System Benchmarking and Measurement

Mayank Jain and Jonathan Woenardi

University of California, San Diego

1 Introduction

As part of the CSE 221: Graduate Operating Systems course, this project focuses on characterizing and understanding system performance, specifically in the context of operating systems and applications. We address the need to empirically determine the performance of hardware components like the CPU, RAM, file system, and network, and how they impact system services and user experience. Many of these performance metrics require experimental investigation, as they are not readily available in documentation.

The team comprises of Mayank Jain, and Jonathan Woenardi. All experiments were coded, debugged and discussed by both. We required a total of 80 hrs for the entire project.

1.1 Testing Methodology

The following steps were taken to ensure accuracy, reproducibility and correctness of all experiments:

1. Used `nice` to boost process priority to avoid frequent context switches.

2. Restricted measurement programs to using a single core to prevent multi-process context switches wherever necessary.

3. Verified `constant_tsc` is set for all processors to workaround dynamically adjusted CPU frequency.

4. Used `rdtsc` and `rdtscp` for reliably reading cycle counters with low-overhead.

5. All programs were written in C programming language, and were compiled with GNU C Compiler (gcc) with `-O0` flag, which completely disables most optimizations wherever necessary.

   ```
   $ gcc --version
   gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
   ```

6. When measuring an operation using multiple iterations, all runs of an iteration were considered as a single trial and statistics were calculated across multiple trials (as opposed to an individual iteration). For example, we ran a function call 1000 times, averaged the execution time and repeated this for 10 times to calculate the reported mean and standard deviation. The values are printed to `stdout` and a separate python script collects these values and calculates the mean and standard deviation of the data. All experiments were run 10 times wherever not specified.

1.2 Format

Every operation contains four sections, namely methodology, estimates, results and analysis. *Methodology* describes how the experiment was performed. *Estimates* outline our predictions for the execution time based on our knowledge. *Results* indicate our estimates based on base hardware performance and software overhead and include a table with mean, standard deviation and graphs showing relationships between variables if any. We discuss the results in the *Analysis* section.

2 Machine Description

We decided to use a Gigabyte P55W v5 laptop running Ubuntu (hereafter, referred to as the test machine). We referred to its technical specifications document [2,4,5], utilities like `sysctl` provided by the operating system, and online databases of laptops and their features to gather more information about the machine.

2.1 Processor

The test machine's processor is an Intel Core i7 6700HQ which is based on the Skylake microarchitecture. It is fabricated on a 14 nm process, has a base frequency of 2.6 GHz and a turbo boost frequency of up to 3.5 GHz. The latency and

throughput of each instruction depend on the microarchitecture. Since clock time is the reciprocal of the clock frequency, every clock cycle takes $\frac{1}{2.6\ GHz} = 0.384\ ns$.

Each core has its own L1-instruction cache, L1-data cache and L2 cache, while the L3 cache is shared between all of the cores. L1 is the fastest and smallest. L3 is far slower, but biggest between the three.

Specification	Intel Core i7 6700HQ
Frequency	2.60GHz
Cycle time	1 / 2.60G = 0.38ns
Number of cores	4
Threads per core	2
Core speed	1994.6 MHz
L1 Data Cache	4 x 32 KB (8-way, 64-byte line)
L1 Instruction Cache	4 x 32 KB (8-way, 64-byte line)
L2 Cache	4 x 256 KB (4-way, 64-byte line)
L3 Cache	6 MB (12-way, 64-byte line)

Table 1: CPU specifications

2.2 Memory and I/O Bus

The machine possesses a single DDR4 8GB memory bank. The I/O bus type is PCIe 3.0.

Memory Size	8 GB
Memory type	DDR4
Memory bus type	DDR4-2133 (1066 MHz)
Max bandwidth	68.2 GB/s

Table 2: Memory specifications

2.3 Network

Name	Intel Dual Band Wireless-AC 8260
Bands	2.4, 5 GHz
Max speed	867 Mbps
Wifi	Wi-Fi 5 (802.11ac)

Table 3: Network specifications

2.4 Disk

The machine has two disks, one SSD and one HDD, and the descriptions of the disks are provided in Table 4 and 5. However, for the purpose of this project, we only use a 128 GB partition of the HDD disk since the operating system was installed on it.

Name	LITEON CV1-8B128
Type	SSD
Capactity	119.2 GB
Bus Type	SATA (II)

Table 4: SSD specifications

Name	HGST HTS721010A9E630
Type	HDD
Capacity	931.5 GB
Bus Type	SATA (II)
Rotation Speed	7200 RPM
Buffer Size	32768 KB
Media Transfer Rate	160.5 MB
Average Latency	4.2ms

Table 5: HDD specifications

2.5 Operating System and Architecture

The machine runs on Ubuntu, a modern operating system which is a Linux distribution based on Debian.

Operating System	Ubuntu
Version	22.04.03 LTS
Hardware Platform	x86_64
Page Table	32448 kB
Byte Order	Little Endian
Address Sizes	39 bits physical, 48 bits virtual

Table 6: Operating System and Architecture

3 CPU, Scheduling and OS Services

3.1 Measurement Overhead

Measurement overhead refers to the extra computational resources and time introduced by the process of measuring code execution time or clock cycles. Since it affects measurement accuracy, we want to account for this overhead in the proceeding experiments while calculating software and hardware performance.

3.1.1 Methodology

Intel offers `RDTSC` and `RDTSCP` instructions to help developers with accurate profiling of their code. `RDTSC` reads the current value of the processor's time-stamp counter into the specified registers. `RDTSCP` returns the current processor's ID in addition to reading the time-stamp counter. However, `RDTSC` is not a serializing instruction i.e. it does not necessarily wait until all previous instructions have been executed before reading the counter. To circumvent the problem of out-of-order execution, we add a `CPUID` instruction (which returns the details

of the processor) right before `RDTSC` since it is a serializing instruction and one with least overhead of its own. To restrict ourselves to measuring the measurement overhead, we simply record the timestamp counter twice without doing any extra operations. The start counter is subtracted from the final counter to calculate the number of cycles that have elapsed.

3.1.2 Estimates

We estimate that the hardware overhead of reading the timestamp registers and calculating the elapsed time is 10+ cycles accounting for the extra `MOV` and `CPUID` instructions we add for synchronization. Software overhead can be 5 cycles to account for the bitwise operations for forming the 64 bit time stamp counter from the two separate 32 bit registers.

3.1.3 Results

- Mean: 35.72 cycles or 13.70ns
- Standard Deviation: 1.55 cycles or 0.59ns

Base hardware performance	3.85+
Estimate of software overhead	1.50+
Prediction of operation time	5.35+
Measured operation time	13.70

Table 7: Measurement Overhead (nanoseconds)

3.1.4 Analysis

Our results show that we underestimated the overhead of our timing mechanism by a small margin. We believed that `CPUID` and `RDTSCP`, which is a serializing version of RDTSC, executes in a handful of cycles only.

