CSC 501 - Operating Systems

Project 1 - Performance Measurement

Name: Parth Bhogate **Unity ID:** pbhogat
Name: Radhika Sakhare **Unity ID:** rssakhar

Introduction

The performance of a computer system depends upon the hardware, and its interaction with the operating system being used to manage the execution of applications. The aim of this project is to measure the baseline performance of the hardware as well as the overhead on account of the operating system. Both CPU performance/software overheads and memory system performance is measured by performing suitable experiments. The machine used for analyzing the performance was a Dell XPS laptop with a Intel i5 processor and 6 GB of RAM. The operating system used was Ubuntu (the laptop has dual Windows and Linux boot).
This project was completed in a group of two members. Contributions of each team member were as follows:
Parth – RDTSC, Procedure call, Task creation, Page fault
Radhika – System call, Context switching, RAM latency, RAM bandwidth

All source code used in this project was written in C. The compiler used was gcc 5.4 (Ubuntu 5.4.0-6ubuntu1~16.04.2). In order to get an unbiased estimation of the hardware performance, it is important to use a unoptimized assembly code. To prevent the compiler from performing aggressive optimizations, the following flags were used to compile all source files -

```
gcc -O0 -pthread
```

The estimated amount of time spent on the project including designing tests, implementing the tests in code, gathering and presenting results – 45 hours total.

Machine Description

All the experiments were performed on a Dell XPS 15z laptop with the following configuration:

1. **Processor Information**

Command used to obtain CPU information -> `/proc/cpuinfo` and `lscpu`

Processor Model: Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz, Quad-Core Processor
Architecture: x86_64
Byte Order: Little Endian
Cycle Frequency: 2300 MHz, upto 2900 MHz with Turbo Boost
L1 data cache: 32K
L1 instruction cache: 32K
L2 cache: 256K
L3 cache: 3072K
DRAM: 6 GB

2. **Memory bus**

Command used to get the information -> `sudo dmidecode --type memory`

```
Total Width: 64 bits
 Data Width: 64 bits
 Size: 4096 MB
 Type: DDR3
 Type Detail: Synchronous
 Speed: 1333 MHz
 Manufacturer: Hynix/Hyundai
 Bandwidth = 10667 MBps
```

3. **I/O bus**

Command to get the information -> `sudo hdparm -Tt /dev/sda`

```
Timing cached reads:   7350 MB in  2.00 seconds = 3676.55 MB/sec
Timing buffered disk reads: 324 MB in  3.01 seconds = 107.79 MB/sec
```

4. **RAM size**

Command to get the information -> `cat /proc/meminfo`

RAM (Main Memory) size = 6022252 kB = 5.743 GB

5. **Disk Information: capacity, RPM, controller cache size**

Command to get the information -> `hdparm -I /dev/sda`

```
Logical           max    current
cylinders     16383   65535
heads               16          1
sectors/track    63      63
CHS current addressable sectors:       4128705
LBA user addressable sectors:  268435455
LBA48  user addressable sectors:  976773168
Logical  Sector size:                512 bytes
Physical Sector size:                4096 bytes
Logical Sector-0 offset:                     0 bytes
device size with M = 1024*1024:      476940 MBytes
device size with M = 1000*1000:      500107 MBytes (500 GB)
cache/buffer size = 16384 KBytes
Nominal Media Rotation Rate: 7200
Queue depth: 32
R/W multiple sector transfer: Max = 16   Current = 16
```

6. **Operating system**

Command used to get the information -> `uname -a`

Operating System: Ubuntu Linux
Version: Ubuntu 16.04 LTS

In order to obtain repeatable and accurate measurements, the following system configuration was setup before running all the experiments to avoid any variations in measurements:

1. **Only a single core was kept active to run the test programs -** Since the base hardware has 4 physical CPU cores, the kernel may choose to run the same process or threads of the same process on different cores during the program execution. To prevent this, 3 cores were disabled (cpu1-cpu3), and only cpu0 was used to run the program code. CPU cores were disabled by writing NULL to cpu'x'/online file -
   ```
   echo 0 > /sys/devices/system/node/node0/cpu'x'/online
   ```
   where, 'x' = 1, 2, 3, denotes each CPU core.
   Only cpu0 was kept online.

