CSE 221: System Measurement Project

Xinyuan Liang, Siran Ma, Wentao Huang
University of California, San Diego

February 24, 2023

1 Introduction

Operating System performance is a crucial aspect in today's computing environments. It determines the overall effectiveness and efficiency of a computer system in handling various tasks. The performance of an OS is influenced by various factors including the underlying hardware, system design, and workload. Understanding how hardware affects the performance of the OS is important in optimizing and tuning the system to meet specific needs and demands.

This report will focus on exploring the relationship between hardware and OS performance. We will analyze the impact of hardware components such as the CPU, RAM, disk, and network on the performance of the OS and how they contribute to the overall system efficiency.

Our group members include Siran Ma, Xinyuan Liang and Wentao Huang. We conducted experiments on an x86-64 and Linux-based machine. See details in the Machine Description section. We compiled the benchmark programs written in C language with GCC-9.4.0 and disable the compiler optimization with the *O0* option. And the estimated workload is 30 to 40 hours for each team member.

2 Machine Description

We get the basic information about the x86-64 and Linux-based machine by reading the hardware specifications directly or gained from Linux commands.

2.1 Notebook Description

We launch our experiments on an ASUSTeK X510UQ machine.

2.2 Processor

The processor details can be obtained through commands "cat /proc/cpuinfo"(for cpu info) and "lscpu | grep cache"(for cache L1, L2, L3) in the terminal.

The processor we use is Intel(R) Core(TM) i5-7200U @ 2.50GHz x 4.

The cycle time of our processor is 800.000 MHz (with a maximum of 3100.0000 MHz and a minimum of 400.0000MHz).

The size of the L1 cache is 64 KB. Notice that 32 KB of it is for the instruction cache, while the other 32 KB is for the data cache. The size of the L2 and L3 cache are 256 KB and 3 MB.

2.3 DRAM and Memory

The DRAM and memory details can be obtained through command "sudo lshw -C memory" in the terminal.

The DRAM model is HMA81GS6AFR8N-UH SK Hynix 8GB 2400 SODIMM. The DRAM type is SODIMM (DDR4). The DRAM's clock frequency is 2400MHz, therefore each clock cycle will cost around 0.4ns. The DRAM has a capacity of 8GiB (64bits).

2.4 I/O Bus

The I/O bus details can be obtained through the command "lspci -v" in the terminal.

The I/O bus link speed is 6.0Gbps, using SATA Controller.

2.5 Disk

The disk details can be obtained from the hardware specifications.

The SSD model is TOSHIBA THNSNK128GVN8 (K8AS4102), with a capacity of 128GB and a transfer rate of 600 MB/s. Unfortunately, the IOPs and the latency information are not reported on the specifications.

2.6 Network

The network details can be obtained through the commands "sudo ethtool name" and "lwconfig name" in the terminal.

The Ethernet network bandwidth is 1000 Mb/s and the wireless network bandwidth is 400 Mb/s.

2.7 Operating System

The operating system is Ubuntu 16.04.4 LTS.

3 CPU, Scheduling, and OS Services

Our first set of measurements focuses on the performance of the CPU, scheduling, and Operating System services. The performance is presented in terms of the clock cycles it takes to finish the operation. We first describe the methodology, followed by its result and interpretation.

3.1 Measurement Cycle Time

To adopt a high-precision timing measurement, we take the advantage of the constant TSC, which forces the update of the counter at a constant frequency. In this way, we can convert from cycles to wall clock times after we have measured the cycle time. To get a more accurate result, however, the number of cycles will be used as the unit of experiment results.

3.1.1 Measurement Methods

We read the value of the cycle counter twice in one second to calculate how many cycles elapsed within one second. To read the cycle counter, we use the special instruction *RDTSC(P)*. To get a reliable result, we also disable the multi-core processors feature by associating the

measurement task with a single core. Apart from this, we fix the CPU frequency at its highest – 3.1GHz.

