Readings

- Chapter 1.2-1.4 of MA:FSPTCM

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 (e.g., bread)
  - Often contradictory (latency *vs.* throughput)
    - Will see many examples of this
  - Choose definition of performance that matches your goals
    - Scientific program: latency; web server: throughput?

- Example: move people 10 miles
  - Car: capacity = 5, speed = 60 miles/hour
  - Bus: capacity = 60, speed = 20 miles/hour
  - Latency: **car = 10 min**, bus = 30 min
  - Throughput: car = 15 PPH (count return trip), **bus = 60 PPH**
Processor Performance and Workloads

Q: what does latency(ChipA) or throughput(ChipA) mean?
A: nothing, there must be some associated workload

Workload: set of tasks someone (you) cares about

Benchmarks: standard workloads
- Used to compare performance across machines
- Either are or 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
- Examples: towers-of-hanoi, 8-queens, etc.

Comparing Performance

A is X times faster than B if
- Latency(A) = Latency(B) / X
- Throughput(A) = Throughput(B) * X

A is X% faster than B if
- Latency(A) = Latency(B) / (1+X/100)
- Throughput(A) = Throughput(B) * (1+X/100)

Car/bus example
- Latency? Car is 3 times (and 200%) faster than bus
- Throughput? Bus is 4 times (and 300%) faster than car

SPEC Benchmarks

SPEC (Standard Performance Evaluation Corporation)
- 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, etc.
- 17 “floating point”: wrf (weather), povray, sphynx3 (speech), etc.
- 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 2000
- 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 and memory components
**SPECmark 2006**

- **Reference machine:** Sun UltraSPARC II (@ 296 MHz)
- **Latency SPECmark**
  - For each benchmark
    - Take odd number of samples
    - Choose median
    - Take latency ratio (reference machine / your machine)
    - Take “average” (Geometric mean) of ratios over all benchmarks
- **Throughput SPECmark**
  - Run multiple benchmarks in parallel on multiple-processor system
- **Recent (latency) leaders**
  - SPECint: Intel 3.3 GHz Xeon W5590 (34.2)
  - SPECFp: Intel 3.2 GHz Xeon W3570 (39.3)
  - (First time I’ve look at this where same chip was top of both)

**Mean (Average) Performance Numbers**

- **Arithmetic:** \( \frac{1}{N} \sum_{P=1..N} \text{Latency}(P) \)
  - For units that are proportional to time (e.g., latency)
- **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 @ 30 miles/hour + 1 mile @ 90 miles/hour
    - Average is not 60 miles/hour
- **Harmonic:** \( \frac{N}{\sum_{P=1..N} 1/\text{Throughput}(P)} \)
  - For units that are inversely proportional to time (e.g., throughput)
- **Geometric:** \( \sqrt[N]{\prod_{P=1..N} \text{Speedup}(P)} \)
  - For unitless quantities (e.g., speedup ratios)

**Processor Performance Equation**

- Multiple aspects to performance: helps to isolate them
- **Program runtime = latency = \“seconds per program\” =**
  \[
  \frac{\text{instructions/program}}{\text{cycles/instruction}} \times \frac{\text{seconds/cycle}}{}
  \]
- **Instructions per program:** “dynamic instruction count”
  - Runtime count of instructions executed by the program
  - Determined by program, compiler, instruction set architecture (ISA)
- **Cycles per instruction:** “CPI” (typical range: 2 to 0.5)
  - On average, 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 per second (Hertz) or cycles per ns (Ghz)
  - Determined by micro-architecture, technology parameters
  - For low latency (better performance) minimize all three
    - Difficult: often pull against one another
- **CPI:** Cycle/instruction for on average
  - \( \text{IPC} = 1/\text{CPI} \)
    - Used more frequently than CPI
    - Favor 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**
  - A program executes equal: integer, floating point (FP), memory ops
  - Cycles per instruction type: integer = 1, memory = 2, FP = 3
  - What is the CPI? \( (33\% \times 1) + (33\% \times 2) + (33\% \times 3) = 2 \)
  - **Caveat:** this sort of calculation ignores many effects
    - Back-of-the-envelope arguments only
Another 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

- Base = 0.5*1 + 0.2*5 + 0.1*1 + 0.2*2 = 2 CPI
- A = 0.5*1 + 0.2*5 + 0.1*1 + 0.2*1 = 1.8 CPI
- B = 0.5*1 + 0.2*3 + 0.1*1 + 0.2*2 = 1.6 CPI (winner)