2. **Dynamic frequency scaling was disabled -** The Intel architecture employs dynamic frequency and voltage to reduce power consumption if the core is in a low activity phase. To measure time intervals accurately, the frequency should be kept constant so that the cycle count read from the TSC can be used to obtain real time. Our platform does not provide ondemand governor. We have only performance and powersave governors. Hence, cpufreq-selector didn't work to set to a particular frequency. We tried to disable dynamic frequency scaling by disabling 'speedstep' in BIOS.

3. **Frequency Turbo Boost was disabled -** The Intel i5 core has the facility to overclock the CPU by running it at a frequency higher than the rated baseline. Turbo boost was disabled by writing '1' to the /cpu/intel_pstate/no_turbo file -
   ```
   echo "1" | sudo tee /sys/devices/system/cpu/intel_pstate/no_turbo 1
   ```

4. **Program priority was kept high using 'nice' utility -** The 'nice' command allows us to manually set the priority level of a process. The priority of the test program was set to high priority to prevent preemption of the process by other active processes during execution. We used the nice() system call in code to raise the priority of the process and passed it a value of -20 as argument

5. **Constant TSC counter was used to obtain precise timing -** The constant TSC feature in Intel x86 architecture allows us to measure the number of elapsed cycles independent of the core clock frequency. The cycle time of the TSC was measured by counting the number of clock cycles in a fixed amount of time, using the sleep() system call. The cycle time of the TSC clock was found to be 0.435 ns (2.3 Ghz frequency).

Experiments

I. CPU, Scheduling, and OS Services

1. **Measurement overhead:**

The 'rdtsc' instruction in the x86 ISA was the mechanism used to measure the execution time of code. This instruction reads the 64-bit Time Stamp Counter (TSC) register present in all x86 architectures. The register holds the number of clock cycles since reset. Since the Intel i5 core executes instructions out-of-order, a serializing instruction such as CPUID needs to be used before reading the time stamp.

Overhead of using RDTSC: To measure the overhead of using RDTSC, the register was read twice without including any other code between the two reads. Multiple runs of the difference in cycles between consecutive TSC reads were performed to get the average value.

Average number of overhead cycles of RDTSC: 52
Standard deviation in the per iteration cycle values of RDTSC: 0

We can estimate the hardware performance by counting the instructions in that operation, estimating the CPI and cycle time.

Cycle time is 0.43 ns. There are four instructions and assuming CPI of 1, we get 2*4*0.43 = 3.44ns. Overhead can also be because of latency to fetch instructions from main memory. As given in specs memory bandwidth is around 10 GB/s. To fetch 4 instructions which has 4*64 bytes we get 0.25 ns. So we can predict the hardware overhead to be 0.25+3.44 ~ 4ns. The instruction cpuid is a fence instruction which can take a lot of time to drain the pipeline. Hence, this increases overhead. This might be waiting for high latency memory operations to be completed which can increase the overhead by around 20 ns.

Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time (Cycles * Cycle Time)
4 ns	10 ns	14 ns	**52*0.435 = 22.62 ns**

The difference in reading is because we are running in a loop to measure the overhead of RDTSC instruction, which can use spatial and temporal locality of cache. Also cache and memory system latency can be lesser.

Overhead of using a ‘for’ loop: In the series of experiments done in this project, a ‘for’ loop of 10000 iterations will be very commonly used to execute a particular test piece of code multiple times to find the average execution latency. The overhead of using the ‘for’ loop is measured by reading the cycle counts before and after an empty ‘for’ loop.

