This Unit

“Speed is not everything but it’s kilometers ahead of whatever is in second place.”
—Ed McCreight, The Dragon Computer System
Xerox PARC September, 1984

- Metrics
  - Latency and throughput

- Reporting performance
  - Benchmarking and averaging

- CPU performance equation

Performance: Latency vs. Throughput

- Latency (execution time): time to finish a fixed task
- Throughput (bandwidth): number of tasks in fixed time
  - Different: exploit parallelism for throughput, not latency
  - Often contradictory (latency vs. throughput)
    - Will see many examples of this
  - Choose definition of performance that matches your goals
    - Single scientific program: latency; web server: throughput?

Problem #1: Car vs. Bus

<table>
<thead>
<tr>
<th>Car</th>
<th>Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>speed = 60 miles/hour, capacity = 5</td>
<td>speed = 20 miles/hour, capacity = 60</td>
</tr>
<tr>
<td>task: transport passengers 10 miles</td>
<td></td>
</tr>
</tbody>
</table>

Latency (min) | Throughput (PPH) |
---------------|------------------|
Car            | 10 min | 15 PPH |
Bus            | 30 min | 60 PPH |

Comparing Performance

- A is X times faster than B if
  \[ \frac{\text{Latency(A)}}{\text{Latency(B)}} = X \]
  \[ \frac{\text{Throughput(A)}}{\text{Throughput(B)}} = X \]

- A is X% faster than B if
  \[ \frac{\text{Latency(A)}}{\text{Latency(B)}} = 1 + \frac{X}{100} \]
  \[ \frac{\text{Throughput(A)}}{\text{Throughput(B)}} = 1 + \frac{X}{100} \]
Problem #2: Car vs. Bus Revisited

- **Latency**
  - Car = 10 min, Bus = 30 min
  - Car is ___ times faster than bus
  - Car is ___% faster than bus

- **Throughput**
  - Car = 15 PPH, Bus = 60 PPH
  - Bus is ___ times faster than car
  - Bus is ___% faster than car

Reporting Performance

Benchmarking & Averaging

**SPEC Benchmarks**

- **SPEC (Standard Performance Evaluation Corporation)**
  - [http://www.spec.org/](http://www.spec.org/)
  - Consortium that collects, standardizes, and distributes benchmarks
  - Post **SPECmark** results for different processors
    - 1 number that represents performance for entire suite
    - Benchmark suites for CPU, Java, I/O, Web, Mail, etc.
    - Updated every few years: so companies don’t target benchmarks
  - **SPEC CPU 2006**
    - 12 “integer”: bzip2, gcc, perl, hmmer (genomics), h264, ...
    - 17 “floating point”: wrf (weather), povray, sphynx3 (speech)...
    - Written in C/C++ and Fortran

**Problem #2: Car vs. Bus Revisited**

- **Latency**
  - Car = 10 min, Bus = 30 min
  - Car is 3 times faster than bus
  - Car is 200% faster than bus

- **Throughput**
  - Car = 15 PPH, Bus = 60 PPH
  - Bus is 4 times faster than car
  - Bus is 300% faster than car

**Processor Performance and Workloads**

Q: what does Latency(Task1, ChipA) or Throughput(Task1, ChipA) mean?
A: nothing, there must be some associated workload

- **Workload**: set of tasks someone cares about
  - Latency(Task1, ChipA) (car/bus Task = drive ppl 10 miles)

- **Benchmarks**: standard workloads
  - Used to compare performance across machines
  - Are highly representative of actual programs people run

- **Micro-benchmarks**: non-standard non-workloads
  - Tiny programs used to isolate certain aspects of performance
  - Not representative of complex behaviors of real applications
  - Frequently helpful to examine isolated performance questions

