供电的设备常处于idle状态, DRAM和外部存储采用低功耗模式工作以降低能耗
- **Overclocking (Turbo Mode)**
 - 当在较高频率运行安全时, 先以较高频率运行一段时间, 直到温度开始上升至不安全区域
 - 一个core以较高频率运行, 同时关闭其他核



Dynamic Voltage Scaling (DVS)

- DVS scales the operating **voltage** of the processor along with the frequency.
- Since energy is proportional to v^2 , ($v \propto f$) DVS can potentially provide significant energy savings through **frequency and voltage scaling** => **DVFS**
- can reduce voltage and frequency by (say) 10%; can slow a program by (say) 8%, but reduce dynamic power by 27%, reduce total power by (say) 23%, reduce total energy by 17%

有一次注意到当服务器的工作负载增加时，完成同一个工作需要的时间却变短了。能解释下这可能是为什么吗？



低功耗技术-DVFS

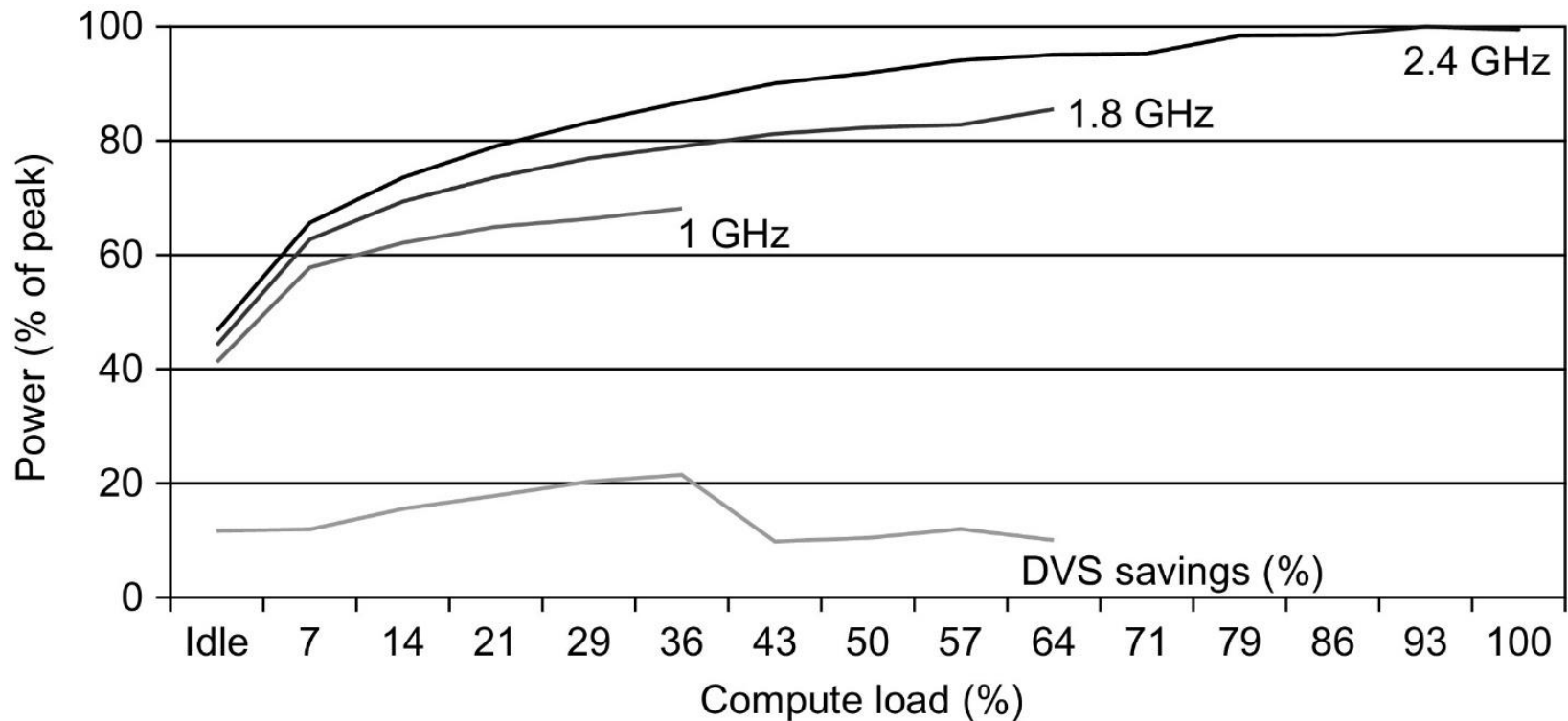


Figure 1.12 Energy savings for a server using an AMD Opteron microprocessor, 8 GB of DRAM, and one ATA disk. At 1.8 GHz, the server can handle at most up to two-thirds of the workload without causing service-level violations, and at 1 GHz, it can safely handle only one-third of the workload (Figure 5.11 in Barroso and Hölzle, 2009).



静态功耗 (Static Power)

- **当晶体管处于off状态时,漏电流产生的功耗称为静态功耗**
- **随着晶体管尺寸的减少漏电流的大小在增加**
- **Static Power = Static Current × Voltage**
 - Static power increases with the number of transistors
- **静态功耗有时会占到全部功耗的50%**
 - Large SRAM caches need static power to maintain their values
- **Power Gating: 通过切断供电减少漏电流**
 - To inactive modules to control the loss of leakage current



有关功耗和能耗小结

- 给定负载情况下能耗越少，能效越高，特别是对电池供电的移动设备。
- 功耗应该被看作一个约束条件
 - A chip might be limited to 120 watts (cooling + power supply)
- **Power Consumed = Dynamic Power + Static Power**
 - 晶体管开和关的切换导致的功耗为动态功耗
 - 由于晶体管静态漏电流导致的功耗称为静态功耗
- 通过降低频率可节省功耗
- 降低电压可降低功耗和能耗

