CSE 221: System Measurement Project

Xinyuan Liang, Siran Ma, Wentao Huang
University of California, San Diego

March 17, 2023

1 Introduction

Operating System performance is a crucial aspect in today's computing environments. It determines the overall effectiveness and efficiency of a computer system in handling various tasks. The performance of an OS is influenced by various factors including the underlying hardware, system design, and workload. Understanding how hardware affects the performance of the OS is important in optimizing and tuning the system to meet specific needs and demands.

This report will focus on exploring the relationship between hardware and OS performance. We will analyze the impact of hardware components such as the cpu, ram, disk, and network on the performance of the OS and how they contribute to the overall system efficiency.

Our group members include Siran Ma, Xinyuan Liang and Wentao Huang. We conducted experiments on an x86-64 and Linux-based machine. See details in the Machine Description section. We compiled the benchmark programs written in C language with GCC-9.4.0 and disable the compiler optimization with the `-O0` option. And the estimated workload is 30 to 40 hours for each team member.

2 Machine Description

We get the basic information about the x86-64 and Linux-based machine by reading the hardware specifications directly or from the Linux commands.

2.1 Notebook Description

We launch our experiments on an ASUSTeK X510UQ machine.

2.2 Processor

We run the Linux command `cat /proc/cpuinfo` to get the cpu information and the command `lscpu|grep cache` to get cache information.

The processor we use is Intel(R) Core(TM) i5-7200U @ 2.50GHz x 4.

The cycle time of our processor is 800.000 MHz (with a maximum of 3100.0000 MHz and a minimum of 400.0000 MHz).

The size of the L1 cache is 64 KB, with 32 KB designated for the instruction cache and the remaining 32 KB for the data cache. Additionally, the L2 cache has a size of 256 KB, and the L3 cache has a size of 3 MB.

2.3 DRAM and Memory

DRAM and memory details can be obtained through the Linux command `sudo lshw -C memory`.

The DRAM model is HMA81GS6AFR8N-UH SK Hynix 8 GB 2400 SODIMM. The DRAM type is SODIMM (DDR4). The DRAM's clock frequency is 2400MHz, therefore each clock cycle will cost around 0.4ns. The DRAM has a capacity of 8 GB (64bits).

2.4 I/O Bus

I/O bus details can be obtained through the Linux command `lspci -v`.

The I/O bus link speed is 6.0 Gbps, using SATA Controller.

2.5 Disk

Disk details can be obtained from the hardware specifications.

The SSD model is TOSHIBA THNSNK128GVN8 (K8AS4102), with a capacity of 128 GB and a transfer rate of 600 MB/s. The IOPs and the latency information are not reported on the specifications.

2.6 Network

The network details can be obtained through the Linux commands `sudo ethtool name` and `lwconfig name`.

The Ethernet network bandwidth is 1000 Mb/s and the wireless network bandwidth is 400 Mb/s.

2.7 Operating System

The operating system is Ubuntu 16.04.4 LTS.

3 CPU, Scheduling, and OS Services

Our initial measurements concentrate on the performance of the CPU, scheduling, and Operating System services, and we present the results in terms of the cycles required to complete the operation. We will describe the methodology first, followed by the results and their interpretation.

3.1 Measurement Cycle Time

To obtain precise timing measurements, we take advantages of the constant TSC, which updates the counter at a consistent frequency. Using this approach, we can convert from cycles to wall clock times after measuring the cycle time. However, for greater accuracy, we use the number of cycles as the unit of experiment results.

3.1.1 Measurement Methods

We read the cycle counter twice within one second to calculate the number of elapsed cycles within that time frame. To read the cycle counter, we use two special instructions `RDTSC` and `RDTSCP`, and to ensure reliable results, we disable the multi-core processor feature by associating

the measurement task with a single core. Additionally, we set the CPU frequency to its maximum of 3.1GHz.

3.1.2 Measurement Result

We measure the average cycles within 1 second 100 times and get the following results,

- Estimated cycles in 1 second: 2.5G to 3.1G
- Measured average cycles in 1 second: 2.7G

3.1.3 Interpreting Measurement

Since we manually set the CPU frequency to 3.10GHz, which is the highest feasible value, and the CPU frequency specified by Intel is 2.5GHz, the results are consistent with our expectations.

3.2 Measurement Overhead

3.2.1 Measurement Methods

However, reading from the cycle counter using instructions also brings overhead. In order to accurately measure the performance of future experiments, we need to quantify this overhead. To accomplish this, we used the `RDTSC` benchmarking method described above to record the time from the timestamp register twice without executing any instructions between them. The difference between the two timestamp registers' values represents the measurement overhead.

To avoid interference from the CPU's out-of-order execution feature, we followed the instructions outlined in [1] by carefully crafting the order of executing instructions. We used the `CPUID`, `RDTSC`, and `RDTSCP` instructions. The `CPUID` instruction is a serialized instruction, while the `RDTSCP` instruction waits until all previous instructions have been executed [1]. By using these instructions, we are able to achieve a fine-grained measurement of timing overhead.

Besides, unlike in [1], we did not implement a kernel module to disable preemption and turn off interrupts. All measurements in this report are conducted in an environment where preemption and interrupts can occur at any time.

3.2.2 Measurement Result

To measure the overhead of timing, we have two nested loops within the measurement process. We designed an inner loop which only contains instructions `RDTSC` or `RDTSCP`, and executed it 1000 times. We ran the entire testbench 1000 times in an outer loop, with each testbench consisting of 1000 iterations of the inner loop. By obtaining 1000 testbench results, we are able to calculate the average time cost, as well as the standard deviation among the testbench results to determine the reliability of the measurements.

- Base hardware performance: 0 cycle
- Estimated software overhead: about 30 cycles[3]
- Measured performance average: 21.92 cycles
- Measured performance standard deviation among single testbench: 2.62 cycles
- Measured standard deviation across 1000 testbenches: 0.5 cycles
- Measured performance maximum deviation: 7629 cycles

Table 1: Procedure Call Overhead

argc	avg(cycles)	std1(cycles)	std2(cycles)	max dev(cycles)
0	32.89	5.74	1.63	22125
1	33.14	2.14	0.91	1662
2	33.37	2.29	0.58	6761
3	33.85	3.36	0.98	17926
4	33.85	3.56	1.31	31225
5	34.37	2.10	0.63	1210
6	34.67	3.67	1.65	11545
7	36.02	4.34	2.03	27811

3.2.3 Interpreting Measurement

Referring to Appendix C - General Purpose Instructions Latency and Throughput from [3], the throughput of RDTSC(P) instruction is around 30 clocks. Therefore, we assume the estimated measurement overhead is around 30 clocks, too. Since in most case, the latency of executing the instruction should be shorter than its throughput, it's acceptable that the measured overhead is a bit shorter than estimation.

3.3 Procedure Call Overhead

3.3.1 Measurement Methods

In the testbench, we create 8 minimum procedures with 0 to 7 arguments to test the overhead of procedure calls with different parameter sizes. We insert the `RDTSCP` instruction before and after the procedure call and iteratively conducted this test 1000 times.