**SPEC Benchmarks**

- **SPEC (Standard Performance Evaluation Corporation)**
  - [http://www.spec.org/](http://www.spec.org/)
  - Consortium that collects, standardizes, and distributes benchmarks
  - Post **SPECmark** results for different processors
    - 1 number that represents performance for entire suite
    - Benchmark suites for CPU, Java, I/O, Web, Mail, etc.
    - Updated every few years: so companies don’t target benchmarks
  - **SPEC CPU 2006**
    - 12 “integer”: bzip2, gcc, perl, hmmer (genomics), h264, ...
    - 17 “floating point”: wrf (weather), povray, sphynx3 (speech)...
    - Written in C/C++ and Fortran

**Other Benchmarks**

- **Parallel benchmarks**
  - SPLASH2: Stanford Parallel Applications for Shared Memory
  - NAS: another parallel benchmark suite
  - SPECopenMP: parallelized versions of SPECfp
  - SPECjbb: Java multithreaded database-like workload

- **Transaction Processing Council (TPC)**
  - TPC-C: On-line transaction processing (OLTP)
  - TPC-H/R: Decision support systems (DSS)
  - TPC-W: E-commerce database backend workload
  - Have parallelism (intra-query and inter-query), heavy I/O, memory

- **Companies have internal benchmarks**
  - What’s going to be important in the future?
  - Overfitting?
Mean (Average) Performance Numbers

3 Types of Means
• Arithmetic
  • for units that are proportional to time (e.g., latency)
• Harmonic
  • for units that are inversely proportional to time (e.g., throughput)
• Geometric
  • For unitless quantities (e.g., speedup ratios)

Know when to use which one & how it is computed.

Arithmetic Mean
For units that are proportional to time (e.g., latency)
Chip A, N programs:
\[
\text{Arithmetic Mean} = \frac{\sum_{i=1}^{N} \text{Latency}(P_i, A)}{N}
\]
You can add latencies, but not throughputs
• \(\text{Latency}(P1+P2, A) = \text{Latency}(P1, A) + \text{Latency}(P2, A)\)
• \(\text{Throughput}(P1+P2,A) \neq \text{Throughput}(P1,A) + \text{Throughput}(P2,A)\)
• 1 mile @ 10 miles/hour + 1 mile @ 100 miles/hour
• Average is not 55 miles/hour
• Need a different mean....

Harmonic Mean
For units that are inversely proportional to time (e.g., throughput)
Chip A, N programs:
\[
\text{Harmonic Mean} = \frac{1}{\sum_{i=1}^{N} \frac{1}{\text{Throughput}(P_i, A)}}
\]
P1: 1 mile @ 30 miles/hour
P2: 1 mile @ 90 miles/hour
\[
\frac{1}{30} + \frac{1}{90} = 45 \text{ mph}
\]

Geometric Mean
For unitless quantities (e.g., speedup ratios)
\[
\text{Geometric Mean} = \sqrt[N]{\prod_{i=1}^{N} \text{Speedup}(P_i, A)}
\]

Processor Performance Equation
Program runtime:
\[
\text{seconds}_{\text{program}} = \frac{\text{instructions}_{\text{program}} \times \text{cycles}_{\text{instruction}} \times \text{seconds}_{\text{cycle}}}{\text{cycles}_{\text{instruction}}}
\]
• Instructions per program: “dynamic instruction count”
• Runtime count of instructions executed by the program
• Determined by program, compiler, ISA
• Cycles per instruction: “CPI” (typical range: 2 to 0.5)
• About how many cycles does an instruction take to execute?
• Determined by program, compiler, ISA, micro-architecture
• Seconds per cycle: clock period, length of each cycle
• Inverse metric: cycles/second (Hertz) or cycles/ns (Ghz)
• Determined by micro-architecture, technology parameters
• For lower latency (=better performance) minimize all three
• Difficult: often pull against one another
**Cycles per Instruction (CPI)**

- **CPI**: Cycle/instruction for on average
  - **IPC = 1/CPI**
    - Used more frequently than CPI
    - Favored because “bigger is better”, but harder to compute with
    - Different instructions have different cycle costs
    - E.g., “add” typically takes 1 cycle, “divide” takes >10 cycles
    - Depends on relative instruction frequencies

- **CPI example**
  - Program has equal ratio: integer, memory ops, floating point
  - Cycles per instruction type: integer = 1, memory = 2, FP = 3
  - What is the CPI? (33% * 1) + (33% * 2) + (33% * 3) = 2

- **Caveat**: this sort of calculation ignores many effects
  - Back-of-the-envelope arguments only

**Problem #3: CPI Example**

- Assume a processor with instruction frequencies and costs
  - Integer ALU: 50%, 1 cycle
  - Load: 20%, 5 cycle
  - Store: 10%, 1 cycle
  - Branch: 20%, 2 cycle

Which change would improve performance more?
- A: “Branch prediction” to reduce branch cost to 1 cycle?
- B: “Cache” to reduce load cost to 3 cycles?