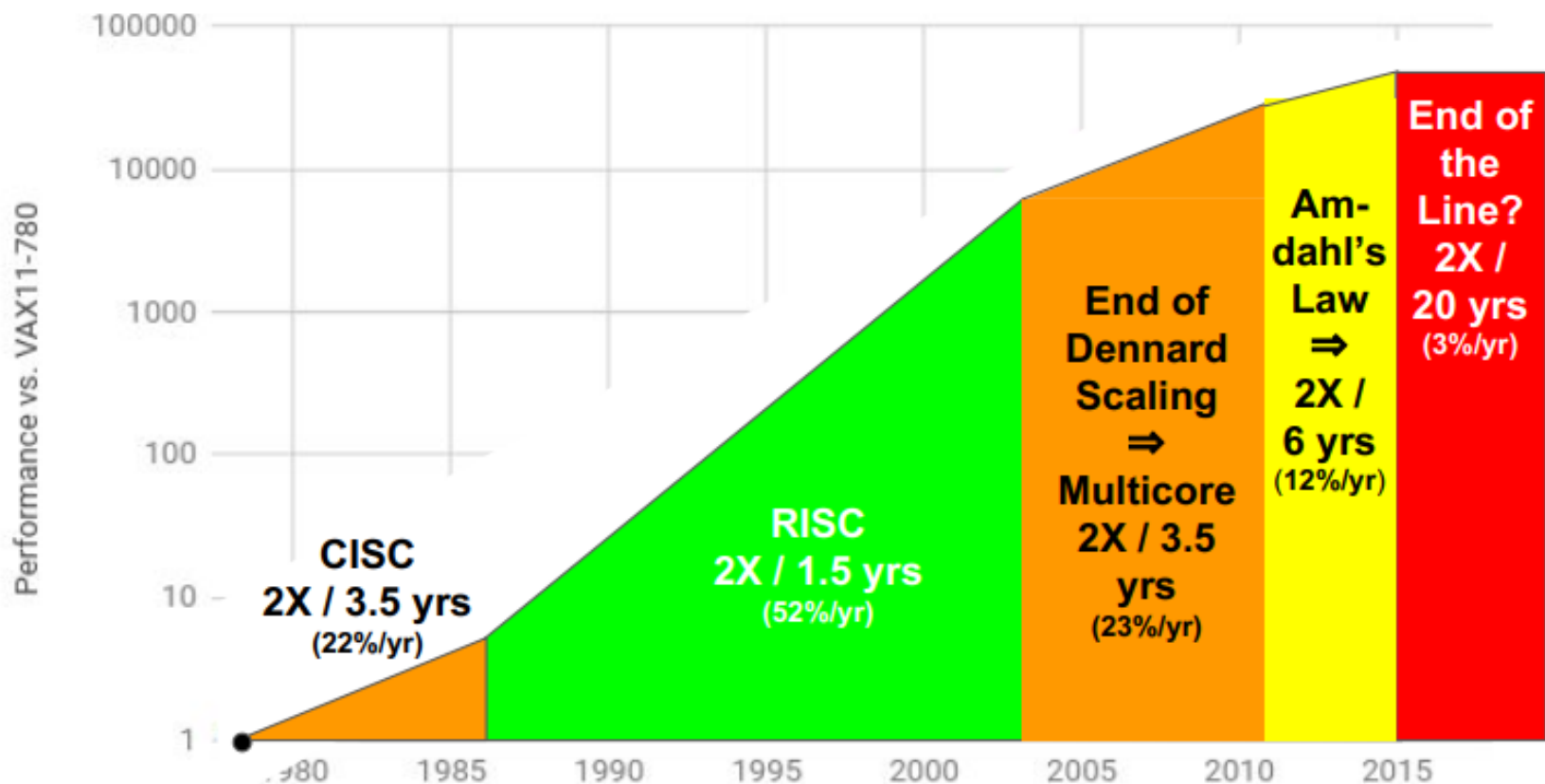


Current and Future Trends



End of Growth of Single Program Speed?

