

Creating Tools for Analyzing the Performance of Applications

Tatiana Elizabeth Sanchez Sanin, *Student, University of Antioquia*

Abstract—This paper emphasizes the importance of application performance analysis in modern software development. Ensuring optimal performance is crucial for enhancing user experience and system efficiency. We leverage established profiling tools to gain a comprehensive understanding of application behavior and identify bottlenecks. These tools offer significant benefits by saving development time, providing detailed insights, and being adaptable to various programming environments and user skill levels.

Index Terms—Performance analysis, application tools, optimization

I. INTRODUCTION

In today's technology-driven world, applications are fundamental to countless operations. However, performance issues can significantly hinder their functionality and user experience. Application performance analysis plays a vital role in identifying and resolving these bottlenecks, ensuring applications operate at peak efficiency.

Developing advanced tools for application performance analysis is critical for several reasons. First, it directly improves user experience. Slow and unresponsive applications frustrate users, prompting them to seek alternatives. Performance analysis helps identify and resolve obstacles to a seamless user experience, enabling developers to refine applications for superior speed and responsiveness.

Second, applications can be significant consumers of system resources like CPU, memory, and storage. Performance analysis helps uncover areas of excessive resource consumption, allowing for code optimization and reduced resource demands. This optimization is crucial for maintaining efficient and sustainable application operations.

Finally, as applications grow in complexity and user base, scalability becomes increasingly important. Performance analysis ensures applications can accommodate escalating workloads without sacrificing speed or stability.

II. BACKGROUND OR THEORETICAL FRAMEWORK

Effective application performance analysis requires a comprehensive understanding of several theoretical domains. These fundamental concepts provide deeper insights into application behavior and enable developers to create high-performing software.

Central to any application is its interaction with underlying hardware. Analyzing performance necessitates understanding computer architecture, including the CPU, memory, and I/O devices. Efficient cache utilization is crucial, as inadequate utilization can lead to bottlenecks. Performance analysis helps

identify areas for cache optimization. Applications also interact with complex memory hierarchies, ranging from high-speed registers to slower main memory and storage devices. Analyzing memory access patterns reveals how efficiently the application utilizes these resources.

Furthermore, the CPU juggles multiple processes, with scheduling algorithms determining which process receives access at any given time. Analyzing the chosen scheduling algorithm (e.g., First-Come-First-Served, Round-Robin) helps assess how effectively the CPU is utilized by the application.

The operating system (OS) is responsible for allocating and utilizing system resources. A deep understanding of core OS functionalities is essential for effective performance analysis. The OS manages process creation, execution, and termination. Context switching, the process of switching between running processes, can impact application performance. Analyzing context switch frequency can reveal potential bottlenecks. Additionally, the OS handles memory allocation for applications. Analyzing memory management techniques employed by the OS helps identify potential issues like memory fragmentation that could impact application performance.

Finally, the algorithms and data structures within the application itself play a significant role in performance. Understanding these concepts allows for analyzing the time and space complexity of the implemented algorithms. Analyzing time complexity helps identify areas where calculations might be taking an excessive amount of time, leading to performance bottlenecks. The amount of memory required by an algorithm (space complexity) also needs consideration, as algorithms with high space complexity might lead to memory exhaustion and impact performance.

III. ASSESSMENT AND PLANNING

This section outlines the assessment and planning process for analyzing application performance using profiling tools.

A. Target Application Identification

The first step involves identifying the target application for performance analysis. This includes specifying:

- **Programming Language:** Python, Java, JavaScript, TypeScript
- **Platform:** Windows, Linux, Web-based

B. Profiling Tool Selection

Based on the target application's programming language and the performance aspect we want to analyze, select suitable profiling tools. Here's a breakdown of potential options:

1) Python Applications:

- **cProfile:** This built-in module provides a lightweight approach to profiling. It generates reports showing the time spent executing different code sections, helping identify performance bottlenecks.
- **Pyinstrument:** This third-party library offers a more user-friendly experience for profiling Python applications. It builds upon cProfile, providing visual representations of profiling data and call graphs that depict function call relationships and execution times.

2) Java Applications:

- **JMeter:** While primarily used for load testing in later stages, JMeter can be a valuable tool for initial assessment. It simulates load scenarios that stress identified bottlenecks, helping measure the impact of optimizations.

