CSE 221 Project: Benchmark

Zekun Chen, Xiuwen Zheng

June 8, 2018

Contents

1 Introduction

Measuring the performance characteristics of underlying hardware components of an operating system is vital for both building an OS and developing applications. Thinking over methodologies to experimentally determining some characteristics should not only help us review and get a deeper understanding towards basic OS concepts such as system call, context switch and so on.

For this project, we mainly use C++ to conduct experiments and use gcc 7.2.0 (MSVC 19.1 for Windows) for compiling. Binaries of o0, o1, o2 optimization settings are both compiled and after compassion, o1 is used. o1 optimization generates purer assembly code, which conducts better experiment results. The time we estimate to finish the project is 100 hour per person.

2 Machine Description

Table 1: Machine Descriptions.

Machine	WinPC	Lab	Mac
CPU	Intel Core i7 4700HQ 3.40GHz (OC)	Intel Xeon E3-1246 v3 3.50GHz	Intel Core i5 5257U 2.70GHz
Architecture	Haswell(AMD64)	Haswell(AMD64)	Broadwell(AMD64)
Cores	4C8T	4C8T	2C4T
TSC Cycle	0.418 ns	0.280 ns	0.370ns
L1d/i cache	32K+32K	32K+32K	32K+32K
L2 cache	256K	256K	256K
L3 cache	8192K	6144K	3072K
RAM Bus	DDR3L Dual Channel	DDR3 Dual Channel	LPDDR3 Dual Channel
RAM Speed	1866MHz(12-12-12-29)	unknown	1867MHz
RAM Size	16GB	16GB	8GB
Disk1	[SSD]OCZ ARC100 240G SATA3 (ATA8-ACS) 512MB cache	[HDD]unknow 1T	[SSD]Apple SM0128G PCI-E 3.0 x4 512MB cache
Disk2	[SSHD]Seagate FireCuda 2T SATA3 (ACS-3) 64MB cache + 8GB NAND	N/A	N/A
Disk3	[HDD]HGST 1T 7200RPM SATA3 (ATA8-ACS) 32MB cahce	N/A	N/A
Network	Realtek RTL8111 GBE PCI-E 2.0 x1 1000Mbps	Intel I217-LM PCI-E x1 1000Mbps	AX88772A USB2.0 100Mbps
Wireless	Atheros AR5BWB222 PCI-E 2.0 x1 802.11abgn (300Mbps)	N/A	Broadcom BCM15700A2 PCI-E 802.11ac
OS (version)	Windows 10 (17134.81)	CentOS 7 (Linux 3.10)	macOS Sierra(10.12.6)

3 Experiment Environment Preparation

To eliminate or decrease the influence of other user processes, multi-core processors and dynamically adjusted CPU frequency on the measurement results, we made some preparations before the experiments.

3.1 Scheduler and Priority

There three schedule strategies in Linux, CFS, FIFO and Round-Robin. Later two are used for realtime process, which has higher priority than normal process. We uses FIFO scheduler, which allows thread to run as long as it need regardless the limit of time slice. Also, the priorities of the process and thread are both set to the highest, which make the benchmark program more competitive when being scheduling.

For Windows, it is unable to set the schedule strategy. According to MSDN, Windows's schedule strategy is similar to Linux's Round-Robin, but each priority class are scheduled seperately. So we set proccess's priority to "High" and set thread's priority to "Critical", which would have similar effect as the settings we applied on Linux.

3.2 Binding Process to Specific Core

We firstly bind our main thread to a specific core (the last core). It would prevent OS from moving the thread to other core, whose cycle counter might have different value. When counting context switch time, both threads or processes are binded to the same core, so that they cannot run in paralle. To eliminate the influence of other user processes such as undesired context switches and make sure that the our measurement programs could occupy the core for most time, we kill all processes that can be killed except some system processes and set the priority of the benchmark process to be the highest, which is -20 in Linux.

3.3 Keeping Steady CPU frequency

We do not find the switch of EIST on BIOS, so we cannot disable the dynamic frequency scaling of CPU. However, CPU's down throttling is only triggered by OS when thermal and power are not reaching the limit. So given the assumption that OS will not halt CPU when system load is heavy, we can increase the benchmark load to force OS to keep CPU running at maximum frequency.

Another thread is created running an endless empty loop. It is binded to a different core than the benchmark thread, so it won't compete for execution resources. Also, the empty loop doesn't need access to cache so its influence to the benchmark is limited to the minimum.

As for Intel's Turbo-Boost, we keep it enabled on 4700HQ, so it can reach maximum boost frequency at 3.4Ghz, which is similar to E3-1246 v3. Sadly we haven't found the way to disable it on Linux.

3.4 Comparing Different Optimization Settings

When adopting different optimization setting, the complier would optimize code at different levels. To get the most desirable measurement results, we try three different optimization settings, o1, o2 and o3, and analyze each assembly codes respectively. Note that *nop* is inserted for preventing compiler from optimizing the loop.

Gcc 7.2 generates assembly below for pain loop in O0:

```
1 mov eax, [rbp-0x1c]
2 cmp eax, [rbp-0x28]
3 jae 0x401415
4 add dword [rbp-0x1c], 0x1
5 jmp 0x401407
```

5 instructions (4 uOPs after fused) in a loop and each iteration needs about 5.2 cycles to execute. There's 3 loads and 1 store at L1, whose latency is 4 cycles. Some operation might be optimized by CPU, but accessing L1 cache is still a non-negligible overhead.

MSVC 19.1 generates assembly below when optimization disabled:

```
1 mov eax, [rsp+0x30]
2 inc eax
3 mov [rsp+0x30], eax
4 mov eax, [rsp+0x58]
5 cmp dword ptr [rsp+0x30], eax
6 jae 0x1400090e2
7 nop
8 jmp 0x1400090cc
```

About 4.4 cycles needed for each iteration and more instructions executed.

While with O1 optimization, gcc7.2 generates assembly below:

```
1 add ecx, 0x1
2 cmp ebx, ecx
3 jnz 0x400b51
```

Only 3 instructions (2 uOPs after fused) in a loop and only about 0.7 cycles needed for an iteration. MSVC generates similar assembly using *dec* rather than *add*. There's no load or store and data is optimized into register, so OoO and loop-buffer can help to reduce cycles.

