CSE 221 Project Report, Fall 2020

Cunji Song
scunji@eng.ucsd.edu

Zongheng Cao
zocao@eng.ucsd.edu

Fan Jin
f1jin@eng.ucsd.edu

1 Introduction

In this project, we measured the performance of AWS EC2. The machine is a hardware virtual machine that provides Linux kernel 4.14 tuned for optimal performance on Amazon EC2, system 219, GCC 7.3. As its virtualization property, the hardware described in the EC2 configuration pages may not reflect its actual performance. For example, the virtualized general purpose SSD in EC2 might skew our measurement. We should consider the variant when evaluating its performance.

We tested the performance of 4 distinct aspects: 1. CPU, Scheduling, and OS Services 2. Memory 3. Network 4. File System. The workload distribution is shown in the figure below. We spend about 40 hours separately and 120 hours in total on the project.

Cunji Song	Fan Jin	Zongheng Cao
Memory 1 File System	Memory 3 Network CPU 1-3	Memory 2 CPU 4-5

2 Machine Description

2.1 Processor

- Model : Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
- Cycle time: 0.4ns
- L1 cache: 32k
- L2 cache: 256k
- L3 cache: 30720k
- Instruction Set: 32-bit, 64-bit

2.2 Memory bus

- Speed: 2400 MHz

2.3 I/O bus

- Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
- ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
- IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
- Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 01)
- VGA compatible controller: Cirrus Logic GD 5446
- Unassigned class [ff80]: XenSource, Inc. Xen Platform Device (rev 01)

2.4 Ram

- Size: 1009132 KB

2.5 Disk

- Capacity: 8G
- RPM: 15000
- Timing cached reads: 21742 MB in 1.99 seconds = 10936.58 MB/sec
- Timing buffered disk reads: 246 MB in 3.03 seconds = 81.14 MB/sec

2.6 Network

- Speed: Elastic but guarentee to be up to 25 Gbps

2.7 operating system

- Version: Linux version 4.14.193-113.317.amzn1.x86 64

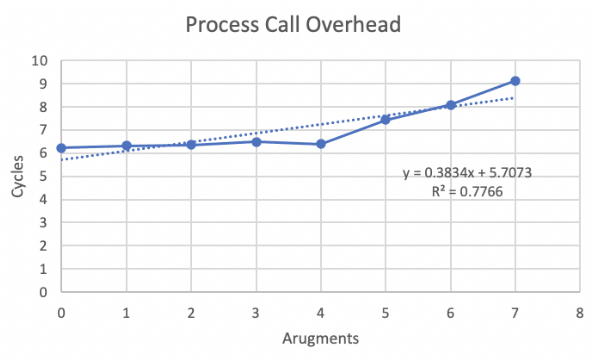


Figure 1: process call vs arguments

3 CPU

3.1 Measurement overhead

Methodology: We use a loop to many iterations. The time is measured using the CPU cycle register (rdtsc). Then, we use the sleep call to sleep for a long time (100 seconds) to get cycles per second. We wait for so long because the sleep system call is not precise. Depending on process scheduling, it can deviate from the specified duration. With a long duration, the error is distributed to every second, improving precision.

Prediction: The virtual CPU runs at 2.40 GHz. So, we expect 2.40G cycles per second.

Result: The CPU cycles per second is 2,399,954,581, very close to 2.40GHz. The measurement overhead is 68 cycles on average, which is 28.3 ns.

3.2 Procedure call overhead

Methodology: Measure the procedure call time for different number of arguments. The more arguemnts, the larger activation record on the stack.

Prediction: The overhead goes linear with the number of arguments with a positive intercept.

Result:

Arguments	Average Call Time (cycles)
0	6.228435
1	6.317052
2	6.349563
3	6.475797
4	6.392004
5	7.435056
6	8.076144
7	9.119760

Analysis: Running linear regression (See Fig 1), the slope is 0.3834 and the intercept is 5.7075. This means an additional integer parameter will bring 0.38 more cycles of overhead.

3.3 System call overhead

Methodology: We use the GetPID system call because it has the simplest kernel implementation.

Prediction: A system call should take much more cycles than a user level call.

