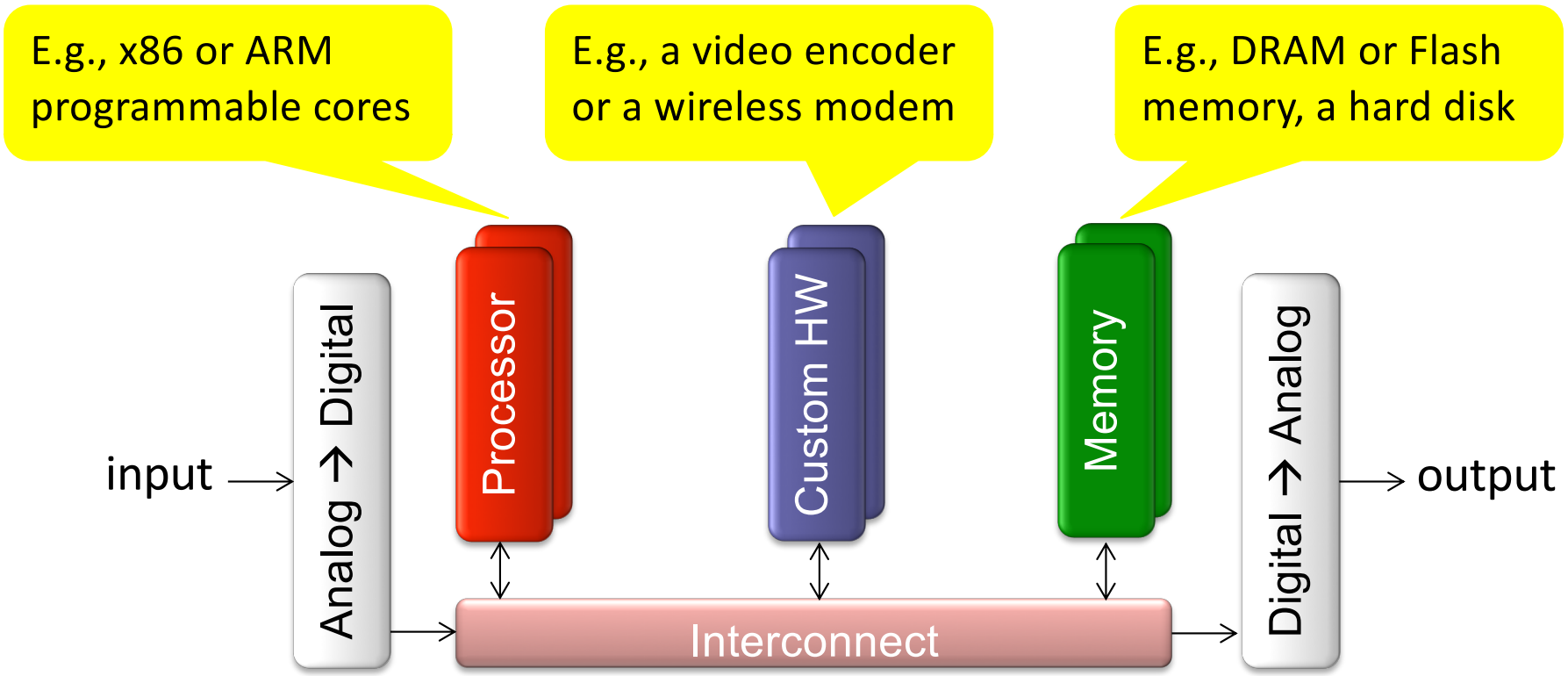
CMPE110 Lecture 02
Performance

Heiner Litz

https://canvas.ucsc.edu/courses/19290

Review

Inside a Typical System



Semiconductor Chips

The dominant technology for integrated circuits

Processing, memory, and I/O functionality

Print-like process

On a semiconductor surface, we print transistors (switches) for logic, memory devices, and their interconnect