3.3.2 Measurement Result

To measure the overhead of the procedure call, we still adopt two nested loops inside the measurement (1000 * 1000). The results of the 8 procedure calls whose number of arguments ranges from 0 to 7 are listed in the table 1 All the time cost in the table is represented in cycles. "std1" stands for performance standard deviation within a single testbench, "std2" stands for standard deviation across all testbenches, and "max dev" stands for performance maximum deviation. All the tables afterward will follow these terms.

- Base hardware performance: 0 cycle
- Estimated software overhead: Varies from the number of arguments passed into the procedure calls

3.3.3 Interpreting Measurement

Linux operating system uses stack and registers to store procedure call arguments, the caller procedure store the first 6 arguments to the %rdi, %rsi, %rdx, %rcx, %r8 and %r9 registers respectively, further arguments are stored on the stack. Thus, the more arguments the procedure has, the more `MOV` instructions are executed, making the procedure call overhead increases as the

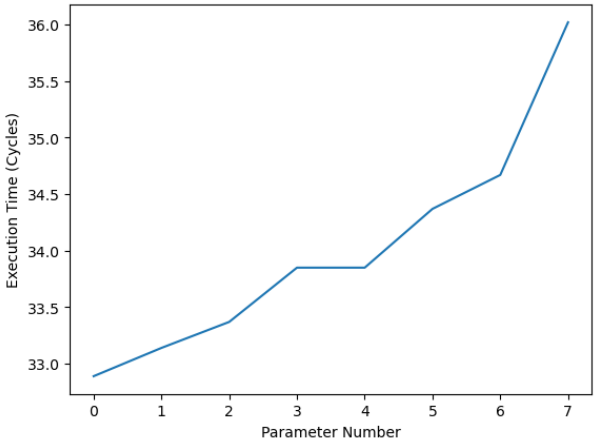


Figure 1: Procedure call overhead with different parameter numbers

number of arguments increases. According to [3], the latency of MOV instructions is 0.5 clocks. The result also shows the trend in this manner.

3.4 System Call Overhead

3.4.1 Measurement Methods

As mentioned in the project description, some operating systems will cache the results of some system calls (e.g. idempotent system calls like `getpid()`), so only the first call by a process will actually trap into the OS. The system call `times()`, however, return current process times, which are not cached by OS. In the testbench, we get the timestamps before and after the system call times. We iteratively conducted the testbench for 1000 times.

3.4.2 Measurement Result

To measure the overhead of the syscall call, we still adopt two nested loops inside the measurement (1000 * 1000).

- Base hardware performance: 0 cycle
- Estimated software overhead: Varies from different system calls. But we assume it would be a much larger cost compared with procedure calls
- Measured performance average: 739.59 cycles
- Measured performance standard deviation among single testbench: 61.25 cycles
- Measured standard deviation across 1000 testbenches: 22.88 cycles
- Measured performance maximum deviation: 5012 cycles

Table 2: Task Creation Overhead

type	avg	std1	std2	max dev
Process	329672.25	2233449.63	152089.56	66203661
Kernel thread	54394.91	278872.91	63085.16	27198348

3.4.3 Interpreting Measurement

During a procedure call, arguments are stored, registers are saved, and a `CALL` instruction is executed to jump to another address and continue execution. Based upon the normal procedure call, on the other hand, system calls also require trapping into the operating system, which incurs much more overhead.

3.5 Task Creation Time

3.5.1 Measurement Methods

To measure the time to create and run a process, we use the `fork()` system call to create a child of the current process, which returns immediately. In the testbench, we inserted `RDTSC` and `RDTSCP` instructions before and after the forking respectively to measure the time. As usual, we iteratively conduct the testbench 1000 times.

According to the project description, kernel threads run at user-level, but they are created and managed by the OS, e.g. `pthread_create()` on Unix creates a kernel-managed thread. We use `pthread_create()` to create and run a kernel thread, which returns immediately. By adding `RDTSC` and `RDTSCP` before and after the creation, we calculate the time used to create a kernel thread.

3.5.2 Measurement Result

To measure the overhead of the creation of processes and kernel threads, we adopt two nested loops inside the measurement (100 * 100) for each of them, considering that these operations are much more time-consuming than the previous ones. The results are shown in the table 2 for comparison.

- Base hardware performance: 0 cycle
- Estimated software overhead: 70,000 cycles for process creation and a much smaller time consumed by the thread creation

3.5.3 Interpreting Measurement

When comparing the overhead of creating processes versus creating threads, it is clear that thread creation is less costly. Threads are lightweight and share the same memory space and system resources as other threads within the same process. As a result, creating a new thread does not require the operating system to allocate new memory or resources for the thread. This makes kernel thread creation much more lightweight than process creation.

According to the results in [2], process creation for Linux-i686 takes 100 times longer than system calls, which translates to an estimated 70,000 cycles. However, the measured results were much shorter than the estimation, which may be due to recent performance improvements on

Table 3: Context switch Overhead

type	avg	std1	std2	max dev
Process	3100.88	168.47	180.63	19068
Kernel thread	2950.13	116.63	32.10	11648
R/W pipe	2353.10	2624.32	2370.52	2362404

Linux. It is also important to mention that our experiment was conducted under conditions where preemptions and interrupts may occur frequently, causing the absolute number of cycles required to finish the procedure call to be much higher than the results reported in [2].

3.6 Context Switch Time

3.6.1 Measurement Methods

Typically, blocking pipes are used to force context switches. When one process attempts to read from an empty pipe, it becomes blocked until data is available. To accomplish this, we created two pipes, named child pip and parent pip, to force the two processes to ping-pong. For example, the child process would initially attempt to read from the parent pip and become blocked since it is empty. The CPU would then switch to the parent process (ideally) to allow it to write into the parent pip. Next, the parent process would attempt to read from an empty child pip and go to sleep, and the process would switch back to the child process. To measure the time cost, we recorded the timestamp before the parent wrote to the parent pip, and after it read from the child pip. Each ping-pong event resulted in two context switches. However, it's important to note that the read and write operations on pipes themselves incur time consumption overhead, so we also measured the overhead of these operations.

Measuring thread context switch performance is similar to measuring process context switch performance, except that threads were used instead of processes.

3.6.2 Measurement Result

To measure the performance of the context switch, we adopt two nested loops inside the measurement (100 * 100) for each of them. The results are shown in the table 3 for comparison.

- Base hardware performance: 0 cycle
- Estimated software overhead: 1500 cycles for process context switch and a smaller time consumed by the thread context switch

3.6.3 Interpreting Measurement

The overhead of a context switch depends on the resources that it needs to save and restore. As described above, a thread context switch overhead is lower compared to a process context switch. Since a thread shares the same memory address space and other system resources with other threads within the same process. Therefore, fewer resources need to be saved and restored during a thread context switch. On the other hand, a process context switch needs to take care of memory space and system resources like file descriptors, page tables, etc.

