Performance Overview

- Execution time is the best measure of performance: simple, intuitive, straightforward.
- Two important quantitative methods:
  - Amdahl’s Law and Speedup
  - CPI - cycles per instruction
- Benchmarks
- Metrics for summarizing performance data
- Pitfalls

Time-based Metrics

<table>
<thead>
<tr>
<th>Plane</th>
<th>DC to Paris</th>
<th>Speed</th>
<th>Passengers</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 747</td>
<td>6.5 hours</td>
<td>610 mph</td>
<td>470</td>
<td>286,700</td>
</tr>
<tr>
<td>Concorde</td>
<td>3 hours</td>
<td>1350 mph</td>
<td>132</td>
<td>178,200</td>
</tr>
</tbody>
</table>

- Elapsed time to run the task
  - Execution time, response time, latency
- Rate: tasks completed per day, hour, week, sec, ns
  - Throughput, bandwidth

In passenger-mile/hour
Performance comparisons

"X is n times faster than Y" means

\[
\frac{\text{ExTime}(Y)}{\text{ExTime}(X)} = \frac{\text{Performance}(X)}{\text{Performance}(Y)} = n
\]

Execution time metric

wall clock time = response time = elapsed time
CPU time = user CPU time + system CPU time

We will measure CPU performance using user CPU time on an unloaded system

Example: Unix time command

90.7u 12.9s 2:39 65%
**Amdahl's Law**

Speedup due to enhancement E:

\[
\frac{\text{ExTime w/o E}}{\text{ExTime w/ E}} = \frac{\text{Performance w/ E}}{\text{Performance w/o E}}
\]

Suppose that enhancement E accelerates a fraction \( F \) of the task by a factor \( S \), and the remainder of the task is unaffected.

\[
\text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (1 - \text{Fraction}_{\text{enhanced}}) + \text{Fraction}_{\text{enhanced}}
\]

\[
\text{Speedup}_{\text{overall}} = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \text{Fraction}_{\text{enhanced}}}
\]
**Amdahl’s Law**

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

\[
\text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (0.9 + 0.1/2) = 0.95 \times \text{ExTime}_{\text{old}}
\]

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

Corollary: Make the common case fast!

**Amdahl’s Law Corollary:**

Make the common case fast!

- All instructions require instruction fetch, only a fraction require data fetch/store.
  - \(\implies\) optimize instruction access over data access
- Access to small memories is faster.
  - \(\implies\) organize the storage hierarchy such that most frequent accesses are to the smallest/closest memory unit.
- Programs exhibit locality (spatial and temporal).
  - \(\implies\) implement pre-fetching of nearby code/data
**CPU Time Analysis**

**Terminology**
- IC = instruction count = number of instructions in the program
- CPI = cycles per instruction (varies for different instructions)
- clock cycle = length of time between clock ticks
  
  Note: clock cycle = 1 / clock frequency
  where frequency is measured in MHz

If we assume the CPI is constant for all instructions, we have:

\[
\text{CPU time} = \text{IC} \times \text{CPI} \times \text{clock cycle}
\]

---

**More realistic CPU time analysis**

- A given machine has several classes of instructions.
- Each class of instructions has its own cycle time.

This equation includes separate IC and CPI for each instruction class:

\[
\text{CPU time} = \sum \left( \frac{\text{CPI}_i \times \text{IC}_i}{\text{freq}_i} \right) \times \text{clock cycle}
\]

Alternatively, if we know the frequency of occurrence of each instruction type:

\[
\text{CPU time} = \text{IC} \times \left( \sum \frac{\text{CPI}_i \times \text{freq}_i}{\text{freq}_i} \right) \times \text{clock cycle}
\]

where \( \sum \frac{\text{CPI}_i \times \text{freq}_i}{\text{freq}_i} = \text{avg. CPI} \)
### Example: Calculating CPI

