CSE 221: Operating Systems

Project Report

Harsh Gondaliya

Contents

1. Introduction

The goal of this project is to characterize and understand the performance of a Lenovo ThinkPad T470 system running the Ubuntu 20.04.2 LTS operating system. The system's performance for CPU, memory, network, and the file system has been characterized. Lenovo ThinkPad T470 is a business-class laptop launched by Lenovo in February 2017 and has more compute and memory resources than other laptops in the market. Ubuntu 20.04.2 LTS is a free open-source operating system based on the Linux kernel.

C programming language was used to write programs for carrying out different measurements. GCC Compiler version 9.3.0 was used for compiling programs and the -O0 optimization flag was set while compiling programs to disable optimizations that can affect this study's measurements. Moreover, to obtain precise measurements, all the measurements programs were run at the highest priority using the *nice* program in Linux and the CPU affinity was explicitly set to core 0 using the *taskset* program in Linux.

Carrying out the measurement experiments precisely while accounting for all possible system overheads has been an arduous journey. A significant amount of time was spent in designing the right methodology for the experiments. The methodology needs to make sure that the experiment is exactly measuring what it's supposed to measure and the results are not getting skewed by any overheads or cache effects. Around 120 hours have been spent to complete this project.

Section 2 provides a description of the machine used in this study. Section 3,4,5,6 describes the methodology and results for various experiments conducted in this study. Finally, Section 7 presents a summary of all the results obtained in this measurement study.

Please note that wherever applicable, results for various experiments are represented in MEAN ± (1* STANDARD_DEVIATION) form.

2. Machine Description

Component	Attribute Name	Attribute Value
Processor	Model	Intel® Core(TM) i7-7500U
	No. of physical CPU Cores	2
	No. of threads per CPU Core	2
	Cycle Time when a CPU Core operates at Base Frequency (2.70 GHz)	0.37 ns
	Cycle Time when a CPU Core operates at Max Turbo Frequency (3.50 GHz)	0.286 ns
	Cycle Time when a CPU Core operates at TDP-up Base Frequency (2.90 GHz)	0.345 ns
	L1d Cache Size	64 KiB
	L1i Cache Size	64 KiB
	L2 Cache Size	512 KiB
	L3 Cache Size	4 MiB
DRAM	Type	SODIMM DDR4 Synchronous Unbuffered
	Clock Frequency	2133 MHz
	Capacity	16 GiB
Memory Bus	Bandwidth	34.1 GB/s
I/O Bus	Type	PCIe 3.0
	Bandwidth (per direction per lane)	8 Gbps
	PCIe slot details with respective bandwidth (per direction)	One PCIe 3.0 x1 (8 Gbps) Two PCIe 3.0 x2 (16 Gbps each) One PCIe 3.0 x4 (32 Gbps)
Disk - SSD	Model	Intel® SSDPEKKF512G7L
	Series	Intel® SSD 600p Series
	Capacity	512 GiB

	Bandwidth - Sequential Read	up to 1800 MB/s
	Bandwidth - Sequential Write	up to 560 MB/s
	Bandwidth - Random Read	up to 155K IOPS
	Bandwidth - Random Write	up to 128K IOPS
	Latencies	Unknown
Disk - Hard Drive	No HDD available on the system	N/A
Network Card - Wired Ethernet	Bandwidth	1 Gbps
Network Card - Wireless Ethernet	Bandwidth	867 Mbps
Operating System	Name	Ubuntu
	Version	Ubuntu 20.04.2 LTS
	Release	February 4, 2021

3. CPU, Scheduling, and OS Services

3.1 Measuring CPU Cycle Time

While reporting the results of different experiments in this study, the count of CPU cycles needs to be converted into wall clock times. Thus, this experiment aims to determine the time taken by the CPU to complete one cycle.

3.1.1 Methodology

The following steps were followed to perform this experiment:

1. Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3. Took a timestamp before and after executing `sleep(2)` command. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions.
4. Computed CPU cycles that passed in two seconds duration using the below formula.
   ```
   CYCLES_IN_2_SECONDS    =    (END_TIME    -    START_TIME)    -
   READ_OVERHEAD
   ```
   Here, the read overhead corresponds to the number of CPU cycles used for taking the timestamps. This value is computed in the experiment described in section 3.1.2.
5. Repeated Steps 3 & 4 five hundred times and then computed the mean and standard deviation for the measurement readings.
6. Computed CPU cycle time in nanoseconds scale using the below formula.
   ```
   CYCLE_TIME = (1000000000*2) / MEAN_CYCLES_IN_2_SECONDS
   ```

3.1.2 Results

Predicted		**Measured**	
CPU Clock Frequency (GHz)	**CPU Cycle Time (ns)**	**CPU Clock Frequency (GHz)**	**CPU Cycle Time (ns)**
2.7	0.37	2.9 ± 0.000016	0.344

Given the fact that we had disabled the Intel Turbo Boost feature while doing measurements and that the system's CPU has a base frequency of 2.7 GHz [1], we predicted the CPU cycle time to be 0.37 ns. However, the measurement results show CPU clock frequency to be 2.9 GHz and thus the CPU cycle time to be 0.344 ns. The system's CPU has certain optimization features that raise the processor frequency to increase performance. This raised frequency is called the TDP-up base frequency of the CPU. Our measured CPU clock frequency and the corresponding cycle time match the TDP-up base frequency specified in the processor's specification [1]. The system doesn't allow us to disable this optimization feature, and hence the measured CPU clock frequency and cycle time need to be considered for all the CPU cycles to wall clock time conversions done in this measurement study.

3.2 Measuring Read Overhead

Different experiments in this study use the CPU cycle clock counters to take timestamp readings. This experiment aims to measure the overhead incurred in reading the cycle counters.

3.2.1 Methodology

The following steps were followed to perform this experiment:

1. Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3. Took two timestamps with no instruction embedded in between them. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions.
4. Computed the reading overhead in terms of CPU cycles using the below formula.
```
READING_OVERHEAD = (END_TIME - START_TIME)
```
5. Repeated Steps 3 & 4 five hundred times and then computed the mean and standard deviation for the measurement readings.
6. Computed the reading overhead in nanoseconds scale by multiplying the above result with the CPU cycle time (i.e., 0.344 ns).

3.2.2 Results

Predicted		**Measured**	
Reading Overhead (CPU Cycles)	**Reading Overhead (ns)**	**Reading Overhead (CPU Cycles)**	**Reading Overhead (ns)**
4	1.38	37.1 ± 1.6	12.76

Since the Cycles Per Instruction (CPI) is not available online for our system's CPU, we safely assumed its CPI to be 1. The `objdump` for the measurement program shows that there are four `MOV` instructions in between the two timestamp instructions. Thus, we had made a rough guess that the reading overhead must be four CPU cycles. However, the measurement results show that the average reading overhead is 37.1 cycles (i.e., 12.76 ns). It turns out that we missed considering the actual delay that will be incurred in reading the clock cycle counters and putting that value into the `edx` and `eax` registers. Once this delay is considered the measured results make more sense.

3.3 Measuring Loop Overhead

Different experiments in this study use a `for` loop to run many iterations of an operation. This experiment aims to measure the overhead incurred per iteration in running a `for` loop.

3.3.1 Methodology

The following steps were followed to perform this experiment:

1. Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3. Took a timestamp before and after executing a single iteration of `for` loop. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions.
4. Computed the loop overhead per iteration in terms of CPU cycles using the below formula.

   ```
   LOOP_OVERHEAD = (END_TIME - START_TIME) - READ_OVERHEAD
   ```
5. Repeated Steps 3 & 4 five hundred times and then computed 10 percent trimmed mean and standard deviation for the measurement readings.
6. Computed the loop overhead per iteration in nanoseconds by multiplying the above result with the CPU cycle time (i.e., 0.344 ns).

3.3.2 Results

Predicted		**Measured**	
Loop Overhead (CPU Cycles)	**Loop Overhead (ns)**	**Loop Overhead (CPU Cycles)**	**Loop Overhead (ns)**
5	1.72	5.8 ± 1.1	1.99

Since the Cycles Per Instruction (CPI) is not available online for our system's CPU, we safely assumed its CPI to be 1. The `objdump` for the measurement program shows that the `for` loop introduces five new instructions—excluding ones required for reading timestamps from registers—in between the two timestamps. Hence, we had made a rough guess that the loop overhead per iteration must be five CPU cycles. The measurement results show that the average loop overhead per iteration is 5.8 cycles (i.e., 1.99 ns) which is pretty close to the predicted results.

3.4 Measuring Procedure Call Overhead

This experiment aims to measure the procedure call overhead and increment overhead of an integer argument in the procedure call.

3.4.1 Methodology

The following steps were followed to measure the procedure call overhead with 0 to 7 integer arguments:

1. Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3. Took a timestamp before and after running a `for` loop that invokes a procedure call one million times. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions.
4. Computed the average procedure call overhead for a single run of one million procedure calls in terms of CPU cycles using the below formula.

