CSC501 Project 1

Performance Measurement

Nishad Sabnis (nsabnis) 001083835 | Alok Kulkarni (akulkar4) 200106986

Introduction

This project is a series of experiments performed in order to get a quantitative performance measurement of an operating system w.r.t. certain operations.The goal of the project is to be able to characterize and understand the operating system and the hardware it runs on. For each operation, the goal is to measure as accurately as possible the time required for the system to execute the operation and compare its relation to the constraints placed by hardware. By comparing the hardware constraints and the software performance, we will be able to quantify the software overhead induced by the operating system.

Language Used	C (along with some inline assembly)
Compiler	gcc version 5.4.0 (ubuntu 16.04 repo)
Optimization	no compiler optimization applied
Environment	ThinkPad T420 Laptop with i5 core
Time Spent	20 hrs

Machine Description

Component	Value
Processor	Intel Core i5-2520M
Cycle Time	0.4 ns
L1 cache	64kB 8-way set-associative data cache
L2 cache	256kB 8-way set-associative data cache
L3 cache	3MB 12-way set-associative unified cache
Memory bus	
I/O Bus	Intel Corporation 6 Series/C200 Series (PCI)
RAM	2 x 4GB RAM@1333MHz
Disk Capacity	320GB
Disk RPM	7200RPM

Operating System	Ubuntu 16.04.1 LTS Xenial
Kernel	Linux 4.4.0-36-generic

Experiments

Prerequisites

In order to have consistent measurements, several advanced features of modern processors need to be switched off as they allow for frequency scaling for power or performance. These were all switched off(accessing the BIOS) and the processor was set to a constant frequency of 2.50GHz. Also, the nice priority of the process was set to -20 (highest priority) and it was run using “taskset -c 0” so as to constrain it to one core.

Measurement Overhead

The Intel Core i5 Sandy Bridge processor being used has both the RDTSC and RDTSCP instructions as part of its ISA and thus a few simple lines of assembly can be used to write the functions that trigger the start and end of the timestamp. Care has to be taken to serialize the instruction flow so that no out-of-order execution instructions creep into the measurement. This is done by use of the CPUID “barrier” instruction.

Methodology: The pseudo code for the measurement overhead is as follows. (MOV instructions to store the results of counter not shown).The RDTSC and RDTSCP calls to signify the start and the end of the measurement have been abstracted as two calls, meas_start() and meas_stop(). Both the functions have been made inline so that no extra overhead is observed.

Usually, the code to be measured will be in the middle as shown, but as we want to measure the overhead of the measurement itself, we will just call meas_start() and meas_stop() immediately after the other and take the difference.

```
CPUID
RDTSC
/*Code to be measured (in this case empty)*/
RDTSCP
CPUID
```

Result: In our case, the measurement overhead comes out to be 52 cycles. At 2.50GHz, that corresponds to 20.8 nanoseconds.

Base Hardware Performance	Estimated s/w Overhead	Estimated Value	Actual Value
N/A	16ns	16ns	20.80

The result is similar to predicted values. Based on certain hardware values seen online, it seemed like 40 cycles is the time taken by the rdtsc measurement itself, but in our case, with our hardware it turned out to be 52 cycles.

To measure loop overhead, we use an empty for loop that runs for 1000 cycles and divide the elapsed time by 1000 to find the overhead of a single iteration of the for loop. We then get 1000 such sets so as to find the average and standard deviation. This comes out to be 2.9 nanoseconds with a standard deviation of 0.395 nanoseconds.

