System Measurement Project

1. Introduction

Operating system is crucial for managing hardware resources and provide basic services to user programs. Its design involves many considerations such as efficiency, reliability, security, etc.. To better understand the mechanisms and performance characteristics of an operating system, we set up a series of experiments to gather data and do analysis on them. We choose to study OS X El Capitan 10.11.4, which is a Unix-like operating system.

To set up the experiments, we use C++ and Python as our programming languages. C++ is “closer to the hardware” and there are many libraries for us to make system calls at a more predictable and relatively lower cost. Python provides good support for network socket programming, which is handful in network measurements. We use CLion and PyCharm as our main development environments. Our laptops all have Intel Core i5(I5-5257U) CPU, which has 1 processor with 2 cores. In order to get more precise result, we switch the system to run on only one core in all following experiments. In this way, we are able to eliminate the scheduling and communication between the 2 cores.

For each test, we run with different settings or repeat a certain number of times. We perform statistical calculations and do qualitative analysis afterward. In this way we can observe the performance pattern and gain more confidence in our results.

We have two people on the team. Both of us are involved in coding, testing, analysis, and report writing. It takes us both about 70 hours to complete this project.

This report contains full description of our experiments with performance of CPU, memory, network, and file system. Section 2 lists the detail hardware specifications of the machine we test with. Section 3 presents the overheads of CPU, Scheduling, and OS Services. Section 4 shows memory performance, including RAM access time, RAM bandwidth, page fault service time. Section 5 discusses on network round trip time, peak bandwidth, and connection overhead. Section 6 shows our analysis on size of file cache, file read time, remote file read time, contention. Section 7 is a comprehensive overview of our results.

2. Machine Description

The target machine is MacBook Pro (Retina, 13-inch, Early 2015).

Hardware	**Specification**

CPU Model	Intel Core i5 (I5-5257U), 2.7GHz
Architecture	64-bit
Processor	1 Processor (2 Cores)
Cycle Time	1 / 2.7 GHz = 0.37ns
L1 Instruction Cache (per Core)	64KB
L1 Data Cache (per Core)	64KB
L2 Cache (per Core)	256 KB
L3 Cache	3MB shared
Memory Bus	64-bit-wide, 5 GT/s (DMI2)
I/O Bus	Thunderbolt Bus (10 GB/s)
Memory	2 * 4GB of 1866MHz LPDDR3 onboard memory
Disk Model	APPLE SSD SM0256G
Disk Capacity	251 GB (251,000,193,024 bytes)
Disk Controller Speed	5.0 GT/s
Disk Read Bandwidth	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 10.11.4

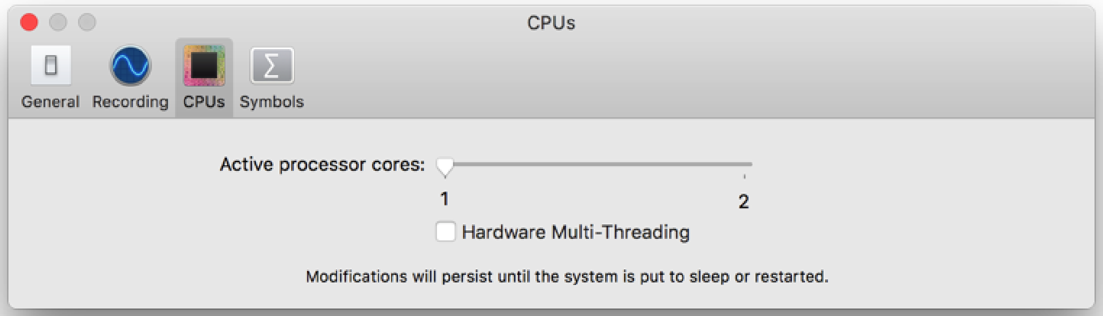
3. CPU, Scheduling, and OS Services

3.1 Time stamping method

Cycle number. In this project, we use Time Stamp Counter (TSC) to count the cycles of C++ instruction executions, which is more fine-grained and accurate compared to measure the program CPU time. The TSC getting the CPU cycle information in a high-resolution, low-overhead way.

Single thread. The TSC falls short in providing accurate cycle results in multi-processor, multi-core, hyper-threaded CPUs. Consequently, in the project we will

- reduce the number of core running to 1
- disable hardware multi-threading



Active processor cores: 1 (range 1–2)

☐ Hardware Multi-Threading

Modifications will persist until the system is put to sleep or restarted.

Implementation[1]. The code of measuring the starting cycle count and ending cycle count is:

[1] "Difference between rdtscp, rdtsc : memory and cpuid ... - Stack Overflow." 2012. 7 Jun. 2016 <http://stackoverflow.com/questions/12631856/difference-between-rdtscp-rdtsc-memory-and-cpuid-rdtsc>

```
static inline uint64_t rdtscStart(void){
    uint32_t cycles_high, cycles_low;
    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", "memory"
    );
    return (((uint64_t) cycles_high << 32)| cycles_low);
}


static inline uint64_t rdtscEnd(void){
    uint32_t cycles_high, cycles_low;
    asm volatile (
            "RDTSCP\n\t" /*read the clock*/
            "mov %%edx, %0\n\t"
            "mov %%eax, %1\n\t"
            "CPUID\n\t"
            : "=r" (cycles_high), "=r"(cycles_low):: "%rax", "%rbx", "%rcx", "%rdx", "memory"
    );
    return (((uint64_t)cycles_high << 32) | cycles_low);
}
```

Instruction reordering. Both the compiler and the processor will reorder the instructions for performance purpose. The out of order execution of instructions will make the measurement inaccurate by measuring something that is not supposed and not measuring something that is supposed. To solve this problem, we impose compiler barrier and processor barrier to ensure the sequential execution of instructions.

Compiler barrier. The `volatile` and `memory` in the `asm` statement act as the compiler barrier to hinder the compiler from reordering instructions.

Processor barrier. The `RDTSCP` and `CPUID` act as the processor barrier to hinder the processor from reordering the instructions.

3.2 Measurement Overhead

3.2.1 Read Overhead

Methodology

We use `RDTSC` to do time stamping throughout our experiments. Reading the code and executing it will incur performance overhead. So at the very first step we want to measure this overhead. To do this, we simply do two time stamping: one call to get start time and one call to get end time. Since in following experiments we will get both start and end times, we measure them together here. Taking the difference of end time and start time, we get the this measurement overhead.

We run 10 **tests** and each test takes 3,000 **repetitions**.

Prediction
In `rdtscStart()` and `rdtscEnd()`, we use embedded assembly language code. Each function consists of 5 instructions, and at the end of each function there is a OR and << operations to get an unsigned 64-bit integer of current counter. Considering these instructions may take more than 1 CPU cycles and C++ inline function call may add slightly more cost. We estimate the overall time to be 20 to 100 CPU cycles.

Result

Test #	1	2	3	4	5	6	7	8	9	10
average CPU cycles	35.992	32.406	35.988	35.994	32.326	31.356	31.351	34.831	34.795	31.398
mean	**33.6437**									
stddev	**1.95117**									

Analysis
The result has relatively low standard deviation, thus it is safe to say that the read overhead falls within 30-40 cycles, which matches our expectation.

3.2.2 Loop Overhead

Methodology
Since loop is quite basic program structure and widely used, it is worthwhile to estimate the overhead. Considering the loop time may have an effect on the result (since more loop times may amortize the cost of loop initialization), we test the overhead with 1,4,16,64,256,1024 loop iterations with basic for loop. We put time stamping code before and after the loop, while we do nothing inside the loop. In the end, we subtract the read overhead and calculate the average overhead of one iteration, which we think is more comparable.

We run 10 tests and each test takes 3,000 repetitions.

Prediction
In each iteration of a for loop, we think CPU will take following steps: first, it loads the variables from memory and check whether the loop condition holds; second, it executes the code within the loop body (in this test case there is no code); last, it modifies the variable

according to the given function and starts the next iteration. In this experiment the condition is a simple variable range check and each iteration increments the variable by 1. Besides the iteration cost, there is a overhead setting up the variables and the read overhead tested in 3.2.1. In terms of the overhead of each iteration, we estimate it to be about 6-10 cycles.

Result

Loop times	**1**	**4**	**16**	**64**	**256**	**1024**
Test 1	0.01	5.13633	7.60967	6.29	6.886	6.37133
Test 2	0.005	4.744	7.612	8.81733	6.00233	6.72933
Test 3	0.014	4.703	7.61033	6.12367	6.01867	6.30733
Test 4	0.012	4.87067	7.57167	6.09267	6.117	6.53767
Test 5	0.017	4.67567	7.56767	6.12433	6.058	6.389
Test 6	0.008	4.65167	7.61033	6.12367	6.021	6.19833
Test 7	0.009	4.68167	7.58467	6.09467	6.02833	6.26033
Test 8	0.01	4.86733	7.576	6.14067	6.017	6.01133
Test 9	0.005	4.86433	7.64833	6.43033	6.10933	7.49133
Test 10	0.009	4.676	7.56967	6.11567	6.814	6.74767
mean	**0.0099**	**4.787067**	**7.596034**	**6.435301**	**6.207166**	**6.504365**
stdev	**0.0037252**	**0.150736**	**0.02626395**	**0.8438356116**	**0.34145231**	**0.4151492**

Analysis
From the result we can see that when the loop time is 1, it barely takes any time. We guess that the compiler may does some optimization here to eliminate the loop. When the loop time grows bigger than 4, we can see that it takes more cycles to do each iteration, and the time seems to converge about a bit more than 6 cycles. Thus we conclude that the loop overhead is 6-7 cycles each iteration, which matches our prediction

3.3 Procedure Call Overhead

Methodology
To measure the overhead of procedure call, we created 8 procedures with 0-7 parameters. Each parameter is of 4-byte int type. Since parameters are passed during the call, we want to observe the effect on overhead with different number of parameters. All procedures do nothing but return immediately. In each tests, we time stamp before and after the procedure.

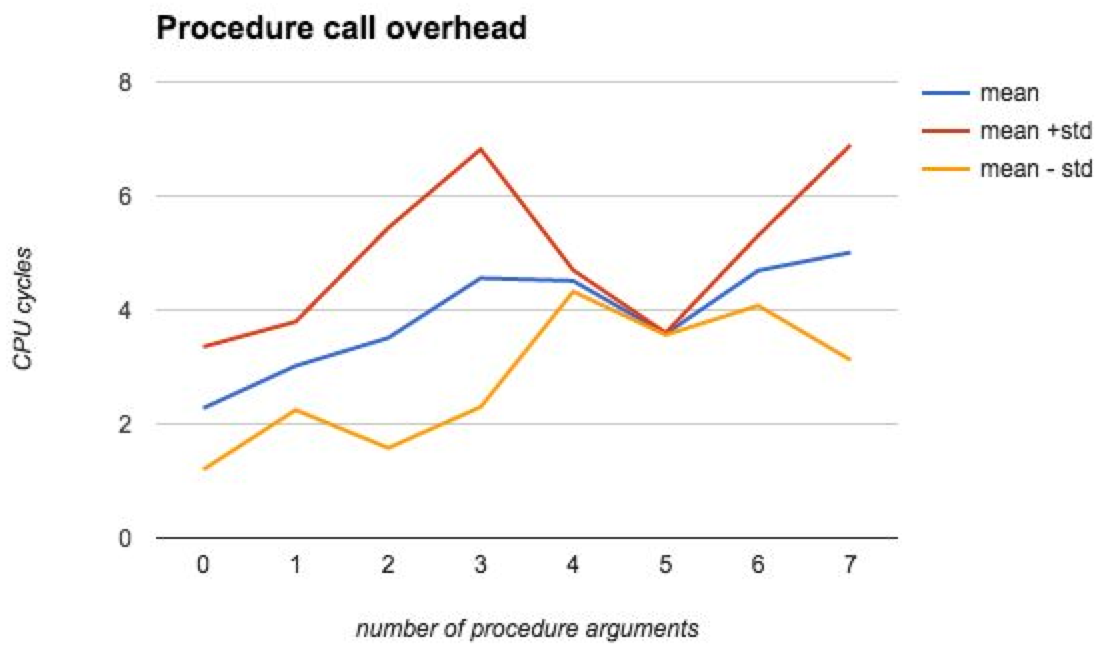
We run 10 tests and each test takes 3,000 repetitions.

