# CS 61C: Great Ideas in Computer Architecture (Machine Structures) *Performance*

Instructor:
Michael Greenbaum



## Agenda

- Floating Point Review
- Defining performance
- **Administrivia**
- Workloads and Benchmarks
- **Break**
- Measuring Performance



## Review: Floating Point Encoding

| 31 | 30 23 | 22 0 |
|---|---|---|
| S | Exponent | Significand |
| 1 bit | 8 bits | 23 bits |

$$(-1)^{S} \times (1 . \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

- Note the implicit 1 in front of the Significand.



## Peer Instruction Question

Suppose Big, Tiny, and NegativeBig are floats in C, with Big initialized to a big number (e.g., age of universe in seconds or $4.32 \times 10^{17}$), Tiny to a small number (e.g., seconds/femtosecond or $1.0 \times 10^{-15}$), NegativeBig = - Big. Here are two conditionals:

I. (Big * Tiny) * NegativeBig == (Big * NegativeBig) * Tiny

II. (Big + Tiny) + NegativeBig == (Big + NegativeBig) + Tiny

Which statement about these is always correct?

Red. I. is false and II. is false

Yellow. I. is false and II. is true

Blue. I. is true and II. is false

Green. I. is true and II. is true



## Peer Instruction Answer

Suppose Big, Tiny, and NegativeBig are floats in C, with Big initialized to a big number (e.g., age of universe in seconds or $4.32 \times 10^{17}$), Tiny to a small number (e.g., seconds/femtosecond or $1.0 \times 10^{-15}$), NegativeBig = - Big. Here are two conditionals:

I. (Big * Tiny) * NegativeBig == (Big * NegativeBig) * Tiny

II. (Big + Tiny) + NegativeBig == (Big + NegativeBig) + Tiny

Which statement about these is always correct?

**Red. I. is false and II. is false**

I. Works ok if no overflow, but because exponents add, if Big * NegBig overflows, then result is overflow, not -1

II. Left hand side is 0, right hand side is tiny



## Pitfalls

- Floating point addition is NOT associative
- Some optimizations can change order of floating point computations, which can change results
- Need to ensure that floating point algorithm is correct even with optimizations

# New-School Machine Structures (It's a bit more complicated!)

*Software* | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time

*Harness Parallelism & Achieve High Performance*

How do we know?




# Agenda

- Floating Point Review
- Defining performance
- **Administrivia**
- Workloads and Benchmarks
- **Break**
- Measuring Performance



# Defining CPU Performance

- What does it mean to say X is faster than Y?
- Ferrari vs. School Bus?
- 2009 Ferrari 599 GTB
  - 2 passengers, 11.1 secs in quarter mile
- 2009 Type D school bus
  - 54 passengers, quarter mile time?

  http://www.youtube.com/watch?v=KwyCoQuhUNA
- *Response Time*, *Latency*: e.g., time to travel ¼ mile
- *Throughput*, *Bandwidth*: e.g., passenger-mi in 1 hour



# What is Performance?

- *Latency* (or *response time* or *execution time*)
  - Time to complete one task
- *Bandwidth* (or *throughput*)
  - Tasks completed per unit time



# Cloud Performance: Why Application Latency Matters

| Server Delay (ms) | Increased time to next click (ms) | Queries/ user | Any clicks/ user | User satisfac-tion | Revenue/ User |
|---|---|---|---|---|---|
| 50 | -- | -- | -- | -- | -- |
| 200 | 500 | -- | -0.3% | -0.4% | -- |
| 500 | 1200 | -- | -1.0% | -0.9% | -1.2% |
| 1000 | 1900 | -0.7% | -1.9% | -1.6% | -2.8% |
| 2000 | 3100 | -1.8% | -4.4% | -3.8% | -4.3% |

Figure 6.10 Negative impact of delays at Bing search server on user behavior [Brutlag and Schurman 2009].

- Key figure of merit: application responsiveness
  - Longer the delay, the fewer the user clicks, the less the user happiness, and the lower the revenue per user



# Defining Relative CPU Performance

- $\text{Performance}_X = 1/\text{Program Execution Time}_X$
- $\text{Performance}_X > \text{Performance}_Y \Rightarrow 1/\text{Execution Time}_X > 1/\text{Execution Time}_y \Rightarrow \text{Execution Time}_Y > \text{Execution Time}_X$
- Computer X is N times faster than Computer Y
  $\text{Performance}_X / \text{Performance}_Y = N$ or
  $\text{Execution Time}_Y / \text{Execution Time}_X = N$
- Bus is to Ferrari as 12 is to 11.1:
  Ferrari is 1.08 times faster than the bus!

