

Trends in cost

- Learning curve : products drop in cost over time
 - Minor improvements in design & manufacturing
 - Amortization of startup costs (R&D, etc.)
- Volume decreases per-unit cost
 - Fixed costs amortized over more units
 - May have more competitors to help keep prices down
 - Commodities: standardized components available from many vendors
 - Little or no profit margin for commodities
 - Often *much* less expensive than low-volume components
- Relative prices of components can change...

Where does the money go?

- Component costs: raw material costs.
- Direct costs: costs incurred to actually make a single item (usually 20% to 40% of component costs)
- Gross margin: overhead not associated with a single item — R&D, plant equipment, profit, taxes, etc. (typically 20% to 55% of average selling price)
- Average selling price: direct cost + gross margin.
- List price: ASP + reseller's margin

Design tradeoff examples

- Designing for low cost
 - Make choices that minimize cost no matter what
 - Design for ease of manufacture as well as low component cost
- Designing for high performance
 - Spend more money on R&D
 - Worry less about how much it costs to manufacture
- Designing for good cost/performance
 - Use more expensive parts if they have “bang for the buck”
 - Control costs, but not if they lower performance
- Cray vs. desktop workstation vs. Palm III



Chip costs

- Chip cost = $\frac{(\text{die cost} + \text{testing cost} + \text{packaging cost})}{\text{final test yield}}$
- Die cost = $\frac{\text{wafer cost}}{\text{dies/wafer} \times \text{die yield}}$
- Bigger dies => higher cost (dies/wafer decreases)
- How does die size affect yield?



Die yields

$$\text{die yield} = \text{wafer yield} \times \left(1 + \frac{\text{defects per unit area} \times \text{die size}^{-\alpha}}{\alpha} \right)$$

- “Wafer yield” accounts for wafers that are completely bad.
- Model assumes randomly distributed defects
 - 1995: 0.6 - 1.2 per cm²
 - Learning curve reduces this value over time
- Alpha corresponds to the complexity of the manufacturing process
 - Roughly equal to the number of masking levels
 - Approximately 3 today

IC cost : the bottom line

- # of good dies/wafer = dies/wafer * die yield.
 - Larger chips => fewer dies/wafer
 - More complex fab process => lower yield
 - Not a linear relationship
- Die cost is proportional to area⁴ for $\alpha = 3.0$
 - Large area => very expensive dies!
 - Reduce feature size rather than increase die size
- Designer controls only the die size
 - Decides which features/functions to include
 - Doesn't control feature size!
- Moral: smaller dies save money!

Computer performance

- Goal: quantitative comparison of two or more systems
 - Figure out which is “faster”
 - Definition of “faster” can vary...
 - User: response time & execution time
 - Sysadmin: throughput & bandwidth
- Measure relative performance using ratios
 - $\text{ExecTime}_A / \text{ExecTime}_B = 2 \Rightarrow B$ is 2x the speed of A
 - Performance = $1/\text{ExecTime}$: this corrects ratios so that $\text{Performance}_A / \text{Performance}_B = 2 \Rightarrow A$ is 2x the speed of B
- Execution time of real programs is the only consistent & reliable measure of performance!

Performance metrics

- Measure time in several ways
 - Wall clock time: total time to complete a task (including all operations & wait time)
 - CPU time: includes only the time spent actually executing, and excludes wait time & I/O time
- User time vs. system time
 - User time = time spent running user code
 - System time = time spent by process in the OS
 - System performance \Rightarrow elapsed time on an unloaded system
 - CPU performance \Rightarrow elapsed *user* CPU time
- Focus on CPU performance (exclude OS effects...)

