SentryRT-1: A Case Study in Evaluating Real-Time Linux for Safety-Critical Robotic Perception

Yuwen Shen*†‡, Jorrit Vander Mynsbrugge*‡, Nima Roshandel*†‡, Robin Bouchez*‡, Hamed FirouziPouyaei*‡, Constantin Scholz*‡, Hoang-Long Cao*§, Bram Vanderborght*‡, Wouter Joosen†, Antonio Paolillo*

*Vrije Universiteit Brussel, Belgium, †KU Leuven, Belgium
‡imec, Belgium, §Can Tho University, Vietnam

***Abstract*—Ensuring timely and safe operation in robotics remains a challenge, especially in systems combining high-throughput perception with real-time safety constraints. SentryRT-1 is a minimal C++/CUDA runtime that integrates multi-camera sensing, GPU-accelerated human detection, and a safety module enforcing Speed and Separation Monitoring (SSM) robot control. In this case study, we model the runtime as a real-time task set and evaluate its behavior under various Linux kernel configurations. Using synthetic interference and replayable camera inputs, we benchmark the latency and determinism of the safety module. Our results show that real-time scheduling policies such as SCHED_DEADLINE significantly reduce both average and worst-case reaction times, and that a real-time kernel with PREEMPT_RT provides further—though less pronounced—improvements. These findings demonstrate the capabilities of Linux-based configurations for safety-critical robotic workloads.**

***Index Terms*—Real-Time Systems, Collaborative Robotics, Safety-Critical, Human-Robot Interaction, Real-Time Linux, Perception Pipeline, Speed and Separation Monitoring**

I. Introduction

Driven by global labor shortages and the accelerating automation trend, robotics is rapidly expanding across industries, particularly in manufacturing [1], [2]. Historically, most robotic deployments occurred in greenfield installations—factories built for automation, typically with fenced-off robotic cells where robots operate at high speed and isolated from humans to ensure safety and throughput [3]. However, many production tasks still rely on manual labor—so-called *brownfield environments*—where automation is introduced incrementally and workspace is often limited [4]. Examples include pharmaceutical packaging, machine tending, and kitting. In these cases, traditional caged robots are impractical due to spatial constraints and the need for continuous human cooperation.

Collaborative robots (cobots) equipped with Power and Force-Limiting (PFL) capabilities can safely operate alongside humans without physical barriers [5]. However, PFL relies on contact-based stopping, which may still cause injuries and imposes strict speed limitations. To overcome these constraints, robots are increasingly equipped with on-robot perception sensors to enable safety through Speed and Separation Monitoring (SSM) [6], standardized by ISO10218 [5], allowing dynamic speed adjustment based on human proximity.

Corresponding author: antonio.paolillo@vub.be

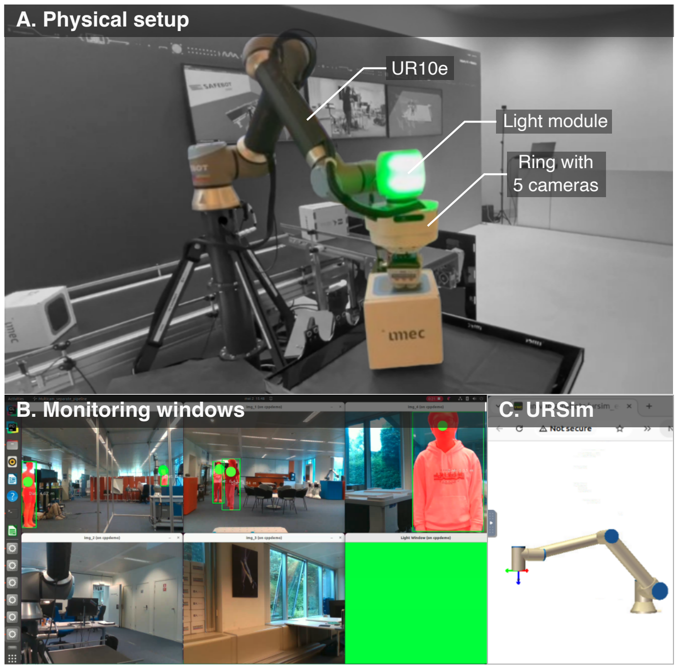

Fig. 1. Overview of the SentryRT-1 platform. (A) The physical setup including a UR10e robotic arm, a ring of five cameras, and a light signaling module. (B) Monitoring windows displaying 5 replay data streams and the virtual light module. (C) URSim, the robot simulation environment.

Achieving such responsiveness requires low-latency, predictable execution across sensing, inference, and actuation pipelines. Linux-based systems—widely used in modern robotic stacks—are attractive due to their ecosystem and hardware support, but it remains unclear how well they meet real-time constraints in these safety-critical scenarios. In particular, there is limited understanding of how different Linux configurations affect end-to-end timing in perception-driven safety loops. This paper addresses this gap through an application-driven analysis of real-time performance under representative a robotic workload.

Our prototype system, illustrated in Figure 1A, consists of a Universal Robots UR10e manipulator, five Intel RealSense D435i depth cameras mounted around the end-effector, an industrial x86 PC with a discrete NVIDIA GPU, and a light signaling module [7] controlled via the robot's digital I/O.

