Why Measure Performance?

• Performance is important
• Identify HW/SW performance problems
• Compare & choose wisely
  • Which system configuration is better?
  • Which ISA is better?
  • Which ISA implementation is better?
• Expose significant issues, ignore others
## What Does Performance Measure?

<table>
<thead>
<tr>
<th>Airplane</th>
<th>Passengers</th>
<th>Range (mi)</th>
<th>Speed (mph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 737-100</td>
<td>101</td>
<td>630</td>
<td>598</td>
</tr>
<tr>
<td>Boeing 747</td>
<td>470</td>
<td>4150</td>
<td>610</td>
</tr>
<tr>
<td>BAC/Sud Concorde</td>
<td>132</td>
<td>4000</td>
<td>1350</td>
</tr>
<tr>
<td>Douglas DC-8-50</td>
<td>146</td>
<td>8720</td>
<td>544</td>
</tr>
</tbody>
</table>
Measures of Computer Performance

• What to measure?
  • Execution time for application
  • Power / temperature / battery life
  • Reliability / availability
  • Cost for acceptable functionality
  • Size
• Measure what matters to you...
Measures of Computer Performance

• **Response Time & Throughput**
  • Time to complete an operation
  • Jobs completed per unit time
  • Often can trade one for the other
• Performance(X) = 1/ExecutionTime(X)
• X is Performance(X)/Performance(Y) times faster than Y, also:
  ExecutionTime(Y)/ExecutionTime(X)
For Whom The Clock Ticks

- Posix uses real, user, system time
- Real “Wall Clock” time always ticks
- CPU time ticks only when CPU is yours
  - User time while in your code
  - System time while in OS code for you
  - Multiplied by #PEs in multiprocessors
- I/O time not reported under Posix
What Is The Clock?

- Not as simple as you think...
- Used to count AC zero crossings in SW
- Legacy of the IBM PC:
  - Clock chip counts seconds (w/battery)
  - Counter/timer chip @ 1.193181 MHz
- Processor tick count performance register
- Borrowed clocks: **NTP, PTP, & GPS**
- **Jiffy** is system interrupt interval (1-10ms)
- Posix counts seconds since 1970, but knows timezones, leaps, etc.
Running What program?

- Different program, different performance
- Application (all that really matters!)
- “Toy” program
- **Benchmark**: representative application
- **Micro Benchmark**: tests a certain feature
- **Synthetic Benchmark**: a program written solely to perform like a particular application, but doing nothing useful
- **Benchmark Suite**: multiple benchmarks
Common Metrics

- Application/benchmark time for specific data: e.g.: Quake updates/s
- LIPS: Logical Inferences/s
- FLOPS: Floating-Point OPerations/s
- MB/s, Mb/s: MegaBytes/Megabits per s
- MIPS: Millions of Instructions/s
- MOPS: “” OPerations/s
- Hz: clock cycles/s
- CPI: Cycles Per Instruction
- IPC: Instructions Per Cycle
CPI

- Clock ticks at a (mostly) constant rate
- Can express program runtime as:

\[
\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Cycles}}{\text{Program}} \times \frac{\text{Seconds}}{\text{Cycle}}
\]

\[
\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Cycles}}{\text{Program}} \div \text{Frequency}
\]
Cycles / Program?

• Programs are made of instructions
• Can use CPI to compute:

\[
\frac{\text{Cycles}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \text{CPI}
\]

\[
\frac{\text{Cycles}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \div \text{IPC}
\]
All Instructions Alike?

- Different instruction type, different CPI
- Can sum over types separately:

\[ \sum \left( \frac{\text{Instructions}}{\text{Program}} \times \text{CPI} \right) \]
Parameters

- Instruction types can be “classes”
- Instructions / Program
  - Expected execution counts
  - % or ratios for relative performance
- CPI
- Clock period same for everything (analyze separately for each clock rate if the processor dynamically throttles)
An Example

This program takes:

$$(((20\times10)+(10\times30))\times10\text{ns} = 5\text{us}$$

What can be changed to make it $$4\text{us}$$?
A Little Disclaimer...

That CPI analysis assumes sequential execution of instructions, but most modern processors are parallel in various ways...

the model is still useful, but approximate, using 1/IPC to approximate CPI; it also works to analyze sequential portions of a design
## What Effects What?

<table>
<thead>
<tr>
<th></th>
<th>Instruction Count</th>
<th>CPI</th>
<th>Clock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program (Algorithm)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impl. Arch.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLSI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instruction Count</td>
<td>CPI</td>
<td>Clock Rate</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------</td>
<td>-----------</td>
<td>------------</td>
</tr>
<tr>
<td>Program (Algorithm)</td>
<td>Yes!</td>
<td>Indirectly...</td>
<td>No!</td>
</tr>
<tr>
<td>Compiler</td>
<td>Yes!</td>
<td>Indirectly...</td>
<td>Power?</td>
</tr>
<tr>
<td>ISA</td>
<td>Yes!</td>
<td>Yes!</td>
<td>Indirectly...</td>
</tr>
<tr>
<td>Impl. Arch.</td>
<td>uOps?</td>
<td>Yes!</td>
<td>Yes!</td>
</tr>
<tr>
<td>VLSI</td>
<td>No!</td>
<td>Indirectly...</td>
<td>Yes!</td>
</tr>
</tbody>
</table>
How Does A Change Affect Design?

- For a particular application
- For a particular compiler
- For a particular ISA
- For a particular implementation arch.
- For a particular VSI technology
- Etc.
Amdahl's Law

- If $1/N$ time is not affected by a change, the best possible speedup is only $N$
- Originally for sequential overhead in parallel code, but applies for any change

Suppose a program spends 80% of its time doing multiplies... you can't get more than a 5X speedup by improving only multiplies!
A Lesson From Top500.org

Projected Performance Development

- 10 EFlop/s
- 1 EFlop/s
- 100 PFlop/s
- 10 PFlop/s
- 1 PFlop/s
- 100 TFlop/s
- 10 TFlop/s
- 1 TFlop/s
- 100 GFlop/s
- 10 GFlop/s
- 1 GFlop/s
- 100 MFlop/s

Lists

- Sum
- #1
- #500
Summary

• Most performance numbers not relevant; measure what you care about
• Relate performance to causes
• Best designs usually make everything (and hence nothing) the bottleneck
• Stuff is getting better fast...
  Don't base design decisions on now, but on when you will need/market it