Distributed Systems Lab Report

Raymond Hoogervorst
2691516
VU Amsterdam
r.g.g.hoogervorst@student.vu.nl

Chris Hoynck van Papendrecht
2829344
VU Amsterdam
c.a.s.hoynckvanpapendrecht@student.vu.nl

Sebastiaan Peters
2832111
VU Amsterdam
s.peters8@student.vu.nl

Stef van Schie
2675049
VU Amsterdam
s.j.van.schie@student.vu.nl

Thea Verburg
2679816
VU Amsterdam
t.h.w.verburg@vu.nl

Chris Wagenaar
2695346
VU Amsterdam
c.c.j.wagenaar@student.vu.nl

Abstract

Application telemetry is a crucial tool to properly understand how codebases behave. The collection of such data can be done with tools such as OpenTelemetry. For the accuracy of performance data, it is crucial that such a tool does not add significant overhead which skews these measurements. Therefore, we perform several experiments to analyze the overhead of tracing done by OpenTelemetry. We conducted an analysis on several benchmarks which put strain on the hardware of the device, to analyze the type of resources OpenTelemetry needs. These benchmarks are performed for Python and C#.

Our measurements show that, while OpenTelemetry incurs a performance overhead, the overhead is minor and does not significantly affect measurements. While the languages have different amounts of overhead, these are proportional to the performance of the languages themselves. We find no significant difference between the overhead of different types of load on different parts of the hardware.

Support Cast

- Lab Supervisor: Sacheendra Talluri
  Email: s.talluri@vu.nl
- Course coordinator: Xiaoyu Chu
  Email: x.chu@vu.nl
- Course coordinator: Tiziano De Matteis
  Email: t.de.matteis@vu.nl
- Course coordinator: Alexandru Iosup
  Email: a.iosup@vu.nl
- Course coordinator: Matthijs Jansen
  Email: m.s.jansen@vu.nl
- Course coordinator: Dante Nieuwenhuis
  Email: d.nieuwenhuis@student.vu.nl

1 Introduction

Distributed applications are composed of many services, each of which can be complex or even a distributed application itself. Some applications can handle millions of users simultaneously, leading to possibly millions of in-progress interactions in the system at any given moment. The scale and complexity of distributed systems introduced by these scenarios make it challenging to identify and debug runtime errors. Understanding the systems' behavior has additional benefits for systems, as it improves developer productivity and reduces debugging and diagnostics time. This stresses the need for a framework which allows observation of an application. To address this, engineers collect data about an application's execution, called a trace. Tracing tools and instrumentation can be used to gather and measure these traces[1][2].

As these systems are growing, tracing tools and instruments need to adapt to these increasing complexities, since they need to measure the performance of these more complex and deeper embedded operations[1]. One problem with the usage of these tracing tools arises when they introduce overhead. Overhead embodies the latency, jitter and decremented throughput that is additional to the raw program, possibly introduced by tools or instrumentation to track or measure metadata [3]. If the overhead is relatively large, this trade-off of performance versus information can cause the instrumentation to become obsolete. To mitigate the unexpected side effects of automated tracing, it is important to provide insights into how tracing can impact the execution time of various program types.

OpenTelemetry provides a tracing specification which is followed by many other tracing tools, as it attempts to conventionalize tracing [4]. Microsoft created OpenTelemetry the official telemetry solution for all .NET base class libraries and Microsoft's frameworks, which illustrates OpenTelemetry's significance [5]. Further background knowledge about OpenTelemetry will be provided in the subsequent background subchapter. As OpenTelemetry functions as a new innovative and general observatory framework, it is largely adopted in the development of distributed systems. This project evaluates the overhead of OpenTelemetry as this can have significant implications for the use of it. OpenTelemetry attempts to provide an observatory framework for eleven programming languages to adopt a cross-platform nature. However, at the time of this project, the release status of OpenTelemetry differs for these languages, ranging from not being implemented to stable in production [5]. For this

project, we opted for two programming languages to assess the performance overhead, ensuring that OpenTelemetry fully supports automatic instrumentation in these languages. We chose two impactful languages from the selection of programming languages that conform to this criterion; C# and Python. Both these languages are within the domain expertise of our team and thus evaluated as suitable for this project. Furthermore, we make distinctions between different types of applications. We introduce several benchmark programs with different types of computational and message passing behavior, to evaluate distinct kinds of overhead when running them instrumented by OpenTelementry. Furthermore, we implement tail-sampling and measure the overhead of the benchmark programs together with the tail-sampling, as tail-sampling is a very common technique to select logs from the log domain.

2 Background

2.1 Background on the Application

System architecture has evolved from monoliths to distributed systems, complicating the creation of an adequate tracing framework. Due to the larger scale, higher complexity and use of multiple processes, the necessity as well as the complexity of creating such a framework has grown. In this section, we discuss the background of tracing in distributed systems and explore previous techniques along with their challenges. These challenges are ultimately overcome by OpenTelemetry.

