Systems Measurement Project - Final Report

Eric Hu, Sarat Sreepathy, Chitra Kharbanda

March 10, 2022

1 Introduction

We performed our project in a school-hosted virtual machine to research the specifications and capabilities of their environment. We performed our measurements in C and used the GCC compiler from the GNU Compiler Collection, using version 7.5.0. For our compiling, we used the $-O$ optimization level to improve the performance of our code, at the cost of a slower compiling time.

Using a school-hosted virtual machine, however, left us with some implications. The host is out of our control and we are unable to manage the status of the virtual machine since it was hosted online. This may greatly affect our experiments for networking. There is also the issue of the network response time between our machines and the virtual machine. While we believe that this will not have any affects on our experiments, as they are built into code programs, it is still worth mentioning. Additionally, as it is a virtual machine, it is missing some machine specifications or the specifications were not very detailed.

- Eric: Task creation and run for measurement, procedure call, and system call overhead, RAM bandwidth for writing and reading, and File read time and remote file read time
- Sarat: Task creation and run for both processes and threads, Access Latency for Memory, L1, and L2 cache, File Cache Size, and Contention
- Chitra: Task creation and run for context switching for both processes and threads, Page fault service time, All network operations

Estimated time spent on project: 75 hours

2 Machine Description

Below are the listed virtual machine specifications. As the virtual machine environment is limited, we were unable to find detailed descriptions of some specifications.

Operating System: Ubuntu 18.04.6 LTS

Processor

- Processor model: Intel ® Xeon ® CPU E5-2690 v2 @ 3.00GHz
- Cycle time: 3000 MHz (0.33 ns)
- Cache size: 25600KB

Memory

- DRAM: Unknown
- Type: DDR3-1866
- Clock: 30 (clock multiplier)
- ECapacity: 2033016KB

- Memory Bus Bandwidth (Speed): 8 GT/s → 985 MB/s

I/O Bus Type: Unknown

Disk

- Hard Drive:
  - Model: Xen Virtual Disk A
  - Capacity: 150G
  - RPM: Unknown
  - Transfer rates: Unknown
  - Latencies: Unknown

Network

- Network Card Bandwidth: Unknown
- Network interface: eth0

3 Experiments & Results

3.1 CPU, Scheduling, and OS Services

For our experiments in CPU, Scheduling, and OS Services, we measured time by counting the number of cycles. This was done by creating 2 instances of *rdtsc*, where one instance gets the cycles before the experiment and the other instance gets the cycles right after the experiment. By subtracting the two cycle values, we can get the number of cycles that occured between the two instances. We ran each experiment for 10 trials. Within each trial, we ran the process (procedure call, system call, context switch, etc.) for 100,000 iterations. We computed the average overhead for each trial, and then got the final average overhead and standard deviation for all of the trials.

3.1.1 Measurement Overhead

We first considered the measurement of the measurement itself. To do this, we created 4 instances of *rdtsc*: one for the start of the outer measurement, one for the start of the inner measurement, one for the end of the inner measurement, and one for the end of the outer measurement. With this, we are able to use the following formula to solve how many cycles it took to start the measurement.

$$MeasurementOverhead = endOuter - startOuter - (endInner - startInner)$$

This formula gets the number of cycles it took for the outer measurement to finish, subtracting from it the number of cycles the inner measurement took. This way we would end up with the number of cycles between *startOuter* and *startInner* and the cycles between *endInner* and *endOuter*, which results in the final measurement overhead.

For the inner measurement, we decided to measure both the measurement of procedure calls and system calls so that we could compare the two if there were any notable differences.

3.1.2 Measurement Overhead Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Measurement Overhead (Process)	N/A	10 μs	10 μs	23 μs	0 μs
Measurement Overhead (System)	N/A	10 μs	10 μs	25 μs	1 μs

Table 1: Measurement Overhead Results

For our estimated values of hardware and software overhead, software overhead is the only one applicable for this measurement. We estimated it to be around 10 microseconds, as we believe this operation will not take very long.

When measuring the measurement overhead, we saw that when it was measuring the measurement of a procedure call it had an average of 23 microseconds whereas on the measurement of a system call it had an average of 25 microseconds. Although it was not a big difference, the system call measurement always had a higher overhead when compared to the process call measurement overhead. The standard deviation for both of these experiments were near 0, so there was little to no variability.

We believe that our measurement is somewhat lacking in accuracy, as we expected that the measurement overhead to be the same for all the measurements we did. It was strange to us that the system call measurement had a consistently greater measurement overhead than the process call measurement overhead.

3.1.3 Process Call Overhead

Our next experiment was conducted on measuring the time it took to complete a process (function) call. We performed this on processes with different number arguments passed to it, ranging from 0 to 7 arguments, so that we could see the effects of passing arguments on cycle time. The multiple processes were named

foo0(), foo1(int a), foo2(int a, int b), foo3(int a, int b, int c), ..., foo7(int a, int b, int c, int d, int e, int f, int g)

and were fairly simple, as none of them performed any operations and immediately returned 0 when called. For the measurements, we created 2 instances of *rdtsc*: one for the right before the process call, and one for right after the process call. The final measurement was then calculated by subtracting the cycles measured right after the process call by the cycles measured before the process call.

3.1.4 Process Call Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Process Call(0 Arguments)	N/A	10 μs	10 μs	11 μs	$\approx$ 0 μs
Process Call(1 Arguments)	N/A	11 μs	11 μs	5 μs	$\approx$ 0 μs
Process Call(2 Arguments)	N/A	12 μs	12 μs	5 μs	$\approx$ 0 μs
Process Call(3 Arguments)	N/A	13 μs	13 μs	6 μs	$\approx$ 0 μs
Process Call(4 Arguments)	N/A	14 μs	14 μs	7 μs	$\approx$ 0 μs
Process Call(5 Arguments)	N/A	15 μs	15 μs	6 μs	$\approx$ 0 μs
Process Call(6 Arguments)	N/A	16 μs	16 μs	7 μs	$\approx$ 0 μs
Process Call(7 Arguments)	N/A	17 μs	17 μs	8 μs	$\approx$ 0 μs

Table 2: Process Call Results

For our estimated values of hardware and software overhead, software overhead is the only one applicable for this measurement. We estimated for a baseline process with no arguments to be around 10 microseconds, and for each additional argument it would take an extra 1 microseconds.

When running the process call measurements, we expected for the average number of microseconds to increase for each additional argument passed, as it would be another resource that the process would have to transfer. However, we were surprised to see that what we thought would be the fastest process, the process with 0 arguments passed, ended having the largest overhead of 11 microseconds. All the other process calls of 1 through 6 arguments followed the expected outcome of having increasingly more overhead for each additional argument. From these results, we can discern the increment overhead of an argument to be about 1 microsecond. The standard deviation for all of these experiments were near 0, so there was little to no variability.

