EECS4030: Computer Architecture

Computer Abstractions and Technology (II)

Prof. Chung-Ta King

Department of Computer Science

National Tsing Hua University, Taiwan

(Adapted from textbook slides https://www.elsevier.com/books-and-journals/book-companion/9780128122754/lecture-slides)

Technology and Computer Summary

- Technology progresses: 0-1 switch
  Mechanic → electro-mechanic → electronic
  (vacuum tube → transistor → integrated circuit)
  Moore's Law
  - Size ↓
  - Switching speed ↑
  - Reliability ↑
  - Power ↓
  - Cost ↓

> 2-fold effects of IC technology scaling on computer performance:
> - Faster without change of design
> - More transistors to implement new architecture features

- Also requires innovative architectural ideas to ride the technology scaling for computer advances

Outline

- Computer: a historical perspective
- Great ideas in computer architecture (Sec. 1.2)
- Below your program (Sec. 1.3)
- Under the covers (Sec. 1.4)
- Technologies for building processors and memory (Sec. 1.5)
- Performance (Sec. 1.6)
- The power wall (Sec. 1.7)
- From uniprocessors to multiprocessors (Sec. 1.8)
- Benchmarking for performance and power (Sec. 1.9)
- Fallacies and Pitfalls (Sec. 1.10)

Why Study Performance?

- As a computer scientist/engineer, your task is not just to **solve a problem**, but to **solve a problem better**
  - Solving a problem is related to correctness
  - Solving a problem better is related to optimization
- As computer architects, we strive to design better computers
- What do we mean by “better”? Better for “what”?
  - Optimization goals

C/P ratio

- How do we know “how good” our design is?
  - Preferable quantized values → performance metrics
  - Performance-guided design

Defining Performance

- Which airplane has the best performance?

Boeing 777

Boeing 747

BAC/Sud Concorde

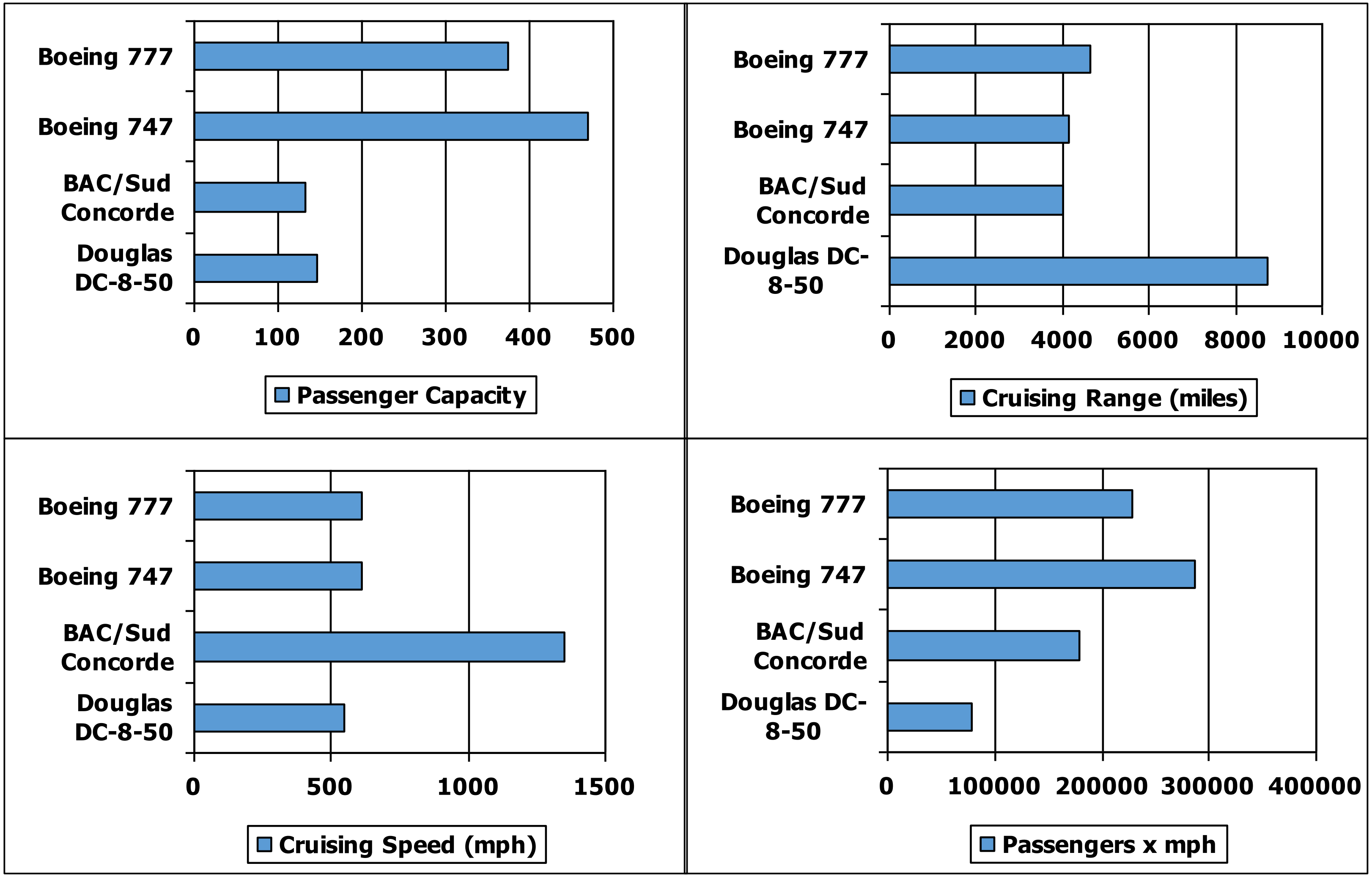
Douglas DC-8-50

Which Airplane Has the Best Performance?

