Review and Fundamentals

Nima Honarmand

Measuring and Reporting Performance

Performance Metrics

- *Latency* (execution/response time): time to finish one task
- *Throughput* (bandwidth): number of tasks/unit time
  - Throughput can exploit parallelism, latency can't
  - Sometimes complimentary, often contradictory

- Example: move people from A to B, 10 miles
  - Car: capacity = 5, speed = 60 miles/hour
  - Bus: capacity = 60, speed = 20 miles/hour
  - Latency: car = 10 min, bus = 30 min
  - Throughput: car = 15 PPH (w/ return trip), bus = 60 PPH

No right answer: pick metric for ***your*** goals

Performance Comparison

- Processor A is X times faster than processor B if
  - Latency(P, A) = Latency(P, B) / X
  - Throughput(P, A) = Throughput(P, B) * X
- Processor A is X% faster than processor B if
  - Latency(P, A) = Latency(P, B) / (1+X/100)
  - Throughput(P, A) = Throughput(P, B) * (1+X/100)
- Car/bus example
  - Latency? Car is 3 times (200%) faster than bus
  - Throughput? Bus is 4 times (300%) faster than car

Latency/throughput of What Program?

- *Very difficult question!*
- Best case: you always run the same set of programs
  - Just measure the execution time of those programs
  - Too idealistic
- Use ***benchmarks***
  - *Representative* programs chosen to measure performance
  - (Hopefully) predict performance of actual workload
  - Prone to Benchmarketing:

> *"The misleading use of unrepresentative benchmark software results in marketing a computer system"*
>
> -- *wikitionary.com*

Types of Benchmarks

- Real programs
  - Example: CAD, text processing, business apps, scientific apps
  - Need to know program inputs and options (not just code)
  - May not know what programs users will run
  - Require a lot of effort to port
- Kernels
  - Small key pieces (inner loops) of scientific programs where program spends most of its time
  - Example: Livermore loops, LINPACK
- Toy Benchmarks
  - e.g. Quicksort, Puzzle
  - Easy to type, predictable results, may use to check correctness of machine but not as performance benchmark.

SPEC Benchmarks

- System Performance Evaluation Corporation

  "non-profit corporation formed to establish, maintain and endorse a standardized set of relevant benchmarks …"

- Different set of benchmarks for different domains:
  - CPU performance (SPEC CINT and SPEC CFP)
  - High Performance Computing (SPEC MPI, SPC OpenMP)
  - Java Client Server (SPECjAppServer, SPECjbb, SPECjEnterprise, SPECjvm)
  - Web Servers
  - Virtualization
  - …

Example: SPEC CINT2006

Program	Language	Description
400.perlbench	C	Programming Language
401.bzip2	C	Compression
403.gcc	C	C Compiler
429.mcf	C	Combinatorial Optimization
445.gobmk	C	Artificial Intelligence: Go
456.hmmer	C	Search Gene Sequence
458.sjeng	C	Artificial Intelligence: chess
462.libquantum	C	Physics / Quantum Computing
464.h264ref	C	Video Compression
471.omnetpp	C++	Discrete Event Simulation
473.astar	C++	Path-finding Algorithms
483.xalancbmk	C++	XML Processing

Example: SPEC CFP2006

Program	Language	Description
410.bwaves	Fortran	Fluid Dynamics
416.gamess	Fortran	Quantum Chemistry.
433.milc	C	Physics / Quantum Chromodynamics
434.zeusmp	Fortran	Physics / CFD
435.gromacs	C, Fortran	Biochemistry / Molecular Dynamics
436.cactusADM	C, Fortran	Physics / General Relativity
437.leslie3d	Fortran	Fluid Dynamics
444.namd	C++	Biology / Molecular Dynamics
447.dealII	C++	Finite Element Analysis
450.soplex	C++	Linear Programming, Optimization
453.povray	C++	Image Ray-tracing
454.calculix	C, Fortran	Structural Mechanics
459.GemsFDTD	Fortran	Computational Electromagnetics
465.tonto	Fortran	Quantum Chemistry
470.lbm	C	Fluid Dynamics
481.wrf	C, Fortran	Weather
482.sphinx3	C	Speech recognition

Benchmark Pitfalls

- Benchmark not representative
  - Your workload is I/O bound → SPECint is useless
- Benchmark is too old
  - Benchmarks age poorly
  - Benchmarketing pressure causes vendors to optimize compiler/hardware/software to benchmarks

  → Need to be periodically refreshed

Summarizing Performance Numbers

- Latency is additive, throughput is not
  - Latency(P1+P2, A) = Latency(P1, A) + Latency(P2, A)
  - Throughput(P1+P2, A) **!=** Throughput(P1, A) + Throughput(P2,A)
- Example:
  - 180 miles @ 30 miles/hour + 180 miles @ 90 miles/hour
  - 6 hours at 30 miles/hour + 2 hours at 90 miles/hour
    - Total latency is 6 + 2 = 8 hours
    - Total throughput is ***not 60*** miles/hour
      - Total throughput is ***only 45*** miles/hour! (360 miles / (6 + 2 hours))

Arithmetic Mean is Not Always the Answer!

