Measurements

General concepts

• Concepts that are often confused
• Repeatability vs. replicability
  • repeatability: repeating measurement on same setup gives more or less the same results
  • replicability: performing measurement on a different but similar setup gives more or less the same results
• Precision vs. accuracy
  • precision: how close are measurements to each other?
    • random error: remove by repeating measurement and taking average
  • accuracy: how close is measurement of a quantity to the actual value of that quantity (ground truth)?
    • systematic error

Overview of lecture

• To understand and improve program performance, need insight into program behavior
  • execution time of program
  • how well program exploits hardware resources
    • memory hierarchy: cache hits and misses
    • processor pipeline: stalls
    • vector units
    • ...
• Measurements
  • basic ideas are quite simple
  • however processors are very complex so getting accurate measurements can be difficult
  • you must have a mental model of how processors execute instructions to make sensible measurements
• Libraries like PAPI simplify some measurements
• Intel Vtune gives you a deep look into program behavior but only on Intel processors

Timing your code
**Basic idea**

- Assume there is a way to get “current time” on the computer
  - for now, don’t worry about precise definition of “current time”
- Timing your code
  - Use the pseudocode on right
- Problems
  - definition of “current time” can be quite subtle
  - modern computer systems are so complex that you may not be measuring what you think you are measuring
  - usually your code is written in C or some other high-level language and compiler may transform your code in unexpected ways
  - ...

**Main issues**

1. Initial conditions matter
   - measured time may depend on state of machine when timing starts
2. Resolution and precision of timer
   - granularity of your measuring device
   - spread in measurements
3. Heisenberg effect
   - measurement may change quantity you are measuring
4. Compiler optimizations
   - may need to look at actual assembly code to make sure compiler has not modified your code in unexpected ways
5. Context-switching by O/S and hardware interrupts
   - you may end up measuring stuff outside your code
6. Out-of-order execution of instructions
   - what you measure may not be what you think you are measuring

**Main issues (1): Initial conditions**

- Computers have a lot of internal state
  - caches, TLBs,....
- Internal state when measurement starts can affect execution time
  - are instructions in I-cache when measurement starts?
  - are memory locations accessed by your code in caches or in memory?
  - Fix: if you want to measure time starting with nothing in the cache, allocate a big array and walk over it to remove your data from cache in between runs

**Main issues (2): Resolution and precision**

- Resolution:
  - how small a quantity can the device measure?
  - example: you can use a tape measure to measure cloth for a suit but not to measure how wide a hydrogen atom is
- If code in R is just a few instructions, your timer may not have resolution to measure this
  - what if timer only measured milliseconds?
  - what about overhead of getCurrentTime itself?
  - Fix: see next slide
- Precision:
  - assuming resolution is not a problem, how variable is the measurement?
  - if you repeat it ten times, how wide is the spread of measurements?
  - Fix: eliminate random error by repeating measurements and taking mean
Main issues(3): Heisenberg effect

• One solution to resolution problem:
  • put a loop around your code and execute it N times
  • divide (tick-tick) by N

• Problems:
  • loop code may change context of measurement
    • if loop counter is allocated to a register, does that affect register allocation in your code?
    • are your instructions still in I-cache?
  • you are including loop overhead in your measurement

Main issues(4): compiler optimizations

• Compiler can optimize your code in unexpected ways so you measure something different from what you are expecting

• Example:
  • to eliminate effect of loop overhead in previous slide, you can try to measure execTime with and without your code in the loop body
  • however, compiler might optimize away the loop in the second piece of code since the loop body is empty

• Solutions
  • examine assembly code to ensure compiler is not changing code in unexpected ways
  • if it is, disable compiler optimizations (but this can change what you are measuring in undesirable ways)
  • you can tweak code to trick compiler to stop it from doing undesirable things

Main issues(5): Context-switching

• Code in R may not be executed in one shot by OS and processor
• OS may de-schedule your process while executing R, schedule code from other processes, and then get back to executing code from R
• This may happen many times during execution of R

• Analogy:
  • taking an exam vs. doing an assignment
  • What is getCurrentTime measuring?
    • if it is elapsed time like “wall-clock time”, process switches will confound your measurement

• Solutions:
  • disable process switches and interrupts before executing code in region (but you may not be able to do this in user mode)
  • find a timer that advances only when processor is executing your program
  • but context-switches may still pollute your caches

Main issues(6): Out-of-order execution of instructions

• Modern processors execute instructions out of program order
  • but ensure dependences are satisfied

• Problem:
  • code from region R may get executed outside of tick and tock
  • code from outside region R may get executed between tick and tock

• Solution:
  • need to insert serializing instructions around region R
    • “fence off” instructions being timed from other instructions
    • similar to memory fences but for instructions of all types, not just memory operations
Example of measurement mistake

- **Data structure**
  - array, each element accessed once
  - little spatial locality
- **Algorithm:** $O(n \log(n))$
- **Time per element**
  - should be almost constant
  - yet pronounced jump at a particular array size
- **Reason?**

Drilling down

- **Key questions:**
  - What can we use for “getCurrentTime” and what is its resolution?
  - How do we avoid timing errors from process-switches and interrupts?
  - How do we insert serialization instructions at tick and tock?

