计算机组织与系统结构

成本 性能 功耗

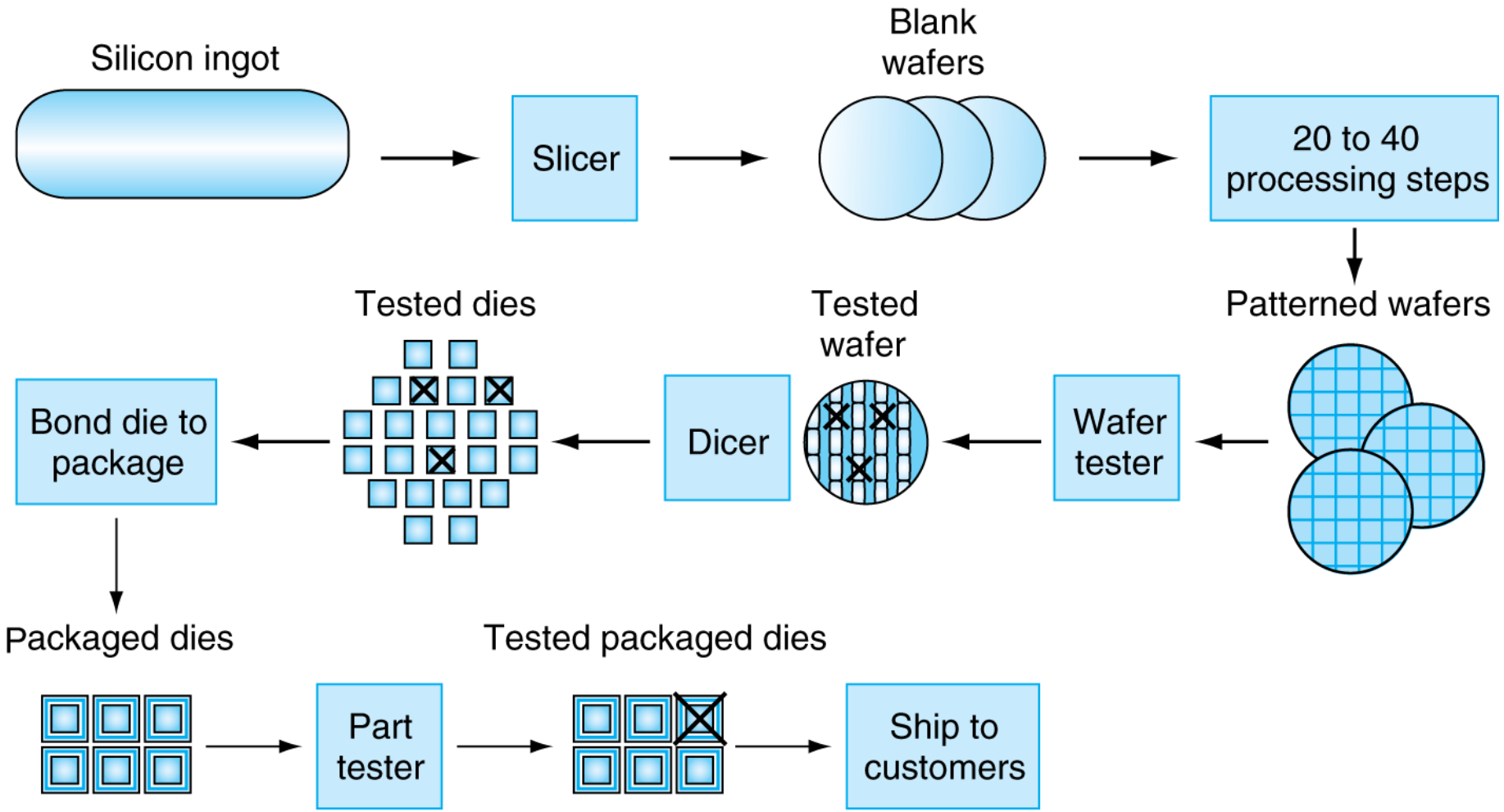
Cost，Performance and Power

（第四讲）

程 旭

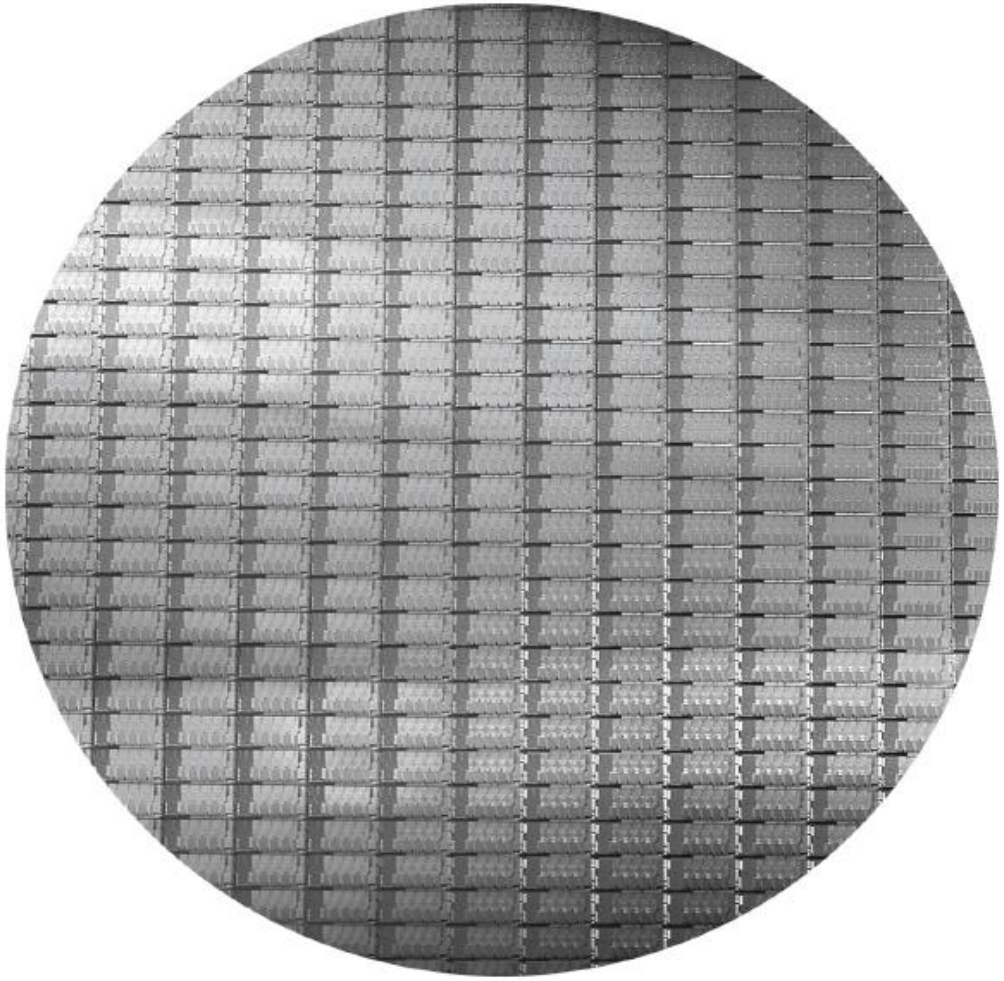
2020.11.12

芯片制作流程



Yield: proportion of working dies per wafer

Intel Core i7 Wafer



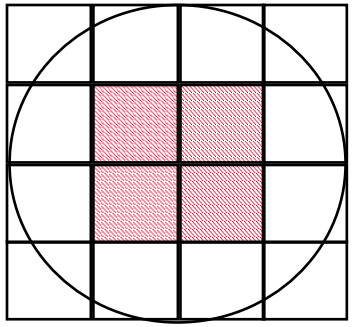
300mm wafer, 280 chips, 32nm technology

Each chip is 20.7 x 10.5 mm

集成电路的成本

$$Die\_cost = \frac{Wafer\_cost}{Dies\_per\_wafer \times Die\_yield}$$

$$Dies\_per\_wafer \approx \frac{Wafer\_area}{Die\_area} = \frac{\pi \times (Wafer\_diameter/2)^2}{Die\_area}$$



考虑Wafer的边界问题(等价于“Square pegs in a round hole” probelm)后:

$$Dies\_per\_wafer = \frac{\pi \times (Wafer\_diameter/2)^2}{Die\_area} - \frac{\pi \times Wafer\_diameter}{\sqrt{2 \times Die\_area}}$$