Summarizing Performance Numbers

- *Arithmetic*: times
  - proportional to time
  - e.g., latency

$$\frac{1}{n}\sum_{i=1}^{n} Time_i$$

- *Harmonic*: rates
  - inversely proportional to time
  - e.g., throughput

$$\frac{n}{\sum_{i=1}^{n} \frac{1}{Rate_i}}$$

- *Geometric*: ratios
  - unit-less quantities
  - e.g., speedups & normalized times

$$\sqrt[n]{\prod_{i=1}^{n} Ratio_i}$$

Used by SPEC CPU

- Any of these can be ***weighted***

Memorize these to avoid looking them up later

Improving Performance

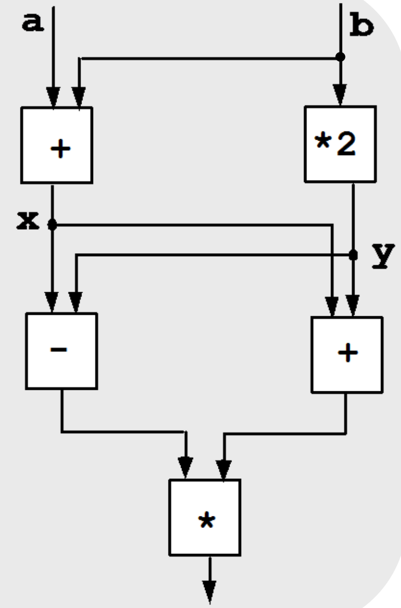
Principles of Computer Design

- Take Advantage of Parallelism
  - E.g., multiple processors, disks, memory banks, pipelining, multiple functional units
  - *Speculate* to create (even more) parallelism
- Principle of Locality
  - Reuse of data and instructions
- Focus on the Common Case
  - Amdahl’s Law

Parallelism: Work and Critical Path

- *Parallelism*: number of independent tasks available
- *Work* (T_1): time on sequential system
- *Critical Path* (T_∞): time on infinitely-parallel system

```
x = a + b;
y = b * 2
z =(x-y) * (x+y)
```



- *Average Parallelism*:
  $P_{avg} = T_1 / T_\infty$
- For a p-wide system:
  $T_p \geq \max\{ T_1/p, T_\infty \}$
  $P_{avg} >> p \Rightarrow T_p \approx T_1/p$

Principle of Locality

- Recent past is a good indication of near future

Temporal Locality: If you looked something up, it is very likely that you will look it up again soon

Spatial Locality: If you looked something up, it is very likely you will look up something nearby soon

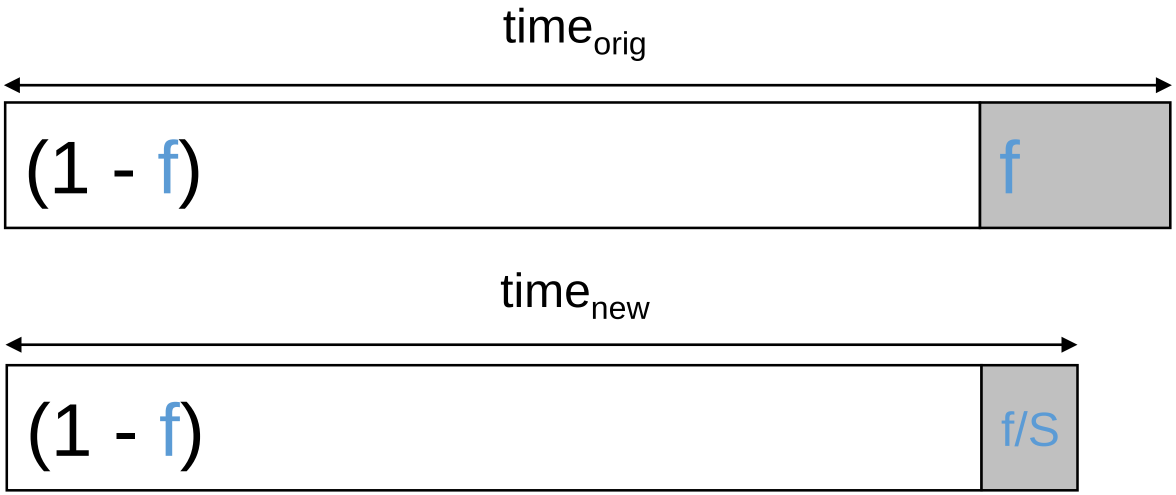
Amdahl's Law

Speedup = $\text{time}_{\text{without enhancement}} / \text{time}_{\text{with enhancement}}$