Base Hardware Performance	Estimated s/w Overhead	Estimated Value	Actual Value
N/A	1.6	1.6ns	2.9

Procedure Call Overhead

Methodology: To measure the procedure call overhead we simply created dummy procedures with varying number of arguments (0-7). The functions themselves consisted of only a return instruction. The "__noinline__" attribute was used for each of these procedures to prevent the compiler from inlining them, thus hiding any overheads to procedure calls. Another check to prevent any kind of measurement altering optimization was to make all the arguments volatile. We took averages of each 100 measurements, and 1000 such sets of 1000 measurements and calculated the standard deviation for the same. As we are calling these in a for loop the for loop's overhead is reduced to get the actual time required for the procedure call. The pseudo code is as follows:

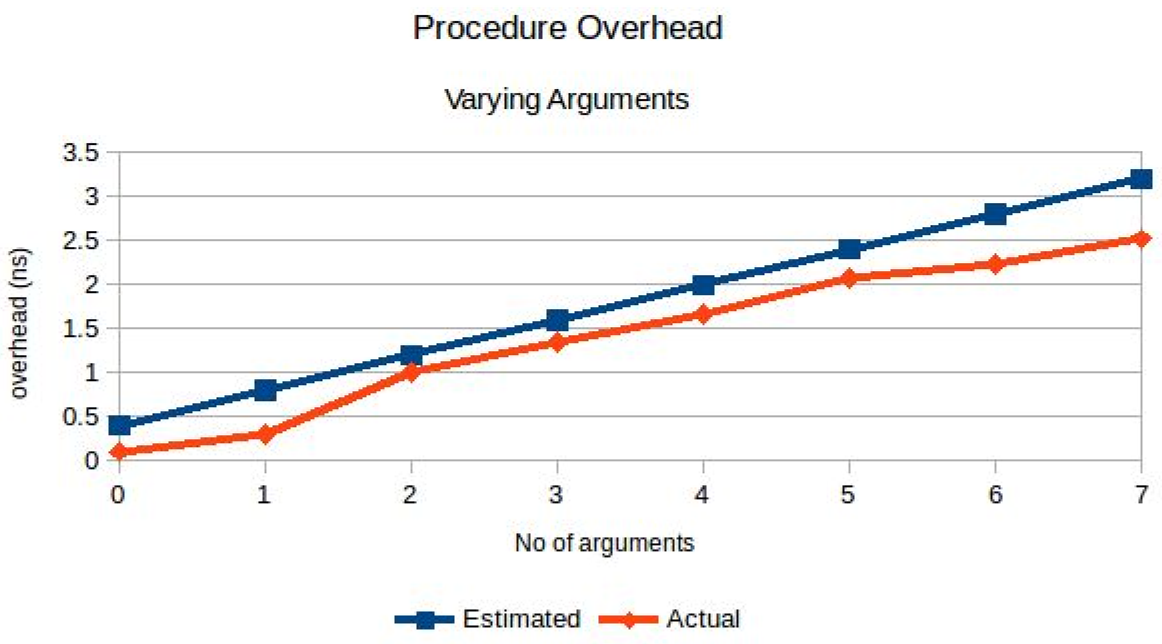
```
void procedure_overhead(void)
{
  for(int i=0;i<1000;i++)
  {
    start = meas_start();
    for(j=0;j<1000;j++)
    {
     procedure_arg_0();
    }
    end = meas_end()
    result[i] = end-start;
  }

  sd = standard_deviation(result,1000); // calculation of standard deviation
}


//procedures are defined as follows
void __attribute__ (noinline) procedure_arg_0(void)
{
  return;
}

void __attribute__ (noinline) procedure_arg_1(uint32_t)
{
  return;
}
```

Result: The graph below shows the expected overhead vs actual overhead for each of the procedure calls.



0 arguments : 0.1 Standard deviation: 0.24
1 argument : 0.3 Standard deviation: 0.234
2 arguments : 1.0 Standard deviation: 0.304
3 arguments : 1.35 Standard deviation:0.264
4 arguments: 1.66 Standard deviation: 0.304
5 arguments: 2.07 Standard deviation: 0.552
6 arguments : 2.23 Standard deviation: 0.633
7 arguments : 2.52 Standard deviation: 0.669

The graph shows that for certain cases, even with increase in argument, there is a very slight increase in the time required to call that function and exit. Ideally, the code will have more pushes and pops from the stack (or movs to register) but in practice it is not necessarily so, due to pipelining and other hardware optimizations. On average, the increment overhead of an argument is 0.357ns.