Result: It takes 12351 cycles on average (3μs), about 10,000 times as a user level process call!

3.4 Task creation time

3.4.1 the creation time of process.

Methodology: We use fork to create a new process, the parent process will give CPU to child process, then we let the child process to end instantly, then the parent process call wait() to collect child process. Count the elapsed time.

Prediction: There are lots of steps that the OS to allocate a new process, including assign a unique process id to the child process, allocate resources and space to the child process, add new entry to the process control block, and set up appropriate linkage. Given that context switches of this sort are on the order of 20 microseconds and a system call is on the order of 5 microseconds, by referencing some of the cases in lmbench paper, we assume the result to be 1ms.

Result: We account the cycles of 10,000 process creation. Use the average of the 5 experiments, the result is 222832.4 CPU cycles, which roughly equals to 0.534ms

	Process Creation Time (cycles)
1	226,651.43
2	244,925.98
3	200,279.17
4	219,301.09
5	223,006.28

Analysis: the result shows that even a relatively simple task is very expensive and is measured in milliseconds while most of the other operations we consider are measured in microseconds.

3.4.2 the creation time of thread.

Methodology: It is mostly very similar to what we have used to test the process creation overhead. However, we use pthread_create() to create a thread, and pthread_join() to force thread to switch. And in that thread, we simply call pthread_exit() to return.

Prediction: since a new thread is much more high-light then a new process, because all the created threads share the same memory resource. We assume that the overhead to be 1/10 of process, i.e., 100μs.

Result: We repeated the test for 10000 times and calculate the average of 5 experiments. The average is 35795.8 CPU cycles, which roughly equals to 85.89μs.

	Thread Creation Time (cycles)
1	36,750.39
2	34,446.72
3	37,438.60
4	36,152.88
5	34,193.85

Analysis: the result shows that the overhead of thread creation is roughly 1/10 of process creation, which fulfill our prediction. So, in real application, if we could use thread to solve concurrency problem, we should not use process.

3.5 Context switch time

3.5.1 process switch time

Methodology: Context switch time is defined here as the time needed to save the state of one process and restore the state of another process. Here, we let the parent process to fork a child process, and then start clock, the parent process keep waiting to collect the child process, and receive a num passed by son process though pipe, this number is the close time count by child process once it get the control.

Prediction: in this process, there will be a switch from user mode to kernel mode and change the address space from the parent's one to the child's one, getting from the previous step, a system call is about 6000 CPU cycles, equals to 14μs. Reference from lmbench paper, the switch time for a 32KB process is 50μs. We predict the switch time to be 64μs, 26000 cycles.

Result: We repeated the test for 10000 times and calculate the average of 5 experiments. The result is twice as our expectation: 124μs, 51771.32 cycles, which may result from the network penalty of the amazon virtual machine.

	Process Switch Time (cycles)
1	51,816.93
2	49,974.32
3	48,806.35
4	54,308.02
5	53,952.38

3.5.2 thread switch time

Methodology: the method we use is very similar to the process switch, by modifying some production and switching functions.

Prediction: the process creation time is roughly 8 times of the thread creation from the previous experiment, so, here, we predict the overhead of thread switch is 6000 CPU cycles.

Result: We repeated the test for 10000 times and calculate the average of 5 experiments. The average cycle is 6268.2, equals to 14.8μs.

	Thread Switch Time (cycles)
1	6,056.00
2	6,284.62
3	6,453.68
4	6,185.99
5	6,363.04

Analysis: Almost the exactly precise result as out prediction. However, compared to the document online, usually, thread context switch time is much faster, about 50-60 times faster than the process's one. But in our case, the ratio is only 8, we think the thread switch might suffer from the unique characteristics of AWS virtual machine.

4 Memory

4.1 Memory bandwidth

Methodology: By definition, bandwidth is the amount of data transferred in a given time [2]. So, we measure how much time it takes to read or write some size of data in memory. The time is measure in CPU cycles. To eliminate the influence of caching, the stride value should be larger than the cache size to avoid any locality. We take a large prime number (34563349) as the stride value to simulate random access. To eliminate the influence of beanching, we iterate 1 million times in a loop, which is enough to train the branch predictor so that misprediction penalty is minimal on average.

