Performance

What do we mean by Performance?

- We must take many different factors into account:
- Technology
  - basic circuit speed (clock speed, usually in MHz: millions of cycles per second)
  - process technology (how many transistors on a chip)
- Organization
  - what style of ISA (RISC or CISC)
  - what type of memory hierarchy
  - how many processors in the *system*
- Software
  - quality of the compiler, OS, database driver, etc...
- There's alot more to measuring performance than clock speed...

Metrics

- *Raw speed* (peak performance, but it is never attained)
- *Execution time* (also called response time, i.e. time to execute one program from beginning to end). Need specific *benchmarks* for:
  - Integer dominated programs (compilers, etc)
  - Scientific (lots of floating point usage)
  - Graphics/multimedia
- *Throughput* (total amount of work in given time)
  - Good metrics for systems managers
  - Database programs (keeping the most people happy at the same time)
- Often, improving execution time will improve throughput, and vice versa.

Execution Time

- Performance:

$$\text{Performance}_A = \frac{1}{\text{Executiontime}_A}$$

- Processor A is faster than processor B if:

$$\text{ExecutiontimeA} < \text{ExecutiontimeB}$$

$$\text{Performance}_A > \text{Performance}_B$$

- Relative performance:

$$\frac{\text{PerformanceA}}{\text{PerformanceB}} = \frac{\text{ExecutiontimeB}}{\text{ExecutiontimeA}}$$

Measuring Execution Time

- Wall clock, response time, elapsed time
- Unix time function:

```
[sanjuan]:~ % time someprogram
346.085u 0.394s 5:48.32 99.4% 5+302k 0+0io 0pf+0w
```

- “time” lists User CPU time, System CPU time, elapsed time, percentage of elapsed time which is total CPU time, as well as information about the process size, quantity of IO, etc.
- Because of OS differences, it is hard to make comparisons from one system to another...
- For the remainder of this lecture, we’ll use *User CPU time* to mean *CPU execution time* (or just *execution time*)

Definition of CPU Execution Time

- CPU Execution Time = CPU clock cycles x clock cycle time
- CPU execution time is program dependent
- CPU clock cycles is program dependent
- clock cycle time (usually in nanoseconds, ns) depends on the particular machine
- Since clock cycle time = 1/clock cycle rate (clock cycle rate is in MHz, millions of cycles per second) an alternate definition is:

$$\text{CPU Execution Time} = \frac{\text{CPU clock cycles}}{\text{clock cycle rate}}$$

CPI - Cycles Per Instruction

- Definition: CPI is the average number of clock cycles per instruction.

$$\text{CPU clock cycles} = \text{Number of Instructions} \times \text{CPI}$$

$$\boxed{\text{CPU Exec Time} = \text{Number of Instructions} \times \text{CPI} \times \text{clock cycle time}}$$

- CPI in isolation is not a measure of performance (program and compiler dependent)
- Ideally, CPI = 1, but this might slow down the clock (compromise)
- CPI can (and usually is) greater than 1 because of breaks in control flow and the impact of the memory hierarchy
- Can we have CPI < 1?

Class of Instructions

- You can give different CPIs for various classes of instructions (e.g. floating point arithmetic instructions take longer than integer instructions, load-store instructions take longer than logical instructions, etc.)

$$\text{CPU Exec time} = \sum_{1}^{n} (\text{CPI}_i \times \text{C}_i) \times \text{clock cycle time}$$

- C_i is the number of instructions in the ith class that have been executed
- Note that minimizing the number of instructions does not necessarily improve the execution time of the program
- Improving part of the architecture can improve a C_j. One often talks about the contribution to CPI of a certain class of instructions.

Measuring average CPI

- Instruction count: need a simulator or (possibly less precise) a profiler
  - Simulator “interprets” every instruction and counts them
  - Profiler can either count how many times each basic block has been executed or use some sampling technique
- CPU Execution time can be measured (elapsed time)
- Clock cycle time is given by the processor
- We know execution time, cycle time, so we can solve for total cycles.
- Knowing the total cycles together with the total number of instructions executed lets us solve for average CPI.

Other Popular Metrics - MIPS

- MIPS = Millions of Instructions Per Second

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Exec time} \times 10^6}$$

or

$$\text{MIPS} = \frac{\text{clock rate}}{\text{CPI} \times 10^6}$$

- Since MIPS is a rate, the higher the better.
- But MIPS in isolation is no better than CPI in isolation. MIPS is:
  - Program dependent
  - Does not take the instruction set into account (CISC programs will typically take fewer instructions than RISC, so we can’t compare different ISAs)

The Trouble with MIPS

- Using MIPS can give “wrong” results:
  - Machine A with compiler C1 executes program P in 10 seconds, using 100,000,000 instructions (10 MIPS)
  - Machine A with compiler C2 executes program P in 15 seconds, using 180,000,000 instructions (12 MIPS)
- While C1 is clearly faster than C2, C1 has a lower MIPS rating than C2....
- ... the trouble with MIPS is that it doesn’t take CPI into account.

Other Popular Metrics - MFLOPS

- MFLOPS = Millions of floating point operations per second

$$\text{MFLOPS} = \frac{\text{Number of floating point instructions}}{\text{Exec time} \times 10^6}$$

- Same problems as MIPS:
  - Program dependent
  - Doesn’t take instruction set into account
  - Counts operations, not the time to execute them...

Benchmarks

- Benchmarks: workload representative of what the computer will actually be used for.
- Industry benchmarks to compare machines: SPEC benchmarks (SPECint, SPECfp), Perfect Club
- Database benchmarks
- Multimedia benchmarks
- Caveats:
  - Compilers optimize specifically for benchmarks
  - Old SPEC benchmarks (1992) were too small (didn't test the memory system sufficiently)
  - Utilities, user interface, etc. are often not in benchmarks

Amdahl's Law

- The amount that we can improve performance with a given improvement is limited by the amount that the improved feature is actually used:

$$\text{Exec time after improvement} = \frac{\text{Exec time affected by improvement}}{\text{Amount of improvement}} + \text{Exec time unaffected}$$

- For instance, if loads/stores take up 33% of our execution time, how much do we need to improve loads/stores to make the program run 1.5 times faster?
- Important corollary: *Make the common case fast.*

Example Measurements

Instruction Category	GCC	SPICE	Ave. CPI
Load/Store	33%	40%	1.4
Branch	16%	8%	1.8
Jumps	2%	2%	1.2
FP Add	-	5%	2.0
FP Sub	-	3%	4.0
FP Mul	-	6%	5.0
FP Div	-	3%	19.0
Other (Integer add/sub, stl, etc)	49%	33%	1.0

- What is the average CPI for gcc? For spice? Should we expect CPI for a given category to be the same btwn two programs?