System Call Overhead
Methodology: To measure a minimal system call and its overhead for the system, we used the getppid() call. The measurement was done 1000 times, averaged to get a mean value and 1000 such iterations were carried out to get a standard deviation over a large set. The getppid() call

is a minimal call often optimized for benchmarks and hence we decided to go for a separate call to fstat() which guarantees a switch to kernel space every time. The pseudo code is as follows.

```
for(i=0;i<1000;i++)
{
    start = meas_start();
    for(j=0;j<1000;j==)
    {
       getppid() OR fstat()
    }
    end = meas_stop();
    result[i]=end-start;
}
sd = standard_deviation(result,1000);
```

Base Hardware Performance	Estimated s/w Overhead	Estimated Value	Actual Value
N/A (for getppid)	few us	few us	2.6us(60 ns cached)
N/A (for fstat)	few us	few us	130.393ns(cached)

Result: The results for these two system calls are much faster than we expected, we suspected this might be due to the modern linux kernels which cache the output of these calls so that they switch to kernel only once. When we tried running it in such a way that it was only called once, we got a latency of 2.3 us . Even when they are cached, we can see that fstat() requires 131 ns as compared to a value of ~4-5 ns for a procedure call with 5 arguments. However the fstat timing value does include the actual code of the system call while our procedure was an empty procedure.

Task Creation Time

Methodolody: Process creation and thread creation are non-trivial tasks and involve a lot of operations on the kernel end. For task creation, we decided to use the fork() API to measure the operating system overhead. On entering the child process, we call exit() immediately and in the parent process we use the wait() API to wait for the child process to terminate. This method of measuring task creation time will include the overhead of the fork() and exit() calls.

```
void process_overhead(void)
{
    for(int i=0;i<100;i++)
    {
      start = meas_start();
      for(int j=0;j<100;j++)
      {
         pid = fork()
         if(pid == 0) //child process
         {
           exit(0); // exit immediately
         }
         else
         {
           wait();
         }
      }
      end = meas_stop();
      result[i]=(end-start)/100.0;
    }
    sd = standard_deviation(result,100);
}
```

Note that this benchmark for process creation is a simple one as we are not considering the effects of adding an exec() after fork() as might usually be the case while creating a new process. This benchmark is also put through two nested loops, one to average the values of 100 fork() calls, and the other to get a standard deviation of 100 such sets grouped together.

For kernel threads, we look to the pthread_library. Similar to process overhead measurement, we call pthread_create(), and exit as soon as we enter the newly created thread. In the main thread, we wait for the new thread to terminate by using the pthread_join() API before measuring the end time.

```
void thread_overhead(void)
{
    for(i=0;i<100;i++)
    {
        start = meas_start();
        for(j=0;j<100;j++)
        {
            pthread_create(&thread_one);
            pthread_join(thread_one);
        }
        end = meas_stop();
        result[i]=(end-start)/100.0;
    }
    sd = standard_deviation(result,100);
}

void thread_one()
{
  return;
}
```

Results:

Process overhead : 115.455us   Standard deviation : 22.45us

Thread overhead : 11.678us Standard deviation: 1.84 us

Base Hardware Performance	Estimated s/w Overhead	Estimated Value	Actual Value
N/A (for process)	100us	100us	115.45us
N/A (for thread)	5us	5us	11.678us

As we can see from the above results, process creation is about 10 times as heavy as thread creation. This is because of the extra handling required by the kernel while creating a completely new process.
The result should be reasonably accurate as it takes into consideration both creation and destruction of the process/thread. This is usually the overhead incurred when trying to solve a computation by adding a process or thread. However, if the new process created has a different purpose then an exec() call is required and that will add to the overhead.

