Can Linux Fly a Drone? A Flight Control Evaluation for Unified UAV Architectures Using PREEMPT_RT Kernel

Luiz Giacomossi, Afshin Ameri, Håkan Forsberg, Ivan Tomasic, Baran Cürüklü
Mälardalen University
Västerås, Sweden

Tommaso Cucinotta
Scuola Superiore Sant'Anna
Pisa, Italy

***Abstract*—Modern UAV architectures increasingly aim to unify high-level autonomy and low-level flight control on a single General-Purpose Operating System, but complex multi-core System-on-Chips introduce significant timing indeterminism. This paper performs an architectural schedulability analysis of the `PREEMPT_RT` Linux kernel on Raspberry Pi 5, isolating the impact of kernel activation paths (deferred SoftIRQs versus real-time direct activation) on a 250-Hz control loop, and evaluates the closed-loop consequences through a Software-in-the-Loop experiment. Under heavy stress, the standard kernel exhibits worst-case latencies above 9 ms, and even POSIX real-time policies spike to 1.8 ms. `PREEMPT_RT`'s direct wake-up path reduces this by 88% to under 225 μs. The SIL experiment indicates the operational consequences: the standard kernel configuration loses stability and crashes, while `PREEMPT_RT` completes the reference trajectory. While `PREEMPT_RT` bounds OS-level latency, the residual jitter on modern edge-AI SoCs is primarily driven by shared memory hierarchy contention.**

***Index Terms*—Real-Time Systems, UAV Flight Control, Real-Time Linux, PREEMPT_RT, Embedded Systems, Real-Time Scheduling, Raspberry Pi**

I. Introduction

Unmanned Aerial Vehicles (UAVs) are evolving from remote-controlled aircraft into autonomous edge-computing nodes capable of onboard tasks including SLAM, computer vision, and swarm coordination [1]–[3]. Historically, these systems relied on a dual-processor architecture: a high-performance "companion computer" for autonomy and a low-power microcontroller (MCU) for flight stabilization. However, constraints on size, weight, and power (SWaP) favor *unified architectures*, where a single high-performance System-on-Chip (SoC) handles both mission logic and the safety-critical Flight Control System (FCS) [4].

The challenge lies in the operating system. While autonomy stacks require a General-Purpose Operating System (GPOS) like Linux, standard kernels are optimized for throughput rather than the determinism required by 250+ Hz control loops [5]. Modern high-performance SoCs (e.g., ARM Cortex-A76-based systems) introduce sources of indeterminism, including deep cache hierarchies, out-of-order execution, and shared memory controllers. On such platforms, scheduling latencies and kernel housekeeping tasks introduce control loop jitter that can affect flight stability [6], [7].

This motivates the `PREEMPT_RT` patch-set, which transforms the Linux kernel into a real-time system by converting spin-locks into rt-mutexes and making interrupt handlers preemptible kernel threads [8]. Despite its established benefits, prior work [5] focuses on older single-board computers such as Raspberry Pi 3 (RPi 3). These studies overlook modern heterogeneous multi-core architectures, where shared resources (L3 cache, DRAM bandwidth) and complex interrupt routing notably affect real-time behavior. Prior work also rarely links scheduling measurements to closed-loop control outcomes, leaving the impact of latency violations unclear.

This paper addresses these gaps with an architectural schedulability analysis of flight control workloads on the RPi 5, complemented by a Software-in-the-Loop (SIL) experiment executing the actual control code under the actual scheduler on the target platform. The BCM2712 SoC (quad-core Cortex-A76) reflects modern edge-AI characteristics, including out-of-order execution and complex memory hierarchies. We isolate the CPU scheduler from hardware I/O to answer three questions:

1) **Architectural Paths:** How do task activation paths (Deferred SoftIRQ vs. Direct Threaded IRQ) in standard and real-time kernels affect worst-case latency?
2) **Resource Contention:** To what extent does `PREEMPT_RT` reduce jitter from shared cache and memory bandwidth contention under heavy autonomy-like workloads?
3) **Operational Effects:** How do scheduling differences translate into closed-loop flight behavior under stress?

The contributions of this work are: 1) A quantitative benchmark of a 250-Hz control loop on a modern multi-core SoC, isolating scheduler effects from bus latency; 2) An architectural analysis showing that the standard kernel's Deferred Execution (SoftIRQ) path causes $>$ 9 ms latency spikes, while `PREEMPT_RT`'s Direct Activation reduces latency by 88% (1.84 ms to $< 225\mu$s); 3) Evidence that residual jitter in `PREEMPT_RT` is driven by hardware memory contention; 4) A SIL evaluation showing that under identical stress, the standard kernel lost stability and crashed while `PREEMPT_RT` completed the trajectory, demonstrating sufficient scheduling determinism for this SIL-based 250-Hz workload.

II. Background and Related Work

This section reviews real-time UAV control, examines existing software architectures, and contextualizes this work.

A. Overview of Real-Time UAV Control Systems

Flight control systems are hard real-time systems. They execute nested control loops, with high-frequency inner loops (200–500 Hz) for attitude stabilization and slower outer loops (50–100 Hz) for navigation [9]–[11]. The stability of these loops depends on the timing guarantees provided by the operating system, captured by the principle of *determinism*. In real-time systems, determinism means that task execution times are bounded so that deadlines can be guaranteed under specified conditions. This relies on two key properties:

- **Bounded Latency:** Scheduling latency is the delay between a task becoming ready and its dispatch to the CPU. Large or unpredictable latencies reduce the control phase margin and can destabilize the system [7]. For a 250-Hz loop (4-ms period) latencies exceeding a fraction of the period (e.g. $> 500\,\mu$s) can destabilize the control loop.
- **Low Jitter:** Jitter is the variation in latency across executions. Excessive jitter makes the control loop period inconsistent, complicates controller tuning (e.g., needs lower gains), and creates unpredictable dynamics [12].

B. Architectures for Embedded FCS

Embedded FCS architectures have evolved to support the increasing demand for onboard intelligence while managing SWaP (Size, Weight, and Power) constraints:

- **Microcontroller + RTOS:** The standard combines a microcontroller (e.g., STM32) with a Real-Time Operating System (RTOS) like FreeRTOS or NuttX. This provides excellent determinism (jitter typically $< 10\,\mu$s) but lacks the memory and compute throughput required for onboard computer vision or SLAM [13].
- **Dual-Processor Setup:** A high-level companion computer (e.g., RPi, Jetson) handles perception and communicates with a separate low-level RTOS microcontroller via a serial link (e.g., MAVLink) [14]. While robust, this adds weight, wiring complexity, and inter-process communication (IPC) latency.
- **Unified GPOS-Based System:** Control and autonomy are integrated onto a single high-performance SoC running a GPOS, typically Linux with real-time extensions [15], [16]. This reduces hardware complexity but introduces the real-time determinism challenge on complex multi-core architectures where cache and memory bus contention can cause unpredictable delays. This architecture is the focus of our study.