MIPS (performance metric, not the ISA)

- (Micro) architects often ignore dynamic instruction count
  - Typically work in one ISA/one compiler → treat it as fixed
- CPU performance equation becomes
  - Latency: seconds / insn = (cycles / insn) * (seconds / cycle)
  - Throughput: insn / second = (insn / cycle) * (cycles / second)

- MIPS (millions of instructions per second)
  - Cycles / second: clock frequency (in MHz)
  - Example: CPI = 2, clock = 500 MHz → 0.5 * 500 MHz = 250 MIPS

- Pitfall: may vary inversely with actual performance
  - Compiler removes insns, program gets faster, MIPS goes down
  - Work per instruction varies (e.g., 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
- (Micro-)architects often ignore dynamic instruction count...
  - ... but general public (mostly) also 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!
  - More recent example: Core i7 faster clock-per-clock than Core 2
  - Same ISA and compiler!
- Meta-point: danger of partial performance metrics!

CPI and Clock Frequency

- Clock frequency implies CPU clock
  - Other system components have their own clocks (or not)
    - E.g., increasing processor clock doesn't accelerate memory latency
- Example: a 1 Ghz processor with (1ns clock period)
  - 80% non-memory instructions @ 1 cycle (1ns)
  - 20% memory instructions @ 6 cycles (6ns)
  - (80%*1) + (20%*6) = 2ns per instruction (also 500 MIPS)
- Impact of double the core clock frequency?
  - Without speeding up the memory
    - Non-memory instructions latency is now 0.5ns (but 1 cycle)
    - Memory instructions keep 6ns latency (now 12 cycles)
    - (80% * 0.5) + (20% * 6) = 1.6ns per instruction (also 625 MIPS)
  - Speedup = 2/1.6 = 1.25, which is << 2
- What about an infinite clock frequency? (non-memory free)
  - Only a factor of 1.66 speedup (example of Amdahl's Law)
Measuring CPI

- How are CPI and execution-time actually measured?
  - Execution time? stopwatch timer (Unix “time” command)
  - CPI = CPU time / (clock frequency * dynamic insn count)
  - How is dynamic instruction count measured?

- More useful is CPI breakdown (CPI\textsubscript{CPU}, CPI\textsubscript{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
    - Calculate CPI using counter frequencies / known event costs
  - 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 (and you)

Improving CPI: Caching and Parallelism

- CIS501 is more about improving CPI than clock frequency
  - Techniques we will look at
    - Caching, speculation, multiple issue, out-of-order issue
    - Vectors, multiprocessing, more...
  - Moore’s Law can help CPI – “more transistors”
    - Best examples are caches (to improve memory component of CPI)
    - Parallelism:
      - IPC > 1 implies instructions in parallel
      - And now multi-processors (multi-cores)
        - But also speculation, wide issue, out-of-order issue, vectors...
  - All roads lead to multi-core
    - Why multi-core over still bigger caches, yet wider issue?
      - Diminishing returns, limits to instruction-level parallelism (ILP)
    - Multi-core can provide linear performance with transistor count

Improving Clock Frequency

- Faster transistors
  - Next topic

- Micro-architectural techniques
  - Multi-cycle processors
    - Break each instruction into small bits
    - Less logic delay -> improved clock frequency
    - Different instructions take different number of cycles
      - CPI > 1
  - Pipelined processors
    - As above, but overlap parts of instruction (parallelism!)
    - Faster clock, but CPI can still be around 1
    - Topic for next week
Performance Rules of Thumb

- **Amdahl’s Law**
  - Literally: total speedup limited by non-accelerated piece
  - \( \text{Speedup}(n, p, s) = \frac{s+p}{s + (p/n)} \)
  - \( p \) is “parallel percentage”, \( s \) is “serial
  - 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 performance: “Performance you are guaranteed not to exceed”
  - Greater than “actual” or “average” or “sustained” performance
    - Why? Caches misses, branch mispredictions, limited ILP, etc.
  - For actual performance \( X \), machine capability must be > \( X \)

Summary

- **Latency** = \( \frac{\text{seconds}}{\text{program}} = \frac{(\text{instructions} / \text{program}) \times (\text{cycles} / \text{instruction}) \times (\text{seconds} / \text{cycle})}{\text{program}} \)
- **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**
  - CIS501 focuses mostly on CPI (caches, parallelism)
  - ...but some on dynamic instruction count (compiler, ISA)
  - ...and some on clock frequency (pipelining, technology)