System Measurement Project

Jiale Xu, Hui Zhi, Zhichao Liu

Date: March 16, 2023

1. Introduction

In working with systems, Operating Systems connect the software/user level with the hardware/machine level, in order to better utilize the hardware resources to perform different tasks. There are several Operating Systems with different features and characteristics to reach the performance goal of utilizing resources. In our project, we want to use different tools to test and determine the performance of our chosen Operating System. After getting the results, we will try to use our abstract knowledge to analyze their performance and provide some conclusions behind the data. In particular, we want to test the following Operating System aspects: CPU Scheduling, Memory, Network, and File System. For each aspect, we will develop testing for several smaller features, which will be described in detail later.

In our project, we plan to use the tools provided by Intel Benchmark to test the main features of our Operating System. The counted time will mainly be measured in terms of CPU cycles using rdtsc(). For the purpose of clear data representation, we may convert cycles to milliseconds (ms), microseconds (μs) or nanoseconds (ns) based on the cycle value we get. For the environment, we restrict it to single-core (the original is dual-core). For language support, we plan to use mainly C/C++ to develop API for testing hardware performance.

In the following report, section 2 will describe our testing machine's configurations, section 3 is for CPU testing, section 4 is for memory testing, section 5 is for network testing, and section 6 is for file system testing. Each will include methodologies for our experiment, followed by our prediction and the result of our experiments. A discussion is added to analyze our experimental results under our background knowledge. Finally, section 7 will summarize our results in one table.

We have three people working as a group. The whole project is estimated for an 8-week period, with 5-8 hours each week.

2. Machine Description

Processor Model: $3.1 GHz,\ Dual - core,\ Intel\ Core\ i5$
CPU cycle times: $\frac{1}{3.1G} = 0.322\ ns$
CPU cache size: $L1\ has\ 64KB,\ L2\ has\ 256KB\ for\ each\ core,\ L3\ has\ 5MB$
DRAM type: $LPDDR3$
DRAM clock: $2133\ MHz$
DRAM capacity: $16GB$
Memory Bus bandwidth: 64 bits
I/O bus type: Thunderbolt 3 / USB 3.0 (Apple T1 controller)
I/O bandwidth: Up to 40 Gbps for Thunderbolt 3, 10 Gbps for USB 3.0
Disk type: SSD (PCI-Express)
Disk Capacity: $512GB$
Disk Transfer read: $639\ MB/s\ for\ 4K$
Disk Transfer write: $437\ MB/s\ for\ 4K$
Network type: Wi-Fi (0x14E4, 0x171) 802.11
Network Card bandwidth: Up to 1300 Mbps
Operating System: $OS\ X12.4$ ($macOS\ Monterey$)

3. CPU Operations

3.1 Measurement overhead

3.1.1 Overhead of reading time

Methodology:

Intel CPUs have a time stamp counter to record each cycle on the CPU. According to Paoloni[1], beginning with the Intel Pentium processor, devices contain a timestamp register to store timestamp value which can be read by calling rdtsc() assembly functions. We recognize the existence of preemption and hard interrupts that might interfere with the CPU performance measurement. We decide to run these experiments at the user level and ignore the effects of these issues.

To measure the read instruction overhead, we call the rdtsc() function twice. Two rdtsc() function calls are consecutive so the returned timestamp values, marked as start and end, indicate the start time and end time of a reading process. To determine the cycle time, we conduct the experiment to sum up 10000 times of two consecutive rdtsc() and take the average to obtain a more accurate result.

Prediction:

On modern Intel processors, the RDTSC instruction typically takes around 20-30 clock cycles to execute, depending on the exact model and clock speed. However, this is just the cost of executing the instruction itself and does not account for any overhead or delays caused by other factors, such as context switches, cache misses, or memory access latencies. To get a more accurate estimate of the number of cycles required for a specific use case, you can benchmark the code using a performance profiler or hardware performance counters. These tools can measure the actual number of cycles and other performance metrics for specific code segments and help identify bottlenecks and optimization opportunities. We guess the overhead of reading time to be about 40 clock cycles.

Results:

Reading time overhead (cycles)

122.2485	116.5212	106.0219	105.5678	105.9807	105.2017	105.3872	106.8727	105.3702	104.9513

Reading time overhead (microseconds)

0.0394	0.0376	0.0342	0.0341	0.0342	0.0339	0.034	0.0345	0.034	0.0339

Discussion:

From the measurement result, reading time overhead is mostly in the range of (104, 107) cycles. We can see the first two values are out of this range, and the following results are well bounded in the range. We first guessed these two values are caused by the cache initialization process. After repeating the test procedure ten times, the results indicate that the first one or two values are always larger than the remaining values, and the following values tend to be in the same range (104, 110). Thus we conclude that the read overhead is approximately 107 cycles by taking an average of the maximum and minimum, that is 0.0345 microseconds in this system. The result is larger than our expectation of around 40 clock cycles. The main cause could be the internal implementation of macOS doing timestamp register access because it’s quite different from the Linux system. In general, the measurement is pretty accurate given it only differs by approximately 10 nanoseconds. And it’s also an efficient operation given its great efficiency in using assembly codes instead of calling timestamp-access API functions.

3.1.2 Overhead of using a loop to measure iterations of an operation

Methodology:

In this part, we will measure the iterations of an operation using a loop. The test starts with storing the timestamp counter at the beginning of the loop. Then execute the loop for a fixed amount of times, iteration_time, with no instructions inside the loop, and record the timestamp counter after the iterations. Now we have received the number of cycles, total_cycle, for the whole loop, and total_cycle / iteration_time gives the average loop overhead count. To examine any abnormal result and reduce such occurrences, we perform this test 10 times to gain a better understanding, using 10000 as the iteration_time.

Prediction:

To obtain an estimation of loop overhead, we want to analyze the assembly instructions within a loop. We think for each iteration, the CPU executes the following assembly instructions:

1. Load the conditional variable v1 to the register r1.
2. Load the conditional variable v2 to the register r2.
3. Perform the condition check with v1 and v2.
4. Update v1’s value.
5. Store v1’s value to the conditional variable.

So we would anticipate the CPU loop overhead to be around 5 cycles.

Result:

Loop overhead (cycles)

4.8817	4.8008	4.7953	4.8109	4.8027	5.2297	5.1451	4.9025	4.8767	5.176

Loop overhead (nanoseconds)

1.5747	1.5486	1.5469	1.5519	1.5493	1.687	1.6597	1.5814	1.5731	1.6697

Discussion:

According to the loop overhead result, the CPU generates around 5 cycles for each iteration of a loop. So the measured performance of the loop operation is close to our predicted performance. The experiment does not include any preparation for handling any system interruptions. We use 10000 iterations for each loop to prevent the result from being largely affected by system interruptions that cannot be recorded.

