CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Performance
Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp11

New-School Machine Structures
(It’s a bit more complicated!)

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time

Agenda
- Defining Performance
- Administrivia
- Workloads and Benchmarks
- Technology Break
- Measuring Performance
- Summary

What is Performance?
- Latency (or response time or execution time)
  - Time to complete one task
- Bandwidth (or throughput)
  - Tasks completed per unit time
Running Systems to 100% Utilization

• Implication of the graph at the right?

• Can you explain why this happens?

L = \lambda W

Average number of customers in system (L) = average interarrival rate (\lambda) x average service time (W)

Cloud Performance: Why Application Latency Matters

<table>
<thead>
<tr>
<th>Server Delay (ms)</th>
<th>Increased time to next click (ms)</th>
<th>Quizzes/ user</th>
<th>Any clicks/ user</th>
<th>User satisfaction</th>
<th>Revenue/ User</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

- Longer the delay, the fewer the user clicks, the less the user happiness, and the lower the revenue per user

Defining CPU Performance

• What does it mean to say X is faster than Y?
• Ferrari vs. School Bus?
• 2009 Ferrari 599 GTB
  - 2 passengers, 11.1 secs in quarter mile
• 2009 Type D school bus
  - 54 passengers, quarter mile time?
  - http://www.youtube.com/watch?v=KwyCoQUhUNA
• Response Time/Latency: e.g., time to travel ¼ mile
• Throughput/Bandwidth: e.g., passenger-mi in 1 hour

Google Instant Search “Instant Efficiency”

Typical search takes 24 seconds, Google’s search algorithm is only 300 ms of this
"It’s not search ‘as you type’, but ‘search before you type’!
“We can predict what you are likely to type and give you those results in real time”

Defining Relative CPU Performance

• Performance_x = 1/Program Execution Time_x
• Performance_x > Performance_y => 1/Execution Time_x > 1/Execution Time_y => Execution Time_y > Execution Time_x
• Computer X is N times faster than Computer Y
  - Performance_x / Performance_y = N or
  - Execution Time_y / Execution Time_x = N
• Bus is to Ferrari as 12 is to 11.1: Ferrari is 1.08 times faster than the bus!
Measuring CPU Performance

- Computers use a clock to determine when events take place within hardware
- **Clock cycles**: discrete time intervals
  - aka clocks, cycles, clock periods, clock ticks
- **Clock rate or clock frequency**: clock cycles per second (inverse of clock cycle time)
- 3 GigaHertz clock rate
  \[ \text{clock cycle time} = \frac{1}{(3 \times 10^9)} \text{ seconds} \]
  \[ \text{clock cycle time} = 333 \text{ picoseconds (ps)} \]

CPU Performance Factors

- To distinguish between processor time and I/O, **CPU time** is time spent in processor
- **CPU Time/Program**
  \[ = \text{Clock Cycles/Program} \times \text{Clock Cycle Time} \]
- Or
  \[ \text{CPU Time/Program} = \frac{\text{Clock Cycles/Program}}{\text{Clock Rate}} \]

Restating Performance Equation

\[ \text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}} \]

What Affects Each Component?

<table>
<thead>
<tr>
<th>Hardware or software component?</th>
<th>Affects What?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td></td>
</tr>
<tr>
<td>Programming Language</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td></td>
</tr>
<tr>
<td>Instruction Set Architecture</td>
<td></td>
</tr>
</tbody>
</table>

Peer Instruction Question

- Computer A clock cycle time 250 ps, \( CPI_A = 2 \)
- Computer B clock cycle time 500 ps, \( CPI_B = 1.2 \)
- Assume A and B have same instruction set
- Which statement is true?
  - Red. Computer A is ~1.2 times faster than B
  - Orange. Computer A is ~4.0 times faster than B
  - Green. Computer B is ~1.7 times faster than A
  - Yellow. Computer B is ~3.4 times faster than A
  - Pink. None of the above
Agenda

• Defining Performance
• Administrivia
• Workloads and Benchmarks
• Technology Break
• Measuring Performance
• Summary

SPEC
(System Performance Evaluation Cooperative)

• Computer Vendor cooperative for benchmarks, started in 1989
• SPECCPU2006
  – 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 to get an effective speed-up

SPECINT2006 on AMD Barcelona

<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction Count (B)</th>
<th>CPI</th>
<th>Clock cycle (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>9,560</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 ...

