UNIVERSITY OF CALIFORNIA, SAN DIEGO
(CSE 221)

PERFORMANCE MEASUREMENT FOR MACOS OPERATING SYSTEMS

PROJECT REPORT

Name: Youliang Liu Student ID: A14003277
Name: Yang Liu Student ID: A59003224
Name: Yixiang Wang Student ID: A13713889

Date: 27 Oct 2020

Contents

1 Introduction

All three members in our team uses Mac as our choice of laptop. We have used this system extensively but still do not understand this operating system enough to claim that we really know this system. This time, we decide to really get to know what kind of a system Mac OS is and what is it capable of doing.

We majorly used C/C++ to implement our measurements because of it's simplicity, speed and ease of use regarding system calls. We compiled our code with Clang (clang-1200.0.32.27) which comes in bundle with current version of Xcode (Version 12.2). Although we all use Apple laptops, we use different makes from different years. We decide on using the newest Macbook Pro we can put our hands on. Based on the current decision, we will run and experiment with our code on our own system and then using that specific configuration to generate real data and analyse them. Each group member spend around 30 hours for this project and we are glad that we can perform some benchmark on the operating system what we used every day to have a better understanding of it.

1.1 Github Url

https://github.com/youliangliu/MacOS-performance

2 Machine Description

Table 1: Add caption

Processor	
Model	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
Clock speed	2.60 GHz
CPU	1
Core	6
CPU Architecture	x86_64
L1d cache size	32K
L1i cache size	32K
L2 cache size	256K
L3 cache size	12M
Memory Bus	
Type	DDR4
Speed	2667MHz
Total Width	64 bits
Data Width	64 bits
Disk	
Type	512G SSD
Name	APPLE SSD AP0512N
Protocol	PCI-Express
Other	
RAM size	16G RAM
Network	Wi-Fi: AirPort
Operating System	macOS Big Sur 11.0.1

3 CPU, Scheduling, and OS Services

3.1 Measurement Overhead

3.1.1 Read Overhead Methodology

When measuring the overhead of reading time, use the configuration of the hardware to calculate how long does it take to have the read instruction executed on the hardware. Go through each software stage of the read operation and estimate the software overhead in each stage. Run a program that performs a single read operation and use time stamp counter to record the time spend. Use the measured time minus the hardware execution time to get the software overhead.

3.1.2 Read Overhead Implementation

We performed two consecutive rdtsc() instruction and then calculate the clock cycle difference between these two rdtsc to estimate the read overhead. We used a loop to perform this measurement 1,000,000 times and

then get the average overhead to reduce the side effect of randomness. And we performed this looped measurement 10 times and write the result to the file read_overhead.txt to see if there is a trend in read overhead.

3.1.3 Read Overhead prediction

According to a post in the Intel Community website, the rdtsc() read overhead should be arount 25 to 35 cycles.

https://community.intel.com/t5/Software-Tuning-Performance/High-impact-of-rdtsc/td-p/1092539.

3.1.4 Read Overhead Result

	1	2	3	4	5	6	7	8	9	10
Cycle	32.143	32.197	24.162	24.038	24.019	24.443	24.153	24.030	24.775	24.726

3.1.5 Analysis

The result is pretty close to what we predicted, we can see the read overhead is a little higher in the beginning, then it dropped to a consistent cycle count quickly, we think this is because of caching.

3.1.6 Loop Overhead Methodology

When using a loop to measure many iterations of an operation, determine how many more instructions was added by using a loop and how the CPU execute those instructions and the estimated time needed to execute those instructions. Also estimate the software overhead added by the loop by examine every software stage. Measure the execution time using the time stamp counter, divide the measured time by iteration counts. Use the measured time minus the hardware execution time to get the overhead by using a loop.

3.1.7 Loop Overhead Implementation

We implemented a loop that will loop for 1,000,000 times and do nothing inside the loop so that we can eliminate any other overheads other than loop overhead. We use rdtsc() instruction to record the clock cycle right before the loop and right after the loop and then calculate the difference to determine how many cycles does the loop took. Then we divide the total cycle count by the number of iterations to get the loop overhead of a single iteration.

3.1.8 Loop Overhead Prediction

Since for each iteration of the loop, we are comparing the index counter and then increment the index counter, and an ALU operation should take just 1 cycle to complete, so that the loop overhead should be about 2 cycles.

