JOHANNES KEPLER
UNIVERSITY LINZ

Author
Mykhailo Sakevych
K12148331

Submission
Institute for Symbolic Artificial Intelligence

Supervisor
DI **Maximilian Heisinger** BSc

June 2023

Analysis of Differences Between Time Tracking APIs

Bachelor Thesis
to obtain the academic degree of
Bachelor of Science
in the Bachelor's Program
Artificial Intelligence

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references. I used the web service Grammarly to improve my writing.

[illegible]

Linz, June 2023

Acknowledgement

I would like to express my sincere gratitude to my supervisor, DI Maximilian Heisinger, for his invaluable guidance and support throughout the duration of my project. His expertise, dedication, and unwavering commitment to excellence have been instrumental in the successful completion of this thesis.

Abstract

This thesis presents the findings of comparing CPU time and wall time measurements using different APIs (`/usr/bin/time` [1], Fish `time` [2], Bash `time` [3], and `BenchExec` [4]) for a highly CPU-bound process. The focus of the analysis was to evaluate the precision and consistency of the APIs' results, without considering other influences such as memory, caches, etc.

Throughout the experiment, CPU time and wall time measurements were collected using each API for every execution of the CPU-intensive process. Statistical analysis was performed to evaluate differences between the measurements obtained from different APIs and determine the level of significance.

No significant differences were observed. In the CPU time and wall time measurements among the various APIs, besides their precision. All the tested APIs demonstrated highly precise results, with negligible variance between measurements under equal experimental conditions.

These findings suggest that when measuring CPU time and wall time for a CPU-intensive process, the choice of API have minimal impact on the precision and consistency of the measurements. The APIs evaluated in this experiment were found to be reliable and provided accurate results. They can thus be used interchangeably for benchmarking tasks, given they are used correctly and results are not invalidated in other ways. However, BenchExec API provides more guaranties for its correct use and also tracks sub-processes, but its precision for individual processes is similar. Future studies could explore additional factors or scenarios to further validate these findings and expand the understanding of API selection as well as perform new analysis and new experiments to reach higher precision and negate influential factors.

Contents

List of Figures

Chapter 1
Introduction

1.1 Problem Statement

The run-time of a process is a critical metric for many applications. It can be used as a means to evaluate performance of an application, to identify bottlenecks, and guide developers to optimise the applications for better performance. However, measuring a process run-time can be challenging.

There are a number of factors that may affect measuring, including CPU speed, memory, and eventual background processes. Additionally, the time-tracking API used can also affect the accuracy of the measurement. As average desktop machines are usually running hundreds of processes, such undesired influences may be frequent.

The findings of this thesis will be valuable for developers who are looking to measure the runtime of their applications. The results will also help to improve our understanding of the factors that affect the accuracy of time-tracking APIs.

1.2 Research Questions

This thesis investigates the accuracy of different time-tracking APIs for measuring run-time without memory utilization on a regular PC. The specific research questions addressed are:

- What are the different time-tracking APIs available?
- How do the different time-tracking APIs perform, and how stark are their differences?
- How can background processes affect run-time?
- How does CPU frequency scaling affect the run-time of a process?

1.3 Structure of the Work

First, the introduction provides an overview of the thesis, The problem statement section discusses the challenges of measuring the run-time of a process without memory utilization on a regular PC, while the research questions section formulates the goal of this work.

Then, we discuss the methodology of our approach. We provide a background on time-tracking APIs, as well as a brief explanation of different times: CPU, wall, usr, etc. The example of time-tracking APIs usage to measure the run-time of a process are also demonstrated and explained . In this chapter we also formulate which results will be considered acceptable and what would constitute negative result.

Afterwards, we move on to the discussion of results. This chapter starts with an explanation of data post processing. It continues with a detailed description of the plot structures and their interpretations and discusses the observed differences in mean and variance for different numbers of parallel processes. It also contains analysis of the results for each API to identify any significant variations. Last but not least, we discuss improvements in time-tracking and the time tracking result, including wall time.

The final chapter present the outcomes of this thesis and possible future work tracks.

1.4 Related Work

In this section, we will examine some relevant studies and research that have investigated the accuracy of time-tracking APIs on a regular PC. We specifically focus on desktop computers that are used as interactive work environments by a single user e.g. running on a Linux system. These works offer valuable insights into different time-tracking APIs, their performance capabilities, and the various factors that can impact measurement accuracy.

One noteworthy study is "EMP: Execution Time Measurement Protocol for Compute-Bound Programs" by Young-Kyoon Suh [5]. This paper introduces a new protocol called EMP, which aims to precisely and accurately measure the execution time of CPU-consuming processes. EMP successfully identifies and addresses external timing factors, resulting in high-quality measurements. Additionally, this protocol takes into account the variability of execution time caused by memory references.

Another study worth mentioning is "BenchExec: A Ready-to-Use, Tool-Independent Benchmarking Framework" authored by Dirk Beyer, Stefan Löwe, and Philipp Wendler [4]. This paper discusses value of reliable benchmarking and problems connected with it. An accessible benchmarking framework BenchExec is one of the tools developed for this paper. The framework offers practicality for measuring program performance reliably. In paper authors use it to measure factors extend of influence.

These papers provide valuable insights into the challenges of benchmarking software systems, and the different techniques that can be used to improve the accuracy of benchmarking measurements.

In summary, previous research has examined different time-tracking APIs, their performance, and the factors that can affect their accuracy for measuring process runtimes. These studies have shed light on the strengths and weaknesses of various APIs and highlighted the influence of factors such as CPU frequency scaling, NUMA and background processes. However, there is still a space for improvement of the accuracy of runtime measurements on regular PCs.

1.5 Data Availability

The code, plots and collected data utilized in this thesis can be accessed at the following GitLab repository. Additionally, for those seeking higher-resolution plots and figures, they are also available on the same repository. This work and data are licensed under the Creative Commons Zero v1.0 Universal.

Chapter 2 Methodology

2.1 Time-tracking APIs

There are a number of different time-tracking APIs available [6, 7, 8] , each with its own strengths and weaknesses. This work focuses on most popular and most accurate one. This chapter introduces the ones evaluated in this thesis: Bash `time`, Fish `time`, `/usr/bin/time`, `RunExec`.

2.1.1 Bash `time`

Bash `time` is a built-in command in Bash shell that can be used to measure the runtime of a process. It prints the total runtime of the process, as well as the runtime of the different phases of the process, such as the user time, system time and real time [3].

The bash `time` command can be used to measure the runtime of any process, regardless of whether it is a Bash script or a binary executable.

The following is an example of the output of the bash `time` command:

```
time ./main 434343
real    0m0.050s
user    0m0.040s
sys     0m0.010s
```

2.1.2 Fish `time`

In Fish the shell `time` command is a built-in command that allows you to measure the execution time of a given command or a block of commands [2].