3.2 Loop Overhead

Loop overhead refers to the additional computational resources and time consumed by the loop control and iteration management operations within a loop structure such as incrementing loop counters, evaluating loop conditions, and branching.

3.2.1 Methodology

Building on the same method for calculating measurement overhead, we simply introduce an empty loop between the two points where we record the time counter. We ensure the loop is not optimized out by setting the `-O0` flag while compiling.

3.2.2 Estimates

We estimate the loop overhead for one iteration to be roughly 5 or less cycles after inspecting the resulting assembly code.

3.2.3 Results

- Mean: 6995.2 cycles or 2675.07ns
- Standard Deviation: 525.49 cycles or 202.11ns
- Loop overhead = (Time taken to run `n` iterations of a emptyloop - Measurement Overhead) / `n`

$$(2675.07 - 13.70)/1000 = 2.66ns$$

Base hardware performance	-
Estimate of software overhead	5
Prediction of operation time	5
Measured operation time	2.66

Table 8: Loop Overhead (nanoseconds)

The measured loop overhead was 7 cycles which is equal to 2.66ns.

3.2.4 Analysis

A loop would not add any hardware overhead. In terms of software overhead, we estimate that 5 additional cycles will be required for each iteration of the loop. This number was arrived at by examining the assembly code that implements the loop and assuming that each instruction executes in a single cycle.

3.3 Procedure call overhead

Procedure call overhead is the additional computational resources and time required to initiate, manage, and return from a function or subroutine call in a program, encompassing things like parameter passing, setting up stack, saving registers, and branching.

3.3.1 Methodology

Building on the method for calculating loop overhead, we now call eight different functions with an empty body. Each function has a different number of integer arguments from 0 to 7. We assume that an empty body is a good approximation for calculating the procedure call overhead itself since it does not involve any computation which could pollute our measurements.

3.3.2 Estimates

After inspecting the resulting assembly code, we estimate an additional overhead of 1 clock cycle for every additional argument added to the procedure call. It roughly translates to 0.38ns additional latency per argument and we think procedure call with no arguments should take about 0.1ns.

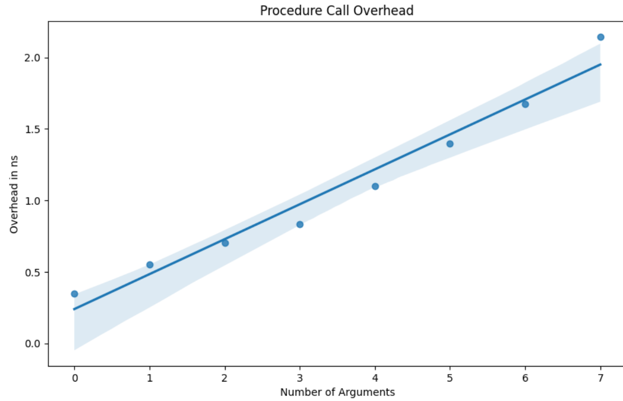

Figure 1: Procedure Call Overhead Time

3.3.3 Results

Following are the observations for trials using different number of arguments:

Argument Count	Mean	Standard Deviation
0	0.350	0.051
1	0.554	0.017
2	0.706	0.029
3	0.835	0.036
4	1.098	0.013
5	1.399	0.008
6	1.676	0.009
7	2.144	0.008

Table 9: Procedure Call Overhead (nanoseconds)

On fitting a regression on the results, we obtained

$$y = 0.244x + 0.24 \tag{1}$$

where y refers to the procedure call overhead (in ns) and x refers to the number of arguments passed to the procedure. The increment overhead of an argument is 0.244 ns.

Base hardware performance	-
Estimate of software overhead	$0.38x + 0.1$
Prediction of operation time	$0.38x + 0.1$
Measured operation time	$0.244x + 0.24$

Table 10: Procedure Call Overhead (nanoseconds)

3.3.4 Analysis

The provided data shows that the procedure call overhead increases with the number of arguments in a generally consistent and precise manner. The overhead is lowest with zero arguments (0.350) and increases incrementally as arguments are added, with some variation in the rate of increase.

3.4 System Call

System call overhead is the additional computational resources and time required to initiate, manage, and return from a request made by a user-level program to the operating system's kernel and includes steps such as transitioning from user mode to kernel mode, passing parameters, and executing the requested kernel service.

3.4.1 Methodology

Building on the method for calculating loop overhead, we now call `syscall(SYS_getpid)` between the two points where we record the time counter.

3.4.2 Estimates

`getpid()` is a simple system call. It only retrieves the process ID (PID) from a readily accessible register within the process's context. This suggests minimal kernel involvement and potential for fast execution. Arkanis Development measured a latency of 120ns for `clock_gettime` and 220ns for `write` on Linux (2017) [6]. While not identical since the author cites vDSO optimizations, these calls involve similar data access and suggest a similar ballpark for `getpid()`. To exclude the caching effect, we think `getpid()` should be upwards of 1000ns.

3.4.3 Results

- Mean: 1.31 μs
- Standard Deviation: 0.045 μs

Base hardware performance	-
Estimate of software overhead	1+
Prediction of operation time	1+
Measured operation time	1.31

Table 11: System Call Overhead (microseconds)

3.4.4 Analysis

Our estimate was right. System calls are costly owing from the need for the processor to switch between these two distinct privilege levels, involving a change in execution mode and a switch in memory context. Comparing them with procedure call, procedure calls are lightweight operations within a program, taking only a few nanoseconds even with arguments. System calls, on the other hand, are heavyweight interactions with the operating system. System calls are almost 1000x slower than a procedure call with no arguments!

3.5 Task Creation

Process creation refers to the time required to fork a child process from a parent. It includes tasks such as duplicating the parent's address space, file descriptors, and other process-related attributes. The `fork` system call creates a nearly identical copy of the parent process, resulting in relatively higher overhead compared to thread creation. Thread creation involves tasks like allocating a new thread control block, initializing the thread's data structures, and setting up its stack.

3.5.1 Methodology

To measure process creation, we employed the `fork` system call, which creates a duplicate of the existing process. We read the time counter before and after each `fork` call to determine the time it took for each `fork` operation. For thread creation, we utilized the `pthread_create` function from the `pthread` library. Like with process creation, we calculated the difference between time counters before and after `pthread_create` to measure the time consumed by each thread creation. Both, the child process and child thread contained no code to ensure that the measurements were not affected by additional factors.

3.5.2 Estimates

No earlier experiments allow us to make an educated guess about running time to create a process or a thread. However, relying on an observation in the Plan 9 operating system about a fork being nearly 200 times as long as a system call, we estimate process creation to take about 200 μs [15].

