Today

• Announcements
  • 1 week extension on project.
  • 1 week extension on Lab 3 for 141L.
• Measuring performance
• Return quiz #1
Evaluating Computers: Bigger, better, faster, more?
Key Points

• What does it mean for a computer to be fast?
• What is latency?
• What is the performance equation?
What do you want in a computer?

- Reliability
- Runs programs quickly
- frames/s @ max settings
- Lower power
- Awesomeness
- Small or volume
- temperature
- Large monitor
- light
- cheap

- quiet
- efficient, but how?
- Fast startup
- keep it busy
- Secure
- Backward compatibility
- Network speed
  - throughput
  - Latency
- Lots of memory
- Convenience
What do you want in a computer?

• Low latency -- one unit of work in minimum time
  • $1/\text{latency} = \text{responsiveness}$
• 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.

\[
\text{Mhz} = \text{cycles/second} \\
\text{Cycle time} = \text{seconds/cycle} \\
\text{Latency} = (\text{seconds/cycle}) \times \text{cycles} = \text{seconds}
\]
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 -- “hard” real time
  • Manufacturing control
  • Multi-media applications -- “soft” real time
• 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 per instructions

• The first fundamental theorem of computer architecture:

Latency = Instructions * Cycles/Instruction * Seconds/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?
What can impact latency?

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
“Dynamic” and “static”

• Static
  • Fixed at compile time or referring to the program as it was compiled
  • ex: The compiled version of that function contains 10 static instructions.

• dynamic
  • having to do with the execution of the program or counted at run time
  • ex: When I ran that program it executed 1 million dynamic instructions.
  • ex: “dynamic instance of an instructions” is one particular execution of a particular static instruction.

• The instruction count in the performance equation in dynamic!
Impacts on Instruction count

• The program itself
  • Your program may do more or less work.

• The inputs to the program
  • e.g., larger data sets

• Compiler optimizations
  • Common sub-expression elimination
  • Use registers to eliminate loads and stores
X86 Examples

- http://cseweb.ucsd.edu/classes/will/cse141/x86/
Computing Average CPI

• Instruction execution time depends on instruction type (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 -- varies
  • 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 Impact on CPI

- 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 \(\rightarrow\) less waiting for instructions
- …and instruction count
  - Common sub-expression elimination
  - Use registers to eliminate loads and stores
Impacts on CPI

• Biggest contributor: Micro architectural implementation
  • More on this later.

• Other contributors
  • Program inputs
    • can change the cycles required for a particular dynamic instruction
  • Instruction mix
    • since different instructions take different numbers of cycles
    • Floating point divide always takes more cycles than an integer add.
```c
int i, sum = 0;
for(i=0; i<10; i++)
    sum += i;
```

```assembly
sw 0($sp), $0 #sum = 0
sw 4($sp), $0 #i = 0
loop:
    lw $1, 4($sp)
    sub $3, $1, 10
    beq $3, $0, end
    lw $2, 0($sp)
    add $2, $2, $1
    st 0($sp), $2
    addi $1, $1, 1
    st 4($sp), $1
    b loop
end:
```

<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>

\[(5 \times 42 + 1 \times 30 + 1 \times 20)/92 = 2.8\]
Smart Compiler

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

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>

\[
\frac{(5 \times 1 + 1 \times 32 + 1 \times 20)}{53} = 1.01
\]
**Live demo**

- [http://cseweb.ucsd.edu/classes/wi11/cse141/x86/](http://cseweb.ucsd.edu/classes/wi11/cse141/x86/)
- arrayloop.c

<table>
<thead>
<tr>
<th></th>
<th>Static inst</th>
<th>dynamic inst</th>
</tr>
</thead>
<tbody>
<tr>
<td>no opt</td>
<td>20</td>
<td>1.2M inst</td>
</tr>
<tr>
<td>opt -O1</td>
<td>17</td>
<td>741 K inst</td>
</tr>
<tr>
<td>Opt -O4</td>
<td>17</td>
<td>752 K inst</td>
</tr>
</tbody>
</table>
Program inputs and CPI

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
Impacts on Cycle time

- Microarchitectural implementation
  - More on this later
- Process technology
  - Moore’s law continues to speed up transistors
  - For a *fixed design* the cycle time will drop as it is “shrunk” from one process generation to the next.
Fun Diversion

• How many instructions in HelloWord?

<table>
<thead>
<tr>
<th>Language</th>
<th>ranking guess</th>
<th>inst count</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1+++</td>
<td>250 k</td>
<td>1</td>
</tr>
<tr>
<td>Java</td>
<td>5 or 2</td>
<td>30 M</td>
<td>5</td>
</tr>
<tr>
<td>perl</td>
<td>2 4</td>
<td>1.6 M</td>
<td>3</td>
</tr>
<tr>
<td>shell</td>
<td>1</td>
<td>319k or 867k</td>
<td>2</td>
</tr>
<tr>
<td>Python</td>
<td>3</td>
<td>15M</td>
<td>4</td>
</tr>
</tbody>
</table>
Limits on Speedup: Amdahl’s Law

• “The fundamental theorem of performance optimization”
• Coined by Gene Amdahl (one of the designers of the IBM 360)
• Optimizations do not (generally) uniformly affect the entire program
  – The more widely applicable a technique is, the more valuable it is
  – Conversely, limited applicability can (drastically) reduce the impact of an optimization.

Always heed Amdahl’s Law!!!
It is central to many many optimization problems
Amdahl’s Law in Action

• SuperJPEG-O-Rama2010 ISA extensions
  **
  – Speeds up JPEG decode by 10x!!!
  – Act now! While Supplies Last!

**
  Increases processor cost by 45%
Amdahl’s Law in Action

• SuperJPEG-O-Rama2010 in the wild
• PictoBench spends 33% of it’s time doing JPEG decode
• How much does JOR2k help?

<table>
<thead>
<tr>
<th></th>
<th>w/o JOR2k</th>
<th>w/ JOR2k</th>
</tr>
</thead>
<tbody>
<tr>
<td>#SBATCH</td>
<td>30s</td>
<td>21s</td>
</tr>
<tr>
<td>JPEG Decode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Performance: \( \frac{30}{21} = 1.4x \) Speedup ≠ 10x

Is this worth the 45% increase in cost?
• The second fundamental theorem of computer architecture.
• If we can speed up $X$ of the program by $S$ times
• Amdahl’s Law gives the total speed up, $S_{tot}$

$$S_{tot} = \frac{1}{(x/S + (1-x))}.$$ 

Sanity check:

$x = 1 \implies S_{tot} = \frac{1}{(1/S + (1-1))} = \frac{1}{1/S} = S$
Amdahl’s Corollary #1

- Maximum possible speedup, $S_{max}$

$$S = \text{infinity}$$

$$S_{max} = \frac{1}{1-x}$$
Amdahl’s Law Example #1

• Protein String Matching Code
  – 200 hours to run on current machine, spends 20% of time doing integer instructions
  – How much faster must you make the integer unit to make the code run 10 hours faster?
  – How much faster must you make the integer unit to make the code run 50 hours faster?

A) 1.1  E) 10.0
B) 1.25  F) 50.0
C) 1.75  G) 1 million times
D) 1.33  H) Other