3.1.2 Measurement Result

We measure the average cycles within 1 second 100 times and get the following results:

- Estimated cycles in 1 second: 2.5G - 3.1G
- Measured average cycles in 1 second: 2.7G

3.1.3 Interpreting Measurement

Since the CPU frequency written on the Intel Specification is 2.5GHz and we previously set the CPU frequency at 3.10GHz manually (feasible highest value), the results match our expectations.

3.2 Measurement Overhead

3.2.1 Measurement Methods

But, reading from the cycle counter using instructions also brings overhead. We need to measure the measurement overhead to facilitate all future experiments. Using the *RDTSC(P)* benchmarking method described above, we recorded the time from the timestamp register twice without executing any instructions between them. The difference between the two timestamp registers' values represents the overhead of measurement.

Following instructions from the [1], we avoid the interference from the CPU's out-of-order execution feature by carefully crafting the order of executing instructions *CPUID*, *RDTSC*, *RDTSCP*. "*CPUID* is a serialized instruction" and "*RDTSCP* instruction waits until all previous instructions have been executed"[1]. In virtue of these instructions, we manage to achieve a fine-grained measurement of timing overhead.

Besides, different from the [1], we didn't implement a kernel module to disable preemption and turn off interrupts. All the measurements in this report are run in an environment where preemption and interrupts can happen anytime.

3.2.2 Measurement Result

To measure the overhead of timing, we have two nested loops inside the measurement. We wrote a loop with no instruction executing other than *RDTSC(P)* and execute it 1000 times. We tested the testbench 1000 times (outer loop), each testbench includes 1000 loops (inner loop). Using these 1000 different testbench results, we are able to compute the average time cost as well as the standard deviation among testbench results to judge whether these results are reliable.

- Base hardware performance: 0 clock cycles
- Estimated software overhead: about 30 clock cycles[3]
- Measured performance average: 21.92 clock cycles
- Measured performance standard deviation among single testbench: 2.62 clock cycles
- Measured standard deviation across 1000 testbenches: 0.5 clock cycles
- Measured performance maximum deviation: 7629 clock cycles

Table 1: Procedure Call Overhead

argc	avg	std1	std2	max dev
0	32.89	5.74	1.63	22125
1	33.14	2.14	0.91	1662
2	33.37	2.29	0.58	6761
3	33.85	3.36	0.98	17926
4	33.85	3.56	1.31	31225
5	34.37	2.10	0.63	1210
6	34.67	3.67	1.65	11545
7	36.02	4.34	2.03	27811

3.2.3 Interpreting Measurement

Referring to Appendix C - General Purpose Instructions Latency and Throughput from [3], the throughput of RDTSC(P) instruction is around 30 clocks. Therefore, we assume the estimated measurement overhead is around 30 clocks, too. Since in most case, the latency of executing the instruction should be shorter than its throughput, it's acceptable that the measured overhead is a bit shorter than estimation.

3.3 Procedure Call Overhead

3.3.1 Measurement Methods

In the testbench, we created 8 minimum procedures with 0-7 arguments to test procedure call overhead of different parameter sizes. We inserted *RDTSCP* instruction before and after the procedure call. We iteratively conducted this testbench for 1000 times.

3.3.2 Measurement Result

To measure the overhead of the procedure call, we still adopt two nested loops inside the measurement (1000 * 1000). The results of the 8 procedure calls whose number of arguments ranges from 0 to 7 are listed in the table 1 All the time cost in the table is represented in clock cycles. "std1" stands for performance standard deviation within a single testbench, "std2" stands for standard deviation across all testbenches, and "max dev" stands for performance maximum deviation. All the tables afterward will follow these terms.

- Base hardware performance: 0 clock cycles
- Estimated software overhead: Varies from the number of arguments passed into the procedure calls

3.3.3 Interpreting Measurement

Linux operating system uses stack and registers to store procedure call arguments, the caller procedure store the first 6 arguments to the %rdi, %rsi, %rdx, %rcx, %r8 and %r9 registers respectively, further arguments are stored on the stack. Thus, the more arguments the procedure has, the more *MOV* instructions are executed, making the procedure call overhead increases as the number of arguments increases. According to [3], the latency of MOV instructions is 0.5 clocks. The result also shows the trend in this manner.