Different instances of a distributed system are identified as individual (micro)services. These microservices can differ in internal logic, language and application [6]. However, there exists a universal type of information in a distributed system, which represents distributed transactions; a distributed trace [7]. The connections between microservices and other parts of a distributed system become increasingly complex as systems grow. Historically, the instrumentation of systems would be manually inserted by the programmer. However, due to this growing complexity, automatic instrumentation is gaining popularity [8]. Automatic tracing is a type of automatic instrumentation, reducing the operational effort required to manage and understand the interdependencies between services [9]. Tracing ultimately provides data representation and tooling to process, instrument and export data by defining the transactions of each of these microservices and propagating context through them [7]. This acquires insights in possible bottlenecks or hotspots of an application, identifying stragglers, after which adequate measures can be made to solve these problems, resulting in a system with a higher performance [4].

Earlier automatic instrumentation are performance metrics such as APM (Application Performance Metrics). APM attempted to provide insights into the application's behavior but was confronted with several problems, including incompatibility of different vendors. This resulted in challenges when migrating these metrics, and libraries struggled to offer sufficient performance metrics for applications utilizing different vendors [9]. In addition to this, the overhead of using APM remained unpredictable. As a solution to these problems, OpenTelemetry was introduced. OpenTelemetry is a community-driven project to define a common format and tooling to collect trace data [4]. Trace data is collected by instrumenting an application at interesting points. Network calls, exceptions, and conditional statements are examples of interesting points. All traces and metrics are produced by the OpenTelemetry library itself. This results in a vendor neutral framework, as vendors are cut out of the gathering information step, and are solely involved in turning the gathered information into useful information [5]. In addition, OpenTelemetry promises consistent as well as high performance. It is especially useful for distributed systems, as it works cross-platform and automatically propagates context through multiple layers of the system and stitches the gathered information together into a trace.

2.2 Related work

When doing research, it is important to investigate other research in the same area. In [1], the researchers also try to investigate the overhead of various tracing instrumentation tools. They also investigated OpenTelemetry. Their findings were mostly that they extended the MooBench to support OpenTelemetry. MooBench is a benchmark which measures the overhead of monitoring frameworks. They then experimented by running the benchmark on a desktop PC and a Raspberry Pi 4. Ultimately, they showed that Kieker was found to have less overhead than inspectIT and OpenTelemetry when processing traces. This is necessary for our research because it showed us that it had been done before, and it verified for our project that there is some overhead when running tracing instrumentation. This means that this verifies our results.

3 System Design

Our research focuses on two types of tests, the first is a set of synthetic tests made to benchmark individual types of workloads. The main focus for this was on two different kinds of CPU/CPU+Memory types of loads as well as different network loads, where CPU or Memory usage is minimized elsewhere. Next to the synthetic set, we also instrumented popular packages on GitHub to see how performance would be impacted in a setup that is closer to a real world scenario. All tests are run both with and without OpenTelemetry's automatic instrumentation. We measure the difference in response time between these two. From there, we can show how much overhead OpenTelemetry's sampling has, if at all.

3.1 Requirements

Before the implementation of these synthetic benchmarks, we determined what our benchmarks and overall project would need to contain in order to show the overhead of OpenTelemetry. For this, we have determined several functional requirements.

FR1 Our synthetic benchmarks must be able to work both independently and with automatic sampling from OpenTelemetry.

FR2 The benchmarks must have some form of network connectivity to measure the response time.

FR3 The different benchmarks should put load on different parts of a system to differentiate between response time of different types of load.

FR4 The distributed system must consist of at least three different subcomponents/services.

FR5 There must be a set of benchmark programs to evaluate the instrumentation overhead. They should cover different application domains. Using the different sampling ratios, we should evaluate the instrumentation on these programs. We should use 2 programming languages.

3.2 System overview

To test the performance overhead of OpenTelemetry we used the following systems:

- A synthetic benchmark suite in C# and Python
- Used a new expansive Microsoft Aspire for .NET

3.3 Synthetic Setup

To test the overhead, we implemented tests for two different languages: C# and Python. We created a setup of 3 servers: one front-end, one middle-end, and one back-end server. Each of these servers communicated in a serial fashion, with the back-end server doing some amount of computation. This setup was chosen to specifically focus on both the networking as well as the computation overhead. This way, we ensure a complete trace consisting of a start, middle part, and an end point. Each of these servers communicated in a serial fashion.

For C#, the servers utilized the ASP .NET MVC framework for the front-end and ASP .NET APIs for the middle and back-end. The Python servers employed Flask. These frameworks were selected as they are supported according to the Open Telemetry documentation. Both implementations supported the following endpoints:

- Small
- Large
- Factorial
- Pi
- Matrix

Endpoints **Small** and **Large** focus on analyzing the overhead on network performance. Specifically, **Small** represents the smallest unit of data transferable between the front and back-end, whereas calling the **Large** endpoint generates payloads of 4 MB from front-end to back-end.

