SysBench: Benchmarking System Performance

Edwin Mascarenhas, Raghav Prasad, Sandhya Jayaraman

1 Introduction

There are a wide variety of systems today, each running different types of Operating Systems (OS), different versions, hardware capabilities etc. While deploying an application on such a system, or set of systems it is useful to understand and characterize the performance of key operations of the system.

The goal of this project is to write a set of benchmarks in C/C++ to measure the performance of key CPU operations, memory operations, network and file system. We will use these benchmarks to evaluate the performance of Edwin Mascarenhas' laptop, which supports an 11^{th} gen Intel Core i7 processor. We write our benchmarks predominantly in C, and compile them using the `gcc 9.3.0` compiler. We compile the code at level 0 optimization by running `gcc` with the flag `-O0`.

In this report, we first present specifications related to the machine on which we execute our benchmarks, and follow this with our experiments for measuring CPU operations. We mention which group member implemented each benchmark while describing experiments in future sections. Additionally, we state that we have ensured a fair division of work among the three of us. We estimate that we spent about 5 hours per member on the project so far.

2 Machine Description

In order to analyze our measurements, and tune benchmarks, we begin by providing an in-depth machine description. Since we are using a Linux based OS, we identified some of the information from the system using `proc` and `/sys/devices/system/cpu`. We found some of the other specs (such as SSD information) online.

Table 1 summarizes the measurements for some key system specifications of the machine we ran our experiments on.

3 Measurement setup

To ensure reliable measurements for the benchmarks we use Read Time-Stamp Counter and Processor ID (RDTSCP) and Read Time-Stamp Counter (RDSTC) instructions for

System Specification	**Measurement**
Processor Model	11^{th} gen Intel Core i7-1165G7 [4]
Physical Cores	4
L1 Data Cache Size	48KB * 4 cores = 192KB
L1 Instruction Cache Size	32KB * 4 cores = 128KB
L2 Cache Size	1280KB * 4 = 5MB
L3 Cache Size (shared)	12MB
Cycle Time	0.5882 ns
DRAM Type	DDR4
DRAM Speed	3200 MTs
DRAM Size	12GB (4GB + 8GB)
Bus Width	64 bits
Memory Bus Bandwidth	51.2 GB/s
IO Bus Type	-
SSD Product	KIOXIA
SSD Model	KBG40ZNS512G
Capacity	512GB
Transfer Rate	32 GTs
Sequential Read	2,200 MB/s
Sequential Write	1,400 MB/s
Random Read	330K IOPS
Random Write	190K IOPS
Network Card Bandwidth	-
Operating System	Ubuntu 20.04.2
Kernel	Linux 5.11.0-40

Table 1: CPU Measurements

measuring the time.

Modern day processors allow operation in a number of different clock frequency and voltage configurations called Operating Performance Points or P-states. The Linux kernel allows dynamic frequency changes depending on CPU capacity to improve performance or efficiency [11]. For our measurements we disable intel pstate performance scaling driver and set the frequency to a constant 1.7Ghz.

To prevent the effects of the OS migrating a process from one CPU to another we set the CPU affinity mask for all the benchmarks using the *sched_setaffinity* call. We also set the priority of the current thread as the highest possible for a user process using the *setpriority* call.

4 CPU Operations

4.1 Measurement overhead

4.1.1 Measurement Overhead

Measurement overhead is an important factor to consider in developing benchmarks to study performance. We implement the benchmark for measurement overhead by reading the clock counter twice using the `rdtsc` instruction, and measuring the time taken in doing so. We follow the methodology described in the Intel paper [4] to read the clock counter. Additionally, we execute the clock fetch instructions once before we run the benchmark loop to ensure the instructions involved are appropriately cached in the Instruction Cache. We executed the benchmark 10000 times.
Pre-Measure Estimate: Measurement overhead does not involve any hardware overheads. Therefore, we estimate based on [3] that the instructions executed in this benchmark might take 160ns or 275 cycles.
Implemented by: Sandhya Jayaraman
Measurement: Measurement overhead took an average of 85.75ns, with a standard deviation of 20.42ns

We summarize the performance of our benchmark across 10000 iterations in Table 2.

Statistic	**Measurement (cycles)**	**Time(ns)**
Mean	145.78	85.75
Standard Deviation	34.72	20.42

Table 2: Measurement Overhead Benchmark Performance

Based on our results, we figure that our estimate was not too far off from the measured overhead value.

4.1.2 Loop Overhead

Determining the loop overhead accounts for the overheads involved in running the code to be benchmarked multiple times in a loop, in order to get as accurate an estimate as possible. To implement this benchmark, we run an empty `for` loop over 100000 iterations and measure its runtime. We read the clock counter before and after running the loop using `rdtsc`, but account for this using the measurement overhead values. Additionally, we specifically ensure that compiler optimization is set to level 0 (`gcc -O0`) to ensure that the compiler doesn't optimize the empty loop away.
Pre-measurement Estimation: Using the values provided in [3], we estimate that the loop overhead to be around 3.7 cycles, which would take about 2.18ns.
Implemented by: Sandhya Jayaraman
Measurement: Loop overhead took about 1.2ns on average.

We summarize the performance of our benchmark across 10000 iterations in Table 3.

Statistic	**Measurement (cycles)**	**Time (ns)**
Mean	2.02	1.2
Standard Deviation	0.27	0.16

Table 3: Loop Overhead Benchmark Performance

In coming up with an estimate for loop overhead, we selected the worst case time for executing, say an instruction, when a range was specified. Apart from that, we figure that our estimate on this is also fairly close to the measured value.