The hardware is able to operate both in live mode – using the setup of Figure 1A – and in simulation – using URSim [8] (Figure 1C) and replayable camera streams. To drive this setup, we develop **SentryRT-1**, a C++/CUDA software runtime that integrates multi-camera perception with GPU-accelerated human detection and a safety module enforcing proximity-based speed control. The system avoids middleware such as ROS, providing direct control over threading, memory, and data flow. A screenshot of the live perception and safety feedback is shown in Figure 1B.

In this paper, we model this runtime as a real-time task set and evaluate it under different Linux configurations—including `PREEMPT_RT` and `SCHED_DEADLINE`—to assess their ability to meet real-time constraints under synthetic interference that models high computational demand. Our benchmarking framework supports replayable inputs and repeatable stress conditions, allowing systematic evaluation of safety responsiveness under load. Our results show that `SCHED_DEADLINE` significantly improves reaction time compared to baseline Linux, while the real-time kernel with `PREEMPT_RT` provides further improvements.

II. Related Work

Significant research has been conducted within the real-time computing and robotics communities to design and evaluate the performance of robotic software frameworks.

ROS. The most widely adopted framework is ROS2 [9], which promotes modularity in robotic system design and uses Data Distribution Service (DDS)-based inter-process communication to transfer data between components. ROS2 adopts an event-driven callback mechanism, making timing analysis based on processing chains of components a natural approach [10]–[12]. Tang et al. [10] analyze response times across processing chains, modeling and improving the behavior of ROS2 executors to optimize task scheduling and system-level response time. Teper et al. [11] evaluate timing performance using two metrics—*maximum reaction time* and *maximum data age*—based on cause-effect chain analysis in ROS2-based autonomous robotic systems. However, prior work [13]–[16] has shown that the DDS-based IPC in ROS2 can introduce millisecond-level delays in data transmission between nodes. In contrast, shared-memory communication in pure C++ implementations incurs significantly lower transmission delays. Moreover, bypassing the ROS2 framework substantially reduces software footprint and debugging complexity. Developers gain finer control over execution paths and avoid the performance overheads and abstraction layers introduced by middleware [17], [18]. Teper et al. [19] show that the ROS2 multithreaded executor is prone to starvation, leading to unbounded response times. These limitations motivate our design independent of the ROS2 framework.

Benchmarking. Bakhshalipour et al. present `RoWild` [20], a comprehensive cross-platform performance benchmark for various mobile robotic systems. It reports end-to-end execution times and identifies algorithmic bottlenecks. `RobotPerf` [21] introduces a benchmarking framework tailored to robotic workloads implemented as ROS2 computational graphs. It supports both black-box and grey-box testing methodologies to evaluate real-time performance across diverse hardware platforms. However, the framework is tightly coupled with the ROS2 ecosystem, limiting its use in non-ROS2 systems. Both studies do not consider the impact of scheduling policies or kernel configurations, nor do they provide a timing model.

Soft real-time scheduling. To address this gap, Sifat et al. [12] propose a safety-performance metric that explicitly incorporates timing considerations in real-time robotic systems. Their approach uses heterogeneous processing units (e.g., CPU and GPU) modeled through a stochastic heterogeneous parallel DAG (SHP-DAG). They evaluate their method using both FIFO and CFS schedulers. However, these schedulers are not designed to meet the hard real-time requirements of safety-critical robotic systems. In contrast, our work explores the use of the real-time scheduling policies and preemptive kernels, which is more suitable for ensuring real-time guarantees.

Evaluate real-time constraints and kernels. Tools such as `cyclictest` [22] and `Timerlat` [23] have been developed to measure execution latency and trace its root causes. The recent tool `LiME` [24] automatically derives task models from real Linux workloads. We previously evaluated the impact of the `PREEMPT_RT` patch using a Raspberry Pi 5 [25], showing its benefits in reducing latency and improving determinism, which motivated further investigation in robotic settings.

III. System Design Overview

Our system is designed to enable cage-free human-robot workspaces by combining GPU-accelerated perception with reactive SSM safety control in a tightly integrated runtime. The hardware includes the following components (Fig. 1A):

- A UR10e [26] industrial robotic arm with an OnRobot VG10 suction gripper [27], connected via LAN to the central computer;
- Five Intel RealSense D435 [28] depth cameras, mounted around the robot Tool Center Point (TCP) on a custom 3D-printed fixture, connected via USB-C (3.2) to the central computer;
- A central computer equipped with an Intel i9-14900KF processor (32 CPUs), an NVIDIA GeForce RTX 4060 Ti (8 GB VRAM), and 2 TB solid-state storage;
- An Atmel ATmega-based Antropo light signaling module [7], [29], connected to the robot's 24 V digital I/O, providing status feedback: safety stop (red), slowed motion (orange), normal operation (green).