These tests were used to approximate the impact on network-heavy workloads in an isolated fashion.

Factorial calculates a certain number of factorial numbers, **Pi** calculates the first 20 digits of Pi, and **Matrix** multiplies a 40x40 matrix. We chose these functions for their varying distribution between CPU and Memory load, allowing us to approximate the overhead of OpenTelemetry on compute-intensive workloads while isolating the network component.

All setups are instrumented to export OpenTelemetry's results to a Zipkin server. Additionally, both language implementations included two special enhancements:

- Python features a custom implementation for tail sampling, which samples only those requests that raised an exception during execution. To test performance on tail sampling we used modified versions of the factorial benchmark with 10%, 5% and 1% probabilities for an exception.
- For C#, load balancing was added. During testing, it was observed that without sufficient load, the overhead was barely measurable. Therefore, test scripts were extended to first run for 7 seconds and poll the current CPU usage of the running server until the average CPU load (excluding the first 2 seconds of measurement) is within a specified margin of error (currently set at 7%).

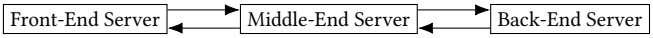


Figure 1. Synthetic Server Setup

4 Implementation

We have implemented our synthetic benchmarks in two different programming languages to be able to understand if the overhead of OpenTelemetry differs significantly depending on the implemented language. We have, arbitrarily, chosen the languages Python and C# for our comparison. Both implementations contain the same set of benchmarks.

4.1 Python

For the Python implementation, we have created a web server which can run all the benchmarks. We also created a middle- and backend server which can connect to the server mentioned above to perform the networking tests.

4.1.1 Tail sampling. Additionally, we have two servers which also compute the factorial, but have a small chance to raise an exception after calculating the factorial. One of these servers still traces every request, but the other server samples only those requests which threw an exception.

For these web servers we have used the Flask framework with which we can run a local web server [10].

4.2 C#

Our servers are running ASP .NET Ubuntu. All benchmarks have been run with K6 to measure the response time of the web servers with and without sampling. We have performed these tests by sending a constant amount of requests per second to our web server, for a duration of five minutes. To determine the rate per second, for each experiment, we tried to find a rate as high as possible without the server dropping requests. From the results, we ignore the first and last 10%, to ensure our measurements are taken when the system is in a consistent state and to avoid measurements at points in time where the system might not have fully allocated its resources to the web server. It also ensures that the just-in-time compilers for the languages have kicked in and are running fully optimized code.

5 Experimental Results

To evaluate the performance of OpenTelemetry we ran our web-servers and used K6 to send a constant rate of requests to the web server to see how they would perform under high load. Each experiment had a duration of 5 minutes, in which K6 measures the duration of each request. This section will specify results for the programming languages Python and C#, together with an overall analysis that combines results from different measurements in an analysis.

5.1 Python

The Python tests have all been run on the DAS-5 cluster. The rates have been manually determined, by trying to find the limit at which the servers could be run using telemetry without dropping requests. For most tests this rate was 400 requests per second, except for the matrix tests which had to be run at 40 requests per second.

The results on the Python benchmarks can be seen in Table 1 and in Figures 5 and 6 and the results on the tail sampling can be found in Table 3 and in Figures 7 and 8.

5.2 C#

The C# tests were run on a personal laptop (an XPS 15 from before 2020). The rates have been determined by automatically trying to reach a certain average CPU load. The results on the C# benchmarks can be seen in Table 2 and in Figures 3 and 4.

For C# it was also tested what the overhead was on different targeted CPU loads. Figure 2 shows for example the effect of the CPU load for the pidigits tests.

All CPU load results can be found in Appendix D.

5.3 Overall Analysis

For both these programming languages, we conducted measurements of four different benchmarks. These benchmarks show different overhead measurements, as illustrated in Table 1 and Table 2 in Appendix D. The average, Medium and maximum are measured. We further measured the affect of using tail-sampling, shown in Table 3. Overall, the results indicate using OpenTelemtry as a tracing method introduces overhead. One interesting point illustrated by the measurements is the difference in average and median, illustrating that OpenTelemetry experiences higher peaks in overhead. The average is more extensively influenced by higher peaks of overhead in comparison to the mean. This is more elaborately illustrated in Figure 3 for C#, Figure 5 for Python and Figure 2 to compare. This is further elaborated in the discussion.

To attempt to distinguish these peaks the longest request duration of the requests is measured. This is shown in Table 1 and Table 2, for both C# and Python. As illustrated in Figure 4 and Figure 6, the impact is very different on the languages C# and Python. This shows the different nature of these programming languages.

The differences in benchmarks show differences in programming languages. This indicates the differences in how OpenTelemetry propagates through different layers in C# and Python.