We also refer to the result in [2] and estimated the process context switch takes about 1500 cycles (twice as executing system calls). However, it turns out that the context switch consumes a

shorter time. Besides, the overhead of reading and writing into pipes occupied a higher proportion in the context switch. It's also mentioned in [2] that the pipe overhead may vary between 30% and 300%. Nevertheless, the kernel thread context switch is indeed more lightweight than the process context switch.

4 Memory

Our second set of measurements focuses on memory performance, including memory access time, memory bandwidth, and page fault service time. We will present the performance in terms of throughput and cycles.

4.1 Memory access time

In this subsection, we will explore memory access performance under different access patterns. Modern processors have sophisticated memory hierarchies with several cache levels. By adjusting the test program's spatial and temporal locality and measuring memory access throughput, we can gain a rough understanding of how the hierarchy is structured.

4.1.1 Measurement Methods

We write a program that performs a tight sequence of reading operations, and the measured read throughput provides an indication of the memory system's performance for that specific sequence of memory access. A specific sequence of memory access entails a specific working set size, which refers to the array size that we are accessing, and a specific stride, which is the distance between two memory accesses. As mentioned earlier, the cache exploits both spatial and temporal locality to improve performance. By changing the working set size, we can alter the temporal locality; the larger the working set, the worse the temporal locality. Similarly, the larger the stride, the worse the spatial locality. Finally, by measuring the cycles required to complete the specific memory access pattern, we can calculate the throughput based on the CPU frequency and the working set size.

Given the hardware information of our machine (32KB L1 data cache, 256KB L2 cache, 3MB L3 cache), we decide to set the working set size (the array size) ranging from 1KB to 64MB and set the stride ranging from 16 bytes to 384 bytes.

4.1.2 Measurement Result

Our measurement results are summarized in Figure 2, Figure 3 and Figure 4, which display the performance for different combinations of array sizes and strides.

4.1.3 Interpreting Measurement

Figure 2 shows that as the stride increases, the overall throughput decreased due to worse spatial locality. And Figure 3 shows that the throughput decreases dramatically when the array size crosses the cache size boundary, e.g. the increasing trend slows down around 32 KB array size, the throughput decreases around 128 KB and 256 KB array size, and the throughput decreases tremendously around 2 MB and 4 MB array size, meaning that the memory access now needs to go down to the memory. To better demonstrate the results, we also attach the latency graphs with different array size ranges in Figure 4. As we can see from the results presented in the figures, the access latency increases noticeably when the array size reaches 32 KB, 128 KB, and around 4 MB, respectively.

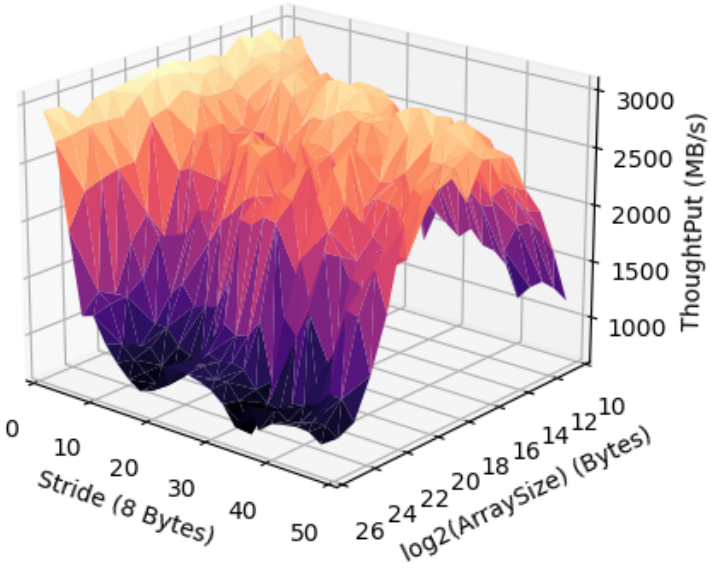


Figure 2: Memory Throughput Mountain

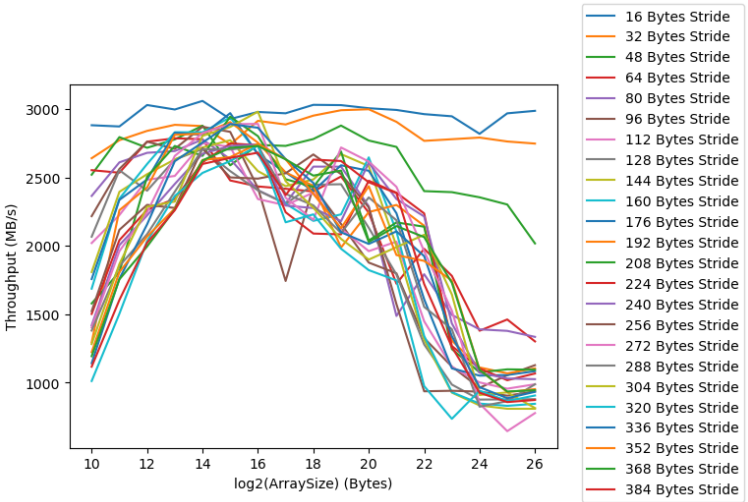
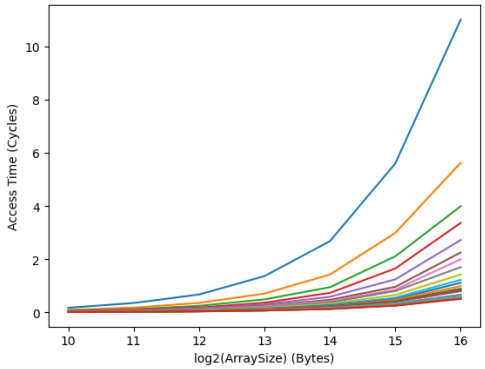
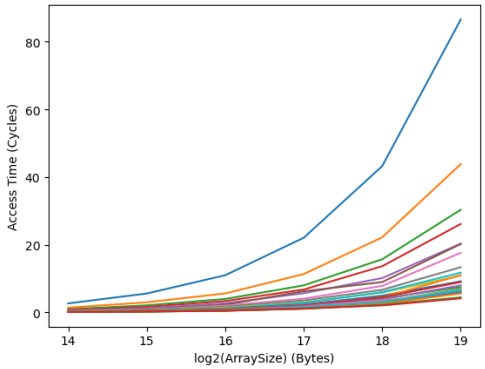


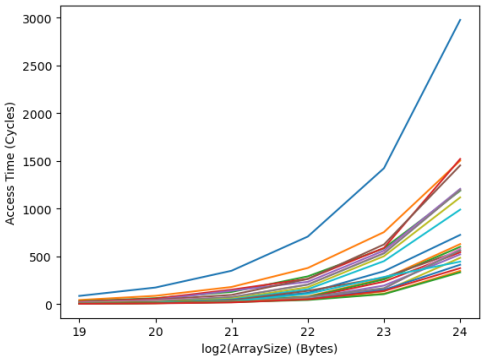
Figure 3: Throughput Graph



(a) L1 Cache



(b) L2 Cache



(c) L3 Cache

Figure 4: Latency Graphs

The graphs also show some unexpected behavior. For small strides, such as 16 bytes and 32 bytes, the reading speed remains constant as the array size increases. We assume that this is due to the cache line prefetching behavior and consistent good locality throughout the experiment. Additionally, for large strides, such as 384 bytes, the throughput increases as the array size increases when the array size is smaller than the L1 cache. This is counter-intuitive as the temporal locality is expected to decrease as the array size increases, but we speculate that this may be due to some caching or prefetching behavior.

