CSE 221: System Measurement Project

Tian Yang
University of California, San Diego
tiyang@ucsd.edu
A99109363

Zhihua Gu
University of California, San Diego
z4gu@ucsd.edu
A99110780

1 INTRODUCTION

The goal of this project is to determine Zhihua's MacBook Pro hardware performance characteristics experimentally. We isolate and reveal many aspect of such performance by testing different components of the hardware, such as CPU, memory, network, and file system, using low-cost measuring mechanisms. In doing the experiments on each operation, we first estimate the base hardware performance, and predict the software overhead, then compare our experiment results with the estimation. The assumptions and methodology choices are also discussed here.

The measurements are implemented with C++ with assembly, and are compiled with Apple LLVM version 9.0.0 (clang-900.0.39.2). The compiler flags used are `-std=c++11 -Wall -Wextra -O3`. Certain measures are taken to disable some optimizations at function level. Those are discussed in individual sections.

The tasks of this project is divided as follows: TODO The amount of time spent on this project is estimated to be TODO.

2 MACHINE DESCRIPTION

The machine we are testing on is a 2015 Macbook Pro, with macOS 10.12.6.

The processor is a 2.7GHz Intel Core i5-5257U with 2 Cores. L1 cache size is 128 KB; L2 cache size is 2 x 256 KB; and L3 cache size is 3 MB. This processor features Intel Turbo Boost that increases the frequency up to 3.1GHz[1], which is outside of our control (a third party app[2] is available but it is not code-signed with a Developer ID certificate[3] and contains a kernel extension so we are wary of installing it).

Data transfer in the system uses memory bus Direct Media Interface (DMI2) with 5 GT/s speed instead of a traditional system bus.[1]

The machine features 8GB RAM on 2 DIMM 1867 MHz 4GB memory banks. Disk has 256GB solid state drive flash storage (model SM0256G). The storage has write speeds of 643.6 MB/s and read speeds of 1.3 GB/s.

Wifi card version is Broadcom BCM43xx 1.0 (7.21.95.178.1a2), supporting standards of 802.11 a/b/g/n/ac.

3 METHODOLOGY AND ENVIRONMENT

3.1 Timestamp Reading

As an essential step in performance benchmarking, reading timestamp needs to be as efficient as possible. For operations that takes very little time to execute (such as CPU operations), we used time stamp counter[4] `rdtsc` and `rdtscp` assembly instructions[5] to record the number of clock cycles it has been system started up, and therefore can be used to compute the number of cycles it takes for a program to run.

For other benchmarks that do not need such high granularity and precision (such as network operations), we uses C++'s `std::chrono::high_resolution_clock`.

3.2 Ensure Process Priority

Since we are testing on a power-efficient modern machine where many other processes are running at the same time, we need to take extra step to minimize the impact of optimization and context switches, in order to maintain consistency and accuracy in our benchmarks. We use a Unix command `nice` that invokes a process with a customized priority, with -20 being highest priority and 20 being lowest[6]. Since only superuser can set the priority ("niceness") to a lower value, we write a script `run.sh` to that asks the permission to elevate the privilege of itself, then execute the measurement program as super-user using `nice`, with priority -20.

3.3 Benchmark and Evaluation

A generic `benchmark()` function was created in `util.hpp` to that takes advantage of C++'s Templates and Variadic features to perform different measurements. It takes in the function to be measured with parameters, along with the number of iteration needed, and measures the number of cycles needed for each of the iteration. The result is then processes and presented in the form of variance, maximum deviation, and median. Median is chosen instead of the mean to minimize the influence of outliers.

For most of the experiments, we choose to collect results from 1000 iterations. Slight adjustments were made to some of the measurements, and these exections are discussed in their individual sections.

4 CPU, SCHEDULING, AND OS SERVICES

4.1 Measurement overhead

4.1.1 Estimation. We first evaluate the overhead of read and loop functions. Loop function is expected to execute 4 assembly-level steps: load the counter to register; compare the counter against the iteration bound; increment the counter; jump back to the address at the beginning of the loop. Assuming that each step takes at least one cycle, we estimate that the overhead of each loop is at least 4.

4.1.2 Methodology. First we benchmark the basic read overhead by measuring the number of clock cycles it takes to run an empty function.

Then we measure the loop overhead by iterating through an empty loop 1000 times, and calculate the average time per iteration. We found out by disassemble the compiled program that the empty for loop was eliminated due to compiler optimization. Therefore we labeled the counter `int` variable of the for loop as `volatile`, to prevent compiler from performing the elimination.

4.1.3 Results.

```
mearsureReadOverhead
  stdev: 5.14636 median: 32
mearsureLoopOverhead
  stdev: 1.27861 median: 6.028
```

4.1.4 Analysis. Each loop takes average of 6 cycles, more than estimated. Each step may take more than 1 cycle, so the result is expected.

4.2 Procedure Call Overhead

4.2.1 Estimation. We expect that procedure call overhead increases linearly with the number of arguments, as adding each argument adds a fixed number of instructions. The total number of cycles should also be low, because the only instruction required are saving the stack pointer, and jump to and from the function.

