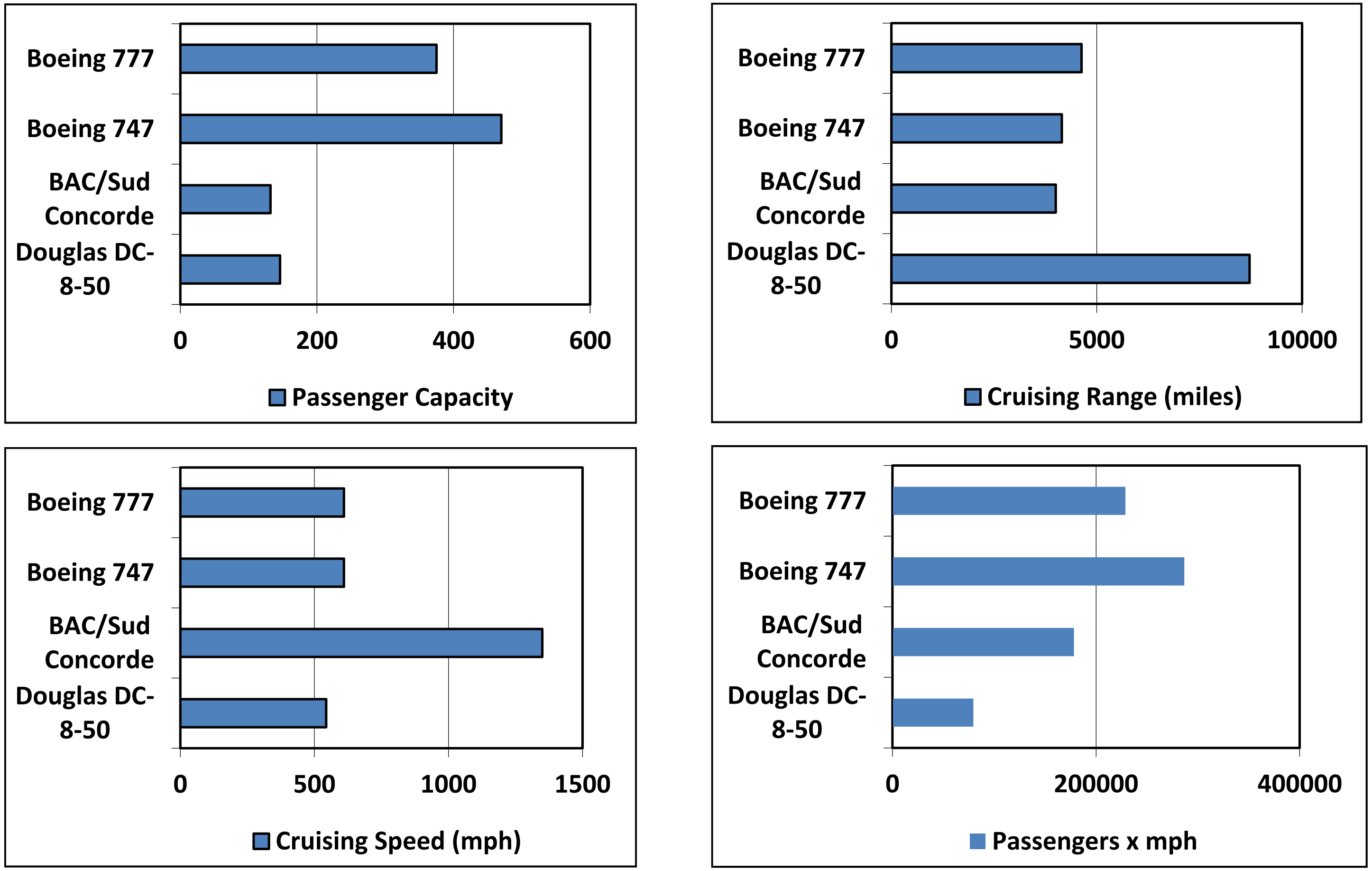
Performance

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Defining Performance (1)

- **Which airplane has the best performance?**



Defining Performance (2)

- **Performance issues**
  - Measure, analyze, report, and summarize
  - Make intelligent choices
  - See through the marketing hype
  - Key to understanding underlying organizational motivation
  - Questions
    - Why is some hardware better than others for different programs?
    - What factors of system performance are hardware related? (e.g., Do we need a new machine, or a new operating system?)
    - How does the machine's instruction set affect performance?