Print resolution improves over time → more devices

Can build more capable digital systems

But cost remain the same

The Famous Moore's Law

Devices get smaller

- Get more devices on a chip
- Devices get faster

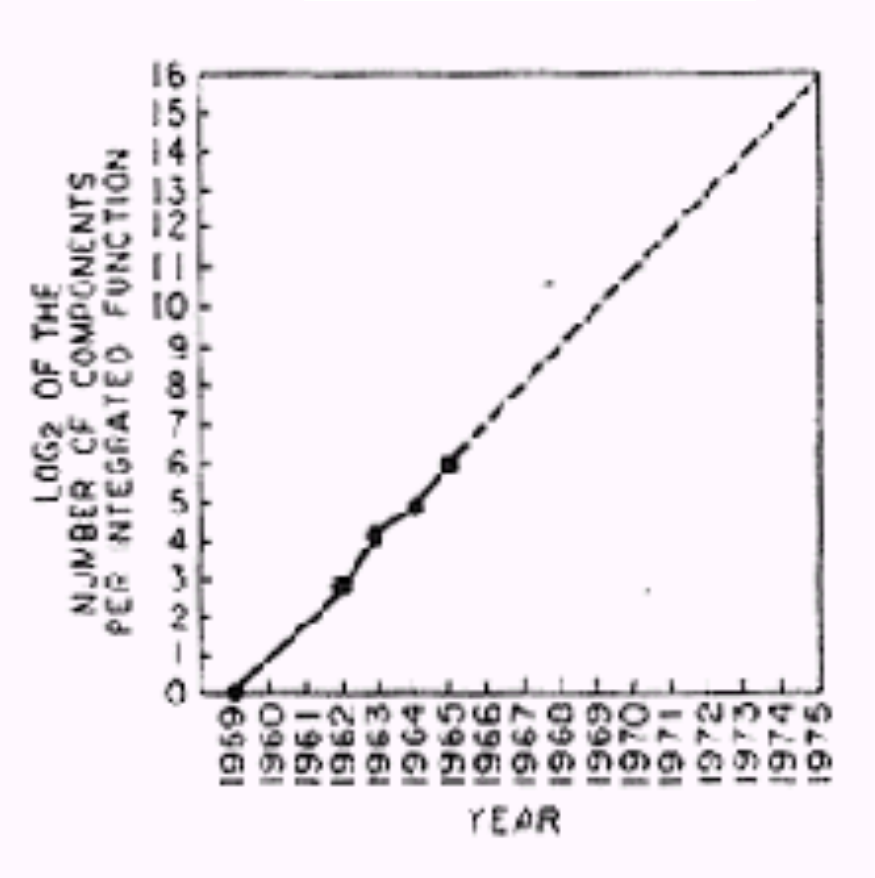
Initial graph from 1965 paper

- Prediction: 2x density per year
- Reality: ~2x density every 3 years

Is Moore's Law really a Law?

What does it say about performance?

Electronics, Volume 38, Number 8, April 19, 1965



Sense of Scale

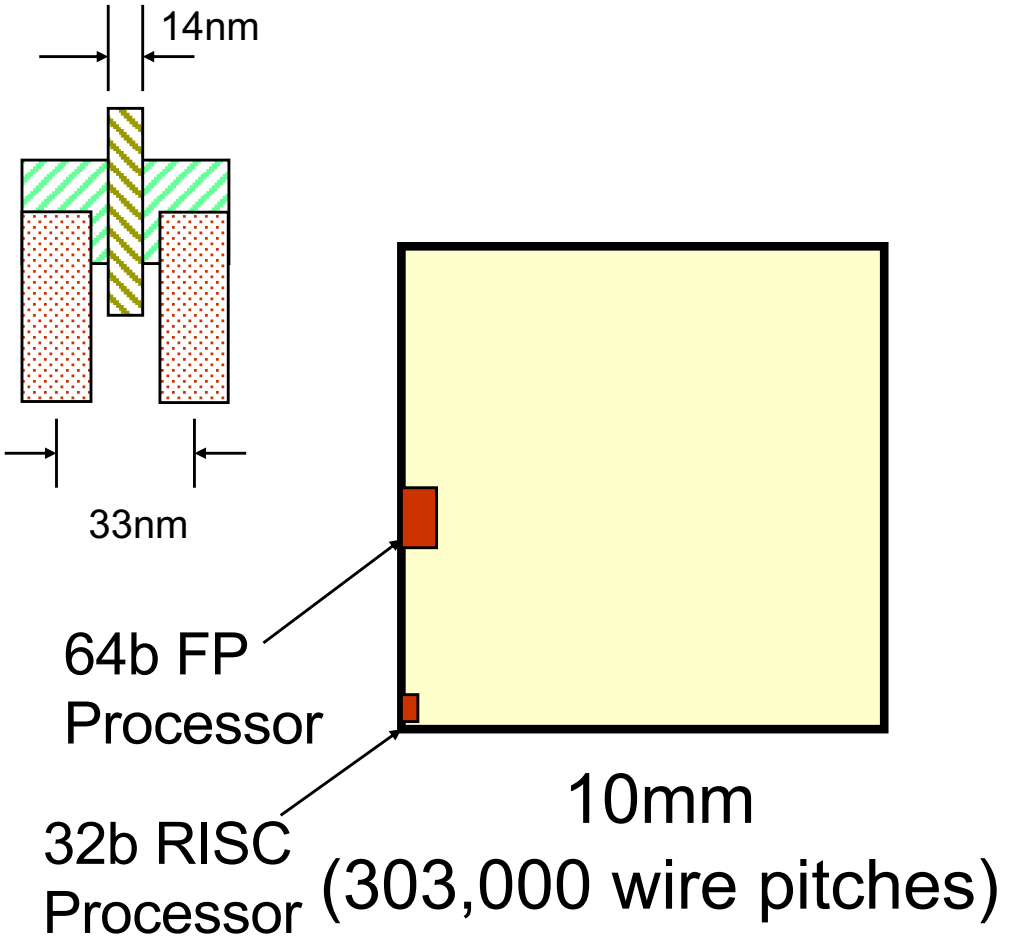
What fits on a chip today?

Mainstream logic chip

- 10mm on a side (100mm^2)
- 14nm drawn gate length
- 33nm wire pitch
- 14 wires levels

For comparison

- 32b RISC integer processor
  - 1K x 2K wire grids
  - **45,000 processors**
- SRAM
  - About 4 x 4 grids / bit
  - **5.5 B SRAM cells**
- DRAM
  - 1 x 2 grids / bit
  - **45 B cells**



Question

Why don’t we implement all the functionality of the iPhone in a single chip?

- Moore’s Law should allow this (sooner or later)

Challenges

1) Complexity

2) Efficiency

The Complexity Challenge

Complexity is the limiting factor in modern chip design

We want to use all the transistors on a chip

- Need HW components for: cellphone, camera, TV, computer, …
- Too many applications to cast all into hardware logic
- Cost: $75M and 1-3 years for a high-end design
- Difficult to verify, fix bugs, upgrade once design is out

Only way to survive:

- Hide complexity using **abstraction**
- Design **reusable** hardware **components**

