Operating Systems Measurements for Linux on x64

1 Introduction

The aim of this project is to build utilities for measuring a set of defined benchmarks related to the following common functionalities provided by modern operating systems:

- Execution time measurement
- Process and thread scheduling, and context switching
- Main memory access time
- Networking round-trip time and bandwidths
- File system related benchmarks

We perform all measurements using C++ code, compiled using version `v13.2.0` of the **g++** compiler. We avoid making use of any optimization provided by the GCC compiler, so as to not impact the functionality or the expected performance of our code, as it can affect the correctness of our measurements.

2 Machine Description

We now present the description of our target machine. The machine is a laptop computer that runs Ubuntu 24.04 LTS on a 10th generation Intel Core i7 CPU which implements the x86_64 instruction set architecture. We make use of the following Linux utilities in order to gather this information:

- `lscpu` to get CPU Model information
- `sysfs` virtual file system (mounted at `/sys/devices/` to get individual cache sizes
- `dmidecode` to get information about main memory size and clock speeds
- `sysfs-block` mounted at `/sys/class/block` to get information about SSD model
- `fdisk` to get information about total disk size
- `lshw` to get network interface information
- `uname` to get Linux kernel version

2.1 CPU Description

Processor Model	Intel Core i7-10750H
L1 Instr. Cache Size	32 KiB
L1 Data Cache Size	32 KiB
L2 Shared Cache Size	256 KiB
L3 Cache Size	12288 KiB (12 MiB)
Base Clock Frequency	2.592 GHz
Cycle Time	0.3858 ns

Table 1: CPU Description

2.2 DRAM Description

DRAM Model	SK Hynix HMA81GS6DJR8N-XN
DRAM Type	DDR4 Synchronous
DRAM Capacity	16 GB
DRAM Clock Frequency	2933 MT/s = 1466.5 MHz
Memory Bus Bandwidth	64 bits

Table 2: DRAM Specifications

2.3 Disk Specifications

Type	Solid State Drive
Model	SAMSUNG MZVLB512HBJQ-000H1
Total Capacity	512110190592 bytes, 476.94 GiB
Sequential Read speed	3500 MB/s
Sequential Write speed	2900 MB/s
Random Read speed	460000 IOPS
Random Write speed	500000 IOPS

Table 3: Disk Specifications

2.4 Network Interface Specification

Network Card	Intel Comet Lake PCH CNVi WiFi
Network Interface Type	WiFi-6
Configured MTU	1500 bytes
Bandwidth	64 bits

Table 4: Network Interface Description

2.5 Operating System Description

OS Name	Linux
Kernel Version	6.8.0-47-generic
Distribution	Ubuntu 24.04.1 LTS x86_64

Table 5: Operating System Description

3 CPU, Scheduling, and OS Services

We take the following precautions to make our measurements as accurate as possible before taking the measurements for the below quantities:

- Ensure that our CPU supports constant timestamp counting. This is done by checking the `/proc/cpuinfo` and ensuring that the `constant_tsc` flag is set for all the CPU cores. In our case, being a relatively modern generation Intel CPU, this flag is supported.
- Turn off Turbo Boost ™ which is Intel's implementation of dynamic frequency scaling as per workload. We do this to ensure that our measurements are slightly more consistent across runs. This is done by writing the value of "1" to the file **/sys/devices/system/cpu/intel_pstate/no_turbo**.
- Restrict execution to a single core. This is done using the `taskset` command [14] that is used to set the CPU affinity of a process to a specific core. Based on our empirical observations, we set the processes to run on core 1 as it has a stable clock frequency.
- Ensure that all benchmarks run at highest priority. We use the C standard library's `nice` function to set the NICE value of the current running process to -20, indicating highest possible priority. In doing so we attempt to mitigate any potential corruption to our benchmarks caused by scheduling and interrupts.

Note that we do not guess any base hardware performance for the below operations, as we measure the instruction latency directly using our testing scripts which gives us a sufficient indication of what operations the CPU hardware is executing.

3.1 Timing overhead

We use RDTSC and RTSCP along with barrier instructions to prevent instruction re-ordering. The barrier instruction used is CPUID. CPUID can be executed at any privilege level to serialize instruction execution. This guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed by the CPU. [12]. This measurement is repeated for an ensemble size of 10^4, over an outer loop running 10^4 times, which leads us to a total measurement of 10^8 i.e. 100 million iterations. We calculate the standard deviation of the sample mean (i.e. the mean of each set of inner loop iterations).

Based on the data available in [11] (Table D-17), we see that the RDTSC and RDTSCP instructions have a throughput of 13 and 20 cycles respectively for the Skylake micro-architecture that the Intel i7-10750H CPU uses. This implies that, according to Intel's definition of throughput in [11], consecutive RDTSC and RDTSCP would occupy the instruction issue ports for 33 cycles (as we are restricting the execution to only a single core). This leads us to our guess of 33 clock cycles as we also find that both `RDTSC` and `RDTSCP` use the same instruction issue ports in the Skylake micro-architecture, i.e. ports 0, 1, 5, and 6 ([4]). Note that chips running the Skylake micro-architecture have 8 instruction issue ports.

This leads is to our **initial guess** of 33 clock cycles, i.e. around **12.7314ns**. Our final results yield a similar figure as shown in table 6.

Predicted Latency (cycles)	33
Predicted Latency (time)	12.731 ns
Actual Latency (cycles)	34.379
Actual Latency (time)	13.263 ns
Measured Std. Deviation (cycles)	1

Table 6: Timing Overhead

3.2 Loop overhead

The `objdump` utility in Linux allows us to disassemble a compiled executable and view it as a series of x86_64 assembly instructions (since that is the hardware platform we have chosen to benchmark). Listing 1 shows the disassembled code for the loop overhead measurement. We see 4 `MOV` instructions and then 5 more instructions that execute the actual empty loop.

We notice that compared to the simple timing loop that was covered in Section 3.1, we have 5 additional instructions that can all be executed in 1 cycle each (assuming that the data is present in L1 cache, if not then the estimate grows to approximately 8 cycles, accounting for a 4 cycle L1 miss penalty as described in (Table D-18, [11]), which explains the variance of the measurement over a large sample) . Hence we

Listing 1: Loop Overhead Code

```
rdtsc
mov    %edx,%edi
mov    %eax,%esi
mov    %edi,-0x78(%rbp)
mov    %esi,-0x74(%rbp)
movl   $0x0,-0x8c(%rbp)
jmp    26c3 <main+0xea>
addl   $0x1,-0x8c(%rbp)
cmpl   $0xf423f,-0x8c(%rbp)
jle    26bc <main+0xe3>
rdtscp
```

estimate an overhead of **5 cycles** or **1.924ns**. The results of our measurements are presented in Table 7.