We believe for our implementation to be correct, but the accuracy of the result of the process call with 0 arguments was surprising. We tested this multiple times, but the initial process call with 0 arguments always resulted with the highest overhead. These results lead us to question the stability of the virtual machine itself, as it gave us contradictory results than expected.

3.1.5 System Call Overhead

We performed experiments on a normal process call, so now we wanted to experiment on a system kernel call. We measured the system process *getpid()*, which accesses the kernel to return the current process's process ID and *getsid()*, which accesses the kernel to return the current session ID. This was so we could compare the overhead for the two different system calls. For the measurements, we created 2 instances of *rdtsc*: one for the right before the system call, and one for right after the system call. The final measurement was then calculated by subtracting the cycles measured right after the system call by the cycles measured before the system call.

3.1.6 System Call Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
System Call (getpid())	N/A	100 μs	100 μs	5474 μs	$\approx$ 0 μs
System Call (getsid())	N/A	100 μs	100 μs	5558 μs	$\approx$ 0 μs

Table 3: System Call Results

For our estimated values of hardware and software overhead, software overhead is the only one applicable for this measurement. We estimated for a baseline system call to be around 100 microseconds. We expect a system call to take a much longer time, compared to a process call which we estimated to be around 10 microseconds.

When we ran the system call measurements, we got an average value of 5474 microseconds for the *getpid()* system call and an average value of 5558 microseconds for the *getsid()* system call. Both these values were much larger than our estimated value of 100 microseconds as we underestimated how much longer a system call would take. Comparing to process calls, the systems calls took much longer, which was expected. Additionally, the two system calls we experimented on had similar overhead times. The standard deviation for both of these experiments were near 0, so there was little to no variability.

We believe our results to be accurate, as we experimented on two different simple system calls and got values close to each other. With a standard deviation of around 0, we believe that our resulting average values are accurate.

3.1.7 Task Creation

One of the CPU operations we want to test on this virtual machine is task creation time. We want to measure how long it takes to create and run a process as well as a thread. For process creation, we want to create and run a single process and measure the time it takes in microseconds. For thread creation, we want to create and run a kernel threads and measure time in microseconds. With Linux, we want to create a kernel-managed thread using (p_thread create). For process creation, we want a single child process so we called *fork()* and exited if the pid == 0 indicated a child and printed "Run process." We waited the parent process until child finished executing so we do not have the child process calling *fork()* and creating even more processes. We used *rdtsc* to track the number of cycles run before and after the child process had been created and executed. We ran this over several iterations and collected the number of cycles that were run each iteration and converted cycles to microseconds. We did this conversion by using *rdtsc* before and after calling sleep for 1 second to measure how many cycles passed in that second and we would use that number to convert the measurement in cycles to microseconds. We then calculated the average process creation and run time as well as the standard deviation.

For threads, we wanted a single thread to be created and run so we used p_thread in Linux. We initialized the p_thread variable and we also called p_thread create and join which created the kernel thread and waited for finished execution, respectively. Similarly to process creation and run, we have a print statement "Run kernel thread" which specifies that the thread has executed. We also used *rdtsc* to track the time before and after thread has executed using similar method for converting cycles to microseconds. We also produce the average time for thread creation and run time as well as standard deviation.

3.1.8 Task Creation Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Process Creation/Run	0.0009 MOpsμs $\approx$ .. μs	400 μs	500 μs	991 μs	0 μs
Thread Creation/Run	0.0018 MOps/μs $\approx$ .. μs	100 μs	150 μs	135 μs	69 μs
CyclesPerSecond	..	..	..	$3*10^9 cycles$	$\approx$ 0 cycles

Table 4: Task Creation Results

When running process creation and execution, we saw that it took an average of 991 microseconds with a standard deviation of 0 microseconds which indicated significant variability within measurements. Processes ran pretty quickly when using this environment.

When running kernel thread creation and execution, we saw that it took an average of 135 microseconds with a standard deviation of 69 microseconds which showed some variability between measurements. The kernel thread was much faster than the creation and execution of the child process which is what we expect since threads are generally supposed to be faster than processes which is why multiple threads are created for execution rather than multiple processes.

Since process creation is a more time intensive operation than thread creation, we predicted that software overhead for process creation/run would be around 400 microseconds while thread creation/run would be around 100 microseconds and we expect software overhead for thread to be faster than process given threads are supposed to be faster. Predicted performance for process creation/execution was 500 microseconds shown in the table. Predicted performance for thread creation was in the 150 microsecond range which is also reflected in the table. We added a 50-100 microseconds to reflect hardware performance when calculating predicted performance, but it was difficult to convert machine operations to a wall clock time. Again, we expected predicted performance for threads to be better than processes. For hardware performance, we got 0.0018 Machine Operation Per Microsecond for a single thread. For hardware performance we got 0.0018/2 = 0.0009 Machine Operations Per Microsecond for a single process using the fact that there are 10 cores and 20 threads so we expect threads to be twice as fast.

When comparing predicted versus actual performance, we see that it was pretty accurate when it came to predict thread creation/run time as it was 150 microseconds predicted and 135 microseconds on average in measurement. Comparing predicted and actual performance for process creation/run time was a bigger difference where we predicted 500 microseconds but it was more like 991 microseconds on average showing a greater discrepancy and could be due to an issue when predicting software overhead or hardware performance.

Hardware Reference: https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+E5-2690+v2+%40+3.00GHz&id=2057

3.1.9 Context Switch

We performed experiments on measuring the overheads of the CPU with respect to context switches among threads and processes. The general steps involved while context switching between processes in a operating system is that the state of the currently running process is saved and then again restore when another process is done executing.

For measuring the context switch time, we make use of the pipes which are inherently blocking in nature. The general property of these pipes are that whenever a process writes a data onto these pipes, they get blocked until another process has read from it. This facilitates us to measure the amount of time the CPU used to context switch from one process to another. The methodology implemented to measure the context switch time is basically starting by creating a pipe. Then we fork a child process and allocate the read descriptor of the pipe to the child process. Meanwhile, the parent process gets the write descriptor of the pipe. As the data gets written on the pipe, a process gets blocked till the data has been read from the child process. This makes the CPU to switch the context from parent to child. We measure the duration by recording the time at the end of the writing process and the start of the reading process. The difference between the two gave us the value of the context switch between the processes. We have used the pipe to transfer a array of 500 elements of integer data, which amount to 2000 bytes of data.

For measuring the context switch between the threads, we have used a similar algorithm. We record the time at the end of the writing of data and at the start of the reading of data at the other thread. We subtract the two which gives us the context switch time between the threads.