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers × m.p.h.)
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

Fig. 1.14

What Do You Mean by Performance?



For Us, Time Is the Ultimate Measure

- For comparing performance of individual computers, time (how fast can we compute) is most important
  - v.s., comparing performance of servers in data centers
- Yet, still two performance metrics related to time:
  - Execution time (response time, latency): how long it takes to do a task (focusing more on non-interactive apps)
  - Throughput: total work done per unit time
    - e.g., tasks/transactions/… per hour
  - Ex.: how are execution time and throughput affected by
    - Replacing the processor with a faster version?
    - Adding more processors?

Which one is easier to improve?

Latency vs. Throughput

Plane	DC to Paris	Speed	Passengers	Throughput (pph)
	6.5 hours	610 mph	470	72.3
	3 hours	1350 mph	132	44

- Latency (flying time) of Boeing 747 vs. Concorde
  - 6.5 hours vs 3 hours (2.17:1)
  - Concord is 2.17 times faster in terms of latency
- Throughput of Boeing 747 vs. Concorde
  - 72.3 pph vs 44 pph (1.63:1) (pph: person / hour)
  - Boeing is 1.63 times faster (better) in terms of throughput

Will focus on latency

(Prof. Jing-Jia Liou)

Performance in Terms of Execution Time

- Time is the measure of computer performance
  - The computer that performs the same amount of work, e.g. same program, in the least time has a higher performance
- So, we can quantize "performance" as

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

  - Just one way to define "performance"
- "*X* is *n* time faster than *Y*" means

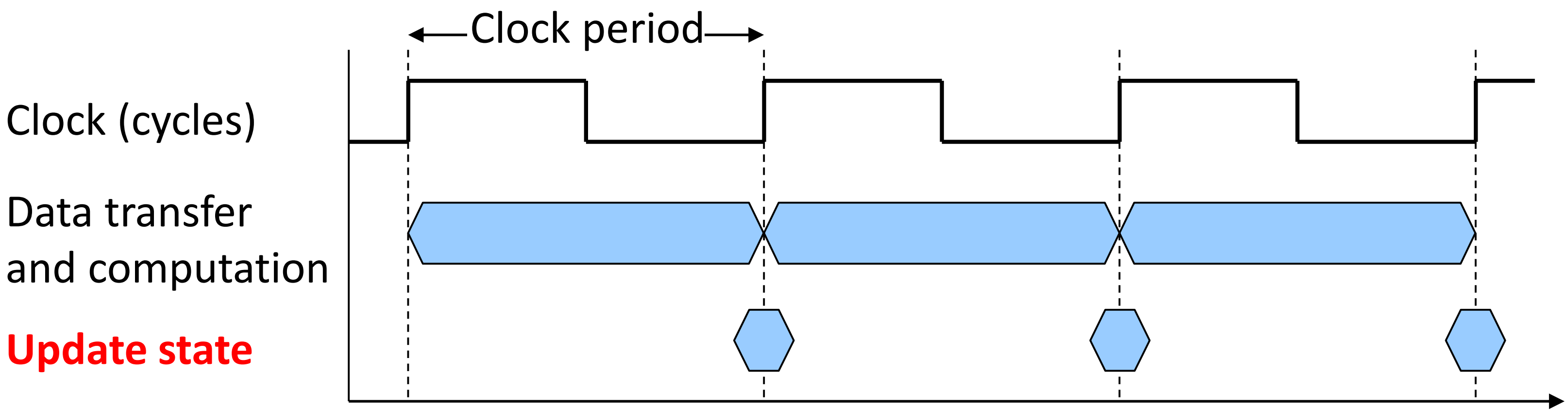
$$\frac{\text{Performance}_{\text{X}}}{\text{Performance}_{\text{Y}}} = \frac{\text{Execution time}_{\text{Y}}}{\text{Execution time}_{\text{X}}} = n$$

What to Measure in Terms of Time?

- But, execution time of a program can be defined differently:
  - Elapsed time: *total response time, wall clock time*
    - Total time to complete a program, including everything: memory accesses, I/O activities, OS overhead, idle time
    - Determines system performance
  - CPU time: time the CPU spent processing this program (discounts I/O time, other jobs' shares)
- CPU time of a program further consists of
  - User CPU time: CPU time spent in the program
  - System CPU time: CPU time spent in OS performing tasks on behalf of this program

How to Express Time?