3.2 Procedure call overhead

Methodology:

We will define 8 functions taking different arguments from 0 to 7, then calling each of them to test their overhead in terms of cycles. To access the overhead for each procedure call on function, we use rdtsc() to measure the time before and after calling the function and calculate the difference as the overhead. To minimize the error occurring in the experiment, we will run each function 1000,000 times and take the average as the overhead result. To ensure accuracy, we perform the above procedures in 10 trials and record their mean and standard deviation to check whether there are any outliers or stay consistent.

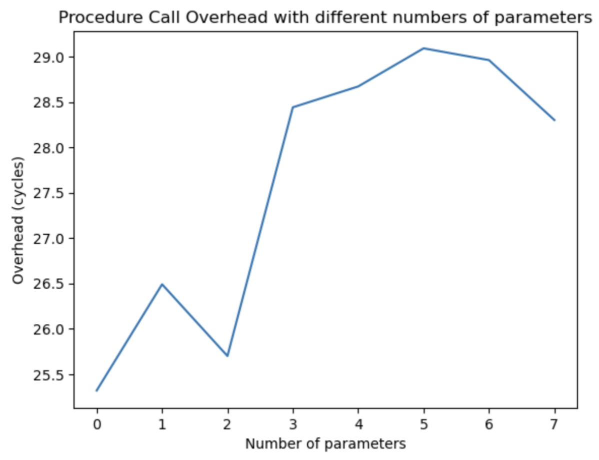
Prediction:

Our prediction is that the increasing parameter will increase the overhead of the procedure calls. The reason is that in terms of the X86 machine operations, before calling the function, the system will push each parameter into the stack. Thus the increasing parameter will increase the cycles for the machine to push parameters. We predict it will increase in 1 cycle. However, compared to the overall instructions done in the procedure call, the increasing parameter may not cause a huge increase in overhead.

Result:

The unit is in CPU cycles:

Parameter number	0	1	2	3	4	5	6	7
Overhead (mean)	25.32	26.49	25.7	28.44	28.67	29.09	28.96	28.3
Standard Deviation	1.84	2.75	2.86	4.85	5.11	6.40	4.84	3.44



Discussion:

In the draft experiment, we only ran the function 1000 times and then calculated the average as overhead. However, the overheads of different arguments stay very close and it seems like the number of arguments doesn't really affect the overhead time. We are suggested by TA to increase the number of iterations.

Thus, in our second experiment, we set the iteration number as 1000,000, and the result, as shown above, truly shows the trend of increasing overhead when the number of arguments increases. However, it also faces a downward trend (from 1 to 2, and 6 to 7) and the std is also not good, indicating that the results are not so consistent. We think the reason is that pushing new arguments to stack only occupies a small portion of procedure call instructions, so increasing parameters may not hugely influence procedure call overhead. As a result, the cost led by adding arguments sometimes is hidden by other dominating factors in procedure calls.

3.3 System call overhead

Methodology:

We would like to consider the following system calls: open, read, write, close, wait, exec, fork, exit, and kill, which are popular system calls in macOS. We choose to find the cost of a minimal system call open() by recording the start and end time from two rdtsc() function calls, and performing an open() function call from the "fcntl.h" file between two rdtsc(). Similarly, one test includes 10000 times repeating the previous procedure. And we repeat this test 10 times and select the minimum system call cost.

Prediction:

On a modern x86-based processor, a system call typically involves a context switch from user mode to kernel mode, where the operating system handles the request and performs any necessary operations before returning control back to user mode. This context switch involves a number of operations, such as saving and restoring the processor state, updating the memory management unit, and performing any necessary security checks and resource allocation.

The overhead of a system call can be measured using a benchmarking tool such as perf or dtrace, which can monitor the number of cycles or instructions executed during a system call. Based on benchmarks of similar systems and assumptions, the overhead of a system call on the device can be estimated to be in the range of tens to hundreds of clock cycles, depending on the specific system call and the system workload. We guess it to be around 70 clock cycles.

Result:

open() system call cost (cycles)

12268.28	11778.17	11941.38	13308.33	11288.47	11205.13	11188.44	11217.06	11430.43	11387.74

open() system call cost (microseconds)

3.9575	3.7994	3.8521	4.293	3.6414	3.6146	3.6092	3.6184	3.6872	3.6735

The cost of a minimal system call using open() is 3.6146 microseconds according to our calculation.

getppid()

cycles	2054.84	2121.66	1968.35	1958.72	1959.35	1990.63	1954.99	1972.45	2179.24	1967.88
microseconds	0.6629	0.6844	0.635	0.6318	0.632	0.6421	0.6306	0.6363	0.703	0.6348

The cost of a minimal system call using getppid() is 0.6306 microseconds.

getpid()

cycles	177.46	117.04	110.58	110.5	108.84	112.59	109.14	112.24	109.12	115.01
microseconds	0.0572	0.0378	0.0357	0.0356	0.0351	0.0363	0.0352	0.0362	0.0352	0.0371

The cost of a minimal system call using getppid() is 0.0351 microseconds.

Discussion:

Generally speaking, the cost of procedure calls is relatively cheaper than system calls. However, some idempotent system calls like getpid() are cached by the operating system, which causes a much smaller cost. And our test result clearly shows that getpid() cost is much smaller than getppid() and open() system calls. So the cost of a minimal system call is 0.0351 microseconds.

The procedure call overhead test is still in the process, and we will present a comparison between the cost of this minimal system call and the procedure call later.

We have chosen open(), getppid(), and getpid() as the system calls to be tested for a minimal cost. However, the cost of a minimal system call could be computed from any of the system calls. More tests will be performed and updated later.

3.4 Task creation time

Methodology:

For the thread creation benchmark, we utilized the system call “pthread_create” from the C standard library. Specifically, we execute the parent process, and then utilize the pthread_create with an empty function (does nothing within the function and returns immediately).

For the process creation benchmark, we utilized the fork() system call, which created and executed a child process that is a copy of the parent process. Similarly, as soon as the program detects that the current process is a child process, it will terminate immediately. The reason we choose fork() rather than exec() is because of the new file loading overhead that will be rendered by executing a file that is not in the memory, as mentioned by hbench[6] paper.

The granularity of this benchmark is a single system call (pthread_create or fork), and time stamps are collected before and after each system call. We will do 2000 trials to get more deterministic results as recommended by the lmbench[4] paper.

Prediction:

The creation of a single process involves a lot of operating system workload, including initializing a new address space, copying the necessary data from the parent process, copying the page table, etc. Such redundant operations would definitely induce a large overhead. Thus, we estimated the process creation overhead to be about milliseconds.

Compared to the process creation, the thread creation overhead should be much smaller, since the child thread would share the same address space as the parent thread, and the related workload would be far less than creating a process. Therefore, we estimated that the thread creation time would be much smaller than the process creation time, at the level of microseconds.