The time command in Fish provides information about the real time, user CPU time, and system CPU time consumed by a command or a series of commands. It can be used to measure the runtime of any process.

The following is an example of the output of the fish `time` command:

```
fish -c "time ./main 434343"
Executed in  44.90 millis  fish            external
   usr time  30.85 millis  696.00 micros   30.16 millis
   sys time  11.10 millis  1045.00 micros  10.05 millis
```

2.1.3 GNU time

GNU time, also known as `/usr/bin/time`, is a command-line utility available in Unix-like operating systems, including Linux. It is distinct from the `time` command built into the shell [9]. `/usr/bin/time` is an external command that provides more extensive timing and resource usage information compared to the shell's built-in time command.

GNU time measures the resource usage of a command and reports various statistics, including the elapsed real time, CPU time, peak memory usage, and more. It is often used to analyze the performance characteristics of programs and determine their resource consumption. A drawback of `/usr/bin/time` is that all times it returns are in centiseconds.

The `/usr/bin/time` command can also be used to measure the runtime of any process

The following is an example of the output of the `/usr/bin/time` command:

```
/usr/bin/time ./main
0.02user 0.01system 0:00.04elapsed 95%CPU
```

2.1.4 RunExec (part of BenchExec)

RunExec is a custom tool developed as hackathon project by SoSy-lab. It can be used to measure the runtime of a process. This API was developed specifically for accurate time-tracking and to isolate processes from all factors that could possibly influence results.

RunExec can measure the runtime of any process. It also correctly treats CPU-time of subprocesses, as it uses the cgroups (cgroups2) feature of the modern Linux kernel.

The following is an example of the output of the RunExec API:

```
2023-03-06 12:54:01,707 - INFO - Starting command echo Test
2023-03-06 12:54:01,708 - INFO - Writing output to output.l
og
returnvalue=0
walltime=0.0024175643920898438s
cputime=0.001671s
memory=131072
```

2.2 Different Types of Trackable Times

For time tracking, we first need to understand which type of time we want to track and what differences there are. There are four main types of time that can be tracked:

CPU time The amount of time that the CPU spends on specific process. This time is equal to sum of system time and user time.

System time The amount of time that the CPU spends executing kernel code for a process. This includes time spent handling system calls, interrupts, and other kernel-level operations, such as accessing files, etc.

User time This is the amount of time that the CPU spends executing user code for a process. This includes time spent executing application code, libraries, and other user-level code.

Wall time This is the total amount of time that elapses from the start to the end of a process. This includes time spent waiting for I/O, time spent blocked by other processes, and time spent executing code in any mode. It is based on a theoretical monotonic wall-clock that only goes forward, disregards time zones, DST or other non-linearities.

Different times can be used to understand how a process is using the CPU and to identify potential bottlenecks in its performance. For example, if a process has a high CPU time but a low wall time, it can mean that the process is spending a lot of time waiting for I/O. This can be addressed by e.g. optimizing the I/O operations or by using a different data structure that is more efficient for the type of I/O that the process is performing. On the other hand, if a process has high system time and low user time, it typically indicates that the majority of CPU time is being consumed by system-level operations rather than the execution of user-space code.

It is also important to note that these times can be affected by the number of other processes that are running on the system. If there are a lot of background processes running, then each process will have less CPU time available. This can lead to longer wall times for all of the processes.

This thesis collects 3 different times:

- User + Sys or Observed. Answers how much actual CPU time your process used.
- Wall. Amount of time from start to end (API reported)
- Reported. Time reported by program (more details in the next section).

2.3 Experimental Setup

For reliable comparison of different APIs all measurements have to be with the same prerequisites and settings. Program we use to measure runtime would be a C++ program that counts the number of divisors of a number. The program will take as input a number X and will return the number of divisors of X and the time difference between `clock()` at the start and end of its execution.

In addition to the C++ program, a Python file is used to conduct all runs and collect data. The Python file runs the C++ program using one of time-tracking APIs 1000 times with a preset seed of 43434343. After this, it collects the return value of the C++ code as well as the time measured by the API. For collecting time reported by the API we use `regex` and for C++ executable return we read the last line from `stdout`. The Python file then saves the data to a `.pkl` file. The python script takes as argument the prefix of data filenames.

In addition to the Python script, we also run similar C++ program with seed 1234567891234 outside of the script to see how the number of background processes influences the runtime.

The experiment is conducted on Raspberry Pi 4B with the following specifications [10]:

CPU: Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz

RAM: 4GB

Operating system: Raspbian 6.1.21-v7l+

The program is compiled using the following command:

```
g++ main.cc -o main
python main.py 0
```

Background processes are compiled using the following command:

```
g++ garb.cc -o garb
./garb 1234567891234
```

Optimizations have no influences on the experiments. We receive 3 different times for each `main` call. For one API it performs 1000 calls, and 4 APIs are tested. Finally, we do all previously mentioned actions for every number of background processes (from 0 up to 6 in my case). In total: $3 \cdot 1000 \cdot 4 \cdot 7 = 84\,000$ or $12\,000$ per run.

2.4 Definition of Significant Differences Between Measurements

In this thesis, a histogram is the main plotting technique for our analysis. In case of histogram, positive results would be a distribution with minimal variance. This means that the reported runtime would fall within this range with the highest probability. Negative results, on the other hand, would be any distributions with large variance. This means that the reported runtime could vary widely, which is undesirable.

A small variance indicates that the data is tightly clustered around the mean, while a large variance indicates that the data is spread out over a wider range [11]. However, a small variance does not necessarily mean that the distribution is close to the desired distribution. For example, a distribution with a small variance but a large mean would still be considered negative result.

We can also use statistical distance as a metric to evaluate the results of our analysis. Statistical distance is a measure of how different two distributions are [12]. In our case, we would use statistical distance to measure how different the distribution of reported runtimes is from the desired distribution.

It is important to note that statistical distance is not the same as variance [13]. A small statistical distance indicates a positive result, whereas large indicates a negative result.

Therefore, we can use both variance and statistical distance to evaluate the results of our analysis. A small variance and a small statistical distance would indicate positive results, while a large variance and a large statistical distance would indicate negative results.

Chapter 3 Discussion

3.1 Post Processing of Gathered Data

In the previous chapter, we discussed the necessary data for our analysis. After collecting the raw time data, we normalize it to milliseconds.

3.2 Eliminating Influence Factors

There are a number of factors that can influence the time it takes to complete tasks on a multi-core system. These factors can include CPU Frequency Scaling, Hyperthreading, NUMA, shared memory bandwidth and caches, and multiple CPUs.

