CSE 221 Project - Fall 2025

Alex Asch
aasch@ucsd.edu, A69033251

Aaron Wu
aaw005@ucsd.edu, A16369174

Sneha De
snde@ucsd.edu, A17282520

1 Introduction

The objective of our project is to create a set of benchmarks to characterize and measure various aspects of our target operating system, including CPU operations, memory operations, networking, and file operations.

Our targeted operating system is Ubuntu 24.04.03, a Linux distribution known for its reliability and ease-of-use. We chose Ubuntu as our OS to benchmark because of our familiarity with well-established UNIX tooling. While our individual setups vary, for the most accurate results, our report is based on a bare-metal, dual-booted installation of Ubuntu Desktop to minimize unknown overheads of virtualization or emulation. More detailed specifications of our chosen machine are listed in section 2.

We split the implementation of the experiments as follows:

Benchmark	**Contributors**
CPU - Measurement overhead	Sneha
CPU - Procedure call overhead	Aaron
CPU - System call overhead	Alex
CPU - Task creation time	Aaron
CPU - Context switch time	Alex, Sneha
Memory - Access time	Sneha, Alex
Memory - Bandwidth	Aaron, Alex
Memory - Page fault service time	Aaron
Network - Round trip time	Alex
Network - Peak bandwidth	Alex
Network - Connection overhead	Alex
File system - Size of file cache	Aaron, Sneha
File system - File read time	Sneha
File system - Remote file read time	Sneha
File system - Contention	Aaron

Our benchmark suite is written in C with `gcc` version 13.3.0 as our compiler, using the following compiler flags: `CFLAGS = -Wall -Wextra -pedantic -O0`. We use the `-O0` flag to disable all compiler optimizations to stay accurate to our original code.

We estimate the total number of hours spent on this project to be 75 hours summed up.

Our source code can be found on the provided GitHub repository.

2 Machine Description

The machine we used to run the experiments had these specifications:

1. Processor: 12th Gen Intel(R) Core(TM) i7-1250U

   From the output of `lscpu`:

   (a) Number of cores: 10

   (b) L1 (data) Cache Size: 352 KiB (10 instances) $\rightarrow$ $\approx$ 35 KiB per core

   (c) L1 (instruction) Cache Size: 576 KiB (10 instances) $\rightarrow \approx$ 58 KiB per core

   (d) L2 Cache Size: 6.5 MiB (4 instances) $\rightarrow \approx$ 1.625 MiB shared per group of four cores

   (e) L3 Cache Size: 12 MiB (1 instance)

   We run our benchmarks on the CPU with id 0 by using `taskset -c 0`, with an estimated CPI of 1.2769 and constant TSC clock speed of 1881.600 MHz, which differs from the max boost clock speed which is 4700MHz. Since we utilize the TSC register for all measurements, we use 1881.600MHz, which means a cycle time of 0.5315 nanoseconds.

$$\text{Cycle time} = \frac{1}{\text{Freq.}} = \frac{1}{1881.600 \times 10^6 \text{ Hz}} = 0.5315 \text{ ns}$$

2. Memory: From the output of `lshw`

   (a) DRAM Type: DDR4

   (b) Clock: 4267 MHz

   (c) Capacity: 16 GiB

   (d) Width: 64 bits (8 bytes)

   (e) Channels (banks): 2 x 8GiB

   (f) Cache line size: 64 bytes

   Obtained via:

```
getconf LEVEL1_DCACHE_LINESIZE
```

3. Memory bus bandwidth: A theoretical memory bandwidth can be calculated with [6]:

$$\begin{aligned}\text{Bandwidth} &= \text{Clock Speed} \times \text{Width} \times \text{Channels} \\ &= 4267 \text{ MHz} \times 8 \text{ bytes} \times 2 \\ &= \frac{4267 \times 10^6}{s} \times 8 \text{ bytes} \times 2 \\ &= 68,272 \times 10^6 = 68.272 \times 10^3 \times 10^6 \\ &\approx 68 \text{ GB/s}\end{aligned}$$

   The theoretical memory bandwidth is therefore **68 GB/s**.

4. I/O
   (a) Bus Type: PCI Express (PCIe)
   (b) Bandwidth: 32 bits (4 bytes per cycle)
   (c) Clock speed: 33 MHz

5. Disk: [5]
   (a) Type: NVMe
   (b) Model: SAMSUNG MZVLQ1T0HBLB-00BH1
   (c) Capacity: 1 TB
   (d) Sequential Read: 2,366 MBytes/Sec
   (e) Sequential Write: 955 MBytes/Sec
   (f) Random seek RW: 821 MBytes/Sec
   (g) IOPS: 52 MBytes/Sec
   (h) The above specifications are from a benchmarking platform, as we were unable to find complete product info for the specific product listed, likely because it is manufacturer supply only, and not sold standalone to consumers.

   The closest rated specifications we could find were for the MZVLQ1T0HBLB-00B00 (different SKU):
      i. Sequential Read: 3,100 MB/s, 380000 IOPS
      ii. Sequential Write: 2,000 MB/s 330000 IOPS

6. Network card:
   (a) Model: Intel Alder Lake-P PCH CNVi WiFi
   (b) Bandwidth: 1.1342 Gb/s

7. Operating system: Ubuntu 24.04.03

3 CPU, Scheduling, and OS Services

3.1 Timing overhead

CPUs maintain a timestamp register that can be used to benchmark the number of cycles that an operation takes on the processor. The most basic instruction is `RDTSC`, Read Time-Stamp Counter, and `RDTSCP`, which in addition reads the processor's ID. The latter instruction is used as a way to serialize the reading of the timestamp and ensure that all previous instructions have been completed in full before reading the timestamp [9].

Additionally, Paoloni [9] suggests guarding these instructions with the `CPUID` instruction, which is another forcibly serialized operation to prevent pipeline pollution before and after our target code and timestamp instructions are running. By guarding measurements with this instruction, we get a common boilerplate for measuring time:

```
asm volatile("CPUID\n\t"
             "RDTSC\n\t"
             "mov %%edx, %0\n\t"
             "mov %%eax, %1\n\t"
             : "=r"(start_high),
             ↪  "=r"(start_low)::"%rax", "%rbx",
             ↪  "%rcx", "%rdx");

// measure something here

asm volatile("RDTSCP\n\t"
             "mov %%edx, %0\n\t"
             "mov %%eax, %1\n\t"
             "CPUID\n\t"
             : "=r"(end_high),
               "=r"(end_low)::"%rax", "%rbx", "%rcx",
                 ↪  "%rdx");

start = ((uint64_t)start_high << 32) | start_low;
end = ((uint64_t)end_high << 32) | end_low;

uint64_t time = end - start;
```

As Paoloni describes, this empty block measure the overhead of these instructions itself. Often times, these instructions are repeated before an actual measurement is taken to warm the instruction cache and prevent over-fluctuations.

3.1.1 Overall Prediction of Performance

Looking at the code for the timing overhead described by Paoloni, it is about 15 instructions, and we have about 1.27 CPI, thus we should be at approximately 19.1535 cycles for measurement or about $\approx$ 10.1801 ns given our 0.5315 ns/cycle.

3.1.2 Measurement Methodology

We use the same setup as Paoloni explained above without anything inside the loop to measure. We experiemented with adding a NOP operation, `;`, which places in a `mov`, but did not see any changes in our results and therefore opted to stay faithful to Paoloni's implementation.

We use the measurement methodology described in 3.1.3 with no changes.

3.1.3 Taking measurements

As we employ a similar measurement technique in most of our benchmarks, we consolidate the information here and mention when any deviations are made.

The boilerpate timing method shown in section 3.1 is wrapped around a loop that runs, by default, 1,000,000 iterations. We call each loop of 1,000,000 a *trial*,

Each measurement is conducted for ten trials to take the standard deviation and average across these trials.

We occasionally change the number of iterations conducted per trial if loops or other types of iteration are being done within the code being measured, so we have a similar count of iterations being done for each.

3.1.4 Results

See table 1 for the results of this benchmark.

The average cycles being very close to 15 instructions means our system is performing consistently with our estimates without any additional overhead or instruction translation occuring. This gives us confidence in our measurements n the target machine as our virtualization and emulation environments used for development purposes showed different overheads, probably due to trapping into a hypervisor to access these low-level registers.

Avg. Cycles	StdDev Cycles	Latency (ns)
15.4071	0.2226	8.1887

Table 1: Results for timing overhead

3.2 Loop overhead

3.2.1 Overall Prediction of Performance

In Paoloni's white paper [9], a nested loop structure is tested with an increasing maximum iteration size to validate the resolution of `RDTSC` being able to capture a single assembly instruction. Indeed, the timing of the loop increased roughly by 1 over time. We therefore **estimated a similar 0-1 cycles** for the overhead of a loop which includes its initialization, comparison, and increment.

3.2.2 Measurement Methodology

Instead of doing the nested structure of a measurement inside the looping described in section 3.1.3, we increment the number of loops to *100,000,000 iterations* and do an arbitrary instruction within the loop so there is enough resolution to capture by timestamping:

```
for (int i = 0; i < NUM_LOOPS; i++) {
    (*dummy_ptr) = 1;
}
```

Looking at the code for the loop, it is about 5 instructions and multiplied with our estimated 1.27 CPI we get approximately 6.3845 cycles. We therefore adjusted our estimate to be roughly 6 cycles or 3.1989 ns given our method.

3.2.3 Results

See table 2 for the results of this benchmark.

The results being close to one cycle matches our estimates and what Paoloni found. We believe there is some amortization or other latency hiding over the loops that cause the real average to appear less than a whole number, but this is not a cause for concern as it is not fully off from 1.

Avg. Cycles	StdDev Cycles	Latency (ns)
0.9337	0.0317	0.4963

Table 2: Results for looping overhead

3.3 Procedure call overhead

3.3.1 Overall Prediction of Performance

To estimate software overhead, we the took the *objdump* of our benchmark binary and counted how many instructions each procedure call required. We then multiplied our instruction count with our estimated CPI of 1.2769 to get our estimate cycles in table 3.

We estimate that each additional argument that adds about three instructions means a jump of relatively 3.8307 cycles or 0.9902 ns.

# Args	Instr. Count	Est. Cycles	Est. Latency (ns)
0	6	7.6614	4.0720
1	7	8.9383	4.7507
2	10	12.769	6.7867
3	13	16.5997	8.8227
4	16	20.4304	10.8588
5	19	24.2611	12.8948
6	22	28.0918	14.9308
7	24	30.6456	16.2881

Table 3: Estimates for procedure calls with arguments. Instruction count gathered from `objdump`

.