We measured the overhead of OpenTelemetry with different CPU loads, as illustrated in Figure 9 and Figure 10 in Appendix E. To properly measure overhead, the CPU load should be adequate. We could not formulate any overall result from the measurements, considering the scope of this project

Lastly, the impact of tail-sampling is illustrated in Figure 7. It shows that using tail-sampling is beneficial for the overhead introduced by OpenTelemetry. This can by the explained due to less information is saved and logged.

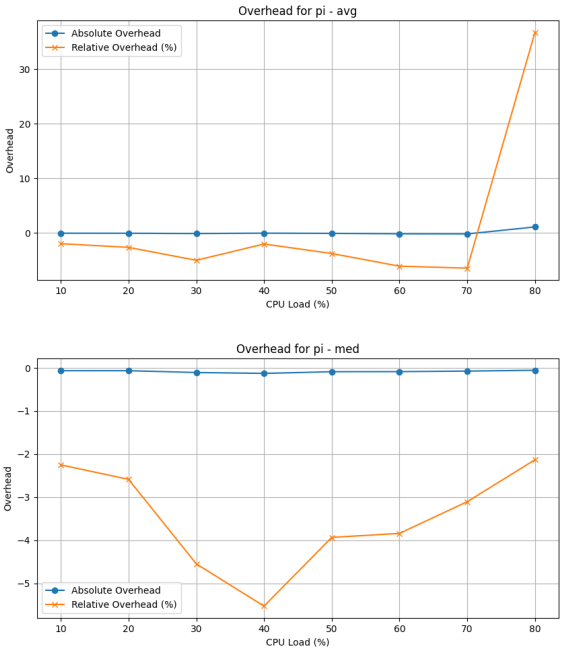
6 Discussion

When comparing the C# and Python Factorial benchmarks, it has been shown that in Python benchmarks, OpenTelemetry resulted in an average performance increase of 37.4%. In comparison, the mean performance increase was much higher at 320.69% in the C# benchmarks. This may suggest that the overhead introduced by OpenTelemetry is larger in C# compared to Python.

6.1 Benchmark Discussion

In the **Factorial benchmark for Python**, the average runtime was 2.08 ms without OpenTelemetry. When it was included, the average runtime was 2.86 ms. What is interesting is that the maximum runtime came out much larger without OpenTelemetry, but this can be seen as an outlier. Furthermore, the median for the average runtime came out to 1.76 ms without OpenTelemetry, and 1.85 ms for the inclusion. This only added a 5.58% overhead. Therefore, the large average overhead may be explained by the large outlier values of 137.86 ms. Interestingly, the C# ran the benchmark faster without overhead, at 1.31 ms, in comparison to Python's 2.08

Figure 2. Median and average overheads for different CPU loads on pidigits (C#)



ms. Yet, including OpenTelemetry, it became 5.53 ms for C#, which signifies a 320.69% increase.

The **Pidigits Benchmark** showed that Python has a substantial 271.44% increase, but C# had a lower impact at 63.4%. Also, the average values were lower for the C#, running at 2.31 ms with no overhead and at 3.77 ms with overhead (increase of 63.4%).

The **Matrix Benchmark** showed that Python's average increase was 9.92%, whereas C# showed an extreme 1501.15% increase. But the overall runtime of each result for C# was much quicker than Python (1.37 ms and 21.98 ms with overhead compared to 571.39 ms and 628.06 ms with overhead). The Matrix benchmark may potentially indicate language-specific differences in handling matrix operations.

The **Network Tests Benchmarks** showed that in both small and large network benchmarks, C# consistently exhibited higher average increases of 124.98% and, 21912.2% respectively, compared to Python (9.52% and 4.64%, respectively). This implies that OpenTelemetry's impact on network-related tasks is more pronounced in C#. But once again, the runtime is lower for C# overall.

The **Tail Sampling Efficiency** showed a notable reduction in runtime across different probabilities (P=0.1, P=0.05, P=0.01), with an average decrease of 92.35%. This shows a significant benefit from Tail Sampling, indicating its potential to mitigate OpenTelemetry's impact on runtime.

Overall, the tests all seem to run much quicker on C#, when looking at the runtime, even though the impact of overhead from OpenTelemetry is much larger when looking at the percentage values. But overall in the C# benchmarks, displayed an order of magnitude increases in runtime when including overhead when compared to Python. This suggests that certain types of computations or operations in C# may be more affected by the integration of OpenTelemetry.

7 Future work

In this report, there was an analysis performed on the overhead of OpenTelemetry for benchmarks written in two languages: Python and C#. OpenTelemetry natively supports many more languages, such as C++, Java, and JavaScript. Community projects also exist that extend the reach of OpenTelemetry even further, to languages such as Dart [11], Julia [12], and Scala [13]. Future work could perform similar benchmarks for other languages to inspect what the overhead of OpenTelemetry is for languages other than Python and C#, as we have seen that the overhead can vary a lot depending on the programming language used.