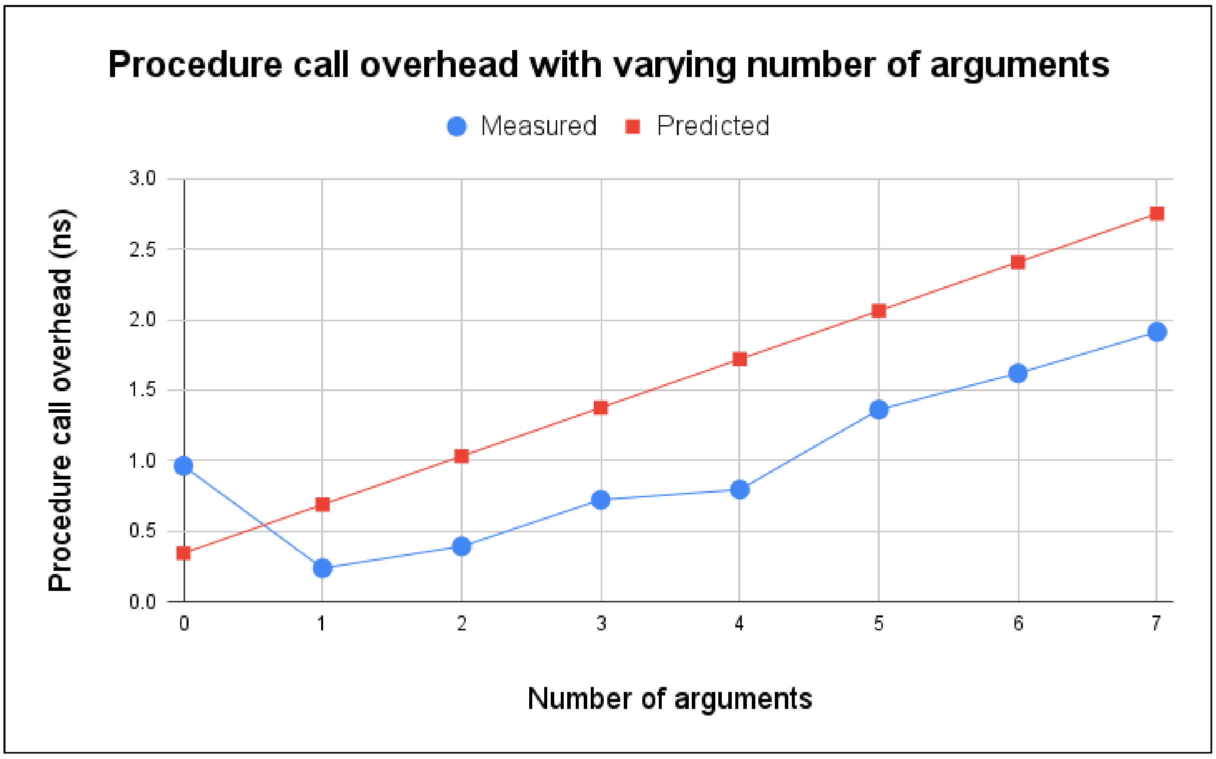
```
PROC_CALL_OVERHEAD =
(END_TIME - START_TIME) - READ_OVERHEAD
- (LOOP_OVERHEAD*NO_OF_ITERATIONS)

AVG_PROC_CALL_OVERHEAD =
PROC_CALL_OVERHEAD/NO_OF_ITERATIONS
```

5. Repeated Steps 3 & 4 two hundred times and then computed 10 percent trimmed mean and standard deviation for the measurement readings.
6. Computed the average procedure call overhead in nanoseconds by multiplying the above result with the CPU cycle time (i.e., 0.344 ns).

3.4.2 Results

Predicted		Measured	
Avg. increment in overhead with each argument (CPU Cycles)	**Avg. increment in overhead with each argument (ns)**	**Avg. increment in overhead with each argument (CPU Cycles)**	**Avg. increment in overhead with each argument (ns)**
1	0.344	0.81	0.279



Since the Cycles Per Instruction (CPI) is not available online for our system's CPU, we safely assumed its CPI to be 1. The `objdump` for the measurement programs shows that with each new argument added to the procedure call, a new `mov` instruction is introduced to store the new argument to stack. Hence, we expected the procedure call overhead to increase by one CPU cycle (i.e, 0.344 ns) for each new argument added. Ignoring the anomalous behavior of procedure call with zero arguments, the expected increment trend is observed for procedure call with one to seven arguments. The average increment overhead of an argument was 0.276 ns (almost equal to CPU cycle time).

Our `objdump` showed the expected sequence of instructions in between the timestamps instruction, and all possible optimizations (setting CPU affinity, disabling power optimization features of CPU, disabling hyperthreading, and setting priority for the program) were made. The standard deviation in all the measurements for this experiment was less than 1.4 percent. Thus, we attribute the odd behavior of procedure call overhead with zero arguments to be a measurement error. There must be some optimization being done by the CPU under the hood that is causing this anomalous result.

3.5 Measuring System Call Overhead

This experiment aims to measure the overhead for a minimal system call called `getpid()`.

3.5.1 Methodology

The following steps were followed to measure the system call overhead:

1. Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3. Took a timestamp before and after running a `for` loop that invokes `getpid()` system call one hundred thousand times. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. `syscall()` library function was used to bypass the cache and invoke the `getpid()` system call.
4. Computed the average procedure call overhead for a single run of one hundred thousand system calls in terms of CPU cycles using the below formula.

```
SYS_CALL_OVERHEAD =
(END_TIME - START_TIME) - READ_OVERHEAD
- (LOOP_OVERHEAD*NO_OF_ITERATIONS)

AVG_SYS_CALL_OVERHEAD = SYS_CALL_OVERHEAD/NO_OF_ITERATIONS
```

5. Repeated Steps 3 & 4 two hundred times and then computed 10 percent trimmed mean and standard deviation for the measurement readings.
6. Computed the average system call overhead in nanoseconds by multiplying the above result with the CPU cycle time (i.e., 0.344 ns).

3.5.2 Results

Predicted		**Measured**	
Avg. system call overhead (CPU Cycles)	**Avg. system call overhead (ns)**	**Avg. system call overhead (CPU Cycles)**	**Avg. system call overhead (ns)**
688.9	237	1313.3 ± 0.6	451.775

Considering the overhead involved in context switching from user to kernel while making a system call, we had made a rough guess that our system call overhead must be around three orders of magnitude more than the procedure call overhead we computed in Section 3.4. Since our results for procedure call with zero arguments overhead in Section 3.4 were anomalous, we took procedure call with one argument as our reference to predict system call overhead.

We predicted the system call overhead to be around 237 ns. The observed system call overhead was 451.775 ns. The measured overhead value was not too close to our prediction but our intuition that system call overhead is three orders of magnitude larger than the procedure call overhead was true.

3.6 Measuring Task Creation Time

This experiment aims to measure the time required to create and run both a process and a kernel thread.

3.6.1 Methodology

The following steps were followed in this experiment. Note that Step 3 differs in our methodology to measure the above entities, but the rest steps remain the same.

1. Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3.
    a. **Process Creation Time:** Took a timestamp before and after running a `for` loop that runs for ten thousand iterations, invoking `fork()` system call followed by immediate `exit()` call to the child process in each iteration. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Computed the average process creation time for a single run in terms of CPU cycles using the below formula.

    ```
    PROC_CREATE_TIME =
    (END_TIME - START_TIME) - READ_OVERHEAD
    - (LOOP_OVERHEAD*NO_OF_ITERATIONS)

    AVG_PROC_CREATE_TIME =
    PROC_CREATE_TIME/NO_OF_ITERATIONS
    ```

    b. **Process Creation and Running Time:** Took a timestamp before and after running a `for` loop that runs for ten thousand iterations. In each iteration, we first invoke `fork()` call in the parent process to create a child process, then invoke `execv()` call in the child process to run a blank program having zero instructions, and finally invoke `wait()` call in the parent process to wait until the child process terminates. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Computed the average process creation and running time for a single run in terms of CPU cycles using the below formula.

    ```
    PROC_CREATE_AND_RUN_TIME =
    ```

```
(END_TIME - START_TIME) - READ_OVERHEAD
- (LOOP_OVERHEAD*NO_OF_ITERATIONS)

AVG_PROC_CREATE_AND_RUN_TIME =
PROC_CREATE_AND_RUN_TIME/NO_OF_ITERATIONS
```

c. **Thread Creation and Running Time:** Took a timestamp before and after running a `for` loop that runs for ten thousand iterations. In each iteration, we first invoke the `pthread_create()` call to create a thread that runs a blank program having zero instructions and then invoke the `pthread_join()` call to wait until the previously created thread terminates. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Computed the average thread creation and running time for a single run in terms of CPU cycles using the below formula.

```
THREAD_CREATE_AND_RUN_TIME =
(END_TIME - START_TIME) - READ_OVERHEAD
- (LOOP_OVERHEAD*NO_OF_ITERATIONS)

AVG_THREAD_CREATE_AND_RUN_TIME =
THREAD_CREATE_AND_RUN_TIME/NO_OF_ITERATIONS
```

4. Repeated Stepf 3 two hundred times and then computed 10 percent trimmed mean and standard deviation for the measurement readings.
5. Computed the average process creation and running time both for processes and threads in nanoseconds by multiplying the above-measured values by cycle time (i.e., 0.344 ns).

3.6.2 Results

Predicted	Measured		
Roughly estimated process creation and running time to be five times more than thread creation and running time.	**Avg. time to create a process (µs)**	**Avg. time to create and run a process (µs)**	**Avg. time to create and run a thread (µs)**
	99.28 ± 0.24	116.27 ± 0.11	18.85 ± 0.10

Unlike new process creation, new thread creation does not require new memory address space allocation. With this insight, we made a rough guess that process creation and running time must be five times more expensive than thread creation and running time. Our results show the process creation and running time is six times more than the thread creation and running time.

3.7 Measuring Context Switch Time

This experiment aims to measure the context switch time from one process to another, and from one thread to another.

3.7.1 Methodology

The following steps were followed for this experiment. Note that Step 3 differs in our methodology to measure the above entities, but the rest steps remain the same.

1. Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3.
    a. **Process to Process Context Switch Time:** Firstly, we computed the pipe overhead involved in passing a token in a ring comprising two blocking pipes. We took a timestamp before and after running a `for` loop that runs for ten thousand iterations. In each iteration, within a single process, a token is passed for one time through all pipes in the ring. We compute the average pipe overhead involved in passing the token for one time in the ring. Each run of for loop is repeated two hundred times and summary statistics are computed for the average pipe overhead readings.

