Introduction

Benchmarking is the most important thing in the world, especially for software engineers. The definition of benchmarking, within the scope of software development, is "*part of the software development life cycle which compares performance testing results against performance metrics to determine current performance and any changes needed to improve performance*." (GeeksForGeeks, 1) Measuring the runtime of different programs, down to the individual operations that various programs carry out, is integral in guiding their further development.

Imagine, for example, that your smartphone OS was filled with inefficient code. The responsiveness of the touch screen would lag and stutter as the process fumbled around in memory, Different apps would likely crash regularly and the product would probably be pretty useless. Or, instead, consider something more pertinent to human safety: vehicle software. One of the biggest tech trends at the moment is the self-driving car. Tesla vehicles have software connected to sensors all around the vehicle that detect nearby vehicles and their relative distance to your vehicle. If two vehicles get close enough to each other, the sensors trigger a warning that alerts the driver of the close proximity. From there, the driver can safely adjust the vehicle to pull away from the other vehicle and avoid an accident. Due to the very delicate nature of road traffic, as well as the human lives contained within each vehicle, imagine if the software's code was deeply inefficient due to a lack of any proper benchmarking throughout the software (and hardware) development. Lives can literally be at stake in these scenarios where software is intended to provide immediate alerts of incoming threats.

More than any example can provide though, one of the most important aspects of benchmarking is its essential function in simply designing better systems. Software and hardware engineers alike are able to learn and more thoroughly understand computer architecture as they benchmark their designs. It deepens their understanding of how their work is processed and helps to guide the design of future systems. In this paper, I'll be doing some benchmarking exercises to demonstrate an understanding of not just how to track time taken for various operations to compute, but also how to chart the subsequent data and analyze it.

History

From the earliest days of computers, performance was a factor in their design. From the days of the ENIAC (1943) to the Harvard Mark-1 (1944) to the IBM Stretch 7030 (1961), the measuring of performance was pretty limited to the "time required to perform an individual operation, such as addition." (Hennessy, 4.7) However, as operations became more complex, the time required for one operation became a dated metric for comparison. The first book on the subject, "Benchmarking: The Search for Industry Best Practices that lead to Superior Performance" by Dr. Robert Camp, was published in 1989. It coined the term "benchmark" and was informed by Camp's experience managing the benchmarking program with Xerox. Prior to this program, the concept of "competitive benchmarking" was largely utilized to examine manufacturing costs through product comparisons. Xerox began to take a markedly different

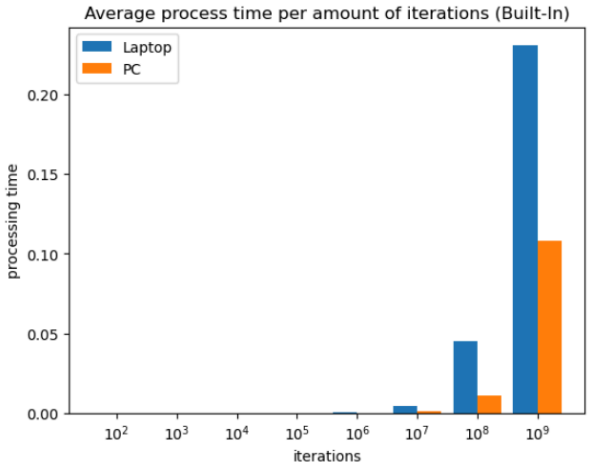
approach that emphasized best practices in the design of their equipment by employing the best irrespective of their industries. During the 11 years between 1981 to 1992, Xerox undertook over 200 benchmarking projects and managed to go from nearly-bankrupt to being recognized as a world-class technology company.

