Performance

Computer Organization
Ellen Walker
Hiram College

Figures from Computer Organization and Design 3ed, D.A. Patterson & J.L. Hennessey, Morgan Kauffman © 2005 unless otherwise specified
Why Measure Performance?

• Choose among computers
• Predict how fast software will run
• Understand how hardware features affect performance
• Determine where the “bottleneck” is, so performance can be improved
Factors in Performance

• How well does software use the underlying machine instructions?
• How well does hardware implement the instructions?
• How well does memory perform?
• How does I/O perform?
Defining Performance

• What does “speed” mean?
  – Raw speed: Concorde @ 1350mph (but only holds 132 passengers)
  – Throughput: Boeing 747 transports 286,700 passenger-miles per hour (470 passengers at 610mph)
Relative Performance

- Typical measure is ratio of performance or execution time
  - \( \frac{\text{Performance}_x}{\text{Performance}_y} = 2 \)
  - \( \frac{\text{Execution}_y}{\text{Execution}_x} = 2 \)
  - X is twice as fast as Y
  - Y takes twice as long as X
Measuring Program Performance

- **Response time (execution time)** - time between start and completion of a single task
- **Throughput** - total amount of work done in a given time
- Decreasing response time almost always increases throughput, but not vice versa
Example

• You are accessing many pages from a complex website over a slow network. Several pages are repeatedly visited. What can be done to make the process faster?

• Which changes…
  – Decrease response (execution) time?
  – Increase throughput?
Performance Metrics Vary

- Desktop computers
- Servers
- Embedded computers
Relative Performance & Execution Time

- Performance_x = 1/ Execution time_x
- Performance_x > Performance_y when Execution time_x < Execution time_y
- “Improving performance” means reducing execution time
- “Improving execution time” also means reducing execution time
Measuring Execution Time

• Response time (wall-clock)
  – Includes disk, memory, I/O, OS etc.

• CPU time
  – Includes only the time CPU is computing (not waiting)
  – User CPU (user’s program)
  – System CPU time (Operating System functions)
Execution Time Examples

• “time cmd” on any Unix-like system
• Reports User time (u), System time (s), Elapsed time and additional information
• 5 results of “time ls -l”
  0.003u 0.006s 0:00.14
  0.003u 0.007s 0:00.07
  0.003u 0.006s 0:00.02
  0.001u 0.004s 0:00.02
  0.001u 0.005s 0:00.10
Measuring CPU Time

- Every system has a clock that “ticks” at a fixed rate
- This clock synchronizes the hardware, e.g. registers all change at once
- CPU time can be measured in clock cycles
- “Overclocking” - running the CPU using a faster clock than intended by the designer
CPU Performance

- Execution time = CPU clock cycles x clock cycle time
- Execution time = CPU clock cycles / clock rate
- Clock cycle time measured in seconds
- Clock rate measured in Hertz (cycles per second)
Improve performance by...

• Reducing number of cycles
  – For example, add hardware to do more in parallel in the same cycle

• Reducing cycle time
  – For example, simplify the instruction set so that less computation is needed to interpret each instruction

• Tradeoff: length of cycle vs. number of cycles for a program
One more measure

- Clock Cycles Per Instruction (CPI) - a measure of the complexity of the instructions in the architecture
- Usually this is an average, since some instructions take more cycles than others
  - Multiplication is one of the longest instructions
  - Unconditional jump is one of the shortest instructions
Basic Performance Equation

- CPU time = \(\frac{\text{instruction count} \times \text{CPI}}{\text{clock rate}}\)
- Equivalently, CPU time = \(\text{instruction count} \times \text{CPI} \times \text{cycle time}\)
- Units
  - Instruction count in “instructions”
  - CPI in “cycles per instruction”
  - Clock rate in “cycles per second”
  - Cycle time in “seconds per cycle”
Measuring Performance

- **CPU Execution time**
  - Run the program

- **Clock cycle time**
  - Published as part of computer specification

- **Instruction count**
  - Use profiling tools, simulator, hardware counters
  - Depends on architecture only

- **CPI**
  - Hardware counters, or compute from CPU execution time
Predicting CPI

• CPI depends on
  – Number of cycles per each instruction
  – Relative frequency of different types of instructions in the program execution

• CPI cannot be computed without reference to a specific program (or set of programs)
Example Performance Comparison

- **System A:**
  - Cycle time = 250 (picoseconds / cycle)
  - CPI = 2.0 (cycles / instruction)

- **System B:**
  - Cycle time = 500 (picoseconds / cycle)
  - CPI = 1.2 (cycles / instruction)

- Which system is faster?
Tradeoffs

• Comparing on only one feature can be misleading.
  – Program A uses fewer instructions but higher CPI
  – Program B uses more instructions but lower CPI