Estimated Loop Overhead (cycles)	5
Estimated Loop Overhead (time)	1.924 ns
Measured Avg. Loop Overhead (cycles)	5
Measured Std. Deviation (cycles)	0.1
Measured Avg. Loop Overhead (time)	1.943 ns

Table 7: Loop Overhead Measurement

3.3 Procedure call overhead

We present the cost of a procedure call as a function of the number of arguments provided to the function with N, number of arguments, ranging from 0-7. We estimate that the function call overhead increases linearly as a function of number of arguments. This conclusion can be drawn by looking at the assembly code generated:

The noticeable difference is that as the number of parameters increase, one additional `MOV` instruction is added to copy the parameter from the function stack (represented by the address of the base pointer `%rbp`) to the registers. Since modern Intel mobile chips like the i7-10750H being tested normally contain multiple data pipelines per core, we assume each move operation to take around half a cycle, as reported in [8]. Also assuming the timing overhead of 33 cycles from Section 3.1 and a 4 cycle overhead for the call instruction using a memory address (again reported in [8]), we get a 37 cycle constant overhead for all functions which yields a rough equation for number of cycles taken:

$$y = 0.5x + 37 \tag{1}$$

Figure 1 shows the linear trend as well as the errors in the estimated values. We can explain the lower values compared to our estimates through various optimizations like instruction re-ordering that Intel CPUs undertake in order to improve memory pipeline performance when executing MOV instructions.

Listing 2: Procedure Call Code

```
<_Z10procedure0>:
    endbr64
    push   %rbp
    mov    %rsp,%rbp
    nop
    pop    %rbp
    ret
<_Z10procedure1i>:
    endbr64
    push   %rbp
    mov    %rsp,%rbp
    mov    %edi,-0x4(%rbp)
    nop
    pop    %rbp
    ret

<_Z10procedure2ii>:
    endbr64
    push   %rbp
    mov    %rsp,%rbp
    mov    %edi,-0x4(%rbp)
    mov    %esi,-0x8(%rbp)
    nop
    pop    %rbp
    ret
```

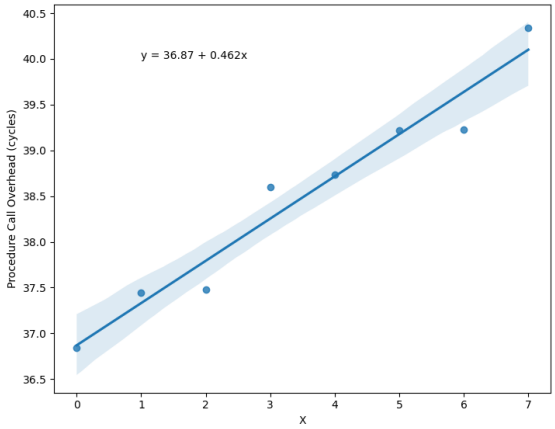

Figure 1: Procedure Call Overhead as a function of number of args N

N	Estimated Time (ns)	Observed Time (ns)	Std. Dev.
0	14.275	14.214	1.032
1	14.467	14.445	1.24
2	14.66	14.457	0.587
3	14.853	14.892	0.624
4	15.046	14.942	0.683
5	15.24	15.130	1.149
6	15.432	15.135	0.442
7	15.624	15.560	0.526

Table 8: Procedure Call Overhead

3.4 System call overhead

We use the `getpid` system call to measure the system call overhead. The LMBench paper [15] provides some arguments against using getpid, including that it may be implemented as a user-level routine instead of a kernel level one. However, versions of `glibc` above 2.25 (released in Feb 2017) do not implement any sorts of optimizations to getpid, treating as a normal system call [1].

We make an initial estimate of the system call time being roughly 10 times of the total time needed for a user-space procedure call. Since the getpid system call has no arguments, we use the measured time needed for a procedure call with 0 arguments from 3.3 and derive our **estimate** of 142.14 ns.

Estimated syscall overhead	142.14 ns
Observed syscall overhead	166.383 ns
Observed Std. Dev. (cycles)	8

Table 9: System Call Overhead for getpid system call

Some modern estimates [18] indicate a latency of around 500 clock cycles or around 192.9 ns, on some older generation Intel Skylake processors that share the same microarchitecture as our test machine.

3.5 Task creation time

3.5.1 Thread Creation Time

We use the popular `pthread` library to study the creation time for a kernel-managed thread. The thread is launched with a very lightweight function that instantly exits, , merigng the launched thread with its parent.

While no recent benchmarks exist for this operation on our specific hardware-software combination, we analyze some older benchmarks to give us an estimate. [9] estimates the thread creation time to be between 254 μs and 312 μs on a 200 MHz Pentium processor from 2000. We can use this as a rough estimate, and estimate our thread creation time to be around 1/13 of this number, because our CPU frequency is around 13 times larger, giving us an initial estimate of 19.5 μs. We measure the overhead of 1 million thread creation operations, and the standard deviation of 1000 averages of these operations.

Estimated thread creation time	19.5 μs
Observed thread create time	9.769 μs
Observed thread create time (cycles)	25322
Observed Std. Dev.	568

Table 10: Overhead for kernel thread creation

Our estimate does not take into account any optimizations of thread creation, as well as the fact that the benchmark was run on a machine with significantly higher memory capacity and bandwidth.

3.5.2 Process Creation Time

We use the `fork()` system call to create a child process. The child process created does not do any work and simply exits, while the parent process measures the end time for the fork and continues with the remaining computations. Forking is an inherently expensive operation as it involves making a copy of the parent's entire address space and state in memory with the default options supplied.

We expect that the overhead of process creation using `fork()` would be an expensive operation. Our initial estimate is 30 times of the thread creation time, which evaluates to 293.07 μs or approximately 0.293 ms based on the results of Section 3.5.1.

Estimated creation time	0.293 ms
Observed creation time	0.350 ms
Observed creation time (cycles)	908127
Calculated Std. Dev.	4335

Table 11: Overhead for process creation using fork

Modern benchmarks suggest that the cost of a single fork call would enter the millisecond range (i.e. $> 1ms$) for even modestly sized processes using around 170MB of memory [20]. Leaner processes that use no additional memory would be faster to create as the overhead of copying a smaller address space is reduced.

3.6 Context switch time

3.6.1 Process Context Switch Time

The `fork()` call is used to create a child process. We also create a pipe [2] that the parent and child process use to communicate.

