CS202: COMPUTER ORGANIZATION

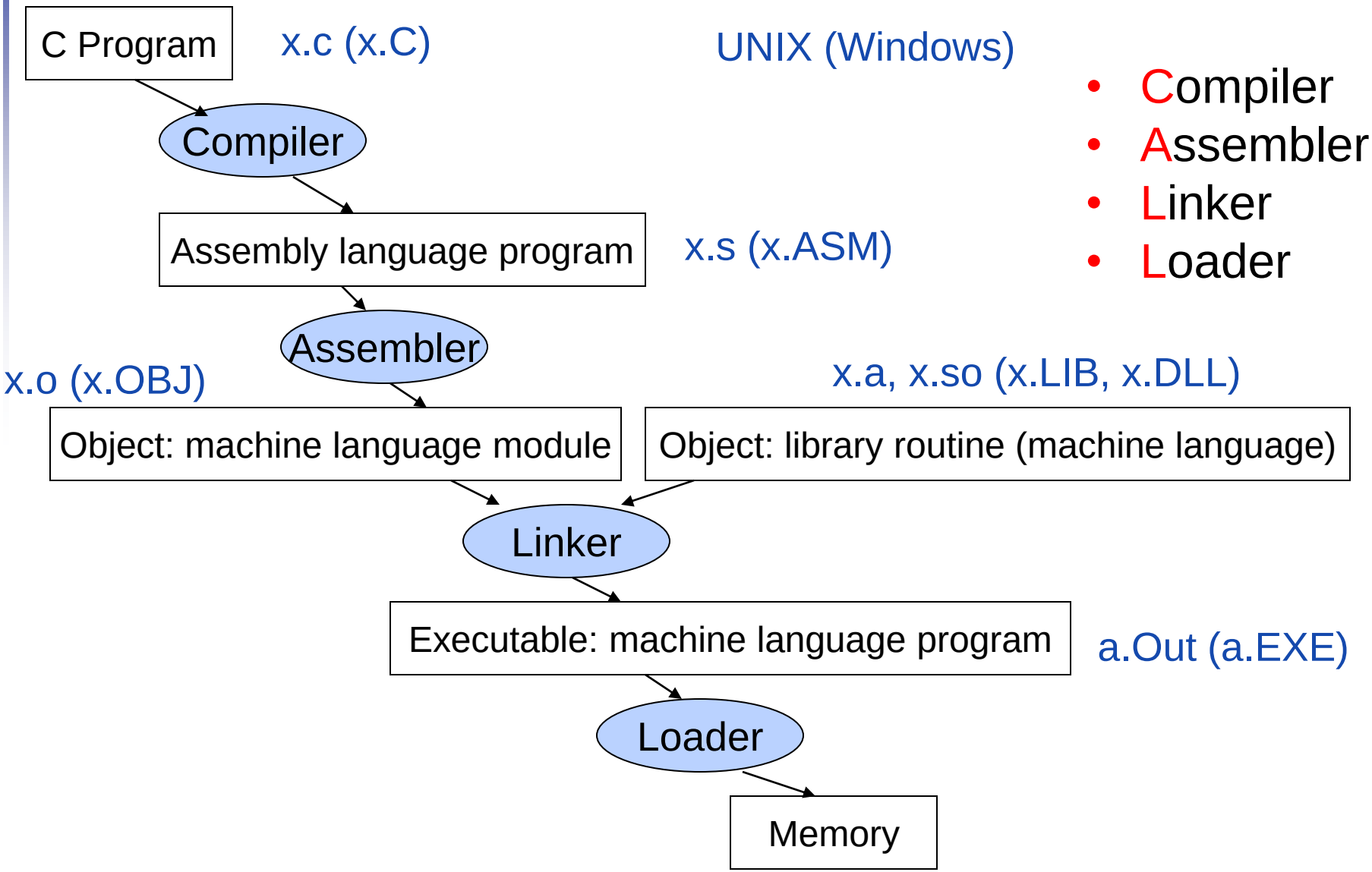
Lecture 5

Performance

Recap – Procedure Calling

- Transfer control instructions
  - Caller: jal
  - Callee: jr
- Argument and return value transfer:
  - Arguments: a0-a7
  - Return value: a0-a1
- Register saving conventions:
  - Use stack push/pop to save register values, $sp, sw, lw
  - Caller-saved registers: a0-a7, t0-t6, ra
  - Callee-saved registers: s0-s11
- Leaf procedure vs. non-leaf procedure
  - Non-leaf procedure should save ra and a# registers

Running a Program: “CALL”



