# CSE 221 Project Milestone 2

Edwin Mascarenhas, Raghav Prasad, Sandhya Jayaraman

## 1 Introduction

There are a wide variety of systems today, each running different types of Operating Systems (OS), different versions, hardware capabilities etc. While deploying an application on such a system, or set of systems it is useful to understand and characterize the performance of key operations of the system.

The goal of this project is to write a set of benchmarks in C/C++ to measure the performance of key CPU operations, memory operations, network and file system. We will use these benchmarks to evaluate the performance of Edwin Mascarenhas' laptop, which supports an $11^{th}$ gen Intel Core i7 processor. We write our benchmarks predominantly in C, and compile them using the `gcc 9.3.0` compiler. We compile the code at level 0 optimization by running `gcc` with the flag `-O0`.

In this report, we first present specifications related to the machine on which we execute our benchmarks, and follow this with our experiments for measuring CPU operations. We mention which group member implemented each benchmark while describing experiments in future sections. Additionally, we state that we have ensured a fair division of work among the three of us. We estimate that we spent about 5 hours per member on the project so far.

## 2 Machine Description

In order to analyze our measurements, and tune benchmarks, we begin by providing an in-depth machine description. Since we are using a Linux based OS, we identified some of the information from the system using `proc` and `/sys/devices/system/cpu`. We found some of the other specs (such as SSD information) online.

Table 1 summarizes the measurements for some key system specifications of the machine we ran our experiments on.

## 3 Measurement setup

To ensure reliable measurements for the benchmarks we use Read Time-Stamp Counter and Processor ID (RDTSCP) and Read Time-Stamp Counter (RDSTC) instructions for measuring the time.

Modern day processors allow operation in a number of different clock frequency and voltage configurations called Operating Performance Points or P-states. The Linux kernel

| **System Specification** | **Measurement** |
|---|---|
| Processor Model | 11th gen Intel Core i7-1165G7 [4] |
| Physical Cores | 4 |
| L1 Data Cache Size | 48KB * 4 cores = 192KB |
| L1 Instruction Cache Size | 32KB * 4 cores = 128KB |
| L2 Cache Size | 1280KB * 4 = 5MB |
| L3 Cache Size (shared) | 12MB |
| Cycle Time | 0.5882 ns |
| DRAM Type | DDR4 |
| DRAM Speed | 3200 MTs |
| DRAM Size | 12GB (4GB + 8GB) |
| Bus Width | 64 bits |
| Memory Bus Bandwidth | 51.2 GB/s |
| IO Bus Type | - |
| SSD Product | KIOXIA |
| SSD Model | KBG40ZNS512G |
| Capacity | 512GB |
| Transfer Rate | 32 GTs |
| Random Read | 330K IOPS |
| Random Write | 190K IOPS |
| Network Card Bandwidth | - |
| Operating System | Ubuntu 20.04.2 |
| Kernel | Linux 5.11.0-40 |

Table 1: CPU Measurements

allows dynamic frequency changes depending on CPU capacity to improve performance or efficiency [10]. For our measurements we disable intel pstate performance scaling driver and set the frequency to a constant 1.7Ghz.

To prevent the effects of the OS migrating a process from one CPU to another we set the CPU affinity mask for all the benchmarks using the *sched_setaffinity* call. We also set the priority of the current thread as the highest possible for a user process using the *setpriority* call.

# 4 CPU Operations

## 4.1 Measurement overhead

### 4.1.1 Measurement Overhead

Measurement overhead is an important factor to consider in developing benchmarks to study performance. We implement the benchmark for measurement overhead by reading the clock counter twice using the `rdtsc` instruction, and measuring the time taken in doing so. We follow the methodology described in the Intel paper [4] to read the clock counter. Additionally, we execute the clock fetch instructions once before we run the benchmark loop to ensure

the instructions involved are appropriately cached in the Instruction Cache. We executed the benchmark 10000 times.
**Pre-Measure Estimate:** Measurement overhead does not involve any hardware overheads. Therefore, we estimate based on [3] that the instructions executed in this benchmark might take 160$ns$ or 275 cycles.
**Implemented by:** Sandhya Jayaraman
**Measurement:** Measurement overhead took an average of 85.75$ns$, with a standard deviation of 20.42$ns$

We summarize the performance of our benchmark across 10000 iterations in Table 2.

| Statistic | Measurement (cycles) | Time(ns) |
|---|---|---|
| Mean | 145.78 | 85.75 |
| Standard Deviation | 34.72 | 20.42 |

Table 2: Measurement Overhead Benchmark Performance

Based on our results, we figure that our estimate was not too far off from the measured overhead value.

