CSE 221 Project Report

Shayan Boghani
University of California, San Diego

1 Introduction

The goal of this project is to gain intuition about the operations of difference aspects of hardware in relation to the operating system. By performing the different tasks outlined in this project, some insight can be gained about the performance of the CPU, memory, network components, and file system and ideally will aid in understanding more complex problems that may be researched in the future. The implementation for the CPU benchmarking was done in C++. Data analysis was done in Python. Some unique includes were done, the main one used to implement JSON in C++ to save the results in a readable format. For this project, measurements will be taken in a virtual machine setting using Linux Mint Xfce Edition. The implications of this are seen in some operations where the virtual machine is limited to certain resources or has policies to limit the performance. This is understandable as the priority of the system is the host OS operation. The project overall took me approximately 60 hours.

2 Machine Description

Given that the benchmarking is being done in a Virtual Machine, this section is somewhat more complex than if benchmarking were to be done on a more straightforward system. The base hardware operates on the Windows 11 Home Operating System, however the Virtual Machine (and the operating system of interest) is Linux Mint 22 Xfce Edition. The virtual machine is allocated less CPU, DRAM, and Disk than what is truly available outside of the virtual machine. These values are marked with a parenthetical which explains the amount allocated to the virtual machine as opposed to the overall hardware.

1. Processor
    - Model: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00 GHz (single core allocated)
    - Cycle Time: 0.33 ns
    - Cache Sizes:
        - L1 instruction: 48 KiB
        - L1 data: 32 KiB
        - L2: 1.3 MiB
        - L3: 12 MiB
2. DRAM
    - Type: LPDDR4X SDRAM
    - Clock: 4267 MT/s
    - Capacity: 2 GB (allocated to Virtual Machine)
3. Memory bus bandwidth: 34.136 GB/s
4. I/O bus type: PCIe 3.0
5. I/O bandwidth: 32 GT/s (x4 * 8 GT/s)
6. Disk:
    - SSD
        - Model: KBG40ZNS512G BG4A KIOXIA
        - Capacity: 25 GB (allocated to Virtual Machine)
        - Transfer Rates:
            * Sequential Read: 2200 MB/s
            * Sequential Write: 1400 MB/s
        - IOPS
            * Random Read: 330K IOPS
            * Random Write: 190K IOPS
7. Network card bandwidth: 1000 Mb/s
8. Operating system: Linux Mint 22 Xfce Edition

3 CPU, Scheduling, and OS Services

The methodology for all the benchmarks can be found here. Each of the lower overhead operations (timing and loop overhead) had ten cycles conducted, each cycle containing 1,000,000 calls. For the remainder of the benchmarks, ten cycles were again conducted, each containing 100,000 calls. The benchmarking was repeated three times to ensure consistency across results.

As a general note CPU operations, the prediction will come from the software overhead and as such base hardware performance has been excluded from the prediction tables.

3.1 Timing overhead

3.1.1 Methodology

For this section, the methodology consisted of choosing a timestamp reading function to utilize. Based on the Intel paper, in-line assembly was considered [9]. However, through testing, intrinsic functions were determined to perform identically to in-line assembly and had the benefit of a more concise implementation. The two intrinsic functions used were cpuid to implement serialization and rdtscp to get the timestamp.

The implementation for the timing overhead was done as follows. Each loop iteration consisted of calling the rdtscp function. Serialization was implemented by placing a call to cpuid before the start time was recorded and after the end time was recorded. The results of each loop were summed and then averaged over the loop count total. This operation was run ten times with 1,000,000 iterations per run. These values were then averaged to produce a final result and a standard deviation was calculated to ensure that a trimmed mean did not need to be applied.

As a note, the time was read in cycles, and then converted to wall clock time. This was done by dividing the clock cycles by the CPU frequency (as seen in the Hardware Specifications this is 3 GHz). This methodology converted cycles to nanoseconds.

3.1.2 Prediction

The predicted timing overhead was expected to be 22 cycles based on the instruction table found to reference this value [5]. This would be 7 ns when converted from cycles to wall clock time on this machine.

Software Overhead Estimate (ns)	Prediction (ns)
~ 7	~ 7

3.1.3 Benchmark

The benchmark yielded a result of 10.874 ns with a standard deviation of 0.343 ns. This standard deviation is acceptable given that it is orders of magnitude smaller than the average benchmark result, and thus a trimmed mean is not needed.

Performance (ns)	Std Dev (ns)
10.874	0.343

3.2 Loop overhead

3.2.1 Methodology

For this section, the methodology was quite simple where an empty for loop was implemented between the read measurements discussed above. Each run's time was totaled and averaged across all runs. Again, ten runs were done, each with 1,000,000 operations conducted.

3.2.2 Prediction

The predicted timing overhead was approximately 10 cycles or 3 ns on this machine. This is based on the instructions which are executed including move, loop, jump, and pushing values to the stack [13].

Software Overhead Estimate (ns)	Prediction (ns)
~ 3	~ 3

3.2.3 Benchmark

The true value for the experimentation was 2.244 ns. The standard deviation was 0.253 ns, which given the magnitude of the mean is an acceptable deviation. As such, no trimmed mean was conducted.

Performance (ns)	Std Dev (ns)
2.244	0.253

3.3 Procedure call overhead

3.3.1 Methodology

This section builds on the previous sections with one consideration made. Rather than having a nested loop to run all the functions necessary (with the changing number of arguments), this was unrolled into seven different for loops to ensure no extra overhead of jumping back to the start of the loop between iterations.

A set of arbitrary functions are defined, each empty with a ranging number of parameters (0 to 7 parameters passed

in). The time is measured around this procedure call. Ten runs were conducted for each of the functions (with 0 to 7 parameters) and each function was called 1,000,000 times per run.