We finally choose to run test in O1 mode so that the overhead of loop is totally covered by the actual test code.

3.5 Reference

https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100.aspx

4 CPU, Scheduling, and OS Services

There is a "constant_tsc" in the flags of the CPU. With this feature, the cycle counter queried by *rdtsc* and *rdtscp* will be updated at a fixed frequency independent of the actual operating frequency of the core. Thus, we use *rdtscp* to measure the overhead.

4.1 Measurement overhead

In this part, we measure the overhead of using *rdtscp* to read time stamp counter and measure the overhead of empty loop.

4.1.1 Overhead of empty loop

We run an empty loop for 1,000,000 times and record the time stamps before and after the loop. Since we have turn on the optimization, the loop body is short and has no access to memmory. We also check the assembly and make sure that the loop is not eliminated. An *nop* is added into the loop for MSVC, which can be ignored. This trail is done for 10 times. Note that the empty loop is executed for 1,000,000 times and we just record time stamps for twice, thus the overhead of reading time can be safely ignored.

4.1.2 Overhead of reading time

We put the *rdtscp* instruction in a loop which execute the instruction for 1,000,000 times and record the time stamp before and after the loop respectively. Since *rdtscp* is serialized, no fence is needed and the overhead can be calculated by taking the difference of the whole loop rather than counting one by one. Note that we also check the non-optimized version and no obvious difference can be observed between two results. The overhead of loop is totally covered and can be ignored.

4.1.3 Results

We execute each measurement trial for 10 times and here presents the result of each trail and the standard deviation across multiple trials. The standard deviation of trails

are low which indicates the stability of our experiment. The CentOS costs more cycles than windows OS, however, the difference is not large.

Table 2: Overhead of Empty Loop and Reading Time.

Test	Machine	Trails(time: cycles)										Mean	Std
Loop	Lab	1.52	0.98	0.98	0.98	0.98	0.98	1.10	0.98	0.98	0.98	1.04	0.154
	WinPC	1.01	1.24	1.02	1.01	1.04	1.03	1.02	1.02	1.01	1.03	1.04	0.063
RDTSCP	Lab	30.66	30.75	30.60	31.01	31.08	30.39	30.57	30.59	30.60	31.05	30.73	0.21
	WinPC	32.13	32.30	32.39	32.19	32.45	32.42	32.41	32.53	32.20	32.54	32.36	0.13

4.2 Procedure Call

4.2.1 Methods

In this part, we compare the overheads of procedure call with different numbers of integer arguments. We wrote eight functions with the number of arguments ranging form 0 to 7. There is no instruction inside each function. In each trial, we execute each function for 1,000,000 times and record the time stamp before and after execution. Take the difference of two time stamps and divide it by 1,000,000 to get the average time cycles cost by each function.

4.2.2 Results and analysis

10 trials are conducted and here only shows the result of CentOS because the difference between the two OSes is not obvious (Windows has slightly lower overhead).

Table 3: Overhead of Procedure Call.

Arg	Trails(time cycles)										Mean	Std
0	4.89	4.89	4.89	4.89	4.89	4.89	5.04	4.89	4.89	5.33	4.95	0.13
1	4.89	4.89	5.25	4.89	4.89	4.89	4.89	4.89	4.89	5.25	4.96	0.14
2	4.89	4.89	5.10	6.09	4.89	4.89	5.31	4.89	4.89	4.89	5.07	0.35
3	4.89	4.89	5.24	4.97	4.89	4.89	5.21	5.06	4.89	4.89	4.98	0.12
4	4.89	4.89	4.89	5.32	4.89	4.89	4.89	4.96	6.27	4.89	5.08	0.40
5	6.71	5.87	5.87	6.24	6.03	5.86	5.87	5.87	5.95	5.86	6.01	0.25
6	7.67	6.85	6.84	6.97	6.96	6.85	6.85	6.85	7.00	6.85	6.97	0.23
7	7.82	7.98	7.82	7.82	7.82	9.31	7.82	7.90	7.82	7.82	7.99	0.42

According to calling conversion, Linux uses “System V ABI” while Windows uses “Microsodt x64”. Linux stores first 6 arguments into register while windows stores first 4 arguments, other arguments need to be pushed into stack. Though *push* is heavier than *mov*, there is no obvious penalty can be observed on Windows. Windows seems to be even slightly faster.

It is easy to find out that there is no obvious increase in time cycles spent by functions with argument number from 0 to 3. However, when the number of arguments is increased to 4, we found an obvious increment in time cycles.

This phenomenon can be explained by the CPU's micro-architecture. Haswell has a 4-way decoder so is regarded as a 4-issue architecture, capable to issue 4 uOPs at a time. It also has 4 ports with ALU which is capable to perform *mov*. We perform the benchmark on Skylake and the time remains stable for arguments from 0 to 5, which matches the fact that Skylake is a 5-issue architecture. Though Skylake has only 4 ALUs capable to perform *mov*, pipeline can hide the laterncy.

4.3 System Call

4.3.1 Methods

In this part, we need to measure the cost of a minimal system call.

getpid is selected for Linux because it is an idempotent one. According to the manual, glibc with version before 2.25 will cache the results of getpid, so only the first call by a process will actually trap into the OS. The glibc we compiled with is 2.23, so in order to get the correct result, we fork a child process to perform the system call.

After forking, the parent process will wait for the child process to finish its task. For the child process, it records time stamp before and after calling *getpid*, and returns the difference of the two time stamps to the parent process. The parent process will not continue the next iteration until child process returns, which eliminates the influence of background threads.

As for Windows, the undocumented API *NtDisplayString* is choosen since it seems to have lowest overhead. (https://stackoverflow.com/questions/25047726/windows-system-call) There's no sign of caching so we simplly call it in a loop.

A variable *total_time* is used to maintain the total time cycles spent for all system calls, and each syscall is timed separately and sumed up. It is tested for 100000 times on Windows but 1000 times on Linux, since forking is much time-consuming.

4.3.2 Results and analysis

This trial is repeated for 10 times and the results are recorded as the following.

Table 4: Overhead of System Call.