- Computer time can be seconds or system clocks
  - Clocks related to how fast HW can perform basic functions
  - CPU clocking: operations of digital hardware, such as CPU, are governed by a constant-rate clock



  - *Clock period*: duration of a clock cycle, e.g., 1 ns=1×10^{-9} s
  - *Clock frequency* (rate): cycles per second, e.g., 4 GHz = 4000 MHz = 4.0×10^{9} Hz

Clocking and Circuits

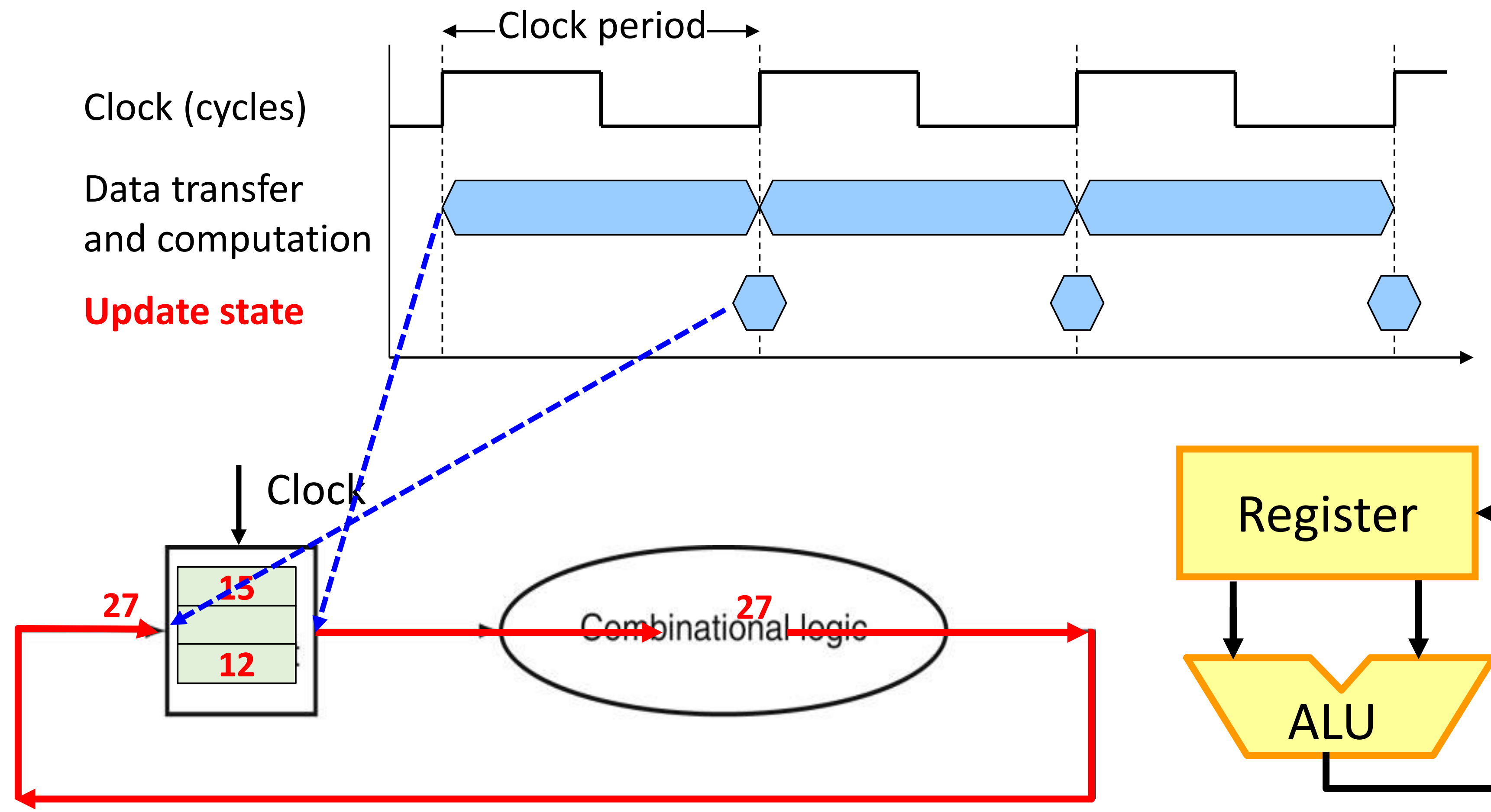


Fig. B.7.3. Clocked sequential circuit

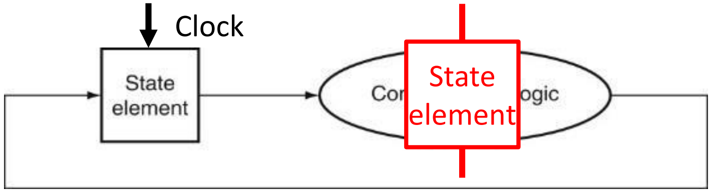
Clock period = longest paths between registers (complexity of computation)

CPU Performance

- The most fundamental CPU performance measure is CPU execution time of a program (*user CPU time*):

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

- CPU performance can be improved by
  - Reducing number of clock cycles
  - Increasing clock rate
- Hardware designers must often trade off clock rate against cycle count (Why?)