CPU Frequency Scaling Dynamic frequency scaling (DFS) technology allows the processor to adjust its frequency in real-time based on the workload [14]. This can improve performance, but it can also lead to more variability in the time it takes to complete tasks [15].

Hyperthreading Allows each physical core on an Intel Core processor to run several threads simultaneously. This can improve performance for tasks that can take advantage of multiple threads, but it can also lead to more variability in the time it takes to complete tasks [16].

NUMA Memory architecture that is used in some multi-core processors. In a NUMA system, each core has its own local memory, and access to remote memory is slower than access to local memory. This can lead to more variability in the time it takes to complete tasks that access memory from multiple cores [17].

The shared memory bandwidth and caches The shared memory bandwidth and caches on a system can also affect the time it takes to complete tasks. If the memory bandwidth is low or the caches are small, then tasks that access memory may take longer to complete [18].

Multiple CPUs If a task is running on multiple CPUs, then the time it takes to complete the task will depend on the synchronization between the CPUs. If the CPUs are not synchronized properly, then the task may take longer to complete [19].

Given that Raspberry Pi does not support Hyperthreading and NUMA, and also have only one CPU socket, only two factors remain as the primary influences: shared CPU-caches and CPU Frequency Scaling. Only CPU Frequency Scaling will be further investigated and mitigated.

3.3 Experiments and Results Analysis

3.3.1 First Results

Plots are presented with increasing complexity.

Plot structure: In this plot structure 3.1, the fixed variables are the API used (specifically Bash) and the type of time being plotted (observed). Each plot in the series represent the distribution of observed times across a given time range. The only distinction among these plots is the number of processes running in background. The X-axis is shared among all plots, and the labels include the mean of the plotted times.

Analysis: Upon analyzing the results, it is evident that there is no significant difference in the first two subplots. The mean and variance did not exhibit substantial variations. However, for two, three, and four background processes, there is a notable difference in mean values compared to the results of zero and one background process. Additionally, these plots display larger variances than those with zero and one background processes. Interestingly, the results for five and six background processes reported CPU times close to 0 and 1, with even smaller means than those of 0 and 1 background processes.

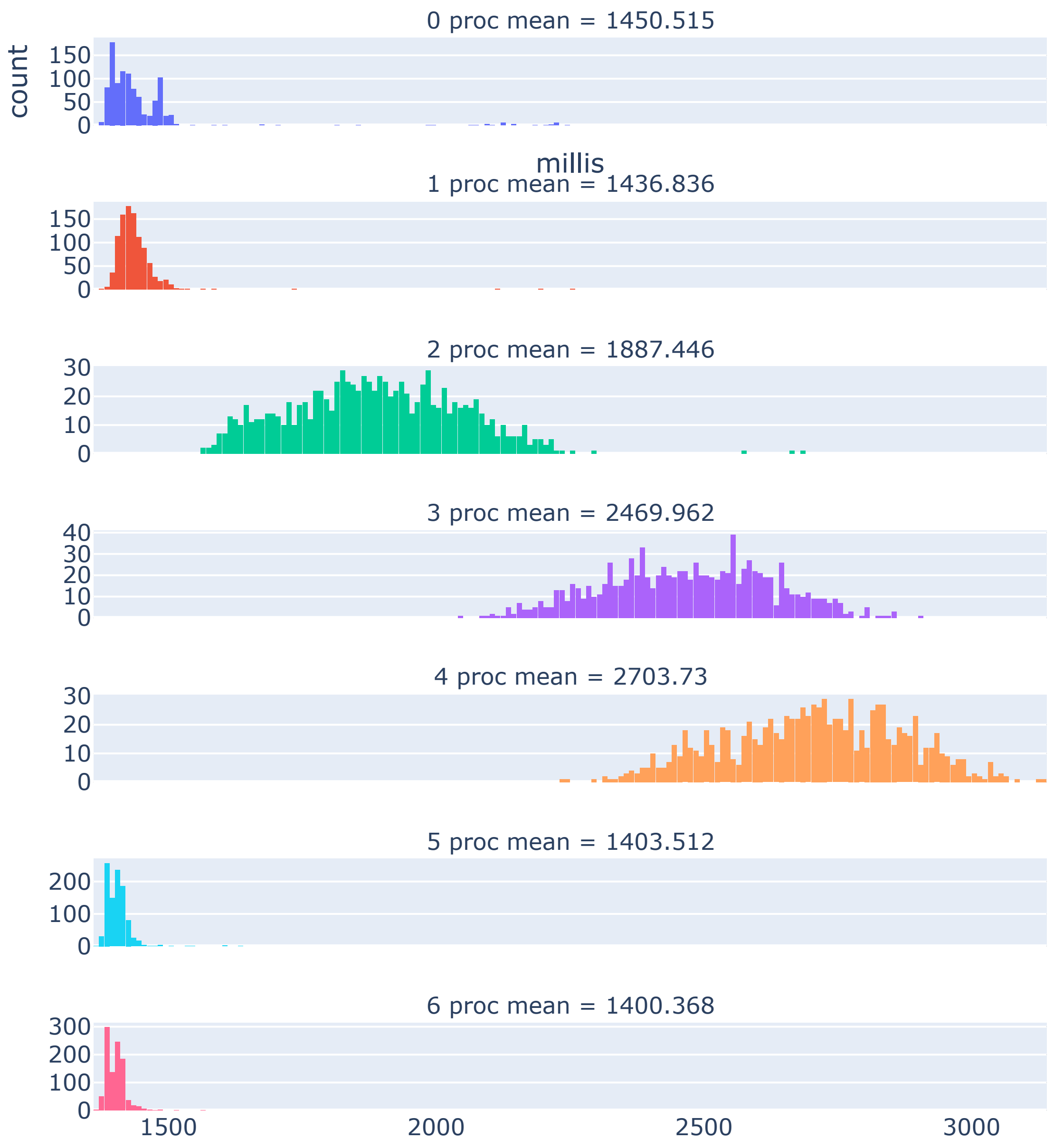
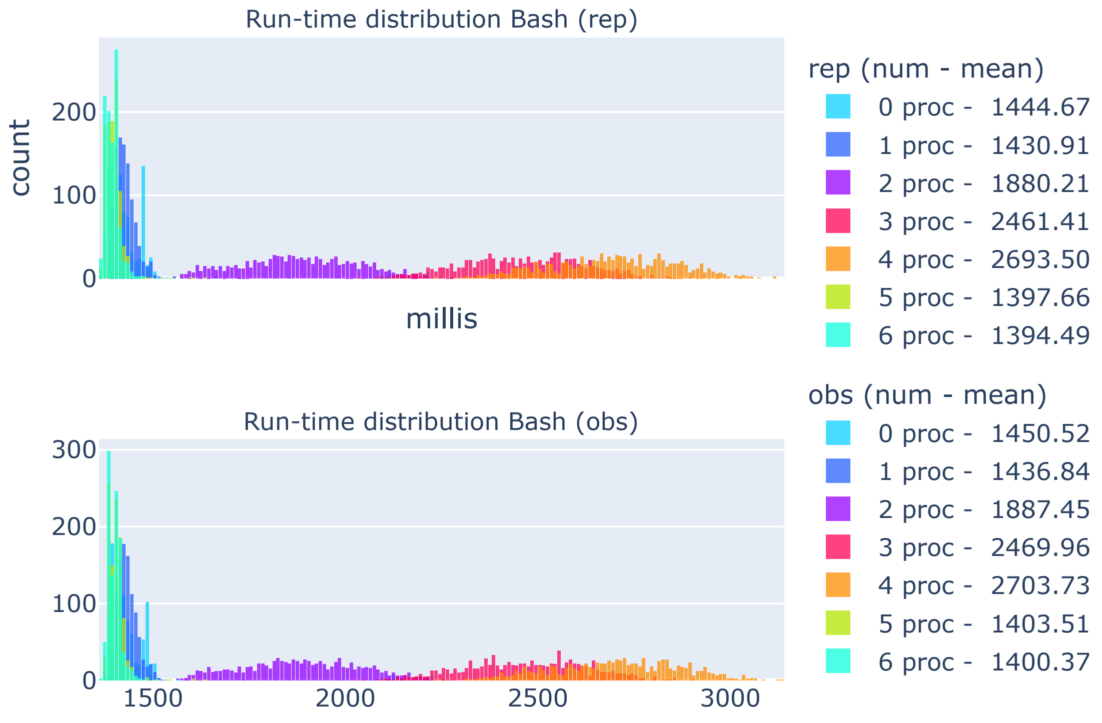