OS	Trails(time cycles)										Mean	Std
Lab	74.39	73.81	74.18	74.08	73.54	73.37	73.64	74.57	73.29	73.61	73.85	0.39
WinPC	6550.4	6555.5	6561.7	6583.4	6576.4	6554.9	6556.5	6564.2	6595.8	6571.9	6567.1	13.17

To compare the overhead of procedure call and system call, we plot Figure.1 which presents the mean and error bar (which uses standard deviation).

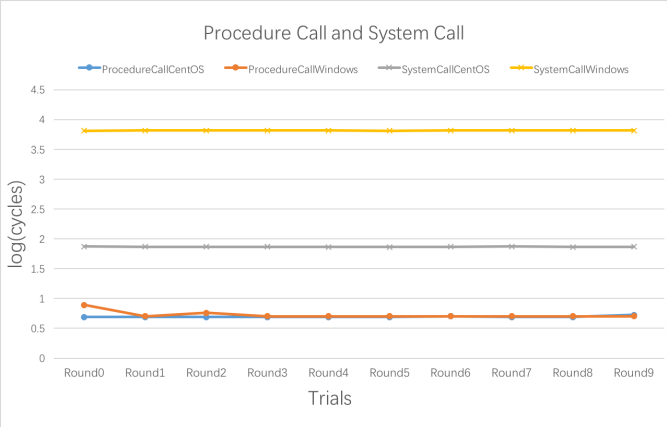

Figure 1: Procedure call and system call.

The overhead of system calls is much higher than that of procedure calls. The reason is that for system calls, it needs to trap into OS and change the privilege mode. For current OS, TLB might not be flushed since it is designed to save the overhead. Every processes share the same kernel space, so kernel space mappings are stored with global flag set while user space mappings are not. When context-switch happened, mappings with global flag set would not be swapped out, so the syscall's overhead is decreased.

It is obvious that Windows cost more time for the syscall than Linux, but that seems to be the difference caused by the API itself. The functionality of two syscall is totally different so they might not be suitable to be compared. Also the patch of KPTI might be the reason, details will be shown in the test of context switch.

4.4 Thread and Process Creation

4.4.1 Methods

It is hard to determine if the creation of a thread or process is synchronized or asynchronized. We choose to measure the time cost by calling the creation syscall as the time of creation. For Linux, *pthread_create* and *fork* are chosen and for Windows, *CreateThread* and *CreateProcess* are chosen.

Note that the main thread will wait until the created thread or process is terminated, so that they will not influence the next iteration of test. For Windows, the flag "CREATE_SUSPENDED" is used so that the created thread won't execute immediately.

4.4.2 Result and analysis

Here presents the results of 10 trails and the error bars.

Table 5: Overhead of process and thread creation.

OS	test	Trails(time cycles)										Mean	Std
Lab	Process	8.81E+04	8.76E+04	8.78E+04	8.78E+04	8.83E+04	8.84E+04	8.71E+04	8.75E+04	8.76E+04	8.76E+04	8.78E+04	363
	Thread	7.72E+03	7.69E+03	7.72E+03	7.89E+03	7.74E+03	7.86E+03	8.16E+03	8.05E+03	7.98E+03	7.83E+03	7.86E+03	144
WinPC	Process	3.71E+07	3.75E+07	3.84E+07	3.82E+07	3.77E+07	3.86E+07	3.77E+07	3.86E+07	3.78E+07	3.84E+07	3.80E+07	4.67E+05
	Thread	8.18E+05	7.54E+05	7.59E+05	7.68E+05	1.78E+06	7.56E+05	7.56E+05	7.22E+05	8.79E+05	7.66E+05	8.76E+05	2.89E+05

In each OS, creating process is much more expensive than creating a thread. The reason is that a process includes not only creating a new thread, but also many other things such as address space and execution state. When a process is created, all of these resources and data structures need to be allocated and initialized, which is extremely costly. Though we uses "fork" to create light-weight process, its overhead is still about 10 times of creating a thread.

As for Windows, creation of thread and process are both significantly higher than CentOS. While on the other machine running the same version of Windows, its speed is about 10 times faster. It might be influenced by anti-virus software or the KPTI patch.

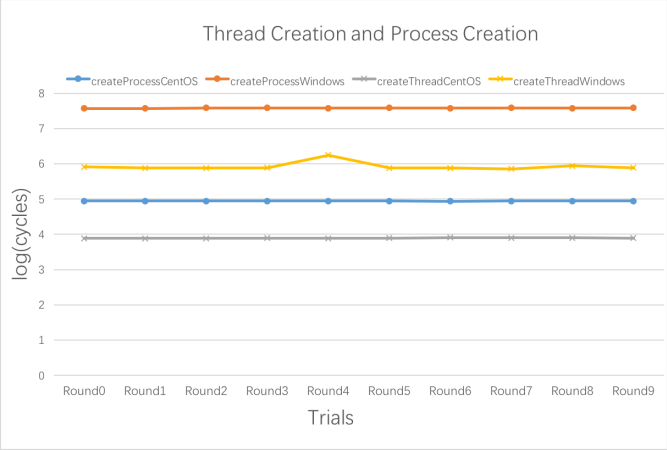

Figure 2: Process and thread creation.

4.5 Context Switch

4.5.1 Methods

The smallest unit for scheduling is a thread, and context-switch happens when OS yield a thread and pick another thread to execute. Especially, when two threads belong to different processes, we can call it a context switch of process.

On Linux, FIFO schedule strategy promises that a thread will not be swapped out for running out of time slice, but there's no such promise on Windows. However, there's little operation needed to record time between context-switches, so it is unlikely to happen. We only need to perform yield manually when needed.

Another concern is that, when we call OS to yield current thread, OS may just treat it as a hint and ignore it. So we need to make sure that a context-switch do happened. Given the fact that TSC on a core is increasing monotonically, we record the timestamp before and after context-switch, and only keep the interleaving timestamps recorded by two threads.

We record the timestamps before and after yielding, and sort data from both threads (processes). The difference between two adjacent timestamps from different threads (processes) are considered a successful context-switch, and they are summed to calculate the average value.

4.5.2 Results and analysis

Here presents the results of 10 trails and the error bars.

Table 6: Overhead of context switch.