3.1.9 Loop Overhead Result

	1	2	3	4	5	6	7	8	9	10
Cycle	3.941	3.493	3.180	2.666	2.624	4.685	3.606	3.819	3.502	2.598

3.1.10 Analysis

The resulting overhead is higher than what we predicted. We think this can be resulted from we are using the rdtsc instruction to record the cycle and we are running our benchmark on x86 architecture intel machine, in which the intel machine will re-order the instruction order and sometime it will start the rdtsc() instruction earlier for optimization. For the loop instruction there must be some form of optimization so that the instruction we wrote are not exactly executed in the order we wanted so that more overheads resulted.

3.2 Procedure Call Overhead

3.2.1 Procedure Call Overhead Methodology

For the procedure call overhead we want to examine how much overhead will be caused by the increment of number of parameters into the function. We plan to create 8 test functions with parameter count range from 0 to 7. We would run these 8 test functions for 10,000 times and use the time stamp counter to record the cycle used, and calculate the execution time. We divide the measured time by 10,000 to get the average time. We compare the average measured time of these 8 test functions to evaluate the increment overhead of an argument.

3.2.2 Procedure Call Overhead Implementation

We create 8 functions that has parameter number ranging from 0 to 7. Then we use rdtsc() instruction to count the cycle before and after the function call to record the function call overhead.

3.2.3 Procedure Call Overhead Prediction

We should see an increase in the result as the number of the parameter increases.

3.2.4 Procedure Call Overhead Result

Parameter Count	0	1	2	3	4	5	6	7
Cycle	26.411	26.094	26.386	26.292	27.173	27.040	26.091	26.539

3.2.5 Procedure Call Overhead Analysis

We didn't see the increase as we predicted, this might be because we use all parameters are an integer type, and integer type is relatively small and it does not create much overhead when comparing to some other factors like the re-ordering of instruction. And after averaging out the randomness, we get very close result to functions with different number of parameters. Another factor can be that we don't have any actual implementation to those functions. Usually more parameters can result to more local variables and more complicated implementations, but since we don't have a implementation to the function, those factors doesn't affect our result.

3.3 System Call Overhead

3.3.1 System Call Overhead Methodology

Since system call can be cached by the OS, we would fork a new process to induce a minimal system call. We would use a time stamp counter to count the clock cycle times right before the system call and right after the system call was complete. We expect system call to induce more overhead than the procedure call since system call calls into the kernel mode, so we need an interrupt to switch from user mode to kernel mode to execute the system call and switch back to the user mode. This process will cause a lot of overhead.

3.3.2 System Call Overhead Implementation

The implementation of our system call overhead function is very similarly to our procedure call function. Instead of calling the function we implemented, we called getpid() which is a system function that will trap to the kernel to get the pid of the process. We still used rdtsc() to record the overhead in cycles.

3.3.3 System Call Overhead Prediction

Since we are trapping to the kernel, we are expecting to have a higher overhead than the procedure call overhead.

3.3.4 System Call Overhead Result

	1	2	3	4	5	6	7	8	9	10
Cycle	29.098	28.081	23.646	24.885	24.113	23.968	23.936	23.738	23.796	23.989

3.3.5 System Call Overhead Analysis

From the result, the first two system call is higher than the average procedure call, then it dropped to a lower level. We think this is because of the system cached the result since we are getting the pid number from the same process.

3.4 Task creation time

3.4.1 Process Creation Time

We use fork() to create a new process. We use RDTSC to acquire time stamps before forking the process and ones after forking. Noticeable, we acquire two timestamps after forking, one in child process and one in parent process. We choose the smaller timestamps as end time because the smaller timestamps represents the scheduled process right after fork. By eliminating the overhead of reading time and process scheduling overhead, we can acquire the time stamps for creating a new process

When creating a new process, OS conducts following operations:

- Generate a new PID and revise process tables.
- Allocate memory space for the new process, including program, data and stack. The space for Process Control Block(PCB) is also allocated here.

- Initialize values in PCB, such as stack pointer, program counter...
- Prepare the process for scheduling.

Noticeably, the forked process only create a copy-on-write copy of data, so OS does not copy the data of original process into new memory space when forking a new process.

3.4.2 Kernel Thread Creation Time

The kernel thread creation is almost same as creating a process in Linux. Both fork() and pthread_create() call a same function: clone(). While processes does not allow memory share and create a copy-on-write copy, threads simply share the memory space.

