Effective Benchmarking Techniques for System Calls and IPC Mechanisms

Nisarg Shah Zubeyr Eryilmaz

{nisargs, eryilmaz}@cs.wisc.edu

Location: /u/n/i/nisargs/cs736_benchmarking

Abstract

Due to optimizations designed for common use case scenarios, modern systems make it non-trivial to benchmark an operating system based on factors such as system call performance and inter-process communication (IPC) mechanisms such as pipes, TCP/IP and UDP sockets. Additionally, aiming for accurate benchmarks can lead to higher overheads in actual measurements. We propose techniques to accurately benchmark system calls, latency and throughput of pipes, TCP/IP and UDP sockets without introducing significant overheads in the techniques themselves.

1. Introduction

Benchmarking system performance is a necessary task to undertake to accurately quantify system performance. General-purpose operating systems and processors have been optimized for power usage and time for common use-case scenarios. Benchmarking a system also introduces overheads imposed by the benchmarks themselves. Section 2 discusses approach used to evaluate the clock precision and subsequently measure system call performance for 2 simple system calls (`getpid` and `getuid`) without incurring significant overheads. Section 3 examines ways to benchmark IPC mechanisms by measuring latency and throughput for pipes, stream and datagram sockets. Section 4 presents our findings and Section 5 concludes.

2. Measuring system call performance

This section describes the approach used to evaluate CPU clock precision and measure two simple system calls.

2.1 Evaluating Clock Precision

We attempted to evaluate the clock precision the C way using the `clock_gettime` function with clock ID CLOCK_MONOTONIC**.** This experiment is performed in two phases. First, we invoke two consequent clock_gettime calls and measure the time elapsed between the two calls. We repeat this experiment 10 million times and consider the smallest value obtained so far. Second, we incrementally insert an increment/decrement operation (var++ or var--) in between the two clock_gettime calls until we observe a change in the elapsed time. Each increment/decrement statement translates into an addl/subl instruction in the compiled code. This is also repeated 10 million times and the minimum value is calculated. It was observed that inserting 2 increment/decrement operations increases the elapsed time by 1 nanosecond. Since this is the smallest unit of time that can be measured by clock_gettime, we cannot conclude that the clock precision is 1 nanosecond until we can verify it with a method that gives us a finer-grained resolution over time measurements.

We referred to [1] for a way to measure elapsed time by using the Intel x86 instructions `rdtsc` and `rdtscp`. These instructions load the processor's time stamp counter into EDX:EAX registers. We add a cpuid instruction before

reading the timestamp counter the first time and one after reading it the second time to prevent out of order execution of instructions by the processor for optimizing performance. We perform the same two-phase experiment as before, and measure the cycles elapsed between the two rdtsc/rdtscp instructions. To measure the corresponding time, we divide the number of cycles obtained by the clock frequency of the processor.

2.2 Evaluating System Call performance

We performed the same experiment as before, with a system call placed in between the rdtsc/rdtscp instructions and measured elapsed time. We measured another simple system call, `getuid`, and observed that execution times for both calls were very close to each other.

3. Benchmarking Interprocess Communications

In this section we benchmarked Unix pipes, TCP/IP (stream) and UDP (datagram) sockets by measuring latency and throughput between two processes running on same as well as different hosts.

3.1 Pipes

We run two experiments – (1) measure the latency for a round-trip packet transmission between two processes, and (2) measure throughput by evaluating total amount of data sent per unit time.

To measure latency, we send a single fixed size packet to another process, send it back to the first process and measure the round-trip time. We repeat this experiment many times and compute the minimum latency. The entire process is repeated for different packet sizes – 4, 16, 64, 256, 1K, 4K, 64K, 256K and 512K bytes.

To measure throughput, we send multiple fixed-size packets to another process, send a single "ack" packet from the receiver when it receives all the packets, and evaluate the total time elapsed during this transmission. As above, we repeat this experiment for the same number of different packet sizes.