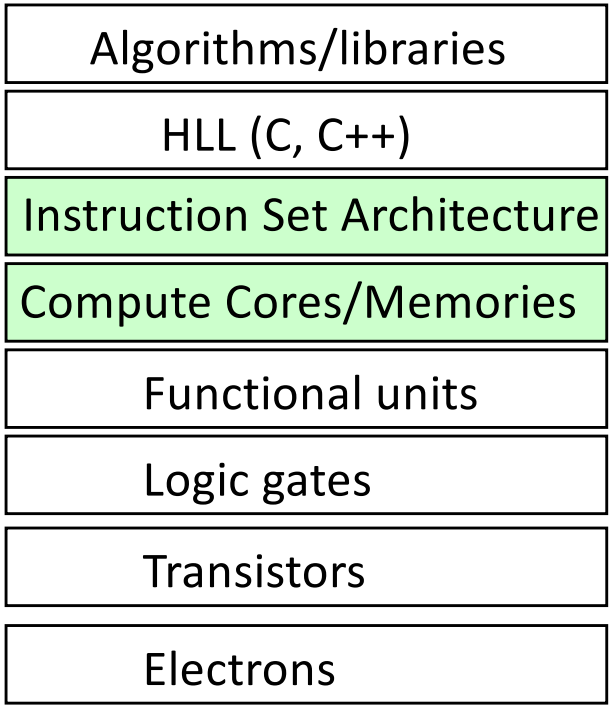
Complexity Management through Abstraction

Key idea

Capture design at different levels of representation

Stable interfaces expose functionality but not low-level implementation details of lower levels

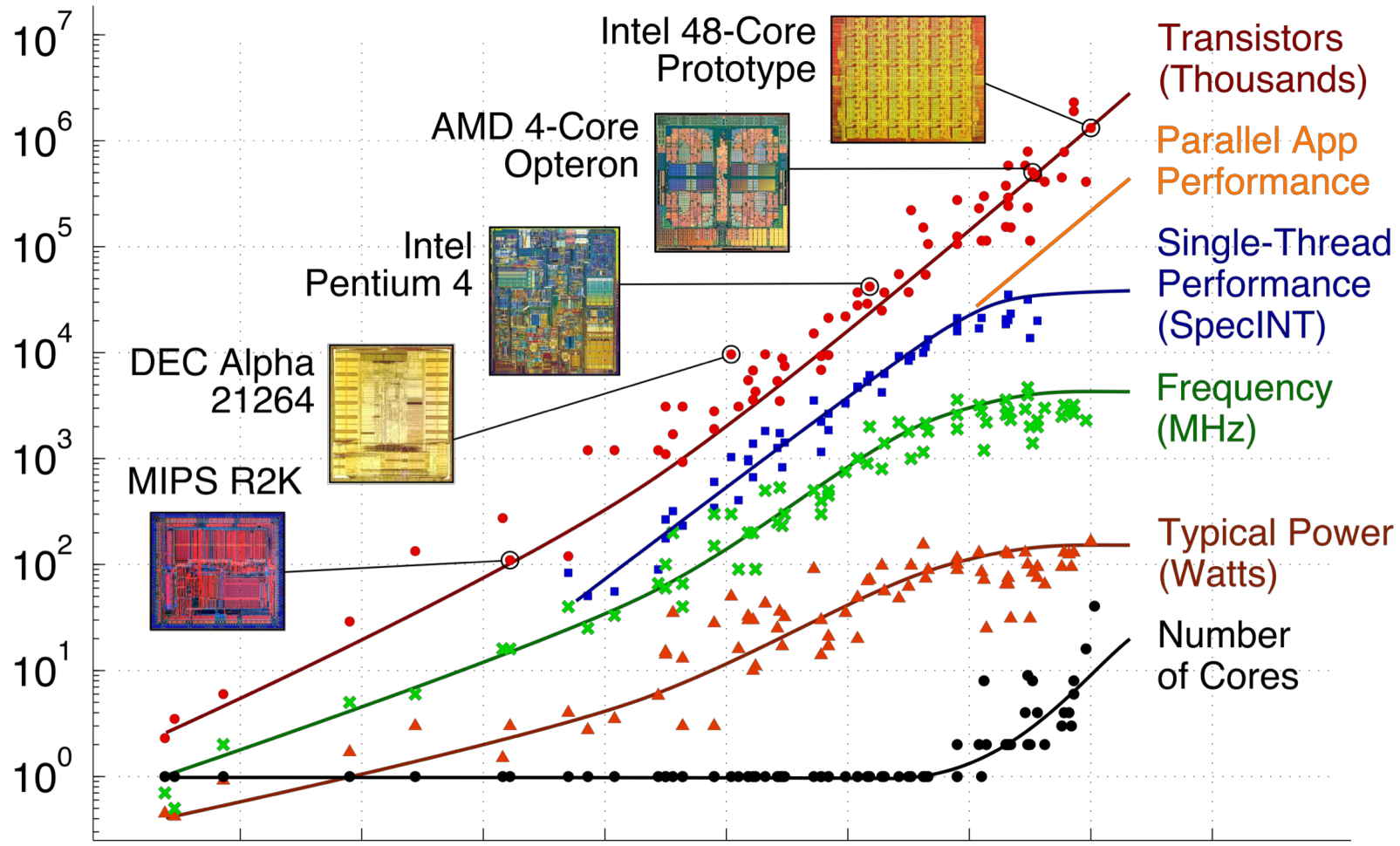
A stable interface allows us to optimize on either side independently of layer below/above