Prediction: The AWS documentation does not disclose the physical bandwidth of the RAM. Public data of DDR3 RAM indicates a peak bandwidth of 6 GB/s.

Result: Given previous measurement of 2,399,954,581 CPU cycles per second, we get the results. The read bandwidth is about 3 times as large as the write bandwidth.

Size	Read Time (cycles)	Write Time (cycles)
10K	9,003	23,331
100K	113,277	315,393
1M	890,403	3,035,247
10M	9,788,727	35,706,063
100M	94,890,831	341,721,099

Size	Read BW (GB/s)	Write BW (GB/s)
10K	2.482	0.958
100K	1.973	0.708
1M	2.510	0.736
10M	2.283	0.626
100M	2.355	0.654

Analysis: The measured read bandwidth is about 40% of the peak bandwidth of a DDR3 RAM (6 GB/s). The read bandwidth does not scale as the data size. However, the write bandwidth decreases as the data size increases.

We observe that the write bandwidth is slower than the read bandwidth. This is a known effect for x86 CPUs because of the cacheline [1]. In order to modify a byte in the RAM, the whole cache line (64 Bytes) must be brought to the cache before writing back. This means the write operation involves both read and write.

4.2 Page fault service time

A page fault is a trap to the software raised by the hardware when a program accesses a page that is mapped in the virtual address space, but not loaded into the physical memory.

Methodology: We use mmap() function to map an ordinary file into memory, which is the same as transfer a page from desk into memory when we get a page fault. We count time cost for a page fault. An extra thing we need to take care of is letting the system ignore the cache. In this experiment, we choose a memory of size 256MB (beyond the L3 cache size) and use the stride of size 4KB.

Prediction: Usually, the dish access time ranges from 100us to 2ms. Since we are using AWS virtual machine, by calculating network penalty, we predict the disk access time to be 3ms. And the disk read speed is roughly 400mb/s, so the transfer time is 0.01us, the total page fault service time is roughly 3ms.

Result: Based on the average of ten experiments, the page fault time is 1.223ms, which is smaller than we have predicted, the reason maybe in nowadays, the Linux kernel serve page fault faster than previous time and the hardware is much faster. Dividing by the size of the page, the overhead for each byte = Page Fault Time / Page Size = 0.29us. However, when we try to access the same data from the main memory, it takes only 0.125us (memory access time without cache). From this comparison, we can conclude that even access to the SSD is slower than access to the main memory.

5 Networking

Methodology: For this part, we measure the roundtrip time, the bandwidth, and the connection overhead all at once. We use two identical EC2 instances running in the same VPC.

At t0, one machine (the client) will establish a TCP connection with the other (the server).

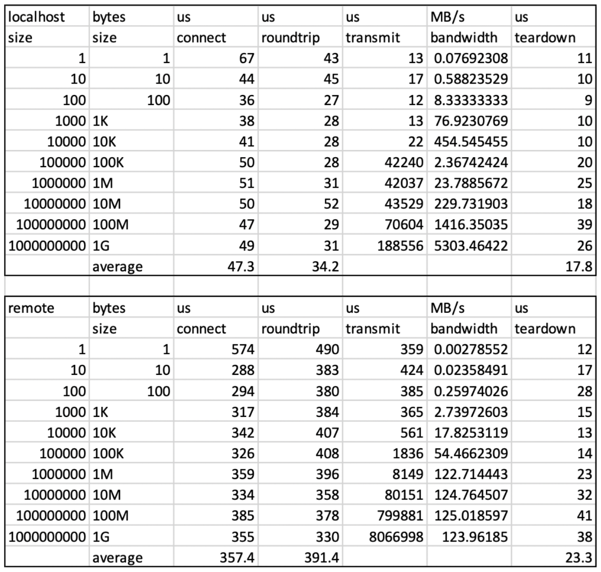
At t1, the TCP socket is returned. The client will send a metadata to the server about the size it will send.

At t2, it gets the acknowledgement from the server and begins transmitting data of the size.

At t3, it gets the acknowledgement from the server that all data has been received. Then, the client will close the connection and establish a new one.