3.1.10 Context Switch time between process and threads

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Context Switch (Process)	N/A	400 μs	400 μs	16.826 μs	0.637 μs
Context Switch (Thread)	N/A	10 μs	10 μs	16.107 μs	0.403 μs

Table 5: Context Switch Results

We observed for the context switch time between process took longer as when compared to the context switch time between kernel thread. This is expected because thread switching is very efficient and cheaper than the process switch. Process switching involves the switching of all the process resources whereas thread switching all of these steps are not present because we are switching within a process itself.

We measured the average context switch time between processes for 10 trials and in each trial we did 1000000 iterations and it came out to be 16.826 microseconds with a standard deviation of 0.637 microseconds. We measured the average context switch time between kernel threads for 100 trials and each trial had about 10000 iterations and the performance for the same was faster. The average for the same was observed to be 16.107 microseconds with a standard deviation of 0.403 microseconds.

For our software overhead estimation, we estimated that the steps involved for context switching in process would be mostly the system call to save the current process state into the CPU's stack, then switch the registers, increase the program counter, and then loading the page table of the next process to run. We estimated the baseline system call to be of 100 microseconds, the total time for the system call involved in context switch came around to be 400 microseconds. For the context switch in threads, the steps involved are those of saving and restoring the minimal context. Since the context switch in threads does not involve much of system call we estimated the software overhead to be around 10 microseconds.

We believe our implementation for both the context switch between thread and process to be accurate. The accuracy for the thread context switch seems to be okay. The process context switch is taking a bit longer than the thread context switch which is the expected result. However, there might be a chance that the results are a skewed a bit because of the interference from other processes which are executing. This is a case which we do not have much control over.

3.2 Memory

3.2.1 Access Time

We performed experiments to find the latency for accessing into main memory (2 GB) and the L1 (32 KB) and L2 caches (256 KB). The goal was to demonstrate the difference in speed of accessing from cache as supposed to memory which is much slower. For measuring the access latency of main memory, we used *malloc()* to create an array of size 1800000 bytes which is larger than the L2 cache (256 KB) and wanted to access the array at strides of 16 (access every 16th element (byte)) to see what the average access latency to main memory. For L1 cache, we do not want to reallocate memory so we do not use *malloc()* and instead access the array we created when doing the main memory measurement. We expect the accesses we did for the main memory measurement to be stored in L1 cache so accessing them would be faster. We accessed the array with strides of 16 and found average access latency for L1 cache. We access a region of the array of around 20000 bytes which is less than the L1 cache meaning we would be likely to hit the L1 cache. For L2 cache, we are going to have to make sure the access is not going to L1 cache. We therefore access the previous array allocated for memory access measurement and access a region of array of around 80000 bytes which is more than the L1 cache (32 KB) meaning we would likely hit the L2 cache. We also randomize the access of different elements instead of sequential access so we avoid prefetching from the L1 cache and we would be accessing L2 cache. We access the array in strides of 16 and find the average latency of L2 cache.

For the measurements for latency, we measured them in cycles using *rdtsc* and converted them to nanoseconds so the average access latency is measured in nanoseconds. Nanoseconds were the

appropriate unit since these accesses are expected to be quite fast especially when we refer to the L1 and L2 caches. We did not report standard deviation in the results because we are relying on caching so running iterations would affect what we would expect in L1 or L2 cache and stuck primarily to an average of access latency.

3.2.2 Access Time Results

Operation	Hardware	Software Overhead	Predicted	Avg
Main Memory Access	80 ns	20 ns	100 ns	81 ns
L1 Cache Access	1 ns	4 ns	5 ns	2 ns
L2 Cache Access	4 ns	10 ns	14 ns	12 ns

Table 6: Access Time Results

CyclesPerSecond: 3000MHz(1 second) Reference for Expected Performance: https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html

Analysis: We saw that the performance for main memory was much slower than L1 and L2 cache which lines up with the expectation that main memory is slower than L1 and L2 cache. We saw that L1 cache latency was around 2 ns which is around what we expected at 5 ns with 1 ns of estimated hardware performance and 4 ns of software overhead. We saw with L2 cache latency an average of 12 ns similar to the predicted latency at 14 ns with 4 ns of estimated hardware performance and 10 ns of software overhead. For main memory access, we had 81 ns on average with a predicted latency of 100 ns with estimated hardware performance of 80 ns and software overhead at 20 ns. The actual and predicted performance for main memory access was similar which also held true for L1 and L2 access latency with similar actual and predicted performances

3.2.3 Graph of Memory Regions and Latencies

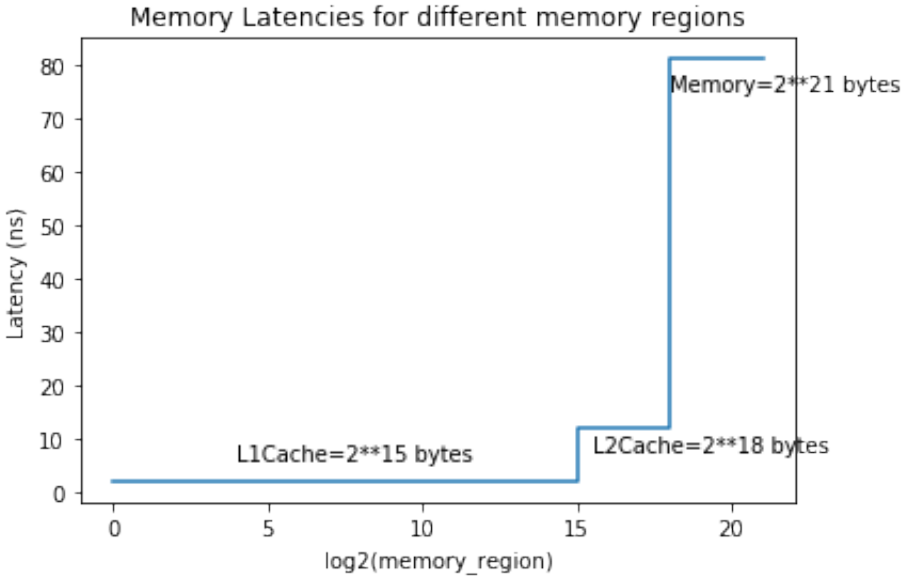

Figure 1: Memory Latencies Graph

3.2.4 RAM Bandwidth

We performed experiments on finding the bandwidth for both writing and reading from memory. We used *malloc()* to create an array of size 10000000, which is equivalent to 10 MB of memory. We used *rdtsc* to measure the number of cycles it took to write and read from the 10 MB of memory. While we are initially measuring in bytes and number of cycles, we convert it to MB/s for our final results.

To achieve more accurate results, we used loop unrolling when writing and reading from memory to limit the effect of any loop operation overhead. In each loop, we wrote 20 bytes to memory and kept running this loop until we have read the entire file.