The following is the assembly dump from our benchmark, explained in the next section.

```
0000000000402a84 <procedure_test_zero_arg>:
```

```
  402a84:        f3 0f 1e fa                 endbr64
  402a88:        55                          push   %rbp
  402a89:        48 89 e5                    mov    %rsp,%rbp
  ; no arguments are passed in here
  402a8c:        b8 00 00 00 00              mov    $0x0,%eax
  402a91:        5d                          pop    %rbp
  402a92:        c3                          ret

0000000000402a93 <procedure_test_one_arg>:
  402a93:        f3 0f 1e fa                 endbr64
  402a97:        55                          push   %rbp
  402a98:        48 89 e5                    mov    %rsp,%rbp
  ; one argument passed
  402a9b:        89 7d fc                    mov    %edi,-0x4(%rbp)
  402a9e:        8b 45 fc                    mov    -0x4(%rbp),%eax
  402aa1:        5d                          pop    %rbp
  402aa2:        c3                          ret

0000000000402aa3 <procedure_test_two_arg>:
  402aa3:        f3 0f 1e fa                 endbr64
  402aa7:        55                          push   %rbp
  402aa8:        48 89 e5                    mov    %rsp,%rbp
  ; two arguments passed
  402aab:        89 7d fc                    mov    %edi,-0x4(%rbp)
  402aae:        89 75 f8                    mov    %esi,-0x8(%rbp)
  402ab1:        8b 55 fc                    mov    -0x4(%rbp),%edx
  402ab4:        8b 45 f8                    mov    -0x8(%rbp),%eax
  402ab7:        01 d0                       add    %edx,%eax
  402ab9:        5d                          pop    %rbp
  402aba:        c3                          ret
```

3.3.2 Measurement Methodology

For each number of arguemnts, 0-7, we defined a function and call it repeatedly in the inner loop of the boilerplate in 3.1.3.

As the previous *objdump* shows, we confirmed that even with a trivial function, the call was not optimized away and instructions were added.

To ensure this, we also added arguments together when possible to add more work to the function, else it would be amortized away through the numerous loops not yielding any substantial numbers.

```c
int procedure_test_two_arg(int a, int b) {
    return a + b;
}
```

3.3.3 Results

See table 4 for the results.

Our estimate was slightly too high, and we see that each additoinal argument adds roughtly ≈ 0.6 ns.

Our results are monotonically increasing with the number of arguments passed in as we expected. Our low cycle count may stem from the way we average out our total measured time, and perhaps some amortization from the procedure being cached within the instruction cache. These could mask latency after several iterations that creates a low average.

The instructions that make up the procedures are releatively simple and might have require less cycles than the instructions used for our CPI estimate. Due to this, our prediction of performance might be overpredicting the impact of each additional instruction.

# Args	Avg. Cycles	StdDev Cycles	Latency (ns)
0	0.9589	0.0197	0.5097
1	1.2971	0.0200	0.6894
2	1.5398	0.0161	0.8184
3	1.8196	0.0186	0.9671
4	2.0670	0.0150	1.0986
5	2.2247	0.0091	1.1824
6	2.5476	0.0112	1.3540
7	2.7040	0.0063	1.4372

Table 4: Cycle counts for procedure calls with arguments

3.4 System call overhead

3.4.1 Overall Prediction of Performance

The cost of a system call is much higher than the cost of a procedure call, since we need to mode switch into the kernel/ring 0, and back, for each system call. This requires a mode switch, which is noted to take around 100 cycles on X86. Thus, our estimate should include the time for a procedure call plus two mode switches: 200 + 7.6 (our estimated for zero argument call) for ≈ 207.6 cycles, ≈ 110.3394 ns.

3.4.2 Measurement Methodology

We use *lmbench*'s method of invoking a `write()` call to `/dev/null` as the authors describe it as a "nontrivial entry into the system" [7] does not have any special optimizations compared to other system calls.

We use the boilerplate in 3.1.3 with `write()` invoked inside the loop repeatedly.

```c
for (int i = 0; i < DEFAULT_NUM_LOOPS; i++) {
    write(dnfd, "", 0);
}
```

3.4.3 Results

See table 5 for results.

Our estimate overshot the actual results but was correct in terms of magnitude; perhaps the cost of a mode switch is lower than what we expect.

A system call taking more than a hundred cycles is understandable due to the amount of work that must be done before the OS gains control. It is evident how this overhead can drastically increase the latency of a workload if there are frequent system calls being made.

Avg. Cycles	StdDev Cycles	Latency (ns)
139.5415	0.1796	74.1663

Table 5: Results for system call overhead

3.5 Task creation time

3.5.1 Overall Prediction of Performance

A kernel thread time to create and run should be much cheaper than a process's time to create and run. To create a process, a process needs to get its own virtual address space, page tables, file descriptor table. In contrast, a kernel thread needs a thread control block (few pointers and register values) and a kernel stack which is less costly compared to the process's creation requirements. The creation of a kernel thread

The *lmbench* paper's measurement of process create and run (fork and exit) is done on a Linuxi686@167Mhz environment which is 1881.600/167 = 11.27 times slower than the processor we are testing on. Their resulting measurement was 0.4ms for fork and exit. This translates to about 400000 nanoseconds, thus we esitmate:

$$\frac{400000ns}{11.27 \text{ (speedup)}} = 35492.4579ns$$

The creation of a kernel thread requires the creation of a new stack, and save and restore of program counter and stack pointer registers. This is much lighter than a process fork. Thus we estimate the kernel thread taking about half of the process creation time.

- Kernel thread: 17746.2289 ns
- User process: 35492.4579 ns

3.5.2 Measurement Methodology

For measuring the time to create and run a kernel thread, we used `pthread_create(&thread, NULL, enter, "test thread")` to create a kernel thread. The start routine, `enter` returns `NULL` when called to exit the thread. To check if the thread has finished running, we call `pthread_join(thread, NULL)` which returns immediately when a thread is finished running. We run this with our timing setup shown in section 3.1 and over 10000 iterations.

For measuring the time to create and run a process we followed MvVoy's *lmbench* process measurement method. In MvVoy's *lmbench* paper [7], they measure process creation by using a combination of `fork()` for creation and `wait()` to block until the child exits. [7]. We used the same method to measure the time to create and run a process. We used the same timing setup shown in section 3.1 for this measurement and iterated over 100 iterations.

3.5.3 Results

See table 6 for results on task creation.

Our estimate for kernel threads was nearly double the actual result, while our estimate for user processes was slightly lower. However, the standard deviation for these measurements may explain some of the difference for the latter.

Our prediction for kernel threads being lighter to create than a user process is consistent and validates our measurements.

	Avg. Cycles	StdDev Cycles	Latency (ns)
Kernel thread	15929.3053	222.5187	8466.4258
User process	80533.1470	8217.7287	42803.3676

Table 6: Results for task creation times

3.6 Context switch time

3.6.1 Overall Prediction of Performance

A process-to-process context switch should be much more expensive than one kernel thread to another. To context switch, the OS needs to save the state of the running process, and restore the context of the swapped process. This call should be much more expensive than any other runs, considering the TLB flush means there is a TLB miss caused for every later memory access.

In contrast, a kernel thread context switch occurs within the same address space in the kernel, so there is less memory movement or saving. Only a few registers for the program counter and stack pointer would need to be saved.

The *lmbench* paper measures context switches on a Linux i686@167Mhz environment which is 1881.600/167 = 11.27 times slower than the processor we are testing on. A (user) process-to-process with size 32KB context switch took about 18 microseconds (18,000 ns).

$$\frac{18000ns}{11.27 \text{ (speedup)}} = 1597.1606ns$$

If we assume that a kernel thread-to-thread context switch is about half of this, we estimate:

- User process-to-process = 1597.1606 ns
- Kernel thread-to-thread = 798.5803 ns

3.6.2 Measurement Methodology

We model our benchmark similar to the *cswitch* benchmark in *lmbench* [8] which utilizes inter-process pipes to communicate and synchronize between processes/threads. The `pipe` system call generates a virtual file-descriptor from an integer buffer which can then be used as `stdin` and `stdout` descriptors to interact with.

We use a "flip-flop" approach where two pipes are used such that a process is a producer (writing a byte) to one pipe while being a consumer (reading a byte) from the other pipe. With two such processes, we essentially create a back-and-forth pattern of unblocking the other thread with a write while getting blocked on a read.

Our implementation times the write that will unblock the other thread, getting blocked upon a read, the context switch (control given to the scheduler and then restoring state of the other), and continuing execution within the other process.

To prevent deadlock, we must ensure both processes are not doing the operations in the read-write order, otherwise there could be a time where both processes are blocked on a `read` for a byte that is never arriving. Conversely, if both processes are doing a write-read order, then the processes will be unblocked far too quickly for the benchmark to make sense. Therefore, we must keep the two processes in opposite operation order to prevent either.

We call a write-read process a *produce-first* process and a read-write process a *consume-first* thread. The produce-first thread will always start the back-and-forth sequence upon its first write.

```
// Process or thread 1: produce-first
write(A_stdout, buf, 1);
read(B_stdin, buf, 1);

// Process or thread 2: consume-first
read(A_stdin, buf, 1);
write(B_stdout, buf, 1);
```

We use the boilerplate measurement code from 3.1.3. The number of loops corresponds for how many iterations of this sequence we execute before it is `kill()`/`pthread_join()`.

User-level processes are created through `fork()`, and the inner body of the time-measuring loop checks the process ID of the two processes to execute either the produce-first versus consume-first code.

```
if (pid == 0) { // Producer-first code
    write(filedes_A[1], buf2, 1);
    read(filedes_B[0], buf2, 1);
} else {        // Consumer-first code
    read(filedes_A[0], buf, 1);
    write(filedes_B[1], buf, 1);
}
```

Kernel-level threads are created through the POSIX library API, `pthread_create`, and time-stamping code is moved into the execution of the produce-first thread and is returned via a pointer upon `pthread_join`. The loop body is present in both threads, and the flipped operation order ensures that all iterations will be fulfilled without deadlock.

3.6.3 Results

See table 7 for the resulting times.

When subtracting the overhead for pipe operations, we arrive at 586.8521 ns for the kernel threads measurement 761.8344 ns for the user processes. Our estimate for kernel threads was below the actual while our estimate for user processes was accurate.

Additionally, our claim for kernel threads having lower context switch times is supported by our results. Overall, these figures support our methodology being accurate and consistent.