At t4, the new connection (socket) is established.

We measure the time t0 - t4 using gettimeofday call. We don't use CPU cycles because the networking does not require nanosenconds of precision.



localhost	bytes	us	us	us	MB/s	us
size	size	connect	roundtrip	transmit	bandwidth	teardown
1	1	67	43	13	0.07692308	11
10	10	44	45	17	0.58823529	10
100	100	36	27	12	8.33333333	9
1000	1K	38	28	13	76.9230769	10
10000	10K	41	28	22	454.545455	10
100000	100K	50	28	42240	2.36742424	20
1000000	1M	51	31	42037	23.7885672	25
10000000	10M	50	52	43529	229.731903	18
100000000	100M	47	29	70604	1416.35035	39
1000000000	1G	49	31	188556	5303.46422	26
	average	47.3	34.2			17.8

remote	bytes	us	us	us	MB/s	us
	size	connect	roundtrip	transmit	bandwidth	teardown
1	1	574	490	359	0.00278552	12
10	10	288	383	424	0.02358491	17
100	100	294	380	385	0.25974026	28
1000	1K	317	384	365	2.73972603	15
10000	10K	342	407	561	17.8253119	13
100000	100K	326	408	1836	54.4662309	14
1000000	1M	359	396	8149	122.714443	23
10000000	10M	334	358	80151	124.764507	32
100000000	100M	385	378	799881	125.018597	41
1000000000	1G	355	330	8066998	123.96185	38
	average	357.4	391.4			23.3

Figure 2: networking results

The connection time is $t_1 - t_0$. The roundtrip time is $t_2 - t_1$. The bandwidth is $N/(t_3 - t_2)$ where N is size of data. The teardown time is $t_4 - t_3$.

Result: See Fig 2.

Analysis: The connection and RTT is much larger on remote than local because they involve real data transmission (handshake). The teardown time is the same for remote and local because the system call returns before the actual packet is sent. The bandwidth grows with the packet size on local host. The bandwidth also grows on remote host for small data size. After reaching 1 MB, it is limited to the network bandwidth. In this case, the limit is 125 MB/s, so AWS must limit the VPC performance to 1 Gbps.

6 File System

In this part, we try to measure the performance of file system. The file system type in our machine is EXT4.

6.1 File Cache Size

Estimation: In our case, the memory size is 1GB. The maximum cache size will be: $MemorySize - KernelSize$. To check the 'KernelSize', we run 'echo 3 > /proc/sys/vm/drop_caches' to drop all other cache. Therefore, the file cache size won't exceed 900MB. And even though we can set the maximum file cache size directly, we choose to just test it out.

Methodology: We created files with size from 8MB to 2GB and implemented a program to reatly read them in a limited time, 5 seconds in our case. For small files, we believe the throughput be high as they can be access from the memory and even L1,L2,L3 cache. However, with the file size increasing,

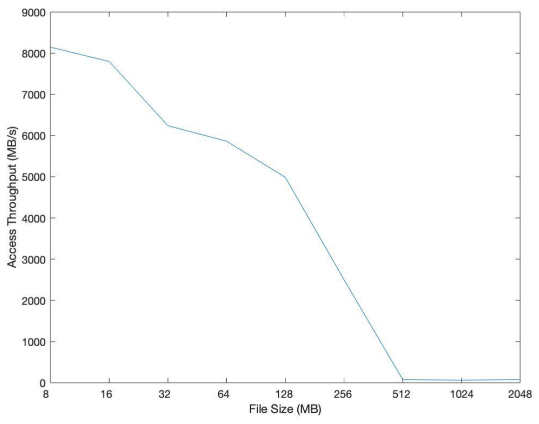


Figure 3: file cache size

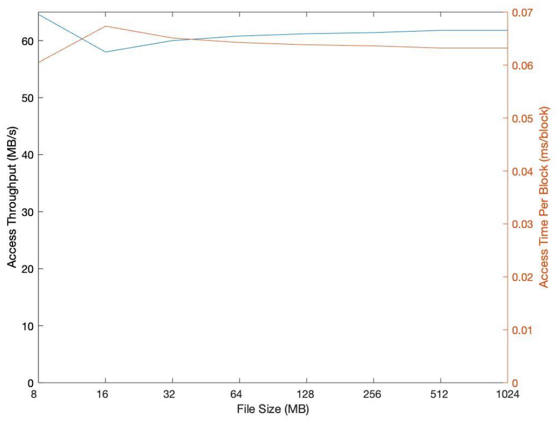


Figure 4: sequential read

the throughput will drop severely as the speed of disk is far less than memory.