- Compiler
- Assembler
- Linker
- Loader

Compiler

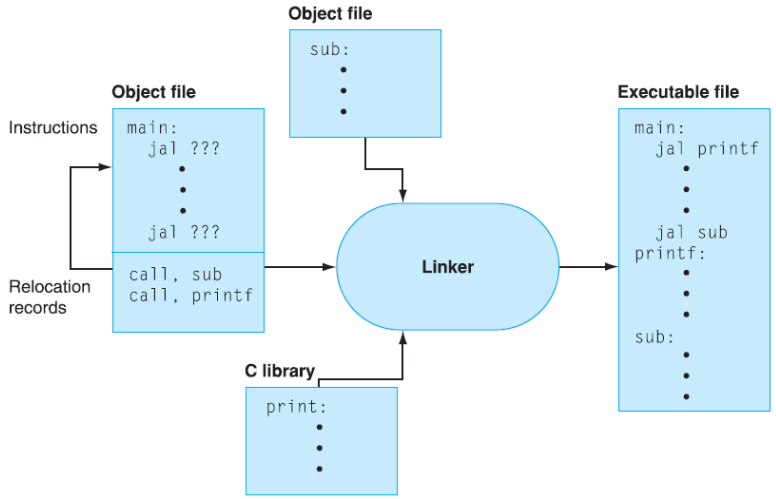
- Input: Higher-level language code
  - e.g. C files such as foo.c
- Output: Assembly Language Code
  - e.g. foo.s
- The output may contain pseudo-instructions
- Optimizing compilers today can produce assembly language programs nearly as well as an assembly language expert, and sometimes even better for large programs.

Assembler

- Input: Assembly language code
  - e.g. foo.s
- Output: Object code, information tables
  - e.g. foo.o – Object file
- Reads and Uses Directives
- Replace Pseudo-instructions
  - pseudo-instrs make it easier to program in assembly
  - examples: “mv”, “j”, etc.
- Produce Machine Language rather than just Assembly
- Creates Object File

Linker

- Input: Object Code files, information tables
  - e.g. foo.o, lib.o
- Output: Executable Code
  - e.g. a.out
- Stitches different object files into a single executable
  - patch internal and external references
  - determine addresses of data and instruction labels
  - organize code and data modules in memory



Role of Linker

- Object files

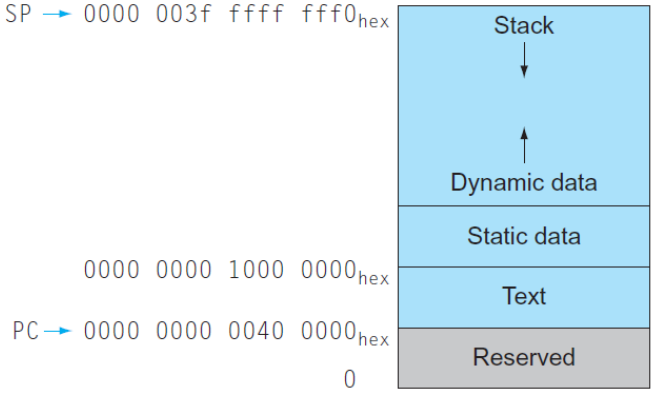
Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`ld x10, 0(x3)`	
	4	`jal x1, 0`	
	. . .	. . .	
Data segment	0	(X)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	`ld`	X
	4	`jal`	B
Symbol table	Label	Ad	
	X		
	B		

	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sd x11, 0(x3)`	
	4	`jal x1, 0`	
	. . .	. . .	
Data segment	0	(Y)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	`sd`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Role of Linker

- Executable file
  - jal x1, 252 # target address: 0x400004 + 252_{ten} = 0x400100
  - jal x1, -260 # target address: 0x400104 – 260_{ten} = 0x400000

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0000\ 0000\ 0040\ 0000_{hex}$	ld x10, 0(x3)
	$0000\ 0000\ 0040\ 0004_{hex}$	jal x1, 252_{ten}
	. . .	. . .
	$0000\ 0000\ 0040\ 0100_{hex}$	sd x11, 32(x3)
	$0000\ 0000\ 0040\ 0104_{hex}$	jal x1, -260_{ten}
	. . .	. . .
Data segment	Address	
	$0000\ 0000\ 1000\ 0000_{hex}$	(X)
	. . .	. . .
	$0000\ 0000\ 1000\ 0020_{hex}$	(Y)
	. . .	. . .