C. Linux Scheduling Policies

The Linux kernel scheduler implements several policies. The default policy, `SCHED_OTHER` (based on the Completely Fair Scheduler, CFS), is optimized for throughput and fair time-slicing but offers no timing guarantees [17]. For real-time applications, Linux provides alternative policies summarized in Tab. I. Foundational works [8] have established that standard kernels suffer from priority inversion and non-preemptible sections (kernel locks), necessitating the `PREEMPT_RT` patch to enforce bounded latencies.

TABLE I
LINUX KERNEL SCHEDULING POLICIES AND CHARACTERISTICS.

Policy	Type	Priority Mechanism	Key Feature
`SCHED_OTHER`	Time-Sharing	Dynamic ('nice' value)	Fair CPU sharing for throughput
`SCHED_FIFO`	Real-Time	Static (1-99)	Runs until block/yield
`SCHED_RR`	Real-Time	Static (1-99)	Round-robin time-slicing
`SCHED_DEADLINE`	Real-Time	Dynamic (EDF)	Temporal isolation via reservation

D. Related Work and Gap Analysis

Research into unified Linux-based flight control generally follows three tracks: dual-kernel extensions, virtualization, and native preemption.

1) Dual-Kernel (Xenomai) vs. Native Preemption: To overcome Linux's non-determinism, approaches like Xenomai (Cobalt) use a dual-kernel architecture, running a real-time microkernel alongside Linux. While Xenomai typically offers lower latency than `PREEMPT_RT` by bypassing the Linux scheduler entirely [18], it requires custom drivers (RTDM) and creates a complex split-system development environment. In the context of UAVs, we prioritize `PREEMPT_RT` because it maintains the standard Linux programming model. This allows standard robotics frameworks (e.g., ROS 2, MAVROS) and device drivers (e.g., V4L2 for cameras) to be used without modification, simplifying unified autonomy development.

2) Isolation via Virtualization: Yang and Shinjo [19] proposed a compounded RTOS (cRTOS) using *Jailhouse* to partition hardware between Linux and NuttX. While this ensures isolation, it replicates a dual-processor constraint in software, limiting resource sharing between autonomy and control layers.

3) Hardware Complexity Gap: Most `PREEMPT_RT` benchmarks focus on simpler Single Board Computers (e.g., RPi 3, Cortex-A53), establishing a baseline of approx. $150\,\mu$s worst-case latency under moderate load [20]. However, these studies do not account for modern *Out-of-Order (OoO)* architectures like the RPi 5's Cortex-A76. On these platforms, despite higher clock speeds, shared resource contention (L3 Cache/DRAM) becomes a primary source of timing violation [21]. Our work builds on this by quantifying how the `PREEMPT_RT` scheduler mitigates interference in these complex, heterogeneous environments.

III. FLIGHT CONTROL ARCHITECTURE

The FCS of the quadcopter is implemented using a cascaded architecture, described in [22] and depicted in Fig. 1. The outer loop features a *Position Controller*, that takes the position reference $\mathbf{r}_r$, the current position $\mathbf{r}$, and the current linear velocity $\mathbf{v}$ as inputs. It calculates the desired force vector $\mathbf{f}_r = [f_{x,r}, f_{y,r}, f_{z,r}]^T$ required to track the reference trajectory. Since the quadcopter is underactuated, $\mathbf{f}_r$ cannot be applied directly. Instead, its magnitude is used via a *Norm* block to produce the total commanded thrust F, while *Control Allocator-1* combines $\mathbf{f}_r$ with the desired yaw angle ψ_r to compute the required roll and pitch references (ϕ_r, θ_r).

The inner loop includes the *Attitude Controller*, which receives the target attitude references $(\phi_r, \theta_r, \psi_r)$, the current attitude $\boldsymbol{\alpha}$ (i.e., Euler angles), and the current body-frame angular velocity $\boldsymbol{\omega}_b$. It uses this information to compute the

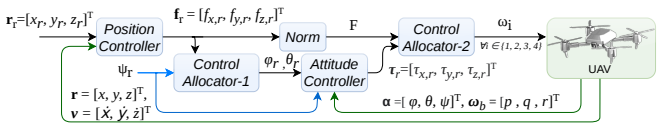


Fig. 1. Architecture of the FCS. *Position Controller* outputs a force vector $\mathbf{f}_r$, split into a thrust magnitude F and target attitude angles (ϕ_r, θ_r). *Attitude Controller* then computes the torque vector $\boldsymbol{\tau}_r$. Finally, *Control Allocator-2* maps the thrust and torques to separate motor speeds ω_i.

desired control torque vector, $\boldsymbol{\tau}_r$. Finally, *Control Allocator-2* receives the total commanded thrust F and the desired torque $\boldsymbol{\tau}_r$. It decomposes these commands into actuator-level inputs by computing the individual rotor speeds ω_i needed to achieve the desired response within actuator limits. The complete cascaded mapping from state and reference to actuator command is denoted $\pi(\cdot)$.

These actuator commands are applied to the UAV plant. In this work, to isolate the performance of the CPU scheduler from hardware I/O delays, the UAV's physical dynamics are handled by an in-process *UAV Simulation Block*. This block integrates the UAV's equations of motion and outputs the state variables (position $\mathbf{r}$, linear velocity $\mathbf{v}$, attitude $\boldsymbol{\alpha}$, and angular velocity $\boldsymbol{\omega}_b$) which are fed back to close the control loops. For more details on the dynamic equations and control laws behind this architecture, see [10], [22].

IV. System Model and Methodology

To empirically evaluate the architectural determinism of the `PREEMPT_RT` kernel, we designed a software framework[1] that explicitly isolates scheduler activation latencies from peripheral I/O delays. This section details the platform architecture, the task model, and the stress framework.

A. Platform Architecture: Raspberry Pi 5

We use RPi 5 (16GB RAM) as the target platform. Unlike previous generations of Single-Board Computers (SBCs) evaluated in literature [20], the RPi 5 utilizes the Broadcom BCM2712 SoC featuring a quad-core ARM Cortex-A76 cluster. To avoid thermal throttling in stress testing, active cooling was employed, and the CPU frequency was strictly locked to 2.4 GHz using the `performance` governor.