We have performed benchmarks for workloads that focus on network and computation. These put strain on the NIC and CPU of the device respectively. Further research could analyze the overhead of OpenTelemetry under load of other components of a system. For example, benchmarks could be created to put strain on the storage device (e.g. an SSD), a GPU, or internal memory. Storage device load would be an important measurement for the application of OpenTelemetry to databases, which store large amounts of data. GPUs could be tested for applications which offload graphics rendering, or the training of machine learning models. For the network tests it would also be interesting to see how much overhead is incurred by the messages sent between different servers, as these will also carry information about traces.

8 Conclusion

To conclude this report, it has been shown that OpenTelemetry generally introduces overhead in both Python and C# benchmarks, with a differing impact on different types of tests. Furthermore, by comparing tail sampling to no sampling, we show significant improvements in reducing runtime overhead. While we have shown that C#'s overhead is significant, it will still run the tests faster than Python. This could mean that C# is more overhead sensitive, but will have an overall faster execution time.

References

[1] D. Reichelt, S. Kühne, and W. Hasselbring, "Towards solving the challenge of minimal overhead monitoring," 04 2023.

[2] W. Hasselbring, "Benchmarking as empirical standard in software engineering research," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 365–372. [Online]. Available: https://doi.org/10.1145/3463274.3463361

[3] A. Gelberger, N. Yemini, and R. Giladi, "Performance analysis of software-defined networking (sdn)," 08 2013, pp. 389–393.
[4] J. Ekenstedt, "Detecting time inefficiencies in service-oriented systems using distributed tracing," 2023.
[5] Dec 2023. [Online]. Available: https://www.cncf.io/projects/opentelemetry/
[6] A. Bento, J. Correia, R. Filipe, F. Araujo, and J. Cardoso, "Automated analysis of distributed tracing: Challenges and research directions," *Journal of Grid Computing*, vol. 19, 03 2021.
[7] D. Gomez Blanco, *Tracing*. Berkeley, CA: Apress, 2023, pp. 85–110. [Online]. Available: https://doi.org/10.1007/978-1-4842-9075-0_6
[8] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–28, 2018.
[9] S. Niedermaier, F. Koetter, A. Freymann, and S. Wagner, "On observability and monitoring of distributed systems–an industry interview study," in *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings 17*. Springer, 2019, pp. 36–52.
[10] [Online]. Available: https://flask.palletsprojects.com/en/3.0.x/
[11] Workiva, "Opentelemetry for dart," https://github.com/Workiva/opentelemetry-dart, 2021.
[12] J. Tian, "Opentelemetry.jl," https://github.com/oolong-dev/OpenTelemetry.jl, 2021.
[13] Typelevel, "otel4s," https://github.com/typelevel/otel4s, 2022.

Appendix A: Timesheets

Name	Raymond	Chris H.	Sebastiaan	Stef	Thea	Chris
Thinking	13	7	13	12	12	13
Development	21	12	19	18	16	11
Experiments	20	10	18	11	13	10
Analysis	6	22	6	11	14	22
Writing	5	21	9	15	15	20
Wasted	13	10	12	14	6	3
Total	78	78	81	81	76	79

Appendix B: codebase

The code for our systems is publicly available at https://github.com/RaymondHoogervorst/DSlab_combined

Appendix C: Result tables

Table 1. Python Benchmark Results

Test	Average		Median		Maximum	
	No Telemetry	OpenTelemetry	No Telemetry	OpenTelemetry	No Telemetry	OpenTelemetry
Benchmark - Factorial	2.08	2.86 (+37.4%)	1.76	1.85 (+5.58%)	142.54	137.86
Benchmark - Pidigits	4.83	17.93 (+271.44%)	2.55	11.09 (+334.64%)	168.28	177.04
Benchmark - Matrix	571.39	628.06 (+9.92%)	575.32	649.88 (+12.96%)	1365.21	1181.22
Network - Small	58.91	64.53 (+9.52%)	58.05	63.86 (+10.02%)	189.04	215.22
Network - Large	1217.91	1274.46 (+4.64%)	1210.03	1255.45 (+3.75%)	3099.72	2457.13

Table 2. C# Benchmark Results

Test	Average		Median		Maximum	
	No Telemetry	OpenTelemetry	No Telemetry	OpenTelemetry	No Telemetry	OpenTelemetry
Benchmark - Factorial	1.31	5.53 (+320.69%)	0.19	1.45 (+661.34%)	93.64	18622.47
Benchmark - Pidigits	2.31	3.77 (+63.4%)	0.30	2.41 (+715.03%)	190.88	1778.83
Benchmark - Matrix	1.37	21.98 (+1501.15%)	0.20	3.73 (+1766.91%)	97.94	41510.45
Network - Small network	1.33	2.99 (+124.98%)	0.20	1.82 (+787.2%)	106.49	1768.57
Network - Large network	1.54	339.96 (+21912.2%)	0.23	335.38 (+148739%)	104.57	828.93