We use similar method in measuring process creation time. We use RDTSC to acquire time stamps before creating a new thread and ones after entering the thread. By eliminating the overhead of reading time and thread scheduling overhead, we can acquire the time stamps for creating a new kernel thread.

3.4.3 Prediction

The creation time of a process and thread really depends on the data space the original process or thread have. The time also depends on other running processes, which might occupy too much CPU time. We can only predict empirically that process creation time will be much larger than thread creation time. The process creation time is usually larger than 0.1ms.

3.4.4 Result

Type	Mean(clks)	Mean(time)	Std. Deviation (time)
Process creation	729742	0.281ms	0.046ms
Thread switch	45045	0.0173ms	0.0035ms

3.4.5 Analysis

The results accord with our expectations: process creation time is much larger than thread creation time, and process creation time is more than 0.1ms.

However, We cannot ensure that there are no other process being scheduled after forking the child process. The only solution is implement a system call and change the scheduler of the kernel, which is infeasible on OS X. Also, when reading rdtsc(), we cannot ensure that there are no other process being switched to.

3.5 Context switch time

In this part, we consider the context switch between processes and kernel thread. When context switching, the state of original process is stored, and state of another process is loaded. The switching of kernel thread is similar to process, since the only difference is that processes have a copy-on-write copy, while threads have shared memory. Threads do not need to change name space.

3.5.1 Methodology

We measure the context switch time by measuring the timestamps before child process write to a pipe and timestamps after parent process read the pipe. We force child process to yield resources by calling sleep after child process write to the pipe. Since processes other than parent process might also be scheduled after child process, we set a threshold on the difference of timestamps to abandon values that are obviously too large. We conduct the measurements for 10 times.

The measurements for thread switching is similar to ones of processes.

3.5.2 Prediction

We make prediction based on task creation time. The task creation includes a context switch. Also, we believe copying process or thread states should occupy a great portion of task creation time. Also, process switching time should be much larger than thread switching time, because thread switching does not need change name space.

3.5.3 Result

Type	Mean(clks)	Mean(time)	Std. Deviation (time)
Process switch	153678	0.0591ms	0.0101ms
Thread switch	14155	0.0054ms	0.0019ms

3.5.4 Analysis

The results show that process switching time is much larger than thread switching time, which is to our expectation. However, the process switching time is much smaller than process creation time. The reasons might be that writing to process table when creating new process might be more time-consuming than our estimation.

Some other factors might increase the inaccuracy of the measurements:

- We measure the time before and after writing or reading the pipe, so the time for writing to and reading from the pipe is also included in the measurements.
- Still, we cannot ensure that thread and process is switched between child and parent process. Other processes are also invloved in scheduling.

4 Memory

4.1 RAM access time

4.1.1 Methodology

According to the Imbench paper, latency is an often-overlooked area of performance problems and it is hard to solve. And the first step toward improving latency is understanding the current latencies in a system, so that in this section we will measure the latency for individual integer access to main mamory and the L1 and

L2 caches to have a better understanding of our current system.

According to the Imbench paper, there are four types of most common memory read latency definitions. In increasing time order, they are memory chip cycle time, processor-pins-to-memory-and-back time, load-in-a-vacuum time, and back-to-back-load time. Due to various reasons, the author of the Imbench paper believes that the back-to-back-load time definition provides the result that most software developers consider as memory latency and it can be easily measured from software. So we will use back-to-back-load time to measure the performance of MacOS on our machine.

