Lecture 05
Performance Metrics and Benchmarking

Which is "the best"?

Measuring & Improving Performance
(if planes were computers...)

Plane	People	Range (miles)	Speed (mph)	Avg. Cost (millions)
737-800	162	3,060	530	63.5
747-8I	467	8000	633	257.5
777-300	368	5995	622	222
787-8	230	8000	630	153

Which is best?

An "architecture" example

1 GHz clock rate, each instruction takes ~1.2 cycles to execute

How do we determine which machine is better?

2 GHz clock rate, each instruction takes ~1.8 cycles to execute

```
...
MOV R1, d(8)
Add R2, R3, R1
Sub R5, R2, R1
MOV d(9) R5
Add R4, R3, R0
...
```

Characterizing Performance

- How can one computer's performance be understood or two computers be compared?
- What factors go into achieving "high performance"?
  - Raw CPU speed?
  - Memory speed or bandwidth?
  - I/O speed or bandwidth?
  - The operating system's overhead?
  - The compiler?
- It is critical to succinctly summarize performance, and be able to meaningfully compare.

The Impact of a Computer Architect

- Number of instructions:
  - ISA design
- Number of clock cycles for each instruction:
  - Computer organization
- Cycle length:
  - Computer organization and lower level implementation
- What about the compiler and others?

Performance Metrics

- latency: response time, execution time
  - good metric for fixed amount of work (minimize time)
- throughput: bandwidth, work per time, "performance"
  - = (1 / latency) when there is NO OVERLAP
  - > (1 / latency) when there is overlap
    - in real processors there is always overlap
  - good metric for fixed amount of time (maximize work)
- comparing performance
  - A is N times faster than B if and only if:
    - perf(A)/perf(B) = time(B)/time(A) = N
  - A is X% faster than B if and only if:
    - perf(A)/perf(B) = time(B)/time(A) = 1 + X/100

A more "qualitative" example…

- What is better?
  - A machine that takes 1 ns to do "task X" 1 time
  - A machine that takes 15 ns to do "task X" 30 times…
    - …but 5 ns to do "task X" 1 time
  - You could say that the 1st machine has a lower latency for a single operation…
  - …while the 2nd machine has better throughput for multiple operations

Measures of Response Time

- Elapsed Time (total)
  - Counts everything:
    - Disk, memory, and I/O access
    - Operating System Overhead
    - Time when the process may be blocked
  - In some ways the most critical number, but often difficult to use for the purposes of enhancement
- CPU Time (execution time)
  - Does not include I/O or the time spent executing other programs
  - Often broken up into system time and user time
  - Generally accounts for memory performance

Comparing Performance

- "X is n times faster than Y"

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

- "Throughput of X is n times that of Y"

$$\frac{\text{Tasks per unit time}_X}{\text{Tasks per unit time}_Y} = n$$

- Goal: Improve overall CPU performance
- $$\frac{\text{Execution time of app A on machine Y}}{\text{Execution time of app A on machine X}} = n$$

Various definitions of speedup.

Execution time and throughput are really good performance metrics in that they're "lowest common denominators"

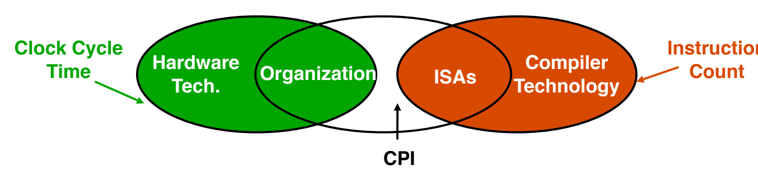
(i.e. if X finishes in 5 seconds and Y finishes in 10, its hard to make the case that Y is faster!)

Later, we discuss a few other performance metrics that you may sometimes see - but are generally not as good and/or misleading.

A CPU : The Bigger Picture

$$\frac{Instructions}{Program} \times \frac{Clock\ cycles}{Instruction} \times \frac{Seconds}{Clock\ Cycle} = \frac{Seconds}{Program} = CPU\ time$$

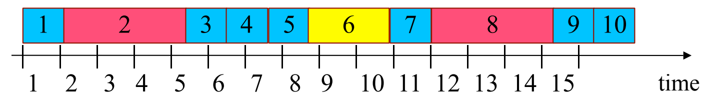
- We can see CPU performance dependent on:
  - Clock rate, CPI, and instruction count
- CPU time is directly proportional to all 3:
  - Therefore an x % improvement in any one variable leads to an x % improvement in CPU performance
- But, everything usually affects everything:



See Problem 1 in Handout

IC, CPI and IPC

Consider the processor we have worked on.
What is its CPI? IPC?