3.4 System Call Overhead

3.4.1 Measurement Methods

As mentioned in the project description, some operating systems will cache the results of some system calls (e.g., idempotent system calls like *getpid()*), so only the first call by a process will actually trap into the OS. The system call *times()*, however, return current process times, which can't be cached by OS. In the testbench, we get the timestamps before and after the system call times. We iteratively conducted the testbench for 1000 times.

3.4.2 Measurement Result

To measure the overhead of the syscall call, we still adopt two nested loops inside the measurement (1000 * 1000).

- Base hardware performance: 0 clock cycles
- Estimated software overhead: Varies from different system calls. But we assume it would be a much larger cost compared with procedure calls
- Measured performance average: 739.59 clock cycles
- Measured performance standard deviation among single testbench: 61.25 clock cycles
- Measured standard deviation across 1000 testbenches: 22.88 clock cycles
- Measured performance maximum deviation: 5012 clock cycles

3.4.3 Interpreting Measurement

Procedure calls store arguments, save registers and execute a *CALL* instruction by jumping to another address and continue executing. On the other hand, system calls need to trap into the operating system, which requires much more overhead.

3.5 Task Creation Time

3.5.1 Measurement Methods

To measure the time to create and run a process, we used the system call *fork()* to create a child of the current process, which returns directly. In the testbench, we inserted *RDTSCP* instruction before and after the system call fork and iteratively conducted this testbench for 1000 times.

According to the project description, kernel threads run at user-level, but they are created and managed by the OS, e.g. *pthread_ create()* on Unix creates a kernel-managed thread. We used *pthread_ create()* to create and run a kernel thread, which returns immediately. By adding *RDTSCP* before and after the creation, we calculated the time used to create a kernel thread.

3.5.2 Measurement Result

To measure the overhead of the creation of processes and kernel threads, we adopt two nested loops inside the measurement (100 * 100) for each of them. The iteration time becomes smaller because those operations consume longer cycle time compared with previous operations. The results are shown in the table 2 for comparison.

Table 2: Task Creation Overhead

type	avg	std1	std2	max dev
Process	329672.25	2233449.63	152089.56	66203661
Kernel thread	54394.91	278872.91	63085.16	27198348

- Base hardware performance: 0 clock cycles
- Estimated software overhead: 70,000 clock cycles for process creation and a much smaller time consumed by the thread creation

3.5.3 Interpreting Measurement

Comparing the process and the thread creation overhead, it is obvious that the cost of thread creation is cheaper compared to process creation because threads are lightweight and share the same memory space and system resources as other threads within the same process. In other words, creating a new thread does not require the operating system to allocate new memory or resources for the thread. The result above indeed shows that kernel thread creation is much more lightweight than process creation.

For the estimated process creation performance, we referred to the result in [2]. It's shown that for Linux-i686, process creation takes 100 times longer than system calls. Therefore we estimate process creation would take around 70,000 clock cycles. We also assume that the reason why the measured result is much shorter than the estimation may be due to the performance improvement in recent Linux Operating Systems. Note that since our experiment is taken in the condition where preemptions and interrupts may happen frequently, the absolute cycles it takes to finish the procedure call are much more than the results in the paper[2].

3.6 Context Switch Time

3.6.1 Measurement Methods

In general, we used blocking pipes to force context switches. A blocking pipe, when one process tries to read from one end of an empty pipe, will be blocked until the data is available. Therefore, we built two pipes, named as child pip and parent pip, to force the two processes to ping-pong. For example, the child process would first try to read from the parent pip and get blocked since it's empty. The CPU will switch to the parent process (ideally) to allow it to write into the parent pip. Then the parent process tries to read from an empty child pip and go to sleep. The process will now be switched back to the child process. To measure the time cost, we collected the timestamp before the parent write the parent pip and after it read the child pip. For a single ping-pong event, it performs two context switches. Note that the read and write operations on pipes themselves have time consumption overhead. So we also have to measure the read and write overhead.

