FlyOS: Integrated Modular Avionics for Autonomous Multicopters

Anam Farrukh
Boston University
afarrukh@bu.edu

Richard West
Boston University
richwest@bu.edu

***Abstract*— Autonomous multicopters often feature federated architectures, which incur relatively high communication costs between separate hardware components. These costs limit the ability to react quickly to new mission objectives. Additionally, federated architectures are not easily upgraded without introducing new hardware that impacts size, weight, power and cost (SWaP-C) constraints. In turn, such constraints restrict the use of redundant hardware to handle faults.**

In response to these challenges, we propose *FlyOS*, an Integrated Modular Avionics (IMA) approach to consolidate mixed-criticality flight functions in software on heterogeneous multicore aerial platforms. FlyOS is based on a separation kernel that statically partitions resources among virtualized sandboxed OSes. We present a dual-sandbox prototype configuration, where *timing- and safety-critical* flight control tasks execute in a real-time OS alongside mission-critical vision-based navigation tasks in a Linux sandbox. Low latency shared memory communication allows flight commands and data to be relayed in real-time between sandboxes. A hypervisor-based fault-tolerance mechanism is also deployed to ensure failover flight control in case of critical function or timing failures. We validate FlyOS's performance and showcase its benefits when compared against traditional architectures in terms of predictable, extensible and efficient flight control.

***Index Terms*—integrated modular avionics, autonomous multicopters, partitioning hypervisor, real-time flight control, fault-tolerance**

I. Introduction

Multicopters have traditionally adopted a federated architecture [43,56], which isolates and distributes flight management functions of different criticalities across separate hardware components [13,20]. Relatively powerful multicore CPUs are managed by a general purpose operating system (GPOS) such as Linux, and execute low time-sensitivity mission tasks. At the same time, an embedded microcontroller, or digital signal processor (DSP), processes the critical low-level flight control stack, often referred to as the autopilot. Connected locally via a slow serial (UART) interface, the loosely-coupled framework suffers from high latency and limited bandwidth communication when transferring commands between the two subsystems. This severely restricts the throughput and responsiveness of autonomous mission tasks, leading to coarse-grained drone control.

In order to ensure fault-tolerance against critical functional failures, the combined hardware and software stack of the low-level flight controller requires redundancy, which quickly becomes prohibitive given the limited size, weight, power and cost (SWaP-C) requirements of small-scale ($<$ 10kg) UAVs [28]. Additionally, constantly evolving autopilot features and functionality updates often render the resource constrained controller architecture obsolete, adding to hardware replacement and maintenance costs over time. Lack of flexibility in customization of autopilot stacks thus restricts a wide-spread deployment of multicopters across the connected aerial infrastructure.

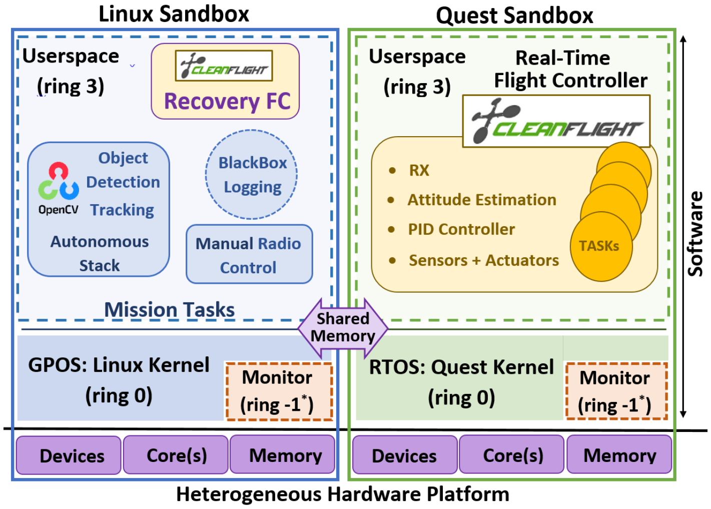


Fig. 1: FlyOS dual-sandbox configuration: Linux + Quest.
**For the purposes of this work, we identify ring -1 to be the root mode software layer, which sits between the hardware and non-root guest.*

In this work, we present FlyOS, which challenges the traditional federated approach to implement a predictable, safe and extensible flight management system. FlyOS takes inspiration from Integrated Modular Avionics (IMA) [84,85] and ARINC-653 (Avionics Application Software Standard Interface) [26,60] partitioning standard for avionic functions. These design guidelines envision consolidation of mixed-criticality flight functions on a centralized hardware platform, while ensuring temporal and spatial isolation of critical software components from execution-time interference.

FlyOS employs a separation kernel [72] to map two or more guest operating systems to virtualized sandbox domains. Virtualization technologies, featured by modern heterogeneous platforms, are used to statically partition hardware resources (processing cores, memory and I/O devices) and software components between separate execution environments. The individual system partitions operate together as a tightly coupled *distributed system-on-a-chip*. Explicitly defined shared memory communication channels set up low-latency and high bandwidth control and data paths between sandboxes. Isolation between guest domains allow for safe, secure and predictable consolidation of functional avionic components.

FlyOS enables software redundancy to meet SWAP-C constraints of small-scale UAVs. The system aims to overcome IMA's inherent limitation to fault containment by providing strict temporal and spatial partitioning between guests. FlyOS therefore protects against fault propagation across guest boundaries avoiding system-wide failure and corruption.

For our prototype implementation, shown in Fig. 1, we map the distributed companion architecture of traditional multicopter systems entirely in software using a dual-sandbox approach. Timing and safety critical flight control modules are implemented as latency-sensitive threads in a lightweight real-time OS (Quest [34]), alongside mission control tasks in Yocto Linux. FlyOS works on the principle of partitioning hypervisors [55,67,73,86] whereby each guest directly manages its own set of allocated resources without any run-time intervention of the most trusted compute base (TCB) of the hypervisor. It differs in its partitioning scheme compared to the state-of-the-art ARINC-653 extended architectures, which predominantly employ consolidating hypervisors [32,81,82]. These systems rely on the hypervisor for management of shared resources on behalf of the hosted guests. Hypervisor-based shared resource management potentially adds undue overheads, which impact predictability and determinism of critical flight control.

In this work, we refactor a performance-critical flight controller to execute with real-time guarantees on Quest. A camera-based vision detection and tracking subsystem is then implemented in a Linux sandbox as part of our mission control functionality (e.g., to represent a search and rescue objective). We also implement a hypervisor-based fault-recovery subsystem for fail-safe flight control in the presence of critical actuation failures.

Contributions: In this paper, we: 1) lay down the foundation for next-generation flight architectures designed around the principle of integrated modular avionics for multicopters, 2) describe FlyOS's separation kernel in the context of a dual-sandbox implementation co-hosting Linux with Quest, 3) implement a *timing*- and *safety-critical* flight stack with a *low-level* attitude (3D orientation) controller by retrofitting a well-known autopilot as a real-time application, 4) introduce *high-level mission-critical* autonomous navigation control, and 5) implement online fault-tolerance for time-bounded activation of failover flight control.

We evaluate FlyOS's performance with real-world experiments on a quadcopter. We also compare inter-sandbox communication overheads against a typical companion-board architecture of a popular drone system manufactured by Intel®. FlyOS opens opportunities for system-wide optimizations, reconfigurability and improved resource usage, while reducing size, weight and power requirements of the underlying hardware.

The following section describes the FlyOS model, by motivating our design goals, followed by an overview of the system framework. Section III presents an extensive evaluation of flight performance with hardware-in-the-loop experiments. Related work is described in Section IV, while conclusions and future work are discussed in Section V.

II. FlyOS: A Flight Management Framework

A. Motivation

FlyOS is designed around a characteristic set of goals for functional safety, timing predictability and efficiency of flight control for multi-rotor UAVs. As such, this work targets timing- and safety-criticality [64] dimensions of the mixed-criticality architecture design-space for drone autopilots. We define safety-criticality as a measure of functional importance of a software component to the overall flight control operation. Timing criticality on the other hand is concerned with guaranteeing real-time flight control responses within prescribed temporal bounds.

Orthogonal to this work, we define a third dimension of security criticality [65,66] for tasks and system components, which directly concerns policies related to the preservation of information integrity and confidentiality. Although an implementation and evaluation of such policies is beyond the scope of this paper, we note that the FlyOS architecture lends itself to support security capabilities such as gateway services at communication interfaces between sandboxed domains. This allows checks to be enforced that mitigate threats from malicious attacks. Carefully designed hypervisor-based security policies [50,79] allow FlyOS to monitor and validate flow of information between sandboxes such as mission commands.

Notwithstanding, we focus our architectural objectives on the following principles of design:

1) *Isolation*. Software consolidation based on the IMA concept and ARINC-653 standard requires temporal and spatial isolation between avionic functions that are critical for correct flight operation from other less-critical and non-essential services. FlyOS employs a novel partitioning approach in this context to allocate hardware resources of a centralized platform to virtualized system-level partitions. The goal is to deploy separate guest environments for locally-hosted tasks of different criticalities. Details of our design are presented in Section II-B.

2) *Extensibility*. Low-criticality sandboxes support reconfigurable and adaptable autonomous mission applications, which reduces redeployment costs. Similarly one or more real-time sandboxes allow hot-plugging of flight controllers tuned to different flight characteristics, e.g., for high maneuverability versus greater stability.

3) *Enhanced Functionality*. FlyOS targets hardware platforms with multiple cores, advanced sensors, high-speed networks, buses, and device interfaces (e.g., Camera Serial Interface), which are often unavailable in simpler autopilot platforms.

FlyOS leverages the capabilities of multicore platforms with hardware virtualization support to build sophisticated flight management software that would otherwise require separate hardware components, increasing the size, weight, power and cost overheads.

4) *Fault Tolerance*. FlyOS's sandboxed design, by virtue of *partitioning*, inherits fault-containment capabilities inherent to federated or hardware-distributed architectures, which operate on the principle of separation of concerns. FlyOS's thin hypervisor layer has a minimal memory foot-print and resides at the most privileged protection domain within each sandbox. The hypervisor (a.k.a., virtual machine monitor (VMM)) implements a run-time health-monitoring subsystem within its trusted compute base. This allows *functional* and *timing* related faults within a guest to be detected.