Result:

With 2000 trials, we get the following results for the thread creation and kernel creation:

Thread	27 µs
Process	700 µs

Discussion:

From the result, we can see that the process creation time is about 30 times more than the thread creation time, which aligns with our expectation. Before the experiment, we suspected that the stochastic behavior of the scheduler may render some effects on the process/thread creation time measurement benchmark. For example, we recorded the time stamp before and after we fork a new child process, but we could not determine if the child process got the CPU or the main process continuously went on. However, after we got the result, we found almost no outliers indicating such a situation. After analyzing the system behavior, we concluded that the execution time of an empty child process as well as the context switching time would be too small to affect our measurement of task creation time.

3.5 Context switch time

Methodology:

We followed the benchmark described in the lmbench[4] paper to measure the context switch time between processes or threads. We utilized a blocked pipe to create a communication channel between child and parent processes (threads), and this blocked pipe would also enforce a context switch from parent to child. Since the usage of pipes to communicate between processes would also induce extra overheads, we first measured the pure overhead of using pipes. We created a pipe within one process to do read and write operations to calculate the overhead of one-time pipe token passing(one read and one write).

To calculate process context switch time, we first created a parent process, then called fork() to spawn a new child process. The parent process called the read from the pipe to wait for the message from the child process. With the blocking pipe, the parent would wait on the read instruction and the child would get the chance to execute. The child process then collected a time stamp and passed it back to the parent process. After the parent process received a timestamp from the child process, the parent would record another timestamp. The time between two timestamp collections includes one write-to-pipe operation, one read-from-pipe operation, and the context switch time from child to parent. Thus, the pure context switch time would be the gap between two timestamps minus the pipe read and write overhead. A similar technique is applied to thread context switch time measurement, with the fork operation changing to pthread_create() function call.

We repeated the process 2000 times to get more generalized results.

Prediction:

Before the experiment, we expected the context switch time would be much more than the procedure call since it involves much more work in the operating system kernel mode, for example, storing the register state and restoring the register state, while it has much fewer overheads than the task creation time since context switch does not have as much as copy workload as task creation. With the information from online[5], we predicted that the process context switch overhead would be 12.5 microseconds and kernel thread overhead would be about 1 microsecond.

Result:

With 2000 trials, we get the following results for the thread context switch and process context switch:

Thread	5.8 μs
Process	25 μs

Discussion:

Before the experiment, the result was much larger than the expectation, and we thought that it might come from other system overheads that we didn't factor out. We have tried several different measurement methods to try to develop the reason behind the result.

The first alternative method we tried was to set timestamps only in the parent process, before the fork() function call. And we recorded the end time after the parent finished reading from the pipe. Such measurement methodology would have more overheads than the methodology we stated in the methodology section. Specifically, it includes two context switches, one process creation, and a read-write pipe operation. We got the result with this benchmark, and it induced almost the same result as the method stated in the methodology.

We then assumed that the inaccuracy of our results came from the inaccurate measurement of pipe overhead calculation. So in our third experiment, we did not use the pipe to enforce the context switch, but utilized pthread_join, and wait, to enforce the context switch. The first timestamp is set right after the fork(), and before the wait() function call and the second timestamp is set after the wait. We think this would be a pure overhead of context switching. However, such methodology induced even larger context-switching overhead. Thus, we concluded that the reason behind this number might be some hardware/machine problem that we could not control with our current methodology.

4. Memory Operations

4.1 RAM access time

Methodology:

To evaluate the RAM access time, we applied the method in the lmbench[4] paper, which is back-to-back-load latency. As defined in the lmbench[4] paper, back-to-back-load latency calculates each load's time, assuming that the instructions before and after are also cache-missing loads.

Intel i5 Core has three-level CPU caches (L1, L2, and L3). L1 cache is 64kb in total for each core, 32kb for the instruction cache, and 32kb for the data cache. L2 cache is 256kb, and L3 is 5MB. Thus to evaluate the latency for main memory as well as different levels of CPU caches, we will test the machine with references to data arrays with different sizes. One thing that has to be factored out is the prefetching effect from the large cache lines. We also applied the method from lmbench[4], introducing the usage of stride.

Thus, our benchmark calculation would be generalized as follows:

1. fix a stride size d, fix an array size n, and malloc an empty array with size n.
2. Loop through the array, the entry value at index i would be assigned at the index value of the entry that is d away from the current index i, i.e.: array[i] = (i + d)%n
3. Walk through the array for the second time, following pointer pt, where pt = array[pt], and set a timer before and after the loop, in which the program will do 100000 array references. We will get the average time for each reference.

In the experiment, we tested the machine with the following parameters:
stride size(bytes): $4 \times 2^i,\ i \in [0,\ 9]$; array size(bytes): $4 \times 2^i,\ i \in [8,\ 17]$

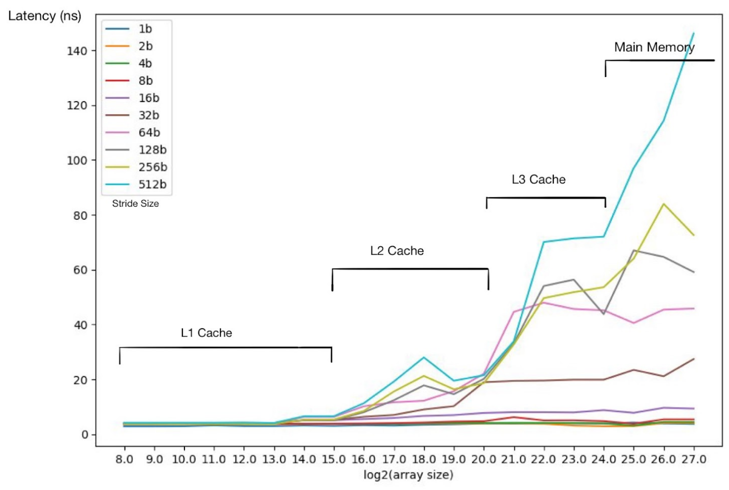
Prediction:

Intel has provided data for the RAM access latency with each level of caches[9], which provides the average performance of a 16 cores machine in 2016. With that as the baseline, before doing the experiment we have made the following estimation for the latency.

Type	Accessing Time (nanoseconds)
L1	2
L2	8
L3	30
Main Memory	60

Result:

The graph of latency with respect to the array size is shown as follows:



From the graph, we deduce 128 bytes as a possible cache line size since it is the smallest stride size that has a great increase of latency when the array size exceeds L3 cache. so we will utilize the result when stride size = 128 bytes as our data for analysis. The following graph is our experiment result for memory access latency.

Type	Accessing Time (ns)
L1	3.919
L2	12.421
L3	43.804
Main Memory	67.04

Discussion:

Our results for latency are comparatively longer than the normal CPU, and we estimate the problem might be that practically, there are too many background processes running that affect the memory reading time. Also, there might be some other reasons that influence the accuracy of the result, such as instruction loading time, which hasn't been factored out in our benchmark.