4.1.2 Implementation

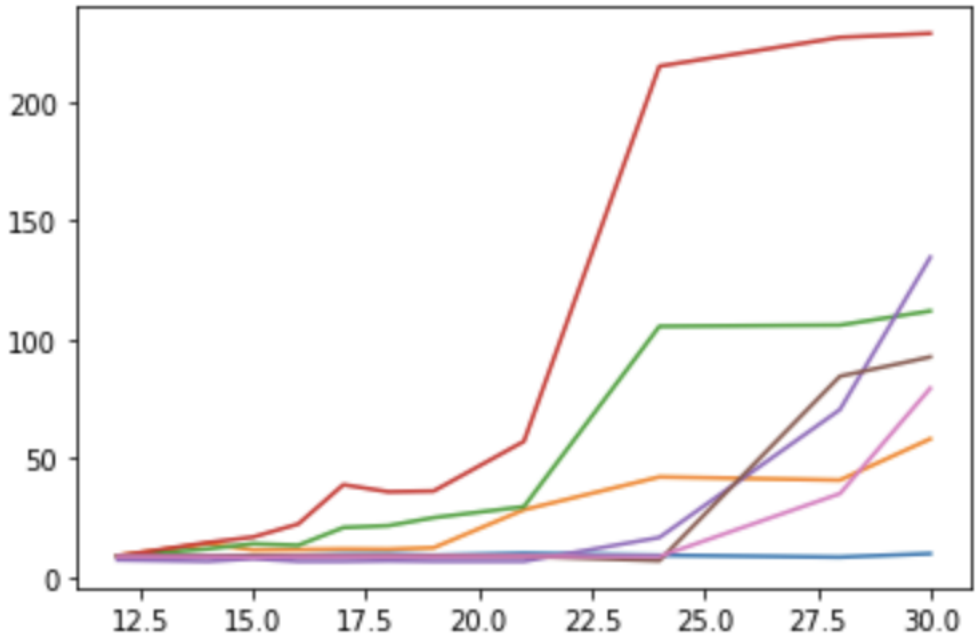
We created different data sets for our measurement. Each data set represents a stride size, with the array size varying from 512 bytes up to 16M so that we can cover memory latencies from L1 cache to main memory. For the ith index of the array we saved the index of the $(i + StrideSize)th$ element in the array. If $i + StrideSize$ is larger than the array size we just mod $i + StrideSize$ with array size so that we can treat the array as a loop and keep getting elements from it. And in our code we make sure that the instructions before and after are also cache missing loads. For each array with different array size and stride size, we use a loop to iterate 1,000,000 times and use rdtsc call to record the clock cycle spend and calculate the time base on that. We will divide our result by 1,000,000 and minus the loop instruction overhead to get the memory read latency value.

4.1.3 Prediction

We should be able to see the RAM access time increases as the array size increases, since larger array will hit larger caches. We should be able to see different stages of the RAM access time, and the array size at different stage should be similar to the cache size of the machine.

4.1.4 Result

Each line the the graph below represents a specific stride size, the x-axis is the $log_2(arraysize)$, the y-axis is the cycle count.



S. \ A.	2^{12}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{21}	2^{24}	2^{28}	2^{30}
2^2	9.170	9.221	9.231	9.310	9.751	9.869	9.332	10.167	9.367	8.580	10.048
2^5	9.141	14.214	11.591	11.751	11.972	11.904	12.515	28.515	42.311	40.981	58.321
2^7	8.937	12.028	14.200	13.557	21.010	21.820	25.178	29.795	105.667	106.169	112.124
2^{10}	9.093	14.734	17.020	22.507	39.037	36.060	36.376	57.281	215.103	227.220	228.892
2^{20}	7.383	6.824	7.920	6.827	6.822	7.037	6.852	6.830	16.835	70.638	134.711
2^{22}	9.221	9.099	9.104	9.104	9.097	9.101	9.128	8.981	7.076	84.664	92.775
2^{24}	8.864	8.790	8.528	8.816	8.857	8.880	8.829	9.002	8.851	35.288	79.613

Note: S. represents Stride Size, and A. represents Array Size

4.1.5 Analysis

We can see that for all the datasets, the RAM access time would increase at approximiatly the same time, that is becasue we hit different caches in our hardware. We can see that when the array size reached about 2^{14}, some data sets started to get more clock cycle for RAM access, that is because our machine has a 12Mb L3 cache, and after the array size reaches about 2^{14}, it started to hit the main memory which will take longer time. And the reason why we didn't see a significant difference in lower array size is that we think maybe our L1 cache and L2 cache are already very fast and the clock cycle count is very small so we can't tell a significant difference.

4.2 RAM Bandwidth

4.2.1 Methodology

We measured the ability of our machine to read and write data over a varying set of sizes, since there are too many results to report all of them, we focused on large memory transfers.