       Next, we took a timestamp before and after running a `for` loop that runs for ten thousand iterations. We designed a ring comprising two blocking pipes between two processes. In each iteration, a process uses a blocking pipe to pass a token to another process, and then the token is sent back to the former process using another pipe in the ring. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Computed the average process to process context switch time for a single run in terms of CPU cycles using the below formula.

       ```
       PROC_CONTEXT_SWITCH_TIME =
       (END_TIME - START_TIME) - READ_OVERHEAD
       - (LOOP_OVERHEAD*NO_OF_ITERATIONS)
       - (PIPE_OVERHEAD*NO_OF_ITERATIONS)

       AVG_PROC_CONTEXT_SWITCH_TIME =
       PROC_CONTEXT_SWITCH_TIME/(2*NO_OF_ITERATIONS)
       ```

    b. **Thread to Thread Context Switch Time:** Firstly, we computed the pipe overhead as described above. Next, we computed the overhead involved in creating two threads using `pthread_create()` call and then using `pthread_join()` call to wait until the two threads terminate. We took a timestamp before and after running a `for` loop that runs for ten thousand iterations. In each iteration, we create two threads using `pthread_create()` call and then use `pthread_join()` call to wait until the two threads terminate. Each run of for loop is repeated two hundred times and summary statistics are computed for the readings.

Next, we took a start timestamp, created two threads, waited until both threads terminate, and then finally took an end timestamp. Before the threads terminate, a token is passed to and fro between the two threads ten thousand times. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Computed the average thread to thread context switch time for a single run in terms of CPU cycles using the below formula.

```
THREAD_CONTEXT_SWITCH_TIME =
(END_TIME - START_TIME) - READ_OVERHEAD
- (LOOP_OVERHEAD*NO_OF_ITERATIONS)
- (PIPE_OVERHEAD*NO_OF_ITERATIONS)
- PTHREAD_CREATION_OVERHEAD

AVG_THREAD_CONTEXT_SWITCH_TIME =
THREAD_CONTEXT_SWITCH_TIME/(2*NO_OF_ITERATIONS)
```

4. Repeated Step 3 two hundred times and then computed 10 percent trimmed mean and standard deviation for the measurement readings.
5. Finally, computed the above-measured values in terms of nanoseconds by multiplying the above-measured results by the CPU cycle time (i.e., 0.344 ns).

3.7.2 Results

Predicted	**Measured**			
Estimated process to process context switch time to be two to three times the thread to thread context switch time.	**Avg. process to process context switch time (CPU Cycles)**	**Avg. thread to thread context switch time (CPU Cycles)**	**Avg. process to process context switch time (µs)**	**Avg. thread to thread context switch time (µs)**
	5072.4 ± 61.6	4812.6 ± 55.4	1.744	1.655

Assuming that threads are lighter than processes, share the same memory address space, and have much less of context to be saved, we assumed that thread to thread context switching time must be two to three times lesser than the process to process context switching time. But, our measured results show that the thread to thread context switching time is just 89 ns less than the process to process context switching time. We confirmed that the threads created using `pthread_create` were kernel-level threads and not user-level threads. Thus, we conclude that the process to process context switch overhead in Linux is slightly higher (not significantly higher) than the thread to thread context switch overhead.

4. Memory

4.1 Measuring RAM Access Time

This experiment aims to measure the individual integer access latency to main memory and L1, L2, L3 cache.

4.1.1 Methodology

The following steps were followed for this experiment:

1. Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Allocated memory for an integer array, initialized all integer array elements' value equal to 1, and created an array consisting of 100 pointers each pointing to random locations of the initialized array. The following algorithm was used while selecting random locations:

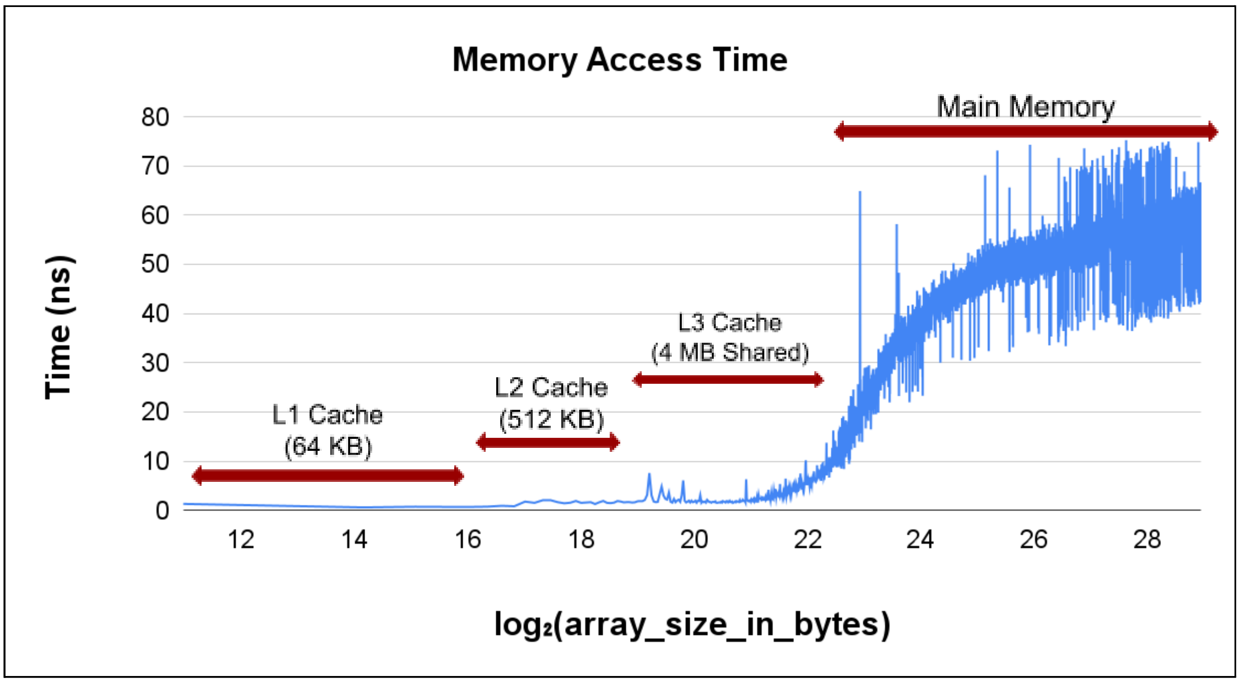
   ```
   stride = array_size/100;
   mem_partition = array_size - 1;
   while mem_partition > stride do:
       random_index = mem_partition - (rand()%stride);
       ptr[access++] = &arr[random_index];
       mem_partition -= stride;
   end
   ```

   The above algorithm makes best attempt to outsmart the CPU pre-fetcher. It has the following three features: (a) access array in reverse order (b) two consecutive accesses are at widely apart memory locations (c) accessed memory locations do not follow a fixed-stride pattern.

3. Took a timestamp before and after accessing the array at 99 random locations computed in Step 2. To obtain more accurate back-to-back load latency, instructions to access the array 99 times were added manually in place of using a loop. Note that the algorithm in Step 2 may only produce only 99 random memory locations for certain array sizes. Hence, we choose the lower bound of 99 and perform that many memory reads. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions.
4. Repeated Steps 2 and 3 for array sizes varying from 2 KB to 514 MB at increments of 16 KB.
5. Overall, we took 31,388 memory access time readings for different array sizes. Using the CPU cycle time of 0.344 ns, we converted the memory access time readings from CPU Cycles to nanoseconds format. Furthermore, we eliminated 164 outlier readings to better visualize results the results. The outlier latency readings ranged from 77 ns to 347 ns.

6. Based on the jumps that we noticed in the memory access time versus array size graph, we grouped our memory access time readings into L1, L2, L3, and main memory access time readings. For readings in each group, we computed a 10 percent trimmed mean. The results are in Section 4.1.2.

4.1.2 Results



Predicted		Measured	
L1 Cache Access Time (CPU Cycles)	**L1 Cache Access Time (ns)**	**L1 Cache Access Time (CPU Cycles)**	**L1 Cache Access Time (ns)**
5	1.72	2.3 ± 0.6	0.80 ± 0.22

Predicted		Measured	
L2 Cache Access Time (CPU Cycles)	**L2 Cache Access Time (ns)**	**L2 Cache Access Time (CPU Cycles)**	**L2 Cache Access Time (ns)**
12	4.13	9.5 ± 5.2	3.25 ± 1.78

Predicted		Measured	
L3 Cache Access Time (CPU Cycles)	**L3 Cache Access Time (ns)**	**L3 Cache Access Time (CPU Cycles)**	**L3 Cache Access Time (ns)**
42	14.45	51.4 ± 21.5	17.69 ± 7.39

Predicted		Measured	
Main Memory Access Time (CPU Cycles)	**Main Memory Access Time (ns)**	**Main Memory Access Time (CPU Cycles)**	**Main Memory Access Time (ns)**
261	90	161.7 ± 21.5	55.61 ± 5.07

We used the benchmark test results for Intel Kaby Lake i7–7660U CPU published online by Andriy Berestovskyy [5] to predict L1, L2, L3 cache and main memory access time. Our processor is from the same family of processors as [5] and is just one release older than it. Our measured L1, L2, L3 cache access time results are pretty close to the predicted values but have a high standard deviation. Various optimizations in modern-day CPUs such as cache line pre-fetching skew the memory access time readings greatly and hence such high standard deviation is measured. Secondly, the measured main memory access time was much lower than the predicted main memory access time. We again attribute this discrepancy between predicted and measured results to the various optimizations introduced by modern-day CPUs.

4.1.3 Experience

The most difficult part of this experiment was to come up with random memory locations that outsmart CPU prefetcher. We tried using fixed-length strides but it was very well predicted by the CPU pre-fetcher. It always gave us 20-30 ns latency readings for large array sizes. We also tried to create random memory locations for a given array size but it did not help entirely. We finally found that a combination of randomness along with wide stride size and reverse direction access can outsmart CPU prefetcher to a reasonable extent.

