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
  - $1/\text{latency} = \text{responsiveness}$
- 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 \ast 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:

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:
  - 1GHz 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
int i, sum = 0;
for(i=0; i<10; i++)
    sum += i;

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>

\(\frac{5\times42 + 1\times30 + 1\times20}{92} = 2.8\)
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 = 2.8
Live demo
Program inputs affect CPI too!

```
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!
Today

- DRAM
- Quiz 1 recap
- HW 1 recap
- Questions about ISAs
- More about the project?
- Amdahl’s law
Key Points

- Amdahl’s law and how to apply it in a variety of situations
- Its role in guiding optimization of a system
- Its role in determining the impact of localized changes on the entire system

●
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-Rama2000 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-Rama2000 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>JOR2k</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG Decode</td>
<td>30s</td>
<td>21s</td>
</tr>
<tr>
<td>Total</td>
<td>30s</td>
<td>21s</td>
</tr>
</tbody>
</table>
Amdahl’s Law in Action

- SuperJPEG-O-Rama2000 in the wild
- PictoBench spends 33% of its time doing JPEG decode
- How much does JOR2k help?

Performance: $\frac{30}{21} = 1.4x$ Speedup != 10x
Amdahl’s Law in Action

• SuperJPEG-O-Rama2000 in the wild
• PictoBench spends 33% of it’s time doing JPEG decode
• How much does JOR2k help?

Performance: \[
\frac{30}{21} = 1.4x \text{ Speedup} \neq 10x
\]

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

- SuperJPEG-O-Rama2000 in the wild
- PictoBench spends 33% of its time doing JPEG decode
- How much does JOR2k help?

Performance: $\frac{30}{21} = 1.4x$ Speedup != 10x

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

• 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, $S_{tot}$

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

• 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, $S_{tot}$

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

Sanity check:

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

- Maximum possible speedup, $S_{\text{max}}$

\[ S = \text{infinity} \]

\[ S_{\text{max}} = \frac{1}{(1-x)} \]
Amdahl’s Law Practice

• 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
B) 1.25
C) 1.75
D) 1.33
E) 10.0
F) 50.0
G) 1 million times
H) Other
Amdahl’s Law Practice

- Protein String Matching Code
  - 4 days ET on current machine
    - 20% of time doing integer instructions
    - 35% percent of time doing I/O
  - Which is the better economic tradeoff?
    - Compiler optimization that reduces number of integer instructions by 25% (assume each integer inst takes the same amount of time)
    - Hardware optimization that makes I/O run 20% faster?
Amdahl's Law Applies All Over

- SSDs use 10x less power than HDs
- But they only save you ~50% overall.
Amdahl’s Law in Memory

- Storage array 90% of area
- Row decoder 4%
- Column decode 2%
- Sense amps 4%

- What’s the benefit of reducing bit size by 10%?
- Reducing column decoder size by 90%?
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
Amdahl’s Corollary #2: Example

Common case

7x => 1.4x
Amdahl’s Corollary #2: Example

Common case

- 7x => 1.4x
- 4x => 1.3x
Amdahl’s Corollary #2: Example

Common case

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

Total = 20/10 = 2x
Amdahl’s Corollary #2: Example

In the end, there is no common case!

Options:
- Global optimizations (faster clock, better compiler)
- Find something common to work on (i.e. memory latency)
- War of attrition
- Total redesign (You are probably well-prepared for this)
Amdahl’s Corollary #3

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

\[
S_{par} = \frac{1}{\left(\frac{x}{p} + (1-x)\right)}.
\]
Amdahl’s Corollary #3

- Benefits of parallel processing
- $p$ processors
- $x\%$ is $p$-way parallizable
- maximum speedup, $S_{par}$

