CSE221 System Measurement Project Report

Qian Wang (A53088113)
Junxia Zhuge (A53099602)
Xiangyu Wang (A53096938)

Grader: Brajesh Kushwaha

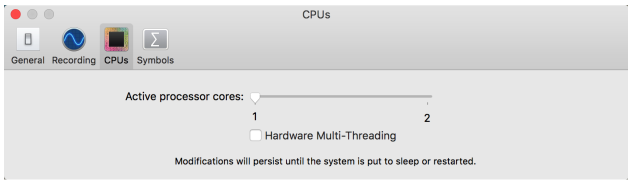
University of California, San Diego
August 21, 2016

Contents

1 Introduction

This project is the course project for CSE221. The purpose of this project is to measure the performance for a machine with the given hardware. It is also meaningful to analyze how this performance affects the software services. To achieve these two goals, we used the C programming language to implement a series of experiments. The compiler version is GCC 4.2.1 nearly without any optimization settings (compiled with "-O0"). The only optimization is to unroll all the loops by compiling with "-funroll-all-loops".

When performing all the experiments, we turned off the hardware multithreading and limited the number of active processor cores to one. This can be achieved with "Instruments" provided in OS X El Capitan directly, as shown in Figure 1.



Active processor cores: 1 (slider from 1 to 2)

☐ Hardware Multi-Threading

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

Figure 1: turn off hardware multithreading and limit the number of active processor cores

We also gave the highest priority when running our program. This can be obtained by typing "sudo nice -20 ./prog" in the terminal, when we were running the program "prog".

To achieve an accurate result, we would like to repeat the test for several times, and calculate the average value and the standard data. But, due to the system performance at a certain instance or any other issues, we may have some outliers. To decrease the influence of the outliers, we always run 60% more tests. Then we will sort all the datas, and only leave the middle ones. For instance, if we would like to test for ten times, we will in fact run for sixteen times and ignore the three smallest and three largest data. In the following section, we will not explictly emphasize this fact any more.

There are three members in our team, Qian Wang, Junxia Zhuge and Xiangyu Wang. There is no clear cut division of labor among us. Generally, we code, debug and discuss together. We think that it will cost us about 80 hours to finish this project.

2 Machine Description

We mainly used three methods to get the basic information of the machine to be tested. We first checked the system report directly. If the corresponding information could not be found easily in the system report, we typed "sysctl -a" in the terminal to get a detailed description about the kernel state of the machine. We could also search online as well.

2.1 Processor

All the processor details can be obtained by typing "sysctl -a | grep cpu" in the terminal. The processor we use is Intel(R) Core(TM) i5-5257U. We would like to emphasize that it has the microarchitecture of Broadwell [1]. This fact is useful for us to get the latency and throughput of each instruction.

The frequency of our processor is 2.70 GHz. Thus, each clock cycle will cost around 0.37 ns. The size of L1 cache is 64 KB. Notice that 32 KB of it is for the instruction cache, while the other 32 KB is for the data cache. The size of L2 and L3 cache are 256 KB and 3 MB, respectively.

2.2 Memory Bus

The information about the memory bus can be found in the system report of the operating system. Use "System Information" provided by the operating system and check the data in "Hardware/Memory", we find that there are two 64-bit wide DDR3 memory slots with the speed of 1867MHz. We also notice that the memory does not support error correcting code and is not upgradeable.

2.3 I/O Bus

The information about the I/O bus can be found in the system report of the operating system. Use "System Information" provided by the operating system and check the data in "Hardware/SATA/SATA Express", we find that the I/O bus link speed is 5.0GT/s with the physical interconnect of PCI.

2.4 RAM Size

The information about the RAM size can be found in the system report of the operating system. Use "System Information" provided by the operating system and check the data in "Hardware/Memory", we find that there are two DDR3 memory slots with the size of 4GB. Hence, the total RAM size is 8GB.

2.5 Disk

The information about the RAM size can be found in the system report of the operating system. Use "System Information" provided by the operating system and check the data in "Storage", we find that there is a solid state drive, whose model is APPLE SSD SM0128G, with the capacity of 120.47GB.

There is no concept of RPM for SSD. So, we use the read and write speed here. The data is from the official datasheet [2], which indicates that the reading speed is at most 505 MB/s and the write speed is at most 330 MB/s.

2.6 Network Card Speed

By now there is no PCI Ethernet cards installed on our machine. What we have is the AirPort Extreme Card supporting 802.11 a/b/g/n/ac WiFi wireless networking.

2.7 Operating System

The information about the operating system can be found in the system report of the operating system. Use "System Information" provided by the operating system and check the data in "Software", we find that the system version is OS X El Captain 10.11.2 (15C50) and the kernel version is Darwin 15.2.0.

3 CPU, Scheduling, and OS Services

3.1 Measurement Overhead

3.1.1 Methodology

In this part of the project, our primary task is to obtain the measurement overhead. Whenever reading from time stamp counter or executing a loop, the processor costs time. Consequently, we need to use the measurement overhead to estimate the time cost for the future tests.

We mainly utilized *rdtscp* to record the time, as inspired by [3]. The exact code script is shown below. This serializing instruction returns the time stamp counter in 64-bit format. We recorded the time twice without executing any other instructions between them. Thus, the difference is the measurement overhead for *rdtscp*. We tested for 500 times and calculated the average as well as the standard deviation afterwards.

```
asm volatile ("cpuid\n\t"
    "rdtscp\n\t"
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t"
    : "=r" (cycles_high0), "=r" (cycles_low0)
    :: "%rax", "%rbx", "%rcx", "%rdx");

// codes to be measured

asm volatile ("rdtscp\n\t"
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t"
    "cpuid\n\t"
    : "=r" (cycles_high1), "=r" (cycles_low1)
    :: "%rax", "%rbx", "%rcx", "%rdx");

start = ( ((uint64_t) cycles_high0 << 32) | cycles_low0 );
end = ( ((uint64_t) cycles_high1 << 32) | cycles_low1 );
data[i] = (end - start);
```

3.1.2 Result

According to [4], the throughput for *rdtscp* is about 30 clock cycles on our machine. Moreover, the assembly instruction *mov*, copying an operand, is much faster than reading time. Its throughput is just around 0.5 clock cycles. Thus, we estimated that the overhead of reading the clock should still be around 30 clock cycles.

a. Base hardware performance: 0 clock cycles

b. Estimated software overhead: about 30 clock cycles

c. Predicted operation time: about 30 clock cycles

d. Average measured time: 32.128 clock cycles

e. Measured time standard deviation: 2.881 clock cycles

3.1.3 Discussion

Our result is close to the prediction. The standard deviation is also small enough. Hence, our result is acceptable.