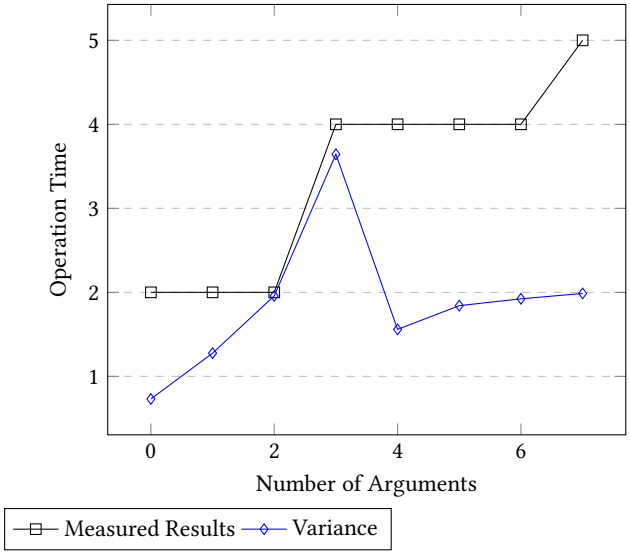
4.2.2 Methodology. In this step of the measurement, we want to visualize how the procedure call overhead changes with the increase of number of arguments passed in the function. We created 7 functions `func0`, $\cdots$, `func7` that take in 0-7 arguments, and have empty function bodies, and benchmarked their performance. Similar to the last section, measures were taken to prevent undesirable compiler optimization: to protect the empty functions from being optimized away, we added function attribute `__attribute__((optnone))` to each of the functions. Clang-specific `#pragma` directives are also used to suppress unused parameter warnings.

4.2.3 Results.

```
0 arg(s)  stdev: 3.35888 median: 2
1 arg(s)  stdev: 2.52113 median: 2
2 arg(s)  stdev: 3.41498 median: 2
3 arg(s)  stdev: 3.42744 median: 4
4 arg(s)  stdev: 3.47347 median: 4
5 arg(s)  stdev: 2.55468 median: 4
6 arg(s)  stdev: 3.57949 median: 4
7 arg(s)  stdev: 4.22762 median: 5
```

Visualization of the results are shown in Figure 1.

Figure 1: Procedural Call Overhead



4.2.4 Analysis. The overhead of procedure call is very small, and does increase somewhat linearly. Comparing to the previous section, where empty loops were executed, these empty functions perform less operations, thus take less cycles to complete.

4.3 System Call Overhead

4.3.1 Estimation. We estimate that the system call would take more than than previously measured procedures, because trapping to the system involves more steps.

4.3.2 Methodology. To accurately measure the overhead of a system call we want a "simplest" system call to measure. Our choice is `clock_gettime(CLOCK_REALTIME, _)`. Since repeated system call can be cached by the OS, and runtime optimized, we picked the `clock_gettime()` which returns different values each time, thus needs to trap every time. Time stamps are taken before and after this call to compute the cycles needed.

4.3.3 Results.

```
measureSystemCallOverhead
  stdev: 521.483 median: 115
```

4.3.4 Analysis. System call does return a much higher median, with a higher variance also. This might be due to the instability of an operation on the kernel level, caused by kernel interruptions and user-kernel mode switching.

4.4 Task Creation time

4.4.1 Estimation. Task creation is done both with kernel-space threads and user-space process forking. We estimate that kernel thread creation will take less time than process creation. That is because a thread shares with other threads their code and data resources, while each process needs to create its own resources.

4.4.2 Methodology. For kernel threads creation, we measured the function `pthread_create(tid, tattr, threadFunc, nullptr)`, where `threadFunc` is a function that simply returns a null pointer.

And for process creation, we measured the time it took to execute `fork()`. A problem that we encountered was that the originally set number of iterations (1000) returns unexpected results, due to machine limitation on the maximum user processes. Too circumvent the problem, we set number of iterations to 700, lower than the max user processes 709. For consistency to compare between thread and process creation, we set the number of iterations to 700 for thread creation as well.

4.4.3 Results.

```
measureKernelThreadOverhead
  stdev: 8351.5 median: 30439
measureProcessCreationOverhead
  stdev: 70723 median: 448039
```

4.4.4 Analysis. As we expected, kernel thread creation takes a lot less time than process creation.

4.5 Context Switch Time

4.5.1 Estimation. In this section, we test and compare the context switch time between both kernel threads and processes. Our estimate is that context switch between processes are more time consuming than that of threads, but because we wouldn't be creating all the new information for a process, simply switching between them, we expect the difference would not be as significant as task creation from the previous section.

4.5.2 Methodology. To measure context switch between threads, we utilized functions `read()` and `write()` through both ends of the pipe to force context switches. We first use `pthread_create()` to create a new thread in which a function writes to the pipe. Immediately afterwards, the program takes a time stamp, then calls `read(pipefd[0], &buff, sizeof(char))`, which blocks the calling thread if writing is not finished, and unblocks and reads from the pipe afterwards. Then we take a final time stamp to calculate the time elapsed. Estimation of a context switch time would be recorded, as a context switch is guaranteed to occur between write and read, in sequential order. To measure context switch time between processes, `fork()` function was used to create a child process, and `read()` and `write()` force context switch. We first start a new process, and once the parent process starts running, we record the time stamp. The process then attempt to read from the pipe, but if the child process has not finished writing to the pipe, it will block the parent process, and wait until write completes, then come back and finish reading. After read, another time stamp is taken. Between the time stamps there is a context switch from the child process to the parent process.

4.5.3 Results.

```
measureProcessSwitchOverhead
  stdev: 90284.8 median: 739335
measureThreadSwitchOverhead
  stdev: 8420.09 median: 23121
```

4.5.4 Analysis. Process switching takes a little less than twice the time of thread switching as expected, but has a higher variance.