3.2.5 RAM Bandwidth Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Writing Bandwidth	985 MB/s	500 MB/s	485 MB/s	377 MB/s	0 MB/s
Reading Bandwidth	985 MB/s	100 MB/s	885 MB/s	571 MB/s	0 MB/s

Table 7: RAM Bandwidth Results

For our predicted values, we estimated that the hardware performance would be 985 MB/s, which is the same as the machine's 985 MB/s bus bandwidth. Software overhead would be present, as we are writing and reading from memory, where we estimated that writing would have a greater overhead compared to reading. We gave writing to memory a software overhead of 500 MB/s and to reading a smaller overhead of 100 MB/s. As such, our final predicted values for writing is 500 MB/s and for reading is 800 MB/s.

We saw as expected that writing performed much slower, at only 377 MB/s, but were surprised to see that reading had performed below our expectations at 571 MB/s. While we overestimated the performance of writing and reading to memory, we believe our experiment to be successful as our results are relatively near our virtual machine's memory bus bandwidth description of 985 MB/s when also considering software overhead. It was also expected that writing to memory would be slower than reading from memory.

3.2.6 Page Fault Service Time

We performed experiments on finding the page fault service time. The total main memory of the system we are working on is 2033016 kB. So the general idea to ensure that page fault occurs was to randomly access a file whose size was bigger than the capacity of main memory so that the entire file could not fit into the memory hence forcing the occurrence of page fault.

As the first step in the experiment, we created a file whose size was larger than the main memory capacity. We created a text file whose size was 2500000 kB. Now after creating a file, the residues of the file content can still be left in the main memory and the cache. To prevent our test from accessing this data, we clean the cache before accessing the file. We make use of "$echo 3 > /proc/sys/vm/drop\_caches$" system call to clear out the cache. Now for accessing the file, we make use of the $mmap()$ system call which creates a new mapping in the virtual address space of the calling process. Then we randomly access the pages (the continuous memory allocated of size 4096 B) of the mapped file for 1000 time. For every access of a page, we measure cpu clock cycle and store the result.

3.2.7 Page Fault Service Time Results

We performed the experiments where we accessed the page and timed every access. We plotted the graph for the access time of each of the page for a 1000 iterations. Attached below is the graph depicting the time taken to access a page (4096 B of data) in milliseconds for every iteration.

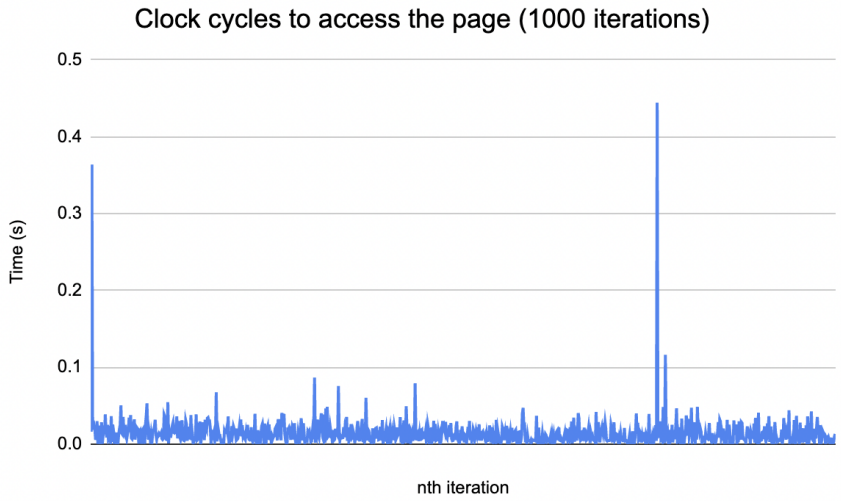

Figure 2: Clock Cycles v. Iterations

Operation	Hardware	Software Overhead	Predicted	Avg
Page Fault Service Time	N/A	100 ms	200 ms	404 ms

Table 8: Page Fault Service Time Results

The graph shows the three peaks which essentially denotes that the page fault occurred. The average time for servicing the page fault came out to be 404.04 milliseconds. Dividing this value by the size of a page gives us the 98.64 microseconds/byte. This is way more than the time of accessing one byte from the main memory which was expected.

The system which we are measuring the performance for is the ubuntu running on virtual machine. We were not able to find the base hardware metrics (the latency for the disk) for the same. For estimating the software overhead, the main contributors to this would essentially be the overhead involving the system calls of accessing the page table, then the trap to os when the requested page is not found, then for finding the page in the backing store, then for bringing the page to the physical memory and then updating the page table. For all these steps, we assume the software overhead to be around 100 milliseconds. Since the transfer of data between the disk to the memory and then while updating the page table also takes time, we predicted the page fault service time to be around 200 milliseconds.

We believe our experiments to be successful. The peaks obtained in the graph are clearly depicting the access time for when the page fault occurred. Since it is expected for the page fault to occur at the start of the experiment (the reason being that we flush out the cache before actually accessing the file) the graph is showing the similar expected behaviour. The final result that accessing one byte of data takes very long time during the page fault versus when directly accessing it from the main memory also makes sense.

3.3 Network

3.3.1 Round Trip Time

We performed experiments to determine the round trip time for the network and compare it with the ping latency for the loop back IP address as well as remote IP address. For the loop back IP address, the IP 127.0.0.1 was used and for the remote network, we used an AWS virtual machine to host our server (Machine description : section 3.3.7).

For measuring the application level round trip time, we created a client and server which would connect with each other over tcp socket. Since the packet used in ping measurement is 56 MB, so in our experiment, the client sent 56 MB of data to the server listening. The server received the data from the client and sent it back to the client. We measure the start time in the client code before sending the data and the end time after receiving the data from the server. The difference between these gave

the round trip time for the network. We did the experiments for 10 iterations and took an average of the results. We expect the ping latency to be faster than the application level latency because the ping requests are handled at the network device of the system and also because ping communicates via ICMP packets which consist of just first part of the TCP 3 way handshake. For the application level connections, we are using TCP to connect the client to the server over socket.

3.3.2 Round Trip Time Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Ping (Loopback)	N/A	N/A	N/A	0.015 ms	0.003 ms
Round Trip Time (Loopback)	N/A	0.8 ms	1 ms	0.0878 ms	0.0063 ms
Ping (Remote)	N/A	N/A	N/A	13.726 ms	0.171 ms
Round Trip Time (Remote)	N/A	0.8 ms	15 ms	13.43 ms	0.186 ms

Table 9: Round Trip Time Results

The round trip time for loopback address was observed to be 0.0878 milliseconds as when compared to ping which was 0.015 milliseconds. Ping taking lesser round trip time than the application level connection is the expected result because the ping commands consist of ICMP packets which are handled at the network interface of the system whereas the application level connection packets have to move all the way from the kernel to the network interface and then again to the kernel to be handled correctly. For the remote case, the round trip time came to be 13.43 milliseconds for the application level request and 13.726 milliseconds for the ping command. Here we see that the ping is slower than the application level latency observed. The reason behind this observation is that most networks deprioritise ICMP packets which the ping uses and delivers it slower than the TCP packets. The TCP in system in general is also optimized to give it better performance than the ICMP packets. So, this difference in the result makes sense.