Prediction
Doing a procedure call, the operating system will do followings:
1. Parameters and return address get pushed onto the stack.
2. CPU finds and jumps to the entry point of the callee.
3. Callee gets the parameters and the return address from the stack.
4. Callee executes and then return to caller.

In these steps, more parameters will add more overhead. Since the whole process involves many operations on CPU and the stack, we estimate it to cause about 4-10 cycles overhead with procedure that takes no argument, excluding the read overhead. And each argument adding 2-4 cycles.

Result

# arguments	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Test 1	2.874	2.843	3.602	4.866	4.61	3.549	4.769	5.325
Test 2	5.2	3.833	1.835	3.59	4.302	3.585	4.543	7.505
Test 3	1.824	4.863	1.845	3.566	4.706	3.588	4.758	0
Test 4	1.835	2.836	0.963	3.566	4.684	3.584	3.927	5.31
Test 5	1.844	2.666	5.5	3.581	4.428	3.618	3.987	5.315
Test 6	1.834	2.827	3.6	3.56	4.709	3.584	4.551	5.322
Test 7	1.84	2.725	1.84	4.375	4.711	3.562	4.554	5.315
Test 8	1.835	2.709	3.515	10.867	4.294	3.583	6.125	5.35
Test 9	1.845	2.823	6.573	3.622	4.325	3.574	4.582	5.335
Test 10	1.834	2.078	5.82	3.995	4.341	3.583	5.119	5.315
mean	**2.2765**	**3.0203**	**3.5093**	**4.5588**	**4.511**	**3.581**	**4.6915**	**5.0092**
stdev	**1.077747265**	**0.7742585055**	**1.930708336**	**2.260299331**	**0.1880053191**	**0.01793197021**	**0.6147787768**	**1.888914668**



Analysis
Subtracting the read overhead, it takes about 2 cycles to call a procedure with no argument, which is small. From the graph we can see the trend that the cost increases with more parameters. Despite the fluctuation in the graph, the increasing rate is lower than our expectation. We think that the OS pushes and fetches the parameters in bulk instead of one by one, thus amortize the overhead.

3.4 System Call Overhead

Methodology
A system call requires a control transfer from user space to the kernel space and an amplification of privileges (operating system changes to kernel mode). We searched in the Linux Manual and decided to experiment with getpid() and getppid(). The reason is that they are system calls of similar category (only fetching information about the processes), but getpid() gets cached by OS and getppid() is not. We think it will be interesting to learn about the difference between cached and uncached system call. To measure the time, we time stamp before and after each system call.

We run 10 tests and each test takes 3,000 repetitions.

Prediction

Since system call requires control transfer and mode switch by operating system, we think the overhead should be two of three magnitudes larger than procedure code overhead (hundreds to thousands cycles). However, if the system call result gets cached, then the more we call it, the less amortized overhead. In the ideal situation, it takes only 1-2 cycles to fetch the cached result.

Result

Test #	1	2	3	4	5	6	7	8	9	10
cached	42.82	34.825	34.837	35.34	34.865	34.776	34.87	34.862	34.836	34.856
mean	**35.6887**									
stdev	**2.38189**									
Un-cached	382.41	372.793	414.417	372.816	428.905	378.271	373.785	443.028	372.862	372.778
mean	**391.207**									
stdev	**25.5871**									

Analysis
From the result we can see that the difference between cached and uncached system call is drastic. The uncached system call will take up to more than 300 cycles. For cached system call, it only takes about 2 cycles extra on top of the read overhead to complete. This result matches our expectation.

3.5 Task Creation Time

3.5.1 Process Creation Overhead

Methodology
We use fork() to create a new process and measure the overhead. The child process will do nothing but returns immediately. We use the pid returned by fork() to distinguish the parent and child process. The parent process will wait for the child process to finish. We time stamp the start time before fork() and end time right after the fork().

Execution sequence. There is complicacy in measuring: the order of running is not guaranteed. Following two situations may happen:

- After the fork(), parent process continues to run. In this case, the creation time is correctly measured.
- After the fork(), the child process starts to run. Then the parent process continues to run (with the child process return or not). In this case, the measurement contains two context switch overhead.

We have tried several fixes.

- The memory-based mutex would not work here since the memory space are not shared between parent process and child process using fork().
- The file-based semaphore would not work here since the File System I/O is a significant overhead (in magnitude of order compared to process context switch overhead), whose large variations will make the measurement of the process context switch time in interest inaccurate.

Thus we choose to adopt the simple experiment and get a ballpark of process creation time. Considering the process creation is time consuming and the process context switch should cost lower overhead by magnitude, we believe this method is acceptable.

Considered above fixes and simplicity of programming, we opt for a simple fix - we only record the cycle times when `start < end` while excluding the other case in benchmarking.

```
diff = end - start;
if (diff > 0) {
    return diff;
}
else {
    return -1;
}
```

We run 10 tests and each test takes 3,000 repetitions.

Prediction

Doing a `fork()` takes many operations. It creates a copy of the parent process and the child process will have its own space to run. There is a lot of space allocation and resource copying during the `fork()`. Therefore, we estimate the process overhead to be much more time-consuming than system calls. The overhead should be at 10,000 to 1,000,000 cycles.

Result

Test #	1	2	3	4	5	6	7	8	9	10
Avg Cycles	1.48E+06	869737	624032	478021	1.17E+06	568741	377210	315869	1.18E+06	239327
mean	730359									
stdev	402325									

Analysis

The average overhead of process creation is 7.3 * 10^5 cycles, which matches our expectation. It’s worth noting that the standard deviation of the result is quite big. We think the possible reasons are:

- The different ordering of execution between the parent and child processes add different overhead of context switches, as discussed above.
- When the child process finishes, the cleaning up process may kick in before the parent process does the time stamp. This will add overhead to the final result.
- Creating new process and cleaning up terminated process may compete system services, hence increase the time to finish the experiment.

Two major learnings from this experiment are :

- Process creation takes about 10^6 cycles, which is very time consuming.
- The scheduling of process creation, allocation, and termination are quite intricate.

3.5.2 Kernel Thread Creation Overhead

Methodology

We use `pthread_create()` call to make new kernel thread. The new thread does nothing but exit immediately. The parent thread will wait for the child thread to finish by performing a `pthread_join()`. We time stamp before and right after the `pthread_create()` call. Similarly, we have the complicacy of execution order but tolerate the difference here. The goal is to get a ballpark figure of the kernel thread creation overhead.

We run 10 tests and each test takes 3,000 repetitions.

Prediction

Kernel thread are much more lightweight than processes (threads will share resources), thus the creation time should be considerably less. We estimate it to be one order of magnitude smaller than process creation overhead, which is about 10,000 to 100,000 cycles to spawn a new thread.

Result

Test #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Cycles	28601.9	43163	68266.3	54000.6	43376.7	43354.5	44648	43287.5	47386.8	46963.1
mean	**46304.8**									
stdev	**9463.25**									

Analysis

As we expected, creating a new thread takes much less time, about 4.6 * 10^4 cycles (while process creation about 7.3 *10^5 cycles). Also the standard deviation is less proportionally. This reinforce the fact that multi-threading is more economic than multi-processing. Moreover, inter-process communication will cost even more overhead, since processes are largely independent from one another.

3.6 Context Switch Time

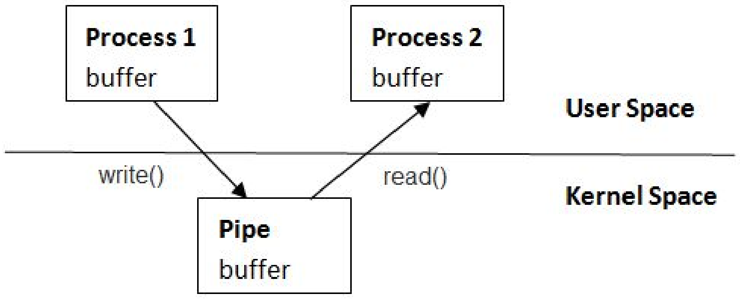
3.6.1 Process Context Switch Overhead

Methodology

Process Fork. We use `fork()` to fork a child process from the parent process. The process is similar to the methodology in Process Creation Overhead. We record the start time at the parent process and do a context switch, and then the

Communication - Pipe.[2] Since the memory space are not shared between parent process and child process using `fork()`, we need pipe to message passing between the two.

1. The parent process records `start` time at the beginning of the execution.
2. The parent process wait for child process thus forcing a context switch.
3. The child process records the `end` time at the beginning of the execution.
4. The child process writes the `end` to the write-end of the pipe
5. The child process exits.
6. The parent process reads the `end` from the read-end of the pipe
7. Consequently, `start` and `end` are recorded.



Pipe. Notice here, the pipe write and read overhead is naturally excluded in this measurements by the above steps.

Prediction

[2] "C++ Tutorial: Multi-Threaded Programming - C++ Class ... - Bogotobogo." 2013. 7 Jun. 2016 <http://www.bogotobogo.com/cplusplus/multithreading_ipc.php>

Heavy-weight process context switch requires complete memory state of the parent process saving and child process restoring. We know the process creation time, and process context switch should be a simpler workflow than process creation but the time consumed should be the same order of process creation. Therefore, we estimate the process context switch time should be half the time of creating a process. The process context switch time is around cycles 365,000 cycles.

Result

Test #	1	2	3	4	5	6	7	8	9	10
Cycles	562962	475115	425574	396518	405184	408729	403470	401686	398288	386652
mean	**426418**									
stdev	**51171.3**									

Analysis
The cycles of Process Context Switch is aligned with the prediction - the process context switch time is around half the time of creating a process.

3.6.2 Kernel Thread Context Switch Overhead

Methodology
Thread creation. We use `pthread_create()`call to make new kernel thread. After creation the new child thread, the main thread records the `start` time. The child thread executes a function `_target`, record the `end` time, and return back to main thread.

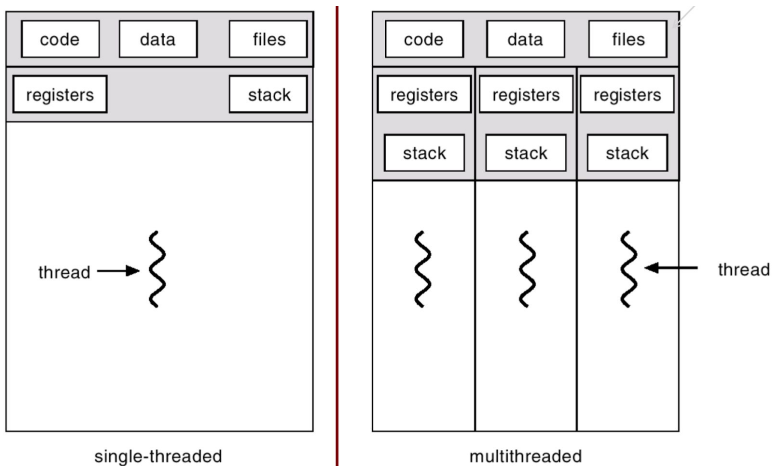
Execution sequence - mutex. Normally, the execution sequence of two threads are not determined. Therefore we use mutex lock to ensure that the main thread is executed first and child thread is executed second. Consequently, `start` and `end` are recorded sequentially.

1. The main thread acquires a `lock`.
2. The main thread creates a child thread.
3. The main thread records the `start` time.
4. The main thread release the `lock`.
5. The main thread calls `phtread_join` to wait for the new thread complete.
6. The child thread acquires the `lock`.
7. The child thread records the `end` time.
8. The child thread return the `end` time to main thread.

Consequently, we ensure the main thread is executed first; however the thread context switch time also includes the lock release and lock acquisition time, which will be measured later.

Prediction

Threads are lightweight processes thus the thread context switch time is significantly less than process context switch time; because, threads share memory spaces for code, data and files; switching threads only needs to records lightweight processor state like execution stack and register.



The cost of thread context switch time would be significantly lower compared to process context switch.

