

# CSE 221: System Measurement Project

Yu Shen (A53092101)  
Yingzhi Wu(A53102471)  
Qingyu Zhou (A53093706)

Instructor: Alex C. Snoeren  
Grader: Brajesh Kushwaha



# 1. Introduction

An operating system plays a vital role in managing computer hardware and software resources and providing common services for computer programs. In order to understand and build operating system better, we decided to set experiments to take a deep study on the performance characteristics of the underlying hardware components working with the operating system, OS X El Capitan version 10.11.

We chose C++ as our primary programming language for the reason that C++ has a powerful low-level capabilities. To compile our C++ code, we choose g++ as our compiler, and the version is Apple LLVM version 7.0.2 (clang-700.1.81). Our machine has dual cores, but we turned off one core to make our system a single-core. Because in modern multi-processor operating system, OS scheduler can switch running processes in and out of all available cores. However, it will severely impact our benchmark results since switching processes among different cores can't make sure the system clock synchronization work as expected. In such case, we switch our test operation system's active processor cores from 4 to 1.

This paper focuses on designing effective experiments to analyze the performance of CPU, memory, network and file system of test machine and operating system. Section 2 lists the machine description of our test machine. Section 3 discusses overhead of CPU, scheduling and OS services operations. Section 4 presents analysis of RAM access time, RAM bandwidth and page fault service time. Section 5 reports 3 factors on the performance of network. Section 6 gives details about how to estimate the performance of file system in 4 different perspectives.

## 2. Machine Description

Hardware	Technical Spec
CPU Model	Intel Core i5 (I5-5257U), 2.7GHz, 1(2 cores)
Cycle Time	$1/2.7\text{GHz} = 0.37\text{ns}$
L1 Instruction Cache(per Core)	32 KB
L1 Data Cache(per Core)	32 KB
L2 Cache(per Core)	256 KB
L3 Cache	3 MB
Memory	2 * 4GB DDR3 (1867 MHz)

Disk Model	APPLE SSD SM0256G
Disk Capacity	251 GB
Disk Controller Speed	5.0 GT/s
Disk Read Bandwidth (spec from vendor)	Sequential Uncached Read 84.19 MB/sec [4K blocks] Uncached Read 753.56 MB/sec [256K blocks] Random Uncached Read 33.61 MB/sec [4K blocks] Uncached Read 420.80 MB/sec [256K blocks]
Network Card Type	AirPort Extreme (0x14E4, 0x133)
Network Card Speed	144 Mbit/s
Operating System	OS X El Capitan version 10.11

## 3. CPU, Scheduling, and OS Services

### 3.1 Time Stamp Counter Mechanism

#### 3.1.1 Cycle Counting

In order to get rid of the problem caused by coarse-grained measuring method like using *time()* function in C language, we can use *RDTSC* to count the cycles of our real executing code. Then we can use the formula

$$time\_int\_seconds = number\_of\_clock\_cycle / frequency \quad [1]$$

to get our result in seconds.

#### 3.1.2 Measurement

After detailed background study on RDTSC, we finally are able to write our first version of the cycle measurement. The general code will like this:

```
asm volatile (
    "RDTSC\n\t" "mov %%edx, %0\n\t" "mov %%eax, %1\n\t"
    : "=r" (cycles_high), "=r" (cycles_low));
```

```

measured_function(&variable);

asm volatile (
    "RDTSC\n\t" "mov %%edx, %0\n\t" "mov %%eax, %1\n\t"
    : "=r" (cycles_high1), "=r" (cycles_low1));

start = ( ((uint64_t)cycles_high << 32) | cycles_low );
end = ( ((uint64_t)cycles_high1 << 32) | cycles_low1 );
deltaTime = end - start;

```

This code works correctly and can achieve what we want. However, there are still some places we can discuss and optimize.

(Notice, we use AT&T syntax here to write the assembly language. The compiler we use is based on LLVM and use Clang. it supports the AT&T syntax.)

### 3.1.3 RDTSCP

We want to guarantee everything is executed before we record time.

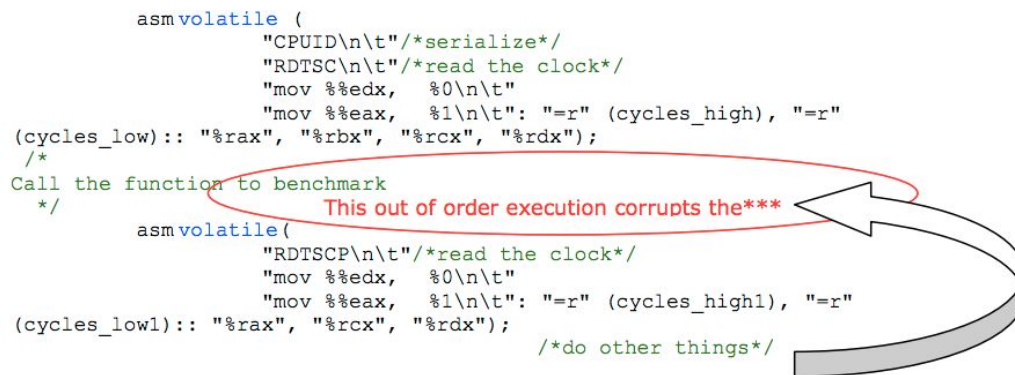
*“The RDTSCP instruction waits until all previous instructions have been executed before reading the counter. However, subsequent instructions may begin execution before the read operation is performed.”[4]*

Once we use *RDTSCP*, a bigger problem. Instructions after our reading and storing counter will be executed in advance sometimes. A picture from [4] articulates this problem in a more clear way.

```

asm volatile (
    "CPUID\n\t"/*serialize*/
    "RDTSC\n\t"/*read the clock*/
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t": "=r" (cycles_high), "=r"
(cycles_low): "%rax", "%rbx", "%rcx", "%rdx");
/*
Call the function to benchmark
*/
asm volatile(
    "RDTSCP\n\t"/*read the clock*/
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t": "=r" (cycles_high1), "=r"
(cycles_low1): "%rax", "%rcx", "%rdx");
/*do other things*/

```



RDTSCP problem [4]

Here, some other instructions may jump into before the “RDTSCP”. The solution is already mentioned in previous chapter. Using serialization here! After using CPUID, we get a modified version of our code.

```
asm volatile ("CUID\nt"
             "RDTSC\nt"
             "mov %%edx, %0\nt"
             "mov %%eax, %1\nt": "=r" (cycles_high), "=r" (cycles_low)::
"%rax", "%rbx", "%rcx", "%rdx");
/*****
/*call the function to measure here*/
*****/
asm volatile("RDTSCP\nt"
             "mov %%edx, %0\nt"
             "mov %%eax, %1\nt"
             "CUID\nt": "=r" (cycles_high1), "=r" (cycles_low1)::
"%rax", "%rbx", "%rcx", "%rdx");
```

RDTSCP solution [4]

Now, we get a quit perfect code which avoids the out-of-order problem and gives us the current cycles number for the measured function.

Because we will call these RDTSC and RDTSCP very frequently, we encapsulate these two assembly lines into two functions. One is called **rdtscStart()** another is called **rdtscEnd()**.

```
static inline uint64_t rdtscStart(void) {
    uint32_t lo, hi;
    asm volatile (
        "CUID\nt"
        "RDTSC\nt"
        "mov %%edx, %0\nt"
        "mov %%eax, %1\nt"
        : "=r" (hi), "=r" (lo)::"%rax", "%rbx", "%rcx", "%rdx"
    );
    return (((uint64_t)hi << 32) | lo);
}

static inline uint64_t rdtscEnd(void) {
    uint32_t lo, hi;
    asm volatile (
        "RDTSCP\nt"
        "mov %%edx, %0\nt"
        "mov %%eax, %1\nt"
        "CUID\nt"
        : "=r" (hi), "=r" (lo)::"%rax", "%rbx", "%rcx", "%rdx"
    );
    return (((uint64_t)hi << 32) | lo);
}
```

```

    );
    return (((uint64_t)hi << 32) | lo);
}

```

## 3.2 Measurement Overhead

### 3.2.1 Reading Time Overhead

#### Methodology

In this section, we are going to test the basic reading time overhead. As mentioned before, using the RDTSC is more accurate than using those system time measurements from C/C++ libraries. Therefore, in the following session, we will keep using this method to measure the whole time to analyze the performance of this operation. However, RDTSC has nothing to do with any operation of our study, using it to measure the execution time of one operation will bring overhead to the result. For more precise observation of our following experiments, we should measure the overhead of invoking our implemented RDTSC function first.

In our experiment, we continuously called `rdtscStart()` and `rdtscEnd()` once respectively in a loop, which we ran over for 100,000 times. After both functions finished, we using the `end_time` to reduce the `start_time` to get the difference. And then we calculated the average of these differences to obtain this measurement result. To get rid of the overhead from the loop, we treat the consecutive calls to `rdtscStart()` and `rdtscEnd()` as a single trial in the loop iteration.

#### Prediction

In our implementation of `rdtscStart()` and `rdtscEnd()`, we are mainly using embed assembly language source code within a C++ program. As we used 5 instructions in each of these functions, and then at the end of these functions, we use OR and << to get an unsigned 64 bit integer of current time counter. However, these instructions are not the basic ones, we are guessing each one may take 3 to 5 cycles and it will takes more than 30 cycles to finish both methods. Besides, although we have implemented these functions by using C inline function, we still think invoking an inline function can also bring overhead. To sum up, we think the results will be 30 to 50 cycles.

#### Result

#	1	2	3	4	5	6	7	8	9	10
Cycles	37.3032	40.1009	32.6056	32.5441	33.3984	31.6247	32.0862	32.5084	32.2587	33.0507

Average CPU cycles is 33.74809 and standard derivation: 2.59588695

## Analysis

From the results above, the measurement result matches up with our prediction.

### 3.2.2 Loop Overhead

#### Methodology

In this section, we will measure the program loop overhead. We separately called `rdtscStart()` before and `rdtscEnd()` after an empty loop to make sure there is by no means other factor bringing more overhead. We set the times of this loop from the set {0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192} and insured that each case will be run for almost 100,000 times in order to get a stable result. For example, when the loop count is 0, we run this case for 100,000 times; while the loop count is 8192, we run this case for  $100,000 / 8192 = 12$  times. After that we got the average overall times for running a loop with a specified loop count and the average times for one loop iteration in each cases.

#### Prediction

In a usual loop operation, there are several steps need to be operated:

1. Check the current counter whether it meets the condition; CPU needs to load the memory to register and then make a condition check operation.
2. Go into the loop to complete the code;
3. After finishing the code, based on the requirement, the value of loop counter will be changed, and then go back to step 1. CPU needs to needs to do calculation on the counter and store the new value back to memory.

In this case, we are assuming each loop will have 6 to 8 CPU cycle operations.

#### Result

Loop Count	Overall time (cycles)	Average time (cycles)
0	36.5884	36.5884
1	39.0297	39.0297
2	43.1936	21.5968
4	56.3275	14.0819
8	85.537	10.6921
16	163.622	10.2264
32	451.603	14.1126



64	2423.41	37.8658
128	963.913	7.53057
256	1861.28	7.27063
512	3664.31	7.15685
1024	7252.25	7.08227
2048	14446	7.05371
4096	28741.7	7.01701
8192	57752	7.0498

## Analysis

From the result above, we can see when the loop count is very small, the result will be affected by the read time overhead as mentioned before. When the loop count is big enough, like bigger than 128, the average time of one loop iteration match our prediction since the read time overhead will be divided into many parts to reduce the influence to the result. But when the loop count is among 4 to 64, as the overall time is increasing and read time overhead is still unneglectable, the result of average time is not stable. However, we don't need to add `rdtscStart()` and `rdtscEnd()` in our real code, so we believe that the number of loop won't impact the performance of loop at all.

## 3.3 Procedure Call Overhead

### Methodology

In order to measure the increment overhead when procedure is called, we wrote a function called `procedureCallOverhead(fstream &file)`. We tried to run `getProcedureOverhead(result)` 10 times to get valuable data. Inside of `getProcedureOverhead`, we run 8 functions all for 100000 times. The 8 functions differ from each other by the number of arguments(i.e. `fun_0()` takes zero argument, and `fun_1(int arg1)` takes one argument, and etc.) There is no instruction in any `fun()` procedure. The procedure instructions reside in memory, and we avoid measuring memory access time to fetch those instructions. We call `rdtscStart()` before the invocation of function and call `rdtscEnd()` after it's returned from the function. Then we divided the total time by 100000 to get the average time for each function with different number of arguments.

### Prediction

Several steps will be executed when invoking a procedure call: Attributes followed with the return address will be pushed to the stack. Then, the invocation of the function jumps to the entry point of the function. After that the local variables will be pushed onto the stack. A local

variables will be popped from the stack. Jump to the address on the top of stack. Pop return address and parameters. The number of parameters will lead to the increase of clock cycles as the arguments added to the procedure. With no argument, we estimate that the procedure call overhead takes about 33 cycles. Our estimation is that the procedure call overhead increases less than 2-3 cycles as the number of arguments increases under the condition of no read time overhead.

## Result

oper	arg = 0	arg = 1	arg = 2	arg = 3	arg = 4	arg = 5	arg = 6	arg = 7
1	36.3037	34.8466	35.6734	35.9826	36.1351	50.8637	41.1352	37.5906
2	34.1203	39.3871	40.5427	35.9032	35.8324	36.6457	36.5807	37.5147
3	34.0046	34.2049	35.7968	35.5459	39.5179	36.5806	36.7446	37.8158
4	33.5044	33.7691	37.6932	36.4714	42.3705	42.6406	38.0035	37.3633
5	34.027	34.7304	37.7093	35.8208	36.1987	36.5186	36.4654	37.3647
6	32.6154	34.197	35.9686	35.8971	35.7838	39.4416	43.3215	37.8969
7	34.1099	35.9319	37.4885	37.0849	35.9894	38.5141	36.9753	38.9286
8	33.0974	34.2824	43.0705	35.9463	35.7918	37.814	39.9996	50.2495
9	33.8783	33.6949	35.7816	35.6287	35.7698	36.5806	36.5804	37.5167
10	34.0368	34.8411	35.816	35.7631	35.8735	36.6048	40.7317	37.3885
avg	33.9697	34.98854	37.55406	36.0044	36.92629	39.22043	38.65379	38.96293
stdev	0.96141	1.674832	2.465581	0.453931	2.226701	4.527658	2.457296	3.993110

## Analysis

We have found that the average of procedure call overhead increase by one cycle as the number of arguments increases by one. This result matches with our prediction. Since there is no instruction inside of procedure function, we avoid the read overhead for fetching the procedure instructions from the memory. This is why our data is relatively small. Each argument is an integer argument that has four bytes length.

## 3.4 System Call Overhead

### Methodology

A system call transfers control from user space to the kernel space and amplifies the privilege to kernel mode. Rather than using other system call which may complicate the analysis of the overhead of switching control from user space to the kernel space, we choose one of the

easiest system call function, getpid(). However, as learned from the function manual[7], since glibc version 2.3.4, the glibc wrapper function for getpid() caches PIDs, so as to avoid additional system calls when a process calls getpid() repeatedly. Therefore, to make a better comparison between non-cached system calls and cached system calls, we also choose a functionally similar system call, getppid(), which returns the process ID of the parent of the calling process, to complete our measurement.

In our experiment, in order to get stable results, we keep using the loop to repeatedly call getpid() and getppid() for 100,000 times. We took into consideration the loop overhead and the overhead of assigning the return value to a local variable. We decided to call rdtscStart(), rdtscEnd() and getpid()/getppid() inside a loop as a single trial and not to assign the return value of these system calls to any variables.

## Prediction

As the getpid() function can cache the PID effectively, while our experiment using the same process to call this function for 100,000 times, we believe that the average CPU cycles will be close to the read time overhead. However, getppid() is a non-cached system call, which means it has to take more time to deal with crossing different protection domains and amplifying the privilege. Therefore, we made a conservative estimate: the average CPU cycles of calling getppid() will increase by almost one order of magnitude.

## Result

The unit is CPU cycles:

	1	2	3	4	5	6	7	8	9	10
getppid()	395.256	378.683	399.705	372.124	408.009	424.603	388.577	379.259	403.472	370.954
getpid()	36.0082	36.9418	39.2232	53.0413	34.2739	34.4467	33.9279	40.2961	34.4088	39.6516

Average CPU cycles of getppid() is 395.8123333, and standard derivation is 18.16879783

Average CPU cycles of getpid() is 38.2, and the standard derivation is 5.443525476

## Analysis

Based on the result above, the average CPU cycles of getppid() is almost 10 times of the average of getpid() or the average of a procedure call(from last experiment, average CPU cycles of a procedure call is about 35 to 40 including the read time overhead), which proves that the cost of switching the control from user space to kernel space is really expensive. Meanwhile, the average CPU cycles of cached system call getpid() is very close to the read time overhead (33.74809 cycles), hence we can say that performance can dramatically be improved through cache and we should take considerations to use a cached system call, especially when invoking such system call too frequently.

## 3. 5 Task Creation Time

### 3. 5.1 Process Creation Overhead

#### Methodology

In this section, we want to measure the overhead of creating a new process by using `fork()` function. Based on the manual of this function[8], the new process(child process) is an exact copy of the calling process (parent process), while the copy includes virtual memory mappings, file tables, signal-handler tables and so on. Since our target is only to measure the time to create and run a process, we decided to make the child process exit immediately after it has been created and started. Therefore, before we call `fork()`, we still used the `rdtscStart()` to mark the start time, and then we called `fork()` function to spawn a child process and assign the return value to the local variable. Learned from the manual of this api, `fork()` returns a value of 0 to the child process and returns the process ID of the child process to the parent process. So we use if-else block to control different branches based on the return value of `fork()`. When the return pid is 0, we called `exit(1)` function to terminate the child process itself immediately; while when the return pid is not 0, we called `wait(NULL)` to make the parent process wait for its child exiting, and then call `rdtscEnd()` to mark the end time. Similarly, we still used loop to run this iteration for 200 times for a stable result and calculated the average CPU cycles difference of end time and start time.