3.3.3 Peak Bandwidth

We performed experiments to determine the peak bandwidth for the network for the loop back IP address as well as remote IP address. For the loop back IP address, the IP 127.0.0.1 was used and for the remote network, we used an AWS virtual machine to host our server (Machine description : section 3.3.7).

For measuring the peak bandwidth for TCP, we create two processes, a reader and a writer, which connects over tcp socket and transfer 1MB data. We also increase the size of the socket buffers to 1M to produce higher throughput. We measure the start and end time in the reader process to make sure we measure duration for the complete data transfer. The result we get is the duration of transferring 1M of data and using this information we calculate the amount of data which can be transferred in one second, which essentially gives the bandwidth of the connection.

3.3.4 Peak Bandwidth Results

Operation	Hardware	Software Overhead	Predicted	Avg
Peak Bandwidth (Loopback)	N/A	500 MB/s	485 MB/s	579 MB/s
Peak Bandwidth (Remote)	N/A	500 MB/s	10 MB/s	1.5 KB/s

Table 10: Peak Bandwidth Results

We observed the average peak bandwidth for the loopback address as 579 MB/s and for the remote, the average peak bandwidth for the remote address was observed to be 1.5 Kb/s. For the loopback address, the bandwidth was almost comparable to the memory bandwidth for reading. We suspect that the since the server and the client are passing data on the same machine, there might be some involvement of cache which is increasing the average bandwidth. For the remote case, the average peak

bandwidth is reflective of the network over which the server and the client communicates. The values are suggestive that the network performance is not good. Since the size of packets is set to 1Mb, so the network overhead is more as when compared to the ping command or the round trip time measure for the remote connection. We are using a ubuntu vm and we did not have the hardware specification for the bandwidth so we are unable to compare our results with the hardware specification of the system used.

While estimating the predicted value for the peak bandwidth of the remote connection, we considered the network performance would hamper the result and so our predicted value is very low.

While doing the experiments we observed that when we performed the experiments for multiple iterations, the average peak bandwidth also changed. We suspect this behaviour due to some mismatch between the time of server and client's connection. As a result, we did the experiment for one iteration multiple times since it represented the base measurement and took an average of it.

3.3.5 Connection Overhead

We performed experiments to determine the connection overhead for the network for the loop back IP address as well as remote IP address. For the loop back IP address, the IP 127.0.0.1 was used and for the remote network, we used an AWS virtual machine to host our server (Machine description : section 3.3.7).

The TCP connection consists of a three way handshake where the client sends SYN to the server, then the server replies with ACK/SYN and then the client finally sends an ACK and the connection is made. Opening a connection consists of these steps and measuring the duration of it gives us the opening connection overhead for the network. Similarly, closing TCP connection consists of a four way handshake. The client sends FIN to server which responds by sending ACK. Then the server sends FIN to client which sends back ACK. The 'connect' and the 'close' functions essentially performs these steps. Measuring the duration of these functions gives us the connection overhead of the network.

3.3.6 Connection Overhead Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Connection Overhead - open (Loopback)	N/A	0.8 ms	1 ms	0.0706 ms	0.00897 ms
Connection Overhead - close (Loopback)	N/A	0.8 ms	1 ms	0.0165 ms	0.0011 ms
Connection Overhead - open (Remote)	N/A	0.8 ms	15 ms	13.33 ms	1.09 ms
Connection Overhead - close (Remote)	N/A	0.8 ms	15 ms	0.0142 ms	0.0069 ms

Table 11: Connection Overhead Results

For the loopback address, the setup overhead was observed to be 0.0706 milliseconds and the tear down overhead was observed to be 0.0165 milliseconds. For the remote address, the setup overhead was observed to be 13.33 milliseconds and the tear down overhead was observed to be 0.0142 milliseconds. We can see that the overhead for opening the connection is almost comparable to the latency measurement we observed. This is expected since TCP is a connection oriented protocol and opening a TCP connection would involve the exchange of packets that accounts for the time observed. The tear down overhead for the remote connection seems to be very low. The network overhead is also not seen in the result. We are not very sure about what might be causing this behaviour.

While doing the experiments we observed that when we performed the experiments for multiple iterations, the average connection overhead for the loopback connection also changed. We suspect this behaviour due to some mismatch between the time of server and client's connection. As a result, we did the experiment for one iteration multiple times since it represented the base measurement and took an average of it.

3.3.7 Remote system description

For the remote system, we deployed instance of AWS EC2 virtual machine. The machine description for this is as follows:

Operating System: Amazon Linux 2

Processor

- Processor model: Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
- Cycle time: 2400 MHz
- Cache size: 30720 KB

Memory

- DRAM: Unknown
- Type: DIMM RAM
- Clock: Unknown
- ECapacity: 965 MB
- Memory Bus Bandwidth (Speed): Unknown

I/O Bus Type: Unknown

Disk

- Hard Drive:
  - Model: /dev/xvda
  - Capacity: 8 Gib
  - RPM: Unknown
  - Transfer rates: Unknown
  - Latencies: Unknown

Network

- Network Card Bandwidth: Unknown
- Network interface: eth0

3.4 File System

We will perform experiment to determine size of file cache, read I/O for local and remote reads, as well as contention (of processes and their I/Os). For these experiments, we were limited in information about the file system in our virtual machine which made it difficult to estimate hardware performance.

3.4.1 Size of File Cache

We performed experiments to determine the size of the file cache. In these experiments, we wanted to measure the read I/O time for different files of different sizes and see how read time changed as the file size had increased. When accessing the file cache, we expect the read time to be much faster as reading from cache is much faster than reading from disk. Therefore, when the file read time increases significantly, it is an indication that we are now accessing the disk instead of the file cache and the file size where this increase occurs can be seen as the file cache size. We want to run different tests reading from different file sizes to find the file cache size on our machine. We would read a single 4KB block from that file and measure the time it took in cycles and convert it to microseconds. Cycle measurement would be done using rdtsc(). We started with 128 MB files all the way to a 32 GB file to see the average read I/O time for reading each file. We did this over 10 trials to find an average read I/O for each of the file sizes and we chose 10 trials because reading these files can be costly as they get larger. We expect that read I/O time would move relatively little for smaller files since they would be

accessing the file cache, but for larger files we would expect a significant increase in read time as you would access the disk. The disk operation is slower because you would have to do the seek operation which slows it down significantly while accessing memory is much faster and has been optimized. We want to plot our results with the x axis representing the size of the file we are reading and the y axis representing the read I/O time for that file.