4.2 Measuring RAM Bandwidth

This experiment aims to measure the read and write bandwidth of RAM.

4.2.1 Methodology

The following steps were followed for this experiment. Note that Step 3 differs in our methodology to measure the above entities, but the rest steps remain the same.

1. Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3.
    a. **Read Bandwidth:** Took a timestamp before and after running an unrolled `for` loop that sums up all the integers at 64 bytes offset in a 400 MB-sized array. Since 64 bytes cache line prefetching is done in x86 architecture, the summing operation ensures that the entire 400 MB data is being read from memory. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Finally, computed the read bandwidth in GB/s using the following calculations.

    ```
    READ_TIME =
    (END_TIME - START_TIME) - READ_OVERHEAD
    - (LOOP_OVERHEAD * NO_OF_ITERATIONS)

    READ_BANDWIDTH = ARR_SIZE/(READ_TIME * 0.344)
    ```

    b. **Write Bandwidth:** Took a timestamp before and after running an unrolled `for` loop that writes integer values at 64 bytes offset in a 400 MB sized array. Since 64 bytes cache line prefetching is done in x86 architecture, the write operation ensures that the entire 400 MB data is being read from memory first and then written back. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Finally, computed the write bandwidth in GB/s using the following calculations.

    ```
    WRITE_TIME =
    (END_TIME - START_TIME) - READ_OVERHEAD
    - (LOOP_OVERHEAD * NO_OF_ITERATIONS)

    WRITE_BANDWIDTH = ARR_SIZE/(WRITE_TIME * 0.344)
    ```

4. Repeated Step 3 one hundred times and then computed 10 percent trimmed mean and standard deviation for the measurement readings.

4.2.2 Results

Predicted		Measured	
Read Bandwidth (GB/s)	**Write Bandwidth (GB/s)**	**Read Bandwidth (GB/s)**	**Write Bandwidth (GB/s)**
17	8.5	14.38 ± 0.04	7.42 ± 0.01

Considering our single channel 2133 MHz RAM and 64 bit wide memory bus, we had predicted RAM read bandwidth to be 17 GB/s. For a memory write operation, since the data is first to read into the cache line before being written back to memory, we had predicted write bandwidth to be half of the read bandwidth. Our measured results are quite close to our predictions. The bandwidth values predicted using hardware specifications are actually the peak bandwidth that can be achieved. Since many other complexities and overhead are involved at the lower hardware level, it is difficult to achieve the peak bandwidth using our measurement code.

4.3 Measuring Page Fault Service Time

This experiment aims to measure the time to fault an entire page from the disk.

4.3.1 Methodology

The following steps were followed for this experiment:

1. Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3. Used `mmap` command to map a 20 GB file to the virtual address space of the process executing this measurement.
4. Took a timestamp before and after running a `for` loop that accesses the memory-mapped file in the reverse direction at a stride of 2 MB. The `for` loop ran for 10.000 iterations and the file access in each iteration resulted in a page fault. We used `/usr/bin/time -v` utility to verify that each iteration resulted in a major page fault in our Linux system. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Finally, computed the average time to fault a single page using the below calculations.

```
PAGE_FAULT_SERVICE_TIME =
(END_TIME - START_TIME)*(0.344)/NO_OF_ITERATIONS
```

5. Repeated Step 4 one hundred times and then computed 10 percent trimmed mean and standard deviation for the measurement readings.

4.3.2 Results

Predicted			**Measured**	
Base Hardware Performance (µs)	**Software Overhead (µs)**	**Total Page Fault Service Time (µs)**	**Page Fault Service Time (CPU Cycles)**	**Page Fault Service Time (µs)**
0.417	100	100.417	1,430,606.2 ± 3037.36	492.13 ± 1.04

Measured Time to Access One Byte	
From Main Memory (ns)	**From Disk after page fault (ns)**
13.9 ± 1.27	349.27 ± 0.74

Considering the fact that L1, L2, L3 cache and main memory will be accessed for finding a page before the page fault occurs, we added the respective cache access time and memory access time to predict the base hardware performance. We added the main memory access time twice while doing this computation because once the desired page is brought into the main memory, it is accessed again by the process. We also used the RAM read bandwidth computed in Section 4.2 to compute the time required to read one page into the main memory and added this time to the base hardware performance computation. Besides this, we roughly guessed the software overhead to be 100 µs. This overhead involved trapping into the OS when a page fault occurs, saving register values, invoking the page fault service handler to find the desired page on disk, allocating frame in main memory, adding page table entry in the page table, resetting registers, and resuming operations. Our results are much higher than our predicted values. We realize that the software overhead involved in the page fault service operation is much more than what we had expected. We verified that our measurement produces the desired number of major page faults and hence are confident of the accuracy of the methodology used. Moreover, we learned that accessing a memory-mapped file in the reverse direction at large strides deterministically produces page faults and outsmarts CPU optimizations.

Dividing the page fault service time by page size (4096 KB), we get 349.27 ns. Dividing the main memory access time measured in Section 4.1 by integer size (4 B), we get 13.9 ns. Hence, accessing a byte that is not in the main memory and causes page fault is about 3x slower than accessing a byte directly from the main memory.

5. Network

The remote machine used for the experiments described in Sections 5.1, 5.2, and 6.3 was an AWS EC2 instance hosted in the Oregon region. Using the TTL value obtained using ping, we inferred that there are 24 hops between our local machine and the remote machine. The following are the specifications of the remote machine:

Component	Attribute Name	Attribute Value
AWS Instance	Type	t2.micro
Processor	Model	Intel® Xeon® CPU E5-2676 v3, 64-bit
	No. of vCPUs	1
	CPU Clock Frequency	2.40 GHz
	Cycle Time when a CPU Core operates at Base Frequency (2.40 GHz)	0.42 ns
	L1d Cache Size	32 KiB
	L1i Cache Size	32 KiB
	L2 Cache Size	256 KiB
	L3 Cache Size	30 MiB
DRAM	Type	DIMM RAM
	Capacity	1 GiB
Disk - SSD	Capacity	32 GiB
	Bandwidth	up to 125 MB/s
	IOPS	3000
Disk - Hard Drive	No HDD available on the system	N/A
Network Card - Elastic Network Interface (ENI)	Bandwidth allotted for AWS t2.micro instance	60-70 Mbps
Operating System	Name and Version	Amazon Linux 2
	Release	February 22, 2022

5.1 Measuring Round Trip Time

This experiment aims to measure the ping and application-layer round trip time (RTT) to a localhost interface and a remote host.

5.1.1 Methodology

For localhost interface measurements, the local machine served as both client and the server machine. For remote host measurements, the local machine served as the client while the remote machine served as the server.

Step 1 was followed for ping measurements while Step 2 to Step 5 were followed for the application layer RTT measurements.

1. Used `ping` command to ping from the client to the server 1000 times. The average RTT value and standard deviation reported by the `ping` command were noted.
2. **Client Machine:** Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
   Server Machine: Used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
3. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
4. Took a start timestamp before sending 64 bytes of data from client to server. The client sends 64 bytes of data to the server using TCP and waits until it receives these 64 bytes of data back from the server. An end timestamp is taken once the 64 bytes of data is received from the server. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Finally, computed the round trip time in nanoseconds scale using the below calculations.
   ```
   RTT = (END_TIME - START_TIME - READ_OVERHEAD) *(0.344)
   ```
5. Repeated Step 4 1000 times and then computed 10 percent trimmed mean and standard deviation for the measurement readings.

5.1.2 Results

Measured Ping Time	
Local (ms)	**Remote (ms)**
0.046 ± 0.008	61.315 ± 36.391

Predicted Application Layer RTT		Measured Application Layer RTT	
Local (ms)	**Remote (ms)**	**Local (ms)**	**Remote (ms)**
0.051	67.446	0.020 ± 0.007	59.039 ± 13.359

We had roughly predicted that the application layer RTT for both local and the remote host will be 10 percent more than the respective ping RTT. It is because we are using TCP connection for measuring application layer RTT and hence an additional TCP header needs to be processed. But, our measured results show that application layer RTT for remote is almost the same as ping RTT for the remote. Surprisingly, the measured application layer RTT was less than the ping RTT for the localhost interface. It turns out that routers on the internet give higher priority to TCP packets than ICMP packets. Thus, the 24 routers in our path to the remote host are very likely to incur an excess delay to our ping ICMP packets than the TCP packets. This explains why application layer RTT is less than ping RTT, and for the same reason, ping is not considered an efficient method to measure round trip time on the internet [6,7]. Thus, our method of measuring RTT at the application layer is more accurate.

Taking the difference between the measured RTT for local and remote, in a single RTT, for a 64 bytes packet, 0.02 ms time overhead is incurred by OS while 59.01 ms overhead is incurred by the network. The overhead by OS involves the time incurred for a system call, data copy between rx_ring and socket buffer, data copy between socket buffer and user-level buffer, and context switch time [8]. The network overhead involves the serialization delay, propagation delay, and the queuing delay faced by the packet in its journey to the remote host.

OS overhead is generally negligible in comparison to network overhead for long-distance communication. Considering the physical distance to our remote host in Oregon and the speed of light in optical fibers, the ideal network overhead must be 15.752 ms. However, our measured network overhead is 59.01 ms. Thus, our measured network overhead is ~4x higher than what it would be in idea hardware. It is because we have 24 routers on our way to the remote host. These internet routers incur a high queuing delay because they are serving a large amount of internet traffic.

5.2 Measuring Peak Bandwidth

This experiment aims to measure the peak bandwidth of a TCP connection to a localhost interface and a remote host.