The parent process calls fork and then starts the timer using the rdtsc assembly instruction. The timer is stopped by the

child process, which then writes the value of the timestamp counter to the pipe, and exits cleanly. The parent then reads the pipe to get the timestamp counter value, and calculates the difference between the start and end times. This operation is repeated 10^3 times, and a group of 10^3 such operations is repeated 100 times for a total of 10^5 data points. This methodology is partially inspired by the work done in [16] to measure context switch overhead.

The work done in [16] is also useful in giving us some ideas on potential initial guesses, in that it clearly shows the worse performance of CISC machines at context switching compared to RISC machines, though not by the margin expected. Works like [5] and [13] give us an idea that the operation requires atleast microsecond latencies. We make a rough guess of around 30-40000 cycles which corresponds to around 15.432 μs on our machine.

Estimated context switch time	15.432 μs
Observed context switch time	22.614 μs
Observed context switch time (cycles)	58615.99
Calculated Std. Dev.	231.1901

Table 12: Overhead for process context switch using fork

3.6.2 Thread Context Switch Time

We use a similar methodology to the above to measure the time needed to switch threads. The difference stems from the use of kernel-managed user threads (implemented as `pthreads` in the case of Linux) instead of processes. We expect a reduction of at least one order of magnitude between thread and process context switching mainly because thread switching avoids the requirement for copying out and copying in large segments of memory (as threads share memory), and swapping register values like the stack pointer etc. is a much cheaper operation.

Our rough initial guess is around 10000 cycles or 3.858 μ s based on an intuitive guess where we estimate thread context switching to be 25% as costly as process context switching.

Estimated creation time	3.858μs
Observed creation time	4.0003755 μs
Observed creation time (cycles)	10369.03958
Calculated Std. Dev.	74.33319

Table 13: Overhead for thread context switch

4 Memory Operations

4.1 RAM access time

The intent of this benchmark is two-fold: to measure the average latency involved in a single access to the main memory system (including the processor caches), while also trying to reveal some insights into the cache organization and structure of the target machine. LMBench [15] defines the **back-to-back latency** of memory access as the time that each load takes, assuming that the instructions before and after are also cache-missing loads. LMBench also cites this particular measurement as being easiest to measure in software, and close to an accurate idea of what software developers consider to be the real memory latency.

The RAM access time benchmark is carried out by allocating fixed-size regions of various sizes as C++ arrays, and stepping through them using a single integer pointer variable. The timing measurement is carried out using `RDTSC` and `RDTSCP` instructions, to measure the time needed to make 1 million (10^6) memory accesses on the array, looping over the length in a circular fashion if the array length is less than 10^6. This approach is inspired by the methodology of LMBench.

First, we set up the array and the pointers (indices) for each stride size. This is achieved by the code snippet in Listing 3.

```
for(int i = 0; i < array_size; ++i)
{
    arr[i] = (i + stride_length) % array_size;
}
```

Listing 3: Setting up array indices to force cache misses

Once the pointers are set up, we ensure that the memory accesses are set up in a manner that forces a cache miss before load, in a manner consistent with our definition of back-to-back memory latency. This is achieved by the snippet in Listing 4 which is inspired by LMBench.

We vary the total **size of the array** between 4KiB and 512 MiB, increasing by powers of 2. For each possible size, we vary the stride length between 4 KiB and 16 MiB, and for each combination of stride and array lengths, we collect 1 million measurements, reporting their final average in Figure 2.

An initial estimate of the cache access latency for each cache level is given by [11] (Table 2-16, p. 2-36). Table 14 shows the **hardware baseline** estimate for cache access latency based on the values given in the Intel Software Optimization guide, as against the experimental results obtained with a stride length of 128 KiB.

We anticipate a very **small software overhead** of 2 cycles per measurement, for the pointer arithmetic operations to calculate the memory address accessed, and a 5 cycle loop overhead. This gives us an **estimated software overhead** of around **7 cycles or 2.7 ns** which is added to each measurement in table 14.

Figure 2, while not showing the memory hierarchy as clearly as Figure 1 in the LMBench paper, still illustrates the memory hierarchy of the system fairly well. The lack of clear boundaries can be put down to complex pipelining

```c
int x = 0;
/*
 Placeholder for RDTSC and
 x86 instr. that place barriers
 for accurate timekeeping
*/
rdtsc();
for (int i = 0; i < NUM_MEASUREMENTS; i++)
{
        x = A[x];
}
/*
 Placeholder for RDTSCP and
 x86 instr. that place barriers
 for accurate timekeeping
*/
rdtscp();
/*  Measure time between rdtsc
and rdtscp calls
by reading register vals
*/
```

Listing 4: Memory accesses and benchmarking code

of memory accesses, as well as optimizations regarding the elimination of L2 to L3 writes when a zero value is being written, that are implemented in recent Intel processors [6].

We notice three significant points of transition in the graph:

- At $\mathbf{X = 5}$, we see a slight upward trend in the graph for all the stride sizes larger than 4 Bytes and smaller than 4 MiB. When the x-axis value is 5, we have an array size of 32 KiB. This increase in latency indicates a transition from the use of the L1 cache to L2 cache, as the array size finally exceeds the limit of the L1 cache size of 32 KiB (refer 1).

- At $\mathbf{X = 8}$, we see an upward trend in the graph for all stride sizes between 64 bytes and 1 KiB. When the x-axis value is 8, we have an array size of 256 KiB. The increase in latency indicates a transition from the L2 cache to the L3 cache, as the array size exceeds the size of the L2 cache (256 KiB, refer 1)

- Between $\mathbf{X = 13}$ and $\mathbf{X = 14}$, we see another sharp upward trend in the memory latency. This indicates a transition from the L3 cache to the use of the main memory, as the array size increases beyond the size of the shared L3 cache (At x = 14, we have array size 2^{14} KiB = 16 MiB. In reality, this transition would occur somewhere between X = 13 and X = 14, as the L3 cache size is 12 MiB, which is halfway between X = 13 and X = 14 on our log-plot).

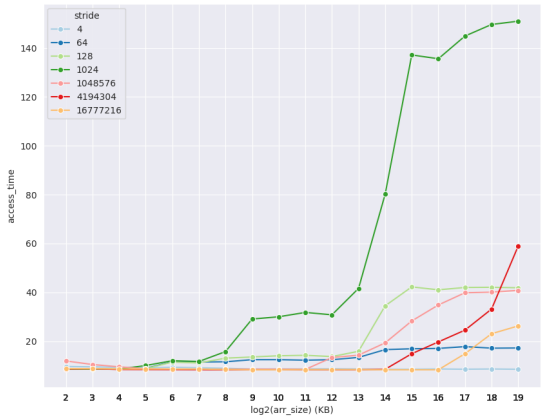

Figure 2: RAM Access Time as a function of Stride Length and Array Size

The choice of stride size also affects the benchmarked values. We see that for smaller stride sizes like 4 bytes, the access time remains almost constant as the array size increases. This is because smaller strides indicate that most of the memory accesses can be accommodated in the faster L1 and L2 caches. For example, a 4 byte stride size will fault from L1 cache only once every 16 array accesses (as each cache line can store 64 bytes, or 16 integers).