This success led to benchmarking becoming known worldwide. Alongside Xerox's success, the System Performance Evaluation Cooperative (SPEC) was formed to "establish, maintain, and standardize benchmarks and tools to evaluate performance for the newest generation of computing systems." (SPEC, 1) Nowadays, benchmarking is considered a "top five tool in terms of popularity and satisfaction of management tools and techniques." (Mann, 4) Formal benchmarking is made up of two types: performance benchmarking and best practice benchmarking. Performance benchmarking is "the comparison of performance data obtained by studying similar processes or activities." Best practice benchmarking is "the comparison of performance data from studying similar processes and identifying, adapting, and implementing the practices that produced the best performance results." (Mann, 135)

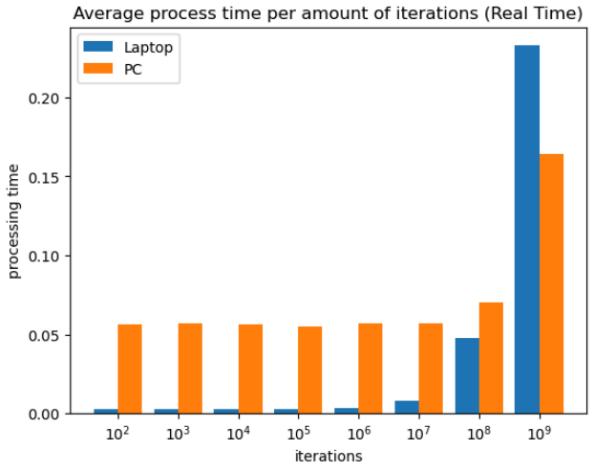
Tasks #2 and #3

For this project, I decided to run each program on two computers. The first is a six-year old Lenovo YOGA 720-12IKB laptop with an Intel Core i5-7200U x 4 with 8GB RAM and 128GB disk space that is running OS Ubuntu 24.04.1. The second is my home PC, which has an AMD Ryzen 5 5600X 6-Core Processor (3.7GHz) with 16GB RAM and 1TB disk space that is running Windows 10 Home.

The first exercise consisted of benchmarking a trivial operation – implementing a simple for loop to increment integer *i* from 0 to 10^2-10^9. When running the program, I ran a simple `test ./benchmark` command and tracked the program output time, as well as the Real, System, and User times separately. I ran the measurements ten times for each exponential interval and tracked each iteration in a spreadsheet for later use in Python. Respectively, I've just labeled these different machines as "Laptop" and "PC" on the figures below.



Built-In Timer Function



Real Time

The general trend of each plot shows that the process runs incredibly quickly up until we get to 10^7, where the run time increases dramatically from an average of 0.0004 (Laptop) / 0.00011 (PC) seconds for the iterations at 10^6, to an average of 0.0045 / 0.00109 for the iterations at 10^7 – roughly a tenfold increase in processing time, consistent on both machines. This trend continues through 10^7 to 10^8. Interestingly, when going from 10^8 to 10^9, the laptop increase was only five-fold but the PC increase was still tenfold.

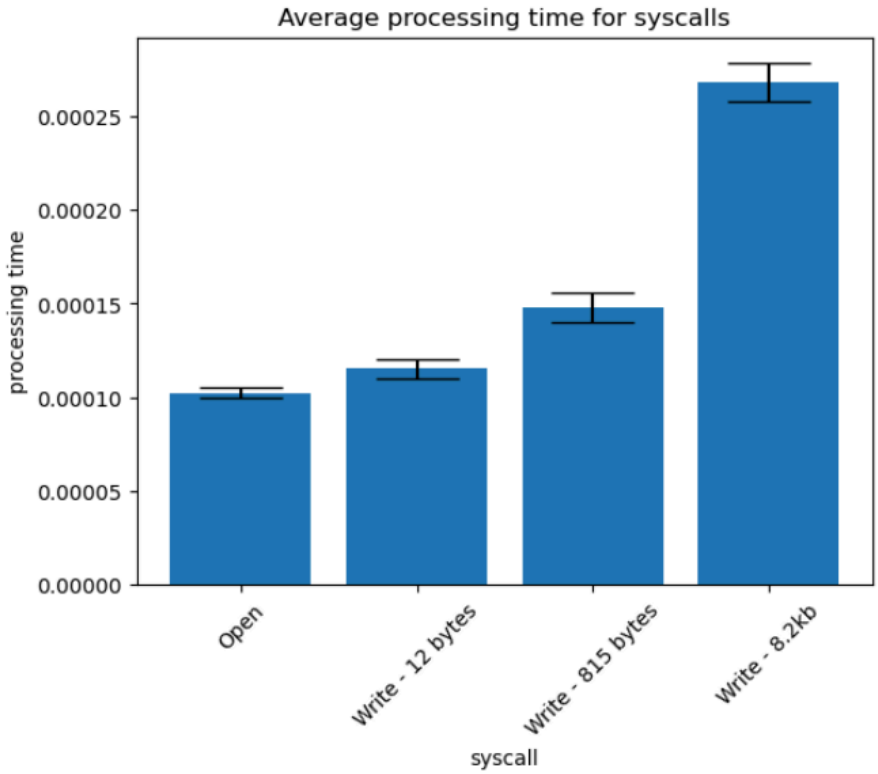
In running the program, the time difference noted from the program output seemed somewhat different from the results in the time output. Namely, the splitting up of the Real, User, and System time measurements necessitated some looking into. Real time is equivalent to wall-clock time, User time is time spent in the user-mode code (outside the kernel) within the process, and System time measures the time spent in the kernel within the process. It seems that the difference between utilizing the time function vs. the built-in technique is explained by the fact that the time function requires some overhead with the time syscall on top of running the process.

