CSE 221: Principles of Computer Operating Systems Final Report

Hao-En Sung
Computer Science and Engineering
A53204772
h3sung@ucsd.edu

Ping-Tsung Hsu
Computer Science and Engineering
A53202251
pihsu@ucsd.edu

Wei-Yuan Wen
Computer Science and Engineering
A53202335
w5wen@ucsd.edu

I. Introduction

In this project, our goal is to design a set of experiments to measure different components in an operating system, including CPU, memory, network, and file system. In this way, we can have a better insight in the characteristic of underlying hardware components and how they influence the performance of the whole operating system. Specifically, we implement our experiments in C along with inline assembly codes and run them on ASUS ZenBook UX31E[1] with Ubuntu 14.04 LTS operating system. To produce robust and stable results, we use gcc 4.8.4 with "-O0" optimizing flag to prevent any unintended optimization.

II. Machine Description

A. ASUS ZenBook UX31E

- Processor
  - Model: Intel Core i5-2557M, 1.7 GHz[2]
  - Cycle Time: 0.625 ns
  - Cache Size:
    L1-I: 2*32 KB, L1-D: 2*32 KB;
    L2: 2*256 KB; L3: 3 MB;
  - Memory Bus: 21.3 GB/s
- I/O Bus: DMI, 5GT/s
- RAM Size: 4 GB, DDR3 1333
- Disk: SanDisk SSD U100, 128 GB
- Network Card Speed: 1 Gbps
- Operating System: Ubuntu 14.04 LTS

III. Experiments

A. CPU, Scheduling, and OS Services

Measurement overhead should be known before conducting any other experiment in order to get precise results. We used the two instructions *rdtsc* (Read Time-Stamp Counter), *rdtscp* (Read Time-Stamp Counter), and Processor ID, to retrieve the number of CPU cycles. To make measurements more precise, we fix CPUs frequencies to 1.6GHz with *cpupower* tools, and run the experiments with a single core with *taskset* command. Also, these measurement programs are set to the highest priority with *nice* program, to reduce potential interferences from other processes.

Each of the experiment is conducted 100000 times to ensure the correctness of the results. Some of the results are found to be multiple-time larger than others, and these results are regarded as outliers and discarded.

1) Measurement Overhead: To measure the measurement overhead, we put nothing in the program except the codes used to retrieve the number of CPU cycles. Pseudocodes are provided below.

Algorithm 1 measure_start()

1: Flush CPU pipeline.
2: Read the number of CPU cycle into registers.
3: Move the data in registers into variables in memory.

Algorithm 2 measure_end()

1: Read the CPU cycle number into register.
2: Move the data in registers into variables in memory.
3: Flush CPU pipeline.

According to the document[3] on the Internet, we found that the latencies of the instructions *rdtsc*, *rdtscp*, are 28 and 36 cycles respectively on Intel 2nd gen. core - Sandy Bridge. Thus, we predicted that the measurement overhead should be at most $28 + 36 = 64$ cycles latency in theory. In the result shown in Table I, the average measurement overhead is 55.63 cycles, comparing to our previous prediction, it is close to our estimation. With only about 8.37 difference in cycle, we think the experiment result is acceptable.

TABLE I. Measurement Overhead

	# Cycles	Time (ns)
Est	64	40
Avg	55.63	34.78
Std	2.42	1.51

To measure the loop overhead, we put a for loop which does nothing between the two cycle measurement units. We calculate the number of CPU cycles it needs to iterate through the loop, and found out the loop overhead should be 8.38 cycle per loop, and the standard deviation is 0.18 cycle.

[1] https://www.asus.com/us/Notebooks/ASUS-ZenBook-UX31E/specifications/

[2] https://ark.intel.com/products/54620/Intel-Core-i5-2557M-Processor-3M-Cache-up-to-2_70-GHz

[3] http://www.agner.org/optimize/instruction_tables.pdf

Algorithm 3 Measurement Loop Overhead

```
1: startCycleNum = measure_start()
2: Do Nothing...
3: endCycleNum = measure_end()
4: return endCycleNum - startCycleNum
```

2) Procedure Call Overhead: To evaluate the time used for each procedure call, we created eight functions, which take different number of arguments ranging from 0 to 7. To estimate the cycle of a procedure call with 0 argument, we navigated through the assembly code and found that there are 8 instructions for zero-argument function call, which indicates that it needs 8 cycles to complete. Another finding is that the number of instructions increases by 1 as the number of argument increases by 1.

Algorithm 4 Procedure call overhead

```
1: startCycleNum = measure_start()
2: call functions with different number of arguments
3: endCycleNum measure_end()
4: return endCycleNum - startCycleNum
```

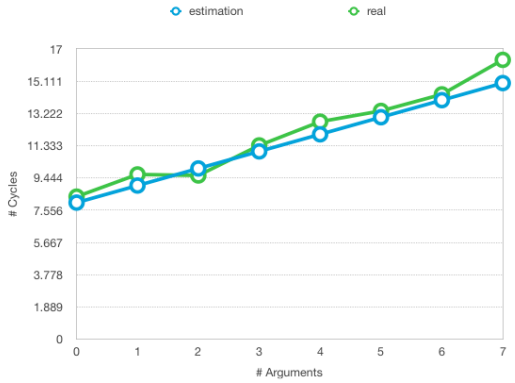

Fig. 1. Procedure Call Overhead

In our experiment, the mean of measurement overhead is excluded from each measurement of procedure call. The result is shown in Table II and Figure 1, which agrees with our intuition that the more arguments a function has, the longer it takes to complete a call. Also, the result is really close to our estimation. Our estimation is that the overhead should grow linearly with respect to the increment of arguments and we can confirm that easily by inspecting the assembly code. We think that this is why our estimation is successful.

