

Evaluating Computer Architecture Designs

- Evaluation is a key component in the iterative design process
- Two dimensions of evaluation:
 - Performance
 - Cost

Performance Evaluation: Metrics

- Depends on the level at which performance is measured
- Examples:
 - Application perspective
 - * Tasks performed per second (*Throughput*)
 - * Time to run the task (*Response time*)
 - System perspective
 - * (millions) of instructions per second: MIPS
 - * (millions) of (FP) operations per second: MFLOPS
 - Datapath (bus): megabytes per second
 - Function units: cycles per instruction
 - Transistors, wires, pins: cycles per second (clock rate)

Performance Measures: Caveat

- Be careful when comparing performance measures
- Example:
 - Program execution time (on CPU): user time + system time
 - Elapsed time/response time: total time to “complete a task”
 - Program execution time \neq elapsed/response time
 - Example:
 - * Unix provides all of these performance measures: 8u 2s 20 50%

Performance Comparison: Terminology

- “X is n% faster than Y” means

$$\frac{\text{ExTime}(Y)}{\text{ExTime}(X)} = \frac{\text{Performance}(X)}{\text{Performance}(Y)} = 1 + \frac{n}{100}$$

$$\Rightarrow n = \frac{100(\text{Performance}(X) - \text{Performance}(Y))}{\text{Performance}(Y)}$$

- Example: Y takes 15 seconds to complete a task, X takes 10 seconds. What % faster is X?
- Similar definitions can be defined for other performance metrics (e.g., throughput)

Performance Comparison: Example

Plane	DC to Paris	Speed	Passengers	Throughput (PMPH)
Boeing 747	6.5 hours	610 mph	470	266,700
Concorde	3 hours	1350 mph	132	178,200

- Performance metrics:
 - Time to run the task (travel time for each passenger)
 - Throughput (person miles per hour - PMPH)
- Performance comparison:
 - Speed of Concorde > Boeing 747
 - Throughput of Boeing 747 > Concorde

Improving Performance: Fundamentals

- Suppose we have a machine with 2 instructions
- Instruction A takes 100 cycles to execute
- Instruction B takes 2 cycles to execute
- We wish to improve the performance, which instruction do we improve?

Amdahl's Law

- Speedup due to enhancement E:

$$\text{Speedup}(E) = \frac{\text{ExTime w/o } E}{\text{ExTime w/ } E} = \frac{\text{Performance w/ } E}{\text{Performance w/o } E}$$

$$\text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times \left[(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right]$$

$$\text{Speedup}(E) = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Amdahl's Law: Example

- Floating point instructions improved to run 2X; but only 10% of actual instructions are FP.

$$\text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (0.9 + .1/2) = 0.95 \times \text{ExTime}_{\text{old}}$$

$$\text{Speedup}_{\text{overall}} = \frac{1}{0.95} = 1.053$$

- The speedup resulting from an enhancement is bounded by the inverse of the fraction that is not enhanced.

Amdahl's Law: Corollary

- Make the Common Case Fast
- Examples:
 - All instructions require an instruction fetch, only a fraction require a data fetch/store \Rightarrow optimize instruction access over data access
 - Programs exhibit *spatial* and *temporal locality*; and accesses to small memories are faster \Rightarrow design a *storage hierarchy* such that the most frequent accesses are to the smallest (closest) memories.

Taking a Closer Look at CPU Performance

- Execution time of a program has 3 components:

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

- Factors affecting CPU execution time:

	Inst. Count	CPI	Clock rate
Program	X		
Compiler	X	(X)	
Inst. Set	X	X	
Organization		X	X
Technology			X

Cycles per Instruction (CPI)

- Depends on the instruction

$$CPI_i = \text{Execution time of instruction } i \times \text{Clock Rate}$$

- Average cycles per instruction

$$CPI = \sum_{i=1}^n CPI_i \times F_i \text{ where } F_i = \frac{IC_i}{\text{Instruction Count}}$$

- Example:

Op	Freq	Cycles	CPI(i)	(% time)
ALU	50%	1	.5	(33%)
Load	20%	2	.4	(27%)
Store	10%	2	.2	(13%)
Branch	20%	2	.4	(27%)
CPI (typical mix)			1.5	

Measuring Processor Performance

- (Toy) benchmarks

- 10-100 line program
- e.g., sieve, puzzle, quicksort

- Synthetic benchmarks

- Attempt to match average frequencies of real workloads
- e.g., Whetstone (scientific applications), dhrystone (non-scientific applications)

- Kernels

- Time critical excerpts of real programs
- e.g., Livermore loops

- Real programs

- e.g., gcc, spice

Comparing and Summarizing Performance

- Is there a fair way to summarize the performance of a set of programs?
- Is there a way to summarize performance in a single number?
- Example: What is the performance ranking of the following three machines?

	Computer A	Computer B	Computer C
Program 1	1	10	20
Program 2	1000	100	20
Total time	1001	110	40

- **Arithmetic Mean** is Simplest
 - Cheating loophole: we can bias the result by appropriately sizing the input of one of the benchmark programs
 - **Weighted arithmetic mean**: $\sum_{i=1}^n \text{Weight}_i \times \text{Time}_i$

Normalized Performance and Geometric Mean

- Performance is normalized against a reference machine (execution time of a benchmark is divided by that on the reference machine)
- **Geometric mean** is used to summarize performance:

$$\sqrt[n]{\prod_{i=1}^n \text{Ratio}_i}$$

- Ratio_i : ratio of execution time of i^{th} program on given machine to that on reference machine
- Note: Ranking based on geometric mean is independent of the reference machine
- SPEC Marks measures the ratio of execution time on the target machine to that on a VAX 11/780

Common Benchmarking Mistakes

- Ignoring monitoring overhead
- Not ensuring same initial conditions
- Load-levels controlled inappropriately
- Caching effects ignored
- Collecting too much data but doing too little analysis
-