3.2 Loop Overhead

3.2.1 Methodology

As for measurement overhead for loops, we wrote an empty loop with nothing inside it and made it iterate for 500 times. Then, we used *rdtscp* to record time both before and after the loop. The difference between the two values should be the cost of 500 iterations. We repeated this process for 500 times.

3.2.2 Result

To estimate the running time, we should first get the assembly corresponding to the loop overhead. Base on the following assembly code, we notice that one “cmp”, two “jump”s, two “mov”s and one “add” is needed. According to [4], “cmp”, “mov” and “add” cost about 0.5 clock cycles, while “jump” cost 2 clock cycles.

```
cmpl    $500, -4088(%rbp)
jge     LBB1_6

......

movl    -4088(%rbp), %eax
addl    $1, %eax
movl    %eax, -4088(%rbp)
jmp     LBB1_3
```

a. Base hardware performance: $0.5 + 2 * 2 + 2 * 0.5 + 0.5 = 6$ clock cycles

b. Estimated software overhead: about $30/500 = 0.06$ clock cycles

c. Predicted operation time: about $6 + 0.06 = 6.06$ clock cycles

d. Average measured time: 6.960 clock cycles

e. Measured time standard deviation: 0.196 clock cycles

3.2.3 Discussion

Our test showed that the loop overhead is around 6.960 clock cycles. We regarded this small difference acceptable.

3.3 Procedure Call Overhead

3.3.1 Methodology

To test procedure call overhead of different parameter sizes, we created 8 minimum procedures with 0-7 arguments respectively, and inserted *rdtscp* instruction before and after the procedure call. We iteratively tested the differences of the two time stamp values for 500 times and computed their average as well their standard deviation afterwards.

3.3.2 Result

From the assembly code, we saw that the increment overhead of an argument came from one more "mov" instruction. Since it only takes about 0.5 clock cycle, the increment of an argument will basically introduce nearly no influence. Considering the measurement overhead of 30 clock cycles, we expect the result of the relation between number of arguments and running time to be:

$$T = 0.5n + 30$$

```
_proc0:                                   ## @proc0
    ......
    ......


_proc1:                                   ## @proc1
    ......
    movl    %edi, -4(%rbp)
    ......

_proc2:                                   ## @proc2
    ......
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    ......
```

The results are presented as a function of number of integer arguments from 0 to 7 in Table 1. To make the discussion as reasonable as possible, we tested for several times and picked up the result with the minimum standard deviation.

As plotted in Figure 2, we fit the data into the following linear equation, where T is the clock cycle and n is the number of arguments:

$$T = 0.5355n + 29.3367$$

Table 1: procedure call overhead as a function of number of integer arguments from 0 to 7

#(aug)	measured t	std	#(aug)	measured t	std
0	28.809 cycles	8.141 cycles	4	31.536 cycles	6.897 cycles
1	29.514 cycles	6.803 cycles	5	31.380 cycles	6.885 cycles
2	31.392 cycles	6.844 cycles	6	32.460 cycles	4.758 cycles
3	31.470 cycles	6.860 cycles	7	33.126 cycles	7.096 cycles

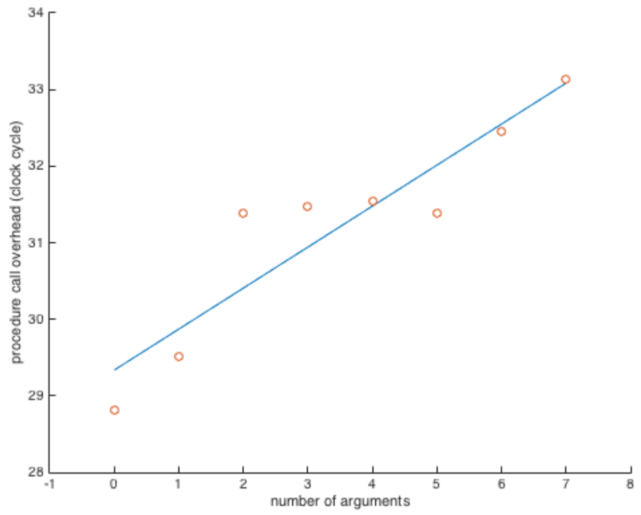


Figure 2: procedure call overhead as a function of number of integer arguments from 0 to 7

3.3.3 Discussion

This result is consistent to the fact that the measurement overhead for reading the time is about 30 clock cycle and the increment overhead of an argument is about 0.5 clock cycles. Hence, we think the result is acceptable.

3.4 System Call Overhead

3.4.1 Methodology

To test the system call overhead, we utilized the system call *getpid* as a benchmark. As mentioned in the project description, the operating system may cache the result of *getpid* when it was called the first time. This will makes the subsequent calls to *getpid* not a complete system call. To solve the problem, in each measurement, we forked a new process and called *getpid* inside the child process. By adding *rdtscp* before and after *getpid* in the child process, we calculated the time cost of the system call *getpid*. After child process finished, it would entirely exit. In the parent process, we kept waiting until the child process exited, before proceeding to the next measurement. We repeated the measurement for 500 times with a loop.

3.4.2 Result

It is hard for us to predict the exact running time of a system call, but we expected this time to be much larger than that of a procedure call. The reason why we did this guess is covered in the discussion subsection.

a. Estimated software overhead: much larger than procedure call

b. Predicted operation time: much larger than procedure call

c. Average measured time: 31795.840 clock cycles

d. Measured time standard deviation: 3390.188 clock cycles

However, we cannot always achieve this kind of result, due to the CPU stabilization. But there is no way to modify the access right to enable CPU stabilization in OS X 10.11. So, this is nearly the best we can do.