5.2.1 Methodology

For localhost interface measurements, the local machine served as both client and the server machine. For remote host measurements, the local machine served as the client while the remote machine served as the server.
The following steps were followed for the peak bandwidth measurements:

1. **Client Machine:** Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
   Server Machine: Used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3. Took a timestamp before and after executing `send` function with `MSG_WAITALL` flag set. The `send` function is provided by the socket programming API and blocks until all the bits of data are sent to the socket buffer. 512 MB data was sent while measuring peak bandwidth for localhost while 10 MB data was sent while measuring peak bandwidth for the remote host. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Finally, computed the round trip time in Mbps scale using the below calculations.
   ```
   SEND_TIME = (END_TIME - START_TIME - READ_OVERHEAD)
   *(0.344)
   PEAK_BW = (DATA_SIZE_IN_BITS/SEND_TIME)*1000
   ```
4. Repeated Step 4 100 times and then computed 10 percent trimmed mean and standard deviation for the measurement readings.

5.2.2 Results

Predicted Peak Bandwidth		**Measured Peak Bandwidth**	
Local (Gbps)	**Remote (Mbps)**	**Local (Gbps)**	**Remote (Mbps)**
59.36	2.74	27.344 ± 2.444	2.889 ± 0.066

Since localhost interface simply places packet from its send queue to receive queue and does not send the packet out on the network, we predicted its bandwidth to be equal to the memory write bandwidth (59.36 Gbps) measured in Section 4.2. The Google Speed Test utility showed our internet connection download speed to be 51.8 Mbps and upload speed to be 2.74 Mbps. Thus, we predicted the peak bandwidth for the remote host connection to be 2.74 Gbps. Our measured peak bandwidth for localhost is four orders of magnitude greater than the peak bandwidth for the remote host.

Our measured peak bandwidth for the localhost is approximately half the value that we had predicted (idea hardware performance). This difference is due to the fact that packets sent to the localhost interface do need to undergo all the packet processing in the network stack. Thus, bandwidth performance will be bottlenecked and we will not achieve bandwidth equal to the memory write bandwidth. Our measured peak bandwidth for the remote host was almost the same as our predicted value (bandwidth provided by ISP) which convinces us that our methodology to measure peak bandwidth is pretty accurate.

5.3 Measuring Connection Overhead

This experiment aims to measure the overhead involved in TCP connection set-up and TCP connection teardown.

5.3.1 Methodology

For localhost interface measurements, the local machine served as both client and the server machine. For remote host measurements, the local machine served as the client while the remote machine served as the server.
The following steps were followed for the connection overhead measurements.

1. **Client Machine:** Disabled hyperthreading from BIOS, turned OFF Intel Turbo Boost feature from the command line, used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
   Server Machine: Used the `nice` program in Linux to run the measurement program with the highest priority, and used the `taskset` program in Linux to set CPU affinity to core 0 for the measurement program.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3. **Connection Set-up Overhead**: Took a timestamp before and after executing `connect` function. The `connect` function is provided by the socket programming API. It creates and sets up a TCP connection to the specified destination.
   Connection Teardown Overhead: Took a timestamp before and after executing the `close` function. The `close` function is provided by the socket programming API. It terminates a TCP connection from the function caller's end.

   The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Finally, computed the overhead time in the nanoseconds scale using the below calculations.

```
OVERHEAD_TIME =
(END_TIME - START_TIME - READ_OVERHEAD) *(0.344)
```

4. Repeated Step 3 1000 times and then computed 10 percent trimmed mean and standard deviation for the measurement readings.

5.3.2 Results

Predicted Connection Set-up Overhead		Measured Connection Set-up Overhead	
Local (ms)	**Remote (ms)**	**Local (ms)**	**Remote (ms)**
0.020	59.039	0.024 ± 0.001	55.174 ± 5.151

Predicted Connection Teardown Overhead		Measured Connection Teardown Overhead	
Local (ms)	**Remote (ms)**	**Local (ms)**	**Remote (ms)**
0.020	59.039	0.007 ± 0.0001	0.056 ± 0.001

We predicted the TCP connection set-up and TCP connection teardown time for both local and remote host to be equal to one RTT. It is because connection set-up requires an SYN and SYN-ACK packet exchange between sender and receiver, while connection teardown requires FIN and FIN-ACK exchange.

Our measured TCP connection set-up overhead values are pretty close to the predicted values (one RTT) which convinces us of the accuracy of the methodology used. Our measured TCP connection teardown overhead doesn't match our predicted values. We realize that the `close` function does not block until it receives an ACK. It immediately returns after sending the FIN packet. Hence, the TCP connection teardown overhead is roughly equal to a system call execution overhead.

6. File Systems

6.1 Measuring File Cache Size

This experiment aims to measure the file cache size of the system.

6.1.1 Methodology

The following steps were followed to measure the file cache size of the system.

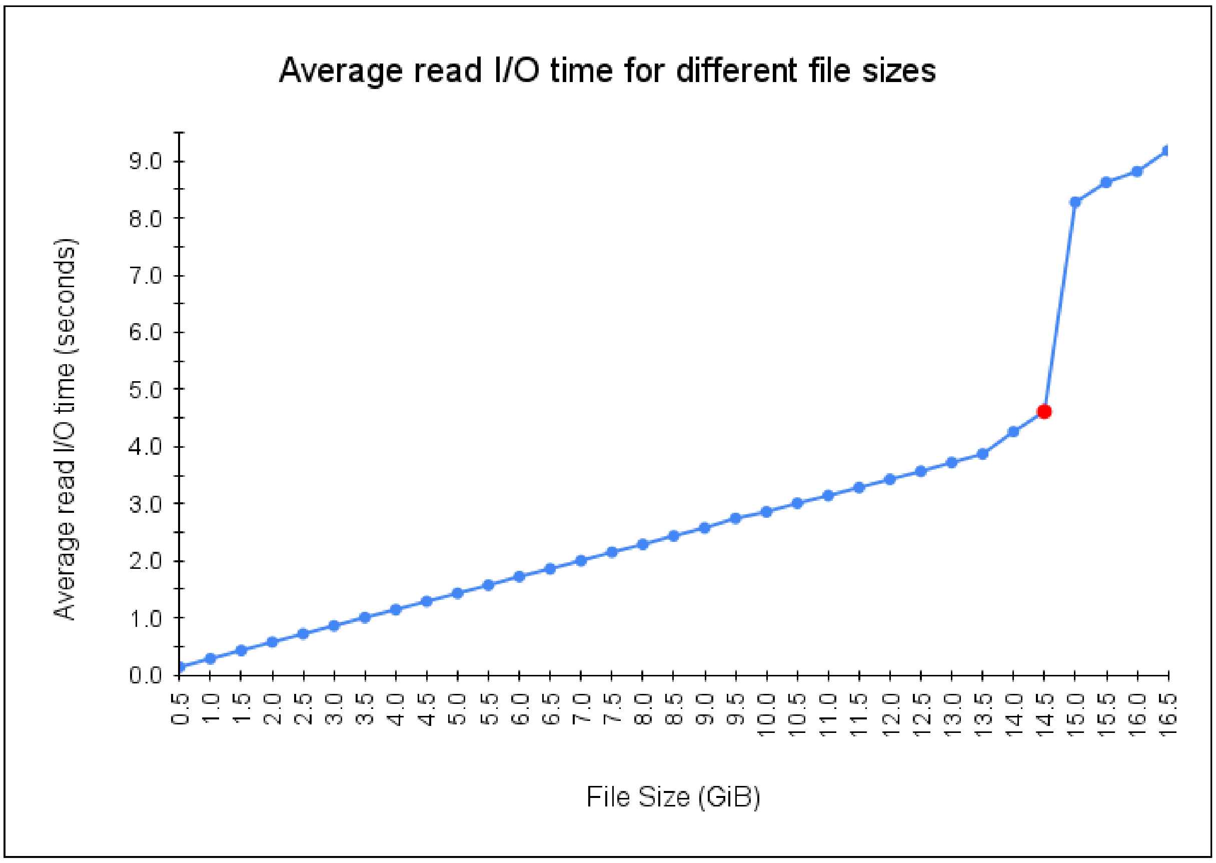
1. Created files in increments of 0.5 GiB `fallocate` command. The file sizes used for the measurement varied from 0.5 GiB to 16.5 GiB.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3. Read a file block by block using `pread` system call in Linux. Our system's block size was 4096 bytes. This step is intended to bring the file into the file cache.
4. Took a timestamp before and after reading the same file block by block in the reverse direction 10 times. The file was read in the reverse direction to avoid prefetching issues. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Finally, computed the average file read time in seconds scale using the below calculations.

   ```
   FILE_READ_TIME =
   (END_TIME - START_TIME)*(0.344)/(NO_OF_FILE_READS * 10^9)
   ```

5. Repeated Step 3 and Step 4 for each file. File cache is cleared before beginning the measurements for a new file.
6. Finally, noticed the file size where a non-linear increase in file read time is noticed for the first time. The non-linear increase indicates that the file cache was not able to accommodate the file and hence the file size gives us a good approximation for the file cache size.

   Note: We were unable to do the experiment at a granularity lower than 0.5 GiB file size due to the limited storage space available on our system.

6.1.2 Results



File Cache Size	
Predicted (GiB)	**Measured (GiB)**
14.6	14.5

We used the `watch` Linux command to determine the available RAM while the system is turned ON and no other applications are running. On our system having 16 GiB RAM, 14.6 GiB RAM was available for use. The rest memory was occupied by the operating system and other background utilities running on the system. Based on this we speculated that the maximum limit of our file cache size must be 14.6 GiB. It turns out that our measured file cache size value closely matches our predicted value. From 14.5 GiB file size onwards, an exponential increase in file read time was noticed. It is because the file cache was not able to accommodate the 15 GiB file entirely and hence there were constant page faults. The measurement methodology proved to be accurate in measuring the file cache size.

