CMSC 411
Computer Systems Architecture
Lecture 3
Reliability and Performance

RELIABILITY

Define and quantify reliability (1/3)

- How decide when a system is operating properly?
- Infrastructure providers now offer Service Level Agreements (SLA) to guarantee that their networking or power service would be dependable
- Systems alternate between 2 states of service with respect to an SLA:

1. Service accomplishment, where the service is delivered as specified in SLA
2. Service interruption, where the delivered service is different from the SLA

- Failure = transition from state 1 to state 2
- Restoration = transition from state 2 to state 1

Define and quantify reliability (2/3)

- *Module reliability* = measure of continuous service accomplishment (or time to failure).
  2 metrics

1. *Mean Time To Failure* (*MTTF*) measures Reliability
2. *Failures In Time* (*FIT*) = 1/MTTF, the rate of failures
   - Traditionally reported as failures per billion hours of operation

- *Mean Time To Repair* (*MTTR*) measures Service Interruption
  - *Mean Time Between Failures* (*MTBF*) = MTTF+MTTR
- *Module availability* measures service as alternate between the 2 states of accomplishment and interruption (number between 0 and 1, e.g. 0.9)
- *Module availability = MTTF / (MTTF + MTTR)*

Example calculating reliability

- If modules have *exponentially distributed lifetimes* (age of module does not affect probability of failure), overall failure rate is the sum of failure rates of the modules
- Calculate FIT and MTTF for 10 disks (1M hour MTTF per disk), 1 disk controller (0.5M hour MTTF), and 1 power supply (0.2M hour MTTF):

$$\begin{aligned} FailureRate &= 10\times(1/1{,}000{,}000)+1/500{,}000+1/200{,}000 \\ &= (10+2+5)/1{,}000{,}000 \\ &= 17/1{,}000{,}000 \\ &= 17{,}000\,FIT \\ MTTF &= 1{,}000{,}000{,}000/17{,}000 \\ &\approx 59{,}000\,hours \end{aligned}$$

COMPUTER PERFORMANCE

Definition: Performance

- Performance is in units of things per second
  - bigger is better
- If we are primarily concerned with response time

$$\text{performance}(x) = \frac{1}{\text{execution\_time}(x)}$$

" X is n times faster than Y" means

$$n = \frac{\text{Performance}(X)}{\text{Performance}(Y)} = \frac{\text{Execution\_time}(Y)}{\text{Execution\_time}(X)}$$

Comparing performance of two machines

- Definition: Performance is equal to 1 divided by execution time
- Problem: How to measure execution time?

What is time?

- Unix time command example:
  - 90.7u 12.9s 2:39 65%
  - The user used the CPU for 90.7 seconds (user CPU time)
  - The system used it for 12.9 seconds (system CPU time)
  - Elapsed time from the user's request to completion of the task was 2 minutes, 39 seconds (159 seconds)
  - And (90.7 + 12.9)/159 = 65%
    - » the rest of the time was spent waiting for I/O or running other programs

Time (cont.)

- Usual measurements of time:
  - system performance measures the elapsed time on unloaded (single user) system
  - CPU performance measures user CPU time on unloaded system

COMPUTER BENCHMARKS

How to measure CPU performance

- Benchmark: a program used to measure performance
  - real programs - what is reality?
  - kernels - loops in which most of time is spent in a real program
  - toy programs
  - synthetic programs
- Fact: Computer manufacturers tune their product to the popular benchmarks
  - ``Your results may vary,'' unless you run benchmark programs and nothing else
  - See Figure 1.13 listing programs in the SPEC CPU2006 benchmark suite

Performance: What to measure

- Usually rely on benchmarks vs. real workloads
- To increase predictability, collections of benchmark applications, called *benchmark suites*, are popular
- SPECCPU: popular desktop benchmark suite
  - CPU only, split between integer and floating point programs
  - SPEC2006 has 12 integer, 17 floating-point C/C++/Fortran programs
  - SPECSFS (NFS file server) and SPECWeb (WebServer) added as server benchmarks

SPEC CPU 2006 (Integer)

Benchmark	Language	Application Area
400.perlbench	C	Programming Language
401.bzip2	C	Compression
403.gcc	C	C Compiler
429.mcf	C	Combinatorial Optimization
445.gobmk	C	Artificial Intelligence: Go
456.hmmer	C	Search Gene Sequence
458.sjeng	C	Artificial Intelligence: chess
462.libquantum	C	Physics / Quantum Computing
464.h264ref	C	Video Compression
471.omnetpp	C++	Discrete Event Simulation
473.astar	C++	Path-finding Algorithms
483.xalancbmk	C++	XML Processing

