Systems Measurement Project - Draft of Intro, Machine Description, and CPU Operations

Eric Hu, Sarat Sreepathy, Chitra Kharbanda

February 1, 2022

1 Introduction

We performed our project in a school-hosted virtual machine to research the specifications and capabilities of their environment. We performed our measurements in C and used the GCC compiler from the GNU Compiler Collection, using version 7.5.0. For our compiling, we used the $-O$ optimization level to improve the performance of our code, at the cost of a slower compiling time.

Using a school-hosted virtual machine, however, left us with some implications. The host is out of our control and we are unable to manage the status of the virtual machine since it was hosted online. This may greatly affect our experiments for networking. There is also the issue of the network response time between our machines and the virtual machine. While we believe that this will not have any affects on our experiments, as they are built into code programs, it is still worth mentioning. Additionally, as it is a virtual machine, it is missing some specifications or were not very detailed.

- Eric: Task creation and run for measurement, procedure call, and system call overhead
- Sarat: Task creation and run for both processes and threads
- Chitra: Task creation and run for context switching for both processes and threads

Estimated time spent on project: 25 hours

2 Machine Description

Below are the listed virtual machine specifications. As the virtual machine environment is limited, we were unable to find detailed descriptions of some specifications.

Operating System: Ubuntu 18.04.6 LTS

Processor

- Processor model: Intel ® Xeon ® CPU E5-2690 v2 @ 3.00GHz
- Cycle time: 3000 MHz (0.33 ns)
- Cache size: 25600KB

Memory

- DRAM: Unknown
- Type: DDR3-1866
- Clock: 30 (clock multiplier)
- ECapacity: 2033016KB
- Memory Bus Bandwidth (Speed): 8 GT/s

I/O Bus Type: Unknown

Disk

- Hard Drive:
  - Model: Xen Virtual Disk A
  - Capacity: 150G
  - RPM: Unknown
  - Transfer rates: Unknown
  - Latencies: Unknown

Network

- Network Card Bandwidth: Unknown
- Network interface: eth0

3 Experiments

3.1 CPU, Scheduling, and OS Services

For our experiments in CPU, Scheduling, and OS Services, we measured time by counting the number of cycles. This done by creating 2 instances of **rdtsc**, where one gets the cycles before the experiment and one gets the cycles right after the experiment.

3.1.1 Measurement Overhead

We first considered the measurement of the measurement itself. To do this, we created 4 instances of **rdtsc**: one for the start of the outer measurement, one for the start of the inner measurement, one for the end of the inner measurement, and one for the end of the outer measurement. With this, we are able to use the following formula to solve how many cycles it took to start the measurement.

$$MeasurementOverhead = endOuter - startOuter - (endInner - startInner)$$

This formula gets the number of cycles it took for the outer measurement to finish, subtracting from it the number of cycles the inner measurement took. This way we would end up with the number of cycles between *startOuter* and *startInner* and the cycles between *endInner* and *endOuter*, which results in the final measurement overhead.

For the inner measurement, we decided to measure both the measurement of procedure calls and system calls so that we could compare the two if there were any notable differences.

3.1.2 Process Call Overhead

Our next experiment was conducted on measuring the time it took to complete a process (function) call. We performed this on processes with different number arguments passed to it, ranging from 0 to 7 arguments, so that we could see the effects of passing arguments on cycle time. The multiple processes were named

foo0(), foo1(int a), foo2(int a, int b), foo3(int a, int b, int c), ..., foo7(int a, int b, int c, int d, int e, int f, int g)

and were fairly simple, as none of them performed any operations and immediately returned 0 when called. For the measurements, we created 2 instances of **rdtsc**: one for the right before the process call, and one for right after the process call. The final measurement was then calculated by subtracting the cycles measured right after the process call by the cycles measured before the process call.

3.1.3 System Call Overhead

We performed experiments on a normal process call, so now we wanted to experiment on a system kernel call. We measured the system process getpid(), which accesses the kernel to return the current process's process ID and getsid(), which accesses the kernel to return the current session ID. This was so we could compare the overhead for the two different system calls. For the measurements, we created 2 instances of **rdtsc**: one for the right before the system call, and one for right after the system call. The final measurement was then calculated by subtracting the cycles measured right after the system call by the cycles measured before the system call.

3.1.4 Task Creation

One of the CPU operations we want to test on this virtual machine is task creation time. We want to measure how long it takes to create and run a process as well as a thread. For process creation, we want to create and run a single process and measure the time it takes in microseconds. For thread creation, we want to create and run a kernel threads and measure time in microseconds. With Linux, we want to create a kernel-managed thread using (p_thread create). For process creation, we want a single child process so we called fork() and exited if the pid == 0 indicated a child and printed "Run process." We waited the parent process until child finished executing so we do not have the child process calling fork() and creating even more processes. We used **rdtsc** to track the number of cycles run before and after the child process had been created and executed. We ran this over several iterations and collected the number of cycles that were run each iteration and converted cycles to microseconds. We did this conversion by using **rdtsc** before and after calling sleep for 1 second to measure how many cycles passed in that second and we would use that number to convert the measurement in cycles to microseconds. We then calculated the average process creation and run time as well as the standard deviation.