OS	test	Trails(time cycles)										Mean	Std
Lab	Process	2955.40	2986.52	2979.64	2987.47	3029.07	3031.61	2987.78	2997.94	2932.13	3025.67	2991.32	29.00
	Thread	1583.09	1584.61	1584.19	1588.81	1586.20	1589.16	1595.82	1585.10	1588.29	1631.73	1591.70	13.14
WinPC	Process	2913.09	2908.48	2895.54	2907.02	2955.66	2893.11	2904.52	2918.73	2895.18	2901.50	2909.28	16.49
	Thread	1670.38	1670.45	1672.70	1672.57	1675.64	1682.22	1678.33	2051.14	1676.78	1705.82	1715.60	107.05

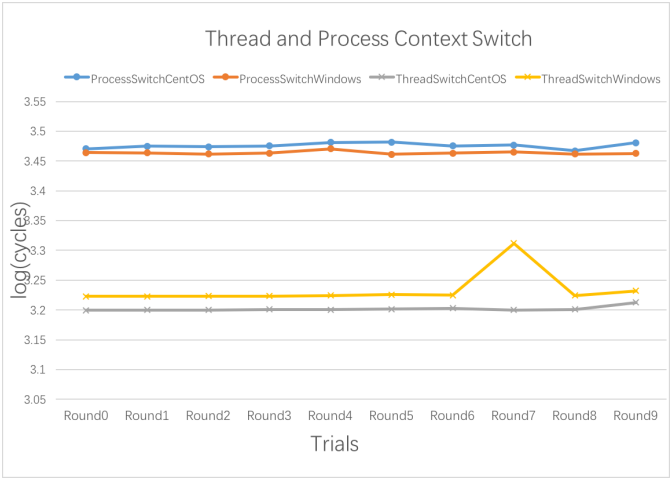

Figure 3: Process and thread context switch.

From the figure, we can get two main findings. First, context switch of thread cost less than that of process. Second, the context switch in Windows 10 is slightly more expensive than that in CentOS 7.

When a thread context switch happens, it saves current thread's hardware state (PC, SP and so on) stored in hardware registers into its stack and reload the hardware registers from the value stored in another thread's stack. Also, OS may includes some extra

overhead for maintaining the schedule queue. When two threads belong to different processes, the context switch between two processes would cost much more mainly because of TLB flushing since they uses different virtual memory space.

It worth mention that the penalty cause by TLB flush is highly related to the memory accessing needed for the thread. In our test, additional memory access is needed (store timestamp to a vector), but remains small. So in actual environment, the overhead of process's context switch would be higher.

The reason why Windows cost slightly more than CentOS may be the reason for KPTI patch, which is designed for fixing the "Meltdown" bug discovered early this year. "Kernel page-table isolation"(KPTI) is a common accepted solution for anti-"Meltdown". According to its description ([1]), kernel space mappings and user space mappings are stored seperately for each process so that accessing kernel data can be blocked by MMU. Its negative effect is that a syscall will require a TLB flush, leading to larger overhead.

For new CPUs, techinics like PCID (introduced in Westmere) can help reduce the overhead of context switch by providing more fine-grained control of TLB flushing. It allows multiple memory space mappings kept in TLB with additional identifier, so TLB miss can be reduced when context switched. It may lead to some problem like TLB shootdown, so Linux kernel partially add PCID support until in version 4.14. CentOS's "3.10" kernel would definitely miss this benefit but the newest Windows 10 should have add support for it. This feature can help reduce the impact of KPTI.

We also perform the test on another laptop with Skylake CPU and same version of Windows. We observer slightly lower overhead on process-switch, around 2780 cycles. It matches the announcement that Skylake is less affected by the patch than older CPU (like Haswell). [2]

Reference:

[1]https://en.wikipedia.org/wiki/Kernel_page-table_isolation
[2] https://cloudblogs.microsoft.com/microsoftsecure/2018/01/09/understanding-the-performance-impact-of-spectre-and-meltdown-mitigations-on-windows-systems/

5 Memory

In this part, we did experiments to measure RAM access time, RAM bandwidth and page fault service time. The memory components have their own clock speed (even CPU's cache can has different clock frequency from the core), so there' no need to measure the speed and latency basing on CPU's frequency. We choose to use *high_resolution_clock* in C++11's chrono library to measure time.

Measurement is currently only performed on Windows platform.

5.1 RAM access time

5.1.1 Methods

In this part, we measured the latency for individual integer accesses to main memory and CPU's three level caches.

Hardware prefetches are quite smart in these new architectures, so there's no way to promise that each load will definitely hit or miss the desire level of memory. But with proper access pattern, we will still be able to see a significant difference on access time when accessing different size of memory. Also, we can still get an approximate so-called "back-to-back" access latency from statistical data.

Since the CPU we tested are all multi-issue OoO processors, another concern is that multiple load instructions may be on-fly at the same time, hiding the actual latency. Speculative execution also make loop useless to stop these paralism. In order to solve this, we fill each integer in the memory with the index of the next integer it should load, and follows the index loaded rather than simply add index with a fixed stride when performing test. The pseudo code is presented as the following. Note that when the index exceed the array size, we do not simply mod the array size but add 1 to it before mod to make the whole piece of memory a linked list. If we simple mod array size, we would only visit $array\_size/stride\_size$ integers no matter how many accesses there are. And larger stride will on the contrary decrease the region we access, so the cachelines needed may be just fit into cache, which leads to a wrong measurement.

We also followed the measurement method used by lmbench paper. In this method, there are two parameters which are array size and array stride size. Different array sizes are chosen to capture the access time for different memory hierarchy. The array size ranges from 4KB to 256MB, covers all three level cache and main memory. Different stride sized are chosen to capture the effect of cache line. The stride sizes are 32byte, 128byte, 1KB and 8KB respectively.

```
1 for (uint32_t i = 0; i < interger_num; i++)
2 {
3     const auto desire = i + step;
4     if (desire >= interger_num)
5         mem[i] = (desire + 1) % interger_num;
6     else
7         mem[i] = desire;
8 }
```

5.1.2 Results and analysis

We examine the time of 1,000,000 access to integers. The experiment is conducted for 10 times and the average time is calculated and reported in the following figure.