Computer Performance (1)

- **Response time (≈ execution time, latency)**
  - The time between the start and completion of a task
  - How long does it take for my job to run?
  - How long must I wait for the database query?
- **Throughput (≈ bandwidth)**
  - The total amount of work done in a given time
  - How much work is getting done per unit time?
  - What is the average execution rate?
- **What if …**
  - We replace the processor with a faster version?
  - We add more processors?

Computer Performance (2)

- **Relative performance**
  - Define

$$Performance = 1 / Execution\ \ Time$$

  - "X is *n* times faster than Y"

$$\frac{Performance_X}{Performance_Y} = \frac{Execution\ time_Y}{Execution\ time_X} = n$$

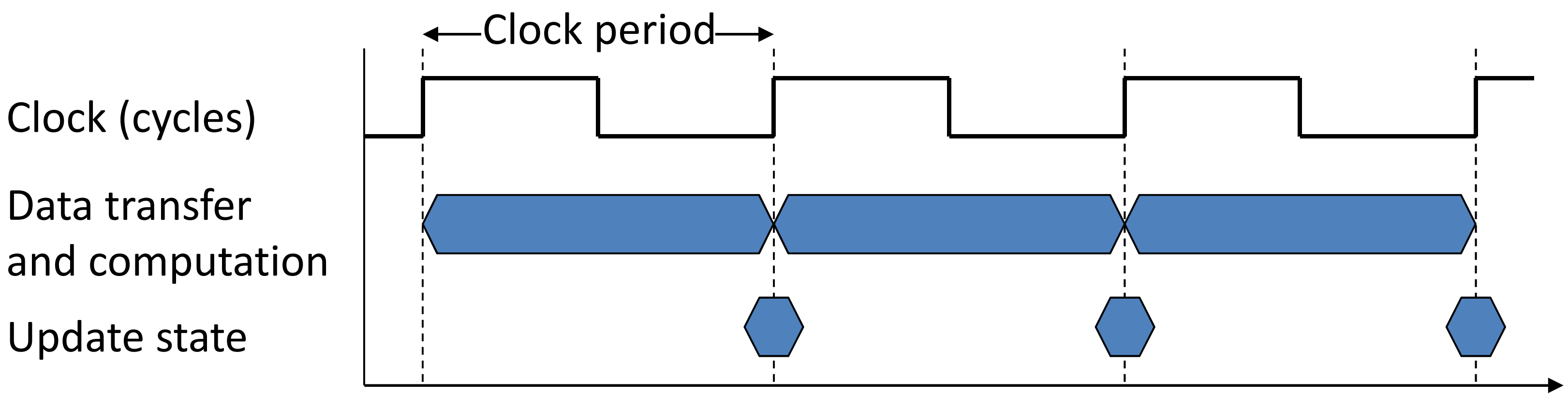
  - Example: time taken to run a program
    - 10s on machine A, 15s on machine B
    - Execution $Time_B$ / Execution $Time_A$ = 15s / 10s = 1.5
    - Machine A is 1.5 times faster than machine B

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
- **Our focus: User CPU time**

CPU Clocking

- **Clock**
  - Operation of digital hardware governed by a constant-rate clock
  - Clock "ticks" indicate when to start activities
  - Clock period: duration of a clock cycle
  - Clock frequency (rate): cycles per second



CPU Time (1)

$$CPU\ Time = CPU\ Clock\ Cycles \times Clock\ Cycle\ Time$$
$$= \frac{CPU\ Clock\ Cycles}{Clock\ Rate}$$

- **Performance improved by**
  - Reducing the number of clock cycles
  - Increasing clock rate
    (or decreasing the clock cycle time)
  - Hardware designer must often trade off clock rate against cycle count

CPU Time (2)

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

$$Clock\ Rate_B = \frac{Clock\ Cycles_B}{CPU\ Time_B} = \frac{1.2 \times Clock\ Cycles_A}{6s}$$

$$Clock\ Cycles_A = CPU\ Time_A \times Clock\ Rate_A$$

$$= 10s \times 2GHz = 20 \times 10^9$$

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

CPI (1)