3) System Call Overhead: To evaluate the system call overhead, it's better to experiment with a simple system call to keep the calculation simple. After surveying on the Internet, we found that *getpid()* is the simplest system call with no argument, and that it returns the id of the current process. Also, We found that the *pid* would be cached after *getpid()* being called the first time. Therefore, we used *vfork()* to deal with the cache problem. *vfork()* is a method that would create a child process, and suspend the parent process until the child process terminates. Thus, we can guarantee there is no context switch between child process and parent process. Moreover,

TABLE II. PROCEDURE CALL OVERHEAD

	# Cycles	Time (ns)
Est	8	5.00
Avg	8.35	5.22
Std	3.55	2.22

(a) No Argument

	# Cycles	Time (ns)
Est	9	5.62
Avg	9.65	6.03
Std	3.70	2.31

(b) One Argument

	# Cycles	Time (ns)
Est	10	6.25
Avg	9.59	5.99
Std	3.67	2.29

(c) Two Arguments

	# Cycles	Time (ns)
Est	11	6.88
Avg	11.35	7.09
Std	3.83	2.39

(d) Three Arguments

	# Cycles	Time (ns)
Est	12	7.50
Avg	12.74	7.96
Std	3.94	2.46

(e) Four Arguments

	# Cycles	Time (ns)
Est	13	8.13
Avg	13.37	8.36
Std	3.9	2.44

(f) Five Arguments

	# Cycles	Time (ns)
Est	14	8.75
Avg	14.34	8.96
Std	4.03	2.52

(g) Six Arguments

	# Cycles	Time (ns)
Est	15	9.38
Avg	16.36	10.225
Std	4.03	2.52

(h) Seven Arguments

they share the same memory, and, therefore, we can guarantee that child process wouldn't encounter cache miss. As a result, other overhead are mostly eliminated.

To estimate the result, we add a *getpid()* call between the two-cycle measurement units, and iterate the whole block of codes for several times. Then average the results except for the first one, because its value is too big, and may result from cache missed. We get an average value, 22.32 cycles, for *getpid()* after removing the measurement overhead. Later, we found that the overhead of an uncached *getpid()* call would be about twice that of the cached one. Therefore, we estimate the result to be 22.32 * 3 = 66.96 cycles. In the result ,shown in Table III, it's easily to see that the cost of a system call is much higher than that of procedure call. Moreover, the experiment result is much higher than we expected, almost three times higher. We think that it may result from some instruction reverting, condition tests and jumps in the kernel. The method is not good enough to measure the system call accurately, and the experiment result is not good.

TABLE III. SYSTEM CALL OVERHEAD

	# Cycles	Time (ns)
Est	66.96	41.85
Avg	276.04	172.53
Std	25.45	15.91

4) Task Creation Time: To evaluate the overhead of process creation and thread creation, we measure the time used for calling fork() and pthread_create(). We observed that the parent process/thread will always start executing before the child

Algorithm 5 System Call Overhead

```
1: pid = vfork()
2: if pid > 0 then
3:     waitpid(pid)
4: else if pid == 0 then
5:     startCycleNum = measure_start()
6:     getpid()
7:     endCycleNum = measure_end()
8:     exit(0)
9: end if
10: return endCycleNum - startCycleNum
```

process/thread does. Thus we stop our time measurement in the parent process before it waits for child process to exit.

The estimation of creating a task like process or thread is relatively hard. We can break pthread_create() down into several steps, including clone() system call (i.e. the minimal system call overhead has 276.04 cycles according to our measurement) and allocating and initializing the necessary resources. Thus, We estimate that it may cost 1000 cycles to create a thread. Then we guess the overhead of creating a process should be about 2 times more than that of thread creation, since that more system calls and kernel functions are needed to be called and completed in process creation. Furthermore, a new page table is created and some information should be updated as well during the fork.

Algorithm 6 Process creation overhead

```
1: startCycleNum = measure_start()
2: pid = fork()
3: if pid == 0 then
4:     exit(0)
5: else
6:     endCycleNum = measure_end()
7:     waitpid(pid)
8: end if
9: return endCycleNum - startCycleNum
```

Algorithm 7 Thread creation overhead

```
1: function start_routine
2:     Do nothing
3: end function
4: startCycleNum = measure_start()
5: pthread_create(&thread, NULL, *start_routine, NULL)
6: endCycleNum = measure_end()
7: pthread_join(thread, NULL)
8: return endCycleNum - startCycleNum
```

TABLE IV. PROCESS CREATION OVERHEAD

	# Cycles	Time (ns)
Est	3000	1875
Avg	67226.17	42016.35
Std	1506.42	941.51

As our result shows in Table IV and Table V, the thread creation overhead is much larger than our estimation and the process creation overhead is about 7 times of that of creating a thread. The difference is almost twice higher than our estimation. The main reason why the overhead of process creation is larger is that it copies tables and creates copy-on-write mapping for memory, while the thread creation is creating a process using shared address space without additional copy, and therefore it is just a unit that can be scheduled. In consequence, unless one needs an independent address space, or creating a new thread is preferred and forking a process should be avoided.

TABLE V. THREAD CREATION OVERHEAD

	# Cycles	Time (ns)
Est	1000	625
Avg	9636.82	6023.01
Std	1048.62	655.39

5) Context Switch Time: To evaluate the time used for process switches, we have to design an experiment where processes need to spend most of the time waiting for each other. In view of this, message communication between two processes or threads best fit our goal. The procedure of the experiment can be summarized as follows.

Algorithm 8 Context Switch Overhead between Processes

```
1: create two pipes: pipeA, pipeB
2: pid = fork()
3: if pid == 0 then
4:     startCycleNum = measure_start()
5:     read(pipeA)
6:     write(pipeB)
7:     endCycleNum = measure_end()
8:     return endCycleNum - startCycleNum
9: else
10:    write(pipeA)
11:    read(pipeB)
12: end if
```

Algorithm 9 Context Switch Overhead between Threads

```
1: create two pipes: pipeA, pipeB
2: function Func
3:     write(pipeA)
4:     read(pipeB)
5: end function
6: pthread_create(&thread, NULL, Func, NULL)
7: startCycleNum = measure_start()
8: read(pipeA)
9: write(pipeB)
10: endCycleNum = measure_end()
11: pthread_join(thread, NULL)
12: return endCycleNum - startCycleNum
```

To estimate the number of cycles from our own knowledge is a little bit challenging to us. From our understanding and online reference [4], context switches between threads include an interrupt from software, i.e. switch from user space to system base, storage and recovery of core states, an interrupt return, where switch from user to kernel space costs 227 cycles,

[4] http://wiki.osdev.org/Context_Switching

return from kernel to user space costs 180 cycles, storing and restoring registers costs 79 cycles. Thus the overall number of cycles should be $227 + 180 + 79 = 486$. However, it differs a lot in real world, since different architectures might require different levels of caches, which might greatly enhance the number of cycles. Context switches between processes are very similar, except it further requires to also switch the address spaces. The minimum cost of it, including a minimal TLB reload of one code page, one data page, and one stack page, costs 177 more cycles, which results in $486 + 177 = 663$ cycles.

However, the above-mentioned procedure not only includes the time for context switch, but also the CPU overhead and time for *read* and *write*. To measure the time more precisely, I need to measure the average cost for *read* and *write* function calls on single process in advance. After that, I subtract both CPU overhead and average cost for *read* and *write* from obtained results. On top of that, from our experiments, we can find some obvious outliers. According to our inference, it occurs when CPU context switched to neither parent process nor child process but other processes. Thus, we can simply ignore those outliers. The results are shown in Table VI. We also work on the context switches between threads. The experiment procedure for thread is pretty similar to the one for process, except I replace *fork* with *pthread_create* and *pthread_join* now. The results are recorded in Table VII.