40 years of Processor Performance

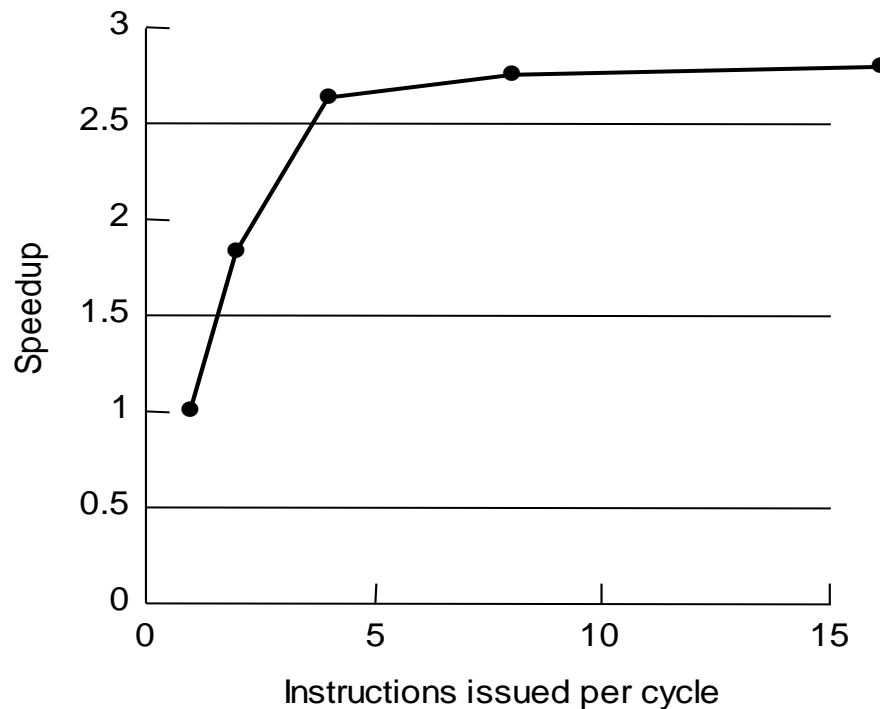
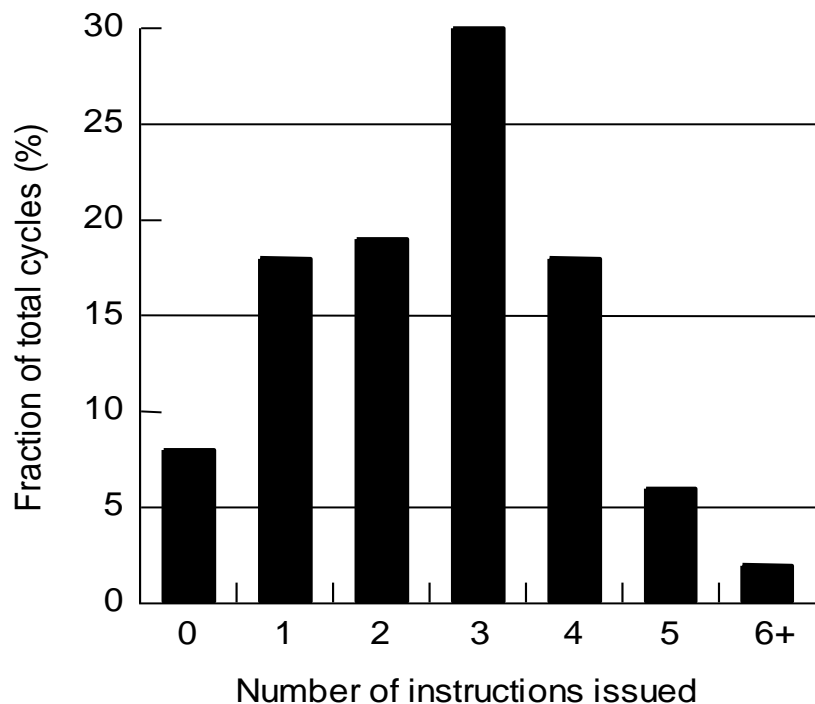


Based on SPECintCPU. Source: John Hennessy and David Patterson, Computer Architecture: A Quantitative Approach, 6/e. 2018

So, What is next?



How far will Instruction Level Parallelism (ILP) go?



理想超标量模型执行下有限的ILP：无限的资源和存取带宽、完善的预取和重命名机制，但是使用现实的Cache。结果：4条指令并行取得了最大的ILP benefit；5以后则收获甚微

平均来讲，每五条指令有一条control transfer instruction

Why do we have very limited ILP?



有关体系结构的新旧观念

- Old **C**onventional **W**isdom: Power is free, Transistors expensive
 - New CW: “**P**ower **w**all” Power expensive, Transistors free
 - Old CW: 通过编译、体系结构创新来增加指令级并行 (Out-of-order, speculation, VLIW, ...)
 - New CW: “**I**LP **w**all” 挖掘指令级并行的收益越来越小
 - Old CW: 乘法器速度较慢, 访存速度比较快
 - New CW: “**M**emory **w**all” 乘法器速度提升了, 访存成为瓶颈 (200 clock cycles to DRAM memory, 4 clocks for multiply)
 - Old CW: 单处理器性能 2X / 1.5 yrs
 - New CW: **P**ower **W**all + **I**LP **W**all + **M**emory **W**all = **B**rick **W**all
 - 单处理器性能 2X / 5(?) yrs
- ⇒ 芯片设计的巨大变化: multiple “cores”
(2X processors per chip / ~ 2 years)
- 越简单的处理器越节能



小结：计算机系统设计方面的巨大变化

- **在过去的50年，Moore's law和Dennard scaling(登纳德缩放比例定律)主宰着芯片产业的发展**
 - Moore 1965年预测：晶体管数量随着尺寸缩小按接近平方关系增长（每18个月2X）
 - Dennard 1974年预测：晶体管尺寸变小，功耗会同比变小（相同面积下功耗不变）
 - 工艺技术的进步可在不改变软件模型的情况下，持续地提高系统性能/能耗比
- **最近10年间，工艺技术的发展受到了很大制约**
 - Dennard scaling over (supply voltage ~fixed)
 - Moore's Law (cost/transistor) over?
 - Energy efficiency constrains everything
 -
- **功耗问题成为系统结构设计必须考虑的问题**
- **软件设计者必须考虑：**
 - Parallel systems
 - Heterogeneous systems



小结：体系结构挑战性问题

- **性能提升处于停滞状态**
 - Moore定律正在减速
 - 邓纳德缩放定律正在失效
 - 微体系架构技术：指令集并行、多核等技术的低效会消耗能量



现代体系结构发展趋势

- 性能提升的基本手段：**并行**
- 应用需求
 - 计算的需求不断增长，如Scientific computing, video, graphics, databases, deep learning...
- 工艺发展的趋势
 - 芯片的集成度不断提高，但提升的速度在放缓
 - 时钟频率的提高在放缓，并有降低的趋势
- 体系结构的发展及机遇
 - 指令集并行受到制约
 - 线程级并行和数据级并行是发展的方向
 - 提高单处理器性能花费的代价呈现上升趋势
 - 面向特定领域的体系结构正蓬勃发展



Summary

- **计算机体系结构的基本概念**
 - ISA+Organization+Implementation
- **本课程将涉及的主要内容**
 - 简单机器设计 (ISA, 基本流水线) 指令级并行
 - 存储系统 (Cache, Virtual Memory)
 - 复杂流水线 (动态指令流调度、动态分支预测)
 - 显式并行处理器 (向量处理器、VLIW, 多线程处理)
 - 多处理器结构
- **体系结构设计面临的新问题**
 - Power Wall + ILP Wall + Memory Wall = Brick Wall