## Measuring CPU Performance

- Computers use a clock to determine when events take place within hardware
- *Clock cycles:* discrete time intervals
  - aka clocks, cycles, clock periods, clock ticks
- *Clock rate* or *clock frequency:* clock cycles per second (inverse of clock cycle time)
- 3 GigaHertz clock rate
  => clock cycle time = $1/(3\times10^9)$ seconds
  clock cycle time = 333 picoseconds (ps)



## CPU Performance Factors

- To distinguish between processor time and I/O, *CPU time* is time spent in processor
- `CPU Time/Program`
  `= Clock Cycles/Program`
  `x Clock Cycle Time`
- Or
  `CPU Time/Program`
  `= Clock Cycles/Program ÷ Clock Rate`



## CPU Performance Factors

- But a program executes instructions
- `CPU Time/Program`
  `= Clock Cycles/Program x Clock Cycle Time`
  `= Instructions/Program`
  `x Average Clock Cycles/Instruction`
  `x Clock Cycle Time`
- 1st term called *Instruction Count*
- 2nd term abbreviated *CPI* for average ***C**lock Cycles **P**er **I**nstruction*
- 3rd term is 1 / Clock rate



## Restating Performance Equation

- $$\text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$



## What Affects Each Component? Instruction Count, CPI, Clock Rate

| Hardware or software component? | Affects What? |
|---|---|
| Algorithm | |
| Programming Language | |
| Compiler | |
| Instruction Set Architecture | |



Student Roulette?

## What Affects Each Component? Instruction Count, CPI, Clock Rate

| Hardware or software component? | Affects What? |
|---|---|
| Algorithm | Instruction Count, CPI |
| Programming Language | Instruction Count, CPI |
| Compiler | Instruction Count, CPI |
| Instruction Set Architecture | Instruction Count, Clock Rate, CPI |

## Peer Instruction Question

- Computer A clock cycle time 250 ps, $CPI_A = 2$
- Computer B clock cycle time 500 ps, $CPI_B = 1.2$
- Assume A and B have same instruction set
- Which statement is true?

Red. Computer A is ~1.2 times faster than B
Blue. Computer A is ~4.0 times faster than B
Green. Computer B is ~1.7 times faster than A
Yellow. Computer B is ~3.4 times faster than A
Purple. None of the above



## Peer Instruction Answer

- Computer A clock cycle time 250 ps, $CPI_A = 2$
- Computer B clock cycle time 500 ps, $CPI_B = 1.2$
- Assume A and B have same instruction set
- Which statement is true?

Red. Computer A is ~1.2 times faster than B
Blue. Computer A is ~4.0 times faster than B
Green. Computer B is ~1.7 times faster than A
Yellow. Computer B is ~3.4 times faster than A
Purple. None of the above



## Agenda

- Floating Point Review
- Defining performance
- **Administrivia**
- Workloads and Benchmarks
- **Break**
- Measuring Performance



## Administrivia

- Project 1 due Sunday.
- Midterm, Friday 7/15.
  - Time and Location still TBD, will keep you updated
  - Closed book. We will supply you with a green sheet.
  - You can bring a one-sided, *handwritten* cheatsheet.
  - 3 hours long.
- Review Session early next week (probably Monday)
- "CS Illustrated" slides at bottom of webpage.
  - Useful illustrations for understanding number representation, floating point, pointers.



## Administrivia

- How should we study for the midterm?
  - Form study groups…don't prepare in isolation!
  - Attend the review session
  - Look over HW, Labs, Projects, class notes!
  - Go over old exams – HKN has put them online (link at top of cs61C home page)
  - Attend TA office hours, my office hours, work out hard probs
  - ASK QUESTIONS!