3.4.2 File Cache Size Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Read I/O for 128 MB	N/A	2 μs	4 μs	6.8 μs	0 μs
Read I/O for 256 MB	N/A	2 μs	4 μs	5.4 μs	0 μs
Read I/O for 512 MB	N/A	2 μs	4 μs	6.1 μs	0 μs
Read I/O for 1 GB	N/A	2 μs	4 μs	6.7 μs	0 μs
Read I/O for 2 GB	N/A	5 μs	40 μs	24 μs	0 μs
Read I/O for 8 GB	N/A	0.5 ms	1.5 ms	2 ms	0 ms
Read I/O for 16 GB	N/A	0.5 ms	3 ms	4.9 ms	0 ms
Read I/O for 32 GB	N/A	0.5 ms	3.5 ms	5.5 ms	0 ms

Table 12: File Cache Size Results

3.4.3 File Cache Size Results Graph

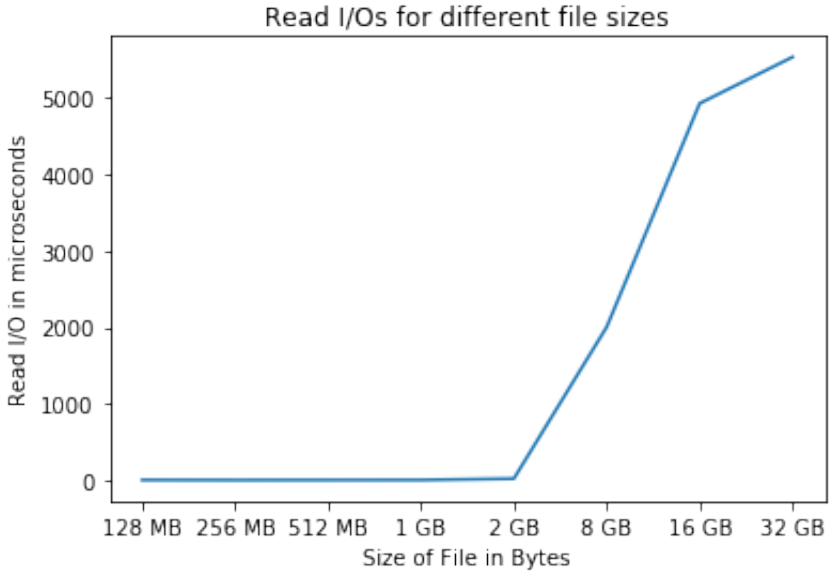

Figure 3: Read I/O v. File Size

For the file cache size results, we saw that from files of size 128 MB to 1 GB, the average read I/O stayed roughly the same at around 6 microseconds which is quite fast indicating that the cache is used for reading those file blocks. For the file of size 2 GB, the average I/O went to 24 microseconds which was slower than the smaller files but not a significant increase to indicate that the reads are not using cache. However, when moving to a 8 GB file, we could see a significant increase to 2 milliseconds (2000 microseconds) indicating that the reads no longer use the file cache and is instead is reading from disk which is much slower. The average read I/Os for 16 and 32 GB were 4.9 and 5.5 milliseconds which also shows that cache is no longer being used. For the file cache size, we conclude that it is a little more than 2 GB because after 2 GB we saw a significant dip in the performance of read for an 8 GB file. This fits with our expectation as the memory in our virtual machine is around 2 GB so any file greater than 2 GB could not be stored in memory and cached. We expect the read I/O to move from a few microseconds to a few milliseconds meaning a (1000x) increase in time. We see that the results of our experiment meet this expectation. We can also see this in the graph where there is a significant

jump in the average read I/O time after 2 GB indicating the limit of the file cache. Access is done with the disk which is much slower. Results shown in graph were roughly similar to expected performance as larger files would have much slower read I/Os.

3.4.4 File Read Time

We performed experiments to measure the time that it took to read a block of data of varying file sizes using sequential access and random access. We first created file sizes to measure of $2^8, 2^9, 2^{10}, \ldots, 2^{28}, 2^{29}$ bytes. We could not measure above 2^{29} bytes due to Github file size limitations. When measuring each of our files, we wanted to make sure that we were not measuring cached data, so we cleared the cache every time before we read the data using the system call "$echo 3 > /proc/sys/vm/drop\_caches$".

For our sequential access experiments, we performed 10 trials on each file. Within each trial, we performed $fileSize/blockSize$ iterations, rounded up, where we measured the time it took to read 1 block of data. We used the *read()* method to read a block and measured it using *rdtsc*. Since we were doing sequential access, we did not have to manipulate the file pointer as after we read a block, the pointer was already pointing to the start of the next block for the next iteration.

For random access, the methodology was the same as it was in sequential access with the addition of manipulating the position of the file pointer before each read. We used the *lseek()* method to randomly move the pointer to point at a block at the beginning of each iteration, so that we could read a random block for each iteration.

For our predictions, we predict that the average block read time will remain consistent across all file sizes. We believe that the file size does not affect the speed at which we read 1 block of data. We also predict that sequential access will be faster than random access, as random access will have to consider the overhead randomly choosing a block and moving the file pointer to that block.

3.4.5 File Read Time Results

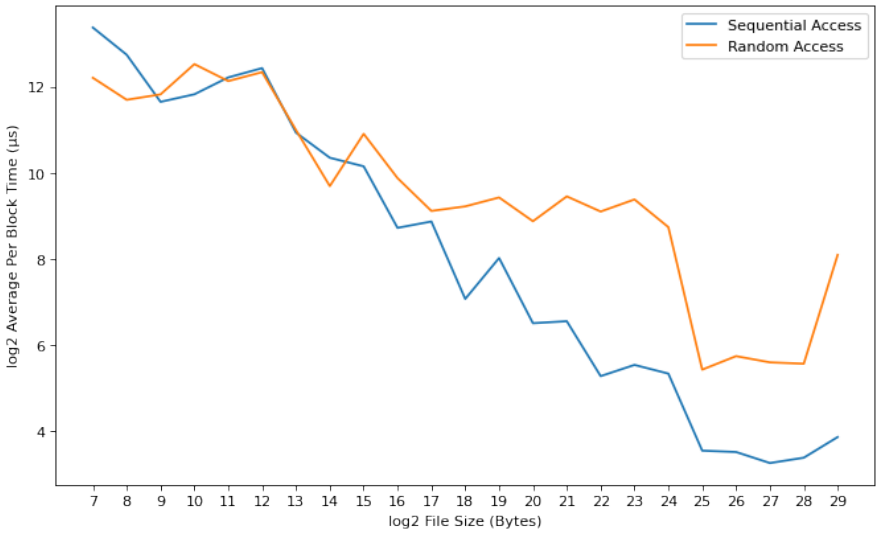

Figure 4: Average Block Read Time v. File Size

For the file read time results, we can see that as the file size increased, the average block time read got continuously got faster for both sequential and random access, which surprised us. We expected that file size would not be a large factor in average block read time and that the results would stay fairly consistent. These results seem to indicate that it may be reading cached data, but we had done a thorough confirmation that the cache was being cleared before each read. After some research into file systems, we believe that our results may be due the scheduler caching future blocks of data before we even read them. By accessing a large file, it would be reasonable to assume that we would want to access the entire file. When we read 1 block, the file system would also cache future blocks that we read later in the iteration, which would explain how the average block file read time got faster as the file size increased.