I should mention that, in the earlier iterations (10^2-10^6), Real time was consistently giving me odd results with the 0.06ms reading from 10^2-10^7. This is most likely due to the limit of three decimal places. The thing that confused me most was that even though the Real time was most closely matching the Built-In time in the later iterations (10^7-10^9), the metric in the time call's output didn't change anything about the output metric (ex. "0m0.031s") to properly scale it to what I was observing in the built-in runtime output. Extended to enough decimals, the numbers would probably match up but I didn't want to make an assumption about how many decimals places I would have to move the numbers. This issue strikes me as an amateur move, but since it caused a bit of a hiccup in my analysis, I figured it would be worth bringing up.

Task #4

The next exercise involved benchmarking non-trivial operations, namely some system calls. Out of the recommended options, I went with `write()`, `open()`, and `fork()`. To keep it

simple in learning how these calls work as well as their arguments, I just went with a simple "Hello world!" message for the first write call. It consists of 12 total bytes (one per character).



Standard error is notated on each bar

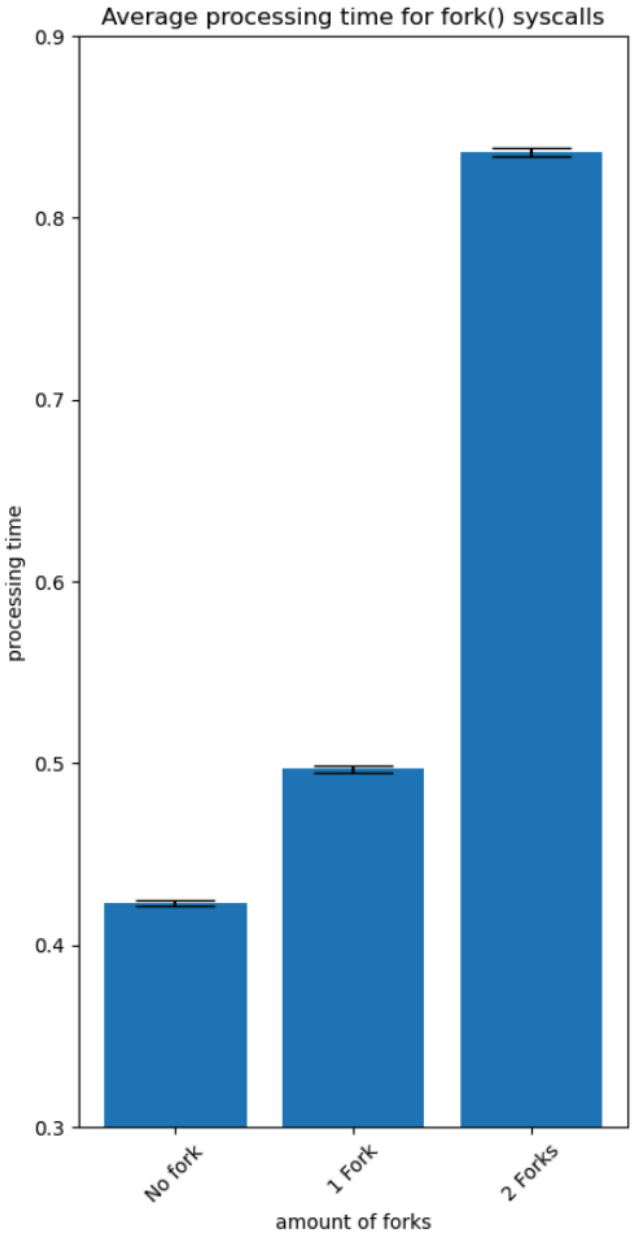
After 20 runs of the program using the built-in timer, the average time for the `write()` call was 0.00012 (Laptop) / 0.000010 (PC). This amounts to roughly 250 thousand iterations of the previous for loop where we incremented integer *i* up to 10^9. I then tried it with one full paragraph of Lorem Ipsum, which amounts to 815 bytes. After another 20 runs of the program, it came out to a very slight average increase of 0.00015 / 0.00001. There was a 6800% increase in byte size but only a 0.00003 increase in average processing time on the laptop (the PC had no change). Finally, I copied the Lorem Ipsum paragraph ten times (8240 bytes total) to see how much more the processing time would increase. With the 1000% increase in byte size, the average write call runtime was 0.00027 / 0.00002 seconds, meaning there was an increase of 0.00012 / 0.00001 from the single Lorem Ipsum paragraph – only about a 180% increase in processing time on the laptop. With this in mind, it seems that, more than anything, most of the overhead is definitely in the `write()` system call itself.