3.4.3 Discussion

From the result, we found that the system call overhead is significantly expensive than the procedure call overhead. Though it seems that the way we make a system call when coding is the same as making a procedure call, what happens under the operating system is different in these two situations.

As mentioned in the textbook [5], when conducting a system call, we will trap into the kernel by switching from the user mode to the kernel mode. After the system call ends, we will return the control to the previous program. Besides, the instructions inside the *getpid* procedure could consume a bunch of CPU cycles. All these factors may lead to a large difference between the running time of a procedure call and a system call.

While executing the system call, there might be TLB and cache misses, which will cost hundreds of cycles. If there are cache and TLB evictions, further cost will be introduced. These cache and TLB issues introduce uncertainty to our measurement. This can be an acceptable explanation to our relatively high standard deviation.

We thought our result is acceptable, since the time for system call has the same order of magnitude as that in the Plan 9 paper [6].

3.5 Task Creation Time

3.5.1 Methodology

In this part, we tested the time cost of two important functions, process creating and thread creating.

For process creating, we used the system call *fork*, which creates a copy of the process itself. By adding *rdtscp* before and after the system call *fork*, we calculated the time consumption for every *fork*. We tested for 500 times and calculated the average as well as the standard deviation of the time cost.

For thread creating, we utilized function *pthread_create* in the *pthread* library. By adding *rdtscp* before and after it, we calculated the time consumption for every *pthread_create*. We tested for 500 times and calculated the average as well as the standard deviation of the time cost.

Note that for both process creating and thread creating, the body of the child process and the child thread is empty. This is to avoid introducing any further bias.

3.5.2 Result

It is hard for us to predict the exact running time to create a process or a thread, but we made some guess. In the paper about Plan 9 during the lecture, it is shown that the time for a fork is nearly 200 times as long as a system call is [6]. Thus, we estimated it as about several million clock cycles.

a. Estimated software overhead: several million clock cycles

b. Predicted operation time: several million clock cycles

c. Average measured time: 3581303.004 clock cycles

d. Measured time standard deviation: 2850589.256 clock cycles

According to what we learnt in the undergraduate operating system course, we also expected the thread creation faster than process creation.

a. Estimated software overhead: much smaller than process creation

b. Predicted operation time: much smaller than process creation

c. Average measured time: 61439.616 clock cycles

d. Measured time standard deviation: 21856.054 clock cycles

3.5.3 Discussion

We found that process creation costed much more time than thread creation. The system call *fork* creates a copy of the caller. The textbook [5] explains that "after the *fork*, two processes have the same memory image, the same environment strings, and the same open files." Consequently, OS needs to copy a bulk of data to *fork* a new process.

As for thread creation, however, OS does not need to copy all of the data from the calling thread, since the caller and the callee will share the address space in this situation. This can save much time. Also, OS might recycle threads. "When a thread is destroyed, it is only marked as not runnable. When a new thread is created, the old thread is reactivated" [5], which saves some overhead.

We thought our result is acceptable, since the process creation time has the same order of magnitude as that in the Plan 9 paper [6] and the thread creation time is much less than process creation time.

3.6 Context Switch Time

3.6.1 Methodology

To test context switch time overhead, we need a way to force context switch. Notice that what we really get is two context switches, so the result we presented later has been divided by two already.

We understand that in the blocking pipe, when one process tries to read from one end of an empty pipe, it will be blocked until any data is available. Therefore, we built a pipe and read from one of its ends in the parent process. The child process would write on the other end. Thus, when we tried to read from the parent process, it would first be blocked and force the context to be switched to the child process. To actually measure the time cost, we inserted two *rdtscp* calls before and after the *read* system call. For context switch from one kernel thread to another, we measured it in a similar way to the process context switch, except that we used *pthread_creat* to create a new kernel thread.

3.6.2 Result

It is hard for us to predict the exact running time for process and thread context switch, but we made some guess. In the paper about Plan 9 during the lecture, it is shown that the time for a context switch is nearly 6.5 times as long as a system call is [6]. Thus, we estimated it as less than one million clock cycles.

a. Estimated software overhead: less than one million clock cycles

b. Predicted operation time: less than one million clock cycles

c. Average measured time: 10447.638 clock cycles

d. Measured time standard deviation: 1704.054 clock cycles

According to what we learned in the undergraduate operating system course, we also expected the thread context switch faster than process context switch.

a. Estimated software overhead: much smaller than process context switch

b. Predicted operation time: much smaller than process context switch

c. Average measured time: 3243.528 clock cycles

d. Measured time standard deviation: 684.544 clock cycles

3.6.3 Discussion

From the result above, we can see clearly that the process context switch is much more expensive than the thread context switch. Under both circumstances, the task of saving and loading the state registers is necessary. The most significant distinction between the process context switch and the thread context switch is that, during the process context switch, the operating system has to switch the page table under using [7]. Depending on both hardware and operation system, the cost of this process may be different.

4 Memory

4.1 Memory Access Time

4.1.1 Methodology

The methodology is inspired by the material in section 6.2 of [8]. We defined an integer list with the size ranging from 2^{10} to 2^{27} bytes. By accessing entries of the array with strides from 2^5 to 2^{10} bytes for 500 times, we can access data in memory or caches. Thanks to the spatial locality [9], when we read or write an entry in the array, the OS will store subsequent entries into the memory or a certain level of the cache as well in order to save time. Consequently, with the stride and array both growing larger, we will gradually access L1 cache, L2 cache, L3 cache and the memory.

4.1.2 Result

We predicted that our measured data should be nearly the twice as the theoretical value reported in [4]. One reason is that it is always hard to achieve the best latency reported in a datasheet. Several affecting factors will be discussed later. The access time for the main memory may vary according to different brands. However, we did not find any official information from Apple Inc. Thus, we predicted this value as 2 times long as that of L3 cache.

Since the time is measured directly from fitting the plot, we did not report the standard deviation here. The detailed data is presented through Figure 3. The blue thick line is the predicted time; the red thick line is the measured time. Recall that we have 32 KB L1 data cache, 256 KB L2 cache and 3MB main memory. This fact can also be illustrate by Figure 3.