Integrated Circuit Cost

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}$$

$$\text{Dies per wafer} \approx \text{Wafer area}/\text{Die area}$$

$$\text{Yield} = \frac{1}{(1+(\text{Defects per area} \times \text{Die area}/2))^2}$$

- **Nonlinear relation to area and defect rate**
  - **Wafer cost and area are fixed**
  - **Defect rate determined by manufacturing process**
  - **Die area determined by architecture and circuit design**

其他成本

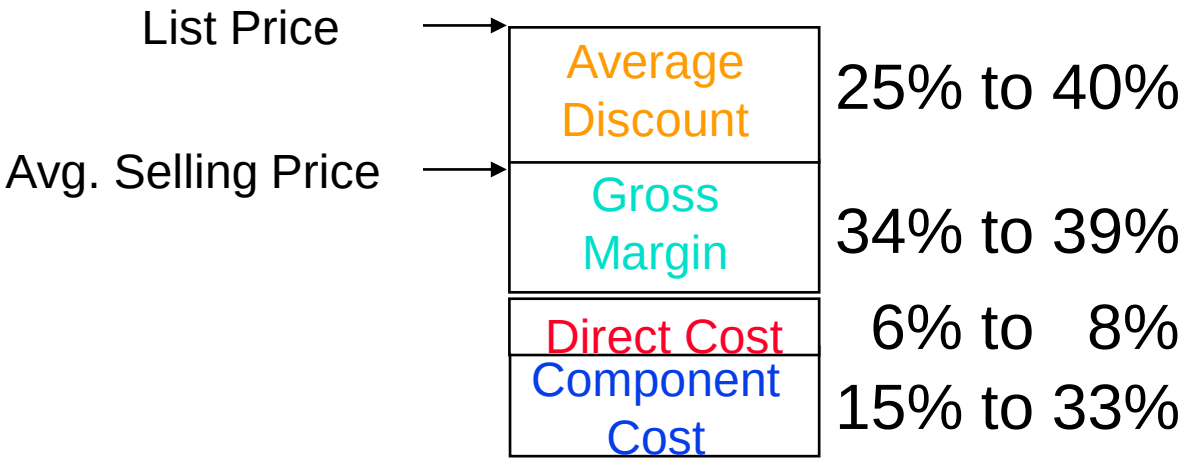
$$\text{IC cost} = \frac{\text{Die cost} + \text{Testing cost} + \text{Packaging cost}}{\text{Final test yield}}$$

封装成本：主要取决于管脚数量和散热要求

Cost/Performance

What is Relationship of Cost to Price?

- **Component Costs**
- **Direct Costs** **(add 25% to 40%) recurring costs: labor, purchasing, scrap, warranty**
- **Gross Margin** **(add 82% to 186%) nonrecurring costs: R&D, marketing, sales, equipment maintenance, rental, financing cost, pretax profits, taxes**
- **Average Discount** **to get List Price (add 33% to 66%): volume discounts and/or retailer markup**



iPad（2010）: Apple's profit comes from margins in hardware

$499

Margin:

40%

$110 **+ Apple margin**

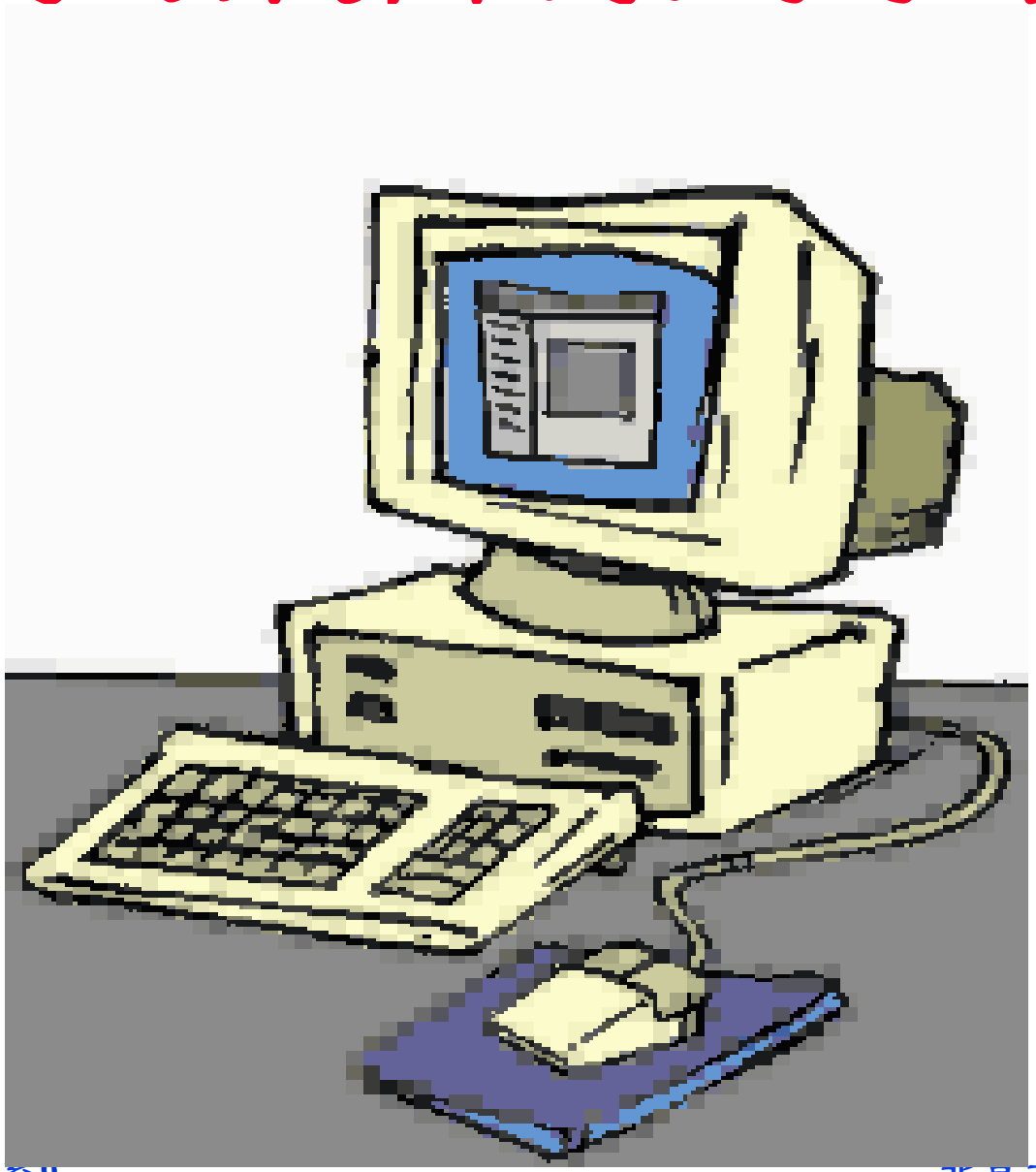
$90 Average industry margin (approx. 30 %)

$70 Cost of sales (approx. 30 %)

$230 Cost of materials and manufacturing1

Source: iSuppli

如何定义计算机系统的 性能?



如何定义计算机系统的 性能?

The goal of performance evaluation in this chapter is to be able to compare, for example,

- **different architectures**
- **different implementations of an architecture**
- **different compilers for a given architecture**

General sense:

how well the computer performs

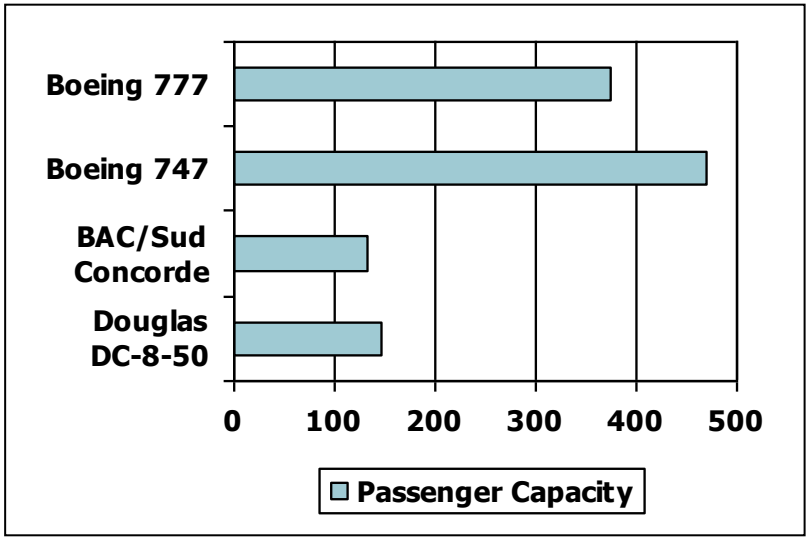
为什么需要了解影响机器性能的要素?

