ECE 454
Computer Systems Programming

Measuring and Profiling

Ashvin Goel
ECE Dept, University of Toronto
http://www.eecg.toronto.edu/~ashvin
“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, e.g., improve algorithm
  - 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
  - E.g.: 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)

\[
time = IC \times CPI \times \text{ClockPeriod}
\]

- IC = instruction count (total instructions executed)
- CPI = cycles per instruction
- ClockPeriod = seconds/cycle
  \[= \frac{1}{\text{ClockFrequency}} = (1 / \text{MHz})\]
Amdahl’s Law: Optimizing Part of a Program

speedup = OldTime / NewTime

- E.g., 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 ($0 < f < 1$)
  - 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 ($0 < f < 1$)

• Let $s$ be the improvement factor (speedup of the optimization)

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

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

\[
\text{speedup} = \frac{1}{1 - f + \frac{f}{s}}
\]
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?

\[
\text{speedup} = \frac{1}{1 - f + \frac{f}{s}}
\]

\[
= \frac{1}{1 - 0.7 + \frac{0.7}{3.0}}
\]

\[
= \frac{1}{0.533333}
\]

\[
= 1.875
\]

• 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

When you optimize the common case, the common case may change!
Tools for Measuring and Understanding Software
Tools for Measuring/Understanding

- **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 – profiling tool, outputs where time is spent in program
  - gcov – coverage tool, outputs how many times each line executed
  - Used together to find commonly executed code where time should be spent on optimization
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

```
real  0m13.860s  real  0m3.515s
user  0m10.669s  user  0m10.837s
sys   0m0.720s   sys   0m0.672s
```

- real – Wall clock time
- user & sys --- CPU time in user mode, kernel mode

Used in Lab 1
Software Timers: Library: Example

- Used to measure time within parts of a program

```c
#include <sys/times.h>  // C library functions for time

unsigned get_time() {
    struct tms t;
    times(&t);  // fills the struct
    // user CPU time (as opposed to OS CPU time)
    return t.tms_utime;
}

unsigned start_time, end_time, elapsed_time;
start_time = get_time();
do_work();  // function to measure
end_time = get_time();
elapsed_time = end_time - start_time;
```

Used in Lab 2
Hardware: Cycle Timers

- Programmer can access on-chip cycle counter
- E.g., via the x86 instruction: rdtsc (read time stamp counter)
- We use this in Lab 2 in clock.c to time your solutions, e.g.

```c
start_cycles = get_tsc(); // executes rdtsc
do_work();
end_cycles = get_tsc();
total_cycles = end_cycles - start_cycles;
```

- Can be used to compute #cycles required to execute code
  - Can be more accurate than library (when used correctly)
- Watch out for multi-threaded programs!
Hardware: Performance Counters

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

- Previously, difficult to use
  - Full OS support was missing
  - Counters can overflow
  - Must be sampled carefully

- Today, software packages can make them easier to use
  - E.g.: Intel’s VTUNE, Linux perf

- We use perf in Lab 2
Instrumentation

• Compiler/tool inserts new code & data-structures
  • Can count/measure anything visible to software
  • E.g., instrument every load instruction to record load address
  • E.g., instrument every function to count #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: counts & measures where time is spent via sampling
Instrumentation: Using gprof

- **gprof**
  - Uses sampling to approximate time spent in each function and # of calls to each function
  - Periodically interrupts program, e.g., roughly every 10ms
  - Determines what function is currently executing
  - Increments the time counter for that function by interval (10ms)

- **Usage:** compile with `-pg`

  ```
  gcc -O2 -pg prog.c -o prog
  ./prog
  gprof prog
  ```

  Executes normally, but also generates file `gmon.out`
  Uses `gmon.out` to show profiling information

*Used in Lab 1*
Instrumentation: Using gcov

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

- Usage: compile with `-g -fprofile-arcs -ftest-coverage`

  ```bash
  gcc -g -fprofile-arcs -ftest-coverage file.c -o file.o
  ```

- ./prog
  - Executes normally, but also generates files file.gcda and file.gcno for each file.o

- gcov -b prog
  - Generates profile output in file.c.gcov

Used in Lab 1
Emulation/Instrumentation: valgrind

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

- valgrind adds instrumentation to the binary dynamically
  - So gcc doesn’t need to be rerun
  - Execution time is 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
Demo: Using gcov