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>CPI</th>
<th>CPI x F</th>
<th>(% Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>0.5</td>
<td>(33%)</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>2</td>
<td>0.4</td>
<td>(27%)</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>2</td>
<td>0.2</td>
<td>(13%)</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>0.4</td>
<td>(27%)</td>
</tr>
</tbody>
</table>

**Typical Mix**

Average CPI = 1.5

### Speedup computation using the CPI eqn

\[
\text{Speedup} = \frac{\text{IC(\text{old})} \times \text{CPI (old)} \times \text{clock cycle (old)}}{\text{IC(new)} \times \text{CPI (new)} \times \text{clock cycle (new)}}
\]

When doing problems, identify which of the three components have changed (old -> new). Need only include those components in the speedup equation since unchanged ones will cancel out.
Programs to evaluate performance

- Real benchmark programs
- Kernels
- Toy benchmarks
- Synthetic benchmarks

SPEC Benchmark Suite
(Standard Performance Evaluation Cooperative
http://www.spec.org)

- First Round 1989
  - 10 programs yielding a single number (“SPECmarks”)

- Second Round 1992
  - SPECInt92 (6 integer programs) and SPECfp92 (14 floating point programs).
  - Compiler Flags unlimited
  - Normalized to VAX-11/780
SPEC Benchmark Suite - continued

- **Third Round 1995** “benchmarks useful for 3 years”
  - SPECint95 (8 integer programs) and SPECfp95 (10 floating point)
  - Single flag setting for all programs: SPECint_base95, SPECfp_base95

- **Fourth Round 2000**
  - CINT2000 (12 integer programs) and CFP2000 (14 floating point)
  - usable across Unix and Windows NT
  - Normalized to Sun Ultra5_10 workstation with 300-MHz SPARC and 256 MB memory
  - base and optimized versions

Newer desktop (PC) benchmarks

- **Winbench**
  scripts that test CPU, video, and disk performance

- **Business Winstone**
  netscape, office suite applications

- **CC Winstone**
  content creation applications such as Photoshop
Specialized Benchmarks

- **Graphics benchmarks**
  - SPECviewperf, SPECapc

- **Embedded systems benchmarks**
  - EEMBC - automotive, consumer, networking, office automation, telecommunications

- **Server benchmarks**
  - SPEC benchmarks for CPU, files system, web server, transaction servers (www.tpc.org)

How to Summarize Performance

Two metrics for summarizing **execution time**

- **Arithmetic mean** (weighted arithmetic mean) tracks execution time: $\frac{\sum(T_i)w_i}{n}$ or $\frac{\sum(W_iT_i)}{\sum(W_i)}$

- **Harmonic mean** (weighted harmonic mean) of rates (e.g., MFLOPS) tracks execution time: $\frac{n}{\sum\left(\frac{1}{R_i}w_i\right)}$ or $\frac{n}{\sum\left(W_i/R_i\right)}$
How to Summarize Performance

- **Normalized execution time** for scaling performance (e.g., X times faster than SPARCstation 10)
  - Arithmetic mean impacted by choice of reference machine

- **Geometric mean** for comparison of normalized execution times
  - $(T_i)^{1/n}$
  - Independent of chosen machine
  - but not good metric for total execution time

Fallacies and Pitfalls

- **Fallacy**: MIPS is an accurate measure of comparative performance.
  \[ \text{MIPS} = \frac{\text{instruction count}}{\text{execution time} \times 10^6} \]

- **Fallacy**: MFLOPS is a consistent and useful measure of performance.
  \[ \text{MFLOPS} = \frac{\text{floating point operations}}{\text{execution time} \times 10^6} \]
More Fallacies and Pitfalls

- **Fallacy**: Synthetic benchmarks predict performance for real programs.
- **Fallacy**: Benchmarks remain valid indefinitely.
- **Fallacy**: Peak performance tracks observed performance.
- **Fallacy**: Your performance in CIS 629 depends on how much you eat in class.