4.2 Procedure call overhead

Procedures are used to modularize and segment code. Procedures may accept 0 or more parameters and return at most 1 value.
Here we look at estimating the overhead associated with procedure calls as a function of the number of parameters it accepts. We have created 8 procedures (or functions) accepting 0 through 7 parameters and no return value. We choose the parameters of these procedures to be `int` and we pass 0s in the procedure calls. We perform each procedure call 1000 times and calculate the average time overhead for one procedure call.
Pre-measurement estimate: N/A
Implemented by: Raghav Prasad

We summarize the performance of our benchmark in Table 4.

Statistic	# Arguments	**Measurement (cycles)**	**Time (ns)**
Mean	1	37.986	22.343
Standard Deviation		1.66	0.976
Mean	2	37.904	22.29
Standard Deviation		1.38	0.81
Mean	3	38.6	22.70
Standard Deviation		2.74	1.61
Mean	4	37.88	22.28
Standard Deviation		1.2	0.71
Mean	5	38.41	22.59
Standard Deviation		4.37	2.57
Mean	6	37.938	22.31
Standard Deviation		1.88	1.105
Mean	7	37.918	22.30
Standard Deviation		1.37	0.805

Table 4: Procedure Call Overhead Benchmark Performance

4.3 System Call overhead

To measure the time overheads in invoking a system call, we run the `getpid` function that returns the process ID of the calling process. However, the system caches the results of a `getpid` call, which makes results dubious, unless we invoke the system call afresh each time. In order to force the system to make a new system call each iteration, we `fork` the process each iteration and measure the time taken to run `getpid` on the new forked process. This ensures that the caching effect does not factor into the accuracy of our measurements. We exit the child process each iteration, making sure that a new process is forked in the next iteration.

Pre-measurement estimate: Based on [9], we estimate that a system call should take between 1000-1500 cycles or $735ns$.
Implemented by: Sandhya Jayaraman
Measurement: System call overheads took about $88.9ns$ on average.

We summarize the performance of our benchmark over 10000 iterations in Table 5.

Statistic	**Measurement (cycles)**	**Time (ns)**
Mean	151.19	88.9
Standard Deviation	49.30	29.0

Table 5: System Call Overhead Benchmark Performance

How does it compare to the cost of a procedure call?

4.4 Task creation time

Here we consider the time overhead associated with task creation. We consider two kinds of tasks a) a process, and b) a thread.
Implemented by: Raghav Prasad

4.4.1 Process

We create a process by using `fork()`. `fork()` returns the PID of the child process to the parent, and 0 to the child process. Using this fact, we are able to distinguish between the parent and child processes and each child process is terminated right after it gets created by calling `return 0` on it. Thus, the parent process creates 1000 child processes in a loop and this is measured to obtain the average process creation time.
Pre-measurement estimate: $\sim$ 750000 cycles [8]

We summarize the performance of our benchmark in Table 6.

4.4.2 Thread

We create a kernel level threads by using POSIX thread libraries (`pthread`). We use `pthread_create` to create a thread and call a function on it that immediately kills the

Statistic	**Measurement (cycles)**	**Time (ns)**
Mean	55463.29	32623.50
Standard Deviation	14224.22	8366.68

Table 6: Process Creation Benchmark Performance

thread. In this way, we create 1000 threads in a loop and this is measured to obtain the average thread creation time.
Pre-measurement estimate: Much lesser than for a process; $\sim$ 75000 cycles

How do they compare? Threads are much more lightweight than processes. They share the same address space and therefore the overhead associated with thread creation is significantly lesser than that for a process.

We summarize the performance of our benchmark in Table 7.

Statistic	**Measurement (cycles)**	**Time (ns)**
Mean	23589.42	13875.30
Standard Deviation	11446.68	6732.93

Table 7: Thread Creation Benchmark Performance

Process and thread creation are expensive tasks, but both process and thread creation performed much better than our estimates.

4.5 Context switch time

To enable multiple processes to share the CPU, the OS' preempt a running process, save it's state and run another process that requires the CPU. This is known as a context switch. To measure the context switch time the approach we use is similar to the one described in lmbench [5]. The parent process and child process are confined to a single CPU using the affinity mask. Two pipes are created for communication between them. The child process sends a token to the parent on pipe1 and waits for the token to come back from the parent on pipe2. The parent process waits for a token from the child on pipe1 and upon receiving writes the token to the child on pipe2. This continues for N iterations. The read and write system calls are used for this purpose. The read call blocks and so it triggers a context switch. Moreover both processes are confined to the same core and so there is a context switch after each process writes and waits for the read. In each iteration there is two context switches occurring and two reads and writes to the pipe.

For isolating the measurement of the context switch time, we first measure the time to write and read to a pipe. We then subtract the read and write time from the benchmark time.
Pre-measurement estimate: 3400 cycles [2]
Implemented by: Edwin Mascarenhas
Measurement: Process context switch took 2135 cycles which comes to 1255ns. Kernel threads context switch took 1142ns.

We summarize the performance of our benchmark in Tables 8 9.

Statistic	**Measurement (cycles)**	**Time (ns)**
Mean	2165.52	1273.45
Standard Deviation	39.10	22.99

Table 8: Context Switch Process Benchmark Performance

Statistic	**Measurement (cycles)**	**Time (ns)**
Mean	2046	1023.45
Standard Deviation	38.37	22.57

Table 9: Context Switch Thread Benchmark Performance

These results align very well with our initial estimates for context switch time.

5 Memory Operations

5.1 RAM Access Time

Modern day processors have a complicated memory hierarchy and the latency of a memory operation depends on which level of cache or main memory is the requested byte resident in. To reliably measure the memory latency, we use an approach similar to the one described in lmbench [5]. We use a pointer chasing approach where the value loaded from memory is used to issue the next load request, $index = buf[index]$;. Before the measurement, the values of the array of size N are initialized in a specific pattern. The array access starts from index 0 by default. The value at any index x is randomly initialized to an index in the other half of the array different from the half x belongs to. We do this for two reasons:

- **Random initialization** CPUs have data based prefetchers which prefetch the next line in cache as well as instruction pointer prefetchers which can prefetch the next address based on a certain stride. So without random initialization, we would end up hitting in the L1, irrespective of the stride for any size.
- **Ping-pong between locations** We randomly initialize it to a location in the other half and ping pong to distribute the accesses throughout the array. This reduces the chances of the next access hitting in the cache.