**Answer** is very system-dependent but we will discuss three solutions for C/Linux/x86:

- Linux call: `clock_gettime`
- x86 code
- PAPI counters

```
clock_gettime

#include <time.h>

struct timespec { time_t tv_sec; /* seconds */ long tv_nsec; /* nanoseconds */};

int clock_gettime(clockid_t clk_id, struct timespec *tp);

#include <stdio.h> /* for printf */
#include <stdint.h> /* for uint64 */
#include <time.h> /* for clock_gettime */

main(int argc, char **argv)
{
    uint64_t execTime; /* time in nanoseconds */
    struct timespec tick, tock;

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tick);
    /* do stuff */
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tock);

    execTime = 1000000000 * (tock.tv_nsec - tick.tv_nsec) + tick.tv_nsec - tick.tv_nsec;
    printf("elapsed process CPU time = \%llu nanoseconds\n", (long long unsigned int) execTime);
}
```

Implementation of `clock_gettime` should use serialization instructions. `CLOCK_PROCESS_CPUTIME_ID` measures the amount of time spent in this process. Resolution on systems I used is 1 nanosecond.

Even if /*do stuff*/ is empty, `execTime` is about 2000 nanosec on these systems.
x86 code

- **Getting time:**
  - **TSC:** 64-bit time-stamp counter that tracks cycles
  - **RDTSC** instruction: read time-stamp counter
  - EDX $\leftarrow$ high-order 32 bits of counter
  - EAX $\leftarrow$ low-order 32 bits of counter
  - no serialization guarantee
  - **RDTSCP** instruction
    - waits until all previous instructions have been executed before reading counter
    - however following instructions may begin execution before read is performed
- **Serialization instruction:**
  - **CPUID** instruction
    - modifies EAX, EBX, ECX, EDX registers
    - can be executed at any privilege level

Further reading

- **Linux man pages:**
  - describes clock_gettime and other clocks
  - [https://linux.die.net/man/3/clock_gettime](https://linux.die.net/man/3/clock_gettime)
- **Technical note from Intel:**
  - shows how to use RDTSC and CPUID for accurate timing measurements

Hardware counters

- Modern CPUs have hardware counters for many events
  - Cycles
  - Instructions
  - Floating-point instructions
  - Loads and stores
  - L1 cache misses
  - L1 data cache misses
  - L2 cache misses
  - TLB misses
  - Pipeline stalls
  - ...
- **Complications**
  - accessing counters directly can be complex
  - code is not portable
  - on many processors, fewer hardware counters than events you can track so only a subset of events can be measured in a given run

PAPI counters
PAPI

- **Performance Application Programming Interface**
- Two interfaces to underlying counter hardware:
  - **High-level interface**: provides ability to start, stop and read counters for a specified list of events
  - **Low-level interface**: manages hardware events in user-defined groups called EventSets
- Timers and system information
- C and Fortran bindings
- PAPI interface to performance counters supported in the Linux 2.6.31 kernel

**PAPI design**

<table>
<thead>
<tr>
<th>Tools</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Portable Layer</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Machine Specific Layer</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PAPI Machine Dependent Substrate</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Kernel Extension</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Operating System</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Hardware Performance Counters</strong></td>
<td></td>
</tr>
</tbody>
</table>

**PAPI Events**

- **Preset events**
  - platform-independent names for events deemed useful for performance tuning
  - examples: accesses to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit and pipeline utilization
  - run PAPI papi_avail utility to determine preset events available on platform
  - PAPI also provides access to native events through low-level interface
    - may be platform-specific

**PAPI preset events**

- PAPI_L1_DCM: Level 1 data cache misses
- PAPI_L1_DCA: Level 1 data cache accesses
- PAPI_L1_ICM: Level 1 I-cache misses
- PAPI_L2_DCM: Level 2 data cache misses
- PAPI_L3_DCM: Level 3 data cache misses
- PAPI_FXU_IDL: cycles floating-point units are idle
- PAPI_TOT_INS: total instructions executed
- PAPI_TOT_CYC: total cycles
- PAPI_IPS: instructions executed per second
  - ....
PAPI_query_event

- Check whether CPU can measure the PAPI event you are interested in.

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <papi.h>

int main( int argc, char *argv[] ) {
    int i, j, k;
    long long counters[3];
    int PAPI_events[] = {
        PAPI_TOT_CYC,
        PAPI_L2_DCM,
        PAPI_L2_DCA
    };
    PAPI_library_init(PAPI_VER_CURRENT);
    i = PAPI_start_counters( PAPI_events, 3 );
    /* your code here */
    PAPI_read_counters( counters, 3 );
    printf("%lld L2 cache misses (%.3lf%% misses) in %lld cycles\n",
        counters[1],
        (double)counters[1] / (double)counters[2],
        counters[0] );
    return 0;
}
```

High Level API

- Meant for application programmers wanting simple but accurate measurements.
  - calls the lower level API
- Eight important functions:
  - `PAPI_num_counters`: How many hardware counters are supported?
  - `PAPI_start_counters`: Start counters
  - `PAPI_stop_counters`: Stop counters
  - `PAPI_read_counters`: Read counters
  - `PAPI_accum_counters`: Adds counters into accumulator array and zeroes them
  - `PAPI_flops`: Floating-point operations per second
  - `PAPI_flips`: Floating-point instructions per second
  - `PAPI_ipc`: Instructions per cycle
Finding timings with Intel® VTune™ Amplifier

Advanced Hotspots Analysis collects clock-ticks and instructions retired

GUI or CLI Collections

Finding timings with Intel® VTune™ Amplifier

Use Hotspots or Hardware Events Viewpoints

Summary

- Measurement
  - basic ideas are quite simple
  - however processors are very complex so getting accurate measurements can be difficult
  - You must have a mental model of how processors execute instructions
  - Libraries like PAPI simplify some measurements
  - Intel VTune gives you an even deeper look into what your program is doing but only on Intel processors