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

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Agenda

- Review
- Benchmarks and Summarizing Performance
- Administrivia
- Technology Break
- Measuring Performance, with Examples

Review

- Time (seconds/program) is measure of performance

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

- Algorithms, Programming Languages, Compilers, Instruction Set affect Instruction Count and CPI
- Instruction Set affects Clock Period/ Clock Rate
- Benchmarks stand in for real workloads to as standardized measure of relative performance

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*: reference execution time on old reference computer divide by execution time on new computer

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

- Barcelona varies from 6 times to 22 times faster than reference computer
  - Average (Arithmetic Mean) is 12.6, Median is 11.5
- *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 as reference
- Geometric Mean for Barcelona is 11.7

SPECPower

- Given rising importance of power and energy, create benchmark for performance and power
- Most servers in Warehouse Scale Computer have avg. utilization between 10% & 50%, so measure power at medium load as well as at high load
- Measure best performance and power, then reduce request rate so that see power for every 10% reduction in performance
- Java server benchmark performance is *operations per second* (*ssj_ops*), so metric is *ssj_ops/Watt*

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

Energy Proportional Computing

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

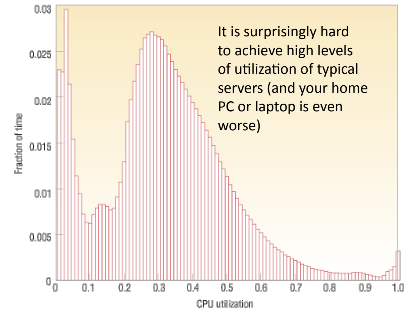
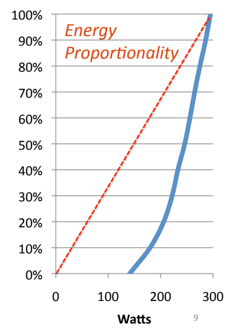


Figure 1. Average CPU utilization of more than 5,000 servers during a six-month period. Servers are rarely completely idle and seldom operate near their maximum utilization, instead operating most of the time at between 10 and 50 percent of their maximum

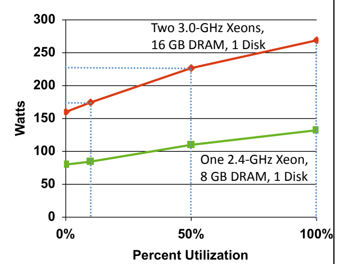
SPECPower on Barcelona

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?

- 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

Agenda

- Review
- Benchmarks and Summarizing Performance
- Administrivia
- Technology Break
- Measuring Performance, with Examples

Other Attempts at Benchmarks

- Rather than run a collection of real programs and take their average (geometric mean), instead 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
- 2nd 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
  - See Lecture Wednesday
- Yet still used in hand held, embedded CPUs!

EE Times Articles

"Samsung and Intrinsity announced they have 1st silicon for Humming bird, an ARM Cortex A8 that … delivers more than 2,000 Dhrystone Mips while consuming 640 mW power" 7/24/09

Compiled Size of Dhystone 9/7/2010

Architecture	Enhanced 8051	Generic MSP430	MSP430F5438 (large memory model)	ARM Cortex-M0	ARM Cortex-M3
Tools	Keil µVision 3.8 PK51 8.18	IAR Embedded Workbench 4.20.1	IAR Embedded Workbench 4.20.1	RVDS 4.0-SP2 with MicroLIB	RVDS 4.0-SP2 with MicroLIB
Program size in bytes*	3186 8 BIT	923 16 BIT	1079 16 BIT	912 32 BIT	900 32 BIT

*All of the compiled results are optimized for size.

Compiler Optimization and Dhrystone

- gcc compiler options
  - -O1: the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time
  - -O2: Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code
  - -O3: Optimize yet more. All -O2 optimizations and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload, -ftree-vectorize and -fipa-cp-clone options

Detailed –O1, -O2 Optimizations

```
-fauto-inc-dec
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fguess-branch-probability
-fif-conversion2
-fif-conversion
-fipa-pure-const
-fipa-profile
-fipa-reference
-fmerge-constants
-fsplit-wide-types
-ftree-bit-ccp
-ftree-builtin-call-dce
-ftree-ccp
-ftree-ch
-ftree-copyrename
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-sra
-ftree-pta
-ftree-ter
-funit-at-a-time
```

```
-fthread-jumps
-falign-functions  -falign-jumps
-falign-loops  -falign-labels
-fcaller-saves
-fcrossjumping
-fcse-follow-jumps  -fcse-skip-blocks
-fdelete-null-pointer-checks
-fexpensive-optimizations
-fgcse  -fgcse-lm
-finline-small-functions
-findirect-inlining
-fipa-sra
-foptimize-sibling-calls
-fpartial-inlining
-fpeephole2
-fregmove
-freorder-blocks  -freorder-functions
-frerun-cse-after-loop
-fsched-interblock  -fsched-spec
-fschedule-insns  -fschedule-insns2
-fstrict-aliasing -fstrict-overflow
-ftree-switch-conversion
-ftree-pre
-ftree-vrp
```

http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

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 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		

Where do you spend the time in your program?

- Profiling program shows which where spend time by function, which code uses most of time
  - E.g., gprof
- Usually a 90/10 rule, where 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
- 3 steps:

1. Compile & link program with profiling enabled
   - cc –pg x.c *{in addition to other flags use}*
2. Execute program to generate a profile data file
3. 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

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

Summary

- Benchmarks stand in for real workloads to as standardized measure of relative performance
  - Synthetic programs don't work, but some still use them!
- 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!