3.3.2 Prediction

The predicted overhead for a procedure call is approximately 30 cycles or $\sim$ 10 ns. This is based on the understanding gained from the previous two experiments in terms of the magnitude of timing and loop overhead. With the additional parameters passed into the function, it would be expected that the values should increase linearly with each added argument. Likely, this will not be a clean linear increase due to other factors in the machine (load, optimization, etc), however this is a good starting point for a prediction.

Software Overhead Estimate (ns)	Prediction (ns)
$\sim$ 10-17	$\sim$ 10-17

3.3.3 Benchmark

The mean of the ten runs of 1,000,000 operations is calculated. From this, the increment between the means is calculated, along with the standard deviation of each of the means. The results show some peculiar results regarding the relation between the number of arguments passed in.

Initially, when the compiler did not have optimization disabled it was found that procedure calls with 4-7 variables actually had a decrement in the average performance time. Functionally, this does not make sense since there should be a jump in time when parameters begin being passed in via the stack rather than registers. However, I believe there was optimizations done to either ignore the empty calls or inlining.

To mitigate this, I disabled compiler optimization and ran the experiment multiple times. The results still displayed some variation, possibly due to the scale at which the measurement was being conducted, but are far more consistent than previous iterations.

Number of args	Mean Time (ns)	Std Dev	Increment
0	11.274	0.253	-
1	11.505	0.303	0.231
2	11.994	0.510	0.489
3	11.863	0.263	-0.131
4	11.652	0.369	-0.212
5	11.744	0.354	0.092
6	12.581	0.188	0.838
7	11.788	0.292	-0.793

3.4 System call overhead

3.4.1 Methodology

For this section, a loop is used and measurement is done around a system call, in this case getuid(). Ten runs are conducted, each with 1,000,000 operations. The results are averaged and a standard deviation is calculated to ensure a trimmed mean does not need to be applied.

3.4.2 Prediction

The predicted cost of a minimal system call is likely to be 873 cycles or $\sim$ 291 ns. This is based on the CPU clock speed as well as another benchmark found while researching the topic [12]. This is by no means an accurate representation since the machine this benchmark was run on is likely vastly different compared to the current machine described in this paper. However, this does give a good starting point for intuition.

Software Overhead Estimate (ns)	Prediction (ns)
$\sim$ 291	$\sim$ 291

3.4.3 Benchmark

The cost of a minimal system call (in this case getuid()) is 494.246 ns with a standard deviation of 32.623 ns.

Some effects of warm up were noted here, as the first few iterations tended to the higher side. Given this, a 10% trimmed mean was applied, yielding the results seen above. The initial results were 538.008 ns with a standard deviation of 56.623 ns.

Performance (ns)	Std Dev (ns)
494.246	32.623

3.5 Task creation time

3.5.1 Methodology

The methodology for this is split into two parts: processes and threads.

For processes, fork() is used to create a new process and an if statement is used to execute code for the parent versus the child to ensure the measurements are taken correctly and do not overwrite each other.

For threads the POSIX library is used and measurement is conducted in a for loop. The pthread_create call is measured to determine the thread creation time.

3.5.2 Prediction

The prediction here will be that process creation will add substantially more overhead compared to thread creation. This is because of the way threads are viewed as "lightweight" processes given that they have their own stack and can access shared data [4].

The prediction for process creation is that it will cost a large number of cycles in the range of 400,000 cycles or 133 μs. This is because when a process created, there are many operations to be done including memory allocation, information storage set up, assigning a PID, and more. This is substantially more overhead than any of the previous benchmarks [1].

The prediction for thread creation is that it will cost approximately ten times less than the cost of process creation or $\sim 13\mu$s. This is because of optimizations for threads which do not exist for processes. Threads share resources, and as such there is no overhead in allocation of memory and smaller data structures. Additionally, context switching between threads generally costs less than process context switching (as we will see in the next set of benchmarks).

	Process	**Thread**
Software Overhead Estimate (μs)	133	13
Prediction (μs)	133	13

3.5.3 Benchmark

The benchmark for the process creation overhead for this machine was 193.378 μs, a value in the same magnitude as the prediction. The standard deviation was 4.958 μs which given the magnitude of the average is reasonable. As such, a trimmed mean is unnecessary to implement.

The benchmark for thread creation is 12.907 μs. The standard deviation was 1.582 μs. This is approximately 15 times smaller than the process creation overhead which is close to the prediction of the relative value of the two benchmarks.

	Process	**Thread**
Average (μs)	193.378	4.958
Std Dev (μs)	12.907	1.582

3.6 Context switch time

3.6.1 Methodology

This methodology is once again split into two parts: process and thread context switching.

For processes, a file descriptor is initialized. A pipe is also initialized and a read and write operation is conducted in the parent and child respectively. This ensures proper ordering for the read and write, however a safe guard is placed to ensure the loops where the net time is negative are excluded.

For threads, a global file descriptor is used. Using the POSIX library, a thread is created and joined while performing the read and write operations.

Initially, A check was done to ensure the loop time is positive and then it is added to a total and averaged. However, based on experimentation I decided to implement mutex locks as a more robust solution to ensure that threads executed in the intended order.

Once again, this is run ten times, each running for 100,000 operations. The individual operations are averaged for each run, and then the runs are averaged. From here, the standard deviation is calculated.

3.6.2 Prediction

Context switching adds a substantial overhead and the prediction is that the process context switch will be more impactful than the kernel thread context switch for similar reasons as discussed in the Section 3.5.

Context switching in general is computationally expensive because of all the tasks which the operating system must do before performing the switch. This includes saving the current process/thread's state and variables, loading the state of the new process, and then executing said process [2] [3]. Numerically, the prediction is 100,000 cycles or 33 μs for the process context switch and 10,000 cycles or 3.3 μs for the thread context switch. This once again holds the factor of ten relative difference from the previous benchmark's prediction.