This choice is architecturally significant: the Cortex-A76 employs an out-of-order execution pipeline with a three-level cache hierarchy (64KB L1, 512KB L2 per core, and a shared 2MB L3). This complexity introduces sources of non-determinism—specifically shared L3 cache contention and DRAM bandwidth saturation—that are not present in simpler in-order microcontrollers. The system runs Ubuntu 24.04 LTS with two kernel configurations: the standard mainline kernel (`6.8.0-raspi`) and the fully preemptive real-time kernel (`6.8.0-raspi-realtime`).

B. Real-Time Software Architecture

The Flight Control System (FCS) is implemented as a set of concurrent, prioritized threads mapped to specific CPU cores. We focus our analysis on the critical high-frequency attitude control loop, as it represents the system's worst-case timing constraint. We define this periodic real-time task τ_{att} as:

$$\tau_{att} = \{C_{att}, T_{att}, D_{att}, P_{att}\} \tag{1}$$

where execution time C_{att} varies based on load, period $T_{att} = 4000\,\mu s$ (250 Hz), deadline $D_{att} = T_{att}$, and priority P_{att} is a variable defined in the experimental setup (see Sec. V). To mitigate OS-induced jitter, we apply an isolation strategy:

- **CPU Isolation:** We utilize the boot parameter `isolcpus=2,3` to exclude Cores 2 and 3 from the SMP balancing algorithms.
- **Task Pinning:** The critical task τ_{att} is pinned exclusively to Core 2 via `pthread_setaffinity_np`.
- **Inter-Thread Isolation:** Less critical tasks (logging and telemetry) are pinned to Core 3, ensuring they do not preempt the control loop.

Unlike decoupled architectures, we synchronously execute the entire control cascade (position, allocation, and attitude) within τ_{att} at 250 Hz. This oversamples the outer loop but intentionally eliminates inter-task latency and upstream failures as confounding variables, ensuring SIL deviations are strictly attributable to OS scheduling delays. The wake-up relies on `clock_nanosleep()` with `CLOCK_MONOTONIC`, which provides high resolution timing via `hrtimer`.

C. Workload Injection Framework

To analyze the architectural limits of the scheduler under worst-case conditions, we utilized `stress-ng`[2]tool to synthesize a unified interference profile. This profile concurrently executes distinct stressor components (Tab. II).

TABLE II
Components of the Stress Profile used for Evaluation.

Component	Stressor Mechanism	Targeted Bottleneck
Compute	`cpu-matrix` (4 workers)	ALU pipeline saturation
Memory	`vm` (75% RAM, 2 workers)	*L3 Cache Thrashing* & DRAM Bandwidth
Kernel	`switch, fork` (High rate)	Runqueue lock contention & *SoftIRQ Latency*

The profile includes: Compute, testing the scheduler's ability to preempt compute-bound threads; Memory, which triggers L3 cache evictions, simulating interference from memory-heavy autonomy workloads (e.g., Computer Vision) on other cores; and Kernel, flooding the system with context switches and interrupts to test the kernel's handling of the SoftIRQs described in Section VI.

D. Methodological Constraints

A challenge in any real-world FCS is the management of I/O latency (e.g., SPI bus transfers for IMU data). While hardware transfers are deterministic, driver stacks introduce variable delays. However, scheduler determinism is the necessary condition for real-time control; if the kernel cannot dispatch the control task within the deadline, the speed of the I/O driver is irrelevant. Therefore, this study intentionally decouples scheduler latency from I/O. For the experiments,

[1] github.com/luizgiacomossi/Real-Time-Flight-Control-With-Linux

[2] The `stress-ng`: github.com/ColinIanKing/stress-ng

τ_{att} executes the full control law mathematics but bypasses physical SPI transactions. This ensures that the measured jitter is attributable solely to CPU scheduling and kernel activation paths, enabling a precise architectural comparison of `PREEMPT_RT` versus standard Linux.

E. SIL Simulation Methodology

We quantify the impact of latency and jitter on flight control performance using a Software-in-the-Loop (SIL) experiment, where the FCS task and UAV dynamics run in real time on the platform. The SIL setup executes the actual control code under the target kernel and scheduler, with the UAV Simulation Block integrated in-process, synchronously on the same thread and core as the FCS task. The jitter affecting the controller is the same jitter the physics observes. An alternative is recording latency traces and replaying them in an offline simulator. This breaks the feedback loop: in a physical UAV, delayed control updates perturb the state, and subsequent inputs depend on that perturbation. Decoupling these stages ignores how errors compound over time, which can make the system appear more stable than it would be in practice.

1) Plant Model and Integration: The UAV dynamics block (Fig. 1) implements the 6-DOF quadrotor model from Bouabdallah [10] with state $x = [\mathbf{r}, \mathbf{v}, \boldsymbol{\alpha}, \boldsymbol{\omega}_b]^\top \in \mathbb{R}12$. The model is integrated using a forward Euler scheme with step size equal to the measured inter-activation interval $\Delta t_k = t_k - t_{k-1}$ (i.e., the actual time elapsed since previous loop iteration rather than the nominal period $T_{\text{att}} = 4\,\text{ms}$). Scheduling jitter's effect on physical deviation is as follows: when the kernel delays the wake-up by δ, the integrator runs for $T_{\text{att}} + \delta$ using the previous actuator command.

2) Control–Integration Ordering and ZOH Assumption: We adopt a zero-order hold (ZOH) model on the actuator command [23]. Within each iteration, the dynamics are integrated *before* the new control law is computed, using the command u_{k-1} from the previous cycle:

$$x_k = x_{k-1} + \Delta t_k \cdot f(x_{k-1}, u_{k-1}), \quad u_k = \pi(x_k, x_{\text{ref}}) \quad (2)$$

where $\pi(\cdot)$ denotes the cascaded position–attitude controller of Section III. This reflects the physical reality that during a scheduler-induced delay δ, the rotors continue at their last commanded speed; the controller cannot affect the plant until it is dispatched and produces a new command.

V. Experimental Evaluation

To quantify the performance difference between the standard Linux kernel and its `PREEMPT_RT` counterpart, we designed an experiment to measure the scheduling latency and jitter of a high-frequency UAV control loop. The inner attitude stabilization loop of our FCS was set to 250 Hz (4 ms period), a common frequency for agile multi-rotor platforms that is fast enough to require real-time determinism but slow enough to be achievable on embedded hardware if the OS is sufficiently responsive [9], [10]. For all tests, this critical task was pinned to an isolated CPU core (core 2) on a RPi 5 with memory locking enabled to prevent swapping delays.