3.2 TCP/IP

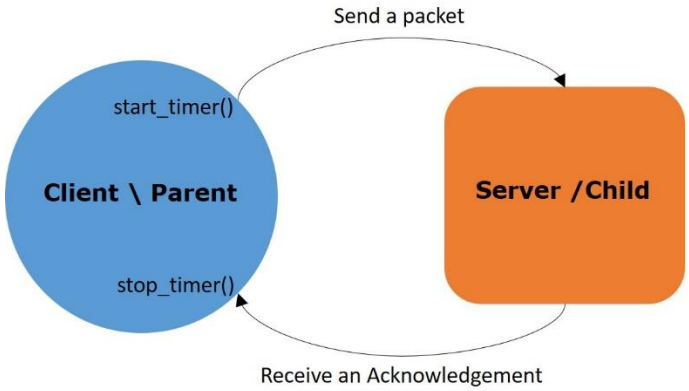


Figure 1. *Pipe, TCP/IP and UDP Experiment diagram for Latency Measurement*

To measure the latency, a server process sends every packet it receives from the client back to client, whereas the client starts the timer, sends the packet, receives the response from the server and stops the timer. In this manner, we

run our experiment for 1 million times and consider the smallest value. The value we get from this experiment is RTT (Round Trip Time). Assuming the latencies are same for back and forth communication between client and server, we divided the RTT by 2 to get the latencies. We repeat this experiment for varying number of packet sizes.

To measure throughput, the client starts the timer, sends many packets to the server and stops the timer on receiving an acknowledgement from the server.

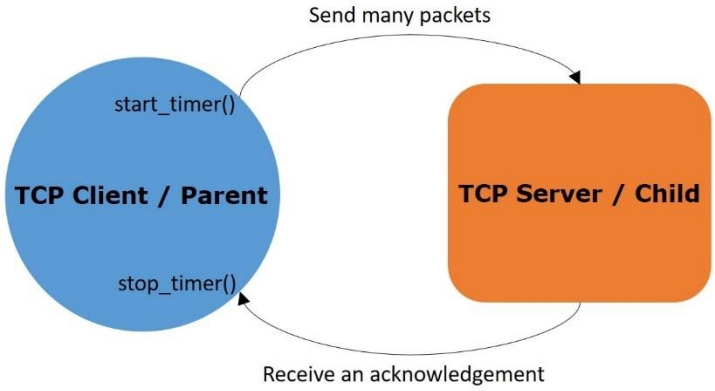


Figure 2. *TCP/IP and Pipe Experiment Diagram for Throughput Measurements*

3.3 UDP

Internet Datagram Socket is not reliable since it is based on UDP. To calculate latency, we have done the same experiment as we do for TCP/IP and Pipe. However, we send packets with unique IDs, and we ignored results if the

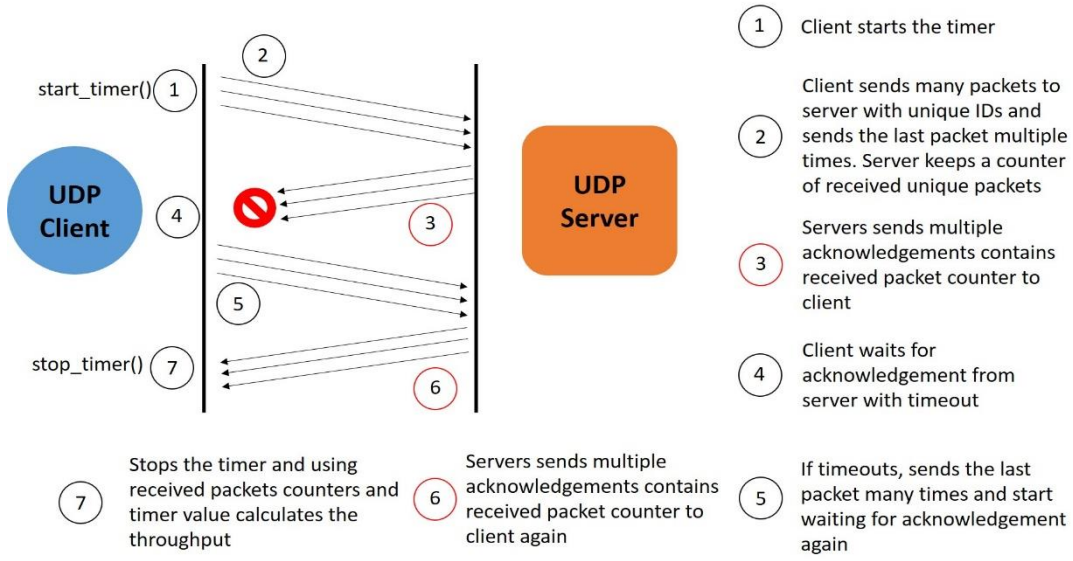


Figure 3. *UDP Experiment Diagram for Throughput Measurements*

receive command timeouts or the received packets ID is not equal to the sent packets ID. The experiment we perform for measuring throughput is detailed in Figure 3.