3) C/C++ Applications:

- **Caliper:** This framework allows embedding performance analysis capabilities directly into applications written in C/C++/Fortran. It facilitates performance measurement at runtime and is particularly suited for High-Performance Computing (HPC) applications but can be used with any language on Unix or Linux systems.

4) General-Purpose APM Tool:

- **New Relic:** This cloud-based application performance monitoring (APM) tool offers a comprehensive view of application health. It collects data on various aspects like server response times, database interactions, and user errors, providing real-time insights that can guide your assessment.

IV. PROFILING PROCESS

Detail the profiling process used by your tools to collect performance data. Explain the types of data collected and how it is used.

1) Python Applications:

- **cProfile:** This built-in module provides a lightweight approach to profiling. It generates reports showing the time spent executing different code sections, helping identify performance bottlenecks. To use cProfile, simply import the module and execute it as a function:

```
import cProfile
cProfile.run('your_function()')
```

```
import random
import cProfile

def calculate_pi(iterations):
    """Estimates pi using the Monte Carlo method."""
    count_in_circle = 0
    for _ in range(iterations):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) <= 1:
            count_in_circle += 1
    pi_estimate = 4 * count_in_circle / iterations
    return pi_estimate

def main():
    """Runs the pi calculation and prints the estimated value."""
    iterations = 1000000 # Adjust this value for different test durations
    pi = calculate_pi(iterations)
    print(f"Estimated value of pi: {pi}")

if __name__ == "__main__":
    main()
    cProfile.run('main()')
```

Fig. 1. Example of Python code for profiling with cProfile

- **Pyinstrument:** This third-party library offers a more user-friendly experience for profiling Python applications. It builds upon cProfile, providing visual representations of profiling data and call graphs that depict function call relationships and execution times. To use Pyinstrument, you need to install the package and execute it via the command line:

```
pip install pyinstrument
pyinstrument your_script.py
```

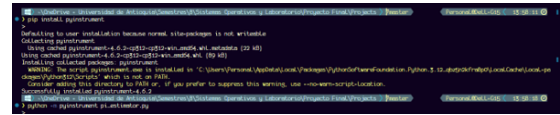


Fig. 2. Example of Python code for profiling with PyInstrument

2) Java Applications:

- **JMeter:** JMeter is an effective tool for performance testing Java applications, particularly for load testing web services. To use JMeter, create a test plan that includes a simple HTTP Request targeting the server on port 8000. Configure JMeter to send multiple concurrent requests to analyze various performance metrics such as response times, throughput, and error rates.

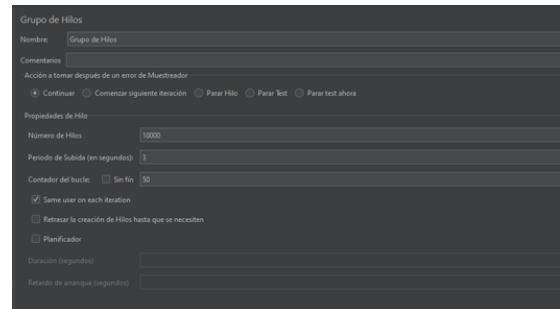


Fig. 3. Creating an HTTP Request Test Plan in JMeter

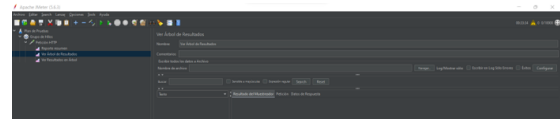


Fig. 4. Analyzing Response Times and Throughput in JMeter

3) **New Relic Agent:** New Relic offers a cloud-based application performance monitoring (APM) tool. This section details the installation process for the New Relic agent and how to access and analyze collected data.

a) **Installation Process:** The installation process involves the following steps:

- 1) **Login:** Access your New Relic account and navigate to the "Add more data" section.
- 2) **Select Operating System:** Choose your operating system from the options provided.

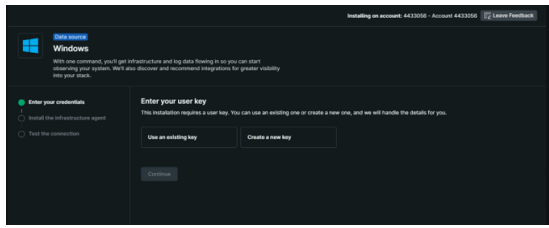


Fig. 5. Example of the Selection of Windows as Operative System

- 3) **Obtain License Key:** New Relic will provide a unique license key specific to your account. Record this key for later use.
- 4) **Download and Install Agent:** New Relic will offer a command tailored to your operating system for installing the agent. Copy this command and execute it in your terminal to begin the installation process.