The experimental matrix (Tab. III) was executed on two kernels: standard Linux (baseline), and a `PREEMPT_RT` kernel to evaluate its real-time capabilities, resulting 32 test runs. 16 configurations were tested on each kernel:

- `SCHED_OTHER` (at nice levels 0 and -19) to establish a non-real-time baseline.
- `SCHED_FIFO` and `SCHED_RR` (at static priorities 50 and 99) to assess POSIX real-time policies.
- `SCHED_DEADLINE` with runtimes of 400 μs and 800 μs, representing a tight and a generous task execution time budget. The period and deadline were set to 4 ms to match the 250 Hz control loop.

TABLE III
EXPERIMENTAL CONFIGURATIONS EXECUTED ON **EACH** KERNEL, FOR A TOTAL OF 32 TESTS. FOR `SCHED_DEADLINE`, THE DEADLINE AND PERIOD SET TO 4.0 MS, MATCHING THE CONTROL LOOP'S PERIOD.

Scheduler	Parameter Values	Period (ms)	Stress-ng	Tests per Kernel
`SCHED_OTHER`	Nice = {0, -19}	4.0	Off / On	$2 \times 2 = 4$
`SCHED_FIFO`	Priority = {50, 99}	4.0	Off / On	$2 \times 2 = 4$
`SCHED_RR`	Priority = {50, 99}	4.0	Off / On	$2 \times 2 = 4$
`SCHED_DEADLINE`	Runtime = {400, 800} μs	4.0	Off / On	$2 \times 2 = 4$
			Total	**16 per kernel**

Each configuration was tested with and without background load using `stress-ng`, configured to simulate a realistic system load representative of a UAV autonomy stack. Thus, a set of stressors were launched, namely, four CPU workers executing matrix multiplications to simulate tasks like path planning, two virtual memory stressors allocating 75% of available memory representing computer vision workloads stressing the system like real autonomy stacks would, without destabilizing the testbed, and multiple inter-process communication (IPC) stressors to mimic the data flow in frameworks like ROS. For determinism in the benchmark, the kernel's real-time runtime throttling was disabled by setting `sched_rt_runtime_us` to -1, and all CPU cores were locked to maximum frequency (2.4 GHz) via `performance` governor. Each of the 32 tests executed $10,000$ iterations of the control loop.

It is important to note that, all latency metrics are presented in microseconds (µs). These values were derived from high-resolution timers with nanosecond precision. The conversion to microseconds can introduce a visual quantization effect, observable as discrete steps in the time-series plots.

A. Analysis of Scheduling Determinism

We designed two follow-up experiments to investigate the origins of the non-determinism observed in the benchmark, particularly for `SCHED_OTHER`. The objective is to distinguish interference from high-level system services and fundamental architectural behaviors of the kernel scheduler. The experiments are:

1) **System-Level Isolation Analysis:** `SCHED_OTHER` benchmarks were re-run with the GUI disabled via `multi-user.target`, isolating scheduler performance from a major source of system nondeterminism.
2) **Task Activation Path Analysis:** Kernel traces collected using the `perf` tool to compare wakeup latency and event sequences under real-time (`SCHED_FIFO`) and standard (`SCHED_OTHER`) policies.

B. Software-in-the-Loop Evaluation

We apply the SIL methodology on the same platform to complement the latency benchmark with closed-loop SIL evidence. The drone tracks a four-waypoint sequence defined in the inertial frame, exercising both translational and yaw control. Each SIL run lasts 30 s, yielding 7,500 control iterations, providing full trajectory coverage across all waypoints.

The SIL evaluation uses two representative cases that bound performance, both under system stress: `SCHED_OTHER` (nice 0) on the standard kernel, characterizing an unmodified Linux deployment without real-time guarantees, and `SCHED_FIFO` (priority 99) on the `PREEMPT_RT` kernel, characterizing the configuration with the strongest preemption model and highest static priority among the evaluated policies. Since the remaining real-time policies exhibit comparable bounded latency behavior, evaluating these two extremes is sufficient to characterize the impact of scheduler choice on closed-loop control performance. Both runs apply the task pinning, memory locking, and CPU isolation parameters set in Sec. IV-B.

For each SIL run, we report per-axis position and attitude tracking against the reference trajectory, and trajectory completion, defined as reaching all four waypoints without state divergence (bounded altitude and attitude throughout the run). Note that the SIL evaluation isolates the impact of scheduler behavior on the FCS. Controller gain tuning and tracking performance optimization are outside the scope of this work. The controller parameters are fixed across all SIL runs so the differences in tracking are attributable to scheduling behavior.

VI. Results

Table IV presents the statistical results of the 32 experiments. While `PREEMPT_RT` paired with `SCHED_OTHER` is included for completeness, its fundamental lack of real-time guarantees excludes it from further plots and discussion. The remaining real-time findings are visualized via statistical latency distributions (Fig. 2). We further provide time-series traces highlighting scheduler-induced timing behavior for the worst-case standard kernel scheduler (Fig. 3) and the best-case `PREEMPT_RT` configuration (Fig. 4).

A. Statistical Distribution of Latency

Figure 2 provides a visual comparison of scheduling latency on both kernels when under heavy system stress.

On the standard kernel, Fig. 2(a), the default scheduler (`SCHED_OTHER`) exhibits very high worst-case latencies, reaching over 9400 μs, far exceeding the 4000 μs deadline (Tab. IV). The real-time schedulers also show significant latency spikes on the standard kernel. For `SCHED_FIFO` with priority 99, the maximum observed latency was 1848 μs. For `SCHED_RR` with priority 99, the maximum was 472 μs, and for `SCHED_DEADLINE` R800, it was 443 μs.

On the `PREEMPT_RT` kernel, in Fig.2(b), the maximum latencies are significantly lower for all RT policies. The data in Tab. IV quantifies this improvement: the peak latency for `SCHED_FIFO` p99 was reduced to 224 μs; for `SCHED_RR` p99, it was reduced to 182 μs; and for `SCHED_DEADLINE` R800, it was reduced to 197 μs.

TABLE IV
Performance Comparison of Standard vs. PREEMPT_RT Kernels for the Control Task.

