Performance Measurement of Personal Computer

Anubhab Majumdar
Department of Computer Science
North Carolina State University
Email: amajumd@ncsu.edu
Unity ID: amajumd

Arun Jaganathan
Department of Computer Science
North Carolina State University
Email: ajagana@ncsu.edu
Unity ID: ajagana

***Abstract*—In this report we describe a set of experiments performed to benchmark the performance of a typical personal computer. The benchmarking is not a performance estimation based on hardware specification - we have measured the different overheads levied by the OS and determined the hardware constrains to gain a true perspective of the system performance. The report details out the experiments and analyzes the results to draw conclusions about the performance of the system being tested.**

I. Introduction

Specifications of a computer system does not always convey it's true performance. The hardware and operating system introduces various overheads which constrains performance. The measure of these overheads are important as they help benchmark the true performance of any system. This knowledge is important for developers as their application runs atop the operating system and heavily uses the services provided by it; thus any bottlenecks in the OS will translate into their applications and degrade performance. Also, the OS determines the baseline "responsiveness" the user expects from the system and applications should not be far from this baseline to ensure smooth customer experience.

A. Goals

Our primary goal was to benchmark the CPU, OS services and memory in details and analyze the results to draw conclusions about their performance. The CPU and OS services experiments are designed and implemented by Anubhab Majumdar. Arun Jaganathan designed the experiments to test memory components and implementation was shared by both the authors.

B. Language

We used trusty C language to design and implement the experiments. The code was compiled with **Apple LLVM version 7.3.0 (clang-703.0.31)** with no optimizations because we wanted the assembly code to be in order of our original program.

C. Duration

We worked on this project for around 70 hours spanning over 3 weeks. This includes determining the deliverables, reading relevant research papers, designing the experiments, coding the experiments, data consolidation, analysis of the data and drafting this report.

II. Machine Description

We have tested one of our personal computer, a MacBook Air. Following are the details of the machine:

1) **Model Name**: MacBook Air
2) **Model Identifier**: MacBookAir7,2
3) **Processor Name**: Intel Core i5
4) **Processor Speed**: 1.6 GHz
5) **Number of Processors**: 1
6) **Total Number of Cores**: 2
7) **L2 Cache**: 256 KB (per core)
8) **L3 Cache**: 3 MB
9) **Memory**: 8 GB
10) **Memory Type**: DDR3
11) **Memory Speed**: 1600 MHz
12) **Memory bus speed**: 1066 MHz
13) **Link Speed**: 5.0 GT/s
14) **Link Width**: x4
15) **Storage**: 128 GB
16) **Medium Type**: Solid State Drive
17) **Operating System**: MacOS Sierra (Version 10.12)

III. Experiments

The experiments are divided into two broad categories:

- CPU, scheduling and OS services experiments
- Memory experiments

Each of these categories are aggregation of small experiments that measure various overheads associated with OS or constraints of hardware. The following subsections describes the methodology, presents the findings and draw inference about it.

Before we dive into explanation about the experiments, we would like to explain the units of measurement. We have measured the operations in term of **cycles** and **time**. CPU cycles were measured using the C function **rdtsc** (Read Time Stamp Counter) to read the CPU cycles before and after any operation and the difference is assumed as the number of cycles consumed by the operation. We should also mention that before using rdtsc, we have used the C function **cpuid** to prevent any out of order execution by CPU. Similarly, time was measured in microseconds by using the **gettimeofday** function before and after an operation and the difference is considered as the time taken to perform the task.

Another important point is that before executing the experiments we have restricted number of active cores in our system from 4 to 1 (see Fig.1).

For some operations like process creation or context switch, we couldn't estimate a base hardware performance because it is difficult to make an educated guess about how much cycles fork() or pipe() is actually consuming. In these cases we have marked the hardware base performance as "Undefined".

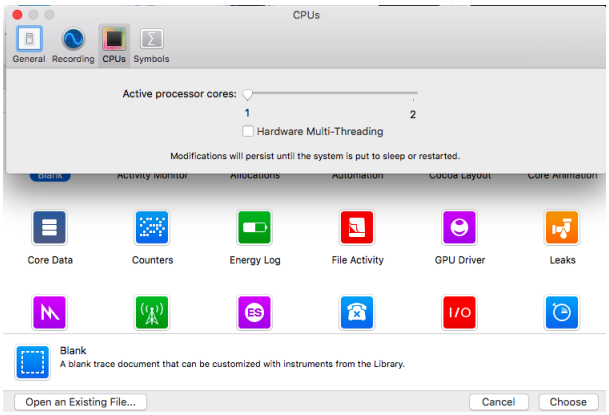

Fig. 1. Restricting use of multi-core

A. CPU, scheduling and OS services experiments

The experiments are conducted to measure 5 key overheads. Their names and descriptions are listed the subsections below.

