ECE 454
Computer Systems Programming
Measuring and profiling

Ding Yuan
ECE Dept., University of Toronto
http://www.eecg.toronto.edu/~yuan

“It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories instead of theories to suit facts.” - Sherlock Holmes
Measuring Programs and Computers

Why Measure a Program/Computer?

- To compare two computers/processors
  - Which one is better/faster? Which one should I buy?
- To optimize a program
  - Which part of the program should I focus my effort on?
- To compare program implementations
  - Which one is better/faster? Did my optimization work?
- To find a bug
  - Why is it running much more slowly than expected?
Basic Measurements

- IPS: instructions per second
- MIPS: millions of IPS
- BIPS: billions of IPS
- FLOPS: floating point operations per second
  - megaFLOPS: $10^6$ FLOPS
  - gigaFLOPS: $10^9$ FLOPS
  - teraFLOPS: $10^{12}$ FLOPS
  - petaFLOPS: $10^{15}$ FLOPS
  - Eg: playstation3 capable of 20 GFLOPS
- IPC: instructions per processor-cycle
- CPI: cycles per instruction
  - CPI = 1 / IPC

How not to compare processors

- Clock frequency (MHz)?
  - IPC for the two processors could be radically different
  - Megahertz Myth
    - Started from 1984

Apple II
CPU: MOS Technology 6503@1MHz
LD: 2 cycles (2 microseconds)

IBM PC
CPU: Intel 8088@4.77MHz
LD: 25 cycles (5.24 microseconds)
How not to compare processors

- Clock frequency (MHz)?
  - IPC for the two processors could be radically different
- CPI/IPC?
  - dependent on instruction sets used
  - dependent on efficiency of code generated by compiler
- FLOPS?
  - only if FLOPS are important for the expected applications
  - also dependent on instruction set used

How to measure a processor

- Use wall-clock time (seconds)

\[ \text{time} = IC \times CPI \times \frac{1}{\text{ClockFrequency}} \]

- IC = instruction count (total instructions executed)
- CPI = cycles per instruction
- ClockPeriod = 1 / ClockFrequency = (1 / MHz)
Amdahl’s Law:
Optimizing part of a program

\[
\text{speedup} = \frac{\text{OldTime}}{\text{NewTime}}
\]

- Eg., my program used to take 10 minutes
- now it only takes 5 minutes after optimization
- speedup = 10min/5min = 2.0 i.e., 2x faster

- If only optimizing part of a program (on following slide):
  - let \( f \) be the fraction of execution time that the optimization applies to (\( 1.0 > f > 0 \))
  - let \( s \) be the improvement factor (speedup of the optimization)

Amdhal’s Law Visualized

the best you can do is eliminate \( f \); 1-f remains
Amdahl’s Law: Equations

- let \( f \) be the fraction of execution time that the optimization applies to (\( 1.0 > f > 0 \))
- let \( s \) be the improvement factor

\[
\text{NewTime} = \text{OldTime} \times [(1-f) + f/s]
\]

\[
\text{speedup} = \text{OldTime} / (\text{OldTime} \times [(1-f) + f/s])
\]

\[
\text{speedup} = 1 / (1 - f + f/s)
\]

Example 1: Amdahl’s Law

- If an optimization makes loops go 3 times faster, and my program spends 70% of its time in loops, how much faster will my program go?

\[
\text{speedup} = 1 / (1 - f + f/s)
\]

\[
= 1 / (1 - 0.7 + 0.7/3.0)
\]

\[
= 1 / (0.533333)
\]

\[
= 1.875
\]

- My program will go 1.875 times faster.
Example 2: Amdahl’s Law

- If an optimization makes loops go 4 times faster, and applying the optimization to my program makes it go twice as fast, what fraction of my program is loops?