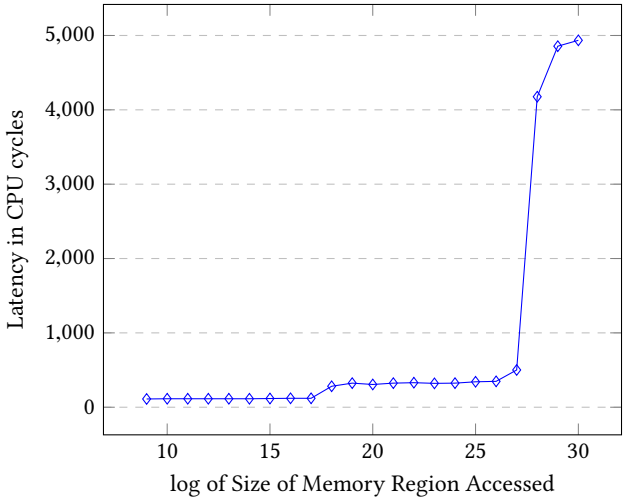
5 MEMORY

In this section, we will measure some of the aspects of performance related to memory, including RAM access time, bandwidth, and page fault time.

5.1 RAM Access Time

5.1.1 Estimation. When measuring memory latency, we assess the result given different memory sizes, because the location where the memory is allocated is based on the size, and thus so does the memory access time. According to the results from McVoy's experiment, L1 cache is accessed until the memory region size is 2^{13} bytes = 8 KB, and L2 cache is accessed until memory size is 2^{19} bytes = 0.5 MB. These correspond to the L1 and L2 cache sizes of the DEC Alpha machine [9].
Similarly, I predict that the L1 cache region would be accessed when 2^{17} byte or less memory is being touched, and L2 cache region would be accessed at size 2^{19}, because of the cache size of the testing system.

Figure 2: RAM Access Time



5.1.2 Methodology. When measuring the memory latency, we measure the "back-to-back-load latency" as described in McVoy's paper, which is measuring each load and "assuming that the instructions before and after are also cache- missing loads" because it is the closest to what developers consider to be latency[9].

The RAM access time is calculated with respect to memory sizes of different orders of 2. The order are chosen as $2^9, \cdots, 2^{30}$, so that each of L1, L2 cache and main memory regions are accessed. For each of the sizes, a memory region corresponding to the memory size is allocated, and pointed to by a pointer. Then we generate a random access point within this memory region by taking a random-sized stride.
The resulting access time (number of cycles) is presented as a function of $\log_2$ of the size of the memory region accessed.

5.1.3 Results. The result is as shown in Figure 2. The first part from 2^9 to 2^{17} corresponds to the L1 cache; the second part 2^{18} - 2^{27} corresponds to L2 cache; and the rest represents main memory.

L1 cache	L2 cache
$2^9 - 2^{17}$	$2^{18} - 2^{27}$

5.1.4 Analysis. The memory regions are indicted in the graph as expected. The transition from L1 to L2 cache due to the 128KB L1 cache size happens at size 2^{17} which is what we predicted. But the transition to main memory happens at a larger memory size (2^26) than what we anticipated. This may be due to the fact that the machine we are testing on have L3 cache which can also be accessed before reaching for main memory, whereas the machine from McVoy's paper only has L1 and L2 cache.

5.2 RAM Bandwidth

5.2.1 Estimation. It is listed on the Intel page [1] that the computer we are testing on has maximum bandwidth 25.6GB/s.
Based on the formula provided on Wikipedia [7], the memory bandwidth is calculated as *Base DRAM clock frequency* × *Number of data transfers per clock* × *Memory bus (interface) width* × *Number*

of interfaces. For the testing system, the above evaluates to:

$$1867MHz \times 2 \text{ lines } \times 64 \text{ bits per line } \times 1 \text{ interface } = 27.8GB/s$$

Since both of these are theoretical optimal values, we predict that our result will fall below this.
We also expect that write will take longer than read.

5.2.2 Methodology. Memory bandwidth is the rate at which the system is able to read or write from memory. Our approach is to allocate some memory pointed to by a pointer and repeated read or write from it, and compute the number of bytes read/written per second.
Below are some of the considerations we made to ensure accuracy of our measurement, and how we formulated the result.

(1) To avoid loop overhead overpowering our test result, we followed the advise from McVoy's paper [9] and performed unrolling by adding loop unroll statement attribute. The command `#pragma unroll` is placed right before the for loop so that the compiler would "directs the loop unroller to attempt to fully unroll the loop if the trip count is known at compile time" [8] to exploit Instruction Level Parallelism.

(2) To further reduce the effect of loop overhead, we choose to use `uint64_t` as a unit of read/write, which is 8 bytes in size. This way, memory access is 8 bytes at a time, instead of 8 individual accesses to each byte. More specifically, the memory is allocated with:
```
auto *ptr = (uint64_t *)malloc(BANDWIDTH_NUM_OF_VAL
                               * sizeof(uint64_t));
```
We choose to read/write 2^{22} times where each time, a piece of memory the size of `uint64_t` is accessed. Therefore, the total memory read/written is $2^{22} \times 8 = 2^{25}$ bytes.

After the time it takes to read/write from memory is recorded, we calculate the bandwidth as follows:

$$\text{Memory Bandwidth(GB/s)} = \frac{2^{25}\text{bytes}}{\text{number of cycles}/2.7GHz}$$

where $2.7GHz$ is the processor base frequency of the test system.

5.2.3 Results.

```
mearsureReadBandwidth
  stdev: 792485 median: 6755400
mearsureWriteBandwidth
  stdev: 1.49604e+06 median: 19140543
```

Then the read and write bandwidth are calculate to be

$$\frac{2^{25}}{6755400/(2.7 \times 10^9)} = 13.411044GB/s$$

and

$$\frac{2^{25}}{19140543/(2.7 \times 10^9)} = 4.73324954GB/s$$

Read Bandwidth	Write Bandwidth
13.4 GB/s	4.73 GB/s

5.2.4 Analysis. The result from our measurement is noticeably less than the theoretical maximum speed calculated earlier. One reason for this might be the fact that memory reads involves multiple sources of overhead, such as the checks for page validity and protection as mentioned in Levy's VAX/VMS paper [10].