Measuring performance: benchmarks

- Goal of performance evaluation: determine the speed of a computer on your specific workload
 - Best accomplished by simply running your workload, but...
 - Difficult to do without a lot of work on your part!
- Alternative: standard workloads called *benchmarks*
 - Real programs: spice, gcc, matmult, etc.
 - + Might match programs you might run
 - May not include the specific programs you run
 - Kernels: key (usually small) pieces of real programs
 - + Isolate individual performance elements
 - Exclude potentially time-consuming setup & other code



More on benchmarks

- Other kinds of benchmarks
 - Toy benchmarks: small programs that produce known results
 - + Easily ported to different architectures
 - Very small, and may not exercise full system
 - Synthetic benchmarks: artificial programs that try to “exercise” the system in the same way as an “average” real program
 - + Easily ported to different architectures
 - Average behavior may not be the same as the behavior of individual programs
- Benchmark suites include several different benchmarks
 - Different kinds of programs test different things
 - Workload can be formed from a combination of benchmarks



Running benchmarks

- Benchmarks are like other kinds of experiments
 - Results must be *reproducible*
 - Conditions need to be controlled as much as possible
- Guidelines for running benchmarks
 - Record as much information as possible
 - Hardware used: CPU, memory size, cache, disk, etc.
 - OS & compiler used (including compiler options, etc.)
 - Program version (preferably entire code!)
 - Program options
 - Program input & output
- Another person should be able to reproduce your results!



Comparing performance

- Goal: compare performance of several computer systems
 - Single number that shows performance?
 - Number should be larger for faster machines (use performance rather than execution time)
- Problem: this isn't as straightforward as it might seem
 - Benchmark INT takes 5 seconds on A, 10 seconds on B
 - Benchmark FP takes 18 seconds on A, 10 seconds on B
 - Which machine is faster, A or B?



Averaging benchmark results



Averaging benchmark results

$$\text{Arithmetic mean} = \frac{1}{n} \sum_{i=1}^n \textit{Execution time}_i$$

$$\text{Harmonic mean} = \frac{n}{\sum_{i=1}^n \frac{1}{\textit{rate}_i}}$$

$$\text{Weighted arithmetic mean} = \frac{1}{n} \sum_{i=1}^n \textit{weight}_i \times \textit{Execution time}_i$$



Combining benchmark results

- Use arithmetic mean for execution times
 - Use weighting to emphasize one benchmark
 - Capture relative frequency with weight
- Use harmonic mean for rates
 - Weighting can apply to harmonic means

$$\text{Arithmetic mean} = \frac{1}{n} \sum_{i=1}^n \text{Execution time}_i$$

$$\text{Harmonic mean} = \frac{n}{\sum_{i=1}^n \frac{1}{\text{rate}_i}}$$

$$\text{Weighted arith. mean} = \frac{1}{n} \sum_{i=1}^n \text{weight}_i \times \text{Execution time}_i$$

Another approach: normalize

- Normalize performance or execution time to that of a reference machine
 - Often use a VAX 11/780, set to 1 MIPS
 - Combine results with geometric mean:

$$\text{Geom mean} = \sqrt[n]{\prod_{i=1}^n \text{Performance}_i}$$

- Doesn't predict execution time
 - Allows easy comparison of two machines
 - Comparison doesn't depend on choice of "base" machine
- Ideal solution: measure a real workload on each machine!

Quantitative design principles

- Make the common case fast!
 - Optimize the most frequently used operations
 - Calculate overall performance to see if an optimization is worthwhile
- Example: arithmetic with overflow
 - Overflow is uncommon, so use exceptions to handle it
 - Don't make programs check every arithmetic operation!
- Use Amdahl's Law to quantify potential improvements in performance from improving particular operations

Amdahl's Law

- Performance improvement for a particular optimization is limited by the fraction of time the optimization is in use
- Amdahl's Law defines this speedup as:
 - $\text{Speedup} = \text{Execution time}_{\text{orig}} / \text{Execution time}_{\text{enhanced}}$
 - $\text{Speedup} = \text{Performance}_{\text{enhanced}} / \text{Performance}_{\text{orig}}$
- Factors affecting execution time:
 - $\text{Fraction}_{\text{enhanced}}$: fraction of execution time that can be sped up by the enhancement ($0 \leq \text{fraction} \leq 1$)
 - $\text{Speedup}_{\text{enhanced}}$: performance improvement on the enhanced portion of the code ($\text{speedup} > 1$)
 - $\text{Speedup} = \text{Execution time}_{\text{orig}} / \text{Execution time}_{\text{enhanced}}$