1) Measurement Overhead: We start our benchmarking experiments by measuring overhead to perform 2 basic tasks - reading from memory and looping through multiple iterations of a task.

a) Methodology: For reading time, we allocated a character array of size 10,000 bytes and read them one by one. The experiment was to measure, in terms of cycles and time, the cost of reading these 10,000 characters. At first, the cost of cpuid and that of running the loop without reading any character is measured. The program is executed 100 times separately using bash script and the readings are noted. Next, the file read operation of 10,000 characters from the array is performed. The time-stamp counter values are measured just before and just after read operation. The difference is noted in a file. This, again, is executed separately 100 times and the results are noted. The mean and median are calculated on the results and noted.

In a similar fashion, the cost is measured in units of time as well. We replaced rdtsc with gettimeofday function and calculated the difference of time measured in microseconds. The results are noted in sheet "Reading Time" in measurements.xls file.

b) Results: The RAM speed of the machine is 1600 MHz (as specified in System Information). Thus to perform 1 read operation it should take 0.6 ns. For reading 10,000 characters it should take 6.25 μs. We estimated it would take thrice the time. We considered page fault, cache miss, TLB miss for overhead and hardware pre-fetching as advantage.

Base Hardware Performance	6.25 μs
Estimated Overhead	12 μs
Predicted Performance	18.75 μs
Measured Performance	21.65 μs
Standard Deviation	3.047038635

From the above table we could see our predicted read time is quite close to measured time. The difference could be because of our estimate of page faults handling time. We believe the results are quite accurate as we have measured, to the best of out ability, only the read time and subtracted the for loop time from the results.

2) Loop Overhead:

a) Methodology: In this experiment we had to "report the overhead of using a loop to measure many iterations of an operation". This experiment is quite similar to the previous one. We wrote a code which contains a for-loop "looping" 10,000 times. We measured the cost, in terms of both cycle and time, twice - once by commenting the for-loop part and once by keeping the for-loop. No OS overhead will be incurred as no system call is made.

b) Results: The CPU speed is 1.6 GHz, i.e., it takes 0.6 ns to run one CPU cycle. The assembly code replacing the for-loop had 4 instructions. We estimated one cycle is required for each instruction.

Base Hardware Performance	24 μs
Estimated Overhead	No OS overhead
Predicted Performance	24 μs
Measured Performance	24.33 μs
Standard Deviation	2.065493313

From the above table we could see our predicted read time is quite close to measured time. We believe the results are quite accurate as we have measured, to the best of out ability, only the for-loop time and subtracted any extra time we measured along with the for-loop (like _cpuid function cost).

3) Procedure Call Overhead:

a) Methodology: Next, we had to measure procedure call overhead. 8 different procedures needed to be tested, each with different number of arguments. The functions themselves do not do anything, they return as soon as they are called. The cost of calling such functions are measured one by one in cycles.

b) Results: The main estimation we had to make was about how much time it takes for the control switch from caller to callee and again come back to the caller function after callee returns. We predicted the function call itself will take 1 cycle and the return from function will take 1 cycle.

Base Hardware Performance	2 cycles
Estimated Overhead	8 cycles
Predicted Performance (0 argument)	10 cycles
Measured Performance (0 argument)	14.33 cycles
Standard Deviation	119.4266145
Measured Performance (1 argument)	15.32 cycles
Standard Deviation	66.55330969
Measured Performance (2 argument)	1.13 cycles
Standard Deviation	35.80560731
Measured Performance (3 argument)	45.3 cycles
Standard Deviation	84.10785524
Measured Performance (4 argument)	31.9 cycles
Standard Deviation	91.47859495
Measured Performance (5 argument)	52.31 cycles
Standard Deviation	272.4169967
Measured Performance (6 argument)	68.3 cycles
Standard Deviation	329.6719367
Measured Performance (7 argument)	39.75 cycles
Standard Deviation	64.24013897

From the above table we could see our predicted read time is quite close to measured time. We believe the results are quite accurate as we have measured, to the best of out ability, only the function call time and subtracted any extra time we measured along with the function call (like _cpuid function cost). We can observe from the table that with increase in the number of arguments, the cost associated with the function call increases, though not linearly. We attribute the uneven pattern in measured performance to optimizations the CPU may have performed. The additional cost of adding another argument is measured as 3.63 cycles. This is the mean of the difference in cost between functions with one argument more than the other.