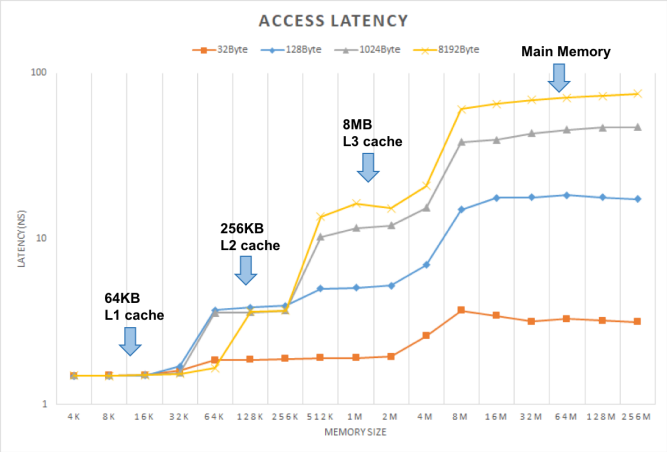

Figure 4: Cache and memory latency.

The result shows that when the stride size is 32 Byte, the latency is much smaller than that of other strides and the latency does not vary much with the increase of array size. The reason is that 32 Byte is within the cacheline size, thus the hit rate is extremely high. As the figure shows, small stride sizes have smaller latency because small stride sizes have higher spatial locality and also suitable for hardware prefetcher. For 8192 Byte stride, it might meets some 4k-aliasing problem which increase access latency for normal access.

The result of 1024 Bytes stride size meets our expectation best. It does not suffer severe effect caused by prefetching since 1k stride is rare and too aggressive for the hardware prefetcher. The latency of Haswell's L3 cache is around 34 cycles (around 10ns), which also matches the result of 1k stride. However the latency of RAM is smaller than expected. Here presents the average latency of 10 experiments when the stride size is 1024 Bytes.

Table 7: Cache and memory latency.

Array Size	Latency(ns)	Array Size	Latency(ns)
4KB	1.5004	2MB	12.0314
8KB	1.4957	4MB	15.3340
16KB	1.5167	8MB	38.4016
32KB	1.5496	16MB	39.4883
64KB	3.5926	32MB	43.2856
128KB	3.5958	64MB	45.4229
256KB	3.7013	128MB	47.0382
512KB	10.3029	256MB	47.2172
1MB	11.5920		

5.2 RAM bandwidth

5.2.1 Methods

In this part, we measured the bandwidth for the main memory. Different from measuring latency of cache, we don't need to bypass the cache influence. With adequate memory sequentially accessed, bandwidth between CPU and RAM will become the bottleneck and cache will no help. We choose 128MB as the size of the memory being tested. According to the previous measurement, it is large enough to show the real bandwidth of RAM.

All the platform we test are using DDR3 two-channel RAM, ann are all running at about 1866MHz, so they should be comparable. Memory timing do influences latency but will not influence the throughput.

The theoretical bandwidth of DDR3 can be calculated as follows: bandwidth = RAMbus_frequency X 2 X RAMbus_width. For DDR3-1866(2 channel), the maximum bandwidth should be 933MHz X 2 X (64bit X 2) = 29.156GB/s.

To achieve maximum bandwidth, we need a efficient way to copy large data from/to RAM. According to Intel's optimization manual, CPU since Ivy-bridge support *Enhanced REP MOVSB*, which generally have higher throughput compared with 128bit SSE instructions. However, since optimization is limited, compiler no promise to generate such pattern. We choose to use 256bit AVX2 instructions to perform bulk load/store. There's additional benefits that we can perform non-temporal load/store to minimize the cache pollution.

For read bandwidth measurement, we calculate the sum of the value to avoid being elimited by compiler. The throughput of AVX2's add is much higher than the memory bandwidth, so it is unlikely to slow down the bandwidth.

5.2.2 Results and analysis

Table 8: RAM bandwidth.

Test	Trails(GB/s)										Mean	Std
Load	20.76	20.57	20.75	20.80	20.63	20.82	20.51	20.69	20.80	20.55	20.69	0.11
Store	18.54	18.47	18.57	18.56	18.55	18.45	18.60	18.63	18.42	18.60	18.54	0.066

Load bandwidth is generally higher than store bandwidth. However, when measuring another laptop with similar environment (DDR3-1866 dual channel with Skylake CPU), we have a store bandwidth of 23.3GB/s and a load bandwidth of 19.5GB/s. Since they all lower than the theoretical bandwidth, this measurement might be limited by single core's execution units. Also, the micro-architecture can also influence the actual bandwidth.

5.3 Page fault service time

5.3.1 Method

Page fault occurs when a page requested does not exists in physical memory. It is handled by OS and loaded from back storage, which is usually the disk. Additionally, Win10 and macOS both have the techinic called “memory comparison ”, which uses compressed memory as back storage to reduce latency.

The service time of page fault is highly dependent with back storage’s access time. Since accessing file is explicit required, we use file mapping to measure the service time. We performed the measurement on both SSD and HDD, which has significant difference on random accessing latency. According to the specification, the HGST 7200RPM HDD has a average seek time of 12ms ad a average latency of 4.2ms. While OCZ’s ARC100 can reach 75000 IOPS for random 4k reading, which is 13us for each operation.

To avoid file caching, we generate access index randomly among 0 and 4M. Also, to prevent OS’s prefetch’s influence on page fault, we uses 16M as the step.

5.3.2 Results and analysis

Table 9: Page fault service time.

disk	Trails(ns)										Mean	Std
SSD	28020.74	21721.21	19245.16	20284.99	17252.86	17727.23	13045.03	8917.27	11066.81	10366.29	16764.76	5391.90
HDD	6397613	6022618	6272783	6101370	6053182	6060459	6053631	5977023	5928504	6025881	6089306	141109

For SSD, the service time is at first 28us and stabled at around 11us. SSD’s cache may help reduce the latency. While for HDD, the service time is around 60ms, which is much longer than SSD’s.

5.3.3 Comparison

L1/L2 cache’s latency is at a level of 10^0 ns, L3 cache’s latency is at 10^1 ns, RAM is approaching 10^2 ns, accessing SSD is at 10^4 ns, access HDD is at 10^6 ns.

As for the read time per bytes, we get around 1ns/bytes for RAM (assuming each load loads a cache line), 2.5ns/bytes for SSD and 1500ns/bytes for HDD.

In fact, reading from disk transfer at least 4K at a time, the huge data covers its latency, so the difference between memory and disk’s service time remains small. In fact, RAM has much lower latency, and that’s why user experience stuttering when OS uses disk (especially HDD) as virtual memory’s back storage.