The data of the base hardware performance are from [4].

a. Base hardware performance: about 4/10/30/60 clock cycles

b. Estimated software overhead: 4/10/30/60 clock cycles

c. Predicted operation time: about 8/20/60/120 clock cycles

d. Average measured time: 10/15/50/100 clock cycles

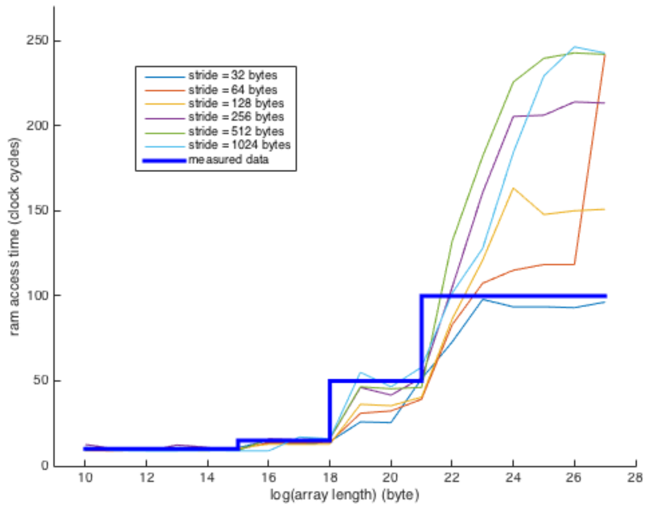


Figure 3: ram access time

4.1.3 Discussion

Apart from the varying environment, there are also some factors bringing the overhead to use. Since we test in a loop there must be some loop overhead. Also, the way in which we access each entry is to maintain a pointer. Thus, the writing operation to update the value of this pointer is also the overhead, since what we really would like to focus on the reading access to an entry in the array.

From the discussion above, we think that our result is acceptable from the view of both time and size.

4.2 RAM Bandwidth

4.2.1 Methodology

The methodology is inspired by the material in section 5.1 of [8]. When testing the read bandwidth, we kept assigning the values in an array to a variable. When testing the write bandwidth, we kept assigning the value of a variable to every entry in an array. This process was repeated by 500 times. We created 2^{30} byte long array, whose entry is a 1.16 MB structure.

```
typedef struct {
    uint64_t a[190000];
} myT;
```

The structure has the size of nearly half of L3 cache. Thus, we can expect that the variable is always in the cache. Besides, there is no space to prefetch any entry in the array. Thus, every assignment can be regarded as exactly one memory access. The structure is also large enough for us to ignore the influence of loop overhead.

4.2.2 Result

In [8], the authors said that "the Power2 is capable of up to 800M/sec read rates", while they actually got about a quarter of it. Thus, we did not expect our result to be close to the theoretical result. The theoretical memory bandwidth can be calculated as:

$$2\ memories * 64\ bit/memory * 1867\ MHz/8\ bits/1024 = 29.17\ GB/s$$

Here, it means that there are two 64 bit memory slots in our machine. The memory bus frequency is 1867 MHz

In [8], it is also said that "the processor cost of each memory operation is approximately the same as the cost in the read case." Hence, we expected the read and write bandwidth would be close to each other.

a. Base hardware performance: 29.17GB/s

b. Estimated software overhead: about 75% of the theoretical value

c. Predicted operation time: 7.5 to 8.5 GB/s for both read and write

d. Average measured time: 8.74 GB/s (read) 8.62 GB/s (write)

e. Measured time standard deviation: 0.0389 GB/s (read) 0.0734 GB/s (write)

4.2.3 Discussion

While our result cannot reach the theoretical speed, we can argue that there are actually many factors which will prevent us to achieve maximum bandwidth. First of all, many kernel applications including the operating system itself will compete for the CPU time. Second, we use loop to iterate through our array to do read or write, which will unavoidably introduce some cost in CPU cycles. Although we have used loop unrolling, it will still reduce our tested result. Third we think that the maximum memory frequency may be designed for video streaming applications with special instructions to achieve, our array read and write may theoretically not be able to get the calculated max bandwidth. By the way, I only get around 10GB/s read/write speed with commercial benchmark software called Geekbench3.

But anyway, compare to the result in [8], our result is acceptable.

4.3 Page Fault Service Time

4.3.1 Methodology

We used the system call *mmap* to append a large file (131.6 Mb log file) to the address space of a process. Due to the lazy load, pages which contain the file data do not reside in the memory at the beginning. By accessing the file data after *mmap* immediately, the page fault occurs. We can test page fault service time after *mmap* a file. In order to prevent accessing a page which is already in the memory, we *munmap* the file and *mmap* the file again every test. This process is repeated for 500 times.

4.3.2 Result

Previously, we have obtained the main memory access time is 100 clock cycle per byte. The page size in one system is 4 KB [10]. When the page fault happens, the system will copy 4 KB data from the disk to the memory. That is to say, the bottle neck is the disk speed. Therefore, we can estimate the page fault time as a value larger than memory access time, such as 1.5 times of memory access time:

$$100/byte * 4096\ bytes * 1.5 \approx 6000000$$

In [5], we learned that every time page fault occurs, "the MMU notices that the page is unmapped and causes the CPU to trap to the operating system. The operating system picks a little-used page frame and writes its contents back to the disk." Consequently, trapping to the OS, picking the page to be evicted and modifying the page table will all cause page fault overhead. But we do not think that these will cost too much time, compared with updating the page.

Our result is as following:

a. Base hardware performance: 600000 clock cycles

b. Estimated software overhead: about 20000 clock cycles

c. Predicted operation time: 620000 clock cycles

d. Average measured time: 623095.32 clock cycles

e. Measured time standard deviation: 17189.88 clock cycles

4.3.3 Discussion

Our testing result shows that we consumed 623095.32 cycles. Given the fact that actually the SSD speed can vary in different conditions, We think the result is acceptable. As we mentioned in the result, this value means that the the disk access time is higher than the memory access time, which is reasonable.

5 Network

In this chapter, we perform all the experiments on two exactly the same laptops. The TCP socket programming code is mainly adapted from the textbook [11].