The software is implemented in C++ and CUDA, without ROS or external middleware, in order to reduce latency and maintain full control over scheduling, memory allocation, and data exchange. This design also facilitates fine-grained debugging, step-through inspection, and performance monitoring, which are often obscured by middleware like ROS [17]. Figure 2 illustrates the structure of the system software and its data flow, highlighting the timing-critical path from camera acquisition to safety actuation. Below, we describe the key software components of our runtime environment.

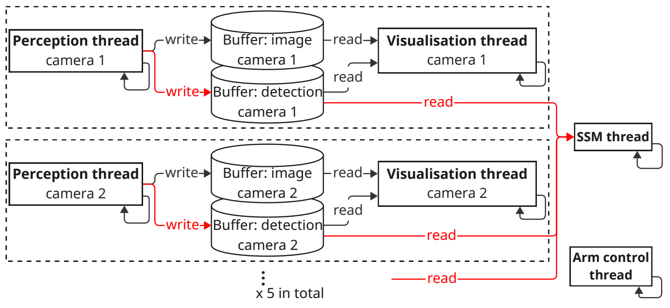
Perception thread camera 1
write
Buffer: image camera 1
read
Visualisation thread camera 1
write
Buffer: detection camera 1
read
read
SSM thread
Perception thread camera 2
write
Buffer: image camera 2
read
Visualisation thread camera 2
write
Buffer: detection camera 2
read
read
x 5 in total
read
Arm control thread

Fig. 2. Task graph of the SentryRT-1 runtime. Perception threads process camera input and write detection results to shared buffers, which are consumed by the SSM thread. The SSM and Arm control threads issue commands to the UR robot for motion and signaling. Visualization threads read images and detections to render annotated views. The critical path is shown in red.

Camera acquisition module. Each RealSense camera is polled at 30 FPS to retrieve aligned depth and RGB frames.

Perception module. A neural network detects and segments humans in the RGB frames, then computes their 3D positions by projecting the segmentation masks onto the corresponding depth images. The positions of humans and distances to the robot's Tool Center Point (TCP) are produced and passed to the SSM module for evaluation.

Trajectory control module. The robot executes predefined missions (e.g., picking and placing boxes with the suction gripper), following motion trajectories programmed via the Real-Time Data Exchange (RTDE) protocol [30] provided by Universal Robots, using the UR RTDE library [31].

SSM module. A continuously running safety loop monitors the distance between detected humans and the robot's Tool Center Point (TCP). When a predefined safety threshold is breached, the module dynamically reduces the robot's speed or halts its motion if the separation distance becomes too small. To implement this behavior, SentryRT-1 currently uses the RTDE speed slider interface [32][1]. Once the minimum required distance is re-established, the robot resumes its normal operating speed [5].

All components run as C++ `std::threads`, exchanging data via shared memory buffers. Inter-thread communication follows a double-buffering pattern, where one thread writes to a back buffer while another reads from a front buffer. This design enables non-blocking data exchange with minimal locking, reducing synchronization overhead and jitter. Although the SSM loop is the final enforcement point for safety decisions, its effectiveness depends on timely and reliable data flowing through the perception pipeline—including camera acquisition and inference threads that lie on the critical path. This motivates a real-time analysis of the system as a whole.

Why real-time matters. Under high system load, race conditions or scheduling delays in the acquisition threads can lead to stale or empty frames entering the pipeline. From the perspective of the SSM logic, this is functionally equivalent to real-world occlusion: in both cases, the robot loses visibility of its environment. To ensure safety, the system must detect such degraded input and trigger a precautionary stop (i.e., red light), but this requires that the SSM thread itself maintain real-time guarantees. If it too is delayed or starved, these safety violations may go undetected, resulting in unbounded latency or unsafe robot behavior. These observations highlight the need for real-time task modeling and motivate our experiments, which quantify reaction times under stress conditions.

[1]Future work will integrate certified safety interfaces, such as PROFIsafe.

TABLE I
SentryRT-1 Task Set Characterization.

Task	Function	Period	Deadline	Criticality
τ_{ssm}	SSM	2 ms	2 ms	High
τ_{p_i}	Perception (cam. i)	33 ms	33 ms	High
τ_{arm}	Arm mission control	Seconds	N/A	Low
τ_{v_i}	Visualization (cam. i)	Best-effort	N/A	Low

Implementation. The current prototype is structured around a single `main()` function, which launches threads for perception, visualization, arm motion, and safety monitoring. Each perception thread handles both camera acquisition and inference for one sensor (i.e., combining camera acquisition and perception modules), writing its results to shared detection buffers. The SSM thread runs periodically, reading the latest detections from all detection buffers, and issues speed updates to the robot, status signals to the LED module. Visualization threads render annotated RGB-frames to the screen for debugging and user feedback but are not latency-critical.

IV. Real-Time Modeling

To evaluate the real-time behavior of our robotic perception and control system, we model the runtime as a set of interacting real-time tasks, each corresponding to a core thread in the implementation. Our goal is to understand how different Linux configurations impact the end-to-end responsiveness of the SSM module—the latency-critical component responsible for enforcing safety constraints. This analysis must also consider upstream dependencies on perception threads (camera acquisition and GPU-accelerated inference), which share compute resources with the SSM loop and may introduce contention under load. Table I summarizes the system's tasks and their timing characteristics, which we detail below.

