Agenda
- Review
- Benchmarks and Summarizing Performance
- Adminstrivia
- Technology Break
- Measuring Performance, with Examples

Review
- Time (seconds/program) is measure of performance
- \(\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}\)
- Algorithms, Programming Languages, Compilers, Instruction Set affect Instruction Count and CPI
- Instruction Set affects Clock Period/ Clock Rate
- Benchmarks stand in for real workloads to as standardized measure of relative performance

SPEC (System Performance Evaluation Cooperative)
- Computer Vendor cooperative for benchmarks, started in 1989
- SPECTCPU2006
  - 12 Integer Programs
  - 17 Floating-Point Programs
- Often turn into number where bigger is faster
- SPECratio: reference execution time on old reference computer divide by execution time on new computer

SPECINT2006 on AMD Barcelona

<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction Count (B)</th>
<th>CPI</th>
<th>Clock cycle time (ps)</th>
<th>Execution Time (s)</th>
<th>Reference Time (s)</th>
<th>SPECratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted string processing</td>
<td>2,118 0.75</td>
<td>400</td>
<td>637</td>
<td>9,770</td>
<td>15.3</td>
<td></td>
</tr>
<tr>
<td>Block-sorting compression</td>
<td>2,389 0.85</td>
<td>400</td>
<td>817</td>
<td>9,650</td>
<td>11.8</td>
<td></td>
</tr>
<tr>
<td>GNU C compiler</td>
<td>1,050 1.72</td>
<td>400</td>
<td>724</td>
<td>8,050</td>
<td>11.1</td>
<td></td>
</tr>
<tr>
<td>Combinatorial optimization</td>
<td>336 10.0</td>
<td>400</td>
<td>1,345</td>
<td>9,120</td>
<td>6.8</td>
<td></td>
</tr>
<tr>
<td>Go game</td>
<td>1,658 1.09</td>
<td>400</td>
<td>721</td>
<td>10,490</td>
<td>14.6</td>
<td></td>
</tr>
<tr>
<td>Search gene sequence</td>
<td>2,783 0.80</td>
<td>400</td>
<td>890</td>
<td>9,330</td>
<td>10.5</td>
<td></td>
</tr>
<tr>
<td>Chess game</td>
<td>2,176 0.96</td>
<td>400</td>
<td>837</td>
<td>12,100</td>
<td>14.5</td>
<td></td>
</tr>
<tr>
<td>Quantum computer simulation</td>
<td>1,623 1.61</td>
<td>400</td>
<td>1,047</td>
<td>20,720</td>
<td>19.8</td>
<td></td>
</tr>
<tr>
<td>Video compression</td>
<td>3,102 0.80</td>
<td>400</td>
<td>993</td>
<td>22,130</td>
<td>22.3</td>
<td></td>
</tr>
<tr>
<td>Discrete event simulation library</td>
<td>587 2.94</td>
<td>400</td>
<td>690</td>
<td>6,250</td>
<td>9.1</td>
<td></td>
</tr>
<tr>
<td>Games/path finding</td>
<td>1,082 1.79</td>
<td>400</td>
<td>773</td>
<td>7,020</td>
<td>9.1</td>
<td></td>
</tr>
<tr>
<td>XML parsing</td>
<td>1,058 2.70</td>
<td>400</td>
<td>1,143</td>
<td>6,900</td>
<td>6.0</td>
<td></td>
</tr>
</tbody>
</table>

Summarizing Performance
- Barcelona varies from 6 times to 22 times faster than reference computer
  - Average (Arithmetic Mean) is 12.6, Median is 11.5
- Geometric mean of ratios:
  - N-th root of product of N ratios
  - Geometric Mean gives same relative answer no matter what computer is as reference
  - Geometric Mean for Barcelona is 11.7
SPECPower

- Given rising importance of power and energy, create benchmark for performance and power
- Most servers in Warehouse Scale Computer have avg. utilization between 10% & 50%, so measure power at medium load as well as at high load
- Measure best performance and power, then reduce request rate so that see power for every 10% reduction in performance
- Java server benchmark performance is operations per second (ssj_ops), so metric is ssj_ops/Watt

Energy Proportional Computing


SpecPower on Barcelona

<table>
<thead>
<tr>
<th>Target Load %</th>
<th>Performance (ssj_ops)</th>
<th>Avg. Power (Watts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>231,867</td>
<td>295</td>
</tr>
<tr>
<td>90%</td>
<td>211,282</td>
<td>286</td>
</tr>
<tr>
<td>80%</td>
<td>185,803</td>
<td>275</td>
</tr>
<tr>
<td>70%</td>
<td>163,427</td>
<td>265</td>
</tr>
<tr>
<td>60%</td>
<td>140,160</td>
<td>256</td>
</tr>
<tr>
<td>50%</td>
<td>118,324</td>
<td>246</td>
</tr>
<tr>
<td>40%</td>
<td>92,035</td>
<td>233</td>
</tr>
<tr>
<td>30%</td>
<td>70,500</td>
<td>222</td>
</tr>
<tr>
<td>20%</td>
<td>47,126</td>
<td>206</td>
</tr>
<tr>
<td>10%</td>
<td>23,066</td>
<td>180</td>
</tr>
<tr>
<td>0%</td>
<td>0</td>
<td>141</td>
</tr>
</tbody>
</table>

Sum: 1,283,590, 2,605

SPECPower on Barcelona

Which is Better?