6 Network

In this section, we measure the Round trip time, Peak bandwidth and Connection overhead of network. In each part, we compare both remote and loopback interfaces to deduce about baseline network performance and the overhead of OS software.

Since the WinPC does not support 802.11ac wireless, its throughput might be the bottleneck while performing remote test. We uses another laptop running the same Windows to perform the test with the mac. That machine has a Intel AC8260 wireless card, with Intel core-m 6Y75 Skylake processor running at 2.9Ghz. Its memory is 2x4GB DDR3L 1866MHz, similar to other platform. We use Mac as client machine.

6.1 Round Trip Time

Round Trip time is the duration from when a browser sends a request to when it receives a response from a server. This is an important factor when determining the web application performance if there are many request/reply pairs being sent one after another ([1]).

6.1.1 Method

As Figure 5 shows, we measure RTT by examining time that it takes from the SYN to the SYN/ACK. We first created client and server, both of which conforming to TCP for exchanging data. The server is always in listen status, the client first tries to establish connection to server. Once the connection is established successfully, the client would send data of 64 bytes to server and then server sends back an acknowledgement message which is also 64 bytes and within one packet. We tried to conducted the send and receive procedures for 1,000,000 times while limiting the whole time within 5s.

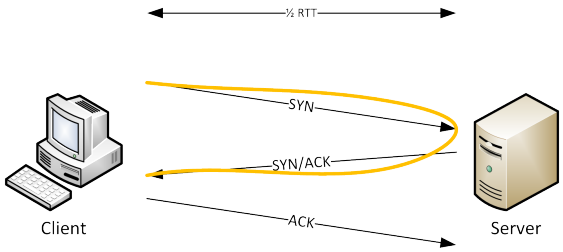

Figure 5: RTT of Network.

6.1.2 Results and analysis

We conducted the trails for 5 times both for loopback and remote interfaces and compare the mean value with the time measured by using *ping* command. The remote machine

and test machine are in the same LAN. Here shows the result.

We also do *ping* command for 1000 times and Table 11 shows the result. By comparison, our experiment result does not have much difference from the RTT measured by Ping. The reason may be that the remote and local machines are in the same LAN and thus there is not server throttling and network congestion.

It is easy to find out that *ping* command has unstable result. The wireless band is shared between multiple users, so they will influence each other and lead to big jitters. Our tests only recorded average time after millions of package transportation, so the result is much more stable. Also, the router may give higher privilege for heavy-load connections, that may leads to our non-stopping test to be slightly faster than *ping* command.

Table 10: RTT for Loopback and Remote Interface.

	Trails(time:ms					Mean(ms)	Std(ms)
Remote	3.8517	3.8753	3.9111	3.9785	3.9113	3.9056	0.0391
Loopback	0.0369	0.0360	0.0425	0.0416	0.0393	0.0393	0.0023

Table 11: Ping for Loopback and Remote Interface.

	Min(ms)	Avg(ms)	Max(ms)	Std(ms)
Remote	2.728	5.948	178.971	12.667
Loopback	0.035	0.075	0.257	0.026

6.2 Peak bandwidth

6.2.1 Method

For peak bandwidth, we measured both download bandwidth and upload bandwidth. In order to achieve the highest speed, we let the client send a large data (we set the size of data as 4MB) and server send back 64 Byte data to measure the upload bandwidth. Equally, we let the client send out 64 Byte data and let server sends back 4MB data to measure download bandwidth.

6.2.2 Results and Analysis

Here shows the result of upload bandwidth and download bandwidth for loopback and remote interface. LAN does not impose different limit on upload and download speed. But there is still some difference between upload bandwidth and download bandwidth. In fact most network card are optimized for downloading, so the buffer size may differ for uplink and downlink.

Also, the bandwidth for loopback is much higher than remote bandwidth, and even exceed PCIE bandwidth. In fact the system is likely to perform extra optimizations for loopback since it's common for different application to communicate via loopback connections, so the packages may not even go through the hardware.

Table 12: Bandwidth for Loopback and Remote Interface.

	Interface	Trails(MB/s)					Mean(MB/s)	Std(MB/s)
Download	Remote	8.94	9.08	6.63	8.67	9.24	8.38	0.96
	Loopback	553.40	632.89	657.18	677.77	681.86	636.88	46.95
Upload	Remote	11.65	13.61	11.93	11.82	11.49	12.05	0.77
	Loopback	516.53	619.91	664.62	671.42	681.15	624.12	60.83

6.3 Connection overhead

6.3.1 Methods

We measure both the setup time and tear-down in the client. We let client try to establish connection and measure the time before and after calling the *connect* function, and then close the connection and also measures the time before and after calling *close* function. We repeat this procedure for 1000 times and after closing connection per time, we will let the thread sleep for 10 ms to prevent the situation that the connection is not closed completely after executing close. The server runs at single thread, so delay closing can hugely influence the setup time of next iteration.

6.3.2 Results and analysis

Here shows the result of setup and teardown for loopback and remote interface. To establish a connection between server and client, there is a three-way hand-shack between client and server, as Figure 5 shows, so the setup time should be around 1.5 times to RTT, however, there is a delay between SYN/ACK and ACK which impose a little more time. From 6.1, we know that the average RTT for remote interface is 3.9056ms, thus the setup time should be a little higher than $1.5 * 3.9056 = 5.8584$ ms, which matches with our exam result here.

For teardown, there is usually a four-way handshake, but according to specification, function *close* can return before the connection actually closed, which means the client may not wait until received ACK from server. The teardown time is much less than setup, which result matches with our expectations.

Reference:

[1]https://blog.packet-foo.com/2014/07/determining-tcp-initial-round-trip-time/
[2] https://www.incapsula.com/cdn-guide/glossary/round-trip-time-rtt.html

Table 13: Setup and Teardown for Loopback and Remote Interface.

	Interface	Trails(ms)					Mean(ms)	Std(ms)
Setup	Remote	5.79	6.08	8.38	8.55	7.35	7.23	1.14
	Loopback	0.26	0.19	0.23	0.22	0.19	0.22	0.02
teardown	Remote	0.08	0.08	0.08	0.07	0.08	0.08	0.00
	Loopback	0.02	0.02	0.02	0.02	0.02	0.02	0.00

7 File System

7.1 Size of file cache

To improve the performance of accessing file in the slow disk, OS will utilize some part of the memory as file cache. It is transparent to high-level applications. The size of file cache is dynamic and OS may drop any cache at any time.