Works well for HW and SW

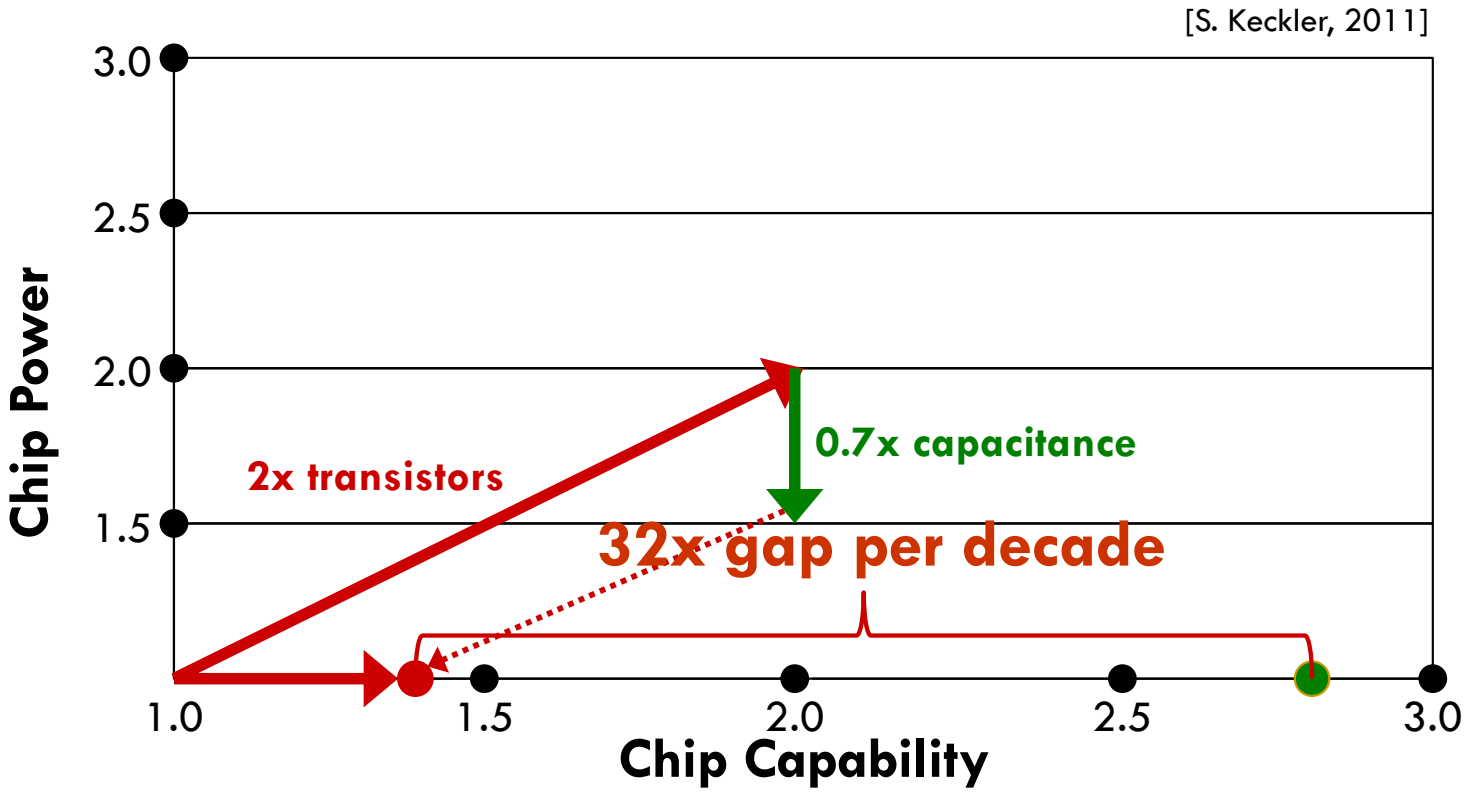
It also allows design of reusable HW and SW

Processor Scaling



Data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten

Technology Scaling: The Present



Moore's Law <u>without</u> Dennard scaling

- 1.4x in chip capability per generation at constant power
- 32x capability gap compared to past scaling

Key Tools for System Architects

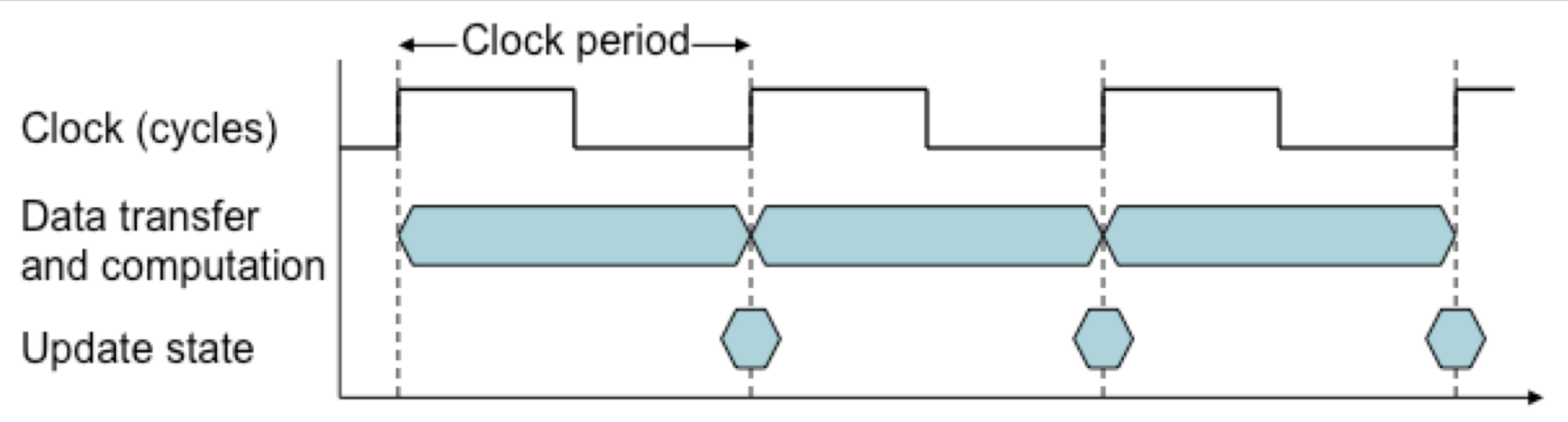
1. **Pipelining**: overlap steps in execution; watch out for dependencies
2. **Parallelism**: execute independent tasks in parallel
3. **Out-of-order execution**: execute task in order of true dependencies
4. **Prediction**: better to ask for forgiveness than permission…
5. **Caching**: keep close a copy of frequently used information
6. **Indirection**: go through a translation step to allow intervention
7. **Amortization**: coarse-grain actions to amortize start/end overheads
8. **Redundancy**: extra information or resources to recover from errors
9. **Specialization**: trim overheads of general-purpose systems
10. **Focus on the common case**: optimize only the critical aspects of the system

Performance

Performance: Latency vs Throughput

- Latency or response/execution time
  - How long it takes to do a task
- Throughput
  - Total work done per unit time (e.g., queries/sec)
- Questions
  - Example of cases where we care about one or the other?
  - Which one improves by speeding up a core?
  - Which one improves by adding more cores?
  - Does improving latency help improve throughput?
  - Does improving throughput help improve latency?