Context Switch Time
Methodology : We have used blocking pipes to measure the time required for a process context switch.
For processes:
We created two processes and used blocking pipes between them to cause context switches. The pipes calls were reading and writing in such a way that control was being switched between the two process in a “ping pong” manner. This was achieved by alternating read() and write() calls to two different pipes. The pseudo code will make the methodology more clear.

```
void process_switch_overhead(void)
{
    pipe(ping);
    pipe(pong);

    pid = fork()
    if(pid == 0) //child process
    {
      write(ping);
      read(pong);    // these alternating read and writes will force context switches.
    }
    else //parent process
    {
      read(ping);
      write(pong);
    }
}
```

We perform this kind of alternate read and write within a for loop 100 times. These 100 loops will consist of 200 context switches (back and forth) hence we divide the total time by 200 to get an average value for the context switch. We then take 100 such sets and find the standard deviation over the set. Note that this method of measuring time for context switches will include some overhead from the pipe read and write calls. This has been removed from the final values by separately counting the time required for read and write calls within a single process.

For threads:
The same philosophy as the process context switch is followed, however the more lightweight semaphores are used in this case. The semaphore overhead is removed from the obtained values to get the final thread context switch value.

Results:
Process Context Switch: 980.804ns Standard Deviation: 75.766ns
Thread Context Switch: 480.342 ns Standard Deviation: 20.912ns

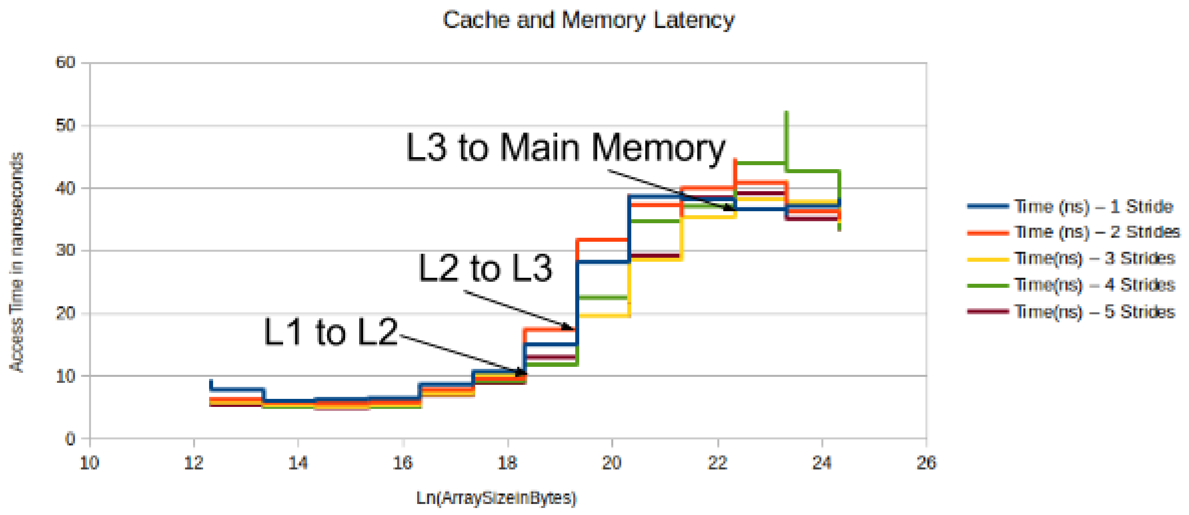
Base Hardware Performance	Estimated s/w Overhead	Estimated Value	Actual Value
N/A(for process)	2us	2us	0.980us
N/A (for thread)	200ns	200ns	480ns

We expected thread switching to be much lighter (order of magnitude) but it turned out to be not as light as expected. This could be because we were running the code on one processor and usually threading performance is higher with multiple cores. Another reason for this low performance might be the mechanism used ot force context switches (i.e. the semaphore). Other implementations were seen which were more lightweight as they used userspace only locks to force the switching. (e.g. FUTEX - fast userspace mutex)

Cache and RAM Access Time
Methodology:
We modeled our tests based on the method mentioned in Lmbench to measure the Cache and RAM access times. We created a linked list, the size of which could be varied over the commandline. The main purpose behind creating a linked list was to prevent the hardware prefetcher from fetching the next memory blocks in advance as this would have affected the timings. It also had to be ensured that each node of the created linked list was at least as big as a cache line width since most modern process (including the one being tested) have cache line prefetching as a part of the caching mechanism. Repeated accesses were made to the created linked list, using a pointer, including looping over it multiple times as a a part of a loop.