Creating a thread is generally less resource-intensive than creating a new process since threads within the same process share the same memory space and resources. So thread creation should take significantly lower time than process creation. We estimate it to take $1/10^{th}$ of the process creation time.

3.5.3 Results

Process

- Mean: 567.67μs
- Standard Deviation: 27.64μs

Base hardware performance	-
Estimate of software overhead	200
Prediction of operation time	200
Measured operation time	567.67

Table 12: Process Creation (microseconds)

Thread

- Mean: 122.87μs
- Standard Deviation: 2.98μs

Base hardware performance	-
Estimate of software overhead	20
Prediction of operation time	20
Measured operation time	122.87

Table 13: Thread Creation (microseconds)

3.5.4 Analysis

Our findings revealed that process creation is considerably more time-consuming than thread creation, by a factor of 4. When using the `fork` system call, the operating system has to duplicate a substantial amount of data to create a new process.

In contrast, when it comes to thread creation, the operating system does not need to replicate all the data from the calling thread. This is because the caller and the new thread will share the same memory address space. This sharing of resources saves a significant amount of time. Additionally, the operating system may recycle threads.

3.6 Context Switch

Context switch time is the time needed to save the state of one process/thread and restore the state of another process/thread.

3.6.1 Methodology

To measure the time it takes for a context switch to occur, we devised a method using pipes to reliably switch between a parent process or thread and its child, and vice versa. In this method, the parent process writes to the first pipe and waits for a read on the second pipe. When the OS detects the first process waiting for data on the second pipe, it puts the first process in a blocked state and switches to the other process. This second process reads from the first pipe and then writes to the second pipe. When the second process attempts to read from the first pipe, it also blocks, leading to a continuous back-and-forth communication cycle. By repeatedly measuring the cost of this communication pattern, we can estimate the context switch time accurately. The result needs to be divided by two since what we measure is essentially two context switches.

3.6.2 Estimates

Based on the provided estimates from [3], a context switch can cost at least 30 μs of CPU overhead. We used this number as an estimate for the thread context switch. A process context

switch should be several times longer than a thread context switch. Thus, we estimate the process context switch to be 3 times longer, which amounts to 90 μs.

3.6.3 Results

Process

- Mean: 78.43 μs
- Standard Deviation: 0.90 μs

Base hardware performance	-
Estimate of software overhead	90
Prediction of operation time	90
Measured operation time	78.43

Table 14: Process Context Switch (microseconds)

Thread

- Mean: 41.46 μs
- Standard Deviation: 0.19 μs

Base hardware performance	-
Estimate of software overhead	30
Prediction of operation time	30
Measured operation time	41.46

Table 15: Thread Context Switch (microseconds)

3.6.4 Analysis

These findings indicate that the time taken for context switching between processes and threads falls within 10 to 100 microseconds. Although this time is less than what is required for creating a new process or thread, the overhead involved in context switching remains significant in today's computing contexts.

4 Memory

4.1 Memory Access Time

Memory access time refers to the time it takes for a CPU to retrieve data from a specific location in the computer's memory hierarchy (L1, L2, L3 cache and main memory). The memory access time is affected by various factors, including the type of memory technology used (e.g., DDR4 RAM), the distance between the CPU and the memory module, the module's latency, and the efficiency of the memory controller.

4.1.1 Methodology

Following the methodology proposed in lmbench [9] for measuring average memory access time, we read repeated references to integer arrays of varying sizes using multiple strides. The stride, representing the distance between consecutive elements during array traversal, was updated to observe the influence of spatial locality on memory access. In this context, memory size denotes the total allocated memory, with larger sizes potentially leading to increased cache misses and impacting memory traversal efficiency.

For each combination of stride size and memory size, we allocated memory, created an integer array with specified links between elements based on the chosen stride, and traversed the array in a loop for one million iterations. We aimed to minimize the impact of cache line sizes and memory prefetching on the measurements. We intend to measure the back-to-back-load time, representing the time each load operation takes, assuming that the instructions before and after are also cache-missing loads.

4.1.2 Estimates

Based on the provided estimates from [18], we estimate the different components of the hierarchy to be in the range highlighted in Table 16.

Level	Access Time	Typical Size
L1 (on-chip)	2-8 ns	8 KB - 128 KB
L2 (off-chip)	5-12 ns	0.25 MB - 8 MB
L3 (off-chip)	12-30 ns	4 MB - 32 MB
Main Memory	10-60 ns	64 MB - 16 GB

Table 16: Estimated Memory Access Time

4.1.3 Results

Level	Access Time
L1	3.41
L2	6.29
L3	10.63
Main memory	14.80

Table 17: Memory Access Time (nanoseconds)

4.1.4 Analysis

Each data set represents a stride size, with the array size varying from 256 KB (2^8 bytes) up to 1 GB (2^{30} bytes). The graph contains four horizontal plateaus, where each plateau represents a level in the memory hierarchy from L1 to main memory. The point where each plateau ends and the line rises

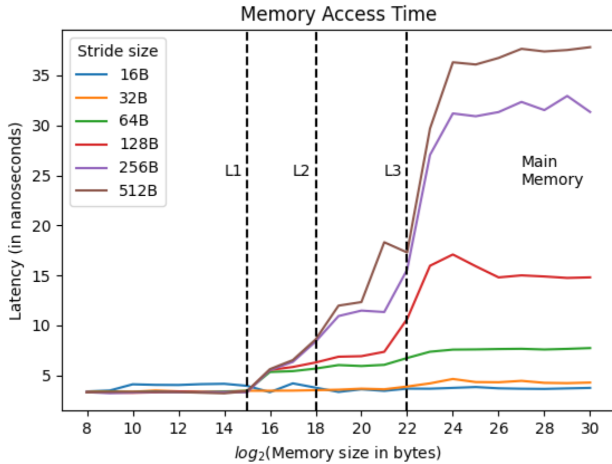

Figure 2: Memory Access Time

marks the end of that portion of the memory hierarchy. There is a rise in latency between different hierarchies because each higher level cache is likely to be further away from the CPU, shared between multiple cores (L1 vs L2) and using a different technology (e.g. SRAM, DRAM). Thus, when the array size exceeds the size limit of the current cache level, the excess data will be stored in the next cache level. This explains the increasing trend in the latency with sudden increases marking the end of current cache level and beginning of the next one.

The three vertical lines correspond to the size of L1, L2 and L3 cache, which are 32 KB, 256 KB, and 6 MB respectively. This observation matches exactly with our machine specification listed in Table 1.

4.2 Memory Bandwidth

Memory bandwidth refers to the rate at which data can be read from or written to the RAM. Several factors influence memory bandwidth, including the type of RAM (e.g., DDR4, DDR5), the memory bus speed, the width of the memory bus, and the overall memory architecture of the computer system.