Pre-measurement estimate: In this document [1], they state Kaby Lake's L1 cycle time as 5 cycles, L2 as 12 cycles, L3 as 42cycles and main memory as 60-80ns. These cache numbers are at 2.5Gz. Using these numbers as estimates, since we are running at 1.7Ghz L1 access should take about 7 cycles, 18cycles and 61 cycles. Main memory we expect 60-80 ns latency.
Implemented by: Edwin Mascarenhas
Measurement The size of the array is varied at different strides and the latency in cycles is plotted as shown in Fig. 1. The x-axis is the $log_2(number of elements)$ in array. Since each element is 4 bytes, the size in bytes is the value on x-axis multiplied by 4.

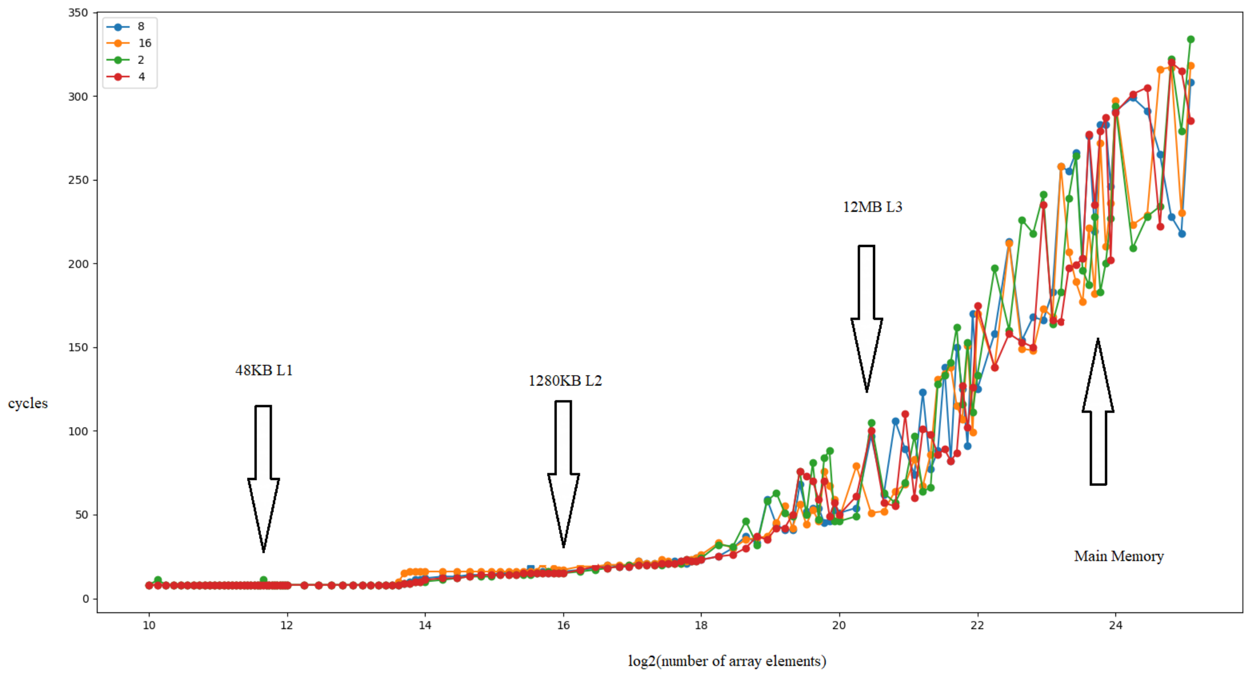

Figure 1: Memory Latency for increasing array sizes.

In the graph the different regions can be identified. The sharp outliers in the LLC cache and DRAM region are possibly because of TLB misses. With 4KB pages it is very likely at higher array sizes the benchmark with it's random accesses is thrashing the TLB. From measurement it is found that L1 cycle time is 7 cycles, L2 is 15 cycles, L3 55 cycles and DRAM latency is 200 cycles.

5.2 RAM Bandwidth

To measure the read bandwidth of our main memory, we initialized an array of size 1000 MB. By default, it contains indeterminate (garbage) values. We then proceeded to read each element of the array into a temporary variable and recorded the number of cycles required to complete a full pass through the array. A similar exercise was done to measure the write bandwidth, except here we wrote to the array and changed each value of the array to 1. We averaged this over 100 iterations, making sure to clear the cache after each iteration.
Thus we get two values, *read_cycles* and *write_cycles* that correspond to the number of cycles required to read from and write to the 1000 MB array. From here, we calculate the read and write bandwidths as follows:

$$read\_bandwidth = \frac{array\_size}{time\_for\_reads} \tag{1}$$

$time\_for\_reads$ is calculated as $time\_per\_cycle * read\_cycles$, where $time\_per\_cycle$ is merely the inverse of the clock frequency, which in our case is 1.7 GHz

Similarly, we calculate write bandwidth as follows:

$$write\_bandwidth = \frac{array\_size}{time\_for\_writes} \tag{2}$$

$time\_for\_writes$ is calculated as $time\_per\_cycle * write\_cycles$

Pre-Measurement Estimate: 52428.8 MB/s (ref. Table 1)
Implemented By: Raghav Prasad
Measurements:
Read bandwidth: 1110.471 MB/s, **Write bandwidth:** 718.380 MB/s

We find that our measured values are lower than our pre-measurement estimate, which is completely valid and expected, because our pre-measurement estimate is the theoretical maximum bandwidth. Another interesting thing to note is that the read bandwidth is higher than the write bandwidth. This is logically correct. Writing to RAM does take more time than reading from RAM, since writing requires more energy.