We know the kernel thread creation time, and kernel thread context switch should be a simpler workflow than kernel thread creation but the time consumed should be the same order of process creation. Therefore, we estimate the kernel thread context switch time should be half the time of creating a kernel thread. The process context switch time is around cycles 23,000 cycles.

Result

Test #	1	2	3	4	5	6	7	8	9	10
Cycles	6563.01	21436	4762.09	5182.7	5839.1	6139.17	7044.99	5825.23	6891.35	7158.96
mean	**7684.26**									
stdev	**4644.64**									

* The result includes lock release and lock acquisition time.

Analysis
The cycles of Thread Context Switch is aligned with the prediction, which is significantly lower compared to process context switch.

Moreover, the kernel thread context switch time is significantly lower than half the time of creating a kernel thread. It actually is around 1/7 of the kernel thread creation time. Therefore, our estimation of the relationship between kernel thread creation and context switch is extremely conservative.

4 Memory

4.1 New Time stamping method

In memory related experiments, we use `std::chrono::steady_clock`[3] to get interval time. `Std::chrono::steady_clock` is newly added class in C++ 11 and provides a simple and precise way to measure monotonic time interval. The `now()` method is cross-platform. The time can be easily casted into seconds, milliseconds, microseconds, and nanoseconds.

In the following experiments, some result are measured in unit of seconds or milliseconds. We find it more straightforward to measure directly in these units rather than in CPU cycles, which require further computation.

4.2 RAM access time

4.2.1 Methodology

Back-to-back-load latency. Back-to-back-load latency is defined as “the time that each load takes, assuming that the instructions before and after are also cache-missing loads”[4]. Back-to-back load is the opposite of loads in a vacuum.

We use the following pseudocode as the back-to-back load as in *lmbench* paper:

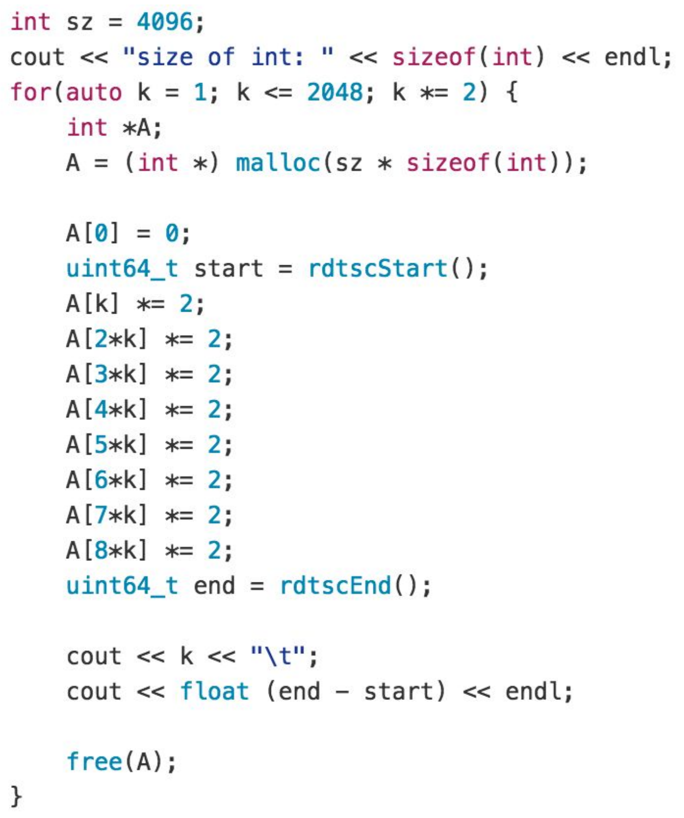
```
p = head;
while (p->p_next)
    p = p->p_next;
```

[3] "std::chrono::steady_clock - cppreference.com." 2011. 3 May. 2016 <http://en.cppreference.com/w/cpp/chrono/steady_clock>
[4] McVoy, Larry W, and Carl Staelin. "lmbench: Portable Tools for Performance Analysis." *USENIX annual technical conference* 22 Jan. 1996: 279-294.

Then benchmark has two variables - array size and stride size for each p_next. As subtly suggested by *lmbench* paper, we create an array and store the p_next pointer as the content of each array element. Then, the array is walked as:

```
mov   r4,(r4)   # C code: p = *p;
```

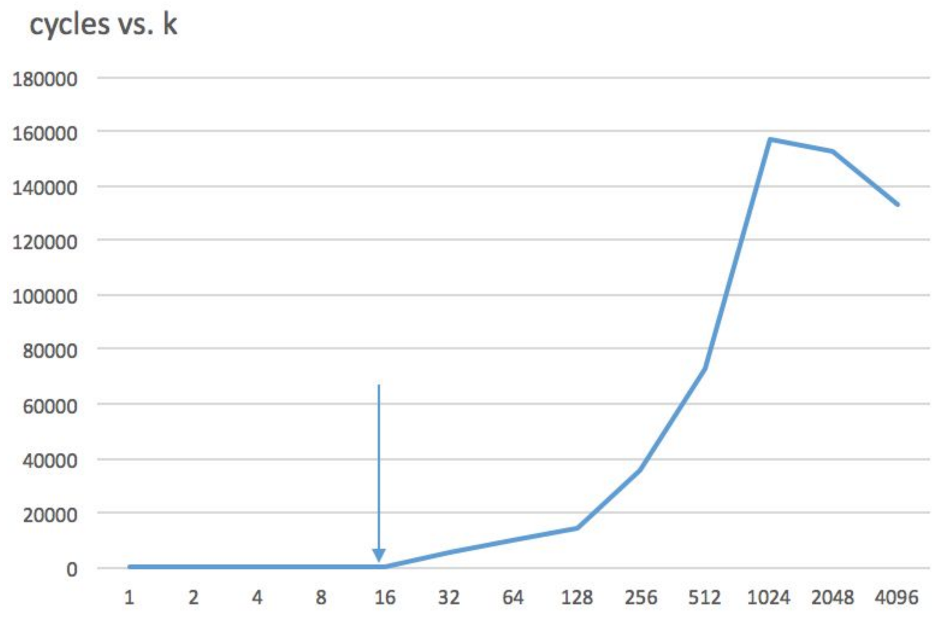
Cache-missing loads. The back-to-back load requires the memory access is cache-missing loads. Consider the opposite case of cache-missing loads, when CPU loads an array element, it not only loads that element, but also **pre-fetches** and caches neighboring elements. Consider the following example:



```
int sz = 4096;
cout << "size of int: " << sizeof(int) << endl;
for(auto k = 1; k <= 2048; k *= 2) {
    int *A;
    A = (int *) malloc(sz * sizeof(int));

    A[0] = 0;
    uint64_t start = rdtscStart();
    A[k] *= 2;
    A[2*k] *= 2;
    A[3*k] *= 2;
    A[4*k] *= 2;
    A[5*k] *= 2;
    A[6*k] *= 2;
    A[7*k] *= 2;
    A[8*k] *= 2;
    uint64_t end = rdtscEnd();

    cout << k << "\t";
    cout << float (end - start) << endl;

    free(A);
}
```

The above code measure the access time for every *k* element in a large array, for 8 accesses. *k* is the stride size.

We plot the access time in number of cycles against *k*, and we get the following graph:



We can see that, before *k,* there is plateau, and after *k* there is an increase in access time. The best explanation that when the machine sequentially access the memory space of the array, it will pre-fetch the elements after the element being accessed.

How large is the pre-fetch size? There is the formal term called **cache line** for this pre-fetch size. From the graph we can see it is *k=4*, the best guess would be 16*4 bytes = 64 bytes, since the size of *int* is 4 bytes. 64 bytes is consistent with the with the fact that the cache line size is 64 bytes in L1, L2, L3 cache.

Avoid prefetches. The g++ compiler optimizes the array access by prefetching the next a few array elements when we accessing the array with a fix stride size. We want to measure the raw memory access time without compiler optimization. To alleviate prefetching, each time we accessing the array, the pointer moves not exactly the stride size, but some deviations. This idea is implemented as followed:

```
if (random) {
    index = (i/k+1)*k+rand()%k;
}
else {
    index = (i/k+1)*k;
}

index %= sz;
A[i] = (int *) &A[index];
```

Construct array to model linked list. For each element of array, the content stores the address of the next target element, so the array looks more like a linked list. And avoid overflow the access, we do also circular linked list.

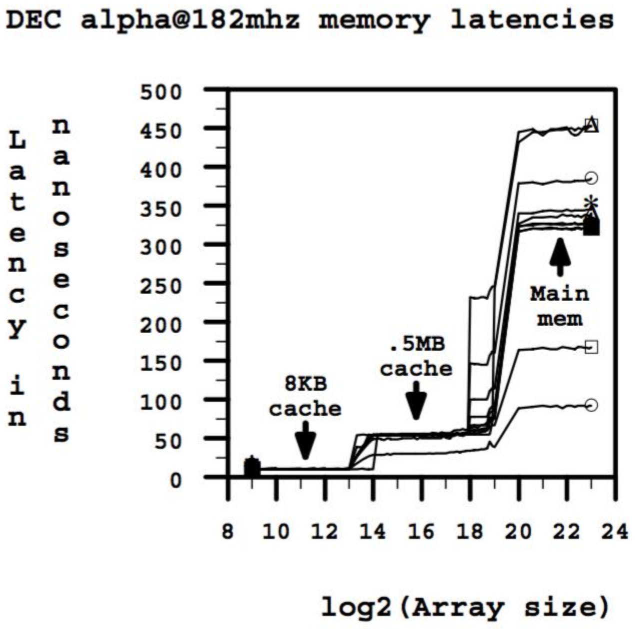
```
A[i] = (int *) &A[index];
```

Reduce overhead of for loop. If we do the for loop in the measurement, there is loop and branch prediction overhead that we do not want to include in the measurement. Thus, to reduce it, we need, we add `-funroll-loops` argument in g++ compiler to unroll the for loop which in turn reduces the loop and branch prediction overhead.

Avoid compiler removing the statements. If we just do *p = (int **) *p* in the loop without accessing p later, the compiler will detect it and remove the loop since the loop essentially does nothing. To avoid such optimization, we assign p to an another variable and the function will return such variable.

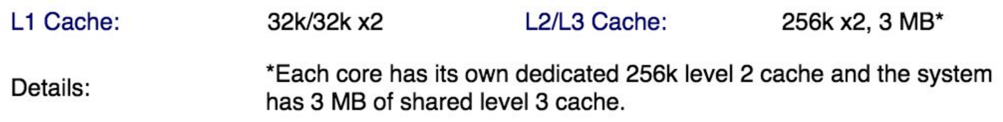
4.2.2. Prediction

The result of memory access time from previous works (*lmbench*) is shown as followed,



From the above figure, there are two caches in DEC alpha, 8KB and 0.5MB. The latency ranges from 10ns to 50ns to 325ns. There are 3 plateaus in the graph .

The machine measured in this project has the following cache information[5]:



L1 Cache:	32k/32k x2	L2/L3 Cache:	256k x2, 3 MB*
Details:	*Each core has its own dedicated 256k level 2 cache and the system has 3 MB of shared level 3 cache.		

The cache line size of each cache is 64Bytes.

Therefore, since there are 3 levels of cache, we predict the memory access time will form 4 plateaus in the graphs. The entire memory hierarchy is measured, including L1, L2, L3 cache access latency and main memory access latency.