An enhancement speeds up fraction f of a task by factor S

$$\text{time}_{\text{new}} = \text{time}_{\text{orig}} \cdot ((1-f) + f/S)$$

$$S_{\text{overall}} = 1 / ((1-f) + f/S)$$



$\text{time}_{\text{orig}}$

(1 - f)

f

time_{new}

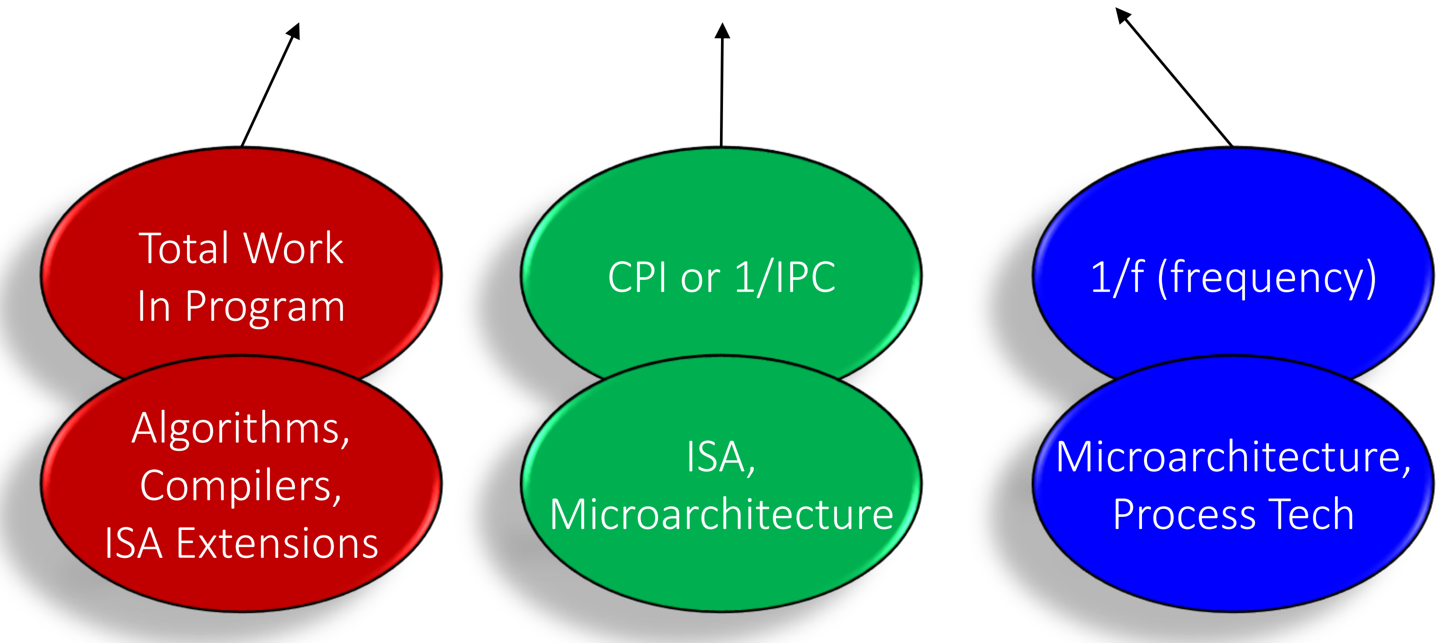
(1 - f)

f/S

Make the common case fast!

The *Iron Law* of Processor Performance

$$\frac{Time}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Time}{Cycle}$$



Total Work In Program

Algorithms, Compilers, ISA Extensions

CPI or 1/IPC

ISA, Microarchitecture

1/f (frequency)

Microarchitecture, Process Tech

Architects target CPI, but ***must*** understand the others

Another View of CPU Performance

- Instruction frequencies for a load/store machine

Instruction Type	Frequency	Cycles
Load	25%	2
Store	15%	2
Branch	20%	2
ALU	40%	1

- What is the average CPI of this machine?

$$\text{Average CPI} = \frac{\sum_{i=1}^{n} InstFrequency_i \times CPI_i}{\sum_{i=1}^{n} InstFrequency_i}$$

$$= \frac{0.25\times 2 + 0.15\times 2 + 0.2\times 2 + 0.4\times 1}{1} = 1.6$$

Another View of CPU Performance

- Assume all conditional branches in this machine use simple tests of equality with zero (BEQZ, BNEZ)
- Consider adding complex comparisons to conditional branches
  - 25% of branches can use complex scheme → no need for preceding ALU instruction
- The CPU cycle time of original machine is 10% faster
- Will this increase CPU performance?

$$\text{New CPU CPI} = \frac{0.25\times 2 + 0.15\times 2 + 0.2\times 2 + (0.4 - 0.25\times 0.2)\times 1}{1 - 0.25\times 0.2} = 1.63$$

Hmm… Both slower clock and increased CPI?
Something smells fishy !!!