In addition, we are not able to prevent system background processes from accessing memories during our benchmark runs.

We use `volatile` keyword to prevent compiler from eliminating the read/write operations. Write might suffer more from the side effects since the memory pointer is marked as `volatile`. As a result, the compiler might be more conservative in applying optimization passes.

5.3 Page Fault Service Time

5.3.1 Estimation. When a page fault happens, the system needs to perform a disk access then transfer the corresponding data over, and our estimation is based on this.
From Wikipedia (original reference is now dead link), a typical random access time from SSD is 0.1ms [12]. Data transfer rate on the test machine is 1.3GB/s, and a page size is 4KB. Therefore, we estimate the page fault service time to be

$$0.1ms + \frac{4kB}{1.3 \times 1024^2 kB/s} = 102.9\mu s$$

5.3.2 Methodology. We measure page fault time by first creating and filling a large temporary file programmatically, and then uses `mmap` to map the file content to memory addresses. In this way, when we access the memory addresses that belong to the file, the system will need to load the corresponding content from disk to memory, exactly like a hard page fault.
In order to perform one page fault each time we access the file, we make sure that each memory access is at least a page size apart. The page size is obtained by calling `getpagesize()` which returns 4096 bytes for the test system.

5.3.3 Results.

```
measurePageFaultService
    median: 290824
```

The resulting page fault service time is:

$$\frac{290824 \text{ cycles}}{2.7 \times 10^9 Hz} = 107.71\mu s/page$$

5.3.4 Analysis. The actual result is slightly higher than our estimates. This might due to the software overhead involved, such as trapping to kernel.

6 NETWORK

We test the network performance of TCP protocol both locally and between remote interfaces. Therefore, we performed the following experiments on round trip time, peak bandwidth, and connection setup/teardown time and obtained both loopback and remote results.

The remote test machine is the login node of UC San Diego's Sorken cluster. It is a Dell PowerEdge 1950 server with dual 8-core Intel Xeon E5-2640 v3 2.6GHz CPUs and 64GB of RAM. The Sorken cluster runs Rocks Cluster Linux 6.2 with g++ 4.8.4. Since this is a UCSD machine and we performed our test while connected to the

UCSD-PROTECTED wireless network, we are able to avoid many of the factors that affect Internet connection quality (e.g. peering arrangement and routing change) that is outside of our control.

6.1 Round Trip Time

6.1.1 Estimation. In this section, we measure the round trip time of loopback and remote connections, with both TCP and ICMP protocols, and compare them.

We predict that loopback RTT will be siginificantly faster than remote RTT since in the former scenario the packects never need to leave the machine and go on the network.

We also predict that the ICMP generally takes shorter time than TCP, because the ICMP does a quicker transfer at the expense of reliability. For example, a common cause of packet drops "does not elicit any ICMP information", while TCP handles such situation[16].

In relation to network distance, we predict that the difference between ICMP and TCP is larger when the network distance increases. More specifically, we would see a larger difference between ICMP and TCP measurements (with ICMP being faster) in remote connection measurements, than in loopback connection measurements. Previous research indicates: "For small delay paths, the mean RTTs measured using Tping have no significant difference compared with which measured using Iping", and "while for large delay paths, the mean RTTs measured using TCP is larger than which measured using Iping with tens of millisecond" [17].

6.1.2 Methodology. For the ICMP round trip time, the time to perform a `ping` command to both loopback interface and to the remote machine is recorded. Loopback result is obtained using the command:

```
ping -c 100 127.0.0.1
```

Remote server result is obtained using the command:

```
ping -c 100 100.81.33.188
```

The command sends 56 data bytes to the provided IP address, and receive back the same data along with 8 bytes of ICMP header. To perform a fair comparison, we also send 64 bytes data through TCP connection. A client and a server are set up to first establish a connection, then the time to setup connection, and to perform a `send()` and `recv()` is measured on the client side, while the server side accepts the connection and send back the same 64 bytes data it receives. We ran the test by first start running a server, (code in in `utilities/Server.cpp` and separately compiled using the same flags described in Section 1), on port 5001, then run our testing client to connect to that server, by running the measurement script `./run.sh`.

All four experiments are performed 100 times, and the results are compiled as follows.

6.1.3 Results.
Results of TCP RTT:

```
mearsureRTT (ns)
127.0.0.1
  stdev: 68134.9 median: 89983
mearsureRTT (ns)
132.239.95.222
  stdev: 339440 median: 2912135
```

Results of ICMP ping:
Loopback:

```
>> ping -c 100 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1:
                    icmp_seq=0 ttl=64 time=0.073 ms
...
--- 127.0.0.1 ping statistics ---
100 packets transmitted, 100 packets received
round-trip min/avg/max/stddev
                       = 0.057/0.083/0.194/0.016 ms
```

Remote interface:

```
>> ping -c 10 132.239.95.222
PING 132.239.95.222 (132.239.95.222): 56 data bytes
64 bytes from 132.239.95.222:
                    icmp_seq=0 ttl=61 time=1.572 ms
...
--- 132.239.95.222 ping statistics ---
10 packets transmitted, 10 packets received
round-trip min/avg/max/stddev
                       = 1.265/1.372/1.672/0.129 ms
```

	Loopback(ms)		Remote(ms)	
	avg	std	avg	std
TCP	0.089983	0.0681349	2.912135	0.33944
ICMP Ping	0.083	0.016	1.372	0.129

6.1.4 Analysis. As we predicted, ICMP perform `ping()` faster than TCP in each situation. With loopback measurements, the difference is small, whereas remote measurements indicates that TCP takes two times as long to do a round trip message passing.