CPU Performance Example

- Computer A: 2GHz clock, 10s CPU time for our prog.
- Want to design Computer B
  - Aim for 6s CPU time for our program
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

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

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

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

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

Observations?

CPU Performance & Program Instructions

- CPU performance is determined by the total number of CPU clock cycles to execute the program

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

```
swap:
      multi  $2, $5,4
      add    $2, $4,$2
      lw     $15, 0($2)
      lw     $16, 4($2)
      sw     $16, 0($2)
      sw     $15, 4($2)
      jr     $31
```

- Total CPU clock cycle is affected by the number of instructions and their types
  - e.g., a multiply takes more CPU cycles than an addition
- Need a way to relate CPU time with instructions in this program

CPU Performance & Program Instructions

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

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

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

- Instruction Count (IC) of execution of a program
  - Determined by program, ISA and compiler
- Average Cycles per Instruction (CPI)
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by *instruction mix*

Not static IC

Which is better?
CPI ↑ or CPI ↓

Can CPI < 1?

CPI Example

- Computer A: clock cycle time = 250 ps, CPI = 2.0
- Computer B: clock cycle time = 500 ps, CPI = 1.2
- Same ISA → same # of instructions for a program
- 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

Observations?

CPI in More Detail

- If different instruction classes take different numbers of cycles (*n* classes of instructions), e.g. add/sub, ld/sd, multi, jr

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

```
swap:
      multi  $2, $5,4
      add    $2, $4,$2
      lw     $15, 0($2)
      lw     $16, 4($2)
      sw     $16, 0($2)
      sw     $15, 4($2)
      jr     $31
```

- Weighted average CPI

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

Instruction mix, relative frequency of ith class of instructions in the program

CPI Example

- Two compilers generate two different code sequences using different instruction mixes in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
Instruction count in sequence 1	2M	1M	2M
Instruction count in sequence 2	4M	1M	1M

- Sequence 1: IC = 5M
  - Clock Cycles
    $= 2M\times1 + 1M\times2 + 2M\times3$
    $= 10M$
  - Avg. CPI = 10M/5M = 2.0
- Sequence 2: IC = 6M
  - Clock Cycles
    $= 4M\times1 + 1M\times2 + 1M\times3$
    $= 9M$
  - Avg. CPI = 9M/6M = 1.5

Observations?

Performance Summary

- The only complete and reliable measure of computer performance is **time**
  - Response time, execution time, latency
  - Throughput
- Execution time of a program on a computer can be
  - Elapsed time (total response time, wall clock time)
  - CPU time
- CPU time of a program consists of
  - *User CPU time*
  - *System CPU time*

  Can be measured by number of CPU cycles and cycle time
- Number of CPU cycles is related to instruction count

Performance Summary

- Computer performance of a program is thus

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

  - Must look at all three when comparing two computers
- Performance depends on

	Inst. Count	CPI	Clock Rate
Algorithm	√	√	
Program Language	√	√	
Compiler	√	√	
ISA	√	√	√

Outline

- Computer: a historical perspective
- Great ideas in computer architecture (Sec. 1.2)
- Below your program (Sec. 1.3)
- Under the covers (Sec. 1.4)
- Technologies for building processors and memory (Sec. 1.5)
- Performance (Sec. 1.6)
- The power wall (Sec. 1.7)
- From uniprocessors to multiprocessors (Sec. 1.8)
- Benchmarking for performance and power (Sec. 1.9)
- Fallacies and Pitfalls (Sec. 1.10)

Power Trends

- Eight generations of x86 processors:

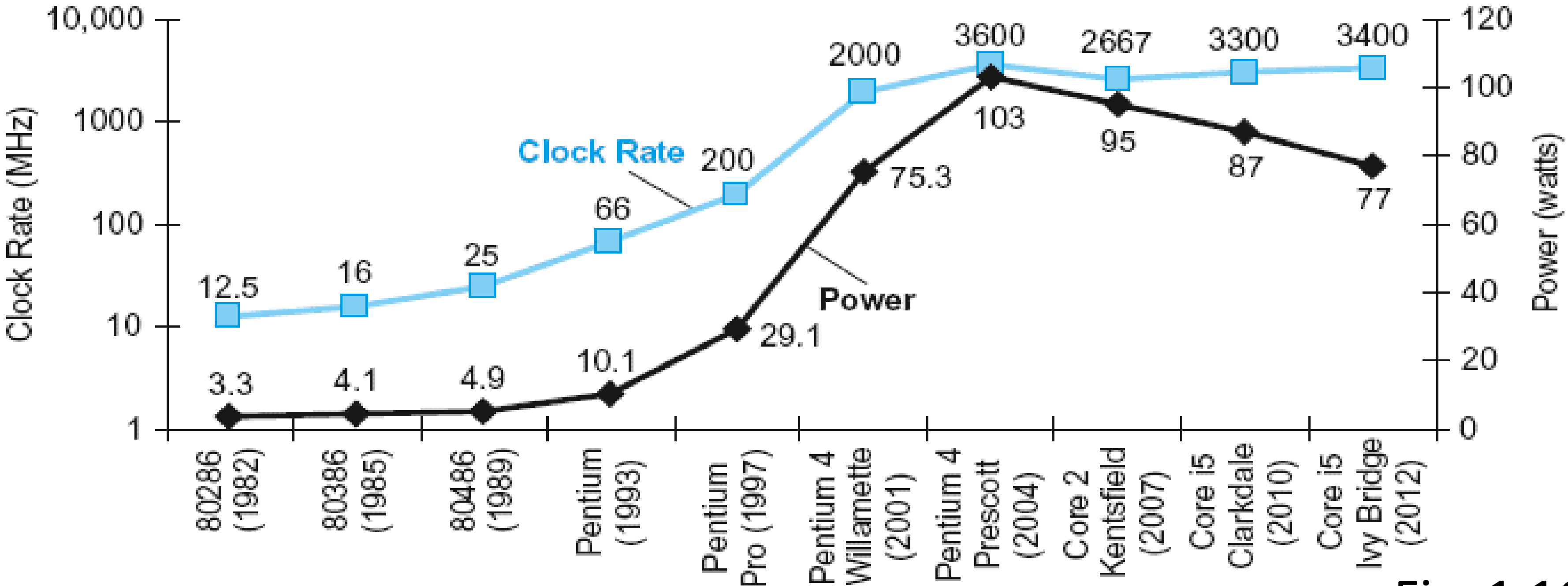


Fig. 1.16

→ power increases as clock rate

- Slowing down after 2004, because of "power wall" (power limit for cooling commodity microprocessors)

Power Trends

- In the PostPC era, the really critical resource is power
  - For personal mobile device → battery life
  - For data centers → powering and cooling servers
- For IC technology based on CMOS, primary energy consumption is dynamic energy
  - Depends on capacitive loading of each transistor and the voltage applied for a single 0 → 1 or 1 → 0 transition

$$\text{Energy} \propto {}^{1}/_{2} \times \text{Capacitive load} \times \text{Voltage}^2$$

  - Power required per transistor

$$\text{Power} \propto {}^{1}/_{2} \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

    - Can be lowered by reducing voltage (15% per IC generation) or clock rate

Power Reduction Example

- Suppose a new CPU uses IC technology with
  - 85% of capacitive load of old CPU
  - 15% voltage reduction and thus 15% frequency reduction

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

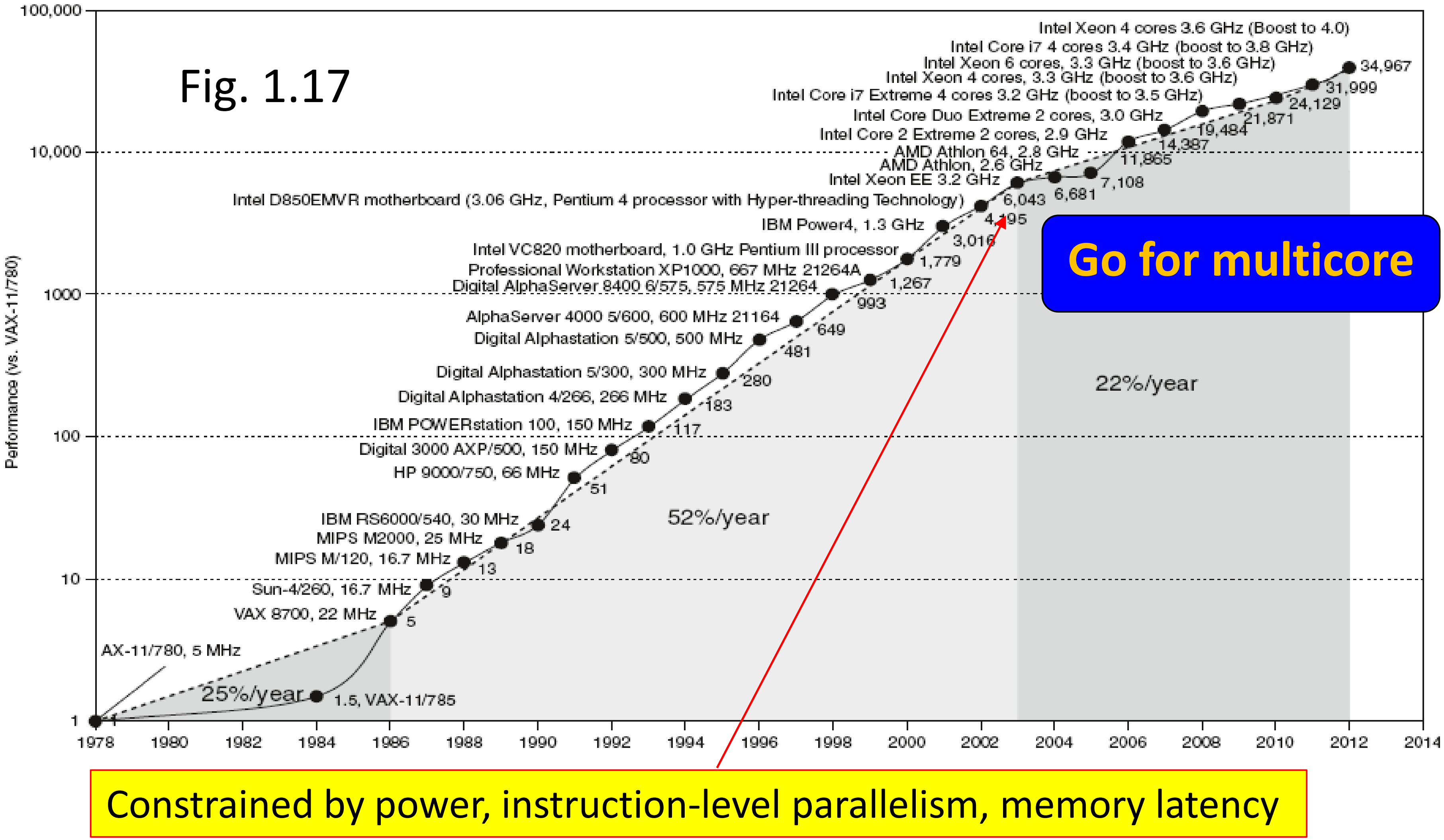
- Problem with lowering voltage
  - *Static energy* consumption (due to leakage current, even when a transistor is off) becomes dominant
- The power wall
  - We cannot reduce voltage further
  - We cannot remove more heat