	Avg. Cycles	StdDev Cycles	Latency (ns)
Pipe overhead	1089.6857	1.9357	579.1649
Kernel threads	3359.9068	1.0581	1785.7905
User processes	4036.8210	4.0732	2145.5704

Table 7: Results for context switch times

4 Memory

4.1 RAM access time

4.1.1 Overall Prediction of Performance

Base hardware performance: Our CPU id 0 has a clock rate of roughly 3.9 GHz, which is 21 times faster than the DEC Alpha@182 MHz that *lmbench* measured. By scaling down their reported times, we estimate:

1. L1 hit time: 0.4762 ns
2. L2 hit time: 2.3810 ns
3. L3 hit time: 10 ns

   The processor in the *lmbench* paper did not have an L3 cache, so this estimate is derived from the 5x difference from 0.4762 ns to 2.3810 ns.

4. Main memory hit time: 35 ns

   Since we have another level of cache, we estimate the main memory access to instead be a factor of 10x faster than the processor in *lmbench*.

Software overhead: The code that makes the memory access is one load instruction: `ptr = *ptr;`. We measure the software overhead to therefore be about five cycles, or $0.5315 \times 5 = 2.6575$ ns.

Summing the estimates in the previous two sections:

1. L1 cache (35 KiB $\approx 2^{15}$ B) : 3.1337 ns
2. L2 cache (1.625 MiB $\approx 2^{20}$ B): 5.0385 ns
3. L3 cache (12MB $\approx 2^{23}$ B): 12.6575 ns
4. Main memory (16GB $\approx 2^{34}$ B): 37.6575 ns

4.1.2 Measurement Methodology

For measuring memory and cache access time, *lmbench* employees a linked list approach where a number of traversals are done on a variable-sized array and strides: how many elements forward does the current index reference. However, today caches can optimize accesses when doing a fixed-length stride and prefetch or rearrange instructions, leading to inaccurate results on modern processors.

While we employ a similar "pointer chasing" mechanism, we use randomized strides so that the compiler cannot predict where an element points to. For array sizes $n \in \{2^x | x \in [10, 30]\}$ of `uint64_t` elements (4 bytes each), and for each index k in the array, we randomly shuffle the array using the Fisher-Yates algorithm (see section 4.1.3). Then, we make another array of the same size and set each index `i` to contain the address of the index array element in front of it. That is:

```
uint64_t indexes[n] = ( array of randomized integers
↪  corresponding to valid indices [0, n - 1] );

uint64_t *arr = ( array of pointers to the elements
↪  of indexes );

arr[i] = &arr[ indexes[i + 1] ];
```

In one iteration, we set a `volatile uint64_t* ptr` to the first element, `arr[0]`, and then conduct the following loop to "pointer chase" through the array, accessing random indices and triggering cache misses:

```
ptr = arr[0];
for (int j = 0; j < ARR_ACCESSES; j++) {
    ptr = (uintptr_t *) *ptr;
}
```

Since the indices and therefore addresses are randomized, this eliminates the potential for the CPU to do any sort of prefetching that would make array accesses and dereferences seem faster than they should be.

`ARR_ACCCESSES` is set to 10,000,000 iterations, and we employ the measurement methodology described in section 3.1.3, but set the number of measurement loops to ten while keeping the same number of trials.

4.1.3 On Randomness

The Fisher-Yates algorithm [13] for pseudo-randomly shuffling an array guarantees that all indices will be referenced in the final array. The algorithm runs in $O(n)$ time complexity.

For every index $i \in \{0, n-1\}$, an index $j \in \{0, i-1\}$ is randomly selected to swap into index i, then i is decremented. In pseudocode:

```
for j = n - 1, decrement to 0
    pick j from [0, i - 1]
    swap(j, i)
```

Since any i is not referenced again, this guarantees the "safety" of the element not being obstructed or removed in the future. Every index is selected once, and only once, so all of the original elements will be placed without loss.

We implement Fisher-Yates as a utility function due to it being used in many of our benchmarks. It takes in an array of `void *`, taking in the size of the elements in the array to properly swap elements through `memcpy`.

4.1.4 Results

See figure 1 for the plot of latency in nanoseconds against array sizes, and table 8 for the raw data. The summary of our estimated latency for each of the memory levels is in table 9.

Our estimates overestimated the latency of the L1 cache while underestimating the lower caches. Overestimating the L1 cache is potentially due to use of registers or other optimizations done by the CPU. Once we leave the realm of CPU cache, the cost of accessing main memory increases rapidly as expected.

The L3 cache is subtle to detect due to it being shared across all the cores in newer Intel processors. While we ran the benchmark pinned to one CPU, the jump between L3 cache and main memory is not as sharp as what is seen in *lmbench* but the steep rise around 2^{24} and after shows the data spilling into main memory.

From the known sizes of each level in section 2 and the curve, we see the transitions for each level at:

1. L1 $\rightarrow$ L2: 2^{15} B
2. L2 $\rightarrow$ L3: 2^{20} B
3. L3 $\rightarrow$ Main memory: Unclear, mostly after 2^{24}.

4.2 RAM bandwidth

4.2.1 Overall prediction of performance

From the section 2, we see that our theoretical memory bandwidth is at 68 GB/s. For read speed, we predict minimal overhead, thus we expect a figure nearing 68 GB/s. For write speeds, we expect more overhead, since there will be cache write back and invalidation for each write, and DRAM architecture is generally prioritized for read. Thus, we expect write bandwidth to be slightly lower than read. Thus, we expect our actual benchmark to be somewhat lower than the real bandwidth, at around 50 GB/s.

4.2.2 Measurement Methodology

To measure memory bandwidth for reads, we used `posix_memalign` to ensure each read we make is within one cache line, to prevent overhead from reading across cachelines, and doubling our latency. We also ensure to set the

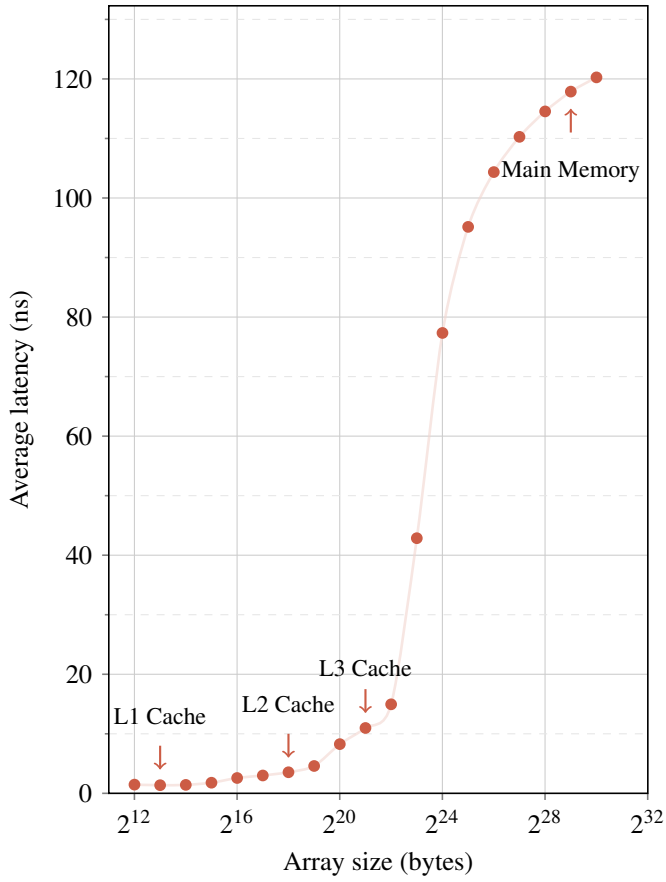

Figure 1: Graph of average memory access latency against the size of the array accessed in bytes.

# Bytes	Avg. Cycles	StdDev Cycles	Latency (ns)
2^{12}	2.7510	0.1823	1.4622
2^{13}	2.5718	0.2122	1.3669
2^{14}	2.6685	0.2291	1.4183
2^{15}	3.3549	0.0587	1.7831
2^{16}	4.8467	0.0684	2.5760
2^{17}	5.6398	0.0675	2.9978
2^{18}	6.6601	0.0486	3.5398
2^{19}	8.6428	0.0504	4.5936
2^{20}	15.5498	0.0383	8.2647
2^{21}	20.6528	0.0481	10.9770
2^{22}	28.1333	0.1567	14.9528
2^{23}	80.6306	0.3845	42.8552
2^{24}	145.4963	0.3906	77.3313
2^{25}	179.0189	0.3961	95.1485
2^{26}	196.3394	0.2029	104.3544
2^{27}	207.4733	0.2302	110.2721
2^{28}	215.5087	0.3039	114.5429
2^{29}	221.7692	0.3299	117.8703
2^{30}	226.2648	0.4286	120.2597

Table 8: Data from memory access measurements

Cache Level	Size (bytes)	Latency (ns)
L1 Cache	2^{15}	$\approx$ 1.5 - 2
L2 Cache	2^{20}	$\approx$ 2.5-5
L3 Cache	$\approx 2^{23}$	40 <
Main Memory	2^{34}	110 <

Table 9: Summary of memory access times

total allocated memory to be 1GB, larger than the L3 cache as defined in 2, by having be much larger than the L3 size, we can ensure the latency of the test is dominated by DRAM read. We unroll the inner memory loop to read 8 at a time, similar to the *lmbench* methodology, such that a constant offset is used, and we incur a load and add for each word. The benchmark methodology for write bandwidth is the same as above, except instead of accumulating and adding values from our byte buffer, we store to it using local dummy variables.

We found that with the above benchmarks on a single thread, we were not able to sufficiently saturate the memory, thus leading to falsely low values bandwidth. Thus, we spawned 2 threads, each performing the above read and accumulate/write loop, and used the combined values as the memory bandwidth.

We utilize the same benchmark boilerplate as in 3.1.3.

4.2.3 Results

See the benchmark's results in table 11. The calculation to obtain the bandwidth in GB/s are as follows:

$$\frac{1 \text{ byte}}{0.0832 \text{ cycles}} \times \frac{1GB}{2^{30}B} \times \frac{1 \text{ cycle}}{0.5315ns} \times \frac{10^9 ns}{1s} = 21.0607 \text{ GB/s (R)}$$

$$\frac{1 \text{ byte}}{0.1141 \text{ cycles}} \times \frac{1GB}{2^{30}B} \times \frac{1 \text{ cycle}}{0.5315ns} \times \frac{10^9 ns}{1s} = 15.3572 \text{ GB/s (W)}$$

Therefore, our measured bandwidths are:

- 21.0607 GB/s for reads
- 15.3572 GB/s for writes

Our measured results are about three times smaller than our estimates and the theoretical bandwidth. This could be due to other system load that is out of our control, software pipelining or context switches that occupy the memory bus in other ways, or simply degradation of hardware from use. Since our measurements seem reasonable for a system of this caliber, we believe our benchmark methods to be correct.