FlyOS's integrated and modular nature opens opportunities to incorporate system-wide redundancy at the software level. However, FlyOS does not address hardware fault redundancy due to SWaP-C restrictions.

B. System Design

The separation kernel approach enables temporal and spatial isolation of multiple guest operating systems $\{$kernel + user space$\}$ by encapsulating their run-time execution within distinct sandboxed domains or system-level partitions. To this end, FlyOS leverages hardware-virtualization to implement a partitioning hypervisor, which statically splits the hardware resources i.e. CPU cores, regions of memory and I/O devices amongst guest sandboxes. Each guest kernel directly manages an allocated set of resources at system run-time without intervention of the hypervisor (VMM). Unlike traditional consolidating hypervisors such as Xen [27] and KVM [49], the VMM is not involved in any run-time resource management on behalf of the guest and is therefore removed from the normal execution control path. This keeps the trusted code base of the VMM to a bare minimum, which is $<$ 4KB in size.

Each sandbox is isolated on mutually exclusive set of cores within the machine. Hardware-managed shadow (extended) page tables securely isolate each operating system image in non-overlapping regions of physical memory and manage guest physical to host (or machine) physical memory translations. Similarly, sandboxes are given direct access to mutually exclusive subsets of I/O devices. Interrupts are delivered directly to the guest kernel to which a device is explicitly assigned. This avoids the run-time overhead of VMM traps for individual device management.

A performance monitoring subsystem employing hardware counters provides FlyOS with the ability to predict last-level shared cache occupancy [87,88]. Such estimates are then used by static page coloring techniques to partition shared caches between sandboxes [52,91]. Consequently, a guest kernel is isolated from any temporal and spatial interference in its execution by another guest.

Explicit communication between pairs of guests is accomplished via secure, low-latency and high-bandwidth shared memory channels [58,63,78] that employ either asynchronous [77] or ring-buffer (semi-synchronous) structural semantics. A standard inter-partition communication library [78] is employed for this purpose. Mixed-criticality task pipelines are then able to span sandbox domains [42], to sustain a diverse set of avionic functions under a common flight objective. FlyOS therefore allows a scalable and capability enriched flight management system to be realized based on the design objectives in Section II-A.

1) The Prototype: Fig. 1 presents our proof-of-concept implementation in a dual-sandbox configuration. The Quest real-time operating system (RTOS) hosts timing- and safety-critical flight control functionality alongside a legacy Yocto Linux system for high-level mission control. FlyOS's separation kernel architecture allows a mutually beneficial *symbiotic relationship* to be established between the two isolated sandboxes: the light-weight RTOS gains access to the pre-existing third-party libraries, run-time frameworks, toolchains, device-drivers and various other legacy services, while the general-purpose system is empowered with hard real-time flight execution capabilities.

FlyOS's execution begins with the Quest RTOS booting up as a standalone bare-metal system. The bootstrapping process proceeds to activate the hypervisor monitor logic baked within the core image. On instantiation, the monitor partitions hardware resources among the two guest domains based on boot-time configuration parameters. A snapshot instance of the Quest kernel along with the minimal monitor code base is replicated in a distinct non-overlapping physical memory region for the Linux guest sandbox. The kernel copy is then replaced with the Yocto Linux binary image, which is thereafter launched on its pre-assigned bootstrap processor.

Depending on the sandbox configuration, one instance of Quest kernel + VMM logic acts as a bootloader for each new guest OS. Both kernels are then allowed to independently proceed with their respective normal boot procedure eventually transitioning into user-space. This marks the completion of each sandbox's initialization.

Our implementation targets multicore x86-based embedded flight computers with hardware-virtualization (VT-x) extensions [24]. For the current work, we utilize the quad-core Aero Compute Flight Hardware by Intel® [9]. Processing cores and I/O devices of the platform are asymmetrically distributed between the two sandboxes.

FlyOS allows a configurable number of CPU cores to be partitioned among guests. For our example implementation, Linux is assigned *one* physical core. This greatly simplifies the use of Linux's SCHED_DEADLINE scheduling policy, and allows relatively easy enforcement of service guarantees for mission tasks with the included PREEMPT-RT patch. In contrast, Quest is configured to work in SMP mode and uses a round-robin load-balancer to statically assign real-time flight control tasks to the *three* remaining processing cores.

Our flight control software runs as a multithreaded application for Quest, taking advantage of the parallelism supported by this core assignment. Spare CPU capacity available to the RTOS supports the addition of future timing critical tasks.

To ensure timing predictability for concurrently executing tasks, techniques are employed that handle both cache and bus contention [92].

In FlyOS, the inertial measurement unit (IMU), motors, electronic speed controllers, and serial debugging ports are exclusively allocated to Quest. In contrast, Linux is given access to the USB host controller for the camera interface discussed in Section II-C2.

Linux and Quest independently manage their assigned resources using their respective guest scheduling policies in isolated execution environments. The memory resident monitor code in each kernel is only invoked at run-time, to set up inter-sandbox communication channels and handle guest preemption timers. Such a timer is enabled for the most critical Quest sandbox, as part of FlyOS's hypervisor-level fault-detection mechanism discussed in Section II-C3.

C. Avionic Capabilities

1) Real-Time Flight Controller: For the example flight controller implementation, we take inspiration from our team's previous work [30,39] on the popular open-source autopilot: Cleanflight [5]. Cleanflight's vanilla flight control features a minimalist software stack targeted towards flight efficiency and functional robustness, reliability and performance. Control tasks are tightly coupled in a linear closed feedback loop, which employs sensor data processing with attitude estimation to regulate motor speeds for tracking a target trajectory [39]. Differential angular velocities of the motors generate net rotational torques to adjust the roll, pitch and yaw attitude about the center-of-gravity of the multicopter.

Cleanflight's responsive attitude maneuverability gives it a competitive edge over other open-source autopilots [2,4,8,14]. However, it is specifically tailored to execute as firmware on resource-constrained microcontrollers. Low-frequency single-core processing with limited memory restricts Cleanflight's ability to implement complex controllers (e.g. model predictive control) or autonomous obstacle avoidance or object tracking missions.

We empower Cleanflight's performance-critical flight control loop with autonomous functionality by retrofitting the native tasks to execute as real-time user-space threads within the Quest sandbox (Fig. 1). The main control components are identified and subsequently classified into flight safety and mission critical task-brackets based on their importance to flight control functionality and corresponding consequences on operational failure.

Table I lists each required Cleanflight task, τ_i, with budget, C_i, and period T_i. These tasks have hard deadlines equal to their corresponding periods. Sensor (IMU) and actuator (MOTOR) tasks are bound to kernel-level threads that handle real-time I/O. These threads read gyroscope and accelerometer data, and write pulse-width modulation (PWM) commands to the motors, respectively. Sensing, processing and actuation tasks form pipelines [30,41], along which data flows from inputs to outputs.

TABLE I: List of essential flight tasks in FlyOS.

Tasks	Budget (us)	Time Period (us)	Freq (Hz)	Criticality	Sandbox Assignment	Description
IMU (GYRO + ACC)	100	1,000	1,000	SAFETY	Quest	Sample sensor data
RX	20	20,000	50	MISSION	Linux + Quest	Specify target commands.
ATTITUDE	20	10,000	100	SAFETY	Quest	Calculate current attitude of copter
PID + MIXER	10	2,000	500	SAFETY	Quest	Attitude controller + throttle mix
MOTOR	1,000	2,500	400	SAFETY	Quest	Process PWM commands
BLACKBOX	20	2,000	500	BACK-GND	Linux + Quest	Log flight + mission data

Quest uses a variant of rate-monotonic scheduling (RMS) [29] algorithm by defining a virtual CPU (vCPU) abstraction as a schedulable entity [34] on top of a physical CPU (pCPU). Threads and their corresponding pipe wrappers are directly mapped to vCPUs, which are then mapped to pCPUs. This two-level scheduling hierarchy guarantees each task τ_i to execute for C_i time units every T_i when runnable [57].

In accordance with RMS, vCPUs are assigned static priorities based on their time periods: highest priority is given to the smallest time period and vice versa. Quest executes interrupt service routines in a separate real-time thread context with a time period inherited from its user-level counterpart. This allows I/O interrupts to be handled at the correct priority of the task issuing the request thus enabling real-time management and deterministic accounting of CPU clock cycles for each device interrupt. The scheduling subsystem therefore guarantees temporal isolation between flight control threads.

Fig. 2 shows the distribution of control functionality between Yocto Linux and Quest in our dual-sandbox setup. *Timing* and *safety-critical* control threads are allocated to Quest while *mission-critical* functionality is mainly ported to Linux. For RX (Table I), a setpoint generator (Process-1) in Linux communicates across an asynchronous shared memory pipe-buffer with a light-weight thread in the RTOS acting

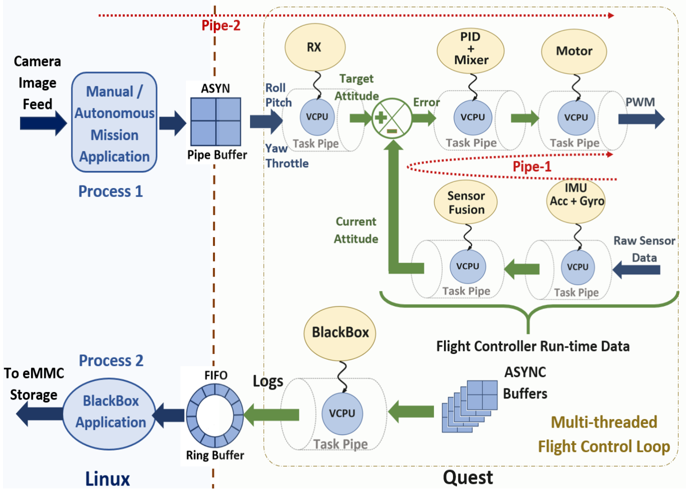


Fig. 2: FlyOS's software-distributed flight-control model with threaded tasks.

as a receiver gateway. Similarly, a background logger thread (BLACKBOX) receives flight data (Process-2) in Linux from the corresponding sender-stub in Quest. A FIFO circular-buffer transfers the time-ordered history of flight logs, which are saved to permanent file storage in Linux.