Another View of CPU Performance

- Recall the Iron Law
- The two programs have different number of instructions

Old CPU Time = $InstCount_{old} \times CPI_{old} \times cycle\_time_{old} = N \times 1.6 \times ct$

New CPU Time =

$$InstCount_{new} \times CPI_{new} \times cycle\_time_{new} = (1 - 0.25 \times 0.2) N \times 1.63 \times 1.1ct$$

Speedup = $\dfrac{1.6}{(1 - 0.25 \times 0.2) \times 1.63 \times 1.1} = 0.94$

The new CPU is ***slower*** for this instruction mix

Partial Performance Metrics Pitfalls

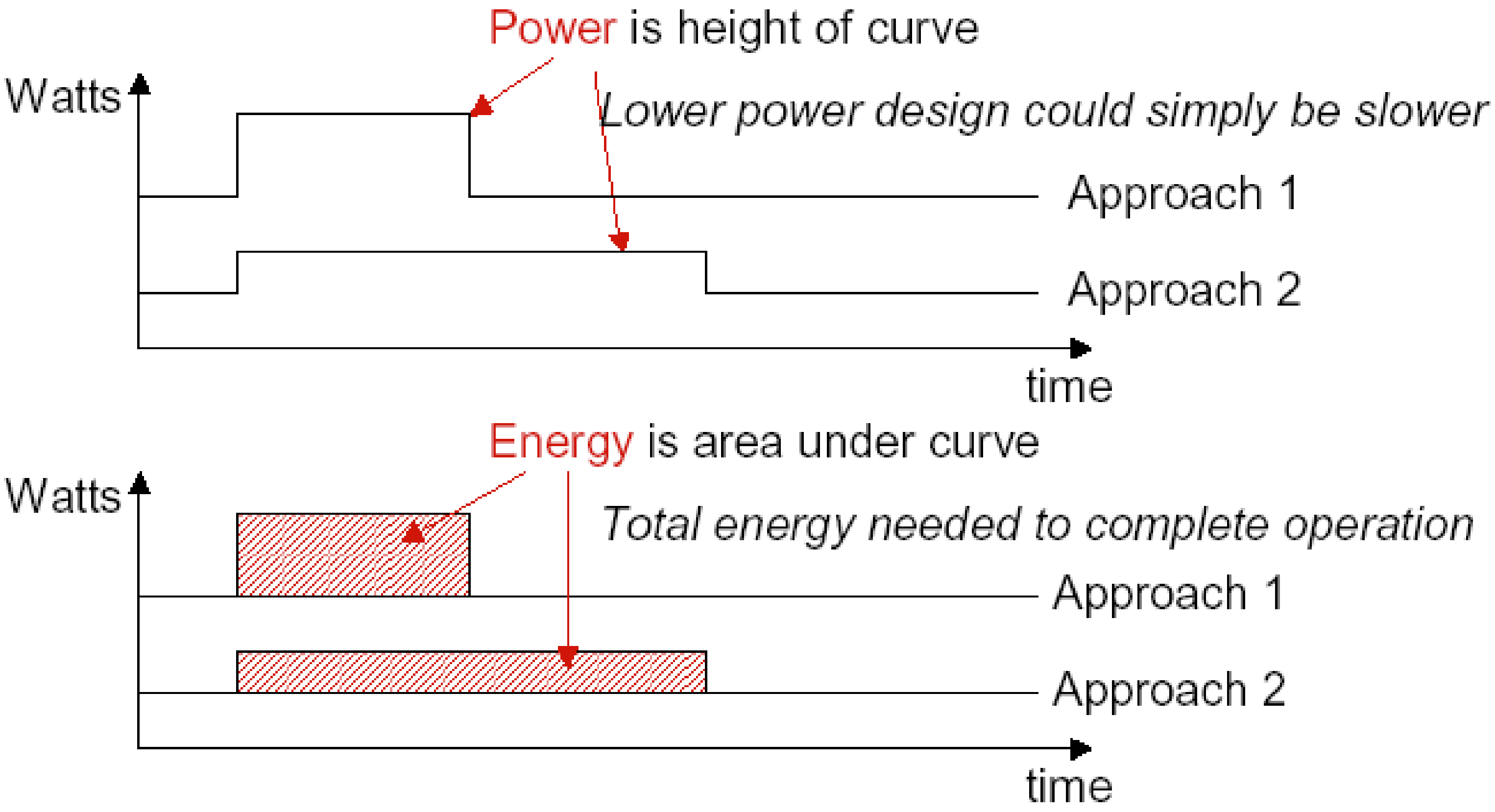
- Which processor would you buy?
  - Processor A: CPI = 2, clock = 2.8 GHz
  - Processor B: CPI = 1, clock = 1.8 GHz
  - Probably A, but B is faster (assuming same ISA/compiler)
- Classic example
  - 800 MHz Pentium III faster than 1 GHz Pentium 4
  - Same ISA and compiler
- Some Famous Partial Performance Metrics
  - MIPS: Million Instruction Per Second
  - MFLOPS: Million Floating-Point Operations Per Second