Total Execution Time = 15 cycles

Instruction Count (IC) = Number of Instructions = 10

Average number of cycles per instruction (CPI) =

Instructions per Cycle (IPC) =

Can CPI < 1?

Lots of examples!

- **See problems in handout distributed in class.**

Different Types of Instructions

- **Multiplication takes more time than addition**
- **Floating point operations take longer than integer operations**
- **Memory accesses take more time than register accesses**
- **NOTE: changing the cycle time often affects the number of cycles an instruction will take**

$$\text{CPU Clock Cycles} = \sum_{i=1}^{n} CPI_i * IC_i = AvgCPI * IC$$

Metrics

- Metrics Discussed:
  - Execution Time (instructions, cycles, seconds)
  - Machine Throughput (programs/second)
  - Cycles Per Instruction (CPI)
  - Instructions Per Cycle (IPC)
- Other Common Measures
  - MIPS (millions of instructions per second)
  - MFLOPS (megaflops) = millions of floating point operations per second

Exercise: Measurement Comparison

- Given that two machines have the same ISA, which measurement is always the same for both machines running program P?
  - Clock Rate:
  - CPI:
  - Execution Time:
  - Number of Instructions:
  - MIPS:

Performance Metric I: MIPS

- MIPS (millions of instructions per second)
  - (instruction count / execution time in seconds) x 10^{-6}
  - instruction count is not a reliable indicator of work
    - Prob #1: some optimizations add instructions
    - Prob #2: work per instruction varies (FP mult >> register move)
    - Prob #3: ISAs not equal (3 Pentium instrs != 3 Alpha instrs)
      - You'll see more when we talk about addressing modes
        - » Auto-increment may be a good example...
  - may vary inversely with actual performance

Performance Metric I: MIPS

- relative MIPS
  - $(\text{time}_{\text{reference}} / \text{time}_{\text{new}}) \times \text{MIPS}_{\text{reference}}$
    - (pro) a little better than native MIPS
    - (con) but very sensitive to reference machine
  - upshot: may be useful if same ISA/compiler/OS/workload

Benchmarks and Benchmarking

- "program" as unit of work
  - millions of them, many different kinds, which to use?
- benchmarks
  - standard programs for measuring/comparing performance
    - + represent programs people care about
    - + repeatable!!
    - benchmarking process
      - define workload
      - extract benchmarks from workload
      - execute benchmarks on candidate machines
      - project performance on new machine
      - run workload on new machine and compare
      - not close enough -> repeat

Benchmarks: Instruction Mixes

- instruction mix: instruction type frequencies
  - (minus) ignores dependences
  - (plus) ok for non-pipelined, scalar processor w/o caches
    - the way all processors used to be
  - example: Gibson Mix - developed in 1950's at IBM
    - load/store: 31%, branches: 17%
    - compare: 4%, shift: 4%, logical: 2%
    - fixed add/sub: 6%, float add/sub: 7%
    - float mult: 4%, float div: 2%, fixed mul: 1%, fixed div: <1%
    - qualitatively, these numbers are still useful today!

Benchmarks: Toys, Kernels, Synthetics

- toy benchmarks: little programs no one really runs
  - e.g., fibonacci, 8 queens
  - little value, what real programs do these represent?
- kernels: important (frequently executed) pieces of real programs
  - e.g., Livermore loops, Linpack (inner product)
  - (plus) good for focusing on individual features not big picture
    - For example, maybe you want to test the design of two different floating point units?
  - (minus) over-emphasize target feature (for better or worse)
- synthetic benchmarks:
  - programs made up for benchmarking
    - toy kernels++, which programs do these represent?