1.2 定量分析技术基础

- **性能的含义**
- **CPU性能度量**
- **计算机系统性能度量**



Defining CPU Performance

- X比Y性能高的含义是什么?
- Ferrari vs. School Bus?
- 2013 Ferrari 599 GTB
 - 2 passengers, 11.1 secs for quarter mile
- 2013 Type D school bus
 - 54 passengers, 36 secs quarter mile time
- 响应时间: e.g., time to travel $\frac{1}{4}$ mile
- 吞吐率/带宽: e.g., passenger-mi in 1 hour





性能的含义

Plane	DC to Paris	Speed	Passengers	Throughput (pmp)
Boeing 747	6.5 hours	610 mph	470	286,700
BAD/Sud Concorde	3 hours	1350 mph	132	178,200

哪个性能高？

- Time to do the task (**Execution Time**)
 - execution time, response time, **latency**
- Tasks per day, hour, week, sec, ns. .. (**Performance**)
 - **throughput**, bandwidth

这两者经常会有冲突的。



举例

- Time of Concord (协和) vs. Boeing 747?
 - Concord is $1350 \text{ mph} / 610 \text{ mph} = 2.2$ times faster than Boeing
= 6.5 hours / 3 hours
- Throughput of Concorde vs. Boeing 747 ?
 - Concord is $178,200 \text{ pmph} / 286,700 \text{ pmph} = 0.62$ "times faster"
 - Boeing is $286,700 \text{ pmph} / 178,200 \text{ pmph} = 1.60$ "times faster"
- Boeing is 1.6 times ("60%") faster in terms of throughput
- Concord is 2.2 times ("120%") faster in terms of flying time

我们主要关注单个任务的执行时间

程序由一组指令构成，指令的吞吐率 (Instruction throughput) 非常重要!



以时间 (Time) 度量性能

- **Response Time**

- 从任务开始到任务完成所经历的时间
- 通常由最终用户观察和测量
- 也称Wall-Clock Time or Elapsed Time
- $\text{Response Time} = \text{CPU Time} + \text{Waiting Time (I/O, scheduling, etc.)}$

- **CPU Execution Time**

- 指执行程序（指令序列）所花费的时间
- 不包括等待I/O或系统调度的开销
- 可以用秒(msec, μsec , ...), 或
- 可以用相对值（CPU的时钟周期数 (clock cycles))



以吞吐率度量性能

- **Throughput = 单位时间完成的任务数**
 - 任务数/小时、事务数/分钟、100Mbits/s
- **缩短任务执行时间可提高吞吐率 (throughput)**
 - Example: 使用更快的处理器
 - 执行单个任务时间少 \Rightarrow 单位时间所完成的任务数增加
- **硬件并行可提高吞吐率和响应时间(response time)**
 - Example: 采用多处理器结构
 - 多个任务可并行执行, 单个任务的执行时间没有减少
 - 减少等待时间可缩短响应时间



相对性能

- 某程序运行在X系统上

$$performance(x) = \frac{1}{execution_time(x)}$$

- “X 性能是Y的n倍” 是指

- $$n = \frac{Performance(x)}{Performance(y)}$$



CPU性能度量

- **Response time (elapsed time): 包括完成一个任务所需要的所有时间**

- User CPU Time (90.7)
- System CPU Time (12.9)
- Elapsed Time (2:39)

例如：unix 中的time命令

90.7s 12.9s 2:39 65% (90.7/159)



CPU 性能公式-CPI

CPU time = CPU clock cycles for a program \times Clock cycle time

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$



不同类型的指令具有不同的CPI

Let CPI_i = clocks per instruction for class i of instructions

Let IC_i = instruction count for class i of instructions

$$\text{CPU cycles} = \sum_{i=1}^n (CPI_i \times IC_i)$$

$$CPI = \frac{\sum_{i=1}^n (CPI_i \times IC_i)}{\sum_{i=1}^n IC_i} \quad Freq_i = \frac{IC_i}{\sum_{i=1}^n IC_i}$$