- **Instruction count and CPI**

$$Clock\ Cycles = Instruction\ Count \times Cycles\ per\ Instruction$$

$$CPU\ Time = Instruction\ Count \times CPI \times Clock\ Cycle\ Time$$

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

- Instruction count for a program
  - Determined by program, ISA, and compiler
- Average cycles per instruction (CPI)
  - Determined by CPU hardware
  - If different instructions have different CPI
    → The average CPI affected by instruction mix

CPI (2)

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?

$$\begin{aligned} CPU\ Time_A &= Instruction\ Count \times CPI_A \times Cycle\ Time_A \\ &= I \times 2.0 \times 250ps = I \times 500ps \end{aligned}$$

$$\begin{aligned} CPU\ Time_B &= Instruction\ Count \times CPI_B \times Cycle\ Time_B \\ &= I \times 1.2 \times 500ps = I \times 600ps \end{aligned}$$

$$\frac{CPU\ Time_B}{CPU\ Time_A} = \frac{I \times 600ps}{I \times 500ps} = 1.2$$

CPI (3)

- **CPI in more detail**
  - If different instruction classes take different numbers of cycles:

$$Clock\ Cycles = \sum_{i=1}^{n} (CPI_i \times Instruction\ Count_i)$$

  - Weighted average CPI

$$CPI = \frac{Clock\ Cycles}{Instruction\ Count} = \sum_{i=1}^{n} \left(CPI_i \times \frac{Instruction\ Count_i}{Instruction\ Count} \right)$$

Relative frequency

CPI (4)

- **Example:**
  - Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- Sequence 1: IC = 5
  - Clock cycles
    = 2x1+1x2+2x3 = 10
  - Avg. CPI = 10/5 = 2.0
- Sequence 2: IC = 6
  - Clock cycles
    = 4x1+1x2+1x3 = 9
  - Avg. CPI = 9/6 = 1.5

MIPS

- **MIPS: Millions of Instructions Per Second**
  - MIPS as a performance metric?
  - Doesn't account for
    - Differences in ISAs between computers
    - Differences in complexity between instructions

$$MIPS = \frac{Instruction\ count}{Execution\ time \times 10^6} = \frac{Instruction\ count}{\frac{Instruction\ count \times CPI}{Clock\ rate} \times 10^6} = \frac{Clock\ rate}{CPI \times 10^6}$$

  - CPI varies between programs on a given CPU

C Sort Example (1)