	Process	**Thread**
Software Overhead Estimate (μs)	33	3.3
Prediction (μs)	33	3.3

3.6.3 Benchmark

In practice, the results showed that the process context switch cost 24.096 μs with a standard deviation of 2.174 μs. These results are quite close to the prediction, at least in order of magnitude. Additionally, the standard deviation is reasonable, so no further action needs to be taken to ensure proper results.

The thread context switch was found to cost 8.481 μs. The standard deviations was 1.359 μs.

The mean calculations are consistent with the prediction, especially given that the comparison shows a substantially higher overhead for process context switches compared to thread context switches.

	Process	**Thread**
Average (μs)	24.096	2.174
Std Dev (μs)	8.481	1.359

4 Memory

As a general note memory operations, the prediction will come mostly from the hardware performance and minimal software overhead.

4.1 RAM access time

4.1.1 Methodology

The integer access operations were performed using two methodologies. They were compared and then one was chosen based on predictions as well as research. The first methodology was creating arrays of increasing size to hit different parts of memory (L1, L2, L3, Main). The idea here was that given that we know the system boundaries, an experiment can be designed to ensure that the largest array size tested does in fact hit main memory, and that there would be enough granularity to see the difference in latency of different caches as well.

While working on this, the results of the experiment were alarmingly similar. I will provide a discussion as to why this is in the Benchmark section, however this led to a second implementation. The implementation was based on the lmbench paper, and utilized a randomized Linked List [7]. By initializing a Linked List, and randomizing the memory accesses to ensure it was not accessing sequential memory, there was a significant improvement in boundaries for the latencies across different caches and memories.

The specifics of both rely on measuring the time it takes for accesses to the array or Linked List, and running iterations for the number of elements in the data structure. A stride of 16 is used to attempt to avoid prefetching and the indexed value attempting to be accessed is wrapped around the length of the data structure using a modulus operation on the length of the data structure. The measured time is then averaged across the number of iterations and returned. This operation is conducted for a set of memory sizes ranging from 4 bytes up to 2^{26} bytes. This range was chosen to ensure the proper granularity as mentioned above, as well as to ensure the maximum value did in fact reach main memory.

4.1.2 Prediction

The prediction for the access latency is based on a few specific things. An interesting starting point is an article written by Peter Norvig, who described the latency of some common PC operations, including RAM access [8].

Understanding that this was published ten years ago, these may not hold, but serve as a good starting point for intuition. Of course, system to system these values may vary drastically. Another article approximates that the L3 cache hit will cost about three times as much as an L2 cache hit [11]. From this, the prediction is 0.5 ns for L1, 7 ns for L2, 21 ns for L3, and 100 ns for main memory. This prediction is the sum of the base hardware performance and the software overhead.

For access time, the main software overhead cost is the translation of a virtual memory address to a physical memory address, however this is negligible compared to the base hardware performance.

Region	Hardware Base (ns)	Software Overhead (ns)	Prediction (ns)
L1	0.5	0	0.5
L2	7	0	7
L3	21	0	21
Main Memory	100	0	100

4.1.3 Benchmark

The results of the experiment can be seen in Figure 1.

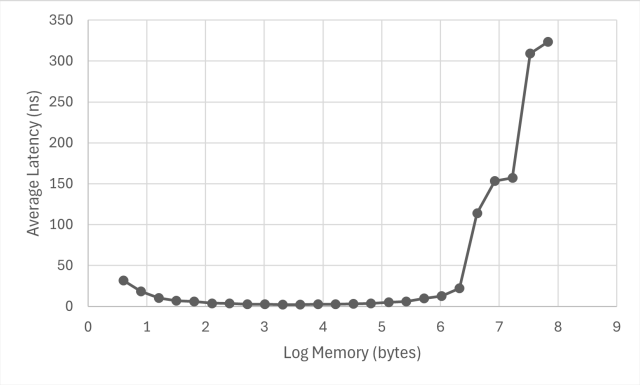

Figure 1: Log Memory vs Average Latency Graph

Region	Results (ns)	Std Dev(ns)
L1	5.424	1.222
L2	92.881	12.323
L3	157.290	27.961
Main Memory	323.321	48.419

From this figure we see clear jumps in latency. These jumps are related to the changes in which section of memory are being accessed. There is a slight spike at the beginning of collection which was observed across multiple tests of the benchmark. This is likely due to there not being enough cache warming done before the experiment begins. A solution to this might be to implement additional cache warming before the function is called.

Discussing the two methodologies presented, there is a reason why the second is superior. The first methodology saw an almost identical latency across the board, likely due

to prefetching and optimizations done to fetch data from an array structure before a call was even made. The second methodology removed this ability by creating a Linked List and randomizing addresses, so there was no way to predict the next value that needed to be accessed.

The results are not quite consistent with the predictions, likely due to the nature of the machine, which is limited in its hardware capacity. This may increase latency, leading to higher predictions than what was expected. Overall, the results do display consistency in different memory region hits, and are in similar magnitudes to the predictions.

4.2 RAM bandwidth

4.2.1 Methodology

The methodologies for read and write memory bandwidths are mostly similar excluding of course the read and write operation. To prevent tree vectorization for the loop, a function level attribute is applied. From there, an array is initialized of a specified size. Then, the start time is measured. From here, a for loop is run for a predefined loop count of 1,000,000 iterations. Loop unrolling is used here to reduce the overhead induced by the loop conditions check every iteration.

Inside the loop is where the read and write differ. One thing to note is that the initial test written accessed multiple integers on the same cache line, yielding poor results. Through experimentation, it was found that a single integer should be read per cache line, since the entire cache line is pulled in during that process. This allowed the results to perform to the expectation.

For the read benchmark, the value from the array is read and added to a sum. The addition is to ensure that the compiler does not optimize away the accesses.

For the write benchmark, a zero integer is written to the indexed memory.