### 4.1.2 Loop Overhead

Determining the loop overhead accounts for the overheads involved in running the code to be benchmarked multiple times in a loop, in order to get as accurate an estimate as possible. To implement this benchmark, we run an empty `for` loop over 100000 iterations and measure its runtime. We read the clock counter before and after running the loop using `rdtsc`, but account for this using the measurement overhead values. Additionally, we specifically ensure that compiler optimization is set to level 0 (`gcc -O0`) to ensure that the compiler doesn't optimize the empty loop away.
**Pre-measurement Estimation:** Using the values provided in [3], we estimate that the loop overhead to be around 3.7 cycles, which would take about 2.18$ns$.
**Implemented by:** Sandhya Jayaraman
**Measurement:** Loop overhead took about 1.2$ns$ on average.

We summarize the performance of our benchmark across 10000 iterations in Table 3.

| Statistic | Measurement (cycles) | Time (ns) |
|---|---|---|
| Mean | 2.02 | 1.2 |
| Standard Deviation | 0.27 | 0.16 |

Table 3: Loop Overhead Benchmark Performance

In coming up with an estimate for loop overhead, we selected the worst case time for executing, say an instruction, when a range was specified. Apart from that, we figure that our estimate on this is also fairly close to the measured value.

## 4.2 Procedure call overhead

Procedures are used to modularize and segment code. Procedures may accept 0 or more parameters and return at most 1 value.
Here we look at estimating the overhead associated with procedure calls as a function of the number of parameters it accepts. We have created 8 procedures (or functions) accepting 0 through 7 parameters and no return value. We choose the parameters of these procedures to be `int` and we pass 0s in the procedure calls. We perform each procedure call 1000 times and calculate the average time overhead for one procedure call.
**Pre-measurement estimate**: N/A
**Implemented by:** Raghav Prasad

We summarize the performance of our benchmark in Table 4.

| **Statistic** | # Arguments | **Measurement (cycles)** | **Time (ns)** |
|---|---|---|---|
| Mean | 1 | 37.986 | 22.343 |
| Standard Deviation | | 1.66 | 0.976 |
| Mean | 2 | 37.904 | 22.29 |
| Standard Deviation | | 1.38 | 0.81 |
| Mean | 3 | 38.6 | 22.70 |
| Standard Deviation | | 2.74 | 1.61 |
| Mean | 4 | 37.88 | 22.28 |
| Standard Deviation | | 1.2 | 0.71 |
| Mean | 5 | 38.41 | 22.59 |
| Standard Deviation | | 4.37 | 2.57 |
| Mean | 6 | 37.938 | 22.31 |
| Standard Deviation | | 1.88 | 1.105 |
| Mean | 7 | 37.918 | 22.30 |
| Standard Deviation | | 1.37 | 0.805 |

Table 4: Procedure Call Overhead Benchmark Performance

## 4.3 System Call overhead

To measure the time overheads in invoking a system call, we run the `getpid` function that returns the process ID of the calling process. However, the system caches the results of a `getpid` call, which makes results dubious, unless we invoke the system call afresh each time. In order to force the system to make a new system call each iteration, we `fork` the process each iteration and measure the time taken to run `getpid` on the new forked process. This ensures that the caching effect does not factor into the accuracy of our measurements. We exit the child process each iteration, making sure that a new process is forked in the next iteration.

**Pre-measurement estimate:** Based on [9], we estimate that a system call should take between 1000-1500 cycles or $735ns$.

**Implemented by:** Sandhya Jayaraman
**Measurement:** System call overheads took about 88.9$ns$ on average.

We summarize the performance of our benchmark over 10000 iterations in Table 5.

| **Statistic** | **Measurement (cycles)** | **Time (ns)** |
|---|---|---|
| Mean | 151.19 | 88.9 |
| Standard Deviation | 49.30 | 29.0 |

Table 5: System Call Overhead Benchmark Performance

**How does it compare to the cost of a procedure call?**

## 4.4 Task creation time

Here we consider the time overhead associated with task creation. We consider two kinds of tasks a) a process, and b) a thread.
**Implemented by**: Raghav Prasad

### 4.4.1 Process

We create a process by using `fork()`. `fork()` returns the PID of the child process to the parent, and 0 to the child process. Using this fact, we are able to distinguish between the parent and child processes and each child process is terminated right after it gets created by calling `return 0` on it. Thus, the parent process creates 1000 child processes in a loop and this is measured to obtain the average process creation time.
**Pre-measurement estimate**: $\sim$ 750000 cycles [8]

