Performance

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Defining Performance (1)

- Which airplane has the best performance?

- **Passenger Capacity**
  - Boeing 777
  - Boeing 747
  - BAC/Sud Concorde
  - Douglas DC-8-50

- **Cruising Range (miles)**
  - Boeing 777
  - Boeing 747
  - BAC/Sud Concorde
  - Douglas DC-8-50

- **Cruising Speed (mph)**
  - Boeing 777
  - Boeing 747
  - BAC/Sud Concorde
  - Douglas DC-8-50

- **Passengers x mph**
  - Boeing 777
  - Boeing 747
  - BAC/Sud Concorde
  - Douglas DC-8-50
Defining Performance (2)

- **Performance issues**
  - Measure, analyze, report, and summarize
  - Make intelligent choices
  - See through the marketing hype
  - Key to understanding underlying organizational motivation
- **Questions**
  - Why is some hardware better than others for different programs?
  - What factors of system performance are hardware related?
    (e.g., Do we need a new machine, or a new operating system?)
  - How does the machine’s instruction set affect performance?
Computer Performance (1)

- **Response time (≈ execution time, latency)**
  - The time between the start and completion of a task
  - How long does it take for my job to run?
  - How long must I wait for the database query?

- **Throughput (≈ bandwidth)**
  - The total amount of work done in a given time
  - How much work is getting done per unit time?
  - What is the average execution rate?

- **What if ...**
  - We replace the processor with a faster version?
  - We add more processors?
Relative performance

- Define

\[ \text{Performance} = \frac{1}{\text{Execution Time}} \]

- “X is \( n \) times faster than Y”

\[ \frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n \]

- Example: time taken to run a program
  - 10s on machine A, 15s on machine B
  - Execution Time_B / Execution Time_A = 15s / 10s = 1.5
  - Machine A is 1.5 times faster than machine B
Measuring Execution Time

- **Elapsed time**
  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - Determines system performance

- **CPU time**
  - Time spent processing a given job
    - Discounts I/O time, other jobs’ shares
  - Comprises user CPU time and system CPU time
  - Different programs are affected differently by CPU and system performance

- **Our focus: User CPU time**
CPU Clocking

- **Clock**
  - Operation of digital hardware governed by a constant-rate clock
  - Clock “ticks” indicate when to start activities
  - Clock period: duration of a clock cycle
  - Clock frequency (rate): cycles per second

![Clock Diagram](image)
CPU Time (1)

\[ CPU \ Time = CPU \ Clock \ Cycles \times Clock \ Cycle \ Time \]

\[ = \frac{CPU \ Clock \ Cycles}{Clock \ Rate} \]

- **Performance improved by**
  - Reducing the number of clock cycles
  - Increasing clock rate (or decreasing the clock cycle time)
  - Hardware designer must often trade off clock rate against cycle count
CPU Time (2)

- **Example:**
  - Computer A: 2GHz clock, 10s CPU time
  - Designing Computer B
    - Aim for 6s CPU time
    - Can do faster clock, but causes 1.2 x clock cycles
  - How fast must Computer B clock be?

\[
\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6s}
\]

\[
\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A
\]

\[
= 10s \times 2GHz = 20 \times 10^9
\]

\[
\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4GHz
\]
CPI (1)

- Instruction count and CPI

\[ \text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction} \]

\[ \text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time} \]

\[ = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}} \]

- Instruction count for a program
  - Determined by program, ISA, and compiler

- Average cycles per instruction (CPI)
  - Determined by CPU hardware
  - If different instructions have different CPI
    - The average CPI affected by instruction mix
CPI (2)

- CPI example
  - Computer A: Cycle time = 250ps, CPI = 2.0
  - Computer B: Cycle time = 500ps, CPI = 1.2
  - Same ISA
  - Which is faster, and by how much?

\[
\begin{align*}
CPU\ Time_A &= Instruction\ Count \times CPI_A \times Cycle\ Time_A \\
&= I \times 2.0 \times 250ps = I \times 500ps
\end{align*}
\]

\[
\begin{align*}
CPU\ Time_B &= Instruction\ Count \times CPI_B \times Cycle\ Time_B \\
&= I \times 1.2 \times 500ps = I \times 600ps
\end{align*}
\]

\[
\frac{CPU\ Time_B}{CPU\ Time_A} = \frac{I \times 600ps}{I \times 500ps} = 1.2
\]
CPI (3)

- **CPI in more detail**
  - If different instruction classes take different numbers of cycles:

\[
\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)
\]

- Weighted average CPI

\[
\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left( \text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)
\]