What else can we do to get performance?

Outline

- Computer: a historical perspective
- Great ideas in computer architecture (Sec. 1.2)
- Below your program (Sec. 1.3)
- Under the covers (Sec. 1.4)
- Technologies for building processors and memory (Sec. 1.5)
- Performance (Sec. 1.6)
- The power wall (Sec. 1.7)
- From uniprocessors to multiprocessors (Sec. 1.8)
- Benchmarking for performance and power (Sec. 1.9)
- Fallacies and Pitfalls (Sec. 1.10)

Uniprocessor Performance



Move into Multiprocessors

- Multicore microprocessors
  - More than one processor (core) per chip
  - A 2-core chip normally consumes less power than a 1-large-core chip of same size and performance
- Free lunch for software in single-core era
  - IC technology scaling (e.g., clock rate) automatically improves program performance
  - Architecture innovation on *instruction level parallelism* lets hardware exploit parallelism among instructions and execute multiple parallel instructions at once
    - Programmer and compiler can view hardware as executing instructions sequentially and no need to change programs
- No more free lunch in multicore

Problems with Multiprocessors

- Require programmers and compilers to be aware of the parallel hardware and to explicitly rewrite their programs to be parallel
- Hard to do:
  - Need to program for performance, not just correctness
  - Load balancing parallel tasks
  - Optimizing communication and synchronization among parallel tasks
- To be elaborated in later chapters

Outline

- Computer: a historical perspective
- Great ideas in computer architecture (Sec. 1.2)
- Below your program (Sec. 1.3)
- Under the covers (Sec. 1.4)
- Technologies for building processors and memory (Sec. 1.5)
- Performance (Sec. 1.6)
- The power wall (Sec. 1.7)
- From uniprocessors to multiprocessors (Sec. 1.8)
- Benchmarking for performance and power (Sec. 1.9)
- Fallacies and Pitfalls (Sec. 1.10)

Comparing Performance

- Recall that "X is *n* time faster than Y" means

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

  - It is easy to compare two computers using one program, but what about multiple programs?
- Why compare performance using multiple programs? → *benchmarking*
  - A standard set of programs specifically chosen to measure and compare computer performance
    - Represent a workload that will predict the performance of the actual workload

How to Summarize Performance Data?

- Two machines with two programs

	Machine A	Machine B
Program 1	2 s	4 s
Program 2	12 s	8 s

- *Arithmetic mean* of execution time:
  - Find average execution time, then find performance ratio
  - Performance of machine B relative to A
    (2+12)/(4+8) = 1.17
  - B is 1.17 times faster than A
  - But, Machine A runs Program 1 twice faster than B!
    → Program B runs longer and dominates the result
  - Need a way to normalize the importance of benchmarks

(Prof. Jing-Jia Liou)

How to Summarize Performance Data?

- Arithmetic mean of execution time ratio:
  - Find performance ratios first, then average them

	Machine A	Machine B
Program 1	2 s	4 s
Program 2	12 s	8 s

  - Performance of machine B relative to A

    (2/4 + 12/8)/2 = 1
    - A is same as B
  - Performance of machine A relative to B

    (4/2 + 8/12)/2 = 4/3
    - A is 1.33 times faster than B
- Different conclusions if different references are used

(Prof. Jing-Jia Liou)

Better Metric to Summarize Performance

- Geometric Mean
  - Consistent performance ratios regardless of reference

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

	Machine A (B as reference)	Machine B (A as reference)
Program 1	4/2 = 2.0	2/4 = 0.5
Program 2	8/12 = 0.667	12/8 = 1.5
Geometric Mean	$(2*0.667)^{1/2} = 1.155$	$(0.5*1.5)^{1/2} = 0.866$
	A : B = 1.155: 1	A : B = 1 : 0.866

(Prof. Jing-Jia Liou)