5.3 Page Fault Service Time

To measure the time taken to service a page fault, we first identified how to force a page fault to occur. To do this we generated a large file of size $1GB$ usind the `dd` command, and used the `mmap` [6] function to map the file into the virtual address space of the calling process. We attempt to access random pages of the file using a random index and page size offset at least 10 pages away from the previous index. This predominantly raises a page fault, which is serviced. Additionally, we clear the page cache every iteration by running `sync; echo 1 > /proc/sys/vm/drop_caches`, and set the `fadvise` flag to `POSIX_MADV_DONTNEED`.

In servicing a page fault, the kernel handles an interrupt, TLB miss, page frame allocation, and the actual page transfer. In our experiment, we only request 1 page, which is $4KB$. Let us assume a disk bandwidth of about $3000MB/s$, so the actual page transfer would take about $1300ns$. We assume that the hardware overheads take about $300000ns$ based on [7]. Therefore, we estimate that the latency of a page fault service is $301300ns$ or 512240.73 cycles.
Pre-measurement estimate: 512240.73 cycles
Implemented by: Sandhya Jayaraman
Measurement: Page Fault Servicing took 3193444.864 cycles which comes to $1878384.27ns$.

We note that our estimate was a lot lower than the measured number of cycles required in servicing a page fault. We assume that this could mean that the hardware operations involved in page fault service have a higher latency than we anticipated, and our disk bandwidth may also be lower. Additionally, we noticed that not clearing the page cache produced consistently lower numbers in terms of number of cycles, indicating that the pages accessed may be cached

Statistic	Measurement (cycles)	Time (ns)
Mean	3193444.864	1878384.27
Standard Deviation	3027999.836	1781069.503

Table 10: Page Fault Servicing Benchmark Performance

leading to lower latencies. We fixed this issue by clearing the page cache each iteration, we gave us more consistent measurements.

Dividing by the size of a page, how does it compare to the latency of accessing a byte from main memory? Page fault servicing 1 byte from the memory (dividing by page size $4KB$) takes 779.65 cycles or $458.58ns$. On the other hand, accessing a single byte from the main memory takes about 200 cycles. We feel an approximate four-fold increase in time to service a page fault makes sense, since each page fault involves at least two accesses to the main memory (to read the page, and to access the page table), and there is the overhead of loading an entire page into the memory.

6 Network

6.1 Remote Server Machine Description

The Table 11 summarizes the machine description for our remote server. We used an Intel(R) Core(TM) i7-8565U system with an installation of Ubuntu 18.04.5. We used the same machine for remote read in the file system section.

System Specification	Measurement
Processor Model	Intel(R) Core(TM) i7-8565U
Processor Frequency	1.80GHz
Physical Cores	4
Memory	7.5 GiB
Capacity	98.9GB
SSD Product	TOSHIBA
SSD Model	KBG30ZMT128G
Network Controller	Intel Corporation Wireless-AC 9260
Operating System	Ubuntu 18.04.5
Kernel	Linux 5.4.0-72-generic $x86_64$

Table 11: Remote Server Measurements

6.2 Round Trip Time

The round trip time is the time taken between a send and response at a single entity (server or client). Since round trip times on the server and client sides are expected to take about the same time, we measure the round trip time of sending a 64 byte message from a client to an echo server. We set up the server and client connections via sockets. We use the standard `listen()` and `accept()` functions to listen for connections on the server side. We use the `connect()` function on the client side to connect to a listening server.

To measure round trip time, we send a 64 byte message from a client to an echo server. We generate a random 64 byte using the `memset()` function. The reason we send this message is to imitate `ping`, which sends 64 bytes for echo.

We run the same code on both remote and loopback interfaces. One point we note is that while we set up TCP which operates at layer 4 of the network stack, `ping` runs on ICMP which operates at level 3 of the network stack. This means that the TCP packet may need more processing at the kernel level, and we suspect this can add overheads.

We used the values of `ping` as a basis to estimate the time it takes for the remote and local TCP RTT. Ping took about 0.035ms for local and 4.41ms for a remote server. We assume the TCP RTT is on the higher side of this at about 0.04ms locally, and 5ms remote.
Pre-measurement estimate: 0.04ms local, 5ms remote.
Implemented by: Sandhya Jayaraman
Measurement: Round trip time took 25024.58 cycles which is about 0.014ms in the case of loopback, and 8113024.248 cycles or 13963113.8ns for remote server.

Interface	**Measurement (ms)**
Local	0.035
Remote	4.41

Table 12: Network Round-Trip Time Benchmark (Ping) Performance

Statistic	**Measurement (cycles)**	**Time (ns)**
Mean	25024.584	14719.46
Standard Deviation	16075.152	9455.32

Table 13: Network Round-Trip Time Benchmark (Loopback) Performance

Statistic	**Measurement (cycles)**	**Time (ns)**
Mean	8113024.248	4772080.72
Standard Deviation	1159205.224	681844.51

Table 14: Network Round-Trip Time Benchmark (Remote) Performance

Surprisingly, the roundtrip time on loopback for TCP (0.014ms) was way lower than that for ping (0.035ms). However, in the remote case, roundtrip time of our benchmark (4.7ms) was comparable to that of ping (4.41ms).