Perception Tasks (τ_{p_i}). For each camera i, a dedicated thread performs image acquisition and human detection inference. These tasks are periodic, with an intended execution rate of 30 Hz (i.e., period of 33.3 ms). Each perception thread writes to a shared double buffer consumed by the SSM and visualization threads. Inference execution time varies depending on the GPU and frame content, but results must be produced within one frame interval to maintain pipeline stability.

SSM Task (τ_{ssm}). This task runs periodically with a period and deadline of 2 ms (i.e., 500 Hz). It reads the latest detections from all perception buffers, computes human-to-robot distances, and updates the robot's speed and LED signals. This task represents the final safety-critical decision point and must complete execution within each period to ensure timely intervention in case of human proximity. The 2 ms budget reflects the worst-case latency for effective speed modulation. Since the task polls perception outputs every 2 ms, a new detection may wait up to one period before being processed by SSM thread; combined with a 2 ms execution deadline, this results in a worst-case reaction time of 4 ms. Failure to trigger a safety control input to stop the robot in the provided time constraint significantly increases the risk of injury.

Arm Control Task ($\tau_{\mathbf{arm}}$). This task executes a pre-defined pick-and-place routine via the robot's controller interface. It is not latency-sensitive and is scheduled as best-effort.

Visualization Tasks (τ_{v_i}). Each perception thread is paired with a UI thread that overlays human detection results on RGB frames and displays annotated output for monitoring and debugging purposes (see Figure 1B). These threads are non-critical and excluded from our real-time evaluation. However, they share buffers with both perception and SSM threads, introducing potential contention in a mixed-criticality setting—a topic we leave for future work.

V. Evaluation

A. Goals and Methodology

Our evaluation focuses on how Linux kernel configurations and scheduler policies affect the real-time behavior of SentryRT-1 under load. We aim to understand the responsiveness and determinism of the perception and SSM loops, which are critical for enforcing human-robot distance constraints. We structure our investigation around the following research questions:

- **RQ1:** How does the system's reaction time degrade under increasing CPU interference?
- **RQ2:** How do different scheduling policies (`CFS`, `RR`, `SCHED_DEADLINE`) impact latency guarantees?
- **RQ3:** What influence does the choice of Linux kernel (`generic`, `lowlatency`, `realtime`) have on worst-case and average latency?

To answer these, we run a series of stress tests on the system while varying scheduler policies and kernel builds. Experiments are repeated across two modes: *Virtual camera mode*, pre-recorded camera streams replayed from disk to simulate identical sensor input, and *Physical camera mode*, live streams from five Intel RealSense cameras attached via USB-C. Each run lasts **30 seconds** and is repeated **3 times** per configuration to capture variability. The experiments process is automated using the **benchkit** open source tool [33].

B. Task Mapping and Scheduling Policies

Each functional module in SentryRT-1 (e.g., perception, safety monitoring) is implemented as a dedicated C++ thread using `std::thread`. We refer to these as *main threads*. However, they rely on several external libraries—such as Intel RealSense, TensorRT, OpenGL, and UR RTDE—which internally spawn additional *subthreads* to handle frame acquisition, inference execution, LED control, and robot actuation.

This architecture introduces a challenge: assigning a scheduling policy to the main thread alone does not guarantee real-time behavior if its subthreads continue to run under the default `SCHED_OTHER` policy, which lacks real-time guarantees. In the presence of CPU interference, these subthreads may be preempted, delaying or even blocking the main thread and breaking end-to-end timing guarantees.

To address this, we implement a mechanism to dynamically identify all threads spawned by each main thread and apply a user-specified fallback scheduling policy to their subthreads. While we cannot assign `SCHED_DEADLINE` to subthreads—since doing so requires explicit knowledge of their execution parameters (e.g., runtime, deadline, period)—we evaluate a fallback option throughout the following combinations.

DL+CFS. Main threads use `SCHED_DEADLINE`; subthreads remain under the default `SCHED_OTHER` policy.

DL+RR. Main threads use `SCHED_DEADLINE`; subthreads are assigned `SCHED_RR` with a fixed static priority of 50.

RR+RR. Both main and subthreads are assigned `SCHED_RR` with fixed priority of 50.

CFS+CFS. Both main and subthreads use the default `SCHED_OTHER` policy. This reflects the unmodified behavior of standard Linux deployments.

This workaround—falling back on the `SCHED_RR` priority class for opaque subthreads—highlights a key limitation of the current Linux real-time scheduling API: assigning a scheduling policy to a parent thread does not automatically propagate to its child threads, necessitating manual intervention to ensure consistent timing behavior across all execution contexts.

For all threads assigned `SCHED_DEADLINE`, the `runtime`, `deadline`, and `period` parameters are configured based on the expected execution rates and constraints of each task, as summarized in Table I. Subthread priorities are equal within each configuration, and thread pinning is not enforced, allowing the Linux scheduler to dynamically manage core assignment. The arm mission control and visualization tasks are considered non-critical and remain scheduled under the default `SCHED_OTHER` policy.

