

Development of a Linux Realtime Latency Monitoring, Storage, and Visualization System

J. SAIKUMAR¹, R. GURUNADHA²

¹M.Tech Student, ECE Department, JNTUGV-CEV. ²Associate Professor, ECE Department, JNTUGV-CEV.

Abstract

Real-time systems require predictable response behavior because excessive and unpredictable execution delays can affect the reliability of time-sensitive applications. This paper presents the development of a Linux-based real-time latency monitoring, storage, and visualization system using the Real-Time Linux Analysis (RTLTA) framework. The proposed system uses the RTLTA timerlat tracer to measure operating-system timer latency and obtain IRQ, thread, and user-level latency observations. A Python-based collector continuously reads the RTLTA output, extracts the required current latency values, and stores the measurements in an SQLite database. A Flask-based web backend retrieves the stored measurements and provides them to a web dashboard through an application programming interface. The dashboard presents the current IRQ, thread, and user latency values together with historical graphs and threshold-based status information. This architecture provides a continuous path from kernel-level latency measurement to persistent storage and web-based visualization, reducing the need for manual observation of terminal output. The experimental observations demonstrate that latency can vary significantly under system activity, highlighting the importance of monitoring both current and historical measurements rather than relying only on average values. A documented cyclictst experiment produced a minimum latency of 26 μ s, an average latency of 973 μ s, and a maximum latency of 16 ms. The proposed system provides a foundation for further studies involving machine-learning-based latency prediction, hardware performance-counter correlation, virtualization analysis, and automated alert notification.

Index Terms— Real-Time Linux, RTLTA, timerlat, latency monitoring, IRQ latency, thread latency, user latency, SQLite, Flask, Linux kernel, real-time systems, latency visualization.

I. INTRODUCTION

Real-time computing systems are required to respond to events within predictable timing constraints. In such systems, the correctness of an operation depends not only on the produced result but also on when the result is produced. Delays introduced by interrupt handling, task scheduling, kernel execution, CPU contention, and other operating-system activities can therefore affect the timing behavior of applications. Linux provides extensive scheduling and tracing facilities, while real-time configurations such as PREEMPT_RT are designed to make important kernel execution paths more preemptible and reduce sources of scheduling latency [6], [7], [13]. Nevertheless, measuring the actual latency experienced by a running system remains important because system behavior can vary with workload, CPU activity, configuration, and hardware conditions.

The Linux Real-Time Analysis (RTLTA) framework provides tools for analyzing real-time properties of Linux systems. RTLTA uses kernel tracing capabilities rather than treating Linux only as a black-box system, allowing latency behavior and possible causes of unexpected results to be investigated

[1]. One of its principal tools, rtda timerlat, measures timer latency at the IRQ and thread stages. The timerlat tracer uses per-CPU kernel threads with periodic timers and reports latency information associated with timer wakeups [2]. The top mode provides continuously updated summary information, while the hist mode presents a distribution of timerlat samples [2], [5].

Although command-line tools are effective for real-time analysis, continuously observing terminal output can make it difficult to maintain historical measurements and visually identify changes in latency. A monitoring system can address this limitation by connecting the measurement tool to a data-collection, storage, processing, and visualization pipeline. In the proposed system, RTLTA provides the measurement source, a Python-based collector.py process extracts the required latency fields, SQLite provides persistent local storage, and a Flask web application provides the interface between stored measurements and the dashboard [3], [4], [12].

The proposed dashboard focuses on three latency values used by the project: IRQ latency, thread latency, and user latency. The current values are displayed numerically, while historical observations are plotted to provide a time-based view of system behavior. A configurable alert threshold is also used to classify the current state. If any monitored latency exceeds the configured threshold, the dashboard reports a high-latency condition. This provides a simple and transparent mechanism for identifying latency excursions without changing the underlying RTLTA measurement process.

The main contribution of this work is therefore an integrated monitoring pipeline that connects operating-system-level latency measurement with persistent storage and web-based visualization. The system preserves the measured data so that current observations can be compared with earlier observations. It also establishes a software architecture that can be extended with automated notifications, hardware performance-counter measurements, virtualization experiments, and machine-learning-based latency prediction. These extensions are considered future work rather than components of the current implementation.

The remainder of this paper describes the technical background, proposed architecture, implementation methodology, experimental setup, results, limitations, and future extensions of the RTLTA-based monitoring system.