Power

Power vs. Energy (1/2)

- *Energy*: capacity to do work or amount of work done
  - Expressed in joules
  - Energy(OP1+OP2)=Energy(OP1)+Energy(OP2)
- *Power*: instantaneous rate of energy transfer
  - Expressed in watts
  - energy / time (watts = joules / seconds)
  - Power(Comp1+Comp2)=Power(Comp1)+Power(Comp2)
- In processors, all consumed energy is converted to heat
  - Hence: power also equals rate of heat generation

Power vs. Energy (2/2)



Power is height of curve

Lower power design could simply be slower

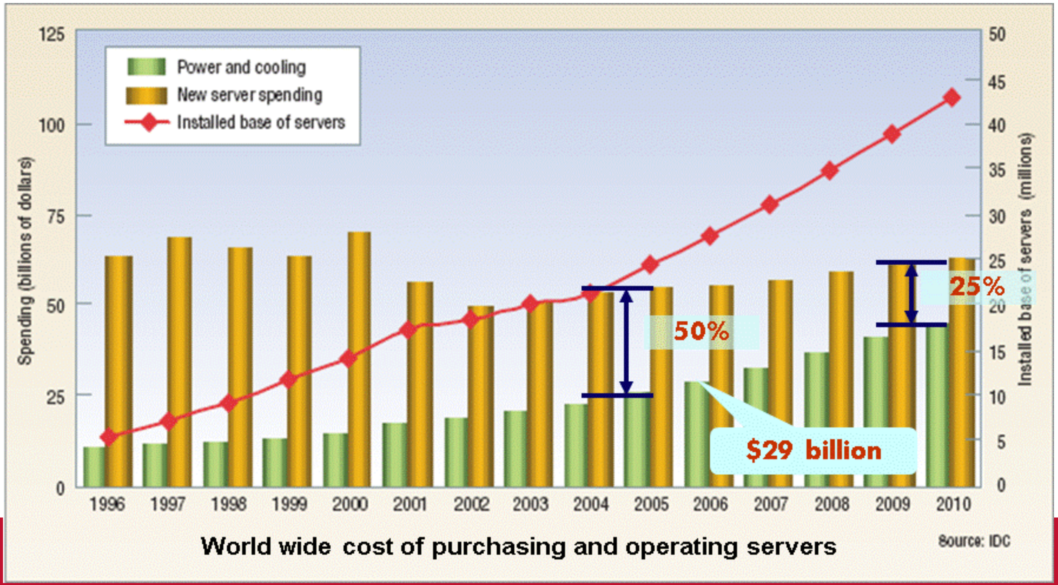
Energy is area under curve

Total energy needed to complete operation

Does this example help or hurt?

Why is Energy Important?

- Impacts battery life for mobile
- Impacts electricity costs for tethered (plugged)
  - You have to buy electricity
    - It costs to produce and deliver electricity
  - You have to remove generated heat
    - It costs to buy and operate cooling systems
- Gets worse with larger data centers
  - $7M for 1000 server racks
  - 2% of US electricity used by DCs in 2010 (Koomey'11)