Figure 3.1: Observed time(usr + cpu), Bash, plot per run with 0-6 background processes

Plot structure: In the subsequent series of plots 3.2. I combined all traces from 0 to 6 background processes into a single plot. This series maintained a fixed API and shared X-axis. The legend provided information about the color of the traces and the mean value of each

Figure 3.2: Observed times VS reported; runs with 0-6 background processes



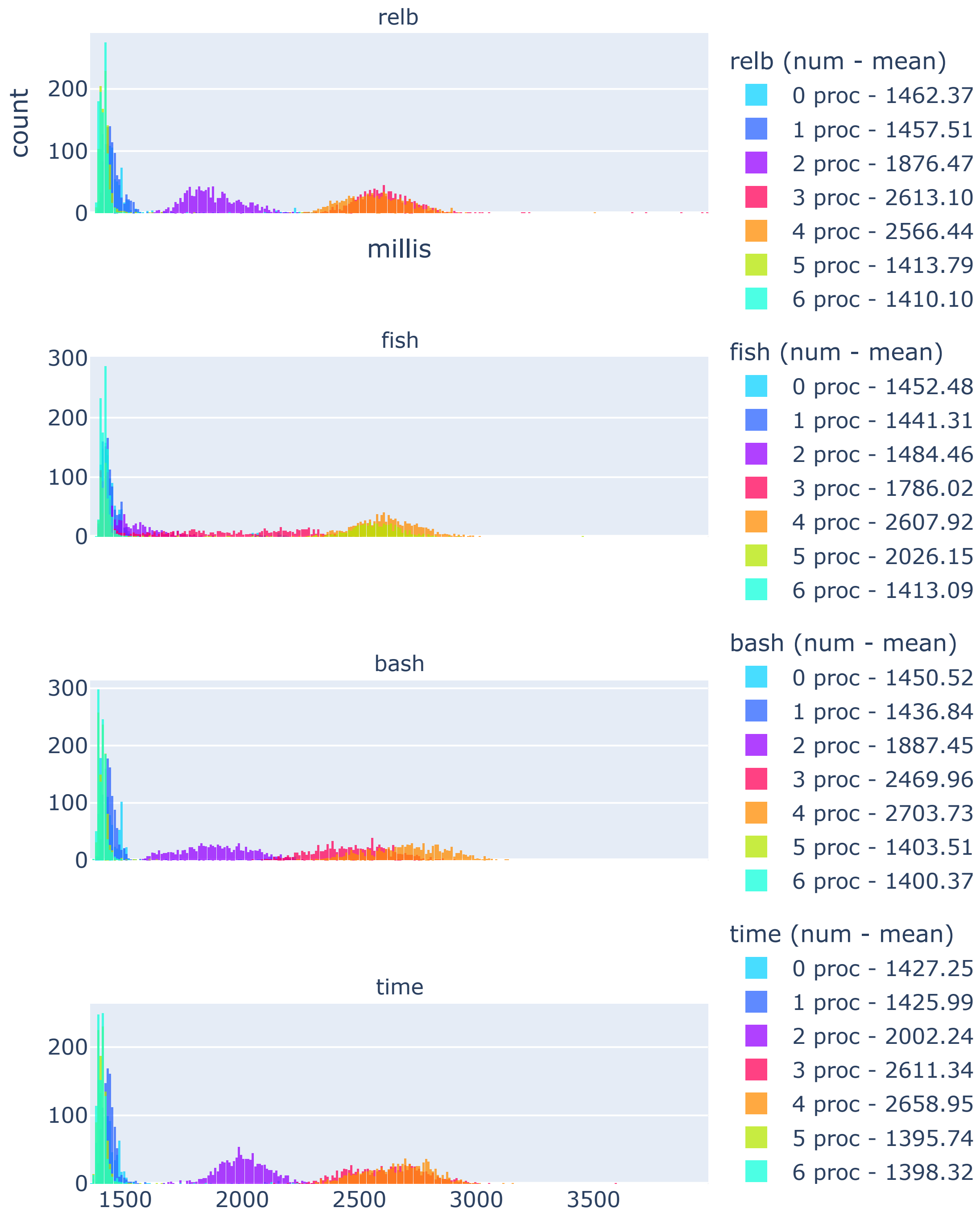
trace. The first plot represented the reported time, while the second plot depicted the observed time.

Analysis: After analyzing these plots, it became apparent that there is an insignificant difference between the two. Consequently, for further analysis, we focus exclusively on the observed time.

Plot structure: Figure 3.3, shows a series of plots with a shared X-axis and a fixed type of time (observed). Similar to the previous series, the legend contains information about the color of the traces and the mean value of each trace. This series comprised four plots, with one plot dedicated to each API.

Analysis Analyzing these plots revealed that all of them exhibited similar distributions. However, the Fish API displayed a larger mean for the results of 5 background processes compared to any other API.

Figure 3.3: Observed times (usr + cpu); different API; runs with 0-6 background processes per plot



Conclusions The observed results can be attributed either to the prioritization of the tested program by the Linux scheduler or to this experiment serving as an exceptional scenario within the Linux scheduler code.