II. RELATED WORK

Real-time Linux analysis has been studied through kernel scheduling, tracing, latency measurement, and real-time kernel technologies. The Linux scheduler provides several scheduling classes and mechanisms for managing CPU execution, including real-time scheduling facilities and deadline scheduling [6]. Scheduling behavior is directly related to latency because a runnable task may have to wait for the CPU while another execution context is active. Linux also provides specialized real-time scheduling mechanisms, while PREEMPT_RT modifies important kernel mechanisms to make execution more preemptible and uses threaded interrupt handling to reduce sections in which preemption is restricted [7], [13].

RTLA extends the Linux tracing infrastructure with tools specifically intended to analyze real-time behavior. The RTLA framework includes osnoise, timerlat, and hardware-noise analysis tools [1]. In particular, timerlat measures timer latency at the interrupt and thread stages and can provide both summary and histogram representations [2]. The timerlat tracer is intended to help identify sources of wakeup latency in real-time threads and uses periodic timers to calculate the difference between expected and actual wakeup times [2], [5]. These capabilities make RTLA an appropriate measurement source for the proposed monitoring system.

Cyclicttest is another established tool for measuring real-time system latency. It repeatedly compares a thread's intended wakeup time with its actual wakeup time and reports latency statistics [5]. The project uses cyclicttest as a complementary reference measurement rather than replacing RTLA. The distinction is important: cyclicttest provides latency statistics for repeated wakeups, whereas the proposed system uses RTLA timerlat as the primary source and additionally preserves the IRQ and thread-related information exposed by timerlat. The documented project experiment produced a minimum latency of 26 μ s, an average of 973 μ s, and a maximum of 16 ms. These values illustrate why minimum, average, and maximum observations can provide different views of system timing behavior.

Kernel tracing is another important area of related work. Linux tracing facilities such as ftrace allow kernel execution behavior to be examined at a lower level [8]. RTLA itself builds on Linux tracing capabilities, making tracing an important part of the broader methodology for understanding operating-system latency [1]. The proposed dashboard does not attempt to replace detailed tracing. Instead, it provides a higher-level monitoring layer in which selected latency measurements can be collected, stored, and visualized continuously.

Persistent storage is also important for latency monitoring because terminal output alone does not provide a convenient historical record. SQLite is a lightweight embedded relational database that can be used without a separate database server [3], [12]. In the proposed architecture, the database stores timestamped latency observations produced by the collector. This enables the Flask backend to retrieve recent measurements and allows the dashboard to plot historical values.

Flask provides the web application layer used to expose the stored measurements to the dashboard [4]. The proposed implementation uses Flask to query SQLite and return structured latency data through an API. The browser then

uses these values to update the current-value indicators, status information, and historical charts. This separation between measurement, storage, backend processing, and visualization allows each stage to be tested independently.

Previous work and documentation also provide a basis for the future extensions considered in this project. Linux performance-counter facilities can expose hardware and software performance events, making it possible to investigate relationships between latency and events such as CPU activity and cache behavior [9], [10], [14]. Real-time virtualization research further motivates evaluation of latency under virtualized configurations [11]. Worst-case execution-time research emphasizes the importance of understanding timing variability rather than relying exclusively on average execution behavior [15].

The proposed work differs from these individual tools and studies by integrating measurement and visualization into a single monitoring pipeline. RTLA remains responsible for latency measurement, while the Python collector, SQLite database, Flask backend, and web dashboard provide continuous data handling and presentation. Thus, the system is intended as a practical monitoring layer around existing Linux real-time analysis facilities rather than as a replacement for the underlying kernel tracing mechanisms.

III. SYSTEM ARCHITECTURE

The proposed RTLA Real-Time Latency Monitoring System consists of multiple software layers that work together to measure, collect, store, process, and visualize Linux real-time latency. The architecture is designed so that the latency measurement mechanism remains independent from the database, web backend, and visualization components. The overall data flow is:

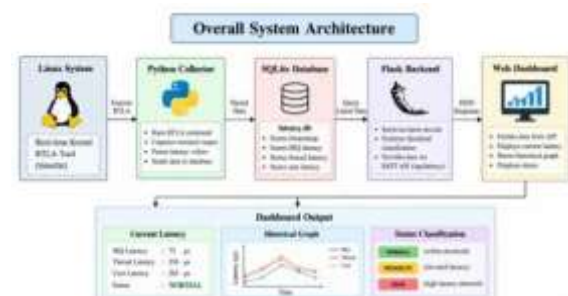


Fig. 1. Overall architecture of the proposed RTLA real-time latency monitoring system

A. Linux System and RTLA Timerlat

The Linux operating system is the measurement environment in the proposed system. RTLA is used to analyze real-time behavior, with rtla timerlat serving as the primary latency measurement mechanism. Timerlat measures the delay associated with timer wakeups and reports latency information for the IRQ and thread stages [1], [2].