4. Evaluation

4.1 Platform

We perform our experiment as a set of C programs on Ubuntu 18.04.3 LTS systems running the default 4.15.0-62-generic Linux kernel. We used Dell Optiplex 3020 systems with the following configuration -

- Intel i5-4570 CPU with clock frequency 3.20 GHz
- 6M L3 cache
- 16 GB RAM
- Ethernet controller - Realtek Semiconductor Co., Ltd. RTL8111/8168/8411

The processor supports invariant timestamp counter (TSC), in which the counter runs at a constant rate (which is 3.2 GHz) and does not changes with CPU frequency changes.

4.2 System Call Measurements

Using clock_gettime, we observe a 1 nanosecond (ns) clock precision. Using the more precise rdtsc/rdtscp instructions, we observe a precision of 3 clock cycles, which translates to 0.94ns, which is not too far from the value obtained using clock_gettime.

To measure the two system calls getpid and getuid, we use the same rdtsc/rdtscp method to measure time. The getpid call took a minimum of 1169 clock cycles (1169 / 3.20 GHz = 365ns) during our measurements and getuid took a minimum of 1143 clock cycles (1143 / 3.20 GHz = 357 ns).

4.3 Pipe Measurements

We observe that latency for packet sizes up to 4KB remains in the range of 4-6 microseconds (µs) as the packet size increases. For sizes of 64KB, 256KB and 512KB, we observe a 3x increase in latency. There is almost an exponential increase in throughput for packet sizes less than 64KB, which saturates at 12000-13000 MBps for packet sizes greater than 4 KB. We attribute these two boosts to efficient utilization of the pipe buffer size of 64KB.

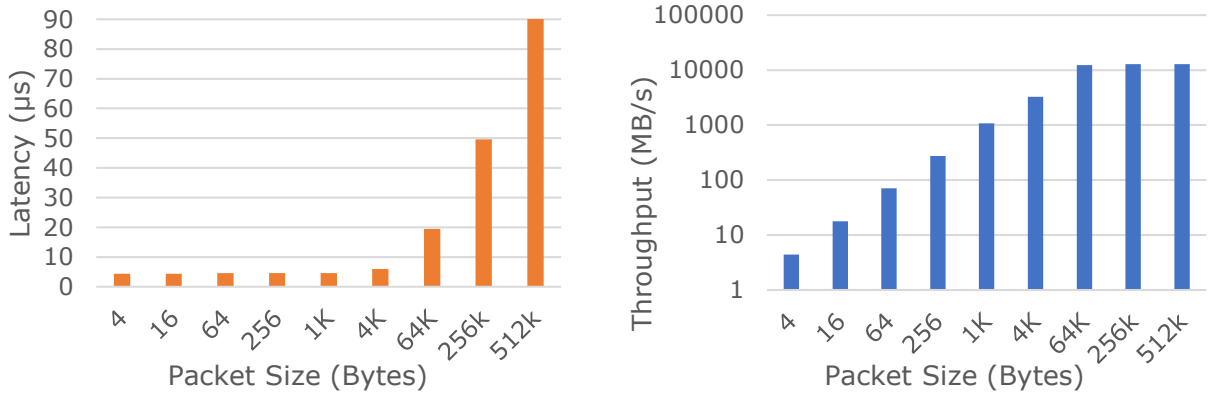


Figure 4. *Latency (left) and Throughput (right) of the Unix Pipe*

4.4 TCP/IP Socket Measurements

The read/write buffer sizes for TCP packets are configured as *tcp_wmem* and *tcp_wmem* vectors, which are defined as (min, default, max) bytes. The values for write buffer on our test system were (4KB, 16KB, 4MB). Packets were sent with TCP_NODELAY option set to avoid nagling.