Before the experiment, we drop all the cache in our system. Our program keep reading one MB a time until it reach the file size and after that it call lseek64() to make the cursor go back to the beginning of that file. The program will do it back and forth within a timeout limit, 5 seconds in our case. After the process, we can get the total size of data, and the throughput is easy to get: $Throughput = TotalSizeRead \div 5second$.

Result: See Fig 3.

Analysis: The maximum size of files we choose is twice larger than the maximum available memory size, which means the system can not cache the whole file in its memory. And from what the figure shown, we know that the severe drop happend between 128MB to 512MB, which is less than the maximum available memory. We think this is because the system need more space to keep the durability and reliability of data.

6.2 File Read Time

6.2.1 Sequential Read

Estimation: We can achieve the maxium disk bandwith through sequential access. Our machine use the General Purpose SSD as its disk and AWS guarentees at least 100 IOPS. IOPS and throughput measure different dimenssions of the disk performance. And the relation between them is: $Throughput(MB/s) = IOPS * KBperIO/1024$. For a small disk (8GB in our case), the throughput won't be very high. We assume it will not exceed 64 KB per IO and 87.5 MB/s.

Methodology: In order to make sure that we do read from disk directly, our program use the 'O_DIRECT' flag. The program read the file in the same way as described above: it read different files with size range from 8MB to 2GB and it read one MB at a time until a timeout of 5 seconds reached. We measure how much data it read and calculate the throughput using $Throughput = TotalSizeRead \div 5second$. We expect the throughtput won't change much as the file size.

Result: See Fig 4.

Analysis: The result fit our assumption and the throughput in the machine is about 60MB/s.

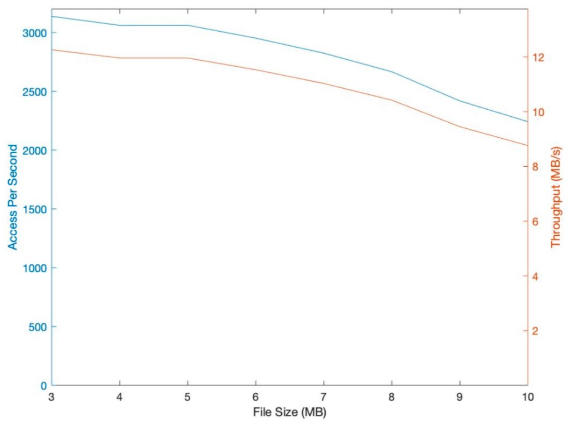


Figure 5: random read

6.2.2 Random Read

Estimation: As our machine use SSD as its disk, we expect there is little penelty for random read.

Methodology: We use the same way as above to read files except that each time we begin from a random offset. Notice that if the size we read each time exceed the block size, the system will read them sequentially. On the other hand, if it is far less than the block size, we won't get best performance. Therefore, we should carefully choose the random offset.

Result: See Fig 5.

Analysis: The result is beyond our expectation as the penalty is severe and it act evev like a HHD. If our program