6.2 Measuring Local File Read Time

This experiment aims to measure the sequential and random access file read time on the local file system.

6.2.1 Methodology

The following steps were followed for this measurement experiment:

1. Created files in increments of power of 2 using `fallocate` command. The file sizes used for the measurement varied from 128 KiB to 16 GiB.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3. Cleared file cache using the following system call:
   `system("sync; echo 3 > /proc/sys/vm/drop_caches");`
4. **Sequential Access:** Took a timestamp before and after reading an entire file block by block.
   Random Access: Took a timestamp before and after reading a series of random blocks of a file. Used the following pseudocode to determine the random block to be accessed.

   ```
   chunks = file_size/4096;
   i = chunks;
   while i > 0:
   access_block = rand()%i;
   i = i - 1;
   ```

   The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. O_DIRECT and O_SYNC flags were set while reading files to prevent kernel caching effects and get more precise measurements.

5. Repeated Step 3 and Step 4 100 times for each file. Finally, computed the average per block file read time in nanoseconds scale using the below calculations.

   ```
   AVG_PER_BLOCK_FILE_READ_TIME =
   (END_TIME - START_TIME)*(0.344)/(NO_OF_BLOCKS_READ * 100)
   ```

6.2.2 Results

As per our disk specification [3], its sequential read bandwidth is 1800 MBps and the random read bandwidth is 155K IOPS (605.47 MBps because 4KB is transferred per IO). We used this bandwidth to predict the hardware performance for reading different file sizes. Each block read from disk involves a read system call and a context switch. Context switch majorly contributes to the software overhead. Due to pre-fetching, a single context switch may bring in multiple blocks into the file cache. Hence, we roughly assumed that 5 percent of the block accesses result in context switch. Accordingly, we used system call time and context switch time results from Section 3 to estimate software overhead for reading files from disk.

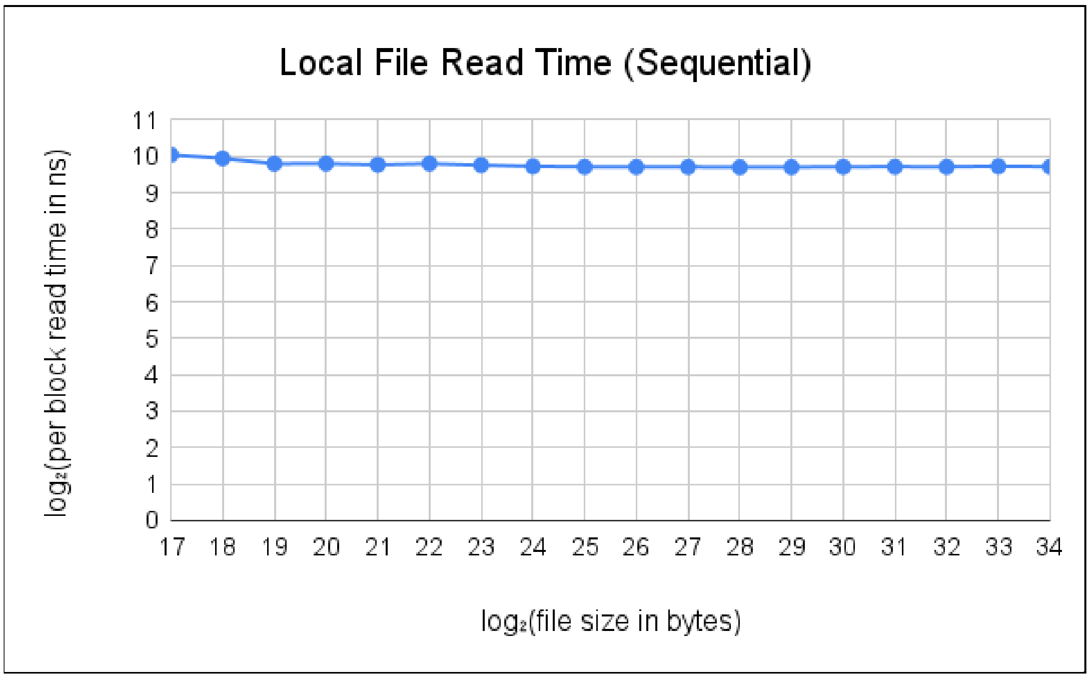
The measured file read times are reasonably lower than our predicted values for both sequential and random access. We speculate that we are not interpreting the bandwidth specified in the disk specifications properly and thus predicting a very high value for hardware performance.

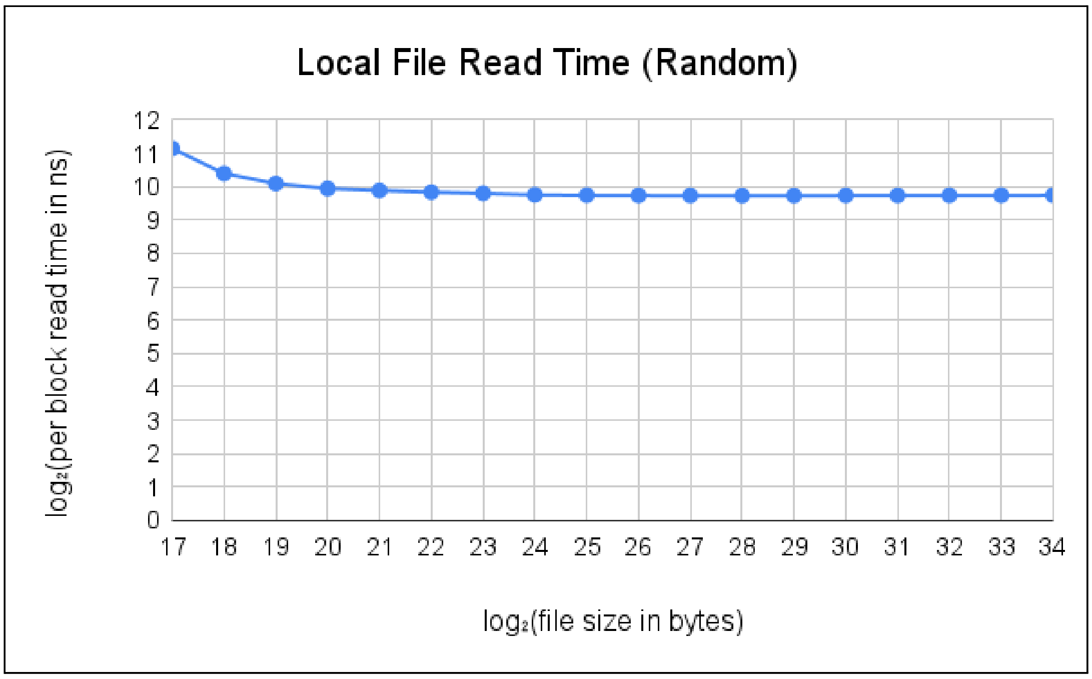
File Read Time (Sequential)			**File Read Time (Random)**		
File Size	**Predicted (ms)**	**Measured (ms)**	**File Size**	**Predicted (ms)**	**Measured (ms)**
128 KB	0.073	0.034	128 KB	0.208	0.036
256 KB	0.146	0.063	256 KB	0.418	0.064
512 KB	0.292	0.114	512 KB	0.838	0.122
1 MB	0.584	0.227	1 MB	1.678	0.236
2 MB	1.167	0.446	2 MB	3.357	0.468
4 MB	2.334	0.910	4 MB	6.717	0.918
8 MB	4.669	1.769	8 MB	13.436	1.806
16 MB	9.339	3.454	16 MB	26.874	3.514
32 MB	18.677	6.837	32 MB	53.749	6.986
64 MB	37.354	13.646	64 MB	107.499	13.912
128 MB	74.707	27.311	128 MB	215.001	27.796
256 MB	149.415	54.348	256 MB	430.003	55.432
512 MB	298.829	108.852	512 MB	860.007	111.013
1 GB	597.659	219.001	1 GB	1720.017	222.874

2 GB	1195.319	438.498	2 GB	3440.035	445.473
4 GB	2390.637	875.432	4 GB	6880.071	893.023
8 GB	4781.273	1769.539	8 GB	13760.146	1788.364
16 GB	9562.547	3511.498	16 GB	27520.293	3581.520

The constant time trend observed in the average block time graph for sequential access is as expected due to the pre-fetching. But, we didn't expect a constant average per block time for random access. For all file sizes except 16 GiB file, our file read times pretty much match the file read times from the file cache experiment. We have verified that we are properly clearing the file cache. Thus, this indicates that heavy pre-fetching is going on and our random accesses are unable to bring a fresh block from the disk.

For the 16 GiB file anomaly, we speculate a higher time value in the file cache experiment is because in that experiment our measurement program reads the entire file block by block 100 times without any pause in between. However, in the file read time experiment we read the file once, took time stamps, paused for clearing buffer cache, and then started re-reading the file again. Due to this, the former might result in blocks missing in the file cache and thus lead to higher file read time.





6.3 Measuring Remote File Read Time

This experiment aims to measure the sequential and random access file read time on a file system on a remote host.