After this, the end time is measured. Since the time is in cycles, this is converted to seconds. Additionally, the memory accessed is recorded based on the number of operations in the loop, the size of a single cache line, and the number of loop operations. This is also converted from bytes to gigabytes. Finally, the memory overhead is divided by the time to get a GB/s rate.

4.2.2 Prediction

The prediction for RAM bandwidth access is 34 GB/s based on the machine information seen earlier in the paper.

Operation	Hardware Base (GB/s)	Software Overhead (GB/s)	Prediction (GB/s)
Read	34	0	34
Write	34	0	34

4.2.3 Benchmark

From the benchmarking, the read and write benchmarks were approximately 32.5 GB/s and 33.24 GB/s respectively. The read benchmark had a standard deviation of 3.277 GB/s while the write benchmark had a standard deviation of 5.511. Due to the slightly high standard deviations, a 10% trimmed mean was applied here, yielding the results seen in the table below. As seen, this trimmed mean allowed for a much tighter standard deviation, indicating some outlier points in the benchmark data.

Operation	Performance (GB/s)	Std Dev (GB/s)
Read	30.427	1.638
Write	29.961	0.748

This is on par with the prediction. There may have been some limitation based on how memory was allocated to the virtual machine, however I believe that the predictions are very close to the results. The experimentation utilized loop unrolling and avoided pre-fetching, however a consideration may be to perform more sets of operations to average to achieve a better result.

4.3 Page fault service time

4.3.1 Methodology

The methodology for measuring the page fault service time is as follows. A dummy file is created and placed in the folder in which this operation will occur. The file descriptor for this file is pulled and then the length of the file is extended to non-zero to ensure mmap can be done. Next, the mmap operation is done to map this file to virtual memory. It is ensured that the file can be read from and written to, and a length is assigned to the file.

From here, a for loop is run where a new page in the mapped memory is accessed to cause a page fault. The time is measured for each access. This time is added to a running total. From here, the time is converted from cycles to nanoseconds, and then averaged over the number of pages accessed.

Finally, the memory is unmapped and the file descriptor is closed and the averaged time is returned. This operation is repeated 100 times and once again averaged and saved.

4.3.2 Prediction

The prediction is that the page fault service time will be approximately 4 ms. This is due to the operations that have to take place. During a page fault, a trap to the kernel occurs, then the kernel fetches the disk page and returns it to the TLB. The main overhead of these operations comes from the fetch from disk. Based on the previous CPU benchmarks, we find that the overhead is quite low for context switching compared

to a disk access. If we consider in microseconds, the software overhead would likely be ∼20-30 μs which would not be substantial for a benchmark measured in milliseconds.

Hardware Base (ms)	Software Overhead (μs)	Prediction (ms)
4	∼ 20-30	4

4.3.3 Benchmark

The benchmark for the page fault service time was 5.656 ms. This is fairly close to the prediction. Compared to the integer access latency, this is substantially larger, at minimum being a factor of 15,000 larger. This makes sense since disk operations cost much more than main memory hits. Additionally, the overhead of context switching between user and kernel processes adds to this cost. This justifies the substantially larger cost of a page fault service compared to an integer access in main memory.

Performance (ms)	Std Dev (ms)
5.656	0.458

5 Network

This benchmark is unique due to the fact that the operating system in question is on a virtual machine. To maintain consistency, a second, identical virtual machine was set up on a separate host device. Both virtual machines were given network privileges by attaching them to the "Bridged Adapter" on their respective host devices. This made them visible on the network, and thus able to communicate.

Through some initial testing, it was found that there was significant latency added to communicate between the virtual machines and the base machines. The communication between the host machines averaged 1 ms, whereas the communication between the virtual machines averaged 5 ms. A possible explanation for this is latency associated with the host network adapter having to forward packets to the virtual machine, adding overhead.

Additionally, the methodology for localhost versus remote are identical outside of the IP Address passed to the client so that it may connect to the correct server (either 127.0.0.1 for localhost or the IP Address of the other VM).

5.1 Round trip time

5.1.1 Methodology

The methodology for this section can be split into two parts - the server and the client.

On the server side, a socket is initialized as well as server information. The socket is then bound to the port. Then, the server listens and accepts incoming requests. From there, we move into a loop which continually receives and sends packets (echoing packets back to the client). Once the client disconnects, the loop is terminated and the socket is closed on the server side.

On the client side, a socket is initialized as well as server information. The socket is connected to the server. From here, we measure the time it takes to send and receive a packet in a loop. This loop is run for 1,000,000 operations. The time is totaled and then averaged over the number of runs. After the end of the loop, the socket is closed on the client side.

These results can then be compared to the ICMP requests made to the remote and loopback interface.

5.1.2 Prediction

The prediction for this section can be considered from a hardware base performance and software overhead consideration.

For loopback, there would be no hardware component since the packet would be handled by the VNI. The prediction is that the software overhead incurred by this will be significantly smaller than in a remote connection, likely in the order of 0.1 ms.

For remote, there would be a hardware component since the packet is sent/received by the respective server/client NIC, as well as travel over the router components in the network. Given the WiFi speed of the current network is 697.4 Mbps, the transmission should be extremely quick. Using an equation of:

$$time = \frac{packetsize(bits)}{speed(bits/s)}$$

The time would be approximately 0.01 μs. This estimation is quite flawed however and it is unlikely the packet transmission would be this quick on a remote connection. The estimate would be that for remote the packet transmission would take approximately 10 ms. This would likely be mostly on the base hardware performance for things like IP lookup as well as packet routing.

	Base Hardware Performance (ms)	Software Overhead (ms)	Prediction (ms)
Loopback	0	0.1	0.1
Remote	10	0	10

5.1.3 Benchmark

This section can be split into the loopback and remote results.