For threads, we wanted a single thread to be created and run so we used p_thread in Linux. We initialized the p_thread variable and we also called p_thread create and join which created the kernel thread and waited for finished execution, respectively. Similarly to process creation and run, we have a print statement "Run kernel thread" which specifies that the thread has executed. We also used **rdtsc** to track the time before and after thread has executed using similar method for converting cycles to microseconds. We also produce the average time for thread creation and run time as well as standard deviation.

3.2 Context Switch

We performed experiments on measuring the overheads of the CPU with respect to context switches among threads and processes. The general steps involved while context switching between processes in a operating system is that the state of the currently running process is saved and then again restore when another process is done executing.

For measuring the context switch time, we make use of the fifo pipes which are also known as blocking pipes. The general property of these pipes are that whenever a process writes a data onto these pipes, they get blocked until another process has read from it. This facilitates us to measure the amount of time the CPU used to context switch from one process to another. The methodology implemented to measure the context switch time is basically starting by creating a pipe. Then we fork a child process and allocate the read descriptor of the pipe to the child process. Meanwhile, the parent process gets the write descriptor of the pipe. As the data gets written on the pipe, e process gets blocked till the data has been read from the child process. This makes the CPU to switch the context from parent to child. We measure the duration by recording the time at the end of the writing process and the start of the reading process. The difference between the two gave us the value of the context switch between the processes. We have used the pipe to transfer a array of 500 elements of integer data, which amount to 2000 bytes of data.

For measuring the context switch between the threads, we have used a similar algorithm. We record the time at the end of the writing of data and at the start of the reading of data at the other thread. We subtract the two which gives us the context switch time between the threads.

3.3 Memory

3.4 Network

3.5 File System

4 Results

4.1 CPU, Scheduling, and OS Services

4.1.1 Measurement Overhead

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Measurement Overhead (Process)	N/A	10 μs	10 μs	45 μs	0 μs
Measurement Overhead (System)	N/A	10 μs	10 μs	52 μs	0 μs

For our estimated values of hardware and software overhead, software overhead is the only one applicable for this measurement. We estimated it to be around 10 μs, as we believe this operation will not take very long.

When measuring the measurement overhead, we saw that when it was measuring the measurement of a procedure call it had an average of 45 microseconds whereas on the measurement of a system call it had an average of 52 microseconds. Although it was not a big difference, the system call measurement always had a higher overhead when compared to the process call measurement overhead. The standard deviation for both of these experiments were near 0, so there was little to no variability.

We believe that our measurement is somewhat lacking in accuracy, as we expected that the measurement overhead to be the same for all the measurements we did. It was strange to us that the system call measurement had a consistently greater measurement overhead than the process call measurement overhead.

4.1.2 Process Call

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Process Call(0 Arguments)	N/A	10 μs	10 μs	27 μs	$\approx$ 0 μs
Process Call(1 Arguments)	N/A	11 μs	11 μs	22 μs	$\approx$ 0 μs
Process Call(2 Arguments)	N/A	12 μs	12 μs	23 μs	$\approx$ 0 μs
Process Call(3 Arguments)	N/A	13 μs	13 μs	22 μs	$\approx$ 0 μs
Process Call(4 Arguments)	N/A	14 μs	14 μs	22 μs	$\approx$ 0 μs
Process Call(5 Arguments)	N/A	15 μs	15 μs	22 μs	$\approx$ 0 μs
Process Call(6 Arguments)	N/A	16 μs	16 μs	23 μs	$\approx$ 0 μs
Process Call(7 Arguments)	N/A	17 μs	17 μs	22 μs	$\approx$ 0 μs

For our estimated values of hardware and software overhead, software overhead is the only one applicable for this measurement. We estimated for a baseline process with no arguments to be around 10 μs, and for each additional argument it would take an extra 1 μs.

When running the process call measurements, we expected for the average number of microseconds to increase for each additional argument passed, as it would be another resource that the process would have to transfer. However, we were surprised to see that what we thought would be the fastest process, the process with 0 arguments passed, ended having the largest overhead of 27 microseconds. All the other process calls of 1 through 6 arguments all had an average overhead of around 22 and 23 microseconds. Due to the nature of the results, we are unable to discern the increment overhead of an argument. The standard deviation for all of these experiments were near 0, so there was little to no variability.

We believe for our implementation to be correct, but the accuracy of the results was surprising. We expected that with each additional argument, it would have greater overhead, but our results were the opposite and instead stayed consistent except for the process call with 0 arguments. These results lead us to question the stability of the virtual machine itself, as it gave us contradictory results than expected.