Remote Host (Virtual Machine) Specifications	
Hypervisor	Oracle VirtualBox 6.1.28
Specifications of Machine Hosting Virtual Machine	Windows 10 Operating System
	Intel® Core™ i7-1165G7 Processor @ 2.80 GHz
	8 Logical Cores
	16 GiB RAM
	512 GiB SSD (No HDD)
Specifications of Virtual Machine	Ubuntu 20.04.4 LTS Operating System
	8 GiB RAM
	4 vCPUs
	32 GiB Virtual Disk Image (Dynamically Allocated)
	Bridged Network Adapter

6.3.1 Methodology

The following steps were followed for this measurement experiment:

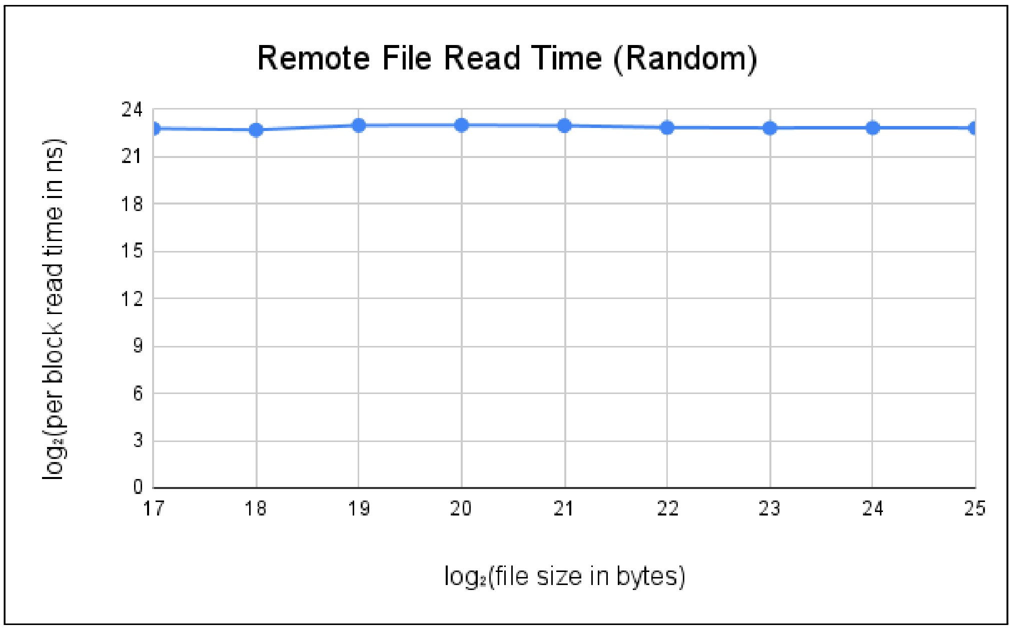
1. Created files in increments of power of 2 using `fallocate` command. The file sizes used for the measurement varied from 128 KiB to 32 MiB.
   Note: We were unable to try larger file sizes and run the measurement program for larger iterations. Our virtual machine crashed often when we tried to read a larger file size or read a large file for multiple iterations. Hence, we had to limit the number of iterations to 5 and the largest file size to 32 MiB.
2. Setup an NFSv4 server on a virtual machine hosted on a laptop in the same home wireless network as our test system. A directory is exported to our test system and firewall rules are altered to allow the test system to access files in the exported directory.
3. Setup NFSv4 client and mounted the directory exported by the NFSv4 server to a local directory. `sync` flag was set while mounting to prevent client-side caching. We were not able to find any documentation that shows ways to prevent client-side caching.
4. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
5. **Sequential Access:** Took a timestamp before and after reading an entire file on remote system block by block.
   Random Access: Took a timestamp before and after reading a series of random blocks of a file on the remote system. Used the following pseudocode to determine the random block to be accessed.
   ```
   chunks = file_size/4096;
   i = chunks;
   while i > 0:
   access_block = rand()%i;
   i = i - 1;
   ```
   The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. O_DIRECT and O_SYNC flags were set while reading files to prevent kernel caching effects and get more precise measurements.

6. Repeated Step 5 five times for each file. Finally, computed the average per block file read time in nanoseconds scale using the below calculations.
   ```
   AVG_PER_BLOCK_FILE_READ_TIME =
   (END_TIME - START_TIME)*(0.344)/(NO_OF_BLOCKS_READ * 100)
   ```

6.3.2 Results

Network penalty per block (4KB) while reading file from remote host	
Predicted (ms)	**Measured (ms)**
9.86	7.21

File Read Time (Sequential)			File Read Time (Random)		
File Size	**Predicted (s)**	**Measured (s)**	**File Size**	**Predicted (s)**	**Measured (s)**
128 KB	0.316	0.232	128 KB	0.316	0.232
256 KB	0.631	0.445	256 KB	0.631	0.435
512 KB	1.262	0.939	512 KB	1.262	1.067
1 MB	2.525	1.789	1 MB	2.525	2.158
2 MB	5.050	3.813	2 MB	5.050	4.220
4 MB	10.100	7.935	4 MB	10.100	7.737
8 MB	20.200	13.836	8 MB	20.200	15.178
16 MB	40.399	29.383	16 MB	40.399	30.635
32 MB	80.798	59.495	32 MB	80.798	60.757



Using our application layer RTT measurement program we measured the RTT between local and remote host to be 8.94 ms. Using `iperf` we measured the TCP bandwidth between the local and remote host to be 4.44 MBps. Thus, the network is going to be the key bottleneck in reading remote file and the disk access overhead is negligible in comparison to it. Adding the serialization delay of entire file and propagation delay per block read (because RPC inovked on each read call), we predicted the overall remote file read time.

We subtracted the remote file read time from the corresponding local file read time to obtain the network penalty for each block, and then took an average. Our predicted values are slightly higher than the measured values. We attribute this minor difference to some parallelism that might be going on in RPC invocation. But, the measured values convinces us that the methodology is reading files from the remote host and not from the client cache. We attribute the constant read time trend for both sequential and random access to the heavy file caching and prefetching discussed in Section 6.2.

6.4 Contention

This experiment aims to measure the average time to read one file system block as a function of multiple processes simultaneously performing same operation on different files on same disk.

6.4.1 Methodology

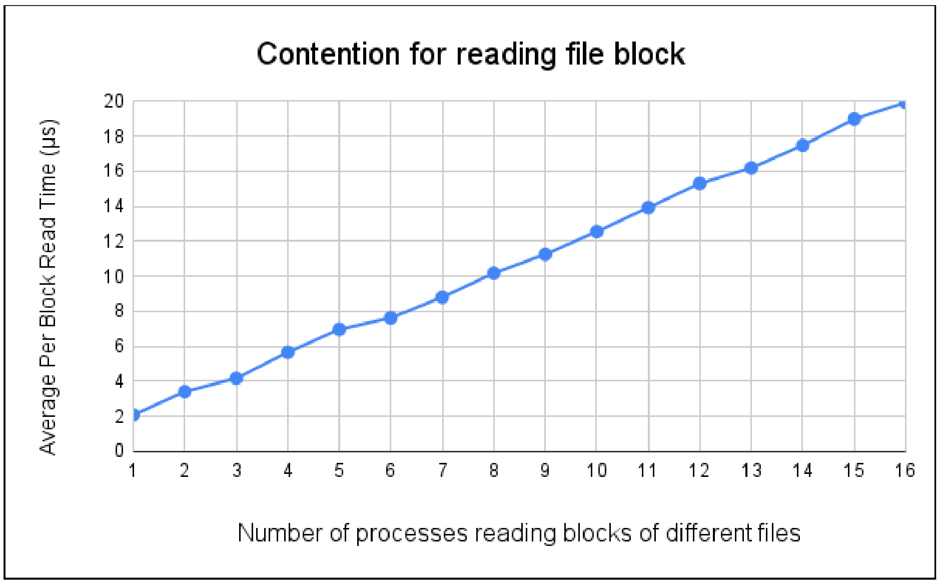
The following steps were followed to measure the file cache size of the system.

1. Created 16 files of 20 MiB size using `fallocate` command.
2. Executed a couple of instructions before starting the actual measurements so that the instruction cache is warm when we begin our measurements.
3. Cleared file cache using the following system call:
   `system("sync; echo 3 > /proc/sys/vm/drop_caches");`
4. Used `fork` system call to create “x” child processes and made the child processes read a block from the above created 20 MiB files at 1000 different offsets. Each child process used a different file for its read operation, but blocks from same offsets were read for all the child processes.
5. The start timestamp was taken before we began `fork` operation, and end timestamp was taken once all the child processes completed their read operations. The timestamps were taken using the `rdtsc` and `rdtscp` CPU cycle counter instructions. Finally, computed the average time to read a block in micro-seconds scale using the below calculations.
   ```
   AVG_PER_BLOCK_READ_TIME =
   (END_TIME - START_TIME)*(0.344)/(1000*1000)
   ```
6. Repeated Step 3, 4, and 5 for number of processes equal to 1 to 16 in increments of one.

6.4.2 Results

We had made a wild guess that average per block read time for “n” contending processes will be equal to “n” multiplied by average per block read time for one process. Using our average per block read time measurements in Section 6.2 and the above realation, we have predicted average per block read times during different contention scenarios.

Our predicted values are pretty close to the measured values for upto seven contending processed. When more than seven processes contend simultaneously for disk read, the we notice relatively larger increase in average per block read times. We observe the expected contention behavior and hence are convinced of the accuracy of the methodology used.