Scheduler	Parameters	Kernel	Stress	Mean (µs)	Median (µs)	Max (µs)	StdDev (µs)	P90 (µs)	P99 (µs)
OTHER	Nice 0	Standard	No	58.53	54.00	2724.00	73.89	57.00	72.00
		Standard	**Yes**	**274.01**	**103.00**	**8626.00**	**563.32**	**451.00**	**3070.00**
		PREEMPT_RT	No	59.64	58.00	2570.00	36.78	59.00	66.00
		PREEMPT_RT	**Yes**	**196.84**	**97.00**	**9015.00**	**414.98**	**290.00**	**2420.00**
OTHER	Nice -19	Standard	No	56.59	54.00	2451.00	45.36	57.00	70.00
		Standard	**Yes**	**136.54**	**101.00**	**9424.00**	**227.57**	**188.00**	**642.00**
		PREEMPT_RT	No	64.79	58.00	2779.00	88.06	60.00	90.00
		PREEMPT_RT	**Yes**	**110.81**	**85.00**	**4351.00**	**186.65**	**124.00**	**669.00**
FIFO	Priority 50	Standard	No	4.07	3.00	45.00	2.90	6.00	21.00
		Standard	**Yes**	**58.37**	**39.00**	**700.00**	**48.83**	**137.00**	**201.00**
		PREEMPT_RT	No	5.29	4.00	135.00	4.55	7.00	21.00
		PREEMPT_RT	**Yes**	**36.07**	**28.00**	**160.00**	**23.78**	**68.00**	**123.00**
FIFO	Priority 99	Standard	No	4.25	4.00	63.00	1.97	5.00	12.00
		Standard	**Yes**	**49.37**	**40.00**	**1848.00**	**40.67**	**89.00**	**156.00**
		PREEMPT_RT	No	4.19	4.00	100.00	1.90	4.00	7.00
		PREEMPT_RT	**Yes**	**31.33**	**25.00**	**224.00**	**21.61**	**50.00**	**130.00**
RR	Priority 50	Standard	No	4.56	4.00	51.00	2.14	6.00	13.00
		Standard	**Yes**	**38.01**	**30.00**	**787.00**	**28.30**	**64.00**	**125.00**
		PREEMPT_RT	No	4.35	4.00	76.00	2.53	5.00	13.00
		PREEMPT_RT	**Yes**	**44.09**	**32.00**	**225.00**	**31.75**	**92.00**	**152.00**
RR	Priority 99	Standard	No	4.76	4.00	86.00	3.40	6.00	16.00
		Standard	**Yes**	**43.18**	**36.00**	**472.00**	**26.12**	**76.00**	**121.00**
		PREEMPT_RT	No	4.14	4.00	51.00	1.15	4.00	6.00
		PREEMPT_RT	**Yes**	**41.08**	**31.00**	**182.00**	**27.01**	**82.00**	**122.00**
DEADLINE	R400, D4000	Standard	No	6.53	7.00	33.00	2.23	8.00	16.00
		Standard	**Yes**	**43.77**	**36.00**	**345.00**	**27.17**	**78.00**	**137.00**
		PREEMPT_RT	No	4.20	4.00	96.00	1.86	5.00	7.00
		PREEMPT_RT	**Yes**	**44.16**	**33.00**	**209.00**	**29.57**	**90.00**	**137.00**
DEADLINE	R800, D4000	Standard	No	6.17	6.00	58.00	1.98	7.00	14.00
		Standard	**Yes**	**40.94**	**35.00**	**443.00**	**22.67**	**70.00**	**112.00**
		PREEMPT_RT	No	4.21	4.00	84.00	2.00	4.00	8.00
		PREEMPT_RT	**Yes**	**32.30**	**27.00**	**197.00**	**18.75**	**51.00**	**116.00**

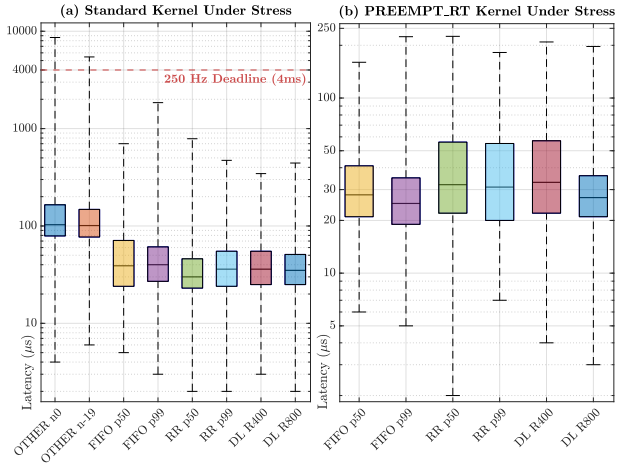


Fig. 2. Box plots - scheduling latency on the standard and `PREEMPT_RT` kernels under heavy system stress. The red dashed line indicates the control-loop deadline. (a) Standard kernel, using a logarithmic y-axis to show outliers exceeding the deadline. Real-time policies still exhibit significant jitter. (b) `PREEMPT_RT` kernel, showing all real-time schedulers, with worst-case latencies an order of magnitude below the deadline.

B. Iteration-by-Iteration Latency Behavior

To provide a view of system responsiveness, Fig. 3 and Fig. 4 depict the latency across 10,000 control loop iterations for the two bounding configurations. The standard kernel trace (Fig. 3) exhibits frequent spikes in execution time, indicating severe variability and reduced predictability under system stress. In contrast, the `PREEMPT_RT` kernel trace (Fig. 4) is more stable, maintaining a baseline with only occasional and much smaller outliers.

C. Analysis of `SCHED_OTHER` Non-Determinism

To investigate the additional latencies with `SCHED_OTHER`, the follow-up experiments in Section V-A were conducted. The first test isolated the impact of system-level services. As shown in Tab. V, disabling the GUI reduced the worst-case latency for `SCHED_OTHER` under stress by up to 61.4%. Despite this improvement, the maximum latency for the high-priority task

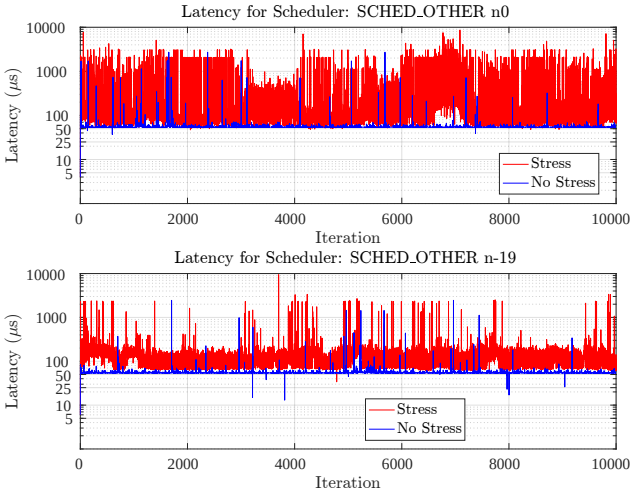


Fig. 3. Time-series latency under `SCHED_OTHER` on standard kernel: standard priority (nice 0, top) and high priority (nice -19, bottom). Note increased latency and jitter under stress (red) compared to the idle state (blue).