Implications of Amdahl’s Law

- Optimize the common case
- The common case may change!
Tools for Measuring and Understanding Software

- Software Timers
  - C library and OS-level timers

- Hardware Timers and Performance Counters
  - Built into the processor chip

- Instrumentation
  - Decorates your program with code that counts & measures
  - gprof
  - gcov

GNU: “Gnu is Not Unix”
--- Founded by Richard Stallman
Software Timers: Command Line

- Example: `/usr/bin/time`
  - Measures the time spent in user code and OS code
  - Measures entire program (can't measure a specific function)
  - Not super-accurate, but good enough for many uses

- `$ time ls` (used in HW1)

<table>
<thead>
<tr>
<th>real</th>
<th>user</th>
<th>sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0m13.860s</td>
<td>0m10.669s</td>
<td>0m0.720s</td>
</tr>
<tr>
<td>real</td>
<td>user</td>
<td>sys</td>
</tr>
<tr>
<td>0m3.515s</td>
<td>0m10.837s</td>
<td>0m0.672s</td>
</tr>
</tbody>
</table>

- user & sys --- CPU time
- `/usr/bin/time` gives you more information

Software Timers: Library: Example

```c
#include <sys/times.h>  // C library functions for time
unsigned get_seconds() {
  struct tms t;
  times(&t);  // fills the struct
  return t.tms_utime;  // user program time
                     // (as opposed to OS time)
}

...  // used in HW2
unsigned start_time, end_time, elapsed_time;
start_time = get_seconds();
do_work();    // function to measure
end_time = get_seconds();
elapsed_time = end_time - start_time;
```

- can measure within a program
  - used in HW2
Hardware: Cycle Timers

- Programmer can access on-chip cycle counter
  - Eg., via the x86 instruction: rdtsc (read time stamp counter)
  - We use this in hw2:clock.c:line94 to time your solutions

- Example use:
  - start_cycles = get_tsc(); // executes rdtsc
  - do_work();
  - end_cycles = get_tsc();
  - total_cycles = end_cycles - start_cycles;

- Can be used to compute #cycles to execute code
- Watch out for multi-threaded program!

(can be more accurate than library (if used right) used in HW2)

Hardware: Performance Counters

- Special on-chip event counters
  - Can be programmed to count low-level architecture events
  - Eg., cache misses, branch mispredictions, etc.

- Can be difficult to use
  - Require OS support
  - Counters can overflow
  - Must be sampled carefully

- Software packages can make them easier to use
  - Eg: Intel’s VTUNE, perf (recent linux)

(perf used in HW2)
Instrumentation

- Compiler/tool inserts new code & data-structures
  - Can count/measure anything visible to software
  - Eg., instrument every load instruction to also record the load address in a trace file.
  - Eg., instrument every function to count how many times it is called

- “Observer effect”:
  - Can’t measure system without disturbing it
  - Instrumentation code can slow down execution

- Example instrumentors (open/freeware):
  - Intel’s PIN: general purpose tool for x86
  - Valgrind: tool for finding bugs and memory leaks
  - gprof: counting/measuring where time is spent via sampling

Instrumentation: Using gprof

- gprof: how it works
  - Periodically (~ every 10ms) interrupt program
  - Determine what function is currently executing
  - Increment the time counter for that function by interval (e.g., 10ms)
  - Approximates time spent in each function, #calls made
  - Note: interval should be random for rigorous sampling!

- Usage: compile with “-pg” to enable
  ```
gcc -O2 -pg prog.c -o prog
./prog
  ```
  - Executes in normal fashion, but also generates file gmon.out
  - Generates profile information based on gmon.out

  used in HW1
  
  detailed example later in lecture
Instrumentation: Using gcov

- Gives profile of execution within a function
  - Eg., how many times each line of C code was executed
  - Can decide which loops are most important
  - Can decide which part of if/else is most important

- Usage: compile with “-g -fprofile-arcs -ftest-coverage” to enable
  gcc -g -fprofile-arcs -ftest-coverage file.c -o file.o
  ./prog
  - Executes in normal fashion
  - Also generates file.gcda and file.gcno for each file.o
  gcov -b prog
  - Generates profile output in file.c.gcov

- used in HW1

Emulation/Instrumentation: valgrind

- Primarily used to find/track memory leaks
  - Eg., if malloc() an item but forget to free it
  - Many other uses for it these days

- valgrind is a fairly sophisticated emulator
  - a virtual machine that just-in-time (JIT) compiles
  - adds instrumentation dynamically (without rerunning gcc)
  - emulates 4-5x slower than native execution

- Usage: (available on ug machines)
  valgrind myprogram
  == LEAK SUMMARY:
  == definitely lost: 0 bytes in 0 blocks
  == indirectly lost: 0 bytes in 0 blocks
  == possibly lost: 0 bytes in 0 blocks
  == still reachable: 330,372 bytes in 11,148 blocks
Demo:
Using gprof