The loopback result for the TCP round trip time was calculated to be 14.611 μs with a standard deviation of 2.238 μs.

This is an acceptable standard deviation for the magnitude of the round trip time benchmark, and as such no trimmed mean needs to be implemented.

The ICMP request was performed for ∼300 iterations and the average value was calculated to be 45 μs with a standard deviation of 13.8 μs. One interesting thing to note was that there were significant outliers for the ICMP requests which were up to 3000 μs for one iteration. This increased the average and standard deviation values significantly. When the interval time was decreased from 1 second to 2 ms, the average value was calculated over ∼3000 iterations to be 10 μs or 0.01 ms with a 51 μs standard deviation 0.051 ms. Again, this indicates that there were individual iterations with excessively high times.

The remote result for the TCP round trip time was calculated to be 6.622 ms with a standard deviation of 1.103 ms.

The ICMP request was performed at an interval of 2 ms for 3000 iterations and the average value was calculated to be 4.59 ms with a standard deviation of 1.562 ms.

In general, the ICMP is expected to be faster due to its handling at the kernel level, while with TCP it is handled at the user level. This incurs its own added overhead, including context switching to the kernel to perform the necessary calls to communicate the packets.

From this benchmark and the comparison between the loopback and remote results, it can be deduced that the baseline hardware performance is orders of magnitude larger than any software overhead from the OS. The TCP benchmark is quite close to the ICMP performance which I would consider the ideal case. However, due to TCP policies like acknowledgment overhead, network latency, and network policies that may act as bottlenecks (e.g. slow start) this ideal case cannot be achieved.

TCP Connection Benchmark

	Performance (ms)	**Std Dev (ms)**
Loopback	0.015	0.00224
Remote	6.622	1.103

ICMP Connection Benchmark

	ICMP Time (ms)	**ICMP Std Dev (ms)**
Loopback	0.01	0.051
Remote	4.59	1.562

5.2 Peak bandwidth

5.2.1 Methodology

For this methodology, we can once again split the approach into the server and client side. The difference between the loopback and remote methodologies is changing the server IP to correspond to the correct server location (127.0.0.1 or the remote server IP).

The server side initializes, binds, and listens on a socket. It then accepts any requests on that socket. Then, it will loop and read all messages coming in. When the client is done, the server will exit and close the socket.

From the client side, we once again initialize and connect a socket. We then write/send data to the server and count the number of bytes sent while also measuring the time. After a certain amount of bytes are sent, the loop ends and we average the bytes sent over the time it took to send them, thus yielding a bandwidth in the form of GB/s.

5.2.2 Prediction

With this benchmark, my understanding is that it will not be possible to meet the maximum bandwidth of the network card which is 1000 MB/s.

My expectation for loopback is that 250-500 Mb/s is achievable. This is because of limitations on network hardware access to the VM, as well as bandwidth limiters that likely ensure that you remain below a certain percentage of the total bandwidth so as not to over consume resources.

My expectation for remote is even lower at around 100 Mb/s due to the added overhead of the network interface, as well as dealing with two NICs. Additionally, there is a dependency on the network itself to be able to deliver the packets.

The software overhead in this case is quite negligible compared to the hardware base performance, and as such can be attributed to zero.

	Base Hardware Performance (Mb/s)	**Software Overhead (Mb/s)**	**Prediction (Mb/s)**
Loopback Bandwidth	250-500	0	250-500
Remote Bandwidth	100	0	100

5.2.3 Benchmark

The benchmarks performed much lower than expected. After some experimentation, it was found that VirtualBox has a

limitation that is enforced on network interfacing. When I ran the same experiment between the host machine the loopback bandwidth jumped up to ∼550 Mb/s which is much closer to the expectation. Additionally, the remote bandwidth jumped up to ∼150 Mb/s which again is closer to the estimation. There was no case where the full 1000 Mb/s were met, and I believe this is simply because there are limitations on how many resources a single source can use at a time. Additionally, for the remote procedure, the network acts as a bottleneck as well.

From this benchmark and the comparison between the loopback and remote results, it can be deduced that the baseline hardware performance is orders of magnitude larger than any software overhead from the OS. The benchmark does not perform and that is partially due to the nature of the VirtualBox VM, as was derived above through experimentation as mentioned above. Additionally, networking policies ensure that one connection cannot take up the full hardware bandwidth, and this applies on both sides of the communication.

	Performance (Mb/s)
Loopback	170.104
Remote	33.095

5.3 Connection overhead

5.3.1 Methodology

The methodology for this section is fairly similar for the loopback and remote client-server model. The only difference is the IP address set for the client to connect to (one being the loopback IP and one being the server IP).

For the setup benchmark, the client initializes the socket for each loop. Then it connects, which is the measured time of the code. Finally, it adds this time to a total count and closes the socket. This is repeated ten times with 100,000 operations each run. The server continuously accepts new connections after initializing the socket as mentioned in the round trip time benchmark.

For the teardown benchmark, the server side remains the same. For the client side, the socket is initialized and connects to the server. Then the client sends a packet to the server. After this the start time is measured and the socket is closed. Finally, the end time is measured. This is once again repeated ten times with 100,000 operations each run.

An interesting aspect to note is that due to the fact that the connect call is asynchronous, it became imperative to send the packet between connecting and closing the socket to ensure that the connection was made before the close occurred. Through experimentation, this adjusted the order of magnitude of the benchmark results.

5.3.2 Prediction

This section can be split into the prediction for the loopback and the remote setup and teardown operations.

Loopback

The setup will likely take 0.15 ms, or a similar value as the previous RTT loopback benchmark. This is because there is no physical network to route the packet through, so there is only software overhead.

For the teardown, it will likely take 0.1 ms, or a slightly lower amount of time than the setup. Again, there is no physical routing, so there would only be software overhead.