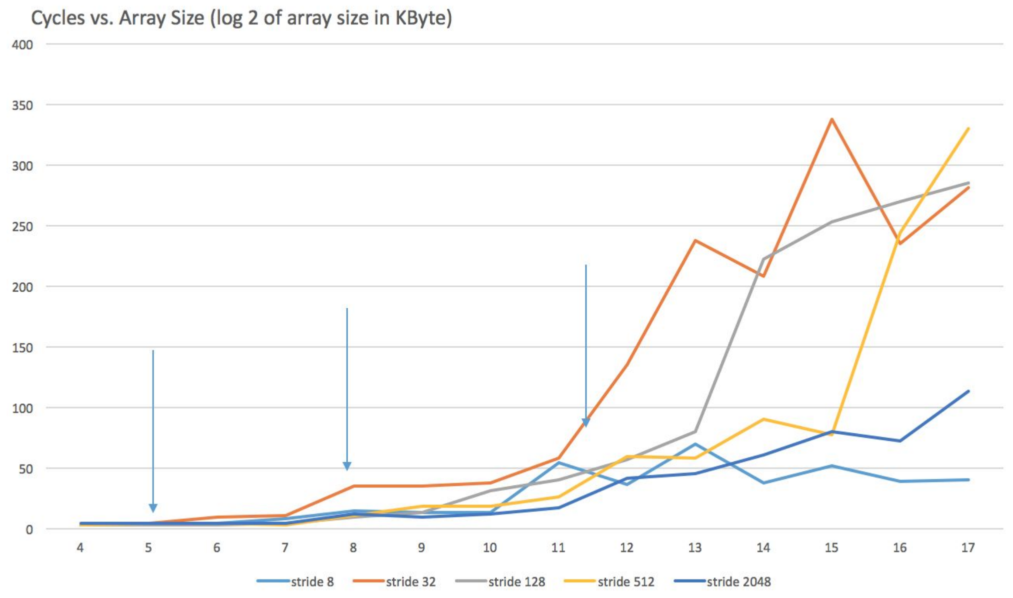
To rough estimate the result, we take a look at the Intel Haskell i7 CPU L1, L2, L3, Memory latency[6]:

- L1 Data Cache Latency = 4 cycles for simple access via pointer
- L1 Data Cache Latency = 5 cycles for access with complex address calculation (size_t n, *p; n = p[n]).
- L2 Cache Latency = 12 cycles
- L3 Cache Latency = 36 cycles
- RAM Latency = 36 cycles + 57 ns

Therefore, we predict the results on our target Core i5 CPU should be:

- L1: 4 cycles
- L2: 15 cycles
- L3: 40 cycles
- Memory: 200 cycles

4.2.3 Result



[5] "MacBook Pro "Core i5" 2.7 13" Early 2015 Specs (Retina ..." 2015. 3 May. 2016 <http://www.everymac.com/systems/apple/macbook_pro/specs/macbook-pro-core-i5-2.7-13-early-2015-retina-display-specs.html>

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

4.2.4 Analysis

Changing points. We have L1 32KB, L2 256KB, L3 3MB, thus there should be changes around the points the cache level changes.

We use log2 of array size in KB, thus we can see the such changes around 5, 8, and 11.585. We have annotated the change point on the graph. 5, 8 and 11.585 are **theoretical** changing points, we can see the **actual** changing points around 5, 8 and 11, which match the theoretical ones closely.

Variances. We can see some variances of different stride sizes increase as the array size increases. The variance in L1, L2, L3 caches are smaller. There is the largest variance in memory access, possibly due to other contributing factors like TLB misses and page fault.

Results.
In the best effort of estimation, average access time:

- L1 access time: 3.8 cycles (1.41 ns)
- L2 access time: 12 cycles (4.44 ns)
- L3 access time: 45 cycles (17 ns)
- Memory access time: 200 cycles (74.07ns)

The results align with our estimation.

4.3 RAM bandwidth

4.3.1 Methodology

According to *lmbench* paper, the read and write RAM bandwidth can be measured in a loop that loads and stores aligned 8-byte words. Also, we found a great article online[7] that describes how to achieve maximum memory bandwidth. To get a better understanding, we implemented several methods to measure RAM bandwidth:

- Read:
    - Basic loop
    - Manually unrolled loop
- Write:
    - Basic loop
    - Manually unrolled loop
    - `memset()`
    - Repeated string instructions

With each method, we create a 1GB array and perform reads/writes through the whole array. We turn off compiler optimization to eliminate its effects. Since we are measuring the

[7] "Achieving maximum memory bandwidth - Code Arcana." 2013. 3 May. 2016 <http://codearcana.com/posts/2013/05/18/achieving-maximum-memory-bandwidth.html>

bandwidth in unit of GB/s, we use `std::chrono::steady_clock` to get duration time in seconds. We repeats each method 20 times and record the result. Since our experiments only yield bandwidth lower than the expected value (27.8GB/s, analysis see prediction part below), we will take the **maximum value** as result. Followings are methods description:

Basic loop / unrolled loop read
In the basic loop, we iterate through the 1GB array, read data, and add the data to a result variable. In our tests, we found that looping creates a huge overhead on the result (we will discuss further in the analysis part) so that we make the array of type `__int128_t` (size of 16 bytes) to reduce the time of iterations.

Unrolled loop is similar to basic loop except that in each iteration we make 8 reads instead of 1, thus further reduce the overhead of loop.

Basic loop / unrolled loop write
Similar to read, we just assign a constant number to each element of the array.

`memset()` **write**
Another way to write the entire array is using `memset()` call. We simply need to specify the memory block pointer, value, and the number of bytes we want to write, without doing a loop. One thing to notice is that `memset()` sets value to memory in bytes, however, this doesn't seem cause much overhead in the result.

Repeated string write
We use `rep stosq` instruction to repeatedly store a word into the array, as another way to write the entire memory block. The assembly code is:

```
asm("cld\n" "rep stosq" : : "D" (data), "c" (SIZE / 8), "a" (0) );
```

4.3.2 Prediction
According to the target machine specs and definition of memory bandwidth:
Bandwidth = Base DRAM clock frequency * memory bus width * number of interfaces
= 1866Mhz * 64 bits * 2
= **27.8 GB/s**

And according to the LPDDR3 wiki[8], a 800Mhz LPDDR3 provides 12.8 GB/s bandwidth, thus a 1866Mhz LPDDR3 provides about a doubled bandwidth, which verified our calculation.

4.3.3 Result

[8] "Mobile DDR - Wikipedia, the free encyclopedia." 2011. 3 May. 2016 <https://en.wikipedia.org/wiki/Mobile_DDR>

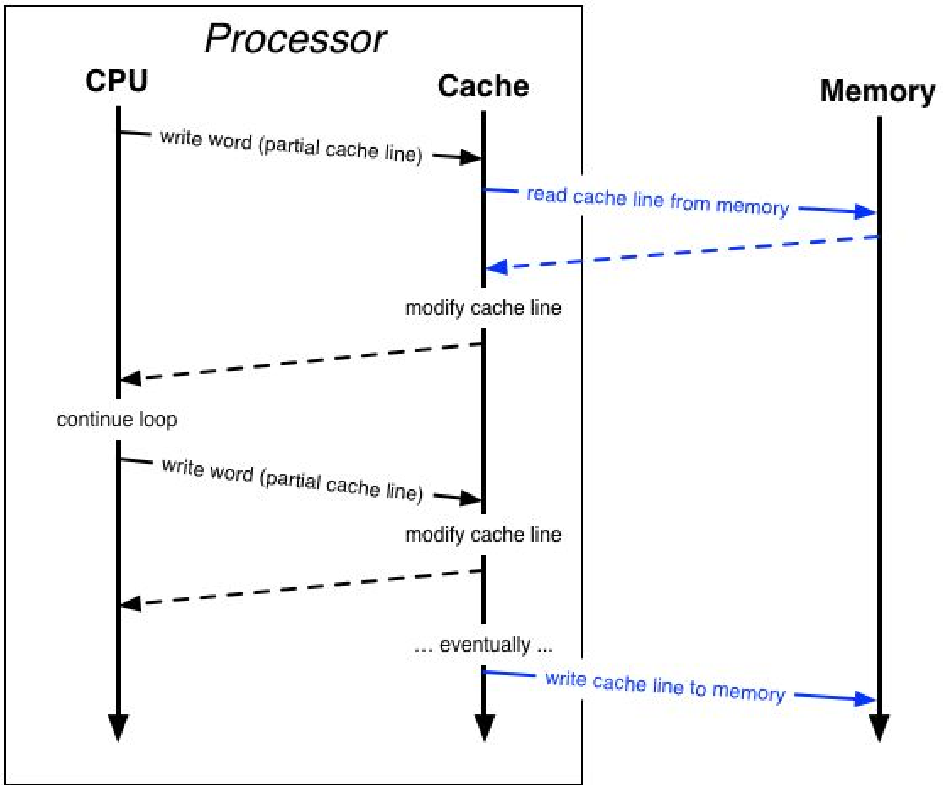
Memory Read (GB/s)	**loop**	**loop unrolled**	**—**	**—**
	5.75466	10.0543	—	—
Memory Write (GB/s)	**loop**	**loop unrolled**	**memset**	**repeated string**
	4.44876	5.96615	13.9932	14.1537

4.3.4 Analysis

The result of loop read is only 5.75 GB/s, which is much lower than the expected 27.8GB/s. With manually unrolled loop read, the result gets improved significantly and gives about 10GB/s. In our experiments, we found that the loop tolls heavy on the result. If we loop the array and read a byte in each iteration, the result is less than 1GB/s. We guess the cost is due to the **branch penalty**: the CPU guess which way a branch will go before this is known for sure and thus may go into wrong direction and waste a few CPU cycles. Using `__int128_t` data type and manual unrolling, we reduce the time that branch penalty happens thus get a more precise result.

Similarly, basic loop or unrolled loop write give poor bandwidth result and unrolled loop improves the result (not as much as read though). However, writing with memset() or repeated string operation give way better result: about 14GB/s. In both methods, there are not loops but built-in system operations, thus the overhead is greatly saved.

Still, branch penalty won't explain the big difference between our experiment result and expectation bandwidth. We believe one important reason is the cache read and write policy. In modern CPUs memory traffic on the bus is done in units of **cache line**, which is 64 bytes on our target machine. To write memory of size less than 64 bytes, the system will first read the entire cache line, modify it, and write back to memory. The process is shown in the picture below. Thus the write operation includes both memory read and write, lowering our bandwidth experiment result.


Processor
CPU
Cache
Memory
write word (partial cache line)
read cache line from memory
modify cache line
continue loop
write word (partial cache line)
modify cache line
... eventually ...
write cache line to memory

4.4 Page fault service time

4.4.1 Methodology

A page fault happens when a user program access a memory page that is mapped into virtual memory but not actually loaded into main memory. The processors's MMU will detects the page fault and traps to the system kernel to handle it. Then system kernel will try to load the page into main memory. There are three types of page fault, and in this experiment we consider the **major page fault**: the page is not loaded in memory at the time of the fault. 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 case will write the victim page back to disk, which is an relatively expensive operation due to disk latency.

In this experiment, we first create a 1GB text file with `ftruncate()` call, which fills the file with null bytes ('\0'). Then we use `mmap()` to creates a new mapping in the virtual address space of the calling process. `mmap()` only changes the kernel data structure but not loads any data from disk to main memory (the allocated region doesn't even point to any physical memory). Therefore, whenever we read something from the file there will be a major page fault.

We read through the file in different **page stride**, the number of memory pages we stride over in each iteration. Every iteration we only read one page. The page size on target machine is 4KB, and we experiment with page stride of size 1, 4, 16, 64, 256, 1024, 4096, 16384, 65536 (4KB - 256MB). We read the file data 100 times, or (1GB / page stride size) times if the page stride is big. Then we calculate the average time and standard deviation. We use `std::chrono::steady_clock` to get monotonic time measurement.

In each experiment with different page stride, we do a `mmap()` before measuring and `munmap()` after it. In addition, we execute a `system("purge")` to force clear the system cache in order to eliminate the its effect on measurement.