# Processes	Avg. Cycles/byte	StdDev Cycles/byte	Latency/byte (ns)
Read	0.0832	0.0083	0.0442
Write	0.1141	0.0154	0.6064

Table 10: Results of estimating cycles for bytes.

Table 11: Results from the memory bandwidth benchmark. Each row represents a trial.

4.3 Page fault Service Time

4.3.1 Overall Prediction of Performance

The cost of page faulting will be dominated by disk read time. Hence, we're basing our estimate around the estimated disk read time for a page in our system (4KB). In our test machine, our NVMe's IOPS for 4KB with queue depth of 1 is around 53 Mbytes/sec. We're using this metric to estimate a page fault because its 4KB size matches our machine's page size and a queue depth of 1 makes it synchronous to match our testing workload. We can then find the latency for one 4KB chunk through the calculations below:

$$\frac{52 * 10^6 \text{ bytes}}{1 \text{ sec}} * \frac{1}{4096 \text{ bytes}} = 12695.3125 \text{ 4KB ops/ sec}$$

$$1/12695.3125 = 0.00007876923 \text{ sec} = 78.76923\mu s$$

Therefore, our prediction for a page fault is $\approx 78.76923\mu s$.

4.3.2 Measurement Methodology

To measure page faults, we used `getrusage()` to track whether we're hard faulting or soft faulting. The kernel may have loaded in pages in memory prior so to make sure that we're not reading from memory we evict pages pointed to by the file descriptior with `posix_fadvise(fd, 0, len, POSIX_FADV_DONTNEED)`. We use `mmap(NULL, len, PROT_READ, MAP_PRIVATE, fd, 0)` to create a virtual mapping of the file into memory in order to programmatically trigger the page faults through virtual address accesses.

Our test file to page into memory is a 512 MB test file generated by `dd if=/dev/zero of=./testfile.bin bs=1M count=512 status=progress`.

In Ubuntu, the kernel optimizes reads by reading ahead and pulling in the consecutive pages, leading to soft faults. To avoid soft faults, we randomize our selection of page and also notify the kernel that we're selecting out of order with `posix_fadvise(fd, 0, len, POSIX_FADV_DONTNEED)`. We also set `madvise(data, len, MADV_RANDOM)` to make sure that when we access our data, the kernel doesn't choose read-ahead or caching techniques that would prevent hard faults. We used the Fisher-Yates algorithm to randomize the access order.

To make sure that we are only triggering hard faults, we use `getrusage(RUSAGE_SELF, &stats)` to check for faults before and and after the running the faults. In `rusage` struct, we used `ru_minflt` to track page reclaims or soft faults, and `ru_majflt` to track page faults or hard faults. In our benchmark, we track take a snapshot of the rusage prior to faults and another snapshot after and take the difference to assure only hard faults are triggered.

We ran 15 iterations of our experiment and each iteration of the experiment in figure 12 consists of the following:

1. Open 512MB test file
2. `mmap` to create a virtual mapping of the file to memory
3. Use Fisher-Yates algorithm to generate a random access order (see section 4.1.3)
4. Record start timestamp register value
5. Access data using randomized access order to trigger page faults
6. Record end timestamp register value
7. Clear file cache and clean up

After every run, to make sure that the file data is evicted from cache, we use employ three tactics. First, we use `madvise(data, len, MADV_DONTNEED)` to advise the kernel not to expect access to the data anymore. We use `madvise(data, len, MADV_DONTNEED)`. Second, we use `posix_fadvise(fd, 0, len, POSIX_FADV_DONTNEED)` to request the kernel to free the cached pages. Third, we use `munmap(data, len)` to free the data in memory.

We run this benchmark with the same measurement boilerplate as in section 3.1.3. The only difference is we ran this experiment for 15 trials.

4.3.3 Results

See table 12 for the table of cycles per fault and latency per fault in each trial, and table 13 for the mean and standard deviation.

Our resulting latencies are within the same mangnitude of our estimate. The key difference is that our estimate is predicted to be around $12\mu s$ slower with the estimate being $\approx 78\mu s$ and our experiments yielding around $91\mu s$. We speculate that the disk benchmark used to calculate IOPs at 4KQ1 may have some additional overhead that we're not aware of. In addition, there could be differences in the setups of the machines when running the benchmark that can differentiate the latency such as how the OS handles faults and the bandwidth between memory and disk.

Our resulting latency is around the expected range of NVMe latencies, $10\mu s$ - $100\mu s$ [12], at $91\mu s$. This demonstrates our experiment is within the ballpark range. Although,

Trial #	Faults	Avg. Cycles/fault	Latency/fault (μs)
1	131072	173540.8528	92.2370
2	131072	168969.9116	89.8075
3	131072	168652.4496	89.6388
4	131072	169095.0107	89.8740
5	131072	169032.6367	89.8408
6	131072	170580.4997	90.6635
7	131072	169946.7702	90.3267
8	131072	170005.5584	90.3580
9	131072	170869.6359	90.8172
10	131072	169563.9709	90.1233
11	131072	170969.1180	90.8701
12	131072	170978.1720	90.8749
13	131072	175018.8471	93.0225
14	131072	169088.2336	89.8704
15	131072	174521.1111	92.7580

Table 12: Page fault latency measurements using CPU frequency of 1881.600 MHz. A 512 MB should have 131072 faults, $(512MB * 2^{20})/4096B$ *page size* $= 131072$ *accesses*

Metric #	Mean	Std. Dev
Avg. Cycles	170722.1852	1983.0779
Latency (μs)	90.7388	1.0910

Table 13: Mean and standard deviation of page fault measurements computed using CPU frequency of 1881.600 MHz.

it's unclear whether or not we're optimistic with our benchmark.

To answer how our experiment compares to the latency of accessing a byte from main memory, it is magnitudes slower. For RAM access time, we were measuring with nanoseconds and for faults, we were measuring with microseconds. In our RAM access benchmark with our largest heap-allocated array at 1 GB, the latency is $58.4895ns$ about 1556x faster than page faults at $91\mu s$.

5 Network

5.1 Remote Windows Machine

For these tests, we used a second device operating Windows connected over LAN to benchmark non loopback tests. An abridged specification of this machine is below. The important specification is the network adapter being 1Gb/s.

Windows Device:

1. OS: Windows 10 Pro
2. CPU: Processor AMD Ryzen 5 5600X 6-Core Processor, 3701 Mhz, 6 Core(s), 12 Logical Processor(s)
3. RAM: 32Gb 3600Mz
4. Network Adapter: Intel(R) Wi-Fi 6 AX200 160MHz, speed, 1Gb/s

5.1.1 Network Skeleton

For all of the below tests, we used a similar skeleton code implementation, and simply changed or added send and receive call of various sizes. On the Windows side, we use the winsock library to setup a TCP listener socket on a different port for each test, then accepting the connection in a while true loop, so that the Ubuntu client can be easily tuned and rerun while leaving the Windows instance running. The inner loop on the Windows side is then tuned for each test, sending and/or receiving data of various sizes as aligned to the request and response of the Linux TCP client.

On the Ubuntu side, we utilized the posix connect function and the arpa/inet.h and netinet/in.h libraries to setup a basic TCP client implementation that we share utilization of for both the remote and loopback testing code. In each test, we begin by creating a socket of type AF_INET for IPv4, and SOCK_STREAM for TCP, then we take an address, either 127.0.0.1 for loopback, or the Windows remote ip address obtained from ipconfig on the Windows server host, and a preset port number for each test (4000 + offset), using inet_pton to convert this to an address, then calling connect to open a connection to either the loopback client or the Windows server. For each test, we then call a separate inner loop, which includes our timing code for the network function or operation of interest 3.1.3.

Our loopback receiver server is essentially a linux port of the Windows code, using the inet and posix libraries to establish a listener socket, and then accepting connections in a while (1) loop, which allows it to remain running. Within each accept and close, a server version that performs the appropriate send and receive for that specific test is implemented.

5.2 Round trip time

5.2.1 Overall Prediction of Performance

To predict the performance of RTT, we must first split our prediction across protocol used. The ICMP protocol is connectionless, does not carry data, and does not guarantee delivery, as opposed to TCP which is built for reliable in order delivery. This initially may seem like thus ICMP should perform much better. However, ICMP packets are treated at a lower priority than TCP by routers and network stacks, thus we predict that our TCP RTT should be lower, as we measure RTT after the connection is established.

ICMP RTT: For estimating remote RTT, we see that in our network configuration, there is one hop through the home router from the Ubuntu laptop device to the Windows "host" machine. Thus, from previous experience with ICMP ping to remote hosts taking around 5ms over the internet with more hops, our estimate should be somewhere below 5ms, which

we will estimate at around 3ms. Since ICMP packet responses are low priority in the network stack, we also predict high standard deviation across these measurements. [2] For the loopback latency since the loopback interface of a device does not utilize the nic and instead is routed in software back to the host device software stack. Due to this, we should observe very low RTT in our loopback rtt test,which we can assume will not differ much from the fastest lmbench UDP benchmark, at 93 microseconds, or 0.093 milliseconds. [7]

TCP RTT: Since we measure the RTT of the TCP protocol, which as discussed above is a higher priority in the network stack, and we measure rtt after the initial establishing of the TCP connection, we assume TCP latency will be lower than that of ICMP. [1] On a remote device, we estimate this to be around 1ms since it is only one hop from the test machine to the remote server. On the loopback interface, this should only be the time it take in the software network stack to be sent and received. Thus, should be similar to the fastest lmbench TCP latency, at 0.162 milliseconds. [7]

5.2.2 Measurement Methodology

For measurement of kernel level RTT, we used the ICMP protocol based ping utility over many runs, the ping tool packages it's own averages and standard deviations as well, so our measurment is cleanly done using it.

For the measurement of the TCP latency, we utilized a TCP send and receive loop between our local environment and the remote or loopback host. By having the host send then receive, and the remote receive then send, we guarantee that we measure 1 rtt of communication each iteration, without our rtt being optimized out by the sliding window of TCP protocol letting us send un-acked packages. We then ran this benchmark over 1000 sends timing the overall time, with 10 runs of the 1000 send benchmark. We utilized a smaller overall trial count for network measurments due to the large latencies of networked trials, especially with regards to bandwidth.

5.2.3 Results

Loopback Latency avg (ms)	Loopback Stddev(ms)
0.03	0.007

Table 14: ICMP RTT to Loopback using the ping utility

Remote Latency(ms)	Remote Stddev(ms)
3.248	4.243

Table 15: ICMP RTT to Remote using the ping utility

Loopback Latency avg (cycles, ns)	Loopback sdev (cycles, ns)
22963.1941, 12204.93766	1346.7967, 715.8224461

Table 16: TCP rtt measurements using CPU frequency of 1881.6 MHz.