#### Prediction

Creating a new process is a very time-consuming task for the operating system, since there are a lot of steps need to complete before the child process can run with its own resources. These steps usually include assigning a new unique process ID to the child process, allocating resources and spaces to its child process(but in reality, with the help of a technique called copy-on-write, the parent process's pages are not copied for the child process when a fork occurs. Whenever a process modifies a page, a separate copy of that particular page along is made for that process which performed the modification.[9]) and so on. Therefore, we estimated that the process creation will take more time than the time of calling former procedures and system calls, probably 500,000 to 1,000,000 cycles.

#### Result

#	1	2	3	4	5	6	7	8	9	10
Cycles	1.37E+06	867683	816213	798111	820733	835284	886118	819538	8.08E+05	839411

Average of CPU cycles without departure results is 832389.2222 and the standard deviation is 28510.33302

## Analysis

From the above result, the average CPU cycles is as huge as what we expected. Besides, from the standard deviation, we can find that unlike the standard deviation of those less time-consuming procedure, the CPU cycles for creating new process from the same process differs a lot from each other. We think there are a lot of possible reasons for such situation. One reason may be with the number of new processes increasing in such short period, it will take more time for some algorithms like allocating new process ID to complete. Another reason may be lots of terminated processes need to be clean up while new processes need to be spawned, both services need system resources to fulfill their tasks and then become competing services to each other. But the main reason for such operation being time-consuming is still the complicated essential steps need to be correctly completed to run a executable child process.

### 3.5.2 Kernel Thread Creation Overhead

## Methodology

In this session, we use pthread library to create a kernel-managed thread on our system. This test process is very similar with what we did for process creation overhead on the previous session. We adopt the pthread\_create to spawn a new kernel thread, and then in the newly created thread, we called the pthread\_exit(0) immediately. While the main process need to call pthread\_join(new\_thread, NULL) to join the new thread to wait for new thread terminating. Finally we calculated the difference the start time which is generated before calling pthread\_create() and the end time which is marked after pthread\_join(). Same as the process test, we run a loop for 200 times for obtain a stable average result.

## Prediction

Unlike creating a new process, a newly created thread can share the same resource with the calling process, which means creating a new thread will be much lighter task than creating a process. Therefore, we are guessing the time to spawn a new thread will be less than creating a process by almost one order of magnitude, probably 50,000 to 100,000 CPU cycles.

## Result

#	1	2	3	4	5	6	7	8	9	10
Cycles	55235.1	54100.3	54276	52773.3	52967.2	54189.3	58955.4	53524.3	540676	54703.8

Average of CPU cycles without departure results is 54,524.97 and the standard deviation is 1837.672284

## Analysis

As we expected, the average CPU cycles of creating a thread is less than creating a process almost one order of magnitude (54524.97 vs 832389.2222). This is quite a convincing evidence that why multi-thread is more reasonable for multi-processing when we develop our program. Besides, there are also some departure results, which is much bigger or smaller than the expected average, as the processor test. Since creating a thread is still a time-consuming task than common procedure calls, these unusual data occur more frequently in time-consuming processing experiments.

## 3. 6 Context Switch Time

### 3.6.1 Process Context Switch Overhead

## Methodology

The main idea is same as previous creation time of the process. However, there are some tricks we should mention here.

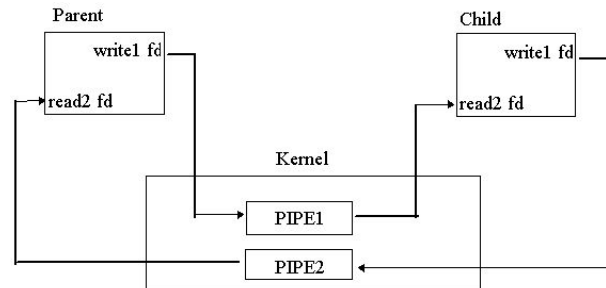
In previous measurement, we only fork a child process and let it do nothing. The `rdtscStart()` is called before the `fork()` and `rdtscEnd` is called after the child process is return. However, we need to change our strategy. We need to place the `sdtsStart()` in the parent process and `sdtsEnd()` in the child process.

The code is:

```
if ((cpid = fork()) != 0) {
    start = rdtscStart();
    wait(NULL);
}
else {
    end = rdtscEnd();
    exit(1);
}
```

In order to measure context switch time, we need to capture the starting time we entered the parent process and the very last second before we leave the parent process. Also, we need to record the beginning time of our arrival in the child process. Hence, we have no choice but place the `rdtsc` in this order.

A major problem comes out. Since two processes own their own image copy, the private's end is not the end in parent process, the start in parent is not the one in the child process. How to make get the difference of these two value? We can use pipe as the communicating method to exchange data.



processes communicate through pipe[20]

Usually, there are four exits for a pipe and we will use two of them. Here, we let the child write into the pipe and parents reads the data from the child process. Hence, we can update the code and get:

```

if ((cpid = fork()) != 0) {
    start = rdtscStart();
    wait(NULL);
    read(fd[0], (void*)&end, sizeof(uint64_t));
}
else {
    end = rdtscEnd();
    write(fd[1], (void*)&end, sizeof(uint64_t));
    exit(1);
}

```

## Prediction

Since we measure the start time of creating a process, we also do experiments on context switch time. Recall from the paper "Plan 9 Bell lab", the context switch plays a significant role in performance. Hence, we will find a way to measure this important value. It is a reasonable prediction that this time will much more large than the simply creation a process. From the kernel's point of view, user->kernel transitions and kernel->user transitions happen during the context switch. Our estimation is that the process context switch time is 5 or 8 times more than the time of creating a process. The process context switch time is around 400000 cycles.

## Result

	1	2	3	4	5	6	7	8	9	10
time	383805	381877	388944	384106	394394	417012	378437	377010	380142	386603
avg	387233									
stdev	11062.41772									

## Analysis

The average process context switch overhead matches with our prediction.

### 3.6.2 Kernel Thread Context Switch Overhead

#### Methodology

In order to measure the thread context switch overhead, we just have to measure the overhead that the main process does a context switching to a new thread. We use `pthread_create()` to create a new thread with a function called `foo()`. In the parent process, we call `start = rdtscStart()` to record the starting point time. Inside of `foo()` function, we call `rdtscEnd()` to record the end time. We subtract start time from end time to get the context switch overhead that it takes to context switch from the process to the new thread. We also used `pthread_join()` to wait for thread termination.

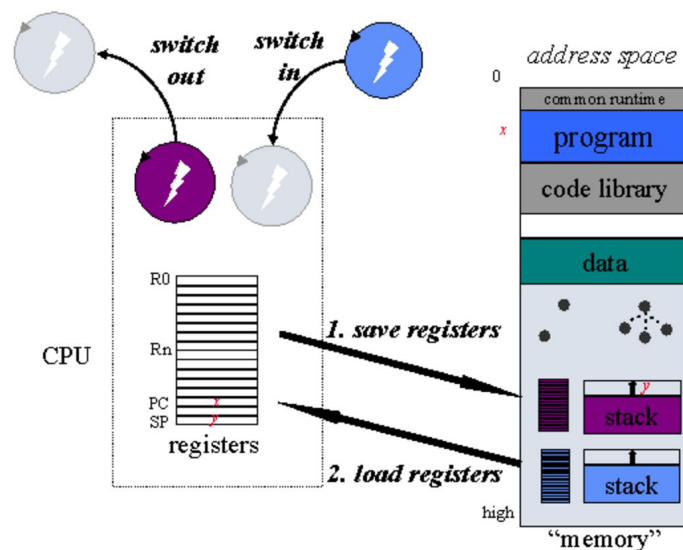
We keep doing this experiment for ten other times to get the total time that kernel thread context switch takes. Then we divide the total overhead by the number of experiments to get the average kernel thread context switch overhead. The purpose of repeating the experiment for multiple times is to improve the accuracy of kernel thread context switch overhead.

#### Prediction

Kernel thread switching is less expensive because only the abstraction that is unique to threads is switched out. The unique abstraction is the processor state. Switching processor state including the program counter and register contents, like what is shown in the figure below, is generally very efficient. The cost of kernel thread context switching is about the same as the cost of entering and exiting the kernel.



## Thread Context Switch



We predict that thread context switch takes less time than process context switch since all threads share the same resources. Conceptually, the main distinction between a kernel thread context switch and a process context switch is that during a thread switch, the virtual memory space remains the same. But during a process switch, the memory space does not remain the same. A kernel thread context switching happens when one thread context switches to another thread within the same process. Our estimation is that the thread context switch is around 6000 cycles.

### Result

	1	2	3	4	5	6	7	8	9	10
time	5754.21	5827.59	5907.23	5795.28	5845.64	5898.54	7076.28	6771.61	7465.8	7625.88
avg	6196.806									

### Analysis:

Kernel thread switching is less expensive. Our measurement result matches with our prediction.

## 4. Memory

### 4.1 RAM Access Time

#### Methodology

The idea of our methodology is inspired by the Imbench paper. As the paper states, we can use “a list of pointers “and changes the sizes of the tested array and stride. [10]

The only thing we need to do is it to create a pointer list and let pointers point to the memory in a certain stride. It sounds like we should use two arrays. One is the pointer array, another is the array used to be pointed to. The thing is, the pointer is also a size of char. In memory, it equals to a word, what is exactly what we want to load each time. Hence, we can simply use one array and just let pointer points to the following index of the pointer array. Just like paper suggests, the code will be like:

```
mov r4,(r4) #c code: p = *p [10]
```

In the project requirement, the “back-to-back-load” is tested here. What is that? Well, in fact, it is just a definition made by Imbench paper. In paper, it defines

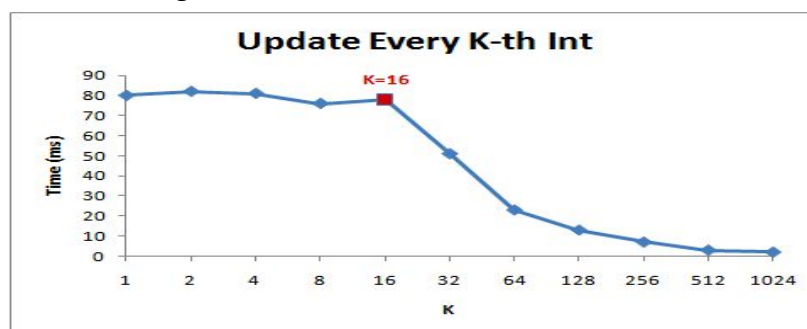
*“Back-to-back-load latency is the time that each load takes, assuming that the instructions before and after are also cache missing loads.”[10]*

In another word, each load is followed by another memory load and after the previous memory word.  $p = *p$  definitely works here. But wait a second, what is the meaning of the “cache missing loads” here? Well, we need to talk about the concept of cache line. In fact, when cpu loads a word (we use word as example here), it loads more than just a word, but also the surrounding memory, called cache line. A nice diagram from Igor’s blog can help us to understand. [11]

If we have a program like this:

```
for (int i = 0; i < arr.Length; i += K) arr[i] *= 3;
```

We will derive this diagram.



[29]

We can find when between 1 to 16 the time barely changes. Why is that? Because in modern computer, each cache line is 64 bytes. Hence, from 1 to 16 words, we read from cache instead of from memory. That is the reason why increasing just 1 could be as fast as  $k = 16$ . The bTobLoad asks us to do every missing at each load. Well, we know that if  $K$  is big enough, then there will be cache miss. But how big is called “big enough”? To answer this, we should consider where we store our cache line. Yes, Of course, the answer is related to the size of cpu L1, L2, L3 cache! We will discuss cpu later.

Back to bTobLoad, there is a hidden problem that paper does not give a clear answer. What if we reach the end of our array? In the paper[10], it requires us to take 1,000,000 loads for each sample and the end of the array is very easy to be reached. Hence, there are **two** solutions.

**One** is

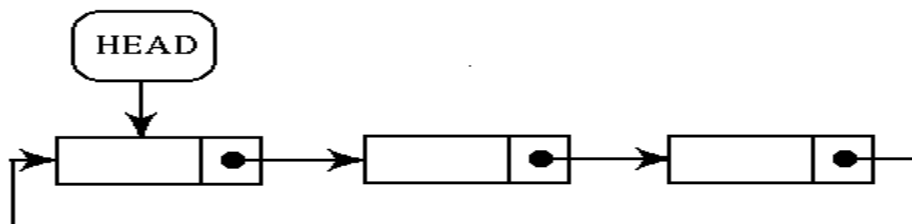
```
p = (char**) *p;
if ( index of p bigger than arraySize)
    p = p_list_head;
```

This is the first solution we use. However, the result is unsatisfactory. It seems that we only introduce a small overhead if statement in each visited. (In Imbench paper, it says memory access will be 200 to 1,000 ns, but the CPU calculation will be only 10 or fewer cycles [10]) . But it turns out that it really influences a lot to our result. Because there are 1,000,000 loads so the total overhead is not negligible. Then how to let the array automatically pointer to the beginning once it reaches the very end of the array?

Well, a more intuitive way to achieve this is that we set the very end of the pointer point to the beginning of the array. Well, this turns out that we have a circulated linked list. Then **second** version of code will be:

```
index = (i/strideSize * strideSize + strideSize)%arraySize;
p_list [i] = (char * ) &p_list[index];    // pointer at i point to index
}
```

Code for Latency1



[12]

However, the compiler is smart. It will optimize the code which has patterns. If we keep moving the pointer with the same distance, the compiler will prefetch the next memory for us. This is not what we want because we want memory to do a normal load each time. Hence, in order to avoid the prefetching, we have to randomize our visit. We will make sure that we move about a stride length each time in memory, but not exactly one stride length. Hence, the compiler cannot let CPU do pre-load for next memory. The code to implement this idea will look like:

```
uint64_t index = ((i/strideSize + 1) * strideSize + rand() % strideSize) % arraySize;
p_list[i] = (char *) &p_list[index];    // pointer at i points to index
```

Code for Latency2

We have a better version of code now. But we can improve the code a little bit more? In the Imbench paper, it mentions that the importance of the unrolled loop.

Normal loop	After loop unrolling
<pre>int x; for (x = 0; x &lt; 100; x++) {     delete(x); }</pre>	<pre>int x; for (x = 0; x &lt; 100; x += 5) {     delete(x);     delete(x + 1);     delete(x + 2);     delete(x + 3);     delete(x + 4); }</pre>

unrolled loop [13]

After unrolling the loop, we get “only 20% of the jumps and conditional branches need to be taken, and represents, over many iterations, a potentially significant decrease in the loop administration overhead.”[13] Let's check our original design of loop.

```
for (int i = 0; i < ITERATION; i++) {
    for (int j = 0; j < Count; j++) {
        p = (char**) *p;
    }
}
```

Here, **Count** means that how many movement do we need to reach the end of the list. As Imbench paper says, we need to traverse the total list for **ITERATION** (1,000,000) times. Here, we try to unroll the loop base 10, i.e.,  $i = i + 10$ .

```

for ( int i = 0 ; i < TenLoadBaseIteration; i ++ ) {
    for ( int j = 0; j < TenLoadBaseCount ; j ++ ) {
        //10 load as base unit
        p = (char**) *p;
        p = (char**) *p;
        p = (char**) *p;
        p = (char**) *p;
        p = (char**) *p;
        p = (char**) *p;
        p = (char**) *p;
        p = (char**) *p;
        p = (char**) *p;
        p = (char**) *p;

    }
}

```

Code for latency3

Here, in order to achieve unrolling, we do not use `j = j + 10` directly. Instead, we recalculate the Count and rename it as `TenLoadBaseCount`. The idea is the same.

## Some Other Facts you may want to know

### How to make the measurement based on the second (or ns)?

In previous section about CPU profiling, we use cycle to make the profile. Indeed, we can use `rdtsc()` to check the cycles difference and use CPU second/cycle to get the time. However, this is way more too complicated and unnecessary. In CPU part, because CPU instruction is so fast that we have no choice but to use `rdtsc`. However, for memory scenario, we can use system time which is accurate at the nanosecond level.

Because we are testing on the Mac OS. Hence, we can use `mach_absolute_time` to do the measurement. A detailed reference can see at here.[7]

### How to avoid compiler optimize away my code?

Compiler nowadays is really smart. It notices that you only traverse the array but do nothing, then it will optimize away your loop. Hence, if you really run you code as we suggest before, the time will be about  $10^{-14}$ s. This is absolutely wrong. So how to let compiler think we are doing some meaningful thing (actually, we do not). Our solution here is to let the function return the last pointer position and send it to the error stream. Compiler does not know the position of `p` until all the loops are finished. Hence the memory loop we used to do the measurement will not be optimized away.

### What sizes are we used to do the measurement? Like array Size, stride sizes?

we use array size start from 512KB and time 2 each time.

We multiply stride 4 each times and start its value from 8.  
Because to are going to use log2 as our x-axis. It's better to use base 2 to increase our array.

## Prediction

Let us recall the figure 1 from Imbench paper. [10]

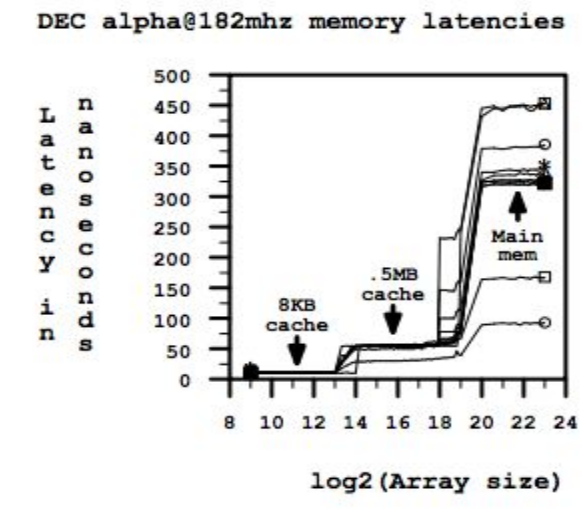


Figure 1. Memory latency

we see there are two steps. 8K is for the L1 cache, and .5MB is for the L2 cache.  
The latency is from 0 ns to 450 ns.

