PERFORMANCE MEASUREMENT

Jalandhar Singh, jsingh8
Rohit Nambisan, rnambis

Table of Contents

1. INTRODUCTION

The goal of the project is to measure the performance of various hardware components and their influence on the operating system services. The project was completed in a group of two. The results of this project came out of numerous discussions held in the group and then dividing the tasks among the group members. The systems used for the projects were of identical specs and the results were similar in both the machines. The compiler used was version 5.4.0. The most amount of time was invested in understanding the various concepts and working of the operating system which took a week. The development of each experiment took approximately a week.

1.1. MACHINE DESCRIPTION

Device: Lenovo ThinkPad Yoga 14
Processor: Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz, 4 cores
L1 cache: 64 KB
L2 cache: 256 KB
L3 cache: 3072K
Memory bus: 1600 MHz (speed), 64 bit(width), DDR3**.**
I/O bus: SATA (AHCI Version 1.30 Supported)
RAM size: 8GB
Disk: SSD (Type), 256 GB (Size)
OS: Ubuntu 16.04

1.2. Testing Environment Setup

In order to accurately benchmarking or profile the system, we need to properly setup the testing environment before we run the program and collect the data.

Given the fact that we are running on a multi-core system, and running a number of other processes at the same time under some specific OS scheduling policy, how to rule out those interferences from multi-core, hardware multi- threading, undesired process context switches becomes our preparation work before we dive into the actual experiments.

1.2.1 Single Core Environment Setup

Currently the system has a 4 CPUs. I got this information by checking the content of "/proc/cpuinfo" file on Linux machine. To get the time or CPU cycles calculation correctly, we need to disable all CPU's except one. This is done by updating the grub file.

The steps are given below,

- Run command, "gksu gedit /etc/default/grub".
- Update GRUB_CMDLINE_LINUX_DEFAULT from "quiet splash" to "quiet splash **maxcpus=1**".
- Then, we need to update the grub by running "sudo update-grub", and then reboot.

- The /proc/cpuinfo showed,

```
processor    : 0
vendor_id    : GenuineIntel
cpu family   : 6
model        : 78
model name   : Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
stepping     : 3
microcode    : 0x82
cpu MHz      : 499.968
cache size   : 3072 KB
physical id  : 0
siblings     : 1
core id      : 0
cpu cores    : 1
apicid       : 0
initial apicid    : 0
fpu          : yes
fpu_exception     : yes
cpuid level  : 22
wp           : yes
flags        : fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse
sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
art arch_perfmon pebs bts rep_good nopl xtopology
bogomips     : 4799.96
clflush size: 64
cache_alignment   : 64
address sizes     : 39 bits physical, 48 bits virtual
power management:
```

- To confirm whether cpu0 was the only online CPU, we also ran “lscpu” command, the output obtained on the terminal is provided below,

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0
Off-line CPU(s) list:  1-3
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 78
Stepping:              3
```

```
CPU MHz:               498.093
BogoMIPS:              4799.96
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              3072K
NUMA node0 CPU(s):     0
```

- Next to understand the machine architecture, how each cache is coupled with particular CPU's, we run "lstopo" command,

```
Machine (7741MB total) + NUMANode L#0 (P#0 7741MB) + Package
L#0 + L3 L#0 (3072KB) + L2 L#0 (256KB) + L1d L#0 (32KB) + L1i
L#0 (32KB) + Core L#0 + PU L#0 (P#0)
  HostBridge
    PCI 00:02.0 (VGA)
    PCI 00:17.0 (SATA)
      Block(Disk) "sda"
    PCIBridge
      PCI 03:00.0 (Network)
        Net "wlan0"
    PCIBridge
      PCI 06:00.0 (3D)
    PCI 00:1f.6 (Ethernet)
      Net "eth0"
```

1.3 Preventing Multi-Process Context Switches

Even I have disable all the CPUs except one but I am trying to be more generic in the report.

To make sure context-switching processes are located on the same processor:

1. Bind a process to a particular processor using "sched_setaffinity()" API.

```
For example:
        cpu_set_t set;

        CPU_ZERO( &set );

       // Assign the CPU having id : 0
       CPU_SET( 0, &set );

       sched_setaffinity( getpid(), sizeof( cpu_set_t ), &set )
```

2. To get the maximum priority value (sched_get_priority_max API) that can be used with the scheduling algorithm identified by policy (SCHED_FIFO).

```
For example:

     struct sched_param prio_param;
     int prio_max;

     if( (prio_max = sched_get_priority_max(SCHED_FIFO)) < 0 )
     {
          perror("sched_get_priority_max");
     }

     prio_param.sched_priority = prio_max;
     if(  sched_setscheduler(getpid(),SCHED_FIFO,&prio_param)  <
0 )
     {
         perror("sched_setscheduler");
         exit(EXIT_FAILURE);
     }
```

Why is it important that context-switching processes should remain on the same process?

We are reading Time Stamp counter value for the CPU cycles calculation in the code. Basically we read the counter before & after starting the test and then we take the difference of the two values to find the overhead. So there might be possibility that process moved to another CPU because of context switch and we got one time stamp counter value from one CPU and one from another. Therefore it is always better to pin the process to one CPU and assign max priority because there is no promise that the timestamp counters of multiple CPUs on a single motherboard will be synchronized always.

1.4 Timestamp Reading Mechanism

There are lot of different ways to get the time or CPU cycles like gettimeofday(), clock() etc. but I have chosen RDTSC to measure the overhead.

CPUs have a timestamp counter to keep track of every cycle that occurs on the CPU. The processor has included a per core timestamp register that stores the value of the timestamp counter and that can be accessed by the RDTSC and RDTSCP assembly instructions.

When running a Linux OS, the developer can check if his CPU supports the RDTSCP instruction by looking at the flags field of "/proc/cpuinfo"; if rdtscp is one of the flags, then it is supported.

```
jsingh@jsingh-ThinkPad-Yoga-14:~$ cat /proc/cpuinfo | grep rdtsc
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
pbe syscall nx pdpe1gb rdtscp lm constant_tsc ida arat pln pts hwp
hwp_notify hwp_act_window hwp_epp
```

Since Processor both support **out-of-order** execution, where instructions are not necessarily performed in the order they appear in the source code. This can be a serious issue when using the RDTSC instruction, because it could potentially be executed before or after its location in the source code, giving a misleading cycle count.

In order to keep the RDTSC instruction from being performed out-of-order, a serializing instruction is required. A serializing instruction will force every preceding instruction in the code to complete before allowing the program to continue. One such instruction is the CPUID instruction, which is normally used to identify the processor on which the program is being run. For the purposes of this paper, the CPUID instruction will only be used to force the n-order execution of the RDTSC instruction.

