Systems Measurement Project - Draft of Memory Operations

Eric Hu, Sarat Sreepathy, Chitra Kharbanda

February 17, 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, RAM bandwidth for writing and reading
- Sarat: Task creation and run for both processes and threads, Access Latency for Memory, L1, and L2 cache
- Chitra: Task creation and run for context switching for both processes and threads, Page fault service time

Estimated time spent on project: 50 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 → 985 MB/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 Results

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. We ran each experiment for 10 trials. Within each trial, we ran the process (procedure call, system call, context switch, etc.) for 100,000 iterations. We computed the average overhead for each trial, and then got the final average overhead and standard deviation for all of the trials.

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 Measurement Overhead Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Measurement Overhead (Process)	N/A	10 μs	10 μs	23 μs	0 μs
Measurement Overhead (System)	N/A	10 μs	10 μs	25 μs	1 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.

3.1.3 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.

3.1.4 Process Call Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Process Call(0 Arguments)	N/A	10 μs	10 μs	11 μs	≈ 0 μs
Process Call(1 Arguments)	N/A	11 μs	11 μs	5 μs	≈ 0 μs
Process Call(2 Arguments)	N/A	12 μs	12 μs	5 μs	≈ 0 μs
Process Call(3 Arguments)	N/A	13 μs	13 μs	6 μs	≈ 0 μs
Process Call(4 Arguments)	N/A	14 μs	14 μs	7 μs	≈ 0 μs
Process Call(5 Arguments)	N/A	15 μs	15 μs	6 μs	≈ 0 μs
Process Call(6 Arguments)	N/A	16 μs	16 μs	7 μs	≈ 0 μs
Process Call(7 Arguments)	N/A	17 μs	17 μs	8 μs	≈ 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 11 microseconds. All the other process calls of 1 through 6 arguments followed the expected outcome of having increasingly more overhead for each additional argument. From these results, we can discern the increment overhead of an argument to be about 1 microsecond. 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 result of the process call with 0 arguments was surprising. These results lead us to question the stability of the virtual machine itself, as it gave us contradictory results than expected.

3.1.5 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.

3.1.6 System Call Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
System Call (getpid())	N/A	100 μs	100 μs	5474 μs	≈ 0 μs
System Call (getsid())	N/A	100 μs	100 μs	5558 μs	≈ 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 5474 microseconds for the getpid() system call and an average value of 5558 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.

3.1.7 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.

3.1.8 Task Creation Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Process Creation/Run	0.0009 MOpsμs $\approx$ .. μs	400 μs	500 μs	991 μs	0 μs
Thread Creation/Run	0.0018 MOps/μs $\approx$ .. μs	100 μs	150 μs	135 μs	69 μs
CyclesPerSecond	..	..	..	3*10^9 *cycles*	≈ 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

3.1.9 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 pipes which are inherently blocking in nature. 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, a 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.

3.1.10 Context Switch time between process and threads

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Context Switch (Process)	N/A	400 μs	400 μs	16.826 μs	0.637 μs
Context Switch (Thread)	N/A	10 μs	10 μs	16.107 μs	0.403 μs

We observed for the context switch time between process took longer 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 10 trials and in each trial we did 1000000 iterations and it came out to be 16.826 microseconds with a standard deviation of 0.637 microseconds. We measured the average context switch time between kernel threads for 100 trials and each trial had about 10000 iterations and the performance for the same was faster. The average for the same was observed to be 16.107 microseconds with a standard deviation of 0.403 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 believe our implementation for both the context switch between thread and process to be accurate. The accuracy for the thread context switch seems to be okay. The process context switch is taking a bit longer than the thread context switch which is the expected result. However, there might be a chance that the results are a skewed a bit because of the interference from other processes which are executing. This is a case which we do not have much control over.

3.2 Memory

3.2.1 Access Time

We performed experiments to find the latency for accessing into main memory and the L1 and L2 caches. The goal was to demonstrate the difference in speed of accessing from cache as supposed to memory which is much slower. For measuring the access latency of main memory, we used malloc() to create an array of the size of the L1 cache (around 32 KB) and wanted to access the array at strides of 10 (access every 10th element) to see what the average access latency to main memory. We used similar sized arrays for L1 cache and L2 cache. For L1 cache, we do not want to reallocate memory so we do not use malloc and instead access the array we created when doing the main memory measurement. We expect the accesses we did for the main memory measurement to be stored in L1 cache so accessing them would be faster. We accessed the array with strides of 10 and found average access latency for L1 cache. For L2 cache, we are going to have to make sure the access is not going to L1 cache. We do this by calling malloc() to create another array of size of the L1 cache and access each element from that newly created array and those accesses will now be what is stored in the L1 cache. When we then access the previous array created for the memory access, those accesses will now be stored in the L2 cache and we access them in strides of 10 and find the average latency of L2 cache.