World wide cost of purchasing and operating servers

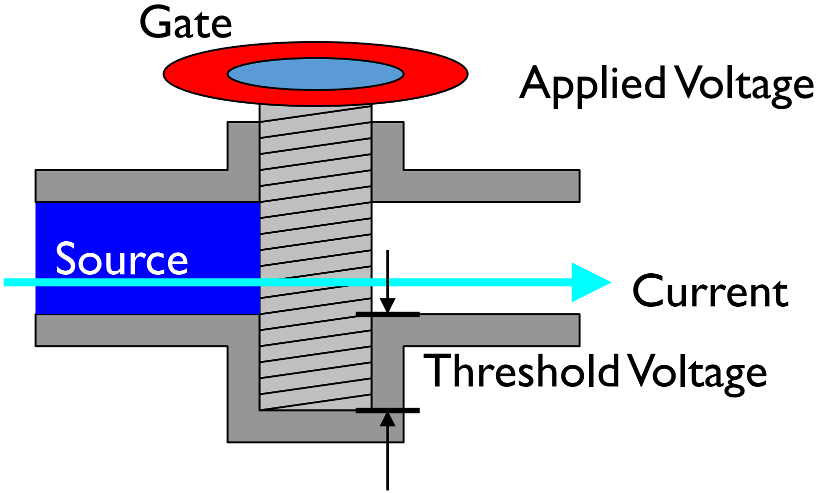
Source: IDC

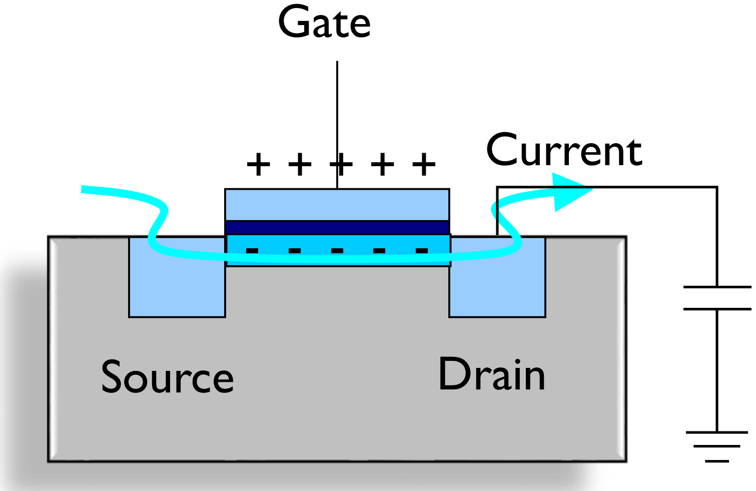
Why is Power Important?

- Because power has a peak
- Power is also heat generation rate
  - Must dissipate the heat
  - Need heat sinks and fans and …
- What if fans not fast enough?
  - Chip powers off (if it's smart enough)
  - Melts otherwise
- Thermal failures even when fans OK
  - 50% server reliability degradation for +10°C
  - 50% decrease in hard disk lifetime for +15°C

Power: The Basics (1/2)

- **Dynamic Power**
  - Related to switching activity of transistors (from 0→1 and 1→0)

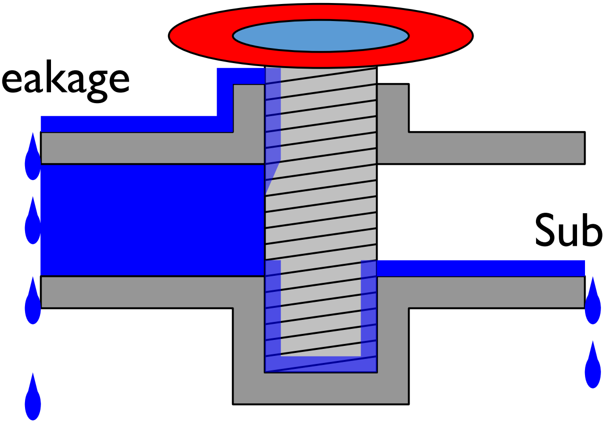




- Dynamic Power ∝ $CV_{dd}^{2}Af$
  - C: capacitance, function of transistor size and wire length
  - V_{dd}: Supply voltage
  - A: Activity factor (average fraction of transistors switching)
  - f: clock frequency
  - About 50-70% of processor power

Power: The Basics (2/2)

- **Static Power**
  - Current leaking from a transistor even if doing nothing (steady, constant energy cost)



- Static Power $\propto\ V_{dd}$ and $\propto e^{-c_1 V_{th}}$ and $\propto\ e^{c_2 T}$
  - This is a first-order model
  - c_1, c_2 : some positive constants
  - V_{th}: Threshold Voltage
  - T: Temperature
  - About 30-50% of processor power

Thermal Runaway

- Leakage is an exponential function of temperature
- ↑ Temp leads to ↑ Leakage
- Which burns more power
- Which leads to ↑ Temp, which leads to…

Positive feedback loop will melt your chip

Why Power Became an Issue? (1/2)

- Good old days of ideal scaling (aka Dennard scaling)
  - Every new semiconductor generation:
    - Transistor dimension: x 0.7
    - Transistor area: x 0.49
    - C and V_{dd}: x 0.7
    - Frequency: 1 / 0.7 = 1.4

  →Constant dynamic power density
  - In those good old days, leakage was not a big deal

→ Faster and more transistors with constant power density ☺

Why Power Became an Issue? (2/2)

- Recent reality: V_{dd} does not decrease much
  - Switching speed is roughly proportional to V_{dd} - V_{th}
    - If too close to threshold voltage (V_{th}) → slow transistor
    - Fast transistor & low V_{dd} → low V_{th} → exponential increase in leakage ✘

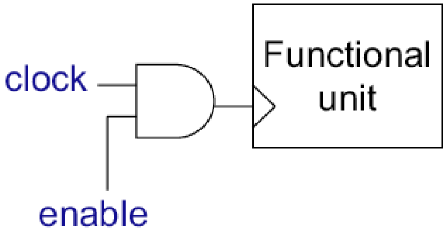
  →Dynamic power density keeps increasing
  - Leakage power has also become a big deal today
    - Due to lower Vth, smaller transistors, higher temperatures, etc.