```
static inline uint64_t rdtsc(void) {
        uint32_t lo, hi;
        __asm__ __volatile__("xor %%eax, %%eax;" "cpuid;" "rdtsc;":
"=a" (lo), "=d" (hi));
        return (((uint64_t)hi << 32) | lo);
}
```

References:

1. https://en.wikipedia.org/wiki/Rdtsc
2. http://www.intel.com/content/www/us/en/embedded/training/ia-32-ia-64-benchmark-code-execution-paper.html

3. Experiments and Operations

3.1 CPU, Scheduling, and OS Services

3.1.1 Measurement Overhead:

Measurement overhead is usually an additional expense occurring in addition to normal cost while doing measurement.

Here we are going to consider the two scenario for which we will find the measurement overhead
1. Read Overhead
2. Loop Overhead

3.1.1.1 Read Overhead

In this section, we are going to measure the read overhead for the time stamp counter.

Methodology:
I calculated the average CPU cycles spend between two rdtsc calls. While doing the experiment, we should make sure that we should not run any other instruction in between two rdtsc calls.

Steps involved in the experiment:

```
1. Read the time stamp counter using rdtsc() API
     start = rdtsc();

2. Read the integer variable.

3. Read the time stamp counter using rdtsc() API.
      end = rdtsc();

4. Calculate the difference of End & Start.

5. Repeat the step 1 to 4 for n times and sum the difference for each
loop.
     sum = sum + (end - Start)

6. Get the read overhead by dividing "sum" with "n".
```

Prediction:

The overhead lies in calling the rdtsc API inside the code to read the time stamp counter value. This results in pushing the base register pointer to the stack and moving the current stack pointer to the base register pointer + calling RDTSC & CPUID assemble instruction and at the end of the function the value in the base register pointer is popped back to the stack.

It is very tough to predict the read overhead of the TSC directly because API involves lot of complexity like CPUID and there is no direct way to calculate the number on paper based on some formula. So I went through the Intel white paper and tried to find the overhead as CPU cycles.

Reading the Time stamp counter register value would take 100 to 145 CPU cycles. It totally depends on the CPU Architecture.

Actual Results:

	1	2	3	4	5	6	7	8	9
Cycles	134	127	123	124	123	123	123	124	123

Analysis:

Our results lies in between the range of results we estimated, showing that the approach to the problem was more or less accurate. The first value is high as compare to other iteration values, it might be because this data might not get populated in the cache during 1st iteration. But Running the executable several times also shows consistency (cycles = 123), which encourages us to implement the rdtsc model in all other experiments we designed.

3.1.1.2 Loop Overhead

In this section, we are going to measure the program loop overhead.

Methodology:

Similarly, to the read overhead, before running the loop, I noticed the time-stamp counter value and after the loop, I have taken time stamp counter value again. Then I took the difference of these two values. Later on, I repeated the same scenario for N times then I took the average.

Steps involved in the experiment:

```
1. read the time stamp counter using rdtsc() API
     start = rdtsc();

2. run the loop
     for (int i=0; i< num; i++)
     {
     }

Note: we should not run any instruction with in the loop.

3. Read the time stamp counter using rdtsc() API.
     end = rdtsc();

4. Calculate the difference of End & Start.

5. Repeat the step 1 to 4 for n times and sum the difference for each
loop.
          sum = sum + (end - Start)

6. Get the loop overhead by dividing "sum" with "n".
```

Prediction:

For loop overhead, I predicted 5 CPU cycles after converting the for loop to the assembly code.

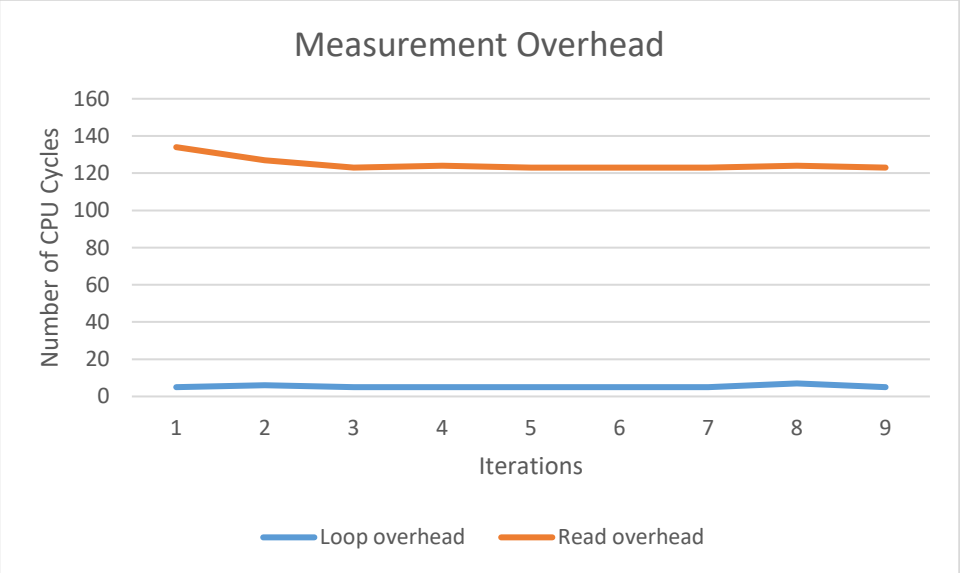
For example: I have considered "for (i=0; i<n; i++)" loop and mentioned the corresponding assembly code below

```
1. move the operand to register
   MOVE  <operand>, R2

2. loop label and its body
   loop1:
          <loop body> ; Whatever you wanna do goes here, should not
          change cx