4.2.1 Methodology

To profile memory bandwidth, we adopted the approach outlined in the lmbench paper [9]. To saturate memory bandwidth, we prepared an integer array approximately six times the size of the L3 cache. We then sequentially accessed the array elements, with a 64-byte stride. The stride was chosen such that it exceeded the cache line size, thus preventing cache hits to ensure accurate measurements. To minimize loop overhead and reduce branch instructions, the loop was manually unrolled 2^8 times. The first element in each cache line was utilized for the read/write operation, mitigating the impact of cache line pre-fetching.

For measuring read bandwidth, the values of the first elements of each cache line in the large integer array were summed. Similarly, to measure write bandwidth, a constant value was stored in the first element of each cache line in the array.

4.2.2 Estimates

According to the codearcana blog post [16], Bandwidth = DRAM Clock Frequency * Memory Bus Width * Number of lines. For our testing machine's DDR4 memory, we have clock frequency = 1066MHz, 2 lines and 64bits wide memory bus. Using these numbers we can calculate the test machine's memory bandwidth as 1066Mhz * 64 / 8 * 2 = 17.06 GB/s. However, this is the theoretical bandwidth and we estimate the real bandwidth to be much lower, somewhere around 25% of the theoretical bandwidth.

We expected the write bandwidth to be roughly half of the read bandwidth due to the cache read and write policies, where a cache line is read into the cache before a modified value is written back to memory. This anticipation arises from the increased memory traffic and potential obfuscation of memory write test results caused by accessing the system cache and reading memory into the cache during write operations.

4.2.3 Results

Memory Read Bandwidth

- Mean: 3.59 GB/s
- Standard Deviation: 69.62 MB/s

Memory Write Bandwidth

- Mean: 1.88 GB/s
- Standard Deviation: 20.11 MB/s

Base hardware performance	17.06
Estimate of software overhead	75%
Prediction of operation time	4.26
Measured operation time (Read)	3.59
Measured operation time (Write)	1.88

Table 18: Memory Bandwidth (in GB/s)

4.2.4 Analysis

In our experiments, standard loop read and write operations consistently yielded a bandwidth significantly below the theoretical maximum, attributing this to the impact of cache read and write policies. Notably, modern CPUs, during write operations involving data sizes below the cache line size (64 bytes),

necessitate reading the entire cache line from main memory, modifying it, and subsequently writing it back. This dual read and write process during a seemingly write-focused operation resulted in a significantly lower than expected bandwidth.

This discrepancy aligns with findings by Alex Reece [16], who highlighted challenges in saturating entire memory bandwidth with a single-threaded program on a single core. Reece achieved higher bandwidth using OpenMP and multiple threads across multiple cores, emphasizing the potential for increased efficiency. While our study did not explore multi-threaded approaches, the observed bandwidth reflects the complexities of fully utilizing memory bandwidth with a single-threaded program.

4.3 Page Fault Service Time

Page fault service time represents the time it takes to handle a page fault. A page fault occurs when a program attempts to access a page of virtual memory that is not currently in the physical RAM. A page fault involves the operating system interrupting the CPU, identifying the required page not in physical RAM, initiating disk I/O to fetch the page, potentially replacing a page in RAM if needed, and updating page tables to reflect the new mapping.

4.3.1 Methodology

We triggered page faults by reading one byte from the initial page of newly mapped pages since `mmap()` performs lazy file-reads. Notably, the first run consistently produced significantly higher results, attributed to system caching effects. To address this, we executed the `sudo sh -c "echo 3 > /proc/sys/vm/drop_caches"` command before each test to clear the cache, ensuring a more consistent and reliable evaluation of page fault metrics.

4.3.2 Estimates

On a major page fault, the operating system needs to bring the required page into the RAM from the secondary storage (usually a hard disk or SSD). Hard disk drives have an average rotational latency of 3 ms, a seek time of 5 ms, and a transfer time of 0.05 ms/page. The software overhead is negligible compared to the hardware overhead. Therefore, we estimate the total time for paging to take approximately 8 ms.

4.3.3 Results

- Mean: 8.21 ms
- Standard Deviation: 0.14 ms

Base hardware performance	8.05
Estimate of software overhead	0.05
Prediction of operation time	8.10
Measured operation time	8.213

Table 19: Page Fault Service Time (milliseconds)

4.3.4 Analysis

The experiment closely matches our estimate. This enunciates the relatively slow nature of page fault handling, making it a noteworthy factor to consider in applications where minimizing access times is crucial. Dividing the page fault service time by the size of page, we get $8213838/4096 = 2005$ ns per byte. Comparing with main memory access time for a byte (14.80 ns), page fault service time per byte is $2005/14.80 = 135$x the memory access time for a single byte!

5 Network

We used a MacBook Pro laptop as our remote test machine in experiments which required a remote setup. Table 20 lists the technical specifications of this machine.

Specification	Apple M1 Pro
Frequency (performance cores)	3228 MHz
Frequency (efficiency cores)	2064 MHz
Number of cores	8
Threads per core	2
Core speed	1994.6 MHz
L1 Data Cache (per core)	192 KB
L1 Instruction Cache (per core)	128 KB
L2 Cache	24 MB
L3 Cache	24 MB
Memory Size	16 GB
Memory type	LPDDR5
Max bandwidth	200 GB/s
Network Bands	2.4, 5 GHz
Maximum network speed	1200 Mbps
Wifi	802.11ax Wi-Fi 6
Disk Name	APPLE SSD AP0512R
Disk Type	SSD
Disk Capactity	494.38 GB
Disk Bus Type	PCIe
Operating System	macOS Monterey
OS Version	12.04

Table 20: Remote Machine Specifications

5.1 Round Trip Time

Round Trip Time (RTT) refers to the total time taken for a signal or packet to travel from a source to a destination and back again.

5.1.1 Methodology

The client initiates a TCP connection with the server, transmitting a fixed-size message in a ping-pong fashion. The time elapsed between sending the data and receiving an acknowledgment is recorded for each iteration over a set number of repetitions. Both the client and server programs are designed to simulate both loopback and remote connections. For loopback, both processes run on the same machine, while for remote, the server operates on a separate machine within the same wireless network.

The experiments involve 10 repetitions. We utilize a predefined message size, specifically a 44-byte message, to mimic the typical data size in a `ping` packet after accounting for headers. The TCP connection logic involves socket creation, server address setup, connection establishment, and iterative ping-pong exchanges. The server runs indefinitely, simulating a continuous communication scenario.

5.1.2 Estimates

Given the simplicity of the round trip time (RTT) estimation process and the comparable nature of ping time, we expect both metrics to be on the same order. In the context of a time-varying network environment, it's challenging to predict which one will be larger, but we anticipate their magnitudes to align closely. Additionally, the straightforward nature of this operation suggests that software overhead can be considered negligible.