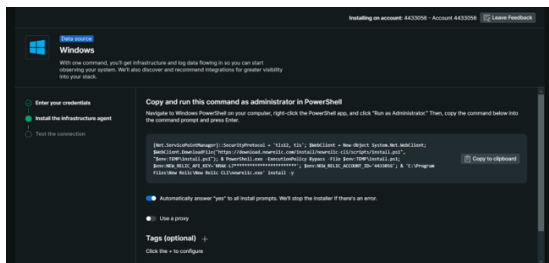


Fig. 6. Example of command for Windows

b) *Verification:* Once the installation is complete, verify it by checking the New Relic dashboard for incoming data from your monitored application.

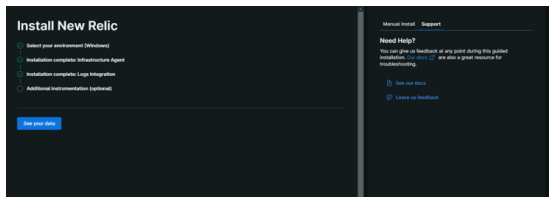


Fig. 7. Example of successful installation

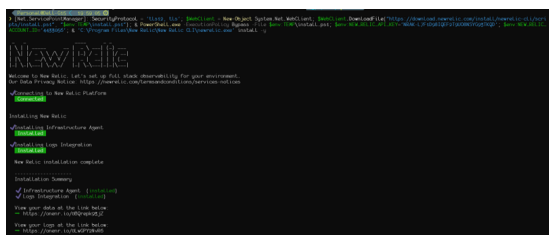


Fig. 8. Example of agent installation on terminal

c) *Data Analysis in New Relic:* New Relic offers a comprehensive web interface for analyzing collected performance data. Here's how to access and utilize this data:

- 1) **Access Logs:** Navigate to the "Logs" section within the New Relic web interface sidebar. This section allows you to analyze the data gathered by the agent.

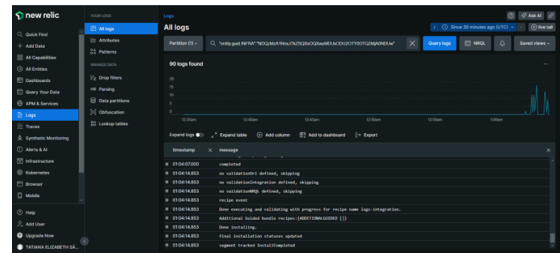


Fig. 9. Example of logs for operative system

By following these steps, you can effectively install the New Relic agent and leverage its capabilities to gain valuable insights into application performance.

V. ANALYSIS AND INTERPRETATION

A. PyInstrument Analysis and Interpretation

PyInstrument offers a different approach for performance analysis in Python.

- 1) **PyInstrument Overview:** PyInstrument is a third-party Python library that utilizes statistical profiling. Unlike cProfile, it doesn't track every function call but periodically samples the call stack at specific intervals (e.g., every millisecond). This approach offers lower overhead on your application's performance compared to cProfile's more comprehensive tracing.
- 2) **PyInstrument Output:** PyInstrument typically generates output in the form of a call graph or a line-by-line profile. The output format might vary depending on the specific configuration used.

1) Call Graph::

- Visualizes a hierarchical representation of function calls.
- Functions with higher call counts appear larger in the graph.
- Arrows indicate call relationships between functions.

2) Line-by-Line Profile::

- Shows the percentage of time spent executing each line of code.
- Lines with higher percentages are potential bottlenecks.

