

# CSE221 - System Profiling

Ruchir Garg (rugarg@ucsd.edu), Max Gao (magao@ucsd.edu)

October 2022

## 1 Introduction

Our project benchmarks various system operations and in doing so, attempts to approximate hardware performance at the software level. The system components we benchmark include:

- CPU Scheduling
- Memory
- Network
- File System

Our measurements are performed by code written in C which we compile with gnu C++=11 with various levels of optimization (level 2 by default, otherwise noted). We developed on our local machines and executed the measurements on a bare metal server rented from an infrastructure provider, Vultr. Work was split evenly: Ruchir laid out the initial codebase and contributed measurement code for the CPU and Filesystem; Max contributed code to generate measurement graphs alongside measurement code for memory and network components.

We estimate that each team member worked a total of **X** hours. Estimate the amount of time you spent on this project.

## 2 Machine Description

To minimize variability in our measurement times arising from virtualization between tenants on a shared physical machine, i.e. inter-tenant resource contention, we opted for a single-tenant, bare-metal server. The machine's hardware specifications are as follows:

Table 1: Bare-Metal Server Specs

CPU Model Name	Intel E3-1270
L1 cache size	32768 B ( 32 KB)
L2 cache size	262144 B ( 262 KB)
L3 cache size	8388608 B ( 8.3 MB)
Frequency (per core)	3118.588 Mhz ( 3 GHz)
TLB0 size	32x2MB pages
TLB size	64x4K pages
Instruction Set	x86-64
Data	(???)
RAM	32 GB
IO Bus	(???)
Disk Capacity	240 GB
RPM	N/A
Control Cache Size	N/A
Network Card Speed	10000 Mbits/s
Distribution Name	Ubuntu
Version	22.10
Linux Kernel	5.19.0-26-generic

### 3 Methodology

#### 3.1 Base Hardware Performance Measurements

Our pre-experimental setup further reduces variability arising from intra-machine, inter-process resource contention. We modify the *isolcpus* kernel parameter before booting to ensure that a single process is given exclusive use to a CPU core. Furthermore, we modify these cores’ *IRQ affinity* so that interrupts are handled by non-isolated cores. Measurement processes are then scheduled on isolated cores after setting their *cpu affinity*.

We run each experiment, unless otherwise noted, for 100,000 iterations and calculate average times using the number of clock-cycles and our machine’s CPU frequency (see Table 1).

##### 3.1.1 CPU, Scheduling, and OS Services

**Measurement Overhead** To measure the time taken by a process, we check the clock time before and after the process is complete. To measure the overhead of reading time, we calculate the average time taken to perform a system call over a set of calls and estimate that the overhead to record time would be twice as much as the recorded average.

To measure the overhead time from iterating a for loop, we measure the time taken by a standalone operation (t) and the total time it to complete a loop

(T), for which the number of iterations performed is N. Hence the overhead per iteration due to for loop would be  $(\frac{T}{N} - t)$

**Procedure Call Overhead** To measure procedure call overhead, we schedule a process on our isolated core which executes 7 variations of a function that prints integer arguments. Each variation accepts one additional integer argument than its preceding variation. We check the clock times between function calls to measure times for each variation.

**System Call Overhead** The isolated core will run a program that invokes a list of system calls (once per program execution to avoid cache effects) and output the elapsed times for each, using, again, clock time differences. We compare average system call time measurements against outputs of the command *strace* to ensure that no significant discrepancies exist (though we anticipate *strace* benchmarked times to be slightly higher than what our program reads. We compare these average times of system calls to procedure calls.

**Task Creation Time** On our isolated core, we will execute a C program that performs the following to measure process creation time:

- Outputs the clock time to stdout
- Calls `fork()` to spawn a child process on the same core.
- Child process outputs the clock time to stdout

Another C program will perform the following to measure pthread creation time:

- Outputs the clock time to stdout.
- Calls `pthread_create()` to request a kernel thread.
- Kernel thread outputs the clock time to stdout.

#### Context Switch Time

- Outputs the clock time to stdout
- Calls `popen(subprocess)`
- inside the subprocess output the clock time.
- Switch back to the master process, and output clock time.

The difference between the times is the time to switch context + task creation. We already measured the time for task creation. Hence the context switch time is the mean of the time differences – time to create task.

#### 3.1.2 Memory







**RAM Access Time** We measure a single integer's load time from main memory, L1, and L2 by the following procedure.

**RAM Bandwidth** TO WRITE

**Page Fault Service Time**

## 3.2 Network

For all remote experiments, we run a client on our original measurement machine and a server on a machine with the same specs located within the same city as the original. We were unable to determine if both machines were colocated within the same rack or data center.

<input type="checkbox"/> Server	OS	Location	Charges	Status	
<input type="checkbox"/> <b>Bare Metal</b> 32768 MB Bare Metal - 45.32.161.112		 Miami	\$1.25	 Running	***
<input type="checkbox"/> <b>Bare Metal</b> 32768 MB Bare Metal - 149.28.102.216		 Miami	\$1.25	 Running	***

### 3.2.1 Round trip time

We evaluated the Round-Trip Time (RTT) for both a TCP and ICMP request. For each protocol, we measured the RTT of packets sent to and received over the loopback interface, i.e. the IP protocol stack is used as an interprocess communication mechanism, and packets sent to and received by remote host within Vultr’s network.