5.1 Round Trip Time

5.1.1 Methodology

In order to test the round trip time, we set up two processes to establish TCP connection and transmit/receive packets. For the loopback connection, we set these two processes on the same machine. While for the remote connection, we set up two processes on two identical machines respectively. We record the time cost between sending a 56 character long string (the same as the data length in ping) on the client machine and receiving *ack* from server machine. The size of the data length in ping is shown directly on the terminal when we are calling it. We repeat the test for 500 times.

To test the average value and standard deviation of ping, we can call it in terminal and run for a relatively long time. After we terminate the process, the required data will be presented by itself.

5.1.2 Result

We estimate that the round trip time and the ping time should generally be the same. Because of the time-varying network environment, we cannot predict which one is larger. However, at least they should be on the same order. Moreover, this operation is so simple that the software overhead can be ignored.

For the loopback operation, the average ping time is 0.122 milliseconds, which is 329729.7 clock cycles. The standard deviation is 0.015 milliseconds, which is 40540.5 clock cycles. We use the average value of ping as our predicted operation time.

a. Base hardware performance: 329729.7 clock cycles

b. Estimated software overhead: ignorable

c. Predicted operation time: 329729.7 clock cycles

d. Average measured time: 365951.1 clock cycles

e. Measured time standard deviation: 5959.8 clock cycles

For the remote operation, the average ping time is 64.063 milliseconds, but the standard deviation is 34.686 milliseconds. Thus, it is not useful to compare our result to this value. Instead, we use the minimum value of ping, 1.798 milliseconds (4859459.5 clock cycles), as our predicted operation time.

a. Base hardware performance: 4859459.5 clock cycles

b. Estimated software overhead: ignorable

c. Predicted operation time: 4859459.5 clock cycles

d. Average measured time: 5538814.8 clock cycles

e. Measured time standard deviation: 170473.8 clock cycles

5.1.3 Discussion

The round trip time we get here is comparable to the ping time. The time spend includes kernel overhead of network stack, the delay of transmitting signal in network channel, and the delay introduced by queue in switch/router. In our experiment of remote connection, we worked on local WiFi network, therefore, signal propogation should not take too long. Thus, we believe our result is acceptable.

5.2 Peak Bandwidth

5.2.1 Methodology

To test the peak bandwidth, we create a string with the size of 128 MB, and transmit the string through the program implemented by socket programming. By dividing the buffer size with time consumption, we can get the peak bandwidth. And we repeated the test for 500 times.

5.2.2 Result

For the local operation, we can view the whole process of client's reading from the string, client's writing into the buffer, client's reading from the buffer and server's writing into the string. In total, there are four read/write operations at the same time. In the previous chapter, we have discuss our result about the memory bandwidth for both operations. Hence, the following data is the data for memory bandwidth divided by four.

a. Base hardware performance: 7.29 GB/s

b. Estimated software overhead: about 75% of the theoretical value

c. Predicted operation time: 1.88 to 2.13 GB/s

d. Average measured time: 2.970 GB/s

e. Measured time standard deviation: 28.178 MB/s

For the remote operation, since there are two devices connected to the same router, we claimed the base hardware performance as half of the bandwidth of the router we used. The router we used is TL-WR841N with a bandwidth of 300 Mbps [12], which is 37.5 MB/s. Thus, we estimated the peak bandwidth should be half of this value, 18.75 MB/s. However, we can hardly achieve this value. Here, we still regard the overhead as about 75% of the theoretical value. The affected factors will be discussed later.

a. Base hardware performance: 18.75 MB/s

b. Estimated software overhead: about 75% of the theoretical value

c. Predicted operation time: 4.5 to 5.5 MB/s

d. Average measured time: 5.602 MB/s

e. Measured time standard deviation: 0.0294 MB/s

5.2.3 Discussion

The packet sent to a local host basically will go down to the network stack, move to network buffer of another process and then go all the way back up to the application receiving it. There are two factors that may limit the bandwidth of connection to local host, memory bandwidth and CPU power. When packet goes down and up the network stack, CPU need to assembly it, parse it and run various network algorithm as well. All this operation will take CPU cycles. When transmission speed reach GB/s domain, the CPU could high likely be the bottleneck as it runs at less than 3 GHz. For memory bandwidth, it's around 7 GB/s. We think 7 GB/s is too high a target for CPU to reach, since we disabled multicore of our platform and the single CPU need to handle two of our TCP applications and two network stack operations. Although network usually won't reach CPU bound until 10 GB/s, we expects our ideal result to be less than half of that considering we have one CPU for both client and server. The CPU utilization we got from system monitor can also prove that we have reached CPU limitation. A We claim that we have reached the peak bandwidth for this machine and the bottleneck is the CPU speed.

Our remote bandwidth was tested under home WiFi environment. Two of our laptops connected to the same WiFi router. As the two laptops were in the same local network, we assume the traffic between them were switched, with no routing involved. Additionally, our WiFi router claims to have bandwidth of 300 Mbps, therefore, we assume the wireless channel was good enough to handle our testing traffic bandwidth. Then the tested 5MB/s bandwidth can only be explained by the limitation of router CPU. As I have tested with a commercial software, 5MB/s throughput did use 100% CPU resources. Note that a router usually has a CPU running at 500MHz with RISC instruction set. Other factors like network protocol overhead, radio interference, physical obstructions on the line of sight between devices, and distance between devices will also have some effects on the result.

5.3 Connection Overhead

5.3.1 Methodology

To test connection overhead, we record the time consumption of *connect* and *close* function call. We do not need to really transfer data in this test. We repeat the test for 500 times.

5.3.2 Result

In order to establish a connection, TCP uses three-way handshake. Before a client attempts to connect with a server, the server must first bind to and listen at a port to open it up for connections: this is called a passive open. Once the passive open is established, a client may initiate an active open. To establish a connection, the three-way handshake occurs. As TCP connection setup involves three packets send back and forth. A reasonable setup time should be 1.5 times of the round trip time, no matter it is on local host or remote host.