4.1.3 System Call

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
System Call (getpid())	N/A	100 μs	100 μs	5493 μs	$\approx$ 0 μs
System Call (getsid())	N/A	100 μs	100 μs	5598 μs	$\approx$ 0 μs

For our estimated values of hardware and software overhead, software overhead is the only one applicable for this measurement. We estimated for a baseline system call to be around 100 μs. We expect a system call to take a much longer time, compared to a process call which we estimated to be around 10 μs.

When we ran the system call measurements, we got an average value of 5493 microseconds for the getpid() system call and an average value of 5598 microseconds for the getsid() system call. Both these values were much larger than our estimated value of 100 μs as we underestimated how much longer a system call would take. Comparing to process calls, the systems calls took much longer, which was expected. Additionally, the two system calls we experimented on had similar overhead times. The standard deviation for both of these experiments were near 0, so there was little to no variability.

We believe our results to be accurate, as we experimented on two different simple system calls and got values close to each other. With a standard deviation of around 0, we believe that our resulting average values are accurate.

4.1.4 Process and Thread Creation and Execution

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Process Creation/Run	0.0009 MOpsμs $\approx$ .. μs	400 μs	500 μs	1085 μs	6387404 μs
Thread Creation/Run	0.0018 MOps/μs $\approx$ .. μs	100 μs	150 μs	115 μs	671512 μs
CyclesPerMicrosecond	..	..	..	3000 cycles	$\approx$ 0 cycles

When running process creation and execution, we saw that it took an average of 1085 microseconds with a standard deviation of 6387404 microseconds which indicated significant variability within measurements. Processes ran pretty quickly when using this environment.

When running kernel thread creation and execution, we saw that it took an average of 115 microseconds with a standard deviation of 671512 microseconds which showed large variability between measurements. The kernel thread was much faster than the creation and execution of the child process which is what we expect since threads are generally supposed to be faster than processes which is why multiple threads are created for execution rather than multiple processes.

Since process creation is a more time intensive operation than thread creation, we predicted that software overhead for process creation/run would be around 400 microseconds while thread creation/run would be around 100 microseconds and we expect software overhead for thread to be faster than process given threads are supposed to be faster. Predicted performance for process creation/execution was 500 microseconds shown in the table. Predicted performance for thread creation was in the 150 microsecond range which is also reflected in the table. We added a 50-100 microseconds to reflect hardware performance when calculating predicted performance, but it was difficult to convert machine operations to a wall clock time. Again, we expected predicted performance for threads to be better than processes. For hardware performance, we got 0.0018 Machine Operation Per Microsecond for a single thread. For hardware performance we got 0.0018/2 = 0.0009 Machine Operations Per Microsecond for a single process using the fact that there are 10 cores and 20 threads so we expect threads to be twice as fast.

When comparing predicted versus actual performance, we see that it was pretty accurate when it came to predict thread creation/run time as it was 150 μs predicted and 115 μs on average in measurement. Comparing predicted and actual performance for process creation/run time was a bigger difference where we predicted 500 μs but it was more like 1085 μs on average showing a greater discrepancy and could be due to an issue when predicting software overhead or hardware performance.

Hardware Reference: https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+E5-2690+v2+%40+3.00GHz&id=2057

4.1.5 Context Switch time between process and threads

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Context Switch (Process)	N/A	400 μs	400 μs	1370.75 ms	761.6 μs
Context Switch (Thread)	N/A	10 μs	10 μs	2.608 μs	1.18 μs

We observed for the context switch time between process took considerably long as when compared to the context switch time between kernel thread. This is expected because thread switching is very efficient and cheaper than the process switch. Process switching involves the switching of all the process resources whereas thread switching all of these steps are not present because we are switching within a process itself.

We measured the average context switch time between processes for 1000000 iterations and it came out to be 1370 milliseconds with a standard deviation of 761 microseconds. We measured the average context switch time between kernel threads for about 1000000 iterations and the performance for the same was much faster. The average for the same was observed to be 2.608 microseconds with a standard deviation of 1.18 microseconds.

For our software overhead estimation, we estimated that the steps involved for context switching in process would be mostly the system call to save the current process state into the CPU's stack, then switch the registers, increase the program counter, and then loading the page table of the next process to run. We estimated the baseline system call to be of 100 μs, the total time for the system call involved in context switch came around to be 400 μs. For the context switch in threads, the steps involved are those of saving and restoring the minimal context. Since the context switch in threads does not involve much of system call we estimated the software overhead to be around 10 μs.

We suspect measured time for the context switch for the process came to be a magnanimous (in milliseconds) because of our unusual platform. We suspect there were additional steps being taken while context switching among the process while in a virtual machine (like switching from VM to actual core). We suspect the CPU was being shared among multiple task at a time which might have been a large factor for getting the huge time. For the hardware performance for context switch, we were not able to get a reference, we believe its because of it's difficult to measure.

The references we got are performances from the actual machines. This is inherently different from our environment. So we suspect the results might not indicate what is actually happening.

4.2 Memory

4.3 Network

4.4 File System

5 References

https://gcc.gnu.org/