4.2 Memory bandwidth

In this section, we will report on the bandwidth of reading and writing data from RAM. RAM bandwidth is defined as the rate at which a processor can read from or write data to the memory. Typically, RAM bandwidth is measured in units of GB/s, but we have chosen to use MB/s for consistency.

4.2.1 Measurement Methods

To avoid interleaving instructions and delayed read requests to memory, we adopted the method of the hand-unrolled loop as described in [5]. To ensure that we were testing memory bandwidth rather than cache bandwidth, we made sure that the destination data stored in memory would not be cached after each read or write. We access the memory at a stride of 64 bytes to avoid cache line prefetch, which is the cache line size of the CPU. Additionally, to account for our L3 cache's size of 3 MB, we define two arrays of 1.5 MB size and use the `memcpy()` system call to copy one array to another before the benchmarks. This ensures that both arrays would be in the L3 cache and there would be no extra space for the later caching of reads and writes. By doing so, we ensure that each read and write could be seen as one memory access.

For testing the memory bandwidth, we allocate a 4 MB array as it is larger than the L3 cache size, which is around 3 MB. When testing the write bandwidth, we assigned values to the array's elements. When testing the read bandwidth, we read values from the array's elements and added them together to avoid C compiler optimization from affecting the results. We loop the above process 100 * 100 times to compute the average throughput and deviation of memory access time for each testbench.

4.2.2 Measurement Result

Read Memory

- Base hardware performance: 38,400 MB/s
- Estimated software overhead: about 28,800 MB/s
- Predicted time: about 9600 MB/s
- Measured performance average: 4757.95 MB/s
- Measured performance standard deviation among single testbench: 56.21 MB/s
- Measured standard deviation across 1000 testbenches: 40.78 MB/s
- Measured performance maximum deviation: 1438 MB/s

Write Memory

- Base hardware performance: 38,400 MB/s
- Estimated software overhead: about 28,800 MB/s
- Predicted time: about 9600 MB/s
- Measured performance average: 1300.60 MB/s
- Measured performance standard deviation among single testbench: 140.67 MB/s
- Measured standard deviation across 1000 testbenches: 77.98 MB/s
- Measured performance maximum deviation: 1101 MB/s

4.2.3 Interpreting Measurement

According to Wikipedia, the theoretical maximum memory bandwidth is calculated by this formula: bandwidth = base DRAM clock frequency * the number of data transfers per clock * bus width * the number of interfaces. According to the specification, our memory theoretical maximum bandwidth would be 38400 MB/s. However, in [5], the author states that measured RAM bandwidth is about 25% of theoretical bandwidth (according to the result of IBM power2). Therefore, we don't expect our experiment result to be close to the theoretical result. The measured RAM bandwidth is much smaller than the theoretical one may be caused by lots of factors, for example, there are lots of applications sharing with the memory at the same time, which would greatly affect the bandwidth. Another reason is that we access the memory in a non-continuous manner, which also decreases the read/write speeds compared with the ideal continuous manner.

4.3 Page fault service time

This section reports the time for faulting an entire page from disk. There are two kinds of page faults, one is a major page fault handled by disk I/O operation (e.g. memory mapped file or swapping pages into memory); the other one is handled without disk I/O operation (e.g. `malloc`)[4].

4.3.1 Measurement Methods

The reason we use the `mmap` operation as our measurement is that it allows for direct buffer sharing between the OS kernel and the user-level process, reducing the non-negligible performance overhead caused by copying data between these two address spaces.

We measure the average cycles it takes to fault a single page. We create non-zero files whose sizes range from 100 to 3000 pages and map them separately into each measurement, followed by reading through the file and adding up the values of bytes. For measuring the time cost, we still adopt two nested loops inside each measurement (100 * 100). Except for the measurement code, we also utilized the Linux command `time` to verify the expected major/minor page fault number. Notice that the Linux kernel would support the file buffer read-ahead mechanism to optimize reading costs, which will dramatically reduce the average page fault cost. Therefore, we set up the `POSIX_MADV_RANDOM` flag for `mmap` to force random page references and disable the prefetching behavior.

4.3.2 Measurement Result

- Base hardware performance: 23600 cycles
- Estimated software overhead: 3135 cycles
- Predicted time: 26735 cycles
- Measured performance average: 29062.11 cycles
- Measured performance standard deviation among single testbench: 73.41 cycles
- Measured standard deviation across 1000 testbenches: 14.85 cycles
- Measured performance maximum deviation: 8972 cycles

4.3.3 Interpreting Measurement

For page fault service, the main bottleneck is the read/write speed of the SSD disk. Therefore, to estimate the base hardware performance, we referred to the specification of the SSD transfer speed, which is 600 MB/s. The estimated software overhead includes the system call, procedure call, memory access (we still need to read through the mapped file from memory to force page faults), and measurement overhead. Moreover, the instruction loading overhead is negligible with a warm-up. Thus, the predicted time cost is the sum of the base hardware performance and the estimated software overhead. Although the average measured page fault service cost is higher than the predicted value, the difference is only 1.2 ns per page, which is still acceptable. Furthermore, the disk used in this experiment has been in use for over 5 years, and it is likely to be slower than its original specification.

Compared with the memory access time, which is around 2350 cycles per page, we can see that page fault latency is about 12 times larger.

5 Network

Our third set of measurements focusing on network performance, including application-level round trip time, peak bandwidth writing from client to server, and connection overhead on client. The performance will be presented in terms of the throughput and cycles.

5.1 Round Trip Time

Round trip time (RTT) refers to the amount of time it takes for a packet of data to travel from one point in a network to another and then return to the original point. RTT is an important metric in network performance because it can affect the responsiveness of applications and the overall user experience.

5.1.1 Measurement Methods

RTT is calculated by sending a small packet of data from one device to another and measuring the time it takes for the packet to reach its destination and for an acknowledgement to be sent back to the original device. This measurement includes both the time it takes for the packet to travel across the network and any processing time required by the devices involved.

For the local loopback measurement, we run a server program listening on port 8080 and use a client program to connect the server. The client sends 56 bytes and the server response

64 bytes to the client, the same as the `ping` command's behavior. By measuring cycles used for `send()` and `recv()` on the client side, we can get the local loopback round trip time given the CPU frequency.