The RTLA timerlat output contains several statistical fields, including current (cur), minimum (min), average (avg), and maximum (max) latency. The proposed collector is configured to extract the required current latency values so that the dashboard represents the latest observed measurement rather than unintentionally displaying an average or maximum value.

The project monitors three values:

- **IRQ Latency:** latency observed at the interrupt-handling stage.

- **Thread Latency:** latency associated with the timerlat thread stage.
- **User Latency:** the user-level latency value collected by the project.

These measurements form the input to the subsequent data-processing pipeline.

B. Python Data Collector

The collector.py program acts as the bridge between RTLA

and the database. It continuously reads the output produced by the timerlat measurement process and parses the relevant fields.

The collector identifies the appropriate latency values from the RTLA output and converts them into structured records. Each record contains the measurement time and the corresponding IRQ, thread, and user latency values. The extracted data is then inserted into the SQLite database.

This component is important because raw terminal output is designed primarily for human observation, whereas the web dashboard requires structured data. The collector therefore converts the continuously generated RTLA measurements into a format that can be stored and processed programmatically.

C. SQLite Database

SQLite is used as the local persistent storage layer. It provides a lightweight relational database without requiring a separate database server [3], [12].

The latency records are stored with fields such as: timestamp, CPU, IRQ latency, Thread latency, and User latency.

The database provides two main advantages. First, it preserves measurements after they have disappeared from the terminal display. Second, it provides historical data that can later be queried by the Flask backend for graph generation and analysis. The database therefore acts as the central storage point between the measurement process and the web application.

D. Flask Backend

The Flask application provides the server-side layer between SQLite and the web interface. Flask is used to create the web application and expose the stored measurements through an API [4].

The backend receives a request from the dashboard and executes an SQL query on the latency database. The selected records are converted into structured response data containing the timestamp and latency fields. The API can therefore provide the latest measurements as well as a set of historical observations required for graph visualization.

This separation prevents the dashboard from directly accessing the database. Instead, the Flask application controls how database information is retrieved and presented to the client.

E. Web Dashboard

The web dashboard is the presentation layer of the proposed architecture. It receives latency information from the Flask API and presents it in an easily understandable form. The dashboard displays:

1. Current IRQ Latency
2. Current Thread Latency
3. Current User Latency
4. Historical latency graphs
5. Threshold-based status

The graphs provide a time-based representation of the stored measurements. This makes it possible to observe whether latency remains relatively stable or experiences an increase during system activity.

F. Threshold and Alert Classification

A configurable threshold is used to classify the observed latency. Let T represent the configured threshold and L_{IRQ} , L_{Thread} , and L_{User}

represent the three current latency values.

The system applies the following rule:

$Status = HIGH\ LATENCY, \text{ if } \max(L_{IRQ}, L_{Thread}, L_{User}) > T;$
 $NORMAL, \text{ otherwise}$

Therefore, the system reports HIGH LATENCY when at least one of the monitored latency components exceeds the configured threshold. If all three values remain at or below the threshold, the dashboard reports NORMAL. The threshold is an application-level classification boundary. It is not a universal Linux real-time limit and should be selected according to the requirements of the monitored application.

G. End-to-End Data Flow

The complete operation of the system can be summarized as follows:

- **Step 1:** The Linux system executes the RTLA timerlat measurement.
- **Step 2:** RTLA generates timer-latency measurements for the monitored CPU or CPUs.
- **Step 3:** collector.py reads the RTLA output and extracts the required current IRQ, thread, and user latency values.
- **Step 4:** The collector stores the measurements in the SQLite latency.db database.
- **Step 5:** The Flask backend queries the database through the /api/latency endpoint.
- **Step 6:** The web dashboard receives the structured data from Flask.
- **Step 7:** The dashboard displays the latest latency values, historical graphs, and threshold-based status.

This architecture establishes a traceable path from the original kernel-level measurement to the value displayed on the web page. Consequently, a displayed latency value can be traced backward through the Flask API and SQLite record to the measurement collected from RTLA.

IV. METHODOLOGY

The proposed system follows a layered methodology in which real-time latency is first measured using RTLA and then collected, stored, processed, and visualized through a web-based monitoring system. The complete methodology consists of latency measurement, data collection, database storage, backend processing, threshold classification, and dashboard visualization.