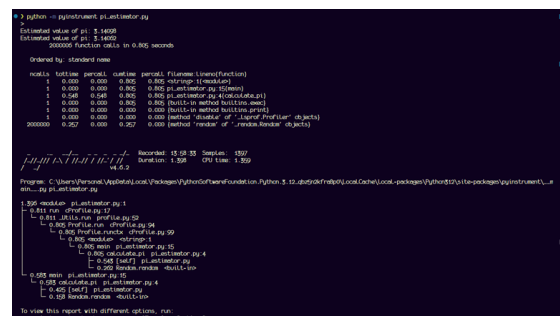


Fig. 10. Results from PyInstrument Analysis

3) Interpretation of the Results:

- **Estimated Value of Pi:** - The script `pi_estimator.py` estimates the value of pi as 3.14902 based on two calculations.
- **Function Call Summary:** - The profiler executed 2,000,000 function calls in 0.805 seconds.
- **Ordered by Standard Name:** - This section lists the functions called during the script execution, ordered by their names:

```

ncalls  tottime  percall  cumtime
percall filename:lineno(function)
1      0.000    0.000    0.805
0.805  <string>:1(<module>)
1      0.000    0.000    0.805
0.805  pi_estimator.py:15(main)
1      0.257    0.257    0.805
0.805  pi_estimator.py:4(calculate_pi)
1      0.000    0.000    0.000
0.000  {built-in method builtins.exec}
1      0.000    0.000    0.000
0.000  {built-in method builtins.print}
1      0.000
0.000    0.000    0.000
{method 'disable' of '_lsprof.Profiler' objects}
2000000 0.548
0.000    0.548    0.000
{method 'random' of '_random.Random' objects}

```

- **ncalls:** Number of calls to the function. - **tottime:** Total time spent in the function (excluding time made in calls to sub-functions). - **percall:** Average time per call. - **cumtime:** Cumulative time spent in this function and all sub-functions (from invocation till exit). - **filename:lineno(function):** Location of the function in the source code.

- **Graphical Representation of Time Consumption:** - This representation shows a hierarchical tree showing how much time was spent in each function, and the call relationships between them:

```

Program: C:\Users\Personal\AppData\Local\
Packages\PythonSoftwareFoundation.Python.3.12
_qbkn8r2jf7nap\LocalCache\Local-packages\
Python312\site-packages\pyinstrument\...\
__main__.py pi_estimator.py

```

- Time spent in various parts of the program is visually represented:

```

- 0.811 <module> pi_estimator.py:1
- 0.811 utils.run_run_profile.py:7
- 0.805 Profile.runcctx Profile.py:194
- 0.805 <module> pi_estimator.py:1
- 0.805 main pi_estimator.py:15
- 0.548 calculate_pi pi_estimator.py
- 0.548 Random.random <built-in>

```

- This tree shows how the total time is divided among the functions:

```

- 0.811 seconds in the <module> at
  pi_estimator.py:1.
- 0.811 seconds in utils.run at
  run_profile.py:7.
- 0.805 seconds in Profile.runcctx at
  Profile.py:194.

```

```

- 0.805 seconds in <module> at
  pi_estimator.py:1.
- 0.805 seconds in main at
  pi_estimator.py:15.
- 0.548 seconds in calculate_pi at
  pi_estimator.py:4.
- 0.548 seconds in Random.random (a built-in
  method).

```

Key Takeaways:

- **Most Time-Consuming Part:** The most time-consuming part of the script is the `calculate_pi` function, specifically the `Random.random` method, which takes 0.548 seconds.
- **Execution Time:** The entire script executes in approximately 0.805 seconds.
- **Profiling Details:** The details of the function calls and their execution times can help identify performance bottlenecks, which in this case are primarily within the random number generation for pi estimation.

By analyzing these details, we can optimize specific parts of the code if needed, especially focusing on the `calculate_pi` function and its usage of random number generation.

B. cProfile Analysis Interpretation

- **cProfile Output Breakdown:** Here is an explanation of the meaning of the columns in the cProfile report.
 - **ncalls:** The total number of times a function was called.
 - **tottime:** The total time spent executing the function, in seconds.
 - **percall:** The average time spent per call to the function, in seconds.
 - **cumtime:** The total time spent executing the function and all its subcalls, in seconds.
 - **filename:lineno(function):** The location of the function definition (file name and line number).
- **Bottleneck Identification:** To identify bottlenecks we have to analyze the cProfile report to pinpoint functions with high values in specific columns.

```

C:\Users\Personal\AppData\Local\Programs\Python\Python312\Scripts>python pi_estimator.py
Estimated value of pi: 3.14236
Estimated value of pi: 3.14296
2000000 function calls in 0.559 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1      0.000    0.000    0.559    0.559 <string>:1(<module>)
1      0.000    0.000    0.559    0.559 pi_estimator.py:15(main)
1      0.415    0.415    0.559    0.559 pi_estimator.py:4(calculate_pi)
1      0.000    0.000    0.559    0.559 {built-in method builtins.exec}
1      0.000    0.000    0.000    0.000 {built-in method builtins.print}
1      0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
2000000 0.145    0.000    0.145    0.000 {method 'random' of '_random.Random' objects}