The test machine we use has three-layer cache and it is released within one year. Hence, we can predict that there are three steps in our final result and the latency should be within 450ns at Main memory. We estimate it should bigger than 10ns because frequency is 100Mhz.[36]

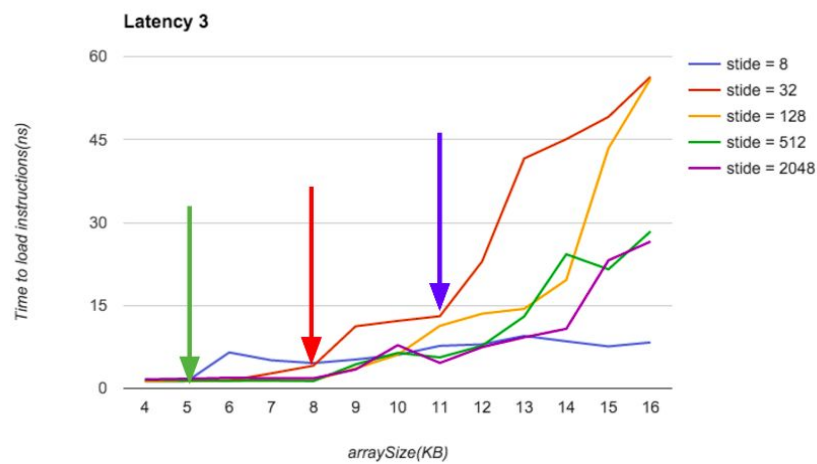
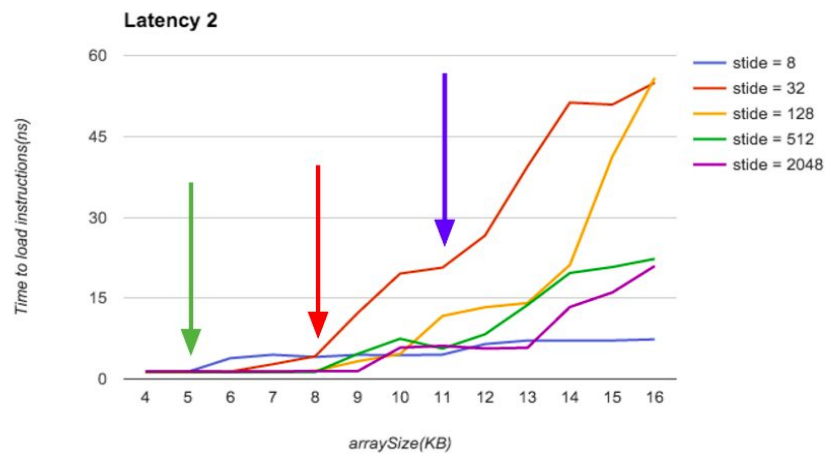
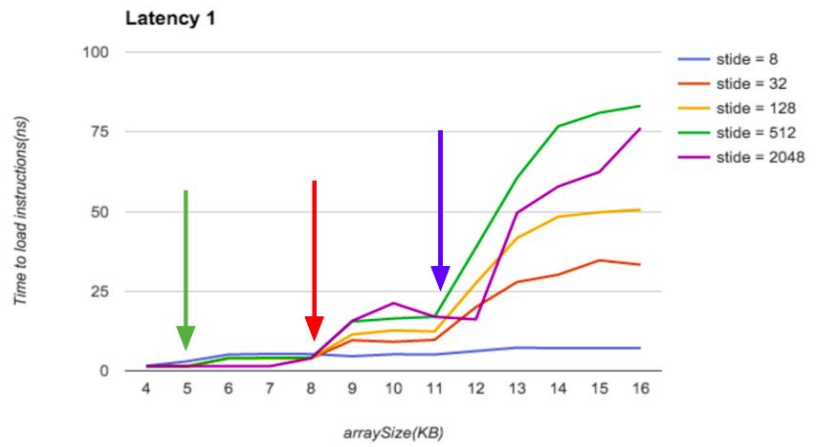
In order to get detailed about our CPU, we get a software to get our test machine CPU information, called “cpuinfo\_x86” from Amit Singh. [15]

Here is the information about the CPU of the test machine.

	L1 Instruction Cache	L1 Data Cache	L2 Unified Cache	L3 Unified Cache
Size	32K	32K	256K	3M
Line Size	64B	64B	64B	64B

We expect that we have clear step changes at L1, L2, L3 level.

## Result



## Analysis

### The L1, L2, L3 size analysis.

We use  $\log_2$  to denote our array size here. We have L1 32k, L2 256k, L3 3M.

Hence we should have a clear change at 5, 8, and 11.58.

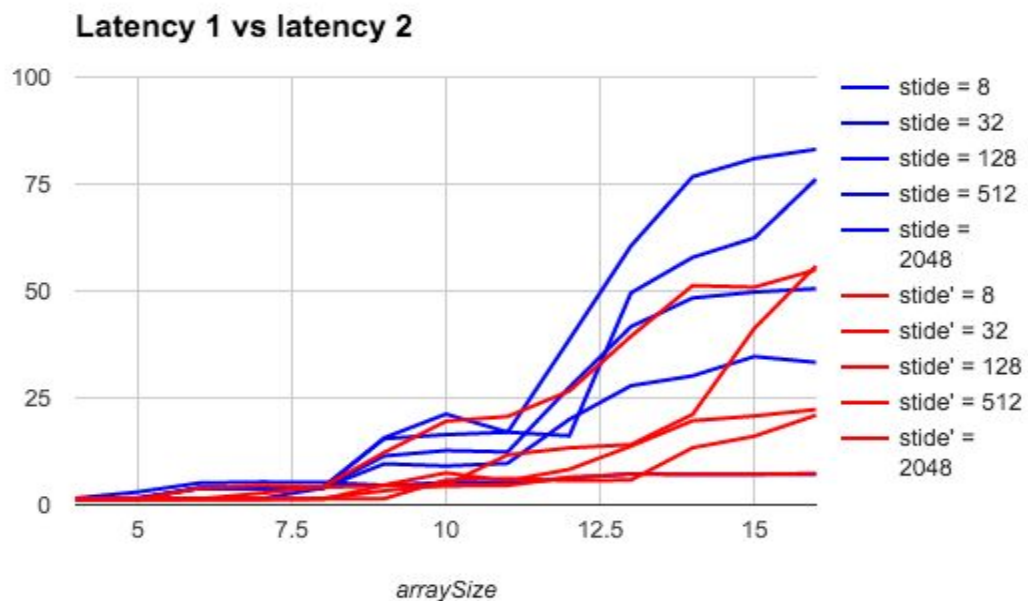
We can see from both three diagram, we have a clear change at these points.

### Latency analysis.

Recall our design for latency 1, latency 2 and latency 3.

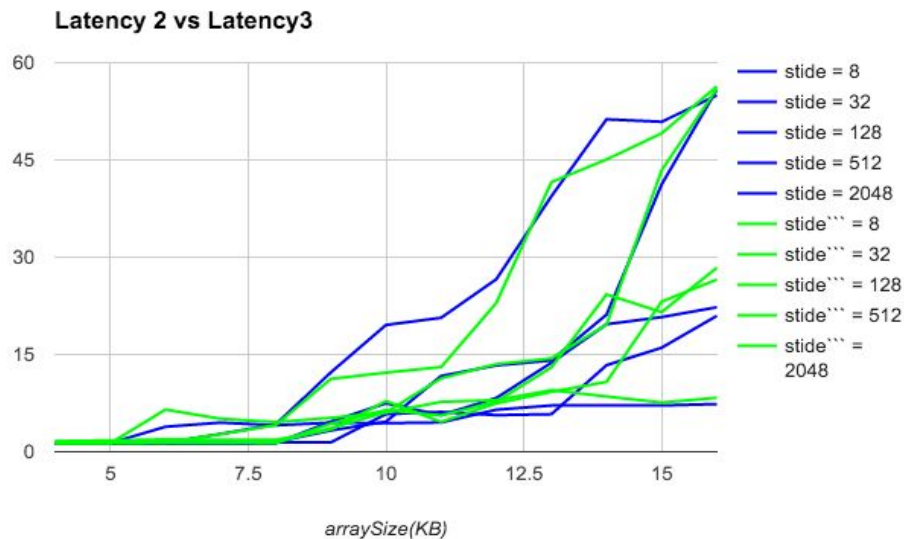
Latency 1 follows the design in Imbench paper. We clearly see that L1, L2 are very steady, just like figure in paper. L3 part has some fluctuations. The reason is that in this case, we increase the memory size by multiply by 2, hence the data here is sparse. This may be one factor leads to fluctuations here. Another possible factor is that L3 is large and its speed is less than L1, L2, so the variance of the time will be large because the base is large.

Latency 2 improves the Latency 1 by adding random in Latency 1 in order to get rid of prefetching. So the latency should be larger. Comparing the latency 1 and Latency 2, we clearly see that our design works. The Blue one is Latency 2 and red one is Latency 1.



Latency 3 we use unrolling idea here. Hence, we expect that the time should be less than latency 2.





In the graph. Latency 3 is the green line. Although it is not that clear, we can tell that green is less than blue line in most cases. This fits our design methodology.

#### Final result:

We will take the latency 3 as the final result.

L1 cache is 1.3 ns

L2 cache is 3 ns

L3 cache is 10.8 ns

Memory is 48.5 ns.

## 3.2 RAM Bandwidth

### Methodology

In the Imbench paper, we have a nice table about bcopy and memory read and write. As the project requirement suggests, we are testing the memory read and write here, hence we do not make the bcopy part.

As the paper suggests, read can use unrolled loop here. After our test, this is not a good idea and get really bad result. Hence, we search the internet and find some better ways to do this. We will use the Imbench method as our base method and use other methods to compare this method. Unfortunately, the Imbench method is the worst one. Let us list the method we found on the internet, we will implement part of them. [16] The code we use will largely follows the design at project of memory bandwidth demo [17]. We try to implement as the same way

as the project does, but turns out that our code is not as good as the code provides. The total methods in that project are listed. The **bold** part is implemented.

- read\_memory\_rep\_lodsl
- **read\_memory\_loop**
- read\_memory\_sse
- **read\_memory\_avx**
- read\_memory\_prefetch\_avx
- **write\_memory\_loop**
- write\_memory\_rep\_stosl
- write\_memory\_sse
- write\_memory\_nontemporal\_sse
- **write\_memory\_avx**
- write\_memory\_nontemporal\_avx
- **write\_memory\_memset**

openMP part:

- read\_memory\_rep\_lodsl\_omp
- read\_memory\_loop\_omp
- read\_memory\_sse\_omp
- read\_memory\_avx\_omp
- read\_memory\_prefetch\_avx\_omp
- write\_memory\_loop\_omp
- write\_memory\_rep\_stosl\_omp
- write\_memory\_sse\_omp
- write\_memory\_nontemporal\_sse\_omp
- write\_memory\_avx\_omp
- write\_memory\_nontemporal\_avx\_omp
- write\_memory\_memset\_omp

### **read\_memory\_loop/ write\_memory\_loop**

In the the reference project, the first method, is loop. Different from Imbench paper, it just uses a simple loop without unrolling.

Like this:

```
void read_memory_loop() {
    unsigned int val = 0;
    for (int i = 0; i < SIZE /sizeof(char); i++) {
        val += array[i];
    }
    assert(!(val==0))
}
```

```

    }

    read_memory_loop().

```

In order to unroll it, we write another version which unrolls the loop.

```

for (i = 0; __builtin_prefetch(&array[i+8],0,1), i < size ; i+=8) {
    val += array[i] ;
    val += array[i+1];
    val += array[i+2];
    val += array[i+3];
    val += array[i+4];
    val += array[i+5];
    val += array[i+6];
    val += array[i+7];
}

```

Assume we get some oracle here, we doubt the unroll version will be worse than the ordinary one and we want a new method to get unrolled loop. (In fact, we actually run the code and find the unroll one is worse in bandwidth. Let us discuss that in the analysis.)

Here is another way to get unrolled loop program. After looking up the g++ manual, we find that adding

```
-funroll-loops
```

will unroll the loop during the compilation. (We also check the -O3 [18], find that unrolled loop is not in the O3.) We compile one version without this flag and one version with this flag on.

#### NOTICE:

You may notice that we use an `assert(!(val == 0))` here. The reason we add this assertion here is that we need to avoid optimization. This is a good method learnt from [10]. Later, we will find that this is not enough for write.

#### read\_memory\_avx/ write\_memory\_avx

Intel is CISC (Complex Instruction Set Computing)[21], hence there are may be some advanced instruction helps us to improve measurement. Here, we are seeking an instrument can read/write massive data from or into the memory. We find that AVX is a good candidate. First, let us talk a little about SIMD. Let us use a nice picture from the Intel Advanced Vector manual[22]

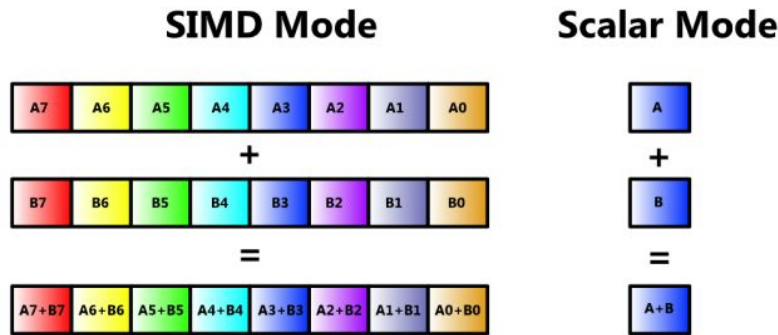


Figure 3. SIMD versus scalar operations

SIMD can operate on the many memory block at the same time. Hence it is faster than the traditional scalar mode. (In fact, it will use YMM registers to do operations.)

AVX is an implement of SIMD Mode. Intel used to use SSE, but it is not on 64-bit and have limit operation on many data types. We just want to use the AVX for measurement, so we do not want to elaborate too many details about AVX. If you are interested, you can read [22] as introduction and [23] for more details.

We do not want discuss our AVX test code here. Only a simple hint for you if you want to check the inside code. Each AVX operation will follow this pattern. (Yes, you are right, it is intrinsic instruction.) [15]

```
_mm256_op_suffix(data_type param1, data_type param2, data_type param3)
```

where `_mm256` is the prefix for working on the new 256-bit registers; `_op` is the operation, like add for addition or sub for subtraction; and `_suffix` denotes the type of data to operate on, with the first letters denoting packed (p), extended packed (ep), or scalar (s). The remaining letters are the types in Table 2.

Table 2. Intel® AVX Suffix Markings

Marking	Meaning
[s/d]	Single- or double-precision floating point
[i/u]nnn	Signed or unsigned integer of bit size <i>nnn</i> , where <i>nnn</i> is 128, 64, 32, 16, or 8
[ps/pd/sd]	Packed single, packed double, or scalar double
epi32	Extended packed 32-bit signed integer
si256	Scalar 256-bit integer

For example, we use `_mm256_add_ps` to add all values in array as read. This means we want to use `_mm256` registers, then we want to add, then we operate on the packaged single.

### write\_memory\_memset

Write\_memory\_memset directly calls the system `memset()`. The `memset()` function “copies the value of c (converted to an unsigned char) into each of the first n characters of the object

pointed to by s.”[17] So we can use it as our write bandwidth test. Later on, you will be surprised by how fast this method is.

## Some other facts you may want to know

### Why there is an extra MemoryWriter in folder?

We code our writer function in a separate file and later link it to the main file. We already mentioned this before. Compiler is really smart! In read, we use assert to fool the compiler. However, in write, this is not enough.

First, we are sure that we will stuff the array with the same value. We can not store it with different values because we need some space to store these difference values. Then, if we are going to assign these to the array which is used to test the bandwidth, there will be additional read load from the memory. Using random is also not a good idea because of the overhead of rand(). Hence, no matter how we use the value from the array. Compiler always has a way to workaround it. For example, we want to sum up all the value of the array. Good, it seems that compiler will have the no choice but to assign all the memory. However, in fact, the compiler will do this:

```
The value you assign to each cell * total_Length_Of_Array
```

Compiler really does a great job to avoid writing the memory! Hence we use a trick here. No matter what strategy I use in the same file, but compiler can realize that I am opening on the same array. So first part is that putting the write code in a separate file and let write function has no idea of what it is going to write.

Secondly, you cannot directly claim the type of the array. In another word, you cannot use :

```
void write_memory_loop_outsider(char * array, int size)
```

The compiler will find that this is a char array and optimize it again. A solution here is that from [17] use void \* pointer to point to the array. In this case, the compiler is not aware of the memory structure of array until it actually read it.

```
void write_memory_loop_outsider(void * array, size_t SIZE)
```

### Why there is an executable file called bandwidth\_unroll?

If you read this report carefully, you may still remember that we use “-funroll-loops” to generate unrolled loop. This is executable file that it generates.

## Prediction

The value should be about 20GB/s. Also, the result should be

```
read_loop < read_unrolled_loop < read_avx
```

**write\_loop < write\_unrolled\_loop < write\_avx < write\_memset**

The manual unrolled should be same as the g++ unrolled loop. Also, the write bandwidth should be less than read bandwidth.

## Result

read_memory_loop	9.18614GB/s
read_memory_unroll	9.15199GB/s
read_memory_avx	13.3005GB/s
write_memory_loop_outsider	7.66534GB/s
write_memory_avx	8.01202GB/s
write_memory_memset	13.5799GB/s

## Analysis

We can see that AVX works. The AVX is 13.3 GB/s but the Imbench paper is about 9 GB/s.

In memory writing part, we find the speed of memset is super fast. It is 13.57 GB.

But wait, does it means our memory has a **faster writing mechanism than reading mechanism?**

Of course, it is impossible. The only reasonable explanation is that our reading, even using AVX, is still not accurate enough.

In order to discover a better way, let us list the data from [17]. Unfortunately, that memory profiler has some compatibility problems. It is runnable on my Macbook air but when I run that on our test machine, It turns that it has some problems. Hence, we decide to provide the data provide on the [17] main page here.

read_memory_rep_lodsl	4.80 GiB/s
read_memory_loop	10.66 GiB/s
read_memory_sse	13.44 GiB/s
read_memory_avx	13.60 GiB/s
read_memory_prefetch_avx	15.06 GiB/s
write_memory_loop	12.84 GiB/s
write_memory_rep_stosl	19.22 GiB/s
write_memory_sse	8.93 GiB/s
write_memory_nontemporal_sse	12.83 GiB/s

write_memory_avx	8.91 GiB/s
write_memory_nontemporal_avx	12.65 GiB/s
write_memory_memset	12.84 GiB/s

From this data, we find that

read_memory_avx	13.60 GiB/s
read_memory_prefetch_avx	15.06 GiB/s

write_memory_memset	12.84 GiB/s
---------------------	-------------

So why we add a magic prefetching here, we can let the reading faster than write?