SPEC CPU 2006 (Floating Point)

Benchmark	Language	Application Area
410.bwaves	Fortran	Fluid Dynamics
416.gamess	Fortran	Quantum Chemistry
433.milc	C	Physics / QCD
434.zeusmp	Fortran	Physics / CFD
435.gromacs	C,Fortran	Molecular Dynamics
436.cactusADM	C,Fortran	Physics
437.leslie3d	Fortran	Fluid Dynamics
444.namd	C++	Biology / Molecular Dynamics
447.dealII	C++	Finite Element Analysis
450.soplex	C++	Linear Programming
453.povray	C++	Image processing / ray-tracing
454.calculix	C,Fortran	Structural mechanics
459.GemsFDTD	Fortran	Computational E&M
465.tonto	Fortran	Quantum Chemistry
470.lbm	C	Fluid Dynamics
481.wrf	C,Fortran	Weather simulation
482.sphinx3	C	Speech recognition

Reproducibility

- Benchmarking is a laboratory experiment, and needs to be documented as fully as a well-run chemistry experiment
  - Identify each variable hardware component
  - Identify compiler flags and measure variability
  - Verify reproducibility and provide data for others to reproduce the benchmarking results

Reporting results

- Example of performance for SPEC CFP2000 benchmark is in H&P Figure 1.14
  - for Sun Ultra5, AMD Opteron, Intel Itanium 2
  - need to look on SPEC web site for all the parameters of the machines, including
    - » data on hardware - CPU (how many, primary and secondary caches), memory, disk
    - » data on software – OS, compilers, file system type, and compiler flags used (very important!)
- How to summarize results?

Sample results

From Figure 1.15 in H&P 3/e – which machine is fastest?

	Computer A	Computer B	Computer C
Program P1 (sec)	1	10	20
Program P2 (sec)	1000	100	20
Program P3 (sec)	1001	110	40

Statistical reporting

- "It is rare indeed when advertisers, politicians, pop economists, and drumbeaters for medical programs offer a statistical argument that is not either misleading or downright deceptive." - Martin Gardner
- "... in mathematics you don't understand things, you just get used to them." - John von Neumann
- "... if you torture your data long enough, they will tell you what you want to hear." - James L. Mills, M.D.

Statistical reporting (cont.)

- The average execution time is the arithmetic mean:
  - sum the execution times for each program and divide by the number of programs *n*
- The harmonic mean is *n* divided by the sum of some function of each execution time
- Often, both of these means are modified to give more weight to more important programs

Statistical reporting (cont.)

- Arithmetic mean is good if the weights represent your own workload - then you can compare how each machine will perform for you
  - *Don't* set the weights based on performance on any particular machine; can get confusing results by doing that
- Harmonic mean gives same sort of information, but is used when rates (e.g., instructions per second) are measured instead of times

Geometric mean

- To measure relative performance, use the geometric mean
  - Choose a *reference machine*, divide all execution times by the corresponding times on the reference machine, multiply those ratios together, and take the *n*th root of the product
- Geometric means have the nice property that you get consistent results (relative performance) regardless of which machine is used to normalize (Figure 1.14)

REPORTING SPEC BENCHMARK RESULTS

How Summarize Suite Performance (1/3)

- Arithmetic average of execution time of all pgms?
  - But they vary by 4X in speed, so some would be more important than others in arithmetic average
- Could add a weights per program, but how pick weight?
  - Different companies want different weights for their products
- SPECRatio: Normalize execution times to reference computer, yielding a ratio proportional to performance =

$$\frac{\text{time on reference computer}}{\text{time on computer being rated}}$$

How Summarize Suite Performance (2/3)

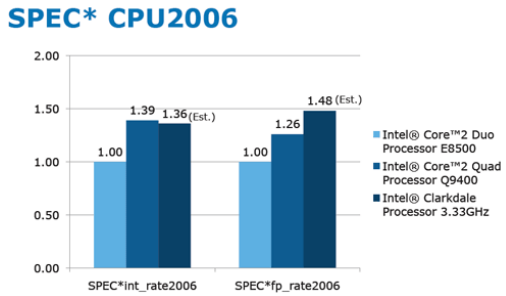
- **If program SPECRatio on Computer A is 1.25 times bigger than Computer B, then**

$$1.25 = \frac{SPECRatio_A}{SPECRatio_B} = \frac{\frac{ExecutionTime_{reference}}{ExecutionTime_A}}{\frac{ExecutionTime_{reference}}{ExecutionTime_B}} = \frac{ExecutionTime_B}{ExecutionTime_A} = \frac{Performance_A}{Performance_B}$$