Considering the nature of round trip time (RTT), the loopback interface, being a virtual network interface with no physical network involvement, is expected to have minimal hardware overhead. Given this, a reasonable estimate for loopback RTT is around 0.1 ms.

For the remote interface, involving data transmission through the network card, routers, and server network card, the ideal RTT is influenced by the link speed of the WIFI network, approximately 800 Mb/s. While the theoretical RTT for transferring 64 bytes data one way is under 0.1 ms, practical considerations suggest an estimate between 1 and 10 ms. The additional latency is primarily attributed to router lookup and packet forwarding in the network, contributing to the overall round trip time.

5.1.3 Results

Loopback Interface

- Mean: 0.062 ms
- Standard Deviation: 0.002 ms

Base hardware performance	≈ 0
Estimate of software overhead	0.1
Prediction of operation time	0.1
Measured operation time	0.062

Table 21: Round Trip Time for Loopback Interface (milliseconds)

Remote Interface

- Mean: 4.48 ms
- Standard Deviation: 0.11 ms

Base hardware performance	1-10
Estimate of software overhead	≈ 0
Prediction of operation time	1-10
Measured operation time	4.48

Table 22: Round Trip Time for Remote Interface (milliseconds)

5.1.4 Analysis

The round trip time we get here is comparable to the ping time. Thus, we believe our result is acceptable.

When analyzing the data from both remote and loopback interfaces, it becomes apparent that the overhead from the operating system's network code is relatively minor, especially when compared to the usual network latency. The loopback interface exhibits a round-trip time (RTT) of 0.062 ms, which serves as a reasonable proxy for the OS overhead, considering that the data packet merely travels down and then back up the network stack without any actual transmission. In contrast, the RTT observed on the remote interface is 4.48 ms, attributed to the Local Area Network's characteristics. However, for typical RTT over the internet, latencies ranging from 100ms to 1s are quite common.

The study referenced in [7] notes that the bidirectional ping latency for two machines connected via Wi-Fi on the same route typically ranges between 4-5 ms. This figure aligns well with the round-trip time (RTT) we observed for the remote interface in our tests.

5.2 Peak Bandwidth

Peak bandwidth refers to the maximum data transfer rate or capacity that a communication channel or network can achieve under ideal conditions. It is typically expressed in bits per second (bps) and serves as a theoretical upper limit, often determined by the protocols and hardware components involved in the network infrastructure.

5.2.1 Methodology

We measure peak bandwidth through a custom client-server setup. The client initiates a TCP connection with the server, and the server sends a large message in a loop to simulate continuous data transfer. We record the time taken for the client to receive the entire message. The chosen message size (1 MB) and repetition count (10,240) stress-test the network. Bandwidth, expressed in megabytes per second, is derived by dividing the total bytes by the total time.

5.2.2 Estimates

To estimate peak bandwidth, we can leverage the maximum TCP window size divided by the round-trip time (RTT) for the given path [19]. In the case of the loopback interface, with an average RTT of 0.062 ms and a TCP window size of 64 KB, the calculated predicted peak bandwidth is approximately 1032.25 MB/s.

For remote interface, we estimated the bandwidth based on the theoretical max bandwidth of our network link. We ran our experiment using a LAN and our machine theoretical max bandwidth is 867 Mbps = 108.37 MB/s.

For software overhead, we estimate it to be around 50%. Thus, the final predicted bandwidth is only a half of the base hardware performance.

5.2.3 Results

Loopback Interface

- Mean: 438.71 MB/s
- Standard Deviation: 8.16 MB/s

Base hardware performance	1032.25
Estimate of software overhead	50%
Prediction of operation time	516.12
Measured operation time	438.71

Table 23: Peak Bandwidth for Loopback Interface (MB/s)

Remote Interface

- Mean: 28.18 MB/s
- Standard Deviation: 0.34 MB/s

Base hardware performance	108.37
Estimate of software overhead	50%
Prediction of operation time	54.18
Measured operation time	28.18

Table 24: Peak Bandwidth for Remote Interface (MB/s)

5.2.4 Analysis

In our experiment, we utilized two laptops linked to the same Local Area Network (LAN). Several factors might explain why we were unable to reach the theoretical maximum bandwidth. One such factor could be the router's processing capacity. Despite its higher theoretical bandwidth limit, the router's CPU might act as a limiting factor. Additionally, during our tests, the router was simultaneously being used by others to access the internet, which would have consumed a portion of the available bandwidth.

Upon comparing outcomes from both the remote and loopback interfaces, it's evident that the operating system's network code overhead is considerably less impactful than the time required to transmit data packets over a real network connection. In remote scenarios, the primary constraints are often the physical network cards of the machines or the network links themselves. The bandwidth observed for the loopback interface, at 438.71 MB/s, offers a benchmark for the network code's capacity to process and construct network packets. However, typical internet network bandwidths may be lower, often due to congestion from multiple users sharing the same network.

The theoretical maximum bandwidth for our machine's network card is 867 Mbps, which translates to 108.37 MB/s. When compared to our findings, it's clear that we aren't fully utilizing the network link. This discrepancy between theoretical and actual performance is common and can be attributed to several factors, including network overheads and quality and age of the network card and other network hardwares used in the experiment too.

5.3 Connection Overhead

TCP is a reliable, connection-oriented protocol that ensures data integrity and delivery by establishing a virtual connection between the sender and receiver. Connection overhead during setup involves a three-way handshake (the exchange of SYN, SYN-ACK, and ACK packets) between the client and server to establish a TCP connection, introducing latency and utilizing additional network resources. During teardown, it involves a four-way handshake to gracefully terminate the connection.

5.3.1 Methodology

To assess connection overhead, we measure the time taken by the `connect()` and `close()` function calls, without the actual data transfer being necessary in this test, and we iterate this process 100 times for comprehensive analysis.

5.3.2 Estimates

In evaluating the overhead for setting up a connection, we only measured the time taken by `connect()` calls. This measurement ends when the client acknowledges the SYN-ACK

message from the server, resulting in the termination of the function call. The time it takes for the server to acknowledge this message was not included in our measurement. As a result, we believe that our calculations represent one Round-Trip Time (RTT). From our previous experiments, we have deduced that the setup time for a loopback interface is approximately 0.062 ms, and for a remote interface, it is around 4.48 ms.

For connection teardown overhead, client only needs to send FIN message to the server and does not need to wait for any reply from the server. Therefore, our estimation of the teardown time is primarily derived from the software overhead, which we approximate to be about 0.01 ms.