- Five machines running at 10% utilization
  - Total Power =

- One machine running at 50% utilization
  - Total Power =

Agenda

- Review
- Benchmarks and Summarizing Performance
- Administrivia
- Technology Break
- Measuring Performance, with Examples

Other Attempts at Benchmarks

- Rather than run a collection of real programs and take their average (geometric mean), instead create a single program that matches the average behavior of a set of programs
- Called a synthetic benchmark
- First example called Whetstone in 1972 for floating point intensive programs in Fortran
- 2nd example called Dhrystone in 1985 for integer programs in Ada and C
  - Pun on Wet vs. Dry ("Whet" vs. "Dhry")
Dhystone Shortcomings

- Dhystone features unusual code that is not usually representative of real-life programs
- Dhystone susceptible to compiler optimizations
- Dhystone’s small code size means always fits in caches, so not representative
  — See Lecture Wednesday
- Yet still used in hand held, embedded CPUs!

Compiler Optimization and Dhystone

- gcc compiler options
  -O1: the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time
  -O2: Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code
  -O3: Optimize yet more. All -O2 optimizations and also turns on the -ffinite-functions, -funswitch-loops, -fpredictive-commoning, -fsgce-after-reload, -ftree-vectorize and -fipa-cp-clone options

Measuring Time

- UNIX time command measures in seconds
  - *Time Stamp Counter*
    - 64-bit counter of clock cycles on Intel 80x86 instruction set computers
    - 80x86 instruction RDTSC (Read TSC) returns TSC in regs EDX (upper 32 bits) and EAX (lower 32 bits)
    - Can read, but can’t set
    - How long can measure?
    - Measures overall time, not just time for 1 program

How get RDTSC access in C?

```c
static inline unsigned long long RDTSC(void)
{
unsigned hi, lo;
asm volatile("rdtsc" : "=a"(lo), "=d"(hi));
return ((unsigned long long)lo) | ((unsigned long long)hi) <<32);
}
```
Gcc Optimization Experiment

<table>
<thead>
<tr>
<th></th>
<th>BubbleSort.c</th>
<th>Dhrystone.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Opt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where do you spend the time in your program?

- Profiling program shows which where spend time by function, which code uses most of time
  - E.g., gprof
- Usually a 90/10 rule, where 10% of code is responsible for 90% of execution time
  - Or 80/20 rule, where 20% of code responsible for 80% of time

Gprof

- Learn where program spent its time
- Learn functions called while it was executing
  - And which functions call other functions
- 3 steps:
  1. Compile & link program with profiling enabled
     - cc --pg x.c  
  2. Execute program to generate a profile data file
  3. Run gprof to analyze the profile data

Gprof example

<table>
<thead>
<tr>
<th>% time</th>
<th>Cumulative (secs)</th>
<th>Self (secs)</th>
<th>calls</th>
<th>Self ms/call</th>
<th>Total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.18</td>
<td>0.06</td>
<td>0.06</td>
<td>23480</td>
<td>0.00</td>
<td>0.00</td>
<td>find_char_unquote</td>
</tr>
<tr>
<td>12.12</td>
<td>0.10</td>
<td>0.04</td>
<td>120</td>
<td>0.33</td>
<td>0.73</td>
<td>pattern_search</td>
</tr>
<tr>
<td>9.09</td>
<td>0.13</td>
<td>0.03</td>
<td>5120</td>
<td>0.01</td>
<td>0.01</td>
<td>collapse_continuations</td>
</tr>
<tr>
<td>9.09</td>
<td>0.16</td>
<td>0.03</td>
<td>148</td>
<td>0.20</td>
<td>0.88</td>
<td>update_file_1</td>
</tr>
<tr>
<td>9.09</td>
<td>0.19</td>
<td>0.03</td>
<td>37</td>
<td>0.81</td>
<td>4.76</td>
<td>eval</td>
</tr>
<tr>
<td>6.06</td>
<td>0.21</td>
<td>0.02</td>
<td>12484</td>
<td>0.00</td>
<td>0.00</td>
<td>file_hash_1</td>
</tr>
<tr>
<td>6.06</td>
<td>0.23</td>
<td>0.02</td>
<td>6596</td>
<td>0.00</td>
<td>0.00</td>
<td>get_next_mword</td>
</tr>
<tr>
<td>3.03</td>
<td>0.24</td>
<td>0.01</td>
<td>29981</td>
<td>0.00</td>
<td>0.00</td>
<td>hash_find_slot</td>
</tr>
<tr>
<td>3.03</td>
<td>0.25</td>
<td>0.01</td>
<td>14769</td>
<td>0.00</td>
<td>0.00</td>
<td>next_token</td>
</tr>
<tr>
<td>3.03</td>
<td>0.26</td>
<td>0.01</td>
<td>5880</td>
<td>0.00</td>
<td>0.00</td>
<td>variable_expand_string</td>
</tr>
</tbody>
</table>

See http://linuxgazette.net/100/vinayak.html

Cautionary Tale

- More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.
  -- William A. Wulf
- We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
  -- Donald E. Knuth

Summary

- Benchmarks stand in for real workloads to as standardized measure of relative performance
- Synthetic programs don’t work, but some still use them!
- Power of increasing concern, and being added to benchmarks
- Time measurement via clock cycles, machine specific
- Profiling tools as way to see where spending time in your program
- Don’t optimize prematurely!