ECE 486/586
Computer Architecture
Lecture # 4
Spring 2019
Portland State University
Lecture Topics

• Performance
  – Processor Performance Equation
  – Calculating Average CPI
  – Amdahl’s Law

• Dependability

Reference: Chapter 1: Sections 1.7, 1.9
Processor Performance Equation

\[ CPU \text{ time} = \frac{\text{Instruction count} \times CPI}{\text{Clock Rate}} \]

Or

\[ CPU \text{ time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} \]
Performance Equation Parameters

- **Clock rate**
  - Instruction’s execution divided into multiple steps, each step 1 *clock cycle* long
  - Clock rate is equal to number of clock cycles per second (Hertz or Hz)
  - Clock period is equal to the length of clock cycle (clock rate = 1 / clock period)

- **Cycles per Instruction (CPI)**
  - Avg. no. of clock cycles required to execute an instruction
    - For ALU instructions, CPI depends on circuit speed and logic complexity
    - For memory instructions, CPI depends on memory access latency

- **Number of Instructions**
  - Count of *dynamic* instructions required to complete a program
    - Depends on the instruction set architecture
  - Not to be confused with code size (no. of instructions in a source program)
    - For example, 8-instruction loop executing 5 times counted as 40 instructions
How to Increase Performance?

- **Faster logic and memory**
  - Design faster implementations of arithmetic and logic circuits
  - Use caches to reduce average latency of memory operations

- **Increase parallelism**
  - Perform multiple instructions in parallel (*instruction-level parallelism*)
    - Fetch next instruction while ALU executes previous instruction (*pipelining*)
    - Use multiple ALUs to execute multiple instrs. concurrently (*superscalar*)
    - **Result:** CPI can become < 1
  - Execute multiple programs at the same time (*thread-level parallelism*)
    - Multiple cores on a single die (*multi/many*-core processors)
How to Increase Performance?

• **Increase processor’s clock speed**
  – Advances in manufacturing technology
    • Faster transistors => Faster circuits => Less time needed per clock cycle
    – Reducing the amount of processing done in each step (*deeper pipeline*)
      • This may increase CPI, unless there is instruction-level-parallelism

• **Better compiler technology**
  – Converts a high-level program into least expensive set of machine instructions with the goal of reducing “*Instruction count*\(\times\)CPI”
  – Optimized according to a specific processor architecture
How to Increase Performance?

• **Choice of Instruction Set Architecture (ISA)**
  
  – Tradeoff between *Instruction count*, *CPI* and/or *Clock Rate*
  
  – **Complex Instruction Set Computers (CISC)**
    
    • Fewer, more complex instructions per program (lower *instruction count*) but higher *CPI* and/or lower clock rate
  
  – **Reduced Instruction Set Computers (RISC)**
    
    • Simpler instructions, more needed to perform the same task (higher *instruction count*) but instructions quicker to execute (lower *CPI* and/or higher clock rate)
**Average CPI**

Performance Equation:

\[
\text{CPU Time} = \text{Cycle time} \times \text{Instruction Count} \times \text{Average CPI}
\]

Assuming \( n \) different type of instructions, each with count \( IC_i \) and requiring \( CPI_i \) cycles:

\[
\text{CPU Time} = \text{Cycle time} \times \sum_{i=1}^{n} (IC_i \times CPI_i)
\]

Then:

\[
\text{Average CPI} = \frac{\sum_{i=1}^{n} (IC_i \times CPI_i)}{IC} = \sum_{i=1}^{n} (CPI_i \times F_i)
\]

where \( F_i \) is the frequency of instruction type \( i \).
### Example: Calculating Average CPI

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
<th>CPI&lt;sub&gt;i&lt;/sub&gt;</th>
<th>CPI&lt;sub&gt;i&lt;/sub&gt; * F&lt;sub&gt;i&lt;/sub&gt;</th>
<th>%Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>40%</td>
<td>1</td>
<td>0.4 * 1 = 0.4</td>
<td>0.4/1.75 = 23%</td>
</tr>
<tr>
<td>Load</td>
<td>30%</td>
<td>2</td>
<td>0.3 * 2 = 0.6</td>
<td>0.6/1.75 = 34%</td>
</tr>
<tr>
<td>Store</td>
<td>15%</td>
<td>2</td>
<td>0.15 * 2 = 0.3</td>
<td>0.3/1.75 = 17%</td>
</tr>
<tr>
<td>Branch</td>
<td>15%</td>
<td>3</td>
<td>0.15 * 3 = 0.45</td>
<td>0.45/1.75 = 26%</td>
</tr>
</tbody>
</table>

**Average CPI = 0.4 + 0.6 + 0.3 + 0.45 = 1.75**
Amdahl’s Law

The performance improvement to be gained from using some enhancement is limited by the fraction of time the enhancement can be used.