For the measurements for latency, we measured them in cycles using rdtsc and converted them to nanoseconds so the average access latency is measured in nanoseconds. Nanoseconds were the appropriate unit since these accesses are expected to be quite fast especially when we refer to the L1 and L2 caches. We did not report standard deviation in the results because we are relying on caching so running iterations would affect what we would expect in L1 or L2 cache and stuck primarily to an average of access latency.

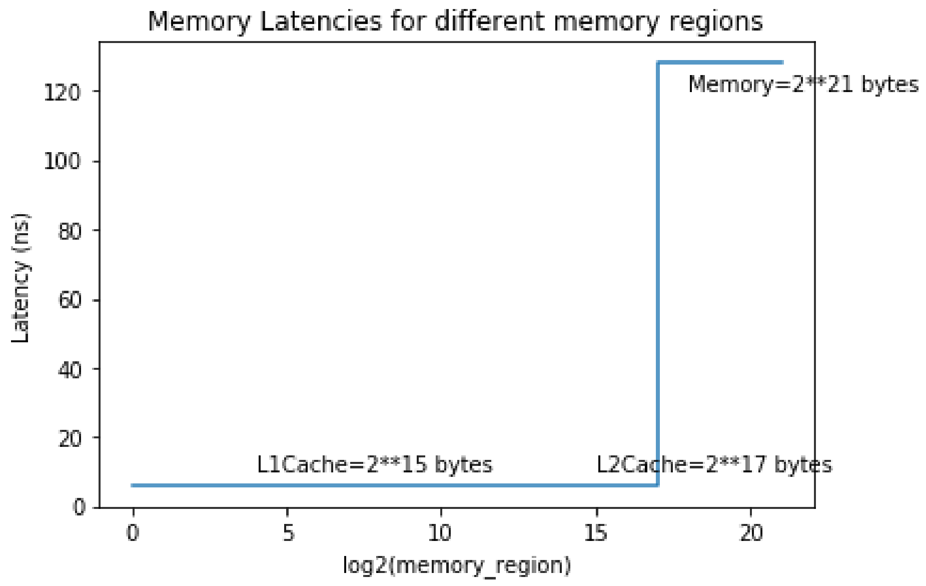
3.2.2 Access Time Results

Operation	Hardware	Software Overhead	Predicted	Avg
Main Memory Access	80 ns	20 ns	100 ns	128 ns
L1 Cache Access	1 ns	4 ns	5 ns	6 ns
L2 Cache Access	4 ns	10 ns	14 ns	6 ns

CyclesPerSecond: 3000MHz(1 second) Reference for Expected Performance: https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html

Analysis: We saw that the performance for main memory was much slower than L1 and L2 cache which lines up with the expectation that main memory is slower than L1 and L2 cache. We saw that L1 cache latency was around 6 ns which is around what we expected at 5 ns with 1 ns of estimated hardware performance and 4 ns of software overhead. We saw with L2 cache latency an average of 6 ns with predicted latency at 14 ns with 4 ns of estimated hardware performance and 10 ns of software overhead. The average compared to predicted L2 cache latency was much faster which indicated there may have been more L1 cache hits than expected and therefore made the access faster. For main memory access, we had 128 ns on average with a predicted latency of 100 ns with estimated hardware performance of 80 ns and software overhead at 20 ns. The actual and predicted performance for main memory access was similar and not the discrepancy we saw with the L2 cache access.

3.2.3 Graph of Memory Regions and Latencies



3.2.4 RAM Bandwidth

We performed experiments on finding the bandwith for both writing and reading from memory. We used malloc() to create an array of size 1000, which is equivalent to 1000 bytes (0.001 MB) of memory. We used **rdtsc** to measure the number of cycles it took to write and read from the 1000 bytes of memory. We ran through this trial 1000 times, so that we could get the average and standard deviation of the bandwidth. While we are initially measuring in bytes and number of cycles, we convert it to MB/s for our final results.

We utilized several optimization techniques. Before performing the measurement, we cleared the memory cache so that we may avoid cache prefetching. We used the system call "$echo3 > /proc/sys/vm/drop\_caches$" to achieve this effect. We also used loop unrolling when writing and reading from memory so that the loop operation overhead will not have any effect on our memory measurements.

3.2.5 RAM Bandwidth Results

Operation	Hardware	Software Overhead	Predicted	Avg	Deviation
Writing Bandwidth	985 MB/s	500 MB/s	485 MB/s	375.38 MB/s	0 MB/s
Reading Bandwidth	985 MB/s	100 MB/s	885 MB/s	1033.06 MB/s	0 MB/s

For our predicted values, we estimated that the hardware performance would be 985 MB/s, which is the same as the machine's 985 MB/s bus bandwidth. Software overhead would be present, as we are writing and reading from memory, where we estimated that writing would have a greater overhead compared to reading. We gave writing to memory a software overhead of 500 MB/s and to reading a smaller overhead of 100 MB/s. As such, our final predicted values for writing is 500 MB/s and for reading is 800 MB/s.

We saw as expected that writing performed much slower, at only 375 MB/s, but were surprised to see that reading had performed slightly above expected, at 1033 MB/s. This performance was even above the machine's defined bus bandwidth of 985 MB/s.