• Cycle time complicates the calculation
  – More complex instructions lead to both smaller Instruction Count and larger Cycle Time
  – Larger cycle time leads to fewer CPI (possibly)
Another Way to Look at the Equation

• Time =
  seconds / program =
  instructions/program x
  cycles/instruction x
  seconds/cycle

• In the end, the only appropriate measure of performance is time
Program Performance Overview

- Algorithm affects Instruction Count
- Programming Language affects Instruction Count, CPI
- Compiler affects Instruction Count, CPI
- Instruction Set Architecture affects Instruction Count, Clock Rate, CPI
Evaluating Performance

• **Workload**
  – Actual mix of applications (for a given user)
  – Best test is to compare machines on actual workload

• **Benchmark**
  – Programs specifically chosen to measure performance
  – Should be similar in relative performance to real workload
Best Benchmarks are “Real”

- Best approximation of workload is reality
- Different applications for different users
- Much harder for system designers to find “trivial speedups”
Risk of Small Benchmarks

• Compilers specially optimized for their code
• Architecture optimized for a specific sequence of instructions that only appears in the benchmark
• Erroneous optimizations introduced specifically for benchmarks
Why Small Benchmarks?

• Easier to hand-compile and simulate
  – Helpful during design phase
  – Analysis before hardware and/or compiler is available

• Even so, use real programs to evaluate working systems
Reproducibility

• Provide complete information so the test can be run again in the same environment
  – Program(s) run
  – Data sets
  – Operating system version
  – Compiler version and flags (e.g. optimizations chosen)
  – Computer configuration
Reporting Results

• Different programs cause different performance
  – A is twice as fast on program 1 (100 vs 200)
  – B is 3x as fast on program 2 (60 vs 20)

• Total execution time is a better measure
  – A is faster than B overall (160 vs 220)
Reporting Complex Workloads

• Time each program in workload
• Compute *weighted arithmetic mean*
  – Each program weighted by its frequency
  – \((1/\text{total weight}) \times \text{sum (weight[i] \times time[i])}\)
Example

- Workload consists of programs X (30%), Y (25%), and Z (45%)
- Runtimes are:

<table>
<thead>
<tr>
<th>Program</th>
<th>System A</th>
<th>System B</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>150</td>
<td>200</td>
</tr>
<tr>
<td>Y</td>
<td>500</td>
<td>400</td>
</tr>
<tr>
<td>Z</td>
<td>30</td>
<td>50</td>
</tr>
</tbody>
</table>
SPEC Benchmarks

• SPEC = System Performance Evaluation Corporation (www.spec.org)
• Offers many different benchmark sets including:
  – CPU performance
  – Graphics
  – High performance computing
  – Java client/server computing
  – Client/server models
  – Web systems
Examples: SPEC 2000 CPU

- **Integer Component of SPEC2000 CPU**
- **Sample Reports**
  - Dell PowerEdge 2800 (3.6Ghz Intel Xeon)
  - HP ProLiant ML570 G3 (3.66GHz, Intel Xeon MP)
Growth in Processor Performance (SPEC vs. Vax)
Ahmdal’s Law

• Relationship between speedup of one aspect and overall performance
  improved time =
    affected time / change + unaffected time

• Example:
  – Original time for benchmark is 100, including 20 for floating point processing
  – Improve floating point processing 5x
  – Old time was 100
  – New time = 20/5 + 80 = 84 (nowhere near 5x)!
Results of Ahmdal’s Law

• Make the common case fast
• Don’t bother optimizing rare cases

• “Ironically, the common case is often simpler … and hence easier to optimize” - p. 267
MIPS as a Performance Measure

- MIPS = million instructions per second
  - MIPS = $10^{-6} \times \frac{1}{CPI} \times$ clock rate
  - Easy to measure and compare
  - Ignores complexity of instructions
  - Ignores different mix of instructions in different programs
The Power Wall

- Limit to how much heat can be dissipated!
- Need to change the way we make processors faster
Multicore

• Create single chips that contain multiple processors
  – Each is a ‘core’
  – Each has CPU, cache, registers, etc.
• Programmers need to rewrite their programs (or compilers) to get the benefit of multicore performance!
Parallel Programming is Hard

• Need to divide up the work “evenly”
• Need to keep all the processors busy
  – If one is waiting for the results of another, no speedup!
• Often requires new and different algorithms
  – Parallel search, parallel sort, etc.
Summary

• Performance equation
  \[ \text{Sec/program} = \frac{\text{inst/program}}{\text{cycles/inst}} \times \frac{\text{sec/cycle}}{\text{cycles/inst}} \]
• Tradeoffs among performance components
• Focusing only on *part* of performance is misleading
  – Examples: clock rate, limited benchmark
• Concentrate on common cases
• The future is multicore / parallel