**Relative frequency**
CPI (4)

- Example:
  - Alternative compiled code sequences using instructions in classes A, B, C

<table>
<thead>
<tr>
<th>Class</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPI for class</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>IC in sequence 1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>IC in sequence 2</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Sequence 1: IC = 5
  - Clock cycles = 2x1+1x2+2x3 = 10
  - Avg. CPI = 10/5 = 2.0

- Sequence 2: IC = 6
  - Clock cycles = 4x1+1x2+1x3 = 9
  - Avg. CPI = 9/6 = 1.5
MIPS

- **MIPS: Millions of Instructions Per Second**
  - MIPS as a performance metric?
  - Doesn’t account for
    - Differences in ISAs between computers
    - Differences in complexity between instructions

\[
MIPS = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}
\]

\[
= \frac{\text{Instruction count}}{\text{Instruction count} \times \text{CPI} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}
\]

- CPI varies between programs on a given CPU
C Sort Example (1)

- **Bubble sort in C**

```c
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v, j);
        }
    }
}
```
C Sort Example (2)

- **Effect of compiler optimization**

![Bar charts showing relative performance, instruction count, clock cycles, and CPI for different compiler optimizations (O0, O1, O2, O3).](image)

Compiled with gcc for Pentium 4 under Linux.
C Sort Example (3)

- Effect of language and algorithm

1. Bubblesort Relative Performance
2. Quicksort Relative Performance
3. Quicksort vs. Bubblesort Speedup
C Sort Example (4)

**Lessons**

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!
### Performance Summary

CPU Time = \[\frac{Instructions}{Program} \times \frac{Clock\ cycles}{Instruction} \times \frac{Seconds}{Clock\ cycle}\]

<table>
<thead>
<tr>
<th></th>
<th>Instruction Count</th>
<th>CPI</th>
<th>Clock Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>○</td>
<td>△</td>
<td></td>
</tr>
<tr>
<td>Programming language</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>Microarchitecture</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td></td>
<td>○</td>
</tr>
</tbody>
</table>
Benchmarks

- **How to measure the performance?**
  - Performance best determined by running a real application
  - Use programs typical of expected workload
  - Or, typical of expected class of applications

- **Small benchmarks**
  - Nice for architects and designers
  - Easy to standardize
  - Can be abused
SPEC CPU Benchmark (1)

- **SPEC (Standard Performance Evaluation Corp.)**
  - Develops benchmarks for CPU, I/O, Web, ...
  - [http://www.spec.org](http://www.spec.org)

- **SPEC CPU benchmark**
  - An industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler.
    - Companies have agreed on a set of real program and inputs
    - Valuable indicator of performance (and compiler technology)
  - CPU89 → CPU92 → CPU95 → CPU2000 → CPU2006
  - Can still be abused
Spec CPU Benchmark (2)

- Benchmark games

An embarrassed Intel Corp. acknowledged Friday that a bug in a software program known as a compiler had led the company to overstate the speed of its microprocessor chips on an industry benchmark by 10 percent. However, industry analysts said the coding error…was a sad commentary on a common industry practice of “cheating” on standardized performance tests…The error was pointed out to Intel two days ago by a competitor, Motorola …came in a test known as SPECint92…Intel acknowledged that it had “optimized” its compiler to improve its test scores. The company had also said that it did not like the practice but felt to compelled to make the optimizations because its competitors were doing the same thing…At the heart of Intel’s problem is the practice of “tuning” compiler programs to recognize certain computing problems in the test and then substituting special handwritten pieces of code…

Saturday, January 6, 1996 New York Times
SPEC CPU Benchmark (3)

- **SPEC CPU2006**
  - Elapsed time to execute a selection of programs
    - Negligible I/O, so focuses on CPU performance
  - Normalize relative to reference machine
    - Sun’s historical “Ultra Enterprise 2” introduced in 1997
    - 296MHz UltraSPARC II processor
  - Summarize as geometric mean of performance ratios
    - CINT2006: 12 integer programs written in C and C++
    - CFP2006: 17 FP programs written in Fortran and C/C++
SPEC CPU Benchmark (4)

- SPEC CPU2006 (cont’d)