Similarly, for the remote connection measurement, we run a server program and a client program, and make them communicate with each other by put them under the same subnet, usually through a personal hotspot. And measuring cycles used for `send()` and `recv()` on the client side, we can get the remote connection round trip time given the CPU frequency.

5.1.2 Measurement Result

Local Loopback

- Base hardware performance: uses virtual interface, no available hardware performance
- Estimated software overhead: two memory copy operations b/w application and kernel memory space, costs 0.1145 ms
- Predicted time: 0.1145 ms
- Measured performance average: 0.1967 ms
- Measured performance standard deviation among single testbench: 0.03028 ms
- Measured standard deviation across 1000 testbenches: 0.04016 ms
- Measured performance maximum deviation: 13.4674 ms

Remote Loopback

- Base hardware performance: client machine WiFi transmit rate : 400 Mbps, remote server WiFi transmit rate: 400 Mbps
- Estimated software overhead: four memory copy operations b/w application and kernel memory space costs 0.2314 ms
- Predicted time: 0.2314 ms + network propagation time about 4 ms
- Measured performance average: 7.06945 ms
- Measured performance standard deviation among single testbench: 3.20329 ms
- Measured standard deviation across 1000 testbenches: 2.13026 ms
- Measured performance maximum deviation: 194.01796 ms

5.1.3 Interpreting Measurement

For the local loop back round trip time measurement, data is being sent from the network stack to the application layer on the same device without leaving the physical network interface. This process is often handled by a special loopback interface, i.e. a virtual network interface, which is used to emulate network traffic on the same device. Since the data does not leave the device, there is no physical transmission involved and the round trip time is typically much faster than for traffic that needs to be transmitted over a physical network. However, there is still some processing overhead involved in handling loopback traffic, including the memory copy operation b/w the application and kernel memory space and the processing time required by the

network stack. Thus, the estimated software overhead is copying 56 bytes and then copying 64 bytes, which is 0.1145 ms. Given the situation where the network stack also need to process the packet as it would for a packet received from a physical network interface, we treat 0.1967 ms a reasonable result.

On the other hand, the local loopback round trip time using `ping` command is 0.037 ms, which is much smaller than the measured application level RTT. And considering the `ping` is implemented at the kernel level, thus no memory copy overhead, we also treat it is a reason result.

For the remote connection case, it is hard to estimate the RTT due to the difficulty to measure the network propagation time. Similar with the local loopback, the client process sends a packet to the server process on another machine, but now the packet travels across the network to the server machine. Then the server process receives the packet and sends a packet back to the client application, the new packet also needs to travel back across the network to the client machine.

To verify correctness, we measure the remote communication RTT using `ping` command and get an average RTT of 7.66 ms, which is similar to the experiment result. Thus, we treat the result reasonable.

5.2 Peak bandwidth

The definition of network bandwidth is the maximum number of bits it can send/receive per second. It measures the capacity available for use in data transmission and presents the performance of the network. Therefore, the network performance has a non-trivial effect on the peak bandwidth.

5.2.1 Measurement Methods

Compared with the measurement of Round Trip Time, the measurement of peak bandwidth only involves sending all the expected length of data to the internet, in other words, it does not need to wait for receiving a response from the other end. Our method records the time it takes to send a certain size of data on a TCP connection. And we calculate the bandwidth based on the formula:

$$\text{bandwidth} = \text{N byte}/(\text{cycles/Mhz}) * 8(\text{Mbps})$$

We express bandwidth in megabits per second, but we calculate the average and variance of the cycles required to send messages. Each measurement includes two nested loops (100 * 100 rounds). The experiments were conducted over the WiFi network.

5.2.2 Measurement Result

Local We measure the cycles it costs to send 2048 bytes to loopback.

- Base hardware performance: uses virtual interface, no available hardware performance
- Estimated software overhead: memory copy speed 0.9 MB/s
- Predicted result: 8108 Mbps
- Measured performance average: 3628.48 Mbps (12252.31 cycles)
- Measured performance standard deviation among single testbench: 252353.03 cycles
- Measured standard deviation across 1000 test benches: 29684.06 cycles
- Measured performance maximum deviation: 2257227950 cycles

Remote We measure the cycles it costs to send 2048 bytes to remote.

- Base hardware performance: client machine WiFi transmit rate : 400 Mbps, remote server WiFi transmit rate: 400 Mbps
- Estimated software overhead: memory copy speed 0.9 MB/s
- Predicted result: 320 Mbps
- Measured performance average: 100.8 Mbps (441186.76 cycles)
- Measured performance standard deviation among single testbench: 148316.54 cycles
- Measured standard deviation across 1000 testbenches: 26937.80 cycles
- Measured performance maximum deviation: 273145222 cycle

5.2.3 Interpreting Measurement

For local bandwidth, we assume that its base hardware performance should be the DRAM bandwidth. The reason is that the loopback packet will not be sent over the internet and just be dispatched to the virtual interface, which means that the entire process is only relative to the memory copy operations (from application memory to kernel network queue and verse back). To estimate the software overhead, we count for the memory read and write overhead. According to the previous experiment, a single copy operation takes 0.9 ms for every 1 MB. Notice that, though the `send` system call needs to be trapped in the kernel, the overhead of it is negligible compared with memory operations. To calculate the predicted result, we also use the estimated overhead (1 MB per 0.9 ms). The reason why we don't use the base hardware performance to predict is that it's too theoretical and hard to achieve.

For remote bandwidth, we use the WiFi interface bandwidth as base hardware performance. For software overhead, besides memory operations, we also count for the connection set-up overhead. To calculate the predicted result, we first calculate the ideal cost to send 1 MB, (which is 1 / 400 * 8), then plus it with the overhead, and get a total of 24.9 ms per MB.

Although the measured bandwidth is much smaller than the predicted one, we still treat it reasonable for the following reasons. The actual network performance is less likely to achieve the ideal NIC bandwidth (the common bandwidth of our WiFi is around 80 Mbps according to the network monitor). Besides, our evaluation is based on the TCP protocol, which is a reliable transfer protocol that involved lots of costly operations, e.g. checksum, acknowledgment, re-transmittance, etc.

Also notice that we still use cycles to calculate bandwidth. The variance and maximum deviation show that the performance of the network is varied and unstable.

The final thing we want to point out is that we choose to send messages in different sizes to see if the message size has an effect on the bandwidth. However, we could not find a stable pattern of it. In the loopback bandwidth measurement, we find that the bandwidth is more likely to be higher as the message becomes larger. In the remote bandwidth measurement, however, we find that the bandwidth of sending shorter messages is a bit higher, which we assume is due to the dominant overhead of sending more TCP packets over the internet.

5.3 Connection overhead on client

Setup time refers to the amount of time it takes for a client to connect to a server. Tear-down time refers to the amount of time it takes for closing the connection between the client and the server.