3. increment the operand
```

```
    inc R2       ; Increment

4. compare the operand with n
    cmp R2,n


5. Branch to the beginning of for loop if less than flag is set

    JNZ loop1
```

Actual Results:

	1	2	3	4	5	6	7	8	9
Cycles	5	6	5	5	5	5	5	7	5



Analysis:
Results are very much same as the predicted value.

3.1.2. Procedure Call Overhead:

Procedures are a set of statements that the process wants to execute to fulfill a functionality. Procedure call overhead is the additional expense occurring to the processor when a procedure is called.

When a procedure call is called the OS is required to perform some tasks, they are,

- Change the mode user to kernel mode.

- Save the state of the process which called the procedure.
- Load the registers with the values of the procedure from the stack.
- Set the program counter to the starting of the new procedure and execute the statements.
- When the execution is complete, save the registers to the stack, load the state of the caller process back to the registers, and,
- Change the mode to user mode and return the control to the caller process.

In this experiment, we are calculating the procedure call overhead and also, the incremental overhead that occurs when the number of arguments are increased.

Methodology:

To calculate the overhead caused by a procedure call we created a function and then we observed the timestamp counter value before and after calling the procedure. To find the incremental overhead we did the following steps:

Steps involved in the experiment:

1) Create the procedure with no arguments initially.
2) Read the timestamp counter value; start = rdstc()
3) call the function
4) read the timestamp counter value; end = rdstc()
5) take the difference of start and end.
6) Increase the number of arguments by one and then repeat steps 2 to 6.

The process is repeated till readings are taken for procedure call with 7 arguments.

Prediction:

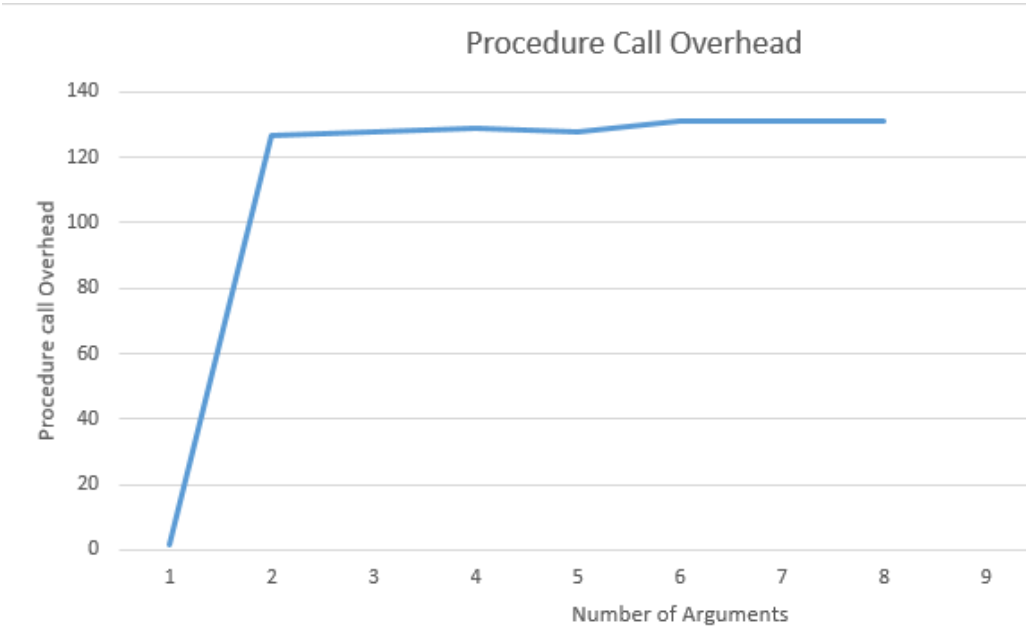
The following tasks are done by the OS for complete execution of the procedure call,

1) Push the arguments to the stack in reverse order.
2) Push the address of the caller to the stack.
3) Update program counter to point to the beginning of the procedure.
4) Place the return value.
5) Push the local variables to the stack.
6) After execution pop the values of the stack.
7) Update the program counter to the caller
8) Pop the return address and parameters off the stack.

In the first step, as the number of parameters increase, there will be an increase in the number of cycles. Taking the incremental overhead to be ‘n’, there should be n+(10-12) cycles.

Actual Results:

Number of Args	T1	T2	T3	T4	T5	T6
1	127	125	127	126	125	125
2	128	127	128	128	127	127
3	128	129	129	128	128	128
4	130	130	129	129	129	129
5	131	131	131	130	130	131
6	131	132	132	131	131	131
7	132	132	132	131	131	132



Analysis:

The values calculated are quite close to what we estimated.

References:

1. http://www.businessdictionary.com/definition/procedure.html
2. http://www.cse.psu.edu/~deh25/cmpsc473/notes/procedure_calls.html

3.1.3. System Call Overhead:

A system call is a request made via a software interrupt by an active process for a service performed by the kernel. The system call overhead is the additional operations the OS has to perform for successful completion of the service the process has requested. This requires a lot of steps to perform, mainly,

1. Change mode from user to kernel.
2. Kernel looks up the system call number in the table, finds the kernel routine and validates the number of parameters required for the system call.
3. It then calculates the address of the first parameter and passes it to the system call routine.
4. After execution if there is no error, then it returns the control to the user mode and restores the stack.

For conducting the experiment, we used the getpid and time system calls.

Getpid system call returns the process id of the current running process. Time system call returns the system time in human understandable time format.

Methodology:

We approached this experiment by observing the timestamp counter value. And then, we invoked each of the system calls in a tight loop and observed the timestamp before and after the iterations.

Steps involved in the experiment:

1. Observe the rdtsc() value, start = rdtsc()
2. run both the system calls in a loop with N iterations.
3. Observe the rdtsc() value, end = rdtsc()
4. difference of start and end gives the measurement.

Prediction:

“getpid” is a low overhead system call compared to the time system call because, getpid just retrieves the value from the current process address space, whereas the time system call has a lot more operations to perform. Therefore, it will take lesser number of cycles than the time system call.

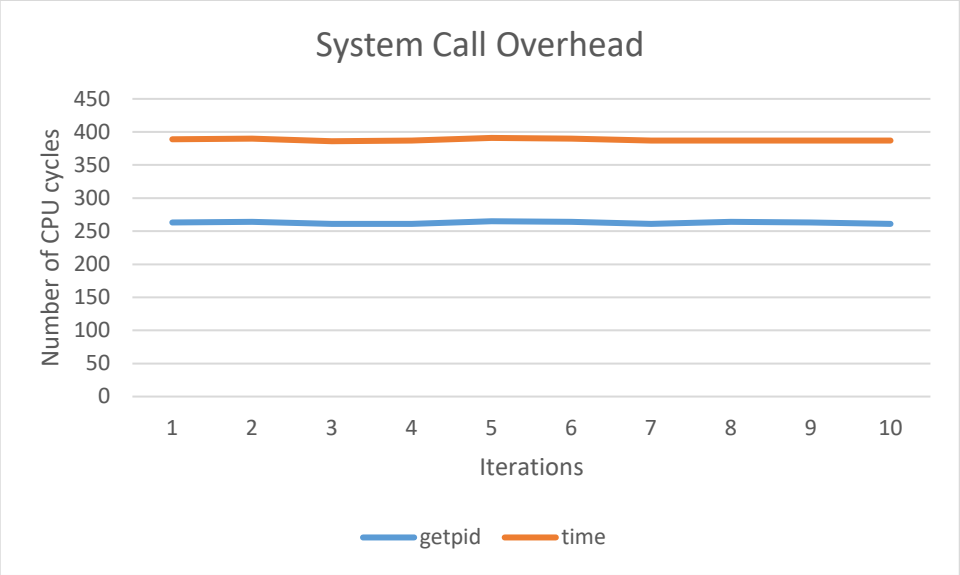
The overhead caused by calling a system overhead is much more than the overhead created by procedure call, provided they perform very few computations because a system call is required to perform operations like

1) perform context switch
2) trap into the kernel
3) Check whether the passed arguments are legal or not.

These operations will require more computational cycles than what is performed during procedure call.

Actual Results:

System Call	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Getpid	263	264	261	261	265	264	261	264	263	261
Time	389	390	386	387	391	390	387	387	387	387



Analysis:

After running the two system calls, getpid showed an average overhead of 263 cycles and time system call showed 387 cycles. This shows that the getpid is a low overhead system call compared to time system call, and thus confirming our prediction. Also, as I said above, a system call has a lot of operations to perform that are costlier than performing a procedure call, like,

1. context switch
2. trapping into the kernel
3. computation of user address of the first parameter.
4. Passing the validated parameters to the system call routine.

These operations make the system call much costlier than procedure call.

Note:
Before running each iteration of getpid(), we flushed the cache.

References:

1. https://www.ibm.com/developerworks/community/blogs/kevgrig/entry/approximate_overhead_of_system_calls9?lang=en
2. http://www.tutorialspoint.com/unix_system_calls/time.htm
3. http://www.di.uevora.pt/~lmr/syscalls.html

3.1.4. Task Creation Time:

In this experiment, we have calculated the time taken by creation of a

1. Process
2. Thread

At the end, we compared the results of both.

3.1.4.1 Process Creation Time:

Methodology:

In order to calculate the process creation time overhead, we need to create a new process from the current running process. To create a new process on Unix-like operating systems, we have to call "fork ()" system call.

Basically process creation time is, difference between time stamp counter value before and after fork() API. So before running the fork() system call, we have saved the time stamp counter (let's say start). When we call fork(), it creates a child process by duplicating the current process address space. And based on the return value of the fork() system API, code execution will go to parent or child process section. So we can take a note of the time stamp counter in the parent process section. Now we have both start and end time stamp counter. So we can take the difference of these values and repeat the same experiment and calculate the average value.

Steps involved in the experiment:

```
1.  save the time stamp counter
     start = rdtsc();

2.  Call the fork() in the current running process.
     RetValue = fork();

Note:
   ➢ If fork() returns a negative value, the creation of a child
     process was unsuccessful.
   ➢ fork() returns a zero to the newly created child process.
   ➢ fork() returns a positive value, the process ID of the child
     process, to the parent. The returned process ID is of type pid_t
     defined in sys/types.h. Normally, the process ID is an integer.
     Moreover, a process can use function getpid() to retrieve the
     process ID assigned to this process.

3. In the parent process, save the current value of the time stamp
counter (call it as end counter).
     end= rdtsc();

4. take the difference between start and end
```

```
     cycles = end - start

4. Repeat the step1 to 4 for n times and take the average value.
```

For example: the pseudo code will be like

```
     pipe p[2];
     retValue = fork();
     if (revalue == 0)
     {    /* child process */
          exit();
     }
     else
     {    /* parent process */
          end = rdtsc();
          wait(NULL);
     }

     time = if (end > start) ? (end – start) : 0;
```

Prediction:

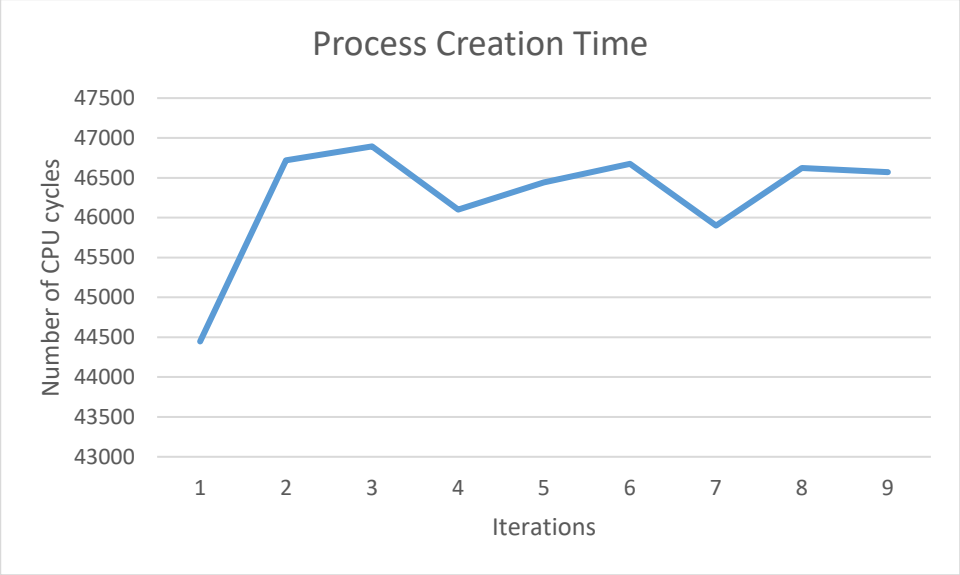
We expect the process creation time using fork() involves lot of steps

1. Creates a new address space.
2. Assigning new pid.
3. Copies text, data, & stack into new address space.
4. Provides child with access to open files.
5. Adding new process into the process list.

If we consider all the steps mentioned above then we believe, it will take around 30,000 to 40,000 CPU cycles.

Actual Results:

	1	2	3	4	5	6	7	8	9
cycles	44447	46719	46894	46102	46441	46674	45900	46622	46572



Analysis:

The above result outcomes are high as compare to the predicted values. The reason could be , result value includes the context switch overhead as well.

3.1.4.2 Thread Creation Time:

Methodology:

In this experiment, we have used the same logic as “process creation time” but we have used different set of API's to create the thread.

We have used the POSIX thread libraries (pthread) to create new kernel level threads.

To create the thread, we have used “**pthread_create()”** API and to join the thread, we have used “**pthread_join()”** API.

Steps involved in the experiment:

```
1. save the time stamp counter value (let's say, start)
     start = rdtsc();

2. create & join a thread
     pthread_create(&td, NULL, threadFunction, NULL);
     pthread_join(td, NULL);

Note: threadFunction() just calls pthread_exit(). It means, we create
a thread and will exit immediately.