- Compute CPI

<table>
<thead>
<tr>
<th></th>
<th>INT</th>
<th>LD</th>
<th>ST</th>
<th>BR</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>0.5</td>
<td>0.2</td>
<td>0.1</td>
<td>0.2</td>
<td>2.0</td>
</tr>
<tr>
<td>A</td>
<td>0.5</td>
<td>0.2</td>
<td>0.1</td>
<td>0.2</td>
<td>1.8</td>
</tr>
<tr>
<td>B</td>
<td>0.5</td>
<td>0.2</td>
<td>0.1</td>
<td>0.2</td>
<td>1.6</td>
</tr>
</tbody>
</table>

*(winner)*

**MIPS (performance metric, not the ISA)**

- (Micro) architects often ignore dynamic instruction count
  - Typically have one ISA, one compiler → treat it as fixed
- CPU performance equation becomes
  
  \[
  \text{Latency (seconds)} = \frac{\text{cycles (inst)}}{\text{cycle}} \\
  \text{Throughput (insns/second)} = \frac{\text{insns (inst)}}{\text{cycle}} \times \frac{\text{cycles (inst)}}{\text{second}}
  \]

- **MIPS** (millions of instructions per second)
  - **Cycles / second**: clock frequency (in MHz)
    - Ex: CPI = 2, clock = 500 MHz → 0.5 * 500 MHz = 250 MIPS
  - Pitfall: may vary inversely with actual performance
    - Compiler removes insns, program faster, but lower MIPS
    - Work per instruction varies (multiply vs. add, FP vs. integer)

**MHz (MegaHertz) and GHz (GigaHertz)**

- 1 Hertz = 1 cycle per second
  - 1 GHz is 1 cycle per nanosecond, 1 GHz = 1000 MHz
- General public (mostly) ignores CPI
  - Equates clock frequency with performance!
- Which processor would you buy?
  - Processor A: CPI = 2, clock = 5 GHz
  - Processor B: CPI = 1, clock = 3 GHz
  - Probably A, but B is faster (assuming same ISA/compiler)

- **Classic example**
  - 800 MHz PentiumIII faster than 1 GHz Pentium4!
- **Recent example**: Core i7 faster clock-per-clock than Core 2
  - **Same ISA and compiler!**
- **Meta-point**: danger of partial performance metrics!

**Latency vs. Throughput Revisited**

- Two views of performance: latency vs. throughput
- Two scopes of performance: instruction vs. program

- **Single instruction latency**
  - Doesn’t matter: programs comprised of billions+ of insns
  - Difficult to reduce anyway
  - Making 1 insn faster doesn’t help unless it’s the slowest insn

- **Instruction throughput** → program latency or throughput
  - Can reduce using parallelism
    - Multiple cores (more units executing instructions)... more later
    - Inter-instruction parallelism example: pipelining
In the Beginning...

...was the single-cycle data path

- Fetch, decode, execute one complete insn every cycle
  - Low CPI: 1 by definition
  - Long clock period: set by longest insn (logic + wire delays)

Inter-Instruction Parallelism: Pipelining

Pipelining: cut datapath into N stages (here 5)

- Separate each stage of logic by latches
- Clock period: maximum logic + wire delay of any stage
  \[ T_{\text{single cycle}} = \max(T_{\text{insn-mem}}, T_{\text{regfile}}, T_{\text{ALU}}, T_{\text{data-mem}}) \]
- Base CPI = 1, but actual CPI > 1: pipeline stalls a lot
- Individual insn latency increases (pipeline overhead), that's ok