We summarize the performance of our benchmark in Table 6.

| **Statistic** | **Measurement (cycles)** | **Time (ns)** |
|---|---|---|
| Mean | 55463.29 | 32623.50 |
| Standard Deviation | 14224.22 | 8366.68 |

Table 6: Process Creation Benchmark Performance

### 4.4.2 Thread

We create a kernel level threads by using POSIX thread libraries (`pthread`). We use `pthread_create` to create a thread and call a function on it that immediately kills the thread. In this way, we create 1000 threads in a loop and this is measured to obtain the average thread creation time.
**Pre-measurement estimate**: Much lesser than for a process; $\sim$ 75000 cycles

**How do they compare?** Threads are much more lightweight than processes. They share the same address space and therefore the overhead associated with thread creation is significantly lesser than that for a process.

We summarize the performance of our benchmark in Table 7.

| Statistic | Measurement (cycles) | Time (ns) |
|---|---|---|
| Mean | 23589.42 | 13875.30 |
| Standard Deviation | 11446.68 | 6732.93 |

Table 7: Thread Creation Benchmark Performance

Process and thread creation are expensive tasks, but both process and thread creation performed much better than our estimates.

## 4.5 Context switch time

To enable multiple processes to share the CPU, the OS' preempt a running process, save it's state and run another process that requires the CPU. This is known as a context switch. To measure the context switch time the approach we use is similar to the one described in lmbench [5]. The parent process and child process are confined to a single CPU using the affinity mask. Two pipes are created for communication between them. The child process sends a token to the parent on pipe1 and waits for the token to come back from the parent on pipe2. The parent process waits for a token from the child on pipe1 and upon receiving writes the token to the child on pipe2. This continues for N iterations. The read and write system calls are used for this purpose. The read call blocks and so it triggers a context switch. Moreover both processes are confined to the same core and so there is a context switch after each process writes and waits for the read. In each iteration there is two context switches occurring and two reads and writes to the pipe.

For isolating the measurement of the context switch time, we first measure the time to write and read to a pipe. We then subtract the read and write time from the benchmark time.

**Pre-measurement estimate**: 3400 cycles [2]
**Implemented by**: Edwin Mascarenhas
**Measurement:** Process context switch took 2135 cycles which comes to 1255$ns$. Kernel threads context switch took 1142$ns$.

We summarize the performance of our benchmark in Tables 10 9.

| Statistic | Measurement (cycles) | Time (ns) |
|---|---|---|
| Mean | 2165.52 | 1273.45 |
| Standard Deviation | 39.10 | 22.99 |

Table 8: Context Switch Process Benchmark Performance

| Statistic | Measurement (cycles) | Time (ns) |
|---|---|---|
| Mean | 2046 | 1023.45 |
| Standard Deviation | 38.37 | 22.57 |

Table 9: Context Switch Thread Benchmark Performance

These results align very well with our initial estimates for context switch time.

## 5 Memory Operations

### 5.1 RAM Access Time

Modern day processors have a complicated memory hierarchy and the latency of a memory operation depends on which level of cache or main memory is the requested byte resident in. To reliably measure the memory latency, we use an approach similar to the one described in lmbench [5]. We use a pointer chasing approach where the value loaded from memory is used to issue the next load request, $index = buf[index]$;. Before the measurement, the values of the array of size $N$ are initialized in a specific pattern. The array access starts from index 0 by default. The value at any index x is randomly initialized to an index in the other half of the array different from the half x belongs to. We do this for two reasons:

- **Random initialization** CPUs have data based prefetchers which prefetch the next line in cache as well as instruction pointer prefetchers which can prefetch the next address based on a certain stride. So without random initialization, we would end up hitting in the L1, irrespective of the stride for any size.
- **Ping-pong between locations** We randomly initialize it to a location in the other half and ping pong to distribute the accesses throughout the array. This reduces the chances of the next access hitting in the cache.

**Pre-measurement estimate**: In this document [1], they state Kaby Lake's L1 cycle time as 5 cycles, L2 as 12 cycles, L3 as 42cycles and main memory as 60-80ns. These cache numbers are at 2.5Gz. Using these numbers as estimates, since we are running at 1.7Ghz L1 access should take about 7 cycles, 18cycles and 61 cycles. Main memory we expect 60-80 ns latency.
**Implemented by**: Edwin Mascarenhas
**Measurement** The size of the array is varied at different strides and the latency in cycles is plotted as shown in Fig. 1. The x-axis is the $log_2(numberofelements)$ in array. Since each element is 4 bytes, the size in bytes is the value on x-axis multiplied by 4.