3. save the time stamp counter value (lets sat, end)
     end = rdtsc();
```

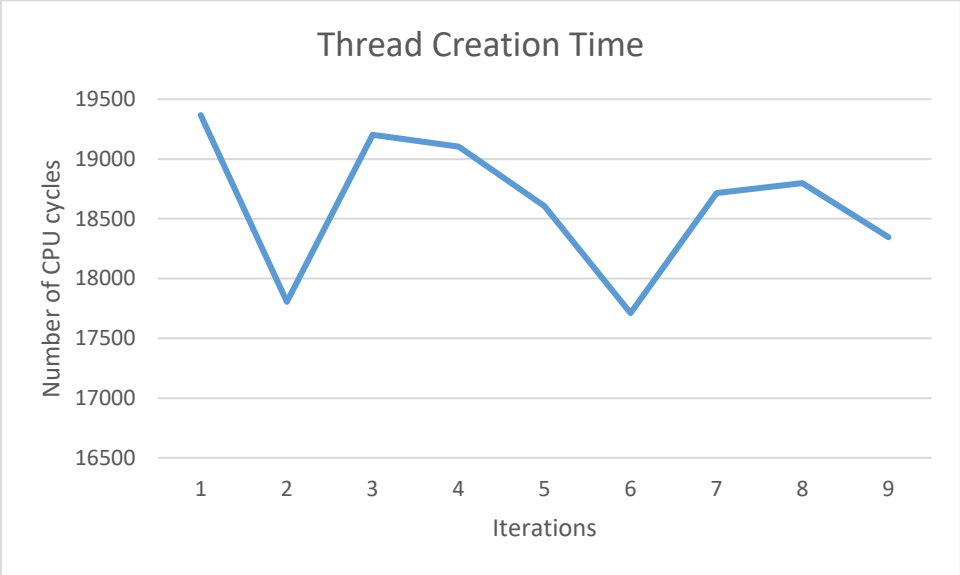
```
4. Take the difference of both the value.
     cycles = end – start;

5. Repeat the Step 1 – 4 for n times and take the average.
```

Prediction:

Threads are very light weight as compare to the process because threads do not need new address space, global data, program code or operating system resources. Thread only need a stack and storage of registers. Therefore, we guess, thread creation time would be 15,000 to 20,000 CPU cycles.

Actual Results:

	1	2	3	4	5	6	7	8	9
cycles	19367	17805	19201	19103	18605	17711	18715	18798	18346



Analysis:

The obtained results are within the range of the predicted values.

→ As I mentioned above, threads are light weight process which doesn't need new address space or process control block , global data, program code etc. + Context switching are fast when working with threads. The reason is that we only have to save and/or restore PC, SP and registers. So thread creation will take less time and it have been proven by the results as well. Thread creation time is 60-65% less than the process creation time.

References:

1. *https://en.wikipedia.org/wiki/POSIX_Threads*
2. *https://man7.org/linux/man-pages/man7/pthreads.7.html*

3.1.5. Context Switch Overhead:

In this experiment, we have calculated the context switch overhead for a

1. Process
2. Thread

At the end, we compared the results of both.

3.1.5.1 Process Context Switch Overhead:

Methodology:

To calculate the context switch overhead, we need two running processes on one CPU. Once we will have two process, we will create a scenario in which one process will be dependent on the activity done by another process like process communication using pipe: one process will write to one end and another process will read data from another end of the pipe.

To create the above scenario, we have opted for parent-child process relationship. So when we call fork(), it creates a child process by duplicating the current process address space. And we cannot follow the similar logic for overhead calculation which we followed in the previous experiment like take the difference between start and end time stamp counter value because to take the difference, we need both the values in one process and we can get the end value only from the process which we are creating (child).

To share the time stamp counter value to the parent process, I have used pipe based inter-process communication. Once the child process will get created, child will write the current time stamp counter to the pipe and on the other end, parent process will read the value and it will take difference. We have repeated the same scenario for n time and taken the average value.

Steps involved in the experiment:

1. Call the fork() in the current running process.
   RetValue = fork();

Note:

- If fork() returns a negative value, the creation of a child process was unsuccessful.
- fork() returns a zero to the newly created child process.
- fork() returns a positive value, the *process ID* of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer.

```
     Moreover, a process can use function getpid() to retrieve the
     process ID assigned to this process.

2.  In the parent process,
     1. Save the current value of the time stamp counter (call it as
end counter).
     2. Read the end counter value which child will write to the pipe
using read API.

3.  In the current process,
     1. Save the current value of the time stamp counter (says end
counter).
     2. Write the counter value to the pipe file descriptor using
write API call.
```

For example: The pseudo code will be like

```
     pipe p[2];
     retValue = fork();
     if( ret == 0 )
     {    /* child process */
          end_cycles = rdtsc();

          write(firstpipe[1],(void *)&end_cycles, sizeof(uint64_t));
          exit(1);
     }
     else
     {    /* parent process */
          start_cycles = rdtsc();

          wait(NULL);
          read(firstpipe[0],(void*)&end_cycles,sizeof(uint64_t));
     }