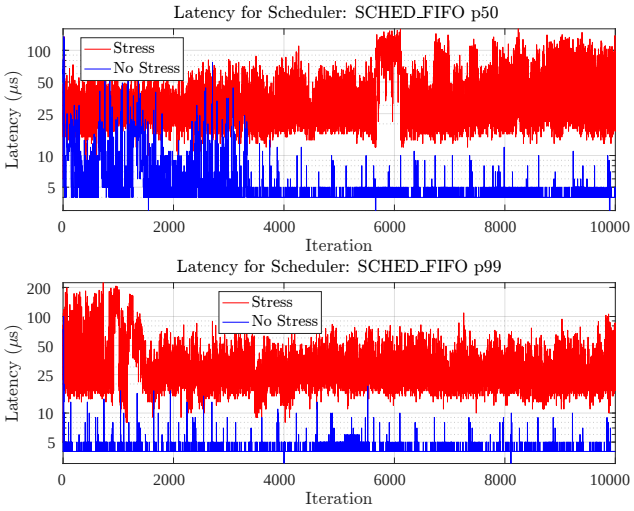


Fig. 4. Latency for `SCHED_FIFO` on the `PREEMPT_RT` kernel for priorities 50 (top) and 99 (bottom). The reduction of outliers under stress (red) shows improved effectiveness of the real-time kernel's preemption model.

(nice -19) still reached 3635 μs. This demonstrates that while high-level services are a significant source of interference, they do not account for all of the observed non-determinism.

TABLE V
IMPACT OF DISABLING THE GRAPHICAL USER INTERFACE (GUI) ON `SCHED_OTHER` WORST-CASE LATENCY UNDER SYSTEM STRESS.

Scheduler (Config)	Max Latency (GUI ON)	Max Latency (GUI OFF)	Improv. (%)
`SCHED_OTHER` (n-19)	9424 μs	**3635 μs**	**-61.4%**
`SCHED_OTHER` (n0)	8626 μs	**5993 μs**	**-30.5%**

To identify the remaining latency, we used `perf` to trace the activation of the control task. Tab. VI compares the timestamped events from a single control loop activation. The traces reveal a contrast in the wake-up mechanism. The `PREEMPT_RT` kernel allows the hardware timer interrupt to wake the user task directly, resulting in a wake-up latency of just 7 μs. In contrast, the standard kernel employs a *Deferred Path*: the interrupt offloads the timer expiry to the intermediate `ktimers` thread (SoftIRQ). In the trace, this indirection introduced a delay of over 100 μs and required three extra context switches before scheduling the flight task.

TABLE VI
KERNEL ACTIVATION SEQUENCE (TRACE DATA). TIMESTAMPS DENOTE Δt FROM THE PHYSICAL INTERRUPT ARRIVAL. THE "DEFERRED PATH" IN THE STANDARD KERNEL INTRODUCES SIGNIFICANT DELAYS ($> 100\,\mu$S) EVEN BEFORE THE TASK BEGINS EXECUTION.

PREEMPT_RT (Direct Path)		**Standard Kernel (Deferred Path)**	
Δt (μs)	**Event**	Δt (μs)	**Event**
0	`irq_handler_entry`	0	`irq_handler_entry`
2	`sched_wakeup` (target: **fcs**)	5	`irq_softirq_raise` (HRTIMER)
4	`sched_switch` ($\rightarrow$ **fcs**)	17	`sched_wakeup` (target: **ktimers**)
7	**Task Execution Begins**	54	`sched_switch` ($\rightarrow$ **ktimers**)
-	-	92	`timer_expire` (ktimers)
-	-	93	`sched_wakeup` (target: **fcs**)
-	-	**117**	**Task Execution Begins**

The traces also revealed a secondary effect: the control algorithm's execution time (excluding scheduling delay) increased from $\approx 3.8\,\mu$s in the RT kernel to $\approx 51\,\mu$s in the standard kernel. With identical payloads, this $13\times$ slowdown suggests cache pollution induced by the intermediate kernel housekeeping threads, though we did not isolate this effect with a dedicated cache-miss experiment. Kernel housekeeping threads (`ktimers`, `kworker`) plausibly flush instructions and data from the L1/L2 caches, forcing the control task to fetch data from slower main memory when it finally runs. This would indicate that the standard kernel suffers not only from scheduling latency but also from induced memory hierarchy contention—a hypothesis we leave for future validation.

D. Software-in-the-Loop Results

The SIL runs[3] reveal a categorical difference in flight outcome between the two kernel and scheduler configurations. Under the worst-case scheduler (`OTHER_n0`, standard kernel, stressed), the controller fails to maintain stability as shown in Fig. 5, the trajectory diverges shortly after the start point with the UAV spinning uncontrollably and crashing, marked as the crash point. The position trace in Fig. 6 (top) confirms this outcome—altitude (Z) collapses and lateral position (X) drifts out of bounds within the first `4s` after take-off. The attitude plot shows uncontrolled angular motion immediately preceding the crash, indicative of the loss of stabilization.

In contrast, the best-case configuration (`FIFO_p99`, `PREEMPT_RT`, stressed) successfully completes the full sequence under identical controller, dynamics, and stress conditions. The trajectory in Fig. 5 closely follows the reference path, with bounded transient overshoot at waypoint transitions. The position plot shows that all three axes track their step references with settling times consistent with the controller dynamics. The attitude plot (Fig. 6, bottom) shows roll and pitch excursions remain bounded within ± 0.35 rad in aggressive maneuvers, returning to zero between waypoints.

VII. DISCUSSION AND ARCHITECTURAL ANALYSIS

The data reveals a distinct separation between the kernel's ability to schedule tasks (SW) and the platform's ability to serve memory requests (HW). This section interprets the

[3]Experiments animation: github.com/luizgiacomossi/Real-Time-Flight-Control-With-Linux/blob/main/experiments_animation.md

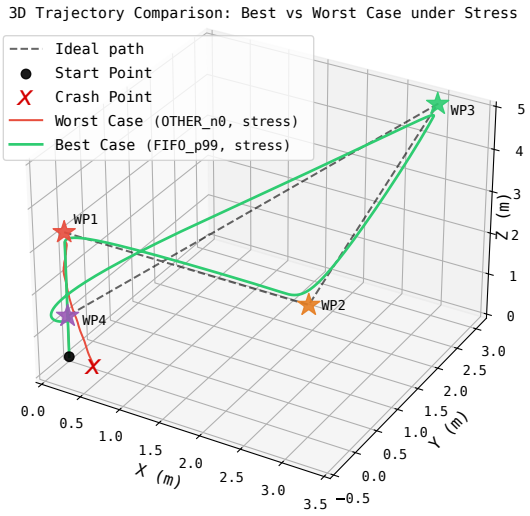


Fig. 5. Trajectory comparison under stress. The worst case (`OTHER_n0`, standard) diverges and terminates at the crash point. The best case (`FIFO_p99`, `PREEMPT_RT`) reaches all waypoints, following the reference.