TABLE VI. CONTEXT SWITCH BETWEEN PROCESSES

	# Cycles	Time (ns)
Est	663	414.375
Avg	3213.10	2008.19
Std	152.55	95.34

TABLE VII. CONTEXT SWITCH BETWEEN THREADS

	# Cycles	Time (ns)
Est	486	303.75
Avg	2695.23	1684.52
Std	153.70	96.06

For a quick conclusion, it can be found that context switches between processes take significant longer time than context switches between threads, which meets our expectation.

B. Memory

1) RAM Access Time: In this section, we are aimed to measure the latencies of cache and memory accessing. There are three levels of cache on our experiment machine, which are L1-D cache with 32 KB, L2 cache with 256KB, and L3 cache with 3 MB. It's important to point out that the L1-D cache is attached to each core, and L2, L3 cache are shared among the 2 cores.

For the method in the experiment, we allocate a memory space with specified size, and set up an integer array inside. The content of each element in the array is the index of the next element which is ahead of the former element for stride size in memory. Therefore, when the experiment runs, it will traverse reversely in the memory in order to minimize the effect of cache pre-fetching. As a result, we conduct the experiments with different memory sizes and stride sizes. We'll fetch the the elements in the array for one million times, and calculate the mean fetching latency after subtract the measurement latency. Also, we need to subtract the loop latency from the mean fetching latency.

According to the non-official document on the Internet[5], the latencies of each level of cache are 4, 12, and 27.85 cycles respectively, and we expected our experiment result should be around these numbers. In Figure 2, it's easily to see that there are three abrupt risings in the number of cycles, and these rising happens at about the size of the caches respectively. The measured cache-access latencies of the three levels of cache are 5.59, 13.05, and 32.38 cycles respectively, shown in Table VIII. For L1 and L2 cache, the measured results are close to our expectation, with less than 2 cycle difference. When it comes to accessing the memory directly, the cost goes high quickly, almost 100 cycles more than accessing cache. The latency may result from the memory controller or the memory bus. We think the experiment result is accurate and, moreover, it points out the importance of the existence of cache in nowadays machine.

Algorithm 10 RAM Access Time

```
numOfElement = ARRAY_SIZE/sizeof(int)
stepOfIndex = STRIDE_SIZE/sizeof(int)
arr = memory allocation.
for (i=0; i < numOfElement; i++) do
    index = ( i + stepOfIndex ) % numOfElement
    arr[index] = i;
end for
startCycleNum = measure_start()
for (ITERATION_TIMES) do
    index = arr[index];
end for
endCycleNum = measure_end()
return endCycleNum - startCycleNum
```

TABLE VIII. RAM ACCESS TIME

	L1	L2	L3	Memory
Est # Cycles	4	12	27.85	100
Avg # Cycles	5.59	13.05	32.38	100 - 170

2) RAM Bandwidth: In this section, we are going to measure the maximum bandwidth for both reading and writing of the RAM. For measuring the read bandwidth, we use a loop to sum up the values in a large integer array.

According to the non-official document on the Internet[6], the L3 cache has 64 Byte-line. In order to saturate the memory bandwidth, we set the size of array to be 4 times of that of the L3 cache and the basic stride size to be 16 bytes since the integer size is 4 bytes on our system. The loop is unrolled 32 times, thus we can avoid too many branch instructions and eliminate the loop overhead. Upon unrolling the loop, the size of stride is updated correspondingly to 512 bytes. To get a more accurate result, we fetch the first element in each cache

[5]http://www.7-cpu.com/cpu/SandyBridge.html
[6]http://www.7-cpu.com/cpu/SandyBridge.html

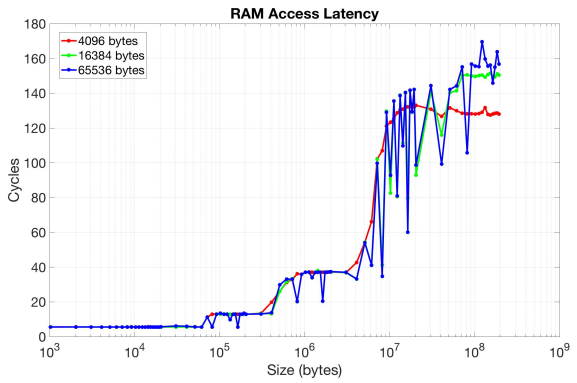

Fig. 2. RAM Access Latency

line in order to eliminate the effect of cache line prefetching. In each iteration, the values, which are 64 bytes away from each other, are accumulated so that the compiler won't optimize the instructions that read the content of the array.

To measure the write bandwidth, a similar methodology is used. We use an unrolled loop and stored values into the first element of each cache line of the large integer array with the same stride sizes and the same unrolling times. The pseudo code for the experiment is as below.

Algorithm 11 RAM Read Bandwidth

```
1: ARRAY_SIZE = 4*L3_CACHE_SIZE
2: BASE_STRIDE = 16, UPDATED_STRIDE = 32
3: sum = 0
4: ptr = array
5: endOfArray = ptr + ARRAY_SIZE
6: startCycleNum = measure_start()
7: for (; ptr!=endOfArray; ptr+=UPDATED_STRIDE) do
8:     sum += ptr[0] + ptr[BASE_STRIDE] + ··· +
       ptr[31*BASE_STRIDE]
9: end for
10: endCycleNum = measure_end()
11: return endCycleNum - startCycleNum
```

Algorithm 12 RAM Write Bandwidth

```
1: ARRAY_SIZE = 4*L3_CACHE_SIZE
2: BASE_STRIDE = 16, UPDATED_STRIDE = 32
3: ptr = array
4: endOfArray = ptr + ARRAY_SIZE
5: startCycleNum = measure_start()
6: for (; ptr!=endOfArray; ptr+=UPDATED_STRIDE) do
7:     ptr[0] = 1
8:     ptr[BASE_STRIDE] = 1
9:     ···
10:    ptr[31*BASE_STRIDE] = 1
11: end for
12: endCycleNum = measure_end()
13: return endCycleNum - startCycleNum
```

For estimation, we expect the write bandwidth to be the half of the read bandwidth, since upon a write a cache line should be read into cache before the value is written back to memory. Therefore, the write speed should be the half of the read speed. In consequence, according to our machine Memory Bus, we estimate the maximum read bandwidth and write bandwidth should be 21.3 GB/s and 10.65 GB/s, respectively.