To simulate non-critical interfering workloads—such as OS background tasks or network stack activity—we launch a configurable number of *noise threads* (from 0 up to 128, to test the limit of the system under high load) using the default `SCHED_OTHER` policy. Each thread executes a tight loop of relaxed atomic increments and decrements on a dummy counter, designed to saturate CPU pipelines. On our system with 32 logical CPUs, launching more than 32 such threads guarantees oversubscription and exposes the impact of CPU interference on real-time tasks.

C. Kernel Variants

In addition to scheduling policy, the kernel configuration plays a role in determining the responsiveness and latency behavior of real-time applications. We compare three Linux kernel variants provided by Ubuntu 22.04 [34], all based on version `5.15.0`:

- **`generic`** (`5.15.0-138-generic`): The standard Ubuntu kernel with voluntary preemption. It is optimized

TABLE II
EXPERIMENTAL VARIABLES AND THEIR EXPLORED VALUES.

Variable	**Values**
Camera mode	`Physical` (D435i) / `Virtual` (dataset)
Linux kernel	`generic` / `lowlatency` / `realtime`
Main threads policy	`SCHED_DEADLINE` (DL) / `SCHED_RR` (RR) / `SCHED_OTHER` (CFS)
Subthreads policy	`SCHED_RR` (RR) / `SCHED_OTHER` (CFS)
Noise thread count	0 / 16 / 32 / 64 / 128

for throughput and general-purpose workloads, but lacks guarantees on worst-case latency.

- **`lowlatency`** (`5.15.0-138-lowlatency`): A soft real-time kernel variant enabling more aggressive preemption to reduce interrupt handling latency and jitter.
- **`realtime`** (`5.15.0-1082-realtime`): A more preemptible real-time kernel distributed via Ubuntu Pro. It includes the `PREEMPT_RT` patchset, which converts most interrupt handlers into schedulable threads and supports bounded latencies.

D. Metric Collected

To ensure the robot does not collide with humans, the primary metric evaluated is the *reaction time* of the system's timing-critical path. We define *reaction time* as the duration between the arrival of a new input frame and the completion of the **first job** of the SSM task τ_{ssm} that processes this frame. Since τ_{ssm} runs at a significantly higher frequency than the perception tasks τ_{p_i}, the system's reaction to an input frame is effectively determined by this first polling instance of τ_{ssm} that corresponds to that frame; later instances for the same frame only maintain the current safety status until a new frame is available. This definition allows us to isolate the latency of the system's critical path—from sensor input to safety actuation—and to measure how kernel and scheduling decisions affect timely responsiveness under varying interference levels.

We evaluate both the **worst-case** and **average-case** reaction times across all frames in each run. Each experimental configuration is repeated 3 times, and we report the aggregated statistics (max and mean across repetitions) to capture variability and ensure repeatability. Table II summarizes the experimental variables and their explored values.

E. Results and Discussion

RQ1: Reaction time under interference. We begin by analyzing how the system's reaction time is affected by increasing CPU load, using configurations that vary the number of noise threads. Figure 3 shows that under the default `CFS+CFS` policy, both average and worst-case reaction times degrade rapidly once the number of noise threads exceeds the number of logical CPUs (32 on our platform). This is expected: critical tasks such as τ_{p_i} and τ_{ssm} receive no prioritization and must contend equally for CPU time.

In contrast, policies that assign real-time priorities to these threads (e.g., `RR+RR`, `DL+RR`, `DL+CFS`) remain resilient even under high contention. This confirms that prioritization—especially for perception and safety-critical threads—is essential to maintain bounded latency.

RQ2: Impact of scheduling policy. Across all interference levels, both `DL+RR` and `RR+RR` configurations show significant improvements in worst-case and average-case reaction time compared to `CFS+CFS`. These two policies are largely on par in our experiments, maintaining stable latency under load and shielding the critical path from CPU interference. The lack of a clear performance gap between `DL+RR` and `RR+RR` is explained by the system's ample resources: with 32 logical CPUs and relatively low per-task utilization, each real-time thread can be effectively isolated. In more constrained systems, where real-time tasks must share cores or operate closer to full CPU utilization, we expect the benefits of EDF-based scheduling (e.g., deadline enforcement and bandwidth guarantees in `SCHED_DEADLINE`) to become more pronounced.

We observe outliers even under real-time configurations. For example, under the `DL+RR` policy on the `generic` kernel with 0 noise thread and virtual cameras, the worst-case reaction time spikes to 81 ms. This is attributable to the unpredictability of GPU inference workloads, which in this case consumed up to 77 ms—far beyond the nominal 33 ms perception period. This underscores a key limitation: while the CPU scheduling is protected, tasks offloaded to the GPU may still introduce non-determinism.

RQ3: Kernel comparison and jitter. To examine the impact of kernel variants on timing variability, we zoom in on individual experimental runs. Figure 4 shows per-frame reaction times for a single run of each kernel configuration under fixed load (128 noise threads) and constant scheduling policy (DL+RR). This fine-grained view shows that the `realtime` kernel (with `PREEMPT_RT`) provides lower jitter and tighter latency bounds than both the `generic` and `lowlatency` kernels. However, its impact is smaller than that of the scheduling policy itself. This suggests that most benefits stem from correct prioritization and task isolation rather than kernel-level preemption improvements alone.