```

Fig. 11. Results from cProfile Analysis

1) Interpretation of the Results:

- **Execution Summary**
 - **Estimated value of pi:** 3.14236
 - **True value of pi:** 3.14296
 - **Execution time:** 0.559 seconds
 - **Total function calls:** 2,000,006

- **Function Call Analysis:** The cProfile output lists function calls, their cumulative time, and the time per call.
- **Definitions**
 - **Total Time (tottime):** The total time spent in the given function (excluding time made in calls to sub-functions).
 - **Per-Call Time (percall):** The time spent per call. It is calculated as tottime/ncalls for the function itself and cumtime/ncalls for cumulative time.
 - **Cumulative Time (cumtime):** The total time spent in this function and all sub-functions.
- **Functions Analysis**
 - **Module Level (<module>):**
 - * **ncalls:** 1
 - * **tottime:** 0.000 seconds
 - * **cumtime:** 0.559 seconds
 - * The total execution time for the script is 0.559 seconds. The script contains the main logic and function calls.
 - **Main Function (pi_estimator.py:15(main)):**
 - * **ncalls:** 1
 - * **tottime:** 0.000 seconds
 - * **cumtime:** 0.559 seconds
 - * This is the main entry point of the script, calling the calculate_pi function.
 - **Calculate Pi (pi_estimator.py:4(calculate_pi)):**
 - * **ncalls:** 1
 - * **tottime:** 0.413 seconds
 - * **cumtime:** 0.413 seconds
 - * The calculate_pi function is where the core logic to estimate pi resides. It takes the majority of the time.
 - **Built-in Methods:**
 - * **exec (builtins.exec):** 0.000 seconds, likely used for executing the script.
 - * **print (builtins.print):** 0.000 seconds, used for outputting results.
 - **Profiler Methods:**
 - * **disable (_lsprof.Profiler):** 0.000 seconds, disables the profiler after profiling is done.
 - **Random Method (_random.Random.random):**
 - * **ncalls:** 2,000,000
 - * **tottime:** 0.146 seconds
 - * **cumtime:** 0.146 seconds
 - * This function is called 2,000,000 times, indicating that random numbers are generated frequently, likely for Monte Carlo simulation to estimate pi.
- **Insights and Recommendations**
 - The majority of the time is spent in the calculate_pi function (0.413 seconds).
 - Random number generation takes a significant amount of time given the high number of calls (0.146 seconds for 2,000,000 calls).
 - Printing and profiler disabling are negligible in terms of time consumption.

Possible Optimizations

- **Optimize Random Number Generation:**
 - * Consider alternative methods or libraries for faster random number generation if feasible.
- **Optimize the Pi Calculation Logic:**
 - * Review the calculate_pi function for any potential optimizations in the algorithm.
- **Reduce Print Statements:**
 - * If there are multiple print statements inside loops, they can significantly slow down the execution. Reduce or buffer output if applicable.

C. Comparison between PyInstrument and cProfile:

- PyInstrument provides a more lightweight profiling approach, suitable for applications where cProfile's overhead might be impactful.
- For a granular understanding of function execution time, cProfile is preferred.
- Combining PyInstrument and cProfile can provide a broader perspective, using PyInstrument for high-level profiling and cProfile for detailed analysis of specific functions.

D. JMeter Result Analysis

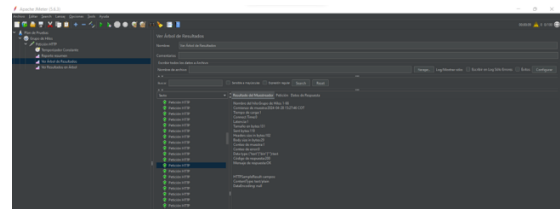


Fig. 12. Results from JMeter Analysis

- **Test Plan Configuration:** These configurations indicate that the test simulates 10,000 concurrent users ramping up over 3 seconds, each performing 50 iterations of the HTTP request.
 - **Threads (Users):** 10,000
 - **Ramp-Up Period:** 3 seconds
 - **Iterations:** 50
- **Observations:** HTTP Requests (Petición HTTP)

