

Linux Systems and Open Source Software

Basics of Performance Analysis Part I

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Fall 2022

CSE 141 Introduction to Computer Architecture. Andrew A. Chien. 2002 http://cseweb.ucsd.edu/groups/csag/html/teaching/cse141-f02/cse141.html
Performance Evaluation and Benchmarking, CRC Press, ISBN 9780367392161, Lizy Kurian John, Lieven Eeckhout



Outline

- What is Performance?
- Time Measurement for Performance Analysis
- Standard Time Measurement (POSIX)
- Native Linux Time Measurement
- Time Measurement for x86 Platforms



WHAT IS PERFORMANCE?



Define *Performance*

Plane	*Speed*	*DC to Paris time*	*Passengers*	*Throughput (p x mph)*
Boeing 747	610 mph	6.5 hours	470	286,700
Concorde	1350 mph	3 hours	132	178,200

Choose an airplane with better ***performance***

- To pick a fastest airplane?
- To pick a high-throughput airplane?



Define *Performance* (Cont'd)

Plane	Speed	DC to Paris time	Passengers	Throughput (p x mph)
Boeing 747	610 mph	6.5 hours	470	286,700
Concorde	1350 mph	3 hours	132	178,200

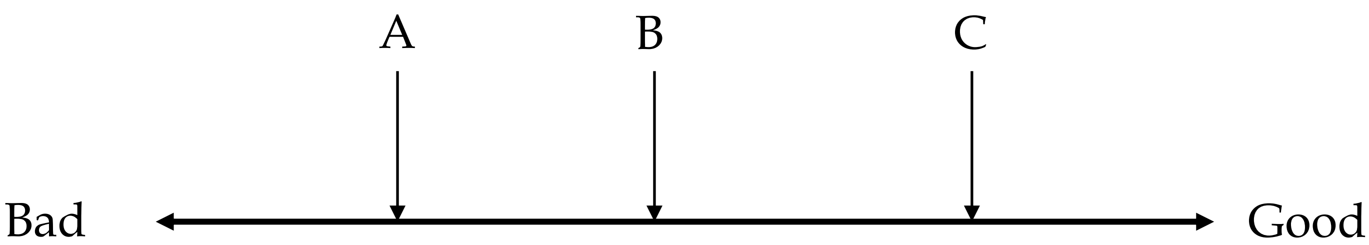
- **Time** to do the task
  - **Execution time**, response time, latency
- **Tasks** per day, hour, week, sec, ns. ..
  - **Throughput**, bandwidth



Performance Measurement

- In CS, when talk about **performance**, we should note…
- What is the **goal** here?
  - A well-tweaked program?
  - A fast machine?
  - ?
- **How** is performance measured?
  - Manually or automatically; will discussed in this course
- Example:
  How to **choose** amongst different machines?
  - Cost: price, technology metrics
  - Performance
    - **Metrics**: time and processing speed; indicate relative performance
    - E.g., run time or X's per second

Performance Metrics



- What is a **metric**?
  - Basis for measuring
  - Basis for comparison
- Metric varies across different situations
  - Job: Salary, Responsibilities; School: Grades;
  - Mutual Funds: Total Return, Risk
  - Cars: Top speed, Acceleration, Impact test
- Metrics are used to **Order** and **Compare**
- Many metrics are possible, many used



Computer Performance Metrics

- **Execution Time**
  - CPU time, wall clock time
- Millions of Instructions Per Second (MIPS)
  - Machine instructions
    small primitive units generated by the compiler
- Cycles Per Instruction (CPI)
  - Fixed length time periods, normalization for technology
- Clock Rate (Megahertz)
  - Millions of cycles per second

=> Many metrics are used
  - Some better, some worse

- **Goal**: use metrics which **reflect performance** delivered by the underlying machine to real user programs, real applications
  - Quantize the **combined effect** of the application (SW) and machine (HW)



Measuring Execution Time

- Steps
  1. Select a program
  2. Execute the program
  3. Measure the CPU time, or the wall clock (stopwatch) time

- Performance = 1 / (Execution time)
  - Smaller execution time -> better performance
  - Larger execution time -> worse performance
- Questions?
  - What to be compared? (the performance of different programs?)
  - How to measure time?