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

Which system is faster?

... Depends Who’s Selling

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>20</td>
<td>15</td>
</tr>
</tbody>
</table>

Average throughput

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.50</td>
<td>2.00</td>
<td>1.25</td>
</tr>
<tr>
<td>B</td>
<td>1.00</td>
<td>0.50</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Throughput relative to B

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.50</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>B</td>
<td>1.00</td>
<td>0.50</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Throughput relative to A

Summarizing SPEC Performance

- Varies from 6x to 22x faster than reference computer
- Geometric mean of ratios: 
  \[ \sqrt[N]{\prod_{i=1}^{N} \text{Execution time ratios}} \]
  - Geometric Mean gives same relative answer no matter what computer is used as reference
- Geometric Mean for Barcelona is 11.7

Energy and Power

(Energy = Power x Time)

- Energy to complete operation (Joules)
  - Correlates approximately to battery life
- Peak power dissipation (Watts = Joules/s)
  - Affects heat (and cooling demands)
  - IT equipment’s power is in the denominator of the Power Utilization Efficiency (PUE) equation, a WSC figure of merit

Peak Power vs. Lower Energy

(Power x Time = Energy)

- Which system has higher peak power?
- Which system has higher energy?

Energy Proportional Computing

"The Case for Energy-Proportional Computing,"
Lyü André Barroso, Urs Höflie, *IEEE Computer* December 2007

It is surprisingly hard to achieve high levels of utilization of typical servers (and your home PC or laptop is even worse)

Figure 1: Average CPU utilization of more than 5,000 servers during a 10-year period. Servers are rarely completely idle and seldom operate near their maximum efficiency, instead operating most of the time at between 10 and 50 percent of their maximum.
SPECPower

- Increasing importance of power and energy: create benchmark for performance and power
- Most servers in WSCs have average utilization between 10% & 50%, so measure power at medium as well as at high load
- Measure best performance and power, then step down request rate to measure power for every 10% reduction in performance
- Java server benchmark performance is operations per second (ssj_ops), so metric is ssj_ops/Watt

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>
<tr>
<td>Sum</td>
<td>1,283,590</td>
<td>2,605</td>
</tr>
<tr>
<td>ssj_ops/Watt</td>
<td></td>
<td>493</td>
</tr>
</tbody>
</table>

Which is Better?
(1 Red Machine vs. 5 Green Machines)

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

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

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
- Yet still used in hand held, embedded CPUs!

Other Benchmark Attempt

- Rather than run a collection of real programs and take their average (geometric mean), 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
- Second 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
- Yet still used in hand held, embedded CPUs!

Agenda

- Defining Performance
- Administrivia
- Workloads and Benchmarks
- Technology Break
- Measuring Performance
- Summary
Compiler Optimization and Dhrystone

- 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 `-ffinalize-functions`, `-funswitch-loops`, `-fpredictive-commoning`, `-fgcse-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 to 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) lo) | ((unsigned 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 a program (e.g., using, `gprof`) shows where it spends its time by function, so you can determine which code consumes most of the execution time.
- Usually a 90/10 rule: 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
- Three steps:
  - Compile & link program with profiling enabled
    - `cc -pg x.c` (in addition to other flags use)
  - Execute program to generate a profile data file
  - Run `gprof` to analyze the profile data

### Test Program to Profile with Saturn

```c
#include <math.h>
#define LIMIT 500000000

void sinfun() {
    double a;
    int i;
    double e;
    int j;
    for (j=1; j < LIMIT; j++)
        a = sin(i/1000.0);
    int main() {
        exponential();
        sinfun();
        return 0;
    }
}
```

(Unfortunately `gprof` isn't supported on my Intel-based mac with mac os; I use an alternative tool called `saturn`).

### gprof example

<table>
<thead>
<tr>
<th>% time</th>
<th>Cumulative (secs)</th>
<th>Self (secs)</th>
<th>calls</th>
<th>Self/mol</th>
<th>Total/mol</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>86-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>14799</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>5890</td>
<td>0.00</td>
<td>0.00</td>
<td>variable_expand_string</td>
</tr>
</tbody>
</table>

See [http://linuxgazette.net/100/vinayak.html](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

And In Conclusion, ...

- Time (seconds/program) is measure of performance
  \[
  \text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}
  \]
- Benchmarks stand in for real workloads to as standardized measure of relative performance
- 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!