Latency: the Base Units



- Digital HW operates using a constant-rate clock
- Clock period: duration of a clock cycle
  - E.g., 250ps = 0.25ns = 250×10^{-12} s
  - This is the basic unit of time in all computers
- Clock frequency (rate): cycles per second
  - E.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

Execution (or CPU) Time

$$\text{Execution Time} = \text{Cycles Per Program} \cdot \text{Clock Cycle Time}$$

$$= \frac{\textit{CyclesPer}\,\text{Pr}\,\textit{ogram}}{\text{Clock Rate}}$$

- Execution time improved by
  - Reducing number of clock cycles
  - Or Increasing clock rate
- These two goals are not always compatible
  - Must often trade off clock rate against cycle count

Instruction Count & 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 (IC) for a program
  - Determined by program, ISA, and compiler
- Average cycles per instruction (CPI)
  - Determined by HW design
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

Performance Summary

$$\textit{Execution Time} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Clock cycles}}{\text{Instruction}} \cdot \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Must take all 3 factors into account
  - Caveat: we will add OS time and I/O time later on
- Performance depends on
  - Algorithm: affects IC and (possibly) CPI
  - Programming language: affects IC and CPI
  - Compiler: affects IC and CPI
  - Instruction set architecture: affects IC and CPI
  - HW design: affects CPI and Tc

Calculating CPI

- If different instruction types take different numbers of cycles

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

- Weighted average CPI

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

CPI Example

- Two alternative code using instructions types 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

- CPI_1 =

- CPI_2 =

CPI Example

- Two alternative code using instructions types 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

- $CPI_1 = (1*2 + 2*1 + 3*2)/5 = 10/5 = 2$

- $CPI_2 = (1*4 + 2*1 + 3*1)/6 = 9/6 = 1.5$

Relative Performance

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

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

- Example:
  - Program runs for 10s on machine A, for 15s on B
  - Execution TimeB / Execution TimeA = 15s / 10s = 1.5
  - So A is 1.5 times faster than B
  - Or A is 50% faster than B

Performance Example

- Processors A and B for the same ISA
  - A: cycle time= 250ps, CPI = 2.0
  - B: Cycle time= 500ps, CPI = 1.2
- Which is one faster, and by how much?

Performance Example

- Processors A and B for the same ISA
  - A: cycle time= 250ps, CPI = 2.0
  - B: Cycle time= 500ps, CPI = 1.2
- Which is one 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

Average vs. Tail Performance

- Average Request Latency:

$$\frac{1}{R} * \sum^{R} Li$$

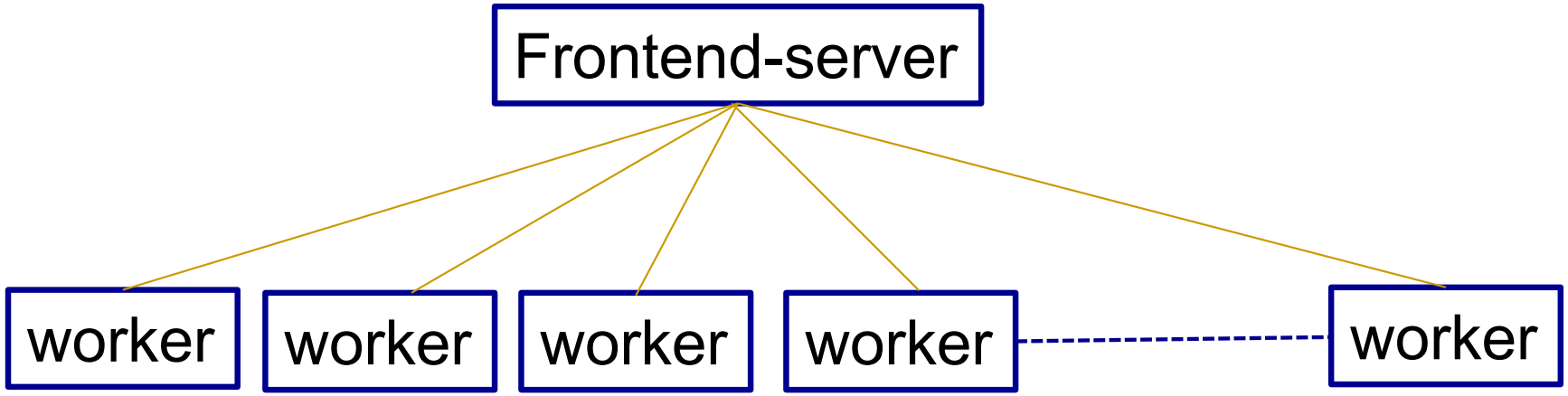
Where Li = Latency of request i,
R = Number of requests

What happens if Li has high variability?

What happens if a request has subrequests?