When the stride size grows to 64 or 128 bytes, i.e. sizes larger than a single cache line but smaller than the L1 cache, this results in accesses either being on different cache lines (for larger arrays), or within the same cache line (for smaller arrays, where the loop back based on modulo would essentially default to consecutive element accesses).

As the stride size grows beyond the size of each level of cache, we see the effect of the access times far more clearly than for smaller stride values. This difference is put down to the larger stride sizes actually forcing cache misses, more frequently than the smaller stride sizes (since there will be more misses for a 64 byte cache line when stride size increases beyond this limit).

Cache Level	**Predicted Latency (cycles)**	**Measured Avg. latency (cycles)**	**Measured Std. Deviation (cycles)**
L1-instr.	11	8.636	3.331
L2	19	13.389	5.166
L3	51	40.629	15.675
Main memory	138	143.703	7.037

Table 14: Memory Hierarchy Latency Measurements

4.2 RAM bandwidth

The primary concern while implementing a good method for RAM Bandwidth benchmarking is to avoid hitting the cache as much as possible. The basic implementation is built around accessing a 4 MB array of integers in memory, using simple read operation or write operation. The following must be taken into consideration while implementing such a benchmark:

- We first ensure that the caches are occupied and none of the data we want to access is in the cache. This is achieved by issuing a `memcpy` on a 32 MB global array (which is significantly larger than the last-level cache size of 12 MB).
- We hand-unroll the loop inside which we perform the read or write tests as much as possible to have a very minimal loop overhead, and avoid the use of caches and the temporal locality that is inherent in the looping operation (since the loop increments accesses a frequently accessed region, it is likely to be cached which could pollute our results)

Given the information in Table 2, it is easy to construct a basic guess of the RAM Bandwidth using its clock speed:

$$\begin{aligned} \text{RAM Clock Frequency, } f &= 1466.5 \text{ MHz} \\ \text{Memory Bus Width} &= 16 \text{ bytes} \\ \text{Bandwidth} &= \textit{bus\_width} \times f \\ &= 1466.5 \times 10^6 \times 16 \text{ bytes/s} \\ &= 23.464 \text{ GiB/s} \end{aligned}$$

Note that this is for a single memory channel. However, the target machine uses *dual channel memory*, in that it has two physical sticks of RAM, each of which has its own channel to the CPU. Hence, the total bandwidth that the CPU can theoretically access is:

$$\begin{aligned} \text{Bandwidth} &= \text{Single Channel Bandwidth } \times 2 \\ &= 23.464 \text{ GiB/s } \times 2 \\ &= 46.928 \text{ GiB/s} \end{aligned}$$

This value acts as our initial estimate for the **hardware baseline** bandwidth in read and write operations. We **avoid** making any software baseline estimates as the use of loop unrolling and cache warming would reduce the loop overhead and pointer address calculation overhead to an insignificant fraction of the cycles measured. We now present our measurements:

Tables 15 and 16 indicate that we use roughly half of the available RAM bandwidth for both reads and writes. We notice that the write bandwidth is less than the read bandwidth.

Estimated Bandwidth (GiB/s)	46.928
Measured Read Bandwidth (GiB/s)	26.057
Measured stdev in reads (cycles)	147
Avg. % of bandwidth used in read	55.527 %

Table 15: Results of memory bandwidth benchmarks in read operations

Estimated Bandwidth (GiB/s)	46.928
Measured Write Bandwidth (GiB/s)	23.152
Measured stdev in writes (cycles)	27
Avg. % of bandwidth used in write	49.335 %

Table 16: Results of memory bandwidth benchmarks in write operations

We restrict the task execution to a single core, as in all of the previous benchmarks. While this has the advantage of providing stable clock measurements, and does not pollute our measurements with overheads in moving a running process between CPUs, it also restricts the available memory bandwidth as each CPU will only be able to access a fraction of the memory bandwidth calculated above. We are satisfied with the result that in the case of both reads and writes, we predict a good value for the single channel bandwidth.

The real bandwidth of DDR4-2933 memory is controlled by various software and hardware states which are controlled by the firmware of the machine, as well as other physical factors like temperature and thermal efficiency.

We can explain the slowdown in write bandwidth with the fact that a single write to RAM is an inherently expensive process due to the energy required to change the state of each bank of capacitors every time a write is performed.

A simple experiment that tests how much memory bandwidth we can extract from our machine is to run our test program for checking the memory bandwidth on two processor cores that are as far apart from each other to ensure that their memory channels overlap to as little an extent as possible, and there is no effect of spatial locality. We choose core 0 and core 11, and trigger the generated executable to run on both cores in parallel using job control operators in the Bash shell. Table 17 summarizes the results.

We see that we are now able to access between 87 and 90% of the maximum theoretical bandwidth of the system, even though the utilization can never reach 100% due to the existing user and kernel level processes that continuously run and consume portions of the memory bandwidth. We note that the read bandwidth still exceeds the write bandwidth across multiple cores too. This set of results validates our initial hypothesis that running the benchmarks on a single core is a decent proxy for the single channel RAM bandwidth.

	Read Bandwidth (GB/s)	Write Bandwidth (GB/s)
Core 0	20.031	19.817
Core 11	21.283	20.089
Total	41.314	39.906
% of maximum b/w	90.2%	87.13%

Table 17: Testing RAM bandwidth on multiple cores

4.3 Page fault service time

The page fault time measurement methodology is inspired by [17]. We use Linux's `mmap` system call to map a 4 GB file to the running process' address space. We back the mmap-ed region of memory against a 4GB file filled with random data (we use the `head` command with `/dev/urandom` as a source of random data to populate the file). We then induce page faults by writing a single byte to each 4KB region in a loop. Care is taken that the number of loop iterations is less than or equal to the size of the backing file, otherwise we run the risk of writing to pages which are already present in memory (loaded in previous iterations) which makes our benchmarks less accurate.

We make the distinction here between a **hard** and a **soft page fault**. A hard page fault is one that must be satisfied by loading a page from the disk, while a soft page fault is one that is serviced by simply retrieving pages from the application's standby list of pages that are already present in memory. We attempt to trigger hard page faults using the above approach and benchmark the overhead of trapping into the kernel, reading a page from disk, updating the page table structures and return to the user level.