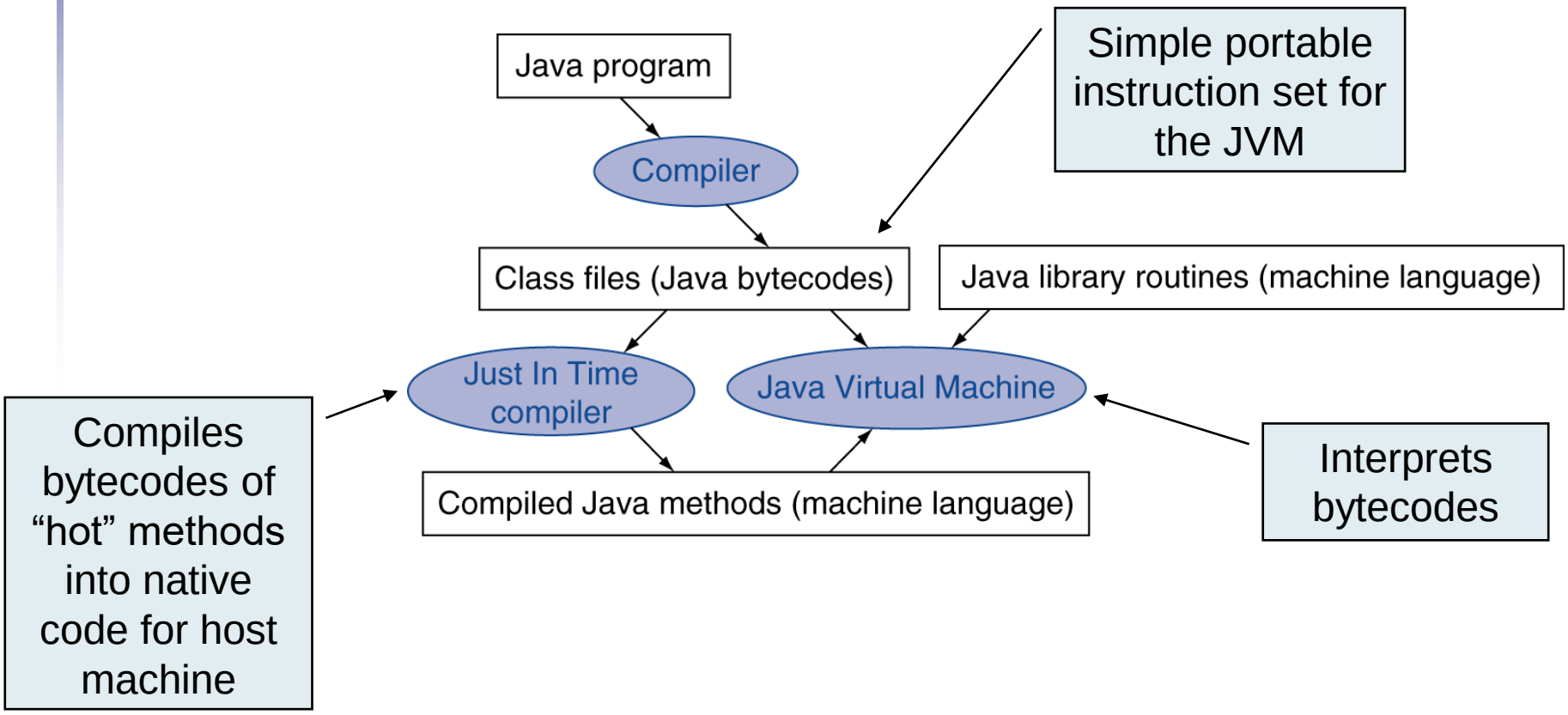


RV64 architecture executable example and memory layout

Loader

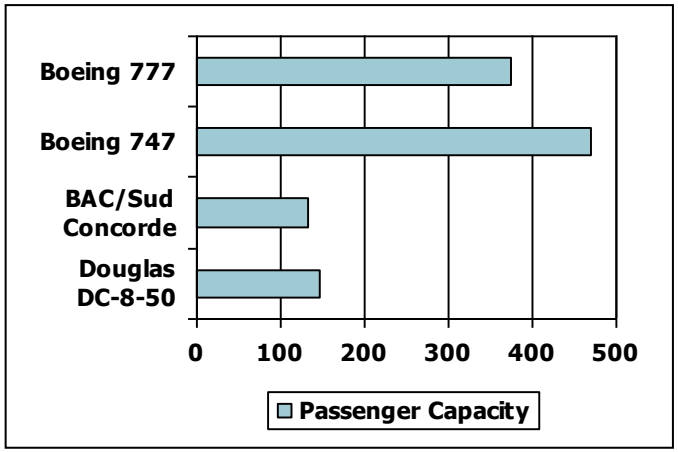
- Input: Executable Code
  - e.g. a.out
- Output: <program is run>
- Executable files are stored on disk
- When one is run, loader’s job is to load it into memory and start running it
- In reality, loader is the operating system (OS)
  - loading is one of the OS tasks
  - And these days, the loader actually does a lot of the linking: Linker's 'executable' is actually only partially linked, instead still having external references (Dynamic Linking)