![Diagram](image)

- Original:
  - Execution time $t_{old}$
  - Target for enhancement
  - Not targeted by enhancement

- After Enhancement:
  - Execution time $t_{new}$
  - Improved execution time

- $t_{new} = \frac{t_{old}}{f + \frac{1-f}{t_{old}}}$
Amdahl’s Law

\[ \text{Speedup} = \frac{\text{Execution time for entire task before the enhancement}}{\text{Execution time for entire task after the enhancement}} \]

\[ = \frac{\text{Execution time old}}{\text{Execution time new}} \]

- Speedup depends on two factors:
  - \( \text{Fraction}_{\text{enhanced}} \): Fraction of computation time in the original computer that can be converted to take advantage of the enhancement (always less than or equal to 1)
  - \( \text{Speedup}_{\text{enhanced}} \): How much faster the task would run if the enhanced mode were used for the entire program? (always greater than 1)
Amdahl’s Law

\[
\text{Execution timenew} = \text{Execution timeold} \times \left[ (1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right]
\]

\[
\text{Speedup} = \frac{\text{Execution timeold}}{\text{Execution timenew}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}
\]
Amdahl’s Law: Example

• We are considering an enhancement to the processor of a web server. The new CPU is 10x faster on computation in the Web serving application than the original CPU. Assuming that the CPU is busy with computation 40% of the time and waiting for I/O 60% of the time. What is the overall speedup gained by incorporating the enhancement?

\[
\text{Fraction}_{\text{enhanced}} = 0.4 \\
\text{Speedup}_{\text{enhanced}} = 10
\]

\[
\text{Overall speedup} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = 1.56
\]
Amdahl’s Law: Example

• What if the enhancement can be applied system-wide?

\[
\begin{align*}
\text{Fraction}_{\text{enhanced}} &= 1 \\
\text{Speedup}_{\text{enhanced}} &= 10
\end{align*}
\]

Overall speedup = \( \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} \) = \( \frac{1}{(1 - 1) + \frac{1}{10}} \) = 10
Amdahl’s Law: Example

- What if the enhancement is so great that it eliminates the execution time formerly contributed by the fraction enhanced?

\[
\text{Fraction}_{\text{enhanced}} = 0.4 \\
\text{Speedup}_{\text{enhanced}} = \text{Infinite}
\]

Overall speedup = \[
\frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} = \frac{1}{1 - 0.4 + \frac{0.4}{\text{Infinite}}} = 1.67
\]
Amdahl’s Law: Diminishing Returns

- If an enhancement is applicable only for a fraction of a task, then we can’t speed up the task by more than the reciprocal of 1 minus that fraction.
Amdahl’s Law: Another Example

- A common transformation required in graphics engines is square root. Implementations of floating point (FP) square root differ considerably in performance. Assume FPSQR is responsible for 20% of the execution time of a critical graphics application. One proposal is to enhance the FPSQR hardware to speed up FPSQR by 10x. Another proposal is to make all floating point operations run faster by 1.6x. Floating point instructions represent 60% of the overall execution time. Assuming the cost of each enhancement is the same, which option would you prefer?

**Option 1:** Enhance FPSQR  
Fraction\(_{enhanced}\) = 0.2, Speedup\(_{enhanced}\) = 10  
Overall speedup = \(\frac{1}{(1 - 0.2) + \frac{0.2}{10}}\) = 1.22

**Option 2:** Enhance all FP instructions  
Fraction\(_{enhanced}\) = 0.6, Speedup\(_{enhanced}\) = 1.6  
Overall speedup = \(\frac{1}{(1 - 0.6) + \frac{0.6}{1.6}}\) = 1.29  
Better
• Service Accomplishment vs. Service Interruption

- System operating according to service-level agreement (SLA)
- System unavailable or performance less than SLA requirements
- Failure causing disruption of service
Dependability Metrics

- **MTTF**: Mean Time to Failure
- **MTTR**: Mean Time to Repair
- **MTBF**: Mean Time between Failures = MTTF + MTTR
- **Failure Rate**: Failures per unit time = 1/MTTF
- **FIT**: Failures per billion hours = $10^9$/MTTF

- **Module Availability**: Fraction of time during which the module is working reliably

  \[
  \text{Module Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} = \frac{\text{MTTF}}{\text{MTBF}}
  \]

- If a collection of modules have independent failures and exponentially distributed lifetimes, then:

  \[
  \text{Failure Rate of System} = \sum \text{Failure rates of each module}
  \]
Example

• Assume a disk subsystem with the following components and MTTF:
  – 10 disks, each rated at 1,000,000-hour MTTF
  – 1 SCSI controller, 500,000-hour MTTF
  – 1 power supply, 200,000-MTTF
  – 1 fan, 200,000-hour MTTF
  – 1 SCSI cable, 1,000,000-hour MTTF

Component lifetimes are exponentially distributed, failures independent.
Compute MTTF of the system as a whole.

\[
\text{Failure rate of System} = 10 \times \frac{1}{1,000,000} + \frac{1}{500,000} + \frac{1}{200,000} + \frac{1}{200,000} + \frac{1}{1,000,000}
\]

\[
= 2.3 \times 10^{-5} \text{ failures per hour (or 23,000 FIT)}
\]

\[\text{MTTF} = \frac{1}{\text{Failure Rate}} = \frac{1}{(2.3 \times 10^{-5})} = 43,500 \text{ hours}\]