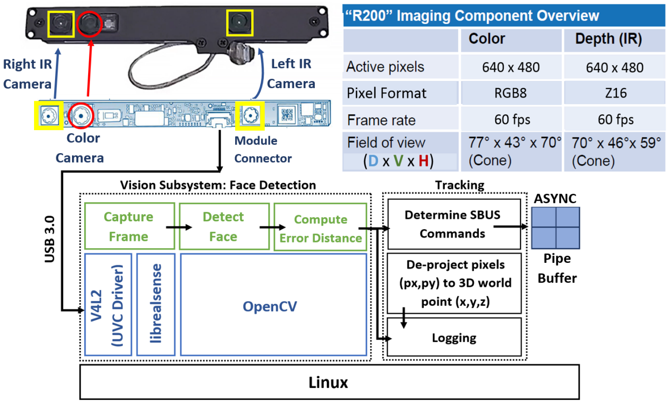
Asynchronous pipe buffers are implemented using Simpson's four-slot algorithm [71,77], which ensures data freshness and integrity. The control loop needs to keep track of the most recently sampled sensor values and target trajectory updates. Pipe-buffers therefore allow accurate data-flow and low-latency attitude control in response to the most up-to-date current and required state of the drone.

We identify two task pipelines within the main flight control loop:1) *intra*-sandbox **Pipe-1**: IMU → MOTOR and 2) *inter*-sandbox **Pipe-2**: RX → MOTOR. Pipe-1 comprises 1) IMU sampling and processing, 2) sensor fusion based on a complementary-filter [74,90] for ATTITUDE estimation, 3) a PID+MIXER that transforms the error between actual and target attitudes into control signals mixed with throttle, and 4) a MOTOR thread that generates PWM waveforms for the multicopter's motors.

Pipe-2 involves the mission task in Linux (Process-1), which computes target attitude and thrust set-points based on the application's flight objective. The reference commands are then sent to the gateway receiver (RX), which forwards the roll, pitch and yaw targets along the feed-forward path of the loop, shared with Pipe-1 (refer to Fig. 2). FlyOS envisions a *criticality-aware distribution* of tasks among guest domains. Task pipelines are thus composed on the basis of each task's role and importance in the perception, planning and control of the drone.

2) Autonomous Vision Subsystem: We implement vision navigation in Linux for our mission application. Linux supports a rich collection of USB video-class drivers for interfacing with hardware cameras. Corresponding libraries and APIs provided by Video4Linux (V4L), OpenCV and CUDA toolkits enable efficient development and testing of autonomous perception applications using state-of-the-art image capture technology.

For autonomous mission control, we design a simple face-image detection and tracking application that relies on librealsense [11] and OpenCV for capturing and processing camera images. We utilize a USB3.0 Intel RealSense (R200) [15] camera module, which features a 3D imaging system that is capable of providing color and depth video streams. Fig. 3 depicts individual task components of our vision framework along with the intrinsic characteristics of the R200 camera. Algorithm 1 details our application loop from image frame capture to generation and communication of mission control commands (setpoints) to the flight controller executing in Quest.



"R200" Imaging Component Overview

	Color	Depth (IR)
Active pixels	640 x 480	640 x 480
Pixel Format	RGB8	Z16
Frame rate	60 fps	60 fps
Field of view (D x V x H)	77° x 43° x 70° (Cone)	70° x 46°x 59° (Cone)

Fig. 3: FlyOS's vision subsystem (Process-1) with the RealSense R200 Camera.

OpenCV supports a ready-to-use face detection algorithm based on the Haar-feature cascade classifier [83] approach. Known for its speed and simplicity, its one of the most popular algorithms still used today for frontal-face detection with high accuracy and image-scale invariance. We utilize OpenCV's built-in repository of pre-trained parameters for the cascade classifier composed of 22 total stages and a sliding window of 20x20 pixels (px). An integrated classifier function (Line 10) detects faces in each frame captured by the color camera at run-time and returns a bounding rectangle.

We calculate the center coordinates (Line 12) of the face to determine an offset distance from the frame center in 2D pixel coordinates. These are forwarded to a linear algorithm, which computes the required direction of movement for the multicopter as well as the target set-points for the pitch and yaw rotational axes to minimize the offset and track the detected face (Lines 13–14).

Our algorithm enables configuration of rate of change of set-point commands in each axis of rotation ($rate\_\{dPitch, dYaw\}$). This allows us to affect the sensitivity and precision of mission control per unit of error distance, which in turn impacts responsiveness of flight

Algorithm 1 Image Detection and Tracking

```
Require: Haar-classier pre-trained XML file containing stage thresholds and
    filter weights: $haarcascades/haarcascade\_frontalface\_alt.xml$
Require: $< cv :: Rect > faces$ /*array to store detected face(s)*/
Require: $rate\_\{dPitch, dYaw\}$ /*rates of change of command*/
 1: async_chan = $create\_shared\_memory$ (ASYNC_TYPE)
 2: $ctx = r200\_create\_context()$
 3: $r200\_enable\_stream(ctx, \{\text{COLOR}, \text{DEPTH}\})$
 4: while true do
 5:     /* Capture and retrieve image frame */
 6:     $data = get\_raw\_frame\_data()$ /*for enabled image streams*/
 7:     $frame = to\_openCV\_matrix(data)$ /*frame vector to matrix*/
 8:     $\{px_0, py_0\}$ = $\{\frac{frame.cols}{2}, \frac{frame.rows}{2}\}$ /*frame center*/
 9:     /* OpenCV: detect face */
10:      cv::CascadeClassifier.$detectMultiScale(frame,faces,$
             min=200×200,max=1000×1000)
11:     /* Estimate distance offset and generate command */
12:      $\{fx_c, fy_c\}$ = $\frac{faces[0].width}{2}, \frac{faces[0].height}{2}$ /*1st face's center*/
13:     dPitch = $rate\_dPitch$×($fy_c$ - $py_0$) /*pitch-up distance*/
14:     dYaw = $rate\_dYaw$×($fx_c$ - $px_0$) /*yaw-right distance*/
15:     /* command in correct format */
16:      $commandData$[Roll,Pitch,Yaw, Throttle] = F({0, dPitch, dYaw, 0})
    /*F(command) is the conversion function specific to the flight controller*/
17:     /* Write to shared memory */
18:      $write\_shared\_memory$(async_chan , $commanData$)
19: end while
```

control to target commands. Data is converted to a compatible RX format (Line 16) for the gateway thread in Quest and sent across shared memory (Line 18). For the Cleanflight autopilot, set-point values are packaged as SBUS [68] protocol frames before transfer.

We use the *depth* stream to de-project the offset distance in pixels into a real-world displacement of the face-image from the camera center, in meters. This allows us to convert between different coordinate systems, and log the multicopter's angular movement against the ground truth trajectory of the image.

SCHED_DEADLINE is used to schedule the vision process allowing mission commands to be generated with sufficient predictability. Our design also caters for face occlusions for a limited time-horizon. We configure a threshold time-out value before the mission is aborted. This allows configurable tolerance against occasional occlusions.

We note that this work does not focus on performance comparisons between different real-time face-detection frameworks. Instead the OpenCV implementation serves as a model example of showcasing the autonomous capability and practical feasibility of FlyOS's architecture. Mission tasks in Linux are able to effectively communicate commands to the flight controller tasks over a low latency inter-sandbox channel interface. FlyOS therefore ensures predictable autonomous control with bounded worst-case end-to-end latencies. Section III validates FlyOS's autonomous tracking capability.

3) Fault-Tolerance Subsystem: FlyOS's virtualized sandboxed architecture lends itself to support high-confidence avionic systems. The partitioning hypervisor prevents access to the separate memory spaces and resources assigned to remote guests. FlyOS's distributed system-on-a-chip design attempts to contain faults within separate sandboxes, similar to how federated architectures isolate faults in separate hardware. Our fault tolerance subsystem enables:

1) *Software component level tolerance* for failures within user-space applications: A functional or timing based failure is detrimental to the safe operation of the multicopter if it directly affects the real-time and safety-critical behavior of the flight control loop. FlyOS allows flight controller redundancy across different sandboxes and implements efficient controller hand-off mechanisms. In this work, we focus on faults within the critical MOTOR task.

FlyOS uses heartbeats to capture a class of functional and timing failures, which jeopardize the progress of critical tasks. For example, if the motor task fails to generate a heartbeat by a certain time, this could jeopardize the control of the drone. Loss or delay of a heartbeat triggers the activation of a failover controller to maintain flight.

2) *Sandbox level (or kernel level) tolerance* for failures impacting the entire guest OS domain: Such failures often involve kernel memory corruption or other types of malicious kernel attacks initiated by external non-certified third-party services. A local copy of the VMM in each guest sandbox allows for sandbox-level redundancy. The VMM is able to quarantine a malicious guest and even re-instantiate or duplicate an entire guest partition with its corresponding application stacks, to replace the corrupted guest instance. I/O device hand-off between sandboxes with replica-coordination mechanisms is implementable in FlyOS's monitor logic. Failover standbys will be activated while the original sandbox is recovered, thereby providing an online and effective way to handle such system-level faults. We reserve further discussion on this topic for future work.

We propose a unified fault-detection mechanism, which operates in the most-secure mode (root-mode) of the system. The VMM keeps a runtime health-check of the critical tasks within its respective sandbox through timer-initiated guest preemptions (VM-exits). x86 hardware-assisted virtualization timers called VMX-preemption timers are leveraged for this purpose. The timer operates at a frequency proportional to the hardware time-stamp counter (TSC) [21] available to each core of the processor. This detection mechanism has the benefit to be agnostic to functional, event or timing related failures within the system.