4.2 RAM bandwidth

Methodology:

To measure the bandwidth for transferring memory, we plan to first construct the transfer data (read/write) in bytes, and then divide it by the transfer time (measured in the experiment). And we split it into two tasks: reading transfer and writing transfer. According to the Imbench paper's approach, "Memory reading is measured by an unrolled loop that sums up a series of integers" (5), and "Memory writing is measured by an unrolled loop that stores a value into an integer (typically a 4-byte integer) and then increments the pointe" (5), we plan to allocate an array of integers with 64MB as memory reference.
For memory writing experiments, as the Imbench paper states, we simply use an unrolled loop (with 32 offsets) to store an integer value (like 1) in each index of an array. After we perform memory writing approaches on 64MB arrays, time consumption is recorded in terms of cycles using rdtsc(). The bandwidth for writing memory is simply calculated by the size of the integer array (64MB) divided by the recorded time (translated from cycles to seconds).
For memory reading experiments, we will reuse the allocated array which is just filled by the memory writing, and perform memory reading for every integer index of the array. Using the Imbench approach, we will use an unrolled loop (with 32 offsets) to sum up every integer in a 64MB array, and calculate the bandwidth as 64MB memory divided by measured consumption time (translate from cycles to seconds).

Prediction:

In our prediction, the memory bandwidth is calculated by Clock Frequency * Memory bus width (in bytes) * data transfer rate. According to the machine description, our memory's frequency is 2133mhz, with 2x data transfer rate and $64\ bits / 8 = 8\ bytes$ Memory bus bytes. As the Imbench paper said: "the processor cost of each memory operation is approximately the same as the cost in the read case", so we expect both reading and writing to have the same memory bandwidth. Thus in our prediction, the memory bandwidth for both reading and writing should be $2133mhz \times 2 \times 8 = 34128\ Mb/s = 34.13\ GB/s$. Since there will be an efficiency loss for software overhead in practice compared to theoretical results, we predict there will be like 50% efficiency loss in software, resulting in 17 GB/s for bandwidth.

Result:

	Memory reading bandwidth	Memory writing bandwidth
Trial 1	8.941 GB/s	10.553 GB/s
Trial 2	9.056 GB/s	10.561 GB/s
Trial 3	8.815 GB/s	10.578 GB/s
Trial 4	8.912 GB/s	10.521 GB/2
Trial 5	9.23 GB/s	10.550 GB/2

Mean	8.98 GB/s	10.55 GB/s
Standard Deviation	0.13	0.02

Discussion:

After testing for both reading and writing memory bandwidth, the results are much lower than our prediction, although the small std indicates the accuracy and consistency of our measurement. I think there are many factors that prevent our experiment from getting the maximum bandwidth, like many other kernel applications will occupy resources when executing the experiment, and our machine is too old (5 years from purchase) which will cause further hardware efficiency loss. The total overhead is around 75% compared to our prediction of 50%. In addition, my unrolled loops method is only with 32 offsets, which may be optimized by using some algorithms that perform fully unrolled loops.

4.3 Page fault service time

Methodology:

A page fault occurs when a process currently executing tries to access a page that is not in the memory. Considering the three types of page faults, minor/major/invalid page faults, we want to focus on the major page fault in this experiment. A major page fault is a mechanism used by the OS to increase the amount of available memory on demand. It occurs when the page being accessed is not in the main memory but is still a valid address.

To measure the time for faulting a page from the disk, we create a large file with a size of 10MB and use mmap() to generate the mapping between the memory and the disk. We use a stride of 40KB (10 pages) and access the memory-mapped file 100 times. With this being a test, we repeat this test 10 times to receive an average result number. Notice that before each test, the disk buffers should be clear in order to get page fault, the purge instruction is executed before each test.

Prediction:

The whole process is mainly disk reading time and the OS overhead. The disk reading time can be estimated by the page size divided by the write rate, which is approximately 0.009 ms. OS overhead might include the context switch and syscall. From the previous experiment results, the approximation is 0.15 ms.

Result:
Measured average page fault service time = 94157.5 * 0.322 ns = 0.303 ms

Discussion:

The test result is 0.303 ms, which is less than the estimation result. This comparison result might be caused by other overheads out of our current consideration.

5 Network

5.1 Round trip time

Methodology:

To get a more accurate estimation of the application-level RTT with the time to perform a ping using the ping command, we would like to conduct the following tests. Our program uses the TCP client-server model to test round trip time, and we use the client to perform the round trip timing which consists of one send() and one recv(). We established a C server that binds to a certain address and port, and serves in a loop reading incoming requests, then sends responses back to the client. We repeat this test for 100 iterations and take the average of the results. To get the RTT local ping, we use a local server running in localhost and command ping 127.0.0.1 in the terminal and observe the output latency of RTT. The PING command will automatically generate the average RTT for us. For the remote tests, we then perform the application requests to send application layer tcp packets to the remote server on AWS ec2, and we try to ping the same remote IP address for contrasts.

Prediction:

The round trip time (RTT) is a network metric that represents the time it takes for a packet to travel from a sender to a receiver and back again. The actual RTT on a MacBook Pro 2015 will depend on various factors, including the network conditions, the distance between the sender and the receiver, the number of routers involved, and the processing speed of the devices.

In general, on our device, we could expect the RTT to range from a few milliseconds to several hundred milliseconds, depending on the above factors. If the network is fast and the devices are located nearby, the RTT could be as low as 1-10 ms. If the network is slow or congested and the devices are located far apart, the RTT could be several hundred ms. According to the daily experience of our local wireless network UCSD-PROTECTED and graduate housing family residents, the ping is about 40 ms most of the time and has unstable connections which give 100 - 140 ms ping.

Result:

RTT Local Server(ms)	RTT Local Ping(ms)	RTT Remote Server(ms)	RTT Remote Ping
0.5001	0.127	69.7	16.8

Discussion:

The round trip time for the loopback interface is very low, typically in the range of a few microseconds. This is because the loopback interface involves sending and receiving data within the same machine, and does not involve actual network communication. The round trip time for the remote interface is higher than that of the loopback interface, typically in the range of a few milliseconds. This is because the actual network communication involves more overhead compared to the loopback interface.

5.2 Peak bandwidth

Methodology:

We use a similar methodology as 5.1.2, C sockets for experiments. To measure the peak bandwidth of the network, we set up a TCP client-server connection model; the client will send a large data stream to the server. And the server will keep reading the socket, and we can calculate the peak bandwidth using the following formula:

Bandwidth = BytesSent / TransferTime.