From [25] we notice that prefetching can improve a lot when we reading something if we know that we are going to read in next turn. [25] gives a good example.

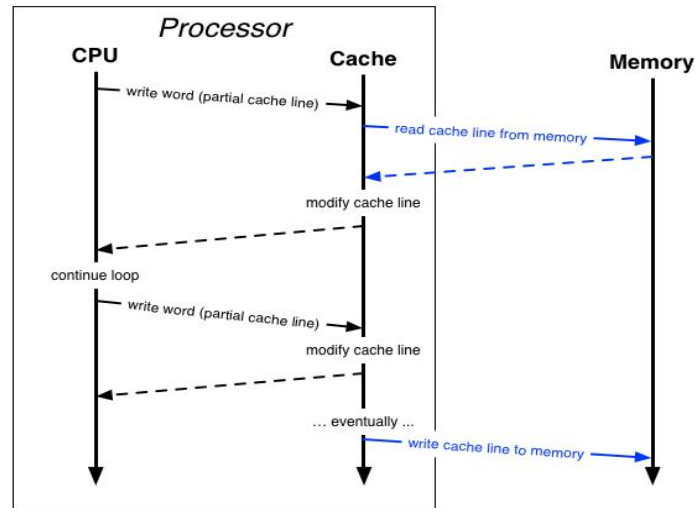
```
#define list_for_each(pos, head) \
    for (pos = (head)->next; prefetch(pos->next), pos != (head); \
         pos = pos->next)
```

in avx, we have a prefetch **\_mm\_prefetch** instruction to boost up our reading. (In face, in our code, we use **\_\_builtin\_prefetch(&array[i+8],0,1)**. But the result is not changing for adding this or not. So we do not mention it in our methodology part.)

### What is non-temporal writing for nontemporal\_avx method?

In Alex blog, it give us a good explanation. We directly use it here. [16]

*“The main problem is that memory traffic on the bus is done in units of cache lines, which tend to be larger than 32 bytes. In order to write only 32 bytes, the cache must first read the entire cache line from memory and then modify it. Unfortunately, this means that my program, which only writes values, will actually cause double the memory traffic I expect because it will cause reads of cache line! As you can see from the picture below, the bus traffic (the blue lines out of the processor) per cache line is a read and a write to memory:”*



Use one sentence to rephrase, we modify the cache line instead of actual memory!

In order to avoid this, we need to use non-temporal operation. The following is the definition in Intel's manual.[23]

Data reference patterns can be classified as follows:

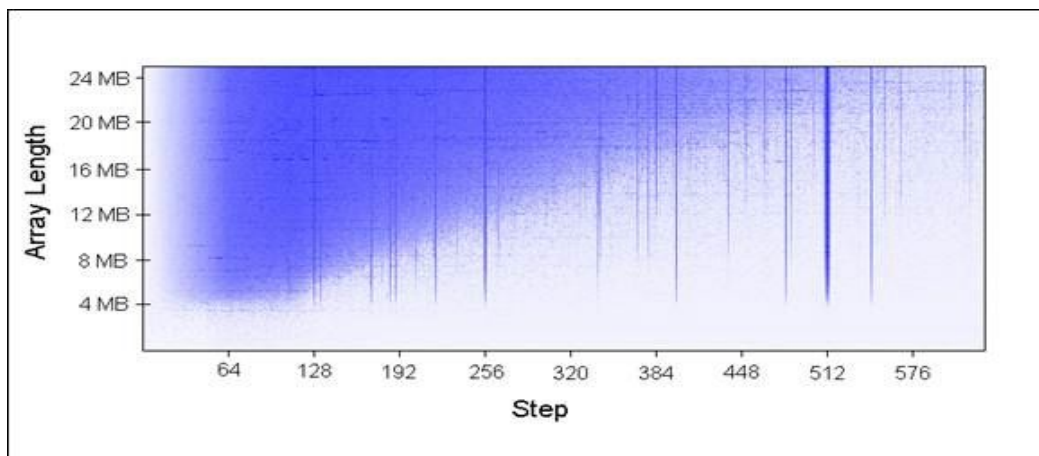
- Temporal — Data will be used again soon.
- Spatial — Data will be used in adjacent locations (for example, on the same cache line).
- Non-temporal — Data which is referenced once and not reused in the immediate future (for example, for some multimedia data types, as the vertex buffer in a 3D graphics application).

These data characteristics are used in the discussions that follow.

We will not use the data again hence the non-temporal write will be a good choice.

### why unrolling is dangerous sometimes?

I mentioned that I try to unroll the loop manually. However, it gives a terrible performance like 1.78 GB/s at reading. Why is that? This is my guess. This is a diagram from[11].





The blue color here means time. The darker, the time is longer. We see that when step is small, time is longer. However, we also notice that like 512 step, there are dark vertical line. Why is that? Because the memory is associated. Some memory locations may be mapped to the same slot in on set. If you unroll the loop badly, then it may fall in these traps. Also, if the memory is not aligned in the memory, the unrolled loop may visit two cache lines. This is also horrible.

### Is this the the best we can get?

If fact, we can get bigger bandwidth if we use the multi core programing, i.e. use OpenMP to do the measure. The following data is from [17]. This is not measured by us because we do not have OPENMP available. We can use this data because this is also from a MBP and the result from previous are almost the same. The list redo all the previous methods in OpenMP.

read_memory_rep_lodsl_omp	19.01 GiB/s
read_memory_loop_omp	22.03 GiB/s
read_memory_sse_omp	22.18 GiB/s
read_memory_avx_omp	22.21 GiB/s
read_memory_prefetch_avx_omp	22.19 GiB/s
write_memory_loop_omp	22.13 GiB/s
write_memory_rep_stosl_omp	21.25 GiB/s
write_memory_sse_omp	9.70 GiB/s
write_memory_nontemporal_sse_omp	22.13 GiB/s
write_memory_avx_omp	9.70 GiB/s
write_memory_nontemporal_avx_omp	22.13 GiB/s
write_memory_memset_omp	22.14 GiB/s

### Final result:

If we do not use parallelism, the read bandwidth should be 13.3 GB/s, the write bandwidth is 15.57 GB/s

If we use parallelism, we expect that, the read bandwidth is 22.21 GB/s and write bandwidth is 22.14 GB/s

## 3.3 Page Fault Service Time

### Methodology

A page fault is a kind of interrupt, called trap, raised by computer hardware when a running program accesses a memory page that is mapped into the virtual address space, but not

actually load into main memory. When handling a page fault, the operating system kernel generally tries to make the required page accessible at the location in physical memory or terminates the program in case of an illegal memory access.[19] There are three different types of page faults, while the major page fault is the mechanism used by an operating system to increase the amount of program memory available on demand. If a major page fault occurs, the page fault handler in OS will find a free location: either a page in memory or another non-free page in memory. The latter one will cause the data in that page be written out to make the page available. Therefore, major page fault is very expensive due to the disk latency.

In this experiment, we are trying to measure the time for faulting an entire page from disk. To implement that, we are using ``mmap`` to simulate the situation when major page fault occurs. Unlike other I/O system call(read or write), the only thing ``mmap()`` function really does is change some kernel data structure, like the page table. It doesn't actually put anything from disk into main memory at all. Instead, after calling the ``mmap()``, the allocated region doesn't even point to any physical memory. In this case, accessing this region will cause a major page fault. Only when this page fault occurs, the page fault handler of operating system will copy the data from disk to memory.[20] Since we are measuring the system performance of OS X El Capitan on the machine with 8GB memory, it will be difficult for us to measure the time for faulting a page from disk when the whole memory is totally occupied. Therefore, based on the mechanism of ``mmap()``, we decided to use this function to estimate the performance of our system coping with the page fault.

To gain the experiment data, we first created a random huge file by executing the following command in our terminal, and it will create a 3GB file in the target directory:

```
dd if=/dev/urandom of=../data/memory/test bs=1048576 count=3072
```

And then, as what we did in the last two experiments, we are using different sizes of stride to access the data from this file. Since in our system the default page size is 4KB, the sizes of stride we were using are 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB and 64MB. In each case, we invoke the ``purge`` system call first to force the system to clear cached memory from the last case in that we are using the same huge file in this experiment. Afterwards, we called ``mmap()`` function to map this huge file to memory without actually copying the data. After that, we started to access the data from the beginning of the file to the end with the distance of specified stride(4KB - 64MB). As usual, in order to accurately measure the execution time, we are using the `rdtscStart()` and `rdtscEnd()` functions as we did in the CPU experiments. Besides, we run our experiment multiple times(in one run, we will measure all strides cases for 100 times separately, and then we repeat run the whole experiment for another four times) to obtain a stable measurements and average the results. Most importantly, to get rid of overhead from other instructions, we called `rdtscStart()` and `rdtscEnd()` functions immediately before and after we accessed the data and decrease the overhead from these two functions before we added the difference of end time and start time into the total time. In addition to the fixed

position accessing in each case, we also run our experiment to access random position in the specified stride to make comparison. Our main code is shown as follows:

```
for(int i = 1; i <= 16384; i *= 4) {
    // call purge to clear cache
    system("purge");

    unsigned int STRIDE = i * PAGE_SIZE;
    unsigned int TEST_TIMES = min<unsigned int>(100, FILESIZE / STRIDE);

    // using mmap to map file into memory
    char* map = (char*) mmap(NULL, FILESIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
testFile, 0);
    ...
    for(int j = 0; j < TEST_TIMES; j++) {
        uint64_t index = min((j * STRIDE + rand() % STRIDE), FILESIZE - 1);
        // uint64_t index = (j * STRIDE); // fixed position accessing
        start = rdtscStart();
        testByte = map[index];
        end = rdtscEnd();
        diff = (double)end - (double)start - overhead;
        total += diff;
    }

    double average = total / TEST_TIMES;
    ...// store the data
    munmap(map, FILESIZE);
}
```

## Prediction

Since we are using `mmap()` to measure the performance of page fault service, the latency of accessing data is mainly caused by accessing the disk, transferring data from disk to memory and the context switch triggered by this page fault. In this case, there are two context switches, the first one occurs when the system find the virtual address hasn't been mapped to the physical address and the other one happens after the data have been copied into memory. From the last CPU experiment, we measured the average cycles of context switching to be 387233. And from the specification of our testing machine, the randomly uncached read speed is 420.80MB/sec (gained from xbench), and our CPU is 2.7GHz which means the cycle time can be  $1 / 2.7\text{GHz} = 0.37\text{ns}$ . Therefore, transferring a 4KB page will cost almost 25089 cycles. Besides of overhead from context switching and data transferring, we think additional overhead can come from locating the specific position of a file, processing the system page table when finishing the data copy and so on. Therefore, we estimated the overall overhead could be larger than 3 times of overhead from context switching, making the prediction  $3 * 387233 + 25089 = 1186788$  cycles.

## Result

First, we fixed the access position in our experiment:

File Size	4KB	2	3	4	5	Average	Standard Derivation
4KB	42,638.44	75,951.00	54,466.50	49,910.90	55,857.10	55,764.79	12,407.12
16KB	169,845.47	314,670.00	160,363.00	166,544.00	356,291.00	233,542.69	94,273.82
64KB	818,137.07	1,078,680.00	761,568.00	812,633.00	1,013,410.00	896,885.61	139,855.13
256KB	1,020,154.96	1,616,510.00	1,287,300.00	1,402,970.00	1,328,720.00	1,331,130.99	215,194.73
1MB	918,660.51	1,201,010.00	1,161,440.00	1,212,800.00	866,596.00	1,072,101.30	165,960.45
4MB	995,173.65	1,439,110.00	1,100,810.00	1,201,090.00	1,047,890.00	1,156,814.73	175,199.80
16MB	1,105,601.68	1,562,440.00	1,076,520.00	1,156,630.00	1,069,100.00	1,194,058.34	208,782.40
64MB	968,683.54	1,497,260.00	1,452,330.00	1,434,490.00	1,278,780.00	1,326,308.71	216,196.46

And then we also make a random access in our experiment:

Stride Size	1	2	3	4	5	Average	Standard Derivation
4KB	48,537.55	57,296.10	45,611.30	48,990.80	51,771.30	50,441.41	4,410.62
16KB	215,655.91	303,821.00	254,153.00	198,809.00	256,286.00	245,744.98	40,813.51
64KB	685,846.23	977,724.00	828,013.00	876,459.00	732,014.00	820,011.25	116,046.82
256KB	1,004,196.25	1,439,260.00	1,522,610.00	1,148,220.00	938,188.00	1,210,494.85	259,970.02
1MB	1,122,575.04	1,142,050.00	1,286,580.00	1,203,890.00	1,026,040.00	1,156,227.01	96,883.65
4MB	1,046,327.05	1,220,330.00	1,349,520.00	1,385,050.00	1,326,030.00	1,265,451.41	137,015.09
16MB	959,775.68	1,240,150.00	1,134,670.00	1,102,700.00	1,401,980.00	1,167,855.14	164,836.66
64MB	960,570.44	1,383,780.00	1,353,290.00	946,458.00	1,300,180.00	1,188,855.69	216,966.60

## Analysis

First, in those smaller stride size (like 4KB and 16KB) results, we noticed that there are a lot data which are obviously smaller than the rest, as 3000+ vs 1000000+. However, in those larger stride size (like 64KB and the rest), although the standard derivation is bigger than the one of

smaller stride size, they concentrate on the range of 800000 to 1300000. Therefore, we think in the smaller stride size result, not all data are gained when a page fault happens. Instead, we strongly believe that those relatively smaller data are generated when our function accesses the data directly from memory rather than from the disk. In such case, these data can be considered irrelative since the goal of this experiment is to measure the time for faulting an entire page from disk. Based on the average result from 256KB to 64MB, the time for faulting pages from disk is among 1,100,000 to 1,300,000, which is very close to our prediction. However, we also notice that the standard derivation of all cases is very high. We are still figuring out the reasons and guessing it may be caused by the time-consuming overhead of disk accessing and context switching, since we also met such high standard derivation in the CPU Context Switch experiment last time.

Besides, by comparing the results of the 4 smallest stride size(4KB-256KB) in both fixed and random accessing, we noticed that when the page fault occurs, the handler of operating system doesn't simply load only one specific page of data into memory. In the chart of 4KB, each peak has the same distance of 16 to each other and in 16KB, each peak has the same distance of 4. But in the char of 64KB or those bigger than 64KB, the results become consecutively large without a series of relatively small number. Therefore, we are guessing that the actual number of pages of data being load from the file is 16. There are a lot of good reasons for doing that, and we think the most important is that loading more pages in one time can bring better performance improvement than only loading one page in one request. In most cases, as for a huge file, it is more possible for a user to read consecutive data. This mechanism is also adopted in other services in our modern operating system. We also noticed that in the results of 4KB and 16KB, the data distribute in a similar way since, in these cases, the actual size of page fault is bigger than the stride and whether randomly accessing or fixed accessing any position of the data between two peaks, they all read from the memory. However, we also notice that when the size of stride is bigger than or equals the actual fault page size, in randomly accessing experiment, we can meet some data which have a very small value, but in fixed accessing case, this situation hardly happens.

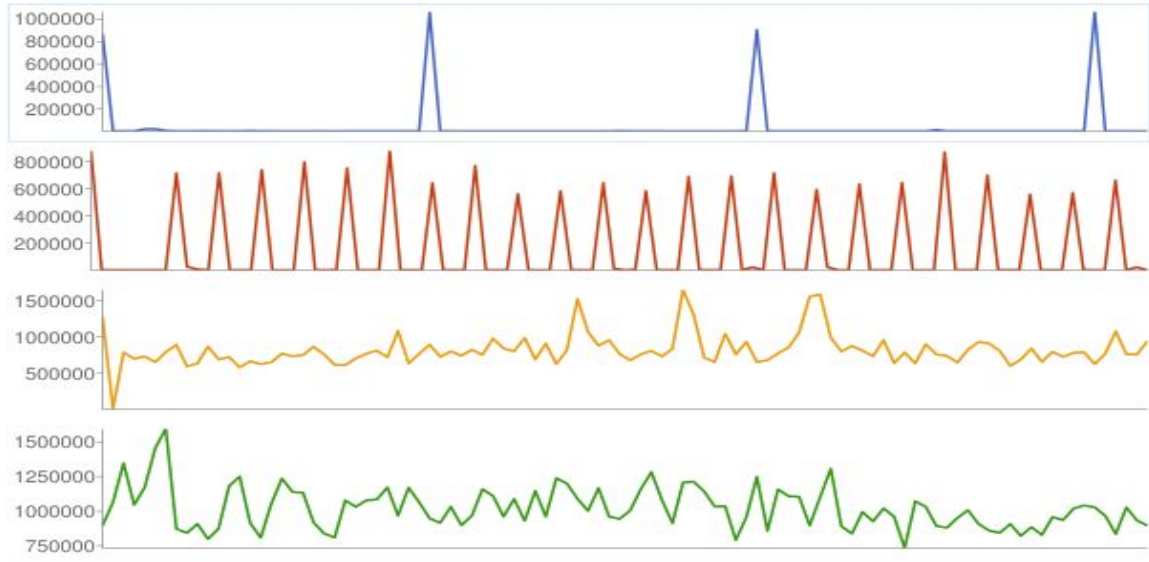


Fig x. In **fixed** position access, the page fault service time of 4KB, 16KB, 64KB and 256KB(from the top to bottom). The x-axis represent the number of the test and the y-axis is the cycles of accessing data.