Further experiment setting To eliminate the influence of CPU Frequency Scaling, we disabled it. However, this decision carried a significant risk. If the CPU temperature were to reach a critical level (within a working range of -40°C to 85°C) [20], the CPU could begin throttling, and in the worst-case scenario, it could potentially overheat. To disable dynamic frequency scaling, we utilized the `cpu freq -set` command [21], first setting the governor to `userspace` mode with the `-q` flag, and then specifying a constant frequency (1GHz) with the `-f` flag. Afterward, we repeated the experiment, ensuring that the highest temperature measured (using `htop`) was 68°C.

3.3.2 Enhancing Tracking

The subsequent figures share the same structure as previously discussed (see Plot Structure)

As depicted in 3.4, there is no significant difference in the mean results among runs with different numbers of background processes. Therefore, it can be concluded that after turning off CPU Frequency Scaling, the number of background processes does not "increase" the CPU time utilized by our program. It does, however, potentially increase variance.

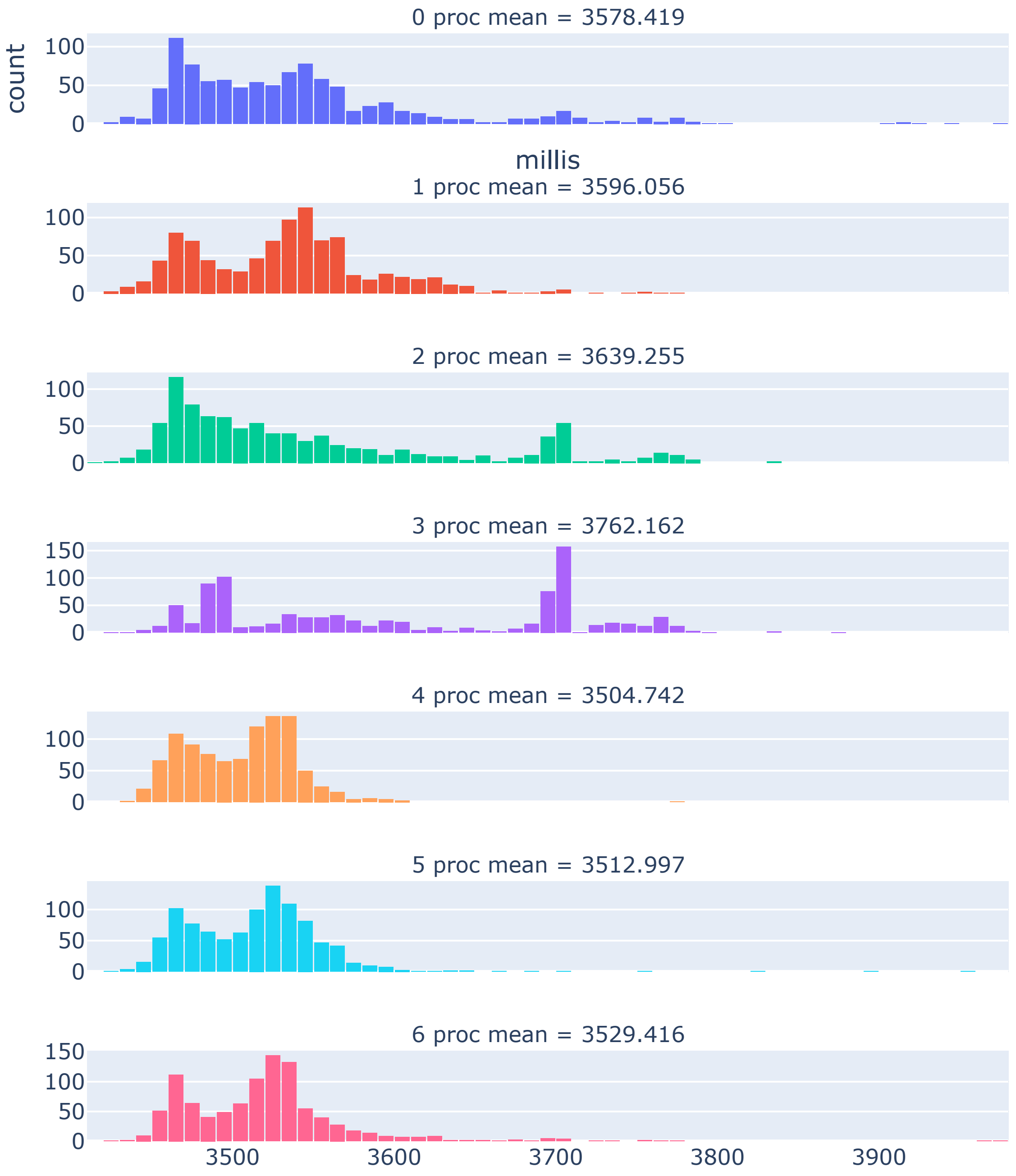
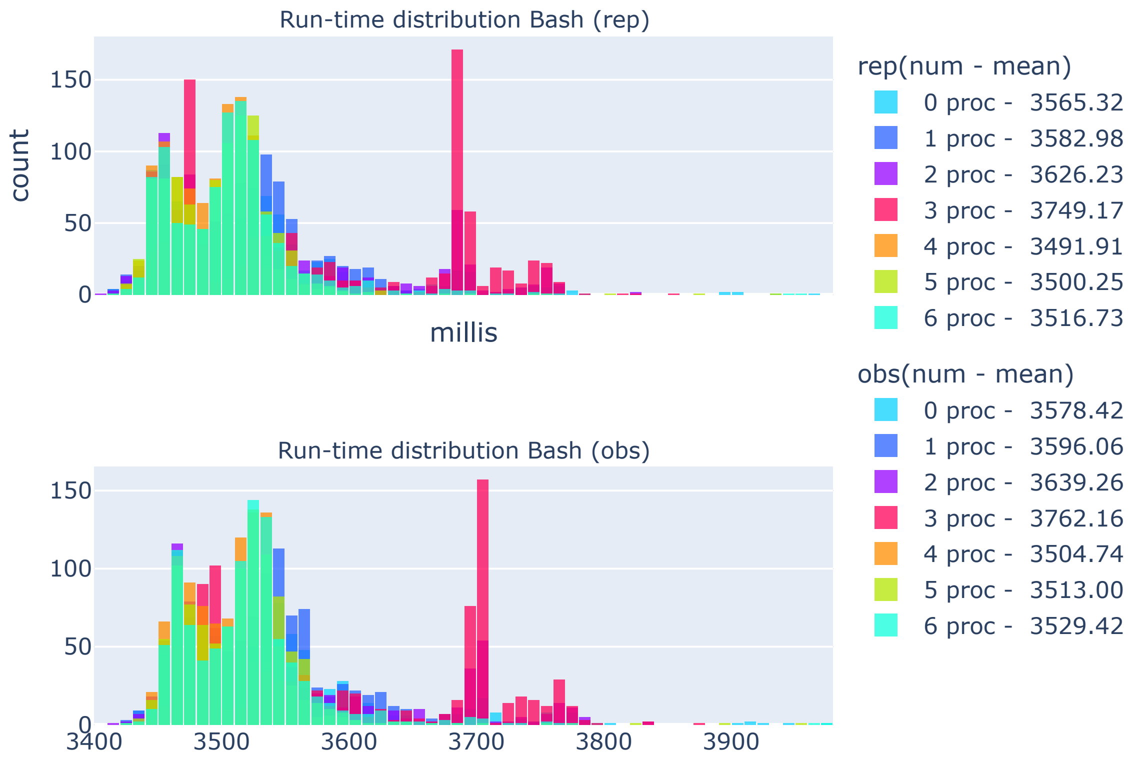