4.2.2 Implementation

We adopted the method proposed in the Imbench paper to measure the memory reading by using an unrolled loop that sums up a series of integers. According to the Imbench paper, the loop is unrolled such that the compiler will generate code that uses a constant offset with the load, resulting in a load and an add for each word of memory. Since in modern processor, the add instruction will be processed in very little time comparing to accessing memory so that our result will be dominated by the memory subsystem, not the processor add unit. We allocated a 40G size array to perform the unrolled loop operation on.
The memory writing is measure by an unrolled loop that stores a value into an integer and then increments the pointer. The processor cost of each memory operation is approximately the same as the cost in the read case according to the Imbench paper.

4.2.3 Prediction

Base on the spec of our machine and an equation we found online, we can calculate the theoretical RAM bandwidth of our machine to be 41.67G/s.
https://superuser.com/questions/736381/what-does-the-mhz-of-ram-really-mean.

4.2.4 Result

The read bandwidth result is $4.69 * 10^9$ cycle per 40G, and since our machine has a frequency of 2.6GHz, that can translate to 22.17G/s. The write bandwidth result is $4.79 * 10^9$ cycle per 40G, which can translate to 21.7G/s.

4.2.5 Analysis

We noticed that our result is about half of our prediction, we think there are couple of potential explaination to this result. First of all, since our machine has multi-core (6 cores), different processors they all need to share the memory bandwidth, and we limited our benchmark program to only run on 1 core, so potentially other cores are also using the memory bus and take up some part of the bandwidth. Second reason is that for an intel cpu, it can change its frequency some time higher than the frequency it was stated on the spec sheet, and if based on a higher frequency, our result calculation from cycle count will be higher than around 20G/s.

4.3 Page fault service time

Page faults occurs when access to a virtual memory is not in physical memory. Then, the system needs to fetch the data block from disk. The time required to fetch blocks from disks to physical memory is page fault service time.

In order to measure page fault service time, we utilize mmap() mechinism.

mmap() establishes mapping between virtual address and disks directly. Initially, mmap() will not bring contents on disk into main memory. Thus, every first access to a page will cause a page fault. In our experiments, we use mmap() to map a large file into memory, and access every pages once to measure the page fault service time.

The OS has 3 layer of cahce and TLB, so we cannot access sequential pages to incur page fault since adjacent pages are also cached after fetching. Thus, we access pages in large file that are apart at a distance much larger than L1,L2,L3 caches. Also, before measurements, we run "purge" command to clear disk buffers.

4.3.1 Prediction

We make prediction based on "dd" command. "dd" command can gives us an relative accurate value of disk read. We first run following code to write a temporary file

- dd if=/dev/zero bs=4096 of=testfile count=10000

Then, we "purge" the disk buffer by running "purge".

Finally we run following code to acquire the time for fetching a single pages:

- dd if=testfile bs=4096 of=/dev/null count=1

We run above command several times, and the average time is 0.000182s (corresponding disk read speed is 22516211 bytes/sec).

4.3.2 Result

The result is following:

Cycles	Mean Time	Std. Deviation
652894	0.00025ms	0.0000313ms

4.3.3 Analysis

The results is slightly bigger than prediction. The reasons might be that the test program has overhead when measuring. Also, disk traffic might be occupied by other processes.

Dividing by page size, fetching a byte from disks requires $0.00025ms/4096 = 61ns$. Compared with fetching a single byte from RAM in previous part, the time fetching single byte from disk is much more expensive.

5 Network

In this section, the computer used above serve as clients. We use another same type of computer as server. The loopback experiments run on computer used in previous sections by communicating through 127.0.0.1 (localhost).

5.1 Round trip time

5.1.1 Backgroud and prediction

Round trip time(rtt) refers to the time between clients sending requests and clients receiving request from server. For loopback interfaces, the data packets will not pass physical layer. The data will only pass a virtual network interface within local OS. The time required is much smaller than the one of remote interface. For remote interfaces, the data packets pass through local network card, router, remote network card and then travel back. The OS provides a command "ping" to measure the RTT.

The result of RTT is very hard to predict theoretically. RTT is affected by both latency and bandwidth. When data packet is small, the latency dominates RTT, which is the case of "ping". The latency is related to router, protocols, network card and many factors, which is difficult to formulate. However, we can predict RTT based on the result of ping. Ping sends ICMP packets, while we use TCP protocol in testing which requires handshaking. Also, the differences in time of loopback and remote interfaces are both the latency of routers and network, but ICMP and TCP layers is implemented differently, and TCP(a transport layer) should contribute more latency. Thus, we have following prediction:

- The RTT from testing should be slightly larger than RTT from ping.
- The differences of remote and loopback RTT of tesing should be slightly larger than ones of ping.