## Agenda

- Floating Point Review
- Defining performance
- **Administrivia**
- Workloads and Benchmarks
- **Break**
- Measuring Performance

## Workload and Benchmark

- *Workload:* Set of programs run on a computer
  - Actual collection of applications run or made from real programs to approximate such a mix
  - Specifies both programs and relative frequencies
- *Benchmark:* Program selected for use in comparing computer performance
  - Benchmarks form a workload
  - Usually standardized so that many use them



## SPEC (System Performance Evaluation Cooperative)

- Computer Vendor cooperative for benchmarks, started in 1989
- SPECCPU2006
  - 12 Integer Programs
  - 17 Floating-Point Programs
- Often turn into number where bigger is faster
- *SPECratio*: execution time on old reference computer divide by execution time on new computer to get an effective speed-up



## SPECINT2006 on AMD Barcelona

| Description | Instruc-tion Count (B) | CPI | Clock cycle time (ps) | Execu-tion Time (s) | Refer-ence Time (s) | SPEC-ratio |
|---|---|---|---|---|---|---|
| Interpreted string processing | 2,118 | 0.75 | 400 | 637 | 9,770 | 15.3 |
| Block-sorting compression | 2,389 | 0.85 | 400 | 817 | 9,650 | 11.8 |
| GNU C compiler | 1,050 | 1.72 | 400 | 724 | 8,050 | 11.1 |
| Combinatorial optimization | 336 | 10.0 | 400 | 1,345 | 9,120 | 6.8 |
| Go game | 1,658 | 1.09 | 400 | 721 | 10,490 | 14.6 |
| Search gene sequence | 2,783 | 0.80 | 400 | 890 | 9,330 | 10.5 |
| Chess game | 2,176 | 0.96 | 400 | 837 | 12,100 | 14.5 |
| Quantum computer simulation | 1,623 | 1.61 | 400 | 1,047 | 20,720 | 19.8 |
| Video compression | 3,102 | 0.80 | 400 | 993 | 22,130 | 22.3 |
| Discrete event simulation library | 587 | 2.94 | 400 | 690 | 6,250 | 9.1 |
| Games/path finding | 1,082 | 1.79 | 400 | 773 | 7,020 | 9.1 |
| XML parsing | 1,058 | 2.70 | 400 | 1,143 | 6,900 | 6.0 |



## Summarizing Performance …

| System | Rate (Task 1) | Rate (Task 2) |
|---|---|---|
| A | 10 | 20 |
| B | 20 | 10 |

*Which system is faster?*



Student Roulette?

## … Depends Who's Selling

| System | Rate (Task 1) | Rate (Task 2) | Average |
|---|---|---|---|
| A | 10 | 20 | 15 |
| B | 20 | 10 | 15 |

Average throughput

| System | Rate (Task 1) | Rate (Task 2) | Average |
|---|---|---|---|
| A | 0.50 | 2.00 | 1.25 |
| B | 1.00 | 1.00 | 1.00 |

Throughput relative to B

| System | Rate (Task 1) | Rate (Task 2) | Average |
|---|---|---|---|
| A | 1.00 | 1.00 | 1.00 |
| B | 2.00 | 0.50 | 1.25 |

Throughput relative to A



## Summarizing SPEC Performance

- Varies from 6x to 22x faster than reference computer
- *Geometric mean* of ratios: N-th root of product of N ratios

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

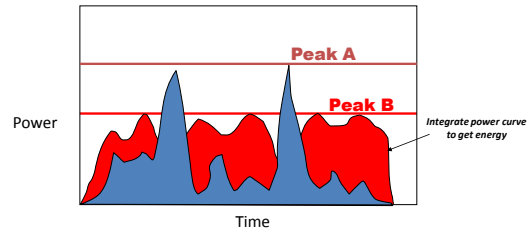
  - Geometric Mean gives same relative answer no matter what computer is used as reference
- Geometric Mean for Barcelona is 11.7

## Energy and Power
(Energy = Power x Time)

- Energy to complete operation (Joules)
  - Sample applications: battery life, total energy costs
- Peak power dissipation (Watts = Joules/s)
  - Affects heat (and cooling demands)
  - IT equipment's power is a term in the Power Utilization Efficiency (PUE) equation, a Warehouse Scale Computer figure of merit



## Peak Power vs. Lower Energy
(Power x Time = Energy)




- Which system has higher peak power?
- Which system has higher energy?


Student Roulette?