Average number of overhead cycles of ‘for’ loop:
Standard deviation in the per iteration cycle values of ‘for’ loop: 347.14

Assuming 2 instructions in for loop with 10000 iterations we get hardware performance = 2*1000*1CPI*0.43ns = 34 us

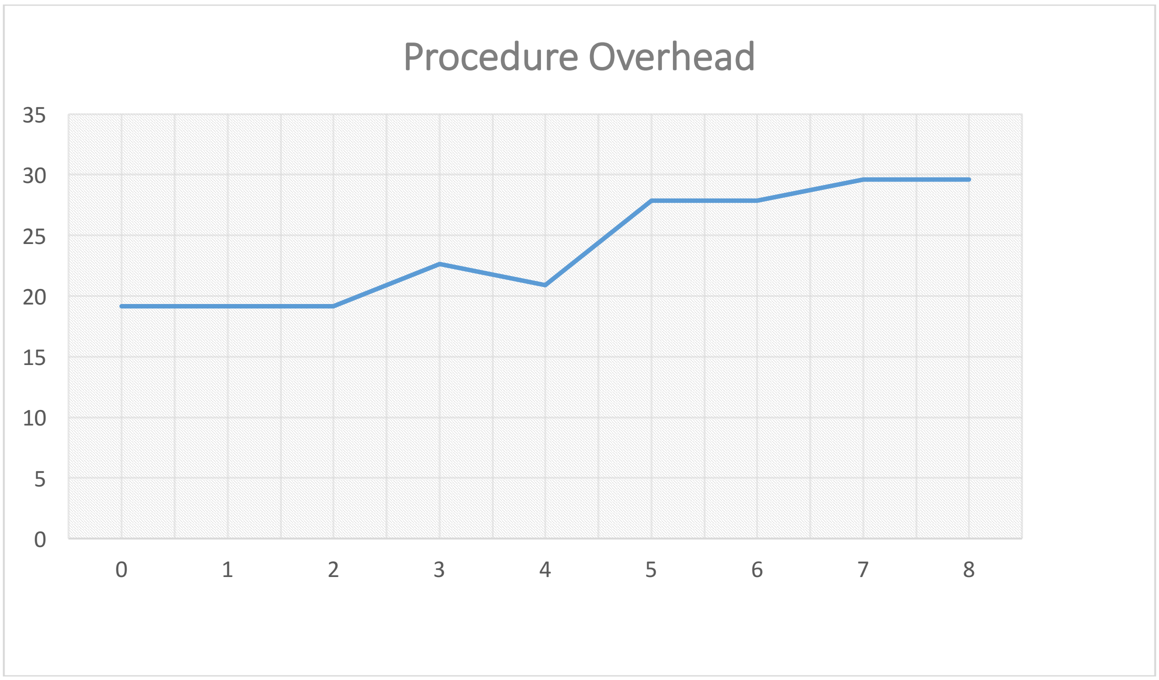
Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time
34 us	10 ns	34.1 us	**76300*0.435=33.190 us**

2. **Procedure call overhead:**
Procedure call overhead is measured by calling user-level functions with an empty body with a variable number of arguments. The number of arguments is varied from 0-8 (arguments are of

the type ‘int’). The rationale behind varying the number of arguments is that a certain number of arguments are passed via registers, while arguments above than this fixed number of arguments are passed via memory.

Hardware overhead by assuming 4 instructions = 4*1CPI*0.43 = 1.72 ns
Here, software overhead is because of pushing and popping registers from stack which preserve the state of original program. This can add around 2 instructions.

Number of arguments	Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time (Cycles * Cycle Time)
0	1.72 ns	15 ns	16.72 ns	**44*0.435 = 19.14 ns**
1	1.72 ns	15 ns	16.72 ns	**44*0.435 = 19.14 ns**
2	1.72 ns	15 ns	16.72 ns	**44*0.435 = 19.14 ns**
3	1.72 ns	15 ns	16.72 ns	**52*0.435 = 22.62 ns**
4	1.72 ns	15 ns	16.72 ns	**48*0.435 = 20.88 ns**
5	2.58 ns	15 ns	17.58 ns	**64*0.435 = 27.84 ns**
6	2.58 ns	15 ns	17.58 ns	**64*0.435 = 27.84 ns**
7	2.58 ns	15 ns	17.58 ns	**68*0.435 = 29.58 ns**
8	2.58 ns	15 ns	17.58 ns	**68*0.435 = 29.58 ns**



The above graph shows number of arguments on x-axis and cycle latency on y-axis.