     time = if (end > start) ? (end – start) : 0;
```

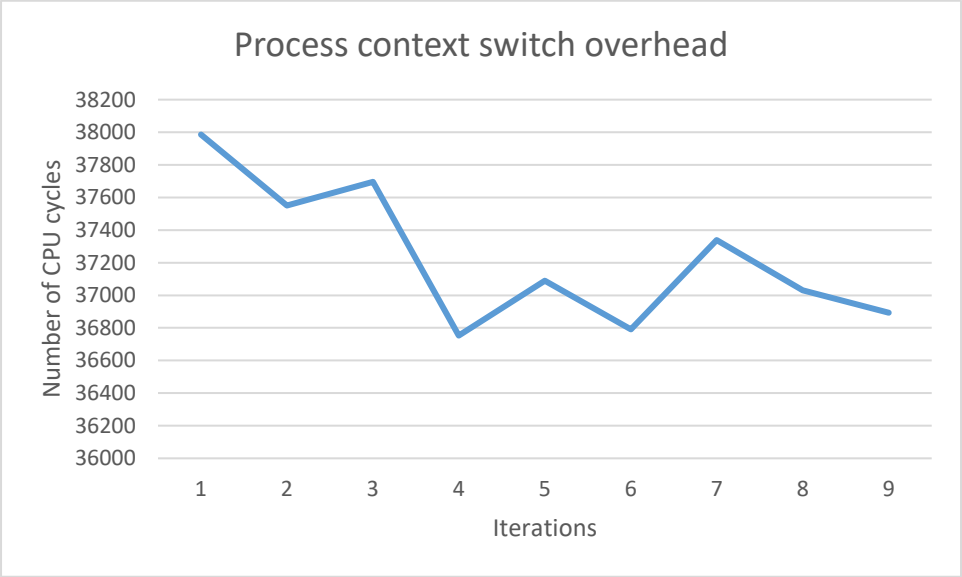
Prediction:

The process context switch is nothing but “Saving and Restoring Context” which involves various operations like
1. Save the state of the current executing program.
2. Save all of its virtual memory configuration.
3. Loading of new program.

By looking at the operation, we would say it is a heavy operation which takes 30000 – 40000 CPU cycles.

Actual Results:

	1	2	3	4	5	6	7	8	9
Cycles	37986	37550	37695	36753	37090	36792	37338	37031	36893



Analysis:

The result we obtained were very similar to our estimates. This was due to the fact that to make the results maximum reliable we avoided the multi core optimizations.

3.1.5.2 Thread Context Switch Overhead:

Methodology:

We have used the same logic which we used in process context switch overhead experiment. Instead of using the fork() system call, we have used “pthread_create” & “pthread_join” API's to create and join the thread respectively.

Steps involved in the experiment:

```
1. Create a thread by calling pthread_create API.
     pthread_create(&tid, NULL, threadFunction, NULL);

Note: threadFunction will get executed after the thread context
switch. So this function saves the current value of the time stamp
counter (let’s say end counter) and writes to the pipe.

2. save the time stamp counter (let’s say start counter)
```

```
     start = rdtsc();

3. Call pthread_join() API.

4. Read the pipe because after the context switch, threadfucntion
will write the end time stamp counter to the pipe.

5. Take the difference between end & start.

6. Repeat the step 1 to 5 n times and calculate the average.
```

For example:

```
 /* create a thread */

pthread_create(&tid,NULL,threadFunction
, NULL);

start = rdtsc();

pthread_join(tid, NULL);

read(pipe[0],(void *)&end, sizeof(end));

time = if (end > start) ? (end – start) :
0;
```

```
 void *threadFunction(void *)
 {
    uint64_t end = rdtsc();

    write(pipe[1],(void*)&end,
sizeof(end));