Results:
We plotted a graph of the average access time vs size of the list and the key observations are:
- As the size of the linked list was increased, the average time taken for an access further increased. were able to observe how the average access time increased as the size increased.

- The access times for smaller sizes, where the data is localized entirely in the L1 cache are almost the same for the times we loop over and access the list.
- As the size becomes big enough for the data to spill over into L2, L3 and the main memory, the average access times clearly show an increase.
- This trend can be observed for both, when higher sizes are compared the lower sizes as well as when higher sizes are compared amongst themselves with respect to single iterations and multiple looped over iterations.
- The boundaries between the cache sizes can be clearly seen that by virtue of the jumps seen in the graph.



Base Hardware Performance	Estimated s/w Overhead	Estimated Value	Actual Value
L1 : 1.2 ns L2: 3.6 ns L3: 8.8ns RAM: ~50ns	One Cycle for the caches: 0.4ns Two to three cycles for the main memory: 1.2ns	L1: ~2ns L2: ~4ns L3: ~9ns RAM: ~50ns	L1: 5ns L2: 6.35ns L3: 12.1ns RAM: 40ns

The base hardware performance of the caches and memory has been obtained from the processor datasheet. The graph we obtained is not exactly as expected but we think there could be a few reasons for this - hardware prefetching which we have not been able to circumvent despite considerable effort. Another reason could be the pollution of the cache with values from different processes. However, it is still quite clear that the latency is accessing the memory is increasing as expected.

Memory Bandwidth

Methodology: The memory write and read speeds are checked by using the popular functions memcpy() and memset(). Results are taken for 8 and 16 MB of data transfer. When using memcpy() we are both reading and writing from memory. To measure only the write time, we used memset(). The pseudo code below shows the procedure for 8MB.

```
void mem_bandwith(void)
{
   start = meas_start();
   memcpy(arrayDst, arraySrc, (MEM_TOTAL_SIZE));
   end = meas_stop();
   count_ns = meas_convert_to_ns(end - start);
   printf("Memcpy (8MB):%Lf MB/s\n",(long double)(1000.0*(MEM_TOTAL_SIZE/count_ns)));
}
```

Result:
memcpy : 8MB - 5.7GB/s , 16MB - 5.2 GB/s
memset : 16MB - 8.9GB/s , 16MB - 8.2GB/s

Base Hardware Performance	Estimated s/w Overhead	Estimated Value	Actual Value
21.3GB/s	5GB/s	15GB/s	8.9GB/s

The RAM Bandwidth = RAM Frequency * data width * 2 = 1333 * 10^6 * 64 * 2 which comes out to be 21.3GB/s. However, this theoretical maximum will not be reached easily as other processes and the operating system are residing in the RAM too. In our experiment, we manage to get a maximum value of 8.9GB/s to write to memory. This difference could be due to various reasons. One is the implementation of memcpy/memset. If memcpy is implemented using special vectorized instructions, a higher bandwidth could be reached. Another improvement could be to use the vector assembly instructions however this would make the benchmark not portable.