Remote Latency (cycles, ns)	Remote sdev (cycles, ns)
46006109.512, 2129247.206	269459.6948,143217.8278

Table 17: TCP rtt measurements using CPU frequency of 1881.6 MHz.

5.2.4 Analysis

From the, the first obvious difference is the remote compared to loopback latencies for both ICMP and TCP, where the difference between the RTT for both protocols is in the factor of around 100x. This demonstrates that, as predicted a majority of the RTT latency between two processes it in the network transfer, from laptop over the wifi NIC, to the router, then to the receiving device. Additionally, we can see that the TCP latency is lower than the ICMP latency. This matches our expectation that while TCP holds state information, it's treated as a higher priority and is thus induces slightly less overhead and thus a faster RTT than ICMP. We also see very large standard deviation for the remote trials, which matches our expectation of a half-duplex wifi device (which can only send or receive and not simultanously), being routed through a router with other traffic on it. From the above results we can conclude that the Loopback Latencies are effectively the raw software overhead latency, and thus the 1 hop network induced latency is 3.245 ms for ICMP, and 2117042.26834ns for TCP. This shows that network latencies are generally dominated by time through NIC hardware and in the network. The discrepancies between our benchmarks and lmbench is likely addressed by running on newer and thus faster hardware allowing the software stack to execute faster.

5.3 Peak bandwidth

5.3.1 Overall Prediction of Performance

From the hardware specification, we see that our network interfaces on both machines are around 1.0Gb/s. Thus our peak possible bandwidth is 1 Gigabit per second. Since the network device on our one side is based on wifi we assume that for simultaneous read and write, we are operate at half peak bandwidth because it is a half duplex connnection. To avoid this, we split our bandwidth into a read and write latency bandwidth.

Remote bandwidth: Since our peak 1.0Gb/s for both sides of the link, we assume the is the effective peak bandwidth. Given that TCP needs to acknowledge every send over the

abovementioned half duplex connection, and that network latencies and bandwidth are a variable factor, due to internals of the TCP protocol like [1] congestion control. Thus, we estimate our peak read to be around 500Mb/s. Since our client and server are mirrored, the read bandwidth should be around the same, as we are on a LAN network, and not limited by a provider for maximum write speed.

Loopback bandwidth: Since we do not go through the network hardware of the laptop device, we may be able to achieve above peak 1.0Gb/s. The transfer rate is limited only by the network stack of the kernel, and the memory/cpu hardware much more strictly than the previous software overhead, we cannot estimate off lmbench figures, as it should only be affected by the time to copy the data from the outbound to inbound network queue in the kernel. Since we are on much newer hardware than the lmbench Thus, we may be able to achieve a fraction of a memory level of latency, thus we estimate that we 10Gb/s.

5.3.2 Measurement Methodology

To measure read bandwidth, we use a client that reads 16000000 Bytes at a time in a `recv` loop, which writes back 1 byte to the remote host between each read to ensure that each write is fully propagated to the buffer before the next write. We mirror this methodology for the read latency, sending 1 byte, then receiving the 16000000 byte chunk. We also use `setsockopt` function on Windows to set the socket's send and receive buffer sized to 16000000, allowing us to more accurately see the latency from the network stack and RTT, instead of getting blocked on being unable to transfer our buffer cleanly to the kernel. we use `sudo sysctl -w net.core.wmem_max=16000000 sudo sysctl -w net.core.wmem_max=16000000 sudo sysctl -w net.core.rmem_max=16000000` on the Ubuntu test machine to mirror this optimization.

5.3.3 Results

Loopback Read Bandwidth avg (Cycle/byte, Mb/s)	Loopback Read Bandwidth stdev (cycles/byte)
0.2463, 60673.97483	0.0127

Table 18: Loopback TCP read bandwidth using CPU frequency of 1881.6 MHz

5.3.4 Analysis

In the above benchmarks, we see that our Loopback Bandwidth is much higher than our remote bandwidth. By converting the Mb/s values to GB/s for the Loopback Latency, we see values around 7GB/s which, while much lower than our memory bandwidth, greatly outpaces our NIC maximum value. This reflects the fact that the Loopback interface does not need to pass through the NIC hardware, and thus shows a much greater bandwidth throughput, as it is only related to the software stack latency to pass the bytes. On the network end, we see a read and write bandwidth of 734.6593646 Mb/s and 655.8125247 Mb/s respectively. Looking at the ideal NIC performance for both of our devices being 1Gb/s, we can reflect that we are around 70% utilization of hardware resource with our benchmark, however this is likely further influenced by the half-duplex nature of Wi-fi connections, meaning that even with or read/write dominanted workloads, there is transmission time that is lost due to Wi-fi being unable to send and receive simultaneously. We also note that Read and Write bandwidths are close to symmetric, unlike standard internet bandwidth test numbers, showing that Read/Write asymmetry is to connect to internet and ISP resources, and not a LAN limitation.

Remote Read Bandwidth avg (Cycle/byte, Mb/s)	Remote Read Bandwidth stdev (cycles/byte)
20.3414, 734.6593646	0.2542

Table 19: Remote TCP read bandwidth using CPU frequency of 1881.6 MHz

Loopback Write Bandwidth avg (Cycle/byte, Mb/s)	Loopback Write Bandwidth stdev (cycles/byte)
0.2802, 53721.62741	0.0405

Table 20: Loopback TCP Write bandwidth using CPU frequency of 1881.6 MHz

Remote Write Bandwidth avg (Cycle/byte, Mb/s)	Remote Write Bandwidth stdev (cycles/byte)
22.7870, 655.8125247	1.3698

Table 21: Remote TCP Write bandwidth using CPU frequency of 1881.6 MHz

5.4 Connection overhead

5.4.1 Overall Prediction of Performance

For connection overhead, for the TCP protocol, at a hardware level, we need at least 1.5 RTT to establish a connection between the client and server. This serves as the baseline time for connection overhead. On top of this, we must factor in whatever overhead is present in the network stack on each device. We can estimate this from our ICMP ping average of 3.2ms, multiplied by 1.5 to get 4.8ms as a likely overestimate, since ICMP is lower priority than TCP. Thus we can bring

our total estimate to around 4 ms.

For the loopback interface, we only have software overhead and kernel accesses to contend with, thus, we can estimate this off the lmbench results of 238 microseconds.

For connection shutdown overhead, since we do not queue any data as a part of this benchmark to avoid the effects of having to wait on a data flush, the connection shutdown overhead should be around 1 RTT, for the TCP fin and fin-ack packets. Since TCP is again higher priority than UDP, we can estimate around 3ms teardown time. For the loopback interface, this should again only be software bound, thus we estimate that we should be a fraction of the lmbench figures, at 100 microseconds.

5.4.2 Measurement Methodology

For connection overhead, we benchmarked the time it takes for the posix connect function to return a valid socket. We run this in a loop over 1000 trials, with 10 trials of each to obtain measurements, as in the above connection skeletion. Since the posix connect function waits until a valid socket is opened and thus the connection is established, we can be confident that the connection is usable and end to end connected by the time the connect function returns, and time the runtime across the 1000 trials.

We used a similar measurement methodology with close for tear down, calling the close(socket) function on linux client, and then timing the time before and after the run. Since we do not send any data in the connection prior to this, we do not need to wait on flushing send buffers, and thus get an accurate measurement just for the time of close.

5.4.3 Results

Loopback Connect Latency avg (Cycles, ns)	Loopback Connect Latency stdev (cycles)
216475.2000, 115056.5688	599784.5187, 318785.4717

Table 22: Loopback TCP connection time using a CPU frequency of 1881.6 MHz

Median: 15663.5 cycles, 8325.15025 ns

Remote Connect Latency avg (Cycles, ns)	Remote Connect Latency stdev (cycles)
5534023222324130, 2941333340000000	8453360104086690, 4492960900000000

Table 23: Remote TCP connection time using a CPU frequency of 1881.6 MHz

Median: 510251 cycles, 271198.4065 ns

Loopback Close Latency avg (Cycles, ns)	Loopback Close Latency stdev (cycles)
6106.0000, 3245.339	1277.7245, 679.1105717

Table 24: Remote TCP connection close overhead using a CPU frequency of 1881.6 MHz

Remote Close Latency avg (Cycles, ns)	Remote Close Latency stdev (cycles)
39945.9000, 21231.24585	2244.3587, 1192.876649

Table 25: Loopback TCP connection close overhead using a CPU frequency of 1881.6 MHz

5.4.4 Analysis

From the above results, the first and most obvious note is that the overhead of connecting to a socket is not only extremely high, but extremly variant. We see that while loopback connection overhead is lower than the remote connection overhead, both are influenced by some having some runs with extremely high latencies. From the implementation of the benchmark, we recall that each "run" has a 1000 connect attempts each. Combining that with the TCP 3 way handshake needing 1.5 RTT with SYN, SYN-ACK, and ACK packets being exchanged. However, even multiplying the rtt of our TCP connection as seen above by 1.5 does not explain this latency. For the remote case, we can see that congestion control situations within the router or LAN network can lead to packet drop or congestion, leading to some runs blocking for a long period and seeing high latency. However, this does not explain the same factors being seen on the loopback interface, since that should not be influenced by network hardware congestion. However, we know that through processes like lsof, the kernel keeps global PID to socket mappings, thus, this high variable latency is likely caused by kernel resources management and allocation when connecting and to new sockets via connect.

The results for close have much less variance and latency. From a network protocol. since the only required network communication is the FIN, and FIN-ACK packet, it makes sense that this latency is much lower than the latency for connect, which is reflected above. Since the close call does not suffer from the same outliers as connect above, for both loopback and remote, we are left to conclude that the kernel implementation of close suffers from less overhead than connect.

6 File System

6.1 Size of file cache

6.1.1 Overall Prediction of Performance

Base hardware performance: Linux systems like Ubuntu automatically manages the file cache in accordance to the system load. We run our benchmarks isolated with only a minimal set of other processes. Therefore, we predict the theoretical upper bound for the file cache to be nearly the entire RAM: 16 GB. [11]

For this benchmark, we will memory-map the file so we can use our estimated ≈ 110 ns access time needed for main memory from sectionsec:mem-access-time.

Software overhead: will need a system call to memory-map our file and do reads, so we can add the cost of two system calls which we found to be 74.1663 ns in section 3.4: $2 * 74.1663ns = 148.3326$ ns.

Therefore, the total estimated time to access one page from a file is roughly 258.3326 ns.

6.1.2 Measurement Methodology

For file sizes ranging from 1 MB to 16 GB, we memory-map the file to access the pages directly: `mmap(NULL, len, PROT_READ, MAP_PRIVATE, fd, 0);`.