Remote

The setup will likely take a similar amount of time to the RTT remote benchmark. It should theoretically be 1.5 times as large given that it is a 3-way handshake, as opposed to the two packet exchange we measured in RTT. Therefore, the expectation is that the setup will take 9 ms. In this case, the hardware time is so much larger than the software overhead, that the software overhead becomes negligible.

For the teardown, there is no added overhead given that it is a single packet sent to notify the server that the connection is closing. As such, the remote teardown should be similar to the loopback teardown at 0.1 ms.

	Base Hardware Performance (ms)	**Software Overhead (ms)**	**Prediction (ms)**
Loopback Setup	0	0.15	0.15
Loopback Teardown	0	0.1	0.1
Remote Setup	9	0	9
Remote Teardown	0	0.1	0.1

5.3.3 Benchmark

Setup Operation

The following table relates to the set up operation for this benchmark. Given the methodology for a TCP 3-way handshake, there is a distinct difference in time between loopback and remote. The 3-way handshake works to send three packets back and forth to solidify the connection. As such, the remote measurement will be similar to our round trip time for remote. For the localhost, this communication can be routed much faster given that there is no network latency, routing, or NIC interaction.

Teardown Operation

The following table relates to the tear down operation for this benchmark. It can be seen that the tear down times are of the same magnitude for both loopback and remote. This is

	Performance (ms)	Std Dev (ms)
Loopback	0.105	0.00397
Remote	7.898	0.281

because there is no acknowledgment needed for tear down, and as such, only one packet is sent to notify the server of the close connection. The measurement should only contain this so it makes sense that these would be similar.

	Performance (ms)	Std Dev (ms)
Loopback	0.166	0.00149
Remote	0.232	0.036

6 File System

6.1 Size of file cache

6.1.1 Methodology

The methodology for this section consists of utilizing increasing sizes of files to attempt to extend beyond the file cache. Intuition is that the file cache should be missed after the size of the file exceeds the memory.

For this experiment, a set of files are created of varying sizes. Each file is then read in first to ensure it ends up in cache. Then, it is read again and the time to read is measured. The file is read backwards to attempt to prevent optimization. The set of times are graphed against their respective file sizes. The idea is that there will be a large jump where the cache is missed.

6.1.2 Prediction

The prediction for this benchmark is that the cache size will be less than the total memory. This is because it is unlikely that the entire memory will be unutilized for this operation. Given the small memory size for this virtual machine, I predict that the cache size will be approximately half the hardware specification which would be 1 GB or 1000 MB. There would not be a relevant software overhead here as this is a measure of the cache size.

Base Hardware Performance (MB)	Software Overhead	Prediction (MB)
1000	0	1000

6.1.3 Benchmark

The benchmark can be seen in Figure 2 as well as the table corresponding to the values. The cache size is likely between 1000 and 1500 MB. This is where the jump in access time is seen in the figure. This is close to the prediction value, and as such the prediction can be considered successful. The limitations that kept from the peak hardware performance is likely other operations running and consuming resources. As mentioned in the benchmark description, this benchmark is sensitive to the OS interactions occuring.

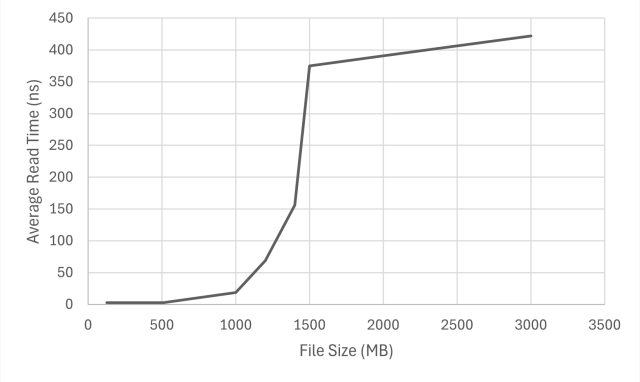

Figure 2: Access Time (ns) vs File Size (MB)

Size (MB)	Access Time (ns)
128	2.677
256	2.556
512	2.742
1000	18.568
1200	68.721
1400	156.232
1500	375.059
2000	390.982
3000	422.158

6.2 File read time

6.2.1 Methodology

This section can be split into the methodology for the sequential read time and the random access read time. One note for all of the operations is that the cache is manually cleared before the start of every operation in this section. A varying set of file sizes is used.

For sequential access, a file is loaded in. Then, it is read in sequentially until the entire file is read, and this time is measured. A posix_fadvise is applied to ensure no caching. The file is closed and the time is returned for the operation.

For random access, a similar process is set up except for how the data is read. For the number of blocks in the file, a random block is chosen and read. Each block access is measured, and then this is averaged across all of the accesses to get an average read time.

6.2.2 Prediction

From an estimation perspective, this is a heavily hardware performance operation with software overhead that is an order of magnitude smaller than the hardware aspect.

Sequential access on an SSD should have a much smaller magnitude than random access. My estimation is that the random access will be $\sim$10 μs.

Random accessing on an SSD should be an operation of magnitude between 50-100 μs [10]. Assuming some added overhead from the virtual machine having to navigate requests through the translation layer, my prediction is that the access would be on the higher end of the spectrum, so 100 μs.

Operation	Base Hardware Performance (μs)	Software Overhead (μs)	Prediction (μs)
Sequential	10	0	10
Random	100	0	100

6.2.3 Benchmark

Below are the results from the benchmarking for the local operations. As seen from the table, the relative value of the sequential versus random access is similar to the prediction. Given that the order of magnitude is similar, the prediction is successful.