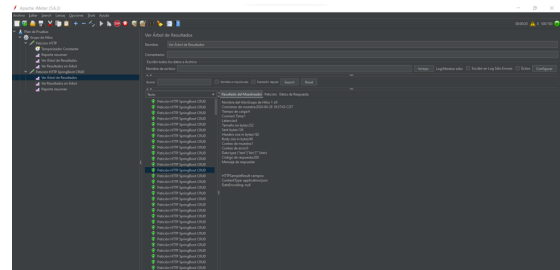


Fig. 13. Results from JMeter Analysis

- **Response Code:** 200 (Success)

– Request Details:

- * **Response Time (ms):** Response time in milliseconds.
- * **Bytes:** The size of the response.
- * **Sent Bytes:** Size of the request sent to the server.
- * **Latency (ms):** Time taken for the first byte to be received.
- * **Connect Time (ms):** Time taken to establish the connection.

• Response Information

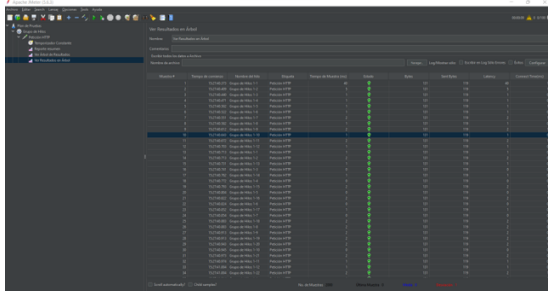


Fig. 14. Results from JMeter Analysis

- **All requests returned a 200 status code:** Indicates successful handling by the server.

– Load Time, Connect Time, and Latency:

- * These metrics are essential to evaluate server and network performance.

• Performance Metrics

- **Load Time:** Measures how long it takes for the request to complete.
- **Connect Time:** The time taken to establish a TCP connection to the server.
- **Latency:** The time until the first byte of the response is received, indicating server responsiveness.

• Insights

- **Server Performance:** The fact that all requests returned a 200 status code suggests the server handled the high load effectively without errors.
- **Potential Bottlenecks:**
 - * **Load Times:** If high, may indicate the server is under strain.
 - * **Connect Times and Latency:** If these are high, it might point to network issues or delays in server response times.

E. New Relic Log Record Analysis

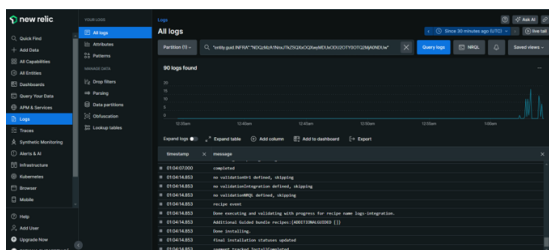


Fig. 15. Results from NewRelic Analysis

1. Successful Account Login:

- **Message ID:** 9db50006-285b-4eb0-a7bc-9fb1be05da34

• Account Details:

- **Security ID:** S-1-5-18
- **Account Name:** DELL-G15\$
- **Account Domain:** WORKGROUP
- **Logon ID:** 0x3E7

• Login Information:

- **Logon Type:** 5 (Service startup)
- **Restricted Admin Mode:** No
- **Virtual Account:** No
- **Elevated Token:** Yes

• Impersonation Level: Impersonation

• New Logon:

- **Security ID:** S-1-5-18
- **Account Name:** SYSTEM
- **Account Domain:** NT AUTHORITY
- **Linked Logon ID:** 0x0

• Process Information:

- **Process ID:** 0x480
- **Process Name:** C:\Windows\System32\services.exe

• Authentication Details:

- **Logon Process:** Advapi
- **Authentication Package:** Negotiate

This log record indicates a successful login event for a system account (SYSTEM) via the Advapi logon process, which is commonly used for service logins. The elevated token (Yes) suggests that the account has administrative privileges.

2. PowerEvent Handled Successfully:

- **Message ID:** 290cea10-1f2e-46eb-9c73-57a271a98eba
- **Message:** “PowerEvent handled successfully by the service.”

• Details:

- **Timestamp:** 1717963266828
- **Hostname:** Dell-G15
- **Process ID:** 10624
- **Event Record ID:** 265286

This log indicates that a PowerEvent (likely a system power state change such as sleep, hibernate, or wake) was handled successfully by a service on the machine named Dell-G15.