5.3.1 Measurement Methods

To test connection and close overhead on client, we adapt similar methodology as measuring the round trip time(RTT). This time, instead of recording the start time of `send()` and the end time of `receive()`, we record the start time and end time of `connect()` and `close()`.

5.3.2 Measurement Result

Local Connection Setup Time

- Base hardware performance: uses virtual interface, no available hardware performance
- Estimated software overhead: three memory copy operations b/w application and kernel costs 0.1718 ms
- Predicted time: 0.3 ms to 0.4 ms
- Measured performance average: 7.1487 ms
- Measured performance standard deviation among single testbench: 5.6230 ms
- Measured standard deviation across 1000 testbenches: 1.0750 ms
- Measured performance maximum deviation: 10.37564 ms

Remote Connection Setup Time

- Base hardware performance: client machine WiFi transmit rate : 400 Mbps, remote server WiFi transmit rate: 400 Mbps
- Estimated software overhead: six memory copy operations b/w application and kernel costs 0.3436 ms
- Predicted time: 1.0 ms to 9.0 ms
- Measured performance average: 8.6419 ms
- Measured performance standard deviation among single testbench: 9.5510 ms
- Measured standard deviation across 1000 testbenches: 3.19002 ms
- Measured performance maximum deviation: 126.0689 ms

Local Connection Close Time

- Base hardware performance: uses virtual interface, no available hardware performance
- Estimated software overhead: negligible system call overhead
- Predicted time: 0.01 ms to 0.09 ms
- Measured performance average: 0.0443 ms
- Measured performance standard deviation among single testbench: 0.0070 ms
- Measured standard deviation across 1000 testbenches: 0.0017 ms
- Measured performance maximum deviation: 0.0413 ms

Remote Connection Close Time

- Base hardware performance: client machine WiFi transmit rate : 400 Mbps, remote server WiFi transmit rate: 400 Mbps
- Estimated software overhead: negligible system call overhead
- Predicted time: 0.01 ms to 0.09 ms
- Measured performance average: 0.0539 ms
- Measured performance standard deviation among single testbench: 0.0074 ms
- Measured standard deviation across 1000 testbenches: 0.0023 ms
- Measured performance maximum deviation: 0.0875 ms

5.3.3 Interpreting Measurement

To establish a TCP connection, the network stack sends and receives segments to and from the application layer. In the case of a local connection, the segment is routed back to the loopback interface and delivered to the server process on the same machine, without any network traffic. However, in a remote connection, packets are sent over the network. In both cases, the setup time should be proportional to the round trip time (RTT), and specifically 1.5 times larger due to the 3-way handshake mechanism. However, our results did not show this pattern, and we suspect that the setup time may be influenced by system load and network conditions.

To gracefully terminate a connection between a client and a server, a TCP four-way handshake is used. However, it is important to note that the actual closing process is handled by the underlying operating system, and the `close()` function simply informs the operating system of the intention to close the connection and returns immediately. Therefore, the time for closing the connection should be significantly smaller than the setup time. The result is reasonable.

6 File System

Our final part of measurement focuses on file system performance, including profiling the file cache size, local and remote file read time, and contention. The performance will be presented in terms of the cycles and read speed.

6.1 File Cache Size

The file buffer cache works by temporarily storing recently accessed disk data in a portion of memory. The data is available to subsequent read requests, allowing them to be satisfied more quickly, without the need to retrieve the data from the slower disk storage. The size of the cache is typically configurable, and the operating system tries to balance the size of the cache with the available memory and other system resources. So in this part, we are going to measure the size of the file cache.

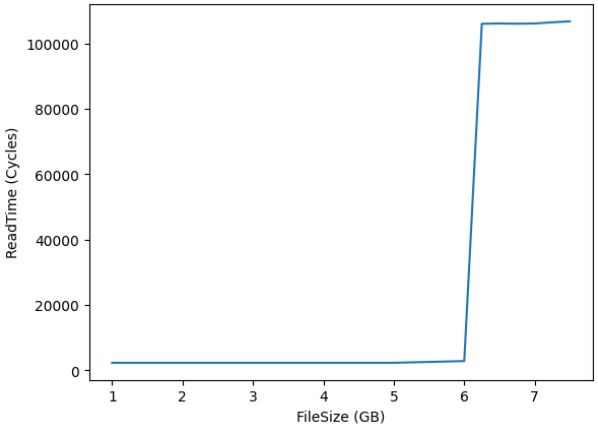


Figure 5: File Cache Measurement

6.1.1 Measurement Methods

As mentioned above, the file buffer cache is dynamic based on the application behavior, e.g. if an application is accessing large files, the file buffer cache size increases to improve performance. However, it is impossible that the cache size increase as large as the memory size, since the operating system needs to run its services. So our measurement method is to measure access time per block to files of different sizes, and make sure most of the access goes to memory, i.e. file cache, by accessing the file from the end to beginning before the measurement. The access time should be increase dramatically after the file size reaches some point, e.g. 6 GB on a machine with 8 GB ram, suggesting that now the file buffer cache size has reached the limit and the accessing needs to go to disk.

6.1.2 Measurement Result

- Base hardware performance: memory reading bandwidth, 4757.95 MB/s
- Estimated software overhead: not available
- Predicted cache size limit: around 6 GB

6.1.3 Interpreting Measurement

As we can see from Figure 5, the access time stays still before the file size reaches 6 GB. Since if the file size is small enough to fit entirely within the file buffer cache, the access time will be relatively fast, as the data can be accessed quickly from the cache. However, as the file size increases and the data no longer fits entirely within the cache, the operating system must perform more disk I/O operations to retrieve or store the data, which can significantly increase the access time. The experiment result exactly shows the pattern as expected.

6.2 Local File Read Time

This section measures the average local read time for files in different sizes. Since the access pattern (sequential or random) has a big effect on the file system read speed, we also take the access pattern into consideration. Also, we need to avoid file caching effects.

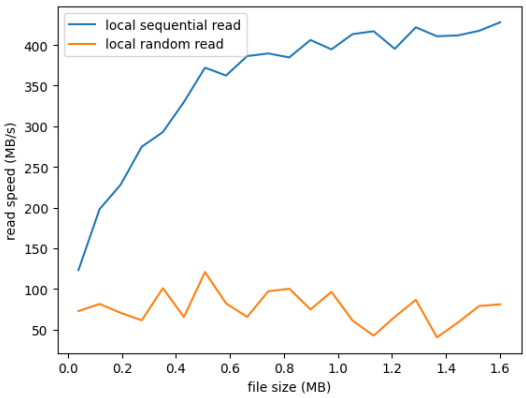


Figure 6: read file local (speed)

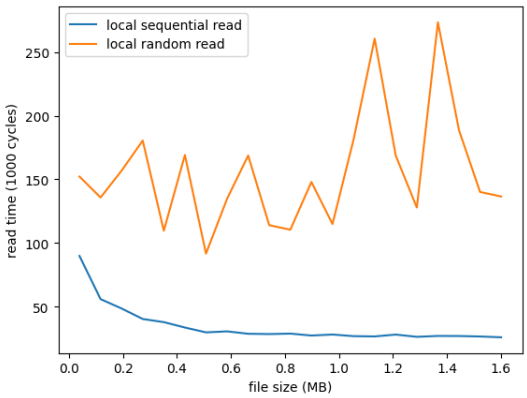


Figure 7: read file local (cycles)

6.2.1 Measurement Methods

We first split the file into blocks (4096 bytes per block) and gain an array of block indexes after assigning each of them a block index, e.g {1, 2, 3, ..}. For sequential reading, we just read the file in the same order as the original block indexes array. For random reading, we first shuffle the block indexes array, e.g. {3, 1, 5, ..}, and read the file randomly according to the shuffled block indexes.