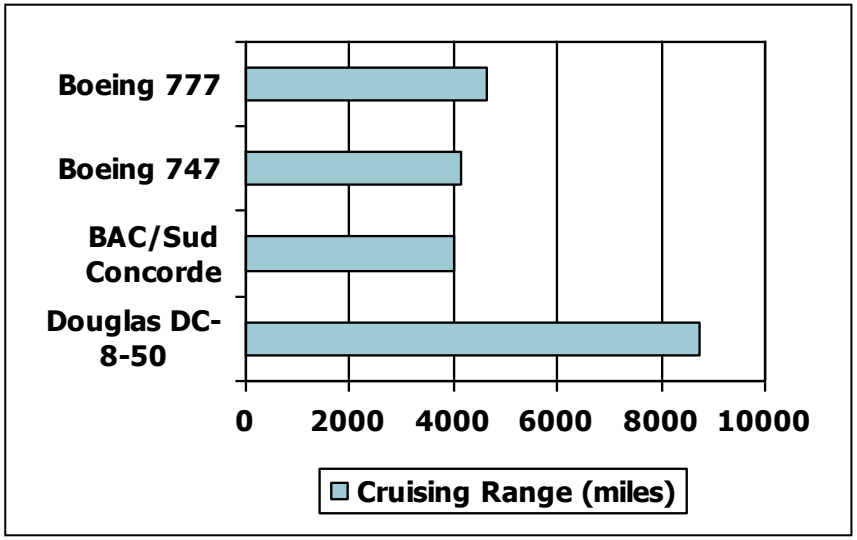
为了了解

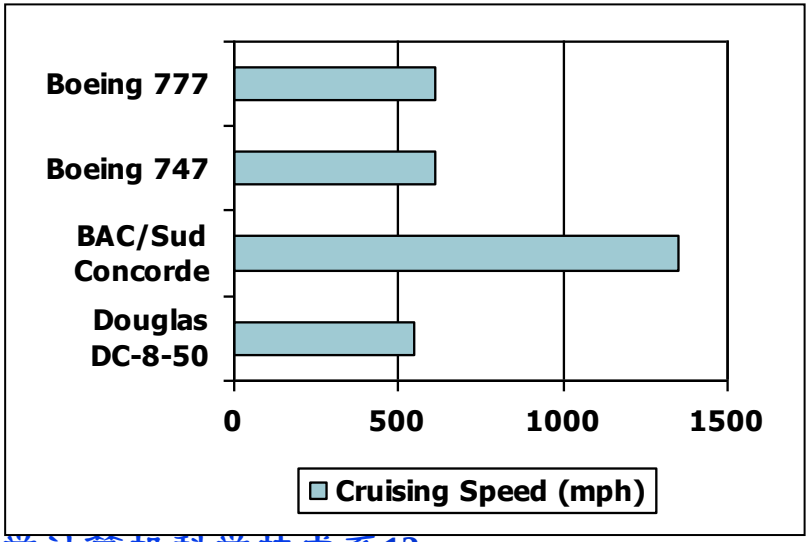
- 程序的执行情况
- 指令系统的不同实现情况
- 一些硬件特征对性能的影响

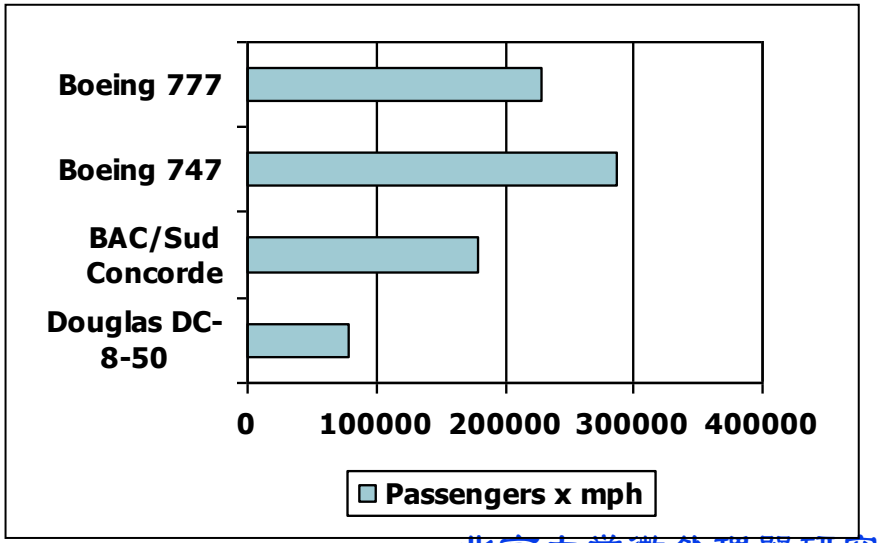
Defining Performance

- Which airplane has the best performance?









性能(和成本)
Performance (and cost)

飞机	华盛顿 - 巴黎	速度	乘客数量	吞吐率 (pmph)
Boeing 747	6.5 小时	610 mph	470	286,700
BAD/Sud Concodre	3 小时	1350 mph	132	178,200

- 完成任务的时间 **(执行时间)**
  - 执行时间**(execution time)**, 响应时间**(response time)**, 延迟**(latency)**
- 单位时间(每天、小时、星期、秒、纳秒…等等)内完成的任务数量(性能)
  - 吞吐率**(throughput)**, 带宽**(bandwidth)**

吞吐率与响应时间

计算机系统发生如下变化，对吞吐率和相应时间分别有如何影响？

- 更换成更快的处理器
- 增加处理器数量，对不同任务分别使用不同的处理器

减少响应时间总是可以改进吞吐率

对于每个任务（不可在并行处理），增加处理器从理论上看，对响应时间没有改进。

实际系统中，通常，任务需要排队等待响应处理，因而，执行时间和吞吐率常常相互影响。

Relative Performance

- **Define Performance = 1/Execution Time**
- **"X is *n* time faster than Y"**

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Execution time}_Y / \text{Execution time}_X = n$$

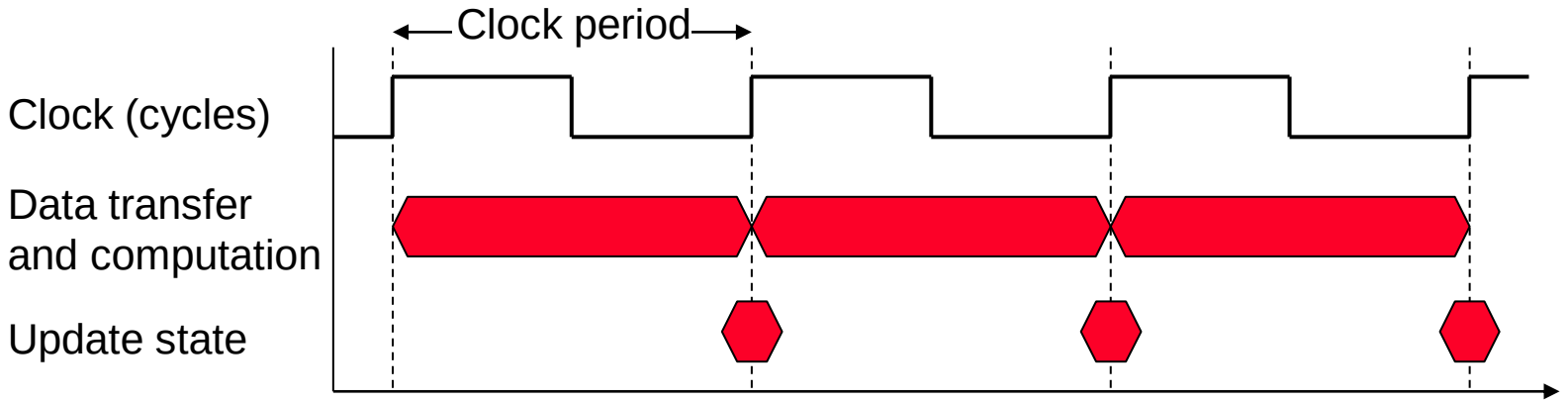
- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time_B / Execution Time_A = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