    pthread_exit(NULL);
  }
```

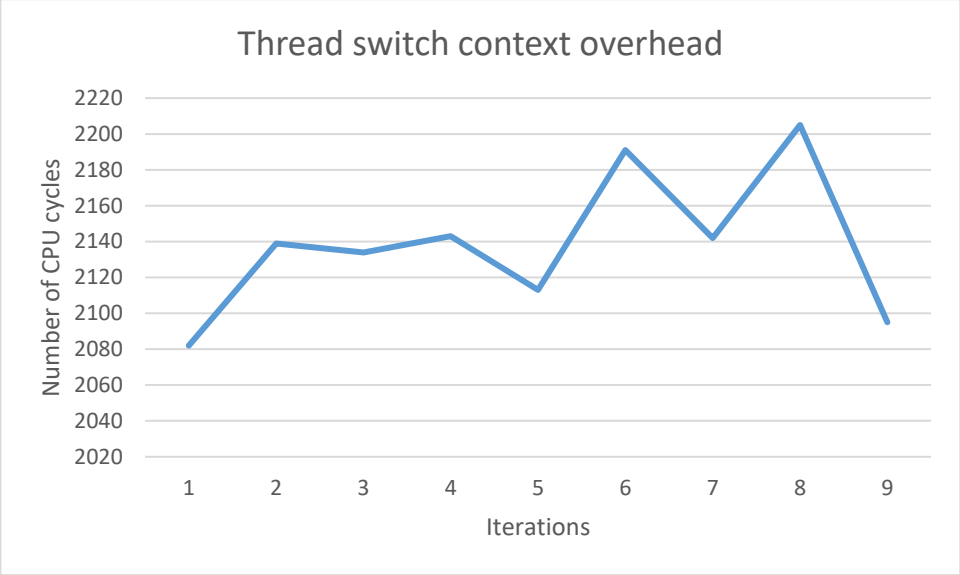
Prediction:

The thread context switch is very light as compare to the process context switch because in the thread context, OS doesn't do the operations which it does for the process like save process state , save memory configuration etc.
Reason for this is, threads share the address space. So OS doesn't need to reconfigure the address space + thread doesn't have the state, it means, there is nothing to save w.r.t state.

By considering all the points, we would say it is a light operation which takes 1500 –3000 CPU cycles.

Actual Results:

	1	2	3	4	5	6	7	8	9
cycles	2082	2139	2134	2143	2113	2191	2142	2205	2095



Analysis:

The result, we obtained were within the range of our estimation.

→ A thread context switch means shifting from one thread to another, within one program. Threads shares one address space with other threads in the program. A thread context switch is cheaper than a process context switch because since the memory management unit does not need to be reconfigured.

If we compare the results, then we can easily see that thread context switch time is 16-20 times lesser than the process context time. So the results are proving the above discussion that threads context switch is a much lighter than process context switch.

3.2 Memory

3.2.1. Memory Access Time:

Memory access time is the time taken to read 1 byte of data from the memory system. The feature of the average memory access time is that it can be also extended across different hierarchical levels of memory management system.

In this experiment we are trying to calculate the time taken to access individual integer from L1,L2,L3 caches and the main memory.

The experiment was conducted on the basis of the Imbench paper. It had defined the memory read latency through 4 key definitions, they were, Memory Chip Cycle latency, pin-to-pin latency, load in a vacuum latency and back-to-back load latency. We had implemented the back-to-back load latency in our experiment. In back-to-back-load latency it calculates time taken for each load, after assuming that the instruction before and after the current instruction are cache-miss loads.

Methodology:

Our approach to this experiment was based on the Imbench paper. Here we instantiated varying chunks of memory to different array sizes starting from 4 KB to 512 MB. Just as in the paper, we also created array strides of different sizes and each integer array was assigned a list of pointers to point to each of them.

Steps involved in the experiment:

1. Create integer arrays of varying sizes from 4KB to 512MB and strides of 4B to 16MB.
2. Place values at each stride positions for each array.
3. Read timestamp counter value; start = rdtsc()
4. read values from the array for N iterations.
5. Read timestamp counter value; end = rdtsc()
6. the average of difference between start and end gives the memory latency for that particular array size and stride size.

Prediction:

According to our prediction, the array range till 64 KB will have the least memory access time as they are in the range of L1 cache. From 16KB to 256KB will show an increase in access time as the L1 cache shifts to L2 in this range. There should be an increase from here, as most of the accesses are in this range.

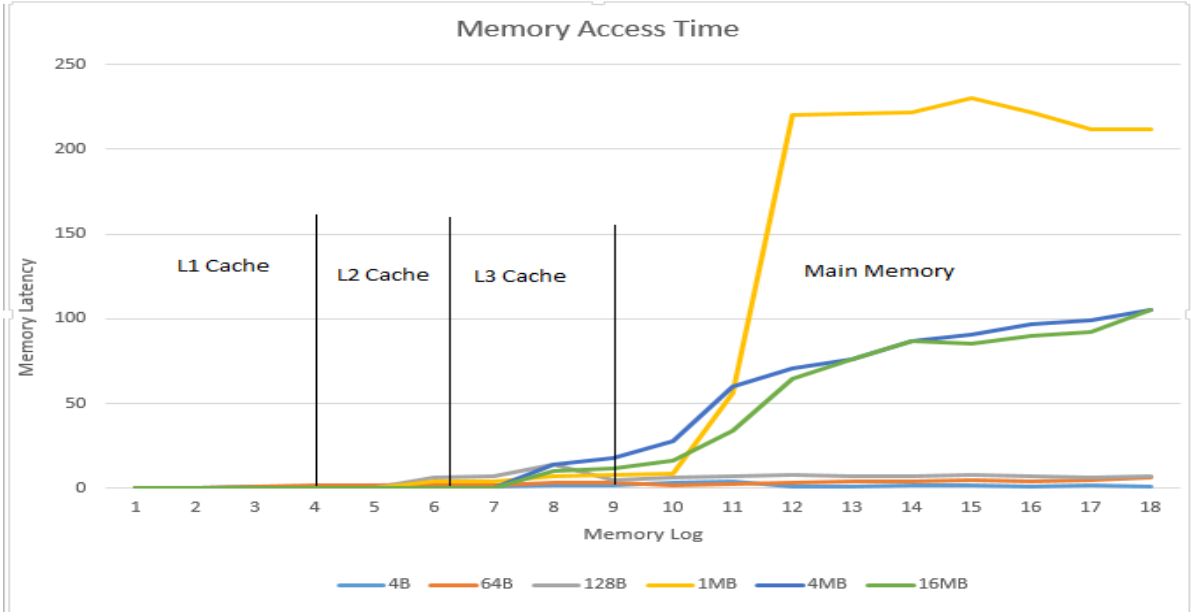
The other overheads that occur are the memory read overhead, measurement overhead and the loop overhead. These have to be accounted for while measuring the latency.

L1 cache: 6 cycles = 0.53ns
L2 cache: 8-12 cycles = 5.5ns
L3 cache: 65 cycles = 20 to 30 ns
Local DRAM: 110 cycles = 90 to 100 ns

Actual Results:

Memory Region	Time for Stride 4B	Time for Stride 64B	Time for Stride 128B	Time for Stride 1MB	Time for Stride 4MB	Time for Stride 16MB
4	0.5	0.3	0.3	0.3	0.3	0.3
8	0.6	0.4	0.4	0.4	0.4	0.4
16	0.7	1	0.5	0.5	0.5	0.5
32	0.4	2	0.6	0.6	0.6	0.6
64	0.5	2	0.5	0.5	0.5	0.5
128	2	2.5	6.5	4	0.5	0.5
256	1	2	7	4	0.4	0.4
512	2	3	14	7	14	10
1024	2	3	5	8	18	12
2048	3	2	6	9	28	16
4096	4	2.5	7	56	60	34
8192	1	3	8	220	71	65

16384	1	4	7	221	76	76
32768	2	4	7.5	222	87	87
65536	2	5	8	230	91	85
131072	1	4	7	222	97	90
262144	2	5	6	212	99	92
524288	1	6	7	212	105	105



Analysis:

After looking into the data we collected it and graph, we arrived at the following inferences,

1. For the initial stride which is 4B, nothing but an integer will be in L1 cache and the next element of the array will also in the cache. So memory access time will be very very low.
2. As the size of the stride grows like from 64B to 1MB, the whole data will be splited into multiple caches because the size of L1 & L2 cache is very less and to accommodate the whole data, few bytes will go to L1, few to L2 and few to L3. So Now the memory access operation will happen from the different cache line. So the access time will be little high as compare to the previous step.
3. Once the stride size will be more than L3 cache size then there will be a high peak for the access time because program/OS will read the data from main memory instead of cache.

In conclusion,

1.. The prediction for the access to our L1 cache was clearly defined. There was a clearly a steady increase after L1 cache as we had predicted.

3. Summing it up the model we had designed looks pretty accurate.

References:

1. http://www.webopedia.com/TERM/A/access_time.html

2. https://www.usenix.org/legacy/publications/library/proceedings/sd96/full_papers/mcvoy.pdf

3.2.2. Page Fault Service Time:

Page faults, also known as hard fault, is a type of interrupt, called trap, that is raised by the hardware when a process tries to access a page that is loaded in the virtual address space, but is not loaded in the main memory. On encountering the page fault, the interrupt handler through a disk-read request tries to replace the requested page in the main memory, and this is called page fault service. The time taken by the OS to replace this page is called page fault service time and that is the objective of this experiment.

Methodology:

We had approached the experiment as described in the Imbench paper. We had created a large file of 2.8 GB and then, memory mapped it using the mmap() system call. We selected 2.8 GB of data so that it will won't be too small to reside in the cache and also it won't be too large so that it doesn't end up in the disk. Next, we chose an array stride of 16 MB so as to avoid caching the data already read. The timestamp counter values are observed before and after the memory accesses.

Steps involved in the experiment:

1. Create file of 2.8 GB and array stride of 16MB
2. the file is mapped to memory using mmap().
3. observe the timestamp counter values; start = rstdc();
4. run random accesses into the mapped file for N iterations.
5. Observe the timestamp counter value; end = rstdc();

Prediction:

Page faults results in loading 4 KB pages from the disk, which is a very slow process compared to memory access. Most of the time was spent on disk seek, latency and the transfer of page from the disk to the main memory. Initial run showed a much higher share of cycle time, as it was loaded from the physical memory and was later stored in cache, which explains the lesser cycle time in the subsequent runs.

Mostly the page fault overhead will be because of accessing the disk + transferring the data to the memory. So, if we have average read rate of the disk and the page size then we can calculate how much the transfer time of a page will be.

Transfer time = Page Size/Average read rate

According to our test setup the page size is 4KB and average read rate is 450 Mbps

Transfer time = 4 KB/450 Mbps = 0.009 msecs

Access Time = 0.09 msecs

Therefore, Page fault overhead = Transfer Time + Access Time + other overhead
= 0.09 + 0.009 + other overhead
= 0.099 + 0.1 (assuming other overhead will be 0.1 msec)
= 0.199 msec

Actual Results:

Predicted Time(Hardware+Software)	Measured Mean Time
0.199 msecs	0.23 msecs

Analysis:

Our predicted model and the measured model were quite similar. The page fault latency for a byte is 0.23/4KB. This is very much less than the latency occurring when a byte is accessed from memory.

References:

1. https://surriel.com/system/files/linux24-vm-freenix01.pdf
2. http://blog.scoutapp.com/articles/2015/04/10/understanding-page-faults-and-memory-swap-in-outs-when-should-you-worry

3.2.3. RAM Bandwidth:

Memory bandwidth is the rate at which data can be read from or stored into a semiconductor memory by a processor. Memory bandwidth is usually expressed in units of bytes/second.

Memory Bandwidth = Total number of bytes / time;

Methodology:

There are lot of different way using which we can benchmark the RAM bandwidth but we have opted the idea proposed by the Imbench paper. We have created two different operations; for read and write bandwidth.

For read bandwidth, we created an array of integers of size 8M so that it won't fit in cache because we have to calculate the RAM bandwidth, not the cache bandwidth. And then summed i and size/2+i elements so that accessing the far reached elements will create a cache miss each time. Then, we run this logic in loop.

For write bandwidth, we created a chunk of memory and every time we accessed the memory to write a value, then memory address was incremented by one. This was run in a loop and the result was averaged and the overheads were included.

Before & after running the above mentioned logic, we have noticed the time stamp counter value and then took the diff. and then averaged the result.

Steps involved in the experiment:

```
1. Save the time stamp counter value to a variable.
     Start = rdtsc();

2. For read bandwidth calculation, run the loop in such a way that
always we miss cache while accessing the value.

3. For write bandwidth calculation, run the loop and write the value
every time.

4. Save the time stamp counter value.
     end = rdtsc();

5. Take the diff of the (end – start).

6. Repeat the step 1 to 4 n times and take the average.

Note: we need to reduce the loop overhead and memory read overhead
while doing calculation.
```

Access time = (end – start) – (loop-overhead) / (number of loops) - (memory-read-overhead)

Prediction:

To calculate the RAM bandwidth, I have used the below mentioned formula.

```
Bandwidth: (memory bus clock rate) × (Number of channels) × (memory bus width).
```

According to our machine,

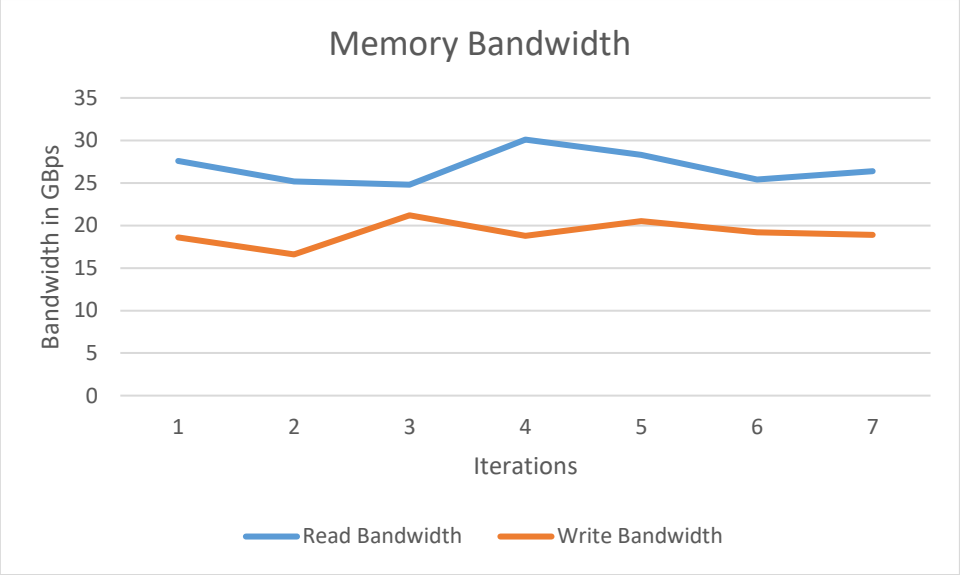
1. Memory bus clock rate = 1600 MHz
2. Number of channels = 2 (because RAM is DDR)
3. Memory bus width = 64 bits or 8 bytes

Therefore, Bandwidth = 1600 *(10^6) * 2 * 8
= 25.6 GBps

Actual Results:

	1	2	3	4	5	6	7
Read Bandwidth	27.6 GBps	25.2	24.8	30.1 GBps	28.3	25.4 GBps	26.4

		GBps	GBps		GBps		GBps
Write Bandwidth	18.6 GBps	16.0 GBps	21.2 GBps	18.8 GBps	20.5 GBps	19.2 GBps	18.9 GBps



Analysis:

The results for the Read Memory Bandwidth looks pretty same as the predicted one but the write memory bandwidth is little low because according to intel, the write transaction turns into a read followed by a write to maintain cache consistency.

References:

1. http://www.makeuseof.com/tag/difference-ddr2-ddr3-ram-technology-explained/
2. https://en.wikipedia.org/wiki/Memory_bandwidth

Conclusion:

There were few hardware/software predictions went wrong but majority of them were close to the obtained results.