CSE 221 Project Milestone 1

Edwin Mascarenhas, Raghav Prasad, Sandhya Jayaraman

January 21, 2022

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 allows dynamic frequency changes depending on CPU capacity to improve performance or efficiency [7]. 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.

System Specification	Measurement
Processor Model	11^{th} gen Intel Core i7-1165G7 [3]
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 GBs
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

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 [3] 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 [2] that the instructions executed in this benchmark might take $160ns$ or 275 cycles.

Implemented by: Sandhya Jayaraman

Measurement: Measurement overhead took an average of $85.75ns$, with a standard deviation of $20.42ns$

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 [2], we estimate that the loop overhead to be around 3.7 cycles, which would take about $2.18ns$.
Implemented by: Sandhya Jayaraman
Measurement: Loop overhead took about $1.2ns$ 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 [6], we estimate that a system call should take between 1000-1500 cycles or 735ns.
Implemented by: Sandhya Jayaraman
Measurement: System call overheads took about 88.9ns 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 [5]

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 [4]. 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 [1]
Implemented by: Edwin Mascarenhas
Measurement: Process context switch took 2135 cycles which comes to $1255ns$. Kernel threads context switch took $1142ns$.

We summarize the performance of our benchmark in Tables 8 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.

References

[1] 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.

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

[3] 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.

[4] 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.

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

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

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