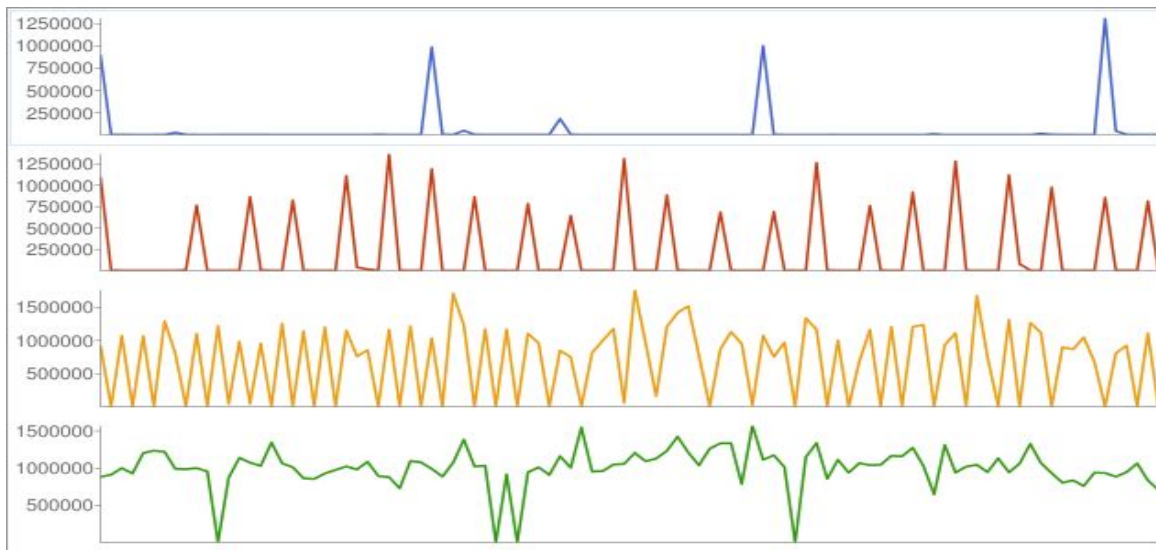


Fig x. In **random** position access, the page fault service time of 4KB, 16KB, 64KB and 256KB(from the top to bottom). The x-axis represent the number of the test and the y-axis is the cycles of accessing data.

The cost of faulting pages(from the last paragraph, we believe each time page fault occurs, the system will load 16 pages instead of only one page) from disk is 1,216,082.81 on average, which is  $1,216,082.81 / (1 / 2.7\text{GHz}) = 1,216,082.81 * 0.37\text{ns} = 449950.6397\text{ns}$ . Thus the cost of accessing one byte from disk through a page fault can be evaluated to  $449950.6397\text{ns} / (4096 * 16) = 6.678125\text{ns}$ . From the latency of accessing a byte from main memory, gained from the previous experiments in which the read bandwidth is 9.18614GB/s =

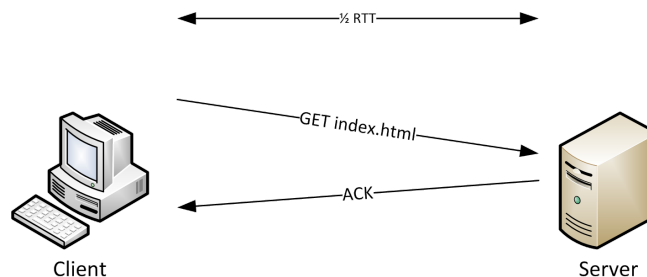
read 1 Byte in 0.10138ns, it is almost 66 times more expensive to accessing one byte from disk via a page fault.

## 4. Network

### 4.1 Round Trip Time

#### Methodology

The **round-trip delay time (RTD)** or **round-trip time (RTT)** is the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgment of that signal to be received.[26]



[27]

One simplest way to profile the RTT is to use Ping. Ping is based on ICMP(Internet Control Message Protocol) which is at **internet layer**. Ping operates by sending Internet Control Message Protocol (ICMP) echo request packets to the target host and waiting for an ICMP echo reply. It measures the round-trip time from transmission to reception. [28]

Since mac OS has the ping utility comes with the system. We can directly use it. Here, we use default package size to do the profile. Check the man for ping, we know its default package size is 56 byte.

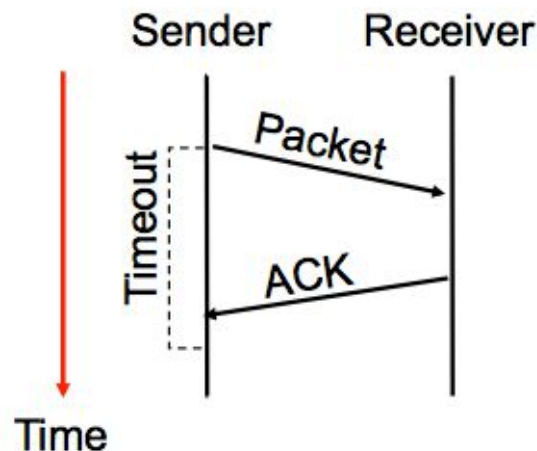
```
-s packetsize  
Specify the number of data bytes to be sent. The default is 56,  
which translates into 64 ICMP data bytes when combined with the 8  
bytes of ICMP header data. This option cannot be used with ping  
sweeps.
```

In addition, let take a look at ip datagram for ICMP. We will later compare this with TCP package.

IP Datagram				
	Bits 0–7	Bits 8–15	Bits 16–23	Bits 24–31
IP Header (20 bytes)	Version/IHL	Type of service	Length	
	Identification		flags and offset	
	Time To Live (TTL)	Protocol	Checksum	
	Source IP address			
	Destination IP address			
ICMP Header (8 bytes)	Type of message	Code	Checksum	
	Header Data			
ICMP Payload (optional)	Payload Data			

[28]

Another way to do measurement is to use TCP protocol. We call this method as **TCPping**. We mimic the ping in this case. Basically, we make a client to write **some** data in the IP datagram and an echo server which responds everything it receives back to the sender. We will discuss the detailed client and server implementation later, but now we start from why this works from the basic idea about TCP. For each package in TCP, we have this model.



[29]

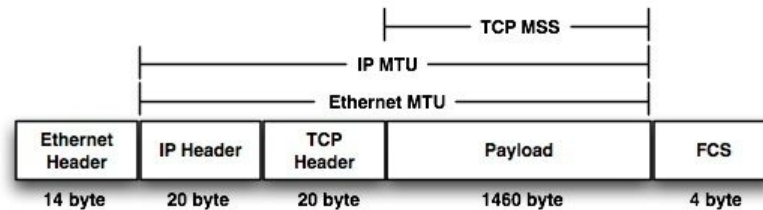
The sender uses ACK to make sure that receiver receives the package. The thing is that whether this model gives different RTT from our ping method because this extra ACK?

The answer is no. The receiver, the echo server in our case, will send the ACK package first but immediately follows to send the echo package. Hence, we start the timer at the time point



when client sends the package and stop the timer when client receives echo from server. This time gap is exactly the round trip time we need.

We now prove the feasibility of our TCPping. The next question is, how large is the package we should send? Let check the model of a TCP package.



[29]

From this, we find the payload of a TCP package is 1460 byte. (MSS means max segment size). Here, we assume MTU as (Max transmission Unit) as 1500 byte because it is the default size of ethernet package. If you wonder what is MTU, great question. We will discuss it in the section of max bandwidth section. For TCPping, we definitely will not send a big package, so we can save us from that problem temporarily.

Now, our problem become what size of data should we fill in the payload? We see that ICMP header is 8 bytes from previous discussion and IP head is 20 bytes. The payload of ICMP is 56 bytes. So, it is debatable that we should stuff the package with 56 bytes or less, like 44 bytes, which comes from  $56 - (20 - 8) = 44$ . We decide to use 56 bytes in our test case. The reason is that in this profiling, we also want to test the difference between ping and TCPping. Hence, to keep the package size at the same level is a good choice.

Still, it is controversial that whether we should let TCPping to carry less data because it has bigger overhead. This does make some sense, however, if you want to let these two at the same startline, we should also consider that ICMP is an internet layer protocol and TCP is a transport level protocol. Hence, it is nearly impossible to treat the TCP and ICMP as the same thing. Just compensate the data size is not enough to make us able to treat these two as the same thing. Hence, in general, we choose to use 56 bytes as payload size. Both these two methods use 100 samples to get min, avg, max and deviation.

### Design of Client and echo Server

The pingClient uses socket to connect the server. It will initialize a 56 byte buffer and send it out. The timer start right before we use send() to send data. The timer stops when we finished receiving the echo package.

The pingServer echoes everything it receives. Since we are using 100 times to test the server. We want to keep the server running to handle multi-client. Multi-client server is a classic problem in undergraduate network class, we reuse part of the code. We use select() there to

detect if there any client is sending data, because we do not want a client block the whole server. Also, in our scenarios, only one client runs at one time, so we do not need to consider the concurrency issue to affect final result.

The idea is to maintain a client pool. Once a new client connect, it will create a thread for it and put it in the client pool. Once the client close the connection, this thread will be removed from the pool.

### The hardware for client and server

For testing loopback, we can directly use our test device. For test remote interface, we use two same mac. Hence, the network interface should be the same. One mac uses ethernet card wired connect to router via RJ45 and another connects to router via Wifi.

### Prediction

We predict that the result should be 1 ms for loopback and for remote it is hard to say because it depends on the network.

The RTT of loopback should be less than the remote RTT.

The ping result should be less than the TCP ping.

### Result

	min(ms)	avg(ms)	max(ms)	stddev(ms)
Local ping	0.061	0.080	0.122	0.009
Local TCPping	0.049641	0.089661	1.354182	0.130014
Remote ping	1.420	8.098	136.922	16.996
Remote TCPping	1.144046	1.996801	10.659761	1.378705

### Analysis

There are some really interesting results!

First, for RTT, minimum value is more important than the average value because it reveals the minimum and how fast a package could travel between two computers.

However, we find an astonishing thing in our result! The TCPping is faster than ping, both in remote and local test! How could this happen? We predicted that the TCP should slower than ping because TCP is one layer higher than ICMP and TCP will send ACK to make sure data is delivered. So, why is the performance of TCPping better than ping? Well, after discussion, we figure out a reasonable explanation.

First, in TCPping, there must be no package lost at the minimum time value. So the ping and TCPping are at the same level at this point. Second, in TCPping, we use timer after we have a stable connection between server and client. So there is nothing like arp or other setup process needed in TCPping. However, ping needs to find the server according to IP table and arp table. Also, server needs to assign a port for listening to the client ICMP package.

In general, we believe is the setup overhead for ping to lead to this result. According to following section, we know that the setup time for local TCP is about 0.009 ms. If we fix the TCPping by adding this setup time  $0.009 + 0.049641 = 0.059s$ , which is almost the same as ping.

Another interesting thing is that the deviation of ping for remote test is so big. It is 16 ms but the minimum value is just 1.42 ms! It is easy to spot that the very reason is that the maximum time for remote ping is 137 ms. Why does this happen? Because the network is not stable, for 100 times test, there maybe some package is delayed or lost. For TCP, if it does not receive ACK, then it will send the package again. But ping does not have this mechanism, hence it just waits until it gets echo. Sometimes in our test, there also has some timeout errors. (The data we present in the table does not cover the case for timeout. We keep running until we get 100 ping without any timeout error) Again, we experience the delicate design of TCP which makes our data transformation stable and efficient.

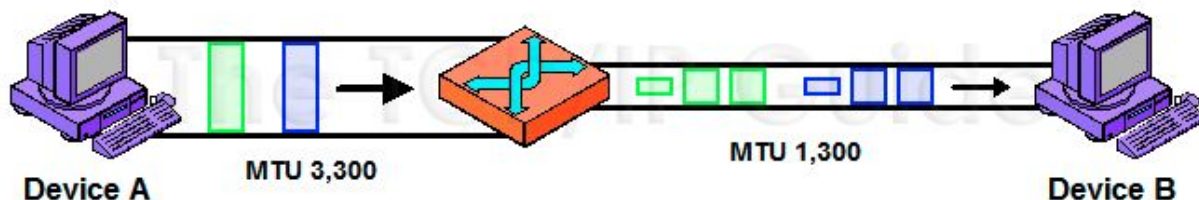
Hence, we use the minimal value we get for all the test.

RTT Time	Time (ms)
local	0.049641
remote	1.144046

## 4.2 Peak Bandwidth

### Methodology

MTU is maximum transmission unit.



[30]

In order to achieve max bandwidth, we need to utilize the max MTU in for data channel. In another word, for our physical layer.

One solution is to get the MTU for our data channel, like wifi. First fill the IP header, TCP header and other additional tail info into MTU. Then fill the max amount data in the the payload.

Another way to think about this problem is that we can use socket to send a very big buffer. In order to send it, the TCP will stuff the as much data as possible in payload each time. Hence we can be sure that every payload is stuffed with the MSS each time except the last transmission. If we keep the package size keeps going big, this effect should be less and less.

Hence, our strategy there is that we start the buffer size at 512 bytes and times 2 until it reaches 32MB. Each buffer size we will test 100 times to get a stable result.

In addition, since IP header and TCP head has size, there are **3% overhead** about the total data transmitted. However, we usually mention the bandwidth as the effective data we want to use, hence we do not recalculate for this 3% but just directly use data transmitted / total time.

### **Where is the timer?**

An interesting question is where the timer is and when we should start it and stop it. If we place the timer at the client, it works. We can start when we send the data and stop it when we receive a simple ack msg from server. Then the time should minus one RTT to get the real time for data to transmission. But if you consider a scenario, that we have a very large RTT. The the time we measure is largely wasted on the RTT. It is hard to say that this channel's bandwidth is xxx based on our measurement. We could measure each RTT time for each package, and deduct them later. But this will lead to other questions.

Hence, we adopt another solution here. We place the timer on the server. The timer start when the first data received and stops when the last data is received. In this case, we do not even need to consider the RTT, because the timer does not start until the actual data is arrived! In addition, they reason why server knows what data is the last piece is that the we design our test cases. So we know the exact size of the data for each test point.

Hence, although it involves to modify the server's code but it is worthy. We write two code here. One is bandwidth Client and the other is bandwidth Server.

### **Prediction**

We has a 100MB network interface on mac. Hence we expect that we get 100MB on loopback and the remote depends on the school library network. We estimate it is 50Mb at least. Also, we estimate that the bandwidth should increase as we grows the buffer size.

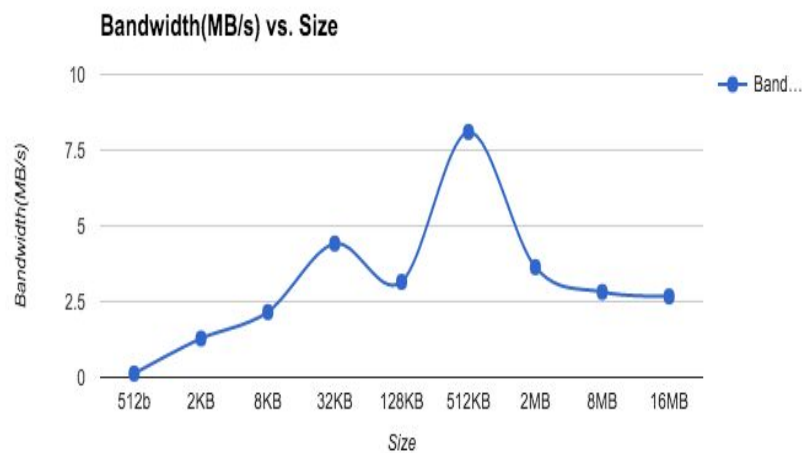
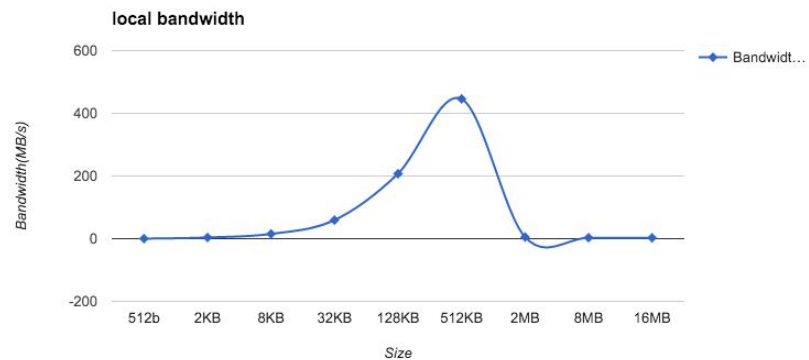
The 512 byte should be the lowest bandwidth because it does not fully use the MSS and as we increase the bandwidth should be become stable. Because we stuff the MSS with data as much as possible. (In fact, we will notice some interesting results which contradict this estimation).

## Result

Since we need peak bandwidth here, so we present the data with maximum value.

Local bandwidth

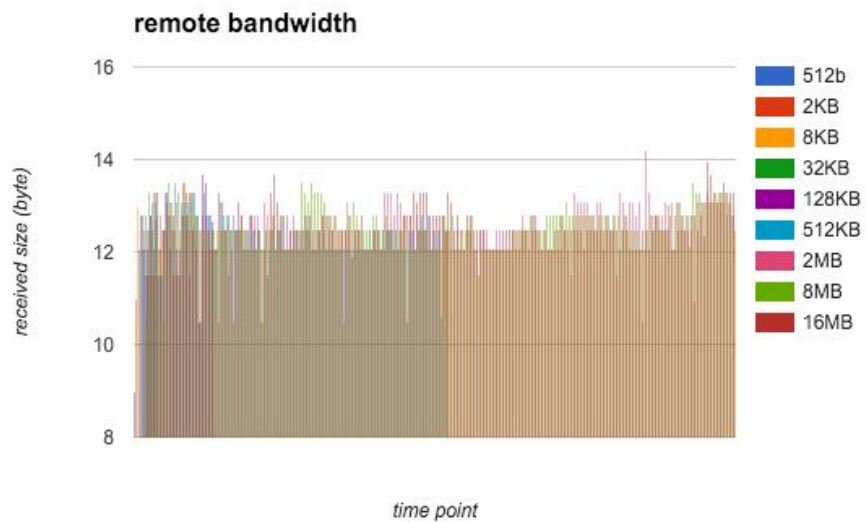
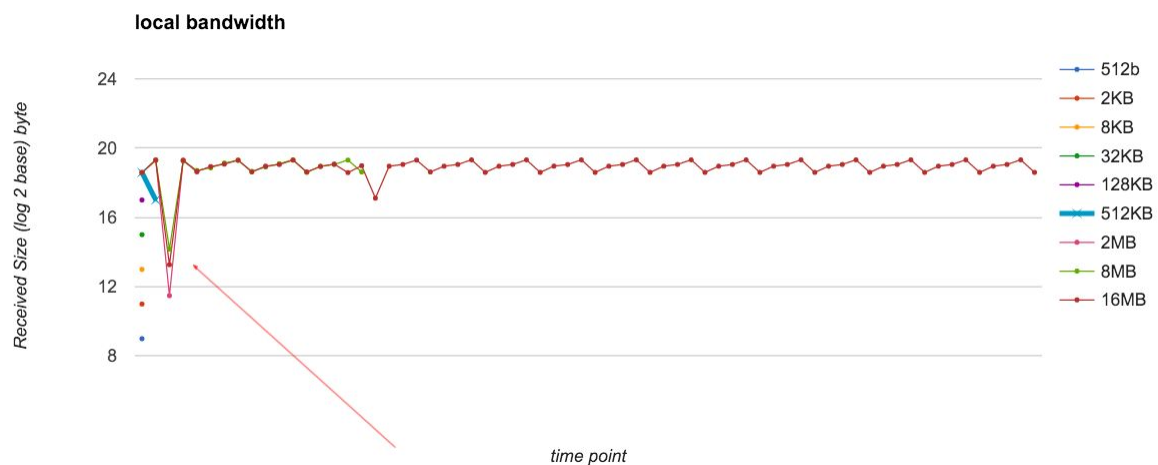
Size	Local Bandwidth(MB/s)	Remote Bandwidth(MB/s)
512b	0.17136	0.122587
2KB	3.868149	1.281212
8KB	15.060821	2.153319
32KB	59.239908	4.414671
128KB	207.241512	3.152048
512KB	445.847374	8.100716
2MB	5.145305	3.637927
8MB	3.080629	2.818187
16MB	2.749426	2.670831



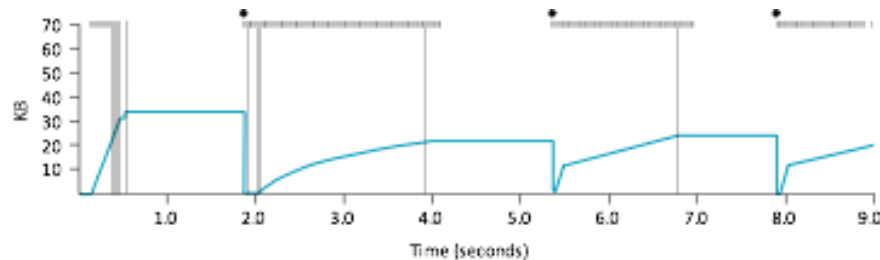
## Analysis

Obviously, this only partially satisfy our prediction. We get the increasing bandwidth from 512b to 512Kb. However, we get a bandwidth dropping in the following instead of a stable bandwidth. Why is that?