We derive a simplistic estimate of the page fault time by simply summing the time needed to read a single page from disk, then write that page to memory, and the system call overhead which is a simulation of the cost needed to enter kernel mode to perform page table manipulation. We calculate this below:

$$\begin{aligned}\text{Page Fault Time} &= \text{Memory write time (4K)} \\ &\quad + \text{Disk read time (4K)} \\ &\quad + \text{System call Overhead} \\ &= \frac{4\text{ KB}}{23.152\text{ GB } s^{-1}} + \frac{4\text{ KB}}{3500\text{ MB } s^{-1}} \\ &\quad + 0.1664\mu s \\ &= 0.1728\mu s + 1.1429\mu s + 0.1664\mu s \\ &= 1.4820\mu s\end{aligned}$$

In this case, the sum of the memory write time and disk read time constitute our **hardware baseline performance estimate** of 1.3157 μs and the system call overhead constitutes our estimate of the **software overhead** of 0.1664 μs.

We use the figure for memory write bandwidth to memory from the experimental results of Section 4.2. The disk read bandwidth is derived from Table 3, while the system call overhead is derived from Table 9.

In a 4GB anonymous region of mmap-ed memory, there are 1 million pages. We measure the overhead of 1 million page faults, and take the standard deviation of 100 such batches of average measurements. We now present our experimental results:

Estimated Page Fault Service Time	1.482 μ s
Measured Page Fault Service time	4.477 μ s
Measured Std. Dev.	0.341 μ s

Table 18: Page Fault Service Time

The simple heuristic calculation above does not take into account the overhead of additional operations like updating the page tables in memory.

We estimate an access time of 150 cycles to read a single byte of main memory, based on the plot in Figure 2 for the stride size of 1024 bytes. This shows us that the page fault time is roughly 76.85 times the time needed to access one byte of main memory.

5 Network Operations

This section covers some benchmarks regarding the network performance of the target machine. For the purposes of this study, we measure all the metrics on both the software-defined **loopback** interface, as well as the hardware defined **wireless** interface present on our target machine. All measurements that involve *non-loopback* (i.e. outside of *localhost*) communications are carried out over a **wireless** network within UCSD's Graduate and Family Housing complex. For non-loopback measurements, we use an AWS EC2 `t2.small` instance. The description of this remote machine is supplied in Section 5.1.

We measure the performance for all the below operations using the **TCP Protocol**. TCP is a transport-layer transmission protocol that offers **reliable**, **ordered** and **error-checked** delivery of packets. All of these requirements cause the network stack to incur additional software and hardware overheads over other transport-layer protocols like UDP. In addition, popular networking utilities like `ping` directly use network-layer protocols like ICMP thus incurring zero overhead from transport layer protocols like TCP.

The aim of these experiments is to measure the total overhead introduced by the TCP protocol, as well as provide a

rough estimate of the basic performance of the network stack (in both hardware and software) provided in our target machine.

5.1 Remote Machine Description

CPU Model	Intel Xeon E5-2686 v4
Number of cores	1
CPU Base Frequency	2300 MHz
L1 Instr. and Data Cache	32KB (each)
L2 Cache Size	256KB
L3 Cache (LLC) Size	46080 KB (45MB)
DRAM Capacity	2GB
DRAM Bus Width	64 bit
Disk Capacity	12 GiB
Network Interface Type	Ethernet
Operating System	Ubuntu 24.04.1 LTS
Linux Kernel Version	6.8.0-1018-aws
AWS Availability Zone	us-west-2c (Oregon)

Table 19: Machine Description of Remote AWS EC2 Instance

5.2 Round trip time

We write a simple program that creates a TCP ([7]) socket to the target interface (either `localhost` via the loopback interface, or our remote AWS EC2 instance via the wireless network interface). We measure the time needed to send and receive back a 56-byte packet of data across the interface. The choice of packet size mimics the size of a single ICMP packet using by the `ping` utility, which has a 56-byte payload. On the server side, use the `ncat` utility [10] to mimic an echo server that listens on port 8888, and responds to each incoming request by echoing the same content back to the sender.

Our base estimate of the network stack's performance is derived by simply performing a `ping` test on the interface for 100 tries.

The loopback interface is a *virtual network interface* that is implemented entirely in software. Any requests directed to the `127.0.0.0/8` subnet will be intercepted by this loopback interface and handled entirely by the OS kernel's networking subsystems. This implies that there is no overhead of physical transmission over a wire, or any sort of physical buffering overhead imposed by the networking hardware within our target machine as well as the remote machine.

	Setup (ms)	**RTT** (ms)	**Teardown** (ms)
Estimate	0.072	0.048	0.024
Actual	0.34	0.876	0.013
Std. Dev.	0.0094	0.0938	0.00019

Table 20: TCP Connection Parameters for loopback interface

	Setup (ms)	**RTT** (ms)	**Teardown** (ms)
Estimate	60.862	40.575	20.287
Actual	44.979	44.08	0.067
Std. Dev.	7.011	7.352	0.0099

Table 21: TCP Connection Parameters for wireless interface

The combined results of experiments in Sections 5.2 and 5.3 for the **loopback interface** are listed in Table 20. We see that there is a 0.828ms overhead between the estimate made by the `ping` command, and our measured RTT. This can be attributed to the extra overhead caused by the TCP protocol, since ICMP (used by ping) is a network-layer protocol and does not incur any transport-layer overhead. Transport layer overhead added by TCP includes forming the packet headers for each outgoing packet, decoding the headers of incoming packets, computing the sequence numbers for the sender and receiver, and ensuring re-transmission in case of packet loss. In addition, the TCP MTU set for this interface is 65536 bytes, which is massive compared to our data unit of 56 bytes, adding further overhead on the network compared to an ICMP packet which is much smaller.

The results for the **remote connection** to the AWS instance are listed in Table 21. We see that the round trip time measured is again around 3.5ms larger than the ping RTT, which is attributed to the above listed overheads added by the TCP transport layer protocol.

5.3 Connection overhead: Setup and Teardown

We measure the time needed to create and close a TCP socket to the remote host and the localhost via the loopback interface. In the POSIX C Standard Library, the `connect()` system call connects the socket referred to by a file descriptor to the specified address. We measure the time needed for the connect call to execute as the connection setup time. For the teardown, we measure the time needed to close the socket using the traditional `close()` system call that takes the socket's file descriptor as input.