Figure 3.4: Observed time(usr + cpu); Bash; plot per run with 0-6 background processes; no CPU Frequency Scaling

Additionally, it is necessary to verify if the observed time and reported time remained close to each other. 3.5 provides confirmation of their proximity.

Figure 3.5: Reported time compared to observed time; runs with 0-6 background processes; no CPU Frequency Scaling



Lastly, the comparison of results across different APIs is examined in 3.6 The plot indicates that all the different APIs exhibit similar means among runs with different numbers of background processes. This finding aligns with expectations, as the CPU time spent on the same computation should be similar or show insignificant differences. Notably, bimodality can be observed in all the results. However, in the results of 5 background processes using `/usr/bin/time`, a triple modality artifact can be noticed, the origin of which remains unknown.

3.7 replicates the same experiment as in 3.6. However, on this graph, the triple modality is present in the results of 5 background processes using the *RunExec* API.

Wall time Unlike CPU time, which measures the amount of time the processor spends executing a specific task, wall time takes into account the overall elapsed time from the start to the completion of a process, including waiting time and other external factors. In the context of this analysis, it is expected that wall time will increase as the number of background processes increases. This is because with more background processes running simultaneously, the total amount of work being performed by the system increases, which in turn leads to a longer overall duration for the processes to complete.

Upon observing the plot 3.8, it is apparent that the mean wall time increases with an increasing number of background processes. However, wall time does not increase till all 4 cores are busy (till 4 background processes mean changes insignificantly).

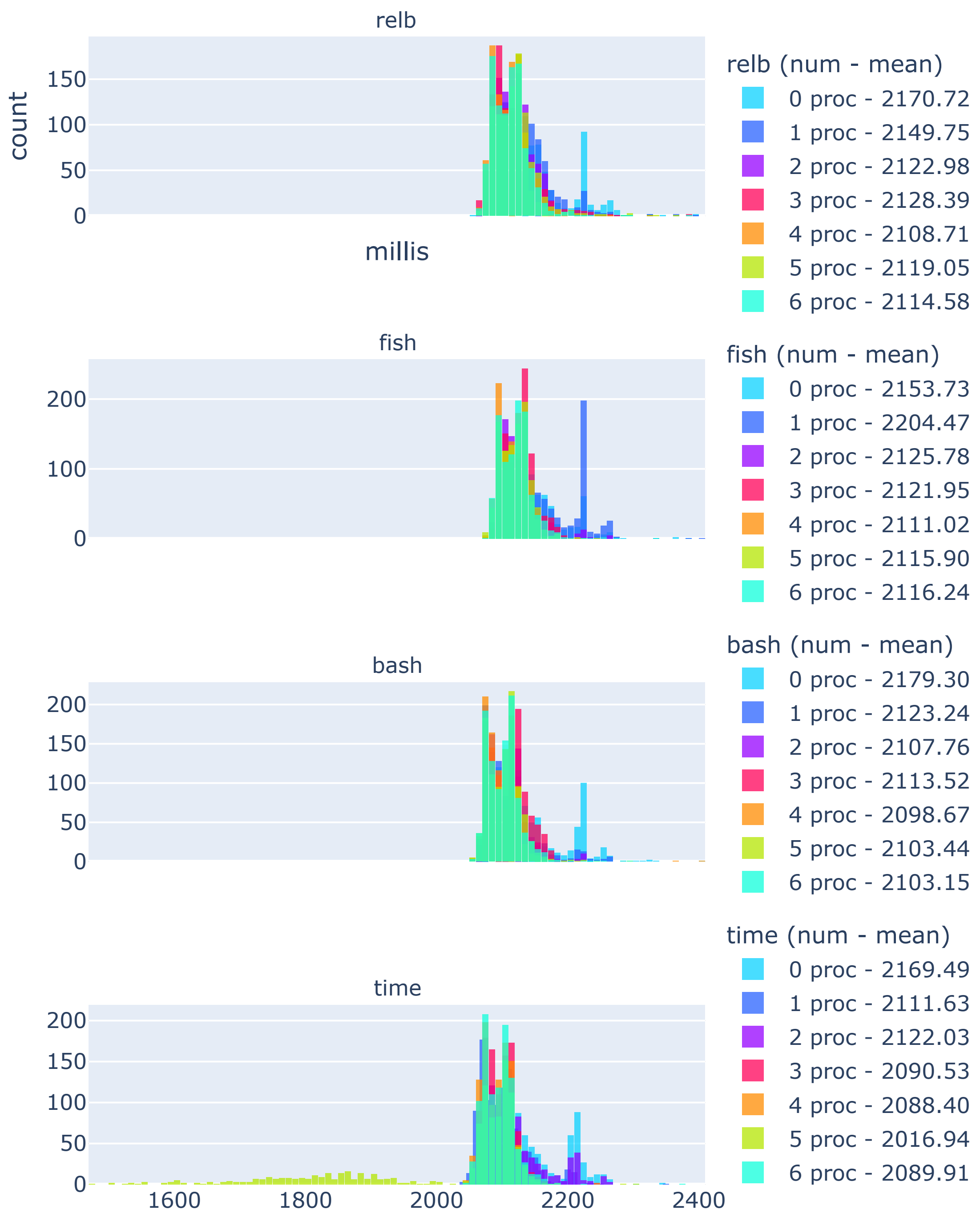


Figure 3.6: Observed times (usr + cpu); different APIs; runs with 0-6 background processes; no CPU Frequency Scaling (1)