7.1.1 Method

The size of file cache will never exceed available physical memory size (regardless of memory compression), so we create a big file of 10GB for testing. We also allocate lots of memory with around 5GB free space left to limit the size of file cache. The file is located at HDD so the difference between cached and uncached file region can be more significant,

To test the cache size, we read the same file repeatedly within different size of region. The test is performed 5 times and the time to read 1 MB data is recorded for comparing.

7.1.2 Results and analysis

As it shows in the table 14 and figure 6, the first test is significantly faster than later tests. The size of cache is around 2GB since the speed of access is much faster than bigger size.

However, the later tests are much slower even when accessing small region of file. It seems that the file cache is removed since it's too large to fit in the memory. The read time for smallest file size is much slower, this might be the negative result for clearing cache.

Since later test show little about cache, we only plot the figure with the first test.

7.2 File read time

The file read time is associate with disk performance and the file system performance. File system needs to locate and read data each time we perform IO operations, which is

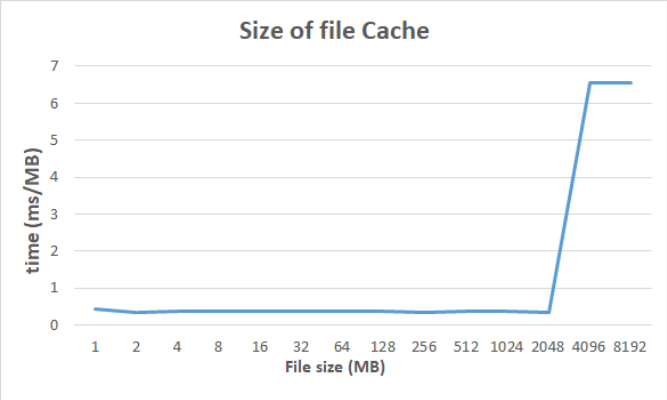

Figure 6: File cache size.

Table 14: Average read I/O time of file with different size.

File Size(MB)	Trails(time:ms/MB)					Mean	Std
1	0.4469	40.4144	38.6277	35.8950	40.4332	31.1635	14.1015
2	0.3391	6.4772	6.3250	6.2916	6.2234	5.1313	2.1886
4	0.3746	6.5707	6.6915	6.6646	6.6507	5.3904	2.2897
8	0.3689	6.5151	6.5018	6.5362	6.5432	5.2930	2.2476
16	0.3851	6.8055	6.8041	6.8674	6.8423	5.5409	2.3534
32	0.3775	6.7066	6.7062	6.6882	6.6660	5.4289	2.3057
64	0.3623	6.7224	6.6720	6.6772	6.7325	5.4333	2.3147
128	0.3845	7.1388	7.0956	7.0793	7.0881	5.7573	2.4524
256	0.3567	6.2322	6.2160	6.2048	6.2086	5.0437	2.1393
512	0.3635	5.7451	5.7637	5.7582	5.7914	4.6844	1.9723
1024	0.3612	6.6601	6.6725	6.6976	6.6771	5.4137	2.3062
2048	0.3575	6.5028	6.5000	6.5240	6.5151	5.2799	2.2468
4096	6.5585	6.5172	6.4981	6.4951	6.5054	6.5149	0.0211
8192	6.5412	6.5472	6.5340	6.5295	6.5376	6.5379	0.0055

the main part of overhead for IO operations.

7.2.1 Method

To measure the accurate overhead of file system, the effect of cache should be removed. Both of three platform provide flag to disable caching while open files. Note that the cache in the disk might still be available since we only perform read operations.

Each file is accessed in a 4K block at a time, with two different strategy: one for sequential read and one for randomly read.

We perform the test on SSD so the effect of file fragment can be reduced. However, SSD still has to lookup the actual file block via FTL, so reading one block at a time is still slower than read a lot of blocks, even when we are reading sequentially.

7.2.2 Results and analysis

As it shows in the table 15 and figure 7, random access is 20 40 us slower than sequential read. The extra overhead might be the lookup of MFT data (inode data in *nix file system) and the query for SSD's FTL.

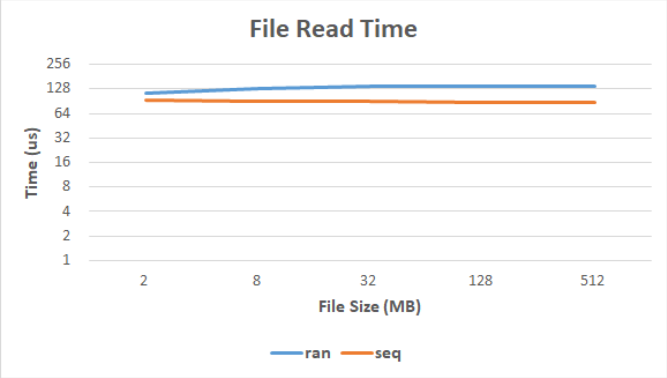

Figure 7: File read time.

Randomly accessing a 2MB region of the file is faster, which might indicate that the detail layout of file is cached by SSD.

7.3 Remote file read time

Remote file access is based on network file system which uses network data packages to transfer data. Network accessing has significantly larger latency, so the read time for remote file system will definitely be larger than that of local file system.

Table 15: File read time for random access and sequential access.

	File Size	Trails(time:us)					Mean (us)	Std (us)
Random	2MB	110.71	115.28	115.79	110.55	110.02	112.47	2.52
	8MB	133.29	128.03	127.22	132.36	131.05	130.39	2.38
	32MB	141.34	138.33	141.51	134.15	140.11	139.09	2.72
	128MB	140.70	136.22	134.17	139.95	137.39	137.69	2.40
	512MB	134.15	139.50	138.40	134.50	140.10	137.33	2.51
Sequential	2MB	92.94	95.79	91.02	92.54	95.14	93.49	1.75
	8MB	91.11	89.82	90.77	90.35	88.94	90.20	0.76
	32MB	91.63	90.33	87.25	91.74	91.44	90.48	1.69
	128MB	85.88	86.97	88.96	86.25	85.00	86.61	1.33
	512MB	89.10	85.69	85.04	88.96	90.67	87.89	2.16

7.3.1 Method

The test is performed on Windows, so SMB is choose to be the file system protocol. SMB does not offer much configuration, but we manually turn off the client cache via registry. Also, we open the file with no-caching flag, which is the same as test2. The test should be able to bypass both server side and client side caching effect.