A. Real-Time Latency Measurement

The first stage uses the Linux Real-Time Linux Analysis (RTLA) framework. The rtla timerlat tool is used as the primary source of latency measurements. Timerlat measures the delay associated with timer wakeups and provides latency information at the IRQ and thread stages. The RTLA output contains several statistical values such as current (cur), minimum (min), average (avg), and maximum (max). For the proposed dashboard, the current (cur) latency values are selected for continuous monitoring. The monitored values are IRQ latency, thread latency, and user latency.

For example, an RTLA observation can provide values such as $L_{IRQ} = 75\ \mu s$, $L_{Thread} = 256\ \mu s$, $L_{User} = 265\ \mu s$. These

values are subsequently passed to the data-collection stage.

B. Latency Data Collection

The second stage is implemented using the Python program collector.py. The collector reads the continuously generated RTLA timerlat output and identifies the required latency fields. The collector performs the following operations:

1. Starts or reads the RTLA timerlat measurement stream.
2. Captures the generated terminal output.
3. Identifies the relevant latency line.
4. Extracts the current IRQ, thread, and user values.
5. Associates the measurement with a timestamp.
6. Stores the structured measurement in SQLite.

The collector acts as the interface between the RTLA measurement tool and the database. This avoids requiring the dashboard to directly interpret raw terminal output.

C. Database Storage

After extraction, the measurements are stored in an SQLite database named latency.db. SQLite is used because it provides lightweight local relational storage without requiring a separate database server. A typical record contains:

Field	Description
Timestamp	Time at which the measurement was recorded
CPU	CPU associated with the observation
IRQ Latency	Current IRQ latency in μ s
Thread Latency	Current thread latency in μ s
User Latency	Current user latency in μ s

The database provides persistent storage for the measurements. Therefore, values that have already disappeared from the terminal output can still be retrieved later for graphing and analysis.

D. Flask Backend Processing

The Flask application provides the backend interface between the SQLite database and the web dashboard. When the dashboard requests latency information, Flask executes an SQL query on latency.db and retrieves the required records. The backend then converts the database records into structured data and returns them through the /api/latency endpoint. The basic data flow is:

SQLite Database $\rightarrow$ SQL Query $\rightarrow$ Flask API $\rightarrow$ JSON Response $\rightarrow$ Web Dashboard

This separation allows the database and user interface to remain independent. The dashboard does not directly access the SQLite database.

E. Threshold-Based Status Classification

The methodology includes a configurable threshold for identifying elevated latency. Let the configured threshold be

T , and let the current measurements be L_{IRQ} , L_{Thread} , and L_{User} .
 $Status = HIGH\ LATENCY$, if $\max(L_{IRQ}, L_{Thread}, L_{User}) > T$;
 $NORMAL$, otherwise

Thus, the system checks all three latency values. If even one value exceeds the configured threshold, the dashboard reports a high-latency condition. For example, with $T = 50\ \mu$ s and $L_{IRQ} = 26\ \mu$ s, $L_{Thread} = 45\ \mu$ s, $L_{User} = 75\ \mu$ s, the status becomes HIGH LATENCY because the user latency of $75\ \mu$ s exceeds the $50\ \mu$ s threshold. The threshold is an application-level classification limit, not a universal Linux

real-time limit. Its value should therefore be selected according to the timing requirements of the application being monitored.

F. Web-Based Visualization

The final stage is the web dashboard. The dashboard receives the JSON data from the Flask API and presents the measurements graphically. The interface contains three main types of information:

1. **Current latency values** — displays the latest IRQ, thread, and user latency.
2. **Historical graph** — plots previously collected measurements against time.
3. **Status classification** — indicates whether the measured latency is within or above the configured threshold.

The historical graph is particularly useful because a single current value cannot show how latency has changed over time. By storing successive measurements in SQLite and retrieving them through Flask, the dashboard can display the evolution of latency.

V. EXPERIMENTAL SETUP

The proposed RTLA-based latency monitoring system was implemented and evaluated in a Linux environment to verify the measurement, collection, storage, and visualization of real-time latency. The experimental setup consists of the RTLA timerlat tool, a Python-based data collector, an SQLite database, a Flask web server, and a browser-based dashboard. The overall experimental workflow follows the sequence RTLA measurement $\rightarrow$ data collection $\rightarrow$ database storage $\rightarrow$ backend processing $\rightarrow$ web visualization.

The primary latency measurements were obtained using rtda timerlat. The tool was used to observe timer latency associated with the IRQ and thread stages and to provide the latency information required by the monitoring application [1], [2]. The RTLA output provides current, minimum, average, and maximum latency values. In the proposed implementation, the current (cur) values were extracted by collector.py and used as the latest latency observations for the dashboard.