Highlighted setting. Under a top-performing setup—real camera input, high interference (128 noise threads), and `DL+RR` scheduling on a `realtime` kernel—the system reacts within **44-70 ms worst-case** and **15-17 ms average**. The default (`CFS+CFS`, `generic` kernel) shows 353 ms worst-case and 117 ms average. This yields about 80% and 85% reductions in worst-case and average times, respectively.

Summary of findings. These results support the conclusion that real-time Linux configurations—with explicit prioritization of safety-critical tasks and runtime visibility into subthreads—can significantly improve latency determinism in robotics workloads. Yet challenges remain: in particular, GPU tasks such as human detection are opaque to the scheduler and can lead to deadline violations even without CPU interference. This motivates further work in designing GPU-aware scheduling models, group-based deadline policies, or extending Linux with budget inheritance mechanisms across threads and heterogeneous resources.

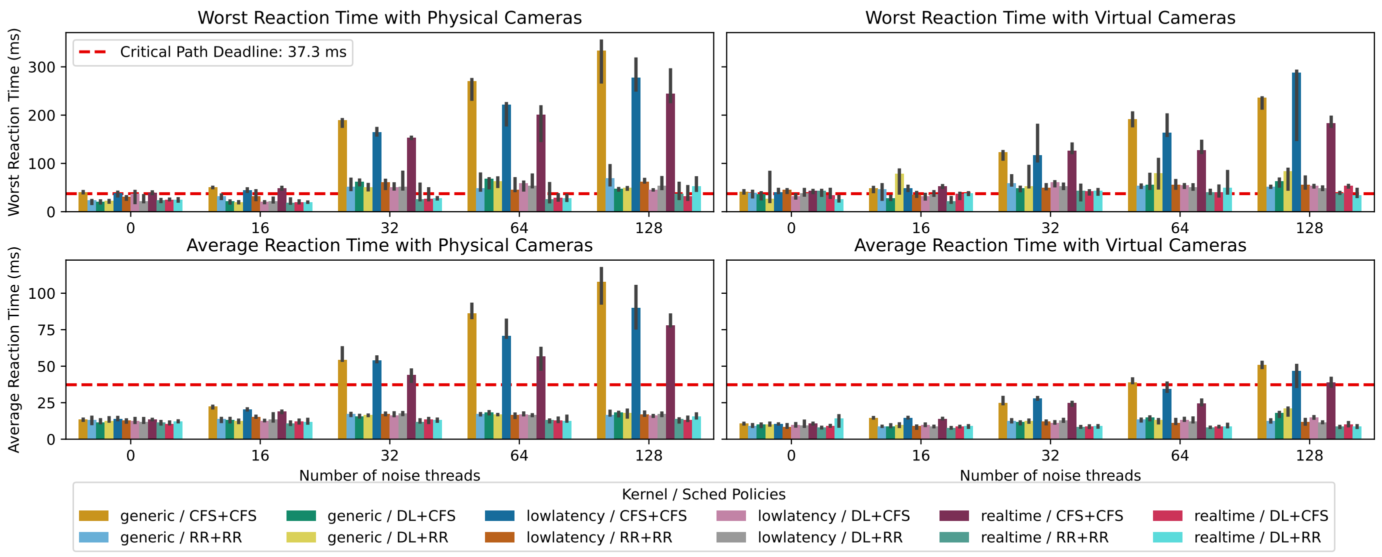

Fig. 3. Reaction times using physical and virtual cameras under different scheduling policies on various Linux kernels. Each 30-second experiment collects all reaction times (worst and average) per configuration. We repeat each experiment 3 times; bars represent the median value across the 3 runs, with error bars showing the minimum and maximum. Using the critical path deadline (37.3 ms) as a reference, the DL+RR policy with physical cameras meets the deadline as long as the number of noise threads does not exceed 32. Average reaction times for RR+RR, DL+CFS, and DL+RR also remain below this threshold. Real-time scheduling policies yield significantly lower reaction times than the default CFS+CFS policy under system interference.

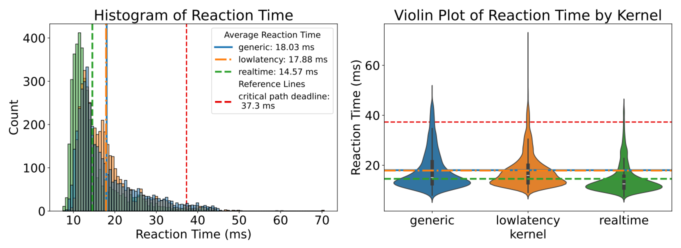

Fig. 4. Sample distribution of system reaction times on `generic`, `lowlatency`, and `realtime` Linux kernels. Each distribution corresponds to a single 30-second run using physical cameras, the DL+RR policy, and 128 noise threads. While the average reaction times on the `generic` and `lowlatency` kernels are nearly identical, the `realtime` kernel reduces the average by 3 ms. Moreover, the reaction time distribution under `realtime` shifts leftward, with a higher concentration of samples below the critical path deadline. Despite this improvement, all kernels experience deadline misses, primarily due to the unpredictable execution time of GPU tasks.