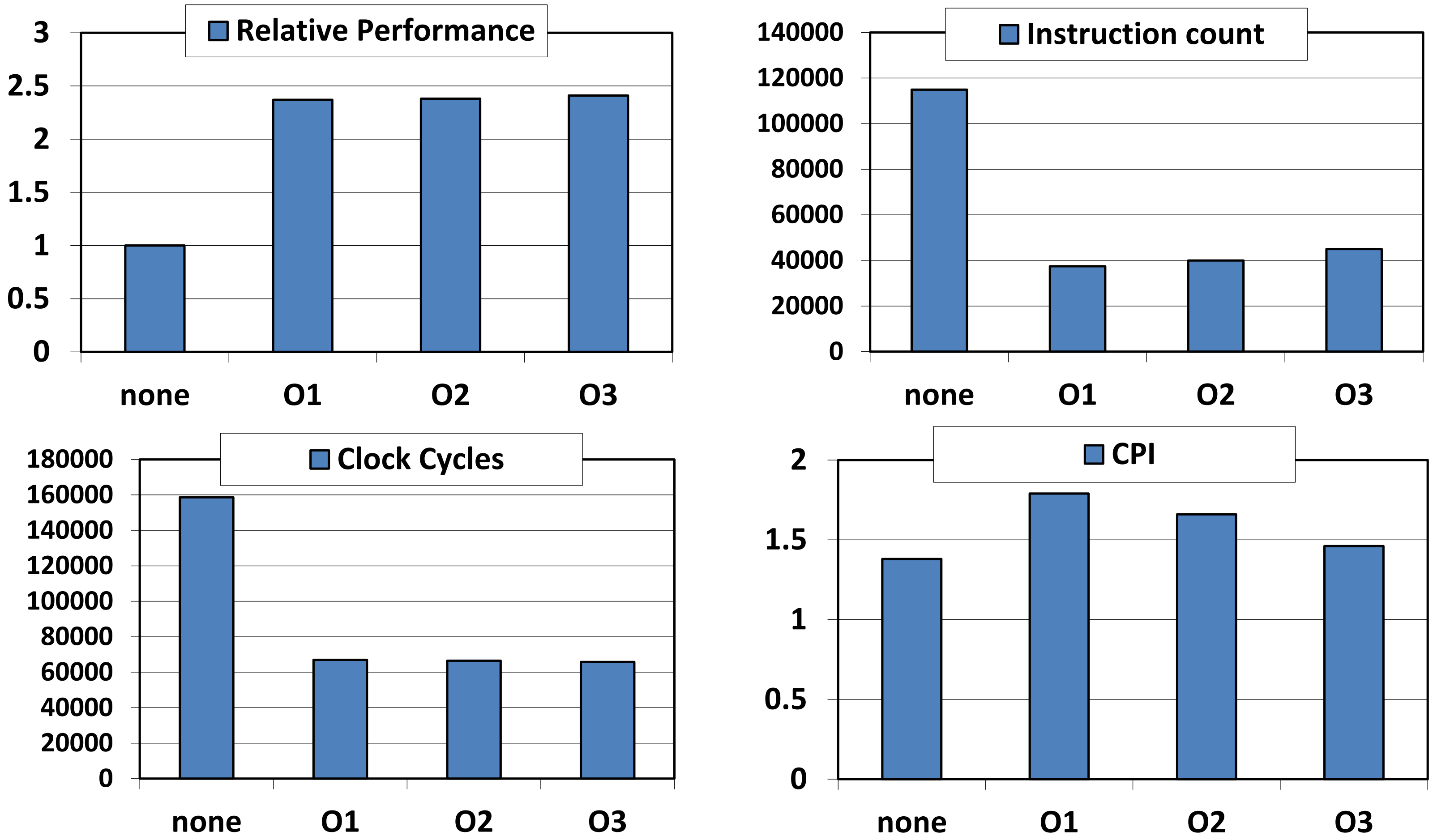
- **Bubble sort in C**

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
       for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j -= 1)
       {
          swap(v, j);
       }
    }
 }
```

C Sort Example (2)

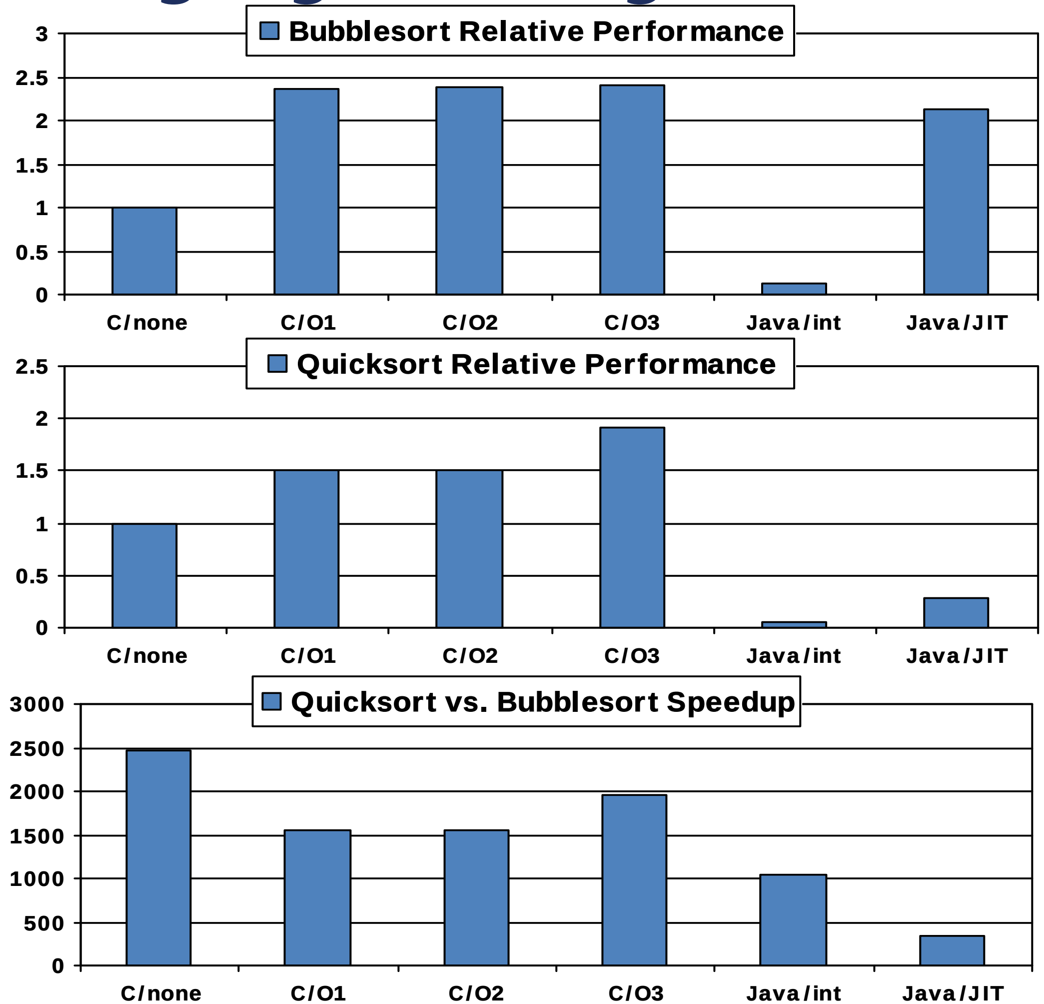
- **Effect of compiler optimization**



Compiled with gcc for Pentium 4 under Linux

C Sort Example (3)

- **Effect of language and algorithm**



C Sort Example (4)

- **Lessons**
  - Instruction count and CPI are not good performance indicators in isolation
  - Compiler optimizations are sensitive to the algorithm
  - Java/JIT compiled code is significantly faster than JVM interpreted
    - Comparable to optimized C in some cases
  - Nothing can fix a dumb algorithm!

Performance Summary

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

	Instruction Count	CPI	Clock Cycle
Algorithm	○	△	
Programming language	○	○	
Compiler	○	○	
ISA	○	○	○
Microarchitecture		○	○
Technology			○

Benchmarks

- **How to measure the performance?**
  - Performance best determined by running a real application
  - Use programs typical of expected workload
  - Or, typical of expected class of applications
- **Small benchmarks**
  - Nice for architects and designers
  - Easy to standardize
  - Can be abused

SPEC CPU Benchmark (1)

- **SPEC (Standard Performance Evaluation Corp.)**
  - Develops benchmarks for CPU, I/O, Web, …
  - http://www.spec.org

- **SPEC CPU benchmark**
  - An industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler.
    - Companies have agreed on a set of real program and inputs
    - Valuable indicator of performance (and compiler technology)
  - CPU89 → CPU92 → CPU95 → CPU2000 → CPU2006
  - Can still be abused

SPEC CPU Benchmark (2)

- **Benchmark games**