The experimental system used a Linux/Ubuntu environment with two online CPUs. The Python collector continuously processed the RTLA output and stored the extracted measurements in the SQLite database latency.db. Each database record contains the measurement timestamp and the corresponding IRQ, thread, and user latency values. SQLite was selected as the storage mechanism because it provides lightweight local relational storage without requiring a separate database server [3], [12].

The Flask application was used as the backend interface between the database and the web dashboard. The backend retrieves the stored latency records through an SQL query and provides the measurements through the /api/latency interface. The dashboard receives this data and displays the current IRQ, thread, and user latency values together with historical latency plots. This arrangement allows the values displayed in the dashboard to be traced back to the

corresponding database records and, ultimately, to the RTLA measurements.

A configurable threshold was used to classify the observed latency condition. Let T denote the configured threshold and

L_{IRQ} , L_{Thread} , and L_{User} denote the measured latency values.

The monitoring system classifies the condition as HIGH LATENCY when at least one of the three measurements

exceeds the threshold:

$Status = HIGH\ LATENCY, if\ max(L_{IRQ}, L_{Thread}, L_{User}) > T;$
NORMAL, otherwise.

The threshold is used as an application-level monitoring boundary and is not considered a universal real-time limit. Therefore, the selected threshold value is reported together with the experimental conditions.

For complementary latency evaluation, `cyclicttest` was also used. The documented `cyclicttest` experiment was performed using the command:

```
sudo cyclicttest -p 99 -m -I 1000 -l 100000
```

The recorded measurements were a minimum latency of 26 μs , an average latency of 973 μs , and a maximum latency of 16 ms. These results demonstrate the difference between typical and worst observed latency during the measurement interval. The maximum value was substantially higher than the average, indicating that occasional latency excursions can be hidden when only average latency is considered.

The experimental setup also supports evaluation under different system conditions. Measurements can be collected under an idle system and under additional CPU workload, such as executing `yes > /dev/null`. The system can also be evaluated with different virtual CPU configurations when running inside a virtual machine. For each condition, the measured latency values can be recorded in the database and compared using the historical dashboard plots.

The evaluation focuses on verifying four aspects of the proposed system: measurement correctness, data consistency, visualization, and threshold classification. Measurement correctness is verified by comparing the values extracted by `collector.py` with the corresponding RTLA output. Data consistency is verified by comparing the collector records with the SQLite database and Flask API response. Visualization is verified by ensuring that the current values and historical graphs correspond to the stored measurements. Finally, threshold classification is verified by determining whether the displayed status changes when one or more latency components exceed the configured threshold.

TABLE III. EXPERIMENTAL SETUP

Parameter	Configuration
Operating system	Linux/Ubuntu
CPU configuration	2 online CPUs
Primary measurement tool	RTLA timerlat
Data collector	Python collector.py
Database	SQLite latency.db
Web framework	Flask
Visualization	Web dashboard / Chart.js
Monitored parameters	IRQ, Thread, User latency
Status mechanism	Configurable threshold
Complementary tool	cyclicttest
Cyclicttest minimum	26 μs
Cyclicttest average	973 μs
Cyclicttest maximum	16 ms

VI. RESULTS AND DISCUSSION

The proposed Linux real-time latency monitoring system was evaluated to verify the complete measurement and visualization process using RTLA, Python-based data collection, SQLite storage, and a Flask-based web interface. The experimental results include basic latency measurements using `cyclicttest`, observation of the RTLA `timerlat` interfaces, and evaluation of the proposed monitoring dashboard under normal- and high-latency conditions. The results demonstrate the ability of the developed system to acquire latency measurements, retain them in a local database, and present the observed latency condition through a web-based interface.

A. Basic Latency Measurement

As an initial latency evaluation, `cyclicttest` was executed using the following configuration: `sudo cyclicttest -p 99 -m -I 1000 -l 100000`

```
cal@kali:~$ sudo cyclicttest -p 99 -l 1000 -l 100000
# /dev/cpu_dma_latency set to 0us
policy: fifo; loadavg: 3.15 3.18 2.53 2/879 6829
T: 0 ( 6828) P:99 I:1000 C: 4728 Min: 26 Act: 155 Avg: 973 Max: 16011
```

The measured minimum latency was **26 μs** , the average latency was **973 μs** , and the maximum latency was **16 ms**. These measurements provide a baseline for observing the timing behavior of the Linux system under the selected experimental configuration. The difference between the average and maximum values indicates that the system experienced latency excursions that were considerably larger than the average observation.