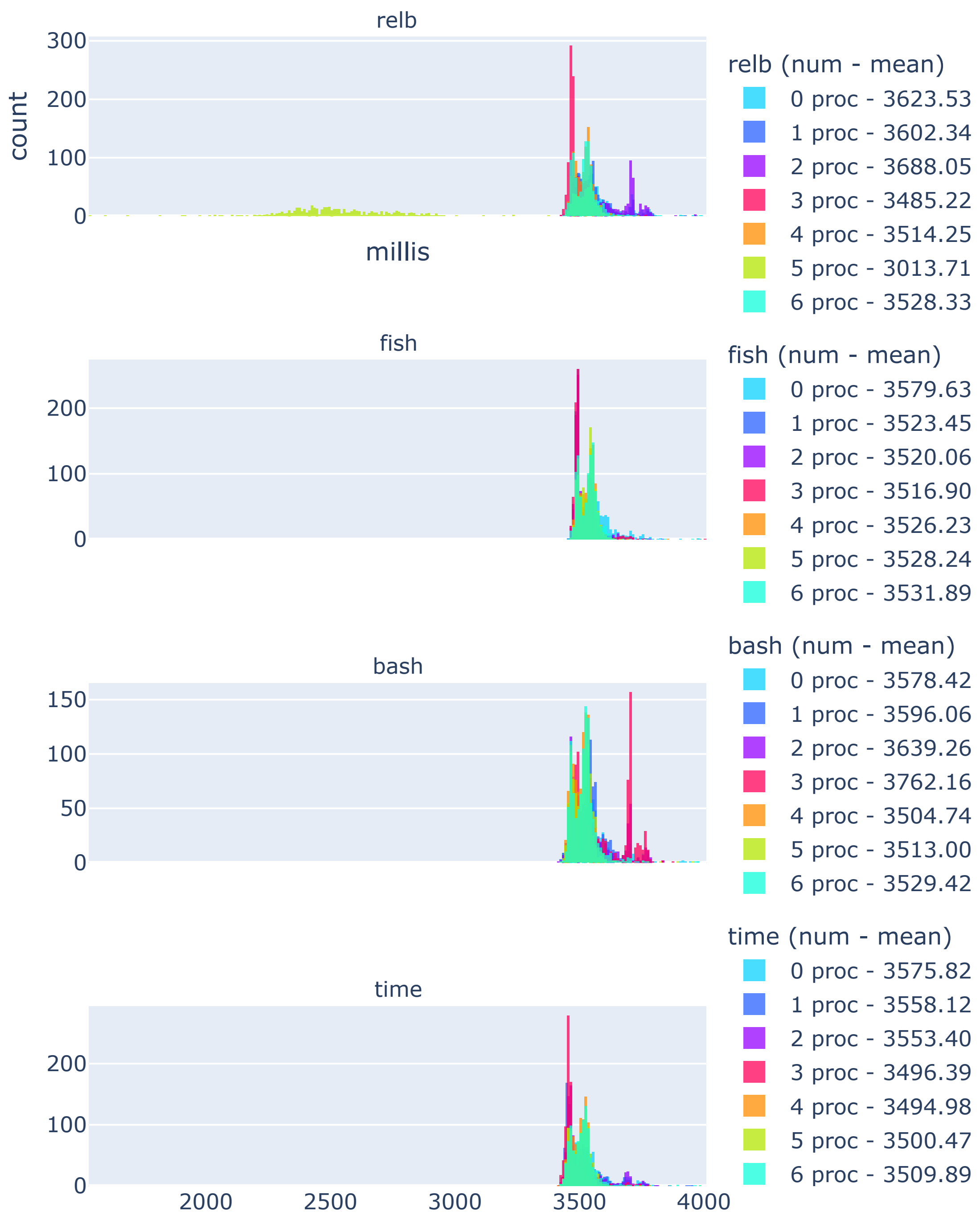


Figure 3.7: Observed times (usr + cpu); different APIs; runs with 0-6 background processes; no CPU Frequency Scaling (2)