5.1.2 Methodology

We measure RTT by establishing a server and a client. First, the client send a 56 bytes packet(same packet size as "ping") to server. Then, the server send the packet back as soon as the server receives the packet. We measure the time of above procedure by rdtsc(). The measurements is repeated for 10 times.

5.1.3 Results

For loopback interface:

Type	Mean(clks)	Means(time)	Std. Deviation (time)
Test	322315	0.124ms	0.016ms
Ping	/	0.071ms	0.012ms

For remote interface:

Type	Mean(clks)	Means(time)	Std. Deviation (time)
Test	1062891	0.424ms	0.103ms
Ping	/	0.282ms	0.092ms

5.1.4 Analysis

Our results accord with our predictions. However, the predictions are not strict, and there are still some factors not taken into account. For example, the test program can have latency even though the server send back data packet as soon as it receives the packet.

Also, the test program does not count setup time and tearing down time into RTT, which reduces the difference in time of TCP and ICMP. TCP requires 3-way and 4-way handshaking when connecting and tearing down, which consumes even more time for transferring single data packet.

In conclusion, the results are to our expectation. The latency dominates RTT, and different protocols adds different overhead. Remote interfaces also have hardware overhead.

5.2 Peak Bandwidth

5.2.1 Background and Prediction

The time for transferring data is restricted by both latency and bandwidth. When size of data is large, the proportion of latency becomes small. Thus, the overhead of latency can be eliminated by sending large files.

The peak bandwidth of loopback interface only depends on CPU and OS. The loopback does not pass physical layer such as routers. The loopback bandwidth is hard to formulate, but we can predict the results based on ”iperf” command.

Since remote computer is under the WiFi of local computer, the peak bandwidth of remote interface is restricted by routers. The bandwidth of routers in this experiment is 120Mbits/s. Thus, the bandwidth is predicted to be 15.0M bytes/s.

5.2.2 Methoology

We establish a client and a server similar to test for RTT. The server will send a 5000000 bytes message to clients, and the program will measure the time consumed by rdtsc(). Clients and servers of loopback test run on the same computer, while clients and servers of remote test run on two computers. We run the test for 10 times.

5.2.3 Result

For loopback interface:

Type	Mean(clks)	Means(Bandwidth)	Std. Deviation (time)
Test	4326871	2.78 GBytes/s	0.32 GBytes/s
iperf	/	4.49 GBytes/s	0.48 GBytes/s

For remote interface:

Type	Mean(clks)	Means(Bandwidth)	Std. Deviation (time)
Test	1172699402	11.1 MBytes/s	1.64 MBytes/s
Prediction	/	15.0 MBytes/s	/

5.2.4 Analysis

The measured bandwidths for both loopback and remote interface are smaller than predicted. There are following possible reasons.

For loopback interfaces, the testing program will occupy some CPU resources because the 5000000 bytes message is generated and stored by testing program, which will occupy some memory. Further, the program cannot takeup all CPU time. The program is only a user process, so CPU schedule as much time as other user process to testing program. Thus, the sending and receiving is idled when the process is not scheduled, which reduced measured bandwidth.

For remote interfaces, the bandwidth of router might be occupied by other devices under the WiFi. Also, TCP protocol usually cannot reach peak bandwidth. TCP protocol adds headers to each data packet, which increases the actual bytes sent. TCP layer required extra time to format the packet. Further, some data packets might be lost, and require re-sending. All these factors reduce the measured bandwidth.

In conclusion, the accuracy of measured peak bandwidth is reduced by protocol overhead, other devices under router, hardware overhead(remote), CPU scheduling(loopback), test program overhead and network instability.

5.3 Connection overhead

5.3.1 Background and prediction

The setup of TCP requires 3-way handshaking, which theoretically consume time equivalent to sending 3 data packets. The tearing down of TCP requires 4-way handshaking, which theoretically consume time equivalent to sending 4 data packets.