We observe a similar pattern for TCP transmission as we observed for pipes. Packets up to 4KB have latency ranging from 3-5 µs and increases by a factor of 2-4x for sizes greater than 4KB. For throughput, we see an exponential increase in throughput for packet sizes up to 64 KB after which it was observed to increase linearly.

For remote TCP, we observe that throughput saturation is achieved at packet sizes starting from 256 bytes.

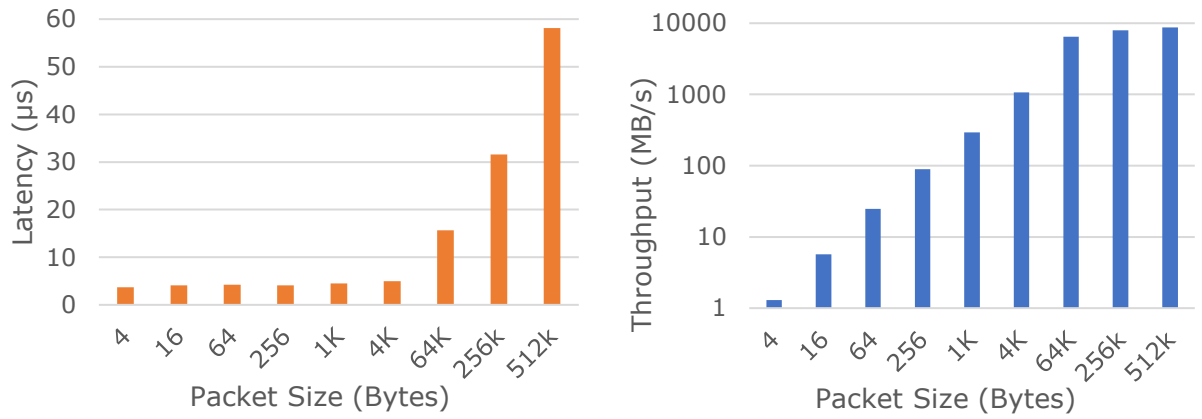


Figure 5. *Latency (left) and Throughput (right) of TCP/IP Socket for processes on the same host*

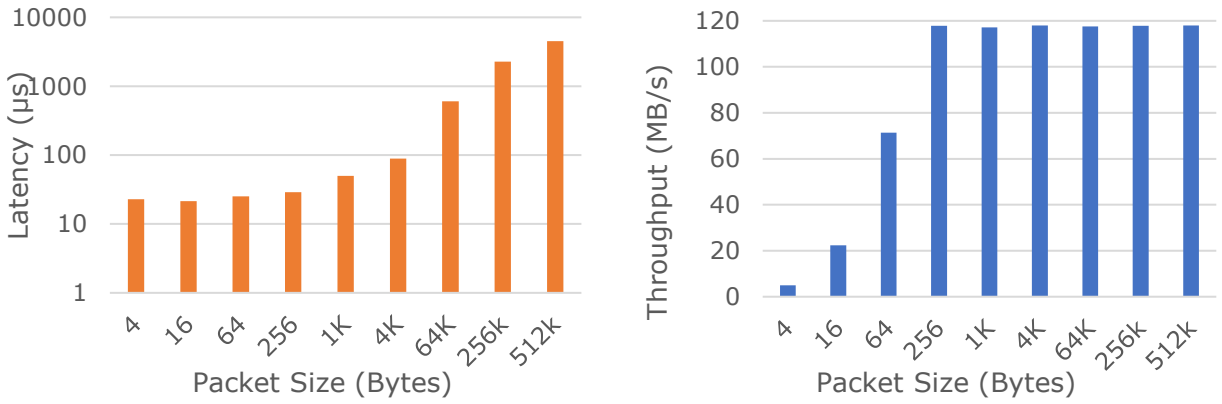


Figure 6. *Latency (left) and Throughput (right) of TCP/IP Socket for processes on different hosts*

4.5 UDP Socket Measurements

For local UDP transmissions we observe the same pattern as that seen for local TCP transmission. For remote transmissions, throughput saturation was observed for packet sizes starting from 1KB.