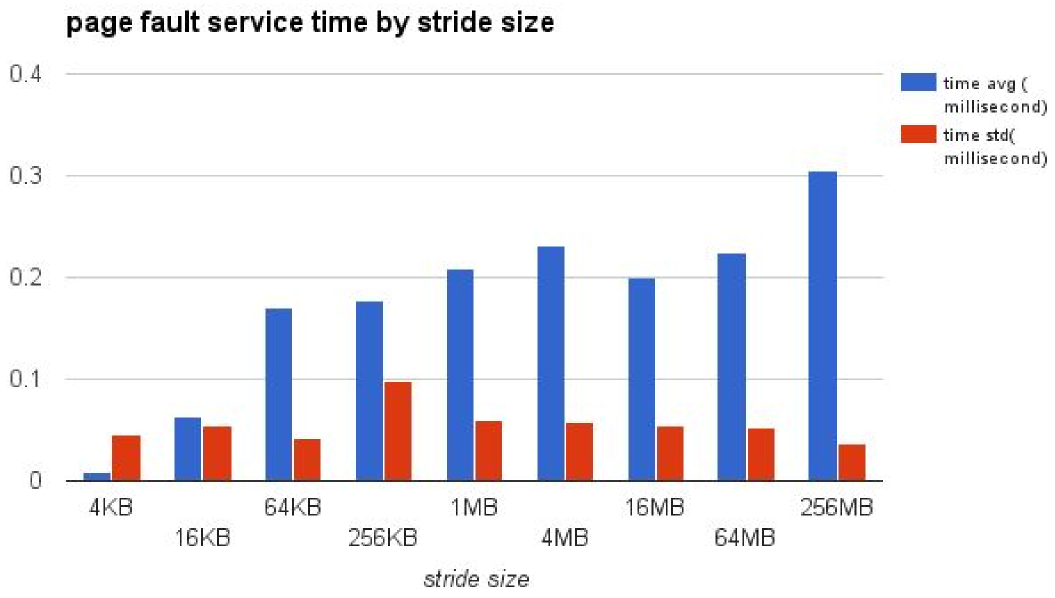
4.4.2 Prediction

Page fault service time includes the time of page fault handling in kernel (context switch, memory management, etc.), disk access time, and disk transfer time. On target machine, files are stored in SSD, and the access time of SSD is under 0.1ms[9]. According the target machine specs, the random uncached read speed is about 33MB/sec. Since a page size is 4KB, the disk transfer time is around 0.12ms. The page fault handling in kernel should add a bit overhead but not exceed the disk access and read time. Thus we estimate the page fault service time to be **0.2-0.4 ms**.

4.4.3 Result

page stride	**1**	**4**	**16**	**64**	**256**	**1024**	**4096**	**16384**	**65536**
stride size	**4KB**	**16KB**	**64KB**	**256KB**	**1MB**	**4MB**	**16MB**	**64MB**	**256MB**
time avg (millisecond)	0.0095247	0.06339743	0.17060542	0.1777686	0.20926224	0.23210015	0.1993291563	0.225086375	0.30536975
time std(millisecond)	0.04498315916	0.05479538348	0.04289761508	0.09817945446	0.05879828183	0.05793737199	0.05355489998	0.05304436315	0.03654250332

[9] "Solid-state drive - Wikipedia, the free encyclopedia." 2011. 3 May. 2016 <https://en.wikipedia.org/wiki/Solid-state_drive>



4.4.4 Analysis
When the stride size is small, especially 4KB, the page fault service time is significantly less than 0.1ms, which is even less than the SSD access time. Clearly the system will read more data than a single page from disk when a page fault happens. Thus many reads are from the main memory rather than disk.

When the stride size becomes larger than 1MB, we can see that the page fault service time becomes **0.2-0.3 ms**. With stride size equal to or more than 64MB, it is unlikely that the system will read more than one entire stride size of data into main memory when a page fault happens. Hence we think these results are more accurate. That means the result matches our prediction.

5 Networking

5.1 Methodology

Python. In this section we swap to Python programming language due to its elegancy in web programming. Unlike CPU measurement and memory measurement, the performance measurement of networking is network-bound rather than CPU-bound, thus the Python is sufficient although its run-time speed is slower than C++.

Time. We use `time.time()` to measure the time in seconds since the epoch as a floating point number. `time.time()` measures the wall time while the `time.clock()` measures a processor’s CPU time. We should measure the wall time rather than the processor’s CPU time, because the operations in the our measurements rarely consume a lot of CPU directly; instead they perform network calls.

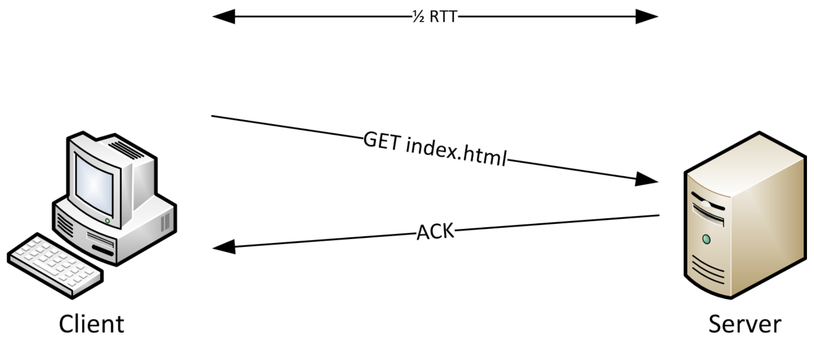
Client-Server. To get measurements on networking, we use client - server model and in each experiment we have separate client and server programs. For loopback interface, the server program runs on localhost (127.0.0.1). For remote interface, the server program runs on another machine, and the client connects to the server via known IP address.

The remote test machine has almost the same specs with test machine except that it has 16GB memory. All measurements are done using the UCSD-PROTECTED network.

5.1 Round Trip Time

5.1.1 Methodology

Round trip time 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. Round trip time is also known as the ping time.[10]



First we establish the connection between client and server. Then the client will send a single packet to the server. Upon receiving the packet, the server will send back a packet containing acknowledgement. We record the time that it takes the client to send and receive the data. We set the network packet size to 64 bytes, which equals to ICMP packet size.

Every client-server test runs 100 times and we compute min, max, average, and standard deviation from the result. Also we perform “ping” on both localhost and remote machine, each for 20 times, and collect the results.

5.1.2 Prediction

[10] "Round-trip delay time - Wikipedia, the free encyclopedia." 2011. 4 Jun. 2016 <https://en.wikipedia.org/wiki/Round-trip_delay_time>

For local loopback test, the packet will only go through the loopback device (a virtual network interface) and handled by the routing table. Since there is no physical network interface involved, the round trip time should be pretty small. The system “ping” command provides a benchmark to evaluate the round trip time. Thus the local loopback round trip time should be about the same with localhost ping time.

For remote interface, packet goes through the network card, routers in WIFI network, server’s network card, and all the way back. The round trip time should increase significantly, due to router lookup and packet forwarding. We estimate remote round trip time should be larger by one order of magnitude compared with local result and be the same with remote ping.

5.1.3 Result

Round Trip Time	local loopback (ms)	localhost ping (ms)	remote (ms)	remote ping (ms)
min	0.041	0.037	3.124	2.283
avg	0.061	0.065	5.056	69.109
max	0.142	0.116	21.059	318.158
std	0.009	0.026	3.863	68.868

5.1.4 Analysis

From the result we can see that the round trip time of local loopback and localhost ping are pretty close, with minimum about 0.04 ms and average about 0.06 ms. This matches our prediction.

The round trip of remote connection grows by an order of magnitude, up to a few milliseconds. Yet we see big difference between the result of remote TCP and remote ping. Interestingly, remote ping has smaller minimum but large variation. Generally speaking, the ping should be faster than remote TCP, because IMCP is at a lower level in the hierarchy, thus the minimum with ping is smaller. And we think the big variation with ping is due to the varying network traffic. For TCP, if ACK is not received, then it will resend the package. But ping keeps waiting until there is an echo. This result reveals the advantage of **error correction mechanism in TCP**, which help provide more stable and efficient data transfer.

5.2 Peak Bandwidth

5.2.1 Methodology

Bandwidth. We want to measure the peak bandwidth of the OS. We use socket to send a large data into buffer from the server to the client. In order to fully utilize the bandwidth, the

TCP will fill the maximal payload (MSS) in the datagram. Hence every datagram is stuffed with the MSS each time except for the last one.

Client-server Architecture. To measure peak bandwidth, we use server client architecture to perform data transfer operation. We transfer a large data from the server's memory to the client and measure the time required to complete the transfer.

We measure the peak bandwidth by sending different size of data to measure the peak bandwidth at different file size.

Timer Placement. The server will send the data to the client. We should put timer on the client. The timer starts when the first data received on the client and stops when the last data is finished transfer. We do not put timer on the server since otherwise extra time of RTT will be included which causes the measured bandwidth smaller.

Loopback. All 127.0.0.1 (localhost) traffic never hits the physical network, it gets processed by a loop back adapter in the kernel. Therefore, the peak bandwidth for loopback is more CPU-bound.

MSS. MSS is the largest amount of data, specified in bytes, that a computer or communications device can receive in a single TCP segment[11]. It is useful to estimate the peak bandwidth of the OS.

5.2.2. Prediction

The maximal bandwidth (peak bandwidth) can be roughly estimated as $throughput \leq \frac{MSS}{RTT}$[12]. In the OS being benchmarked, the MSS is default as 64 kb. In the following calculation, we just need to estimate an upper limit of the maximal bandwidth, thus, the overhead for handling data loss during the transfer is not factored in.

Local. For the local transfer, the minimal RTT is 0.041 ms. Thus the expected peak bandwidth is 1560.98 MB/s.

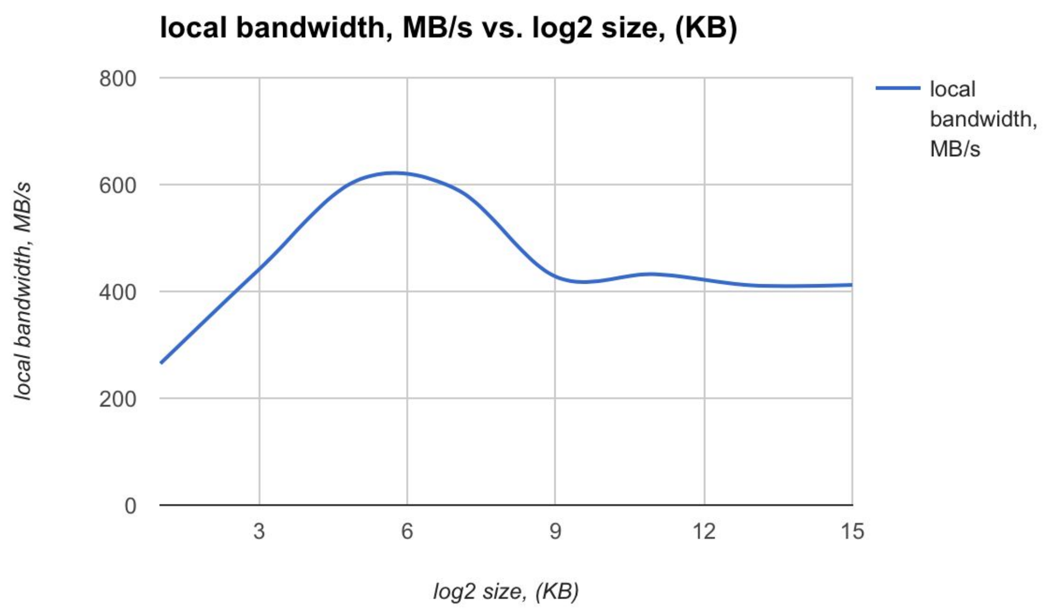
Remote. For the remote transfer, the minimal RTT is 3.12 ms. Thus the expected peak bandwidth is 20.51 MB/s.

