

# Performance Measurement of Personal Computer

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

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

**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 constraints 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".

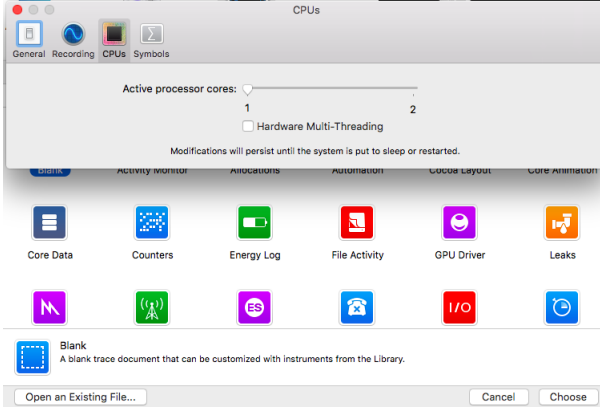


Fig. 1. Restricting use of multi-core

#### 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 initialized 'a' in all the locations. 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 is measured using rdtsc function. The program is executed 100 times separately using bash script and the readings the noted. Next, the file read operation of 100,000 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  $\mu$ s. We estimated it would take thrice the time. We considered page fault, cache miss, TLB miss for overhead and hardware pre-fetching as advantage.

<b>Base Hardware Performance</b>	6.25 $\mu$ s
<b>Estimated Overhead</b>	12 $\mu$ s
<b>Predicted Performance</b>	18.75 $\mu$ s
<b>Measured Performance</b>	21.65 $\mu$ s

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.

<b>Base Hardware Performance</b>	24 $\mu$ s
<b>Estimated Overhead</b>	No OS overhead
<b>Predicted Performance</b>	24 $\mu$ s
<b>Measured Performance</b>	24.33 $\mu$ s

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.

<b>Base Hardware Performance</b>	2 cycles
<b>Estimated Overhead</b>	8 cycles
<b>Predicted Performance (0 argument)</b>	10 cycles
<b>Measured Performance (0 argument)</b>	14.33 cycles
<b>Measured Performance (1 argument)</b>	15.32 cycles
<b>Measured Performance (2 argument)</b>	1.13 cycles
<b>Measured Performance (3 argument)</b>	45.3 cycles
<b>Measured Performance (4 argument)</b>	31.9 cycles
<b>Measured Performance (5 argument)</b>	52.31 cycles
<b>Measured Performance (6 argument)</b>	68.3 cycles
<b>Measured Performance (7 argument)</b>	39.75 cycles

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  $\mu s$ . So we predicted the performance to be little more than 1  $\mu s$  with OS overhead.

<b>Base Hardware Performance</b>	Undetermined
<b>Estimated Overhead</b>	1 $\mu s$
<b>Predicted Performance</b>	Greater than 1 $\mu s$
<b>Measured Performance</b>	2.06 $\mu s$

We believe the results are quite accurate as we have measured, to the best of our 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 than `time()` system call we used before as it is not a minimalistic system call.

<b>Base Hardware Performance</b>	Undetermined
<b>Estimated Overhead</b>	20 $\mu s$
<b>Predicted Performance</b>	Greater than 20 $\mu s$
<b>Measured Performance</b>	153.41 $\mu s$

<b>Base Hardware Performance</b>	Undetermined
<b>Estimated Overhead</b>	200 $\mu s$
<b>Predicted Performance</b>	Greater than 200 $\mu s$
<b>Measured Performance</b>	522.41 $\mu s$

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.

<b>Base Hardware Performance</b>	Undetermined
<b>Estimated Overhead</b>	10 $\mu s$
<b>Predicted Performance</b>	Greater than 10 $\mu s$
<b>Measured Performance</b>	21.34 $\mu s$

<b>Base Hardware Performance</b>	Undetermined
<b>Estimated Overhead</b>	50 $\mu s$
<b>Predicted Performance</b>	Greater than 50 $\mu s$
<b>Measured Performance</b>	25.5 $\mu s$

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 its 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.

<b>Base Hardware Performance</b>	Undetermined
<b>Estimated Overhead</b>	150 $\mu s$
<b>Predicted Performance</b>	Greater than 150 $\mu s$
<b>Measured Performance</b>	161.83 $\mu s$

#### B. Memory experiments

##### 1) RAM access time:

a) Methodology:

b) Results:

##### 2) RAM bandwidth:

a) Methodology:

b) Results:

##### 3) Page fault service time:

a) Methodology:

b) Results:

#### 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 $\mu s$	NA	24 $\mu s$	24.33 $\mu s$
Read	6.25 $\mu s$	12 $\mu s$	18.25 $\mu s$	21.65 $\mu s$
Procedure call	2 cycles	8 cycles	10 cycles	14.33 $\mu s$
System Call	Undetermined	1 $\mu s$	>1 $\mu s$	2.06 $\mu s$
Process Creation	Undetermined	20 $\mu s$	>20 $\mu s$	153.41 $\mu s$
Process running	Undetermined	200 $\mu s$	>200 $\mu s$	522.41 $\mu s$
Thread creation	Undetermined	10 $\mu s$	>10 $\mu s$	21.34 $\mu s$
Thread running	Undetermined	50 $\mu s$	>50 $\mu s$	25.5 $\mu s$
Context switch	Undetermined	150 $\mu s$	>150 $\mu s$	161.83 $\mu s$
Memory Latency	24 $\mu s$	NA	24 $\mu s$	24.33 $\mu s$
RAM bandwidth	24 $\mu s$	NA	24 $\mu s$	24.33 $\mu s$
Page fault	24 $\mu s$	NA	24 $\mu s$	24.33 $\mu s$

For a more visual comparison, we have plotted the CPU and OS services experimental results in a graph. (See Fig. 2)

The graph clearly shows the contrast between cost associated with the different tasks. Some noteworthy points are:

- Process creation and process running is costliest of the tasks
- Thread is lightweight process - its cost is much less than process

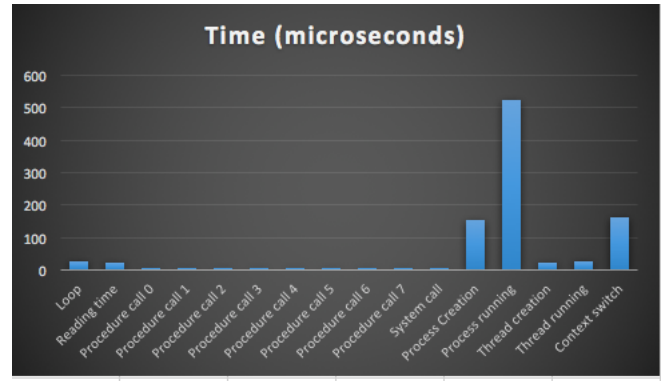


Fig. 2. Results of CPU experiments

- 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

#### REFERENCES

- [1] H. Kopka and P. W. Daly, *A Guide to L<sup>A</sup>T<sub>E</sub>X*, 3rd ed. Harlow, England: Addison-Wesley, 1999.