Table 3. Tail Sampling Results

Test	Average		Median		Maximum	
	No Sampling	Tail Sampling	No Sampling	Tail Sampling	No Sampling	Tail Sampling
P=0.1	20.84	2.19 (-89.51%)	1.29	1.25 (-3.45%)	4311.13	129.96
P=0.05	22.15	1.73 (-92.21%)	1.44	1.24 (-13.57%)	3481.81	144.76
P=0.01	25.69	1.45 (-94.34%)	1.45	1.22 (-15.76%)	5533.66	126.77

9 Appendix D: Figures

Figure 3. Median and average request durations for C# benchmarks

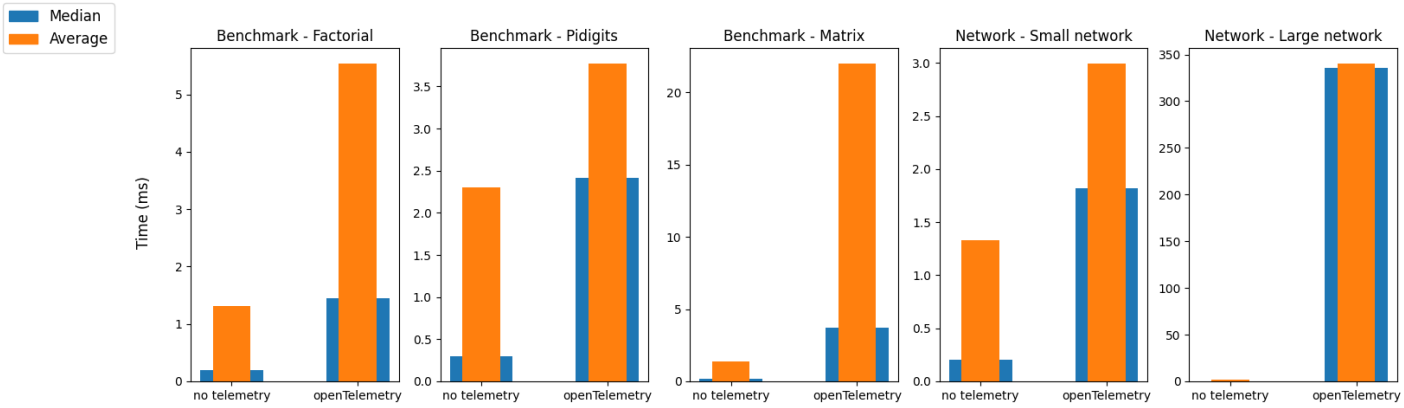


Figure 4. Maximum request durations for C#

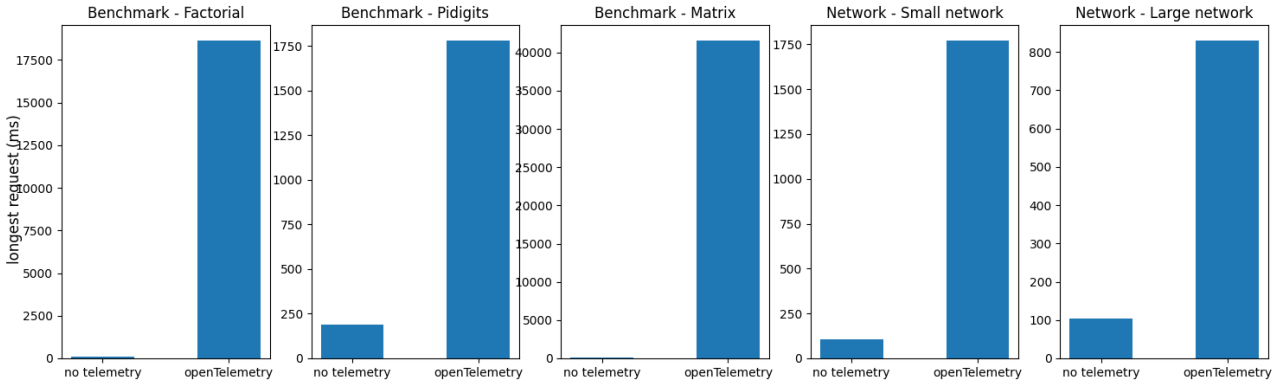


Figure 5. Median and average request durations for Python benchmarks

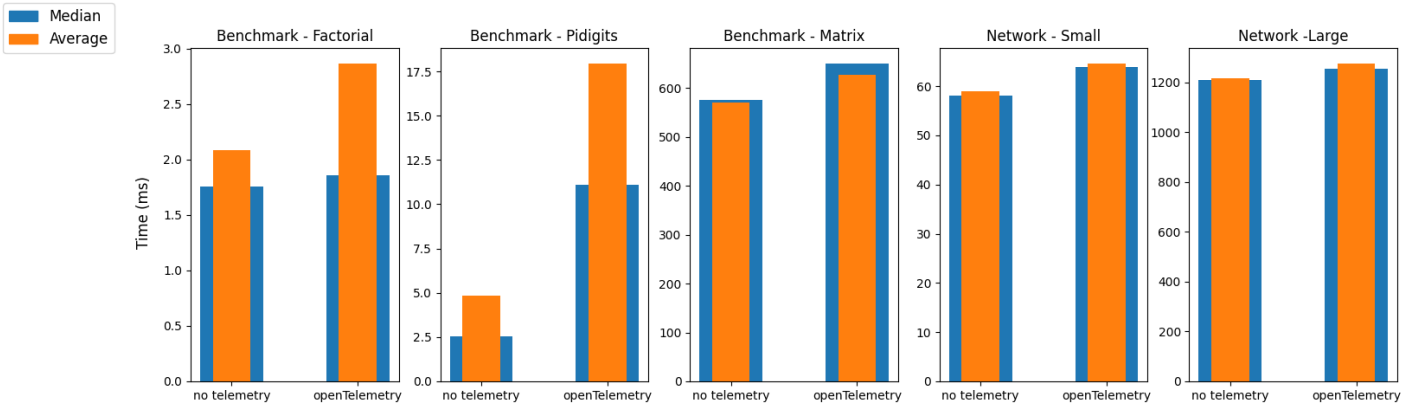


Figure 6. Maximum request durations for Python benchmarks

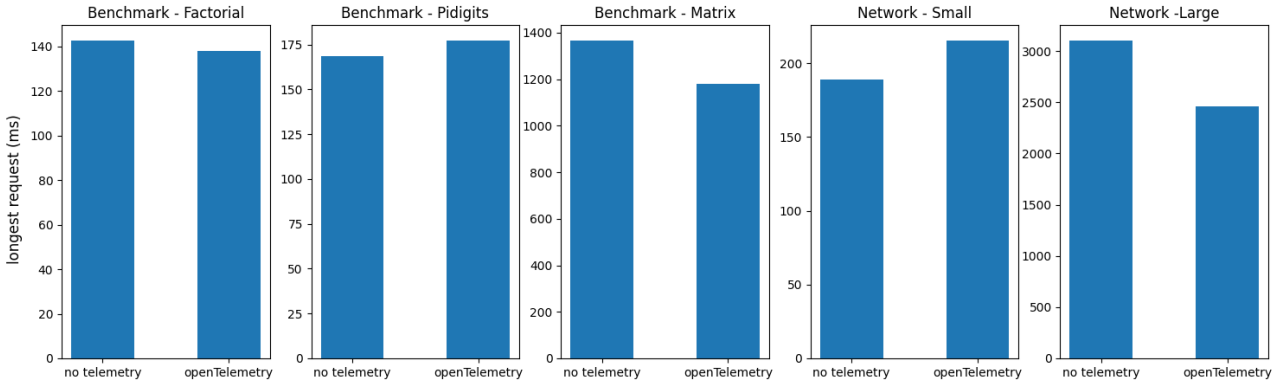