Application task failures that compromise the correctness of real-time flight control are attributed to factors such as delayed mission commands, incorrect tuning of the PID controller, motor runaway or stale motor updates. Due to the closed loop nature of the flight control, it is possible for a fault originating in a single thread to propagate through the entire application. Fig. 4 enumerates the steps involved in the workflow from fault-detection to recovery for the dual-sandbox system.

The MOTOR task is instrumented to generate periodic heartbeat messages (Step-1). A VMX-preemption timer is enabled within the VMM logic of the Quest sandbox. It counts down during the execution cycles of the guest, in non-root mode, based on a configured timeout value. This directly controls the fault-detection latency. On expiration, a low-cost VM-exit [19,76] is triggered causing a soft-trap to the hypervisor (Step-2). If the monitor observes a heartbeat message inconsistency, a system mode change is initiated (Step-3) following a distributed recovery response (Step-4&5). Consequently, the faulting flight controller is marked as compromised (depicted

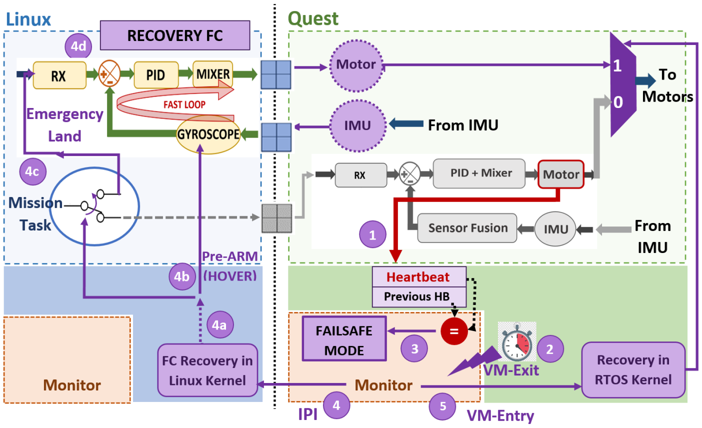


Fig. 4: Flight controller fault tolerance: detection and remote recovery.

TABLE II: Recovery path latencies for **pipe-delayed** in Linux.

Tasks: Linux (Detection → Recovery)	**Min**	**Average**	**Max**
(Step-4a) IPI RX: Kernel → **Userspace** (ms)	0.004	0.005	0.007
(Step-4b) IPI RX Userspace → **Mission Process** (ms)	24	43	81
(Step-4c) Mission Process → **Backup-FC: RX** (ms)	19	29	35
(Step-4d) Backup-FC: RX → **Shared Memory** (ms)	0.005	0.075	0.4

in grey in Fig. 4) and all corresponding threads terminated.

Two *proxy* real-time tasks are activated, to control the sensor (IMU) and actuator (motor) devices (Step-5). This retains predictable safety-critical I/O control within Quest. The local monitor sends an inter-processor interrupt (IPI) to trigger the remote recovery pipeline, in parallel, within Linux (Step-4). A kernel module listens for the IPI and acknowledges receipt with an interrupt-handler routine (Step-4a).

Two userspace task pipelines (Step-4b) are then launched: 1) fast-loop response (**pipe-fast**), which pre-arms the backup flight controller (Linux port of vanilla Cleanflight) to transfer simple hover commands to the virtual device *proxy* interfaces in Quest, and 2) delayed response (**pipe-delayed**) employing the Linux mission task to initiate relatively more complex maneuvers such as radio over-ride to return the multicopter to base or force an emergency-land.

Table II shows a preliminary set of latency measurements for each step of the online recovery within Linux. The timing measurements incorporate processing, scheduling and transition delays, which may cause the drone to experience motor downtime. To avoid crashing, we activate a first-response (pipe-fast) recovery, which complements the delayed response.

To keep execution costs low, we ported vanilla Cleanflight as a backup-controller supporting only the most critical functionality of the fast-loop. Asynchronous communication channels between the failover controller and real-time device gateway threads ensure timely transfer of commands. Despite SCHED_DEADLINE scheduling optimizations, the overheads experienced by Linux indicate it is only really suitable as a temporary backup until the primary hard real-time controller is restored.

III. Evaluation

We evaluate FlyOS for three different scenarios: 1) *Manual radio control* with attitude stabilization in the presence of an external disturbance, 2) *Autonomous mission control* with face-image detection and tracking, and 3) *Failover flight control* to recover from a critical actuator fault. We conduct hardware-in-the-loop (HIL) experiments and latency-benchmark simulations. Our testbed setup common to all HIL experiments is presented next.

A. Experimental Setup

1) Hardware: Figure 5 shows our custom built S500 (500mm) quadcopter mounted at the center of a 3-axis mechanical gyroscope, called the *BirdCage* [39]. The three orthogonal gimbal rings (annotated in the diagram) allow the drone to freely rotate about its roll, pitch and yaw axes thus enabling repeatable attitude adjustments in a controlled environment. We also mount a flat passive screen (40” x 30”) in front of the drone to project images for vision-based tracking experiments.

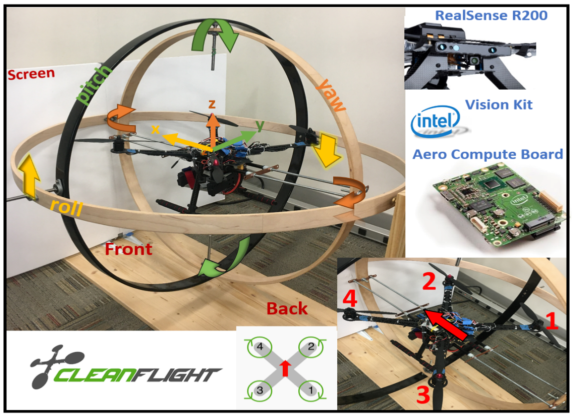


Fig. 5: BirdCage testbed for real-world experiments. The quadcopter motor configuration is enumerated at the bottom-right.

The quadcopter's frame is fitted in an X motor configuration for symmetrical mass distribution in all three rotation axes. This ensures 100% motor output performance [59] with 4 EMAX 935kV brushless DC motors.

We host FlyOS on Intel's purpose-built UAV developer kit featuring the Aero Compute Board and a complementary vision accessory kit [10], which includes the RealSense R200 camera module.

Fig. 6 shows a block diagram layout for the hardware modules integrated into the Aero Compute Board. This board features 4GB RAM, a quad-core Intel Atom x7-Z8750 processor, a GPIO expander, a 6 degrees-of-freedom BMI160 IMU (for 3-axis gyroscope + accelerometer), and an Altera MAX10 FPGA. The processor nominally runs at 1.6GHz and supports Intel VT-x virtualization technology. The FPGA generates PWM motor signals from commands issued by the Atom processor. A RadioLink R9DS receiver connects to the GPIO expander to receive raw SBUS commands required for manual radio control. We deploy FlyOS on the Aero Compute Board with the task distribution shown in Fig. 2.

2) Performance Metrics and Settings: For the **BirdCage** experiments, we record attitude variation profiles of the quad-

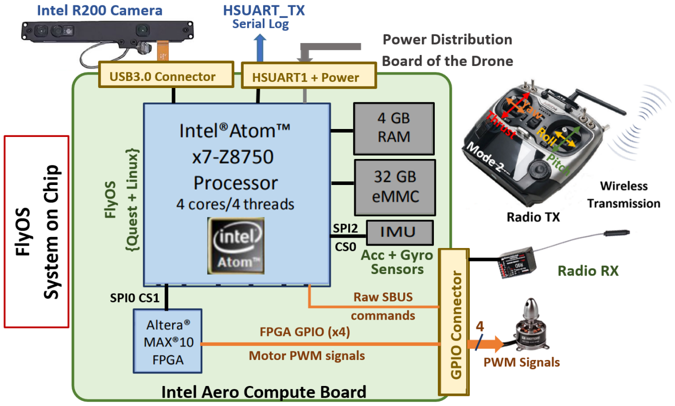


Fig. 6: Aero Compute Board with manual radio control setup.

copter over time in response to an appropriate stimulus. We measure the *response time* to achieve a steady-state target attitude with an error-band of $\pm 0.5°$ (shown as horizontal red lines in our plotted results). This allows us to account for data imprecision and any inherent imperfections in the drone hardware or positioning of the payload. We additionally compute *error* statistics for each flight, to quantify the impact on the accuracy of flight control. Results are averaged over at least 3 flights.

We design **microbenchmarks** to draw conclusions about the worst-case observed *end-to-end (E2E) delays* along critical flight control paths. We now present FlyOS's performance results for each of the three flight scenarios described above.

B. Manual Radio Control

FlyOS is compared with vanilla Cleanflight (CF) firmware leveraging manual radio control capability. The mission process in FlyOS's Linux sandbox reads raw SBUS radio input from the GPIO connector of the Aero Board and sends processed SBUS commands to the RX gateway thread in Quest.

1) Setup: Vanilla Cleanflight is flashed on an SPRACINGF3 [31] flight microcontroller and installed on the drone. Featuring the STM32F3 processor, it offers native support for the original flight stack. A TX/RX pair is used to ARM the drone and transfer throttle and attitude target commands to the autopilot. Cleanflight is configured to run the same subset of critical flight control tasks as FlyOS (Section II-C). The main loop-time for the *fast-loop* is set to the maximum supported frequency of 1000Hz, which represents the best response time performance [39] on the microcontroller platform. PID constants for both FlyOS and Cleanflight autopilots are tuned to yield stable flight control behavior with minimal response time.

2) Results: For the BirdCage experiment, our steady-state target is set at a horizontal hover $(0° \pm 0.5°)$ in the **Roll** axis. A transient step-input attitude disturbance is introduced in the *Roll-Right* direction by displacing the corresponding axial ring of the BirdCage by 15°. The quadcopter is then allowed to stabilize to target hover. Due to the symmetrical nature of the motor + mixer configuration, Roll axis proves sufficient to showcase the attitude correction behavior. Fig. 7 shows that FlyOS's integrated architecture yields the same control integrity and functional correctness as the vanilla firmware. FlyOS, however exhibits a slightly better response time of 10.87s compared to 11s of Vanilla-CF despite running a more complex software stack involving two guest OS domains. Smaller peak-amplitude oscillations by FlyOS lead to lower mean error values reported in Table III. FlyOS's predictable task execution therefore exhibits higher accuracy of control with a timely and precise response from the motors. This manifests as lower magnitude of under- and over-shoots from the hover target.