Measuring thread context switch performance is quite similar to the process one, we created threads instead of processes.

3.6.2 Measurement Result

To measure the performance of the context switch, we adopt two nested loops inside the measurement (100 * 100) for each of them. The results are shown in the table 3 for comparison.

Table 3: Context switch Overhead

type	avg	std1	std2	max dev
Process	3100.88	168.47	180.63	19068
Kernel thread	2950.13	116.63	32.10	11648
R/W pipe	2353.10	2624.32	2370.52	2362404

- Base hardware performance: 0 clock cycles
- Estimated software overhead: 1500 clock cycles for process context switch and a smaller time consumed by the thread context switch

3.6.3 Interpreting Measurement

The overhead of a context switch depends on the resources that it needs to save and restore. As described above, a thread context switch overhead is lower compared to a process context switch. Since a thread shares the same memory address space and other system resources with other threads within the same process. Therefore, fewer resources need to be saved and restored during a thread context switch. On the other hand, a process context switch needs to take care of memory space and system resources like file descriptors, page tables, etc.

We also referred to the result in [2] and estimated the process context switch takes about 1500 clock cycles (twice as executing system calls). However, it turns out that the context switch consumes a shorter time. Besides, the overhead of reading and writing into pipes occupied a higher proportion in the context switch. It's also mentioned in [2] that the pipe overhead may vary between 30% and 300%. Nevertheless, the kernel thread context switch is indeed more lightweight than the process context switch.

4 Memory

Our second set of measurements focusing on memory performance, including memory access time, memory bandwidth and page fault service time. The performance will be presented in terms of the throughput and cycles.

4.1 Memory access time

In this subsection, we are going to explore the memory access performance under different access pattern. Modern processors have sophisticated memory hierarchy with several cache levels, and by tweaking the test program's locality, both spatial and temporal, and measuring the memory access throughput, we are going to see a rough picture how the hierarchy looks like.

4.1.1 Measurement Methods

We are going to write a program which issues a tight sequence of reading operations, and the measured read throughput would show the performance of the memory system for that specific sequence of memory access. A specific sequence of memory access means specific working set size, i.e. the array size we are going to access, and specific stride, i.e. the distance between two memory accesses. As mentioned above, cache exploits both spatial and temporal locality to improve performance, and by changing working set size, we are able to change temporal locality,

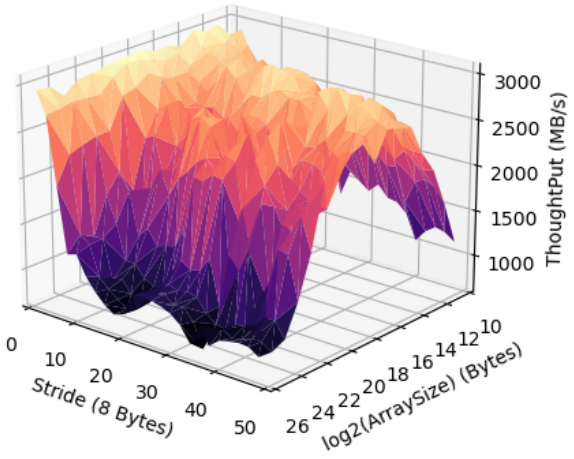

Figure 1: Memory Mountain

the larger the working set is, the worse the temporal locality would be. And the same to the stride, the larger the stride is, the worse the spatial locality would be. Finally, by measuring the cycle required to finish the specific memory access pattern, we can get the throughput given the CPU frequency and the working set size. Given the hardware information of our machine, 32KB L1 data cache, 256KB L2 cache and 3MB L3 cache, the working set size, i.e. the array size is ranging from 1KB to 64MB, the stride is ranging from 16 bytes to 384 bytes.

4.1.2 Measurement Result

Combining different array sizes and strides, we get the measurement results as shown in the Figure 1 and Figure 2.