Benchmarks: Real Programs

- real programs
  - (plus) only accurate way to characterize performance
  - (minus) requires considerable work (porting)
- Standard Performance Evaluation Corporation (SPEC)
  - http://www.spec.org
  - collects, standardizes and distributes benchmark suites
  - consortium made up of industry leaders
  - SPEC CPU (CPU intensive benchmarks)
    - SPEC89, SPEC92, SPEC95, SPEC2000, SPEC2006
  - other benchmark suites
    - SPECjvm, SPECmail, SPECweb
- Other benchmark suite examples: TPC-C, TPC-H for databases

SPEC CPU 2000

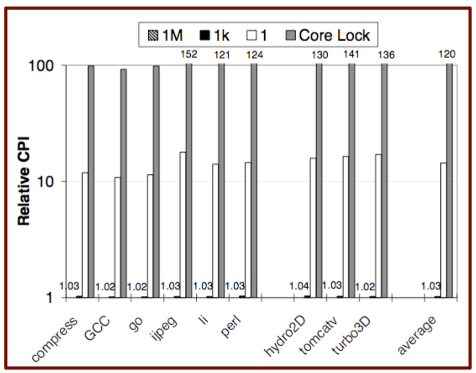
- 12 integer programs (C, C++)
  - gcc (compiler), perl (interpreter), vortex (database)
  - bzip2, gzip (replace compress), crafty (chess, replaces go)
  - eon (rendering), gap (group theoretic enumerations)
  - twolf, vpr (FPGA place and route)
  - parser (grammar checker), mcf (network optimization)
- 14 floating point programs (C, FORTRAN)
  - swim (shallow water model), mgrid (multigrid field solver)
  - applu (partial diffeq's), apsi (air pollution simulation)
  - wupwise (quantum chromodynamics), mesa (OpenGL library)
  - art (neural network image recognition), equake (wave propagation)
  - fma3d (crash simulation), sixtrack (accelerator design)
  - lucas (primality testing), galgel (fluid dynamics), ammp (chemistry)

SPEC 2000

- Different programs in the suite stress different parts of the architecture
  - For example:
    - One benchmark may be memory intensive...
    - ...another may be compute intensive...
    - ...another may be I/O intensive...
  - Ideally, show wins on all aspects, but most often not the case - or the point

A common architecture graph:

Often see graphs like this...



(and interestingly, now such a graph without accompanying power analysis is viewed as incomplete)

Benchmarking Pitfalls

- benchmark properties mismatch with features studied
  - e.g., using SPEC for large cache studies
- careless scaling
  - using only first few million instructions (init. phase)
  - reducing program data size
- choosing performance from wrong application space
  - e.g., in a realtime environment, choosing troff
- using old benchmarks
  - "benchmark specials": benchmark-specific optimizations
- Benchmarks must be continuously maintained and updated

Amdahl's Law

- Qualifies performance gain
- Amdahl's Law defined…
  - The performance improvement to be gained from using some faster mode of execution is limited by the amount of time the enhancement is actually used.
- Amdahl's Law defines speedup:

$$\text{Speedup} = \frac{\text{Perf. for entire task using enhancement when possible}}{\text{Perf. For entire task without using enhancement}}$$

Or

$$\text{Speedup} = \frac{\text{Execution time for entire task without enhancement}}{\text{Execution time for entire task using enhancement when possible}}$$

Amdahl's Law and Speedup

- Speedup tells us how much faster the machine will run with an enhancement
- 2 things to consider:
  - 1st…
    - Fraction of the computation time in the original machine that can use the enhancement
    - i.e. if a program executes in 30 seconds and 15 seconds of exec. uses enhancement, fraction = ½ (always < 1)
  - 2nd…
    - Improvement gained by enhancement (i.e. how much faster does the program run overall)
    - i.e. if enhanced task takes 3.5 seconds and original task took 7, we say the speedup is 2 (always > 1)

Deriving the previous formula

$$\text{Speedup}_{overall} = \frac{\text{Execution Time}_{old}}{\text{Execution Time}_{new}} = \boxed{\frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}}$$

$$\frac{1}{\underbrace{(1 - \text{Fraction}_{enhanced})}_{\text{1 - \% enhanced}} + \boxed{\frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}}$$

normalized old execution time

1 - % enhanced (i.e. part of the task will take the same amount of time as before)

$$\frac{\text{\% of task that will run faster}}{\text{how much faster it will run}}$$

(note: # should be > 1)
(otherwise, performance gets worse)
(represents new component of ex. time)

See class handout for Amdahl's Law Examples