5.3.3 Results

Loopback Interface

- Mean: 39 μs
- Standard Deviation: 1 μs

	Setup	Teardown
Base hardware performance	62	0
Estimate of software overhead	10	10
Prediction of operation time	72	10
Measured operation time	28	11

Table 25: Connection Overhead for Loopback Interface (microseconds)

Remote Interface

- Mean: 7.061 ms
- Standard Deviation: 0.154 ms

	Setup	Teardown
Base hardware performance	4.48	0
Estimate of software overhead	0.01	0.01
Prediction of operation time	4.49	0.01
Measured operation time	6.99	0.06

Table 26: Connection Overhead for Remote Interface (milliseconds)

5.3.4 Analysis

Analyzing the results from both remote and loopback interfaces reveals that the operating system's network code overhead is relatively minor compared to the time required for sending SYN or FIN packets to the server. In a remote setup, the latency closely mirrors the round-trip time. On the other hand, the loopback setup shows that the time taken to establish a connection on the loopback interface, at 0.028 ms, can serve as a good estimate for the duration the OS takes to set up a new socket for the client's connection. During the teardown phase, the timings are more aligned in both scenarios, as there's no waiting involved for a response in either measurement.

Regarding the outcomes observed with the remote interface, we must consider the extra overhead associated with data transmission. Additionally, variations in network traffic at the time of our experiments could account for some differences in the results. Despite these potential discrepancies, the outcomes and our estimates are roughly within the same order of magnitude, suggesting that our findings are reasonably reliable.

6 File System

6.1 Size of File Cache

A file cache is used to store frequently accessed data from the file system in memory, reducing the need to repeatedly read from or write to the slower storage devices (such as hard drives or SSDs). The size of the file cache is dynamically managed by the operating system based on factors like available system memory, the demand for other processes, and the access patterns of different files.

6.1.1 Methodology

In this experiment, we aim to measure the system's maximum file cache size by reading the same large file at varying lengths to investigate the file cache effect. Initially, we read the experimental file, placing it into the main memory. Subsequent reads of data blocks from the same file benefit from the file being cached in memory, resulting in faster read times—a demonstration of the file cache effect.

However, when the file size surpasses the maximum file cache limitation, the entire file cannot fit into main memory. Subsequent reads lead to cache misses, requiring retrieval from the disk and significantly increasing data block read times. This abrupt change indicates that the experimenting file size equals or exceeds the file cache size for the testing machine. To pinpoint the precise maximum file cache boundaries, we iteratively decrease the file size.

The experimental setup involves measuring the maximum file cache size by reading data ranging from 1GB to 6GB file. Each experiment starts with an initial read to ensure the file's data is in main memory. Subsequently, we perform another read of the same file, calculating the average reading time per data block. We read the file backward instead of from the beginning. This decision is made to prevent the prefetching of disk blocks and to better isolate the impact of the file cache.

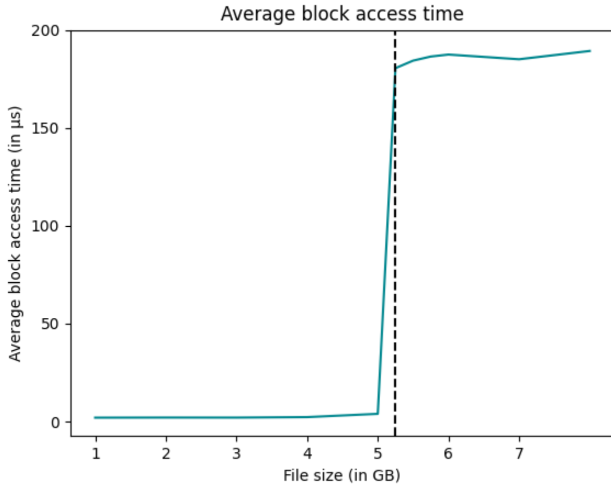

Figure 3: Size of File Cache

6.1.2 Estimates

Taking into account the 8GB main memory on the testing machine running Ubuntu, we can estimate the maximum file cache size by subtracting the memory space consumed by essential OS processes. Referring to the System Monitor on Ubuntu, it's observed that around 2GB is utilized by the OS and other user-level processes, along with approximately 1GB each for wired memory and compressed memory. This deduction leads to an approximation of an available file cache size in the range of 5-6GB. This estimation relies on the understanding that any unused RAM can potentially be allocated for file caching in the Ubuntu environment.

6.1.3 Results

Base hardware performance	8
Estimate of software overhead	2-3
Prediction of operation time	5-6
Measured operation time	5.2-5.3

Table 27: Size of File Cache (GB)

6.1.4 Analysis

The experimental results indicate a file cache size of approximately 5.2GB to 5.3GB. The methodology successfully identified a transition point around 5GB, distinguishing between file cache and disk reading. This provides a reasonable estimate of file cache boundaries with a clear change in reading times.

6.2 File Read Time

File read time refers to the duration it takes to access and retrieve data from a file stored on a storage device. It is typically influenced by factors such as storage device speed, file size, access patterns, and the efficiency of file and I/O subsystem implementations.

6.2.1 Methodology

The methodology involves configuring file access with specific flags, allocating buffers, setting up timing mechanisms, and implementing sequential and random access patterns. Sequential access entails reading the file block by block. Random access involves generating random offsets within the file and reading blocks at those offsets. The `O_DIRECT` flag ensures direct I/O by bypassing the file system cache, while `O_SYNC` guarantees synchronous operations, requiring data and metadata updates to be written to the storage device before system calls are considered complete. Data is collected by recording time per read operation, calculating average times. We close the file descriptor and free allocated resources for cleanup. To avoid any kind of file cache effects across runs, we created a new test file for every run.

6.2.2 Estimates

On average, disks take 0.85 ms to read 1 MB of data sequentially [17]. For our disk described in Table 5, the average latency measures the average time it takes for the disk to access a random piece of data on the platter, which is 4.2 ms in our case. It is a good estimate for the random read time. Since sequential is always faster, we estimate it to be 10x as fast as reading data randomly. Our estimate for sequential file access is 0.42 ms. Also, an important point often missed is "sequential" file access, although implying linear reading from start to end, can be influenced by factors like disk fragmentation, degrading performance. File system characteristics, and hardware buffers may make sequential reads seem more efficient.

6.2.3 Results

Sequential

- Mean: 0.212 ms
- Standard Deviation: 0.012 ms

Base hardware performance	0.42
Estimate of software overhead	-
Prediction of operation time	0.42
Measured operation time	0.21

Table 28: File Read Time for Sequential Access (milliseconds)

Random

- Mean: 4.935 ms
- Standard Deviation: 0.149 ms

Base hardware performance	4.20
Estimate of software overhead	-
Prediction of operation time	4.20
Measured operation time	4.935

Table 29: File Read Time for Random Access (milliseconds)

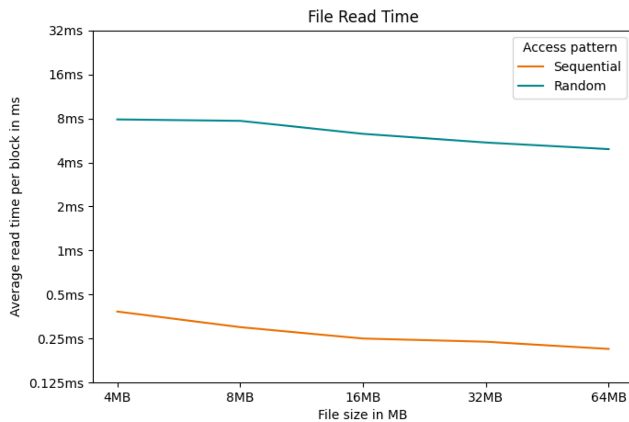

Figure 4: File Read Time

6.2.4 Analysis

For sequential access, the stable and consistent time measurements across various file sizes highlight the efficiency of HDDs in handling sequential data retrieval. The low deviation further supports the reliability of this methodology, showcasing the consistent nature of sequential reads on HDDs.