For the loopback operation, the result is:

a. Base hardware performance: 494593.5 clock cycles

b. Estimated software overhead: ignorable

c. Predicted operation time: 494593.5 clock cycles

d. Average measured time: 493382.7 clock cycles

e. Measured time standard deviation: 60057.35 clock cycles

For the remote operation, the overhead may be larger than the local case. We will discuss in more details later in the discussion part.

a. Base hardware performance: 8308222.2 clock cycles

b. Estimated software overhead: 17000000 clock cycles

c. Predicted operation time: 25000000 clock cycles

d. Average measured time: 21756389.7 clock cycles

e. Measured time standard deviation: 257902.25 clock cycles

In the real world, according to [13], a four-way handshake is used during the teardown process. The server and the client can close the connection independently. The one willing to end a connection can send a FIN packet, and the other side send back an ACK. The side sending the FIN packet will wait for a timeout after receiving the ACK, before eventually closing the connection.

In socket programming, to tear down a connection, we can directly use the system call *close*. Thus, we expect our result to be nearly the same as what we got previously when measuring the time of system call. There should be nearly no difference for either loopback or remote operation. Notice that there is no need to really transmit a message and to wait for a reply. So, compared with the setup time, the teardown time should be much smaller.

For the loopback operation, the result is:

a. Estimated software overhead: much smaller than setup time
b. Predicted operation time: about 35000 clock cycles
c. Average measured time: 34394.8 clock cycles
d. Measured time standard deviation: 1771.3 clock cycles

For the remote operation, the result is:

a. Estimated software overhead: much smaller than setup time
b. Predicted operation time: about 35000 clock cycles
c. Average measured time: 36027.6 clock cycles
d. Measured time standard deviation: 800.2 clock cycles

5.3.3 Discussion

As TCP connection setup involves three packets send back and forth. A reasonable setup time should be 1.5 times of the round trip time whether it's on local host or remote host. I think our result is reasonable enough for local case. In the remote case, we observe 5 times of round trip time.

This part is tricky in our view. We checked the TCP state transaction in the connection process. We did measurement in different time of day under different time interval. We checked SNR of our WiFi router. However, we got the same result whatever we tried. Here are some possible reason. The OS X may schedule idle connection, that is just connecting but not sending data, with lower priority. Our program just stay in the ready queue longer after the *connect* returns. Or, there are some weird queuing behavior in the WiFi router which delay our connection packets longer.

An interesting thing to discuss here is the tear down process. Tear down operation involves TCP state changes beyond the *close* system call. That is to say, there will be TCP state transition even after *close* returns. As far as we know, we can track each TCP connection's state through the *netstate* command. Therefore we wrote a bash script to repeatedly capture TCP states of our TCP connections. The result is not so accurate since the *netstat* and *grep* and *gdate* commands takes more time than a tear down time. Our measuring accuracy can't even reach the tear down time. But it do provide us a upbound of teardown time of about 30ms on remote connection. This should be reasonable.

6 File System

6.1 Size of File Cache

6.1.1 Methodology

In order to test the size of file cache, we created ten files with size from 1 GB to 10 GB. At first, we read the file sequentially. If the file size is smaller than the file cache, the file will be stored in the file cache completely. And then we read the file sequentially for the second time. Because the file is in the file cache, the read operation will cost much less time. On the other hand, if the size of the file is larger than the file cache, the file will be partially stored in the file size. If we read the file sequentially for the second time, we might fetch the file data from the disk storage, which costs much more time than file cache. Consequently, we can roughly determine that the file cache size is between 5 GB to 6 GB. And then, we created files with range from 5.1 GB to 6.0 GB to test the accurate size of file cache.

6.1.2 Result

All of the file cache is from the main memory. Our machine has a main memory of 8GB. But it is hard to use up all of these space. We estimated that there will be about 30% of main memory used by other processes. This issue will be discussed in details in the discussion part.

Figure 4 shows the relation between the block access time and the file size. From this figure, we can find a sudden change when the file size increases in the stage from 5.3 GB to 5.7 GB. Thus, we claim that the file cache should actually be 5.5 GB. The blues line is the average value of the access time, and the red line is the standard deviation for the access time. The small standard deviation means that our result is acceptable. Since we gradually increase the file size by 0.1 GB during the measurement, we will regard the standard deviation of the file cache as $0.1/5.5 = 1.82\%$

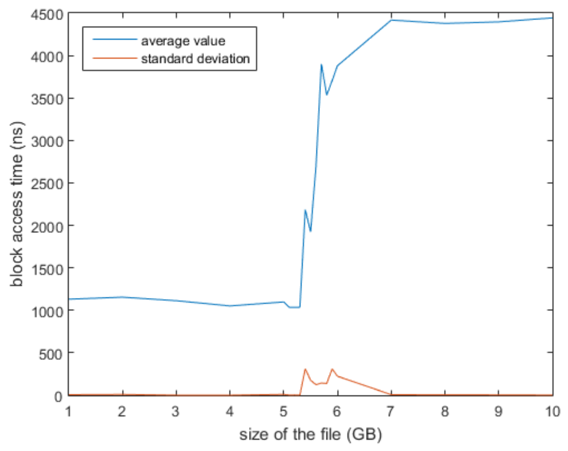


Figure 4: relation between block access time and file size

a. Base hardware performance: 8 GB

b. Estimated software overhead: about 30% of the theoretical value

c. Predicted operation time: 5.6 GB

d. Average measured time: 5.5 GB

e. Measured time standard deviation: 0.1 GB

6.1.3 Result

The main memory is not only used to store file cache, but also, it needs to store virtual memories for each process as well as the files opened by other processes. Consequently, we think that the result is acceptable.

6.2 File Read Time

6.2.1 Methodology

In order to test the file read time, we created files with size of $4, 8, 16, \cdots, 512$ KB. And then, we read the files sequentially and randomly. For the sake of eliminating the overhead of calculating random number, we created an array with size of the file block number ahead. For sequential read, we stored $0, 1, 2, \cdots, N-1$ into the array, in which N represents the number of file blocks. For random read, we stored random number into the array. Consequently, we did not need to consider the time consumed by generating random number.