Our local ICMP ping implementation (`icmp_loopback_RTT.c`) is a single process that sends and receives an ICMP Echo request packet through the same socket. We calculate the elapsed times between `sendto()` and `recvfrom()` system calls for each iteration. The remote implementation separates the sending and receiving logic into two separate processes across two cores, each belonging to a separate machine.

Our local TCP implementation (`tcp_loopback_client.c` and `tcp_loopback_server.c`) comprises of two processes, each with its own socket bound to a different port on the loopback interface. We calculate the elapsed time between the `write()` and `read()` calls on the client process, discounting the connection setup time. We repeat the procedure for the remote implementation (`tcp_remote_client_RTT.c` and `tcp_remote_server_RTT.c`) except the two processes instead execute on separate cores belonging to separate machines.

### 3.2.2 Peak bandwidth

We measure bandwidth by sending a payload of fixed size, 1.07GB ( $2^{30}$  bytes). We test this over 8 different TCP payload lengths: 512, 1024, 2048, 4096, 8192, 16384, 32768, and 64000 (the maximum TCP payload length assuming a maximum MTU of 65536). For each length, we send the total payload partitioned by the payload length and measure the number of cycles it took to transmit the payload (`tcp_remote_client_bw.c`). For simplicity sake, we calculate bandwidth over the entire window of time, though it’s likely that there are fluctuations at smaller time intervals.

### 3.2.3 Connection overhead

The connection overhead.....

## 4 File System

### 4.1 Size of file cache

Note that the file cache size is determined by the OS and will be sensitive to other load on the machine; for an application accessing lots of file system data, an OS will use a notable fraction of main memory (GBs) for the file system cache. Report results as a graph whose x-axis is the size of the file being accessed and the y-axis is the average read I/O time. Do not use a system call or utility program to determine this metric except to sanity check.

### 4.2 File read time

Report for both sequential and random access as a function of file size. Discuss the sense in which your "sequential" access might not be sequential. Ensure that you are not measuring cached data (e.g., use the raw device interface). Report as a graph with a log/log plot with the x-axis the size of the file and y-axis the average per-block time.

### 4.3 Remote file read time

Repeat the previous experiment for a remote file system. What is the "network penalty" of accessing files over the network? You can either configure your second machine to provide remote file access, or you can perform the experiment on a department machine (e.g., APE lab). On these machines your home directory is mounted over NFS, so accessing a file under your home directory will be a remote file access (although, again, keep in mind file caching effects).

### 4.4 Contention

Report the average time to read one file system block of data as a function of the number of processes simultaneously performing the same operation on different files on the same disk (and not in the file buffer cache).

## 5 Results and Discussion

### 5.1 CPU, Scheduling, and OS Services Estimates and Results

### 5.2 Memory Estimates and Results

Please see attached pdfs.

### 5.3 Network Estimates and Results

Though our bandwidth measurements in the remote case approaches the maximum speed reported for the network card on our machine (1000Mbit/s or

Table 2: CPU, Scheduling, and OS Services Estimates and Measurements 1

Operation	Base Hard- ware Perf.	Est. ware head	Soft- ware Over- head	Pred. Time	Measured Time (Avg ms)
Task Creation (pro- cesses)					55.318
Task Creation (kthreads)					7.833
Context Switch (pro- cesses)					1.384
Context Switch (Kthreads)					1.380

Table 3: CPU, Scheduling, and OS Services Estimates and Measurements 2

Operation	Base Hard- ware Perf.	Est. ware head	Soft- ware Over- head	Pred. Time	Measured Time (Avg ns)
Loop Overhead					1328
Reading Time Over- head					7514
Procedure Call Over- head					
System Call Overhead					48

1.25GB/s) as we increase the payload size, a 64K payload only yields 76% of the theoretical hardware maximum. We did not have time to investigate the cause for this. If we were to estimate the bandwidth, we'd establish a lower bound on the number of syscalls for the various phases of TCP and express bandwidth as a function of this fixed cost lower bound and network latency.

## 5.4 Filesystem Estimates and Results

Please see attached pdfs for results.

Table 4: Network Measurements (\* denotes sending to loopback interface)

Operation	Base Hard-ware Perf.	Est. ware Over-head	Pred. Time	Measured Time
RTT* (ICMP)				3.9ms
RTT (ICMP)				N/A
RTT* (TCP)				14.9ms
RTT (TCP)				78.3ms
Conn. Overhead* (TCP)				43.7ms
Conn. Overhead (TCP)				67.2

Table 5: Network Bandwidth Measurements (\* denotes sending to loopback interface)

Operation	Base Hard-ware Perf.	Est. ware Over-head	Pred. Time	Measured Bandwidth (MB/s)
Peak Bandwidth* (TCP)	N/A	N/A	N/A	2253.56
Peak Bandwidth (TCP)	N/A	N/A	N/A	962.22