What did conform to our expectations was that overall, sequential access was much faster when compared to random access. The plot for sequential access tended to be less jagged and had smaller spikes, whereas the plot for random access showed larger spikes between file sizes, such as between 2^{24} bytes to 2^{25} bytes and 2^{28} bytes to 2^{29} bytes. As such, we believe for our results to be accurate.

3.4.6 Remote File Read Time

For the remote file read time experiments, our method of clearing the cache and getting results are the same as the previous section. What differed was where the files were stored and how we accessed those files. The files were located in our AWS server, the same server that we used in our network experiments. To access these files, we created a mount point in our virtual machine and enabled an NFS mount share on our files in the AWS server. By mounting the files in the AWS server onto the mount point in our virtual machine, we were able to access the files over the network using ordinary file operations.

We predict, based off our results from local file read time, that the average block read time will either stay consistent or increase, as we have to consider the network penalty of accessing files over the network.

3.4.7 Remote File Read Time Results

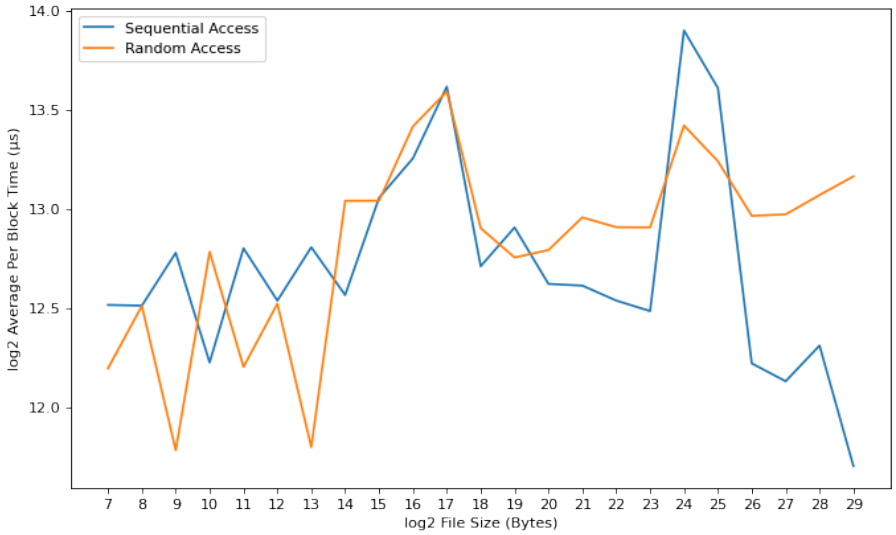

Figure 5: Average Remote Block Read Time v. File Size

For our remote file read time results, we can see that the average block read time were more consistent across all files sizes, ranging between 2^{12} microseconds and 2^{14} microseconds. Based off of the results from the local file read time results, we believe that this consistency in the plot is due to the network penalty of accessing a file over the network. As seen in our results in the previous section, the average block read time should be increasing as the file size increases, but this is not the case for remote file read. Here, the network penalty is the overhead of sharing a file over the network, where the amount of overhead and the file size are directly proportional to each other. This explains the consistent average block read times for our remote file read time results; although it will read blocks faster for a larger file, there will be a larger network penalty for sharing the file over the network. In addition, this also explains why the average remote block read time centered around 2^{12} microseconds and 2^{14} microseconds, since the highest value we saw in the average local block read time was around 2^{13} microseconds.

However, we were surprised that sequential access and random access both had very similar results throughout the whole process. We expected for random access to be slower than sequential access, but it ended up staying very similar to sequential access and was occasionally faster than it between file sizes 2^7 bytes and 2^{13} bytes. This may be due to the nature of the virtual machine, as we are unable to control or monitor on the status of the network connection during runtime.

3.4.8 Contention

We performed experiments to see how the number of processes running affect read time for a file block. In these experiments, we want to measure read time for a single file block and do this over a different number of processes. We calculate the average read time per block over 100 iterations and we also calculate deviation to see if there is significant variability in measurements. We start with reading a file block with one process and move to reading a file block for 16 different processes. We use *fork()* to create the child processes that we run the experiment on and increase the number of forks based on the number of processes we want by creating child processes. For each process, we read a file block from a different file and calculate the average read time in cycles for one block among all the processes. Cycles calculated using *rdtsc()* would then be converted to time in seconds. We will do this experiment over several iterations to find the average time to read 1 block for 1, 2, ..., 16 processes. We also want to measure standard deviation to see the variability in our measurements. We expect that the average read time to slow down as we increase the number of processes because of potential overhead. We expect the size of the file block to be around 4 KB so we create files around that size and replicate them so each process has a different file to read. One thing that is important within this experiment is that we do not want to access the file cache so we have to clear the cache after each read from a process using the drop caches command. We want to plot our results with the x axis being the number of processes running and the y axis being the average time of reading a file block across all processes.

3.4.9 Contention Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Read I/O for 1 process	N/A	30 μs	60 μs	63 μs	7.3 μs
Read I/O for 2 process	N/A	40 μs	75 μs	87 μs	0 μs
Read I/O for 4 process	N/A	55 μs	100 μs	118 μs	11.3 μs
Read I/O for 8 process	N/A	60 μs	125 μs	186 μs	3.5 μs
Read I/O for 16 process	N/A	70 μs	200 μs	316 μs	0 μs

Table 13: Contention Results (Process Number vs Read I/O)