6.3 Peak Bandwidth

Peak bandwidth was measured by calculating the smallest amount of time taken to send a relatively large payload using socket programming. We first construct a 100 MB string at the client-side and construct TCP sockets to send this 100 MB string over to the server side in blocks of 4 KB. We obtain the number of cycles taken for this operation, divide it by the base frequency of our processor (1.7 GHz), thus getting the time required for this operation. We perform 20 iterations of this and choose the smallest time, which corresponds to *peak bandwidth* and multiply that with our payload size (100 MB). This yields peak bandwidth.

$$min\_time = min(\frac{num\_cycles}{1.7 * 10^9})\ over\ n\ iterations \tag{3}$$

$$peak\_bandwidth = \frac{payload\_size}{min\_time} \tag{4}$$

num_cycles is the number of cycles taken for payload delivery.
Parameters: *payload_size* is 100 MB and n is 20

We perform this entire exercise twice, once with the server on `localhost` and once again with the server hosted on a remote machine.
Pre-measurement estimate: Our system uses the `802.11ac` WiFi standard. This was ascertained using `iw dev`. According to the LinkSys documentation (https://www.linksys.com/us/support-article?articleNum=186891), the peak bandwidth estimate is **1300 Mbps**
Implemented by: Raghav Prasad
Measurement:

Statistic	**Bandwidth (MBps)**
Mean	6454.277
Standard Deviation	254.4470

Table 15: Peak Bandwidth Benchmark Performance for localhost

Statistic	**Bandwidth (MBps)**
Mean	26.586
Standard Deviation	7.241

Table 16: Peak Bandwidth Benchmark Performance for remote

We find that the remote server peak bandwidth is much lower than that for loopback (and pre-measurement estimate). We can attribute this to bottlenecks placed by the network infrastructure (e.g. slow connection). Thus, we find that for remote connections, the peak bandwidth can be highly influenced by external factors such as the network quality.

6.4 Connection Overhead

To measure the connection overhead, we measure the time taken for a 3-way TCP handshake for connection setup and the time taken to close the connection using a 4-way handshake. We set up a server-client connections with sockets, using the same method of setting up the RTT experiments. In this experiment, we time the `connect()` and `close()` functions on the client side.

We estimate that the time for connection overhead is about the same time as a that of round trip time, which is about $0.014ms$ in the case of loopback. We assume that the teardown takes about half the time taken for connection setup, since it involves sending one `FIN` packet followed by an `ACK` and `FIN`. We estimate that it takes about $0.007ms$ for a loopback. For remote connection, we estimate the same: connection setup take about the same time as a round-trip. We estimate this to be about $4.41ms$. We estimate teardown to take about $2ms$, resulting in a net connection overhead time of about $6.5ms$.
Pre-measurement estimate: local - $0.021ms$, remote - $6.5ms$.
Implemented by: Sandhya Jayaraman
Measurement: Connection overheads took $0.0109ms$ in the case of loopback, and $4.79ms$ for remote connection.

Statistic	**Measurement (cycles)**	**Time (ns)**
Mean Connection setup	18639.546	10963.45
Standard Deviation Connection setup	54206.173	31883.981
Mean Connection teardown	5737.02	3374.515
Standard Deviation Connection teardown	3494.702	2055.583

Table 17: Connection Overhead (Loopback) Benchmark Performance

Statistic	**Measurement (cycles)**	**Time (ns)**
Mean Connection setup	8151775.448	4794874.31
Standard Deviation Connection setup	2407822.586	1416281.24
Mean Connection teardown	65863.846	38740.616
Standard Deviation Connection teardown	18551.227	10911.83

Table 18: Connection Overhead (Remote) Benchmark Performance

For a loopback connection, we measured connection overheads to be about $0.014ms$ which is the same as roundtrip time. We observed that our assumption that connection teardown takes negligible time to execute compared to connection setup proved to be true in both cases. For the remote connection case, we estimated $4.41ms$, and observed the connection overhead time to be $4.79ms$, which is about the same.

7 File System

7.1 File Cache Size

The file cache size is tested by sequentially reading files of varying sizes (starting from 4 KB all the way up to 8 GB). The mechanism we use is that we initially load a file into memory (*first read*) and then proceed to read it again (*second read*). The rationale is that if the file size is lesser than the file cache size, the read time will be low on the *second read.* Else, if the file size exceeds that of the file cache size, then the *second read* will be as slow as the *first read.*
Note: We flush the disk (`sync()`) and purge the file cache (by writing to `/proc/sys/vm/drop_caches`) before every *first read.*

Thus, at a particular file size, we will observe an inflection point in the plot of *file size* vs *second read time* (see Figure 2). In Table 19, note that read times increase linearly with file size, i.e., a 2 times increase in file size results in a 2 times increase in read time. This holds true until we transition from a 4 GB file to an 8 GB file, which indicates that we have found an estimate for our file cache size.
Pre-measurement estimate: Our system has 12 GB of DRAM. It would be unreasonable to assume that the kernel would allocate all of it as the file buffer cache. Thus, a reasonable estimate would be anywhere between 50 - 75% of the DRAM, which would be **6-9 GB**.
Implemented by: Raghav Prasad
Measurement: We observe an inflection point in Figure 2 between file sizes 4 GB and 8 GB, which indicates that our file cache size is around **6 GB**.