In the case of random access, the higher and stable time values compared to sequential access are a characteristic of HDD behavior. HDDs are inherently slower in responding to random access requests due to mechanical components, seek times, and potential file fragmentation. The absence of prefetching advantages in random access contribute to the observed higher time values.

6.3 Remote File Read Time

We measure and analyze the average per-block time for sequential and random file reads of various sizes on a remote file system to assess the network penalty. The exact impact of the network penalty depends on factors such as the network speed, latency, and the efficiency of the file transfer protocol.

6.3.1 Methodology

In our experiment, we utilized a MacBook Pro laptop as the remote test machine, which is connected to the same Local Area Network (LAN). Detailed specifications of this machine are provided in Table 20.

We configured an NFS server on the remote test machine, as outlined in [1], and subsequently mounted the shared directory on our main machine following the guidelines in [8].

However, there are notable differences to be considered:

- The remote test machine is equipped with an SSD, whereas our main machine utilizes an HDD.
- The NFS operates with a larger block size of 1MB, which could significantly affect transmission latency.

To ensure accuracy in our measurements, we cleared the file cache on both the client and server machines before each test to prevent the influence of cached data.

For assessing the network penalty, we compared the average time taken per block for sequential local file reads (performed on the same machine) with the average time per block for sequential remote file reads.

6.3.2 Estimates

We approximated that the read latency on our remote test machine, which utilizes an SSD, would be nearly identical to the sequential read time on our primary test machine. Consequently, our focus shifted to calculating the network penalty.

Our assessment identifies two principal factors contributing to the network penalty: the round-trip time and the data transmission time. Based on earlier network tests, we estimated the round-trip time at 4.48 ms. For the transmission time, we calculated it as 1MB (the NFS block size) divided by 28 MB/s, resulting in 35.71 ms. Therefore, we estimated the total network penalty to be 4.48 ms + 35.71 ms = 40.19 ms.

The total estimated time amounts to 0.21 ms + 40.19 ms = 40.4 ms. This estimation holds for both sequential and random access, as the use of SSDs eliminates any distinction between the two.

6.3.3 Results

Sequential

- Mean: 49.155 ms
- Standard Deviation: 3.449 ms

Random

- Mean: 49.843 ms
- Standard Deviation: 1.434 ms

Base hardware performance	40.4
Estimate of software overhead	≈ 0
Prediction of operation time	40.4
Measured operation time	49.1

Table 30: Remote File Read Time for Sequential Access (milliseconds)

Base hardware performance	40.4
Estimate of software overhead	≈ 0
Prediction of operation time	40.4
Measured operation time	49.8

Table 31: Remote File Read Time for Random Access (milliseconds)

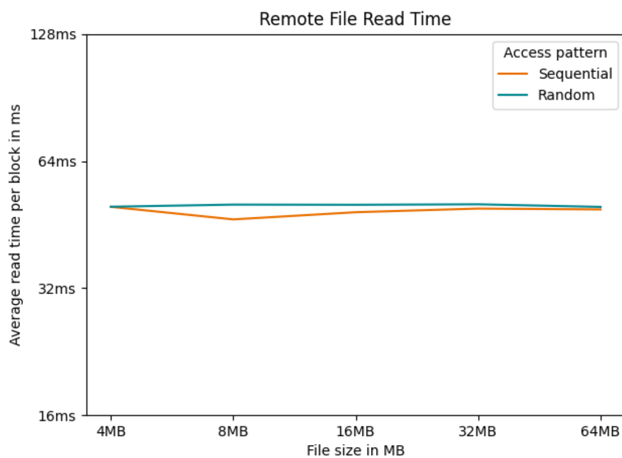

Figure 5: Remote File Read Time

6.3.4 Analysis

The observed times for sequential and remote reads are remarkably similar, aligning with the fact that the remote test machine employs an SSD, resulting in negligible disparity between sequential and remote reads.

To determine the network penalty, we deducted the average time for sequential file reading of remote files from the average time for sequential file reads locally. This calculation revealed a network penalty of 49.155 ms - 0.21 ms = 48.943 ms.

The calculated network penalty aligns with our initial estimate. The slight variation can likely be ascribed to changes in network traffic and the extra processing time as packets navigate through the network stack.

6.4 Contention

Contention refers to the competition for shared resources among concurrent processes or threads, often leading to interference and performance degradation. We measure and analyze the average time to read one file system block of data as the number of simultaneous processes performing the same operation on different files on the same disk increases, excluding the impact of the file buffer cache.

6.4.1 Methodology

To conduct this experiment, multiple processes simultaneously read distinct 64MB files on the same disk. Each process reads a different file. Similar to the previous experiment, the file caching is bypassed by setting `O_DIRECT | O_SYNC` flags. The experiment measures file block read times against the number of concurrently reading processes for both sequential and random reads.

6.4.2 Estimates

As contention rises, wait times may increase. In the case of random reads, which involve non-sequential access patterns, contention is likely to lead to more significant increases in disk seek times. Since contention degrades performance, we expect the sequential and random access time to read one file system block of data to be much worse than when read without contention. For 10 processes, we estimate the latency with contention to be approximately 10x the sequential access and random access latency for reading a file, i.e. 2.1 ms and 49.3 ms respectively.