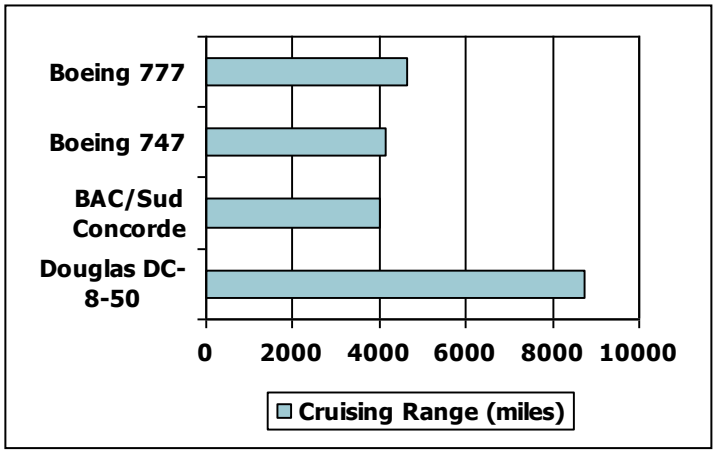
Starting Java Applications

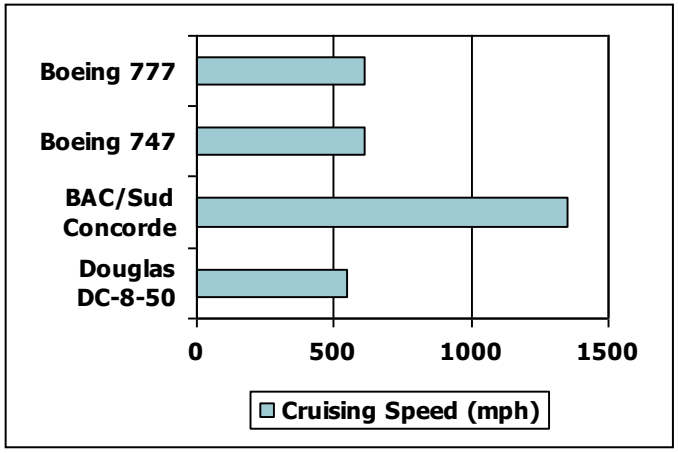


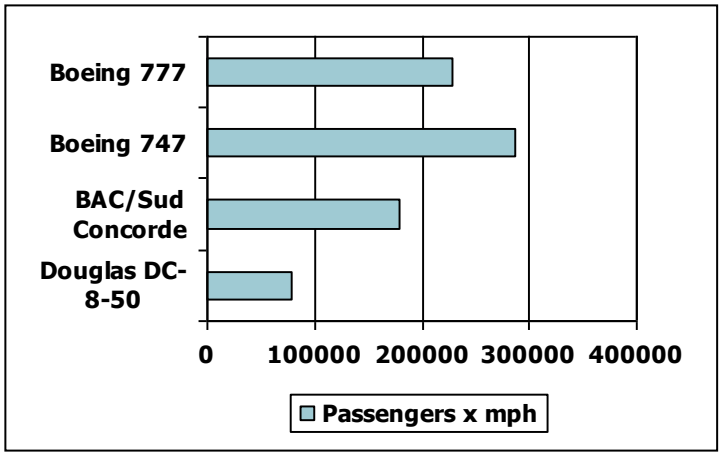
Defining Performance

- Which airplane has the best performance?
  - To evaluate the performance, we must define the metric first!









Response Time and Throughput

- Performance of a computer is based on the following criteria:
- Response time
  - Elapsed time between the start and the end of one task
  - i.e. How long it takes to do a task
- Throughput
  - Total number of tasks finished in a given interval of time.
  - e.g., tasks/transactions/… per hour
- How are response time and throughput affected by
  - Replacing the processor with a faster version?
  - Adding more processors?
- We'll focus on **response time** for now…

Relative Performance

- Defining performance:

$$\text{Performance} = \frac{1}{\text{Execution time}}$$

- What does "X is n times as fast as Y" mean?

$$\text{Performance ratio (n)} = \frac{\text{Performance}_x}{\text{Performance}_y} = \frac{\text{Execution time}_y}{\text{Execution time}_x}$$

- Example:
  - 10s on x, 15s on y for the same program running

$$\frac{\text{Execution time}_y}{\text{Execution time}_x} = \frac{15\text{s}}{10\text{s}} = 1.5$$

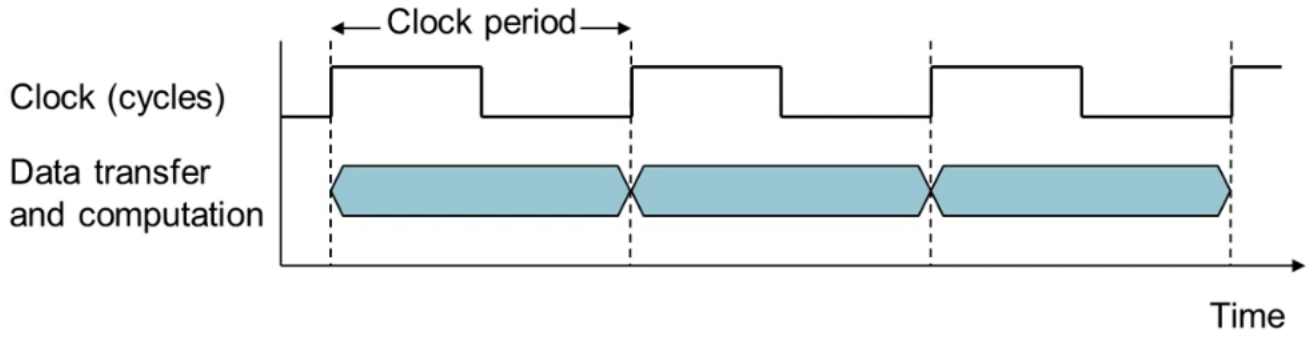
  - So x is 1.5 times as fast as y
- Increase performance = decrease execution time
  - "**improve**" performance/execution time

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 specific task
    - Minus I/O time, other jobs' shares
    - Includes user CPU time and system CPU time
  - Different programs are affected differently by CPU and system performance
    - Running on servers-I/O performance-hardware and software
    - Total elapsed time is of interest
    - Define performance metric and then proceed

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock (Notions: Clock cycle, Clock period, Clock rate/frequency)



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

CPU Time

$$\text{CPU Time} = \text{No. of Clock Cycles} \times \text{Clock Period}$$

$$= \frac{\text{No. of Clock Cycles}}{\text{Clock Rate}}$$

- Performance can be improved by
  - Reducing the number of clock cycles (cycle count)
  - 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 × No. of clock cycles
- How fast must Computer B clock be?