Next, I ran the same sort of test on the `open()` call. I created three .txt files – the first was made up of "Hello World!", the second contained one paragraph of Lorem Ipsum (same as the one used in the `write()` call), and the third contained ten paragraphs of Lorem Ipsum. Interestingly, when I made an open call on each of them, the runtime was always the same despite their difference in file size, averaging 0.00010 / 0.00004 seconds over 20 runs.

Finally, I started to tinker with the `fork()` system call, especially since we had to write about it this week and I had to present it. The process that I decided to time was the same loop used in Tasks #2 and #3, but I decided to have some fun and went with the maximum possible integer: 2147483647. To establish a baseline, I ran twenty iterations of the loop without a `fork()` call. After that, I ran another twenty with one `fork()` call, and then another twenty with two `fork()` calls.

The figure to the right may be a tad confusing, so I'll clarify what it demonstrates. When I first charted these numbers, the standard error was only showing up as a single line. I confirmed that there was a standard error that should be displayed, so what you're seeing here is essentially a zoomed in graph. There were two things I needed to do to show a distinct standard error range:

1.) *Scale the graph vertically to stretch the bars more.*
2.) *Reset the y-limit from 0 to 0.3.*



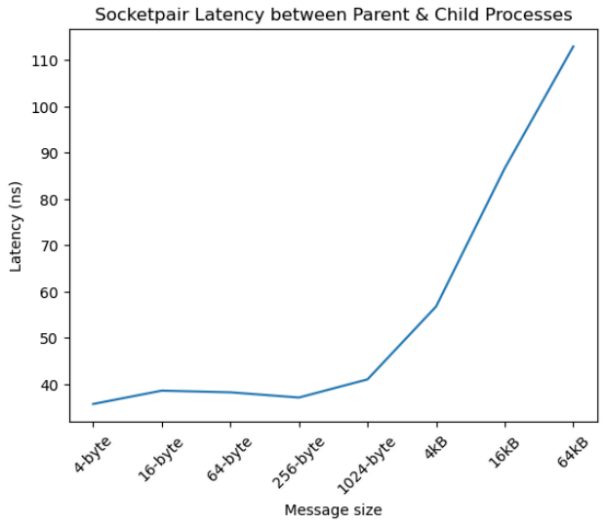
This is worth clarifying because even though it looks like the "2 forks" bar is 4x or even 5x the size of the "0 forks" bar, it's really only doubling it. My main goal was to display the standard error.

This information does lead me to a very interesting question though: the two forked processes followed by the iteration up to the max integer means that, in total, the iteration will execute four times. With that in mind, why isn't the total runtime with two forks 4x as long as the runtime with zero forks?