The measurement is shown in Table IX. The result is surprisingly small comparing to our expectation, which is about half of the full memory bandwidth. After read the article written by Alex Reece[7], it seems that using a single threaded program running on a single core is hard to saturate the entire memory bandwidth. He used OpenMP to run multiple threaded program over 4 cores and achieved up to 93% of the theoretical bandwidth. However, we didn't do the further experiment to achieve the maximum bandwidth.

TABLE IX. RAM BANDWIDTH

	Bandwidth (GB/s)
Est	21.3
Avg	9.82
Std	0.21

(a) Read

	Bandwidth (GB/s)
Est	10.65
Avg	6.32
Std	0.11

(b) Write

3) Page Fault service time: A page fault occurs when a page of virtual memory blocks is not actually loaded into physical memory. When this error is captured by hardware, page fault handler will search for the corresponding contents on disk and reload them into main memory. If there is no sufficient space to load these new contents, some other existing pages will need to be swapped out.

To put experiment on this measurement, we need to first create a huge enough file on disk with the command

dd if=/dev/urandom of=FILE bs=1048576 count=4096,

and map the entire file into main memory through *mmap* function. In case the file is cached by the hardware, we also need to use */proc/sys/vm/drop_caches* to flush file cache. For example:

echo 3 | /proc/sys/vm/drop_caches.

The pseudo code for the following experiment can be summarized as follows. It is noticeable that we read 16 MB in each loop, since each disk read will load nearby 32 $4K$ pages. Reading in 16 MB per cycle can effectively avoid this issue.

Algorithm 13 Connection Overhead - Establish (Clinet)

```
1: fd = open(FILE, O_DIRECT + O_SYNC)
2: my_mmap = mmap(FILE, fd)
3: startCycleNum = measure_start()
4: read(my_mmap)
5: endCycleNum = measure_end()
6: munmap(my_mmap)
7: close(fd)
8: return endCycleNum - startCycleNum
```

[7]http://codearcana.com/posts/2013/05/18/achieving-maximum-memory-bandwidth.html

The page fault service is mainly consisted of data access and data transfer from disk to memory. Since our computer is using SDD (SanDisk U100), we only need to focus on its data access rate, which is 480 MB/s according to the official spec. Thus, the overall time should be around $16/480 * 1000 \approx 33$ ms. The experiment results are recorded in Table X.

TABLE X. PAGE FAULT SERVICE TIME

	# Cycles	Time (ns)
Est	53333333.33	3333333.33
Avg	1401022.71	875639.19
Std	117301.28	73313.30

From the table, it can be found out that our estimated time is much larger ($\sim 4x$) than the real running time. We believe it might be caused by the disk cache, which we might estimate directly in the later section.

C. Network

For this section, we acquire another computer to complete the measurement. Information provided below is the machine description of the computer used in the measurement.

Apple MacBook Pro Retina 2016[8]

- Processor
  - Model: Intel Core i5-5257U, 2.70GHz[9]
  - Cycle Time: 0.363 ns
  - Cache Size:
    L1-I: 2*32 KB, L1-D: 2*32 KB;
    L2: 2*256 KB; L3: 3 MB;
  - Memory Bus: 25.6 GB/s
- I/O Bus: 5GT/s, DMI
- RAM Size: 16GB DDR3, 1867
- Disk: 256 GB SSD
- Network Card Speed: 1 Gbps
- Operating System: OS X EI Capitan Version 10.11.6

1) Round Trip Time: In this section, we are aimed to measure the round trip time for network latency. We set up a TCP connection between two machines with one server and one client. Client sends 64 bytes of data to the server and waits for the echo from the server. The reason for choosing 64 bytes for the data is that the *ping* command sends out 64 bytes of data. For *ping*, we utilize the default *ping* command to ping the other machine. We measure the time from sending out the data to receiving the echo from server.

For local network, we run the experiment on the ASUS laptop and run the server and client programs on the same core using *taskset* command. For remote network, we run the client and server on ASUS and Mac laptop with the specifications provided above. The experiment result is shown in table VIII.

We think that ping should take less time comparing to TCP, because that ping is processed in the kernel, and it does not need to cross the kernel, user-space interface. In the result, we found that the ping uses a little more time than TCP in local, but the small difference is acceptable. The remote one matches our thought that TCP uses more time than ping.

Algorithm 14 Round Trip Time

```
1: Connect to server
2: startCycleNum = measure_start()
3: Send 64 Bytes to server
4: Wait for response from server
5: endCycleNum = measure_end()
6: return endCycleNum - startCycleNum
```

TABLE XI. ROUND TRIP TIME

	TCP (remote)	TCP (local)	ping (remote)	ping (local)
Avg	929.98 us	32.03 us	766 us	37 us
Std	125.81 us	9.62 us	111 us	7 us

2) Peak Bandwidth: In this section, we are aimed to measure the peak bandwidth for our network. We evaluate for TCP protocol and compare both remote and local hosts. To measure the bandwidth, we record the average time elapsed when client sending 16MB of data to server, repeated 1000 times and find the maximum value among records. The pseudo code for server and client is provided below.

Algorithm 15 Peak Bandwidth (Server)

```
1: startCycleNum = measure_start()
2: for res != 16MB do
3:     res += read 4KB message to client
4: end for
5: endCycleNum = measure_end()
6: return endCycleNum - startCycleNum
```

Algorithm 16 Peak Bandwidth (Client)

```
1: for res != 16MB do
2:     res += write 4KB message from server
3: end for
```

Since we are using a full-duplex link, we expect our bandwidth can achieve almost the theoretical value of network card speed, which is 125MB/s (1Gbps). However, we runs the experiment on a local network. We assume the maximum network bandwidth will be limited by the USB2.0 to Ethernet Cable[10] we used in measuring remote peak bandwidth, which only support 10/100M LAN. Thus the maximum possible bandwidth is 12.5MB/s. The Ethernet frame is 38 octects[11], the size of IP header and TCP header is 20 octects each. With the Ethernet MTU of 1500, we expect to get (1500-40)/(1500+38) = 94.93% of theoretical bandwidth, which is 11.86MB/s. For local measurement, we expect the bandwidth should be much greater than the remote one. Furthermore, we think the bandwidth should be limited by the system call time and the memory copy time. To estimate the local network bandwidth, the RAM bandwidth can give us some guidance. Since the data need to be copied from user space to kernel

[8]https://support.apple.com/kb/SP715?viewlocale=en_US&locale=zh_TW
[9]https://ark.intel.com/products/84985/Intel-Core-i5-5257U-Processor-3M-Cache-up-to-3_10-GHz
[10]https://www.asus.com/Laptops-Accessory/USB_Ethernet_Cable/specifications/
[11]https://en.wikipedia.org/wiki/Ethernet_frame

space and vice versa, we need to count the time of copy twice, which gives us $2 * (1/21.3 + 1/10.65) = 0.28$s of copy time of copying 1GB data. Thus we estimate the local bandwidth is about 3550MB/s.

