Evaluating Computers: Bigger, better, faster, more?
What do you want in a computer?
What do you want in a computer?

- Low latency -- one unit of work in minimum time
  - 1/latency = responsiveness
- High throughput -- maximum work per time
  - High bandwidth (BW)
- Low cost
- Low power -- minimum jules per time
- Low energy -- minimum jules per work
- Reliability -- Mean time to failure (MTTF)
- Derived metrics
  - responsiveness/dollar
  - BW/$
  - BW/Watt
  - Work/Jule
  - Energy * latency -- Energy delay product
  - MTTF/$
Latency

• This is the simplest kind of performance
• How long does it take the computer to perform a task?
  • The task at hand depends on the situation.
• Usually measured in seconds
• Also measured in clock cycles
  • Caution: if you are comparing two different system, you must ensure that the cycle times are the same.
Measuring Latency

- Stop watch!
- System calls
  - `gettimeofday()`
  - `System.currentTimeMillis()`
- Command line
  - `time <command>`
Where latency matters

• Application responsiveness
  • Any time a person is waiting.
  • GUIs
  • Games
  • Internet services (from the users perspective)
• “Real-time” applications
  • Tight constraints enforced by the real world
  • Anti-lock braking systems
  • Manufacturing control
  • Multi-media applications
• The cost of poor latency
  • If you are selling computer time, latency is money.
Latency and Performance

- By definition:
  - Performance = 1/Latency
- If Performance(X) > Performance(Y), X is faster.
- If Perf(X)/Perf(Y) = S, X is S times faster than Y.
- Equivalently: Latency(Y)/Latency(X) = S

- When we need to talk about specifically about other kinds of “performance” we must be more specific.
The Performance Equation

• We would like to model how architecture impacts performance (latency)
• This means we need to quantify performance in terms of architectural parameters.
  • Instructions -- this is the basic unit of work for a processor
  • Cycle time -- these two give us a notion of time.
  • Cycles

• The first fundamental theorem of computer architecture:

  \[ \text{Latency} = \text{Instructions} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}} \]
The Performance Equation

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

• The units work out! Remember your dimensional analysis!
• Cycles/Instruction == CPI
• Seconds/Cycle == 1/Hz
• Example:
  • 1 GHz clock
  • 1 billion instructions
  • CPI = 4
  • What is the latency?
Examples

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

• gcc runs in 100 sec on a 1 GHz machine
  – How many cycles does it take?

    100G cycles

• gcc runs in 75 sec on a 600 MHz machine
  – How many cycles does it take?

    45G cycles
How can this be?

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

• Different Instruction count?
  • Different ISAs?
  • Different compilers?
• Different CPI?
  • underlying machine implementation
  • Microarchitecture
• Different cycle time?
  • New process technology
  • Microarchitecture
Computing Average CPI

• Instruction execution time depends on instruction time (we’ll get into why this is so later on)
  • Integer +, -, <<, |, & -- 1 cycle
  • Integer *, /, -- 5-10 cycles
  • Floating point +, - -- 3-4 cycles
  • Floating point *, /, sqrt() -- 10-30 cycles
  • Loads/stores -- variable
  • All theses values depend on the particular implementation, not the ISA

• Total CPI depends on the workload’s Instruction mix -- how many of each type of instruction executes
  • What program is running?
  • How was it compiled?
The Compiler’s Role

• Compilers affect CPI…
  • Wise instruction selection
    • “Strength reduction”: $x \times 2^n \rightarrow x \ll n$
    • Use registers to eliminate loads and stores
  • More compact code -> less waiting for instructions
• …and instruction count
  • Common sub-expression elimination
  • Use registers to eliminate loads and stores
Stupid Compiler

```plaintext
int i, sum = 0;
for (i=0; i<10; i++)
    sum += i;
```

<table>
<thead>
<tr>
<th>Type</th>
<th>CPI</th>
<th>Static #</th>
<th>dyn #</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>5</td>
<td>6</td>
<td>42</td>
</tr>
<tr>
<td>int</td>
<td>1</td>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>br</td>
<td>1</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td>2.8</td>
<td>11</td>
<td>92</td>
</tr>
</tbody>
</table>

\[(5 \times 42 + 1 \times 30 + 1 \times 20)/92 = 2.8\]
int i, sum = 0;
for(i=0; i<10; i++)
    sum += i;

code:
add $1, $0, $0 # i
add $2, $0, $0 # sum
loop:
    sub $3, $1, 10
    beq $3, $0, end
    add $2, $2, $1
    addi $1, $1, 1
    b loop
end:
    sw 0($sp), $2

<table>
<thead>
<tr>
<th>Type</th>
<th>CPI</th>
<th>Static #</th>
<th>dyn #</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td>1</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>br</td>
<td>1</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td>1.01</td>
<td>8</td>
<td>53</td>
</tr>
</tbody>
</table>

\[(5 \times 1 + 1 \times 32 + 1 \times 20)/53 = 2.8\]
Live demo
Program inputs affect CPI too!

```c
int rand[1000] = {random 0s and 1s }
for(i=0;i<1000;i++)
    if(rand[i]) sum -= i;
    else sum *= i;

int ones[1000] = {1, 1, ...}
for(i=0;i<1000;i++)
    if(ones[i]) sum -= i;
    else sum *= i;
```

- Data-dependent computation
- Data-dependent micro-architectural behavior
  - Processors are faster when the computation is predictable (more later)
Live demo
Making Meaningful Comparisons

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

- Meaningful CPI exists only:
  - For a particular program with a particular compiler
  - ....with a particular input.

- You MUST consider all 3 to get accurate latency estimations or machine speed comparisons
  - Instruction Set
  - Compiler
  - Implementation of Instruction Set (386 vs Pentium)
  - Processor Freq (600 Mhz vs 1 GHz)
  - Same high level program with same input

- “wall clock” measurements are always comparable.
  - If the workloads (app + inputs) are the same
The Performance Equation

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

- Clock rate =
- Instruction count =
- Latency =
- Find the CPI!