7.3.2 Results and analysis

As it shows in the table 16, the performance of remote file reading is much more unstable and much slower than that of local file access.

A significant difference is that sequential access and random access has similar performance, which means that most of overhead lies on network transportation, so the difference between random and sequential access are covered.

With the results of previous tests of network, the RTT between two local computer is around 3ms, which contributes most part of the remote file read time.

7.4 Contention

File system need to handle multiple IO operation at the same time with promise for data consistency. So there might be some contention overhead even when multiple threads are all performing read-only operations. Also, the AHCI protocol, which is widely used by SATA disks, has a limited request queue, so too few IO requests might not fully utilize disk.

Table 16: Remote file read time for random access and sequential access.

	File Size	Trails(time:us)						Mean (us)	Std (us)
Random	2MB	3005	1893	1744	2866	2121	2085	2225	365
	8MB	5551	1790	1907	4639	4576	2155	4202	3701
	32MB	4173	3475	1715	4507	2076	2079	3156	938
	128MB	5822	3416	1962	4466	3056	1903	3491	1437
	512MB	7362	1983	3126	4567	4798	1850	3534	1530
Sequential	2MB	1954	4795	1736	3028	7715	4681	3271	1751
	8MB	2303	4412	1842	3127	3537	4529	2950	885
	32MB	2945	4152	1576	3719	1830	4550	3021	937
	128MB	2996	1678	3175	3863	1849	4471	2755	845
	512MB	3304	1632	1576	5783	3848	5498	3108	1371

7.4.1 Method

The test is based on File Read Test, with multiple thread accessing different region of the same file at the same time. The test only read data sequentially.

We test 2,4,16,64 thread, where 16 and 64 have exceed CPU core count. In fact, OS is likely to block current thread while performing IO requests, leaving the opportunity for other thread to execution, so the CPU won't be block by the slow IO accessing. So more thread than CPU core is affordable. The AHCI protocol has a queue of size 64, which is also the reason for us to test 64 threads' contention.

7.4.2 Results and analysis

As it shows in the following table and figure, cost for accessing a page is growing with more threads executing together. So multi-thread will definitely lead to extra overhead. But the growing speed is slower than thread count, so the overhead might partially covered by the file system and disk protocol. In another words, multi-thread with IO can improve performance.

When thread number reaches 64, the time needed is almost 4 times of that of 16 threads. They all exceed the CPU core count, so the Os kernel might not be able to cover the extra overhead with thread scheduling.

FileSize	Threads	Trails(time:us)					Mean(us)	Std(us)
2MB	2	26.6	26.375	27.415	26.7	27.245	26.865	0.395
	4	44.885	44.745	43.555	40.91	39.325	42.685	2.205
	16	59.22	60.2	63.64	58.96	59.63	60.33	1.705
	64	208.84	213.15	214.87	219.01	213.525	213.88	3.265
4MB	2	27.03	26.38	27.34	26.11	25.96	26.56	0.53
	4	41.61	41.96	45.56	41.34	41.92	42.48	1.56
	16	58.56	59.78	59.62	56.41	59.93	58.86	1.31
	64	211.52	212.39	213.89	212.31	213.32	212.69	0.83
8MB	2	27.14	26.48	26.24	26.38	26.08	26.46	0.36
	4	39.7	39.98	40.6	41.22	42.88	40.88	1.14
	16	58.92	59.84	59.84	58.84	59.5	59.38	0.44
	64	216.14	215.16	217.3	215.06	214.74	215.68	0.94
16MB	2	26.44	27.52	26.88	26.64	26.4	26.76	0.4
	4	42.84	40.24	44.12	41.04	41.36	41.92	1.4
	16	59.16	60.88	60	59	59.92	59.8	0.68
	64	215.88	214.52	214.44	209.88	213.36	213.6	2.04
32MB	2	26.48	26.64	26.56	26.4	26.64	26.56	0.08
	4	45.04	41.28	47.44	45.2	44.56	44.64	2
	16	58.88	60.32	60.32	59.52	60.24	59.84	0.56
	64	213.84	215.44	213.92	210	219.28	214.48	2.96

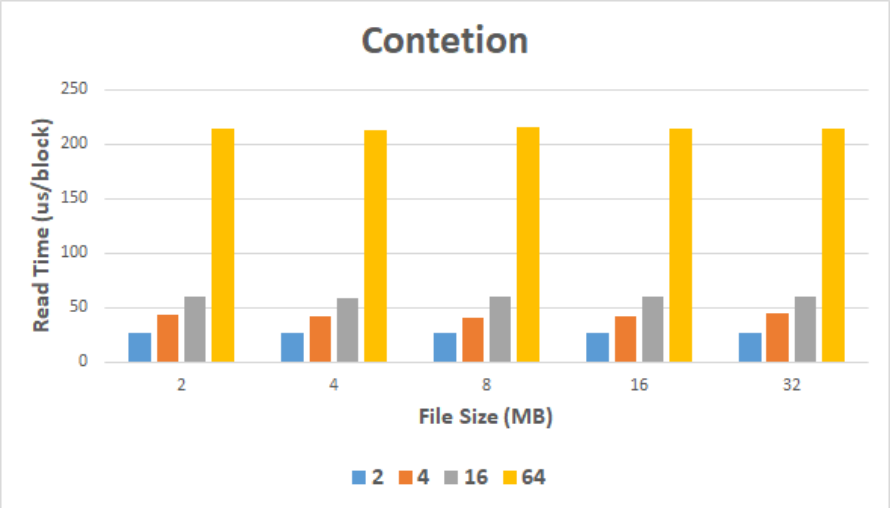

Figure 8: Contention.

7.5 Conclusion

For most tests, no much difference can be seen with different size of file. The reason is that SSD don't need to seek the track, so accessing any region has almost identical response time. This also explains why random accessing has no much difference than sequential access.