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

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

$$\begin{aligned}\text{Clock Cycles}_A &= \text{CPU Time}_A \times \text{Clock Rate}_A \\ &= 10\text{s} \times 2\text{GHz} = 20 \times 10^9\end{aligned}$$

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

Instruction Count and CPI

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

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

- CPU executes instructions sequentially
- Instruction Count for a program
  - Determined by program, ISA, and compiler
- Clock cycles per instruction (CPI)
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI gets affected by instruction mix
  - Average cycles per instruction: No. of cycles/Instruction Count

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?

$$\text{CPU Time} = \text{Instructions} \times \text{CPI} \times \text{Clock Period} = \frac{\text{Instructions} \times \text{CPI}}{\text{Clock Rate}}$$

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

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

$$\frac{\text{Performance}_x}{\text{Performance}_y} = \frac{\text{Execution time}_y}{\text{Execution time}_x} = n$$

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

- So, computer A is 1.2 times as fast as computer B

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

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

Relative frequency

Compiler design - Code Segments

- Alternative compiled code sequences using instructions in classes A, B, C. Instruction counts? CPI? Which is faster?

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

Performance Summary

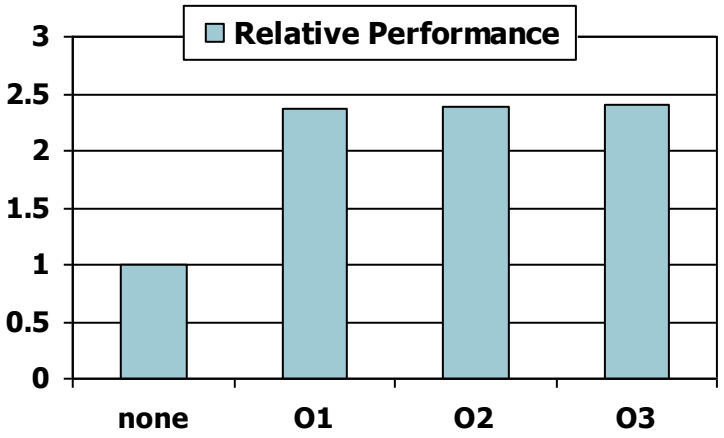
- **The Classic CPU Performance Equation**

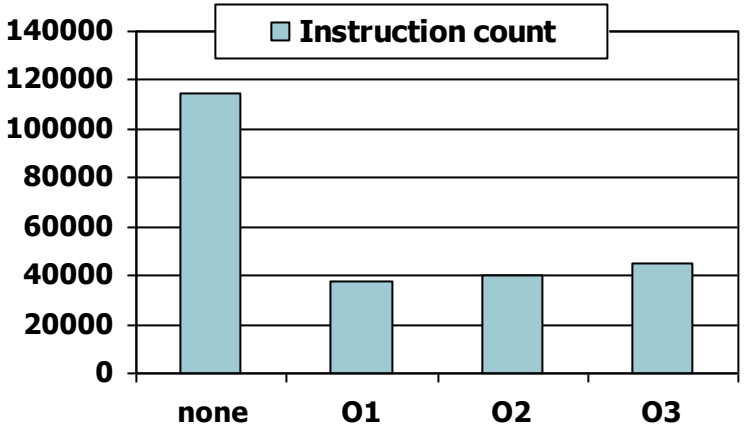
$$\begin{aligned}\text{CPU Time} &= \text{Instruction Count(IC)} \\ &\quad \times \text{Cycles per Instruction(CPI)} \\ &\quad \times \text{Clock Period}(T_c) \\ &= IC \times CPI \times T_c \\ &= IC \times CPI / f \quad (f = 1/T_c)\end{aligned}$$

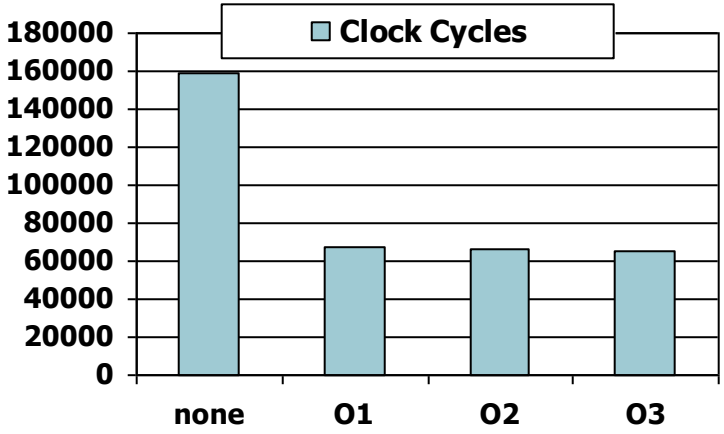
- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