6.4.3 Results

The following results are for contention between 10 processes.
Sequential

- Mean: 4.50 ms
- Standard Deviation: 0.52 ms

Base hardware performance	2.10
Estimate of software overhead	-
Prediction of operation time	2.10
Measured operation time	4.50

Table 32: Contention for Sequential Access (milliseconds)

Random

- Mean: 26.61 ms
- Standard Deviation: 0.89 ms

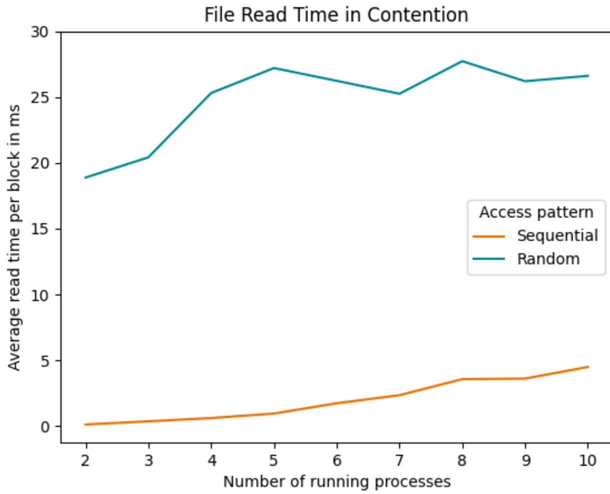

Figure 6: Contention

Base hardware performance	49.3
Estimate of software overhead	-
Prediction of operation time	49.3
Measured operation time	26.61

Table 33: Contention for Random Access (milliseconds)

6.4.4 Analysis

When examining sequential access results, the file block access time increases as the number of simultaneous read operations grows. This is attributed to the diminishing effect of prefetching as more processes are introduced. In scenarios with a single process, prefetching is effective as data blocks can be stored in main memory. However, as processes reading different files increase, newly read data blocks may replace those read by previous processes, undermining prefetching efficiency. Consequently, a stable reading time is observed after a certain point as the diminishing impact of prefetching becomes apparent.

For random access, as the number of processes increases, random access time gradually rises. Disk controller can also reorder read requests for efficiency, which might improve the performance slightly as seen in the graph for a few data points.

Section	Operation	Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time
CPU, Scheduling, and OS Services	Measurement Overhead	3.85 ns	1.50 ns	5.35 ns	13.70 ns
	Loop Overhead	-	5.00 ns	5.00 ns	2.66 ns
	Procedure Call (n args)	-	$0.38n + 0.1$ ns	$0.38n + 0.1$ ns	$0.244n + 0.24$ ns
	System Call	-	1 μs	1 μs	1.31 μs
	Process Creation	-	200 μs	200 μs	567.67 μs
	Thread Creation	-	20 μs	20 μs	122.87 μs
	Process Context Switch	-	90 μs	90 μs	78.43 μs
	Thread Context Switch	-	30 μs	30 μs	41.46 μs
Memory	L1 Access Time	2-8 ns	-	2-8 ns	3.41 ns
	L2 Access Time	5-12 ns	-	5-12 ns	6.29 ns
	L3 Access Time	12-30 ns	-	12-30 ns	10.63 ns
	Main Memory Access Time	10-60 ns	-	10-60 ns	14.80 ns
	Read Bandwidth	17.06 GB/s	75% overhead	4.26 GB/s	3.59 GB/s
	Write Bandwidth	17.06 GB/s	75% overhead	4.26 GB/s	1.88 GB/s
	Page Fault Service	8.05 ms	0.05 ms	8.10 ms	8.21 ms
Network	Loopback Round Trip	≈ 0	0.1 ms	0.1 ms	0.06 ms
	Remote Round Trip	1-10 ms	≈ 0	1-10 ms	4.48 ms
	Loopback Bandwidth	1032 MB/s	50% overhead	516 MB/s	438 MB/s
	Remote Bandwidth	108 MB/s	50% overhead	54 MB/s	28 MB/s
	Loopback Connection Overhead	62 μs	20 μs	80 μs	39 μs
	Remote Connection Overhead	4.48 ms	0.02 ms	4.5 ms	7.06 ms
File System	File Cache Size	8 GB	2-3 GB	5-6 GB	5.2 GB
	Sequential File Read	0.42 ms	-	0.42 ms	0.21 ms
	Random File Read	4.2 ms	-	4.2 ms	4.93 ms
	Sequential Remote File Read	40.4 ms	≈ 0	40.4 ms	49.15 ms
	Random Remote File Read	40.4 ms	≈ 0	40.4 ms	49.84 ms
	Sequential Access Contention (10 processes)	2.10 ms	-	2.10 ms	4.50 ms
	Random Access Contention (10 processes)	49.30 ms	-	49.30 ms	26.61 ms

Table 34: Summary

References

[1] Enabling network (NFS) shares in Mac OS X. URL: https://archive.li/67HS2.

[2] Gigabyte P55W v5 - full specs, details and review. URL: https://www.productindetail.com/pn/gigabyte-p55w-v5.

[3] How long does it take to make a context switch? URL: https://archive.li/kmLwF.

[4] Intel Core i7-6700HQ Mobile processor. URL: https://www.cpu-world.com/CPUs/Core_i7/Intel-Core%20i7-6700HQ%20Mobile%20processor.html.

[5] Laptop Specification - GIGABYTE U.S.A. URL: https://www.gigabyte.com/us/Laptop/P55W-v5/sp#sp.

[6] Measurements of system call performance and overhead - Arkanis Development. URL: https://archive.li/Eovsy.

[7] The Benefits of OFDMA for Wi-Fi 6, publisher=Qualcomm Technologies, Inc. (QTI). URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/ofdma_white_paper.pdf.

[8] Brian Boucheron and Tony Tran. How To Set Up an NFS Mount on Ubuntu 22.04, Apr 2022. URL: https://www.digitalocean.com/community/tutorials/how-to-set-up-an-nfs-mount-on-ubuntu-22-04.

[9] Aaron B. Brown and Margo I. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel X86 Architecture. *SIGMETRICS Perform. Eval. Rev.*, 25(1):214–224, jun 1997. doi:10.1145/258623.258690.

[10] J. B. Chen, Y. Endo, K. Chan, D. Mazieres, A. Dias, M. Seltzer, and M. D. Smith. The Measured Performance of Personal Computer Operating Systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 299–313, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224056.224079.

[11] Agner Fog. Instruction tables. *Technical University of Denmark*, 2018.

[12] Torbjörn Granlund. Instruction latencies and throughput for AMD and Intel x86 Processors. *Technical report, KTH*, 2012.

[13] John Ousterhout. Always Measure One Level Deeper. *Commun. ACM*, 61(7):74–83, jun 2018. doi:10.1145/3213770.

[14] John K Ousterhout. Why aren't operating systems getting faster as fast as hardware?, Jun 1990. URL: https://web.stanford.edu/~ouster/cgi-bin/papers/osfaster.pdf.

[15] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. *Comput. Syst.*, 8, 07 2000.

[16] Alex Reese. Achieving maximum memory bandwidth, May 2013. URL: https://archive.li/W4Ad4.

[17] Colin Scott. Latency numbers every programmer should know. URL: https://colin-scott.github.io/personal_website/research/interactive_latency.html.

[18] Jon Stokes. Understanding CPU caching and performance, Jul 2002. URL: https://archive.li/BCPXi.

[19] Wikipedia. Measuring network throughput — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Measuring%20network%20throughput&oldid=1185432147, 2023. [Online; accessed 11-December-2023].