TABLE III: Error statistics of the flight profiles in Fig. 7.

Autopilot	Mean Absolute error	RMSE
FlyOS	3.85°	5.18°
Vanilla-CF: 1000Hz	4.93°	6.53°

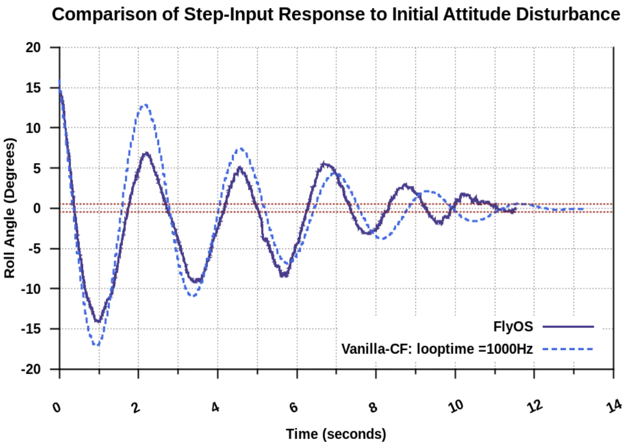


Fig. 7: Roll-Right attitude correction profile.

E2E latency is reported in Fig. 8 for the two task pipelines (**Pipe-1** and **Pipe-2**) within FlyOS's flight control loop. Vanilla-CF latencies provide a baseline reference. FlyOS performs 59% and 20% better for Pipe-1 and Pipe-2, respectively, in the worst-case. This ensures low-latency responsiveness and expedited recovery from anomalous attitude shifts. FlyOS takes advantage of the higher clock rate and powerful processing capabilities of embedded multicore platforms to ensure predictable flight behavior. Low E2E pipeline latencies are crucial for high frequency mission control to track a trajectory target in real-time.

C. Autonomous Mission Control

We now demonstrate how FlyOS supports autonomous mission control. Our sample mission requires the quadcopter to use face detection to locate and track a target image, which is projected onto a 2D screen (Fig. 5).

1) Setup: The RealSense R200 camera is mounted at the front of the quadcopter, such that the image plane [1] center is aligned to the middle of the screen. The image plane is set to a resolution of 640×480 pixels, with the middle of the screen having the origin coordinates, $x_0 = 0, y_o = 0$.

The autonomous flight objective is three-fold: 1) detect an image of a face of size 10x10 pixels (target) on the screen, 2) determine its horizontal or vertical displacement from the origin, and 3) adjust the drone's pitch or yaw attitude in the direction of the target, to accurately align the center of the camera plane with the projected image. In case of a moving target, the aforementioned steps are repeated every time the target location updates.

We interpret a static image to be equivalent to a step input target signal, whereas a moving image corresponds to a ramp input signal to the flight controller. The BirdCage is placed at a fixed distance from the screen (Fig. 5), which we measure using the native DEPTH stream from the IR-sensors in the Realsense module. We record updates in the horizontal (x)

[1] We refer to image and camera plane interchangeably.

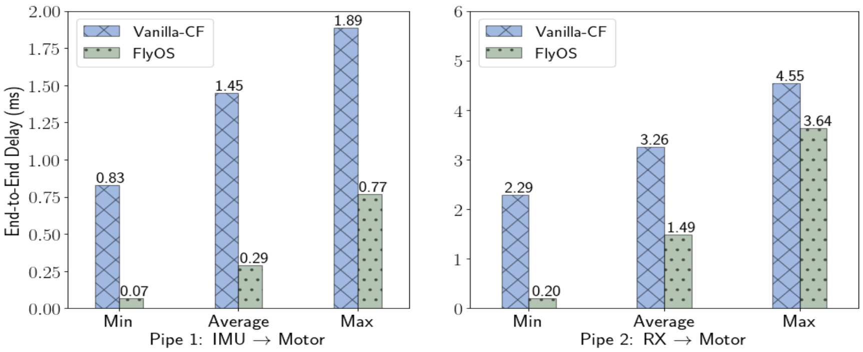


Fig. 8: E2E latencies for two critical flight control paths within FlyOS. Vanilla-CF provides a reference for comparison.

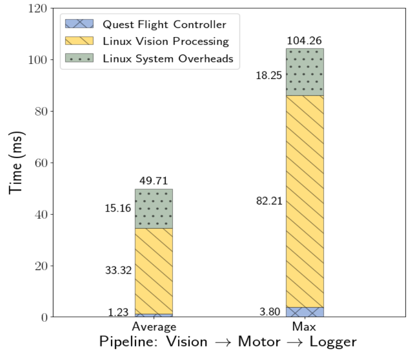


Fig. 9: Round-trip times for vision pipeline with constituent task latencies.

and vertical (y) displacement of the center of the image-plane over time in meters, as the drone rotates in the yaw and pitch axis, respectively. This distance is then converted into an angular rotation in degrees using trigonometry. A similar technique is used to record the ground truth for the target's movement in a pre-programmed trajectory for the duration of each experiment.

2) Results: **Pitch** and **Yaw** attitude adjustment profiles in response to step- and ramp-input stimuli are shown in Fig. 10 and 11 respectively. Target image location is restricted to the positive y-axis of the screen for *Pitch-Up* experiments and positive x-axis for *Yaw-Right* experiments. Corresponding error and response time values are reported in Table IV. Steady-state *alignment* and root-mean-square *tracking* error is measured for statically positioned and moving targets respectively.

For step-profiles shown in Fig. 10, the drone eventually settles to an accurate steady-state, aligning with the image center within ±0.5° error threshold in both axes. The response times for pitch and yaw are also within 0.9s of each other. The transient control response however, shows higher magnitudes of over- and under-shoots and sharper corrections in the pitch axis compared to yaw's smoother and heavily damped trace. Similarly, ramp profiles in Fig. 11 show that the drone is successfully able to track the target with a root-mean-square error of 1.19° and 0.5° in pitch and yaw axis respectively. These fall within the boundary of the 10px by 10px target-image, which translates to the drone's angular span of ±1.25° from the image center. We again observe that compared to pitch, yaw response exhibits a greater accuracy of control (lower RMSE) and smaller transient lag leading to a lower average response time.

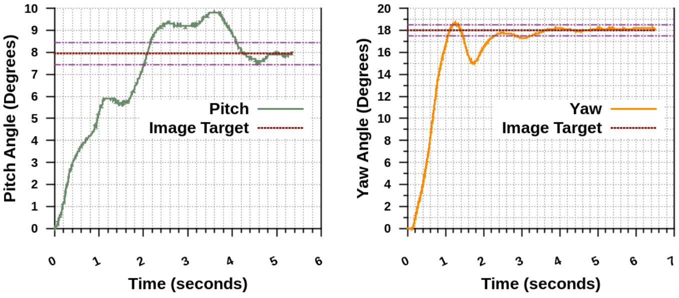


Fig. 10: Detecting a static image: step-response in Pitch and Yaw axis.

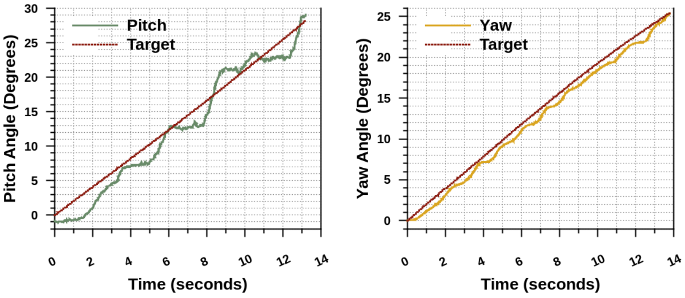


Fig. 11: Tracking a moving image: ramp-response in Pitch and Yaw axis.

TABLE IV: Response time and error statistics for vision experiments.

Parameters	Static Image		Moving Image	
	Pitch	Yaw	Pitch	Yaw
Mean Steady-State (S.S) Error (deg)	0.16	0.15	-	-
Root Mean Square (RMS) of Total Error (deg)	-	-	1.19	0.50
Avg. Response Time to reach Target Angle (s)	4.10	3.21	1.26	1.22

This performance difference results from the hyper-sensitivity of the pitch axis to changes in airflow dynamics, ground-effect and external environmental forces like gravity [53,75]. As the quadcopter pitches up towards the target, downstream turbulence produced by the front two propellers interferes with the rotation of the rear propellers. This prop-wash effect is largely absent in yaw rotations with all 4 propellers operating in the same horizontal plane. We also note that the weight of the hardware payload, including the battery, is predominately distributed along the pitch axis. The resultant center-of-gravity vector therefore has a direct impact on pitch sensitivity to slight changes in motor thrusts. We thus observe a less damped transient response, which is possible to improve with a more finely tuned PID controller. Despite the differences in performance between the axes, FlyOS exhibits efficient, autonomous detection and tracking behavior for both static and moving targets, with reasonable accuracy and responsiveness.

FlyOS's vision detection pipeline spans across Linux and Quest sandboxes. We measure round-trip latencies of the autonomous pipeline: Image detection & tracking mission application (*Linux*) → RX stub processing (*Quest*) → PID+MIXER

(*Quest*) → MOTOR Update (*Quest*)→ BLACKBOX Logger (*Linux*). Average and worst-case end-to-end latencies of the entire workflow and constituent software modules are reported as stacked bar graphs in Fig. 9. Vision processing in Linux includes frame retrieval and processing delays, as well as object inference delays. Inter-sandbox communication delays and SCHED_DEADLINE scheduling overheads are aggregated as "System Overheads". Quest delays involve the execution times of flight control tasks along **Pipe-2**. These tasks read and process vision commands sent via shared memory, and generate corresponding PWM commands. On average, our vision application is able to maintain a frame processing rate of ≈30 fps. Even in the rare worst case, the processing rate still results in the drone tracing an accurate tracking trajectory as seen in the attitude profiles.