These experiments support the design direction of SentryRT-1, which aims to serve as a lightweight safety perception layer for robotics. An runtime capable of supporting real-time embedded systems with GPU acceleration and modular real-time scheduling is a key requirement for deployments.

VI. Conclusion

This paper presented SentryRT-1, a minimal C++/CUDA runtime that integrates multi-camera sensing and GPU-accelerated human detection for safety-critical robot control without middleware. By modeling the system as a real-time task set and evaluating its responsiveness under various Linux kernel and scheduler configurations, we demonstrated that proper use of `SCHED_DEADLINE` combined with a real-time kernel (with `PREEMPT_RT`) can significantly improve both average and worst-case reaction times. Compared to the default setup (`CFS` scheduler, `generic` kernel), our configuration reduced worst-case reaction time by **80%** and average time by **85%**, even under high CPU interference. This underscores the viability of Linux-based systems for enforcing Speed and Separation Monitoring (SSM) constraints.

To further improve performance under high interference, we plan to use the `timerlat` [23] tool to analyze deadline miss conditions, and use the `LiME` [24] tool to further refine the task model. Our results expose limitations in the current Linux real-time scheduling API, especially its inability to manage multi-threaded real-time tasks as unified entities. Ensuring consistent scheduling across main threads and library-spawned subthreads is a manual and error-prone process. This motivates the need for future work on group-level scheduling, graceful `SCHED_DEADLINE` inheritance, and deeper integration of scheduling policies with real-world runtime dependencies.

Beyond scheduler design, several other factors warrant investigation: the impact of thread pinning, I/O and memory contention, GPU sharing between critical and non-critical tasks, and interference from network traffic or unpredictable thread placement. Newer kernel features such as EEVDF scheduling, or alternative platforms including embedded SoCs like Jetson or heterogeneous CPU architectures with Performance- and Efficient-cores (e.g., Intel Alder Lake), offer further opportunities to test and refine our assumptions. Finally, extending SentryRT-1 to support heterogeneous sensor fusion, both rule-based and AI-driven scene interpretation, and modular perception pipelines will help evaluate its applicability in increasingly complex collaborative robotic scenarios.

Acknowledgment

This work was supported by imec through the SAFEBOT research program. The authors also acknowledge Universal Robots for their in-kind contribution of hardware components, which facilitated the system integration and evaluation.

References

[1] W. Dauth and S. Findeisen and Jens Suedekum and Nicole Woessner, "The Adjustment of Labor Markets to Robots," *Journal of the European Economic Association*, 2021.

[2] Shahab Sharfaei and Jan Bittner, "Technological employment: Evidence from worldwide robot adoption," *Technological Forecasting and Social Change*, 2024.

[3] M. Vasic and A. Billard, "Safety issues in human-robot interactions," in *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 197–204.

[4] T.-A. Tran, T. Ruppert, G. Eigner, and J. Abonyi, "Retrofitting-based development of brownfield industry 4.0 and industry 5.0 solutions," *IEEE Access*, vol. 10, pp. 64 348–64 374, 2022.

[5] International Organization for Standardization, *Robotics — Safety requirements — Part 2: Industrial robot applications and robot cells*, International Organization for Standardization Std. ISO 10 218-2:2025, 2025, second edition, published February 2025. [Online]. Available: https://www.iso.org/standard/73934.html

[6] C. Scholz, H.-L. Cao, E. Imrith, N. Roshandel, H. Firouzipouyaei, A. Burkiewicz, M. Amighi, S. Menet, D. W. Sisavath, A. Paolillo, X. Rottenberg, P. Gerets, D. Cheyns, M. Dahlem, I. Ocket, J. Genoe, K. Philips, B. Stoffelen, J. Van den Bosch, S. Latre, and B. Vanderborght, "Sensor-Enabled Safety Systems for Human–Robot Collaboration: A Review," *IEEE Sensors Journal*, vol. 25, no. 1, pp. 65–88, 2025.

[7] C. Scholz, H.-L. Cao, I. El Makrini, and B. Vanderborght, "Antropo: An open-source platform to increase the anthropomorphism of the Franka Emika collaborative robot arm," *Plos one*, vol. 18, no. 10, p. e0292078, 2023.

[8] "Offline Simulator - e-Series and UR20/UR30 - UR Sim for Linux," https://perma.cc/8DVY-6RG9, accessed: 2025-05-01.

[9] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074

[10] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 231–243.

[11] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J.-J. Chen, "End-To-End Timing Analysis in ROS2," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 53–65.

[12] A. H. Sifat, X. Deng, B. Bharmal, S. Wang, S. Huang, J. Huang, C. Jung, H. Zeng, and R. Williams, "A Safety-Performance Metric Enabling Computational Awareness in Autonomous Robots," *IEEE Robotics and Automation Letters*, vol. 8, no. 9, pp. 5727–5734, 2023.