## Energy Proportional Computing

"The Case for Energy-Proportional Computing," Luiz André Barroso, Urs Hölzle, *IEEE Computer* December 2007

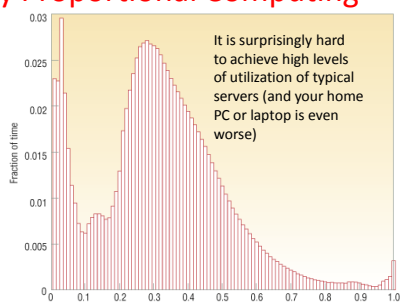



Figure 1. Average CPU utilization of more than 5,000 servers during a six-month period. Servers are rarely completely idle and seldom operate near their maximum utilization, instead operating most of the time at between 10 and 50 percent of their maximum



## SPECPower

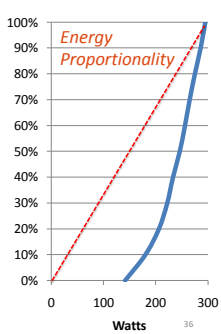
- Increasing importance of power and energy: create benchmark for performance and power
- Most servers in WSCs have average utilization between 10% & 50%, so measure power at medium as well as at high load
- Measure best performance and power, then step down request rate to measure power for every 10% reduction in performance
- Java server benchmark performance is *operations per second (ssj_ops)*, so metric is *ssj_ops/Watt*

$$\text{overall ssj\_ops per Watt} = \left(\sum_{i=0}^{10} \text{ssj\_ops}_i\right) / \left(\sum_{i=0}^{10} \text{power}_i\right)$$



## SPECPower on Barcelona

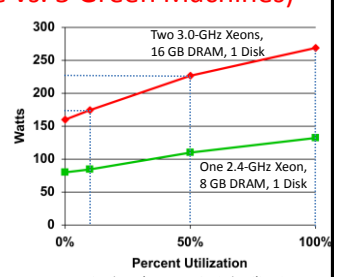
| Target Load % | Performance (ssj_ops) | Avg. Power (Watts) |
|---|---|---|
| 100% | 231,867 | 295 |
| 90% | 211,282 | 286 |
| 80% | 185,803 | 275 |
| 70% | 163,427 | 265 |
| 60% | 140,160 | 256 |
| 50% | 118,324 | 246 |
| 40% | 92,035 | 233 |
| 30% | 70,500 | 222 |
| 20% | 47,126 | 206 |
| 10% | 23,066 | 180 |
| 0% | 0 | 141 |
| Sum | 1,283,590 | 2,605 |
| | ssj_ops/Watt | 493 |






## Which is Better?
(1 Red Machine vs. 5 Green Machines)

- Five machines running at 10% utilization
  - Total Power =
- One machine running at 50% utilization
  - Total Power =

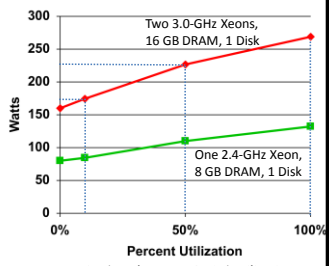



85% Peak Power@50% utilization
65% Peak Power@10% utilization


Student Roulette?

## Which is Better? (1 Red Machine vs. 5 Green Machines)

- Five machines running at 10% utilization
  - Total Power = 90 x 5 = 450w
- One machine running at 50% utilization
  - Total Power = 1 x 230 = 230w




85% Peak Power@50% utilization
65% Peak Power@10% utilization



## Other Benchmark Attempts

- Rather than run a collection of real programs and take their average (geometric mean), create a single program that matches the average behavior of a set of programs
- Called a *synthetic benchmark*
- First example called *Whetstone* in 1972 for floating point intensive programs in Fortran
- Second example called *Dhrystone* in 1985 for integer programs in Ada and C
  - Pun on Wet vs. Dry ("Whet" vs. "Dhry")



## Dhystone Shortcomings

- Dhrystone features unusual code that is not usually representative of real-life programs
- Dhrystone susceptible to compiler optimizations
- Dhrystone's small code size means always fits in caches, so not representative
- Yet still used in hand held, embedded CPUs!



## Agenda

- Floating Point Review
- Defining performance
- **Administrivia**
- Workloads and Benchmarks
- **Break**
- Measuring Performance



## Agenda

- Floating Point Review
- Defining performance
- **Administrivia**
- Workloads and Benchmarks
- **Break**
- Measuring Performance



## Measuring Time

- UNIX time command measures in seconds
- *Time Stamp Counter*
  - 64-bit counter of clock cycles on Intel 80x86 instruction set computers
  - 80x86 instruction RDTSC (Read TSC) returns TSC in regs EDX (upper 32 bits) and EAX (lower 32 bits)
  - Can read, but can't set
  - How long can measure?
  - Measures overall time, not just time for 1 program