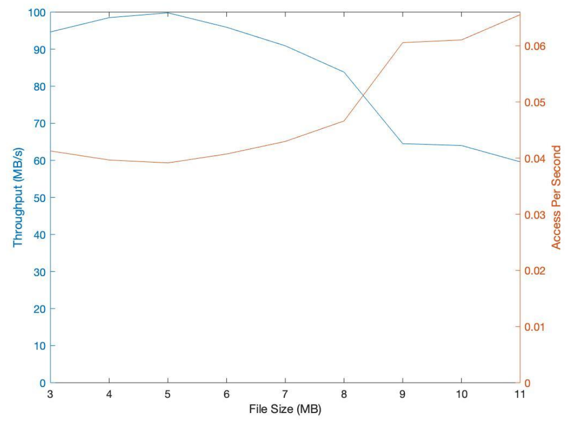


Figure 6: remote sequential read

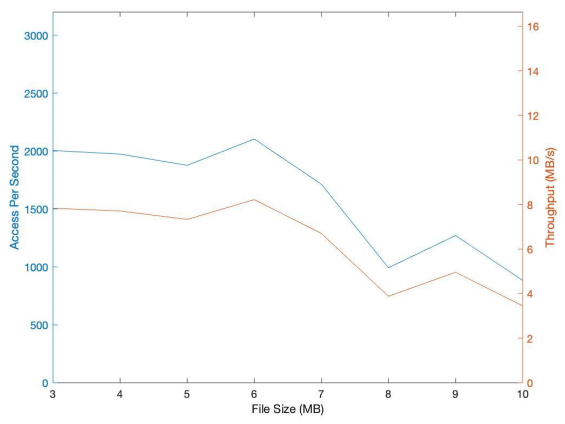


Figure 7: remote random read

work right, we don't know why it act like this. Some possible reasons might be that the overhead of our program is too high and take too much time to measure. Or there is some prefetch mechanism for the read process and when the read become totally random, it hurt performance rather than help.

But the trendency shows that the throughput won't change much as the file size increase.

6.3 Remote File Read Time

Estimation: in the experiment, we used NFS version4 as our remote file system. NFS use TCP for data transport. The network throughput between our test machines is about 125MB/s and is higher than the sequential access throughput. Therefore, we expect the network won't be the bottleneck and the overall throughput won't change much.

Methodology: In our experiment, we install NFS in two seperate EC2 virtual machine and they are in the same VPC. We use the machine we tested before as the server and another machine as client. And we mount the NFS in '/shared'. After the installation, we do the same experiments as we did before except that we read file from the mounted file system this time.

Result: See Figs 6 and 7.

Analysis: The results show that the there is no network connection penelty and the performance is even better than the original test. The reasons might be:

1. The network throughput is even higher than the disk access throughput so there is no way the network will influence the performance passively.

2. Even though we read from the NFS in client side without any cache, the server side might do some cache and the client side read data directly from teh networkk card rather than disk. The throughput of the network card in our machine is higher than the disk, therefore the overall performance will get better.

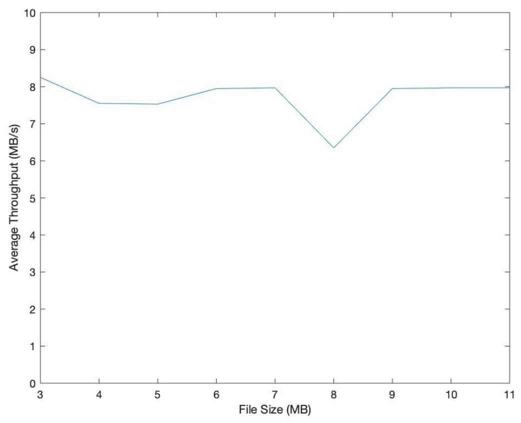


Figure 8: contention

6.4 Contention

Estimation: Disk access is the bottleneck in the whole process of data access. And if there are multiple process trying to access the data, the throughput for a single one might decrease while the overall performance remain unchanged.

Methodology: To measure the throughput in contention situation, we fork 8 processes to repeat the same experiment simultaneously. During the process, the 8 processes will content for the same data.

Result: See Fig 8.

Analysis: From the result we notice that the average throughput is

$$ThroughputWeTestedBefore \div ProcessNumber = 7.5MB/s.$$

This is reasonable result and fit our expectation. As the bottleneck of our process is disk access and even though there is some overhead during context switch, it won't influence the disk throughput.

7 Conclusion

After tens of hours of work on the EC2 machine, we get a deeper understanding of the operating system and the hardware underneath. The benchmark result shows its performance meets well with the hardware that amazon said it provides to its customer.

References

[1] Code Arcana. Achieving maximum memory bandwidth https://codearcana.com/posts/2013/05/18/achieving-maximum-memory-bandwidth.html.

[2] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. pages 279–294, 01 1996.