Let's check data size the server received each time, i.e. the data server receives at each read().



For previous two diagrams, the y-axis is the size of received data and x-axis is time. It is easy to find that as the size become bigger, there is a valley after the beginning point and there are bumps in the following. Why is this ? It is easy to explain. TCP has congestion control and TCP is slow start!



[31]

See our figure fits the characteristic diagram of TCP. This explain why we cannot measure the TCP's real max bandwidth.

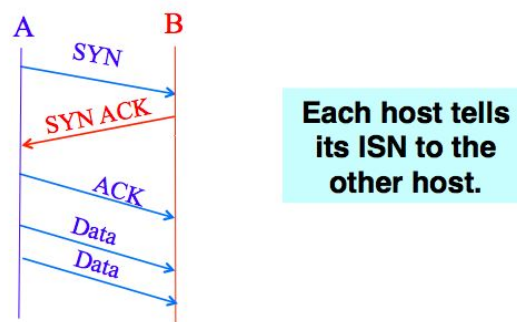
In general, for local TCP, it is 445.8 MB/s, for remote, it is 8 MB/s. The actual values for should be bigger than our measurement.

### 4.3 Connection overhead: setup and teardown

#### Methodology

Let do a quick review about setup and teardown in TCP connection.

TCP has three-way handshake setup.



[30]

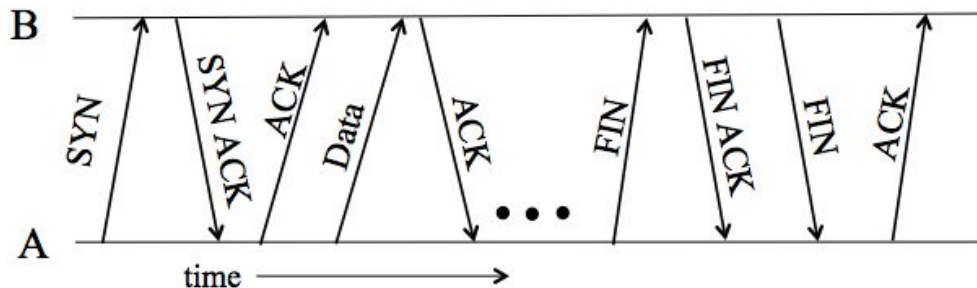
For a TCP setup, it will go through these three steps.

- Host A sends a SYN (open) to the host B
- Host B returns a SYN acknowledgment (SYN ACK)
- Host A sends an ACK to acknowledge the SYN ACK

Hence, if we treat the RTT as 2 unit time, then the setup time should be the 3 unit time. A very intuitive way to code this problem is to start timer before the client use connect() and stop timer when the connection is established. This seems work, however, you will notice

that this only measure the 2 unit times. Because it just includes the time gap between first ACK and the SYN ACK from the server. If we want to stop timer when the server actually receive the ACK from client, it will be extremely difficult. Hence, let us rethink about the setup time. The setup time we measure here is not the exact of three-way handshake time, but the perceivable time for the client that it make sure that it has established a connect to the server and succeed.

Tear down:



[29]

Let us recall the mechanism of the termination of a TCP.

Closing the connection

1. Finish (FIN) to close and receive remaining bytes
2. And other host sends a FIN ACK to acknowledge
3. Reset (RST) to close and not receive remaining bytes

Hence, if the RTT is 2 unit time, then the teardown time should be 4 unit time.

However, we meet the same dilemma as we met in the measurement time. In fact, in this case, the thing is even worse. If we start the timer before the `socket.close()` is called and stop timer the timer when `socket.close()` is finished. What are we measuring? In fact, we are just measuring the time of destroying a file descriptor. Remember, everything in the linux can be treated as a file. There is alternative function called `socket.shutdown()`. It takes `shutdown(SD_SEND)` or `shutdown(SD_BOTH)`. The thing is that this method only block the transmission or reception for the socket, so it is also not applicable for our case. In general, we can treat the tear down time as the perceivable time for client to end a TCP connection.

## Prediction

For setup time, it should be nearly the same as the RTT time or just a little bigger than RTT for some overhead like binding port.

For teardown time, it should be very fast since it only sends a FIN and destroy the socket. It should within 1ms. Hence, the remote teardown and local teardown is nearly the same.



## Result

	min(ms)	avg(ms)	max(ms)	stddev(ms)
Local setup	0.085778	0.108107	0.151910	0.020768
Remote setup	1.351129	2.614392	5.914362	1.347776
Local teardown	0.010013	0.012127	0.018655	0.002326
Remote teardown	0.013924	0.022124	0.039323	0.008834

## Analysis

Finally, there is a set of data totally fits our estimation.

In previous, we get RTT is about 1.14 for remote, hence for remote setup is about 1.35. These two values are pretty close. The reason why remote setup is a little bigger is that the connection overhead on local machine.

Also, for local teardown 0.010013 and remote tear down 0.022124, they are also very close. Because teardown is just simply closing the socket descriptor so it really does not matter whether it is remote or not.

# 5. File System

## 5.1 Size of File Cache

### Methodology

The overall approach we use to measure the file system cache is to sequentially access files with difference size and then to measure the disk bandwidth. We need to prepare the input file with varied sizes first. A shell script was written to generate eleven input files filled with randomly generated characters. The size of input files range from 32 MB to 32768MB. Each input file's size is twice previous input file.

Before we run the program on each input file, we use `sudo purge` to free up inactive memory in the machine. Within an infinite loop, we do `read(fd, buf, BLOCKSIZE)` repetitively until the return value is zero. If it returns zero, we reach the end of the input file. We are guaranteed that the input file is in the cache after the first time we read it. Then within another infinite while loop, we do `read(fd, buf, BLOCKSIZE)` repetitively from the same input file. Before

read(), we use rdtscStart() to record the starting time. After reading a block size of bytes, we use rdtscEnd() to record the ending time. If the bytes we have read equals to the size of the input file, we stop the program. Now we can get the disk bandwidth by calculating the number of bytes read per cycle for each input file.

Assume an input file has size less than the file system cache's maximum size. The first time that an input file is read, it will be cached in the memory. After that, whenever we read the same file again. The input file is cached in the memory. If the file is already cached in the memory, reading bytes from the file should be very fast since we don't have to access the same input file accessing the disk. We will observe performance by measuring the disk bandwidth.

However, for input files that have sizes exceeding the maximum size of the file system cache. The entire input file cannot fit in the file system cache. Since such large input files do not reside in file system cache the first time they are read, the following read from these files will lead to cache miss. In order to keep reading bytes from the large input file, we have to read data blocks from the disk. The time of reading data blocks from the disk will be longer than that from the cache.

Since the time of reading data blocks increases significantly, we can observe a dramatic change of the disk bandwidth when reading bytes from input files with different sizes. The dramatic change reveals the size of the file system cache. When the input size is 8GB, we notice the bandwidth is 0.27 bytes/cycle, which is 749MB/s(close to uncached sequential read bandwidth for SSD). We know that the file cache size is between 4GB - 8GB.

Once we can narrow down to the range of the input files' sizes, we can do the same experiment on another list of input files with finer granularity on file sizes. We decide to increase the size of files by 100MB. By doing so we can achieve the file cache size accurately.

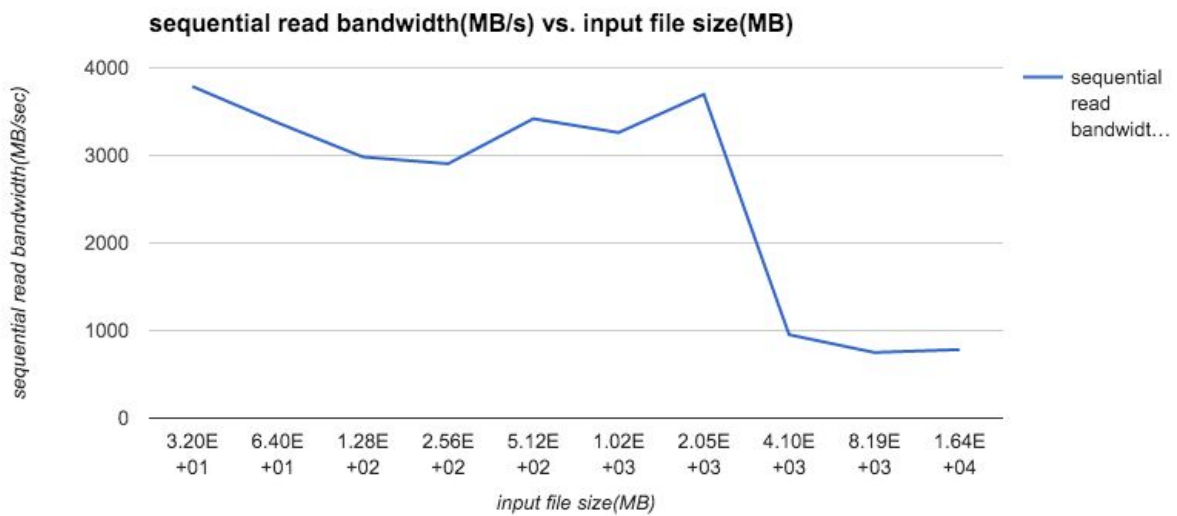
## **Prediction**

The machine that is being tested has a RAM with size of 8 GB. Other than the memory needed for kernel, the rest of memory can all be used for file system cache. So we just have to figure out the memory needed for Operating System's processes.

By using activity monitor, our machine has 1.85 GB wired memory. Wired memory is a part of memory that has to stay in RAM since it can't be compressed or paged out to the startup drive. The statistics also shows that the machine has 911 MB compressed memory. There is only 616MB cached files. There are about 3.12GB memory space used by OS and user-level processes. We estimate that the file system cache size is around 4GB.

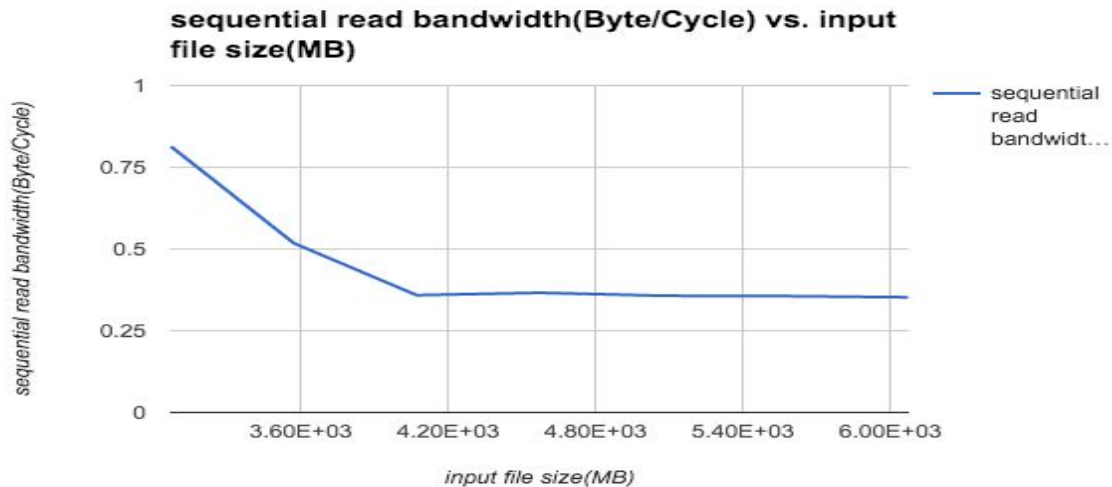
## **Result**

input file size(MB)	Read Bandwidth(Bytes/cycle)	sequential read bandwidth(MB/s)
3.20E+01	1.401525	3787.905405
6.40E+01	1.248114	3373.281081
1.28E+02	1.10351	2982.459459
2.56E+02	1.074822	2904.924324
5.12E+02	1.264974	3418.848649
1.02E+03	1.206486	3260.772973
2.05E+03	1.368072	3697.491892
4.10E+03	0.352402	952.4378378
8.19E+03	0.277427	749.8027027
1.64E+04	0.289645	782.8243243



input file size(MB)	sequential read bandwidth(byte/cycle)
3.07E+03	0.813587
3.57E+03	0.517686
4.07E+03	0.358726

4.57E+03	0.366098
5.07E+03	0.357054
5.57E+03	0.356015
6.07E+03	0.352255



## Analysis

The results of our experiment show that the sequential read bandwidth remains around 1.2byte/cycle(3243MB/s) when the file size is less than 2GB. But the bandwidth drop to 0.35byte/cycle when the file size is 4GB. There are two 4GB RAM on the machine. Using “top” command, I see that 1.5GB is for unused memory. Other processes running on the machine are also trying to use cache space. The computer can dump that RAM in a nanosecond if it were needed for something else. The sequential read bandwidth starts to decrease after the input file size is bigger than 2GB.

When the input size is 4GB, we notice that the bandwidth is 0.35 bytes/cycle, which is 952 MB/s. When the input size is 8GB, we notice the bandwidth is 0.27 bytes/cycle, which is 749MB/s. The uncached sequential read bandwidth for SSD is 753 MB/sec. This indicates that contents of file cannot be cached since the file size is too big for the file system cache. We know that the file cache size is between 4GB - 8GB. For input files with sizes from 3GB to 6GB, the bandwidth gets stable at 0.35MB/s when the file size is bigger than 4GB. The input file’s size varies from each other with a finer granularity. The cache size is 4GB, which matches with our prediction.

## 5.2 File Read Time

### Methodology

In this section, we are going to compare two types of file read time: sequentially reading and randomly reading without the influence of data caching. In order to calculate the time of making an actual disk read operation, we decide to use some flags to get rid of automatically data cache features of those file operation functions in C++. Since we are holding all experiments in OSX, we can't use `O_DIRECT` and `O_SYNC` when we call `open()` which is only available in Linux. [32] Instead, in BSD system like Mac OS, there is a function `fcntl()` which can turn data caching off/on. Therefore, in the rest experiments, we use this function to disable data caching before we invoke `read()` function. Another approach to get rid of influence of memory caching, like we did in those previous memory experiments, before we start these file experiments, we manually execute the `purge` command to clear out file cache in our system.

Learned from the command `stat -f %k .` and `diskutil info / | grep "Block Size"`, our testing environment machine use 4K bytes as a block size. Based on this information, we decide to capture the read time for different sizes of random file from size 4KB to size 64MB with the difference of a factor of 4 to the previous file. Besides, in order to keep the standard deviation low across the experiment, we repeat each experiment for 100 times. For sequential access, we read the file from the beginning to the end and each time we get 4K bytes from the file. To avoid introduce any irrelevant overhead, we use our previously decided `rdtscStart()` function directly before we call the `read()` function and use `rdtscEnd()` function directly after `read()`. In each of these 100 tests, we sum up the difference of the results of these two `rdtsc` function, and use the total cycles to divide file block size to get the average block read time. Similarly, we design the random access experiment using the same approach of the sequential access one. The most different part is to add `lseek()` function [34] before calling the `read()` function and include the time of calling `lseek()` function to our final result. `lseek()` function is used to reposition read/write file offset. Therefore, we utilize the `rand() % blockNumber` to generate a random offset of the target file, and then use this offset to reposition of current offset by calling `lseek()` and finally call `read()` to read the 4K size of data from the new position. In the random access experiment, we randomly read the file for the times equals to the block number of the target file and each file has been test for 100 times as sequentially read test.

### Prediction

According to our vendor's specification, the maximum uncached read throughput during 4K blocks test is 84.19MB/sec, or 0.046ms/block. Since sequential access of a file is mainly doing data accessing hardly affected by seeking overhead, we are guessing the average sequential read time will be close to the performance data given by our SSD vendor.

As for random access, there is another overhead, seeking time, having influence on the result, which will cause the average time is much slower than sequentially access. Fortunately, we are using SSD as our storage device, which is known to have much better I/O performance than the traditional hard disk drive commonly used in most of our lecture paper. Learned from the Wikipedia, in common SSD benchmark, seek time is about 0.1ms. Given the estimation we made in sequential access and our benchmark code, the average block read time in random access includes the seek time and data accessing. So we predict it will take  $0.1 + 0.046 = 0.146\text{ms}$  to randomly access a block.

