CIS 501
Computer Architecture

Unit 2: Performance

This Unit

- Metrics
  - Latency and throughput
- Reporting performance
  - Benchmarking and averaging
- CPU performance equation & performance trends

Readings

- H+P
  - Chapter 1: Section 1.8

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**

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

Processor Performance and Workloads

- Q: what does latency(Pentium) or thruput(Pentium) 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.

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

- Reference machine: Sun SPARC 10
- Latency SPECmark
  - For each benchmark
    - Take odd number of samples
    - Choose median
    - Take latency ratio (Sun SPARC 10 / 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 2.3 GHz Core2 Extreme (3119)
  - SPECfp: IBM 2.1 GHz Power5+ (4051)

Mean (Average) Performance Numbers

- **Arithmetic**: $(1/N) * \Sigma_{P=1..N}$ Latency(P)
  - For units that are proportional to time (e.g., latency)
- You can add latencies, but not throughputs
  - Latency(P1+P2,A) = Latency(P1,A) + Latency(P2,A)
  - Throughput(P1+P2,A) != Throughput(P1,A) + Throughput(P2,A)
    - 1 mile @ 30 miles/hour + 1 mile @ 90 miles/hour
    - Average is **not** 60 miles/hour
- **Harmonic**: $N / \Sigma_{P=1..N}$ 1/Throughput(P)
  - For units that are inversely proportional to time (e.g., throughput)
- **Geometric**: $^{N}\sqrt{\Pi_{P=1..N}}$ Speedup(P)
  - For unitless quantities (e.g., speedup ratios)

CPU Performance Equation

- Multiple aspects to performance: helps to isolate them
  - But ignore any one at your own risk!
- Latency = seconds / program =
  - (instructions / program) * (cycles / instruction) * (seconds / 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
- For low latency (better performance) minimize all three
  - Difficult: often pull against one another

Cycles per Instruction (CPI)

- **CPI**: Cycle/instruction for **average instruction**
  - **IPC** = 1/CPI
    - Used more frequently than CPI, but harder to compute with
  - Different instructions have different cycle costs
    - E.g., integer add typically takes 1 cycle, FP divide takes > 10
  - Assumes you know something about instruction frequencies
- CPI example
  - A program executes equal integer, FP, and memory operations
  - 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

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. Better data cache to reduce load cost to 3 cycles?
- Compute CPI
  - Base = 0.5*1 + 0.2*5 + 0.1*1 + 0.2*2 = 2
  - A = 0.5*1 + 0.2*5 + 0.1*1 + 0.2*1 = 1.8
  - B = 0.5*1 + 0.2*3 + 0.1*1 + 0.2*2 = 1.6 (**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)

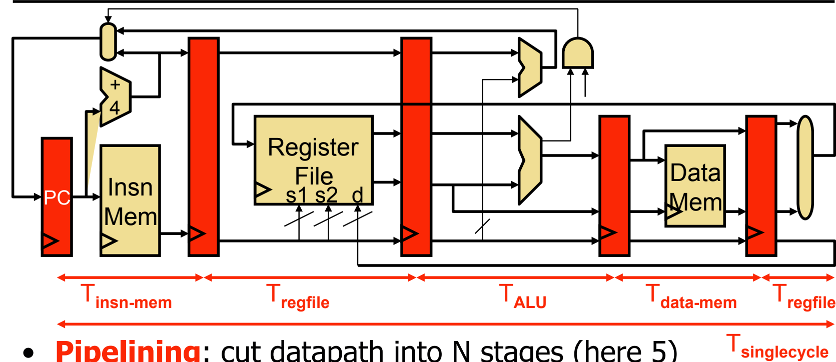
Mhz (MegaHertz) and Ghz (GigaHertz)

- 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!
  - Same ISA and compiler!
- **Meta-point: danger of partial performance metrics!**

Latency vs. Throughput Revisited

- Latency and throughput: two views of performance …
  - … at the program level
  - ... not at the instructions level
- Single instruction latency
  - – Doesn't matter: programs comprised of [billions]$^+$ of instructions
  - – Difficult to reduce anyway
- As number of dynamic instructions is large…
  - Instruction throughput → program latency or throughput
  - + Can reduce using ***parallelism***
    - Inter-instruction parallelism example: pipelining

Inter-Insn 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 = $\max(T_{insn-mem}, T_{regfile}, T_{ALU}, T_{data-mem})$
  - Base CPI = 1, but actual CPI > 1: pipeline must often stall
  - Individual insn latency increases (pipeline overhead), not the point

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 and unbalanced stages cause sub-linear scaling
    - Double the number of stages won't quite double the frequency
  - – Decreases IPC (increase CPI)
  - – At some point, actually causes performance to decrease
  - "Optimal" pipeline depth is program and technology specific
- Classic example
  - PentiumIII: 12 stage pipeline, 800 MHz
  - Pentium4: 22 stage pipeline, 1 GHz
    - – Actually slower (because of lower IPC)
  - Core2: 15 stage pipeline
    - \+ Intel learned its lesson

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
- Example: a 1 Ghz processor with
  - 50% memory instructions, 50% non-memory, all 1-cycle latency
  - Base: CPI is 1, frequency is 1Ghz → MIPS is 1000
- Impact of double the core clock freq?
  - ***Without*** speeding up the memory
    - Non-memory instructions retain 1-cycle latency
    - Memory instructions now have 2-cycle latency
  - CPI = (50% * 1) + (50% * 2) = 1.5
  - New: CPI is 1.5, frequency is 2Ghz → MIPS is 1333
  - Speedup = 1333/1000 = 1.33 << 2
- What about an infinite clock frequency?
  - Only a factor of 2 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 insn 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
    - 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)

Performance Trends

	386	486	Pentium	PentiumII	Pentium4	Core2
Year	1985	1989	1993	1998	2001	2006
Technode (nm)	1500	800	350	180	130	65
Transistors (M)	0.3	1.2	3.1	5.5	42	291
Clock (MHz)	**16**	**25**	**66**	**200**	**1500**	3000
Pipe stages	"1"	5	5	10	22	~15
(Peak) IPC	0.4	1	2	3	**3**	**"8"**
(Peak) MIPS	6	25	132	600	4500	24000

- Historically, clock provides 75%+ of performance gains…
  - Achieved via both faster transistors and deeper pipelines
- … that's changing: 1GHz: '99, 2GHz: '01, 3GHz: '02, 4Ghz?
  - Deep pipelining is not power efficient
  - Physical scaling limits are approaching

Improving CPI: Caching and Parallelism

- CIS501 is more about improving CPI than frequency
  - Techniques we will look at
    - Caching, speculation, multiple issue, out-of-order issue
    - Vectors, multiprocessing, more…
- **Moore's Law can help IPC** – "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, limited ILP in programs
  - Multi-core can provide linear performance with transistor count

Performance Rules of Thumb

- **Amdahl's Law**
  - 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 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 = seconds / program =
  - (instructions / program) * (cycles / instruction) * (seconds / 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
  - CIS501 focuses mostly on CPI (caches, parallelism)
  - …but some on dynamic instruction count (compiler, ISA)
  - …and some on clock frequency (pipelining, technology)