- **Note that when comparing 2 computers as a ratio, execution times on the reference computer drop out, so choice of reference computer is irrelevant**

How Summarize Suite Performance (2/3)

- **Example**



How Summarize Suite Performance (3/3)

$$GeometricMean = \sqrt[n]{\prod_{i=1}^{n} SPECRatio_i}$$

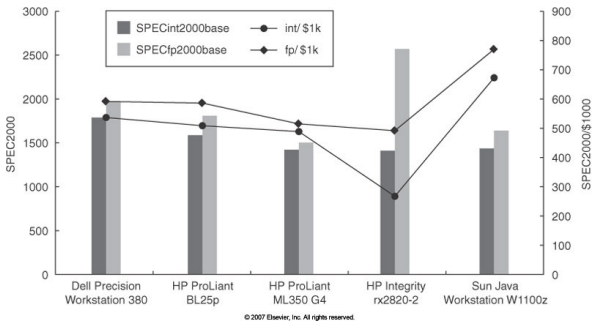
COMPUTER PRICE/PERFORMANCE

Performance/Price Performance

For desktop systems (H&P Figure 1.15) – 1GB ECC SDRAM, August 2005

Model	Processor	Clock rate (GHz)	Price
Dell 380	Intel PIV Xeon	3.8	$3346
HP BL25p	AMD Opteron 252	2.6	$3099
HP ML350 G4	Intel PIV Xeon	3.4	$2907
HP rx2620-2	Intel Itanium 2	1.6	$5201
Sun Java WS W1100z	AMD Opteron 150	2.4	$2145

SPEC CINT2000 & CFP2000 – H&P 1.16



IMPROVING COMPUTER PERFORMANCE – AMDAHL'S LAW

How to make computers faster

- Make the common case faster!
- Example: Put more effort and funds into optimizing the hardware for addition than to optimize square root
- Amdahl's law quantifies this principle:
  - Define speedup as the time the task took originally divided by the time the task takes after improvement

Amdahl's Law

- Then Amdahl tells us what the speedup of a particular task is, given
  - fraction *f* of the original execution time that the task could use the improvement
  - the speedup *s* of a task that always uses the improvement
- Then what is the speedup of the task?

Amdahl's Law (cont.)

- Suppose that the original task runs for 1 second so it takes *f* seconds in the critical piece, and *1-f* in other things
- Then the task on the improved machine will take only *f/s* seconds in the critical piece, but will still take *1-f* seconds in other things
- speedup = $\frac{old\_time}{new\_time}$ = $\frac{1}{(1-f)+\frac{f}{s}}$

Example 1

- Suppose we work very hard improving the square root hardware, and that our task originally spends *1%* of its time doing square roots. Even if the improvement reduces the square root time to zero, the speedup is no better than
- speedup = $\frac{1}{1-f}$ = $\frac{1}{.99}$ = 1.01
- *And we might be better off putting our effort into a more important part of the task*

Example 2

- Suppose that, for the same cost, we can speed up integer arithmetic by a factor of 20, or speed up floating point arithmetic by a factor of 2. If our task spends *10%* of its time in integer arithmetic, and *40%* of its time in floating point arithmetic, which should we do?

Example 2 (cont.)

- Option 1:
  speedup = $\frac{1}{.9 + \frac{.1}{20}}$ = 1.105
- Option 2:
  speedup = $\frac{1}{.6 + \frac{.4}{2}}$ = 1.25

CPU Performance

- More jargon …
- Something new happens in a computer during every clock cycle
- The clock rate is what manufacturers usually advertise to indicate a chip's speed:
  - e.g., a 2.8GHz Pentium 4
- But how fast the machine is depends on the clock rate *and* the number of clock cycles per instruction (CPI)

CPU Performance (cont.)

- So total CPU time = instruction count (IC) times CPI divided by clock rate (MHz/GHz)
  - IC * CPI / GHz
- There's an example on p. 43 that illustrates the overall effect
- Note: CPI is not a good measure of performance all by itself since it varies by type of instruction!

How to design a fast computer

- Amdahl's law says put your effort into optimizing instructions that are often used.
- The locality of reference rule-of-thumb says that a program spends *90%* of its time in *10%* of the code, so we should put effort into taking advantage of this, through a memory hierarchy
- For data accesses, 2 types of locality
  - temporal
  - spatial

Take advantage of parallelism

- At system level
  - e.g., use multiple CPUs (Unit 7), disks (Unit 6)
- At processor level
  - among instructions (Unit 4)
  - pipelining (Unit 3)
- At digital design level
  - e.g., set associative caches (Unit 5), carry-lookahead in ALU design