If we compare the RTT for to the loopback address and a remote address, we can see the obvious result that loopback RTT is magnitudes faster than remote RTT, just as we predicted.

6.2 Peak Bandwidth

6.2.1 Estimation. We predict the peak bandwidth using the measured round trip time from last section:

$$Loopback = \frac{64 \text{ bytes } * 2}{0.08998ms} = 1.4MB/s$$

$$Remote = \frac{64 \text{ bytes } * 2}{2.9121ms} = 0.044MB/s$$

We foresee that the actual peak bandwidth will be much higher, because sending larger data chunks will reduce the effect of connection overhead.

6.2.2 Methodology. We measure the peak bandwidth by measuring the time it takes to receive a fixed size 1MB data package over many iterations, and compute its median.

The implementation is similar to last section. We set up a simple server and run our client on it. We ran the test by first start running a server, (code in `utilities/BandwidthServer.cpp`, and separately compiled using the same flags described in Section 1), on port 5002, then run our testing client to connect to that server, by running the measurement script `./run.sh`.

6.2.3 Results.
Loopback:
Remote Interface:

```
mearsureBandwidth (ns)
127.0.0.1
  stdev: 92457 median: 359785
mearsureBandwidth (ns)
132.239.95.222
  stdev: 2.26838e+07 median: 459446055
```

Loopback	2.77 GB/s	Remote	0.02 GB/s

6.2.4 Analysis. The peak bandwidth is, as we predicted, much higher than the theoretical prediction from RTT. This could be because of the protocol overhead of TCP, which is heavily rooted in TCP's focus on reliability. TCP connection checks the receipt of a message, and resend if they message wasn't received. This feature compromises speed when establishing connection and when transmitting. Since the measurements in this section sends data a lot bigger than during RTT testing, this overhead is less noticeable.

6.3 Connection Overhead

6.3.1 Estimation. Similarly, we use the RTT measured in Section 6.1.2 to estimate the connection overhead. Round trip time is the time to set up the connection and to send and receive back a small message, measured on the client side. TCP connection is setup with a three-way handshake, and together with the round trip of the actual message, constitute 5 transfers. Therefore I predict the set up time to be 3/5 of RTT:

$$Loopback\ setup = \frac{3}{5} \times 89983 = 53989.8ns$$

$$Remote\ setup = \frac{3}{5} \times 2912135 = 1747281ns$$

For teardown, the process should be faster because the client only need to send out the FIN message, without waiting for an response.

6.3.2 Methodology. We use the server used in earlier section `utilities/Server.cpp`, and only establish connection and tear it down. After the server starts up and be ready by calling `socket()`, `bind()`, `listen()`, and `accept()`, we measure the time it takes for the client to run `socket()` and `connect()`, which is the three-way handshake process to set up connection.

For tear down, we first set up the connection in the same way, then measure the time to complete `close()` which tears down the established connection.

6.3.3 Results.

```
mearsureSetupOverhead (ns)
127.0.0.1
  stdev: 24722.4 median: 45498
mearsureSetupOverhead (ns)
132.239.95.222
  stdev: 518549 median: 1494980
mearsureTearDownOverhead (ns)
127.0.0.1
  stdev: 2710.71 median: 6034
mearsureTearDownOverhead (ns)
132.239.95.222
  stdev: 6984.62 median: 20734
```

	Loopback(ns)	Remote(ns)
Setup Overhead	45498	1494980
Teardown Overhead	6034	20734

6.3.4 Analysis. As we predicted, the setup times correlates to RTT measured earlier, and teardown time is faster because we are only testing the time it takes to run `close()` on the client side.

7 FILE SYSTEM

7.1 Size of File Cache

7.1.1 Estimation. We performed empirical testing by checking the statistics on macOS' Activity Monitor app, which displays the size of "Cached Files" in memory. We closed all but one applications to reduce the amount of "Memory Used" (by applications), and opened as much big local files as possible to increase the size of "Cached Files", we saw the maximum of 3.3GB of cached files. So we assume the theoretical lower bound of file cache is 3.3GB. The testing machine has 8GB of RAM, which is the loose upper bound.

7.1.2 Methodology. When a file is written to disk, it is cached. We exploit this property to determine the size of file cache by writing increasingly large files to disk and timing the subsequent read time to calculate and compare the normalized results. The result is then calculated by dividing the number of bytes of the file, to obtain the read time per byte of each file size.

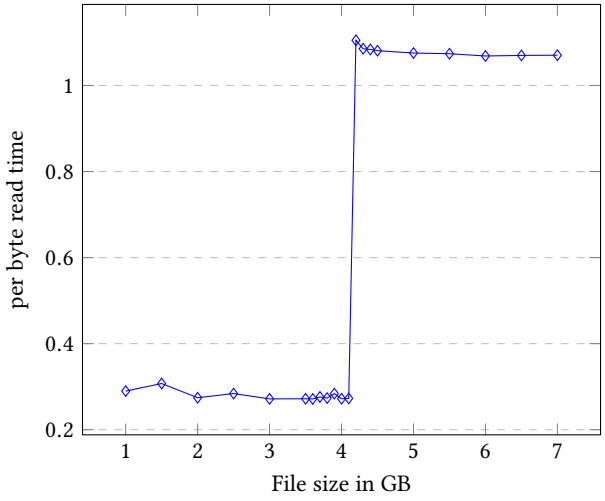
The first group of iteration includes 1 – 7GB of file written and read, with 0.5 GB step size. Then we performed another iteration of 3.5 – 4.5GB of file size, with 0.1GB step size to obtain a more detail on part of the result.