Also, we need to restrict caching, because file cache will reduce the time to read file from disk storage. We utilized *fcntl* function with parameter *F_NOCACHE* to prohibit the use of file cache.

6.2.2 Result

From the hardware description, we know that our machine has a solid state drive. So, we expect the sequential read time and the random access time to be nearly the same. Figure 5 proves the validity of our expectation. To predict the block access time, we basically refer to the SanDisk SSD datasheet [2], which says that the read latency should be 60 microseconds. But this theoretical maximum is hard to achieve, so we predict the overhead to be 60 microseconds as well.

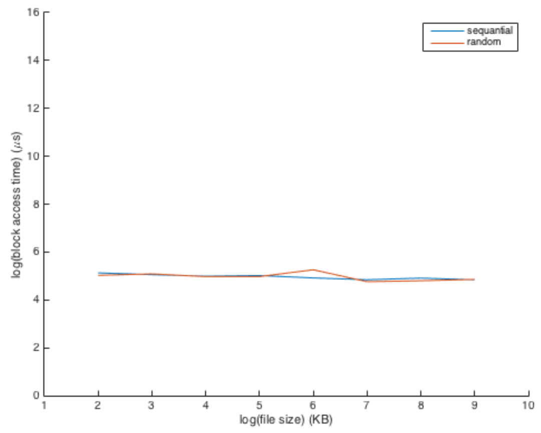


Figure 5: relation between block access time and file size

For sequential access, our result is:

a. Base hardware performance: 60 μs

b. Estimated software overhead: 60 μs

c. Predicted operation time: 120 μs

d. Average measured time: 130.06 μs

e. Measured time standard deviation: 0.1964 μs

For random access, our result is:

a. Base hardware performance: 60 μs

b. Estimated software overhead: 60 μs

c. Predicted operation time: 120 μs

d. Average measured time: 127.53 μs

e. Measured time standard deviation: 0.2268 μs

6.2.3 Discussion

Due to the utilize of SSD in our test computer, we think the result is acceptable. "SSDs have the same characteristics as traditional hard disks but can be more reliable because they have no moving parts and faster because they have no seek time or latency."[14] Consequently, we do not need to cost extra time with random read.

6.3 Remote File Read Time

6.3.1 Methodology

For remote file read time test, we just utilize the same method with the previous test to read a file from an identical machine.

6.3.2 Result

The bottleneck for remote access is the network latency and bandwidth. We expect the access time to be equal to about 2 to 3 times of remote round trip time, which is about 5 milliseconds. Compared with this bottleneck, the time to really access the block from the disk is ignorable. Also, we expect the sequential read time and the random access time to be nearly the same as well. Figure 6 proves the validity of our expectation.

For sequential access, our result is:

a. Base hardware performance: 5 ms

b. Estimated software overhead: ignorable

c. Predicted operation time: 5 ms

d. Average measured time: 6.676 ms

e. Measured time standard deviation: 0.0767 ms

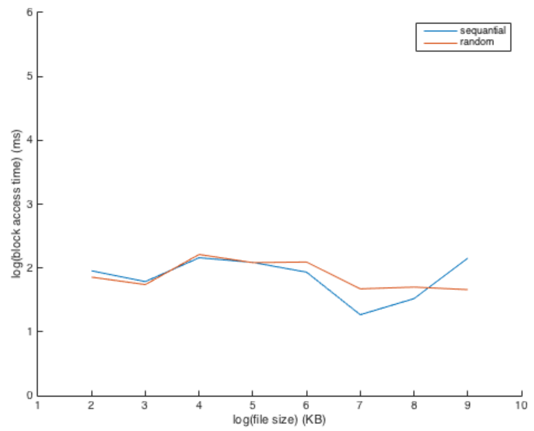


Figure 6: relation between block access time and file size

For random access, our result is:

a. Base hardware performance: 5 ms

b. Estimated software overhead: ignorable

c. Predicted operation time: 5 ms

d. Average measured time: 6.677 ms

e. Measured time standard deviation: 0.0789 ms

6.3.3 Discussion

The remote file read time in our case is about 2.5 times of the remote round trip time. This is reasonably, since when we read, a remote procedure call will be made to the remote file server and the sever will read the file on behave of the client and send the data back through packets. In our experiment, we read one block a time which is 4 KB per read. There is a limitation on maximum segment size for TCP segment which is negotiated by file server and client. The MSS usually is smaller than MTU of underlying layers, which makes it around 1500 Bytes. This means the 4 KB block file might need 3 packets to be transmitted. However, the default congestion control window size for OS X might be 2. Then the file transfer takes about two Round trip time. Therefore, the total cost should be one RPC ($0.5 roundtriptime$) + 3 packets for 4 KB file ($2 roundtriptime$). They add up to 2.5 round trip time.

6.4 Contention

6.4.1 Methodology

In order to test contention time cost, we created 1 to 10 processes to read files with size of 32 MB sequentially from the disk storage concurrently. Different process read different file. We utilized *fcntl* function with parameter *F_NOCACHE* to prohibit the use of file cache. For every different number of processes, we test for 10 times.

6.4.2 Result

We predicted that, with the number of processes increasing, the block access time should also increase.

The relatively small standard deviation and the linear growth and 60-μs long block access time for a single process (nealy the same as the official latency in [2]) in Figure 7 indicates that our result is acceptable. We fit the result linearly to obtain:

$$T = 28.3555n + 53.0963$$

which means that increasing one process will increase the block access time by 28.3555 microseconds.

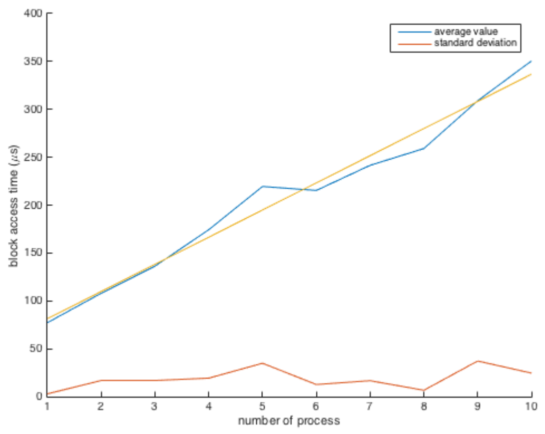


Figure 7: block access time increases linearly with the number of process

We predicted that the addition of one process may introduce a 60-μs long increase of block access time. But it is possible that the CPU may prefetch the blocks for one process not taking the processor. This optimization may make the increase rate lower.

a. Base hardware performance: 60 μs

b. Estimated software overhead: minus half of the theoretical value

c. Predicted operation time: 60 μs

d. Average measured time: 28.35 μs

Since the result is derived by fitting the plot, we will not report the standard deviation here.