Analyzing the latency results for procedure calls with variable number of input arguments, we observe that there is slight increase in the call overhead as the number of arguments is increased. The argument values to a function call are passed by storing them in fixed registers according to the procedure call standard. When arguments to a function exceed a particular number, the arguments need to be passed through memory. We observe a jump in the call overhead when the number of arguments increases from 4 to 5. So, we can predict that upto 4 registers values doesn't need stack access to preserve the data. But as soon as we cross 4 registers, we need to preserve old state of registers on stack and jump to the function call.

3. **System call overhead:**

A system call is a special procedure call used by a user-level program to access kernel-level functionality. System calls have the special 'trap' instruction in their body, which changes the processor mode to 'kernel' mode. To service a system call, the state of the program is saved on the kernel stack before executing the trap handler. System calls are expected to have a higher overhead as compared to procedure calls because of the added overhead of saving the process state, trapping into the kernel, executing the particular system call and then returning back to the user process.
A system call is expected to be significantly more expensive than a procedure call (provided that both perform very little actual computation). We used the gettimeofday() system call to measure the syscall overhead. The system call overhead is not measured in a loop since the result could be cached after the first call. Therefore, the average is calculated by running the program multiple times with each run calling the system call once.

Average number of system call overhead cycles: 2074
Standard deviation in the system call overhead: 153.43

Because of the additional trap to OS, execute the OS instructions and return back to user space there is software overhead.

Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time (Cycles * Cycle Time)
10 ns	700 ns	710 ns	**2074*0.435 = 902.19 ns**

Comparing the overhead of a user program procedure call and a system call, we observe that the syscall overhead is much greater than a procedure call overhead. This observation can be explained by the fact that a system call involves the following actions :

- A context switch
- A trap to a specific location in the interrupt vector
- Control passes to a service routine, which runs in 'monitor' mode
- The monitor determines what system call has occurred
- Monitor verifies that the parameters passed are correct and legal

A procedure just branches to the address of called function and returns to the callee program after finishing execution.

4. **Process and Kernel Thread creation overhead:**

Process is the fundamental abstraction in Unix-based systems to denote a program in execution. The fork()syscall is used in Linux to create a new child process. The child process has a replica of the code and data segment of the parent process, but has its own address space and Process ID (PID). To measure the overhead of creating a process, a child process is spawned by calling fork(), while the parent process is halted by calling wait(). The delay incurred to return to executing the parent process is the process creation overhead.

Average number of process creation overhead cycles: 241133
Standard deviation in the process creation overhead cycles: 4421.69

For process, OS has to setup page table, call kernel code for that, execute that code and the return back. Hence it has lot of overhead which is in microseconds.

Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time (Cycles * Cycle Time)
10 us	80 us	90 us	**241133*0.435=104.89us**

Kernel threads have the same code and data segment as the parent process, but each thread maintains its own stack and executes independently of other threads. The kernel threads share the same address space as the parent process. The pthread_create() API in the <pthread.h> C standard library implements kernel threads.

Average number of thread creation overhead cycles: 27457

Standard deviation in the thread creation overhead: 507.42

For thread creation, we need to save user space information, call kernel code for allocating new stack and set up thread space and then return back. This incurs lot of software head but lesser than process as we do not have to set up a new page.

Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time
25 ns	10 us	10.25 us	**27457*0.435=11.94 us**

We observe that the process creation overhead is an order of magnitude greater than thread creation thread. Creating a process involves trapping into the kernel by the clone() syscall, and then setting up the address space for the new process. The code and data for the new process is also copied into memory before executing the process code.
Since threads share the same address space, (they only have separate stacks), thread creation does not involve setting up a new address space and page table.etc. Thus, the creation overhead for threads is lesser than a process, as our experimental results demonstrate.

5. **Process and thread context switch overhead:**
A process context switch occurs when the processor stops executing the process being currently executed and starts running another process. Switching context involves saving the register and architectural state on the process stack, and changing the hardware stack pointer to point to the process being switched in so that its state can be restored.

Average number of process context switch overhead cycles: 4014
Standard deviation in the process switch overhead: 46.07

For context switching of a process, we need to save the process information and call kernel to switch to another process. Kernel also has scheduler code which decides which process will execute and get CPU. Hence, overhead will be more but it will be much lesser than process creation overhead.

Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time
25 ns	1100 ns	1125 ns	**4014*0.435 = 1757.83 ns**

Kernel thread switch implies that one kernel thread ‘yields’ to allow the execution of some other kernel thread. The context switching process for threads involves changing the stack pointer (since each thread maintains its own stack) and copying the architectural register state from the stack. Kernel threads are implemented by using the ‘pthread’ library interface.
Since Linux implements processes and threads using the same clone() system call, the only difference between threads and process is the amount of data shared among two processes or two

threads. The switch overhead for threads is expected to be lesser than processes since threads share more data than processes.

Average number of thread context switch overhead cycles: 3304
Standard deviation in the thread switch overhead: 81.98

For thread context switching, we need to save running thread information in user space, then invoke kernel to switch to another thread. Now, kernel runs its code to switch to a new thread and then returns back to old thread. This incurs a lot of overhead but less than thread creation.

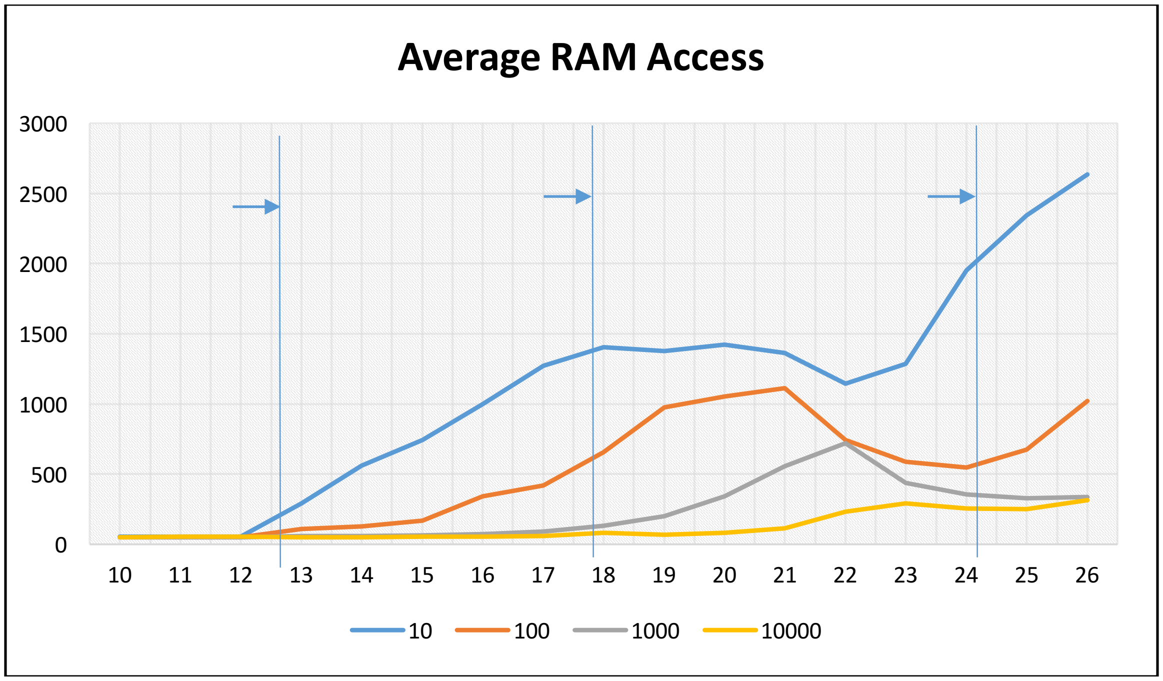
Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time
25 ns	900 ns	925 ns	**3304*0.435 = 1437.24ns**

Our experiments suggest that the context switch overhead from one process to another is greater than the switch overhead from one kernel thread to another. Switching from one process to another involves saving the execution state of one process onto stack, changing the stack pointer to point to the other process' stack, and finally loading the state of the other process.
Thread context switch involves similar steps to a process switch, the only difference being that thread share more common data than distinct processes. The observed context switch overhead for threads is slightly lower than processes.