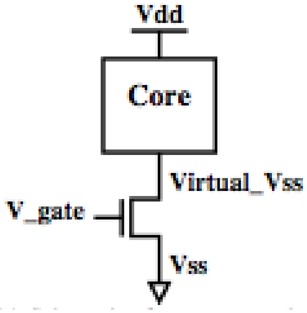
→ We hit the power wall ☹

- Example: power consumption in Intel processors
  - Intel 80386 consumed ~ 2 W
  - 3.3 GHz Intel Core i7 consumes ~ 130 W
  - Heat must be dissipated from 1.5 x 1.5 cm^2 chip
  - This is the limit of what can be cooled by air

How to Reduce Processor Power? (1/3)

- *Clock gating*
  - Stop switching in unused components
  - Done automatically in most designs
  - Near instantaneous on/off behavior





- *Power gating*
  - Turn off power to unused cores/caches
  - High latency for on/off
    - Saving SW state, flushing dirty cache lines, turning off clock tree
    - Carefully done to avoid voltage spikes or memory bottlenecks
  - Issue: Area & power consumption of power gate
  - Opportunity: use thermal headroom for other cores

How to Reduce Processor Power? (2/3)

- Reduce Voltage (V): quadratic effect on dyn. power
  - Negative (~linear) effect on frequency
- Dynamic Voltage/Frequency Scaling (DVFS): set frequency to the lowest needed
  - Execution time = IC * CPI * f
- Scale back V to lowest for that frequency
  - Lower voltage → slower transistors
  - Dyn. Power ≈ C * V^2 * F

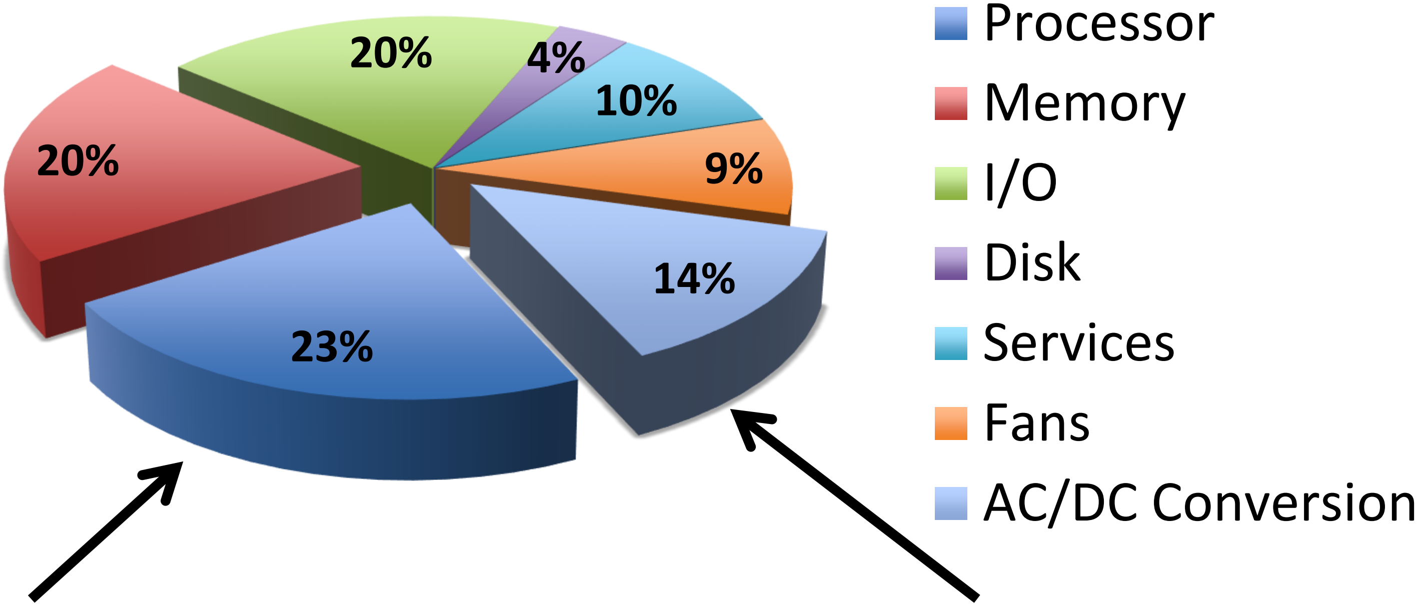
Not Enough! Need Much More!