The results are shown in Table XII, the data indicates that our methodology is doing quite well. For remote measurement, our result is close to that of our estimation, it is about 90% of the estimation. For local measurement, our data is quite close to our estimation but somewhat a little bit higher. We get this accurate measurement since we have done the RAM bandwidth experiment before, so we have some information to base on. After comparing between remote and local results, we regard that the memory copy time and system call time is much faster than network card speed, so that we can ignore the software overhead.

TABLE XII. PEAK BANDWIDTH

	Bandwidth (MB/s)
Est	3550
Peak	3822.04
Avg	2006.20
Std	992.24

(a) Local Host

	Bandwidth (MB/s)
Est	11.86
Peak	10.60
Avg	10.59
Std	0.005119

(b) Remote Host

3) Connection Overhead: It is known that three-way handshake is required to establish a TCP connection between a server and a client. In the beginning, the client will send a *SYN* message to the server. After receiving this information, the server will reply a message with *SYN+ACK* to the client. At the end, the client will send back a message with ACK as a confirmation.

On the other hand, to terminate a TCP connection requires four steps. When a client plans to terminate a connection, it will send out a *FIN* message to the server. While receiving this message, the system will reply two messages individually: one is *ACK* as the confirmation of receiving *FIN* from client; another one is *FIN* to indicate the termination of another half-duplex communication. Finally, the client will reply with a *ACK* as a confirmation.

To examine the connection overhead, we use the same pseudo code for server from previous sections and write down the one for client as follows.

Algorithm 17 Connection Overhead - Establish (Clinet)

```
1: create socket: sockfd
2: configure server socket address
3: startCycleNum = measure_start()
4: connect(socked)
5: endCycleNum = measure_end()
6: close(socked)
7: return endCycleNum - startCycleNum
```

Here, we directly refer the results from *Round Trip Time* section as our expected values for handshake establishment. We can tell our expectations are very close to the real experiment results.

Algorithm 18 Connection Overhead - Terminate (Clinet)

```
1: create socket: sockfd
2: configure server socket address
3: connect(socked)
4: startCycleNum = measure_start()
5: close(socked)
6: endCycleNum = measure_end()
7: return endCycleNum - startCycleNum
```

From the result Table XIII and XIV, one can tell that the time for TCP termination is much smaller than TCP establishment. We believe the reason is that TCP termination is an asynchronized function call. In other words, what we measure in our codes is only the time to establish the system call but not the time to terminate TCP connection, i.e. the receive of second *ACK*. Thus, we directly use the overhead for *getuid* system as our estimation.

TABLE XIII. CONNECTION OVERHEAD: ESTABLISH

	# Cycles	Time (us)
Est	51248.00	32.03
Avg	60737.38	37.96
Std	15255.02	9.53

(a) Local Host

	# Cycles	Time (us)
Est	1487968.00	929.98
Avg	1080009.16	675.73
Std	156078.93	97.55

(b) Remote Host

TABLE XIV. CONNECTION OVERHEAD: TERMINATE

	# Cycles	Time (us)
Est	276.04	172.53
Avg	17065.94	10.67
Std	8527.71	5.33

(a) Local Host

	# Cycles	Time (us)
Est	276.04	172.53
Avg	23902.36	14.94
Std	9318.80	5.82

(b) Remote Host

D. File System

1) Size of File Cache: To measure the size of the file cache, we measure the time required to read a block from a file. The file cache should work similar to memory cache: if the file is small enough to fit in the file cache, then the time required to read a block from cache should be a lot smaller than that from disk. Therefore, we used the time required as a benchmark to estimate the size of file cache.

We experimented with different sizes of files ranging from $1GB$ to $4GB$, the memory size of the machine, to acquire the time required to read a block. We used the *dd* command to create files with various sizes. Right after the file is created and is put into the file cache (if applicable), we read the file block by block, whose size is 4096 bytes, and measure the time required to read the file per block.

By plotting out the figure with various required time and file sizes, in Figure 3, it's easily to find there's a abrupt rising of the required time when the file size is around 3.5GB. Before the file size reaches 3.5GB, the required time to read a block of the file is less than 1 us. After if, the required time to read a block jumps to around 9 to 10 us. Therefore, we believe that 3.5GB is the size of file cache on the machine.

Also, we can utilize these commands below to estimate the size of the file cache. *free* command provides users information about the usage of RAM in Linux machine and *echo 3 | /proc/sys/vm/drop_caches*. enable users to flush the file cache on the machine. With these two commands, we can create different size of files with the *dd* command, and see the size that is cached on the RAM by *free* command. When the cached size doesn't grows as the file size, and it's the size of the file cache on the machine. With these two methods, they both led to the same result telling that the file cache of the machine is about $3.5GB$. Also after calculation, the speed to read file from disk (9 us/per block) is around 444 MB/S, which is almost the same as the information provided on the specifications (450 MB/s). Therefore, we think the experiment result is accurate.

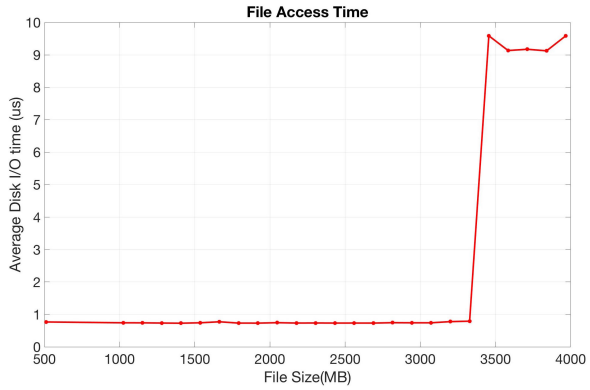

Fig. 3. File Cache Size

2) File Read Time: To measure the time for file reading, it is crucial to exempt the file cache mechanism. It can be achieved by using

echo 3 | /proc/sys/vm/drop_caches.

command again to remove all existing file caches and adding parameter *O_DIRECT + O_SYNC* when one opens a file.

We use 4096 (4 KB) as the block size, which is the same as physical page size. In order to examine the factor of random access, we put experiments on both sequential and random accesses, whose pseudo code are listed as follows.