$$CPI = \sum_{i=1}^n (CPI_i \times Freq_i)$$



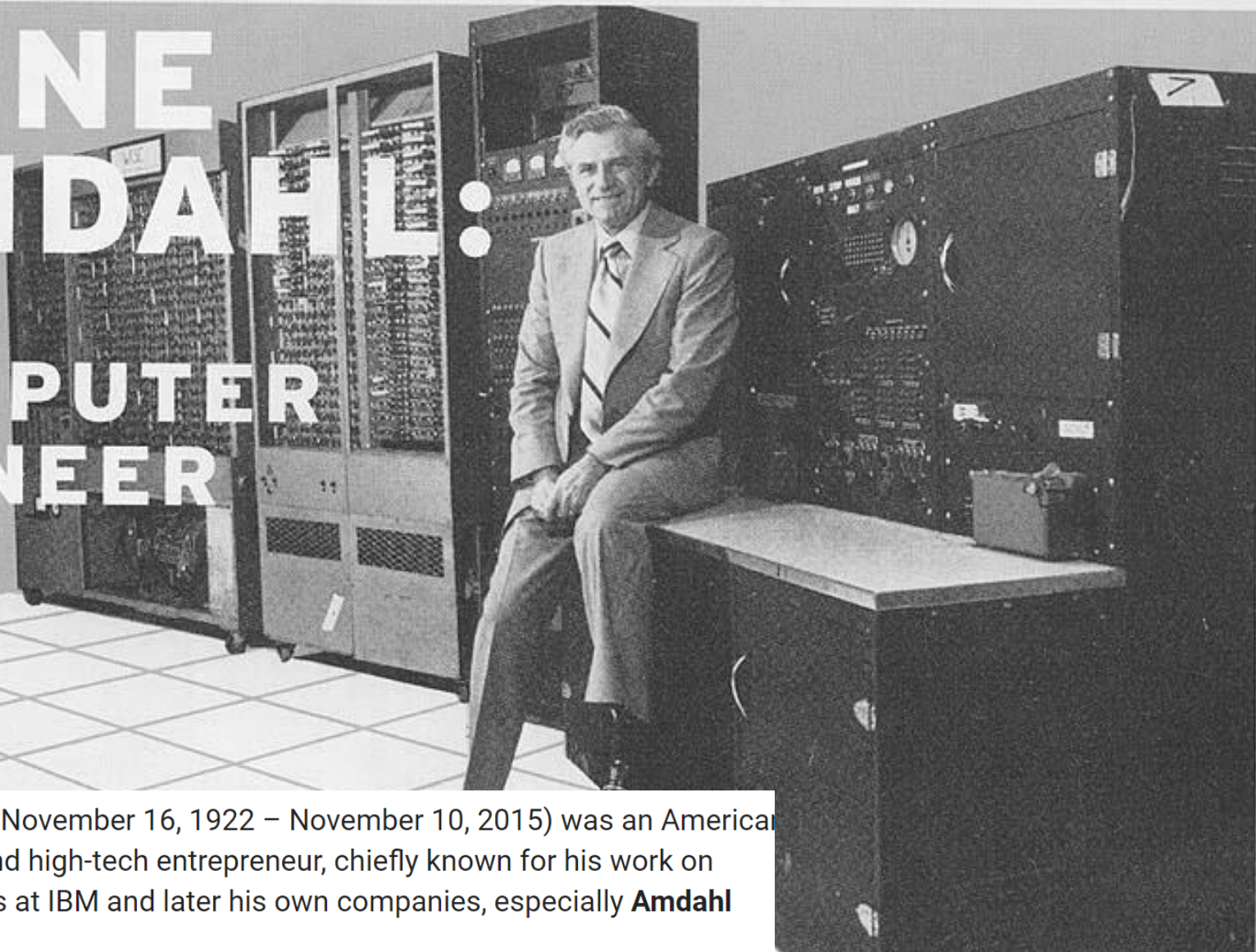
CPI计算举例

Base Machine (Reg / Reg)

Op	Freq	CPI_i	$CPI_i * F_i$	(% Time)
ALU	50%	1	.5	(33%)
Load	20%	2	.4	(27%)
Store	10%	2	.2	(13%)
Branch	20%	2	.4	(27%)
			<u>1.5</u>	

GENE AMD AHL: COMPUTER PIONEER

ALEXIS DANIELS



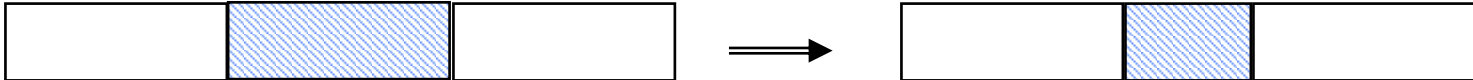
Gene Myron Amdahl (November 16, 1922 – November 10, 2015) was an American computer architect and high-tech entrepreneur, chiefly known for his work on mainframe computers at IBM and later his own companies, especially **Amdahl Corporation**.

3/15/2021



Amdahl's Law

假设对机器的部件进行了改进（加速比的概念）

$$\text{Speedup}(E) = \frac{\text{ExTime w/o } E}{\text{ExTime w/ } E} = \frac{\text{Performance w/ } E}{\text{Performance w/o } E}$$


假设可改进部分E在原来的计算时间所占的比例为F，而部件加速比为S，任务的其他部分不受影响，则

- $\text{ExTime}(\text{with } E) = ((1-F) + F/S) \times \text{ExTime}(\text{without } E)$
- $\text{Speedup}(\text{with } E) = 1/((1-F) + F/S)$

重要结论(性能提高的递减原则): 如果只针对整个任务的一部分进行优化，那么所获得的加速比不大于 $1/(1-F)$

Current processor has 1 core; next generation has n cores. Can we achieve a speedup of n?