To avoid reading cached data, we utilize the `posix_fadvise` system call, that is, after reading the block, we drop it from the cache explicitly by `POSIX_FADV_DONTNEED` flag. Before doing the experiment, we also use the Linux command `/proc/sys/vm/drop_caches` to flush the entire cache.

We still adopt two nested loops inside each measurement (100 * 100 rounds). We choose the file size ranging from 10 blocks (40K) to 410 blocks (approximately 1.6M).

6.2.2 Measurement Result

The result is shown in Figure 6 and Figure 7. We produce two different figures, one uses cycles as the y-axis, and another uses speed (MB/s) as the y-axis. The latter illustrates the file read performance more intuitively.

Sequential Read

- Base hardware performance: 600 MB/s (18432 cycles per block)
- Estimated software overhead: 10240 cycles per block
- Predicted result: 28672 cycles
- Measured performance average: 34307.71 cycles (mean of the later half: 27053 cycles)

Random Read

- Base hardware performance: 100 MB/s (110592 cycles per block)
- Estimated software overhead: 10242 cycles per block

- Predicted result: 120834 cycles
- Measured performance average: 155392 cycles

6.2.3 Interpreting Measurement

To evaluate the base hardware performance, we converted the transfer rate into the read time for a block. Regarding software overhead, the majority of it should be related to system call overhead and memory copying from the kernel file buffer cache to the user memory space. Based on previous experiments, we found that it takes 0.9ms to read and write into memory per MB, which implies that copying a block takes around 9500 cycles. Furthermore, a system call takes about 740 cycles.

For the sequential read, the average measured read time is 34307 cycles. But since the latter half of the result is more stable, we also calculate their average measured read time, which is 27053, which is pretty close to the predicted result. We are surprised to see that the measured result outperforms the predicted result for the first time. But it indeed may happen because Linux uses zero-copy to optimize the overhead in copying data from one memory area to another, which is unnecessary.

For the random read, the average measured read time is 155392, which is a bit higher than the predicted one. It's reasonable as the speed of the random read is much more varied than that of the sequential read.

Also, notice that the sequential read speed grows up at the beginning and keeps at a relatively stable speed. The growing trend is due to the fact that the SSD chunk is actually larger than the block size and it tries to learn from the reading pattern (which is 4096 bytes every time) as well as prefetches more data. When the file size grows to 512 KB, however, it reaches the maximum chunk size and the reading speed becomes relatively stable.

6.3 Remote File Read Time

This section measures the read time for files on remote NFS. We create an NFS (server side) on MacOS and connect it from the Ubuntu (client side) with SMB protocol. The remaining part is exactly the same as the previous section.

6.3.1 Measurement Methods

The measurement method is the same as the previous one, except that we shrink the loop size from 100 * 100 to 10 * 10 for the sake of saving time. We still conduct this experiment in the same WiFi network as previously.

6.3.2 Measurement Result

The result is shown in Figure 8 and Figure 9. We still produce two different figures for better illustration.

Sequential Read

- Base hardware performance: client machine WiFi transmit rate : 400 Mbps, remote server WiFi transmit rate: 400 Mbps
- Estimated software overhead: 10240 cycles per block
- Predicted result: 895249 cycles

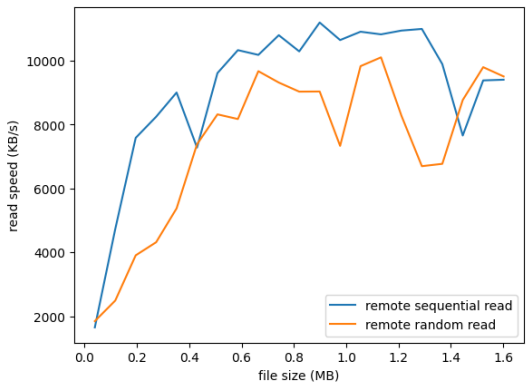


Figure 8: read file remote (speed)

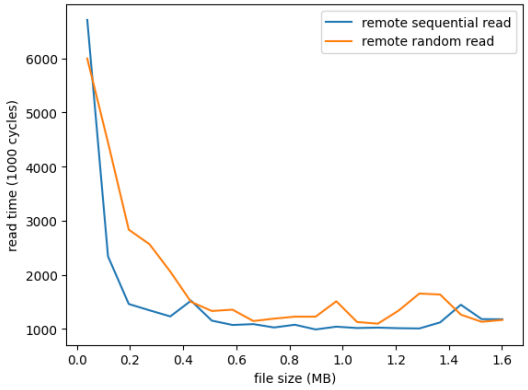


Figure 9: read file remote (cycles)

- Measured performance average: 1476495 cycles (mean of the file read time for files with more than 110 blocks: 1093754 cycles)

Random Read

- Base hardware performance: client machine WiFi transmit rate : 400 Mbps, remote server WiFi transmit rate: 400 Mbps
- Estimated software overhead: 10242 cycles
- Predicted result: 895249 cycles
- Measured performance average: 1845940 cycles (mean of the file read time for files with more than 110 blocks: 1291307 cycles)

6.3.3 Interpreting Measurement

The bottleneck of random file reading lies in the network transfer speed. Thus we choose the WiFi interface bandwidth as base hardware performance, which is 400 Mbps for both the client and server machine. The estimated software overhead is still memory copy and system call. The performance of the server (MacBook Pro 2021) is much better than the experiment machine (Memory bandwidth: 200GB/s, SSD transfer rate: 3.2GB/s). We can just ignore the software overhead on the server side. The predicted result is calculated based on the previous peak network bandwidth, which is 100 Mbps. We can now know that transferring a block takes about 885000 cycles. Thus we estimate the predicted result is 895249 cycles.

As before, the average of the file reading time is calculated with two methods: the first one is basically an average over different files, and the second one only considers the average of the relatively reliable part of the results – when the file size exceeds 110 blocks – since its trend becomes more steady.

The measured read time is larger than the predicted one because the predicted result actually overlooks some unknown factors. First, the NFS is transferring over SMB protocol, whose speed should differ from TCP protocol. And we also ignore the disk and memory transfer costs on the server side (although they should be small). What's more important, the network is unstable and our result variance is pretty large.