7.1.3 Results. The result is plotted in Figure 3.

```
mearsureFileCacheSize (ns)
  size: 1073741824  stdev: 0.193401 median: 0.289734
  size: 1610612736  stdev: 0.21363 median: 0.307312
  size: 2147483648  stdev: 0.256739 median: 0.274281
  size: 2684354560  stdev: 0.225854 median: 0.284034
  size: 3221225472  stdev: 0.230289 median: 0.271604
  size: 3758096384  stdev: 0.00995019 median: 0.279358
  size: 4294967296  stdev: 0.248127 median: 0.270789
  size: 4831838208  stdev: 0.00810092 median: 1.08085
  size: 5368709120  stdev: 0.00439324 median: 1.07544
  size: 5905580032  stdev: 0.00486344 median: 1.07378
  size: 6442450944  stdev: 0.00290641 median: 1.06858
  size: 6979321856  stdev: 0.00356847 median: 1.06986
  size: 7516192768  stdev: 0.00504614 median: 1.07037

mearsureFileCacheSize (ns)
  size: 3758096384  stdev: 0.241628 median: 0.271887
  size: 3865470566  stdev: 0.00221888 median: 0.271466
  size: 3972844748  stdev: 0.00415491 median: 0.276136
  size: 4080218930  stdev: 0.00276407 median: 0.273519
  size: 4187593112  stdev: 0.0106033 median: 0.283844
  size: 4294967294  stdev: 0.00169286 median: 0.271974
  size: 4402341476  stdev: 0.246867 median: 0.272774
  size: 4509715658  stdev: 0.0155539 median: 1.10497
```

Figure 3: Read time with increasing file size




```
size: 4617089840  stdev: 0.00499582 median: 1.08541
size: 4724464022  stdev: 0.0165208 median: 1.08375
size: 4831838204  stdev: 0.00538228 median: 1.08235
```

7.1.4 Analysis. We observe that there is a drastic increase of read time per byte between 4.1 – 4.2 GB file size, and conclude that the file cache size is approximately 4.1GB. This is because if the file written is smaller than a certain size, it would be completely cached, and then reading the file would only require accessing cached data, which is very fast. Then at one point, the file size exceeds the file cache size, then part of the file written to disk would not be in cache, which would be the first part of the file because it is written first. Then, reading sequentially from the beginning would mean fetching at least part of the data from disk instead of cache.

7.2 File Read Time

7.2.1 Estimation. From the results of previous section, we predict that the file read time would be at least the read time of the not cached portion, which is around $1ns/byte = 4096ns/block$. We also predict that random file read time is higher than sequential file read time, because of cache line prefetching.

7.2.2 Methodology. To measure the file read time, we first write a sufficiently large temporary file to disk. To avoid the file from being cached, we make use of macOS's `fcntl(2)` system call to set `F_NOCACHE` to a non-zero value on the file descriptor to turn off data caching when writing. When reading, we additionally set `F_RDAHEAD` to `0` to disable read ahead for accurate measurement [18].

The measurements is repeated for files with increasing sizes (in powers of 2), then the time in ns to read each block is calculated, where a block size is 4096 bytes.

7.2.3 Results. The results are presented in Figure 4.

```
mearsureFileReadTime (ns)
  size: 2^0 MB
    seq:   stdev: 13468.8 median: 46678.1
    rand:   stdev: 3179.76 median: 108529
  size: 2^1 MB
    seq:   stdev: 2093.15 median: 45767.7
    rand:   stdev: 2248.67 median: 109674
  size: 2^2 MB
    seq:   stdev: 2373.33 median: 47422.7
    rand:   stdev: 1703.38 median: 109614
  size: 2^3 MB
    seq:   stdev: 2011.22 median: 46894.1
    rand:   stdev: 1983.99 median: 113424
  size: 2^4 MB
    seq:   stdev: 1604.71 median: 48766
    rand:   stdev: 2703.27 median: 111814
  size: 2^5 MB
    seq:   stdev: 871.144 median: 48721.4
    rand:   stdev: 823.097 median: 110625
  size: 2^6 MB
    seq:   stdev: 1174.22 median: 50242.6
    rand:   stdev: 928.531 median: 112510
  size: 2^7 MB
    seq:   stdev: 580.113 median: 49119.1
    rand:   stdev: 1197.89 median: 112802
  size: 2^8 MB
    seq:   stdev: 1069.09 median: 50738.2
    rand:   stdev: 779.069 median: 112495
```

7.2.4 Analysis. We observe that as predicted, the sequential read time is shorter than random read time, and that neither of them fluctuate much around their average with increasing file size read. The former is because random read requires a seek to be performed, whereas sequential does not. The latter is because we already prevented memory cache, as discussed in the methodology.

The sequential access in a sense is not exactly "sequential" because of the way test machine's file system is structured. Legacy macOS' file system, HFS+ (Hierarchical File System Plus) uses volume bitmaps where allocation of blocks are documented. This feature reduces head travel when growing files, but randomize file location, as discussed by HFS+ archive: "The allocation file does not have to be contiguous, which allows allocation information and user data to be interleaved."[15].