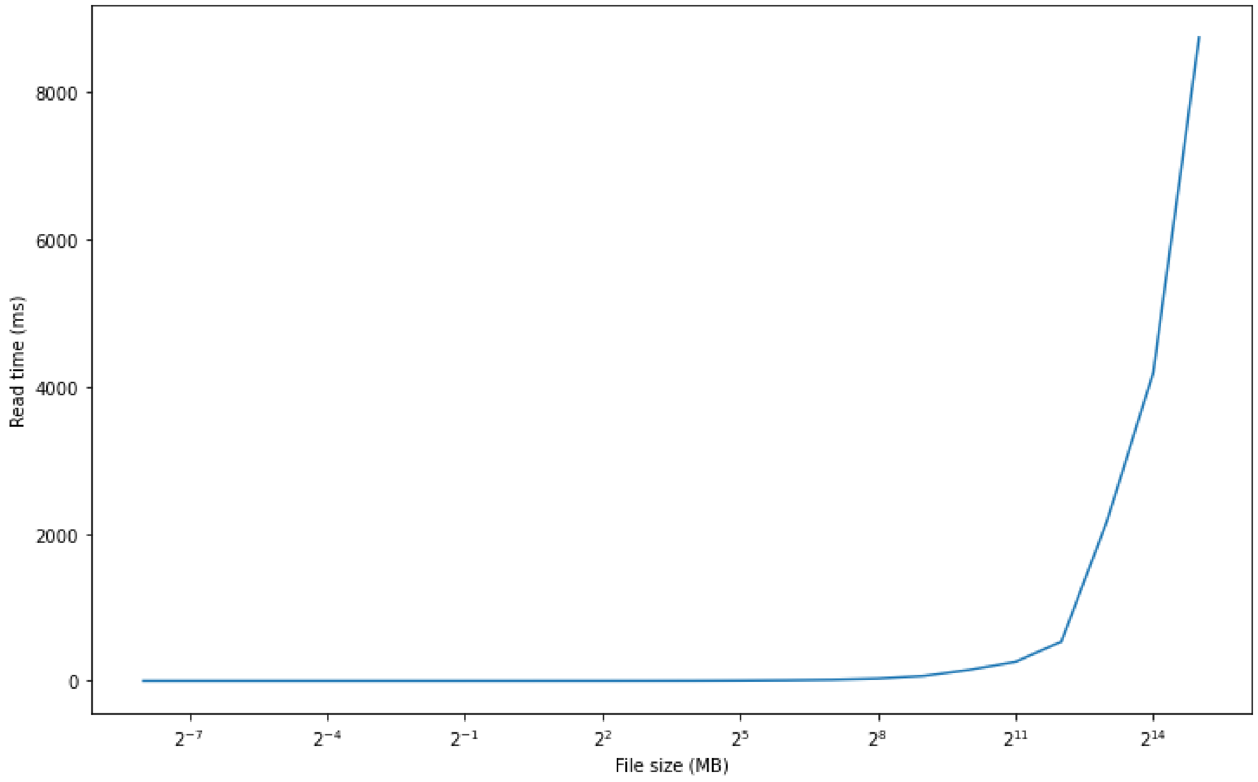

Figure 2: File cache size measurement: File Size vs Read time

7.2 File Read Time

To measure the time taken for reading a file using sequential and random access, we first had to bypass the file buffer cache that the kernel may cache into for performance improvement. To to this, we used the `read()` function with the `O_DIRECT` flag. Using `O_DIRECT` requires that the buffer we read file data into must be aligned to a multiple of the file system block size (4KB). We perform this alignment using the `posix_memalign()` function. For sequential access, we created a file descriptor to the file we wanted to read, and used the `read()` function to read into the aligned buffer in memory. For random access, we used the `lseek()`

File size	Read time (ms)
4 KB	0.00096
8 KB	0.00148
16 KB	0.00300
32 KB	0.00554
64 KB	0.00947
128 KB	0.01919
256 KB	0.03810
512 KB	0.07677
1 MB	0.16174
2 MB	0.37347
4 MB	0.41055
8 MB	0.92619
16 MB	1.72215
32 MB	4.10193
64 MB	8.37829
128 MB	15.13846
256 MB	31.19871
512 MB	66.28389
1 GB	151.13297
2 GB	259.49596
4 GB	534.27087
8 GB	2180.91315
16 GB	4184.93965
32 GB	8737.39726

Table 19: File Cache Size Benchmark Performance

function to access random blocks in the file. We accessed random blocks by offsetting the file descriptor to a random block in the file using `rand()`%(file_size/4096 − 1). We used the `dd` utility to generate files of different sizes.

The sequential read time of our disk was specified to be 2200 MB/s. We multiply this factor by the file size to estimate the time taken to sequentially read a file.

Pre-measurement estimate: From the benchmark in [10], SSD access time per block is about $0.031ms$ per block. We estimate file read to take approximately about this much.
Implemented by: Sandhya Jayaraman
Measurement: The measurement with different file sizes is summarized in Table 20 for sequential accesses and Table 21.

We observe that the read time per block (4096KB) is consistent across all file sizes. This makes sense because we are checking the time taken to read a single block in each measurement, irrespective of file size. We also observe that the average time taken for sequential read is a bit lower than the average time take for random accesses. This also makes sense, because in a random access attempts to access blocks in a non-contiguous manner.

File Size	Read Time per block (cycles)	Read Time per block (ms)
512K	109535.24	0.0644
1M	95780.58	0.0563
4M	88355.62	0.0519
16M	87251.18	0.0513
64M	86728.23	0.0510
128M	89417.52	0.0525
256M	86893.70	0.0511

Table 20: File Read Benchmark Performance - Sequential (per block of size $4KB$)

File Size	Read Time (cycles)	Read Time (ms)
512K	137218.90	0.0807
1M	132067.08	0.0776
4M	131358.82	0.0772
16M	131327.56	0.0772
64M	131762.31	0.0775
128M	132838.92	0.0781
256M	131109.24	0.0771

Table 21: File Read Benchmark Performance - Random Access (per block of size $4KB$)

7.3 Remote File Read Time

For measuring remote file read time we set up an NFS file server on another machine running Ubuntu 16.04 linux OS connected to the same wireless hotspot. The NFS server machine created a shared folder with read and write file access for the our machine. The shared folder is mounted via the command:

```
mount [NFS server IP]:[PATH TO SHARED FOLDER ON SERVER] [LOCAL MOUNT FOLDER]
```

Once the folder is mounted we use the same benchmark as section 7.2 to measure the read times with varying file sizes.

Pre-measurement estimate: The round trip time for remote access was measured to be 4.77ms for a payload of 64 bytes. For remote file read time we estimate the time as round trip time for remote access plus the time to read the file from memory from remote machine which is 0.06 ms. Although the payload is 4096 bytes, but it will be in the same packet and so we estimate the round trip time as the same as the measured round trip time. So the total estiamte is 4.83 ms.
Implemented by: Edwin Mascarenhas
Measurement: The measurement with different file sizes is summarized in table 22 for sequential accesses and table 23.