We can predict the connection overhead based on previously measured RTT. Since RTT measured previously does not take setup and tear down time into account, the RTT time only consists of round-tripping a single data packet. i.e. Measured RTT is equivalent to time sending 2 data packets.Thus, the predicted setup and tear down overhead for loopback interface are $0.124/2 * 3 = 0.186ms$ and $0.124/2 * 4 = 0.248ms$ respectively. The predicted setup and tear down overhead for remote interface are $0.424/2 * 3 = 0.636ms$ and $0.424/2 * 4 = 0.848ms$

5.3.2 Methodology

We establish clients and servers similar to previous experiments. We measure rdtsc() before and after the connection is established to acquire setup time. We measure rdtsc() before and after the client close the connection to acquire tear down time. The measurement is repeated for 10 times.

5.3.3 Result

For loopback interface:

Type	Mean(clks)	Mean(time)	Std. Deviation (time)
Setup	423421	0.176ms	0.032ms
Setup Prediction	/	0.186ms	/
Tear down	169335	0.065ms	0.009ms
Tear down Prediction	/	0.248ms	/

For remote interface:

Type	Mean(clks)	Mean(time)	Std. Deviation (time)
Setup	1502796	0.578ms	0.102ms
Setup Prediction	/	0.636ms	/
Tear down	298032	0.115ms	0.025ms
Tear down Prediction	/	0.848ms	/

5.3.4 Analysis

The measured setup time match predicted setup time well, while measured tear down is much smaller than predicted. Also, measured tear down time is much smaller than measured setup time, which conflict with analysis in background part.

Many reasons might explain the results:

- The measured RTT cannot represent actual time for sending data packets during connection. Too many factors in RTT measurement affect the accuracy, such as protocol overhead, network instability and packet lose.
- The close() function (which is used for tearing down in measurement) in C might be implemented differently from TCP implementation. The clients in C simply close the connection and does not wait for the response from serve, which can explain why tearing down time is much smaller than setup time. Also, some TCP tearing down only requires 3-way handshaking.

In conclusion, the overhead for both setup and tear down including protocol overhead, detail implementation and hardware overhead(remote interfaces).

6 File System

In this section, we will be testing RAM and disk performance and utilization.

6.1 Size of file cache

6.1.1 Background and prediction

The testing machine has 16GB of physical memory. Based on information from Mac's built in activity monitor, there are about 7.5GB of memory used already. About 5.2GB of which is for app memory and another 2.3 GB for wired memory and compressed. Cached files took up about 8.3GB of the cache. Since we will be trying to cache the file into RAM, I will predict that all 8.3GB of cache that is currently used for

cached files can be used. If we are being conservative, at least half of those space can be used so that the other half can ensure the smooth operations of other processes on the system.

6.1.2 Methodology

The basic idea is rather simple. We will read in the file for the first time and thus most of the file should be written into cache. Then we will read the same file again and this time we should be reading from the cache, or at least parts that is still on cache. If file can be fit into the cache, our first read will just put all of that file to cache. In this case, the second read will be rather fast since reading from cache took far less time comparing to reading from a disk. However, there will be a point where as the file will just be too big for the cache. In that case, when we read it for the first time, parts of the cache will be kicked out in order to put in new blocks. If this is the case, we will not get much improvement from second read since all of the blocks we need are already been kicked out from the cache and we just need to read from disk again. Although our files are randomly generated, our read is sequential. To avoid the cache prefetch next block needed, our code is organized in the way to from backward. We will be testing 6 files in order to find the boundary of cache size. We will start from 0.25GB, and double the size each round. We start the time measurement right before reading operation and stop right after. We will be adding the time together and divide that sum with how many reads are actually performed. The result will be average time taken to read a block.

6.1.3 Results

Here is the result of average time taken to read a block from couple randomly generated files.

Table 2: Cache Size Test

File Size (GB)	Average Time Taken
0.25	4183
0.5	3954
1	109884
2	110555
4	108531
6	108836

6.1.4 Analysis

From the results, we can clearly see that there is a major performance impact from 0.5GB to 1GB. We can infer from this result that available cache to use is between 0.5GB to 1GB. Reading below 1GB is fast and stable, which can only be explained by there location in cache. Reading above 1GB is almost 3 times slower, which should be a result from reading from disk. The size of available cache is quit surprising. Mac is very conservative about how much RAM it allows a single process to use and very reluctant to kick out other cache files in favour of ours. Maybe if our process has much higher priority, the OS will locate much larger size to it. But as of a normal process, it can only utilize about 1GB of RAM from the total of 16GB.

