Evaluating Computers:
Defining what it means to be “better.”
Key Points

- What does it mean for a computer to be fast?
- What is latency?
- What is the performance equation?
- What is Amdahl’s Law?
A hypothetical trip to Best Buy...

What do YOU want in your next computer?
Important traits in a computer

- Crysis -- Frame rate
- Responsiveness
- Real time -- deadlines, responsiveness
- Operations/sec -- throughput
- Cost
- Volume
- Weight
- Battery life
- Power/energy consumption
- Storage size
- Reliability
- Latency -- how long does it take?
- Happiness
- Usability
- Flexibility
- Shininess
- Connectivity -- interesting IO
- Ruggedness
Metrics to compare traits

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

• Latency is the simplest kind of performance
• How long does it take the computer to perform a task?
  • Always need to specify the task
• Usually measured in (milli/micro/nano)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
  • GUIs
  • Games
  • Internet services (from the user’s perspective)
  • Any time a person is waiting

• “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)
Latency and Performance

• Definition: performance is the inverse of latency
  • Performance = 1/Latency

• If X is faster, Perf(X) > Perf(Y)

• If X is S times faster than Y, Perf(X)/Perf(Y) = S
  • Equivalently: Latency(Y)/Latency(X) = S

• Unless specified, this definition is what is assumed when you talk about “performance”
The Performance Equation

- We would like to model how architecture impacts performance (i.e. latency)
- This means we need to quantify performance in terms of architectural parameters.
  - **Instructions**: these are the basic unit of work for a processor
  - **Cycles**: the number of clock cycles required
  - **Cycle time**: the length of time for a clock cycle

- Latency equation for simple machines:

  \[ \text{Latency} = \text{Instructions} \times \left( \frac{\text{Cycles}}{\text{Instruction}} \right) \div \text{Freq} \]
The Performance Equation

Latency = Instructions \* Cycles/Instruction \* Seconds/Cycle

- Do the units work out?
- Cycles/Instruction == CPI
- Seconds/Cycle == 1/hz == 1 / Freq
- Example:
  - 1 GHz clock freq. (cycle time = ______? )
  - 1 billion instructions
  - CPI = 4
  - What is the latency?
Be careful when comparing systems running the same program

Machine A has the same frequency as Machine B; and has a lower CPI, but the program runs slower. Why?

Latency = Instructions * Cycles/Instruction * Seconds/Cycle
Comparing the program latency of two different systems

Latency = Instructions * Cycles/Instruction * Seconds/Cycle

- Different Instruction count:
  - ISA
  - Compiler / Compiler options

- Different CPI:
  - Underlying machine implementation
  - Microarchitecture

- Different cycle time:
  - New process technology
  - Microarchitecture
Computing Average CPI

- 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?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Integer or Floating Point</th>
<th>Typical Latency in Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>+,-,&lt;&lt;,</td>
<td>, &amp;</td>
<td>Int</td>
</tr>
<tr>
<td>/</td>
<td>Int</td>
<td>32</td>
</tr>
<tr>
<td>*</td>
<td>Int</td>
<td>2</td>
</tr>
<tr>
<td>+ - *</td>
<td>FP</td>
<td>4</td>
</tr>
<tr>
<td>/ Sqrt</td>
<td>FP</td>
<td>12</td>
</tr>
<tr>
<td>Loads, Stores</td>
<td>Int or FP</td>
<td>3,8,30, or 200</td>
</tr>
</tbody>
</table>
The Compiler’s Role