Pipelining: Clock Frequency vs. IPC

- Increase number of pipeline stages ("pipeline depth")
- Keep cutting datapath into finer pieces
  - Increases clock frequency (decreases clock period)
  - Latch overhead & unbalanced stages cause sub-linear scaling
- Decreases IPC (increase CPI)
  - More pipeline "hazards", higher branch penalty
  - Memory latency relatively higher (same absolute lat., more cycles)
- Result: at some point, deeper pipelines decrease performance

- "Optimal" pipeline depth is program and technology specific
- Classic example
  - Pentium III: 12 stage pipeline, 800 MHz
  - Pentium 4: 22 stage pipeline, 1 GHz (and slower due to IPC)

Note: clock frequency implies CPU clock. Other system components have own clocks (or not).

Problem #4: CPI and Clock Frequency

1 GHz processor with

- 80% non-memory instructions @ 1 cycle
- 20% memory insns @ 6 nanoseconds (= 6 cycles)

Double the core clock frequency?

- Increasing processor clock doesn't accelerate memory!
  - Non-memory instructions retain 1-cycle latency
  - Memory instructions now have 12-cycle latency

Infinite clock frequency?

- Hello, Amdahl's Law!

<table>
<thead>
<tr>
<th>Non-Mem</th>
<th>Mem</th>
<th>CPI</th>
<th>MIPS</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GHz</td>
<td>0.8  x 1</td>
<td>0.2  x 6</td>
<td>2.0</td>
<td>500</td>
</tr>
<tr>
<td>2 GHz</td>
<td>0.8  x 1</td>
<td>0.2  x 6</td>
<td>3.2</td>
<td>625</td>
</tr>
<tr>
<td>&gt;&gt; GHz</td>
<td>1M insn / (200K x 6 ns)</td>
<td>813</td>
<td>1.66</td>
<td></td>
</tr>
</tbody>
</table>

Measuring CPI

- How are CPI and execution-time actually measured?
  - Execution time? stopwatch timer (Unix "time" command)
  - CPI = CPU time / (clock period * dynamic insn count)
  - How is dynamic instruction count measured?
  - More useful is CPI breakdown (CPI_{CPU}, CPI_{MEM}, etc.)
  - So we know what performance problems are and what to fix

- Hardware event counters
  - Available in most processors today
  - One way to measure dynamic instruction count

- Cycle-level micro-architecture simulation (e.g., SimpleScalar)
  - Measure exactly what you want ... and impact of potential fixes!
- Method of choice for many micro-architects
- Hardware emulation (e.g., on FPGAs) becoming common
Performance Rules of Thumb

**Amdahl’s Law:** “Make the common case fast”
- Literally: total speedup limited by non-accelerated piece
- Example: can optimize 50% of program A
  - Even “magic” optimization that makes this 50% disappear...
  - …only yields a 2X speedup
**Corollary:** build a balanced system
- Don’t optimize 1% to the detriment of other 99%
- Don’t over-engineer capabilities that cannot be utilized

Design for actual performance, not peak performance
- Peak perf: “Performance you are guaranteed not to exceed”
- Greater than “actual” or “average” or “sustained” performance
- Why? Caches misses, branch mispredictions, limited IIP, etc.
- For actual performance X, machine capability must be > X

Summary

- Latency:
  \[
  \text{seconds} = \frac{\text{instructions}}{\text{cycles}} \times \text{seconds} \\
  \text{program} = \text{program} \times \text{instruction} \times \text{cycle}
  \]
- **Instructions / program:** dynamic instruction count
  - Function of program, compiler, instruction set architecture (ISA)
- **Cycles / instruction:** CPI
  - Function of program, compiler, ISA, micro-architecture
- **Seconds / cycle:** clock period
  - Function of micro-architecture, technology parameters
- Optimize each component
  - CSE 560 focuses mostly on CPI (caches, parallelism)
  - …but some on dynamic instruction count (compiler, ISA)
  - …and some on clock frequency (pipelining, technology)