TABLE IV. BASIC LATENCY MEASUREMENT

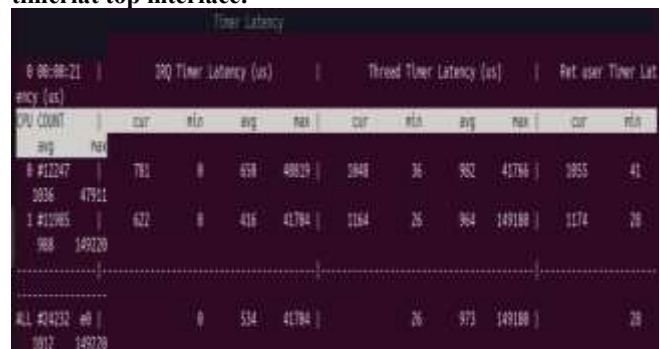
Measurement Parameter	Observed Value
Minimum latency	26 μs
Average latency	973 μs
Maximum latency	16 ms

The `cyclicttest` measurement is used as a baseline measurement in this work. The proposed system subsequently extends the monitoring process by obtaining latency information through RTLA and presenting the measurements through an integrated dashboard.

B. RTLA Timerlat Observation

The RTLA `timerlat` interfaces were used to observe the timing behavior of the Linux system. The project results document both the `timerlat top` and `timerlat hist` interfaces. The `timerlat top` interface provides an observation of the timer latency behavior, whereas the histogram interface provides a distribution-oriented representation of the measured latency.

timerlat top interface:



This architecture separates measurement, storage, processing, and visualization functions. Such separation allows the latency measurements obtained from the operating system to be retained and subsequently analyzed through the web interface.

The project results include a NORMAL LATENCY STATUS ALERT obtained from the implemented monitoring system. The corresponding experimental sequence includes the execution of collector.py, storage in the SQLite database, execution of app.py, and presentation of the result through the dashboard.

```

root@ip-10-0-1-204:~# python3 collector.py
Starting RTLA timerlat...
Waiting for latency data...

Saved -> CPU: 0 | IRQ: 468 us | Thread: 536 us | User: 540 us
Saved -> CPU: 1 | IRQ: 218 us | Thread: 342 us | User: 346 us
Saved -> CPU: 0 | IRQ: 56 us | Thread: 131 us | User: 139 us
Saved -> CPU: 1 | IRQ: 17 us | Thread: 106 us | User: 115 us
Saved -> CPU: 0 | IRQ: 90 us | Thread: 186 us | User: 190 us
Saved -> CPU: 1 | IRQ: 276 us | Thread: 391 us | User: 401 us
Saved -> CPU: 0 | IRQ: 192 us | Thread: 278 us | User: 287 us
Saved -> CPU: 1 | IRQ: 75 us | Thread: 256 us | User: 265 us
^C
RTLA collector stopped.

```

[illegible]

```

root@kali:~/Documents# docker run -it --rm --name flask-app python:3.8-slim /bin/bash
# Serving Flask app "app"
# Debug mode: on
WARNING: This is a development server - do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://10.0.2.15:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 496-252-338

127.0.0.1 - - [20/Sep/2020 20:00:00] "GET /api/latency HTTP/1.1" 200 -
127.0.0.1 - - [20/Sep/2020 20:00:00] "GET /api/latency HTTP/1.1" 200 -
127.0.0.1 - - [20/Sep/2020 20:00:00] "GET /api/latency HTTP/1.1" 200 -
127.0.0.1 - - [20/Sep/2020 20:00:01] "GET /api/latency HTTP/1.1" 200 -
127.0.0.1 - - [20/Sep/2020 20:00:02] "GET /api/latency HTTP/1.1" 200 -
127.0.0.1 - - [20/Sep/2020 20:00:04] "GET /api/latency HTTP/1.1" 200 -

```

The normal-status result demonstrates that the developed application can process the collected latency information and provide a corresponding system-status indication through the web interface. Rather than requiring continuous observation of the RTLA terminal output, the user can obtain the monitoring result through the dashboard.



This result also verifies the functional relationship between the backend components. The collector supplies the

measurement data, SQLite provides persistent storage, and the Flask application retrieves the stored information for presentation. Therefore, the normal-status result represents not only a dashboard output but also the successful operation of the complete data-processing chain.

E. High-Latency Monitoring Result

A separate experimental result documented in the project shows a HIGH LATENCY STATUS ALERT. The corresponding workflow again includes collector.py, the SQLite database, app.py, and the dashboard.