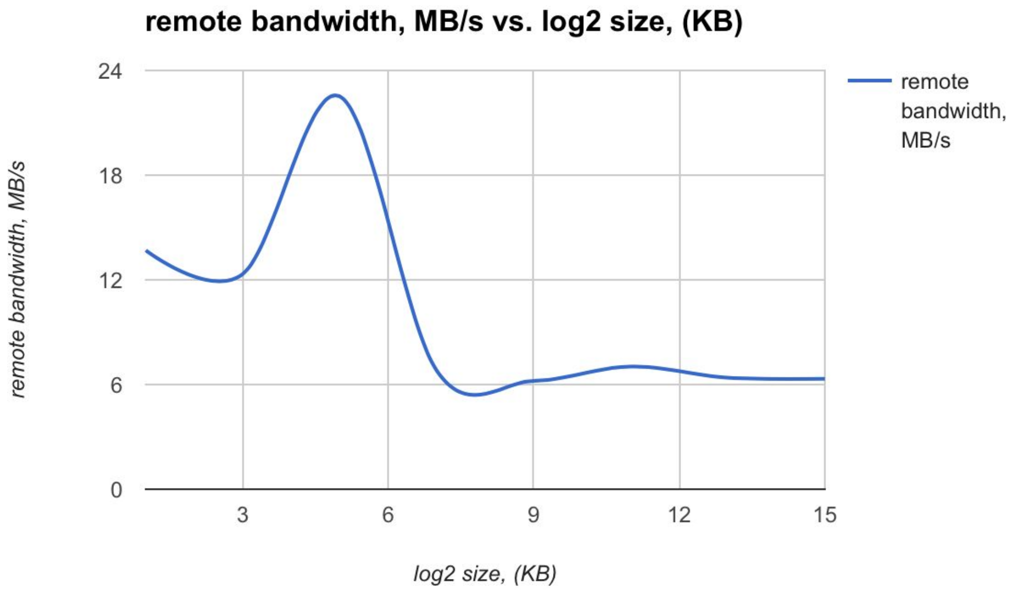
5.2.3 Result

Size, KB	log2 size, (KB)	local bandwidth, MB/s	remote bandwidth, MB/s
2	1	265	13.69

[11] https://en.wikipedia.org/wiki/Maximum_segment_size
[12] https://en.wikipedia.org/wiki/Measuring_network_throughput

8	3	441.48	12.34
32	5	608.47	22.51
128	7	590.67	6.79
512	9	427.67	6.21
2048	11	432.18	7.03
8192	13	411.16	6.39
32768	15	412.17	6.33



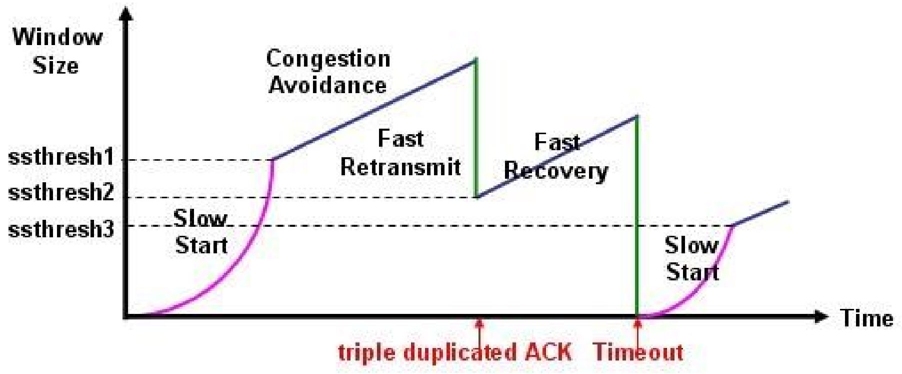


5.2.4 Analysis
Observations. We get the increasing in measured in bandwidth with respect to data size from 2KB to 32KB (2^5). However, we get a bandwidth plunge in the subsequent data size around 128KB (2^6) until a stable bandwidth around data size 512KB (2^9).

Results and Predictions Comparison. In terms of local TCP, the peak bandwidth is 608.47 MB/s, for remote, it is 22.51 MB/s. The remote bandwidth matches well with the predicted bandwidth (20.51 MB/s).

The local bandwidth (608.47 MB/) falls far below the predicted one (1560.98 MB/s). It may caused by the Python code runtime overhead of timer measurement in such fast transfer speed in kernel loopback. In such fast transfer speed, the overhead would not be negligible.

Explanations. The increase, plunge, and subsequently flat in bandwidth can be explained by the TCP protocol - TCP is slow start and TCP has congestion control.



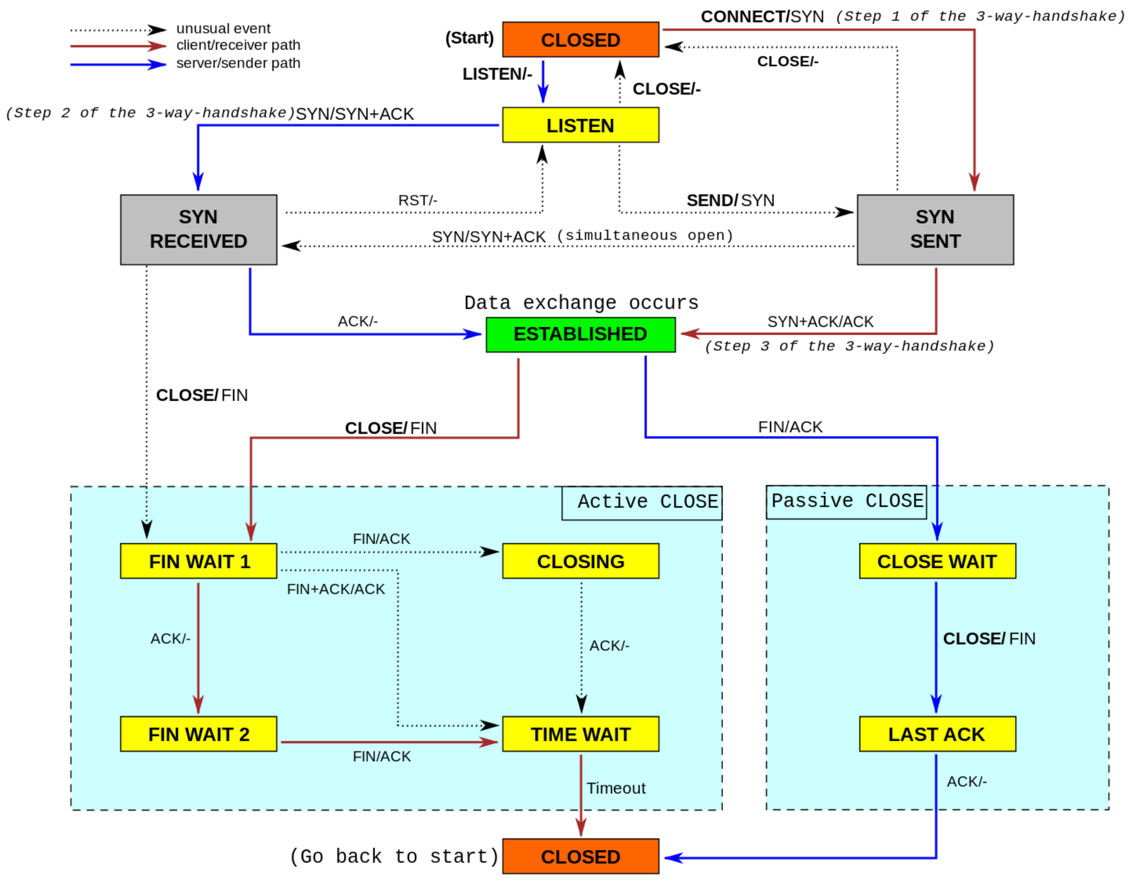
For smaller file sizes, the window size gradually increases in slow start phase, thus, there is an increasingly large measured bandwidth. From 128KB to 512KB, the plunge can be explained by TCP congestion avoidance and restart. For file size larger than 512KB, the TCP found a suitable stable window size after a few rounds to determine the network transmission capacity, therefore subsequently the measured bandwidth is stably flat.

5.3 Connection overhead: setup and teardown

5.3.1 Methodology

For connection setup and teardown, TCP requires handshakes as shown below[13]:

[13] "Transmission Control Protocol - Wikipedia, the free encyclopedia." 2011. 4 Jun. 2016 <https://en.wikipedia.org/wiki/Transmission_Control_Protocol>



We use the same experiment structure of part 5.1, except that this time we don't transfer any data and just simply record the time to make a socket `connect()` and `close()` call. We have the server program keep listening and two separate client program for setup and teardown. We conduct each test for 100 times.

5.3.2 Prediction

Connection setup. The setup process takes three handshakes and the round trip time records the time for two handshakes. Thus the setup time should be **1.5 times RTT** we measured in part 5.1, for both loopback and remote.

Connection teardown. To tear down a TCP connection, the client only needs to send a FIN packet and doesn't necessarily have to wait for the following packets. After sending the FIN packet, the client can destroy the socket. Since there is no involvement of server, the teardown time for loopback and remote should be about the same. And the time should be about 50% of loopback RTT (the time for one handshake).

5.3.3 Result

	loopback setup (ms)	remote setup (ms)	loopback teardown (ms)	remote teardown (ms)
min	0.0551	2.3389	0.0179	0.0260
avg	0.0973	5.9391	0.0309	0.0540
max	0.2670	159.9050	0.1481	0.1540
std	0.0367	18.7526	0.0202	0.0228

RTT for comparison

Round Trip Time	local loopback (ms)	localhost ping (ms)	remote (ms)	remote ping (ms)
min	0.041	0.037	3.124	2.283
avg	0.061	0.065	5.056	69.109
max	0.142	0.116	21.059	318.158
std	0.009	0.026	3.863	68.868

5.3.4 Analysis

Loopback setup. The minimum setup time is **1.34 times** the previous loopback minimum RTT, and the average setup time is **1.60 times** the previous loopback average RTT. This matches our prediction.

Remote setup. The minimum setup time is **75%** of the previous remote minimum RTT, and the average setup time is **1.18 times** the previous remote average RTT. This deviation from our prediction can be explained by the network traffic effect. In the result we can see the standard deviation of remote setup time is 18 ms, which indicates a pretty unstable network environment.

Loopback & remote teardown. We can see that both teardown time are pretty close, and the standard deviation are relatively small. And the tear down time is only **50-80%** of loopback RTT (in terms of minimum and average time) . Considering the test error, we would say our prediction is correct.

6 File System

6.1 Size of File Cache

6.1.1 Methodology

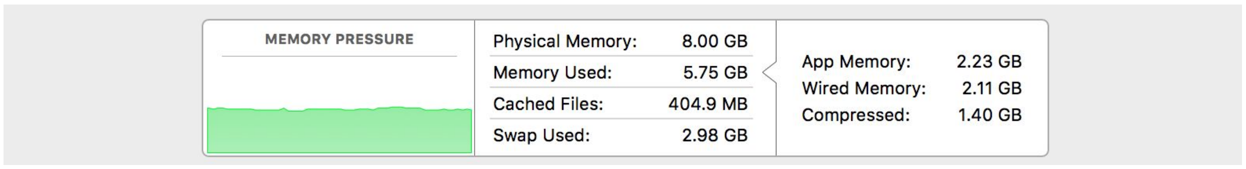
OS uses available memory as file cache for applications. When the file data is present in file cache, the read will be a lot faster than loading the data from disk. Based on this knowledge, we design experiments that sequentially read files of different size twice: the first read will put data in file cache and we record the time of second read as measurement. When the file fits into the file cache, the time of second read will be minimal; If the file doesn't fit into the file cache, then some read will access data from disk, leading to a noticeable increase in read time. By testing various file sizes, we can approximate the file cache size.

The experiment has two stages: first we test on files with big size difference (1GB), then we do a finer-grained test on files with small size difference (0.1GB) in the range that sees a read time increase. We use a shell script to create random file of a certain size. Before every test, we we execute a `system("purge")` to force clear the system cache. We take timestamps with `std::chrono::steady_clock()::now()`.

One complication is **prefetching**. Since we are doing sequential read on files, the system can do prefetching to speed up following read. However, despite the use of prefetching, the penalty of disk access is still obvious. To determine the size of file cache, we just want to spot the very file size that causes read time increase. Thus we don't really have to eliminate this effect.