6.2 Local file read time

6.2.1 Background and Prediction

The machine we use to analysis has a SSD. Comparing to traditional HDD, SSD do not have any physical moving parts and can read from any location. Based on the characters of our disk, we predict that sequential read will be really fast since only overhead will be file access time. Random read may be a bit slower since it need to locate the file first but much faster than any HDD machine since there will be no physical parts moving back and forth to find some file.

6.2.2 Methodology

We will read files of different sizes and add the time each read took. We will also be using fcntl() system call to set FNOCACHE, which will prevent any kinds of memory caching for our read. The final result will be the sum generated from each read divide by the total read performed. With caching turning off, this average read time should just be the disk access time. For sequential read, we will just open a file and read from start to the end. And for random read, we will just open the file and read random blocks from that file until we have reached our reading goal.

6.2.3 Results

Here is the result of average reading time from reading various sizes of files sequentially and randomly.

Table 3: Local Read

File Size(MB)	Sequential Read Time	Random Read Time
2	17174348	17373462
4	8686731	8842696
5	4421348	4607755
16	2303877	2504063
32	1252031	1473313
64	736656	979036
128	489518	744467

6.2.4 Analysis

Firstly, it is very interesting to see that larger the file size, shorter time on average is needed for the read. We believe this is because that current SSDs has built on-board CPU and cache to improve performances. This caching cannot be turned off on OS level because it is really hardware based and do not communicate with the OS. When reading larger files, CPU and caching system will catch up on what is going on and make predictions, which will decrease the time needed for each read. It is very constant that random read is always a little bit slower than sequential read. This result is on par with our prediction since reading randomly will require the disk to locate the file. This process will bring extra overhead to the reading operation.

6.3 Remote file read time

6.3.1 Background and Prediction

We will be using LAN to mount the disk of another Mac and access files through LAN. Reading process should be exactly same comparing to read locally except that now we added another layer of transmission. In particular, files will be transferred via Wifi to the router and then to our testing machine. Our testing machine will then decode the transmission and then finally get the result. We predict that this process will create a big overhead comparing to reading files locally.

6.3.2 Methodology

We used another Mac as a NFS server and access the file on that Mac from our testing machine via TCP connection. All other method are exactly the same comparing to local read test.

6.3.3 Results

Here is the result of average reading time from reading remotely various sizes of files sequentially and randomly.

Table 4: Remote Read

File Size(MB)	Sequential Read Time	Random Read Time
2	47645933	64765431
4	32382715	48502901
8	24251450	43948467
16	21974233	40784894
32	20392447	39713279
64	19856639	47379340
128	23689670	43303834

6.3.4 Analysis

As we have expected, the network layer added to the reading process gives a huge overhead. We are now looking at almost 10X average accessing time. It is understandable since there are way too many limiting factors here. We can't determine which factor is the dominant one but we do know that combined together, reading remote files performs much worse than local files.

6.4 Contention

6.4.1 Background and Prediction

With contention, we are now reading from multiple different files from multiple different processes running at the same time. In this case, multiple reads need to share the resources like disk utilization and CPU. We predict that this will lead to a big overhead since disk resources need to be split into multiple pieces and there will be extra overhead to switch between different reading processes.

6.4.2 Methodology

We will be reading various numbers of files in the background and time the average time per read for another process, both sequentially and randomly. The reading process is exactly the same comparing to local read test and remote read test where all caching is disable to better reflect disk capability.

6.4.3 Results

Here is the result of reading from local files both sequentially and randomly with various background reading processes.

Table 5: Remote Read

Processes Number	Sequential Read Time	Random Read Time
1	19856639	47379340
2	17179757791	17180029903
3	17179790506	17180059518
4	17179835881	17180108250

6.4.4 Analysis

It is very obvious that contention reading gives a huge overhead. This is because of how disk resources are shared and prefetching doesn't work the same. If there is only one reading process, then data can be prefetched into the memory and hugely increased the reading time. However, when there are mulciple processes reading at the same time, OS need to do context switching very often and context switchings are really expensive. Also, all prefetched caches need to be flushed in order to server the new process. That's why there is a big difference between reading sequentially and reading randomly with only one process but reading sequentially and reading randomly with multiple processes are barely noticalbe. When there are no prefectching, reading randomly and reading sequentially doesn't make much of a difference with SSD.