We disable kernel readahead by advising the kernel we will be accessing the file randomly. This prevents prefetching blocks of the file.

```
posix_fadvise(fd, 0, len, POSIX_FADV_RANDOM);
madvise(data, len, MADV_RANDOM);
```

We use the same measurement boilerplate as in section 3.1.3.

To measure I/O access time for the file, we allocate an array of page indices and shuffle them using Fisher-Yates (see section 4.1.3). This is another prefetching prevention technique. In the inner loop, we go through the number of the pages and access a byte on the page and another byte half way through the page to add some extra work and not have it optimized away. The access is done through a XOR which cannot be optimized by the compiler:

```
for (size_t k = 0; k < n_pages; k++) {
    size_t off = order[k] * (size_t) page_size;
    sink ^= (unsigned char) data[off];
    sink ^= (unsigned char) data[off + page_size / 2];
}
```

At the start of every trial, we advise the kernel to drop the file from cache with `posix_fadvise(fd, 0, len, POSIX_FADV_DONTNEED);`. Before and after the inner loop, we use `getrusage` to meausre how many soft and hard faults we get. This is how we determine that the file has then been loaded into the file cache.

6.1.3 Results

The first trial out of ten causes n hard page faults, where n is the number of pages within the file. We separate out the first run from each size below for this reason, as they have a significantly higher latency. After the hard faults occur, the file is now within the file buffer cache, and since we memory-mapped it, access times become much faster.

See tables 26, 27, and figure 2. Table 28 gives the mean over all file sizes.

We were unable to run our 16 GB file size to completion as the system began thrashing, unable to ever fit into the file cache even after three hours of running while other similarly sized benchmarks at least finished within less than an hour. This supports our estimate of the file cache size being the same size as the RAM, in our case 16 GB.

The large latency on the hard-faulting 1 MB (2^{20} bytes) may be due to the "pseudo-amortization" phenomneon we see in later benchmarks, where smaller files have less pages/blocks to normalize against, so their averages feel more drastic.

# Bytes	Avg. Cycles/page	Latency/page (ns)
2^{20}	228.5352	121.4664
2^{21}	172.5469	91.7087
2^{22}	167.9980	89.2909
2^{23}	173.0654	91.9843
2^{24}	190.0049	100.9876
2^{25}	200.1318	106.3701
2^{26}	201.8721	104.3544
2^{27}	203.8389	108.3404
2^{28}	205.4590	109.2015
2^{29}	205.7637	109.3634
2^{30}	205.9033	109.4376
2^{31}	209.5781	111.3908
2^{32}	210.0889	111.6623
2^{33}	216.2480	114.9358

Table 26: Hard fault time for each file size. This happens in the first trial, so there is no standard deviation to mesaure.

6.2 File read time

6.2.1 Overall Prediction of Performance

Base hardware performance: Given sequential read speeds of 2,366 MB/s, sequential writes of 955 MB/s on the measured disk and reading in 4096B blocks, we can estimate the per-block performance as:

# Bytes	Avg. Cycles/page	StdDev Cycles/page	Latency/page (ns)
2^{20}	5.0841	2.3922	2.7022
2^{21}	6.4453	0.2220	3.4257
2^{22}	6.7066	0.3305	3.5645
2^{23}	11.5735	0.6379	6.1513
2^{24}	22.6301	0.2600	12.0279
2^{25}	34.4351	0.8415	18.3023
2^{26}	39.3171	0.4146	20.8970
2^{27}	40.8753	0.1139	21.7048
2^{28}	41.5825	0.0520	22.1011
2^{29}	41.8167	0.0626	22.2256
2^{30}	42.5048	0.0321	22.5913
2^{31}	42.8878	0.0382	22.7949
2^{32}	43.5095	0.2992	23.1253
2^{33}	49.3040	0.0131	26.2051

Table 27: Non-faulting read time for each file size. These are trials 1-9 (excluding first one).

	Avg. Cycles/page	Latency (ns)/page
Hard faults	198.7167	97.7931
No faults	30.6195	16.2728

Table 28: Results of estimating time for a page access

$$\frac{4096\text{B}}{\text{block}} \times \frac{s}{2366 \times 2^{20}\text{B}} \times \frac{10^9 ns}{1s} = 1650.9932 \text{ ns/blk (read)}$$

$$\frac{4096\text{B}}{\text{block}} \times \frac{10^9 ns}{955 \times 2^{20}\text{B}} = 4090.3141 \text{ ns/blk (write)}$$

For random access, we have an estimated 821 MB/s for random-seek reads:

$$\frac{4096\text{B}}{\text{block}} \times \frac{10^9 ns}{821 \times 2^{20}\text{B}} = 4757.9172 \text{ ns/blk (random seek)}$$

Software overhead: The software overhead would be the cost of trapping into kernel when performing a `read()` system call which we found to be about 74.1663 ns in section 3.4; we add this to the above hardware predictions. As expected, the cost of a system call would be magnitudes smaller than the cost of a disk read.

Therefore, our predictions are:

- Sequential-access read per block: 1725.1595 ns/block
- Random-access read per block: 4832.0835 ns/block

6.2.2 Measurement Methodology

We use the measurement boilerplate outlined in section 3.1.3 but lower the number of loops to 32, as higher number of loops did not yield us substantially different results. Additionally, since we are normalizing our times to per block, the number of blocks in a file provide enough normalization for our results.

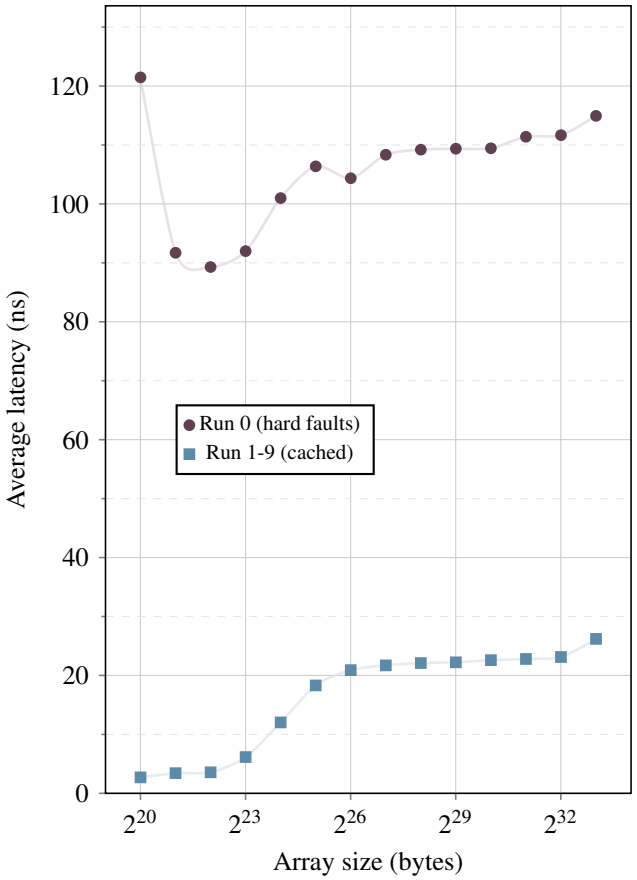

Figure 2: Latency for accessing a page within a file.

For both sequential and random access measurements, we read in 4096 byte blocks, which corresponds to the default in UNIX systems. We measure how long it takes to access and read in every block within a file whose size ranges from 1 MB to 1 GB within each loop of the trial. Notably, since we use powers-of-two for our file, our blocks are aligned within a file and can be easily indexed from the start of a file.

For sequential-access, we do not disable kernel read-ahead and prefetching for blocks, since this makes each read become random as the kernel has to fetch blocks once more. At the beginning of each loop, but before we start timing, the file is re-seeked to the beginning of the files using `lseek()`, and we simply iterate through all the blocks of the file and perform `read()` system calls.

For random-access, we collect all block indices within an array and randomize them using the Fisher-Yates algorithm 4.1.3. During each read-through of a file, we loop through the randomized index array and read a block offsetted by `BLOCK_SIZE * rand_idx` from the start of the file. The `pread()` system call performs an atomic combination of `lseek` and `read` together by taking in the offset directly in its

function signature.

6.2.3 Results

See table 29 for the cycles and latency for sequential-access reads, and table 30 for random, and figure 3 for the latencies plotted on a log-log plot. Table 33 gives a summary of both local and remote read times.

Evidently, randomly accessing blocks within files has a higher latency than sequential access since the file cache cannot predict which blocks to fetch next.

Our estimates were slightly higher than what we actually got, but are within the ballpark so we believe our benchmark methodology was successful and accurate.

We see the read latencies slightly decrease as the file size grows, especially for sequential reads at the lower file sizes below 16 MB (2^{24} bytes). Since smaller files may more easily fit into cache, perhaps even contiguously, blocks from these files may be highly optimal to fetch by virtue of spatial locality.

We also suspect our measurement methodology may cause "pseudo-amortization" as larger files are measured for several thousand more blocks than smaller files, which is evident when seeing the very high standard deviation for 1 MB (2^{20} bytes) versus later sizes.

Additionally, since we do not advise the kernel to drop cached blocks for sequential access to measure true access patterns, the high standard deviation may be attributed to kernel optimizations that vary trial-to-trial.