Programs under Test (What)

- Target program
  - Would like to run EXACTLY the **same program**, but…
  - Programs are large and unwieldy, input data is critical, and they and their usage would change over time
  - Need to **run entire application programs at full data set sizes** to get good performance information
  - Best **predictors**, tricky, difficult, sometimes misleading
- "*Benchmarking*"
  - How applications will really run
  - Lies? and Benchmarks
  - Common benchmark programs are used to evaluate the performance amongst ***commercial machines***
    - SPEC, TPC, Dhrystones, Antutu, etc.



Time Measurement (How)

- How to measure execution time?
  - Watch (start/stop the meter; wall clock)
  - Computer timing: User (processor time) + System (operating system -- I/O, etc.)
    - Compare based on *processor time* for *processor performance*
    - Important, but of decreasing importance for SYSTEM performance
  - Concurrent users? Measurement errors?
    (Involves What to measure!!!)
- Usually, you should know **What** and then you will know **How**

Time

- **CPU Execution Time**
  = CPU clock cycles * Clock cycle time
  = CPU clock cycles / Clock rate
- **Elapsed time (or wall-clock time)**
  - It is the actual time taken between the start and the end of the task
- **CPU Execution Time** does not necessarily mean the **wall-clock time**
  - I.e., the elapsed time for running the program may be different from the time that CPU spent on the program

- Every conventional processor has a clock with an associated clock cycle time or clock rate
- Every program runs in an integer number of clock cycles
  x MHz = x millions of cycles/second (**clock rate**)
  1/ (x MHz) = **cycle time**; 1/(500 MHz) = 2 ns



MIPS and MFLOPS

- MIPS and MFLOPS are common metrics for describing the capabilities of the machines
  - But, are they suitable for comparing performance of different machines all the time?

- MIPS: million instructions per second

  =(number of **instructions** executed in program)/

  (execution time in seconds * 10^6)

- MFLOPS: million floating point operations per second

  =(number of **floating point operations** executed in program)/(execution time in seconds * 10^6)



A Benchmarking Example

- Pentium III 2.5Ghz system, Microsoft C++ compiler
  - Compile the program, execute, count instructions
  - Measured at **<u>2,100 MIPs</u>**
- What does this tell you about performance?

- Compile again, this time with optimization **ON**!
  - Compilation takes a lot longer, execute, count instructions
  - Measured performance at **<u>1,600 MIPs</u>**

- Intuitively, the optimized version should be faster than the original version (and in fact, the optimized version takes less time)
- But, the MIPS of the former is less than that of the latter. What happened?

A Benchmarking Example (Cont'd)

of Inst_A / ExecTime_A

# of Inst_B / ExecTime_B

- There are **fewer instructions** executed in the optimized program!
  - And the optimized program takes less time
- MIPS rating depends on **compiler**
  - Quality of generated code
  - **Optimized for instruction execution time, not MIPS rating**
  - Compilers are always benchmarked with the machine
- How could you "***cheat***" to get a high MIPS rating?



Another Benchmarking Example

- Power Macintosh, 1Ghz, PowerPC G4
  - Compile same program, "optimized"
  - Execute, assuming no obvious cheating
  - Experiment produces **1,500 MIPS** rating
  - Is this faster than the Pentium III?

=> There's no easy way to tell from this information!

- Why?
  - The **unit of work** has changed.
  - **Pentium Instruction != PowerPC instruction**
- Hard to compare MIPS across architectures
  - MIPS is of little use for comparing architectures
  - **Resort to execution time**



Another Benchmarking Example (Cont'd)

- **Unit of work**
  - **The real target** when we try to benchmark a program/system
  - E.g., Instructions, Floating Point Operations, Window updates, etc.

- The benchmarking result can be considered as <u>the time taken for executing the **works**</u>, which are affected by different factors:
  - Instructions: compiler, architecture
  - Floating Point operations: compiler, algorithm
  - Window updates: algorithm
- That is, the benchmarking result is the combined effect of the HW and SW environment
  - **Depends** on compiler, architecture, algorithm, implementation, execution environment, etc.



Example: Unit of Work

- Floating Point Operations
- Window Updates
- Frames/Polygons (rendering)
- Megabytes (communication)
- Limitations of each of these?
- How can you cheat/reduce each of these?