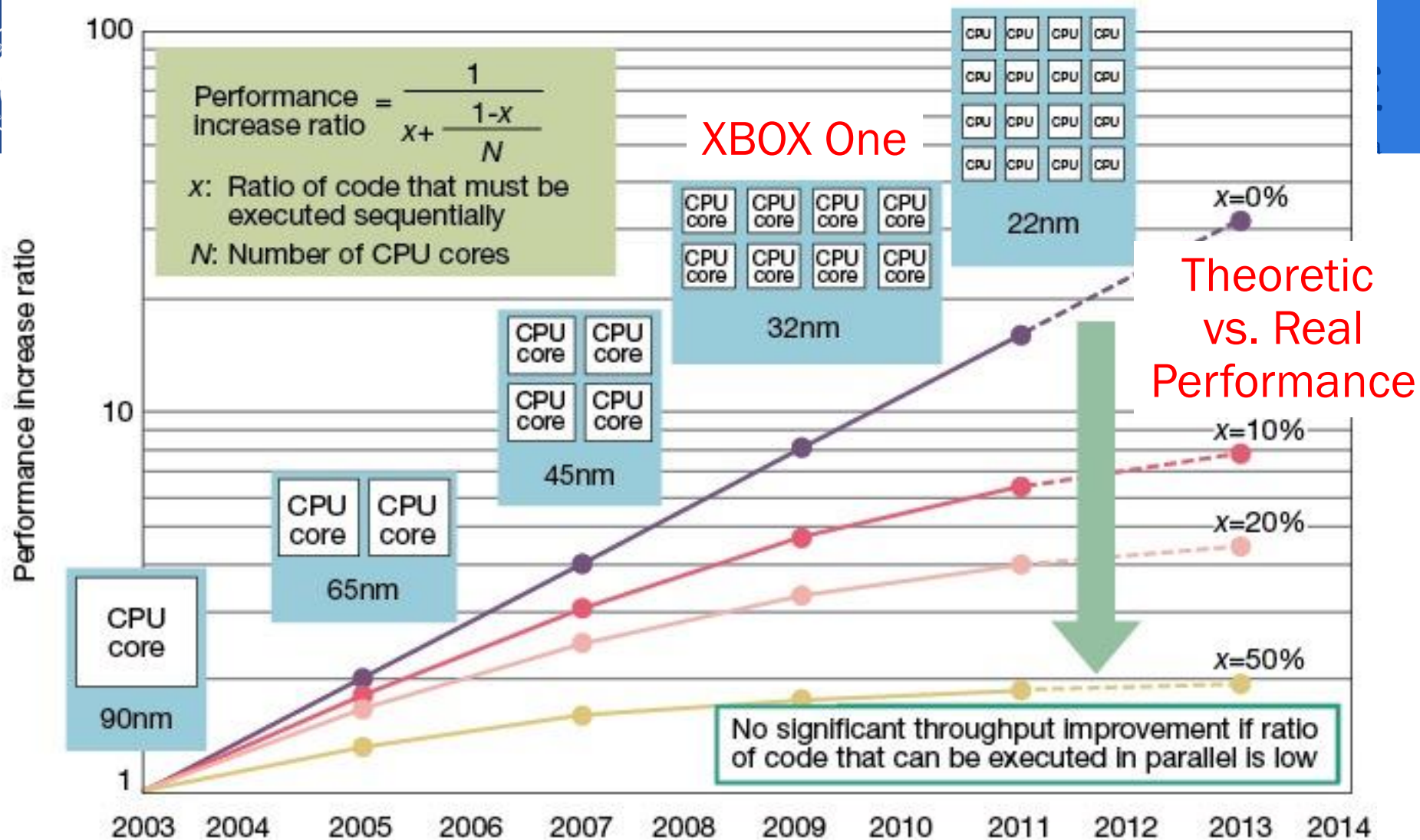


Fig 3 Amdahl's Law an Obstacle to Improved Performance Performance will not rise in the same proportion as the increase in CPU cores. Performance gains are limited by the ratio of software processing that must be executed sequentially. Amdahl's Law is a major obstacle in boosting multicore microprocessor performance. Diagram assumes no overhead in parallel processing. Years shown for design rules based on Intel planned and actual technology. Core count assumed to double for each rule generation.



举例

- **假设给定一体系结构硬件不支持乘法运算，乘法需要通过软件来实现。在软件中做一次乘法需要200个周期，而用硬件来实现只要4个时钟周期。如果假设在程序中有10%的乘法操作，问整个程序的加速比？ (1.11)**
- **如果有40%的乘法操作，问整个程序的加速比又是多少？ (2.45)**



举例

- **假设一计算机在运行给定的一程序时，有90%的时间用于处理某一类特定的计算。现将用于该类计算的部件性能提高到原来的10倍。**
 - 如果该程序在原来的机器上运行需100秒，那么该程序在改进后的机器上运行时间是多少？
 - 新的系统相对于原来的系统加速比是多少？
 - 在新的系统中，原来特定的计算占整个计算的比例是多少？



Amdahl定律的假设

- **Amdahl 定律的假设**

- 问题规模是常量
- 研究随着处理器数目的增加性能的变化

- **有别于Amdahl定律的另一视角**

- 我们通常用更快的计算机解决更大规模的问题
- 将处理时间视为常量，研究随着处理器数目的增加问题规模的增加情况（可扩放性）
- Gustafson-Barsis's Law （古斯塔夫森定律）



基本评估方法 - 市场评估方法

- **MIPS: 每秒百万条指令数 【millions of instructions per second】**
 - $\text{MIPS} = \text{IC} / (\text{execution time} \times 10^6) = 1 / (\text{CPI} \times T \times 10^6)$
 - MIPS依赖于指令集
 - 在同一台机器上, MIPS因程序不同而变化, 有时差别较大
 - MIPS可能与性能相反
 - 举例: 在一台load-store型机器上, 有一程序优化编译可以使ALU操作减少到原来的50%, 其他操作数量不变。
 - $F = 500\text{MHz}$
 - ALU (43% 1) loads (21% 2) stores (12% 2)
 - Branches (24% 2)
- **MFLOPS 基于操作而非指令, 它可以用来比较两种不同的机器。但MFLOPS也并非可靠, 因为不同机器上浮点运算集不同。CRAY-2没有除法指令, Motorola 68882有**
- **SPEC测试 (Standard Performance Evaluation Corporation)**



基本评估方法 - benchmark测试

- 五种类的测试程序（预测的精度逐级下降）

- (1) 真实程序：这是最可靠的方法，但是很多情况下不可行。
- (2) 修改过的程序：通过修改或改编真实程序来构造基准程序模块。原因：增强移植性或集中测试某种特定的系统性能
- (3) 核心程序(Kernels)：由从真实程序中提取的较短但很关键的代码构成。Livermore Loops及LINPACK是其中使用比较广泛的例子。一般是小几千行。
- (4) 小测试程序 (toy programs)：一般在100行以内。
- (5) 合成测试程序(Synthetic benchmarks)：首先对大量的应用程序中的操作进行统计，得到各种操作比例，再按这个比例人造出测试程序。Whetstone与Dhrystone是最流行的合成测试程序。



基准测试程序套件 (此页之后课堂没讲)

- **Embedded Microprocessor Benchmark Consortium (EEMBC)**
- **Desktop Benchmarks**
 - SPEC2017
 - SPEC2006
 - SPEC2000
 - SPEC 95
 - SPEC 92
 - SPEC 89
- **Server Benchmarks**
 - Processor Throughput-oriented benchmarks (基于SPEC CPU benchmarks->SPECrate)
 - SPECintFS, SPECWeb
 - Transaction-processing (TP) benchmarks (TPC-A, TPC-C, ...)
-

Standard Performance Evaluation Corporation (www.spec.org)