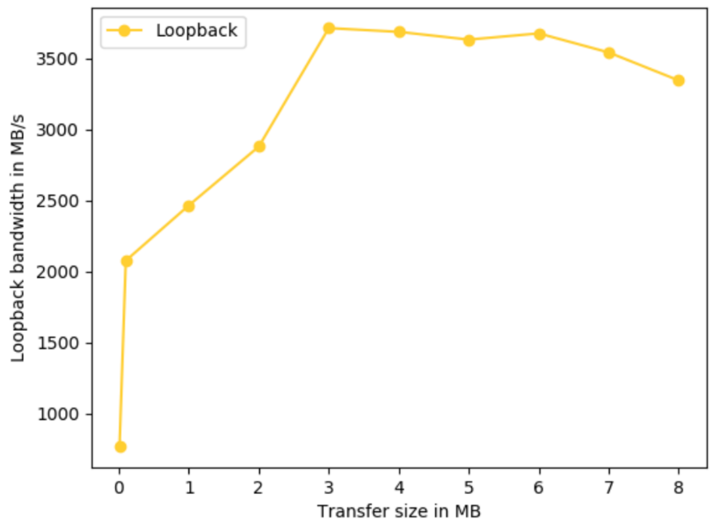
Since we want to measure the peak bandwidth, the peak bandwidth will be reached when the transfer size is close to the socket buffer size. So we tried different transfer sizes in the experiment. And the largest bandwidth is chosen as the peak bandwidth. To calculate the remote server we used another Mac OS computer with similar machine performance.

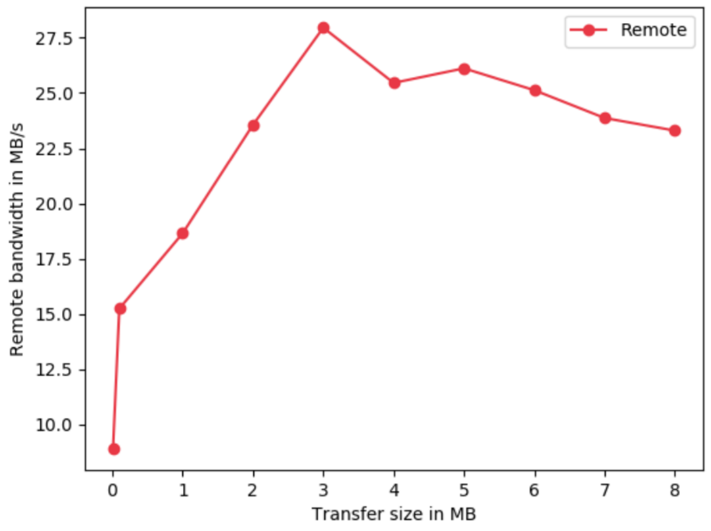
Prediction:

The peak bandwidth for the loopback interface is very high, typically in the range of several gigabits per second. This is because the loopback interface does not involve actual network communication and only involves sending and receiving data within the same machine. The peak bandwidth for the remote interface is lower than that of the loopback interface, typically in the range of several hundred megabits per second. This is because the actual network communication involves more overhead compared to the loopback interface, and is also affected by factors such as network congestion. Since the maximum socket buffer size of our computer is 3MB, and the round trip time for loopback is 0.5001ms, we can estimate the peak bandwidth for loopback is 3 MB / 0.5001 ms = 5998 MB/s. Using the same method, we can estimate the peak bandwidth for the remote to be 3 MB / 69.7 ms = 43.04 MB/s

Result:

We measure the bandwidths with different transfer sizes.





	Estimation Peak Bandwidth	Measured Peak Bandwidth
Loopback	5998 MB/s	3713.6 MB/s
Remote	43.04 MB/s	27.96 MB/s

The graphs show the peak bandwidth is reached when the transfer size is 3 MB, so for the loopback. And the exact result is shown in the table above.

Discussion:

For both round trip time and bandwidth, we may not achieve ideal hardware performance due to various overheads such as protocol overhead, CPU overhead, and network congestion. The TCP protocol has a

relatively high overhead compared to other protocols such as UDP, which can affect its performance. Network congestion can also affect the bandwidth achieved.
The measured results are quite close to our estimation, and the peak bandwidth was got when the transfer size was around 3 MB, which is exactly the socket buffer size of our computer. And we found our measured results are generally smaller than we expected, and we believe there are several reasons for this:

First, the network we used may dramatically influence our experiment. In this experiment, we used the public wifi in UCSD, UCSD-PROTECTED, and this wifi is shared by all of the students in that domain, so it is possible the bandwidth of the network we used was the bottleneck in our experiment. Second, the client and server we used were connected through TCP protocol, and since TCP will ensure reliable transmission of packets if during the experiment some packets were lost, TCP protocol will try to re-send the packets. And also, every time we send some messages through the TCP protocol, the TCP protocol will first package the messages, so the protocol itself may have lots of overhead during the experiment. And finally, we made an assumption that the socket buffer is fully used by our program during the experiment, but it is not possible, even if we had closed as many other processes as possible during the experiment, it is hard for you to make sure the socket buffer is fully used, and if the socket buffer is not fully used, our experimental results will be smaller than expected.

5.3 Connection overhead on client

Methodology:

We use a similar approach to the experiments above. For setup time, we measure the time before the call to connect and just after the end of the call to connect. For tear-down time, we measure the time before the call closes and just after the end of the call to close. We repeat this process 100 times and get the average value.

Prediction:

The connection overhead for the loopback interface is very low, typically in the range of a few microseconds. This is because the loopback interface involves sending and receiving data within the same machine, and does not involve actual network communication. The connection overhead for the remote interface is higher than that of the loopback interface, typically in the range of a few milliseconds. This is because the actual network communication involves more overhead compared to the loopback interface.

The setup of TCP needs a 3-way handshake, which is roughly 1.5 times round trip if we treat a round trip as sending two packages. According to the round trip time we measured before, the estimated local setup time should be $0.127 \times 1.5 = 0.191$ ms, and the remote setup time should be $16.8 \times 1.5 = 25.2$ ms. Similarly, the teardown of TCP needs a 4-way handshake, which is roughly 2 times round trip. According to the round trip time we measured before, the estimated local teardown time should be $0.127 \times 2 = 0.254$ ms, and the remote teardown time should be $16.8 \times 2 = 33$ ms.

Result:

Local setup time (ms)	Local tear down time	Remote setup time	Remote tear down time
0.175	0.079	22.74	0.161

Discussion:

The setup time is in line with our expectations. However, the teardown time is a lot less than our expectation. The underlying reason may be the specific implementation of close in c/c++. We assumed that the teardown is a 4-way handshake, but close may merely send a FIN package and close the connection immediately without waiting for any ACK. If it were the case, then our measurements seem reasonable.