6.4.3 Discussion

When there is only one process, operating system cannot conduct prefetching because CPU is faster than fetching data from disk storage. However, when there are two or more processes, operating system can prefetch file blocks from disk storage for the processes in the ready list. Due to the sequential read, prefetching could decrease read time dramatically. "To improve performance, the cache manager keeps a small history of read requests and from this history attempts to predict future requests. If the cache manager finds a pattern in the previous three requests, such as sequential access forward or backward, it prefetches data into the cache before the next request is submitted by the application. In this way, the application may find its data already cached and not need to wait for disk I/O."[14] Consequently, we think the result is acceptable.

7 Summary

All the results from our experiments are listed in Table 2 on the final page. For convenience, we use the second as the unit. The conversion follows the relation we derived in the machine description section. We also use the percentage to present the standard deviation for clarity. The original data of average value and standard deviation for any measurement in clock cycle can be found in previous chapters.

Table 2: result summary for all the experiments

operation	hw base	sw overhead	predicted	measured	std
measurement	0 ns	11 ns	11 ns	11.899 ns	8.967%
loop	2.22 ns	0.022 ns	2.242 ns	2.575 ns	2.835%
procedure(0)	-	11.1 ns	11.1 ns	10.659 ns	28.25%
procedure(1)	-	11.285 ns	11.285 ns	10.920 ns	23.049%
procedure(2)	-	11.470 ns	11.470 ns	11.615 ns	21.799%
procedure(3)	-	11.655 ns	11.655 ns	11.644 ns	21.797%
procedure(4)	-	11.840 ns	11.840 ns	11.668 ns	21.834%
procedure(5)	-	12.025 ns	12.025 ns	11.611 ns	21.936%
procedure(6)	-	12.210 ns	12.210 ns	12.010 ns	14.662%
procedure(7)	-	12.395 ns	12.395 ns	12.257 ns	21.424%
add argument	-	0.185 ns	0.185 ns	0.221 ns	-
system call	-	$\gg$ procedure()	$\gg$ procedure()	11.944 μs	10.662%
process creation	-	several ms	several ms	1.325 ms	79.623%
thread creation	-	$\ll$ fork()	$\ll$ fork()	22.733 μs	35.574%
process switch	-	$<$ 370 μs	$<$ 370 μs	3.866 μs	16.309%
thread switch	-	$<$ process switch	$<$ process switch	1.200 μs	21.107%
L1 cache	1.48 ns	1.48 ns	2.96 ns	3.7 ns	-
L2 cache	3.7 ns	3.7 ns	7.4 ns	5.55 ns	-
L3 cache	11.1 ns	11.1 ns	22.2 ns	18.5 ns	-
main memory	22.2 ns	22.2 ns	44.4 ns	37.0 ns	-
read BW	29.17GB/s	75%	7.5 to 8.5 GB/s	8.74 GB/s	0.445%
write BW	29.17GB/s	75%	7.5 to 8.5 GB/s	8.62 GB/s	0.851%
page fault	222 μs	7.4 μs	230 μs	230.5 μs	2.76%
local rtt	122 μs	ignorable	122 μs	135.4 μs	1.629%
remote rtt	1.798 ms	ignorable	1.798 ms	2.049 ms	3.077%
local BW	7.29 GB/s	75%	1.88 to 2.13 GB/s	2.97 GB/s	0.927%
remote BW	18.75 MB/s	75%	4.5 to 5.5 MB/s	5.602 MB/s	0.524527%
local setup	183 μs	ignorable	183 μs	182 μs	12.2%
remote setup	3.074 ms	6.29 ms	9.25 ms	8.05 ms	1.19%
local close	-	$\ll$ setup	12.95 μs	12.73 μs	5.15%
remote close	-	$\ll$ setup	12.95 μs	13.33 μs	2.22%
file cache	8 GB	30%	5.6 GB	5.5 GB	1.82%
sequential	60 μs	60 μs	120 μs	130.06 μs	0.15%
random	60 μs	60 μs	120 μs	127.53 μs	0.18%
remote sequential	5 ms	ignorable	5 ms	6.676 ms	1.15%
remote random	5 ms	ignorable	5 ms	6.677 ms	1.18%
add process	60 μs	-30 μs	60 μs	28.35 μs	-

References

[1] Intel Corporation. News fact sheet. the next generation of computing has arrived: Performance to power amazing experiences. 2015.

[2] TP-LINK Technologies Co. Ltd. `http://www.sandisk.com/Assets/docs/X210-sata-ssd-datasheet.pdf`.

[3] Gabriele Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. *Intel Corporation*, 2010.

[4] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual. 2012.

[5] Andrew S Tanenbaum. Modern operating systems. 2009.

[6] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. 1995.

[7] Remzi H Arpaci-Dusseau and Andrea Carol Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books, 2014.

[8] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[9] David A Patterson and John L Hennessy. Computer organization and design. *Morgan Kaufmann*, pages 474–476, 2007.

[10] SanDisk Corporation. `https://developer.apple.com/library/mac/documentation/Performance/Conceptual/ManagingMemory/Articles/AboutMemory.html`.

[11] Alberto Leon-Garcia and Indra Widjaja. *Communication networks*. McGraw-Hill, Inc., 2003.

[12] TP-LINK Technologies Co. Ltd. `http://www.tp-link.us/products/details/cat-9_TL-WR841N.html#specifications`.

[13] Pallapa Venkataram, Sunilkumar S Manvi, and B Sathish Babu. *Communication protocol Engineering*. PHI Learning Pvt. Ltd., 2014.

[14] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.