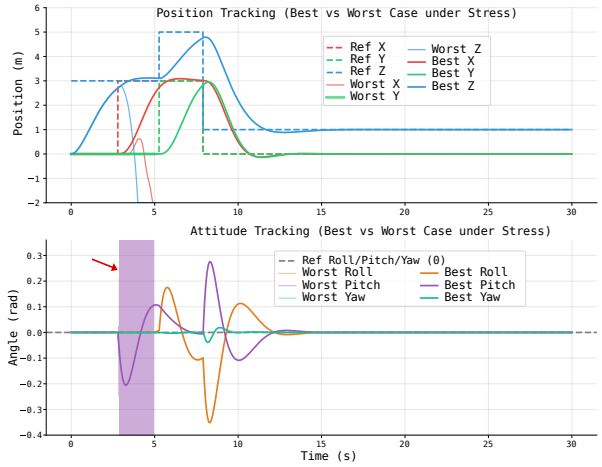


Fig. 6. Position (top) and attitude (bottom) tracking under stress. Worst-case position diverges within the first 4 s (with $Z < -2$ m indicating a crash), also illustrated by the red arrow marking uncontrollable spinning of the UAV. Best-case tracks all step references with bounded attitude excursions.

results through architectural lenses, isolating the sources of determinism and establishing a feasibility baseline relative to traditional microcontroller-based architectures.

A. Determinism: Direct vs. Deferred Activation

The primary source of the $>$ 9 ms latency spikes in the standard kernel (Tab. IV) is the Deferred Activation Path. In standard Linux, a timer interrupt does not instantly wake the user-space task but schedules a Software Interrupt (SoftIRQ), often deferred to `ksoftirqd` to maintain throughput. Under stress, runqueue lock contention delays this SoftIRQ, decoupling the physical interrupt from the task wake-up.

In contrast, `PREEMPT_RT` enforces a Direct Activation Path by turning interrupt handlers into preemptible threads, flattening the activation hierarchy. Our traces (Tab. VI) show that the critical path becomes $IRQ \rightarrow Task$, bypassing the non-deterministic SoftIRQ layer. This change yields an 87.9% reduction in worst-case latency for `SCHED_FIFO`, enabling sub-ms response for 250 Hz control.

B. The Residual Jitter: Memory Hierarchy Contention

While `PREEMPT_RT` bounds OS scheduling latency, a residual worst-case jitter of 200–225 μs persists even on isolated CPU cores. This points to the architectural complexity of the Pi 5's BCM2712 SoC, rather than the kernel scheduler.

Unlike MCUs with dedicated SRAM, the quad-core Cortex-A76 shares a unified L3 cache and DRAM controller. We hypothesize that under synthetic memory stress, "noisy neighbor" threads on Cores 0–1 induce memory bus contention and L3 cache thrashing. This is supported by the 13$\times$ execution time increase of the control payload (from $\approx 3.8\,\mu$s to $\approx 51\,\mu$s) seen in the trace analysis. Even with the flight control task pinned to Core 2, it likely suffers memory stalls, repeatedly fetching evicted instructions and data from slower main memory. Consequently, on modern edge-AI SoCs, reducing jitter to MCU levels ($< 50\,\mu$s) requires more than kernel preemption; it needs hardware-level spatial partitioning (e.g., cache coloring), which remains an open challenge in Linux.

C. Feasibility Analysis: RT-Linux vs. MCU Baseline

To contextualize these results for UAV designers, we compare the measured performance of the RPi 5 against the industry-standard microcontroller (MCU) baseline, typified by an STM32F7 running an RTOS (e.g., FreeRTOS/NuttX).

TABLE VII
ARCHITECTURAL COMPARISON: RPI 5 VS. TYPICAL MCU FC. RPI 5 VALUES ARE FROM OUR EXPERIMENTAL DATA; STM32 VALUES ARE REFERENCE METRICS FOR CORTEX-M7 RTOS [7], [13].

Metric	RPi 5 (RT-Linux)	STM32F7 (RTOS) [7], [13]
Clock Speed	2.4 GHz	216 MHz
Worst-Case Jitter	$\approx 224\,\mu s$	$\approx 10 - 20\,\mu s$
Control Period (250Hz)	$4000\,\mu s$	$4000\,\mu s$
Jitter % of Period	**5.6%**	**0.5%**
Compute Headroom	High (Vision/AI)	Low (Control Only)

The `PREEMPT_RT` Linux solution exhibits (Table VII) near 10$\times$ higher jitter than a bare-metal RTOS [13]. However, in the context of a 250 Hz control loop, the 224 μs worst-case jitter represents only 5.6% of the timing budget. For flight stability, this margin is generally acceptable [6]. While Linux cannot match an MCU's microsecond-level precision, the RPi 5 with `PREEMPT_RT` provides *sufficient* scheduling determinism for a 250 Hz loop decoupled from physical I/O, unlocking significant computational power for autonomy.

D. Implications for Flight Stability

Flight stability is contingent on the controller's reaction to disturbances within a bounded phase margin. The standard kernel exhibited worst-case latencies of 9.4 ms, exceeding the 4.0 ms period of a 250 Hz attitude loop by a factor of 2.3$\times$. In a physical system, this control blackout forces actuators to hold outdated commands for multiple cycles, effectively opening the control loop. During aggressive maneuvers, such delays lead to state divergence and potential loss of control. The SIL results illustrate this empirically: Under stress, the standard kernel configuration diverges within the first 4 s of the trajectory, and the UAV crashes shortly after takeoff.

However, the `PREEMPT_RT` kernel bounds the worst-case jitter to 225 μs (5.6% of the period). This ensures that the

control law is computed with fresh state estimates in every cycle, maintaining the phase margin required for safe flight even under heavy CPU saturation. Under identical stress and controller settings, the FIFO_P99 configuration completes the full four-waypoint trajectory, indicating that the latency bound is sufficient for closed-loop stability in practice.