  → Time to finish a fixed amount of work

  → Ratio of the finished amount of work per time unit



Summary

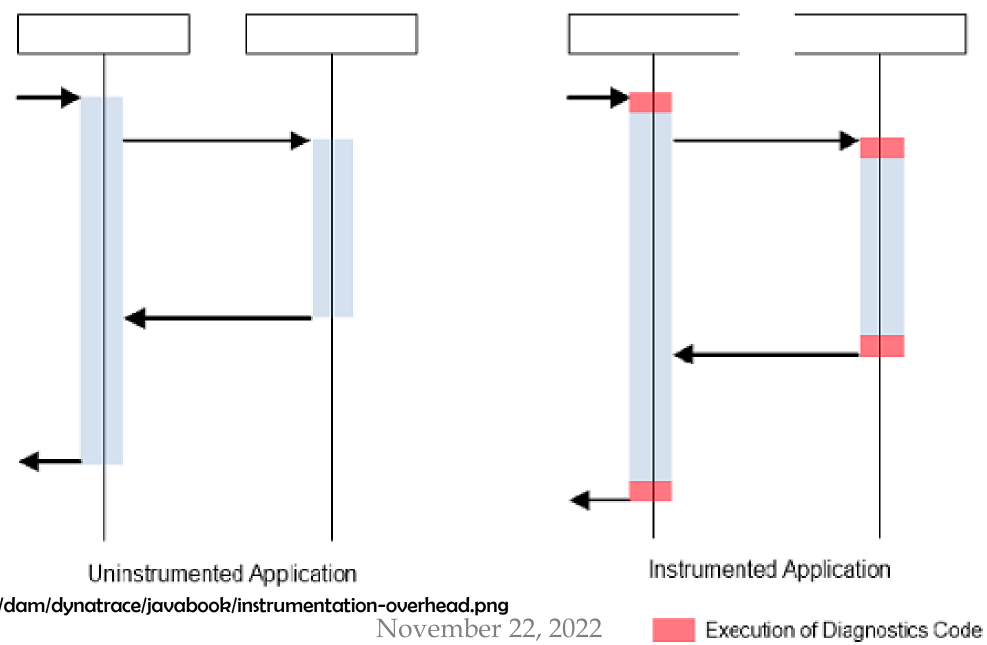
- Architecture involves a tension of programmability, compilers, implementability, and technology
  - Optimize the design given these ever changing constraints
- Many possible measures of **work** / **performance metrics**
- Choosing is rife with potential errors
- Because it includes everything, **execution time** is the safest choice
- Still need to analyze the other influences carefully before you can draw any conclusions about the causes



TIME MEASUREMENT FOR PERFORMANCE ANALYSIS

Instrumentation-based Time Measurement

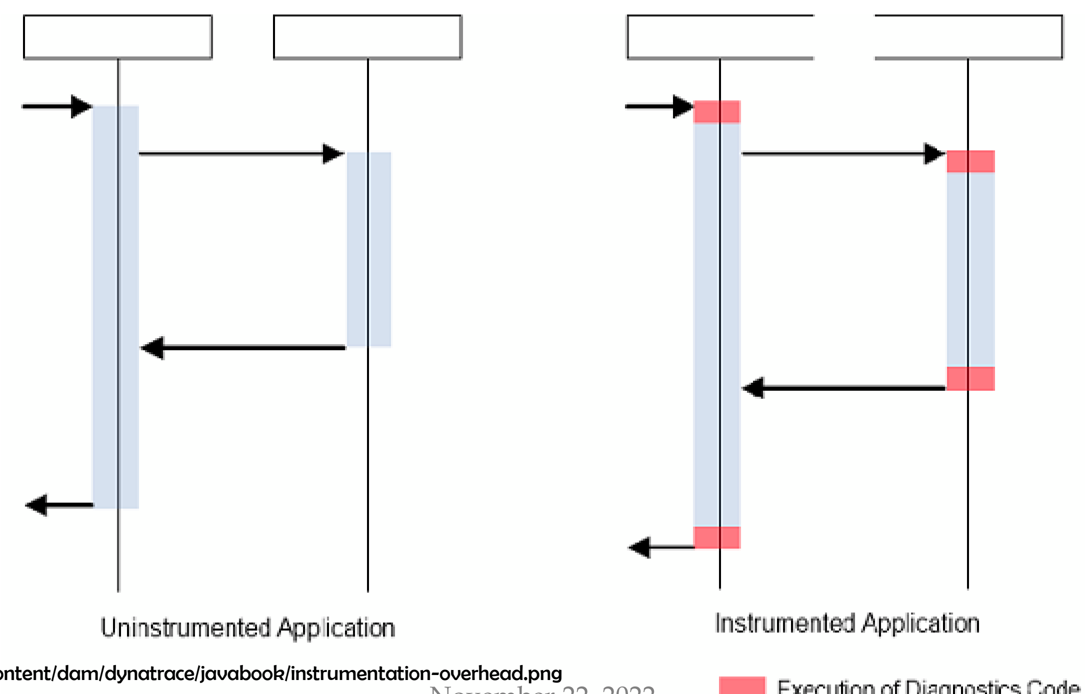
- Code instrumentation
  - Inject the *time functions* into the target application
  - It can be done manually or automatically in the source or binary program
- Example
  - Insert ***time*** at the prolog and epilog of each function to get the elapsed times of the functions