6. File System Operations

6.1 Size of File Buffer Cache

Methodology:

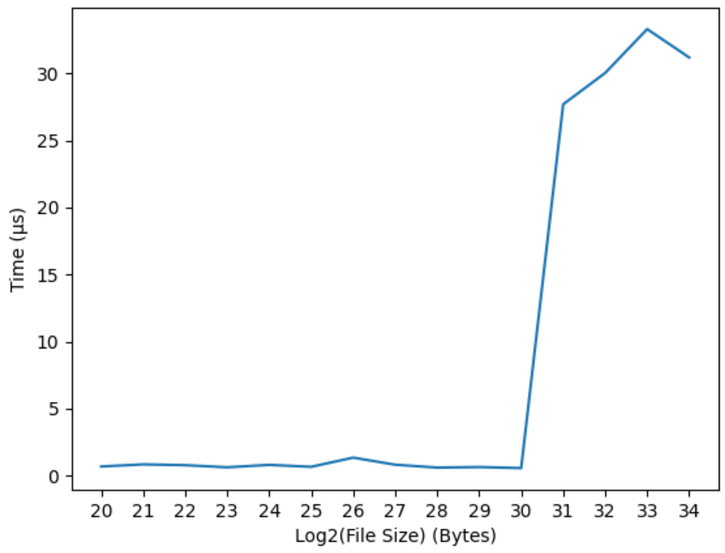
File buffer cache size is a dynamically changed variable. The operating system will increase or decrease the file buffer cache size and coordinate with the memory size for processes based on a current working set of requirements to reach the best performance. Since the file buffer cache size can reach as large as the DRAM size, we have created 15 fixed-size files from 1MB (2^{20} bytes) to 16GB(2^{34}) bytes. For each of file size, we will measure the block reading time. To measure the effect of caching, before measuring the block reading time, we will first read each file sequentially to warm up the file buffer cache. Then, we do 5 times read-through of the entire file in the unit of block. Finally, we calculate the average reading time per block.

Prediction:

Since our test machine has 16GB RAM, we estimated the file buffer cache size to be about 4G, which is a fair amount to balance the memory usage of processes and file buffer cache. Also, we predict that this number would change as the activities on the machine change.

Result:

The following is the graph of per block reading time of different file sizes. File sizes from 1MB - 1GB have almost constant latency (about 0.8 microseconds). We can see a spike when the file size reaches 2GB, which means it has exceeded the limit of file buffer cache size.



Discussion:

Before doing the measurement test, we expect the graph to be constant with a spike indicating the size of the file buffer cache. However, we also guessed that there might be no spike until the file size reaches the DRAM size because our test program actively using the file buffer cache could result in increases in the file buffer cache. From the result, we can see a vast increase in access time when the file size reaches 2G. It reaches the climax when the file size reaches 8G, but drops a little bit when the file size reaches 16G. Such behavior aligns with our expectation that the file buffer size might grow as the demand for reading increases.

6.2 File Read Time

Methodology:

Different from the previous part in which we want to measure the effect of the file buffer cache, in this section, we want to factor out the effect of caching and focus on the direct I/O time statistics. We utilize the system call fcntl() and specify the flag as NO_CACHE to close the cache effect. For this part, we use files from 1 MB to 512 MB. For the sequential access pattern, we read through each file from the start to the end. For the random access pattern, we create an array with the block number of the file and shuffle the array. Then we reach each block in the order specified by the shuffled array.

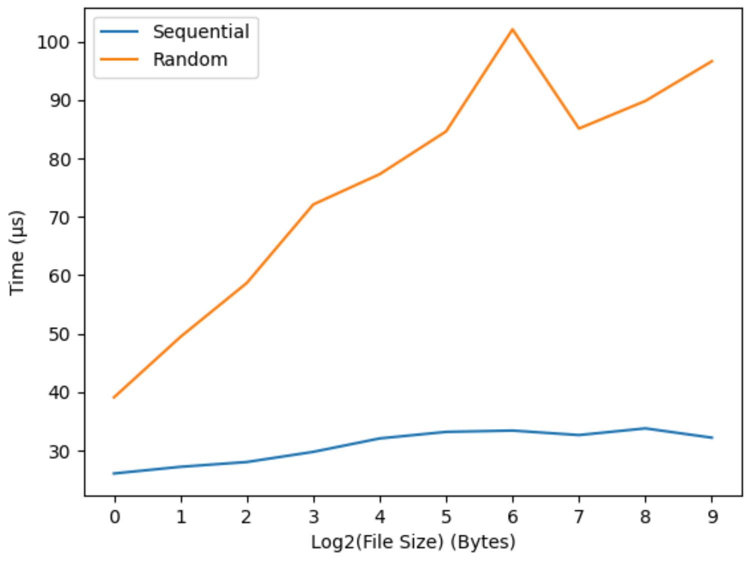
Prediction:

The traditional HDD disk would need a long time to spin and seeking for a specific position to get the desired data. According to the Fast File System paper[10], most of the file systems by then only utilized 5% of the bandwidth to do the read/write operations, and the rest of them were devoted to the overhead of disk-related operations. Therefore, it is natural to predict that random access to the filesystem would be much less efficient than the sequential access pattern since more seek operations are needed for random access. Even though our machine has a filesystem based on SSD, which might be less susceptible to disk-seeking operations, the effect will be apparent when the disk read time gets larger. Moreover, since we closed the cache, we expect the per-block reading time with different file sizes would be similar to each other.

Result:

Following are the graph and numerical results of the per-block access time of different file sizes and access methods.

Remote file size (MB)	Sequential read (µs)	Random read (µs)
1	26.09	39.5
2	27.23	49.89
4	28.24	58.81
8	29.57	72.15
16	32.45	77.84
32	33.97	84.31
64	32.16	102.85
128	33.24	85.93
256	32.63	89.03
512	33.62	96.80



Discussion:

From the result, we can observe an apparent discrepancy between the latency of sequential access and the latency of the random access. Specifically, for sequential access, the access time almost remains constant for files of different sizes. However, for the random access file, we observed an apparent increase in latency as the file size increases. We think this behavior is because of the fragmentation of the file system since larger files are less likely to be placed sequentially on the disk. For the sequential access pattern, even if a big file is fragmented into several parts, we only have to pay a small number of overheads for the several seekings between the fragments. However, for example, for random access, if a big file is fragmented into three parts (A, B, C) across the disk, and supposed that our random access pattern was (A, B, C, A, B, C….), the operating system has to jump back and forth between these blocks to get the desired data, and this would render an enormous overhead.

6.3 Remote file read time

Methodology:

To build a remote file system, we select our experiment machine as a server to store files, and choose another machine (MBP 2018) as a client to access files on the server. To accomplish that goal, we utilize the sharing functionality between Macbooks to build a sharing directory in our server machine. The connection is via WIFI. The path for the sharing directory located in the server is named "/Volumes/leo" in the client's views, which can be directly accessed using "open()" as remote files.
Since reading remote files requires a huge amount of network overhead, we only test the remote file reading up to size of 64MB. The experiment details are the same as 6.2 except we change the file path to "/Volumes/leo/filenames" after setting up the server/client model. The blocksize is still 4K, and there are 7 files with the sizes of 2^0 *to* 2^6 MB. To avoid the influence of file cache, we use the same method of fcntl(fd, F_NOCACHE, 1) to disable the file caching. The sequential and random read time for remote files are recorded in units of cycles (rdtsc()), and then divided by CPU frequency to translate to milliseconds.