## Results

First, we use 4KB to 64MB files to read in our local sequential read experiment:

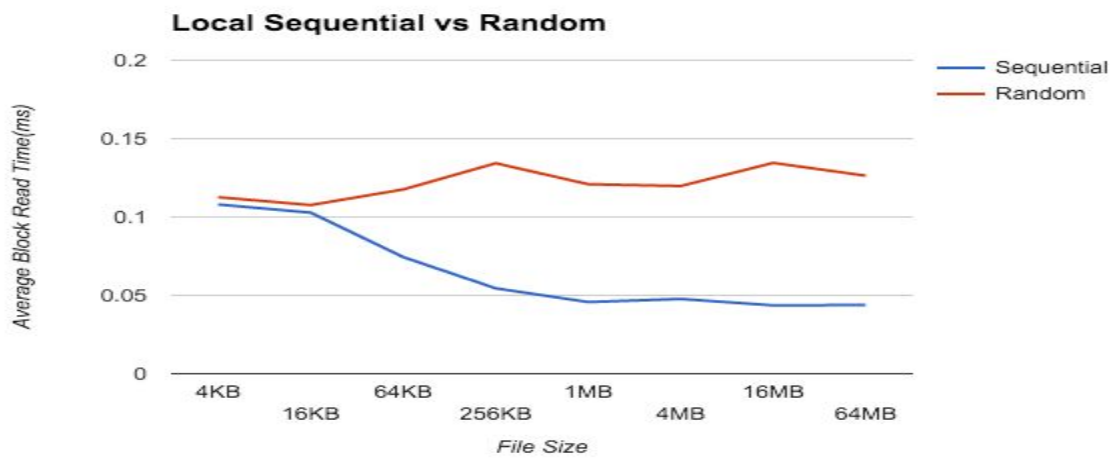
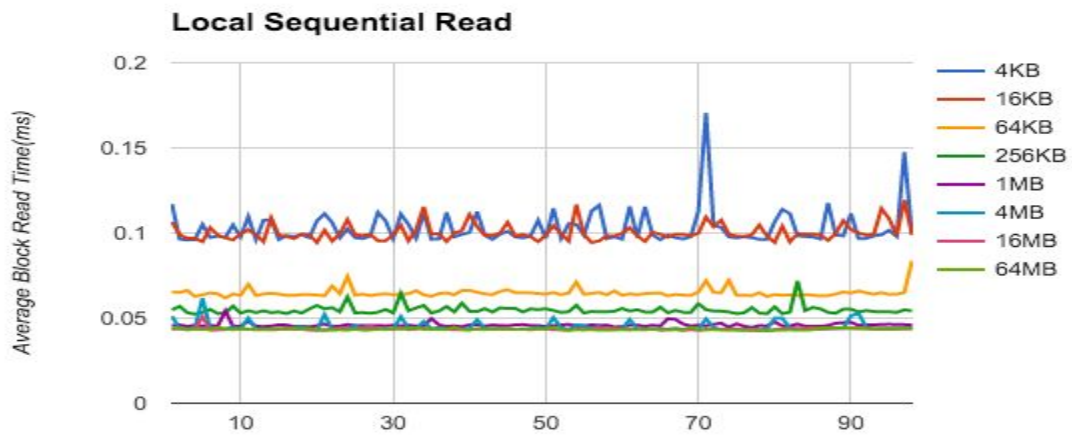
File Size	4KB	16KB	64KB	256KB	1MB	4MB	16MB	64MB
Average	0.10798	0.10277	0.07442	0.05449	0.04572	0.0477	0.04361	0.04385
Standard Derivation	0.01278	0.00517	0.01295	0.00041	0.00034	0.00476	0.00035	0.00038
Min	0.09895	0.09912	0.06527	0.0542	0.04548	0.04433	0.04336	0.04359
Max	0.11702	0.10642	0.08358	0.05478	0.04597	0.05106	0.04386	0.04412

And then we also make a random access in our experiment:

File Size	4KB	16KB	64KB	256KB	1MB	4MB	16MB	64MB
Average	0.11262	0.10761	0.11757	0.1343	0.12091	0.11976	0.13452	0.12634
Standard Derivation	0.01214	0.00591	0.00066	0.00187	0.00644	0.00236	0.00081	0.0077
Min	0.10404	0.10343	0.1171	0.13298	0.11635	0.11809	0.13394	0.12089
Max	0.12121	0.11179	0.11803	0.13562	0.12546	0.12143	0.13509	0.13178

## Analysis

After we plot these 100 average block read time in a distribution graph, we find something interesting results beyond our prediction.



- 1) Compared to our estimation from the data tables in Results section, the average read time we gained is very close to our prediction. As for sequential access, the results of bigger files (like 1MB-64MB, from 0.043 to 0.047 ms) are close to our predicted value 0.046ms; and for the random access, the results of all files (from 0.117 to 0.134ms) are also close to our predicted value 0.146ms. Actually, the seek time we gained from Wikipedia is generated a few years ago, while our test machine is released in early 2015, and we believe our test SSD device can have better performance than the one in Wikipedia. Therefore, the random access time is lower than what we expected.
- 2) From the result table of sequential access experiment above, the average block read time of smaller size files is obviously larger than the read time of bigger files. Although the main overhead of sequential access of a file is the data process, the first time seeking beginning offset of target file in SSD is not to be neglected for smaller files. As we described in the Methodology session, the average block read time is gained from the total time of reading a whole file divided by the block number of this file. Therefore, bigger files have more blocks leading to more time on data processing and reducing the influence of the first time seeking, and finally have a lower average read time. That's one of the reason why most benchmarks use a big file to capture storage device performance.
- 3) Although in sequential read experiment, bigger files can have better performance than smaller files, it doesn't mean the performance can keep increase without limit. From the result table and the distribution graph, there is an obvious decrease in read time when the file size increases to 64KB, but the average read time becomes very close when the file size keeps increasing. From the Local Sequential vs Random graph, we can see the blue line becomes flat and smooth when the file size is bigger than 1MB. The main reason is because every storage device has its own disk bandwidth and the performance will be limited as the amount of data read grows bigger.
- 4) Although we keep saying that in sequential access experiment, the main overhead is caused by the data process(read from the disk), we notice the distribution of average block read time is not as flat as we expect, in other words, the "sequential" access might not be sequential. We can still see peaks in Local Sequential Read graph. The main reason to cause this "seems random" result is we can't insure the allocation of our testing files as sequential as possible, especially for those bigger files. In this case, we might have to fetch metadata on some other partition and read file data from non continuous spaces. Therefore, even our code can make sure we invoke `read()` function sequentially, we still can promise the system can actually read sequentially from the disk.
- 5) From the random access result table and the distribution graph, we notice that the average time of different files is very close to each other since every time we read a block we need to randomly pick an offset. The seek overhead can't be amortized with



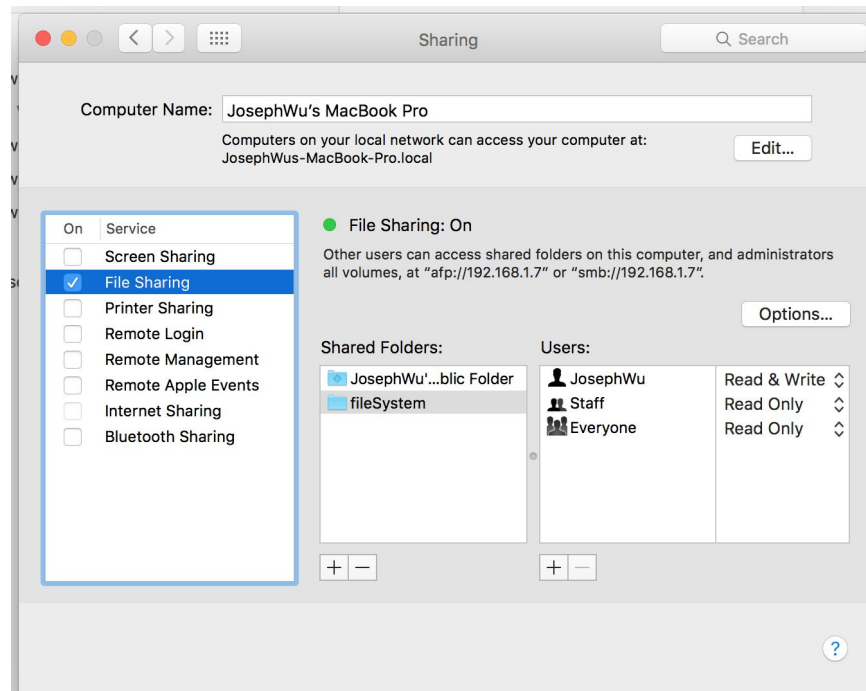
the growth of the file size as sequential access experiment does. Therefore, we can obviously notice that the overall read time of random access is always more than the sequential access. However, we also notice from the distribution graph of random access, the read time distribution is not as flat as sequential read. We think the main reason for this case is because we use `rand()` to generate the next block we will read, it is possible that we will visit “closer” blocks while sometimes we will visit “farther” blocks, making the seek time can also vary randomly. Therefore, the standard deviation of random access is bigger than sequential access. However, both experiments’ standard deviation is very small to make sure that our approach is correct.

- 6) There is a slight increment in random access experiment when file size grows but the read time seems to keep unchanged in some arrange. In another word, judged from the result we think the performance of reading small files in random pattern could be better than reading large files. Although we tried hard to search relevant information of this situation, it is still very difficult for us to make a conclusion about the reason why in random pattern, the distribution of the file size vs average read time is different from the distribution in sequential pattern. In sequential access, the read time slightly decreases and in random access, the read time slightly increases when the file size grows. The possible reason we think is some optimization mechanism within SSD, which might be very helpful for SSD to randomly access small files. Unfortunately, we are still not so sure yet.

## 5.3 Remote File Read

### Methodology

In this section, we are going to use the another Macbook Pro with the same specification as our remote file server and the original Macbook Pro as file client. Instead using other NFS tools, we decide to use Mac OSX self-installed ‘Sharing’ feature[35] to share a file folder in our home wifi environment (same with the testing environment of the network experiment). In MAC OS X El Capitan operating system, when you connect from a Mac to another Mac using file sharing, your MAC automatically tries to use the Service Message Block(SMB) protocol[33] to communicate. When you successfully connect to the remote file server using this feature, the remote shared folder will automatically mount to your local `/Volumes/shared_folder`, and then you can access this folder as accessing local folder.



Because of this mount remote shared folder to local feature, we keep using the same codes from the previous local file read experiment, and only need to change the testing files directory and our captured performance data csv files. Therefore, we are still using `fcntl()` and `F_NOCACHE` flag to disable file caching effect. And we still use 4KB to 64MB difference sizes files to measure the read performance over network. Since we are holding our tests in home wifi environment, although there are only very few devices using the network, it is still possible that we can't keep continuously high network performance for a long time when doing our experiment. So we decide to change our original testing times for one experiment from 100 to 10, and do the experiment for 3 times with about 5-10 minutes pause between each experiment. And finally we will use the lowest standard deviation as our final results.

## Prediction

The most difference of this experiment compared to the previous one is of course the network penalty from accessing files over the network. From the previous local read experiment, we have average block read time in sequential access from 0.04361 to 0.10798 ms and in random access from 0.10761 to 0.13452. And from the previous network experiment, our result shows that bandwidth can reach 2.153319MB/s, which means 1.814ms/block. Since we are using SMB protocol for sharing files, which is an application-layer network protocol, there must be some processes in the server, like serializing and encrypting file data to transportable network data; and meanwhile in the client, there must be some processes like deserializing and decrypting network data to memory. We tried to look up relevant performance of SMB in MAC OS, but

couldn't find anything useful. So we have to give a conservative assumption, this data process in SMB protocol might take about 1ms. Based on this assumption, the overall network penalty should be about  $1.814\text{ms} + 1\text{ms} = 2.814\text{ms/block}$ . Learned from the previous result, we can predict remote sequential average block read time could be 2.85 to 3ms; and remote random average block read time could be about 3ms.

## Results

First, we use 4KB to 64MB files to read in our remote sequential read experiment:

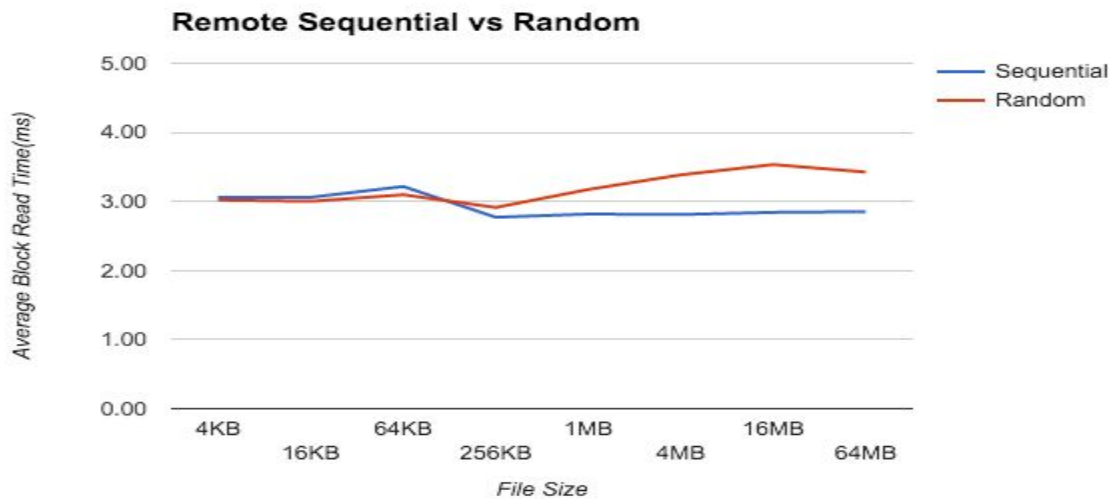
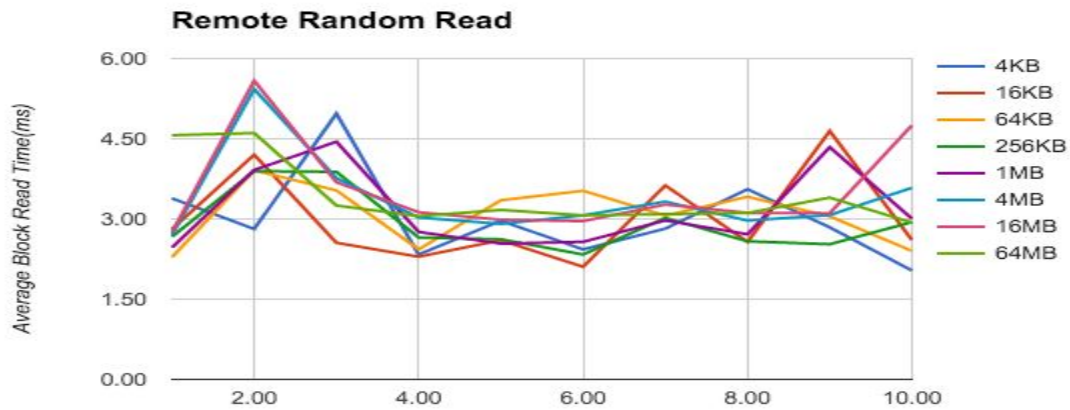
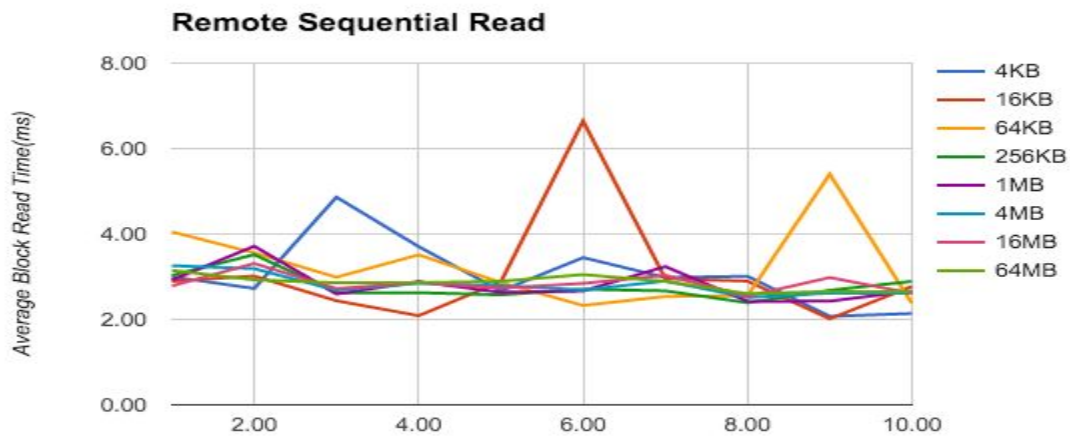
File Size	4KB	16KB	64KB	256KB	1MB	4MB	16MB	64MB
Average(ms)	3.06	3.06	3.22	2.77	2.82	2.81	2.84	2.85
Standard Derivation	0.81	1.31	0.96	0.31	0.40	0.25	0.22	0.18
Min	2.07	2.01	2.33	2.39	2.42	2.52	2.54	2.60
Max	4.86	6.65	5.40	3.52	3.71	3.26	3.31	3.15
Local Average(ms)	0.10798	0.10277	0.07442	0.05449	0.04572	0.04770	0.04361	0.04385
Penalty	2.95272	2.95711	3.14192	2.71705	2.77147	2.76260	2.79861	2.80875

And then we also make a random access in our experiment:

File Size	4KB	16KB	64KB	256KB	1MB	4MB	16MB	64MB
Average(ms)	3.02	3.00	3.10	2.91	3.17	3.38	3.53	3.43
Standard Derivation	0.83	0.85	0.56	0.55	0.76	0.79	0.92	0.63
Min	2.04	2.11	2.28	2.34	2.47	2.70	2.75	2.93
Max	4.97	4.64	3.91	3.90	4.45	5.43	5.58	4.61
Local Average(ms)	0.11262	0.10761	0.11757	0.13430	0.12091	0.11976	0.13452	0.12634
Penalty	2.90341	2.89408	2.97943	2.77852	3.05207	3.26506	3.39935	3.29871

## Analysis

Following graphs are the distribution of those 10 times average block read time we gained from different sizes of files and the distribution of different file sizes' average block read time in sequential access and random access.