We believe our experiment to be successful, as our results are near our virtual machine's memory bus bandwidth description of 985 MB/s. It was also expected that writing to memory would be slower than reading from memory.

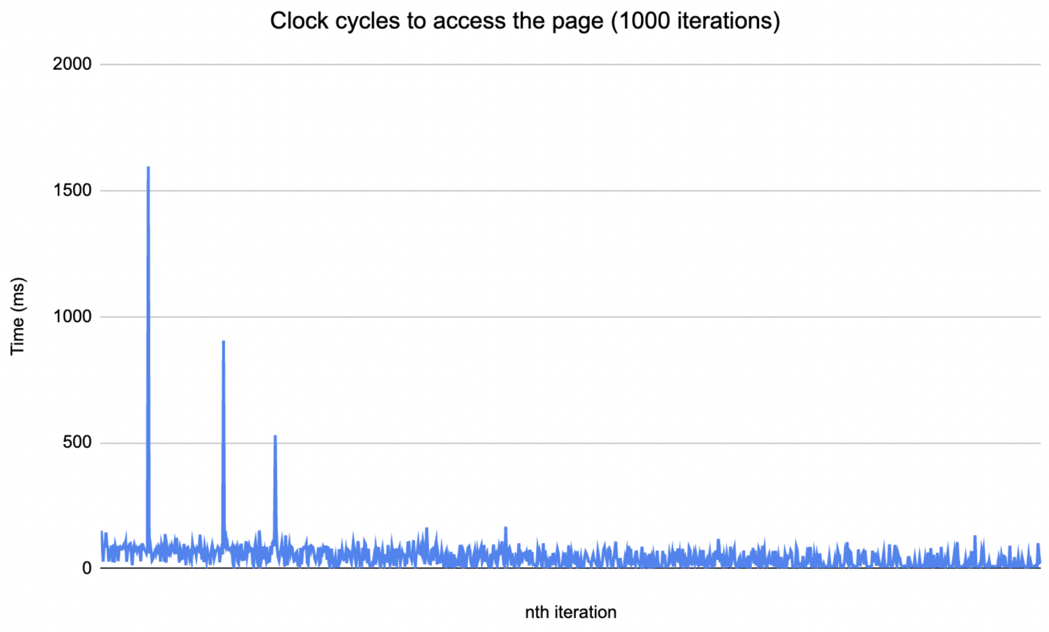
3.2.6 Page Fault Service Time

We performed experiments on finding the page fault service time. The total main memory of the system we are working on is 2033016 kB. So the general idea to ensure that page fault occurs was to randomly access a file whose size was bigger than the capacity of main memory so that the entire file could not fit into the memory hence forcing the occurrence of page fault.

As the first step in the experiment, we created a file whose size was larger than the main memory capacity. We created a text file whose size was 2500000 kB. Now after creating a file, the residues of the file content can still be left in the main memory and the cache. To prevent our test from accessing this data, we clean the cache before accessing the file. We make use of "$echo3 > /proc/sys/vm/drop\_caches$" system call to clear out the cache. Now for accessing the file, we make use of the $mmap()$ system call which creates a new mapping in the virtual address space of the calling process. Then we randomly access the pages (the continuous memory allocated of size 4096 kB) of the mapped file for 1000 time. For every access of a page, we measure cpu clock cycle and store the result.

3.2.7 Page Fault Service Time Results

We performed the experiments where we accessed the page and timed every access. We plotted the graph for the access time of each of the page for a 1000 iterations. Attached below is the graph depicting the time taken to access a page (4096 kB of data) in milliseconds for every iteration.



Operation	Hardware	Software Overhead	Predicted	Avg
Page Fault Service Time	N/A	100 ms	500 ms	1010.43 ms

The graph shows the three peaks which essentially denotes that the page fault occurred. The average time for servicing the page fault came out to be 1010.43 milliseconds. Dividing this value by the size of a page gives us the 246 nanoseconds/byte. This is basically almost twice (≈ 1.92) the time of accessing one byte from the main memory.

The system which we are measuring the performance for is the ubuntu running on virtual machine. We were not able to find the base hardware metrics (the latency for the disk) for the same. For estimating the software overhead, the main contributors to this would essentially be the overhead involving the system calls of accessing the page table, then the trap to os when the requested page is not found, then for finding the page in the backing store, then for bringing the page to the physical memory and then updating the page table. For all these steps, we assume the software overhead to be around 100 ms. Since the transfer of data between the disk to the memory and then while updating the page table also takes time, we predicted the page fault service time to be around 500 ms.

We believe our experiments to be successful. The peaks obtained in the graph are clearly depicting the access time for when the page fault occurred. Since it is expected for the page fault to occur at the start of the experiment (the reason being that we flush out the cache before actually accessing the file) the graph is showing the similar expected behaviour. The final result that accessing one byte of data takes twice as time during the page fault versus when directly accessing it from the main memory also makes sense.

3.3 Network

3.4 File System

4 References

https://gcc.gnu.org/