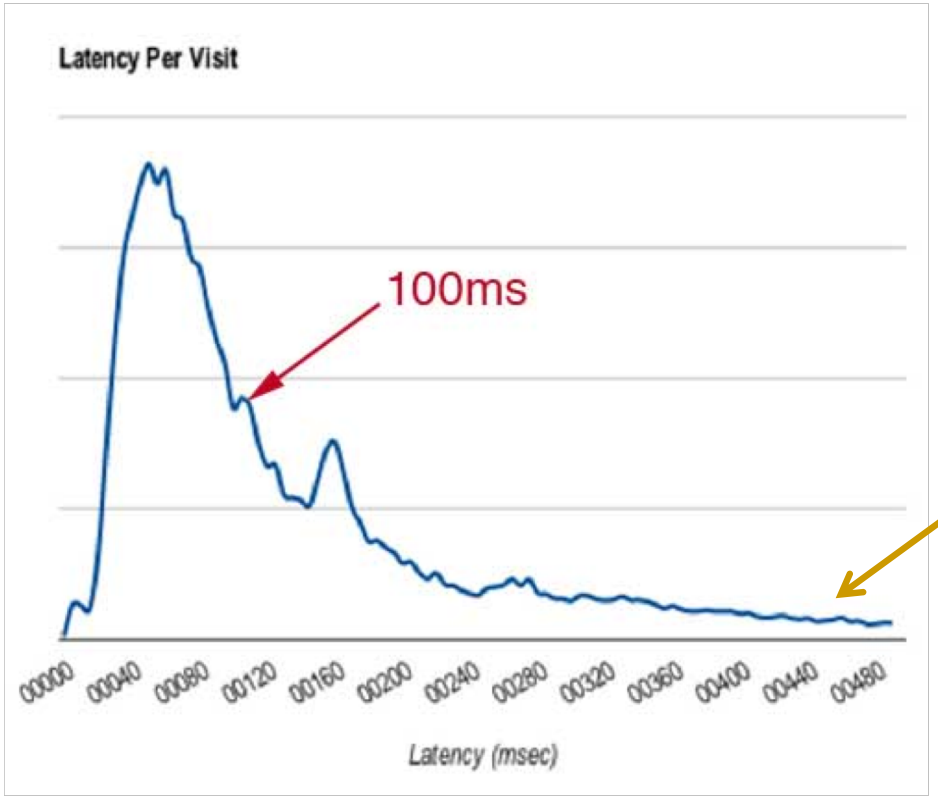
The Tail at Scale

- High fan-out webservices



Request Latency = MAX(Worker Latency)

Latency Distribution



Percentile Latency: 99th, 99.9th, 99.99th

Power and Energy

Why Are Power & Energy Important?

- Power density (cooling)
  - Limits compaction & integration
  - E.g., a cellphone chip cannot exceed 1-2W
- Battery life for mobile devices
- Reliability at high temperatures
- Cost
  - Energy cost
  - Cost of power delivery, cooling system, packaging
- Environmental issues
  - IT responsible for 0.53 billion tons of CO_2 in 2002

Power Consumption in Chips

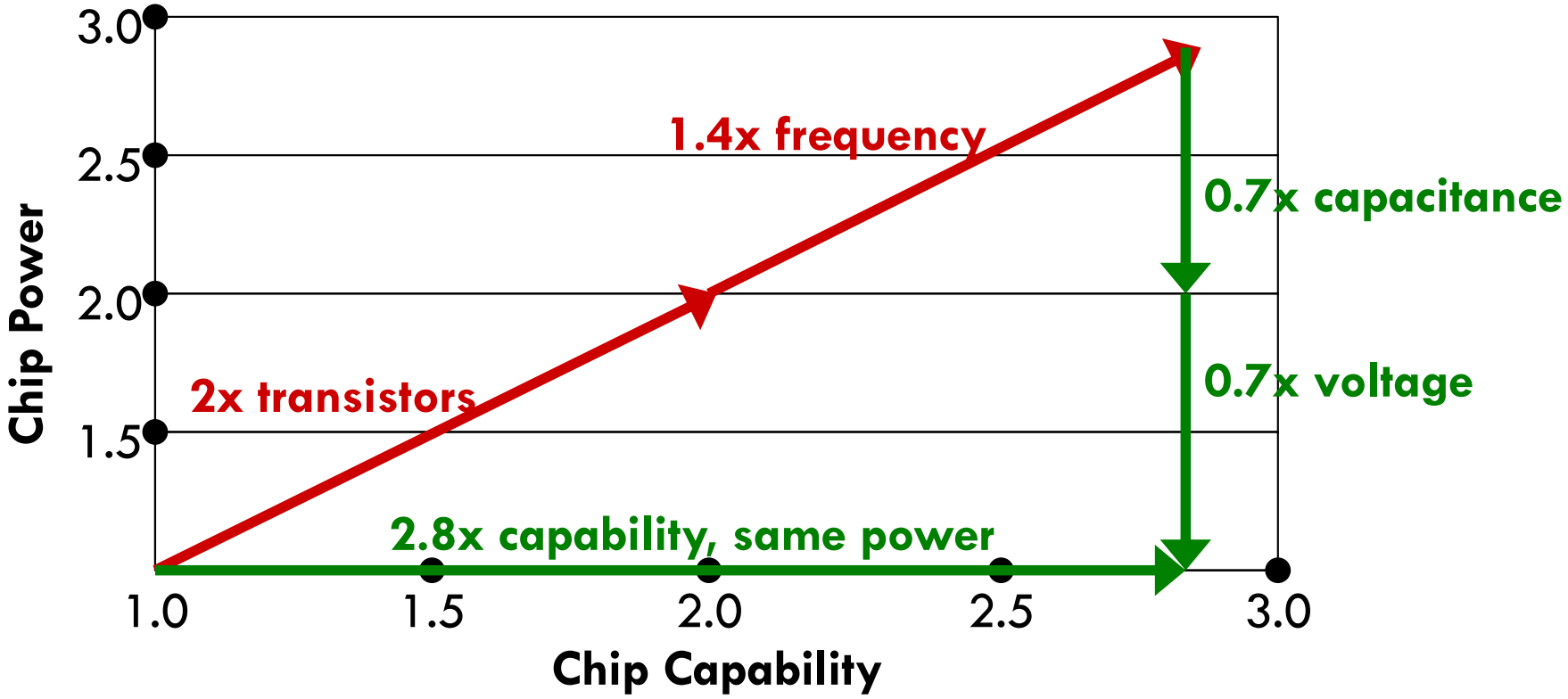
$$\text{Power} = C * Vdd^2 * F_{0\rightarrow1} + Vdd * I_{leakage}$$