Example Benchmark Suite: SPEC

- Standard Performance Evaluation Corp (SPEC)
  - Began in 1989 on benchmarking workstation and servers
  - Developed benchmarks for cloud, CPU, Web, …
- SPEC CPU2017 (43 benchmarks)
  - SPECspeed 2017 Integer and Floating Point suites
    - Elapse time to execute each benchmark from the suite
  - SPECrate 2017 Integer and Floating Point suites
    - Elapse time to execute *n* copies of each benchmark
  - Normalize relative to a reference machine
  - Summarize as *geometric mean* of performance ratios
  - For desktop computers or servers using non-interactive applications

spec

SPEC CPU2017 Integer Benchmarks

SPECrate 2017	SPECspeed 2017	Lang.	KLOC	Application Area
500.perlbench_r	600.perlbench_s	C	362	Perl interpreter
502.gcc_r	602.gcc_s	C	1,304	GNU C compiler
505.mcf_r	605.mcf_s	C	3	Route planning
520.omnetpp_r	620.omnetpp_s	C++	134	Discrete Event simulation - computer network
523.xalancbmk_r	623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT
525.x264_r	625.x264_s	C	96	Video compression
531.deepsjeng_r	631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
541.leela_r	641.leela_s	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
548.exchange2_r	648.exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
557.xz_r	657.xz_s	C	33	General data compression

KLOC = line count (including comments/whitespace) for source files used in a build / 1000

SPEC CPU2017 Floating-Point Benchmarks

SPECrate 2017	SPECspeed 2017	Lang.	KLOC	Application Area
503.bwaves_r	603.bwaves_s	Fortran	1	Explosion modeling
507.cactuBSSN_r	607.cactuBSSN_s	C++, C, Fortran	257	Physics: relativity
508.namd_r		C++	8	Molecular dynamics
510.parest_r		C++	427	Biomedical imaging
511.povray_r		C++, C	170	Ray tracing
519.lbm_r	619.lbm_s	C	1	Fluid dynamics
521.wrf_r	621.wrf_s	Fortran, C	991	Weather forecasting
526.blender_r		C++, C	1,577	3D rendering & animation
527.cam4_r	627.cam4_s	Fortran, C	407	Atmosphere modeling
	628.pop2_s	Fortran, C	338	Wide-scale ocean modeling
538.imagick_r	638.imagick_s	C	259	Image manipulation
544.nab_r	644.nab_s	C	24	Molecular dynamics
549.fotonik3d_r	649.fotonik3d_s	Fortran	14	Computational Electromagnetics
554.roms_r	654.roms_s	Fortran	210	Regional ocean modeling

Performance Report

- ASUS WS C621E SAGE Server System
  Intel Xeon Platinum 8180, 2.50 GHz
  - Note how the performance is reported! → repeatable

Benchmark	Base							Peak						
	Threads	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Threads	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
600.perlbench_s	112	**270**	**6.57**	270	6.56	270	6.58	112	226	7.87	227	7.84	**226**	**7.87**
602.gcc_s	112	**389**	**10.2**	391	10.2	389	10.2	112	376	10.6	**377**	**10.6**	386	10.3
605.mcf_s	112	403	11.7	**406**	**11.6**	413	11.4	112	401	11.8	394	12.0	**396**	**11.9**
620.omnetpp_s	112	**202**	**8.06**	198	8.23	203	8.02	112	195	8.38	**195**	**8.36**	207	7.89
623.xalancbmk_s	112	142	9.95	**141**	**10.1**	140	10.1	112	**132**	**10.7**	133	10.6	132	10.7
625.x264_s	112	140	12.6	140	12.6	**140**	**12.6**	112	**140**	**12.6**	140	12.6	140	12.6
631.deepsjeng_s	112	265	5.40	266	5.39	**266**	**5.40**	112	**267**	**5.36**	267	5.36	267	5.37
641.leela_s	112	**371**	**4.59**	371	4.60	372	4.59	112	370	4.60	**370**	**4.61**	370	4.61
648.exchange2_s	112	207	14.2	**207**	**14.2**	207	14.2	112	207	14.2	207	14.2	**207**	**14.2**
657.xz_s	112	252	24.6	**252**	**24.5**	253	24.4	112	251	24.6	252	24.5	**252**	**24.6**
SPECspeed2017_int_base			9.64											
SPECspeed2017_int_peak			9.96											

Results appear in the order in which they were run. Bold underlined text indicates a median measurement.