Collector.py

```
cal@kali:~/RTLA$ python3 collector.py
Starting RTLA timerlat...
Waiting for latency data...

Saved -> CPU: 0 | IRQ: 184 us | Thread: 248 us | User: 252 us
Saved -> CPU: 1 | IRQ: 52 us | Thread: 158 us | User: 161 us
Saved -> CPU: 0 | IRQ: 678 us | Thread: 728 us | User: 734 us
Saved -> CPU: 1 | IRQ: 209 us | Thread: 335 us | User: 339 us
Saved -> CPU: 0 | IRQ: 114 us | Thread: 149 us | User: 152 us
Saved -> CPU: 1 | IRQ: 79 us | Thread: 136 us | User: 148 us
Saved -> CPU: 0 | IRQ: 21 us | Thread: 78 us | User: 82 us
Saved -> CPU: 1 | IRQ: 476 us | Thread: 559 us | User: 554 us
^C
RTLA collector stopped.
```

The high-latency result demonstrates the monitoring system's ability to distinguish an abnormal latency condition from a normal condition according to the configured threshold. This provides a simple mechanism for identifying latency excursions during system operation.

HIGH LATENCY



The comparison between the normal- and high-latency dashboard results demonstrates that the proposed system is capable of converting raw latency measurements into an immediately interpretable monitoring status. The dashboard therefore serves as the final visualization layer of the measurement pipeline.

F. Discussion

The experimental results demonstrate that the proposed system provides an integrated approach for Linux latency monitoring. The cyclictst experiment establishes the presence of measurable timing variation, with the recorded values ranging from a minimum of 26 μ s to a maximum of 16 ms. The RTLA interfaces provide additional visibility into timer latency behavior through both top-level and histogram-based representations. These measurements are subsequently incorporated into the developed software pipeline consisting of collector.py, SQLite, app.py, and the web dashboard.

The normal- and high-latency results further demonstrate the practical operation of the proposed monitoring mechanism. Under the normal condition, the dashboard reports a normal latency status, whereas the high-latency experiment produces a corresponding high-latency status.

An important feature of the proposed system is that it does not depend solely on the RTLA terminal interface. Instead, the measurements are collected and retained in a database before being presented through the web application. This provides a structured path for converting low-level Linux latency measurements into information that can be monitored through a graphical interface.

The results therefore support the primary objective of the work: to develop a Linux-based system capable of measuring, collecting, storing, and visually monitoring real-time latency information. The current implementation focuses on measurement and visualization. Advanced functions such as machine-learning-based prediction, hardware performance-counter correlation, and automated external notification can be considered extensions of the developed framework.

VII. LIMITATIONS

The proposed system provides an integrated framework for Linux real-time latency monitoring, data storage, and web-based visualization; however, several limitations remain. First, the experimental evaluation is based on the Linux environment and workload conditions used during the project, and therefore the obtained latency values cannot be generalized to all hardware platforms, kernel configurations, or workloads. The measured latency can vary with processor configuration, scheduling behavior, system load, and other operating-system conditions.

Second, the current system primarily focuses on collecting and visualizing the latency measurements generated by RTLA. Although the dashboard identifies a high-latency condition using a configurable threshold, it does not automatically determine the underlying cause of each latency excursion. Detailed root-cause analysis would require additional kernel tracing and system-level information.

Third, the threshold used for status classification is an application-defined monitoring parameter. It provides a practical criterion for distinguishing normal and high-latency observations within the developed application, but it should not be interpreted as a universal real-time performance limit.

Finally, the current implementation does not include machine-learning-based prediction or hardware performance-counter correlation. Consequently, the system currently reports and analyzes measured latency rather than predicting future latency or correlating latency excursions with processor-level events.

VIII. FUTURE WORK

The proposed system provides a foundation for several extensions. One important direction is the integration of a machine-learning-based latency prediction mechanism. Historical IRQ, thread, and user latency measurements stored in the SQLite database can be used as input data for developing a prediction model. The trained model could estimate future latency values and display predicted and actual measurements together on the dashboard. Different prediction algorithms and evaluation metrics can subsequently be investigated under different workloads and system configurations.

Another extension is the integration of hardware performance counters. Hardware-level information such as CPU cycles, cache misses, branch events, and memory-related events can be collected and correlated with RTLA latency measurements. Such correlation could help investigate whether particular hardware-level activities occur together with increased latency.

The system can also be extended for virtualized and workload-based experiments. Measurements can be collected under different virtual CPU configurations and controlled CPU workloads to study the effect of virtualization and system activity on real-time latency.