4.1.3 Interpreting Measurement

Though less obvious and straight forward, the Figure 1 shows that as the stride increases, the overall throughput decreased due to the worse spatial locality. And the Figure 2 shows that the throughput decreases dramatically when the array size crosses the cache size boundary, e.g. the increasing trend slows down around 32KB array size, the throughput decreases around 128KB and 256KB array size, and the throughput decreases tremendously around 2MB and 4MB array size, meaning that the memory access now needs to go down to the memory.

The graphs also show some unexpected behavior. For small strides, e.g. 16 bytes and 32 bytes, the reading speed stays still as the array size increases. We assume that this is because of the cache line prefetching behavior and consistent well-locality through out the whole experiment. Also, for large strides, e.g. 384 bytes, the throughput increases as the array size increases when the array size is smaller than the L1 cache. We assume that the temporal locality is counter-intuitively increased when the array size has not reached the cache size.

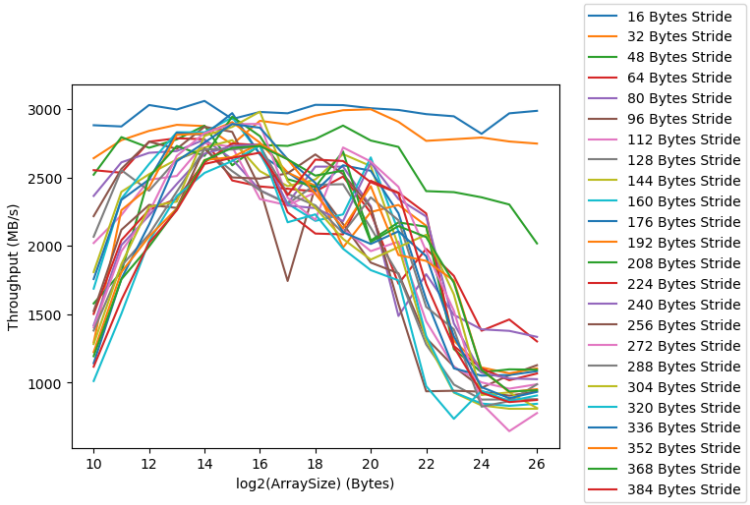

Figure 2: Result Graph

4.2 Memory bandwidth

This section reports the bandwidth of reading and writing RAM. RAM bandwidth is defined as the rate at which data can be read from or stored in the memory by a processor. RAM bandwidth is usually expressed in units of GB/s, but we choose to use MB/s for consistency.

4.2.1 Measurement Methods

Inspired by the Memory Measurement introduced by Section 5.1[5], we adopted the method of the hand-unrolled loop to avoid interleaving instruction and delayed read requests to memory. In order to test the memory bandwidth rather than the cache bandwidth, we are supposed to ensure that the destination data stored in memory would not be cached after each read or write. We accessed the memory at a stride of 64 bytes to avoid cache line prefetch, which is the cache line size of the CPU. Besides, considering that our L3 cache's size is 3MB, we defined two arrays of 1.5MB size and use *memcpy()* to copy one array to another before benchmarks. Thus we can expect that both arrays would be in the L3 cache and there would be no extra space for the later caching of reads and writes. In this way, we ensure that each read and write can be seen as one memory access.

We allocated a 4MB array for testing. We choose this size because it is larger than the L3 cache size, which is around 3MB. When testing the write bandwidth, we assign values to the array's elements. When testing the read bandwidth, we read values from the array's elements and add them together. According to the paper[5], it can avoid C compiler optimization to affect the result. We loop the above process 100 * 100 times to compute the average throughput and deviation of memory access time for each testbench.