Algorithm 19 File Read - Sequential Access

```
fd = open(FILE, O_DIRECT + O_SYNC)
for i = 0 to MAX_ITER do
    startCycleNum = measure_start()
    read(fd, BLOCK_SIZE)
    endCycleNum = measure_end()
    record.append(endCycleNum - startCycleNum)
end for
close(ed)
return record
```

We put both sequential access and random access experiments on 8 different file sizes ranging from $1(=2^0)$ MB to $128(=2^7)$ MB, and the results are recorded as Table XV and Table XVI, respectively.

According to SANDISK U100 official documentation and UserBenchmark, the sequential read speed should be around 450 MB per second, which equals to 0.01 ms or around 13888.89 cycles for a 4 KB page. Our experiment shows approximately the same performance and thus verify our algorithm correctness.

Algorithm 20 File Read - Random Access

```
fd = open(FILE, O_DIRECT + O_SYNC)
for i = 0 to MAX_ITER do
    idx = rand() % (FILE_SIZE / BLOCK_SIZE);
    lseek(fd, idx * BLOCK_SIZE, SEEK_SET);
    startCycleNum = measure_start()
    read(fd, BLOCK_SIZE)
    endCycleNum = measure_end()
    record.append(endCycleNum - startCycleNum)
end for
close(ed)
return record
```

For random access of SSD, UserBenchmark says it can read approximately 13 MB per second, which is 0.30 ms or 480769.23 cycles for a 4 KB page. My implementation provides slightly faster performance; while both of documented results and our results are still at the same scale. We believe this is caused by randomness.

If we compare the results for sequential access and random access, we can clearly tell the benefits of sequential access: it is much more efficient. Due to the pre-fetch mechanism, disk will not read only one page for each *read* command, instead, it will pre-fetch a block of pages at once. We believe that is the main reason for two different accessing times.

3) Remote File Read Time: The experiment for this section is very similar to the previous one, except we need to regard one computer as NFS server and another one as client. To be specific, we plan to regard our MAC as server and edit the file */etc/exports* as follows.

/Users/rand-dat -alldirs -ro

By doing so, the *rand-dat* folder is visible for all machines on the same LAN. Later, we use our Ubuntu as client and mount the shared folder using command below.

*sudo mount -o lookupcache=none,actimeo=0,noac *
*10.42.0.12:/Users/rand-dat *
dat/remote-rand-dat/

Other procedures are exactly the same. We need to flush disk caches with *echo 3 | /proc/sys/vm/drop_caches* and open files with parameter *O_DIRECT + O_SYNC*. Experiments for sequential and random access on 8 different file sizes ranging from $1(=2^0)$ MB to $128(=2^7)$ MB are provided in Table XV and XVI.

For both sequential and random access measurements, we need to further consider the cost of network transfer. The adapter we use to connect our two machines: MAC OS and Ubuntu is ASUS USB Ethernet Cable, which provides 100 Mb per second or said 12.5 MB per second. In view of this, the estimation of sequential access to a 4 K page becomes $0.01 + 0.31 = 0.32$ ms, or said 513888.89 cycles. On the other hand, the estimation of random access to a 4 K page should be $0.30 + 0.31 = 0.61$ ms, or said 980769.23 cycles.