6.1.2 Prediction

The measurement of file cache size is sensitive to system workload. The following is a snapshot of memory usage on the test machine (from **activity monitor**) before our experiment:



Physical Memory:	8.00 GB
Memory Used:	5.75 GB
Cached Files:	404.9 MB
Swap Used:	2.98 GB

App Memory:	2.23 GB
Wired Memory:	2.11 GB
Compressed:	1.40 GB

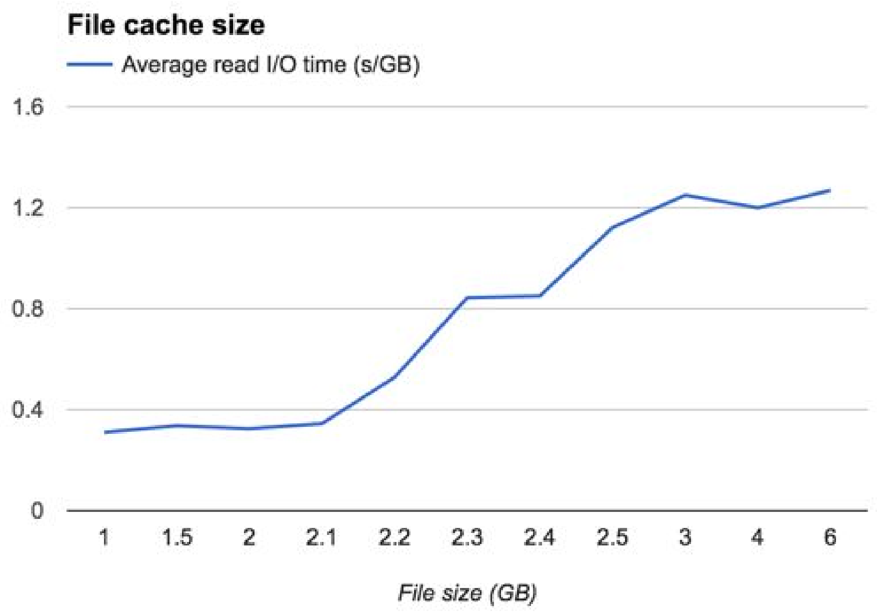
We can see that there has already about 400MB cached files, and the free memory is about **2.25 GB**. Therefore, we predict that the file cache size should be **2-2.25 GB**.

6.1.3 Result

We record the time of second sequential read and compute the average read I/O time in unit of **s/GB**.

File size (GB)	**1**	**1.5**	**2**	**2.1**	**2.2**	**2.3**	**2.4**	**2.5**	**3**	**4**	**6**
Transfer rate (GB/s)	3.22	2.97	3.08	2.90	1.89	1.18	1.17	0.89	0.80	0.83	0.79

Avg read I/O time (s/GB)	0.31	0.34	0.32	0.34	0.53	0.84	0.85	1.12	1.25	1.20	1.27



6.1.4 Analysis

In the first stage we tested on 1GB, 1.5GB, 2GB, 3GB, 4GB, and 6GB file. We noticed that read time increases from 2GB to 3GB. It's clear that the file cache size falls in this range. Then we conducted a size searching in the range. From the graph we can see the line rises up starting from 2.1 GB. Thus we conclude that the file cache size is about **2.1-2.2GB**, and this matches our prediction.

One interesting thing is that the read time stops growing again after 3GB. The possible reason is that, since then, the read time is dominated by disk access time. In other words, the file cache contributes very little to the performance improvement. However, this is because of our experiment setting: read the whole large file again in the second pass. This doesn't reflect the general file access workload.

6.2 File Read Time

6.2.1 Methodology

We will measure the file read time in absence of file cache.

Disable file cache. Having obtained the file descriptor using `open()`, we manipulate the file descriptor using `fcntl()`. `fcntl()` sets the `F_NOCACHE` flag to disable the memory file cache. Additionally, we call `system(“purge”)` to clean out any cached file previously stored before the measurement.

Block size and file size. From `diskutil info / | grep "Block Size"`, we obtain the block size of the targeting OS File System is 4KB; therefore, starting from 4KB to 256MB with increasing factor of multiplication of 4, we measure the read file time.

Sequential read and random read. In terms of sequential read, we read one block at a time from the beginning of the file until the end of the file.

In terms of random read, we randomly access the file using a random offset *k* block size and file descriptor seeks to that offset using `lseek()`.

6.2.2 Prediction

The OS is using a PCI-based solid state drive manufactured by Samsung.

Sequential read. According to QuickBench[14], a commercial MacBook benchmark tool, 4KB sequential block read is around 113.6MB/s, that is 34.38 us/block. The sequential read time of a file mainly consists of data transferring rather than file seek time.

Random read. According to QuickBench, 4KB block random read is around 34.5 MB/s, that is 113.22 us/block. The random read time of a file mainly consists of both data transferring and file seek time. In common SSD benchmark, seek time is around 100 us[15]. Thus, the 113.22 us/block approximately equals to 100us/block + 34.38 us/block, adding seek time overhead to the sequential block read.

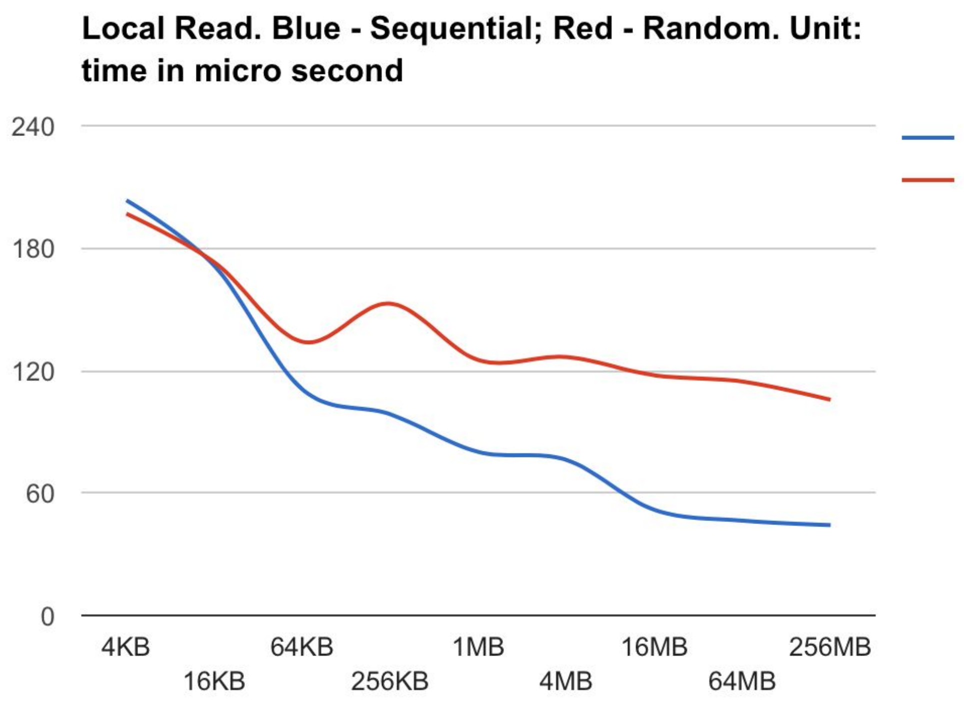
6.2.3 Result

We range the file sizes from 4KB to 256MB with increasing factor of multiplication of 4 to read in the local sequential and random read measurements. We run experiment 10 times to get the average, stdev, min, and max of the file reading time.

[14] http://www.speedtools.com/QuickBench.html
[15] https://en.wikipedia.org/wiki/Solid-state_drive

File Size	4KB	16KB	64KB	256KB	1MB	4MB	16MB	64MB	256MB
Sequential									
mean	203.423	171.284	110.78	98.5391	80.1155	76.1633	51.7337	46.4422	44.2668
stdev	78.0023	36.7683	23.432	12.9813	17.0554	16.3906	3.50419	2.88448	2.14601
min	124.302	128.523	82.1363	76.6473	63.1005	58.57	46.0349	43.6894	41.9851
max	761.769	217.482	159.863	119.584	121.996	109.232	56.6536	53.9855	47.432
Random									
mean	196.852	173.016	134.075	152.776	125.136	126.606	117.677	114.591	105.675
stdev	87.4644	59.6896	19.9016	20.1338	13.6537	9.57494	5.18635	8.26012	1.46692
min	117.66	123.919	115.511	120.79	113.792	110.156	109.646	105.644	103.655
max	878.673	334.902	189.854	177.528	156.813	143.219	123.134	132.178	107.643

Unit: time in microsecond



6.2.4 Analysis

Results. For both sequential read and random read, the read time decreases, and they becomes gradually stable with smaller standard deviation, for file sizes ranging from 4MB to 256MB.

The random read time is higher than the sequential read time, which reflects the additional seek time additional seek time in SSD for random read.

Small files. In the graph, the read time for small files is significantly longer (2x-4x) than the larger files. An explanation will be the first-time seeking the beginning offset of the target file is SSD is not negligible.

For larger files, such first-time seeking time can be amortized, because we calculate read time per block by dividing the total time of reading the entire file by the number of blocks in the file; therefore the average read time over large files is shorter than small files.

Compared to predictions. At 256MB file size, the sequential read time measured is **44.2668 us/block**; thus the prediction of 34.38 us/block is closed to the measurement result. The random read time measured is **105.675 us/block**; thus the prediction of 113.22 us/block is closed to the measurement result.

6.3 Remote File Read Time

6.3.1 Methodology

Remote File System. We use two Macbook Pros for the measurements. Macbook Pro A as the server of remote file system serves the files and the Macbook Pro B as the server accesses the file.

Macbook Pro B will mount the shared folder on its own FS, and then it will access the remote file through this mount point.

Service Message Block (SMB). In OS X built-in file sharing system, the mounting of the remote file system is through Service Message Block. Server Message Block operates as an application-layer network protocol mainly used for providing shared access to files, printers, and serial ports between nodes on a network[16].

LAN. We connect two Macbook Pros through LAN rather than public WiFi to avoid congestion issue in public WiFi, which will decrease the variance in the remote file access time.

Measurements. After mounting the remote file system onto the client's file system hierarchical tree, the measurement methodology is similar to 6.2.1.

6.3.2 Prediction

[16] https://en.wikipedia.org/wiki/Server_Message_Block

Compared to 6.2 local file access, this section measures the extra overhead of network overhead, besides the file access time.

Transfer time. In the section 5.2, the stable remote bandwidth is around 6.33 MB/s, which is 617.10 us/block for a 4KB file block. This is the transfer time from the server to the client.

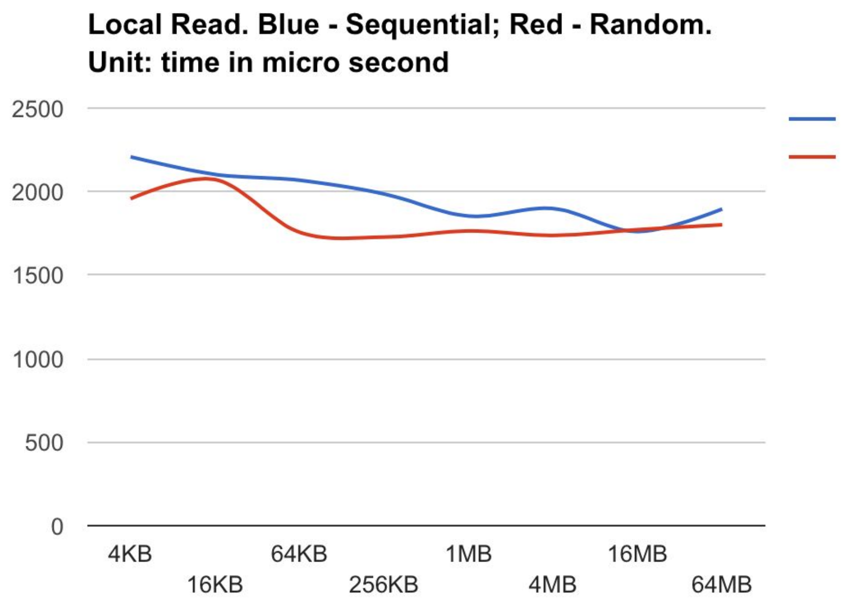
Local access time. In the section 6.2.3, the sequential file access time for 256MB file is 44.27 us/block, and random file access time is 105.68 us/block.

Overall, we predict the sequential remote file access time will be around 661.37 us/block and random remote file access time be around 722.78 us/block.

6.3.3 Result

File Size	**4KB**	**16KB**	**64KB**	**256KB**	**1MB**	**4MB**	**16MB**	**64MB**
Sequential								
mean	2208.25	2103.07	2068.26	1985.69	1853.71	1898.42	1760.76	1895.96
stdev	1262.33	553.326	742.028	377.827	296.547	615.942	180.182	187.388
min	1242.9	1453.47	1518.26	1520.87	1578.22	1529.24	1566.31	1761.21
max	5803.52	3340.16	3827.08	2647.83	2665.86	3717.25	2272.07	2368.16
Random								
mean	1957.18	2072.6	1756.84	1727.73	1764.62	1737.72	1772.29	1801.4
stdev	476.485	563.822	270.796	235.515	165.887	111.681	65.9137	112.166
min	1320.36	1552.92	1461.95	1488.02	1643.48	1569.78	1710.81	1609.41
max	2900.54	3381.44	2391.41	2219.26	2219.1	1906.81	1916.48	2051.71

Unit: time in microsecond



6.3.4 Analysis

Compared to predictions. The sequential remote read is around 1895.96 us/block. The random remote read is around 1801.4 us/block.