4) System Call Overhead:

a) Methodology: The purpose of this experiment was to measure a minimalistic system call cost. We choose to test with the **time** system call. It is lightweight and the result is never cached. So it served out purpose perfectly. The procedure of measuring cost is similar to the previous two. We measure cost by running the code twice, once commenting the system call and once letting it run.

b) Results: The main estimate we had to make was the OS TRAP and OS TRAP handler time and it was difficult to estimate as we didn't had information about the TRAP handler code. We estimated it to be more than procedure call. For the procedure call we were getting results less than 1 μs. So we predicted the performance to be little more than 1 μs with OS overhead.

Base Hardware Performance	Undetermined
Estimated Overhead	1 μs
Predicted Performance	Greater than 1 μs
Measured Performance	2.06 μs
Standard Deviation	1.096366819

We believe the results are quite accurate as we have measured, to the best of out ability, only the system call time and subtracted any extra time we measured along with the system call (like _cpuid function cost). Also since time system call is not cached, repeated execution of the program does not affect result.

5) Task creation and running time:

a) Methodology: In this experiment, we measured the cost associated with process creation. We measured the time/time-stamp-counter before and after fork() system call. The difference will provide the task creation time. For measuring the running time, we included a for loop of 100 cycles in the child process followed by exit(0). Also, we added wait(childId) in the parent before measuring the time again. This forces the parent to wait for the child to exit before measuring the time for the second time. Thus this time we are measuring the child process running time.

b) Results: We predicted the process creation and running to be a costly affair as the entire code and data needs to be copied for child process creation. We couldn't come up with a logical estimate for the time it would take to create or run a process. What we could guess is fork() would be more costly that time() system call we used before as it is not a minimalistic system call.

Base Hardware Performance	Undetermined
Estimated Overhead	20 μs
Predicted Performance	Greater than 20 μs
Measured Performance	153.41 μs
Standard Deviation	60.707349

Base Hardware Performance	Undetermined
Estimated Overhead	200 μs
Predicted Performance	Greater than 200 μs
Measured Performance	522.41 μs
Standard Deviation	1121.665014

The details of all the measurement is present in the Process creation and Process running tabs of measurement.xls file. There we can see that the measured performance sometimes varies between individual executions. We attribute this variation to the non-determinism surfacing from scheduler.

6) Thread creation and running time:

a) Methodology: This is very familiar to the last experiment except one difference - instead of creating and running a new process we need to create and run a new thread performing the same task as previous. The cost measurement process is exactly the same.

b) Results: Thread is a lightweight process. Thus we estimated its cost to be much less than process.

Base Hardware Performance	Undetermined
Estimated Overhead	10 μs
Predicted Performance	Greater than 10 μs
Measured Performance	21.34 μs
Standard Deviation	2.877621773

Base Hardware Performance	Undetermined
Estimated Overhead	50 μs
Predicted Performance	Greater than 50 μs
Measured Performance	25.5 μs
Standard Deviation	10.69804098

We believe our measurements to be quite accurate as we were careful in subtracting any extra instructions within our time measuring functions.

7) Context switch time:

a) Methodology: In this experiment we measured the context switch time using the pipe system call. First, we created a pipe in parent process and used fork() to create a child process. Inside the child process we closed the input side of pipe, measured the time/time-stamp-counter and sent that value through the pipe and exit. Meanwhile, in the parent process, we close off the input side of the pipe and read the value sent by the child process. Once we get the value, we immediately measure time/time-stamp-counter value again and subtract the one we received from this. Here, the parent process has to wait till the child process sends the value, thus eliminating the need to include wait(childID) call. Also we are measuring the final time in parent process but only after child process sends it's data. This ensures a context switch has taken place and we have measurements before and after it takes place.

b) Results: We have been very careful in measuring the cost in this experiment. We have measured, individually, the time taken to send and receive message from pipe. We have subtracted this costs from our measurement to ensure we achieve correct results. We also estimated software overhead to be more than thread running overhead but less than process running overhead.