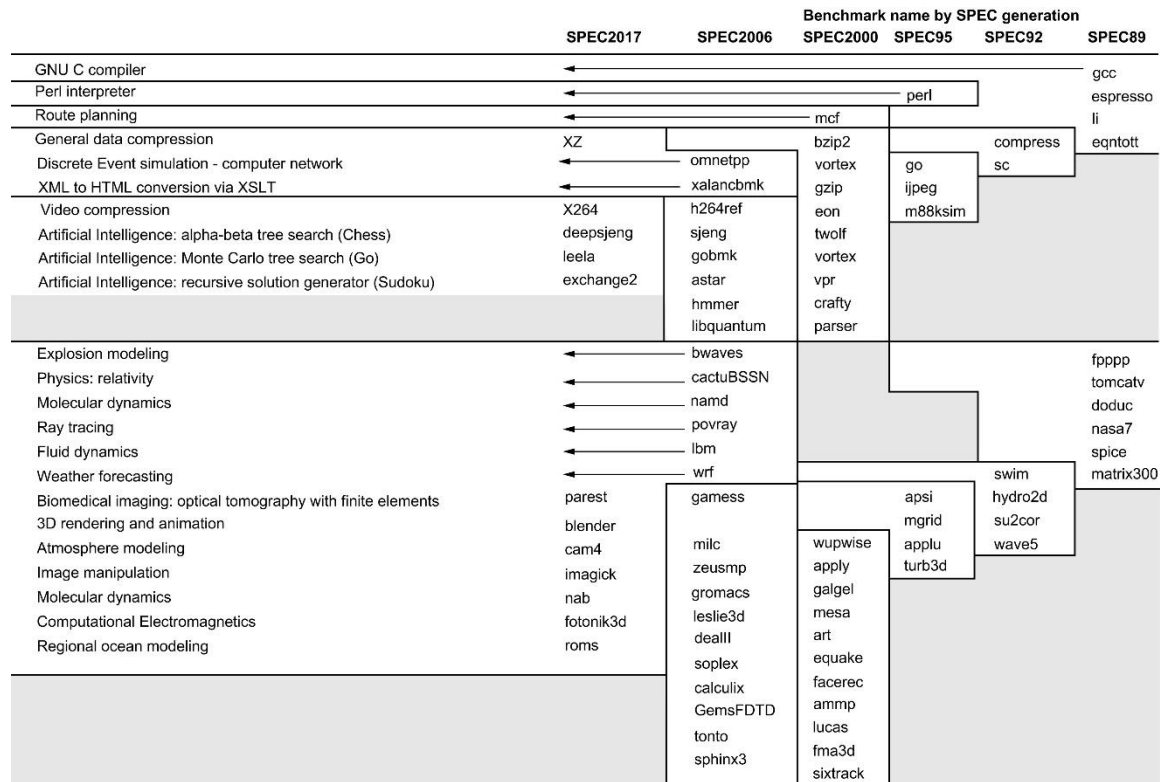


Figure 1.17 SPEC2017 programs and the evolution of the SPEC benchmarks over time, with integer programs above the line and floating-point programs below the line. Of the 10 SPEC2017 integer programs, 5 are written in C, 4 in C++, and 1 in Fortran. For the floating-point programs, the split is 3 in Fortran, 2 in C++, 2 in C, and 6 in mixed C, C++, and Fortran. The figure shows all 82 of the programs in the 1989, 1992, 1995, 2000, 2006, and 2017 releases. Gcc is the senior citizen of the group. Only 3 integer programs and 3 floating-point programs survived three or more generations. Although a few are carried over from generation to generation, the version of the program changes and either the input or the size of the benchmark is often expanded to increase its running time and to avoid perturbation in measurement or domination of the execution time by some factor other than CPU time. The benchmark descriptions on the left are for SPEC2017 only and do not apply to earlier versions. Programs in the same row from different generations of SPEC are generally not related; for example, fpppp is not a CFD code like bwaves.



Category	Name	Measures performance of
Cloud	Cloud_IaaS 2016	Cloud using NoSQL database transaction and K-Means clustering using map/reduce
CPU	CPU2017	Compute-intensive integer and floating-point workloads
Graphics and workstation performance	SPECviewperf [®] 12	3D graphics in systems running OpenGL and Direct X
	SPECwpc V2.0	Workstations running professional apps under the Windows OS
	SPECapcSM for 3ds Max 2015™	3D graphics running the proprietary Autodesk 3ds Max 2015 app
	SPECapcSM for Maya [®] 2012	3D graphics running the proprietary Autodesk 3ds Max 2012 app
	SPECapcSM for PTC Creo 3.0	3D graphics running the proprietary PTC Creo 3.0 app
	SPECapcSM for Siemens NX 9.0 and 10.0	3D graphics running the proprietary Siemens NX 9.0 or 10.0 app
High performance computing	SPECapcSM for SolidWorks 2015	3D graphics of systems running the proprietary SolidWorks 2015 CAD/CAM app
	ACCEL	Accelerator and host CPU running parallel applications using OpenCL and OpenACC
	MPI2007	MPI-parallel, floating-point, compute-intensive programs running on clusters and SMPs
Java client/server	OMP2012	Parallel apps running OpenMP
	SPECjbb2015	Java servers
Power	SPECpower_ssj2008	Power of volume server class computers running SPECjbb2015
Solution File Server (SFS)	SFS2014	File server throughput and response time
	SPECsfs2008	File servers utilizing the NFSv3 and CIFS protocols
Virtualization	SPECvirt_sc2013	Datacenter servers used in virtualized server consolidation

Figure 1.18 Active benchmarks from SPEC as of 2017.