TABLE XV. FILE ACCESS - SEQUENTIAL ACCESS

	# Cycles	Time (ms)
Est	13888.89	0.01
Avg	47532.06	0.03
Std	251875.43	0.16

(a) FILE_SIZE: 2^0 MB

	# Cycles	Time (ms)
Est	13888.89	0.01
Avg	28351.68	0.02
Std	182189.11	0.11

(b) FILE_SIZE: 2^1 MB

	# Cycles	Time (ms)
Est	13888.89	0.01
Avg	22877.98	0.01
Std	133939.07	0.08

(c) FILE_SIZE: 2^2 MB

	# Cycles	Time (ms)
Est	13888.89	0.01
Avg	21233.65	0.01
Std	125802.63	0.08

(d) FILE_SIZE: 2^3 MB

	# Cycles	Time (ms)
Est	13888.89	0.01
Avg	22342.32	0.01
Std	138049.87	0.09

(e) FILE_SIZE: 2^4 MB

	# Cycles	Time (ms)
Est	13888.89	0.01
Avg	22676.50	0.01
Std	137301.71	0.09

(f) FILE_SIZE: 2^5 MB

	# Cycles	Time (ms)
Est	13888.89	0.01
Avg	22019.51	0.01
Std	132162.12	0.08

(g) FILE_SIZE: 2^6 MB

	# Cycles	Time (ms)
Est	13888.89	0.01
Avg	24313.62	0.02
Std	158732.27	0.10

(h) FILE_SIZE: 2^7 MB

TABLE XVI. FILE ACCESS - RANDOM ACCESS

	# Cycles	Time (ms)
Est	480769.23	0.30
Avg	282767.06	0.18
Std	288316.96	0.18

(a) FILE_SIZE: 2^0 MB

	# Cycles	Time (ms)
Est	480769.23	0.30
Avg	263074.51	0.16
Std	251677.41	0.16

(b) FILE_SIZE: 2^1 MB

	# Cycles	Time (ms)
Est	480769.23	0.30
Avg	248593.64	0.16
Std	251473.22	0.16

(c) FILE_SIZE: 2^2 MB

	# Cycles	Time (ms)
Est	480769.23	0.30
Avg	248132.90	0.16
Std	261733.64	0.16

(d) FILE_SIZE: 2^3 MB

	# Cycles	Time (ms)
Est	480769.23	0.30
Avg	262991.20	0.16
Std	253916.10	0.16

(e) FILE_SIZE: 2^4 MB

	# Cycles	Time (ms)
Est	480769.23	0.30
Avg	258142.60	0.16
Std	257959.04	0.16

(f) FILE_SIZE: 2^5 MB

	# Cycles	Time (ms)
Est	480769.23	0.30
Avg	251628.54	0.16
Std	222946.16	0.14

(g) FILE_SIZE: 2^6 MB

	# Cycles	Time (ms)
Est	480769.23	0.30
Avg	255746.18	0.16
Std	222838.36	0.14

(h) FILE_SIZE: 2^7 MB

TABLE XVII. REMOTE FILE ACCESS - SEQUENTIAL ACCESS

	# Cycles	Time (ms)
Est	513888.89	0.32
Avg	2011161.76	1.26
Std	4463867.52	2.79

(a) FILE_SIZE: 2^0 MB

	# Cycles	Time (ms)
Est	513888.89	0.32
Avg	2041930.84	1.28
Std	5820507.81	3.64

(b) FILE_SIZE: 2^1 MB

	# Cycles	Time (ms)
Est	513888.89	0.32
Avg	1983875.24	1.24
Std	8063135.61	5.04

(c) FILE_SIZE: 2^2 MB

	# Cycles	Time (ms)
Est	513888.89	0.32
Avg	2228949.08	1.39
Std	9854997.99	6.16

(d) FILE_SIZE: 2^3 MB

	# Cycles	Time (ms)
Est	513888.89	0.32
Avg	2190427.30	1.37
Std	8636959.44	5.40

(e) FILE_SIZE: 2^4 MB

	# Cycles	Time (ms)
Est	513888.89	0.32
Avg	2142221.94	1.34
Std	8538765.72	5.34

(f) FILE_SIZE: 2^5 MB

	# Cycles	Time (ms)
Est	513888.89	0.32
Avg	2178314.12	1.36
Std	8738268.27	5.46

(g) FILE_SIZE: 2^6 MB

	# Cycles	Time (ms)
Est	513888.89	0.32
Avg	2281115.90	1.43
Std	8102417.27	5.06

(h) FILE_SIZE: 2^7 MB

TABLE XVIII. REMOTE FILE ACCESS - RANDOM ACCESS

	# Cycles	Time (ms)
Est	980769.23	0.51
Avg	2618483.93	1.64
Std	1296805.34	0.81

(a) FILE_SIZE: 2^0 MB

	# Cycles	Time (ms)
Est	980769.23	0.51
Avg	2643743.85	1.65
Std	1535277.00	0.96

(b) FILE_SIZE: 2^1 MB

	# Cycles	Time (ms)
Est	980769.23	0.51
Avg	2582424.25	1.61
Std	1752686.31	1.10

(c) FILE_SIZE: 2^2 MB

	# Cycles	Time (ms)
Est	980769.23	0.51
Avg	2559437.90	1.60
Std	2390581.31	1.49

(d) FILE_SIZE: 2^3 MB

	# Cycles	Time (ms)
Est	980769.23	0.51
Avg	2598934.82	1.62
Std	1300645.86	0.81

(e) FILE_SIZE: 2^4 MB

	# Cycles	Time (ms)
Est	980769.23	0.51
Avg	2636123.22	1.65
Std	2316360.07	1.45

(f) FILE_SIZE: 2^5 MB

	# Cycles	Time (ms)
Est	980769.23	0.51
Avg	2556644.82	1.60
Std	2151206.96	1.34

(g) FILE_SIZE: 2^6 MB

	# Cycles	Time (ms)
Est	980769.23	0.51
Avg	2562594.39	1.60
Std	1255458.21	0.78

(h) FILE_SIZE: 2^7 MB

Our experiment results are a little bit slower than our estimation. Our explanation for it is that the network overhead is too uncertain and hard to measure accurately. It is also known that, most of the time, Ethernet cable cannot provide full-speed data transfer. On top of that, we can tell some interesting conclusions from the table: the difference between sequential access and random access are shorten greatly, since the execution time is most dominated by the network transfer.

4) Contention: To measure the contention, we first created 16 files with 64 MB using *dd* command. These files are used to read sequentially from the disk. Then we run the experiment multiple times, each time using different amount of threads (from 1 to 16) to read different files simultaneously. We use threads instead of processes in order to get a more accurate result, since the context switch overhead of processes is much larger than threads' switch. As mentioned in previous section, it is important to flush file cache in case the file is cached by the hardware and to exempt the file cache mechanism. For flushing file cache, we use *echo 3 | /proc/sys/vm/drop_caches*, and for disabling the file system cache, we open files with parameter *O_DIRECT + O_SYNC*. The pseudo code is provided below.

Algorithm 21 Contention

```
1: for each thread do
2:     fd = open(FILE, O_DIRECT + O_SYNC)
3:     startCycleNum = measure_start()
4:     sequential read all bytes of every block of the file
5:     endCycleNum = measure_end()
6:     close(ed)
7:     return endCycleNum - startCycleNum)
8: end for
```

For the prediction, it is quite hard to estimate the time reading the disk, since it includes thread waiting time and the time to read blocks. Although our disk is SSD, the seek time still exists. According to Wikipedia[12], the seek time is 0.1 ms and is independent from the access disk location. Furthermore, the thread switch overhead can be ignored since it is so small comparing to disk access. Our SSD uses Serial ATA-600[13]. According to the documentation, the interface is SATA 6 Gb/s, which equals to 750 MB/s. And the sequential read speed should be around 450 MB/s, which equals to 0.01 ms or around 13888.89 cycles for a 4 KB page. Since sequential read speed is smaller than the interface, it should be the upper limit of our bandwidth. For using a thread to read a file, we estimate the time should be 0.01 ms. For using two or more threads, we think the time spend should increase linearly with respect to the number of threads used plus seek time, which may be 0.1ms for each extra threads, since threads would share the interface.

Figure 4 shows that the average time of reading file using multiple threads grows linearly as the increment of threads, which also indicate that our methodology is doing quite well. This result is quite match with our intuition that the increasing trend is linear. However , the values of result are separate from that of our estimation. The real measurement is almost an order of magnitude slower than our estimation . We think it is due

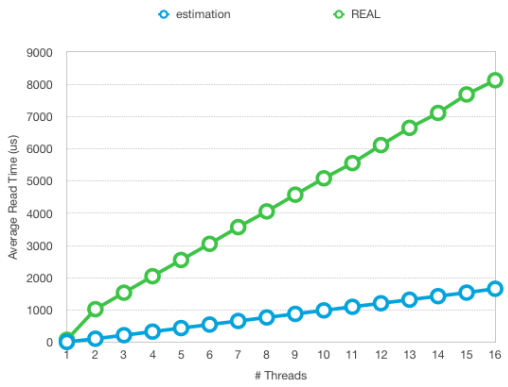

Fig. 4. Contention

to the thread scheduling issue. That is to say, when multiple threads are running simultaneously, they spend most of the time waiting to be chosen to executed. And as the increment of the running threads, the waiting time of each thread will increase linearly as well. We think that is the reason why the gap is getting larger when the number of running thread increases. There is another interesting finding that the time spent using one thread is much lesser than that of using two threads. This is the only part that slightly doesn't grow linear. We think that when using two threads, disk needs seek time to read between two different areas, while using one thread don't. For using even more threads, the seek time overhead is not that prominent comparing to the situation we just discussed. The experiment result is recorded in Table XIX.