An embarrassed Intel Corp. acknowledged Friday that a bug in a software program known as a compiler had led the company to overstate the speed of its microprocessor chips on an industry benchmark by 10 percent. However, industry analysts said the coding error…was a sad commentary on a common industry practice of "cheating" on standardized performance tests…The error was pointed out to Intel two days ago by a competitor, Motorola …came in a test known as SPECint92…Intel acknowledged that it had "optimized" its compiler to improve its test scores. The company had also said that it did not like the practice but felt to compelled to make the optimizations because its competitors were doing the same thing…At the heart of Intel's problem is the practice of "tuning" compiler programs to recognize certain computing problems in the test and then substituting special handwritten pieces of code…

Saturday, January 6, 1996 New York Times

SPEC CPU Benchmark (3)

- **SPEC CPU2006**
  - Elapsed time to execute a selection of programs
    - Negligible I/O, so focuses on CPU performance
  - Normalize relative to reference machine
    - Sun's historical "Ultra Enterprise 2" introduced in 1997
    - 296MHz UltraSPARC II processor
  - Summarize as geometric mean of performance ratios
    - CINT2006: 12 integer programs written in C and C++
    - CFP2006: 17 FP programs written in Fortran and C/C++

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

SPEC CPU Benchmark (4)

- **SPEC CPU2006 (cont'd)**

Integer Benchmarks (CINT2006)			Floating Point Benchmarks (CFP2006)		
perlbench	C	Perl programming language	bwaves	Fortran	Fluid dynamics
bzip2	C	Compression	gamess	Fortran	Quantum chemistry
gcc	C	C compiler	milc	C	Physics: Quantum chromodynamics
mcf	C	Combinatorial optimization	zeusmp	Fortran	Physics / CFD
gobmk	C	Artificial intelligence: Go	gromacs	C/Fortran	Biochemistry / Molecular dynamics
hmmer	C	Search gene sequence	cactusADM	C/Fortran	Physics / General relativity
sjeng	C	Artificial intelligence: Chess	leslie3d	Fortran	Fluid dynamics
libquantum	C	Physics: Quantum computing	namd	C++	Biology / Molecular dynamics
h264ref	C	Video compression	dealII	C++	Finite element analysis
omnetpp	C++	Discrete event simulation	soplex	C++	Linear programming, optimization