Both sequential access time and random access time are larger than the predictions, there is 1100 us/block difference between the actual results and predictions.

The difference can be explained by the SMB protocol overhead. SMB remote file protocol is an application-layer network protocol, which includes protocol negotiation, user authentication and end-to-end encryption[17]. We roughly estimates that such overhead will take 1100 us/block transfer.

Compared to local file access. The remote file access time is significantly larger than the local file access time. The network overhead plays a larger role in the remote file access time than the disk seek time.

There difference between random and sequential access time is smaller in remote compared to those in local because the network overhead plays a dominant rule in the total time.

Random vs. Sequential. In the graph, we see that the random access time (red line) takes less time than the sequential access (blue line),. Such abnormal phenomenon may be strange at the first glance.

17 http://www.snia.org/sites/default/education/tutorials/2012/fall/file/JoseBarreto_SMB3_Remote_File_Protocol_revision.pdf

Although we disable the file cache in the file system, there is additional layer of cache in the SMB protocol. Network cache will affect the access time. For each file size, we first test sequential access and then we test random access, although we have disabled cache on the client's cache, there may be cache on server side; therefore there may be server cache, which makes the random read testing faster because it runs right after the sequential read testing.

Small files. The access per block time for smaller files is larger than larger files. It can be explained by the fact the network first-time setup overhead, first-time caching overhead, and first-time disk seeking overhead are not well-amortized since there are fewer blocks to access in small files compared to large files.

6.4 Contention

6.4.1 Methodology

To create contention, we measure the file read time of one main process while having multiple other processes read different files. The main process performs two tasks: creating other reading processes and read a file sequentially. The main process will repeat the reading for 20 times, while other competing processes read other files for 40 times. In this way we can make sure there is a contention when the main process is doing the read.

We first create 16 random files of 4MB (since we just want to know the time of reading one file system block of data, this small file size is good enough) using the same shell script mentioned in 6.1.1. Then we conducted tests with 0-15 competing processes. In each test, we record the time of the main process reading the whole file and repeat 20 times. Then we get the average reading time of one file system block (4KB) of data by calculation.

To get a better understanding of the file system contention, we designed four test cases:

1. the main process do sequential read and others all do sequential read.
2. the main process do random read and others all do random read
3. the main process do sequential read and others do **mixed read (half sequential, half random)**
4. the main process do random read and others do mixed read.

We use `std::chrono::steady_clock()::now()` to get elapsed time. Also, in `fcntl()` we set the `F_NOCACHE` flag to disable the memory file cache and we call `system("purge")` to clear system cache. The sequential read time is purely the time loading the data, while the random read time includes the time to do `lseek()`.

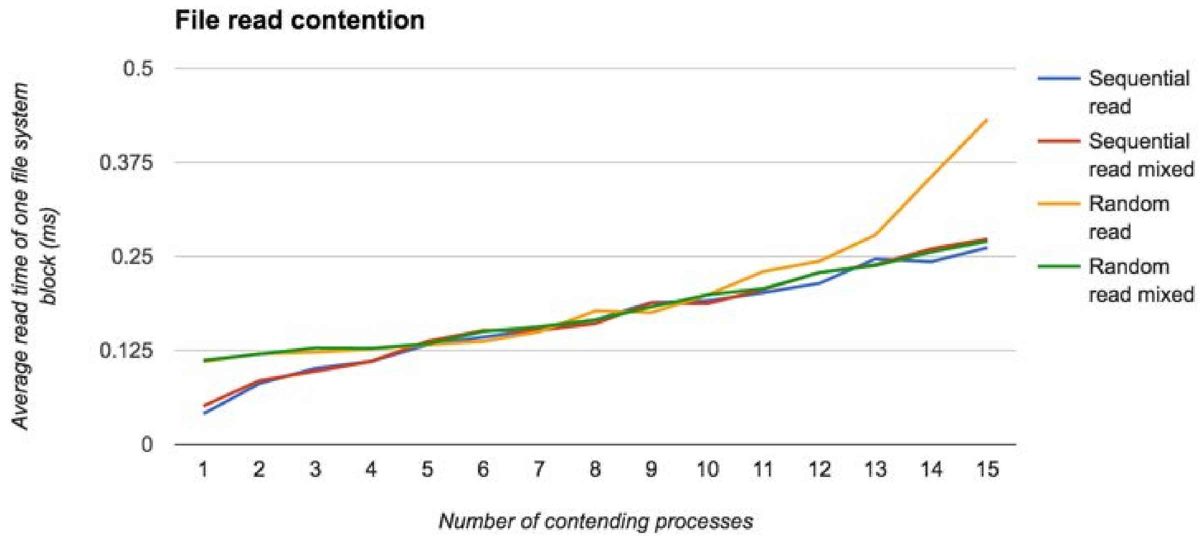
6.4.2 Prediction

Contention will incur an overhead on the read time because of process context switch and flushing prefetched data (with sequential read). The time of reading one file system block of data should grow linearly as a function of the number of competing processes.

Generally, the random read time should be larger than that of sequential read, because to do a random read, the process needs to do a `lseek()` and the pre-fetching won't help to reduce read time. For competing processes, the more they do random reads, the larger the overhead will be, because it takes more CPU cycles to do the random reads.

6.4.3 Result

Number of competing processes	Sequential read(ms)		Sequential read mixed (ms)		Random read (ms)		Random read mixed (ms)	
	avg	std	avg	std	avg	std	avg	std
0	0.04072	0.00096	0.05118	0.00295	0.11021	0.00146	0.11182	0.00644
1	0.08095	0.01005	0.08484	0.01205	0.12052	0.00349	0.12007	0.01422
2	0.10137	0.00599	0.09728	0.02326	0.12254	0.00605	0.12846	0.00947
3	0.11044	0.00352	0.11057	0.01604	0.12699	0.00182	0.12799	0.00334
4	0.13260	0.00712	0.13718	0.00674	0.13262	0.00152	0.13371	0.00254
5	0.14279	0.00302	0.15163	0.00326	0.13713	0.00391	0.15016	0.01847
6	0.15279	0.00455	0.15170	0.00297	0.14982	0.01265	0.15658	0.00418
7	0.16569	0.00986	0.16075	0.00702	0.17761	0.01597	0.16582	0.00855
8	0.18884	0.03125	0.18823	0.03041	0.17544	0.02344	0.18288	0.01345
9	0.19104	0.02022	0.18739	0.01091	0.19863	0.02134	0.19894	0.02090
10	0.20181	0.01928	0.20617	0.02108	0.22997	0.04577	0.20717	0.01781
11	0.21428	0.02272	0.22854	0.01731	0.24372	0.04945	0.22863	0.03287
12	0.24678	0.04265	0.23917	0.01996	0.27868	0.02290	0.23802	0.02480
13	0.24314	0.02877	0.26021	0.02499	0.35614	0.04726	0.25598	0.04170
14	0.26172	0.04370	0.27328	0.03289	0.43243	0.07078	0.27023	0.04401
15	0.27041	0.03066	0.29623	0.03712	0.45637	0.07649	0.30585	0.05084



6.4.4 Analysis

From the graph we can see that in all four case, the average time of reading one file system block of data grows linearly with the number of competing processes. This matches our prediction.Interestingly, the four lines are kind of identical between 5 - 10 competing processes. However, there are two divergences:

First, the sequential read has shorter read time in the beginning. The reason for this phenomenon, we believe, lies with the **prefetching effect**. With more competing processes, it’s more likely that the pre-fetched data for the main process gets flushed due to process switch. The prefetching effect becomes insignificant later on.

Second, while the random read mixed case converges with two sequential read cases (one competing process adds about 0.01 ms), the random read with all competing processes doing random reads grows much more quickly (one competing process adds 0.03-0.07ms) after 11 competing processes. We think this can be explained by the increased scheduling pressure. The more processes doing random reads, the more CPU work, and thus the more context switches. This leads to higher contention overhead.

7. Summary

Category	**Operation**	**Base hardware performance**	**Estimated software overhead**	**Predicted time**	**Measured time**
CPU	**Reading time overhead**	NA	20-100 CPU cycles	20-100 cycles	33 cycles (12.21 ns)
	Loop overhead	NA	Reading time overhead (33 cycles)	6-10 cycles (reading overhead subtracted)	6 - 8 cycles
	Procedure call overhead	NA	Reading time overhead (33 cycles)	2-4 cycles (reading overhead subtracted)	~1 cycle each extra argument
	System call overhead	NA	Reading time overhead (33 cycles)	getpid(): 33 + 2 = 35 cycles getppid(): 1-2 order of magnitude higher than getpid()	getpid(): 35 cycles getppid(): 391 cycles
	Process creation time	NA	Reading time overhead (33 cycles)	10,000 - 1,000,000 cycles	730,000 cycles
	Kernel thread creation overhead	NA	Reading time overhead (33 cycles)	One order of magnitude lower than process creation: 10,000-100,000 cycles	46,304 cycles
	Process context switch overhead	NA	Reading time overhead (33 cycles)	Half the time of creating a process: 365,000 cycles.	426,418 cycles
	Kernel thread context switch overhead	NA	Reading time overhead (33 cycles)	Half the time of creating kernel thread: 23,000 cycles.	7,684 cycles

Memory	**RAM access time**	L1 4 cycles L2 12 cycles L3 36 cycles RAM 36 cycles + 57 ns	Cache line read/write overhead	L1: 4 cycles L2: 15 cycles L3: 40 cycles Memory: 200 cycles	L1: 3.8 cycles (1.41 ns) L2: 12 cycles (4.44 ns) L3: 45 cycles (17 ns) Memory: 200 cycles (74.07ns)
	RAM bandwidth (R as read, W as write)	From machine spec: 27.8 GB/s	Cache line read/write overhead	R: 27.8 GB/s W: 27.8 GB/s	R: 10.05 GB/s W: 14.15 GB/s (both max value)
	Page fault service time	SSD access time: <0.1 ms Disk transfer time: 0.12 ms (from machine spec)	Context switch	0.2 - 0.4 ms	0.2 - 0.3 ms
Network (L as loopback , R as remote)	**Round trip time (RTT)**	System ping L: 0.037 ms R: 2.283 ms (both **min** value)	Reading overhead (12.21 ns). Negligible	Same to the system ping result	L: 0.041 ms R: 3.124 ms (both min value)
	Peak bandwidth	L: NA R: ≤100MB/s	Reading overhead (12.21 ns). Negligible	L:1560.98 MB/s R: 20.51 MB/s	L: 608.47 MB/ R: 22.51 MB/s (both max value)
	Connection setup	NA	Reading overhead (12.21 ns). Negligible	should be 1.5 times RTT L: 0.061 ms R: 5.056 ms (both avg value)	L: 0.0973 ms R: 5.9391 ms (both avg value)
	Connection teardown	NA	Reading overhead (12.21 ns). Negligible	should be 50% loopback RTT	L: 0.03 ms R: 0.05 ms (bot avg value)
File System	**Size of file cache**	The free memory space can be used as file cache (out of 2*4 GB)	None	2.25 GB (the free memory space at time of test)	2.1-2.2 GB
	File read	Sequential:	Reading overhead	Sequential: 34.38	Sequential: 44.267

	time	34.38 us/block Random: 113.22 us/block	(12.21 ns). Negligible	us/block Random: 134.38 us/block	us/block Random: 105.675 us/block
	Remote file read time	NA, depends on remote network conditions.	1100 us/block transfer from SMB protocol.	Sequential: 661.37 us/block Random: 722.78 us/block	Sequential: 1895.96 us/block Random: 1801.4 us/block. (factoring in the SMB protocol overhead)
	Contention	NA	Reading overhead (12.21 ns)	Reading time grows linearly to the number of competing processes	Reading time grows linearly to the number of competing processes (each adds approximately 0.01 ms)