Section 5.2 leads us to the overhead caused by the TCP protocol needed for a single round trip. Since a TCP connection establishment typically requires 1.5 RTTs (the syn-ack phase happens in a single transaction), our initial estimate of the setup time in both remote and loopback cases is simply 1.5 times the measured RTT.

Our estimate of the teardown time is half of the measured RTT in Section 5.2. This is because we are measuring the connection teardown time from the client's perspective, which

involves sending a FIN packet to the server. If the server tries to send any data after the `close()` call, the protocol stack on the client will simply return an "RST" message which leads to the commonly observed "Connection reset by peer" message.

Our results are presented in Tables 20 and 21. In both cases, we appear to have overestimated the setup time and teardown time. We suspect that the `close()` function simply sends the FIN packet to the server and exits, while still remaining in the `FIN_WAIT_1` state. This is referred to as a "half shutdown" in TCP since TCP is a bi-directional protocol. The additional time taken for the teardown in the remote case as compared to the loopback case is attributed to the fact that the teardown of the TCP connection involves additional overheads like cleaning up the receive and send buffers for the connection, which take longer for the network hardware than the loopback software.

In the case of the setup time, we see that the remote setup time is roughly similar to the measured RTT. From the client's perspective this makes sense as the client enters the `ESTABLISHED` state after it receives the server's `SYN/ACK` message. The extra half RTT required for sending the client's ack to the server is not measured from the client's perspective.

We acknowledge the high variance introduced in the remote measurements as a consequence of the fact that our remote and target machines are on different subnets, across a large geographical distance and connected over multiple hops. We accept a standard deviation which is up to 10% of the measured average, as an accurate depiction of the real-life performance of our machine's network stack, and the inconsistencies that TCP/IP networking is associated with.

5.4 Peak bandwidth

We attempt to measure the bandwidth of both the software-defined loopback interface as well as the hardware wireless interface. The basic setup of our experiment is to have a separate client and a server program. The server program runs on the AWS instance when we take the measurements for the remote case. The client program initiates a TCP socket connection to the server, and the client and server exchange 128 KB chunks of data between each other, over a time duration of 20 seconds. We choose the time duration to be intentionally long to make up for the effects of the TCP slow start and other congestion avoidance strategies that are part of the TCP protocol.

For the **loopback interface** which is defined purely in software, we consider the speed of the memory to be the most significant bottleneck. We take the average of the combined read and write bandwidths obtained in Table 17 as our estimate of the loopback bandwidth. Note that the quantity is multiplied by 8 to convert from megabytes to megabits/s.

For the **remote interface**, the `iwconfig` tool reveals the bit rate of the UCSD-Protected network as 135.4 Mb/s. As a heuristic, our estimate will be half of this quantity since we are conducting the experiment on a public network (UCSD-Protected) with a very large probability of packet collision and retries being undertaken due to contention over the bandwidth.

	Loopback	**Remote**
Estimated B/W	324.88 Gb/s	67.7 Mb/s
Actual B/W	49.351 Gb/s	4.34 Mb/s

Table 22: Peak Network Bandwidth

There are a number of reasons for our estimates being much higher than the actual measured quantities. In the case of the **loopback interface**, we suspect that sending 128KB packets constantly without any sort of delays in between causes a saturation of the receiver's receive window, causing TCP congestion control protocols to kick in and restrict the bandwidth to avoid denial of service. For a sanity test, we attempt to run the `iperf3` benchmark on our machine, and across 10 iterations receive a bandwidth of 34.9 Gb/s. This, being on the same order of magnitude as our measured value, gives us confidence in the accuracy of our measurement.

In the case of the **remote interface**, it is fairly clear that high congestion (the experiment was conducted in the afternoon which is a peak working hour) and competition for the available bandwidth restrict the amount of bandwidth available to us. As a sanity check, we use Google's online speed-test utility which yields an average bandwidth of 6 Mb/s. We accept our results as being on the same order as our sanity-checked values.

6 File System

6.1 Size of file cache

In order to estimate the size of the file cache, we write a simple program that reads 64KB chunks of data from different-sized files containing random data (generated from /dev/urandom). The program first syncs all pending disk writes to the disk (using the `sync` command) and then clears the file buffer cache in memory by writing the value **"3"** to the file `/proc/sys/vm/drop_caches` which frees all the cached pages, directory entries and inodes from the file buffer cache on disk [19]. We start from a file size of 128MB and double it until we reach 16GB, which gives us 8 samples. Our measurements are made on an unstressed system and we take care to close as many foreground and background processes to leave as much memory available for caching file reads. We make sequential reads in 64KB chunks from the beginning of the file as we want to observe the extent of caching and read look-ahead that the file system is capable of performing.

Our initial estimate of the buffer cache size is the size of the main memory, i.e. 16 GB. We do not expect the entire

main memory to be used for buffering the file, but we do expect a significant portion of it to be used, with less than 1 GB remaining free when we try to read large files.

File Size (MB)	Access Time (ms)
128	16.593
256	34.726
512	67.208
1024	131.502
2048	252.340
4096	484.61
8192	936.426
16384	5052.110

Table 23: File Read Time as a function of file size

Table 23 gives us the file access time for the above mentioned file sizes. We notice that until the file size reaches 8GB, the access time roughly doubles as the file sizes doubles. Post 8GB, we notice a significant spike in access time as the file size finally hits the main memory size, forcing the OS to hit the disk directly, instead of being able to buffer the file data in cache. This spike is noticeable in the log-log plot below.

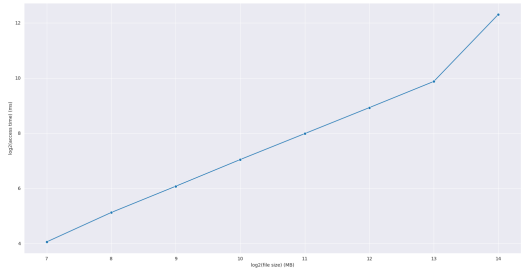

Figure 3: Text file read time as a function of file size

We attempt to make a more precise measurement by performing the same experiment with files ranging from 12GB to 15GB increasing in increments of 1GB.

File Size (GB)	Access Time (ms)
12	1438.01
13	1561.03
14	1688.29
15	6636.67

Table 24: File Read Time as a function of file size for large files

Table 24 shows us that the read time actually spikes between 14 and 15 GiB. This leads to us a final measured file cache of around 14 GB. A simple sanity check with the `free` command also tells us that the size of the cache never increases above 14GiB.