In the graph the different regions can be identified. The sharp outliers in the LLC cache and DRAM region are possibly because of TLB misses. With 4KB pages it is very likely at higher array sizes the benchmark with it's random accesses is thrashing the TLB. From measurement it is found that L1 cycle time is 7 cycles, L2 is 15 cycles, L3 55 cycles and DRAM latency is 200 cycles.

### 5.2 RAM Bandwidth

To measure the read bandwidth of our main memory, we initialized an array of size 1000 MB. By default, it contains indeterminate (garbage) values. We then proceeded to read each element of the array into a temporary variable and recorded the number of cycles required to complete a full pass through the array. A similar exercise was done to measure the write bandwidth, except here we wrote to the array and changed each value of the array to 1. We averaged this over 100 iterations, making sure to clear the cache after each iteration.
Thus we get two values, *read_cycles* and *write_cycles* that correspond to the number of

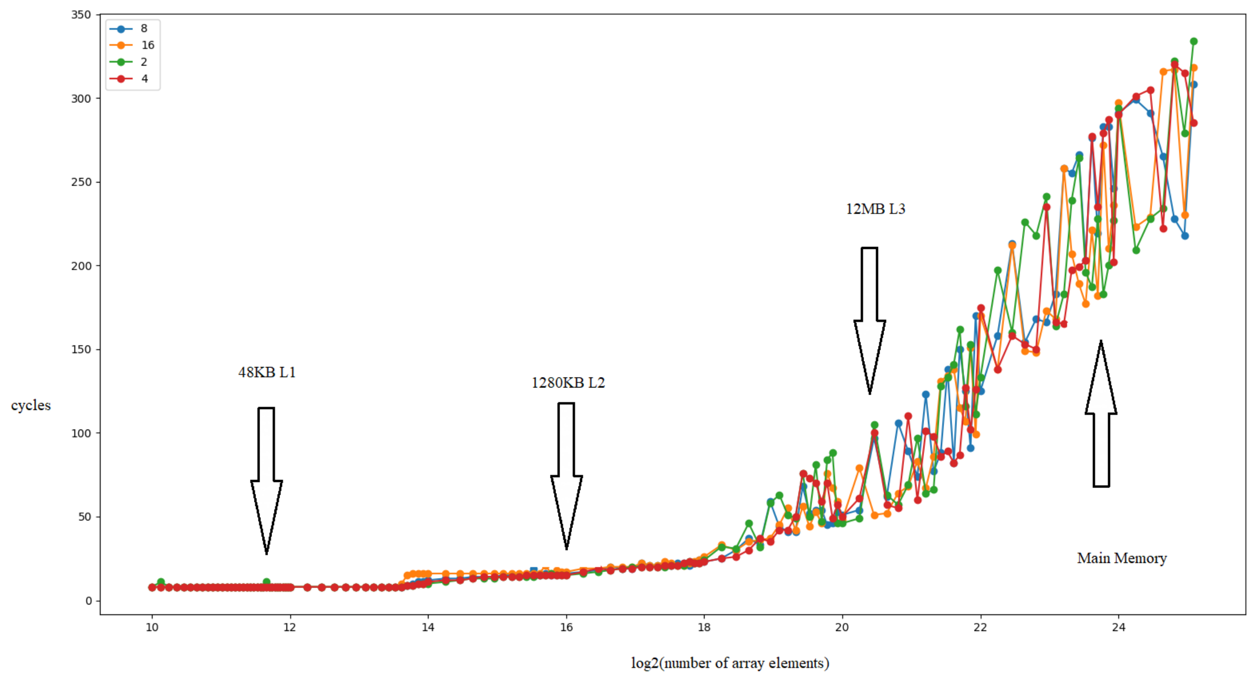

Figure 1: Memory Latency for increasing array sizes.

cycles required to read from and write to the 1000 MB array. From here, we calculate the read and write bandwidths as follows:

$$read\_bandwidth = \frac{array\_size}{time\_for\_reads} \tag{1}$$

$time\_for\_reads$ is calculated as $time\_per\_cycle * read\_cycles$, where $time\_per\_cycle$ is merely the inverse of the clock frequency, which in our case is 1.7 GHz

Similarly, we calculate write bandwidth as follows:

$$write\_bandwidth = \frac{array\_size}{time\_for\_writes} \tag{2}$$

$time\_for\_writes$ is calculated as $time\_per\_cycle * write\_cycles$

**Pre-Measurement Estimate**: 52428.8 MB/s (ref. Table 1)
**Implemented By**: Raghav Prasad
**Measurements:**
**Read bandwidth:** 1110.471 MB/s, **Write bandwidth:** 718.380 MB/s

We find that our measured values are lower than our pre-measurement estimate, which is completely valid and expected, because our pre-measurement estimate is the theoretical maximum bandwidth. Another interesting thing to note is that the read bandwidth is higher than the write bandwidth. This is logically correct. Writing to RAM does take more time than reading from RAM, since writing requires more energy.