## How to get RDTSC access in C?

```
static inline unsigned long long RDTSC(void)
{
  unsigned hi, lo;
  asm volatile ("rdtsc" : "=a"(lo), "=d"(hi));
  return ( (unsigned long long) lo) |
  ( ((unsigned long long) hi) <<32 );
}
```



## gcc Optimization Experiment

| | BubbleSort.c | Dhrystone.c |
|---|---|---|
| No Opt | | |
| -O1 | | |
| -O2 | | |
| -O3 | | |



## gcc Optimization Experiment

| | BubbleSort.c | Dhrystone.c |
|---|---|---|
| No Opt | 3.215 s | 7.230 s |
| -O1 | 1.460 s | 2.805 s |
| -O2 | 1.457 s | 2.711 s |
| -O3 | 1.458 s | 1.340 s |



## Where Do You Spend the Time in Your Program?

- Profiling a program (e.g., using, `gprof`) shows where it spends its time by function, so you can determine which code consumes most of the execution time
- Usually a 90/10 rule: 10% of code is responsible for 90% of execution time
  - Or 80/20 rule, where 20% of code responsible for 80% of time



## gprof

- Learn where program spent its time
- Learn functions called while it was executing
  - And which functions call other functions
- Three steps:
  - Compile & link program with profiling enabled
    - cc –pg x.c *{in addition to other flags use}*
  - Execute program to generate a profile data file
  - Run `gprof` to analyze the profile data



## gprof example

| % time | Cumulative (secs) | Self (secs) | calls | Self ms/call | Total ms/call | name |
|---|---|---|---|---|---|---|
| 18.18 | 0.06 | 0.06 | 23480 | 0.00 | 0.00 | find_char_unquote |
| 12.12 | 0.10 | 0.04 | 120 | 0.33 | 0.73 | pattern_search |
| 9.09 | 0.13 | 0.03 | 5120 | 0.01 | 0.01 | collapse_continuations |
| 9.09 | 0.16 | 0.03 | 148 | 0.20 | 0.88 | update_file_1 |
| 9.09 | 0.19 | 0.03 | 37 | 0.81 | 4.76 | eval |
| 6.06 | 0.21 | 0.02 | 12484 | 0.00 | 0.00 | file_hash_1 |
| 6.06 | 0.23 | 0.02 | 6596 | 0.00 | 0.00 | get_next_mword |
| 3.03 | 0.24 | 0.01 | 29981 | 0.00 | 0.00 | hash_find_slot |
| 3.03 | 0.25 | 0.01 | 14769 | 0.00 | 0.00 | next_token |
| 3.03 | 0.26 | 0.01 | 5800 | 0.00 | 0.00 | variable_expand_string |

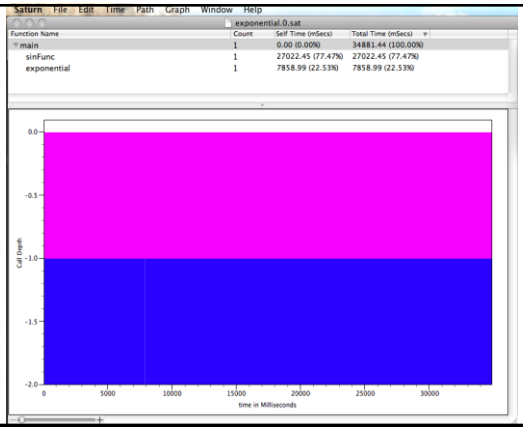
See http://linuxgazette.net/100/vinayak.html

## Test Program to Profile with Saturn

```
#include <math.h>
#define LIMIT 500000000

void exponential()
{
    double a;
    int i;

    for (i=1; i != LIMIT; i++)
    a = exp(i/1000.0);
}

void sinFunc()
{
    double a;
    int i;

    for (i=1; i != LIMIT; i++)
    a = sin(i/1000.0);
}

int main()
{
    exponential();
    sinFunc();
    return 0;
}
```

(`saturn` is an alternate tool supported on mac osx)






## Cautionary Tale

- "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity"
  **-- William A. Wulf**
- "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil"
  **-- Donald E. Knuth**



## And In Conclusion, …

- Time (seconds/program) is measure of performance

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

- Benchmarks stand in for real workloads to as standardized measure of relative performance
- Power of increasing concern, and being added to benchmarks
- Time measurement via clock cycles, machine specific
- Profiling tools as way to see where spending time in your program
- Don't optimize prematurely!