$$S_{par} = \frac{1}{(x/p + (1-x))}.$$  

$x$ is pretty small for desktop applications, even for $p = 2$
Amdahl’s Corollary #3

- Benefits of parallel processing
- \( p \) processors
- \( x\% \) is \( p \)-way parallizable
- 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?
Amdahl’s Corollary #4

• Amdahl’s law for latency (L)

\[ L_{\text{new}} = L_{\text{base}} \times \frac{1}{\text{Speedup}} \]
\[ L_{\text{new}} = L_{\text{base}} \times \left(\frac{x}{S} + (1-x)\right) \]
\[ L_{\text{new}} = \left(\frac{L_{\text{base}}}{S}\right)x + ET_{\text{base}} \times (1-x) \]

• If you can speed up y% of the remaining (1-x), you can apply Amdahl’s law recursively

\[ L_{\text{new}} = \left(\frac{L_{\text{base}}}{S_1}\right)x + \]
\[ \left(\frac{S_{\text{base}} \times (1-x)}{S_2}y + L_{\text{base}} \times (1-x) \times (1-y)\right) \]

• This is how we will analyze memory system performance
Amdahl’s Non-Corollary

• Amdahl’s law does not bound slowdown
  \[ L_{\text{new}} = \left( \frac{L_{\text{base}}}{S} \right) x + L_{\text{base}} (1-x) \]
  
• \( L_{\text{new}} \) is linear in \( 1/S \)

• Example: \( x = 0.01 \) of execution, \( L_{\text{base}} = 1 \)
  - \( S = 0.001; \)
    - \( E_{\text{new}} = 1000L_{\text{base}} \times 0.01 + L_{\text{base}} \times (0.99) \sim 10L_{\text{base}} \)
  - \( S = 0.00001; \)
    - \( E_{\text{new}} = 100000L_{\text{base}} \times 0.01 + L_{\text{base}} \times (0.99) \sim 1000L_{\text{base}} \)

• Things can only get so fast, but they can get arbitrarily slow.
  - Do not hurt the non-common case too much!
Benchmarks: Standard Candles for Performance

• It’s hard to convince manufacturers to run your program (unless you’re a BIG customer)

• A benchmark is a set of programs that are representative of a class of problems.

• To increase predictability, collections of benchmark applications, called benchmark suites, are popular
  – “Easy” to set up
  – Portable
  – Well-understood
  – Stand-alone
  – Standardized conditions
  – These are all things that real software is not.
Classes of benchmarks

- **Microbenchmark** – measure one feature of system
  - e.g. memory accesses or communication speed
- **Kernels** – most compute-intensive part of applications
  - e.g. Linpack and NAS kernel b’marks (for supercomputers)
- **Full application:**
  - SpecInt / SpecFP (int and float) (for Unix workstations)
  - Other suites for databases, web servers, graphics,...
Bandwidth

• The amount of work (or data) per time
  • MB/s, GB/s -- network BW, disk BW, etc.
  • Frames per second -- Games, video transcoding
    • (why are games under both latency and BW?)

• Also called “throughput”
Measuring Bandwidth

- Measure how much work is done
- Measure latency
- Divide
Latency-BW Trade-offs

- Often, increasing latency for one task and increase BW for many tasks.
  - Think of waiting in line for one of 4 bank tellers
  - If the line is empty, your response time is minimized, but throughput is low because utilization is low.
  - If there is always a line, you wait longer (your latency goes up), but there is always work available for tellers.

- Much of computer performance is about scheduling work onto resources
  - Network links.
  - Memory ports.
  - Processors, functional units, etc.
  - IO channels.
  - Increasing contention for these resources generally increases throughput but hurts latency.
Stationwagon Digression

- IPv6 Internet 2: 272,400 terabit-meters per second
  - 585GB in 30 minutes over 30,000 Km
  - 9.08 Gb/s

- Subaru outback wagon
  - Max load = 408Kg
  - 21Mpg

- MHX2 BT 300 Laptop drive
  - 300GB/Drive
  - 0.135Kg

- 906TB

- Legal speed: 75MPH (33.3 m/s)

- BW = 8.2 Gb/s

- Latency = 10 days

- 241,535 terabit-meters per second
Prius Digression

- IPv6 Internet 2: 272,400 terabit-meters per second
  - 585GB in 30 minutes over 30,000 Km
  - 9.08 Gb/s

- My Toyota Prius
  - Max load = 374 Kg
  - 44 Mpg (2x power efficiency)

- MHX2 BT 300
  - 300 GB/Drive
  - 0.135 Kg

- 831 TB

- Legal speed: 75 MPH (33.3 m/s)

- BW = 7.5 Gb/s

- Latency = 10 days

- 221,407 terabit-meters per second (13% performance hit)