## 5.3 Page Fault Service Time

To measure the time taken to service a page fault, we first identified how to force a page fault to occur. To do this we generated a large file of size $1GB$ usind the `dd` command, and used the `mmap` [6] function to map the file into the virtual address space of the calling process. We attempt to access random pages of the file using a random index and page size offset at least 10 pages away from the previous index. This predominantly raises a page fault, which is serviced. Additionally, we clear the page cache every iteration by running `sync; echo 1 > /proc/sys/vm/drop_caches`, and set the `fadvise` flag to `POSIX_MADV_DONTNEED`.

In servicing a page fault, the kernel handles an interrupt, TLB miss, page frame allocation, and the actual page transfer. In our experiment, we only request 1 page, which is $4KB$. Let us assume a disk bandwidth of about $3000MB/s$, so the actual page transfer would take about $1300ns$. We assume that the hardware overheads take about $300000ns$ based on [7]. Therefore, we estimate that the latency of a page fault service is $301300ns$ or 512240.73 cycles.
**Pre-measurement estimate**: 512240.73 cycles
**Implemented by**: Sandhya Jayaraman
**Measurement:** Page Fault Servicing took 3193444.864 cycles which comes to $1878384.27ns$.

| Statistic | Measurement (cycles) | Time (ns) |
|---|---|---|
| Mean | 3193444.864 | 1878384.27 |
| Standard Deviation | 3027999.836 | 1781069.503 |

Table 10: Page Fault Servicing Benchmark Performance

We note that our estimate was a lot lower than the measured number of cycles required in servicing a page fault. We assume that this could mean that the hardware operations involved in page fault service have a higher latency than we anticipated, and our disk bandwidth may also be lower. Additionally, we noticed that not clearing the page cache produced consistently lower numbers in terms of number of cycles, indicating that the pages accessed may be cached leading to lower latencies. We fixed this issue by clearing the page cache each iteration, we gave us more consistent measurements.

**Dividing by the size of a page, how does it compare to the latency of accessing a byte from main memory?** Page fault servicing 1 byte from the memory (dividing by page size $4KB$) takes 779.65 cycles or $458.58ns$. On the other hand, accessing a single byte from the main memory takes about 200 cycles. We feel an approximate four-fold increase in time to service a page fault makes sense, since each page fault involves at least two accesses to the main memory (to read the page, and to access the page table), and there is the overhead of loading an entire page into the memory.

## References

[1] *Applied C++: Memory latency*. URL: https://medium.com/applied/applied-c-memory-latency-d05a42fe354e.

[2] Martin Becker and Samarjit Chakraborty. "Measuring Software Performance on Linux". In: *CoRR* abs/1811.01412 (2018). arXiv: 1811.01412. URL: http://arxiv.org/abs/1811.01412.

[3] Agner Fog. *Instruction Tables*. Aug. 2021. URL: https://www.agner.org/optimize/instruction_tables.pdf.

[4] Intel. *Intel® Core™ i7-1165G7 Processor*. https://www.intel.com/content/www/us/en/products/sku/208662/intel-core-i71165g7-processor-12m-cache-up-to-4-70-ghz/specifications.html.

[5] Larry W McVoy, Carl Staelin, et al. "lmbench: Portable Tools for Performance Analysis." In: *USENIX annual technical conference*. San Diego, CA, USA. 1996, pp. 279–294.

[6] *mmap(2) — Linux manual page*. https://man7.org/linux/man-pages/man2/mmap.2.html.

[7] *Page fault*. URL: https://en.wikipedia.org/wiki/Page_fault.

[8] Rob Pike et al. *Plan 9 from Bell Labs*. 1995.

[9] *Protection Ring*. URL: https://en.wikipedia.org/wiki/Protection_ring.

[10] Rafael J. Wysocki. *intel_pstate CPU Performance Scaling Driver*. https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html.