Base Hardware Performance	Undetermined
Estimated Overhead	150 μs
Predicted Performance	Greater than 150 μs
Measured Performance	161.83 μs
Standard Deviation	72.20929947

B. Memory experiments

1) RAM access time: The machine we are testing contains L1, L2 and L3 cache. The size of the caches are already mentioned in the machine description part. However, during execution we are reducing number of active core to 1, thereby reducing the L2 and L3 cache to 128KB and 768KB respectively.

a) Methodology: We measured the "back-to-back" load latency of arrays ranging from 2KB to 2MB and plot the curve of log2(ARRAY_SIZE) VS the load latency. We then divide the graph into 3 parts, using our knowledge of cache size, and try to estimate the latency time of each part - which provides us the latency of caches and memory.

First, we allocate the required memory for the array. Next, we read each array element and assign unique values to them. This is done to make sure all array elements are in cache and all page faults are handled. This ensures that page fault handling time won't interfere with our back-to-back load latency measurement.

Now that our data is in cache, we start accessing them again. This time we are accessing them from caches of different level (depending on array size they would be spread across different caches). However, while accessing them to read their values, we also measure the time required to access them each. The individual access time of all the elements are added and written in an output file along with the log2 of array size. This data is used to plot the graph.

b) Results: The graph (Fig. 2) is an exponentially increasing graph. The inference drawn by us are:

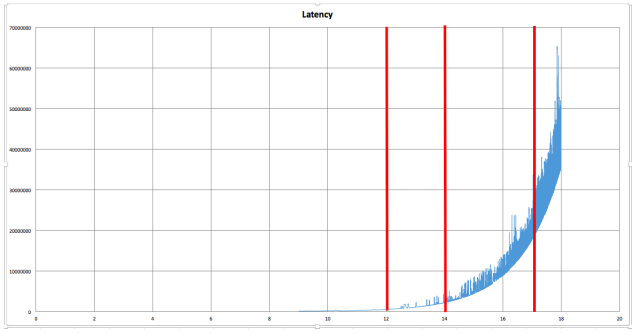

Fig. 2. Latency curve

- The graph is increasing because the array size is increasing which needs more time to read all its values
- The graph is not increasing linearly. What this means is that the time taken to access 2x elements is not double of accessing x elements, it is more than that. Infact, this difference increases in the border of cache sizes, clearly demarcating the fact that the latency between different levels of cache are quite prominent.

2) RAM bandwidth: Memory bandwidth is defined as the rate at which data can be read or written into the memory. The memory bandwidth is measured in bytes/second. But in modern RAMs, the read and write bandwidth are very high, that we have started measuring them in Gigabytes/seconds. For DDR3 RAMs, we can expect a transfer rate as high as 15 gigabytes/second with DDR3 RAMs, under ideal conditions. But as there are overheads to be considered, we can still expect a read rate of about 10 GB/s and write rate of about 1 GB/s for DDR3 RAM with 1600 Mhz.

The software overheads to be considered for measuring memory bandwidth would be the general error of about 200 microseconds (for an array of 8MB) and the latency in reading and writing. Since writing into the array takes more cycles, the write speed will be lower than the read speed. Estimating the overhead in terms of time is not that easy we are handling huge amount of data. Thus we are guessing the performance after considering the above mentioned overheads. The practical read bandwidth can be expected about 10GB/s against the theoretical 12.5 GB/s, as there will be always a small latency when accessing. Similarly, the write bandwidth can be expected to be about 8 Gb/s against the theoretical 12.5Gb/s.

a) Methodology: We classify the memory bandwidth into two components - Read bandwidth and Write bandwidth. In general, the read bandwidth will be higher than the write bandwidth. We have written a program that creates a very big array (of size about 8MB). The write bandwidth is computed by finding the time taken in assigning the entire array (of size 8MB) with some random value. i.e., we start the timer before the assignment and end the timer after the values are

assigned, and we find the time difference. Using this value, we can compute the write speed of our RAM.

Similarly, for computing the read speed, we just use the same concept as above, and we try to read the value of the array elements individually and assign into a variable.

b) Results: We have found the read and write transfer rates, i.e the read/write bandwidth of our RAM. As expected, the read speed is higher than the write speed (approximately twice).