E. I/O Latency Considerations

While this study finds that PREEMPT_RT bounds *computational* scheduling latency to levels compatible with 250 Hz control ($< 225\,\mu$s), scheduler determinism is a necessary but insufficient condition for flight stability. End-to-end performance remains dependent on the I/O subsystem. In unified architectures, standard Linux device drivers can introduce unbounded latencies via spinlocks or non-threaded interrupts, potentially negating the scheduler's guarantees. Thus, achieving theoretical performance demonstrated in this work likely requires bypassing standard kernel driver stacks in favor of Userspace I/O or DMA-based transfer mechanisms. This isolates the high-frequency control loop from the blocking behavior inherent to standard Linux peripherals.

VIII. Conclusion

We evaluated the PREEMPT_RT Linux kernel against the standard kernel for a 250 Hz UAV flight control task on a RPi 5, and quantified the closed-loop consequences through a SIL experiment on the target platform. The standard kernel showed to be unsuitable: its default scheduler produced worst-case latencies above 9 ms and its POSIX real-time schedulers exhibited spikes up to $1848\,\mu$s under stress. The SIL run confirmed the operational consequence: the UAV lost stability and crashed before completing the trajectory.

In contrast, PREEMPT_RT with any real-time policy bounded latencies below $225\,\mu$s under load, and FIFO_p99 completed the four-waypoint trajectory under identical stress. Because this behavior is tied to the threaded-IRQ mechanism, the results extend to recent real-time kernel versions (now in Linux mainline), providing evidence that PREEMPT_RT provides bounded scheduling latency for this control workload. Since our study isolated CPU scheduling from I/O and focused on a single critical control task, future work could evaluate the scheduling interactions between multiple concurrent high-priority tasks, integrate sensor and actuator I/O, extend the analysis to more demanding controllers such as Model Predictive Control and Sliding Mode Control, and explore the trade-off between real-time performance and power consumption.

Acknowledgment

This research was funded by the Horizon Europe research and innovation program of the European Union and the Chips Joint Venture under GA No. 101194287, NexTArc (Next Generation Open Innovations in Trustworthy Embedded AI Architectures for Smart Cities, Mobility, and Logistics).

References

[1] S. Javed, A. Hassan, R. Ahmad, W. Ahmed, R. Ahmed, A. Saadat, and M. Guizani, "State-of-the-art and future research challenges in uav swarms," *IEEE Internet of Things Journal*, vol. 11, no. 11, 2024.

[2] J. A. Ricardo, L. Giacomossi, J. F. S. Trentin, J. F. B. Brancalion, M. R. O. A. Maximo, and D. A. Santos, "Cooperative threat engagement using drone swarms," *IEEE Access*, vol. 11, pp. 9529–9546, 2023.

[3] L. Giacomossi, M. R. O. A. Maximo, N. Sundelius, P. Funk, J. F. B. Brancalion, and R. Sohlberg, "Cooperative search and rescue with drone swarm," in *International Congress and Workshop on Industrial AI and eMaintenance 2023*. Springer Nature Switzerland, 2024, pp. 381–393.

[4] V. Kangunde, R. S. Jamisola Jr, and E. K. Theophilus, "A review on drones controlled in real-time," *International journal of dynamics and control*, vol. 9, no. 4, pp. 1832–1846, 2021.

[5] A. Carvalho, C. Machado, and F. Moraes, "Raspberry pi performance analysis in real-time applications with the rt-preempt patch," in *2019 Latin American Robotics Symposium (LARS)*, 2019, pp. 162–167.

[6] J. A. Ricardo Jr and D. A. Santos, "Smooth second-order sliding mode control for fully actuated multirotor aerial vehicles," *ISA Transactions*, vol. 129, pp. 169–178, 2022.

[7] R. Rico, J. Rico-Azagra, and M. Gil-Martínez, "Hardware and rtos design of a flight controller for professional applications," *IEEE Access*, vol. 10, pp. 134 870–134 883, 2022.

[8] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on preempt_rt," *ACM Comput. Surv.*, vol. 52-1, 2019.

[9] W. Giernacki, M. Skwierczyński, W. Witwicki, P. Wroński, and P. Kozierski, "Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering," in *2017 22nd international conference on methods and models in automation and robotics (MMAR)*. IEEE, 2017, pp. 37–42.

[10] S. Bouabdallah, "Design and control of quadrotors with application to autonomous flying," Ph.D. dissertation, École Polytechnique Fédérale de Lausanne, 2007.

[11] R. W. Beard and T. W. McLain, *Small unmanned aircraft: Theory and practice*. Princeton university press, 2012.

[12] K. Smeds and X. Lu, "Effect of sampling jitter and control jitter on positioning error in motion control systems," *Precision Engineering*, vol. 36, no. 2, pp. 175–192, 2012.

[13] L. Meier, D. Honegger, and M. Pollefeys, "Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 6235–6240.

[14] X. Qu, Y. Wei, Y. Liu, and X. Su, "Design of automatic search and rescue uav based on jetson nano combined with px4 pixhawk flight controller and color recognition technology," in *2024 Int. Conf. Electr. Drives Power Electron. Eng.*, 2024, p. 460–466.

[15] A. Mårtensson, "Flying penguins: Building and evaluating the viability of a linux-based drone," 2016, bachelor Thesis in Computer Science, Educational program: DVGDS Computer and System Science.

[16] W. E. Hong, J. S. Lee, L. Rai, and S. J. Kang, "Rt-linux based hard real-time software architecture for unmanned autonomous helicopters," in *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, 2005, pp. 555–558.

[17] R. Love, *Linux kernel development*. Pearson Education, 2010.

[18] J. H. Brown and B. Martin, "How fast is fast enough? choosing between xenomai and linux for real-time applications," in *Proceedings of the 12th Real-Time Linux Workshop (RTLWS)*, 2010, pp. 1–17.

[19] C.-F. Yang and Y. Shinjo, "Obtaining hard real-time performance and rich linux features in a compounded real-time operating system by a partitioning hypervisor," in *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020, pp. 59–72.

[20] G. K. Adam, "Real-time performance and response latency measurements of linux kernels on single-board computers," *Computers*, vol. 10, no. 5, p. 64, 2021.

[21] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson, "Slow and steady: Measuring and tuning multicore interference," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 1–13.

[22] L. Giacomossi, A. Caregnato-Neto, and M. R. O. A. Maximo, "Optimization of force and torque bounds for the flight control system of a quadcopter using pso," in *2022 Latin American Robotics Symposium (LARS)*, 2022, pp. 1–6.

[23] K. J. Åström and B. Wittenmark, *Computer-Controlled Systems: Theory and Design*, 3rd ed. Dover Publications, 2011.