https://www.spec.org/cpu2017/results/res2018q1/cpu2017-20180121-02622.html

CINT2006 for 2.66GHz 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

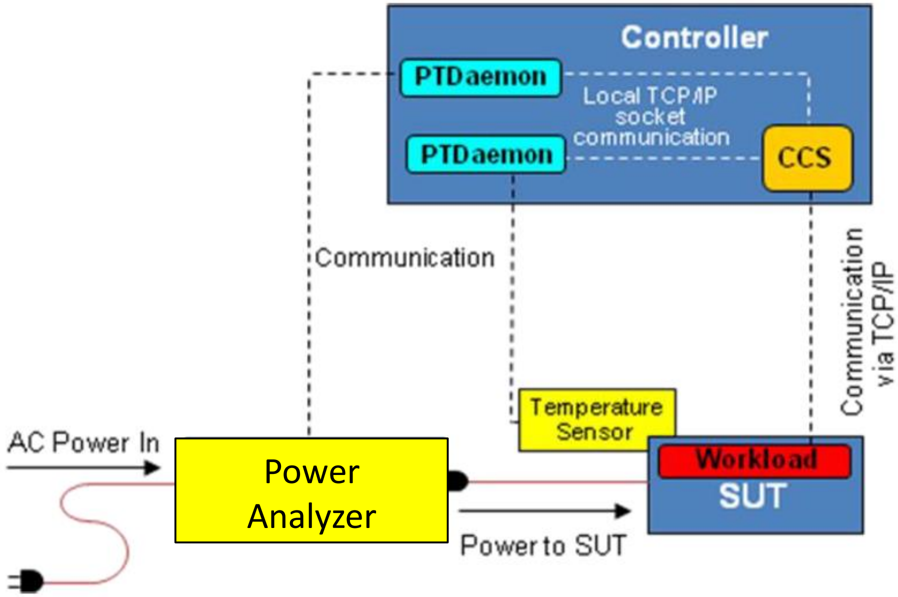
Fig. 1.18

SPEC Power Benchmark

- **SPECpower_ssj2008**: power consumption to generate performance of servers (at 10 workload levels)
  - Performance: ssj_ops
    - Server Side Java (SSJ) workload: Java business application to generate transactions, performance measured in throughput (transactions per sec)
  - Power: Watts (Joules/sec)
  - Metric:

Overall ssj_ops per Watt =

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



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

Fig. 1.19

Outline

- Computer: a historical perspective
- Great ideas in computer architecture (Sec. 1.2)
- Below your program (Sec. 1.3)
- Under the covers (Sec. 1.4)
- Technologies for building processors and memory (Sec. 1.5)
- Performance (Sec. 1.6)
- The power wall (Sec. 1.7)
- From uniprocessors to multiprocessors (Sec. 1.8)
- Benchmarking for performance and power (Sec. 1.9)
- Fallacies and Pitfalls (Sec. 1.10)

Pitfall: Amdahl's Law

Improving one aspect of a computer and expecting a proportional improvement in overall performance

- Example:
  - Traveling from Taipei to Kaohsiung by train would take 4 hr
  - High speed rail shortens the time to 1.5 hr
  - *Improvement factor* = 4/1.5 = 2.67
  - Can we expect the overall performance (door-to-door) to be improved also by the same factor (2.67)?

- Actually, we cannot
  - There is a limitation, as stated by the **Amdahl's Law**

Pitfall: Amdahl's Law

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

- Example: door-to-door improvement by high speed rail
  - T_{affected} = 4 hr
  - $T_{\text{unaffected}}$ = 0.5 hr + 0.5 hr = 1 hr
  - T_{improved} = 4 hr / 2.67 + 1 hr = 2.5 hr
  - Overall improvement factor (speedup) = (4 + 1) hr / 2.5 hr = 2.0 which is less than 2.67
  - Time that cannot be enhanced (local traveling) is more dominant

0.5 hr (local)

0.5 hr (local)

Pitfall: Amdahl's Law

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

- Amdahl's Law is often expressed as *speedup*:

$$\text{Speedup} = \frac{T_{\text{original}}}{T_{\text{improved}}} = \frac{T_{\text{original}}}{\frac{T_{\text{affected}}}{n} + T_{\text{original}} - T_{\text{affected}}}$$

$$\text{Speedup} = \frac{1}{\frac{f}{n} + (1-f)}$$

f: % that can be improved (T_{aff}/T_{orig})
n: improvement factor

$$\lim_{n\to\infty}\left(\frac{1}{\frac{f}{n} + (1-f)}\right) = \frac{1}{1-f}$$

Performance improvement from using enhancement E is limited by the fraction that E cannot be applied

Fallacy: Low Power at Idle

Computers at low utilization use little power

- Look back at i7 power benchmark
  - At 100% load: 258W
  - At 50% load: 170W (66%)
  - At 10% load: 121W (47%)

Ideally, should be 10%!

- Google datacenter
  - Mostly operates at 10% ~ 50% load
  - At 100% load less than 1% of the time
- Should consider designing processors to make power proportional to load (*energy-proportional computing*)

Pitfall: MIPS as a Performance Metric

Using a subset of the performance equation as a performance metric

- Ex.: MIPS (Millions of Instructions Per Second)
  - Doesn't account for capacities of instructions → cannot compare computers with different ISAs
  - MIPS varies between programs on the same computer

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

  - MIPS can vary independently from performance (e.g., more instructions but each is faster)

Concluding Remarks

- Cost/performance is improving
  - Due to underlying technology development
  - Also need architecture innovations to scale performance
- Eight great architecture ideas and hierarchical layers of abstraction
  - In both hardware and software
- Instruction set architecture
  - The hardware/software interface
- Execution time: the best performance measure
- Power is a limiting factor
  - Use parallelism to improve performance