astar	C++	Path-finding algorithms	povray	C++	Image ray-tracing
xalancbmk	C++	XML processing	calculix	C/Fortran	Structural mechanics
			GemsFDTD	Fortran	Computational electromagnetics
			tonto	Fortran	Quantum chemistry
			lbm	C	Fluid dynamics
			wrf	C/Fortran	Weather prediction
			sphinx3	C	Speech recognition

SPEC CPU Benchmark (5)

- **CINT2006 for Opteron X4 2356**

Name	Description	IC×10^9	CPI	Tc (ns)	Exec time	Ref time	SPECratio
perl	Interpreted string processing	2,118	0.75	0.40	637	9,770	15.3
bzip2	Block-sorting compression	2,389	0.85	0.40	817	9,650	11.8
gcc	GNU C Compiler	1,050	1.72	0.40	724	8,050	11.1
mcf	Combinatorial optimization	336	10.00	0.40	1,345	9,120	6.8
go	Go game (AI)	1,658	1.09	0.40	721	10,490	14.6
hmmer	Search gene sequence	2,783	0.80	0.40	890	9,330	10.5
sjeng	Chess game (AI)	2,176	0.96	0.40	837	12,100	14.5
libquantum	Quantum computer simulation	1,623	1.61	0.40	1,047	20,720	19.8
h264avc	Video compression	3,102	0.80	0.40	993	22,130	22.3
omnetpp	Discrete event simulation	587	2.94	0.40	690	6,250	9.1
astar	Games/path finding	1,082	1.79	0.40	773	7,020	9.1
xalancbmk	XML parsing	1,058	2.70	0.40	1,143	6,900	6.0
Geometric mean							**11.7**

SPEC Power Benchmark (1)

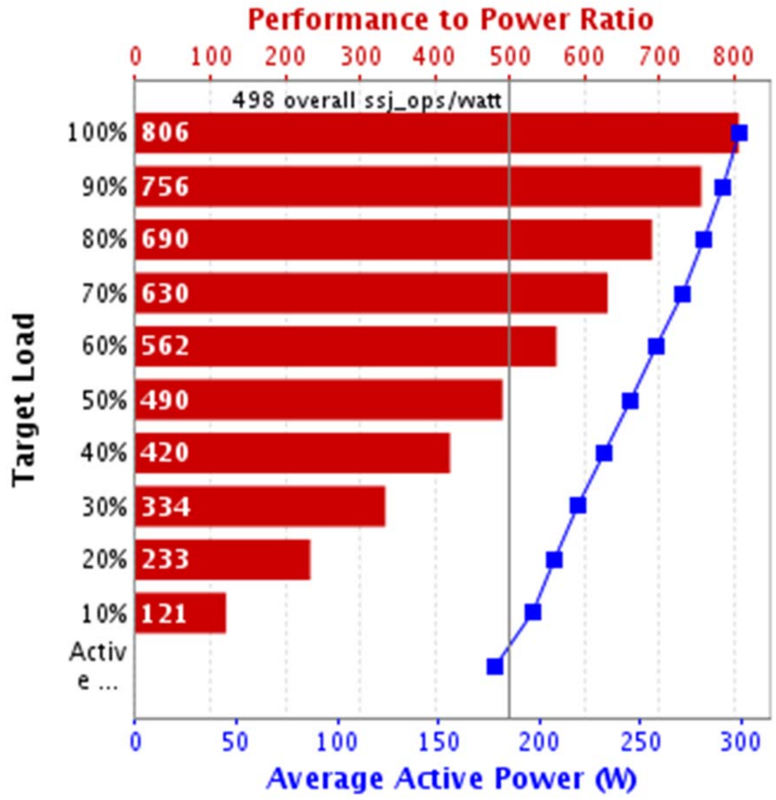
- **SPECpower_ssj2008**
  - The first industry-standard SPEC benchmark for evaluating the power and performance characteristics of server class computers
  - Initially targets the performance of server-side Java
  - Power consumption of server at different workload levels (0% ~ 100%)
    - Performance: ssj_ops/sec
    - Power: Watts (Joules/sec)

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

SPEC Power Benchmark (2)

- **SPECpower_ssj2008 for X4 2356**

Performance			Power	Performance to Power Ratio
Target Load	Actual Load	ssj_ops	Avg. Active Power (W)	
100%	99.3%	240,914	299	806
90%	90.7%	219,979	291	756
80%	80.1%	194,276	282	690
70%	70.5%	170,927	271	630
60%	59.9%	145,299	258	562
50%	49.5%	120,062	245	490
40%	40.2%	97,534	232	420
30%	30.2%	73,199	219	334
20%	19.9%	48,386	207	233