Latency = Instructions * CPI * Cycle Time

• Compilers influence CPI…
  • Wise instruction selection
    • “Strength reduction”: \( \frac{x}{2^n} \rightarrow x \gg n \)
    • 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

```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>

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

(5*1 + 1*32 + 1*20)/53 = 1.01
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 microarchitectural behavior
  – Processors are faster when the computation is predictable (more later)
Making Meaningful Comparisons

Latency = Instructions * CPI * Cycle Time

• For comparisons, meaningful CPI exists only:
  • For a particular program
  • ... with a particular compiler
  • ... with a particular input.

• You MUST consider all 3 latency components 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
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
A Special Offer for You!

• SuperJPEG-O-Rama2011 ISA extensions **
  – Speeds up JPEG decode by 10x!!!
  – Act now! While Supplies Last!

** Increases processor cost by 45%
Amdahl’s Law to the Rescue!

- SuperJPEG-O-Rama2011 in the wild
- PictoBench spends 33% of its time doing JPEG decode
- How much does JOR2k help?

\[
\begin{array}{c}
\text{w/o JOR2k} \\
\text{30s}
\end{array}
\begin{array}{c}
\text{JPEG Decode} \\
\text{21s}
\end{array}
\begin{array}{c}
\text{w/ JOR2k} \\
\text{Amdahl ate our Speedup!}
\end{array}
\]

\[
\text{Speedup: } 30/21 = 1.4x \quad \text{Speedup != 10x}
\]

Is this worth the 45% increase in cost?
Amdahl’s Law

• If we can speed up $X$ of the program by $S$ times:

  Amdahl’s Law gives the total speed up, $S_{tot}$

  \[ S_{tot} = \frac{1}{\left(\frac{x}{S} + (1-x)\right)} \]

Sanity check:

\[ x = 1 \Rightarrow S_{tot} = \frac{1}{\left(\frac{1}{S} + (1-1)\right)} = \frac{1}{1/S} = S \]
Amdahl’s Corollary #1

• Maximum possible speedup, $S_{max}$

$$S = \text{infinity}$$

$$S_{max} = \frac{1}{1-X}$$

*Maximum speedup limited only by the fraction of time spent in region being optimized*
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 reduce latency by 10 hours?
  - How many times faster must you make the integer unit to make the code run 50 hours faster?

A) 1.25
B) 1.33
C) 10.0
D) impossible!

\[ S_{tot} = \frac{1}{\left(\frac{x}{S} + (1-x)\right)} \]
Amdahl’s Law Example #2

- Protein String Matching Code
  - 4 hours execution time on current machine
    - 20% of time doing integer instructions
    - 35% percent of time doing I/O
  - Which option is the best?
    - Option A: Compiler optimization that reduces number of integer instructions by 25% (assume each integer instruction takes the same amount of time)
    - Option B: Hardware optimization that reduces the latency of each IO operations from 6us to 5us.
Amdahl’s Law Applies All Over

- SSDs use 10x less power than HDs
- But they only save you ~50% overall.
Amdahl’s Corollary #2

• Make the common case fast (i.e., x should be large)!
  – *Common == “most time consuming”*
    • Not necessarily “most frequent”
  – The uncommon case doesn’t make much difference
  – Be sure of what the common case is
  – The common case changes.

• Repeat…
  – With optimization, the common becomes uncommon and vice versa.
Amdahl’s Corollary #2: Example

Common case

- 7x => 1.4x
- 4x => 1.3x
- 1.3x => 1.1x

Total = 20/10 = 2x

- In the end, there is no common case!
- Options:
  - Global optimizations (e.g. faster clock, better compiler)
  - Find something common to work on (e.g. memory latency)
  - Total redesign
Amdahl’s Corollary #3

- Benefits of parallel processing
- \( p \) processors
- \( x\% \) is \( p \)-way parallelizable
- maximum speedup, \( S_{par} \)

\[
S_{par} = \frac{1}{(x/p + (1-x))}
\]

\( x \) is pretty small for desktop applications, even for \( p = 2 \)

Does Intel’s 80-core processor make much sense?
Example #3

• Recent advances in process technology have doubled the number transistors you can fit on your die.

• Currently, your key customer can use up to 16 processors for 50% of their application
  – Can only use 1 processor the other 50%

• You have two options:
  – Option A: Double the number of processors from 2 to 4
  – Option B: Stick with 2 processors but add features that will allow the applications to use two cores for 70% of execution.

• Which will you choose?