# Bytes	Avg. Cycles/blk	StdDev Cycles	Latency/blk (ns)
2^{20}	8870.5625	687.4803	4714.7040
2^{21}	6988.5281	36.4263	3714.4027
2^{22}	6303.9500	37.9880	3350.5494
2^{23}	6121.4438	44.6470	3253.5474
2^{24}	5959.7188	21.3574	3167.5905
2^{25}	5959.3094	36.1609	3167.3729
2^{26}	5637.8906	14.5914	2996.5389
2^{27}	5587.4344	21.9012	2969.7214
2^{28}	5612.9937	18.9179	2983.3062
2^{29}	5582.6594	38.9079	2967.1835
2^{30}	5596.4250	39.6650	2974.4999

Table 29: **Sequential-access** read performance

6.3 Remote file read time

We used the `ieng6` department machines to run this experiment, as like the assignment specification mentions, a file access on a particular user's partition mounted on NFS simulates doing a remote file access. The `ieng6` timeshare is a Linux Mint machine with a Intel(R) Xeon(R) Gold 5220

# Bytes	Avg. Cycles/blk	StdDev Cycles	Latency/blk (ns)
2^{20}	165343.2250	231.5868	87879.9241
2^{21}	163254.4594	233.4643	86769.7452
2^{22}	164288.1187	364.5908	87319.1351
2^{23}	164675.2344	78.8387	87524.8871
2^{24}	165132.4469	114.7126	87767.8955
2^{25}	165694.3750	65.4647	88066.5603
2^{26}	165761.6875	36.9684	88102.3369
2^{27}	166340.5063	12.1403	88409.9791
2^{28}	166844.6250	13.9942	88677.9182
2^{29}	167030.5562	15.6218	88776.7459
2^{30}	167464.9500	9.3423	89007.6209

Table 30: **Random-access** read performance

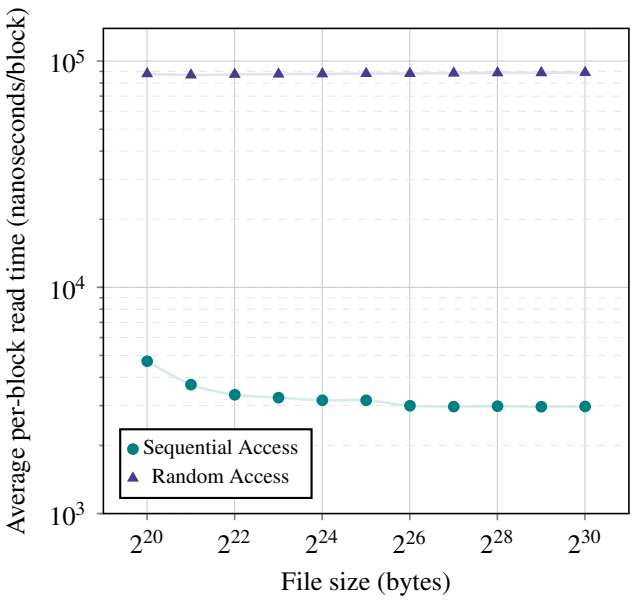

Figure 3: Average per-block read time for sequential and random file access against file size (local)

CPU @ 2.20GHz CPU, and 64 GB of RAM shared between all. Without superuser permissions, we were unable to obtain other substantial information about its hardware.

Assuming the single core we run the benchmarks has 2.20 GHz frequency, this corresponds to a **cycle time of 0.4545 ns** on `ieng6`.

$$\texttt{ieng cycle time} = \frac{1}{2.20 \times 10^9 Hz} = 0.4545 ns$$

6.3.1 Overall Prediction of Performance

To get a rough estimate on the read speeds for the 13 GB of disk that each user has a maximum on, we ran the following

command with a generated 1 GB file from our test setup [4]:

```
time dd if=1G.bin of=/dev/null bs=4k
```

which gives us 225 MB/s, therefore an estimated per block time of:

$$\frac{4096\text{B}}{\text{block}} \times \frac{s}{225 \times 2^{20}\text{B}} \times \frac{10^9 ns}{1s} = 17361.1111 \text{ ns/blk (read)}$$

Without superuser permissions, we cannot install benchmarking tools such as `fio` to obtain disk random seek time. Since in the local benchmark (see section 6.2) the random seek time was about $4832.0835/1725.1595 = 2.8$ times larger than the sequential, we can estimate the random seek to be:

$$17361.1111 \text{ ns/blk (read)} \times 2.88 = 48611.1111 \text{ ns/blk (read)}$$

Without knowing the network topology of the `ieng6` servers, we arbitrarily guess they could impose a 25% penalty on these times.

Therefore, with the same 74.1663 ns added for the software overhead as in the local benchmark, we arrive at:

- Sequential-access read per block: 21775.5552 ns/block
- Random-access read per block: 60838.0552 ns/block

6.3.2 Measurement Methodology

No changes are made to the local file read time benchmark highlighted in 6.2 in regards to code.

To note, since this benchmark was performed over a school-shared timeshare, these findings are influenced by system load. These measurements were conducted on `ieng-203` when there were about 50 other users logged in at the time.

6.3.3 Results

See table 31 for sequential access, table 32 for random access, and figure 4 for these plotted. Table 33 gives a summary of both local and remote read times. Note that the latencies calculated are based off an assumed 0.4545 ns on the `ieng` server.

Surprisingly, our estimate for the sequential access was greatly overestimated, and the latency for sequential was only a smaller factor of 7-9 times slower than on local. Meanwhile, our random access estimates were about 2 times faster than the actual results. Comparing local random versus remote random, there is a factor of about 10.

Therefore, when comparing the local times with the remote times, we see a **network penalty of roughly 8-10 times the local disk read times**.

Since we used a timeshare environment where our process does not have full priority, along with the added load of other users, our benchmark hit a significant bottleneck when measuring. We ran the experiment several times to validate these numbers as being mostly consistent within their range.

# Bytes	Avg. Cycles/blk	StdDev Cycles	Latency/blk (ns)
2^{20}	36568.6281	9115.9669	16620.44147
2^{21}	35746.0000	4703.6274	16246.5570
2^{22}	30128.0563	2328.3412	13693.2016
2^{23}	27347.4188	1532.7754	12429.4018
2^{24}	27893.6063	1045.0982	12677.6441
2^{25}	26653.1000	758.8176	12113.8340
2^{26}	25713.6469	863.4528	11686.8525
2^{27}	24935.6781	481.1890	11333.2657
2^{28}	24594.4062	899.2932	11178.1576
2^{29}	25413.2125	585.6443	11550.3051
2^{30}	25092.0438	542.0669	11404.3339

Table 31: **Sequential-access** read performance, remote

# Bytes	Avg. Cycles/blk	StdDev Cycles	Latency/blk (ns)
2^{20}	979205.3344	150355.6800	445048.8245
2^{21}	934031.8750	41371.8548	424517.4872
2^{22}	950828.9313	16152.3494	432151.7493
2^{23}	931623.2656	20223.0482	423422.7742
2^{24}	972046.9062	51313.3624	441795.3189
2^{25}	968680.4250	27633.0571	440265.2532
2^{26}	1107872.8281	216246.9904	503528.2004
2^{27}	949922.0312	91831.6754	431739.5632
2^{28}	852735.4000	50170.8104	387568.2393
2^{29}	902179.0062	57262.0126	410040.3829
2^{30}	856967.4000	17719.9629	389491.6833

Table 32: **Random-access** read performance, remote

The follwing is a summary of both local and remote, by taking the mean of the averages and latencies.

	Avg. Cycles/blk	Latency/blk (ns)
Sequential (local)	6201.9014	3296.3106
Random (local)	165620.9259	88027.5226
Sequential (remote)	28189.6179	12812.1814
Random (remote)	946008.4912	429960.8615

Table 33: Mean of all averaged cycles and latencies across all file sizes per-block

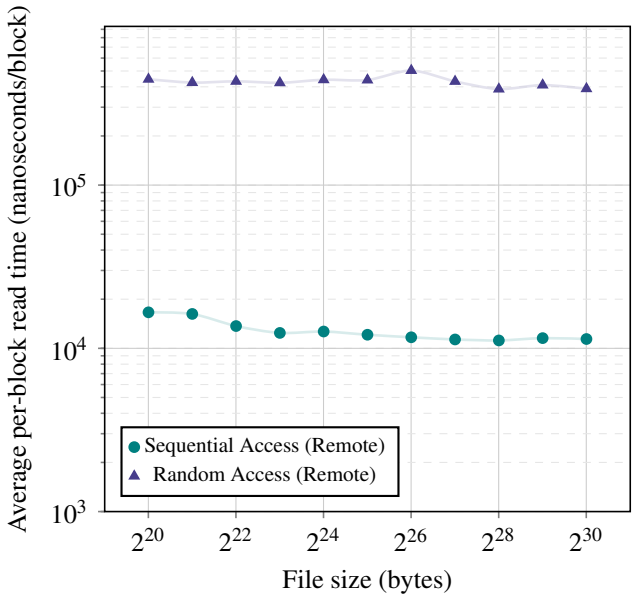

Figure 4: Average per-block read time for sequential and random file access against file size (remote)

6.4 Contention

6.4.1 Overall Prediction of Performance

Base hardware performance: can ultize the same estimates from our file read time benchmark 6.2 to get the estimations of per-block performance as:

- Sequential-access read per block: 1725.1595 ns/block

To estimate the contention for reads, we can multiply our random-access time by the expected additional reads. For each additional process, we estimate a linearly scaled amount of additional reads for disk to handle in a serialzed way. For a given P processes we then estimate the access time per block would be $P *$ Est. sequential-access read per block or $P * 1725.1595$. We predict that this estimate should be an upper bound for the reads.

Software overhead: The software overhead here would be the read syscall which can be estimated by 3.4 at 36.0715ns.

Our final estimate of the contention overhead as a function of number of processes P is: P * Est. sequential-access read per block*or*P * 1761.2665.

6.4.2 Measurement Methodology

To get the average time to read one file system block under contention, we systematically ran through a process to create N different files, N processes, and for each process, read one unique file block. In our benchmark, we set N as 50 and we would loop through i iterations from 1 to N creating i files to test under i processes for contention. For each file, we write into the file a 1000 blocks of data, our machine's block size is 4096KB. We then reset the file pointer with `lseek`, clear the cache with `posix_fadvise`, and close the file descriptors. We then reopen them with flag, `O_DIRECT` to force reading from disk and record all the various file descriptors in an array.

We then create N processes to conduct the reads. We use a pipe to make sure all processes are created and ready to execute prior to starting any of the reads. Once, the processes are ready, we start timing utilizing the timestamp register method (see 3.1). Within the timing instructions, we loop through and read though 1000 blocks of data within the file using `read`. Our resulting measurement is the average read time for one block. To keep track of average cycles per process, we map a region of shared memory with `mmap(NULL, count * sizeof(uint64_t), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0)` where each process has dedicate entry to add write their average read cycles for one block.

After every run, we sum up the average read cycles for a block for every process to get the average reads across all files. Also we clear the shared memory to keep our runs consistent.We repeat this experiment 10 times per file count.

6.4.3 Results

Our resulting latencies for block read with a fixed process count from 1-50 are listed on table 34. We graph our resulting block latency under a fixed number of processes reading simultaneously in figure 5.

Our results for a single file and a single process fall into the range of between a sequential singular block access and a sequential block access 4MB. Our hypothesis of why they are not a closer match in cycles even for one file, is that we're running sequential reads while using `O_DIRECT` to avoid using the file cache. Due to sequential reads, there might be some prefetching done on the disk which can explain some additional speed up. This latency is also within the range of our estimated sequential-access read per block.

For contention, we found that the cycles started to exponentially grow from 1 process to 5 processes. After that, the cycles started growing linearly instead with every additional process. Our hypothesis here is that for lower process counts, the disk cache is able to optimize reads but as quickly fills up and starts reaching it's maximum performance. That's where we're starting to see linear growth where the workload is still increasing, more file reads, but the disk cache speedup is now amortized. We also noticed that the standard deviation dropped from steeply from 5 and beyond which can be a result of reaching a steady state as the cache optimizations are amortized.