Although our application employs a relatively simple object detection algorithm, it serves as proof-of-concept for FlyOS's ability to support real-time autonomous missions.

D. Comparison with Intel Drone

To further motivate our architectural framework, we compare the communication overheads of FlyOS and the Intel Ready-to-Fly (Intel-RTF) [7,13] drone. The Intel-RTF drone runs the Ardupilot [2] flight controller hosted on the Pixhawk [35] companion microcontroller board.

1) Setup: The Intel-RTF drone is a pre-assembled quadcopter, which supports programmable UAV applications and mission control. The platform is a dual-board solution to flight management. The main compute engine comprises the Aero Compute Board connected via an HSUART (high-speed universal asynchronous receiver-transmitter) serial bus to the Pixhawk flight controller hardware. Pixhawk offers native support for Ardupilot's flight stack, which ships as a binary with the Intel-RTF drone. We flashed the Compute Board with the Ubuntu Linux 16.04 operating system, to develop and host our microbenchmark for measuring communication latencies.

Communication over the serial link is managed by the MAVLink-router soft-service within Linux. MAVLink commands [36] and corresponding acknowledgment (ACK) messages are transferred between high-level mission applications in Linux and Ardupilot's control loop executing on the Pixhawk. The control loop logic within the flight stack is split into two parts [38]: critical flight controller tasks (termed the fast-loop) and non-critical application tasks, including MAVLink message retrieval, processing and ACK generation. Priority is given to the fast-loop, which executes controller sub-tasks in a sequential manner. Remaining time of the control loop is then distributed between application tasks that are scheduled in a best-effort preemptive manner. In contrast, all threads within FlyOS's critical flight control loop, including RX processing, are managed by a real-time scheduler that guarantees each task's (τ_i) execution time budget (C_i time units) every time period (T_i time units).

We measure the round-trip latencies of the MAVLink communication protocol using DroneKit's python API [23], to send "set-yaw-attitude" commands to the flight controller and receive corresponding ACK messages. Similarly for FlyOS, we use our vision-detector Yocto Linux application to transfer yaw commands to the flight controller executing in Quest, using asynchronous shared memory communication. For every message sent, an ACK message is received, timed and logged on the Linux side.

TABLE V: Communication overheads in federated & FlyOS architecture.

Communication protocol	**Min**	**Average**	**Max**
Asynchronous Shared Memory (ms)	0.0004	0.00052	0.0091
MAVlink on UART-serial (ms)	4.13	9.99	301.54

2) Results: Table V presents our results averaged over 2000 transferred messages. As shown, the MAVLink protocol incurs a significant delay.

We also note that FlyOS's shared memory inter-sandbox communication exhibits lower overhead latencies than inter-partition communication based on a data-distribution service (DDS) network as evaluated by Pérez et al. in [62]. The authors analyze an ARINC-653 compatible DDS communication link between two MaRTE [12] RTOS virtualized partitions, hosted by the XtratuM [33] hypervisor on a multicore x86 platform. Their results show average round-trip latencies of 100s of microseconds for simple data transfers. Such delays result from the ARINC-653 virtual network service, DDS middleware stack, hypervisor-based processing of interrupts and other operating system overheads.

With reduced data transfer costs, FlyOS allows mission tasks the flexibility to execute at high frequencies, while incurring minimal delays for communicating target commands to the flight controller. It thus ensures agile and responsive flight control with enhanced maneuverability.

E. Failover Flight Control

We next study the performance impact of the fault identification and failover subsystem. We measure the latencies of Detection→Recovery pipelines within each guest OS. An artificial fault is injected within the motor-update (MOTOR) thread, which sends stale commands to the motors after the flight controller has been operational under *Normal* mode for some time. This causes the heartbeat messages sent to the hypervisor to stall after the fault is encountered, resulting in a *Fault-Tolerance* system mode switch.

We utilize vanilla Cleanflight's fast-loop operating at 1000Hz frequency (looptime=1ms) as our ported failover controller. The VMX-preemption timer for Quest's bootstrap processor (BSP) core within the Aero Compute Board is configured to expire periodically at intervals of 2ms (500Hz). This defines our worst-case time bound for fault-detection. Each sandbox's corresponding recovery response is tracked in parallel based on the steps enumerated in Fig. 4. End-to-end delay statistics are presented in Fig. 12. The measured worst-case recovery time to reach the hover state in Quest is 0.77ms. This represents the duration from the system mode change (Step-3 in Fig. 4) to the first set of valid hover commands sent to the motors (Step-5).

For Linux, the measured worst-case end-to-end recovery time for the longer pipeline (**pipe-delayed**) (Steps:4-4d) is 84ms. This is less than the total sum of the worst-case constituent step latencies of the pipeline as shown in Table II. The pipeline thus activates the emergency landing mode for the backup flight controller within the practical latency upper bound. Comparatively, the first response pipeline (**pipe-fast**) is activated with a maximum delay of 0.41ms. This latency comprises the combined delays for Step-4a: 6.5μs and Step-4d: 0.4ms, and falls well within the upper bound latency of 1ms (1000Hz) for the fast-loop vanilla controller. PWM hover commands are therefore sent to the shared memory channel with a lower latency than the pipe-delayed pipeline. The motor proxy task in Quest reads the asynchronous channel on activation and transfers processed commands from the Linux-side flight controller to the motors. This allows the quadcopter to stabilize until emergency landing is activated at a later time.

A comparative *primary-backup* partitioned system built for helicopters by Jeong et al. [37] reports the first response time to be 11ms using a hardware-in-the-loop simulation environment. This is at-least 90% slower than FlyOS's pipe-fast failover hover response. A primary reason is the temporal multiplexing approach taken by the authors for scheduling virtualized primary and backup partitions onto shared hardware resources. FlyOS's hypervisor allows each sandbox partition to directly and independently manage its local resources, thus allowing activation of parallel recovery pipelines. This leads to a more timely first response for a fault that originates in one or more critical flight control tasks.

Fig. 13 shows the real-world attitude response profile of the quadcopter in the BirdCage. For normal mode operation, the primary flight controller within Quest tracks a static image along the x-axis of the screen with corresponding yaw-right rotations. We observe a failover response time of 2.51s from when the motor fault is detected to the time when the quadcopter achieves a stable hover under the control of the backup Cleanflight. This experiment provides a practical latency bound to regain stable flight, when the quadcopter is subjected to physical constraints, considering factors such as the rotational inertia of the motors and rotor drag. We note that during the dynamic hand-off between the primary and backup flight controllers, the motors do not exhibit any visible downtime but only a change in the update frequency of corresponding angular velocities. This example fault-tolerance subsystem shows FlyOS to be capable of maintaining safe failover flight control.

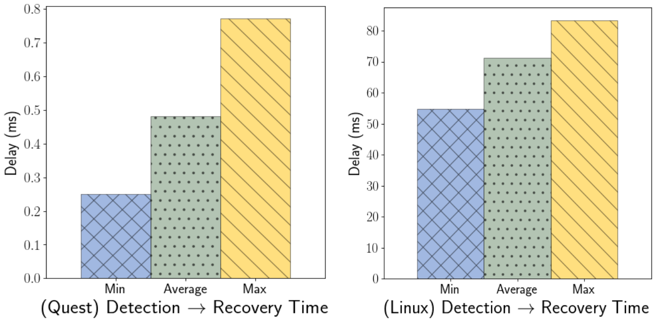


Fig. 12: E2E latencies from fault detection to flight recovery within Quest (*left*) and Linux (*right*).

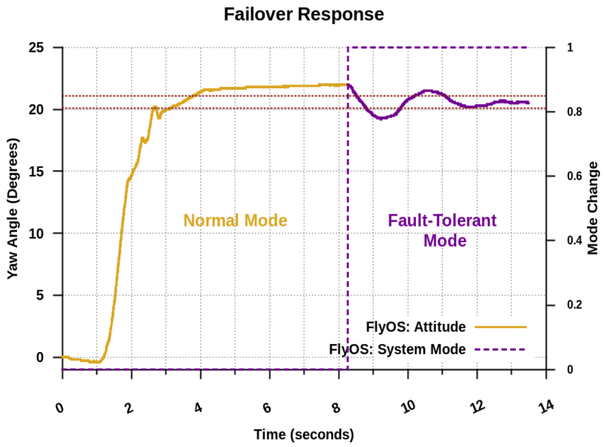


Fig. 13: Static image detection (Normal Mode) and Fault recovery hover stabilization (Fault-Tolerant Mode) with fault injection at 8.27s.

Our current implementation relies primarily on Linux to be the warm standby sandbox for failover flight control. In an effort to reduce the response time even further, we propose using an RTOS sandbox for real-time failover recovery. This would allow us to overcome the timing shortcomings of Linux, and implement a real-time safety-critical backup flight controller. However, this warrants further investigation and is left for future work.

IV. Related Work

Multicopter flight management relies primarily on federated architectures for functional segregation. Hardware segregation of flight control stacks from mission applications is provided by Cube Autopilot's co-processors [16], Intel's Ready-to-Fly (RTF) Drone [13,40,80], Qualcomm's digital signal processors (DSPs) [1,6,17,22], and various companion board solutions [3,43,56]. In each of these cases, timing and functional failures are isolated from tasks running on remote hardware.

Other researchers have focused on the security of mission components used in a federated flight management architecture. For example, Klein et al. [50] use seL4 to separate trusted from untrusted software in separate VMs of a mission computer that is distinct from the flight control hardware.

However, to reduce size, weight, power and cost (SWaP-C), the research community has recently considered a software-based integrated modular avionics (IMA) approach to flight architectures [28,70,85]. IMA in UAVs takes its inspiration from the commercial aerospace domain led by Airbus and Boeing [46,84], to employ temporal and spatial partitioning techniques in compliance with ARINC-653 software development standards.