4.2.2 Measurement Result

Read Memory

- Base hardware performance: 38,400MB/s

- Estimated software overhead: about 28,800MB/s
- Predicted time: about 9600MB/s
- Measured performance average: 4757.95 MB/s
- Measured performance standard deviation among single testbench: 56.21
- Measured standard deviation across 1000 testbenches: 40.78
- Measured performance maximum deviation: 1438

Write Memory

- Base hardware performance: 38,400MB/s
- Estimated software overhead: about 28,800MB/s
- Predicted time: about 9600MB/s
- Measured performance average: 1300.60 MB/s
- Measured performance standard deviation among single testbench: 140.67
- Measured standard deviation across 1000 testbenches: 77.98
- Measured performance maximum deviation: 1101

4.2.3 Interpreting Measurement

According to wikepedia[6], the theoretical maximum memory bandwidth is calculated by this formula: bandwidth = base DRAM clock frequency * the number of data transfers per clock * bus width * the number of interfaces. According to the specification, our memory theoretical maximum bandwidth would be 38400MB/s. However, in Imbench[5], the author states that measured RAM bandwidth is about 25% of theoretical bandwidth. Therefore, we don't expect our experiment result to be close to the theoretical result. The measured RAM bandwidth is much smaller than the theoretical one may be caused by lots of factors, for example, there are lots of applications sharing with the memory at the same time, which would greatly affect the bandwidth. Another reason is that we access the memory in a non-continuous manner, which also decreases the read/write speeds compared with the ideal continuous manner.

4.3 Page fault service time

This section reports the time for faulting an entire page from disk. There are two kinds of page faults – one is major page fault handled by disk I/O operation (e.g. memory mapped file or swapping pages into memory); the other one is handled without disk I/O operation (e.g. *malloc*)[4].

4.3.1 Measurement Methods

The reason why we execute *mmap* operation as our measurement lies in that it can directly share the buffer between the OS kernel and the user-level process, which can reduce the non-negligible performance overhead caused by copying data between these two address spaces. The *mmap* system call maps the file given in the disk to a set of virtual pages in user space and can simulate the situation of a major page fault.

We measure the average clock cycles it takes to fault a single page. We create non-zero files whose sizes range from 100-3000 pages (400M-12G) and map them separately into each measurement, followed by reading through the file and adding up the values of bytes. For measuring the time cost, we still adopt two nested loops inside each measurement (100 * 100). Except for the measurement code, we also utilized the existing Linux executable "time" to verify the expected major/minor page fault number. Notice that the Linux kernel would support the file buffer read-ahead mechanism to optimize reading costs, which will dramatically reduce the average page fault cost. Therefore, we set up the "MADV RANDOM" flag for *mmap* to force random page references and disable the prefetching behavior.

4.3.2 Measurement Result

- Base hardware performance: 23600 cycles
- Estimated software overhead: 3135 cycles
- Predicted time: 26735 cycles
- Measured performance average: 29062.11 cycles
- Measured performance standard deviation among single testbench: 73.41
- Measured standard deviation across 1000 testbenches: 14.85
- Measured performance maximum deviation: 8972

4.3.3 Interpreting Measurement

For page fault service, the main bottleneck lies in the read/write speed of the SSD disk. Therefore, when estimating the base hardware performance, we refer to the specification of the SSD transfer speed, which is 600 MB/s. The estimated software overhead is combined with the system call, the procedure call, memory access (we still need to read through the mapped file from memory to force page faults), and measurement overhead. Besides, the instruction loading overhead is negligible with a warm-up. Therefore the predicted time cost is the result of adding base hardware performance to the estimated software overhead. We can see that the average measured page fault service cost is higher than the predicted one, but the difference, which in fact is 1.2 ns per page, is still acceptable. What's more, the disk used in this experiment has been used for over 5 years, and it's very likely to be slower than the specification.

Compared with the memory access time (which is around 2350 cycles/page), we can see that page fault latency is about 12X.

References

We have referred to the following papers for estimate overhead and measurement design.

[1] Gabriele Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. Intel Corporation, 2010.

[2] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In USENIX annual technical conference, pages 279–294. San Diego, CA, USA, 1996.

[3] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual. 2012

[4] Slide from CS423 https://courses.engr.illinois.edu/cs423/fa2011/lectures/cs423_mp3_final.pdf

[5] Larry McVoy and Carl Staelin, lmbench: Portable Tools for Performance Analysis, Proc. of USENIX Annual Technical Conference, January 1996.

[6] Memory bandwidth https://en.wikipedia.org/wiki/Memory$_b andwidth: :text = Memory\%20bandwidth\%20is\%20the\%$