Measuring Execution Time

- **Elapsed time**
  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - Determines system performance
- **CPU time**
  - Time spent processing a given job
    - Discounts I/O time, other jobs' shares
  - Comprises user CPU time and system CPU time
  - Different programs are affected differently by CPU and system performance

CPU Clocking

- **Operation of digital hardware governed by a constant-rate clock**



- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns = 250×10^{-12}s
- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- **Performance improved by**
  - **Reducing number of clock cycles**
  - **Increasing clock rate**
  - **Hardware designer must often trade off clock rate against cycle count**

CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- **Instruction Count for a program**
  - **Determined by program, ISA and compiler**
- **Average cycles per instruction**
  - **Determined by CPU hardware**
  - **If different instructions have different CPI**
    - **Average CPI affected by instruction mix**

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$
$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$
$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

…by this much

CPI in More Detail

- **If different instruction classes take different numbers of cycles**

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

  - Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

示例

A Compiler designer is trying to decide between two code sequences for a particular machine

示例

The hardware designers supplies:

Instruction class	CPI for this instruction set
A	1
B	2
C	3

For a particular High-Level-Language statement:

Code Sequence	Instruction counts for instruction class		
	A	B	C
1	2	1	2
2	4	1	1

示例

示例

Sequence 1 executes: 2+1+2= 5

Sequence 2 executes: 4+1+1= 6

$$CPU\_clock\_cycles = \sum_{j=1}^{n} (CPI_j \times I_j)$$

$$CPU\_clock\_cycles_1 = (2\times 1)+(1\times 2)+(2\times 3)$$
$$= 2+2+6 = 10 cycles$$

$$CPU\_clock\_cycles_2 = (4\times 1)+(1\times 2)+(1\times 3)$$
$$= 4+2+3 = 9 cycles$$

$$CPI_1 = \frac{10}{5} = 2 \qquad CPI_2 = \frac{9}{6} = 1.5$$

CPU性能

$$CPU时间 = \frac{秒数}{程序} = \frac{指令数}{程序} \times \frac{时钟数}{指令} \times \frac{秒数}{周期}$$

	指令总数	CPI	时钟周期
算法	X	(X)	
编程语言	X	X	
编译	X	X	
指令系统	X	X	(X)
组成		X	X
实现			X

Performance expressed as a rate

- Rates are performance measures expressed in units of work per unit time.

- Examples:
  - millions of instructions / sec (MIPS)
  - millions of floating point instructions / sec (MFLOPS)
  - millions of bytes / sec (MBytes/sec)
  - millions of bits / sec (Mbits/sec)
  - images / sec
  - samples / sec
  - transactions / sec (TPS)

行销计量(Marketing Metrics)

$$MIPS = \frac{\text{指令总数}}{\text{时间}} \times 10^{-6}$$

$$= \frac{\text{时钟频率}}{CPI} \times 10^{-6}$$

- 具有不同指令系统的 机器？
- 具有不同指令频度的 程序？
  - 指令的动态频度
- 与性能没有直接相关关系

$$MFLOPS = \frac{\text{浮点操作总数总数}}{\text{时间}} \times 10^{-6}$$

- 与机器有关
- 通常，没能揭示 最费时 的 操作

MIPS

$$Native\_MIPS = \frac{\text{指令总数}}{\text{时间}} \times 10^{-6}$$

Instruction class	CPI for this instruction set
A	1
B	2
C	3

Code From	Instruction counts (in billion) for each instruction class		
	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

Clock Rate=500 MHz

MIPS

$$CPU\_ClockCycles_1 = (5\times1+1\times2+1\times3)\times10^9 = 10\times10^9$$

$$CPU\_ClockCycles_2 = (10\times1+1\times2+1\times3)\times10^9 = 15\times10^9$$

$$ExTime_1 = \frac{10\times10^9}{500\times10^6} = 20s$$

$$ExTime_2 = \frac{15\times10^9}{500\times10^6} = 30s$$

⇒ Performance1 > Performacne2

$$MIPS_1 = \frac{(5+1+1)\times10^9}{20\times10^6} = 350$$

⇒ Performance2 > Performacne1

$$MIPS_2 = \frac{(10+1+1)\times10^9}{30\times10^6} = 400$$

MIPS

Peak MIPS?

Native MIPS?

$$Relative\_MIPS = \frac{Time_{reference}}{Time_{unrated}} \times MIPS_{reference}$$

MIPS为什么不好?

Meaningless Indicator of Performance of System

Marketing's Invention to Promote Sales

- **All instructions are not equally powerful**
- **Depends on compiler**
- **Depends on instruction mix and sequencing**

Performance expressed as a rate(cont)

- Key idea: Report rates that track execution time.
- Example: Suppose we are measuring a program that convolves a stream of images from a video camera.
- Bad performance measure: MFLOPS
  - number of floating point operations depends on the particular convolution algorithm: n^2 matix-vector product vs nlogn fast Fourier transform. An FFT with a bad MFLOPS rate may run faster than a matrix-vector product with a good MFLOPS rate.
- Good performance measure: images/sec
  - a program that runs faster will convolve more images per second.

Performance expressed as a rate(cont)

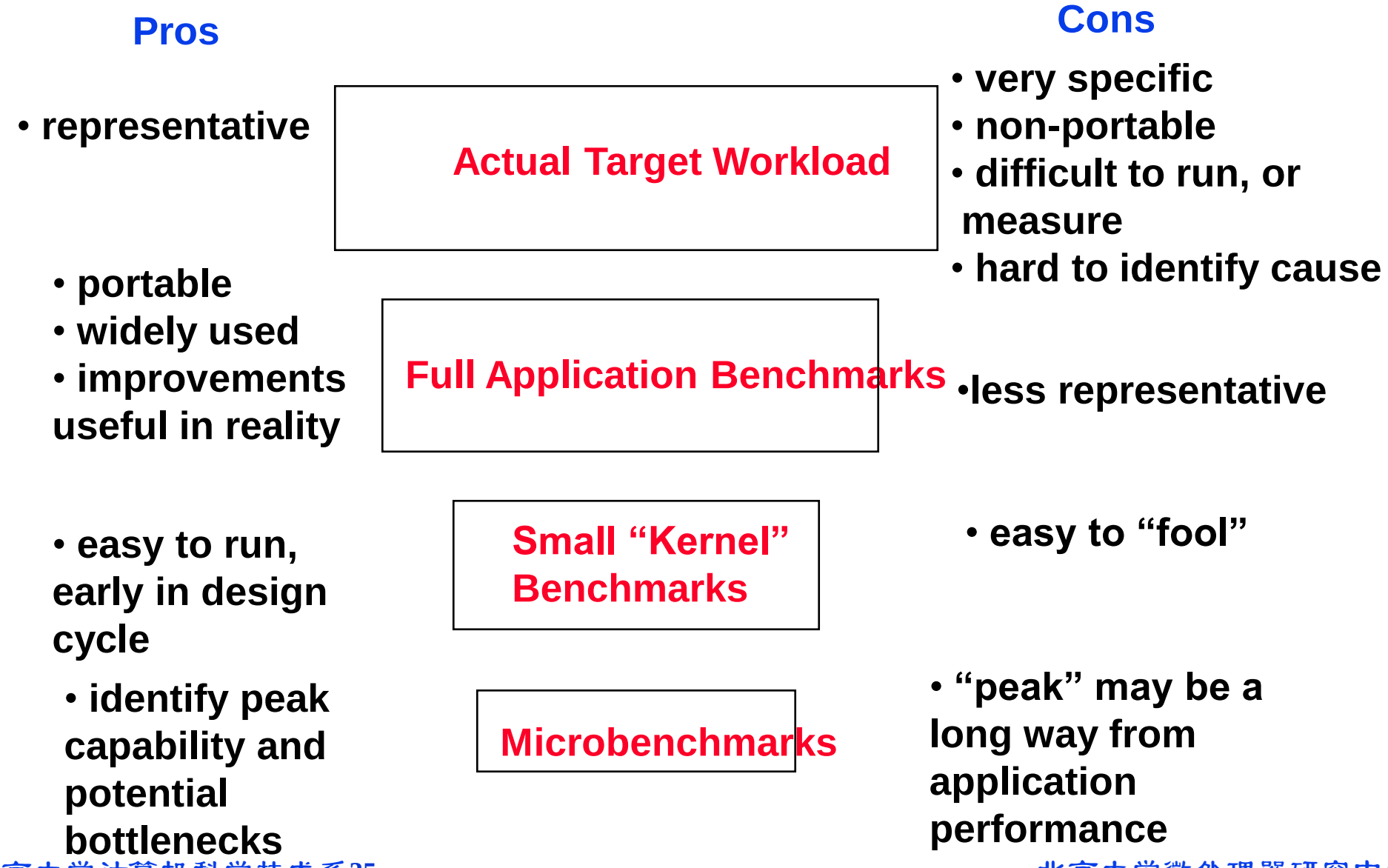
- Fallacy: Peak rates track running time.
- Example: the i860 is advertised as having a peak rate of 80 MFLOPS (40 MHz with 2 flops per cycle).
- However, the measured performance of some compiled linear algebra kernels (icc -O2) tells a different story:

Kernel	1d fft	sasum	saxpy	sdot	sgemm	sgemv	spvma
MFLOPS	8.5	3.2	6.1	10.3	6.2	15.0	8.1
%peak	11%	4%	7%	13%	8%	19%	10%

基准程序(Benchmarks)

- 如何评价差异
  - 不同系统
  - 与单一系统相比
- 提供一个目标
  - 基准程序应该能够代表一大类重要程序
  - 改进基准程序的性能应该对大多数程序有益
- **For better or worse, benchmarks shape a field**
- 好基准程序可以加速计算机的发展进程
  - 好的发展目标
- 坏基准程序不利于计算机的发展
  - 是有益于运行真实程序, 还是销售机器/发表论文?
  - 创造 真正 有益于 真实程序 的 基准程序
    而不是 有益于 基准程序 的 基准程序

Basis of Evaluation



评价处理器性能的程序

- (玩具Toy) 基准程序
  - 10-100 行
  - 例如: sieve, puzzle, quicksort
- 合成(Synthetic)基准程序
  - 试图匹配真实工作负载的平均频度
  - 例如, Whetstone, Dhrystone
- 内核(Kernels)程序
  - 时间密集(Time critical excerpts)
  - 例如, Livermore loops
- 真实程序
  - 例如, gcc, spice

成功的基准程序： SPEC

- 1987年，RISC工业陷入营销困境
  (只有8 MIPS性能的机器, 却被声称具有10 MIPS性能!)
- 1988年, EE Times + 5 家公司(Sun, MIPS, HP, Apollo, DEC) 联合起来 完成

Systems Performance Evaluation Cooperative (SPEC)

Systems Performance Evaluation Committee (SPEC)

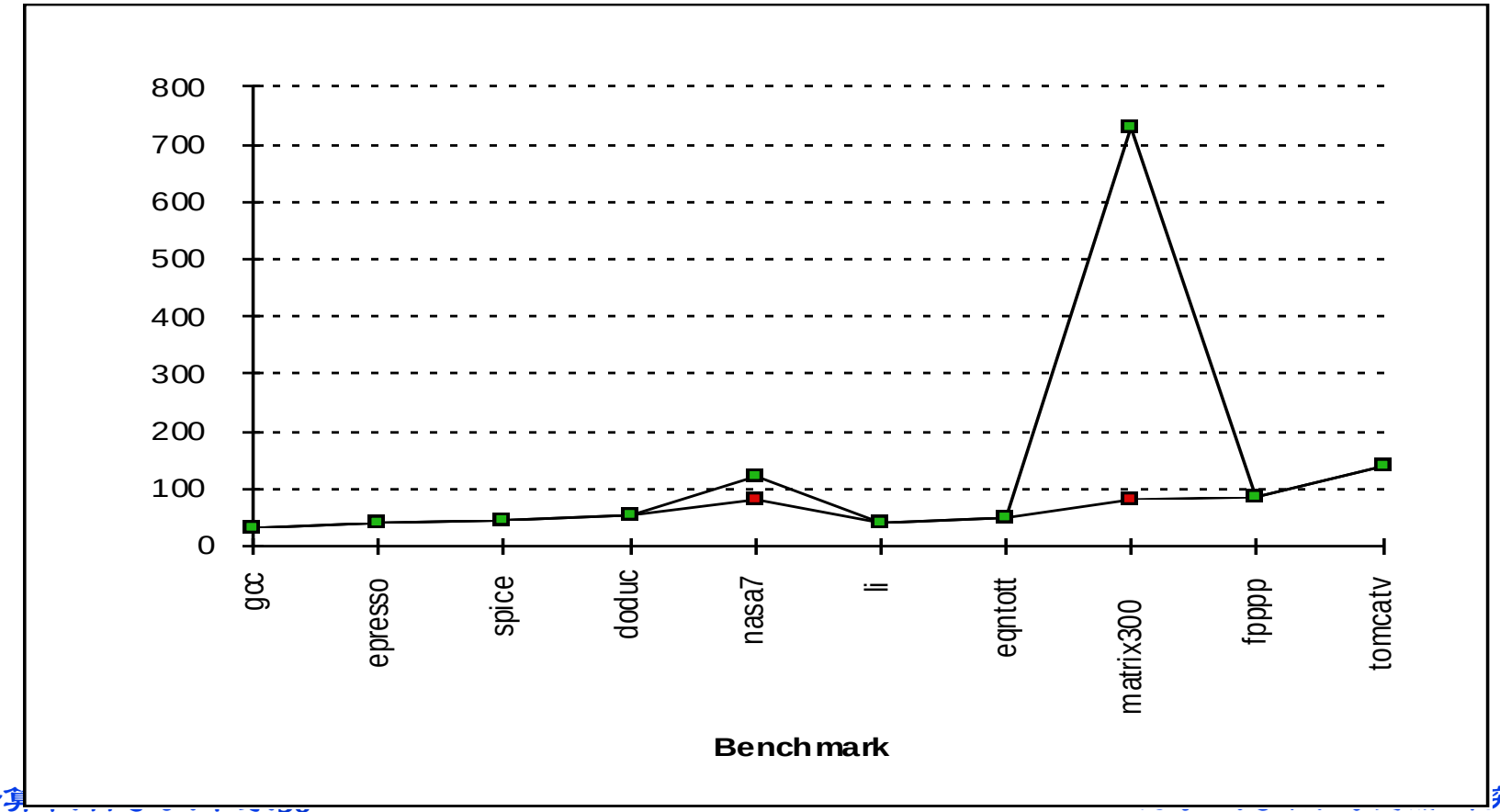
- 寻找出 一组标准的程序、输入、报告

(standard list of programs, inputs, reporting）:

一些真实程序（包括操作系统调用和一些I/O）

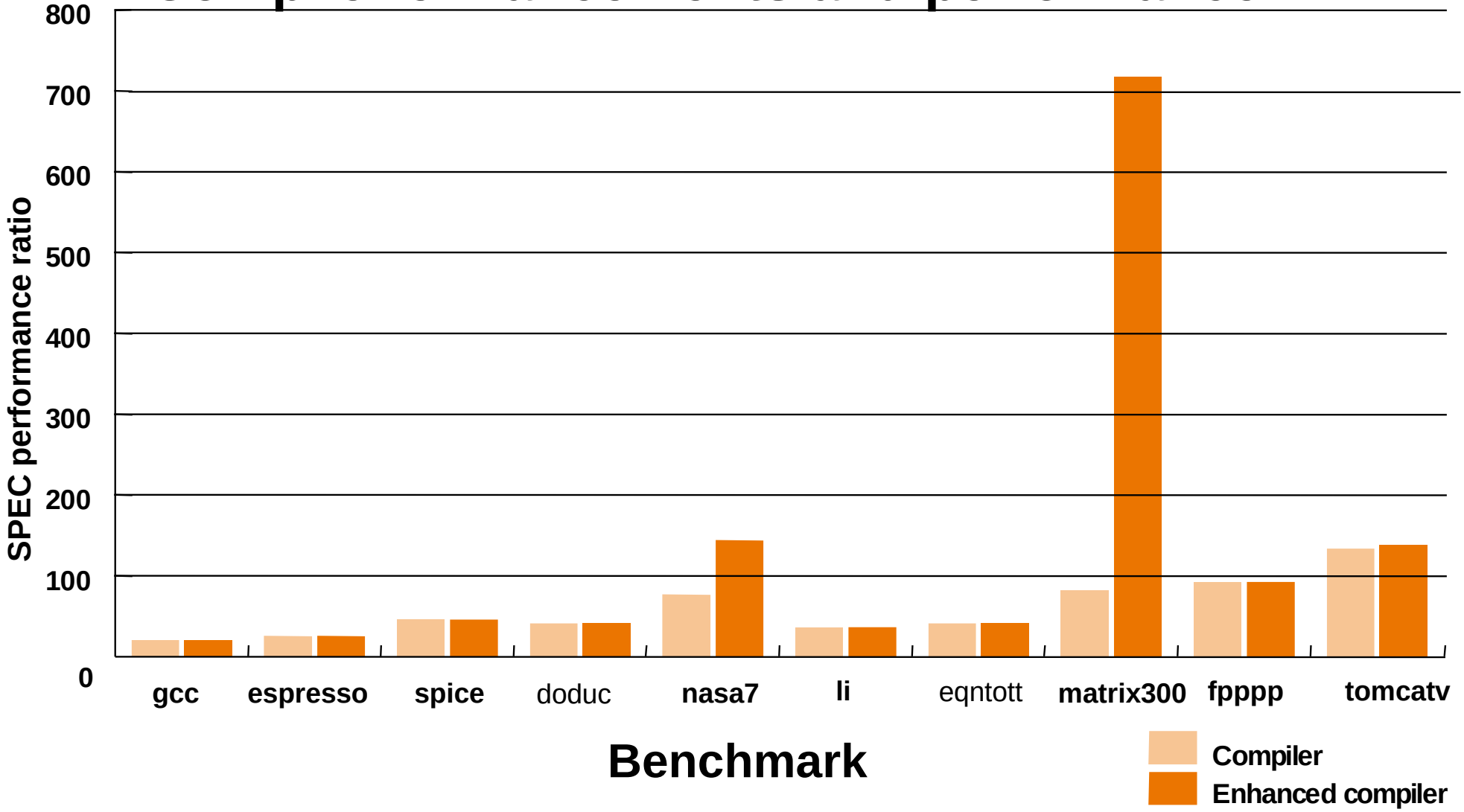
SPEC的发展早期情况

- 1989年，第一版; 10 个程序, 用单一数值来总结性能 (6Fp+4Int), 相对于VAX 11/780
- 其中有一个程序: 99%的时间耗费在该程序的单一一行代码上
- 新型前端编译器可以非常显著地改进它的性能



SPEC 89

Compiler enhancements and performance



SPEC 发展情况

- 第二版; SpecInt92 (6 整数程序) 和 SpecFP92 (14 个浮点程序)

  不限制编译器的标志（Flags）时, DEC 4000 Model 610:

  spice: `unix.c:/def=(sysv,has_bcopy,copy(a,b,c)=memcpy(b,a,c)`

  wave5: `/ali=(all,dcom=nat)/ag=a/ur=4/ur=200`

  nasa7: `/norecu/ag=a/ur=4/ur2=200/lc=blas`

- 新增 SPECbase: 对整数程序用一种标志设置 &对浮点程序有另一种标志设置

- 第三版; 1995; 一组新程序SpecInt95 (8 整数程序) 和 SpecFP92 (10 个浮点程序)
  - 相对于Sun SPARCstation 10/40的执行时间进行规格化
  - 基准程序的有效期： 3-5年(Spec2000)

SPEC95

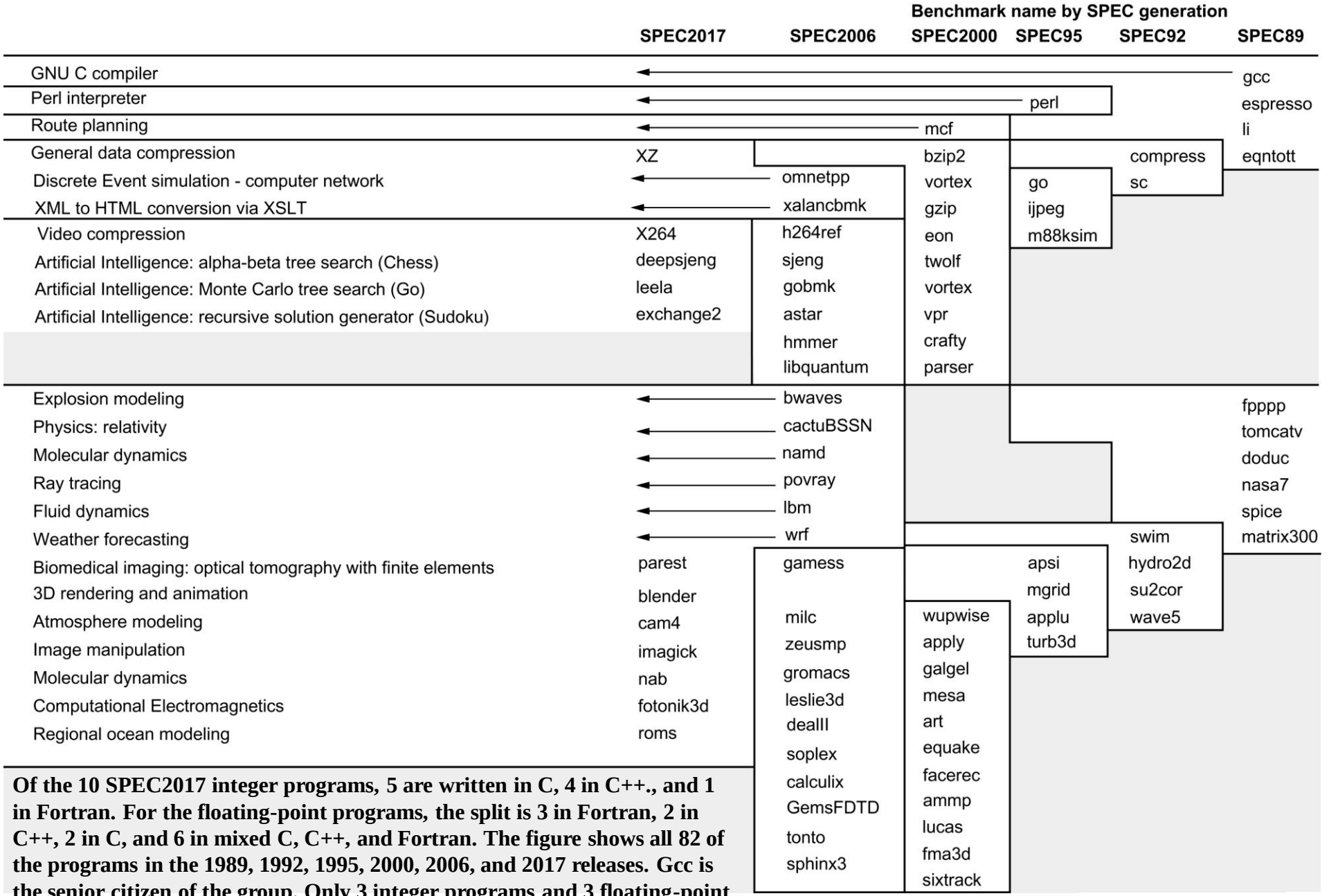
- Eighteen application benchmarks (with inputs) reflecting a technical computing workload
- Eight integer
  - go, m88ksim, gcc, compress, li, ijpeg, perl, vortex
- Ten floating-point intensive
  - tomcatv, swim, su2cor, hydro2d, mgrid, applu, turb3d, apsi, fppp, wave5
- Must run with standard compiler flags
  - eliminate special undocumented incantations that may not even generate working code for real programs