<table>
<thead>
<tr>
<th>Integer Benchmarks (CINT2006)</th>
<th>Floating Point Benchmarks (CFP2006)</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench C Perl programming language</td>
<td>bwaves Fortran Fluid dynamics</td>
</tr>
<tr>
<td>bzip2 C Compression</td>
<td>gamess Fortran Quantum chemistry</td>
</tr>
<tr>
<td>gcc C C compiler</td>
<td>milc C Physics: Quantum chromodynamics</td>
</tr>
<tr>
<td>mcf C Combinatorial optimization</td>
<td>zeusmp Fortran Physics / CFD</td>
</tr>
<tr>
<td>gobmk C Artificial intelligence: Go</td>
<td>gromacs C/Fortran Biochemistry / Molecular dynamics</td>
</tr>
<tr>
<td>hmer C Search gene sequence</td>
<td>cactusADM C/Fortran Physics / General relativity</td>
</tr>
<tr>
<td>sjeng C Artificial intelligence: Chess</td>
<td>leslie3d Fortran Fluid dynamics</td>
</tr>
<tr>
<td>libquantum C Physics: Quantum computing</td>
<td>namd C++ Biology / Molecular dynamics</td>
</tr>
<tr>
<td>h264ref C Video compression</td>
<td>dealII C++ Finite element analysis</td>
</tr>
<tr>
<td>omnetpp C++ Discrete event simulation</td>
<td>soplex C++ Linear programming, optimization</td>
</tr>
<tr>
<td>astar C++ Path-finding algorithms</td>
<td>povray C++ Image ray-tracing</td>
</tr>
<tr>
<td>xalancbmk C++ XML processing</td>
<td>calculix C/Fortran Structural mechanics</td>
</tr>
<tr>
<td></td>
<td>GemsFDTD Fortran Computational electromagnetics</td>
</tr>
<tr>
<td></td>
<td>tonto Fortran Quantum chemistry</td>
</tr>
<tr>
<td></td>
<td>ibm C Fluid dynamics</td>
</tr>
<tr>
<td></td>
<td>wrf C/Fortran Weather prediction</td>
</tr>
<tr>
<td></td>
<td>sphinx3 C Speech recognition</td>
</tr>
</tbody>
</table>
## SPEC CPU Benchmark (5)

### CINT2006 for Opteron X4 2356

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>$IC \times 10^9$</th>
<th>CPI</th>
<th>$T_c$ (ns)</th>
<th>Exec time</th>
<th>Ref time</th>
<th>SPECratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>perl</td>
<td>Interpreted string processing</td>
<td>2,118</td>
<td>0.75</td>
<td>0.40</td>
<td>637</td>
<td>9,770</td>
<td>15.3</td>
</tr>
<tr>
<td>bzip2</td>
<td>Block-sorting compression</td>
<td>2,389</td>
<td>0.85</td>
<td>0.40</td>
<td>817</td>
<td>9,650</td>
<td>11.8</td>
</tr>
<tr>
<td>gcc</td>
<td>GNU C Compiler</td>
<td>1,050</td>
<td>1.72</td>
<td>0.40</td>
<td>724</td>
<td>8,050</td>
<td>11.1</td>
</tr>
<tr>
<td>mcf</td>
<td>Combinatorial optimization</td>
<td>336</td>
<td>10.00</td>
<td>0.40</td>
<td>1,345</td>
<td>9,120</td>
<td>6.8</td>
</tr>
<tr>
<td>go</td>
<td>Go game (AI)</td>
<td>1,658</td>
<td>1.09</td>
<td>0.40</td>
<td>721</td>
<td>10,490</td>
<td>14.6</td>
</tr>
<tr>
<td>hmmer</td>
<td>Search gene sequence</td>
<td>2,783</td>
<td>0.80</td>
<td>0.40</td>
<td>890</td>
<td>9,330</td>
<td>10.5</td>
</tr>
<tr>
<td>sjeng</td>
<td>Chess game (AI)</td>
<td>2,176</td>
<td>0.96</td>
<td>0.40</td>
<td>837</td>
<td>12,100</td>
<td>14.5</td>
</tr>
<tr>
<td>libquantum</td>
<td>Quantum computer simulation</td>
<td>1,623</td>
<td>1.61</td>
<td>0.40</td>
<td>1,047</td>
<td>20,720</td>
<td>19.8</td>
</tr>
<tr>
<td>h264avc</td>
<td>Video compression</td>
<td>3,102</td>
<td>0.80</td>
<td>0.40</td>
<td>993</td>
<td>22,130</td>
<td>22.3</td>
</tr>
<tr>
<td>omnetpp</td>
<td>Discrete event simulation</td>
<td>587</td>
<td>2.94</td>
<td>0.40</td>
<td>690</td>
<td>6,250</td>
<td>9.1</td>
</tr>
<tr>
<td>astar</td>
<td>Games/path finding</td>
<td>1,082</td>
<td>1.79</td>
<td>0.40</td>
<td>773</td>
<td>7,020</td>
<td>9.1</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>XML parsing</td>
<td>1,058</td>
<td>2.70</td>
<td>0.40</td>
<td>1,143</td>
<td>6,900</td>
<td>6.0</td>
</tr>
</tbody>
</table>