Work Distribution

- Hao-En Sung: Instruction, Context Switch Overhead, Page Fault Service Time, Connection Overhead, File Read Time, Remote File Read Time
- Ping-Tsung Hsu: Machine Description, Measurement Overhead, System Call Overhead, RAM Access Time, Round Trip Time, Size of File cache
- Wei-Yuan Wen: Instruction, Procedure Overhead, Task Creation Overhead, RAM Bandwidth, Peak Bandwidth, Contention

[12]https://en.wikipedia.org/wiki/Solid-state_drive#cite_note-diffen-109

[13]https://www.cnet.com/products/sandisk-ssd-u100-solid-state-drive-32-gb-sata-600-series/specs/

TABLE XIX. CONTENTION

	# Cycles	Time (ms)
Est	13889	0.01
Avg	122976	0.08
Std	38782	0.02

(a) # Threads: 1

	# Cycles	Time (ms)
Est	152778	0.11
Avg	1642507	1.03
Std	282539	0.18

(b) # Threads: 2

	# Cycles	Time (ms)
Est	305556	0.22
Avg	2459464	1.54
Std	201545	0.13

(c) # Threads: 3

	# Cycles	Time (ms)
Est	458334	0.33
Avg	3277981	2.05
Std	490441	0.30

(d) # Threads: 4

	# Cycles	Time (ms)
Est	611112	0.44
Avg	4083048	2.55
Std	481640	0.47

(e) # Threads: 5

	# Cycles	Time (ms)
Est	763890	0.55
Avg	4884434	3.05
Std	749999	0.47

(f) # Threads: 6

	# Cycles	Time (ms)
Est	916668	0.66
Avg	5713302	3.57
Std	992228	0.62

(g) # Threads: 7

	# Cycles	Time (ms)
Est	1069446	0.77
Avg	6494068	4.06
Std	1468062	0.92

(h) # Threads: 8

	# Cycles	Time (ms)
Est	1222224	0.88
Avg	7323234	4.58
Std	1005475	0.63

(i) # Threads: 9

	# Cycles	Time (ms)
Est	1375002	0.99
Avg	8135629	5.08
Std	1272885	0.80

(j) # Threads: 10

	# Cycles	Time (ms)
Est	1527780	1.10
Avg	8899627	5.56
Std	2488483	1.56

(k) # Threads: 11

	# Cycles	Time (ms)
Est	1680558	1.21
Avg	9782653	6.11
Std	1216782	0.76

(l) # Threads: 12

	# Cycles	Time (ms)
Est	1833336	1.32
Avg	10638220	6.65
Std	1224988	0.77

(m) # Threads: 13

	# Cycles	Time (ms)
Est	1986114	1.43
Avg	11376920	7.11
Std	1692154	1.06

(n) # Threads: 14

	# Cycles	Time (ms)
Est	2138892	1.54
Avg	12295930	7.68
Std	1376607	0.86

(o) # Threads: 15

	# Cycles	Time (ms)
Est	2291670	1.65
Avg	13001060	8.13
Std	2775679	1.73

(p) # Threads: 16

TABLE XX. SUMMARY

	Operation	Estimated Cycles	Estimated Time	Measured Cycles	Measured Time
CPU and OS Service	Measurement Overhead	64	40 (ns)	55.63	34.78 (ns)
	Loop Overhead	—	—	8.38	5.24 (ns)
	Procedure Call Overhead(no argument)	8	5.00 (ns)	8.35	5.22 (ns)
	Procedure Call Overhead (7 arguments)	15	9.38 (ns)	16.36	10.23 (ns)
	System Call Overhead	66.96	41.85 (ns)	276.04	172.53 (ns)
	Task Creation Overhead (Process)	3000	1875 (ns)	67226.17	42016.35 (ns)
	Task Creation Overhead (Thread)	1000	625 (ns)	9632.82	6023.01 (ns)
	Context Switch Overhead (Process)	663	414.38 (ns)	3213.10	2008.19 (ns)
	Context Switch Overhead (Thread)	486	303.75 (ns)	2695.23	1684.52 (ns)
Memory	RAM Access Time (L1)	4	2.5 (ns)	5.59	3.49 (ns)
	RAM Access Time (L2)	12	7.5 (ns)	13.05	8.75 (ns)
	RAM Access Time (L3)	27.85	16.71 (ns)	32.38	20.24 (ns)
	Band Width (Read)	—	21.3 (GB/s)	—	9.82 (GB/s)
	Band Width (Write)	—	10.65 (GB/s)	—	6.32 (GB/s)
	Page Fault Service Time	53333333.33	3333333.33 (ns)	1401022.71	875639.19 (ns)
Network	Round Trip Time (TCP, local)	—	—	51.25	32.03 (ns)
	Round Trip Time (TCP, remote)	—	—	1487.97	929.98 (ns)
	Round Trip Time (ping, local)	—	—	11.20	7.00 (ns)
	Round Trip Time (ping, remote)	—	—	59.20	37.00 (ns)
	Peak Band Width (local)	—	3550 (MB/s)	—	3822.04 (MB/s)
	Peak Bandwidth (remote)	—	11.86 (MB/s)	—	10.60 (MB/s)
	Connection Overhead (establish, local)	51248.00	32.03 (ms)	60737.38	37.96 (ms)
	Connection Overhead (establish, remote)	1487968.00	929.98 (ms)	1080009.16	675.73 (ms)
	Connection Overhead (terminate, local)	276.04	172.53 (ms)	17065.94	10.67 (ms)
	Connection Overhead (terminate, remote)	276.04	172.53 (ms)	23902.36	14.94 (ms)
File System	Size of File Cache	—	—	—	3.5 (GB)
	File Sequential Read Time (2^0 M, local)	13888.89	0.01 (ms)	47532.06	0.03 (ms)
	File Sequential Read Time (2^7 M, local)	13888.89	0.01 (ms)	24313.62	0.02 (ms)
	File Random Read Time (2^0 M, local)	480769.23	0.30 (ms)	282767.06	0.18 (ms)
	File Random Read Time (2^7 M, local)	480769.23	0.30 (ms)	255746.18	0.16 (ms)
	File Sequential Read Time (2^0 M, remote)	513888.89	0.32 (ms)	2011161.76	1.26 (ms)
	File Sequential Read Time (2^7 M, remote)	513888.89	0.32 (ms)	2281115.90	1.43 (ms)
	File Random Read Time (2^0 M, remote)	980769.23	0.51 (ms)	2618483.93	1.64 (ms)
	File Random Read Time (2^7 M, remote)	980769.23	0.51 (ms)	2562594.39	1.60 (ms)
	Contention (1 thread)	13889	0.01 (ms)	122976	0.08 (ms)
	Contention (2 threads)	152778	0.11 (ms)	1642507	1.03 (ms)
	Contention (16 threads)	2291670	1.65 (ms)	13001060	8.13 (ms)