SPEC 95

Benchmark	Description
go	Artificial intelligence; plays the game of Go
m88ksim	Motorola 88k chip simulator; runs test program
gcc	The Gnu C compiler generating SPARC code
compress	Compresses and decompresses file in memory
li	Lisp interpreter
ijpeg	Graphic compression and decompression
perl	Manipulates strings and prime numbers in the special-purpose programming language Perl
vortex	A database program
tomcatv	A mesh generation program
swim	Shallow water model with 513 x 513 grid
su2cor	quantum physics; Monte Carlo simulation
hydro2d	Astrophysics; Hydrodynamic Naiver Stokes equations
mgrid	Multigrid solver in 3-D potential field
applu	Parabolic/elliptic partial differential equations
trub3d	Simulates isotropic, homogeneous turbulence in a cube
apsi	Solves problems regarding temperature, wind velocity, and distribution of pollutant
fpppp	Quantum chemistry
wave5	Plasma physics; electromagnetic particle simulation

SPEC2Kcpu

Benchmark	Type	Source	Description
gzip	Integer	C	Compression using the Lempel-Ziv algorithm
vpr	Integer	C	FPGA circuit placement and routing
gcc	Integer	C	Consists of the GNU C compiler generating optimized machine code
mcf	Integer	C	Combinatorial optimization of public transit scheduling
crafty	Integer	C	Chess-playing program
parser	Integer	C	Syntactic English language parser
eon	Integer	C++	Graphics visualization using probabilistic ray tracing
perlmbk	Integer	C	Perl (an interpreted string-processing language) with four input scripts
gap	Integer	C	A group theory application package
vortex	Integer	C	An object-oriented database system
bzip2	Integer	C	A block-sorting compression algorithm
twolf	Integer	C	Timberwolf: a simulated annealing algorithm for VLSI place and route
wupwise	FP	F77	Lattice gauge theory model of quantum chromodynamics
swim	FP	F77	Solves shallow water equations using finite difference equations
mgrid	FP	F77	Multigrid solver over three-dimensional field
apply	FP	F77	Parabolic and elliptic partial differential equation solver
mesa	FP	C	Three-dimensional graphics library
galgel	FP	F90	Computational fluid dynamics
art	FP	C	Image recognition of a thermal image using neural networks
equake	FP	C	Simulation of seismic wave propagation
facerec	FP	C	Face recognition using wavelets and graph matching
ammp	FP	C	Molecular dynamics simulation of a protein in water
lucas	FP	F90	Performs primality testing for Mersenne primes
fma3d	FP	F90	Finite element modeling of crash simulation
sixtrack	FP	F77	High-energy physics accelerator design simulation
apsi	FP	F77	A meteorological simulation of pollution distribution



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.

EEMBC

Benchmark type	Number of kernels	Example benchmarks
Automotive/industrial	16	6 microbenchmarks (arithmetic operations, pointer chasing, memory performance, matrix arithmetic, table lookup, bit manipulation), 5 automobile control benchmarks, and 5 filter or FFT benchmarks
Consumer	5	5 multimedia benchmarks (JPEG compress/decompress, filtering, and RGB Conversions)
Networking	3	Shortest-path calculation, IP routing, and packet flow operations
Office automation	4	Graphics and text benchmarks (Bézier curve calculation, dithering, image rotation, text processing)
Telecommunications	6	Filtering and DSP benchmarks (autocorrelation, FFT, decoder, encoder)

正确地总结性能

- 算术平均值 (或者加权算术平均值) 追踪 执行时间: **SUM(Ti)/n** 或者 **SUM(Wi*Ti)**
- 比率（例如**MFLOPS**） 的 调和平均值 (或者加权调和平均值) 追踪 执行时间:

  n/SUM(1/Ri) 或者 **n/SUM(Wi/Ri)**
- 为了按比例伸缩性能，规格化执行时间 是 非常便捷的！

  例如， 参照机器的时间 ÷ 被评测机器的时间
- 注意，不可使用规格化的执行时间的算术平均值，而应该使用几何平均值！

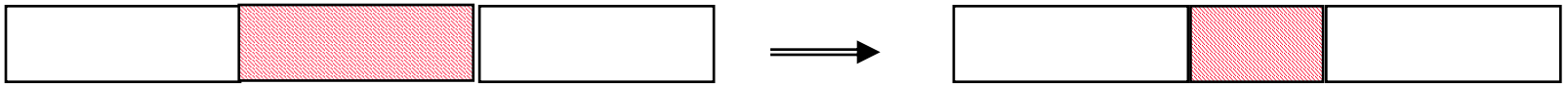
为什么对规格化数值要进行几何平均值?

	Time on A	Time on B	Normalized to A		Normalized to B	
			A	B	A	B
Program 1	1	10	1	10	0.1	1
Program 2	10	1	1	0.1	10	1
Arithmetic mean	55	55	1	5.05	5.05	1
Geometric mean	31.6	31.6	1	1	1	1

Amdahl定律（Amdahl's Law）

通过 增加 E，获得的 加速比（Speedup）：

$$加速比(E)=\frac{没有E时的执行时间}{增加E后的执行时间}=\frac{增加E后的性能}{没有E时的性能}$$



假设，增加 E 可以加速整个任务的 F 部分，加速因子为 S；并且这个任务的其他部分不受影响.

那么

$$执行时间(增加E)=\left((1-F)+\frac{F}{S}\right)\times 执行时间(没有E)$$

$$加速比(增加E)=\frac{执行时间(没有E)}{\left((1-F)+\frac{F}{S}\right)\times 执行时间(没有E)}$$

Amdahl's Law

Example:

"Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?"

How about making it 5 times faster?

Principle: Make the common case fast

解答

$$执行时间(增加E) = \left((1-F)+\frac{F}{S}\right)\times 执行时间(没有E)$$

$$\frac{100}{4} = \left(\left(1-\frac{80}{100}\right)+\frac{80/100}{S_1}\right)\times 100$$

$$S_1 = \frac{0.8}{\frac{1}{4}-0.2} = 16$$

$$S_2 = \frac{0.8}{\frac{1}{5}-0.2} = ?$$

解答

性能评价概要

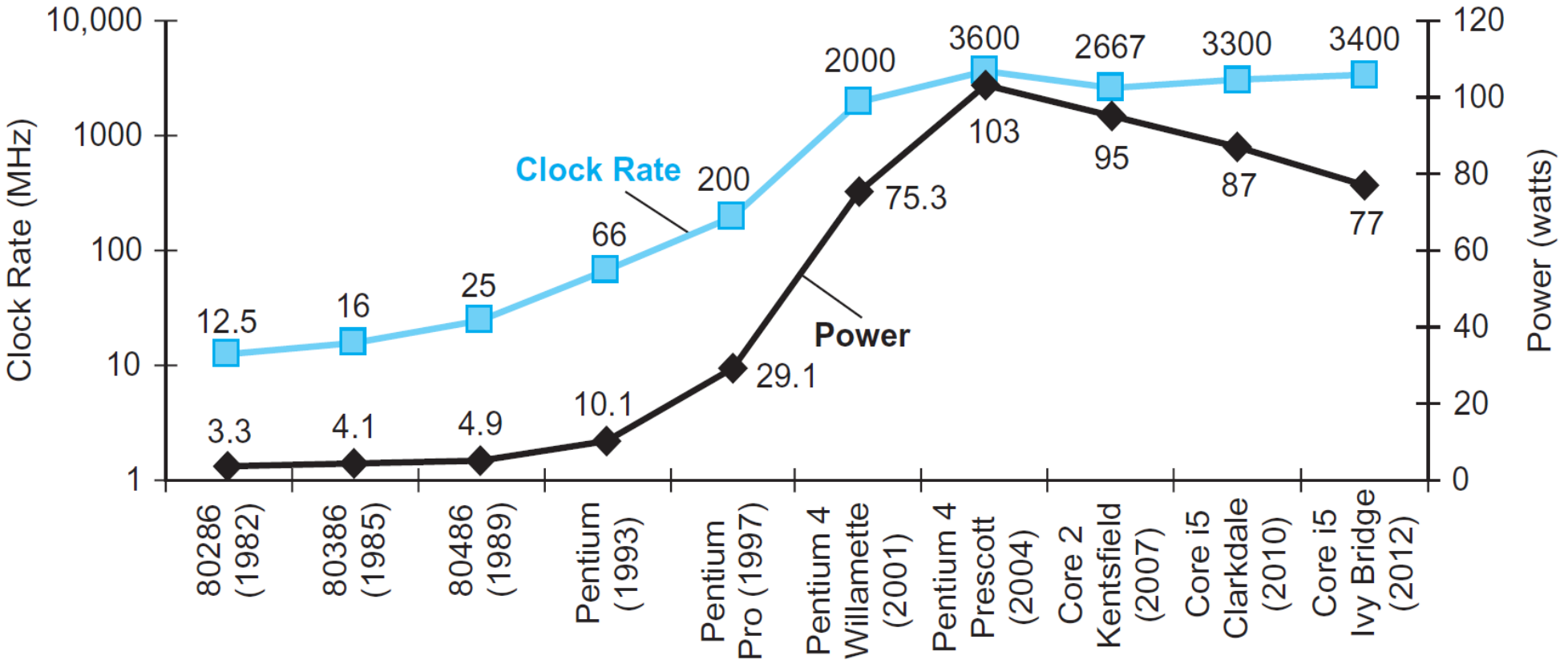
$$CPU时间 = \frac{秒数}{程序} = \frac{指令数}{程序} \times \frac{时钟数}{指令} \times \frac{秒数}{周期}$$