3.4.10 Contention Results Graph

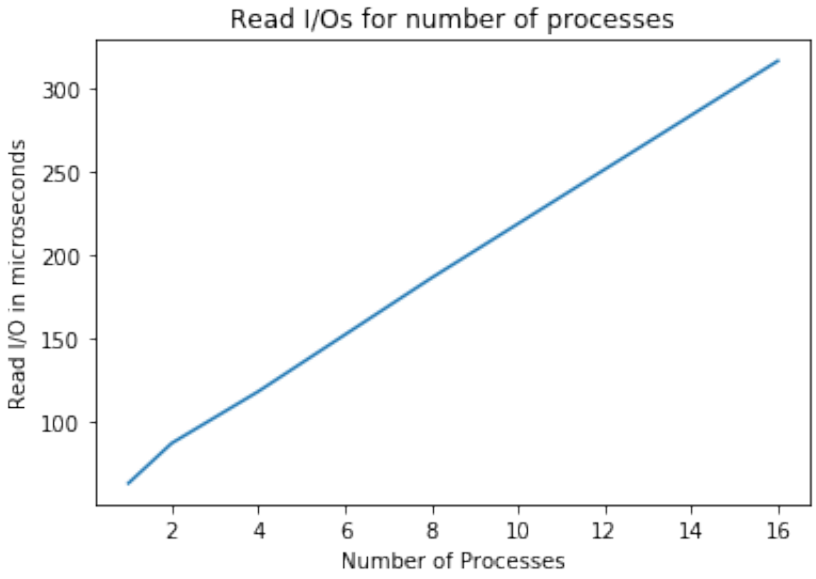

Figure 6: Read I/O v. Number of Processes

For the contention results, we saw that the average read time per block increased as the number of processes increased. This follows our expectation that more processes would lead more overhead and could lead to slower read times. We saw that with one process reading from one file that the average

read I/O time was around 63 microseconds. When we created a second process and reading from two different files, the average read I/O time was around 87 microseconds. The average read I/O time continued to increase to 118 microseconds for 4 processes, 186 microseconds for 8 processes, and 316 microseconds for 16 processes. The read I/O time continued to get higher and that made sense as you have to access more files with more processes. We can see in the graph that read I/O increased steadily which also meets our initial expectations. The results we got for each number of processes was roughly similar to expected performance or was at least reasonable.

4 Final Results Table

CPU Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Measurement Overhead (Process)	N/A	10 μs	10 μs	23 μs	0 μs
Measurement Overhead (System)	N/A	10 μs	10 μs	25 μs	1 μs
CPU Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Process Call(0 Arguments)	N/A	10 μs	10 μs	11 μs	$\approx$ 0 μs
Process Call(1 Arguments)	N/A	11 μs	11 μs	5 μs	$\approx$ 0 μs
Process Call(2 Arguments)	N/A	12 μs	12 μs	5 μs	$\approx$ 0 μs
Process Call(3 Arguments)	N/A	13 μs	13 μs	6 μs	$\approx$ 0 μs
Process Call(4 Arguments)	N/A	14 μs	14 μs	7 μs	$\approx$ 0 μs
Process Call(5 Arguments)	N/A	15 μs	15 μs	6 μs	$\approx$ 0 μs
Process Call(6 Arguments)	N/A	16 μs	16 μs	7 μs	$\approx$ 0 μs
Process Call(7 Arguments)	N/A	17 μs	17 μs	8 μs	$\approx$ 0 μs
CPU Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
System Call (getpid())	N/A	100 μs	100 μs	5474 μs	$\approx$ 0 μs
System Call (getsid())	N/A	100 μs	100 μs	5558 μs	$\approx$ 0 μs
CPU Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Process Creation/Run	N/A	400 μs	500 μs	991 μs	0 μs
Thread Creation/Run	N/A	100 μs	150 μs	135 μs	69 μs
CPU Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Context Switch (Process)	N/A	400 μs	400 μs	16.826 μs	0.637 μs
CPU Context Switch (Thread)	N/A	10 μs	10 μs	16.107 μs	0.403 μs
Memory Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Main Memory Access	80 ns	20 ns	100 ns	81 ns	N/A
L1 Cache Access	1 ns	4 ns	5 ns	2 ns	N/A
L2 Cache Access	4 ns	10 ns	14 ns	12 ns	N/A
Memory Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Writing Bandwidth	985 MB/s	500 MB/s	485 MB/s	375.38 MB/s	0 MB/s
Reading Bandwidth	985 MB/s	100 MB/s	885 MB/s	1033.06 MB/s	0 MB/s
Memory Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Page Fault Service Time	N/A	100 ms	200 ms	0.404 s	N/A
Network Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Ping (Loopback)	N/A	N/A	N/A	0.015 ms	0.003 ms
Round Trip Time (Loopback)	N/A	0.8 ms	1 ms	0.0878 ms	0.0063 ms
Ping (Remote)	N/A	N/A	N/A	13.726 ms	0.171 ms
Round Trip Time (Remote)	N/A	0.8 ms	15 ms	13.43 ms	0.186 ms

Network Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Peak Bandwidth (Loopback)	N/A	500 MB/s	485 MB/s	579 MB/s	N/A
Peak Bandwidth (Remote)	N/A	500 MB/s	10 MB/s	1.5 KB/s	N/A

Network Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Connection - open (Loopback)	N/A	0.8 ms	1 ms	0.0706 ms	0.00897 ms
Connection - close (Loopback)	N/A	0.8 ms	1 ms	0.0165 ms	0.0011 ms
Connection - open (Remote)	N/A	0.8 ms	15 ms	13.33 ms	1.09 ms
Connection - close (Remote)	N/A	0.8 ms	15 ms	0.0142 ms	0.0069 ms

File System Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Read I/O for 128 MB	N/A	2 μs	4 μs	6.8 μs	0 μs
Read I/O for 256 MB	N/A	2 μs	4 μs	5.4 μs	0 μs
Read I/O for 512 MB	N/A	2 μs	4 μs	6.1 μs	0 μs
Read I/O for 1 GB	N/A	2 μs	4 μs	6.7 μs	0 μs
Read I/O for 2 GB	N/A	5 μs	40 μs	24 μs	0 μs
Read I/O for 8 GB	N/A	0.5 ms	1.5 ms	2 ms	0 ms
Read I/O for 16 GB	N/A	0.5 ms	3 ms	4.9 ms	0 ms
Read I/O for 32 GB	N/A	0.5 ms	3.5 ms	5.5 ms	0 ms

File System Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Read I/O for 1 process	N/A	30 μs	60 μs	63 μs	7.3 μs
Read I/O for 2 process	N/A	40 μs	75 μs	87 μs	0 μs
Read I/O for 4 process	N/A	55 μs	100 μs	118 μs	11.3 μs
Read I/O for 8 process	N/A	60 μs	125 μs	186 μs	3.5 μs
Read I/O for 16 process	N/A	70 μs	200 μs	316 μs	0 μs

Table 15: All Final Results Table

5 References

- https://gcc.gnu.org/
- https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html
- https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+E5-2690+v2+%40+3.00GHzid=2057
- Larry McVoy and Carl Staelin, lmbench: Portable Tools for Performance Analysis, Proc. of USENIX Annual Technical Conference, January 1996