10%	9.8%	23,819	197	121
Active Idle		0	178	0
			∑ssj_ops / ∑power =	**498**



SPEC Power Benchmark (3)

- **Low power at idle?**
  - Look back at X4 power benchmark
    - At 100% load: 299W
    - At 50% load: 245W (82%)
    - At 10% load: 180W (60%)
- **Google data center**
  - Mostly operates at 10% – 50% load
  - At 100% load less than 1% of the time
- **Designing processors to make power proportional to load?**

Other Benchmarks

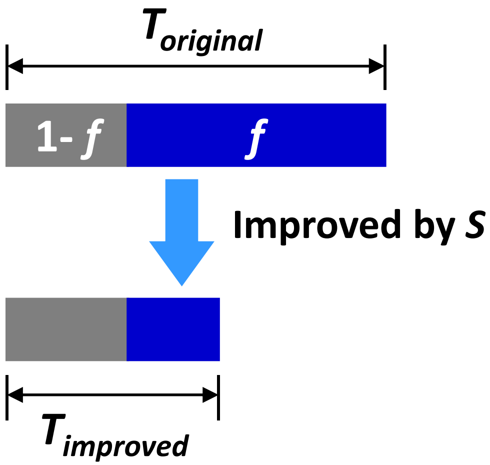
- **EEMBC**
  - Applications on embedded systems such as communication devices, automobiles, etc.
- **Mediabench**
  - Set of multimedia applications (codecs, graphics, …)
- **NAS**
  - Parallel benchmarks from NASA
- **SPLASH, PARSEC**
  - Multithreaded benchmarks for multiprocessors

Amdahl's Law (1)

- **Execution time after improvement**

$$T_{improved} = \frac{T_{affected}}{Improvement\ \ factor} + T_{unaffected}$$

$$= T_{original} \times ((1 - f) + f / S)$$



- **Example: multiply accounts for 80s/100s**
  - How much improvement in multiply performance to run a program 4 times faster?
  - How about making it 5 times faster?

Amdahl's Law (2)

- **Speedup and Amdahl's law**

$$Speedup = \frac{T_{original}}{T_{improved}} = \frac{1}{((1 - f) + f / S)}$$

- **Principles**
  - Make the common case fast
    - As f → 1, speedup → S
  - Speedup is limited by the fraction of code that can be optimized
    - As S → ∞, speedup → $1 / (1 - f)$
  - Uncommon case can become the common one after improvement

Summary

- **Performance is specific to a particular program(s)**
  - Total execution time is a consistent summary of the performance
- **For a given architecture, performance increases come from**
  - Increases in clock rate (without adverse CPI affects)
  - Improvements in processor organization that lower CPI
  - Compiler enhancements that lower CPI and/or instruction count
  - Algorithm/Language choices that affect instruction count
- **Pitfall:**
  - Expecting improvement in one aspect of a machine's performance to affect the total performance