**Geometric mean**

| 11.7 |
SPEC Power Benchmark (1)

- **SPECpower_ssj2008**
  - The first industry-standard SPEC benchmark for evaluating the power and performance characteristics of server class computers
  - Initially targets the performance of server-side Java
  - Power consumption of server at different workload levels (0% ~ 100%)
    - Performance: \( \text{ssj\_ops/sec} \)
    - Power: Watts (Joules/sec)

\[
\text{Overall ssj\_ops per Watt} = \left( \frac{\sum_{i=0}^{10} \text{ssj\_ops}_i}{\sum_{i=0}^{10} \text{power}_i} \right)
\]
### SPEC Power Benchmark (2)

#### SPECpower_ssj2008 for X4 2356

<table>
<thead>
<tr>
<th>Target Load</th>
<th>Actual Load</th>
<th>ssj_ops</th>
<th>Avg. Active Power (W)</th>
<th>Performance to Power Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>99.3%</td>
<td>240,914</td>
<td>299</td>
<td>806</td>
</tr>
<tr>
<td>90%</td>
<td>90.7%</td>
<td>219,979</td>
<td>291</td>
<td>756</td>
</tr>
<tr>
<td>80%</td>
<td>80.1%</td>
<td>194,276</td>
<td>282</td>
<td>690</td>
</tr>
<tr>
<td>70%</td>
<td>70.5%</td>
<td>170,927</td>
<td>271</td>
<td>630</td>
</tr>
<tr>
<td>60%</td>
<td>59.9%</td>
<td>145,299</td>
<td>258</td>
<td>562</td>
</tr>
<tr>
<td>50%</td>
<td>49.5%</td>
<td>120,062</td>
<td>245</td>
<td>490</td>
</tr>
<tr>
<td>40%</td>
<td>40.2%</td>
<td>97,534</td>
<td>232</td>
<td>420</td>
</tr>
<tr>
<td>30%</td>
<td>30.2%</td>
<td>73,199</td>
<td>219</td>
<td>334</td>
</tr>
<tr>
<td>20%</td>
<td>19.9%</td>
<td>48,386</td>
<td>207</td>
<td>233</td>
</tr>
<tr>
<td>10%</td>
<td>9.8%</td>
<td>23,819</td>
<td>197</td>
<td>121</td>
</tr>
<tr>
<td>Active Idle</td>
<td></td>
<td>0</td>
<td>178</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \frac{\sum_{\text{ssj\_ops}}}{\sum_{\text{power}}} = 498 \]
SPEC Power Benchmark (3)

- **Low power at idle?**
  - Look back at X4 power benchmark
    - At 100% load: 299W
    - At 50% load: 245W (82%)
    - At 10% load: 180W (60%)

- **Google data center**
  - Mostly operates at 10% – 50% load
  - At 100% load less than 1% of the time

- **Designing processors to make power proportional to load?**
Other Benchmarks

- **EEMBC**
  - Applications on embedded systems such as communication devices, automobiles, etc.

- **Mediabench**
  - Set of multimedia applications (codecs, graphics, ...)

- **NAS**
  - Parallel benchmarks from NASA

- **SPLASH, PARSEC**
  - Multithreaded benchmarks for multiprocessors
Amdahl’s Law (1)

- Execution time after improvement

\[ T_{improved} = \frac{T_{affected}}{\text{Improvement factor}} + T_{unaffected} \]

\[ = T_{original} \times ((1 - f) + f / S) \]

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to run a program 4 times faster?
  - How about making it 5 times faster?
Amdahl’s Law (2)

- **Speedup and Amdahl’s law**

\[ \text{Speedup} = \frac{T_{\text{original}}}{T_{\text{improved}}} = \frac{1}{((1 - f) + f / S)} \]

- **Principles**
  - Make the common case fast
    - As \( f \to 1 \), speedup \( \to S \)
  - Speedup is limited by the fraction of code that can be optimized
    - As \( S \to \infty \), speedup \( \to 1 / (1 - f) \)
  - Uncommon case can become the common one after improvement
Summary

- **Performance is specific to a particular program(s)**
  - Total execution time is a consistent summary of the performance

- **For a given architecture, performance increases come from**
  - Increases in clock rate (without adverse CPI affects)
  - Improvements in processor organization that lower CPI
  - Compiler enhancements that lower CPI and/or instruction count
  - Algorithm/Language choices that affect instruction count

- **Pitfall:**
  - Expecting improvement in one aspect of a machine’s performance to affect the total performance