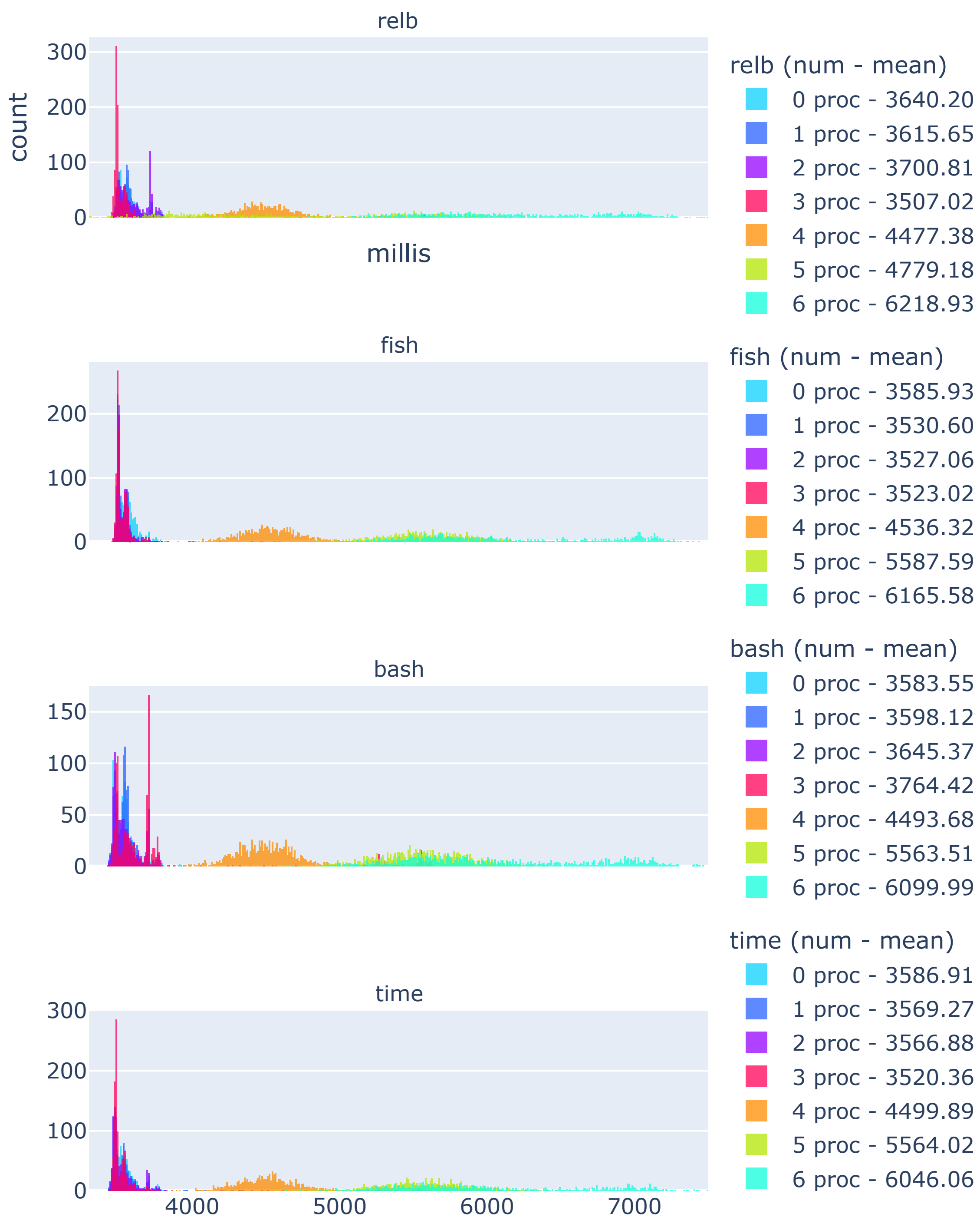


Figure 3.8: Wall times for runs with 0-6 background processes

Chapter 4
Conclusion and Future Work

Conclusions

The results showed that CPU Frequency Scaling can significantly influence measurement of CPU time, and that the choice of API does not have a significant impact on the accuracy of CPU time measurements.

The study was limited to a single computer and a limited set of programs. Therefore, the results may not be generalized to other computers or programs. However, the findings of the study have implications for the accuracy of CPU time measurements and the optimization of program performance.

Further Analysis

Further analysis plan can include:

- Investigation of triple-modality on figure 3.6 and 3.7. What is its nature? Since it influences the precision of time tracking, we need to know how to mitigate it.
- Further analysis of collected data. Using other analysis techniques and statistical testing can help in analysis and bring up new findings and questions.
- Investigation of difference in means of experiments 3.6 and 3.7. What causes difference in 1 second between any two experiments with same conditions?

Implications

The findings of this study have several implications for the accuracy of CPU time measurements and the optimization of program performance. First, the study showed that CPU Frequency Scaling can significantly influence measurement of CPU time. This means that it is important to disable CPU Frequency Scaling when measuring CPU time, or to use a method that can account for the effects of CPU Frequency Scaling.

Second, the study showed that the choice of API does not have a significant impact on the accuracy of CPU time measurements. This means that it is not necessary to use a specialized API for measuring CPU time, and that any of the APIs that were tested in this study can be used to get accurate measurements, if following conditions of experiment.

Call to Action

The findings of this study can be used to improve the time tracking of CPU time and the optimization of program performance. I encourage readers to replicate the study or to use the findings of the study in their own work.

Recommendation for researchers to repeat experiments and to do significance testing of the benchmark results. Evaluate the time-related distribution of ones specific benchmarking setup.

Bibliography

[1] Michael Kerrisk. *time(1) — Linux manual page*. URL: https://man7.org/linux/man-pages/man1/time.1.html (cit. on p. iii).

[2] Axel Liljencrantz. "*Fish - A user-friendly shell*". URL: https://lwn.net/Articles/136232/ (cit. on pp. iii, 4).

[3] Wikipedia contributors. *Time (Unix) — Wikipedia, The Free Encyclopedia*. [Online; accessed 2-July-2023]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Time_(Unix)&oldid=1115593213 (cit. on pp. iii, 4).

[4] Dirk Beyer, Stefan Löwe, and Philipp Wendler. “Reliable benchmarking: requirements and solutions”. In: *International Journal on Software Tools for Technology Transfer* 21 (2019), pp. 1–29 (cit. on pp. iii, 3).

[5] Young-Kyoon Suh et al. “EMP: execution time measurement protocol for compute-bound programs”. In: *Software: Practice and Experience* 47.4 (2017), pp. 559–597 (cit. on p. 2).

[6] Cloyce D Spradling. “SPEC CPU2006 benchmark tools”. In: *ACM SIGARCH Computer Architecture News* 35.1 (2007), pp. 130–134 (cit. on p. 4).

[7] Sergey V. Zubkov. *std::chrono::system*$_c lock$. URL: https://en.cppreference.com/w/cpp/chrono/steady_clock (cit. on p. 4).

[8] Python Software Foundation © Copyright 2001-2023. *timeit — Measure execution time of small code snippets*. URL: https://docs.python.org/3/library/timeit.html (cit. on p. 4).

[9] Yunming Zhang. */usr/bin/time notes*. URL: https://yunmingzhang.wordpress.com/2016/03/15/usrbintime-notes (cit. on p. 5).

[10] Raspberry Pi Trading Ltd. *Raspberry PI 4B Specifications*. 2019. URL: https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-4-Product-Brief.pdf (cit. on p. 7).

[11] Robert V Hogg and Allen T Craig. “Introduction to mathematical statistics.(5"" edition)”. In: *Englewood Hills, New Jersey* (1995) (cit. on p. 8).

[12] DA Freedman, Robert Pisani, and Roger Purves. “Statistics. WW Norton & Company”. In: *Inc.,* (2007) (cit. on p. 8).

[13] Maurice George Kendall, Alan Stuart, and J Keith Ord. *Kendall’s advanced theory of statistics*. Oxford University Press, Inc., 1987 (cit. on p. 9).

[14] Bilge Acun, Phil Miller, and Laxmikant V. Kale. “Variation Among Processors Under Turbo Boost in HPC Systems”. In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS ’16. Istanbul, Turkey: Association for Computing Machinery, 2016. ISBN: 9781450343619. DOI: `10.1145/2925426.2926289`. URL: `https://doi.org/10.1145/2925426.2926289` (cit. on p. 10).

[15] © 2017 Intel Corporation; Rafael J. Wysocki. *CPU Performance Scaling*. URL: `https://docs.kernel.org/admin-guide/pm/cpufreq.html#cpu-performance-scaling` (cit. on p. 10).

[16] Deborah T Marr et al. “Hyper-Threading Technology Architecture and Microarchitecture.” In: *Intel Technology Journal* 6.1 (2002) (cit. on p. 10).

[17] Christoph Lameter. “An overview of non-uniform memory access”. In: *Communications of the ACM* 56.9 (2013), pp. 59–54 (cit. on p. 10).

[18] John D McCalpin et al. “Memory bandwidth and machine balance in current high performance computers”. In: *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2.19-25 (1995) (cit. on p. 11).

[19] Vijayalakshmi Saravanan et al. “A study on factors influencing power consumption in multithreaded and multicore cpus”. In: *WSEAS Transactions on Computers* 10.3 (2011), pp. 93–103 (cit. on p. 11).

[20] 2023 Brainboxes Limited. *How does Pi deal with overheating?* URL: `https://www.brainboxes.com/white-papers/raspberry-pi-overheating` (cit. on p. 15).

[21] Mattia Dongili Dominik Brodowski. *cpufreq-set(1) - Linux man page*. URL: `https://linux.die.net/man/1/cpufreq-set` (cit. on p. 15).