3. Another Successful Account Login: This section has similar details to the first log record but indicates another successful login event.

General Structure of Logs:

- **Message:** Describes the event or action.
- **Timestamp:** When the event occurred.
- **Hostname:** The machine where the event was logged.
- **Process ID:** The ID of the process that logged the event.
- **Event Record ID:** Unique identifier for the event record.
- **Account and Logon Information:** Details about the accounts involved, including security IDs, logon IDs, and domains.

Usage of Fields:

- **Security ID (SID):** Unique identifier for the user or group.
- **Logon Type:** Indicates the kind of logon that was requested (e.g., interactive, network, service).
- **Process Information:** Helps identify which executable is responsible for the event.
- **Authentication Details:** Information about the authentication method and process.

Recommendations:

- **Security Monitoring:** Pay attention to logon types and elevated tokens to ensure that only authorized logins are taking place.
- **Event Tracking:** Regularly review `PowerEvent` logs to track system power state changes and ensure they are handled correctly.
- **System Health:** Use process and event record IDs to diagnose and troubleshoot any issues with system services.

VI. CONCLUSION

In this paper, we presented a comprehensive analysis of application performance using various profiling tools, including cProfile, PyInstrument, New Relic, and JMeter. Our findings demonstrate that these tools are invaluable for identifying performance bottlenecks and optimizing application efficiency.

The cProfile tool, with its detailed function call tracking, provided deep insights into the computational aspects of our applications, allowing us to pinpoint exact locations of inefficiency. However, its overhead can sometimes affect the application's runtime performance, making it less suitable for real-time profiling in production environments.

PyInstrument, on the other hand, offered a lower-overhead alternative with its statistical sampling approach. This method provided a clear visualization of performance hotspots without significantly impacting the application's performance, making it more suitable for live systems where minimal intrusion is necessary.

New Relic's cloud-based APM capabilities extended our analysis to include not only application-level insights but also server-side performance metrics. This holistic view enabled us to correlate application performance with underlying system behavior, leading to more targeted and effective optimization strategies.

JMeter added another dimension to our analysis by simulating various load conditions to assess application performance under stress. This allowed us to identify potential scalability issues and ensure that our application could handle high volumes of traffic without degradation in performance.

Through the integration and comparison of these tools, we illustrated their complementary strengths and the necessity of a multi-faceted approach to performance analysis. By leveraging these tools, developers can gain detailed insights, reduce resource consumption, and enhance the scalability and responsiveness of their applications.

Ultimately, the use of advanced performance analysis tools is crucial for developing high-performing, efficient, and scalable software applications. Our analysis underscores the importance of selecting the appropriate tool based on specific

performance requirements and operational contexts to achieve optimal results.

REFERENCES

- [1] R. K. Lenka, M. R. Dey, P. Bhanse, and R. K. Barik, "Performance and Load Testing: Tools and Challenges," *2018 International Conference on Recent Innovations in Electrical, Electronics & Communication Engineering (ICRIEECE)*, Bhubaneswar, India, 2018, pp. 2257-2261.
- [2] M. Weber, et al., "Detection and Visualization of Performance Variations to Guide Identification of Application Bottlenecks," *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, Philadelphia, PA, USA, 2016, pp. 289-298.
- [3] K. S. Arif and U. Ali, "Mobile Application testing tools and their challenges: A comparative study," *2019 2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*, Sukkur, Pakistan, 2019, pp. 1-6.
- [4] P. Tran and I. Gorton, "Analyzing the scalability of transactional CORBA applications," *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 38*, Zurich, Switzerland, 2001, pp. 102-110.
- [5] J. P. Sotomayor, S. C. Allala, D. Santiago, et al., "Comparison of open-source runtime testing tools for microservices," *Software Qual J*, vol. 31, pp. 55-87, 2023.
- [6] Python Software Foundation, "cProfile Documentation," <https://docs.python.org/3/library/profile.html#module-cProfile>
- [7] "PyInstrument Documentation," <https://pyinstrument.readthedocs.io/>
- [8] New Relic, "New Relic Documentation," <https://docs.newrelic.com/>
- [9] Apache JMeter, "JMeter Documentation," <https://jmeter.apache.org/usermanual/index.html>