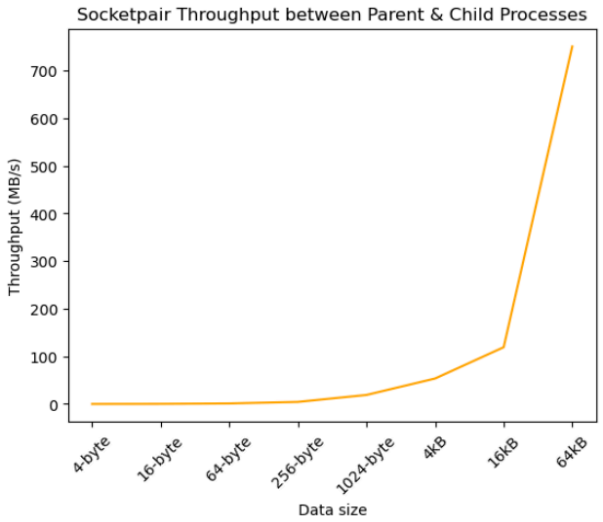
Task #5

The final stop was to measure both the latency and the throughput of a socket communication between a parent and child process. I found this step to be the hardest, mainly due to not fully understanding how to specify the metrics of each within the code. Eventually, it kind of clicked and I was able to find some decent results where the increase in latency and throughput were consistent with each other.

I ran the tests from 4-bytes up to 64kB because anything above that (256kB, 512kB) was not working. I ran into a similar issue in my initial tests of char array size when, in an earlier version of the code, I was trying to manually place 256k characters into a char array since I wasn't seeing a latency difference without it. I may be wrong, but I think it's worth considering that this 6-year old laptop may just not be able to process even mildly demanding programs.



Measurements of Latency



Measurements of Throughput

For each of these measurements, the variables were message/data size on the x-axis, and then latency (ns) and throughput (MB/s) on the y-axis. My hypothesis in this section was that for each 4x increase in size along the x-axis, the corresponding latency/throughput would reflect a similar increase along the y-axis. I ran twenty tests on each of them to find the mean for each x-tick. As you can see, the results don't display a steady, gradual increase along the y-axis to mirror the byte size increase on the x-axis. I actually think that this is mostly due to the fact that I had trouble finding how to disable nagling in the code. On further review, that might even be responsible for the odd dip in latency around the 256-byte mark right before the sudden jump. Either way, the fact that both measurements looked this similar in the end was a pretty cool observation.

Conclusion

Having benchmarked all kinds of different measurements (trivial operations, syscalls, latency, and throughput) and compared built-in results to those of the time command, I definitely

felt like I learned a lot about how to measure program performance. It was also fun that, up until the `fork()` system call, I was able to compare the performance of my PC with this old laptop. Having put the PC together from top notch parts at the beginning of 2021 (NVidia GeForce RTX 3070/AMD Ryzen 5600 were both very hard to find at the time), I'm glad to know that the CPU is still performing well in processing the small programs from this project. After going over the "`fork()` in the road" paper, I wonder if I would have found a consistent performance difference between the PC and the laptop if I would have learned how to use the `CreateProcess()` call in place of `fork()` since my Windows PC does not allow forking. At some point, I'll have to explore that API and maybe I'll remember to run those tests then. Either way, this project was definitely an intense crash-course in benchmarking and I'm glad I finally learned the basics of what the process entails.

Works Cited

GeeksForGeeks. "Benchmark Testing in Software Testing". Jan 30th, 2024. https://www.geeksforgeeks.org/benchmark-testing-in-software-testing/

John L. Hennessey, David A. Patterson. "Computer Organization and Design - The Hardware/Software Interface". Morgan Kaufmann. October 10th, 2013. http://staff.ustc.edu.cn/~han/CS152CD/Content/COD3e/CDSections/CD4.7.pdf

Mann, Robin. "The History of Benchmarking and Its Role in Inspiration". Journal of Inspiration Economy. September 1st, 2016.

SPEC. "Standard Performance Evaluation Corporation". 1995-2024. https://www.spec.org/spec/