7.3 Remote File Read Time

7.3.1 Estimation. For this section, the same measurement was performed as the previous section, but with a remote file system. Therefore, we predict that the change in file read time would be caused by network overhead. Thus, our predict is based on behavior of remote connection bandwidth which was measured in 6.2 Peak Bandwidth. The peak bandwidth of loopback (1.4MB/s) is 35 times that of remote connection (0.044MB/s). Since each remote file read depend on the connection peak bandwidth, we predict a similar multiplier on top of local file read time (around $1 \times 10^5 ns$):

$$\text{Predicted remote read time} = 35{\times}1{\times}10^5 ns/block = 3.5{\times}10^6 ns/block$$

However, since remote file access performs with less protocol overhead than TCP connection measured in section 6. Therefore, our prediction is less than $3.5 \times 10^6 ns/block$.

Figure 4: Sequential and random file read time

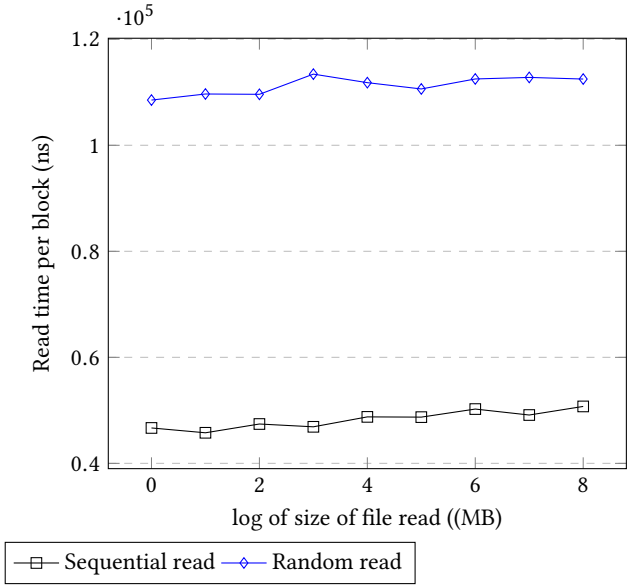
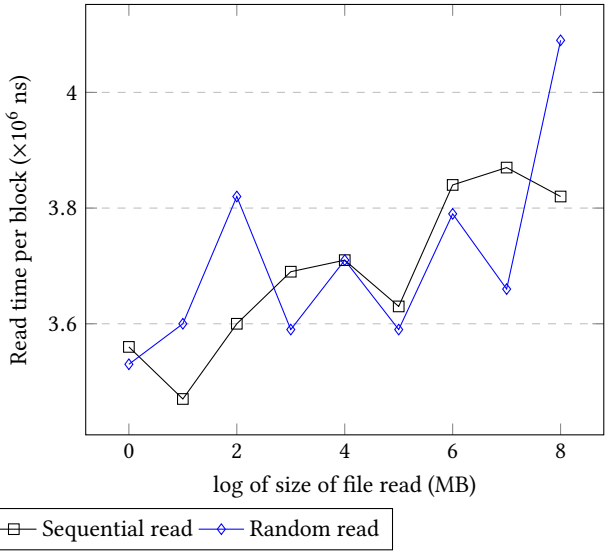


Figure 5: Remote file read time



7.3.2 Methodology. A directory on another computer running macOS is shared via the Server Message Block protocol (version 3.02 with signing enabled) and mounted on the benchmark target computer as `/Volume/smb/`. While NFS was suggested for use for this benchmark, on modern macOS versions, SMB is the natively supported protocol is we opted for SMB.

Same as the local version, to measure the file read time, we first write a sufficiently large temporary file to disk. To avoid the file from being cached, we make use of macOS's `fcntl(2)` system call to set `F_NOCACHE` to a non-zero value on the file descriptor to turn off data caching when writing. When reading, we additionally set `F_RDAHEAD` to `0` to disable read ahead for accurate measurement [18].

7.3.3 Results. The result of this experiment is plotted in Figure5.

```
mearsureFileReadTime (ns)
  size: 2^0 MB
    seq:   stdev: 199088 median: 3.55872e+06
    rand:   stdev: 123425 median: 3.52701e+06
  size: 2^1 MB
    seq:   stdev: 131704 median: 3.47254e+06
    rand:   stdev: 147305 median: 3.60445e+06
  size: 2^2 MB
    seq:   stdev: 239640 median: 3.60134e+06
    rand:   stdev: 217349 median: 3.82488e+06
  size: 2^3 MB
    seq:   stdev: 90687.6 median: 3.69081e+06
    rand:   stdev: 79199.8 median: 3.59297e+06
  size: 2^4 MB
    seq:   stdev: 121633 median: 3.70633e+06
    rand:   stdev: 165603 median: 3.71652e+06
  size: 2^5 MB
    seq:   stdev: 292245 median: 3.62986e+06
    rand:   stdev: 152144 median: 3.59049e+06
  size: 2^6 MB
    seq:   stdev: 170419 median: 3.84217e+06
    rand:   stdev: 410992 median: 3.79536e+06
  size: 2^7 MB
    seq:   stdev: 186451 median: 3.87079e+06
    rand:   stdev: 210538 median: 3.65797e+06
  size: 2^8 MB
    seq:   stdev: 222326 median: 3.82113e+06
    rand:   stdev: 262324 median: 4.0892e+06
```

7.3.4 Analysis. The performance of remote file read over SMB is unexpectedly poor. Upon research, we identified several contributing factors.

The fact that we are reading one 4096-byte block at a time works good for local read but not remote read over SMB. Such a large number of read calls added a significant amount of overhead at every protocol level.

SMB 3's packet signing also impacted the performance [19] but is necessary to ensure security.