The log/log plot of average time to access a block of file with varying file size is shown in Fig. 3. From the figure it is clear that the remote access time is orders of magnitude larger than the local accesses. On the local accesses, the random accesses take noticeably longer

File size	Read time per block (cycles)	Read time per block (ms)
512K	9025360.92	5.30
1M	9096497.78	5.35
4M	9357756.84	5.504
16M	9370478.42	5.51
64M	9344573.2	5.49
128M	9371141.4	5.51
256M	9435892.2	5.55

Table 22: Remote Sequential File Read Benchmark Performance

File size	Read time per block (cycles)	Read time per block (ms)
512K	9357851.58	5.50
1M	9624156.1	5.66
4M	9494149.1	5.58
16M	9898590.8	5.82
64M	9714643.2	5.71
128M	10039557.0	5.90
256M	9879671.8	5.81

Table 23: Remote Random File Read Benchmark Performance

time than local accesses (30% more for 512K case), however in the case of remote accesses, the difference isn't as much (3% more for the 512K case). This is likely because majority of the overhead in remote accesses is from the network and the impact of random vs sequential access is diminished for remote accesses.

7.4 Contention

The file system contention benchmark takes number of processes N and number of blocks to read B. It spawns additional N-1 child processes, and creates N different files. The files are created using the dd utility making use of the /dev/urandom random number generation to write random values to the file. Each of the N processes open their respective file. The files are opened with O_DIRECT flag in addition to the O_RDONLY flag so that file I/O is directly done to the benchmark buffer without caching. We also use the posix_fadvise64 call with POSIX_MADV_RANDOM to advise the OS that run ahead prefetching of file pages will not be useful. This will prevent the OS from aggressively prefetching file pages and thus getting more reliable benchmark measurements. The file is read into a per-process buffer at the size of 4096 bytes ie size of 1 file block. The buffer address is aligned to 4096 bytes using the posix_memalign call which is a requirement for read to work from a file opened with the O_DIRECT flag. Each process reads B number of 4096 byte blocks from their respective files and the total time is divided by B to get the average time to read a block.

Pre-measurement estimate: From the benchmark studied in [10], they see that SSD access latency is measured to be 0.031ms and so we estimate that even our benchmark for 1

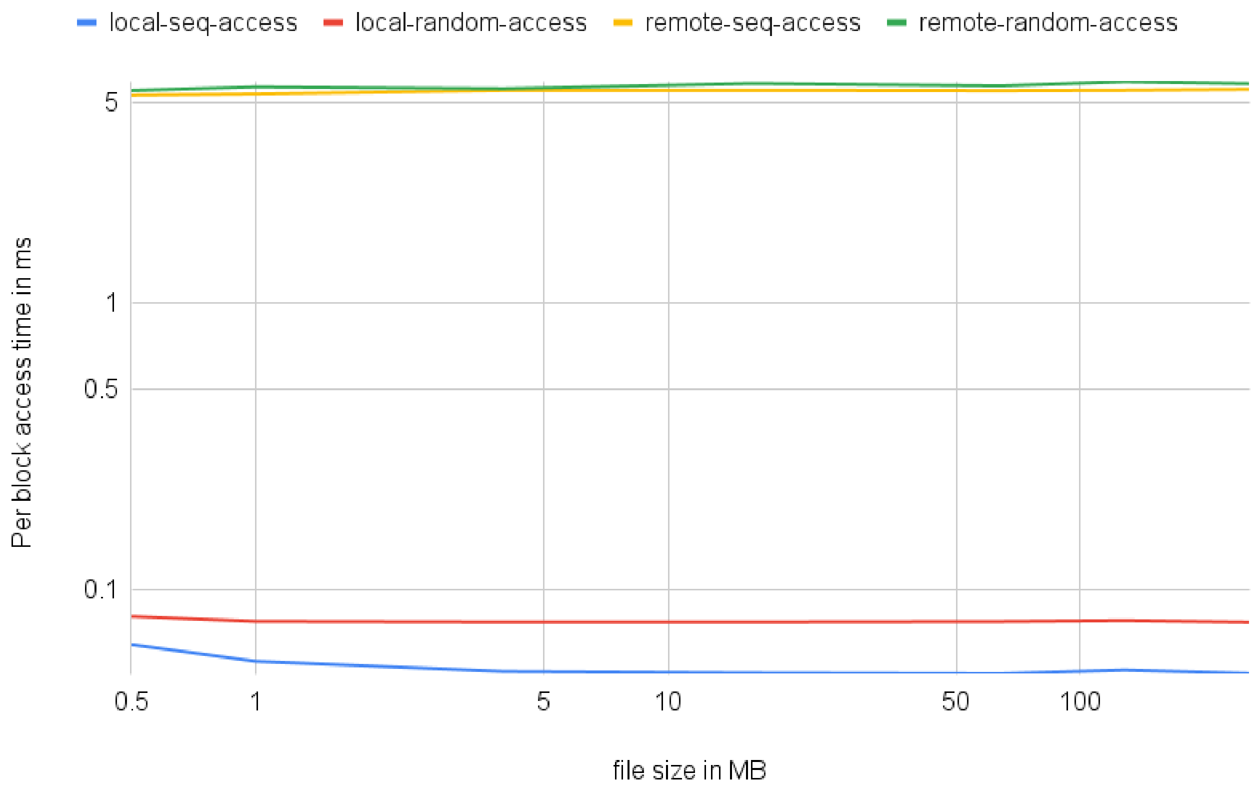

Figure 3: log/log plot of file size and average block read time

process will take similar amount of time. Moreover as the number of processess increase we estimate an increase in average time to read a single 4K block.
Implemented by: Edwin Mascarenhas
Measurement: In the benchmark, the process reads 1000 blocks of 4096 bytes each and the average time for each is reported. The number of processes are increased to see the impact of contention on the read time and is summarized in Table 24

From table 24, the measurements indicate a significant increase in access time with increases in number of processess from 1 to 2. As we go from 2 to 4 processes, there is an increase after which increasing the number of processess has negligible impact. This could be attributed to the fact that there are only 4 physical cores on the machine we are testing on. Each core can run two hardware threads. With increase in number of processess especially beyond the number of cores, the number of context switches will increase and possibly lead to less number of concurrent I/O requests due to which the increase in latency isn't much.