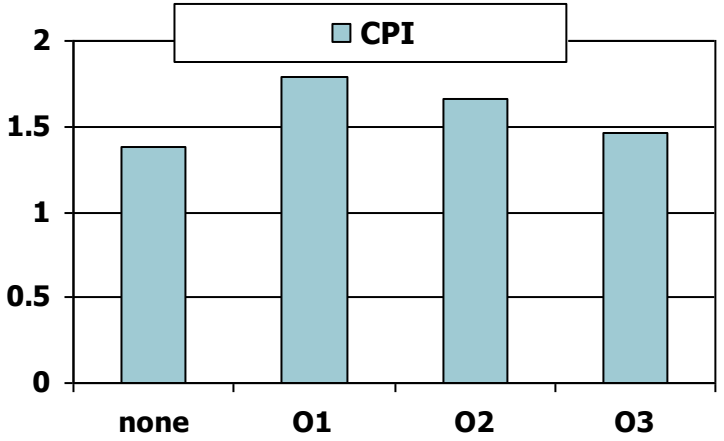
Effect of Compiler Optimization

- Compiled with gcc for Pentium 4 under Linux



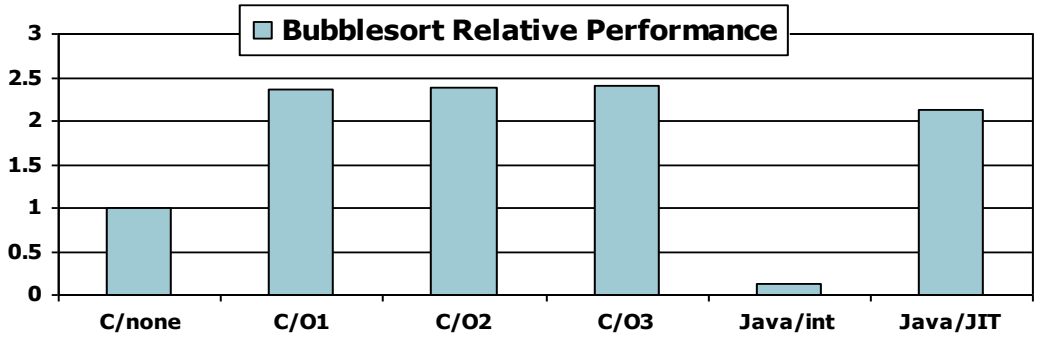


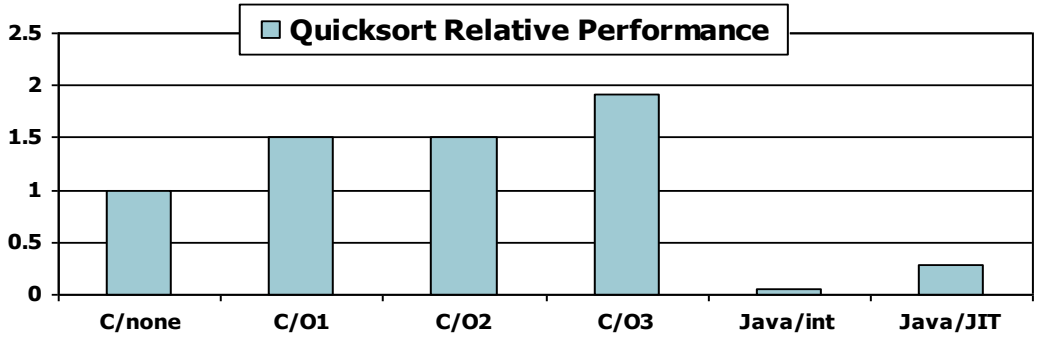


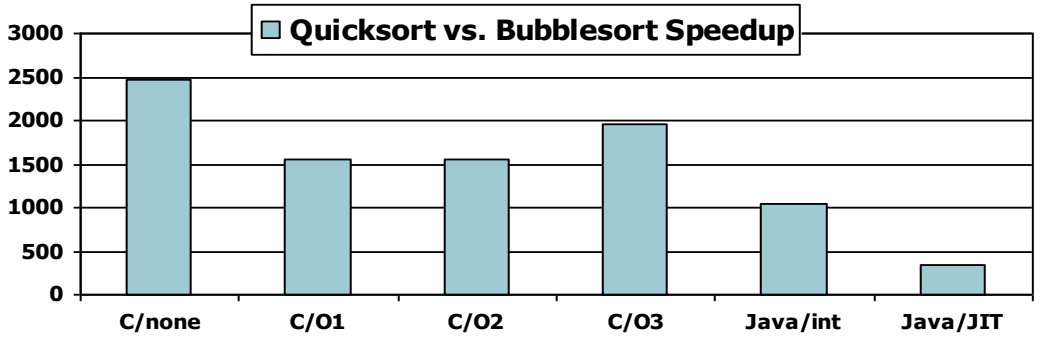


Effect of Language and Algorithm

- Sort Algorithm Comparison







Remarks

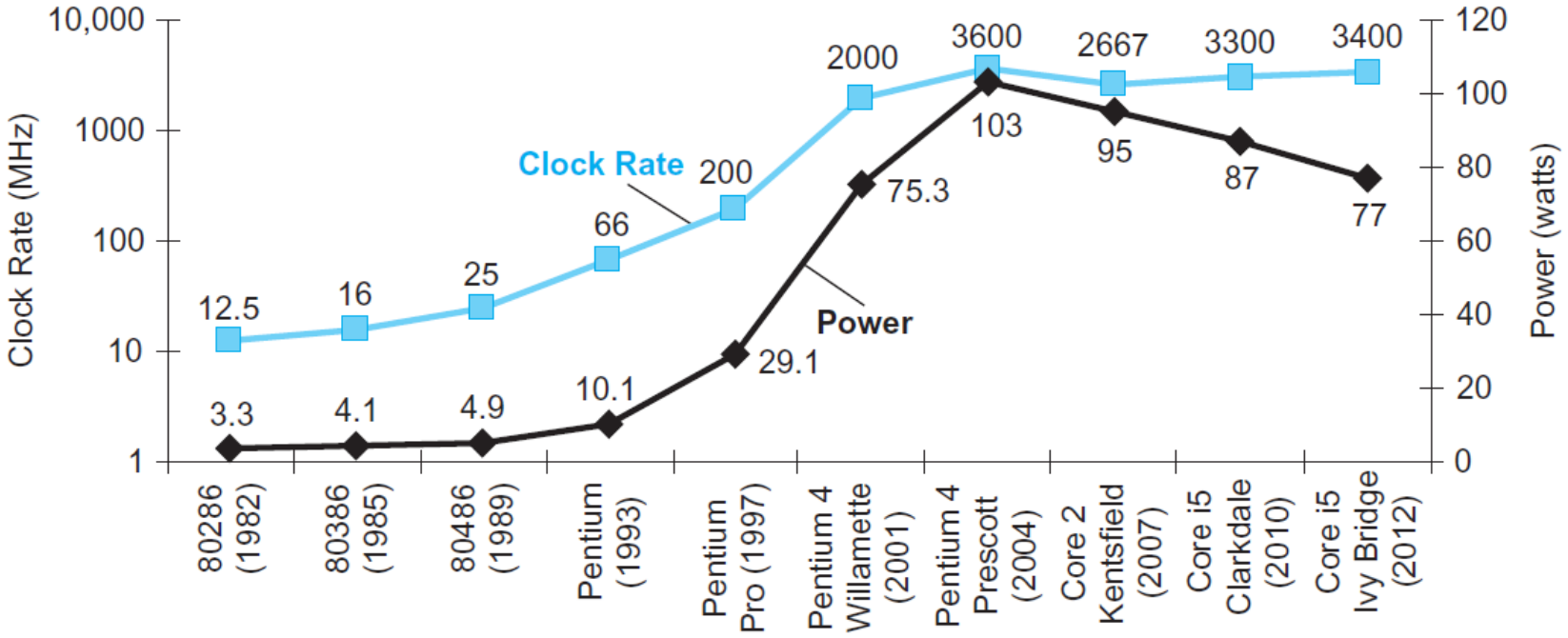
- Instruction count and CPI are not good performance indicators in isolation
- Execution time(CPU Time) is the only complete and reliable measure of computer performance
- Compiler optimizations are sensitive to the algorithm
- Nothing can fix a dumb algorithm!

Energy Consumption of a chip

- Energy consumption = dynamic energy + static energy
  - Dynamic energy (energy spent when transistors switch from 0→1 1→0) is primary
  - Static energy is the energy cost when no transistor switches
- Energy for 0→1→0: $Energy \propto Capacitive\ load \times Voltage^2$
- Energy for 0→1 or 1→0: $Energy \propto 1/2 \times Capacitive\ load \times Voltage^2$
- Energy per second (power):

$$Power \propto 1/2 \times Capacitive\ load \times Voltage^2 \times Frequency\ switched$$

Power Trends



- In CMOS IC technology

$$\text{Power} \propto \frac{1}{2}\text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×23