- 1) There is no doubt the average block read time of sequential read and random read in remote environment is much larger than the one in local environment. Therefore, we use the local average time to compare with the remote average time to gain network penalty. We notice the distribution of the average block read time in sequential and random read becomes different from each other when the file size grows. In sequential read experiment, when the file size equals to or is bigger than 256KB, the average read time become very stable and the standard deviation is also very trivial. Besides, all the average time of large files is less than the average time of small files(<256KB). On the contrary, in random read experiment, all the average time of large files is more than the average time of small files. Meanwhile, the standard deviation of big files in random pattern is almost of a factor of 3 to the corresponding sequential read. Based on this results, we are thinking there might be caching mechanism within this remote file sharing protocol, which we believe all those most commonly used protocols have some cache mechanism to improve cross network performance. The reasons why we think caching mechanism can lead to the difference between remote sequential and random read are:
  - A) Prefetching a big sequential data and then direct returning these data without synchronously accessing disk can reduce the average time of sequential read and insure average time of reading big files can be very stable.
  - B) Every cache mechanism should have a limit cache size, so if the random read can't hit the cache in the protocol, it will trigger a prefetching action. In this case, it will introduce more overhead, like flushing old cached data out, accessing disk to gain more than requested size data and caching the data into its cache space. That may be the reason why in random read pattern, the average block read time will increase when the file size is bigger than 256KB. Besides, these time-consuming actions could also cause the standard deviation grows.
- 2) The average block read time of small file(<256KB) in both patterns is really close to each other. The main reason is still because of the possible cache mechanism within this file sharing protocol. Since this file size is probably less than the cache size, so it might be need one prefetching action to load a whole file data into the cache space. After that, no matter sequential or random read, this cache mechanism can directly return the corresponding block data from its cache spaces. However, since the small file has less block numbers, the overhead of prefetching action can't be amortized very well. Therefore, the average block read time of small files in these two patterns is larger than the one of big files in sequential read pattern.

## 5.4 Contention

### Methodology

In this section, we are going to report the average time to read one file system block of data similar to previous experiments but while there are other processes simultaneously performing the same operation on different files on the same disk. To measure contention, we continue using what we coded for previous experiments to capture the average block read time in sequential and random access situation, which means, once again, we still use ``fcntl()`` and ``F_NOCACHE`` flag to disable file caching effect.

Since in this section we focus on the influence of competing read-disk processes on our main process, which we used to capture the read time, we don't need to operate this experiment on different sizes files. Instead we use ``dd`` command to randomly generate 16 1MB files as our testing files and observe the impact of different number of competing processes on the main process's read time. Therefore, in our experiment, we will first use ``fork()`` function to create target number of new processes and when these new processes start to work, they will start to open their corresponding files and read from the start to the end for 100 times. After forking these new processes, the main process will use the same approach from previous experiments to open and read its own files and capture the time of reading a block of data for 50 times and then store the average block read time to our data csv files. After the main process finishes collect data, it will wait for its child processes to finishing reading and then continue the next round of the whole experiment with increased number of competing processes.

In this section, we will hold the experiments in three different read patterns: 100% sequential read(main process and competing processes are all sequential read); 100% random read(main process and competing processes are all random read); sequential mixed read(main process is sequential read and 50% competing processes are sequential read and the other half are random read); and random mixed read(main process is random read and 50% competing processes are sequential read and the other half are random).

### Prediction

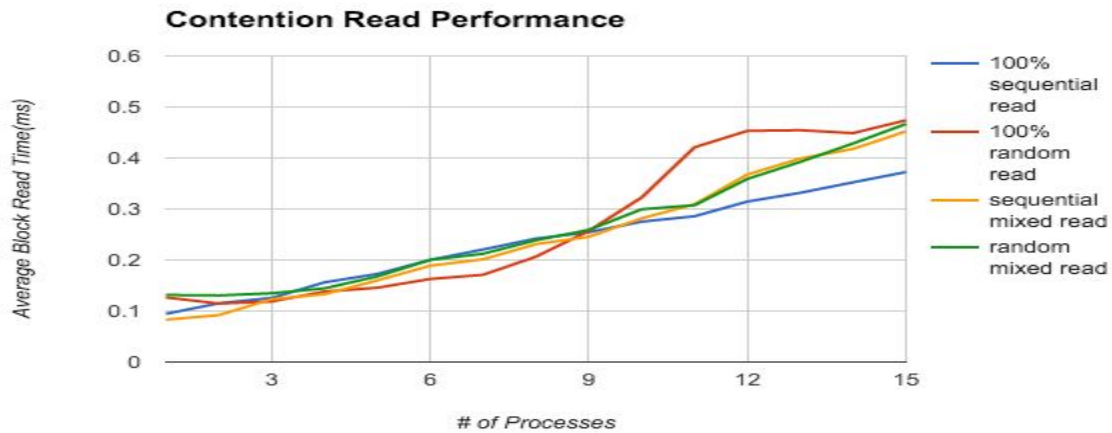
Since direct disk operation is limited resource, we believe as the number of competing processes grows, the read performance of main process could decrease. Besides of the waiting time for fetching data from SSD, the read throughput of main process could also be affected by other processes competing the fixed disk bandwidth. Theoretically, if two processes are simultaneously reading from the same disk, each one will possibly end up with using only half of the whole bandwidth separately. Therefore, we predict the read time could increase in a linear coefficient of 2. Comparing these 4 different patterns: 100% sequential read, 100% random

read, main process is sequential and others are 50%-50%, and main process is random and others are 50%-50%, we predict the average read time will be 100% sequential read < Sequential mixed read < Random mixed read < 100% random read. We think in the condition of the same number of competing processes, the contention penalty may be very similar to each other and then the difference between sequential and random read will be discovered.

## Results

The following are 4 different patterns average block read time(ms) table and distribution graph in the condition of different number of competing processes.

# of Competing Processes	100% sequential read(ms)	Standard Deviation	100% random read(ms)	Standard Deviation	Sequential mixed read(ms)	Standard Deviation	Random mixed read(ms)	Standard Deviation
1	0.09443	0.03153	0.1266	0.01905	0.08301	0.02997	0.13148	0.0129
2	0.11512	0.03685	0.11464	0.00516	0.0919	0.02221	0.13055	0.00936
3	0.12556	0.01356	0.11873	0.00449	0.12316	0.04769	0.13475	0.0045
4	0.15618	0.01178	0.13806	0.00976	0.13294	0.01135	0.14437	0.00768
5	0.17264	0.01478	0.14563	0.00779	0.16003	0.00929	0.16779	0.01777
6	0.2002	0.01751	0.1628	0.01656	0.18876	0.05185	0.20024	0.06036
7	0.22059	0.02279	0.17096	0.01455	0.20128	0.0238	0.21224	0.03733
8	0.24195	0.04088	0.20646	0.02839	0.23109	0.02713	0.23871	0.03481
9	0.25441	0.03502	0.257	0.06721	0.24545	0.03971	0.2591	0.03808
10	0.2752	0.04521	0.3226	0.10353	0.28167	0.07699	0.29944	0.06401
11	0.28575	0.04965	0.42092	0.18394	0.30915	0.07481	0.30739	0.07287
12	0.3146	0.08226	0.45316	0.18918	0.36763	0.13895	0.35903	0.12389
13	0.3317	0.07835	0.45448	0.18718	0.3988	0.16441	0.39235	0.13517
14	0.35229	0.1012	0.44873	0.14784	0.41778	0.16295	0.42852	0.1616
15	0.37244	0.09651	0.47392	0.15027	0.45221	0.19481	0.46682	0.1991



## Analysis

- 1) As we expected, the average block read time increases when the number of processes increases. However, the linear coefficient is not exactly 2. Instead from the table and the distribution graph, the linear coefficient is closer to 1.5 rather than to 2 and is not constant. There may be two main reason. First, the data we gained is only from the main process rather making every process collect data and sum up to calculate the average. The reason why we don't do that is it can hardly make sure there are always specified number of processes running when doing the experiment. However, our approach can at least insure when the main process is collecting data, there are always required number of competing processes also working. In this case, there is a chance for main process finish its task before any competing process, making the time waiting for system resume main process from others shorter. The second reason is possibly some kind of optimization is implemented within SSD in order to provide better performance when simultaneous operations occur, like SSD can happen to prefetch data for multiple processes. The caching invalidation we mention in the Methodology session is to avoid cache data to memory and read from memory, there should still be some optimized design in SSD to improve performance which is not easy to turn it off. Therefore, the read time doesn't increase as fast as we expected.
- 2) Comparing the small number of competing processes(<5) with the large number of competing processes(>10), we find the later one has much bigger standard deviation. With more and more competing processes working, the execution time of read a block, which includes the time of waiting for the system resuming itself, vary more obviously. The main process could possibly wait for any number from 1 to 15 of competing process finished and then continue its task. So the standard deviation increases when the number of processes grows.



- 3) Learned from the distribution graph, average read time of these four different patterns doesn't simply follow the rule: 100% sequential read < Sequential mixed read < Random mixed read < 100% random read. In fact, when the number of competing processes is less than 9, the 100% sequential read uses the most average time although the difference is very trivial. One of the reason is that the advantage of sequential read to avoid seeking is gone when there are multiple competing processes. Due to the competition, sequential read need time to wait for SSD relocate disk offset to next block offset after other competing processes finishing their operations. In this case, the sequential read becomes randomly access, as the other three patterns, in the disk and fixed offset may take more time to relocation than random offset. However, the random read is supposed to read block randomly, so this change have less influence on its results than the sequential read. Moreover, when the number of competing processes becomes very big(>9), the overhead of switching among different processes and waiting for other processes finished becomes more and more vital. There is a chance that after the random read process relocate the file offset, the system schedules other processes to run and pause this process, and after other processes finished, the previous read process needs to wait for system relocation to its desired offset. However, with the growth of number of competing process, process switching happens more frequently making the first time seeking and relocation become a waste of time as an extra overhead. So as the number of competing processes increases, the total useless first time relocation becomes the main reason causing those three patterns related to random read have worse performance than the sequential read pattern.

## 6. Summary

Category	Operation	Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time
CPU	Reading Time Overhead	NA	30-40 cycles	30-40 cycles	33.74809 cycles
	Loop Overhead	6-8 cycles/loop	33.75 cycles	$33.75 + (6-8) * N$ cycles	7.05 cycles/loop
	Procedure Call Overhead	<2 cycles with one more parameter	33.75 cycles	$33.75 + 1.5 * N$ cycles	1 cycles with one more parameter
	System Call Overhead	NA	33.75 cycles `getpid()` = 5 `getppid()` = 300 [unit: cycles]	`getpid()` = 38.75 `getppid()` = 333.75 [unit: cycles]	`getpid()` = 38.2 `getppid()` = 395.81 [unit: cycles]

	Process Creation Overhead	NA	33.75 cycles 500,000 to 1,000,000 cycles	500,000 to 1,000,000 cycles	832389.22 cycles
	Kernel Thread Creation Overhead	NA	33.75 cycles 50,000 to 100,000 cycles	50,000 to 100,000 cycles	54,524.97 cycles
	Process Context Switch Overhead	NA	400,000 cycles	400,000 cycles	387,233 cycles
	Kernel Thread Context Switch Overhead	NA	6000 cycles	6000 cycles	6196 cycles
Memory	RAM Access Time	10ns	Ns level overhead	More than 10 ns	48.5 ns
	RAM Bandwidth	25.6 GB/s [36]	Guess about -5 GB	R-20GB/s W-20GB/s	R-22.21GB/s W-22.14GB/s
	Page Fault Service Time	Transfer 4KB page = 25089 cycles	About 3*context switch = 3 * 387233 cycles	3 * 387233 + 25089 = 1186788 cycles	1,216,082.81cycles 6.678125ns/byte
Network	Round Trip Time Loopback (L) Remote (R)	(ping) L < 0.122ms R - depends	NEGLIGIBLE	L <1 ms R- depends	L-0.049641ms R-1.144046ms
	Peak Bandwidth	100MB	NEGLIGIBLE	L-100 MB/s R- depends	L-445.8 MB/s R- 8 MB/s
	Connection Setup	NA	NEGLIGIBLE	L <1ms R >=RTT	L-0.108107 ms R-2.614392 ms
	Connection teardown	NA	NEGLIGIBLE	L=R<1ms	L-0.012127 ms R-0.022124 ms
File System	Size of File Cache	RAM: 2*4G	Rdtsc: 33.75 cycles = 12.48ns	file system cache size is around 4GB	file cache is around 4G
	File Read Time	Sequential = 0.046ms/block Random = 0.146ms/block	Rdtsc: 33.75 cycles = 12.48ns	Sequential = 0.046ms/block Random = 0.146ms/block	Sequential = 0.043ms/block Random = 0.134ms/block [varied from file size]

	Remote File Read	Network bandwidth = 1.814ms/block Sequential = 0.046ms/block Random = 0.146ms/block	1 ms/block from SMB protocol	Sequential read = 2.85 to 3ms/block; Random read = about 3ms/block.	Sequential read = 2.85 ms/block; Random read = 3.43 ms/block [varied from file size]
	Contention			read time could increase in a linear coefficient of 2	the linear coefficient is closer to 1.5 rather than to 2

## Reference

[1]Performance measurements with RDTSC

[http://www.strchr.com/performance\\_measurements\\_with\\_rdtsc](http://www.strchr.com/performance_measurements_with_rdtsc)

[2]Using the RDTSC Instruction for Performance Monitoring, Intel pentium II processor,

<https://www.ccsf.carleton.ca/~jamuir/rdtscpm1.pdf>

[3]X86\_instruction\_listings

[https://en.wikipedia.org/wiki/X86\\_instruction\\_listings](https://en.wikipedia.org/wiki/X86_instruction_listings)

[4] Intel® 64 and IA-32 Architectures Software Developer's Manual, intel,

<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>

[5] How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set

Architectures,Gabriele Paoloni,

<http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf>

[6] c on linux: IPC Socket Pairs(quick full-duplex pipes)

<http://chattrawits.blogspot.com/2015/01/c-on-linux-ipc-socket-pairsquick-full.html>

[7] getpid()

<http://man7.org/linux/man-pages/man2/getpid.2.html>

[8] fork()

<https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/fork.2.html>

[9] copy-on-write mechanism

<http://unix.stackexchange.com/questions/58145/how-does-copy-on-write-in-fork-handle-multiple-forks>

[10] Imbench: Portable tools for performance analysis. Larry McVoy Carl Staelin

[11] Gallery of Processor Cache Effects. Igor Ostrovsky.

<http://igoro.com/archive/gallery-of-processor-cache-effects/>

[12] More Linked Lists

<http://ironbark.xtelco.net.au/subjects/IOO/lectures/lecture19.html>

[13] loop unroll, wiki entry.[https://en.wikipedia.org/wiki/Loop\\_unrolling](https://en.wikipedia.org/wiki/Loop_unrolling)

- [14] Across platform monotonic timer.  
<http://codearcana.com/posts/2013/05/15/a-cross-platform-monotonic-timer.html>
- [15] cpu info x86,  
[http://osxbook.com/book/bonus/misc/cpuinfo\\_x86/cpuinfo\\_x86.c](http://osxbook.com/book/bonus/misc/cpuinfo_x86/cpuinfo_x86.c)
- [16] Achieving maximum bandwidth,  
<http://codearcana.com/posts/2013/05/18/achieving-maximum-memory-bandwidth.html>
- [17] memory bandwidth demo,  
<https://github.com/awreece/memory-bandwidth-demo>
- [18] gcc optimization <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>
- [19] Page fault [https://en.wikipedia.org/wiki/Page\\_fault](https://en.wikipedia.org/wiki/Page_fault)
- [20] mmap  
<https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/mmap.2.html>  
<http://stackoverflow.com/questions/12383900/does-mmap-really-copy-data-to-the-memory>
- [21] Intel VS ARM  
<http://www.androidauthority.com/arm-vs-x86-key-differences-explained-568718/>
- [22] Intel advanced Vector Extensions  
[https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro\\_to\\_Intel\\_AVX.pdf](https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf)
- [23] Intel Optimization Reference Manual  
<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>
- [24] *C standard library:string.h:memset* [http://clc-wiki.net/wiki/C\\_standard\\_library:string.h:memset](http://clc-wiki.net/wiki/C_standard_library:string.h:memset)
- [25] The problem with prefetch <https://lwn.net/Articles/444336/>
- [26] round-trip time wiki entry. [https://en.wikipedia.org/wiki/Round-trip\\_delay\\_time](https://en.wikipedia.org/wiki/Round-trip_delay_time)
- [27] RTT <https://blog.packet-foo.com/2014/07/determining-tcp-initial-round-trip-time/>
- [28] ping, wiki entry. [https://en.wikipedia.org/wiki/Ping\\_\(networking\\_utility\)](https://en.wikipedia.org/wiki/Ping_(networking_utility))
- [29] TCP package, TCP and UDP, COS 461,princeton transport protocol  
<http://blog.apnic.net/2014/12/15/ip-mtu-and-tcp-mss-mismatch-an-evil-for-network-performance>
- [30] MTU,  
[http://www.tcpipguide.com/free/t\\_IPDatagramSizetheMaximumTransmissionUnitMTUandFrag.htm](http://www.tcpipguide.com/free/t_IPDatagramSizetheMaximumTransmissionUnitMTUandFrag.htm)
- [31] Computer Networks A Systems Approach", Third Ed.,Peterson and Davie,  
Morgan Kaufmann, 2003.
- [32] How does one do Raw IO on Mac OS X? (ie. equivalent to Linux's O\_DIRECT flag)  
<http://stackoverflow.com/questions/2299402/how-does-one-do-raw-io-on-mac-os-x-ie-equivalent-to-linuxs-o-direct-flag>
- [33] Server Message Block & SAMBA <https://www.samba.org/samba/>  
[https://en.wikipedia.org/wiki/Server\\_Message\\_Block](https://en.wikipedia.org/wiki/Server_Message_Block)
- [34] lseek -- reposition read/write file offset  
<https://developer.apple.com/library/prerelease/mac/documentation/Darwin/Reference/ManPages/man2/lseek.2.html>
- [35] How to connect with File Sharing on you mac  
<https://support.apple.com/en-us/HT204445>
- [36] Intel datasheet

[http://ark.intel.com/products/64891/Intel-Core-i7-3720QM-Processor-6M-Cache-up-to-3\\_60-GHz](http://ark.intel.com/products/64891/Intel-Core-i7-3720QM-Processor-6M-Cache-up-to-3_60-GHz)