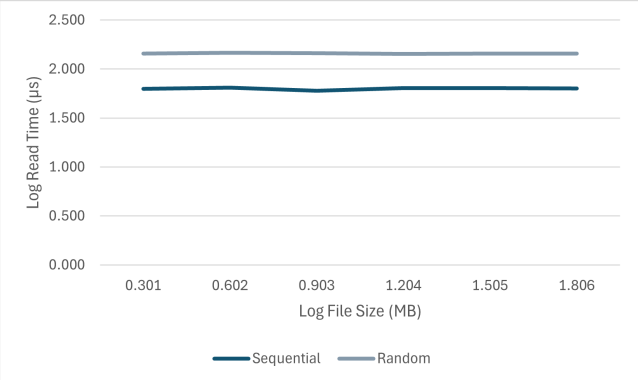

Figure 3: Log-Log Plot for Local Access Sequential/Remote

File Size (MB)	Sequential (μs)	Random (μs)
2	62.539	143.67
4	64.365	147.239
8	60.02	144.962
16	64.134	143.227
32	63.986	143.602
64	63.588	143.335

6.3 Remote file read time

6.3.1 Methodology

For this experiment, the first step was to mount the remote directory. This required some research into options for mounting a remote directory in Linux. The same machine from the networking operations was reused as the server in this benchmark.

The three mounting options considered were SSHFS, NFS, and Samba. The lowest barrier of entry for use was SSHFS, and the simplicity lent it to being simple to set up using a simple SSH key. One consideration that will impact the results of this benchmark are the performance limitations of SSHFS. From some general research, it can be found that NFS outperforms SSHFS in many relevant benchmarks [6]. As such, this benchmark should be viewed within the constraints imposed by SSHFS alongside the operating itself.

The methodology for the experiment is identical, except for that the file directory passed in to access is now the remote directory that has been mounted on the system.

6.3.2 Prediction

The prediction for the remote access is that it will likely add a significant network penalty. This is because based on the networking benchmarks, the order of magnitude is much higher for network versus the file system access operations. From the previous benchmark, I am estimating that the remote sequential access will be 6 ms, and the random access will add a slight amount of so it will be closer to 7 ms.

Operation	Base Hardware Performance (ms)	Software Overhead (ms)	Prediction (ms)
Sequential	6	0	6
Random	7	0	7

6.3.3 Benchmark

The benchmark for the remote read is as follows. As predicted, the random access had a higher latency than the sequential, and both were orders of magnitude higher than the local operation. The network penalty is quite large at 5-7 ms, and makes

up a majority of the benchmark result. Different aspects come in to play here with the interactions with the hardware (NICs), routing, and additional network overhead which was discussed in the networking benchmarks. The network penalty is much larger than the local access time, and thus we find that the disk block access is actually not the relevant part of this benchmark when comparing the local and remote reads. As for the prediction, the relative magnitude was correct, and the idea that sequential will always be less than random held so I would say the prediction was fairly accurate. The results also have a small standard deviation which is reasonable compared to the average value.

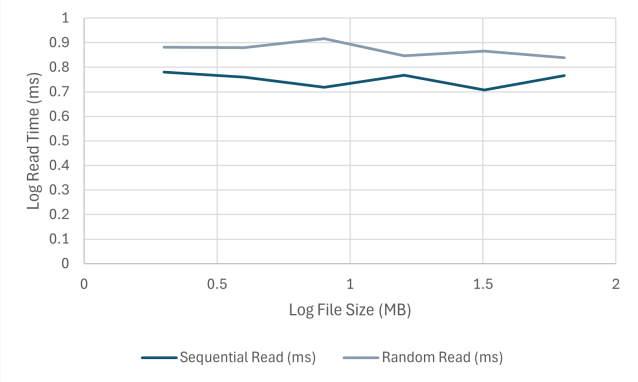

Figure 4: Log-Log Plot for Remote Access Sequential/Remote

File Size (MB)	Sequential (ms)	Random (ms)
2	6.021	7.623
4	5.749	7.592
8	5.23	8.257
16	5.844	7.022
32	5.09	7.333
64	5.832	6.887

6.4 Contention

6.4.1 Methodology

The methodology for contention is as follows. The measurement to perform is the change in time to access a block when more processes exist. To do this, we measure up to forty processes at a time, and each of these, we create that number of processes. Each of these processes will be produced by forking the parent, and then opening and then reading a block from a file passed in. This is measured and averaged across the number of accesses per process. There are ten trials run for each set of processes. The result is then graphed.

For this benchmark, sequential access was utilized so as to keep performance variance isolated to an increase in the number of processes

6.4.2 Prediction

The prediction for this benchmark is that the increase of access time should proportionally increase with the number of processes. It is difficult to classify the prediction in the normal format.

The reason to believe the benchmark will be proportional is because of how the system will be operating. Given that there is only one core, there will be waiting to ensure that all the processes can read, and there will be context switches. These will add overhead, however it will be consistently more overhead as the number of processes increase. Given this, it makes sense that the time would scale with the number of processes.

6.4.3 Benchmark

The benchmark proves that as the number of processes increases, the access time also increases linearly. Discussing the prediction, it is intuitive that there would be a proportional penalty for every additional process added given that resources are consumed at a similar rate between these processes. I believe that the benchmark was successful in verifying the prediction. There is an interesting spike in the graph, however that may be due to variability in the system. The general trend remains consistently positive.

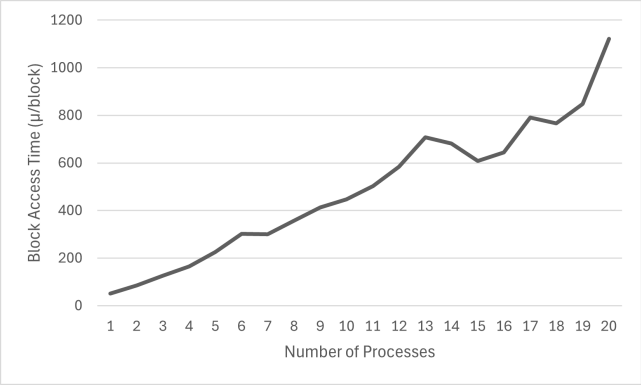

Figure 5: Number of Processes vs Block Access Time (μsec/block)

The data for this can be seen in the table that follows. There is a clear proportional trend between the number of processes and the access time.