Courtesy of https://assets.dynatrace.com/content/dam/dynatrace/javabook/instrumentation-overhead.png

Watch Out for Its Overhead

- Code instrumentation overhead
  - The **red blocks** refer to the **overhead** incurred by the **time functions**
  - The overhead is often ignored; but, you should pay attention to it
  - Here, we assume it is negligible compared with the **targets** (e.g., the functions to be analyzed)
- Instrumentation always:
  - **introduces overhead** (low or high)
  - **alters the program execution**



Courtesy of https://assets.dynatrace.com/content/dam/dynatrace/javabook/instrumentation-overhead.png



Measurement of *Time* Overhead

- It is crucial to understand the **cost** of the time measurement
- You may use the following simple method to get the **overhead** of the time function you used
  - Example below shows the averaged time required by each **time()** function invocation
- Note: **one million** is a magic number

```
start = time();
for i= 1 ~ 1,000,000
   time();
end = time();

overhead = (end-start)/1,000,000
```



Note

- The following methods are simple and good for the **sequential** programs

- As for **parallel programs**, you should consider the concurrency issues
  - E.g., Lock and Wait
  - In this case, performance profiling tools should be used to characterize the runtime activities of multiple threads

STANDARD TIME MEASUREMENT



Be Careful

- The length of code for time measurement
- Unit & Resolution
  - Minute, seconds, microseconds
  - Length of code vs. Time Resolution
- Variables to hold the *time* values
  - Each time function has its own rules and way to ***represent*** the time
  - Types/Formats of the variables are important
- ***Clock source*** of the time function
  - System-wide, per-process, per-thread
- Code portability
  - Is the time measurement code portable across HW platforms?
  - E.g., Windows system has its time functions



time()

- Get time in seconds
  - Return the current calendar time or -1 if there is an error
  - If the argument time is given, then the current time is stored in time
  - Only measure the time up to seconds

```
#include <ctime>

time_t time( time_t *time );
```

clock()

- Determine processor time (clocks)
  - Return the processes CPU time (time since the program started)
  - Return -1 if that information is not available
- Conversion to seconds by division by CLOCKS_PER_SEC
  - Note: if your compiler is POSIX compliant, then CLOCKS_PER_SEC is always defined as 1,000,000
  - On a 32-bit system where CLOCKS_PER_SEC equals 1000000 this function will **return the same value approximately every 72 minutes**
  - For improved accuracy, since glibc 2.18, it is implemented on top of **clock_gettime**(2) (using the CLOCK_PROCESS_CPUTIME_ID clock)

```
#include <ctime>

clock_t clock( void );
```

Example: Time Measurement using clock()

```
#include <ctime>
// Time stamp before the computations
clock_t start = clock();
... /* Computations to be measured */
// Time stamp after the computations
clock_t end = clock();
double cpu_time = static_cast<double>( end - start ) /
CLOCKS_PER_SEC;
```

Report the elapsed time for the code section enclosed by the two **clock** functions



clock_gettime()