6.2 File read time

The code for this experiment and the measurement of file cache size is similar in its setup, in that we attempt to read 64KB blocks of data from files of different sizes, and clear the file caches/sync pending writes to disk before starting each experiment (i.e. sequential and random reads). The difference is that we add the ability to make random reads by generating a random offset, we read the file backwards (i.e. from the last 64KB block, all the way till the file pointer is at position 0) to counter the effect of the OS fetching blocks in a look-ahead manner. In addition to this, we use the `posix_fadvise` system call to mark the entire file with the `POSIX_FADV_DONTNEED` macro, which advises the OS that these file blocks will not be read in the future, and hence need not be cached. It is important to note that the OS is free to ignore this advice [3], but we hope that on an unstressed system with very low memory pressure, the advice will be followed as it is the best mechanism to disable file caching available to us.

The hardware base estimate of the file read is fairly simple to calculate. For a sequential access, we have a read rate of 3500 MB/s from Table 3, and a write speed of around 39.906 GB/s for the RAM (from Table 17) which leads us to a read time of:

$$\begin{aligned}\text{Read Time for 128MB File} &= \text{Time to read 128MB from disk} + \\ &\quad \text{Time to write 128MB to RAM} \\ &= \frac{128\text{ MB}}{3500\text{ MB/s}} + \frac{128\text{ MB}}{39.906\text{ GB/s}} \\ &= 39.778\text{ ms}\end{aligned}$$

Our estimated software overhead consists of the overhead of making 2048 `read()` system calls (since we read a 128MB file in 64KB chunks) which evaluates to 0.3407 ms. Our combined estimate is thus **40.119ms** for reading 128MB from the disk. Since the hardware estimate (which doubles as the file size doubles) dominates this calculate, we assume for the sake of simplicity that this initial estimate doubles as the file sizes doubles.

The hardware estimate for **random reads** remains the same. However, we add the overhead of the `lseek` call to move the file pointer. This is purely a software operation as the SSD does not have any rotational or seek overhead like a traditional hard disk drive. We measure the time for a single lseek as around **14.637** μ**s**, which means that for 2048 lseeks in a 128MB file, it adds 29.976ms to the total read time, giving us a random file read time of **70.096ms** for a 128MB file. The time required for a single lseek is measured by seeking randomly through the file 2048 times and taking the average, using our standard timing harness.

The log-log plot in Figure 4 shows that the file access time doubles as the file size doubles, a similar exponential

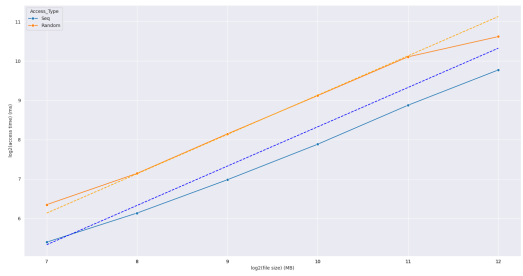

Figure 4: Local File Access Time (Sequential and Random)

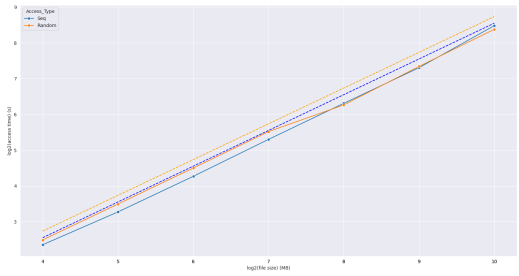

Figure 5: Remote File Access Time (Sequential and Random)

relationship to the buffered access time. However, as expected, the disk read time is larger than the read time from the file cache that was measured in Section 6.1. The read time of 42.003ms for sequential read and 81.332ms for random reads is in line with our estimates. The dotted lines in Figure 4 show the corresponding estimated values as calculated above, and we notice that our estimates roughly line up with the measured value.

6.3 Remote file read time

We repeat the above experiment, with a similar experimental setup. The difference is that we restrict the file size to 1 GB, and add some smaller file sizes to our measurements, in order to ensure that our network is not overstressed, causing our measurements to get skewed. We configure the AWS instance mentioned above as an NFS server, but are forced to access it via the UCSD Campus VPN due to AWS EC2 security group restrictions within our AWS setup. In addition, we perform this experiment on a private network within UCSD Graduate Housing instead of on the public UCSD Protected network, so as to be able to get better bandwidth to complete the experiment in a reasonable amount of time! We use the same methodology as Section 5.4 to measure a bandwidth of **21.82 Mb/s** for the connection between our NFS share.

This leads us to our base estimate for the NFS read performance. For a 128MB file, the network transfer time would be $128\,\text{MB}/21.82\,\text{Mb/s} = \mathbf{46.92s}$. The software overhead, as in Section 6.2, consists of the time needed to make 2048 `read()` system calls, which yields a total estimate of **46.933s** for a 128MB file.

For the random read case, we add the overhead of 2048 seek calls, which is measured in the same way as the local file read time experiment. Over a network, the `lseek` operation requires roughly 3.149ms, which leads to a total seek time of around **6.45s** for a 128MB file. This gives us a total estimate of **53.383s** for a remote file read of 128MB.

Figure 5 depicts the results of the remote file read time for file sizes ranging from 16MB to 1GB, mounted on NFS v4. The dotted lines are the estimated read times for sequential (blue) and random (orange) accesses. We notice that we have overestimated the read times for both sequential and random reads by a small margin. We suspect that since NFS is a very lightweight and optimized protocol, there is very little overhead of communication which means that NFS can access more of the network's bandwidth than our testing script described in Section 5.4 can. An interesting feature to note is that the random access time and the sequential access time seem to line up as the file size increases. This, we suspect is because the network access time starts to vastly dominate the total time as compared to the minimal seek time, which is an entire order of magnitude smaller than the network transfer time.

6.4 Contention in the file system

The aim of this experiment is to measure the effect that an increasing number of processes performing file operations concurrently, has on the file block access time. We spawn n concurrent processes that perform file read operations (one block at a time) from separate files on the same disk. We aim to establish some relationship between the increasing number of processes, and the time needed to read one block of data from the file system on average.

We first implement a simple program that reads a 64 MB file on the disk in chunks of 4096 bytes (i.e. the configured file system block size). This program contains the same methodologies used to ensure that we do not use the cache to read the file (i.e. using `posix_fadvise` with the appropriate options, as well as reading the file backwards to ensure there is no prefetching by the OS). We use the `stat` command to yield the block size of the file system, which is 4096 bytes for our ext4 file system. We then write a simple Bash script that runs the above program n times in parallel, each program accessing a separate 64MB file, with n ranging from 1 to 8. We report the average time needed to read a single block (each file has 16K blocks, and the experiment is repeated 100 times) in Table 25.