Thus from the varied growths in our graph and to better represent its values, we have one function to represent count of contending processes from 1-5 and another from 6 - 50. For each function, we based it off of the min and max latencies in

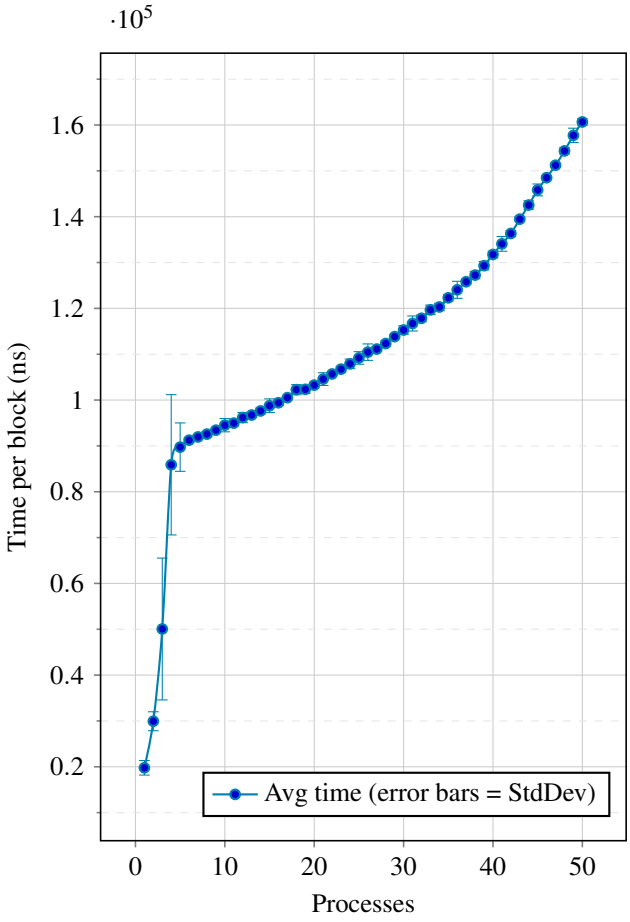

Figure 5: File system contention; average time per block vs number of processes

each respective range and derived a line of best fit.

For calculating estimated block read time for contending processes 1-5:

$block\ access\ ns = 17488.75 * P + 2289.25$, where P is the count of contending processes.

For calculating estimated block read time for contending processes 6-50:

$block\ access\ ns = 1577.59 * P + 81785.43$, where P is the count of contending processes.

Our estimates were pretty close to the growth of latency under contending processes. The growth was estimated to be $1725.1595 * P$ and the actual results were about 8% faster at $1577 * P$ on the higher end of processes. This could be due to slight variation in how our benchmark tested reads versus the estimate. In our benchmark, we used `O_DIRECT` to force disk reads, which was still susceptible to optimization on the disk end. The online benchmark may had other settings that removed this influence.

Overall, we found that our contention benchmark scales monotonically with additional workload which we expected. There is still some unknown causes to deviation within the lower process counts. The causes of the higher deviation on the lower processes could be caused by variability of how the disk optimizes retrievals. A further experiment could be testing on various file sizes to see if the deviations change.

7 Summary

A summary of our results can be found in table 35.

Measurement overhead is the summation of timing overhead (section 3.1) and looping overhead (section 3.2).

Many of the CPU measurements were predicted using CPI, which we categorize as a base hardware performance. When we scaled according to a paper, we assume that the hardware and software overhead are both captured when we scale our CPU to their final results.

# Processes	Avg. Cycles/block	StdDev Cycles/block	Latency/block (ns)
1	37212.7000	1585.4160	19778.5501
2	56326.3500	2066.7258	29937.4550
3	94183.0667	15473.9480	50058.3000
4	161580.8000	15300.1794	85880.1952
5	168831.4600	5278.8860	89733.9210
6	171685.9833	493.3227	91251.1001
7	173007.3857	431.6586	91953.4255
8	174165.7125	572.7438	92569.0762
9	175784.9556	417.9996	93429.7039
10	177830.6700	1445.2776	94517.0011
11	178677.3091	622.3632	94966.9898
12	180957.9750	1110.7490	96179.1637
13	181994.0846	513.3220	96729.8560
14	183679.3786	612.0981	97625.5897
15	185832.9800	1488.5600	98770.2289
16	187038.9250	608.6152	99411.1886
17	189129.5000	333.7642	100522.3293
18	192404.1389	1100.6954	102262.7998
19	192550.0263	878.4807	102340.3390
20	194278.0700	611.2966	103258.7942
21	196747.7667	1385.5861	104571.4380
22	198850.4227	283.4275	105688.9997
23	200818.4783	626.9963	106735.0212
24	203011.3000	1048.5089	107900.5060
25	205429.5600	1368.8614	109185.8111
26	207816.2000	1795.9981	110454.3103
27	209067.8444	485.4251	111119.5593
28	211346.5393	584.5340	112330.6856
29	214183.1759	409.0074	113838.3580
30	216891.1967	926.3467	115277.6710
31	219565.6774	1644.6265	116699.1575
32	221699.7344	724.4048	117833.4088
33	225162.0545	1037.3765	119673.6320
34	226347.0412	799.7277	120303.4524
35	230109.6114	694.2451	122303.2585
36	233399.8111	1855.1927	124051.9996
37	236697.8568	496.5905	125804.9109
38	239440.1289	789.5356	127262.4285
39	243298.8821	906.1411	129313.3558
40	247895.9700	744.8524	131756.7081
41	252250.7268	1609.5251	134071.2613
42	256514.8929	652.3576	136337.6656
43	262402.8860	469.0068	139467.1339
44	268188.4432	923.0616	142542.1576
45	274396.9133	1294.4712	145841.9594
46	279403.5739	441.1923	148502.9995
47	284520.5745	464.7928	151222.6853
48	290406.6271	694.6219	154351.1223
49	296773.9755	1566.5251	157735.3680
50	302286.3120	690.6586	160665.1748

Table 34: File System: Contention: average cycles and latency per file block access with a fixed number processes reading at once

Table 35: Summary of our benchmarks.

Operation	Base Hardware Performance (ns)	Estimated Software Overhead (ns)	Predicted Time (ns)	Measured Time (ns)
CPU - Measurement overhead	13.3691	N/A	13.3691	8.685
CPU - Procedure call overhead	0.9902 (per added arg.)	N/A	0.9902	$\approx$ 0.6 (per added arg.)
CPU - System call overhead	N/A	53.6646	53.6646	36.0715
CPU - Task creation time (kernel threads)	8695.6522	N/A	8695.6522	4117.7254
CPU - Task creation time (user processes)	17391.3043	N/A	17391.3043	20817.8185
CPU - Context switch time (kernel threads)	782.6	N/A	782.6	868.5359
CPU - Context switch time (user processes)	391.3	N/A	391.3	20817.8185
Memory - Access time (main memory)	35	1.2925	36.2925	55 <
Memory - Bandwidth (reads)	68 GB/s	N/A	68 GB/s	21.0607 GB/s, 0.0442 ns/byte
Memory - Bandwidth (writes)	50 GB/s	N/A	50 GB/s	15.3572 GB/s, 0.6064 ns/byte
Memory - Page fault service time	78.7623 μs	N/A	78.7623 μs	44.1327 μs
Network - Round trip time	N/A	N/A	93000 ICMP local 162000 TCP local	30000 local icmp 3248000 remote icmp 12204.93766 loopback TCP 2129247.206 remote TCP
Network - Peak bandwidth	1.0 Gb/s	N/A	N/a	60673.97483 Mb/s Loopback Read 53721.62741 Mb/s Loopback Write Remote Read 734.6593646 Mb/s Remote Write 655.8125247 Mb/s
Network - Connection overhead	N/a	N/a	100000 Loopback Connect	Loopback Connect 115056.5688 ns Remote Connect 2941333340000000 ns Loopback Close 3245.339 ns Remote Close 21231.24585 ns
File system - Size of file cache	110	148.3326	258.3326/page	97.7931/page on hard fault, estimated 16 GB file cache
File system - File read time (sequential)	1650.9932	36.0715	1687.0647	3296.3106/blk

Table 35: Summary of our benchmarks.

Operation	Base Hardware Performance (ns)	Estimated Software Overhead (ns)	Predicted Time (ns)	Measured Time (ns)
File system - File read time (random)	4757.9172	74.1663	4793.9887	88027.5226/blk
File system - Remote file read time (sequential)	17361.1111 $\times$ 1.25	74.1663	21775.5552	12812.1814/blk
File system - Remote file read time (random)	48611.1111 $\times$ 1.25	36.0715	60838.0552	429960.8615/blk
File system - Contention	1650.9932 - 4757.9172 (block access)	36.0715	$P * 1761.2665$	$\begin{cases} 17488P + 2289, & 1 \leq P \leq 5, \\ 1577P + 81785, & 6 \leq P \leq 50. \end{cases}$

References

[1] Rfc 793 — transmission control protocol, Sept 1981.

[2] Limitations of icmp echo for network measurement, 2025.

[3] CHEN, J. B., ENDO, Y., CHAN, K., MAZIERES, D., DIAS, A., SELTZER, M., AND SMITH, M. D. The measured performance of personal computer operating systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, Association for Computing Machinery, p. 299–313.

[4] GITE, V. Linux and unix test disk i/o performance with dd command, Aug 2015.

[5] HARDDRIVEBENCHMARK.NET. Samsung mzvlq1t0hblb-00bh1 benchmarks. https://www.harddrivebenchmark.net/hdd.php?hdd=SAMSUNG%20MZVLQ1T0HBLB-00BH1&id=29801, 2025. Accessed: 2025-11-14.

[6] INTEL. Theoretical maximum memory bandwidth for intel® coretm x-series, 2021.

[7] MCVOY, L., AND STAELIN, C. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference (USENIX ATC 96)* (San Diego, CA, Jan. 1996), USENIX Association.

[8] OUSTERHOUT, J. Why aren't operating systems getting faster as fast as hardware?

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

[10] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Organization and Design The Hardware/Software Interface - MIPS Edition*. Morgan Kaufmann, 2021.

[11] PROJECT, T. L. D. The buffer cache.

[12] PURE STORAGE, INC. What is nvme? the complete guide to non-volatile memory express. https://www.purestorage.com/knowledge/what-is-nvme.html, 2025. Accessed: 2025-11-14.

[13] WIKIPEDIA CONTRIBUTORS. Fisher–yates shuffle — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Fisher%E2%80%93Yates_shuffle&oldid=1321879330, 2025.