II. Memory

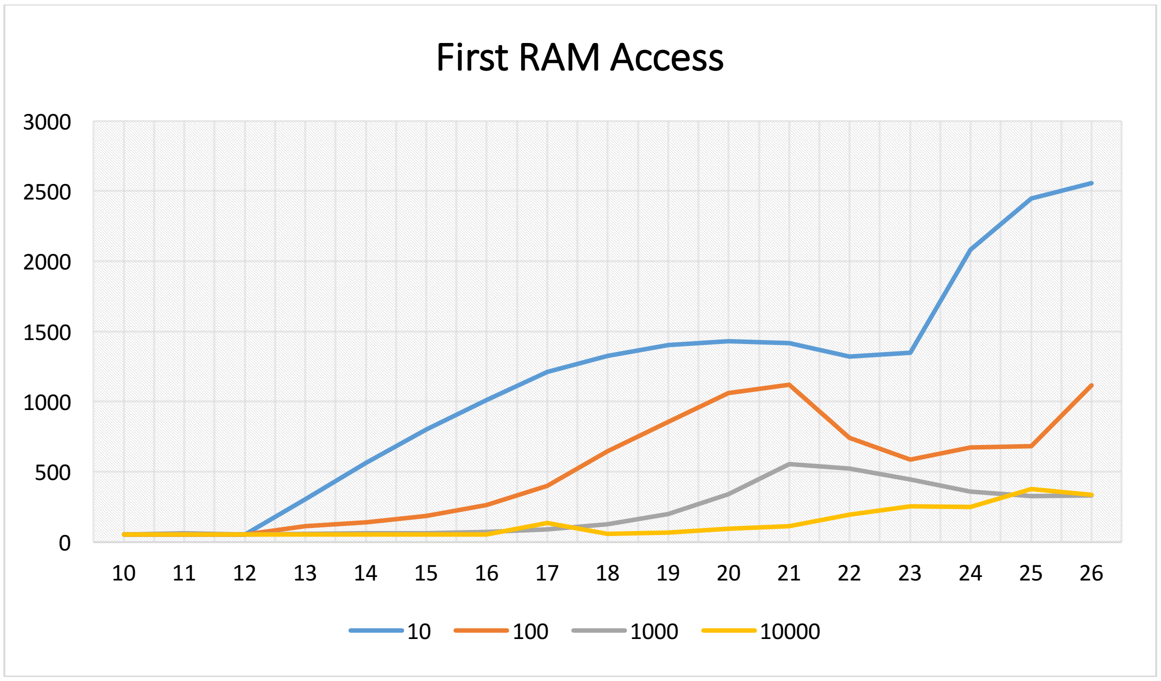
1. **RAM access time:**
   The memory hierarchy in modern systems is usually built of multiple levels of fast small caches located near the processor and a large slow RAM memory. The access latency of each level in the hierarchy progressively increases as we move down the hierarchy.
   To measure the access of each memory hierarchy level, we created an array on heap. Then the array was accessed randomly for various random indexes from 10 to 10000. We varied the array size from 2^10 to 2^26 bytes and took measurements. The plot is as below.

   The x-axis is array size in bytes. The y-axis is average latency of accessing 100 random elements.



We also tried another experiment in which we varied array size and allocated memory for that array on heap. Then the latency to access first random element was noted. This gives a similar graph to lmbench paper. It is plotted below.

The x-axis is log of array size in bytes and y-axis is latency in cycles to access first random element.



Further, we tried varying array size and changing stride for each array size. But because of hardware prefetcher, readings were not as per expected. This could be because on the machine, there was no option to disable the pre-fetcher. Hence, we decided to go with random index accesses as described above.
We can see clear transition from L1 to L2 cache with increase in memory access time. Also, transition from L2 to L3 as well as from L3 to DRAM is clear with further increase in the access time/latency. However, the latency decreases abruptly in between because of cache hits and probably because the rand() function generates similar values.
From the graph, L1 size can be predicted to be approx 16KB, L2 size and L3 size can be predicted to be approx 128 KB and 8 MB respectively. However, it is just the fair estimate of the sizes and differ a bit from actual sizes due to other limiting factors like prefetching, preemption, voltage frequency scaling, etc.
Also from graph, individual integer access to L1 cache is around 100 cycles, to L2 is around 800 cycles, L3 is around 1500 cycles and memory is 2500 cycles.