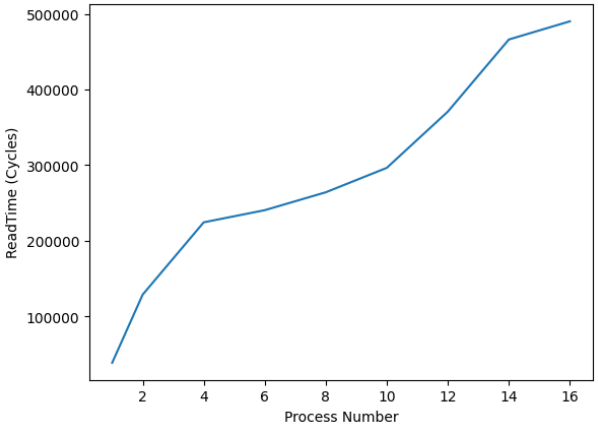


Figure 10: File Cache Measurement

But Figure 8 and Figure 9 indeed show some good attributes which match our assumption: the random read speed is still slower than the sequential read speed, and the network bottleneck also narrows the gap between sequential and random read speed.

6.4 Contention

This section measures the average time to read one file system block of data when multiple processes simultaneously performing the same operation on different files on the same disk. We need to avoid file caching effects to measure real disk I/O time.

6.4.1 Measurement Methods

We split the files into blocks (4096 bytes per block) and we read files in 1 MB size for each child process. We measure the start time and end time of reading one block as "block reading time" to eliminate loops overhead and average the "average block reading time" over n processes. To test the average time to read one file system block of data as a function of the number of processes, we created 1 to 16 processes to read simultaneously. For every different number of processes, we test 10 * 20 times.

To avoid reading cached data, we utilize the `posix_fadvise` system call: after reading the block, we drop it from the cache explicitly by `POSIX_FADV_DONTNEED` flag. Before doing the experiment, we also use the Linux command `/proc/sys/vm/drop_caches` to flush the entire cache.

6.4.2 Measurement Result

- Base hardware performance: 600 MB/s (18432 cycles per block)
- Estimated software overhead: 10240 cycles per block
- Predicted result: 28672 cycles (for #process = 1)
- Measured performance: shown as Figure 10

6.4.3 Interpreting Measurement

We still use the base hardware performance (which is the SSD transfer rate) as well as the software overhead used in the previous section (which is combined with system call and memory copy). It's hard to predict the exact performance in this experiment, but we predicted that as the number of processes reading files increases, the average time of reading one file system block would increase as well (which is indeed shown in Figure 10). This is because when multiple processes are accessing different files on disk, the disk head has to seek across larger areas to get files in different locations. The overhead of seeking would increase. Besides, the disk bandwidth is limited and shared among all the processes. If the number of processes increases, the processes need to compete with the disk bandwidth.

To mitigate this problem, most modern computers adapt memory prefetching techniques with SSDs to improve performance. For example, the Linux kernel includes a feature called `readahead` that prefetches data from storage into memory based on the access patterns of the system. Note that in our experiment we flush the caches between each file system block's read, so our results don't reflect the prefetching mechanism.

Summary

Operation	Hardware Base	Software Overhead	Predicted Result	Measured Result
Measurement	0 ns	11.062 ns	11 ns	8.083 ns
Procedure Call(0)	0 ns	11.062 ns	11.062 ns	12.128 ns
Procedure Call(1)	0 ns	11.246 ns	11.246 ns	12.220 ns
Procedure Call(2)	0 ns	11.431 ns	11.431 ns	12.305 ns
Procedure Call(3)	0 ns	11.615 ns	11.615 ns	12.482 ns
Procedure Call(4)	0 ns	11.799 ns	11.799 ns	12.482 ns
Procedure Call(5)	0 ns	11.984 ns	11.984 ns	12.673 ns
Procedure Call(6)	0 ns	12.168 ns	12.168 ns	12.784 ns
Procedure Call(7)	0 ns	12.353 ns	12.353 ns	13.282 ns
System Call	0 ns	$\gg$ Procedure Call	$\gg$ Procedure Call	272.71 ns
Process Creation	0 ns	25.811 µs	25.811 µs	121.56 µs
Thread Creation	0 ns	$\ll$ Process Creation	$\ll$ Process Creation	20.057 µs
Process Switch	0 ns	0.553 µs	0.553 µs	1.143 µs
Thread Switch	0 ns	$\ll$ Process Switch	$\ll$ Process Switch	1.088 µs
L1 Cache	Figure 2 3 4	Figure 2 3 4	Figure 2 3 4	Figure 2 3 4
L2 Cache	Figure 2 3 4	Figure 2 3 4	Figure 2 3 4	Figure 2 3 4
L3 Cache	Figure 2 3 4	Figure 2 3 4	Figure 2 3 4	Figure 2 3 4
ReadMem Bandwidth	38,400 MB/s	28,800 MB/s	9600 MB/s	4757.95 MB/s
WriteMem Bandwidth	38,400 MB/s	28,800 MB/s	9600 MB/s	1300.60 MB/s
Page Fault	8.702 µs	1.156 µs	9.858 µs	9.610 µs
Local RTT	not available	0.1145 ms	0.1145 ms	0.1967ms
Remote RTT	400 Mbps	0.2314 ms	4.2314 ms	7.06945 ms
Local Bandwidth	38,400 MB/s	0.9 ms/MB	8108 Mbps	3628.48 Mbps
Remote Bandwidth	400 Mbps	0.9 ms/MB	320 Mbps	100.8 Mbps
Local Setup Time	not available	0.1718 ms	0.295 ms	7.1487 ms
Remote Setup Time	400 Mbps	0.3436 ms	10.604 ms	8.6419 ms
Local Close Time	not available	272.71 ns	0.01 ms to 0.09 ms	0.0443 ms
Remote Close Time	not available	272.71 ns	0.001 ms to 0.090 ms	0.0539 ms
File Cache Size	4757.95 MB/s	not available	around 6 GB	6.5 GB
Local Sequential Read	600 MB/s	3.776 µs/block	10.572 µs	12.650 µs
Local Random Read	100 MB/s	3.776 µs/block	44.555 µs	57.298 µs
Remote Sequential Read	400 Mbps	3.776 µs/block	330.107 µs	544.430 µs
Remote Random Read	400 Mbps	3.776 µs/block	330.107 µs	680.656 µs
Multiple Process Read(1-16)	600 MB/s	3.776 µs/block	330.107 µs	Figure 10

Table 4: result summary for all the experiments

References

We have consulted the following papers to estimate overhead and design our measurements.

[1] Gabriele Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. Intel Corporation, 2010.

[2] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In USENIX annual technical conference, pages 279–294. San Diego, CA, USA, 1996.

[3] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual. 2012

[4] Slide from CS423 https://courses.engr.illinois.edu/cs423/fa2011/lectures/cs423_mp3_final.pdf

[5] Larry McVoy and Carl Staelin, lmbench: Portable Tools for Performance Analysis, Proc. of USENIX Annual Technical Conference, January 1996.