How to Reduce Processor Power? (3/3)

- Design for E & P efficiency rather than speed
- New architectural designs:
  - Simplify the processor, shallow pipeline, less speculation
  - Efficient support for high concurrency (think GPUs)
  - Augment processing nodes with accelerators
  - New memory architectures and layouts
  - Data transfer minimization
  - …
- New technologies:
  - Low supply voltage (V_{dd}) operation: Near-Threshold Voltage Computing
  - Non-volatile memory (Resistive memory, STTRAM, …)
  - 3D die stacking
  - Efficient on-chip voltage conversion
  - Photonic interconnects
  - …

Processor Is Not Alone

SunFire T2000



< ¼ System Power

> ½ CPU Power

No single component dominates power consumption

Need whole-system approaches to save energy

Instruction Set Architecture (ISA)

ISA: A Contract Between HW and SW

- *ISA*: Instruction Set Architecture
  - A well-defined hardware/software interface
  - Old days: target language for human programmers
  - More recently: target language for compilers
- The "contract" between software and hardware
  - Functional definition of operations supported by hardware
  - Precise description of how to invoke all features
- No guarantees regarding
  - How operations are implemented
  - Which operations are fast and which are slow (and when)
  - Which operations take more energy (and which take less)

Components of an ISA (1/2)

- Programmer-visible machine states
  - Program counter, general purpose registers, control registers, etc.
  - Memory
  - Page table, interrupt descriptor table, etc.
- Programmer-visible operations
  - Operations: ALU ops, floating-point ops, control-flow ops, string ops, etc.
  - Type and size of operands for each op: byte, half-word, word, double word, single precision, double precision, etc.
- Addressing modes for each operand of an instruction
  - Immediate mode (for immediate operands)
  - Register addressing modes: stack-based, accumulator-based, general-purpose registers, etc.
  - Memory addressing modes: displacement, register indirect, indexed, direct, memory-indirect, auto-increment(decrement), scaled, etc.

ISAs last forever, don’t add stuff you don’t need

Components of an ISA (2/2)

- Programmer-visible behaviors
  - What to do, when to do it
- A binary encoding

```
if imem[rip]==“add rd, rs, rt”
then
    rip ⇐ rip+1
    gpr[rd]=gpr[rs]+gpr[rt]
```

Example “register-transfer-level” description of an instruction

ISAs last forever, don’t add stuff you don’t need

RISC vs. CISC

- Recall Iron Law:
  - (instructions/program) * (cycles/instruction) * (seconds/cycle)

- *CISC* (Complex Instruction Set Computing)
  - Improve "instructions/program" with "complex" instructions
  - Easy for assembly-level programmers, good code density

- *RISC* (Reduced Instruction Set Computing)
  - Improve "cycles/instruction" with many single-cycle instructions
  - Increases "instruction/program", but hopefully not as much
    - Help from smart compiler
  - Perhaps improve clock cycle time (seconds/cycle)
    - via aggressive implementation allowed by simpler instructions

Today's x86 chips translate CISC into ~RISC

RISC ISA

- Focus on simple instructions
  - Easy to use for compilers
    - Simple (basic) operations, many registers
  - Easy to design high-performance implementations
    - Easy to fetch and decode, simpler pipeline control, faster caches
- Fixed-length
  - MIPS and SPARCv8 all insts are 32-bits/4 bytes
  - Especially useful when decoding multiple instruction simultaneously
- Few instruction formats
  - MIPS has 3: R (reg, reg, reg), I (reg, reg, imm), J (addr)
  - Alpha has 5: Operate, Op w/ Imm, Mem, Branch, FP
- Regularity across formats (when possible/practical)
  - MIPS & Alpha opcode in same bit-position for all formats
  - MIPS rs & rt fields in same bit-position for R and I formats
  - Alpha ra/fa field in same bit-position for all 5 formats

CISC ISA

- Focus on max expressiveness per min space
  - Designed in era with fewer transistors
  - Each memory access very expensive
    - Pack as much work into as few bytes as possible
- Difficult to use for compilers
  - Complex instructions are not compiler friendly → many instructions remain unused
  - Fewer registers: register IDs take space in instructions
  - For fun: compare x86 vs. MIPS backend in LLVM
- Difficult to build high-performance processor pipelines
  - Difficult to decode: Variable length (1-18 bytes in x86), many formats
  - Complex pipeline control logic
  - Deeper pipelines
- Modern x86 processors translate CISC code to RISC first
  - Called "μ-ops" by Intel and "ROPs" (RISC-ops) by AMD
  - And then execute the RISC code