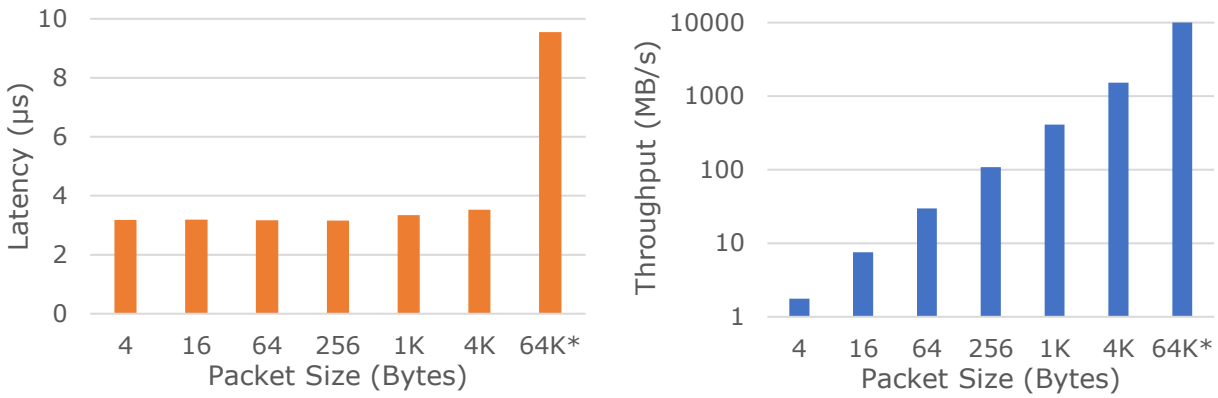
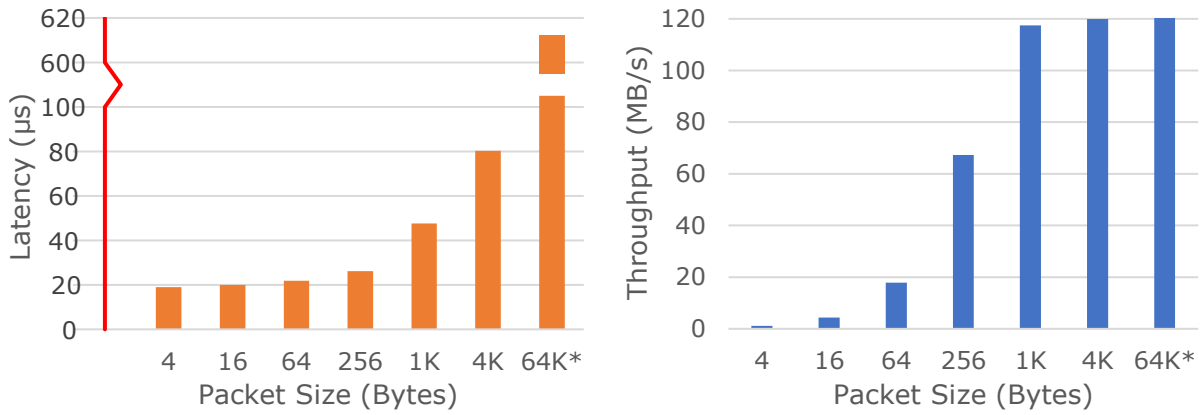


***Figure 7**. Latency (left) and Throughput (right) of UDP Socket for processes on the same host*



5 Conclusion

Using rdtsc/tdtscp instructions we were able to accurately benchmark a system call with high precision with minimal overhead (2 mov instructions following the rdtsc call). For pipes, TCP/IP and UDP sockets, we were able to benchmark the latency with the overhead of the return packet being transmitted via a path which might have different characteristics than the path used to send the packet. UDP throughput benchmarks also carry an additional overhead of sending extra packets at the end, which is negligible compared to total size of data sent. We also observed contrasting behavior of latency and throughput in relation to increasing packet sizes.

6 References

1. Gabriele Paoloni. How to Benchmark Code Execution Times on Intel®IA-32 and IA-64 Instruction Set Architectures (white paper). Intel Corporation, September 2010.