Figure 7. Median and average request durations for Python tail sampling

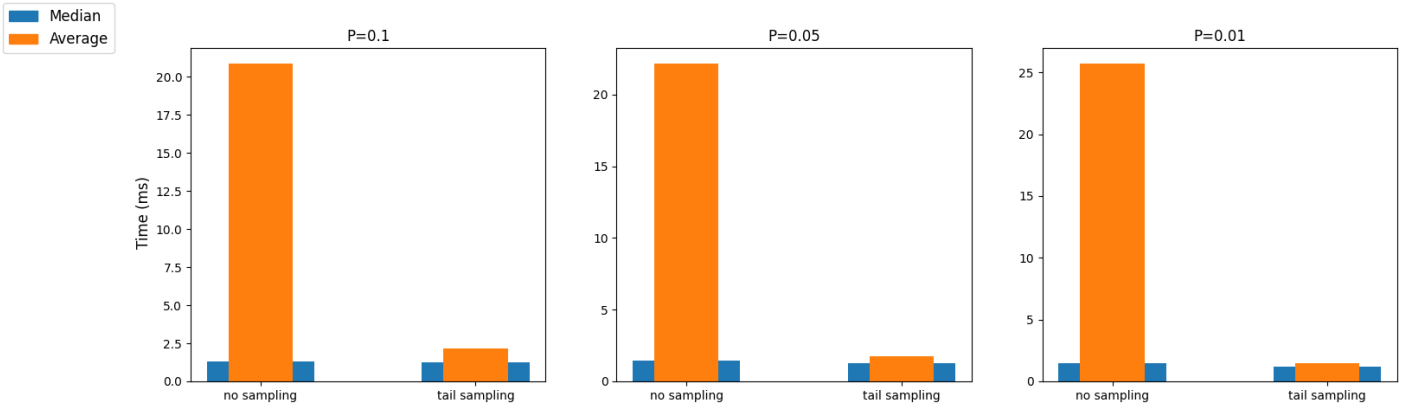
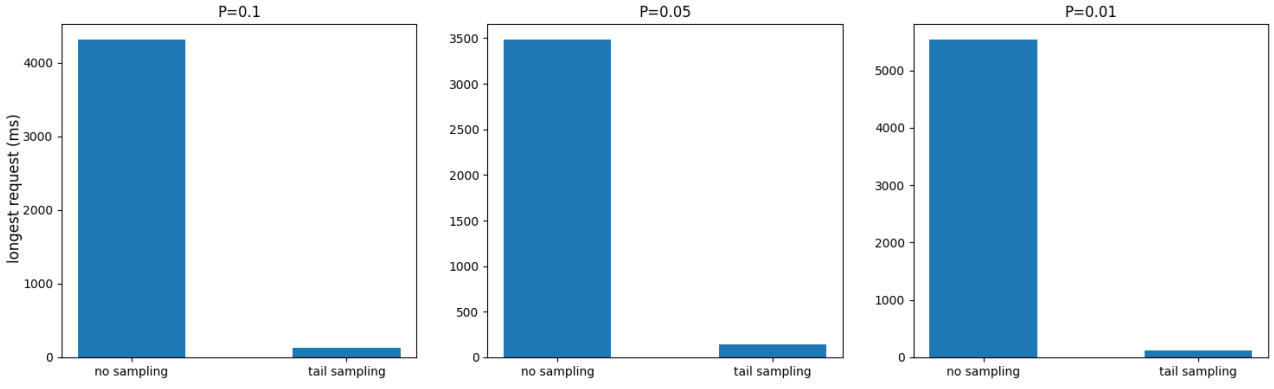


Figure 8. Maximum request durations for Python tail sampling



10 Appendix E: CPU load comparisons

Figure 9. Average overheads per CPU load for C#

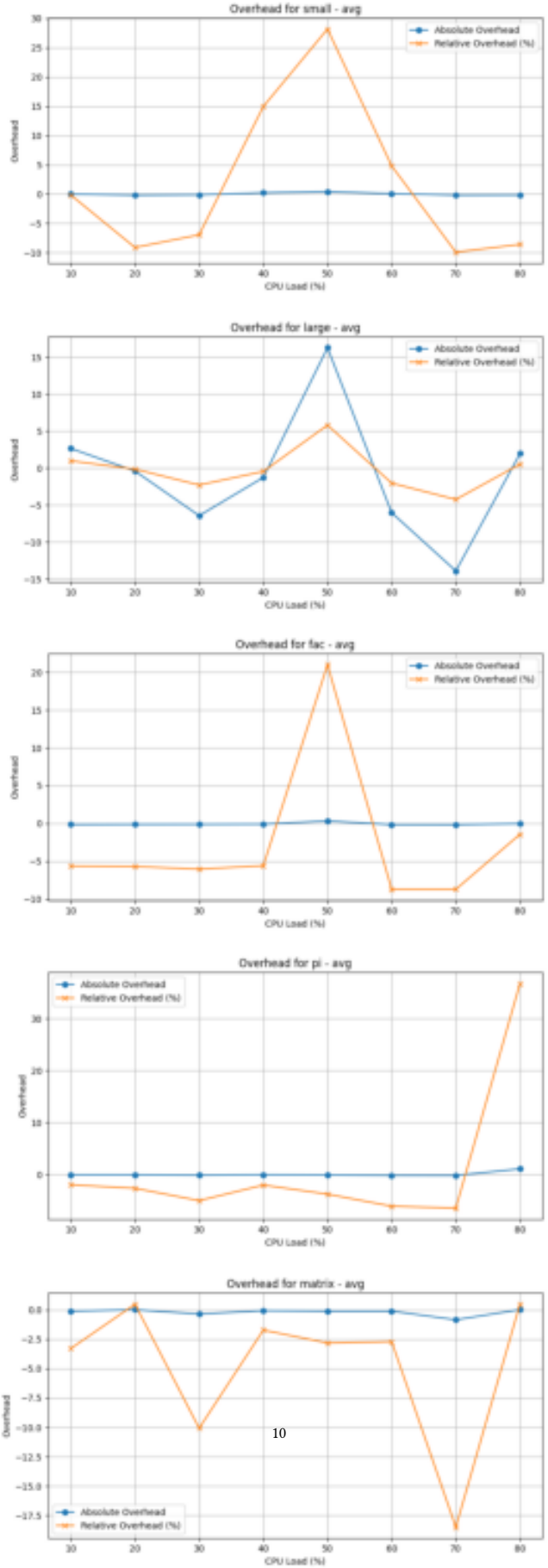


Figure 10. Median overheads per CPU load for C#