- Dynamic or active power consumption
  - Charging and discharging capacitors
  - Depends on switching transistors and switching activity
- Leakage current or static power consumption
  - Leaking diodes and transistors
  - Gets worse with smaller devices and lower Vdd
  - Gets worse with higher temperatures

Energy (≠ Power)

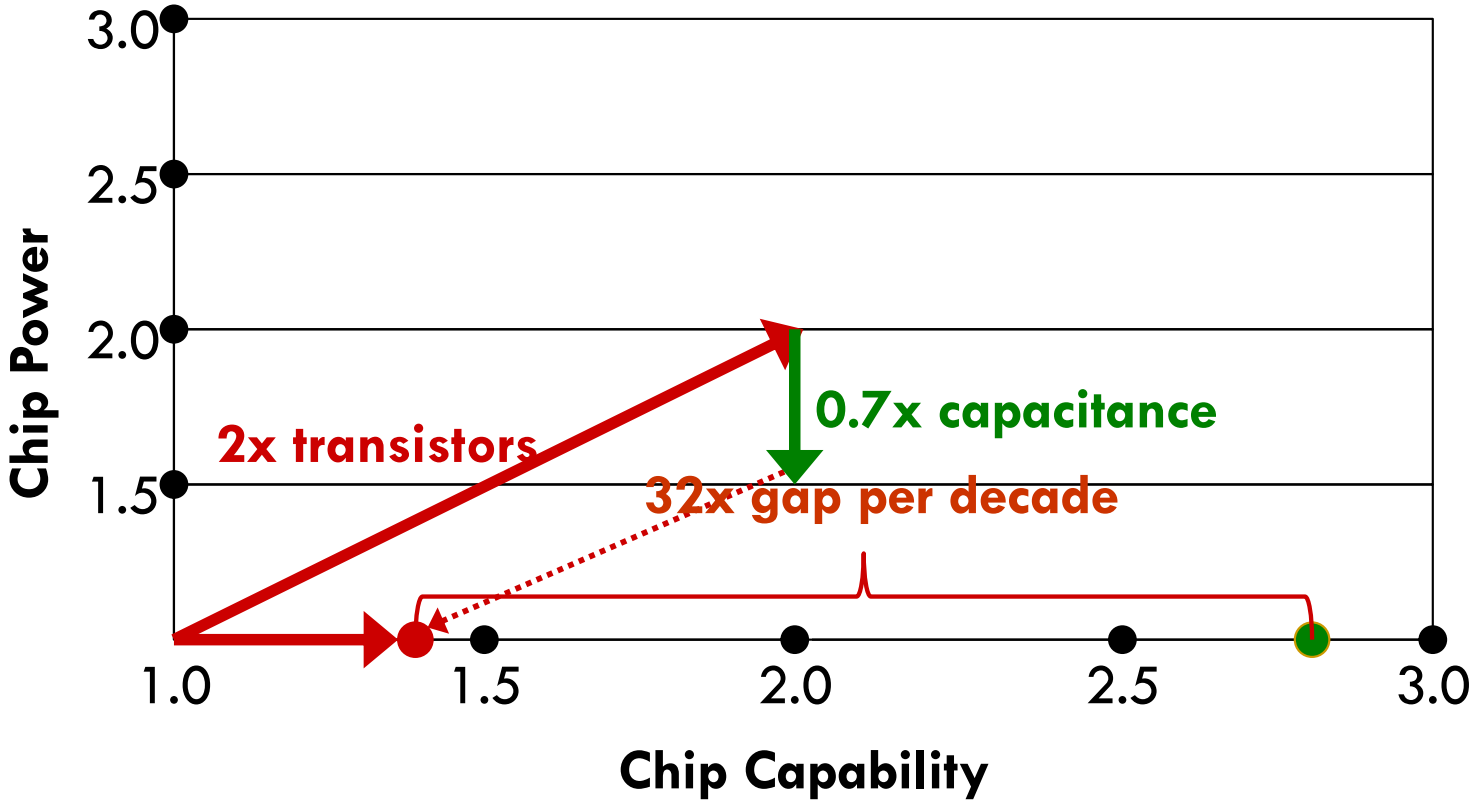
- Energy = Average power * Execution time
  - Joules = Watts * sec
  - Power is limited by infrastructure (e.g., power supply)
  - Energy: what the utilities charge for or battery can store
- You can improve energy by
  - Reducing power consumption
  - Or by improving execution time
- Race to halt!

Semiconductor Scaling: The Past



- Moore's law (more transistors) + Dennard scaling (lower V_{dd})
  - 2.8x in chip capability per CMOS generation at constant power

Semiconductor Scaling: The Present



- Moore's Law without Dennard scaling
  - 1.4x in chip capability per generation at constant power
  - 32x capability gap compared to past scaling

Question

- So, what do we do now?
  - Remember: $\mathbf{Power = C * Vdd^2 * F_{0\rightarrow1} + Vdd * I_{leakage}}$
- What should we optimize processors for?