性能的综合评价

- **算术平均或加权的算术平均**

- $SUM(T_i)/n$ 或 $SUM(W_i \times T_i)/n$

- **规格化执行时间，采用几何平均**

$$\sqrt[n]{\prod_{i=1}^n Execution_time_ratio_i}$$

- SPEC采用这种方法(SPECRatio)



SPEC 性能综合

$$\text{SPEC Ratio} = \frac{\text{Time on Reference Computer}}{\text{Time on Computer Being Rated}}$$

$$\frac{\text{SPEC Ratio}_A}{\text{SPEC Ratio}_B} = \frac{\frac{\text{ExecutionTime}_{\text{Ref}}}{\text{ExecutionTime}_A}}{\frac{\text{ExecutionTime}_{\text{Ref}}}{\text{ExecutionTime}_B}} = \frac{\text{ExecutionTime}_B}{\text{ExecutionTime}_A} = \frac{\text{Performance}_A}{\text{Performance}_B}$$

$$\text{Geometric Mean of SPEC Ratios} = \sqrt[n]{\prod_{i=1}^n \text{SPEC Ratio}_i}$$



SPECfp2000 Execution Times & SPEC Ratios

Benchmark	Ultra 5 Time (sec)	Opteron Time (sec)	SpecRatio Opteron	Itanium2 Time (sec)	SpecRatio Itanium2	Opteron/ Itanium2 Times	Itanium2/ Opteron SpecRatios
wupwise	1600	51.5	31.06	56.1	28.53	0.92	0.92
swim	3100	125.0	24.73	70.7	43.85	1.77	1.77
mgrid	1800	98.0	18.37	65.8	27.36	1.49	1.49
applu	2100	94.0	22.34	50.9	41.25	1.85	1.85
mesa	1400	64.6	21.69	108.0	12.99	0.60	0.60
galgel	2900	86.4	33.57	40.0	72.47	2.16	2.16
art	2600	92.4	28.13	21.0	123.67	4.40	4.40
equake	1300	72.6	17.92	36.3	35.78	2.00	2.00
facerec	1900	73.6	25.80	86.9	21.86	0.85	0.85
ampp	2200	136.0	16.14	132.0	16.63	1.03	1.03
lucas	2000	88.8	22.52	107.0	18.76	0.83	0.83
fma3d	2100	120.0	17.48	131.0	16.09	0.92	0.92
sixtrack	1100	123.0	8.95	68.8	15.99	1.79	1.79
apsi	2600	150.0	17.36	231.0	11.27	0.65	0.65
Geometric Mean			20.86		27.12	1.30	1.30

Geometric mean of ratios = 1.30 = Ratio of Geometric means = 27.12 / 20.86



几何平均的两个重要特性

$$\begin{aligned} \frac{\text{Geometric mean}_A}{\text{Geometric mean}_B} &= \frac{\sqrt[n]{\prod_{i=1}^n \text{SPECRatio } A_i}}{\sqrt[n]{\prod_{i=1}^n \text{SPECRatio } B_i}} = \sqrt[n]{\prod_{i=1}^n \frac{\text{SPECRatio } A_i}{\text{SPECRatio } B_i}} \\ &= \sqrt[n]{\prod_{i=1}^n \frac{\frac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{A_i}}}{\frac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{B_i}}}} = \sqrt[n]{\prod_{i=1}^n \frac{\text{Execution time}_{B_i}}{\text{Execution time}_{A_i}}} = \sqrt[n]{\prod_{i=1}^n \frac{\text{Performance}_{A_i}}{\text{Performance}_{B_i}}} \end{aligned}$$

- 几何平均的比率等于比率的几何平均
- 几何平均的比率 等于 性能比率的几何平均
 - 与参考机器的选择无关



为什么对规格化数采用几何平均?

	Computer A	Computer B	Computer C
Program P1 (secs)	1	10	20
Program P2 (secs)	1000	100	20
Total time (secs)	1001	110	40

	Normalized to A			Normalized to B			Normalized to C		
	A	B	C	A	B	C	A	B	C
Program P1	1.0	10.0	20.0	0.1	1.0	2.0	0.05	0.5	1.0
Program P2	1.0	0.1	0.02	10.0	1.0	0.2	50.0	5.0	1.0
Arithmetic mean	1.0	5.05	10.01	5.05	1.0	1.1	25.03	2.75	1.0
Geometric mean	1.0	1.0	0.63	1.0	1.0	0.63	1.58	1.58	1.0
Total time	1.0	0.11	0.04	9.1	1.0	0.36	25.03	2.75	1.0



小结：定量分析基础

- **性能度量**
 - 响应时间 (response time)
 - 吞吐率 (Throughput)
- **CPU 执行时间 = IC × CPI × T**
 - CPI (Cycles per Instruction)
- **MIPS = Millions of Instructions Per Second**
- **Latency versus Bandwidth**
 - Latency指单个任务的执行时间, Bandwidth 指单位时间完成的任务量 (rate)
 - Latency 的提升滞后于带宽的提升 (在过去的30年)
- **Amdahl' s Law 用来度量加速比 (speedup)**
 - 性能提升受限于任务中可加速部分所占的比例
- **Benchmarks: 指一组用于测试的程序**
 - 比较计算机系统的性能
 - SPEC benchmark: 针对一组应用综合性能值采用SPEC ratios 的几何平均



本章小结

- 设计发展趋势

	<u>Capacity</u>	<u>Speed</u>
Logic	2x in 3 years	2x in 3 years
DRAM	4x in 3 years	2x in 10 years
Disk	4x in 3 years	2x in 10 years

- 运行任务的时间

- Execution time, response time, latency

- 单位时间内完成的任务数

- Throughput, bandwidth

- “X性能是Y的n倍” :

$$\frac{\text{ExTime}(Y)}{\text{ExTime}(X)} = \frac{\text{Performance}(X)}{\text{Performance}(Y)}$$



本章小结(续)

- **Amdahl's 定律:**

$$\text{Speedup}_{\text{overall}} = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

- **CPI Law:**

CPU time	=	$\frac{\text{Seconds}}{\text{Program}}$	=	$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$
----------	---	---	---	---

- **执行时间是计算机系统度量的最实际，最可靠的方式**



Acknowledgements

- **These slides contain material developed and copyright by:**
 - John Kubiawicz (UCB)
 - Krste Asanovic (UCB)
 - John Hennessy (Stanford) and David Patterson (UCB)
 - Chenxi Zhang (Tongji)
 - Muhamed Mudawar (KFUPM)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**