5V → 1V

×270

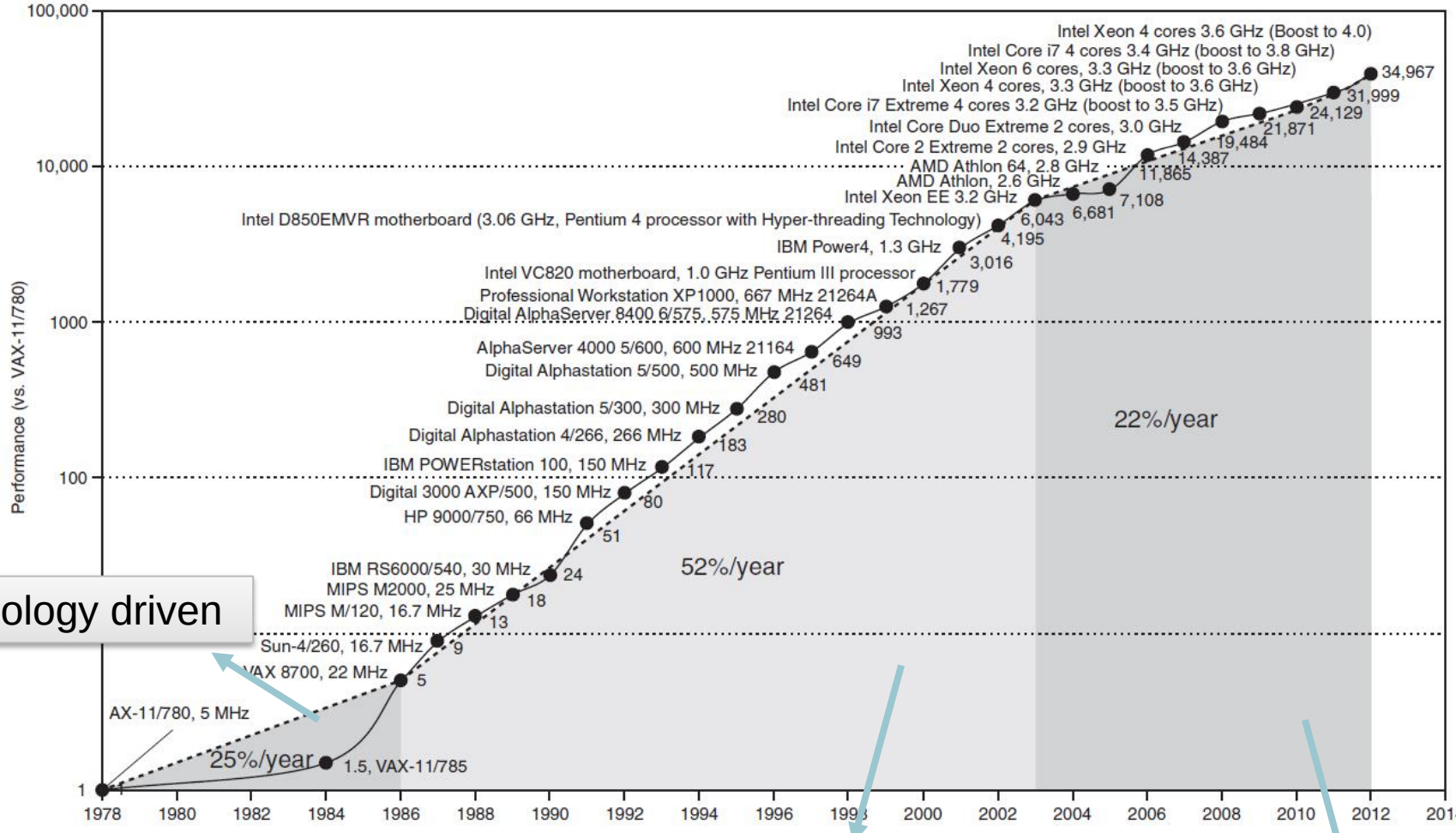
Reducing Power

- 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 (leakage power)
  - We can't remove more heat
- How else can we improve performance?

Uniprocessor Performance



Technology driven

Advanced architectural and organizational ideas

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 (Why?)
  - Programming for performance
  - Load balancing
  - Optimizing communication and synchronization

Benchmark Suites

- Each vendor announces a SPEC (Standard Performance Evaluation Cooperative) rating for their system
  - a measure of execution time for a fixed collection of programs
  - is a function of a specific CPU, memory system, IO system, operating system, compiler
  - enables easy comparison of different systems
- The key is coming up with a collection of relevant programs

SPEC CPU Benchmark

- Programs used to measure performance
  - Supposedly typical of actual workload
- Standard Performance Evaluation Cooperative (SPEC)
  - Develops benchmarks for CPU, I/O, Web, …
- SPEC CPU2006
  - Elapsed time to execute a selection of programs
    - Negligible I/O, so focuses on CPU performance
  - Normalized relative to reference machine
  - Summarize as geometric mean of performance ratios
    - CINT2006 (integer) and CFP2006 (floating-point)

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

CINT2006 for Intel Core i7 920

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7

SPEC Power Benchmark

- Power consumption of server at different workload levels
  - Performance: ssj_ops (server side Java operations per second)
  - Power: Watts (Joules/sec)

$$\text{overall ssj\_ops per watt} = \left(\sum_{i=0}^{10} \text{ssj}_o\text{ps}_i\right) / \left(\sum_{i=0}^{10} \text{power}_i\right)$$

- 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	1922
∑ssj_ops / ∑power =		2490

load:负载

Fallacy: Low Power at Idle

- Fallacy (谬误)
- Look back at i7 power benchmark
  - At 100% load: 258W
  - At 50% load: 170W (66%)
  - At 10% load: 121W (47%)
- Google data center
  - Mostly operates at 10% – 50% load
  - At 100% load less than 1% of time
- Consider designing processors to make power proportional to the load !

Amdahl's Law

- Architecture design is very bottleneck-driven – make the common case fast, do not waste resources on a component that has little impact on overall performance/power
- Amdahl's Law: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play
- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

→ Can't be done!

Summary

- Knowledge of hardware improves software quality:
  - compilers, OS, threaded programs, memory management
- Important trends:
  - growing density of transistors
  - move to multi-core
  - slowing rate of performance improvement
  - power/thermal constraints
- Reasoning about performance:
  - clock speeds, CPI, benchmark suites, performance equations