Question

- So, what do we do now?
  - Remember: **Power = C *Vdd^2*$F_{0\rightarrow1}$ + Vdd*$I_{leakage}$**
- What should we optimize processors for?
  - Answer: energy per instruction (EPI)

$$Power = \frac{energy}{second} = \frac{energy}{instruction} \times \frac{instructions}{second}$$

  - After minimizing EPI, tune performance & power as needed
    - Higher for server, lower for cellphone

Useful Techniques for Evaluating Efficiency

Amdahl's Law: Make Common Case Efficient

- Given an optimization x that accelerates fraction f_x of program by a factor of S_x, how much is the overall speedup?

$$Speedup = \frac{CPUTime_{old}}{CPUTime_{new}} = \frac{CPUTime_{old}}{CPUTime_{old}[(1-f_x)+\frac{f_x}{S_x}]} = \frac{1}{(1-f_x)+\frac{f_x}{S_x}}$$

- Lesson's from Amdahl's law
  - Make **common cases** fast: as $f_x \rightarrow 1$, speedup$\rightarrow S_x$
  - But don't overoptimize common case: as $S_x \rightarrow \infty$, speedup$\rightarrow 1 / (1-f_x)$
    - Speedup is limited by the fraction of the code accelerated
    - Uncommon case will eventually become the common one
- Amdahl's law applies to cost, power consumption, energy …

Benchmarks

- Programs used to measure performance
  - Supposedly typical of actual workload
- Benchmark suite: collection of benchmarks
  - Plus datasets, metrics, and rules for evaluation
  - Plus a way to summarize performance in one number
- Examples
  - SPEC CPU2006 (integer and FP benchmarks)
  - TPC-H and TCP-W (database benchmarks)
  - EEMBC (embedded benchmarks)
- Warning
  - Different benchmarks focus on different workloads
  - All benchmarks have shortcomings
  - Your design will be as good as the benchmarks you use

Example: 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

Summarizing Benchmarks

Arithmetic mean $\frac{1}{n}\sum_{i=1}^{n} T_i$ — Use with times, not with rates

Represents total execution time

Harmonic mean $\frac{n}{\sum_{i=1}^{n} \frac{1}{R_i}}$ — Use with rates not with times

Geometric mean $\left(\prod_{i=1}^{n} \frac{T_i}{T_{ri}}\right)^{\frac{1}{n}} = \exp\left(\frac{1}{n}\sum_{i=1}^{n} \log\left(\frac{T_i}{T_{ri}}\right)\right)$ — Good with normalized performance

Does not represent total execution time

- Can also use weighted version
- Be careful which one you use!

Example: SPECpower_ssj2008 for Xeon X5650

- Power/performance benchmark

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

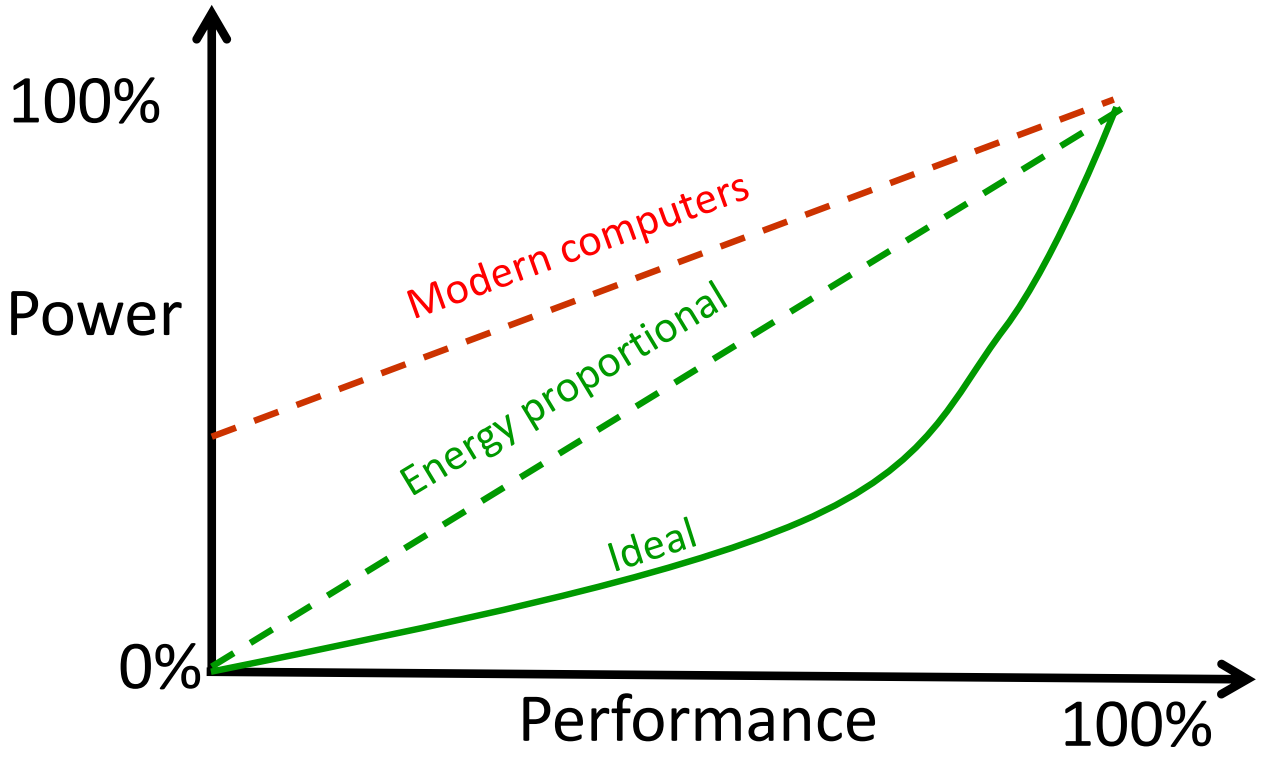
Target Load %	Performance (ssj_ops)	Average Power (Watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1,922
Σssj_ops/Σpower =		2,490

A Closer Look

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

- How does power scale with load?
- Is that good? Why?

Energy Proportionality



- Power consumption should scale with performance
- But most modern computers don't (why?)