Write speed is noted below:

Base Hardware Performance	12.5 GB/sec
Predicted Performance	8 GB/s
Measured Performance	1.36 GB/s

The read speed is noted below:

Base Hardware Performance	12.5 GB/s
Predicted Performance	10 GB/s
Measured Performance	3.11 GB/s

3) Page fault service time: The page size was found to be 4096 Bytes. Page fault service time is the time taken to allocate an entire page into the cache/ram, from the disk. Whenever a page fault occurs i.e., when the file is not present in the memory, there will be a flag raised and the data will be fetched from the disk into the memory. The software overheads to be considered for this would be the delay in fetching the data from the disk, and the speed of the disk also influence the time.

a) Methodology: We create a character pointer and using the mmap() function, we are allocating memory equivalent to the size of the page to this pointer. We mentioned the start address and the size (4096Bytes) and we are finding the time taken to execute this function. The pointer resides in the memory and thus, it can be used to estimate the service time. To find the time taken to access a single byte, we divide it by the size of the page.

b) Results: After running the program for several times, the page fault service time comes to about 21 ms, which is more than what we guessed.

Base Hardware Performance	8 μs
Estimated Overhead	5-7 μs
Predicted Performance	13-15 μs
Measured Performance	21 μs

Therefore page fault service time for 4096 bytes is 21 μs, i.e., 5.1 ns for 1 byte. Comparing this to our read time experimental results, it takes 21.65 μs to read 10,000 characters from RAM. This averages to 2.1 ns per byte.

IV. Conclusion

The experiments provide results which validates many of our theoretical understandings of how an operating systems works in tandem with hardware and the close relation between them.

The results from all the experiments are summarized below:

Operation	Base Hardware Performance	Estimated Overhead	Predicted Performance	Measured Performance
Loop	24 μs	NA	24 μs	24.33 μs
Read	6.25 μs	12 μs	18.25 μs	21.65 μs
Procedure call	2 cycles	8 cycles	10 cycles	14.33 μs
System Call	Undetermined	1 μs	>1 μs	2.06 μs
Process Creation	Undetermined	20 μs	>20 μs	153.41 μs
Process running	Undetermined	200 μs	>200 μs	522.41 μs
Thread creation	Undetermined	10 μs	>10 μs	21.34 μs
Thread running	Undetermined	50 μs	>50 μs	25.5 μs
Context switch	Undetermined	150 μs	>150 μs	161.83 μs
RAM bandwidth (R)	12.5 GB/s	4 GB/s	8 GB/s	1.36 GB/s
RAM bandwidth (W)	12.5 GB/s	2 GB/s	10 GB/s	3.11 GB/s
Page fault	8 μs	5-7 μs	13-15 μs	21 μs

For a more visual comparison, we have plotted the CPU and OS services experimental results in a graph. (See Fig. 3)

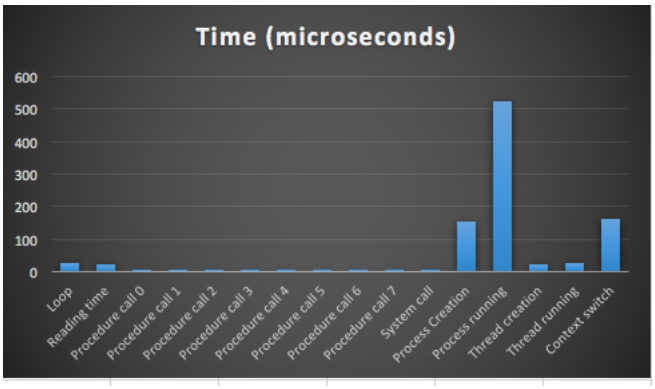

Fig. 3. Results of CPU experiments

The graph clearly shows the contrast between cost associated with the different tasks.
Some noteworthy points from the doing the whole project:

- Process creation and process running is costliest of the tasks
- Thread is lightweight process - its cost is much less than process
- Context switch is costly, but less than process creation or running
- Reading time from memory is higher compared to procedure calls
- Procedure calls are cheaper, but their cost increases with increasing parameters
- System calls vary by cost; minimalist ones like time() takes less time where as fork() takes much longer to finish
- Memory read bandwidth is greater than read. Thus read takes less time than writing to memory
- Latency increases as level of cache increases. Reading from caches take much lesser time than reading from RAM.
- Page fault cost is large. Infact, we saw through experiment results that it is much costlier than RAM read time.

REFERENCES

[1] L. McVoy and C. Staelin, *lmbench: Portable Tools for Performance Analysis*, San Diego, California, January 1996

[2] Page Fault, https://en.wikipedia.org/wiki/Page_fault

[3] Understanding mmap(), http://pubs.opengroup.org/onlinepubs/009695399/functions/mmap.html

[4] Memory Bandwidth, https://en.wikipedia.org/wiki/Memory_bandwidth

[5] Understanding RAM, http://www.bleepingcomputer.com/tutorials/identifying-and-upgrading-ram/