No. of processes	Predicted Avg. Per Block Read Time (µs)	Measured Avg. Per Block Read Time (µs)
1	1	2.088
2	2	3.414
3	3	4.182
4	4	5.663
5	5	6.973
6	6	7.629
7	7	8.809
8	8	10.184
9	9	11.260
10	10	12.556
11	11	13.926
12	12	15.306
13	13	16.193
14	14	17.479
15	15	18.989
16	16	19.929

7. Summary

CPU, Scheduling, and OS Services				
	Base Hardware Performance	**Estimated Software Overhead**	**Predicted Time**	**Measured Time**
CPU Cycle Time	0.37 ns	-	0.37 ns	0.344 ns
Read Overhead	-	1.38 ns	1.38 ns	12.76 ns
Loop Overhead	-	1.72 ns	1.72 ns	1.99 ns
Procedure Call (0 args)	-	0.344 ns	0.344 ns	0.963 ns
Procedure Call (1 args)	-	0.688 ns	0.688 ns	0.237 ns
Procedure Call (2 args)	-	1.032 ns	1.032 ns	0.392 ns
Procedure Call (3 args)	-	1.376 ns	1.376 ns	0.722 ns
Procedure Call (4 args)	-	1.720 ns	1.720 ns	0.795 ns
Procedure Call (5 args)	-	2.064 ns	2.064 ns	1.362 ns
Procedure Call (6 args)	-	2.408 ns	2.408 ns	1.620 ns
Procedure Call (7 args)	-	2.752 ns	2.752 ns	1.913 ns
System Call	-	237 ns	237 ns	451.775 ns
Process Creation	-	predicted process creation 5 times more costly than thread creation		116268.33 ns
Thread Creation	-			18852.31 ns
Process Context Switch	-	predicted process context switch 2-3 times more costly than thread context switch		1744.91 ns
Thread Context Switch	-			1655.53 ns

Memory				
	Base Hardware Performance	**Estimated Software Overhead**	**Predicted**	**Measured**
L1 Cache Access Time	1.72 ns	-	1.72 ns	0.80 ± 0.22 ns
L2 Cache Access Time	4.13 ns	-	4.13 ns	3.25 ± 1.78 ns
L3 Cache Access Time	14.45 ns	-	14.45 ns	17.69 ± 7.39 ns

Main Memory Access Time	90 ns	-	90 ns	55.61 ± 5.07 ns
RAM Read Bandwidth	17 GB/s	-	17 GB/s	14.38 ± 0.04 GB/s
RAM Write Bandwidth	8.5 GB/s	-	8.5 GB/s	7.42 ± 0.01 GB/s
Page Fault Service Time	0.417 µs	100 µs	100.417 µs	492.13 ± 1.04 µs

Network				
	Base Hardware Performance	**Estimated Software Overhead**	**Predicted**	**Measured**
Ping RTT (Local)	-	-	-	0.046 ± 0.008 ms
Ping RTT (Remote)	-	-	-	61.315 ± 36.391 ms
Application Layer RTT (Local)	0.051 ms	-	0.051 ms	0.020 ± 0.007 ms
Application Layer RTT (Remote)	67.446 ms	-	67.446 ms	59.039 ± 13.359 ms
Peak Bandwidth (Local)	59.36 Gbps	-	59.36 Gbps	27.344 ± 2.444 Gbps
Peak Bandwidth (Remote)	2.74 Mbps	-	2.74 Mbps	2.889 ± 0.066 Mbps
Connection Setup Overhead (Local)	-	0.020 ms	0.020 ms	0.024 ± 0.001 ms
Connection Setup Overhead (Remote)	-	59.039 ms	59.039 ms	55.174 ± 5.151 ms
Connection Teardown Overhead (Local)	-	0.020 ms	0.020 ms	0.007 ± 0.0001 ms
Connection Teardown Overhead (Remote)	-	59.039 ms	59.039 ms	0.056 ± 0.001 ms

File System				
	Base Hardware Performance	**Estimated Software Overhead**	**Predicted**	**Measured**
File Cache Size	-	-	14.6 GiB	14.5 GiB
Local File Read Time (Sequential)				
128 KB	0.069 ms	0.004 ms	0.073 ms	0.034 ms

256 KB	0.139 ms	0.007 ms	0.146 ms	0.063 ms
512 KB	0.278 ms	0.014 ms	0.292 ms	0.114 ms
1 MB	0.556 ms	0.028 ms	0.584 ms	0.227 ms
2 MB	1.111 ms	0.056 ms	1.167 ms	0.446 ms
4 MB	2.222 ms	0.112 ms	2.334 ms	0.910 ms
8 MB	4.444 ms	0.225 ms	4.669 ms	1.769 ms
16 MB	8.889 ms	0.450 ms	9.339 ms	3.454 ms
32 MB	17.778 ms	0.899 ms	18.677 ms	6.837 ms
64 MB	35.556 ms	1.798 ms	37.354 ms	13.646 ms
128 MB	71.111 ms	3.596 ms	74.707 ms	27.311 ms
256 MB	142.222 ms	7.193 ms	149.415 ms	54.348 ms
512 MB	284.444 ms	14.385 ms	298.829 ms	108.852 ms
1 GB	568.889 ms	28.770 ms	597.659 ms	219.001 ms
2 GB	1137.778 ms	57.54 ms	1195.319 ms	438.498 ms
4 GB	2275.556 ms	115.081 ms	2390.637 ms	875.432 ms
8 GB	4551.111 ms	230.162 ms	4781.273 ms	1769.539 ms
16 GB	9102.222 ms	460.325 ms	9562.547 ms	3511.498 ms
Local File Read Time (Random)				
128 KB	0.206 ms	0.002 ms	0.208 ms	0.036 ms
256 KB	0.413 ms	0.005 ms	0.418 ms	0.064 ms
512 KB	0.826 ms	0.012 ms	0.838 ms	0.122 ms
1 MB	1.652 ms	0.026 ms	1.678 ms	0.236 ms
2 MB	3.303 ms	0.054 ms	3.357 ms	0.468 ms
4 MB	6.606 ms	0.111 ms	6.717 ms	0.918 ms
8 MB	13.213 ms	0.223 ms	13.436 ms	1.806 ms
16 MB	26.426 ms	0.448 ms	26.874 ms	3.514 ms
32 MB	52.852 ms	0.897 ms	53.749 ms	6.986 ms

64 MB	105.703 ms	1.796 ms	107.499 ms	13.912 ms
128 MB	211.406 ms	3.595 ms	215.001 ms	27.796 ms
256 MB	422.812 ms	7.191 ms	430.003 ms	55.432 ms
512 MB	845.624 ms	14.383 ms	860.007 ms	111.013 ms
1 GB	1691.248 ms	28.769 ms	1720.017 ms	222.874 ms
2 GB	3382.496 ms	57.539 ms	3440.035 ms	445.473 ms
4 GB	6764.992 ms	115.079 ms	6880.071 ms	893.023 ms
8 GB	13529.985 ms	230.161 ms	13760.146 ms	1788.364 ms
16 GB	27059.970 ms	460.323 ms	27520.293 ms	3581.520 ms
Remote File Read Time (Sequential)				
128 KB	0.316 sec	-	0.316 sec	0.232 sec
256 KB	0.631 sec	-	0.631 sec	0.445 sec
512 KB	1.262 sec	-	1.262 sec	0.939 sec
1 MB	2.525 sec	-	2.525 sec	1.789 sec
2 MB	5.050 sec	-	5.050 sec	3.813 sec
4 MB	10.100 sec	-	10.100 sec	7.935 sec
8 MB	20.200 sec	-	20.200 sec	13.836 sec
16 MB	40.399 sec	-	40.399 sec	29.383 sec
32 MB	80.798 sec	-	80.798 sec	59.495 sec
Remote File Read Time (Random)				
128 KB	0.316 sec	-	0.316 sec	0.232 sec
256 KB	0.631 sec	-	0.631 sec	0.435 sec
512 KB	1.262 sec	-	1.262 sec	1.067 sec
1 MB	2.525 sec	-	2.525 sec	2.158 sec
2 MB	5.050 sec	-	5.050 sec	4.220 sec
4 MB	10.100 sec	-	10.100 sec	7.737 sec
8 MB	20.200 sec	-	20.200 sec	15.178 sec

16 MB	40.399 sec	-	40.399 sec	30.635 sec
32 MB	80.798 sec	-	80.798 sec	60.757 sec
Average Per Block Read Time During Contention				
1 Process	-	1 µs	1 µs	2.088 µs
2 Processes	-	2 µs	2 µs	3.414 µs
3 Processes	-	3 µs	3 µs	4.182 µs
4 Processes	-	4 µs	4 µs	5.663 µs
5 Processes	-	5 µs	5 µs	6.973 µs
6 Processes	-	6 µs	6 µs	7.629 µs
7 Processes	-	7 µs	7 µs	8.809 µs
8 Processes	-	8 µs	8 µs	10.184 µs
9 Processes	-	9 µs	9 µs	11.260 µs
10 Processes	-	10 µs	10 µs	12.556 µs
11 Processes	-	11 µs	11 µs	13.926 µs
12 Processes	-	12 µs	12 µs	15.306 µs
13 Processes	-	13 µs	13 µs	16.193 µs
14 Processes	-	14 µs	14 µs	17.479 µs
15 Processes	-	15 µs	15 µs	18.989 µs
16 Processes	-	16 µs	16 µs	19.929 µs

References

[1] Intel® Core™ i7-7500U Processor Specifications
https://ark.intel.com/content/www/us/en/ark/products/95451/intel-core-i77500u-processor-4m-cache-up-to-3-50-ghz.html

[2] Intel® Turbo Boost Technology
https://www.intel.com/content/www/us/en/gaming/resources/turbo-boost.html

[3] Intel® Solid State Drive 600p Series Specifications
https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/ssd-600p-brief.html

[4] Intel® Dual Band Wireless-AC 8265 Specifications
https://ark.intel.com/content/www/us/en/ark/products/94150/intel-dual-band-wirelessac-8265.html

[5] Smarter CPU Testing – How to Benchmark Kaby Lake & Haswell Memory Latency, Andriy Berestovskyy
https://www.nexthink.com/blog/smarter-cpu-testing-kaby-lake-haswell-memory/

[6] ICMP Ping Showing Latency for Host Inbound and Outbound traffic, Sept 2019.
https://kb.juniper.net/InfoCenter/index?page=content&id=KB28157&cat=MX_SERIES&actp=LIST

[7] The 5 best tools to measure network latency, Jan 2021.
https://www.kadiska.com/blog-measure-network-latency/

[8] Path of a Packet in Linux Network Stack, Ashwin Kumar Chimata, July 11, 2005.
https://www.cs.dartmouth.edu/~sergey/netreads/path-of-packet/Network_stack.pdf