A further extension is automated alert notification. When IRQ, thread, or user latency exceeds the configured threshold, the system could generate an external notification containing the measured values and the component that exceeded the threshold. A cooldown mechanism could be incorporated to prevent repeated notifications for the same continuing latency condition.

These extensions would transform the current monitoring-oriented framework into a more comprehensive platform for latency prediction, hardware-level correlation, workload analysis, and automated real-time monitoring.

IX. CONCLUSION

This paper presented the development of a Linux-based real-time latency monitoring, storage, and visualization system using RTLA timerlat. The proposed architecture integrates RTLA-based latency measurement with a Python data collector, SQLite database, Flask backend, and web-based dashboard. This provides a continuous data path from operating-system-level latency measurement to persistent storage and graphical visualization.

The experimental results demonstrate the practical operation of the developed system. The documented cyclictst experiment produced a minimum latency of 26 μ s, an average latency of 973 μ s, and a maximum latency of 16 ms. The project also demonstrated the use of the RTLA timerlat top and timerlat hist interfaces and the subsequent processing of latency information through collector.py, SQLite, app.py, and the dashboard.

The normal- and high-latency experiments further demonstrated the ability of the system to classify and visualize different latency conditions through the web dashboard. The documented results show the complete processing sequence involving the collector, database, Python application, and dashboard for both conditions.

The developed system therefore provides a practical foundation for continuous Linux latency observation without depending solely on manual inspection of command-line output. The current implementation focuses on measurement, collection, storage, threshold-based classification, and visualization, while machine-learning prediction, hardware performance-counter integration, virtualization analysis, and automated notification remain potential areas for future development.

REFERENCES

- [1] The Linux Kernel Documentation, “rtla—Real-time Linux Analysis tool,” Linux Kernel Documentation, accessed Sep. 2026.
- [2] The Linux Kernel Documentation, “rtla-timerlat—Measures the operating system timer latency,” Linux Kernel Documentation, accessed Sep. 2026.
- [3] Python Software Foundation, “sqlite3—DB-API 2.0 interface for SQLite databases,” Python Documentation, accessed Sep. 2026.
- [4] Pallets Projects, “Quickstart—Flask Documentation,” Flask Documentation, accessed Sep. 2026.
- [5] Linux Foundation, “Cyclictst,” Real-Time Linux Documentation, accessed Sep. 2026.
- [6] The Linux Kernel Documentation, “Scheduler,” Linux Kernel Documentation, accessed Sep. 2026.
- [7] The Linux Kernel Documentation, “Real-time preemption,” Linux Kernel Documentation, accessed Sep. 2026.
- [8] S. Rostedt, “ftrace—Function Tracer,” The Linux Kernel Documentation, Linux Kernel Documentation, accessed Sep. 2026.
- [9] The Linux Kernel Documentation, “Performance Counters for Linux,” Linux Kernel Source Documentation, accessed Sep. 2026.
- [10] J. M. Domingos, P. Tomas, and L. Sousa, “Supporting RISC-V performance counters through performance analysis tools for Linux (Perf),” arXiv preprint arXiv:2112.11767, 2021.
- [11] “Real-Time Performance Analysis and Tuning of Embedded System Virtualization Architecture Based on KVM,” International Journal of Embedded and Real-Time Communication Systems, vol. 13, no. 1, 2022, doi: 10.4018/IJERTCS.302113.
- [12] K. P. Gaffney, M. Prammer, L. Brasfield, D. R. Hipp, D. Kennedy, and J. M. Patel, “SQLite: Past, Present, and Future,” Proc. VLDB Endowment, vol. 15, no. 12, pp. 3535–3547, 2022, doi: 10.14778/3554821.3554842.
- [13] F. Reghenzani, G. Massari, and W. Fornaciari, “The Real-Time Linux Kernel: A Survey on PREEMPT_RT,” ACM Computing Surveys, vol. 52, no. 1, Art. no. 18, 2019, doi: 10.1145/3297714.
- [14] J. Kukunas, “Perf,” in Power and Performance: Software Analysis and Optimization. Morgan Kaufmann, 2015, pp. 137–165, doi: 10.1016/B978-0-12-800726-6.00008-2.
- [15] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, “The Mälardalen WCET Benchmarks: Past, Present and Future,” in Proc. 10th Int. Workshop on Worst-Case Execution Time Analysis (WCET 2010), OASlcs, vol. 15, pp. 136–146, 2010, doi: 10.4230/OASlcs.WCET.2010.136.