Several mechanisms have emerged that target partitioning at either the application [48], kernel or hypervisor level of the consolidated flight management system. LynxOS-178 [51,54] is a small partitioning kernel, which establishes encapsulated domains for applications, and schedules them on shared hardware in dedicated timeslots. Jo et al. [47] define an OS abstraction layer (OSAL) for Linux and RTEMS, along with an ARINC-653 core layer tailored for small civilian UAV applications.

Other kernel-level partitioning approaches [45,69] extend existing operating systems with ARINC-653 API support. In these approaches, user-level partitions are typically multiplexed on processing cores, resulting in frequent context switching, and potentially increased system overheads. Lack of temporal and spatial isolation in the shared interrupt handling subsystem for I/O devices results in unpredictable worst-case execution times at the task level. This negatively impacts the timing predictability of flight control, and responsiveness of mission control.

In contrast, research in virtualization technology for avionics has approached IMA's partitioning requirement at the system-level by employing consolidating hypervisors. These allow multiple operating systems to run simultaneously as virtual machines on shared flight hardware. PikeOS [18] and AIR [32] are two micro-kernels with support for a virtualization layer responsible for partitioning of resources between hosted guest operating systems. These approaches to IMA however have only been deployed in spacecraft applications [25,89]. State-of-the-art multicopters, on the other hand, employ traditional hypervisors like Xen [81,82], VMware and VirtualBox [44]. These offer support to host Linux VMs extended with ARINC-653 standard APIs. Linux however lacks hard real-time support for I/O interrupt scheduling [93], which is needed for sensing, processing and actuation tasks in a flight controller.

Pérez et al. [61,62] integrate DDS (data distribution service) with ARINC-653's port-based communication. The authors validate their approach by implementing RTOS-based publisher-subscriber partitions on the Xtratum [33] hypervisor. The inter-partition communication is presented as a general avionic solution applicable to all IMA based flight management systems.

Contrary to the current virtualization solutions, FlyOS presents a *partitioning hypervisor* approach tailored towards efficient flight control for multicopters. To the best of our knowledge, FlyOS is the first consolidated avionic system to statically partition hardware resources between guest sandboxes that remain under the direct management of their respective OS kernels at run-time. Consequently, the system incurs minimal operational overheads. FlyOS's separation kernel thereby achieves spatial and temporal isolation in the context of Integrated Modular Avionics for multicopters. Additionally, mixed-criticality avionic services mapped to different sandboxes are able to communicate with low latencies using shared memory mapped into user-level address spaces.

V. Conclusions & Future Work

This paper presents FlyOS, an integrated modular avionics (IMA) framework for next-generation multicopter flight management systems. FlyOS employs a partitioning hypervisor to statically partition hardware resources among virtualized guest OS domains or sandboxes. Our prototype implementation hosts a built-in RTOS (Quest) with a legacy feature-rich Linux system in a dual-sandbox configuration. A real-time safety-critical flight controller ported to Quest communicates via shared memory with autonomous mission critical application services in Linux.

FlyOS guarantees temporal and spatial isolation of mixed-criticality avionic tasks consolidated onto a centralized flight platform. Hardware virtualization support is used to implement fault isolation, detection and recovery mechanisms for critical flight controller failures. An empirical evaluation validates the effectiveness of FlyOS's approach for sustaining safe, predictable and efficient autonomous control of a real-world quadcopter in the presence of critical task failures.

FlyOS's architecture opens up future possibilities to extend the system with additional avionic capabilities for an enriched flight solution. We intend to expand our fault-tolerance subsystem to handle kernel- and sandbox-level failures in a time-bounded manner, while still maintaining the original flight performance. In addition to redundant failover mechanisms, complete fault-recovery will also be considered. We also aim to incorporate real-time capabilities for adaptive flight control, and in-flight mission re-configurability, to maintain flight stability in varied environmental conditions.

VI. Acknowledgements

Thanks to the shepherd and reviewers for their help improving this work, which is funded in part by the National Science Foundation (NSF) Grant # 2007707. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

[1] "Archived: Qualcomm Snapdragon Flight Kit," Accessed Oct. 2021. [Online]. Available: https://ardupilot.org/copter/docs/common-qualcomm-snapdragon-flight-kit.html

[2] "Ardupilot [Home.]." [Online]. Available: https://ardupilot.org/

[3] "Automatic Control System for UAV with a Takeoff Weight of 100 kg up to 4000 kg," Accessed Oct. 2021. [Online]. Available: https://www.uavos.com/products/autopilots/ap10-1-automatic-control-system-for-uav/

[4] "Betaflight [Home.]." [Online]. Available: https://betaflight.com/

[5] "Cleanflight [Home.]." [Online]. Available: goo.gl/uCGmr4

[6] "Flight RB5 5G Platform," Accessed Oct. 2021. [Online]. Available: https://www.qualcomm.com/products/qualcomm-flight-robotics-rb5-5g-platform

[7] "Github Documentation Wiki for Intel Ready to Fly Drone," Accessed Oct. 2021. [Online]. Available: https://github.com/intel-aero/meta-intel-aero/wiki/02-Initial-setup

[8] "iNAV [Home.]." [Online]. Available: https://github.com/iNavFlight/inav/wiki

[9] "Intel Aero Compute Board," Accessed Oct. 2021. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/97178/intel-aero-compute-board.html

[10] "Intel Aero Vision Accessory Kit," Accessed Oct. 2021. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/97175/intel-aero-vision-accessory-kit.html

[11] "Intel RealSense Github," Accessed Oct. 2021. [Online]. Available: https://github.com/IntelRealSense/librealsense

[12] "MaRTE." [Online]. Available: https://www.osrtos.com/rtos/marte/

[13] "Overview of Intel Ready to Fly Drone," Accessed Oct. 2021. [Online]. Available: https://intel.ly/3b8WwGz

[14] "PX4 [Home.]." [Online]. Available: http://px4.io/

[15] "Support for Intel RealSense Camera," Accessed Oct. 2021. [Online]. Available: https://intel.ly/3uX0KKe

[16] "The Cube Autopilot," Accessed Oct. 2021. [Online]. Available: https://bit.ly/3vPxbLo

[17] "First Public Demo of Snapdragon Flight Robotics Dev Platform in One of Worlds Smallest 4K Drones," 2015. [Online]. Available: https://www.qualcomm.com/news/onq/2015/09/10/first-public-demo-snapdragon-flight-robotics-dev-platform-one-worlds-smallest-4k

[18] "SYSGO PikeOS Hypervisor," 2015. [Online]. Available: http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept

[19] "[V4,4/4] Utilize the VMX Preemption Timer for TSC Deadline Timer," Accessed Oct. 2021, 2016. [Online]. Available: https://bit.ly/3pihmM0

[20] "Qualcomm Snapdragon Flight Kit," March 2017. [Online]. Available: https://www.intrinsyc.com/vertical-development-platforms/qualcomm-snapdragon-flight/

[21] "Timer Interrupt Sources," 2019. [Online]. Available: https://wiki.osdev.org/Timer_Interrupt_Sources

[22] "Journey to Mars: How our Collaboration with Jet Propulsion Laboratory Fostered Innovation," 2021. [Online]. Available: https://www.qualcomm.com/news/onq/2021/03/17/journey-mars-how-our-collaboration-jet-propulsion-laboratory-fostered-innovation

[23] 3D Robotics Inc., "DroneKit Python." [Online]. Available: https://github.com/dronekit/dronekit-python

[24] K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization," in *ACM SIGOPS Operating Systems Review*, vol. 40, December 2006, pp. 2–13.

[25] J. Almeida and M. Prochazka, "Safe and Secure Partitioning with Pikeos: Towards Integrated Modular Avionics in Space," in *Proceedings of DASIA 2009 Data Systems in Aerospace, by Ouwehand, L. Noordwijk, Netherlands: European Space Agency*, 2009.

[26] ARINC Std. 653P1-3, "Avionics Application Standard Software Interface, Part 1 - Required Services," Wind River Systems / IEEE Seminar, 2010.

[27] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *ACM SIGOPS OSR*, 2003.

[28] F. Boniol and V. Wiels, "Towards Modular and Certified Avionics for UAV," in *Journal Aerospace Lab, Alain Appriou*, December 2014, pp. 1–8.

[29] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," in *Journal of the ACM*, vol. 20, no. 1, 1973, pp. 46–61.

[30] Z. Cheng, R. West, and C. Einstein, "End-to-End Analysis and Design of a Drone Flight Controller," in *Proceedings of the ACM SIGBED International Conference on Embedded Software (EMSOFT)*, Torino, Italy, September 30-October 5 2018.

[31] D. Clifton, "SPRACINGF3 Flight Controller Manual (Revision 4)," 2015. [Online]. Available: https://bit.ly/2Mx9dRV

[32] J. Craveiro, J. Rufino, T. Schoofs, and J. Windsor, "Flexible Operating System Integration in Partitioned Aerospace Systems," in *Actas do INForum - Simposio de Informatica*, 2009, pp. 49–60.

[33] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach," in *EDCC*, 2010, pp. 67–72.

[34] M. Danish, Y. Li, and R. West, "Virtual-CPU Scheduling in the Quest Operating System," in *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011, pp. 169–179.

[35] DroneCode, "Pixhawk Home." [Online]. Available: https://pixhawk.org/

[36] Dronecode Project, "MAVLink Developer Guide." [Online]. Available: https://mavlink.io/en/

[37] E. -H. Jeong and J. -G. Kim, "S/W Fault-tolerant OFP System for UAVs Based on Partition Computing," in *2013 International Conference on Electronic Engineering and Computer Science*, 2013.

[38] E. Bregu, N. Casamassima, D. Cantoni, L. Mottola, and K. Whitehouse, "Reactive Control of Autonomous Drones," in *14th Annual International Conference on Mobile Systems, Applications and Services (MobiSys'16)*, June 2016, pp. 207–219.

[39] A. Farrukh and R. West, "smARTflight: An Environmentally-Aware Adaptive Real-Time Flight Management System," in *32nd Euromicro Conference on Real-Time Systems (ECRTS)*, July 2020.