[13] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *2016 International Conference on Embedded Software (EMSOFT)*, 2016, pp. 1–10.

[14] Y. Ye, Z. Nie, X. Liu, F. Xie, Z. Li, and P. Li, "ROS2 Real-time Performance Optimization and Evaluation," *Chinese Journal of Mechanical Engineering*, vol. 36, no. 1, p. 144, 2023. [Online]. Available: https://doi.org/10.1186/s10033-023-00976-5

[15] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 264–277.

[16] H. Abaza, D. Roy, B. Trach, W. Chang, S. Saidi, A. Motakis, W. Ren, and Y. Liu, "Managing End-to-End Timing Jitters in ROS2 Computation Chains," in *Proceedings of the 32nd International Conference on Real-Time Networks and Systems*, ser. RTNS '24. New York, NY, USA: Association for Computing Machinery, 2025, p. 229–241. [Online]. Available: https://doi.org/10.1145/3696355.3696363

[17] S. Barut, M. Boneberger, P. Mohammadi, and J. J. Steil, "Benchmarking Real-Time Capabilities of ROS 2 and OROCOS for Robotics Applications," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 708–714.

[18] A. Alhonen, "Why don't we use ROS?" http://pulurobotics.fi/blog/pulurobotics-blog-1/post/why-don-t-we-use-ros-7, 2017, pulurobotics Blog.

[19] H. Teper, D. Kuhse, M. Günzel, G. v. d. Brüggen, F. Howar, and J.-J. Chen, "Thread Carefully: Preventing Starvation in the ROS 2 Multithreaded Executor," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3588–3599, 2024.

[20] M. Bakhshalipour and P. B. Gibbons, "Agents of Autonomy: A Systematic Study of Robotics on Modern Hardware," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 7, no. 3, Dec. 2023. [Online]. Available: https://doi.org/10.1145/3626774

[21] V. Mayoral-Vilches, J. Jabbour, Y.-S. Hsiao, Z. Wan, M. Crespo-Álvarez, M. Stewart, J. M. Reina-Muñoz, P. Nagras, G. Vikhe, M. Bakhshalipour, M. Pinzger, S. Rass, S. Panigrahi, G. Corradi, N. Roy, P. B. Gibbons, S. M. Neuman, B. Plancher, and V. J. Reddi, "RobotPerf: An Open-Source, Vendor-Agnostic, Benchmarking Suite for Evaluating Robotics Computing System Performance," 2024. [Online]. Available: https://arxiv.org/abs/2309.09212

[22] L. T. Foundation, "Linux Foundation Realtime Wiki - HowTo - Cyclictest," https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start, accessed: 2025-05-01.

[23] D. B. D. Oliveira, D. Casini, J. Lelli, and T. Cucinotta, "Timerlat: Real-time Linux Scheduling Latency Measurements, Tracing, and Analysis," *IEEE Transactions on Computers*, pp. 1–13, 2025.

[24] Björn B. Brandenburg and Cédric Courtaud and Filip Marković and Bite Ye, "LiME: The Linux Real-Time Task Model Extractor," in *2025 IEEE 31th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2025.

[25] W. Dewit, A. Paolillo, and J. Goossens, "A Preliminary Assessment of the real-time capabilities of Real-Time Linux on Raspberry Pi 5," in *Proceedings of the 18th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2024, pp. 7–12. [Online]. Available: https://www.ecrts.org/wp-content/uploads/2024/07/ospert24-proceedings.pdf

[26] "UR10e, Medium-sized, versatile cobot," https://perma.cc/4PCV-C9WL, Accessed: 2025-05-01.

[27] "VG10 – Flexibler, Einstellbarer Vakuumgreifer," https://perma.cc/US4B-ZLDK, Accessed: 2025-05-01.

[28] "Intel RealSense D435," https://perma.cc/K5EN-7WHV, Accessed: 2025-05-01.

[29] H.-L. Cao, S. A. Elprama, C. Scholz, P. Siahaya, I. El Makrini, A. Jacobs, A. Ajoudani, and B. Vanderborght, "Designing interaction interface for supportive human-robot collaboration: A co-creation study involving factory employees," *Computers & Industrial Engineering*, vol. 192, p. 110208, 2024.

[30] "Real-Time Data Exchange (RTDE) Guide," https://perma.cc/6VCE-9KD6, Accessed: 2025-05-01.

[31] A. P. Lindvig, I. Iturrate, U. Kindler, and C. Sloth, "ur_rtde: An Interface for Controlling Universal Robots (UR) using the Real-Time Data Exchange (RTDE)," in *2025 IEEE/SICE International Symposium on System Integration (SII)*, 2025, pp. 1118–1123.

[32] "RTDE IO Interface API," https://sdurobotics.gitlab.io/ur_rtde/api/api.html#rtde-io-api, accessed: 2025-05-01.

[33] "Benchkit: Performance Evaluation Framework," https://github.com/open-s4c/benchkit/tree/main, accessed: 2025-05-01.

[34] "Ubuntu Kernel Database," https://kernel.ubuntu.com/mainline/, accessed: 2025-05-01.