Processes	Avg cycles / 4096B read	Average time (ms) / 4096B read
1	79795	0.0469
2	110061	0.0647
3	110600	0.0651
4	115365	0.0679
5	117564	0.0692
6	118417	0.0697
7	121045	0.0712
8	122302	0.0719
9	126296	0.0743
10	119766	0.0705
11	127275	0.0749
12	126698	0.0745

Table 24: Contention Benchmark Performance

References

[1] *Applied C++: Memory latency.* URL: `https://medium.com/applied/applied-c-memory-latency-d05a42fe354e`.

[2] Martin Becker and Samarjit Chakraborty. "Measuring Software Performance on Linux". In: *CoRR* abs/1811.01412 (2018). arXiv: `1811.01412`. URL: `http://arxiv.org/abs/1811.01412`.

[3] Agner Fog. *Instruction Tables.* Aug. 2021. URL: `https://www.agner.org/optimize/instruction_tables.pdf`.

[4] Intel. *Intel® Core™ i7-1165G7 Processor.* `https://www.intel.com/content/www/us/en/products/sku/208662/intel-core-i71165g7-processor-12m-cache-up-to-4-70-ghz/specifications.html`.

[5] Larry W McVoy, Carl Staelin, et al. "lmbench: Portable Tools for Performance Analysis." In: *USENIX annual technical conference.* San Diego, CA, USA. 1996, pp. 279–294.

[6] *mmap(2) — Linux manual page.* `https://man7.org/linux/man-pages/man2/mmap.2.html`.

[7] *Page fault.* URL: `https://en.wikipedia.org/wiki/Page_fault`.

[8] Rob Pike et al. *Plan 9 from Bell Labs.* 1995.

[9] *Protection Ring.* URL: `https://en.wikipedia.org/wiki/Protection_ring`.

[10] *SSD throughput and latency performance.* URL: `https://www.heelpbook.net/2017/ssd-benchmark-throughput-latency-performance/`.

[11] Rafael J. Wysocki. *intel_pstate CPU Performance Scaling Driver.* `https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html`.

A CPU, Scheduling and OS Services

	Estimate	**Measurement (cycles)**	**Measurement (ns)**
Measurement Overhead	275 cycles	145.78	85.75
Loop Overhead	3.7 cycles	2.02	1.2
Procedure Call Overhead			
1 argument		37.986	22.343
2 arguments		37.904	22.29
3 arguments		38.6	22.7
4 arguments		37.88	22.28
5 arguments		38.41	22.59
6 arguments		37.938	22.31
7 arguments		37.918	22.3
System Call Overhead	1250 cycles	151.19	88.99
Task Creation Time			
Process	750000 cycles	55463.29	32623.5
Thread	75000 cycles	23589.42	13875.3
Context Switch Time	3400 cycles	2165.52	1273.45

Table 25: CPU, Scheduling and OS Services Benchmarks

B Memory

	Estimate	**Measurement**	**Measurement (ns)**
RAM Access Time			
L1 Cache	7 cycles	7 cycles	4.11
L2 Cache	18 cycles	15 cycles	8.823
L3 Cache	61 cycles	55 cycles	32.351
Main Memory	136 cycles	200 cycles	117.64
RAM Bandwidth			
Read	52428.8MB/s	1110.471MB/s	
Write	52428.8MB/s	718.38MB/s	
Page Fault Service Time	512240.73 cycles	3193444.864	1878384.27

Table 26: Memory Benchmarks

C Network

	Estimate	**Measurement**	**Measurement (ms)**
Round Trip Time			
Ping Local	-	-	0.035
Ping Remote	-	-	4.41
Loopback	0.04ms	25024.584 cycles	0.0147
Remote	5ms	8113024.248 cycles	4.772
Peak Bandwidth			
Local		6454.277MB/s	
Remote		26.586MB/s	
Connection Overheads			
Local Setup	0.014ms	18639.546	0.0109
Local Teardown	0.07ms	5737.02	0.0033
Remote Setup	4.41ms	8151775.448	4.79
Remote Teardown	2ms	65863.846	0.0387

Table 27: Network Benchmarks

D File System

	Estimate	**Measurement**
File Cache Size	6-9GB	6GB
File Read Time	0.031ms per block	
File Size	Sequential Measurement(ms)	Random Measurement(ms)
512K	0.0644	0.0807
1M	0.0563	0.0776
4M	0.0519	0.0772
16M	0.0513	0.0772
64M	0.0510	0.0775
128M	0.0525	0.0781
256M	0.0511	0.0771
Remote File Read Time	4.83ms per block	
File Size	Sequential Measurement(ms)	Random Measurement(ms)
512K	5.3	5.50
1M	5.35	5.66
4M	5.504	5.58
16M	5.51	5.82
64M	5.49	5.71
128M	5.51	5.9
256M	5.55	5.81
File Contention		
Processes	Avg cycles	Average time (ms)
1	79795	0.0469
2	110061	0.0647
3	110600	0.0651
4	115365	0.0679
5	117564	0.0692
6	118417	0.0697
7	121045	0.0712
8	122302	0.0719
9	126296	0.0743
10	119766	0.0705
11	127275	0.0749
12	126698	0.0745

Table 28: File System Benchmarks