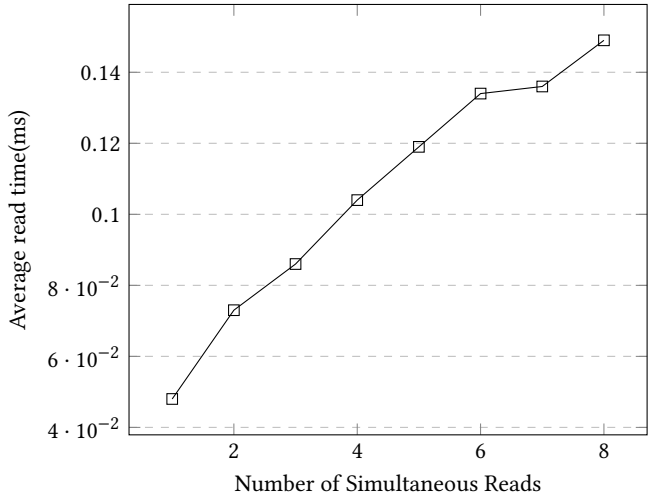
We performed our measurements while the two computers are in close proximity physically and while both are connected to the UCSD-PROTECTED network in a public area. Thus, the remote read performance is likely impacted by the fluctuating wireless channel utilization level and by the airtime fairness feature [13] that is likely enabled on the wireless access point.

Unlike local file reads, the difference between sequential and random read times are not pronounced. This is likely due to the fact that transmit time dominates the overall time.

7.4 Contention

7.4.1 Estimation. We predict that, as the number of sequential read increases, performance degrades. This is because context switch would occur with each time the system switch between one

Figure 6: Average read time under contention



read and another. With more number of simultaneous read, more prefetched data would need to be flushed and overwritten with context switch. Therefore, it is likely that the file read time would grow proportionally with the number of reads performed at the same time.

7.4.2 Methodology. To benchmark read contention's impact on file read time, we first write a number of temporary files (again, with `F_NOCACHE` enabled) and then issues different number of simultaneous reads (again, additionally with `F_RDAHEAD` disabled), each to a different file that we have just written. We measure and calculate the average time for the reads to complete.

7.4.3 Results.

```
mearsureReadContention (ns)
 num of simultaneous read: 1  stdev: 1764.41 median: 48083.1
 num of simultaneous read: 2  stdev: 4646.59 median: 73320.8
 num of simultaneous read: 3  stdev: 9115.58 median: 85887.1
 num of simultaneous read: 4  stdev: 9830.67 median: 104135
 num of simultaneous read: 5  stdev: 9860.79 median: 119022
 num of simultaneous read: 6  stdev: 3906.58 median: 133938
 num of simultaneous read: 7  stdev: 1123.33 median: 136223
 num of simultaneous read: 8  stdev: 2448.11 median: 149301
```

7.4.4 Analysis. We can observe from the results that as number of sequential read increases, performance indeed suffers, even if the each read request is to a different file. As the number increases, additional costs incur, including context switch cost, in addition I/O bandwidth bottlenet.

REFERENCES

[1] Intel *Intel Core i5-5257U Processor.* https://ark.intel.com/products/84985/Intel-Core-i5-5257U-Processor-3M-Cache-up-to-3_10-GHz
[2] *Turbo Boost Switcher for OS X.* https://www.rugarciap.com/turbo-boost-switcher-for-os-x/
[3] Apple. *Developer ID and Gatekeeper.* https://developer.apple.com/developer-id/
[4] https://en.wikipedia.org/wiki/Time_Stamp_Counter
[5] Intel. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures.* https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf
[6] Apple. *BSD General Commands Manual NICE(1.* https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man1/nice.1.html
[7] https://en.wikipedia.org/wiki/Memory_bandwidth
[8] The Clang Team. *Attributes in Clang.* https://clang.llvm.org/docs/AttributeReference.html#pragma-unroll-pragma-nounroll
[9] Larry McVoy and Carl Staelin. *lmbench: Portable Tools for Performance Analysis.* http://www.usenix.org/publications/library/proceedings/sd96/full_papers/mcvoy.pdf
[10] Henry M. Levy and Peter H. Lipman. *Virtual Memory Management in the VAX/VMS Operating System.* http://cseweb.ucsd.edu/~voelker/cse221/papers/vms.pdf
[11] https://en.wikipedia.org/wiki/Page_fault#Invalid_conditions
[12] https://en.wikipedia.org/wiki/Solid-state_drive#cite_ref-105
[13] Cisco. *Air Time Fairness(ATF) Phase1 and Phase 2 Deployment Guide.* https://www.cisco.com/c/en/us/td/docs/wireless/technology/mesh/8-2/b_Air_Time_Fairness_Phase1_and_Phase2_Deployment_Guide.html
[14] Scott Baden. *CSE260 Fa15 - Getting Started With Sorken.* http://cseweb.ucsd.edu/classes/fa15/cse260-a/Sorken.html
[15] Apple. *HFS Plus Volume Format.* https://developer.apple.com/library/archive/technotes/tn/tn1150.html#AllocationFile
[16] https://notes.shichao.io/tcpv1/ch8/
[17] Li Wenwei, Zhang Dafang, Yang Jinmin, Xie Gaogang. *On evaluating the differences of TCP and ICMP in network measurement.* https://pdfs.semanticscholar.org/fd14/cbb8ad261f12c479f21953fd0c5dfac1b574.pdf
[18] Apple. *BSD System Calls Manual FCNTL(2).* https://www.unix.com/man-page/osx/2/fcntl/
[19] Apple. *Turn off packet signing for SMB 2 and SMB 3 connections.* https://support.apple.com/en-us/HT205926