2. **RAM bandwidth:**

The RAM bandwidth refers to the overall data rate that is supported by the main memory (RAM) on the system. To measure the RAM bandwidth, a large amount of data is read from the RAM in a program and the average latency required to bring in data is calculated. The RAM bandwidth ignores effects of cache hits and misses, and just reports the overall observed throughput as seen by the user program.
It was implemented by taking an array of a large size and copying the data from the array to another array using the memcpy() call. To test the write bandwidth, random data was written to all the elements in the array using the memset() method. The RAM bandwidth was calculated by dividing the total array size by the time taken to read/write the array. Different array sizes were used to compute the average bandwidth.

Array Size	Time taken to read array	Time taken to write array	RAM read bandwidth	RAM write bandwidth
2^(29) bytes	106451798*0.435 ns	92534365*0.435 ns	9.714 GB/s	12.422 GB/s
2^(30) bytes	223483492*0.435 ns	182624678*0.435 ns	9.361 GB/s	12.588 GB/s
2^(31) bytes	468785436*0.435 ns	368476375*0.435 ns	9.459 GB/s	12.478 GB/s

3. **Page fault service time:**

Page fault refers to the condition when the page requested by the CPU is present on the disk instead of main memory. To service a page fault, the page is brought in from disk swap partition to main memory by the operating system.
To measure the latency of a page fault, the mmap() system call was used. The mmap() procedure maps a file from disk to a memory region in the DRAM. Once the file is brought in the memory, it can be accessed like any other data using address pointers. Thus, the extra I/O latency of using the file system can be avoided. This can be used to emulate a page fault since the file is brought into memory only it is first accessed by the user program. Thus, by creating a file on disk with size equal to the system page size, the latency of bringing in an entire page from disk to memory can be measured. The time to access first byte of the page was measured as it gives the exact page fault measurement. But this measurement may not be accurate if the processor system uses Critical Word First scheme.

Average number of cycles to bring an entire page from disk to memory: 2702
Standard deviation in the page fault service cycles: 291.32

The overhead of pagefault is more than memory latency. The disk bandwidth is 107 MB/s. For accessing one byte the latency is 4/BW = 3738 ns

Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time
3738 ns	10 ns	3748 ns	**2702*0.435 = 1175.37 ns**

This overhead can be less because of Critical Word First optimization done by processor.

Summary of Results

The following table gives an overview of all the experiments performed as a part of this project, and the obtained experimental overhead measurements -

Sr. No.	Operation	Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time
1	RDTSC	4 ns	10 ns	14 ns	22.62 ns
2	For loop	34 us	10 ns	34.1 us	
3	Procedure call (8 args)	2.58 ns	15 ns	17.58 ns	22.61 ns
4	System Call	10 ns	700 ns	710 ns	902.19 ns
5	Process Creation	10 us	80 us	90 us	104.89 us
6	Thread Creation	25 ns	10 us	10.25 us	11.94 us
7	Process Context Switch	25 ns	1100 ns	1125 ns	1757.83 ns
8	Thread Context Switch	25 ns	900 ns	925 ns	1437.24 ns
9	L1 Access Latency	-	-	-	43.5 ns
10	L2 Access Latency	-	-	-	0.348 us

11	Main Memory Latency	-	-	-	1.087 us
12	RAM Read Bandwidth	10 GB/s	-	10 GB/s	9.512 GB/s
13	RAM Write Bandwidth	10 GB/s	-	10 GB/s	12.496 GB/s
14	Page Fault Latency	3738 ns	10 ns	3748 ns	1175.37 ns

Conclusion

The operating systems provides an interface between the system hardware and the user application programs. Since the user programs are run in a virtualized environment, extra overheads are added because of the kernel process management functionality. In this project, we have looked at the overhead due to common operating system functionality such as system calls, process creation and context switching, and kernel thread creation and context switching. We have also analyzed the memory hierarchy latencies for cache and main memory accesses. The page fault virtual memory operation is also analyzed.

During the course of this project, we gained a good understanding of the low-level actions in various operating system tasks, and also the causes for overheads added by various operations. The measurements also helped to validate and to correct some of our initial performance estimates obtained from hardware specification.