Our estimate for the file read time is based on the results of Section 6.2. For a 64MB file, the read time for the entire file would be half of the time taken for a 128MB file, giving us 20.06ms. The file has 16384 blocks, giving us a per-block read time of 0.00122ms. We estimate that this time scales linearly with the increasing number of processes (i.e. the read time for n concurrent processes is n times that of the read time for one process).

No. of concurrent readers	Estimated Block Read Time (ms)	Measured Block Read Time (ms)
1	0.00122	0.002389
2	0.00224	0.002647
3	0.0037	0.004111
4	0.00489	0.004962
5	0.00612	0.006245
6	0.00735	0.007031
7	0.00857	0.008794
8	0.00979	0.009737

Table 25: File System Contention with concurrent processes accessing the file system

Our initial hypothesis seems to be validated by the linear growth in the per-block access time as the number of concurrent processes doing file system accesses increases. The closeness of our estimates means that our methodology to avoid cache accesses and hit the disk only, is validated.

7 Summary

We present the summary of the results of the above experiments in Table 26.

References

[1] *getpid(2) — Linux manual page*, linux man-pages 6.9.1 ed., May 2024. Available at https://man7.org/linux/man-pages/man2/getpid.2.html#HISTORY.

[2] *pipe(2) — Linux manual page*, linux man-pages 6.9.1 ed., June 2024. Available at https://man7.org/linux/man-pages/man2/pipe.2.html.

[3] *posix_fadvise(2) — Linux manual page*, linux man-pages 6.9.1 ed., May 2024. Available at https://linux.die.net/man/2/posix_fadvise.

[4] ABEL, A., AND REINEKE, J. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS* (New York, NY, USA, 2019), ASPLOS '19, ACM, pp. 673–686.

[5] CHEN, S., WU, Y. J., JIANG, X., AND HU, W. Deep dive into the cost of context switch. *Department of Electrical Engineering and Computer Science, Uni. of Michigan* (2019).

[6] DOWNS, T. Hardware Store Elimination — travisdowns.github.io. https://travisdowns.github.io/blog/2020/05/13/intel-zero-opt.html. [Accessed 10-11-2024].

[7] EDDY, W. Transmission Control Protocol (TCP). RFC 9293, Aug. 2022.

[8] FOG, A. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*, November 2022. Available at https://www.agner.org/optimize/instruction_tables.pdf.

[9] GARCIA, F., AND FERNANDEZ, J. Posix thread libraries. *Linux J. 2000*, 70es (Feb. 2000), 36–es.

[10] GIBSON, C., LYON, G., AND KATTERJOHN, K. Chapter 17. ncat reference guide | nmap network scanning, 2022.

[11] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 85th ed., January 2023. Available at https://cdrdv2-public.intel.com/814198/248966-046A-software-optimization-manual.pdf.

[12] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-L*, 85th ed., October 2024. Available at https://cdrdv2-public.intel.com/835751/253666-sdm-vol-2a.pdf.

[13] LI, C., DING, C., AND SHEN, K. Quantifying the cost of context switch. In *ExpCS '07* (2007).

[14] LOVE, R. M. *taskset(1) — Linux manual page*, util-linux 2.39.3 ed., July 2023. Available at https://man7.org/linux/man-pages/man1/taskset.1.html.

[15] MCVOY, L., AND STAELIN, C. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference* (USA, 1996), ATEC '96, USENIX Association, p. 23.

[16] OUSTERHOUT, J. K. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer* (1990), USENIX Association, pp. 247–256.

[17] REN, X. J., RODRIGUES, K., CHEN, L., VEGA, C., STUMM, M., AND YUAN, D. An analysis of performance evolution of linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 554–569.

[18] SAUTHOFF, G. On the costs of syscalls. https://gms.tf/on-the-costs-of-syscalls.html, August 2021.

[19] VAN RIEL, R., AND MORREALE, P. Documentation for /proc/sys/vm/; the linux kernel documentation. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html#drop-caches, 2008.

[20] ZHAO, K., GONG, S., AND FONSECA, P. On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems* (New York, NY, USA, 2021), EuroSys '21, Association for Computing Machinery, p. 540–555.

Operation	**H/W Estimate**	**S/W Estimate**	**Total Estimate**	**Measured Time**
Timing Overhead	-	12.731 ns	12.731 ns	13.263 ns
Loop Overhead	-	1.924 ns	1.924 ns	1.943ns
System Call Overhead	-	142.14 ns	142.14 ns	166.383 ns
Thread Creation Overhead	-	19.5 μs	19.5 μs	9.769 μs
Process Creation Overhead	-	0.293 ms	0.293 ms	0.350 ms
Thread Context Switch Overhead	-	3.858 μs	3.858 μs	4.0003 μs
Process Context Switch Overhead	-	15.432 μs	15.432 μs	22.614 μs
L1 Cache Access Time	1.543 ns	2.7ns	4.244 ns	3.332 ns
L2 Cache Access Time	4.63 ns	2.7ns	7.33ns	5.165ns
L3 Cache Access Time	16.975 ns	2.7ns	19.675 ns	15.675 ns
Main Mem. Access Time	50.54 ns	2.7 ns	53.24ns	55.44 ns
Main Mem. Bandwidth (Read)	46.928 GB/s	-	46.928 GB/s	41.314 GB/s
Main Mem. Bandwidth (Write)	46.928 GB/s	-	46.928 GB/s	39.906 GB/s
Page Fault Service time	1.3157 μs	0.1664 μs	1.482 μs	4.477 μs
Network RTT (Loopback)	-	0.048 ms	0.048 ms	0.876 ms
Network RTT (Wireless)	-	40.575 ms	40.575ms	44.08 ms
Network Setup Time (Loopback)	-	0.072 ms	0.072 ms	0.34 ms
Network Setup Time (Wireless)	-	60.862 ms	60.862 ms	44.979 ms
Network Teardown Time (Loopback)	-	0.024 ms	0.024 ms	0.013 ms
Network Teardown Time (Wireless)	-	20.287 ms	20.287 ms	0.067 ms
Peak Network Bandwidth (Loopback)	-	324.88 Gb/s	324.88 Gb/s	49.351 Gb/s
Peak Network Bandwidth (Wireless)	-	67.7 Mb/s	67.7 Mb/s	4.34 Mb/s
File Read Time (128MB, local, file cache)	-	-	-	16.593 ms
File Read Time (128 MB, local, disk)	39.778 ms	0.3407 ms	40.119 ms	42.003 ms
File Read Time (128 MB, NFS)	46.92 s	0.3407 ms	46.933 s	39.444 s

Table 26: Summary of Performance Results