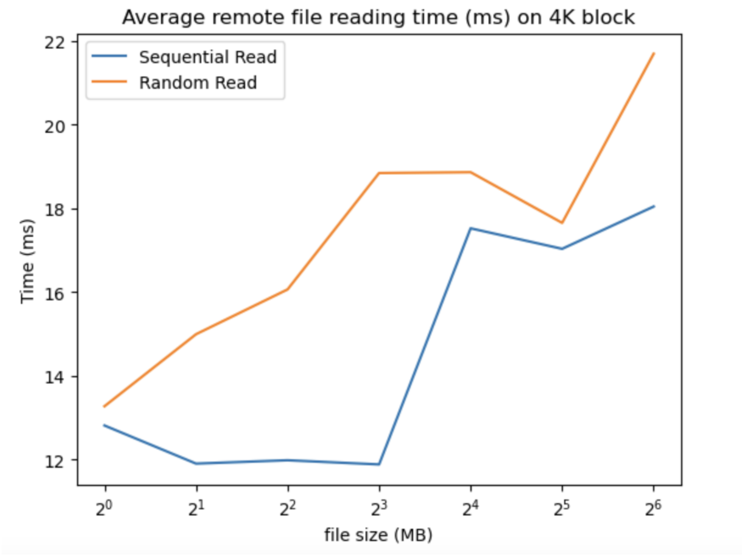
The configuration for client machine is included below:
Processor Model: $2.9 GHz,\ 6\ cores,\ Intel\ Core\ i9$
DRAM type: DRR4
DRAM clock: $2400\ MHz$
DRAM capacity: $16GB$
Disk type: SSD (PCI-Express)
Disk Capacity: $256GB$
Network type: Wi-Fi (0x14E4, 0x7BF) 802.11n
Network Card bandwidth: Up to 1300 Mbps

Prediction:
The only difference between remote file access (6.3) and local file access (6.2) is the network overhead for transferring data. According to our previous experiments on network bandwith and RTT, we predict the network overhead will be around 6 - 7 ms, which should dominate the overall time cost (compared to local read). Since we use Wifi as a remote connection, which is not so stable in transmission, we estimate 2-4 ms overhead due to network traffic. So, overall prediction is 7-9 ms for Sequential Read, and 9-12 ms for Random Read.

For random reading, we predict the overhead will be larger than sequential reading caused by the time for seeking the offsets. So the overhead for random file reading should be larger than sequential file reading. We also predict increased file size will cause the larger overhead, since the reading larger time may more likely face the network traffic, resulting in the larger remote file reading time.

Result:
We get our results in terms of cycles (using rtdsc()), which is very huge (around 35 to 60 million cycles). So we translate it into ms (milliseconds) by dividing the CPU frequency. Below is our experiment results:

Remote file size (MB)	Sequential read (ms)	Random read (ms)
1	12.81	13.27
2	11.90	14.99
4	11.98	16.06
8	11.88	18.84
16	17.52	18.86
32	17.03	17.65
64	18.04	21.69



Discussion:
From the experimental result, we can see that it is somehow close to what we predicted (but the real penalty is bigger than our expection). The network overhead dominates the main cost of file reading, compared to their performances locally (from <100us to 10+ms). But the network overhead is still little large compared to our prediction, and it is not very stable such that the graph doesn't maintain a consistent trend. We think the reason is that there may be network traffic involved during the remote file reading. We use WIFI to build the TCP connection, and during the experiment there are 4-5 people (my housemates) with around 10 devices utilizing the WIFI at the same time, which may lead to more overhead when reading large file data.
The difference between sequential and random reads is not as big as in the local file reads experiment, which is reasonable since the network penalty occupies the most of the overhead. Like when reading the 16MB and 32MB files, the costs of sequential and random reads are pretty closed. In addition, for sequential reads, we can see that after the testing for an 8MB file, the time cost increases dramatically. And random file remote read also faces an dramatic overhead increase when reading 64MB files. We think these are due to the network traffic which leads to huge network penalties.

6.4 Contention

Methodology:

To measure the file reading performance under multi-processes circumstances, we utilize the fork() function to generate a different number of processes and repeat the previous experiments on Sequential and Random Read. In our experiment, we let the process number vary from 1 to 10, and keep the 10 different files with the same size of 32MB. Each process will perform Sequential and Random read operations on specific files (like process[i] will read file[i], so we can make sure they are reading different files simultaneously in our experiment.
The block size is still 4K, and we calculate the average Sequential and Random reading time on each block among all processes. The time is measured by rtdst() with a unit of CPU cycles, and we translate it to us for a clear representation. To avoid the influence of file caching, we use the same method of fcntl(fd, F_NOCACHE, 1) to disable file caching when reading files.
To ensure the accuracy, we perform 10 trials for each Sequential and Random reading experiment with a different number of simultaneous processes. And we calculate the average reading time (mean) among 10 trials and its standard deviation, which will be shown in the result part.

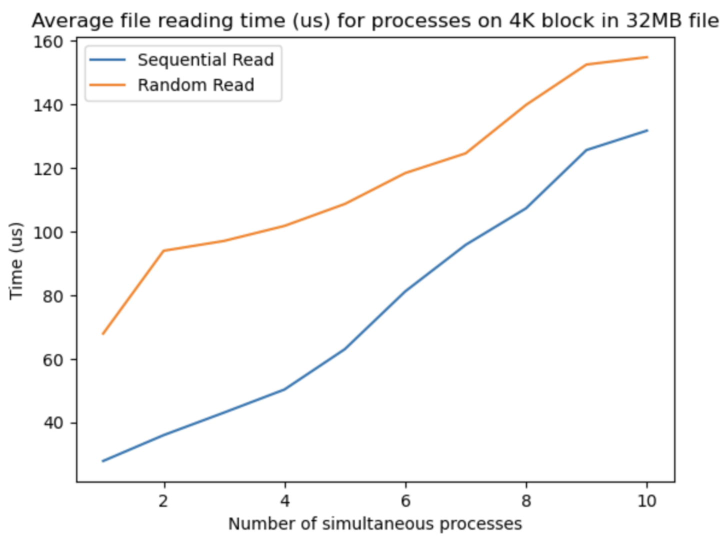
Prediction:

From our previous experiments on file reading, we can see that Random reading is much more costly than sequential reading. Under the new multi-processes circumstances, we predict this conclusion will not change, such that random reading cost is still much higher. With the increased number of simultaneous processes, the contention will occur between processes when accessing the files, resulting in increasing overhead. Thus, we predict that both Sequential and Random read overhead will increase with the

increase of process number. There are two main overheads: context switching between processes, and the flush of prefetching data after the switching process. For Sequential reads, we think its cost will increase further since both of these overheads are included. For random reads, since it will not be affected by flushing prefetching data (only by context switch), we predict its cost increase will be smaller than Sequential read's cost increase.
For base hardware performance, as our previous experiment on process context switch, it is around 25 µs. The paper "The Context-Switch Overhead Inflicted by Hardware Interrupts" states the process context switch is around 10 µs. The testing file size is 32MB, which contains 8000 4K blocks. Since there may be many context switches happening, we predict that there will be 9-16 µs overheads caused by context switches on each 4K block when increasing the process number by 1. And we estimate the overheads caused by flush of prefetching data is 3-5 µs. So, we predict that starting from a single process (30 µs for Sequential and 70 µs for Random), each additional process will increase Sequential Read cost by 12-20 µs, and increase Random Read cost by 9 - 16 µs.

Result:

Process number	Sequential Mean (µs/us)	std	Random Mean (µs/us)	std
1	28.86	0.62	70.90	2.90
2	35.98	0.82	93.94	5.99
3	43.07	1.23	97.03	5.07
4	50.32	2.81	101.75	3.03
5	63.01	3.36	108.65	3.34
6	81.17	4.76	118.36	3.47
7	95.8	5.46	124.56	2.28
8	107.3	5.20	139.80	5.32
9	125.6	5.48	152.48	3.63
10	131.7	5.73	154.79	2.08



Discussion:

From the experimental result, our general predictions are all met. Firstly, both sequential and random reading's costs increase when the process number increases. It is reasonable since more processes means more contentions and more context switches, which will definitely cause overheads. Secondly, the random reading time is still more costly than sequential reading under multi-processes circumstances. However, we can see that their differences are smaller when the process number increases to 8+ compared to a single process. The reason is explained in our estimation part: random reading doesn't involve prefetching data for each block, so the flush of prefetching data in context switches doesn't increase its overhead.
In our prediction, we estimate the overhead on reading each 4K block when adding one more process is 12 - 20 µs for Sequential read, and 9 - 16 µs for Random read. Our result shows that Sequential read has 8-16 µs overhead and Random read has 4-14 µs overhead, which is pretty close to our prediction. The experimental result is even smaller than our prediction, implying that there are less context switches during process contention than we estimated.
In addition, we collect 10 trials for each reading test and calculate their mean as result and standard deviation for referencing its consistency. From our standard deviation values, we can see the reading time among different trails is kind of consistent. And the standard deviation is relatively increasing when the process number increases, which makes sense since more contentions implies more variations.

7 Result summary

Operation	Base H/W Performance	Estimated S/W Overhead	Predicted Time	Measured Time
Measurement Overhead	Difficult to Measure	Around 40 cycles	0.03 microseconds	0.036 microseconds
Procedure call Overhead	Difficult to Measure	0-1 cycle	1 cycle increase for each additional argument	Having trend of cycle increase (1-2 cycles) but not so consistent, see Table in 3.2 for details
System call Overhead	Difficult to Measure	Around 70 cycles	0.385 microseconds	0.572 microseconds
Thread Creation Time	Difficult to Measure	-	20 µs	27 µs
Process Creation Time	Difficult to Measure	-	1 ms	700 µs
Thread Context Switch Tim	Difficult to Measure	-	1 µs	5.8 µs
Process Context Switch Time	Difficult to Measure	-	10 µs	25 µs
L1 Cache	1 ns	1 ns	2 ns	3.9 ns
L2 Cache	4 ns	4 ns	8 ns	12.4 ns
L3 Cache	10 ns	10 ns	16 ns	43.8 ns
Main Memory	20 ns	20 ns	32 ns	67 ns
RAM Bandwidth	$34.13\ GB/s$ for both read and write	-50%	17 GB/s for both read and write	8.98 GB/s for read, 10.55 GB for write
Page Fault service Time	-	0.009 microseconds	0.15 microseconds	0.303 µs
Round trip Time	-	-	RTT Remote Server: 100 microseconds	RTT Local Server: 0.5001 µs; RTT Local Ping: 0.127 µs;

			RTT Remote Ping: 20 µs	RTT Remote Server: 69.7 µs; RTT Remote Ping: 16.8 µs
Peak Bandwidth	-	-	5998 MB/s	3713.6 MB/s
Connection Overhead	-	-	43.04 MB/s	27.96 MB/s
Size of File Cache	-	-	-	1GB - 2GB
File Read Time (32 MB)	20 µs	20 µs	30 µs for sequential read 60 µs for random read	33.9 µs for sequential read 84.31 µs for random read
Remote file read Time	6-7 ms for network penalty	2-4 ms during network traffic	7-9 ms for Sequential Read. 9-12 ms for Random Read.	For sequential read, 11-12 ms for 1MB- 8MB, 17-18 ms for 16MB - 64MB. For Random read, 13-16ms for 1MB-4MB. 17-19ms for 8MB-32MB. 21ms for 64MB.
Contention	Each context switch costs 25µs	3-5µs due to flush of prefetching data	From single process result, each additional process increases Sequential cost by 12-20 µs, and Random cost by 8 9-16 µs	For sequential reads, starting from 28.86 µs for a single process, having 8-16 µs increase for each additional process. For random reads, starting from 70.9µs for a single process, having 4-14µs increase for each additional process.

Reference:

[1]https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf

[2]https://en.wikichip.org/wiki/intel/microarchitectures/saltwell

[3] Dan Tsafrir IBM T.J. Watson Research Center, “The Context-Switch Overhead Inflicted by Hardware Interrupts (and the Enigma of Do-Nothing Loops) ”, P.O. Box 218, Yorktown Heights, NY 10598

[4] Larry McVoy and Carl Staelin, lmbench: Portable Tools for Performance Analysis, Proc. of USENIX Annual Technical Conference, January 1996

[5] http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

[6] Aaron B. Brown and Margo I. Seltzer, Operating system benchmarking in the wake of lmbench: a case study of the performance of NetBSD on the Intel x86 architecture, Proc. of ACM SIGMETRICS, pp. 214-224, June 1997. (Colloquially known as the "hbench" paper.)

[7] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazières, Antonio Dias, Margo Seltzer, and Michael D. Smith, The Measured Performance of Personal Computer Operating Systems, Proc. of ACM SOSP, pp. 299-313, December 1995.

[8]https://learn.microsoft.com/en-us/gaming/gdk/_content/gc/system/overviews/finding-threading-issues/high-context-switches

[9]https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html

[10] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. 1984. A fast file system for UNIX. ACM Trans. Comput. Syst. 2, 3 (Aug. 1984), 181–197.