Number of Processes	Block Access Time (μs/block)
1	51.871
2	84.66
3	126.574
4	164.754
5	226.02
6	302.376
7	300.994
8	357.807
9	412.388
10	447.065
11	502.333
12	583.363
13	708.284
14	681.942
15	608.826
16	644.338
17	790.423
18	766.268
19	847.408
20	1121.523

7 Summary

Operation	Base HW Performance	Estimated SW Overhead	Predicted Result	Measured Result
Read Time	0 ns	∼7 ns	∼7 ns	10.874 ns
Loop Overhead	0 ns	∼3 ns	∼3 ns	2.244 ns
Proc Call 0 Arg	0 ns	10 ns	10 ns	11.274 ns
Proc Call 1 Arg	0 ns	11 ns	11 ns	11.505 ns
Proc Call 2 Arg	0 ns	12 ns	12 ns	11.994 ns
Proc Call 3 Arg	0 ns	13 ns	13 ns	11.863 ns
Proc Call 4 Arg	0 ns	14 ns	14 ns	11.652 ns
Proc Call 5 Arg	0 ns	15 ns	15 ns	11.744 ns
Proc Call 6 Arg	0 ns	16 ns	16 ns	12.581 ns
Proc Call 7 Arg	0 ns	17 ns	17 ns	11.788 ns
System Call	0 ns	∼291 ns	∼291 ns	494.246 ns
Proc Creation	0 ns	133 ns	133 ns	193.378 ns
Thread Creation	0 ns	13 ns	13 ns	12.907 ns
Proc Context Switch	0 ns	33 ns	33 ns	24.096 ns
Thread Context Switch	0 ns	3.3 ns	3.3 ns	8.481 ns
RAM Access L1 [Figure 1]	0.5 ns	0 ns	0.5 ns	5.424 ns
RAM Access L2 [Figure 1]	7 ns	0 ns	7 ns	92.881 ns
RAM Access L3 [Figure 1]	21 ns	0 ns	21 ns	157.290 ns
RAM Access Main Memory [Figure 1]	100 ns	0 ns	100 ns	323.321 ns
RAM Read BW	34 GB/s	0 GB/s	34 GB/s	30.427 GB/s
RAM Write BW	34 GB/s	0 GB/s	34 GB/s	29.961 GB/s
Page Fault Service Time	4 ms	$\sim$ 20-30 μs	4 ms	5.656 ms
Network RTT Local	0 ms	0.1 ms	0.1 ms	0.015 ms
Network RTT Remote	10 ms	0 ms	10 ms	6.622 ms
Network Peak BW Local	250-500 Mb/s	0 Mb/s	250-500 Mb/s	170.104 Mb/s
Network Peak BW Remote	100 Mb/s	0 Mb/s	100 Mb/s	33.095 Mb/s
Network Setup Local	0 ms	0.15 ms	0.15 ms	0.105 ms
Network Setup Remote	9 ms	0 ms	9 ms	7.898 ms
Network Teardown Local	0 ms	0.1 ms	0.1 ms	0.166 ms
Network Teardown Remote	0 ms	0.1 ms	0.1 ms	0.232 ms
File Cache Size [Figure 2]	1000 MB	0 MB	1000 MB	1000-1500 MB
Local Seq Read Time	-	-	-	see Figure 3
Local Random Read Time	-	-	-	see Figure 3
Remote Seq Read Time	-	-	-	see Figure 4
Remote Random Read Time	-	-	-	see Figure 4
Contention	-	-	-	See Figure 5

References

[1] ABRAHAM SILBERSCHATZ, G. G., AND GALVIN, P. B. *Operating System Concepts, Ninth Edition*. Wiley, 2012.

[2] ASHWATHNARAYANA, S. Understanding context switching and its impact on system performance. https://www.netdata.cloud/blog/understanding-context-switching-and-its-impact-on-system-performance/#:~:text=This%20is%20because%20each%20context,can%20slow%20down%20the%20system., 2023.

[3] BHARGAV, N., AND SIMIC, M. Context switches in operating systems. *Baeldung Computer Science* (2023).

[4] CLANCY, M. What's the diff: Programs, processes, and threads. https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/#:~:text=Some%20people%20call%20threads%20lightweight,to%20communicate%20between%20the%20threads., 2024.

[5] FOG, A. Instruction tables. https://www.agner.org/optimize/instruction_tables.pdf, Nov. 2022.

[6] JAKELER. Nas performance: Nfs vs. smb vs. sshfs. https://blog.ja-ke.tech/2019/08/27/nas-performance-sshfs-nfs-smb.html, 2019.

[7] MCVOY, L., AND STAELIN, C. lmbench: Portable tools for performance analysis, 1996.

[8] NORVIG, P. Teach yourself programming in ten years. https://norvig.com/21-days.html#answers, 2014.

[9] PAOLONI, G. How to benchmark code execution times on intel® ia 32 and ia-64 instruction set architectures, 2010.

[10] PCMAG. Disk vs. ssd. https://www.pcmag.com/encyclopedia/term/access-time#:~:text=SSD,25%20to%20100%20microsecond%20range., 2024.

[11] PODGORNYY, S. Cpu cache basics. https://dev.to/larapulse/cpu-cache-basics-57ej#:~:text=L1%20cache%20is%20the%20fastest,L3%20cache%20access%20time:, 2023.

[12] SKOCIK, P. https://stackoverflow.com/questions/23599074/system-calls-overhead, 2019.

[13] TUTORIALSPOINT. Assembly - loops. https://www.tutorialspoint.com/assembly_programming/assembly_loops.htm, 2024.