Page Fault Service Time

Methodology: The Linux operating system uses an on-demand paging algorithm. Hence, to generate a page fault, we need to access an area of memory which is not allocated and will thus have to be “brought in” from disk. The easiest way to do this is to memory map a disk file using the mmap system call and then access the newly created page to trigger a major page fault. While doing this experiment, we first used a loop to trigger page faults during each iteration of the loop. On looking at the results, we found that while the first two page faults for the mmap() file were as expected, the third access was around a hundred times faster and continued to be as fast for the rest of the iterations. This could be due to disk caching that is taking place after the same piece of data was asked for repeatedly. Hence, we have included the measurements for both as part of the final observations. The pseudo code is as follows.

```
void pagefault_overhead(void)
```

```
{
  fd=open("GRTAGS",O_RDONLY);
  struct stat stats;
  fstat(fd,&stats);
  posix_fadvise(fd,0,stats.st_size,POSIX_FADV_DONTNEED);
  dataPtr = mmap (NULL,stats.st_size,PROT_READ,MAP_SHARED,fd,0);
  start = meas_start();
  ret = dataPtr[10]; // random access to trigger pagefault
  end = meas_stop();
  munmap(dataPtr,stats.st_size);
  result = (end-start);
  count_ns = meas_convert_to_ns(result);
}
```

Note: Due to modern Linux kernels, once a file is opened it is sometimes brought into memory just in case it might be accessed as such behaviour is likely in programs. Thus calling mmap() would not trigger a major pagefault unless this behaviour of the kernel is nullified. This is done by using the posix_fadvise() API, passing the POSIX_FADV_DONTNEED to trick the kernel into thinking that we will not be accessing this file. We then mmap(), and access it to trigger a major pagefault. We used the time utility with the “-v” verbose flag to ensure that these set of steps always triggered a major page fault.

Results: The pagefault overhead using this method comes out to be 18117232 ns or 18.11 milliseconds. This is for an entire 4KB page. Hence it comes out to 4.52us per byte. From main memory, we require ~100ns to access a single byte.

For a hard disk, access time = seek time + rotational latency. Rotational latency is 30000/RPM = 4.1667ms For this particular hard disk, the typical seek time is 13ms. (varies between 1 to 25 milliseconds). Hence the base hardware performance could be anything between 5 - 29 milliseconds.

Base Hardware Performance	Estimated s/w Overhead	Estimated Value	Actual Value
5 - 29 ms(typical 17.1)	200us	17.3ms	18.1ms

We found that the overhead of the pagefault was always between around 12 - 20 ms which fits well in line with the base hardware performance expected. In this case, the software overhead is not the major factor, the access to disk is the limiting factor.

OBSERVATION TABLE

Operation	Base Hardware Performance	Estimated s/w overhead	Estimated result	Actual Result
Measurement overhead	N/A	**16ns**	**16ns**	**20.8ns**
Procedure call overhead	N/A	**1 cycle for each argument (0.4ns) 1 cycle to call (jmp instruction)**	**0: 0.4ns 1: 0.8ns 2: 1.2ns 3: 1.6ns 4: 2.0ns 5: 2.4ns 6: 2.8ns 7: 3.2ns**	**0 : 0.1ns 1 : 0.3ns 2 : 1.0ns 3 : 1.35ns 4 : 1.66ns 5 : 2.07ns 6 : 2.23ns 7 : 2.52ns**
System call overhead	N/A	**few us**	**few us**	**2.6 us (60ns cached)**
Process Creation Thread Creation	N/A	**100us**	**100us**	**115.45 us**
	N/A	**5us**	**5us**	**11.678us**
Process ctxt switch Thread ctxt switch	N/A	**2us**	**2us**	**0.98us**
	N/A	**200ns**	**200ns**	**480ns**
RAM Access time	L1 : 1.2 ns L2: 3.6 ns L3: 8.8ns RAM: ~50ns	**One Cycle for the caches: 0.4ns Two cycles for the main memory: 1.2ns**	**L1: ~2ns L2: ~4ns L3: ~12ns RAM: ~50ns**	**L1: 5ns L2: 6.35ns L3: 12.1ns RAM: 40ns**
RAM Bandwidth	21.3GB/s	**5GB/s**	**15GB/s**	**8.9GB/s**
Page Fault service Time	17.1ms	**200us**	**17.3ms**	**18.1ms**