- 时间 是 计算机性能 的 唯一可靠的 测度!
- 当具有以下条件时，才能创造出好的产品：
  - 好的基准程序
  - 好的方式来总结性能
- 如果没有好基准程序和总结方法, 那么在

  为真实程序的性能 而 改进产品 与

  为增加销售额 而 改进产品 之间进行选择时

  => 后者几乎总是赢家!
- 记住Amdahl定律: 加速比 受到 程序中没有改进的部分 制约



Clock rate and power for Intel x86 microprocessors over eight generations and 30 years.

CMOS的能耗和功耗

CMOS动态能耗：

Logical transition of $0 \longrightarrow 1 \longrightarrow 0$ *or* $1 \longrightarrow 0 \longrightarrow 1$:

$$\text{Energy} \propto Capacityive\ Load \times Voltage^2$$

Single transition:

$$\text{Energy} \propto \frac{1}{2} Capacityive\ Load \times Voltage^2$$

The power required per transition:

$$Power \propto \frac{1}{2} Capacityive\ Load \times Voltage^2 \times \text{Frequency switched}$$

Reducing Power

- **Suppose a new CPU has**
  - **85% of capacitive load of old CPU**
  - **15% voltage and 15% frequency reduction**

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^{\ 2} \times F_{old}} = 0.85^4 = 0.52$$

- The power wall
  - We can't reduce voltage further
  - We can't remove more heat
- How else can we improve performance?

SPEC Power Benchmark

- **Power consumption of server at different workload levels**
  - **Performance: ssj_ops/sec**
  - **Power: Watts (Joules/sec)**

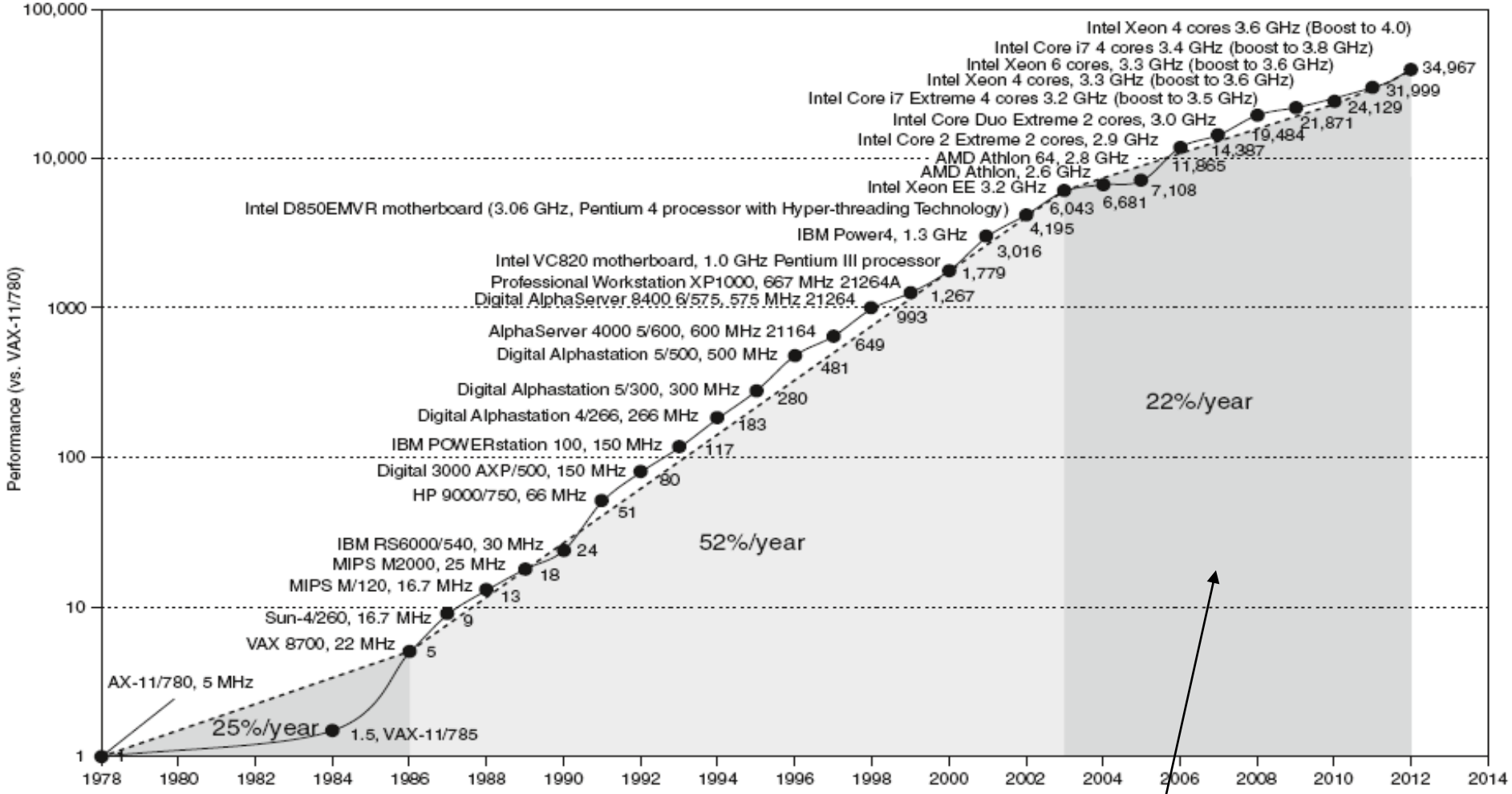
$$\text{Overall ssj\_ops per Watt} = \left(\sum_{i=0}^{10} \text{ssj\_ops}_i\right) \Big/ \left(\sum_{i=0}^{10} \text{power}_i\right)$$

SPECpower2008是SPECpower_ssj2008的简称，其以overall ssj_ops/watt为计算单位，即平均ssj每秒性能/每瓦.
ssj: server side java business application
Java 服务器是目前最常用的服务器端，因此该基准默认采用BEA的Java虚拟机。

SPECpower_ssj2008 for Xeon X5650

Target Load %	Performance (ssj_ops)	Average Power (Watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1,922
Σssj_ops/Σpower =		2,490

Uniprocessor Performance



Constrained by power, instruction-level parallelism, memory latency

Multiprocessors

- Multicore microprocessors
  - More than one processor per chip
- Requires explicitly parallel programming
  - Compare with instruction level parallelism
    - Hardware executes multiple instructions at once
    - Hidden from the programmer
  - Hard to do
    - Programming for performance
    - Load balancing
    - Optimizing communication and synchronization

How to Mislead with Performance Reports

- Select pieces of workload that work well on your design, ignore others
- Use unrealistic data set sizes for application (too big or too small)
- Report throughput numbers for a latency benchmark
- Report latency numbers for a throughput benchmark
- Report performance on a kernel and claim it represents an entire application
- Use 16-bit fixed-point arithmetic (because it's fastest on your system) even though application requires 64-bit floating-point arithmetic
- Use a less efficient algorithm on the competing machine
- Report speedup for an inefficient algorithm (bubblesort)
- Compare hand-optimized assembly code with unoptimized C code
- Compare your design using next year's technology against competitor's year old design (1% performance improvement per week)
- Ignore the relative cost of the systems being compared
- Report averages and not individual results
- Report speedup over unspecified base system, not absolute times
- Report efficiency not absolute times
- Report MFLOPS not absolute times (use inefficient algorithm)

[David Bailey "Twelve ways to fool the masses when giving performance results for parallel supercomputers"]