[40] G. Brunner, B. Szebedy, S. Tanner, and R. Wattenhofer, "The Urban Last Mile Problem: Autonomous Drone Delivery to Your Balcony," in *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2019, pp. 1005–1012.

[41] A. Golchin, Z. Cheng, and R. West, "Tuned Pipes: End-to-end Throughput and Delay Guarantees for USB Devices," in *39th IEEE Real-Time Systems Symposium (RTSS)*, 2018.

[42] A. Golchin, S. Sinha, and R. West, "Boomerang: Real-Time I/O Meets Legacy Systems," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 390–402.

[43] Q. Gu, D. R. Michanowicz, and C. Jia, "Developing a Modular Unmanned Aerial Vehicle (UAV) Platform for Air Pollution Profiling," in *Sensors*, 2018.

[44] S. Han and H. W. Jin, "Full Virtualization Based ARINC 653 Partitioning," in *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, 2011.

[45] S. Han and H. W. Jin, "Kernel-Level ARINC 653 Partitioning for Linux," in *SAC '12: Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 2012, pp. 1632–1637.

[46] D. Jensen, "B787 Cockpit: Boeing's Bold Move," in *Aviation Today*, 2005.

[47] H. C. Jo, J. K. Park, H. W. Jin, H. S. Yoon, and S. H. Lee, "Portable and Configurable Implementation of ARINC-653 Temporal Partitioning for Small Civilian UAVs," in *IEEE Access*, vol. 7, 2019, pp. 142 478–142 487.

[48] Q. Kang, C. Yuan, X. Wei, Y. Gao, and L. Wang, "A User-Level Approach for ARINC 653 Temporal Partitioning in seL4," in *ISSSR*, 2016, pp. 106–110.

[49] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux Virtual Machine Monitor," in *The Linux Symposium*, vol. 1, 2007, pp. 225–230.

[50] G. Klein, J. Andronick, M. Fernandez, I. Kuz, T. Murray, and G. Heiser, "Formally Verified Software in the Real World," *Communications of the ACM*, vol. 61, no. 10, pp. 68–77, October 2018.

[51] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber, "A Comparison of Partitioning Operating Systems for Integrated Systems," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2007, pp. 342–355.

[52] J. Liedtke, H. Härtig, and M. Hohmuth, "OS-Controlled Cache Predictability for Real-Time Systems," in *the 3rd IEEE Real-time Technology and Applications Symposium*, 1997.

[53] A. Liszewski, "NASA's Supercomputers Reveal the Incredible Turbulence Produced By a Drone." [Online]. Available: https://gizmodo.com/nasas-supercomputers-reveal-the-incredible-turbulence-p-1791179507

[54] LYNX Software Technologies, "LynxSecure Embedded Hypervisor and Separation Kernel," 2015. [Online]. Available: http://www.lynx.com/products/hypervisors/

[55] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 77. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, pp. 3:1–3:14.

[56] L. Mejias, J. P. Diguet, C. Dezan, D. Campbell, J. Kok, and G. Coppin, "Embedded Computation Architectures for Autonomy in Unmanned Aircraft Systems (UAS)," in *Sensors*, 2021.

[57] C. M. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves for Multimedia Operating Systems," in *Technical Report*. Carnegie Mellon University, 1993.

[58] A. B. Montz, D. Mosberger, S. W. O'Mally, L. L. Peterson, and T. A. Proebsting, "Scout: A Communications-Oriented Operating System," in *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*. IEEE, 1995, pp. 58–61.

[59] OscarLiang.com, "Custom Motor Mixing Multirotor | What | Calculate | Uses," April. 22 2017. [Online]. Available: https://www.oscarliang.com/custom-motor-output-mix-quadcopter

[60] P. J. Prisaznuk, "ARINC 653 role in Integrated Modular Avionics (IMA)," in *IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008, pp. 1.E.5–1–1.E.5–10.

[61] H. Pérez and J. J. Gutiérrez, "Handling Heterogeneous Partitioned Systems through ARINC-653 and DDS," in *Computer Standards & Interfaces*, vol. 50, 2017, pp. 258–268.

[62] H. Pérez, J. J. Gutiérrez, S. Peiró, and A. Crespo, "Distributed architecture for developing mixed-criticality systems in multi-core platforms," *Journal of Systems and Software*, vol. 123, pp. 145–159, 2017.

[63] R. Pineiro, K. Ioannidou, S. A. Brandt, and C. Maltzahn, "Rad-flows: Buffering for Predictable Communication," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2011, pp. 23–33.

[64] Radio Technical Commission for Aeronautics (RTCA) Std., "DO-178C/ED-12C Software Considerations in Airborne Systems and Equipment Certification," 2011.

[65] Radio Technical Commission for Aeronautics (RTCA) Std., "DO-326A Airworthiness Security Process Specification," 2014.

[66] Radio Technical Commission for Aeronautics (RTCA) Std., "DO-356A Airworthiness Security Methods and Considerations," 2018.

[67] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look Mum, no VM Exits! (Almost)," *arXiv preprint arXiv:1705.06932*, 2017.

[68] ROBOTmaker, "Real-Time Graphical Representation | S.BUS Protocol." [Online]. Available: http://www.robotmaker.eu/ROBOTmaker/quadcopter-3d-proximity-sensing/sbus-graphical-representation

[69] W. Ruan and Z. Zhai, "Kernel-Level Design to Support Partitioning and Hierarchical Real-Time Scheduling of ARINC 653 for VxWorks," in *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, 2014, pp. 388–393.

[70] J. Rushby, "Partitioning for Avionics Architectures: Requirements, Mechanisms and Assurance," in *NASA contractor report CR-1999-209347, NASA Langley Research Center*, 1999.

[71] J. Rushby, "Model Checking Simpsons Four-Slot Fully Asynchronous Communication Mechanism," in *Computer Science Laboratory–SRI International, Tech. Rep. Issued*, July 2002.

[72] J. M. Rushby, "Design and Verification of Secure Systems," in *8th ACM Symposium on Operating Systems Principles*, 1981, pp. 12–21.

[73] S. C. Technology, "Jailhouse Partitioning Hypervisor," 2014. [Online]. Available: https://github.com/siemens/jailhouse

[74] S. O. H. Madgwick., A. J. L. Harrison, and R. Vaidyanathan, "Estimation of IMU and MARG Orientation using a Gradient Descent Algorithm," in *International Conference on Rehabilitation Robotics (IEEE-ICORR)*, 2011, pp. 1–7.

[75] P. Sanchez-Cuevas, G. Heredia, and A. Ollero, "Characterization of the Aerodynamic Ground Effect and Its Influence in Multirotor Control," in *International Journal of Aerospace Engineering*, vol. 2017, no. 1823056, 2017.

[76] S. Schildermans, K. Aerts, J. Shan, and X. Ding, "Paratick: Reducing Timer Overhead in Virtual Machines," in *50th International Conference on Parallel Processing (ICPP)*. Association for Computing Machinery, August 2021, pp. 1–10.

[77] H. R. Simpson, "Four-slot Fully Asynchronous Communication Mechanism," in *IEEE Computers and Digital Techniques 137*, January 1990, pp. 17–30.

[78] S. Sinha and R. West, "Towards an Integrated Vehicle Management System in DriveOS," in *ACM Transactions on Embedded Computing Systems*, vol. 20. ACM, 2021, pp. 1–24.

[79] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-Based Secure Virtualization Architecture," in *Proceedings of the 5th European Conference on Computer Systems (Eurosys)*. Association for Computing Machinery, 2010, pp. 209–222.

[80] T. Morales, A. Sarabakha, and E. Kayacan, "Image Generation for Efficient Neural Network Training in Autonomous Drone Racing," in *2020 International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–8.

[81] S. H. VanderLeest, " Benefits and Implications of an ARINC 653 Hypervisor ." [Online]. Available: https://dornerworks.com/about/whitepapers/arinc-653-benefits-implications/

[82] S. H. VanderLeest, "ARINC 653 Hypervisor," in *29th Digital Avionics Systems Conference*, 2010, pp. 5.E.2–1–5.E.2–20.

[83] P. Viola and M. J. Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, February 2001.

[84] C. B. Watkins, "Integrated Modular Avionics: Managing the Allocation of Shared Intersystem Resources," in *Proceedings of the 25th Digital Avionics Systems Conference*, 2006, pp. 1–12.

[85] C. B. Watkins and R. Walter, "Transitioning from Federated Avionics Architectures to Integrated Modular Avionics," in *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, 2007, pp. 2.A.1–1–2.A.1–10.

[86] R. West, Y. Li, E. Missimer, and M. Danish, "A Virtualized Separation Kernel for Mixed-Criticality Systems," in *ACM Transactions on Computer Systems*, vol. 34, no. 3. New York, NY, USA: ACM, Jun. 2016, pp. 8:1–8:41.

[87] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang, "Online Cache Modeling for Commodity Multicore Processors," in *SIGOPS Oper. Syst. Rev.*, vol. 44, 2010, pp. 19–29.

[88] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang, "CAF: Cache-Aware Fair and Efficient Scheduling for CMPs," in *Multicore Technology: Architecture, Reconfiguration and Modeling, CRC Press*, 2013, pp. 221–253.

[89] J. Windsor, K. Eckstein, P. Mendham, and T. Pareaud, "Time And Space Partitioning Security Components For Spacecraft Flight Software," in *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, 2011.

[90] X-IO Technologies, "Open Source IMU and AHRS algorithms." [Online]. Available: https://x-io.co.uk/open-source-imu-and-ahrs-algorithms/

[91] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A Dynamic Cache Partitioning System using Page Coloring," in *23rd International Conference on Parallel Architectures and Compilation Techniques*, 2014.

[92] Y. Ye, R. West, J. Zhang, and Z. Cheng, "MARACAS: A Real-Time Multicore VCPU Scheduling Framework," in *37th IEEE Real-Time Systems Symposium (RTSS)*, 2016.

[93] Y. Zhang and R. West, "Process-Aware Interrupt Scheduling and Accounting," in *27th IEEE Real-Time Systems Symposium (RTSS)*, 2006.