- High Precision Event Timer
- The functions **clock_gettime()** and **clock_settime()** retrieve and set the time of the specified clock
- The function **clock_getres()** finds the resolution (precision) of the specified clock ***clk_id*** (should also consider the incurred overhead)

```
#include <time.h>

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

struct timespec {

    time_t   tv_sec;        /* seconds */

    long     tv_nsec;       /* nanoseconds */

};
```



Supported Clocks for clock_gettime()

- The **clk_id** argument is the identifier of the particular clock on which to act
- Sufficiently recent versions of GNU libc and the Linux kernel support the following clocks:
  - CLOCK_REALTIME
    System-wide wall-clock clock. Set this clock requiring appropriate privileges
  - CLOCK_MONOTONIC
    Clock that cannot be set and represents monotonic time since some unspecified starting point
    Good for total elapsed time, including I/O & block overhead
  - CLOCK_PROCESS_CPUTIME_ID
    High-resolution per-process timer from the CPU (not for block/sleep code)
  - CLOCK_THREAD_CPUTIME_ID
    Thread-specific CPU-time clock



NATIVE LINUX TIME MEASUREMENT



gettimeofday()

Used for measuring wall clock time

```
#include <sys/time.h>
#include <sys/types.h>
struct timeval tp;
double sec, usec, start, end;
// Time stamp before the computations
gettimeofday( &tp, NULL );
sec = static_cast<double>( tp.tv_sec );
usec = static_cast<double>( tp.tv_usec )/1E6;
start = sec + usec;
// Computations to be measured
...
...
// Time stamp after the computations
gettimeofday( &tp, NULL );
sec = static_cast<double>( tp.tv_sec );
usec = static_cast<double>( tp.tv_usec )/1E6;
end = sec + usec;
// Time calculation (in seconds)
double time = end - start;
```



TIME MEASUREMENT FOR X86 PLATFORMS



Read Time Stamp Counter (RDTSC)

- Time Stamp Counter (TSC)
  - Record CPU clock cycles
  - Monotonically increase for each CPU clock
  - Accessed by the Read Time Stamp Counter (**RDTSC**) instruction introduced in Pentium processors
- Simple and neat to use the Time Stamp Counter for time measurement
  - by calling the inline function below to get the current timestamp (in cycles)
  - Need to translate the cycles into the actual time

```
/* assembly code to read the TSC */
static inline uint64_t RDTSC()
{
  unsigned int hi, lo;
  __asm__ volatile("rdtsc" : "=a" (lo), "=d" (hi));
  return ((uint64_t)hi << 32) | lo;
}
```

Courtesy of http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf
http://stackoverflow.com/questions/3388134/rdtsc-accuracy-across-cpu-cores

Also read: http://oliveryang.net/2015/09/pitfalls-of-TSC-usage/



Why RDTSC?

- Compared with **clock_gettime()**
  - Higher resolution
    - We can get the resolution through the API **clock_getres()**
    - On the Dell XPS 1530 with Intel core2duo T7500 CPU running Ubuntu 10.04, it has a resolution of 1 nanosecond
    - On the other hand, RDTSC instruction can have resolution of up to a CPU clock time
    - On the 2.2 GHz CPU that means resolution is 0.45 nanoseconds
  - Low cost
    - Measure the time taken for 1 million calls to both HPET and RDTSC
    - HPET : 1 s 482 ms 188 us 38 ns
    - RDTSC: 0 s 103 ms 311 us 752 ns
    - RDTSC is 14x faster than HPET

Courtesy of https://aufather.wordpress.com/2010/09/08/high-performance-time-measuremen-in-linux/



Be Careful for the Following Situations!!!

- The per-CPU TSC value may not the same across CPU cores
  - Multiple cores having different TSC values
  - One should bind the process to same CPU
  - Invariant TSC support for ensuring consistent across multiple cores; but, one should pay attention to multi-CPUs
- CPU frequency scaling for power saving
  - Fixed CPU power governor policy, e.g., high performance
- Hibernation of system will reset TSC value
  - Disable the hibernate; or, check the TSC values
- Impact on portability due to varying implementation of CPUs
  - Fixed Intel CPUs with same settings
- Out-of-order execution of code
  - Use the instruction, e.g., CPUID, for serializing instructions
  - Please refer to [the document]



QUESTIONS?