Defining Amdahl's Law

$$Exec\ time_{new} = \underbrace{Exec\ time_{old} \times (1 - Fraction_{enhanced})}_{\text{Time spent in original program}} + \underbrace{\frac{Fraction_{enhanced}}{Speedup_{enhanced}}}_{\text{Time spent in enhanced program}}$$

$$Speedup_{overall} = \frac{Exec\ time_{old}}{Exec\ time_{new}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

- This formula gives the overall speedup for a single enhancement

Amdahl's Law: Example

- A CPU without floating point hardware spends 40% of its time doing FP calculations
- Adding FP hardware will speed up FP calculations by 8x
- How much faster will the computer run with FP hardware?
- Answer:
 - $Fraction_{enhanced} = 0.4$
 - $Speedup_{enhanced} = 8$
 - $Speedup_{overall} = 1 / ((1 - Fraction_{enhanced}) + (Fraction_{enhanced} / Speedup_{enhanced}))$
 $= 1 / ((1 - 0.4) + (0.4 / 8)) = 1 / 0.65 = 1.54$
 - The CPU with FP hardware will run the program 1.54x faster

Implications of Amdahl's Law

- Law of diminishing returns
 - Speedup is limited by the fraction of execution time that can be sped up
 - If we only enhance 75% of the code, maximum speedup is 4!
- Parallel processing is hard!
 - In most parallel programs, there is some serial portion of the code that can't be parallelized
 - Suppose $\text{fraction}_{\text{serial}} = 0.02$
 - Maximum speedup is then 50x!
- In general, $\text{maximum speedup} = 1 / (1 - \text{Fraction}_{\text{enhanced}})$



CPU performance equations

- It can be difficult to directly measure performance improvements using new techniques
 - Must build (or simulate) the system first!
 - What about different programs?
- Another method: break execution time into three components
 - Instruction count (IC): instructions executed for a task
 - Cycles per instruction (CPI): average number of cycles each instruction requires
 - Clock cycle time (CCT): frequency of the CPU's clock
 - Execution time = $IC * CPI * CCT$
 - This method can easily predict performance gains from reducing any factor in the equation



CPI as a performance metric

- Cycles per instruction (CPI) is an important metric
 - Cycle time improvements usually come from advances in VLSI techniques
 - Instruction count reductions require changing the instruction set
 - CPI can be reduced for a given instruction set
- Calculate average CPI by:
 - CPU clock cycles in a program / instructions executed
 - Cycles in a program = execution time * clock frequency
- Lower CPIs are better
 - CPIs below 1 are possible (execute > 1 instruction per cycle)
 - Reducing average CPI improves performance for a given instruction set and cycle time

Improving CPU performance

- It can be difficult to change one factor in isolation
 - Cycle time: hardware (VLSI) & organization
 - CPI: organization & instruction set architecture (ISA)
 - Instruction count: ISA & compiler technology
- Often, different instructions behave differently:
$$CPU\ time = \left(\sum_{i=1}^n CPI_i \times IC_i \right) \times clock\ cycle\ time$$
 - CPI_i is the average CPI for instruction i
 - CPI_i isn't a constant for all instructions i !
 - IC_i is the number of times instruction i appears in the program
 - Measuring individual instruction CPIs and counts isn't too hard, and allows performance prediction for new programs

Fallacies & pitfalls

- MIPS (millions of instrs per second) isn't a good metric!
 - MIPS is dependent on the instruction set: difficult to compare across ISAs
 - MIPS varies between programs on the same computer
 - MIPS can be lower on computers with special purpose hardware and good compilers
 - Example: floating point hardware vs. emulation
 - Example: vector instructions
 - MIPS considers only CCT and CPI, but not instruction count
- More important: how much work gets done
- MIPS = “Meaningless Indication of Processor Speed”



More fallacies & pitfalls

- Beware synthetic benchmarks (ie, Whetstone & Dhrystone)
 - Often contain code sequences not in real code
 - Compilers can be tuned specifically to optimize for them
 - Behave unlike real programs in many ways
 - Fit into cache, unlike real programs
- Peak performance \neq observed performance
 - Peak = “guaranteed not to exceed”
 - Most programs run nowhere near peak performance
 - Pipeline hazards, superscalar issue, branches, and cache issues all slow down the CPU and prevent peak performance

