Real-Time Systems (2023) 59:256–301
https://doi.org/10.1007/s11241-023-09399-w

FlyOS: rethinking integrated modular avionics for autonomous multicopters

Anam Farrukh[1] · **Richard West**[1]

Accepted: 26 April 2023 / Published online: 23 May 2023

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 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 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.

Keywords Integrated modular avionics · ARINC-653 · Multicore · UAVs · Partitioning hypervisor · Autonomous mission · Real-time flight management · Fault tolerance

✉ Richard West
richwest@bu.edu

Anam Farrukh
afarrukh@bu.edu

[1] Department of Computer Science, Boston University, Boston, MA, USA

1 Introduction

Multicopters have traditionally adopted a federated architecture (Gu et al. 2018; Mejias et al. 2021), which isolates and distributes flight management functions of different criticalities across separate hardware components (Intel; Qualcomm 2017). 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.

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 (Boniol and Wiels 2014). 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.

In this work, we present an integrated flight management system called FlyOS, which contrasts with the traditional federated approach that uses multiple separate hardware components. FlyOS takes inspiration from Integrated Modular Avionics (IMA) (Watkins 2006; Watkins and Walter 2007) and ARINC-653 (Avionics Application Software Standard Interface) (ARINC Std. 653P1-5 2019; Prisaznuk 2008) 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 (Rushby 1981) to map two or more guest operating systems to virtualized sandbox domains or partitions. For the purposes of this paper, we use the terms guest OSes, sandboxes and partitions interchangeably in the context of FlyOS. Separation kernels (Green Hills Software Inc 2010; Leiner et al. 2007; Li et al. 2014a; Lynx Software Technologies; McDermott et al. 2012; West et al. 2016) allow guests to co-exist on a common hardware platform as isolated regimes, which communicate only through explicit and secure channels. 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 of the guests. 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.

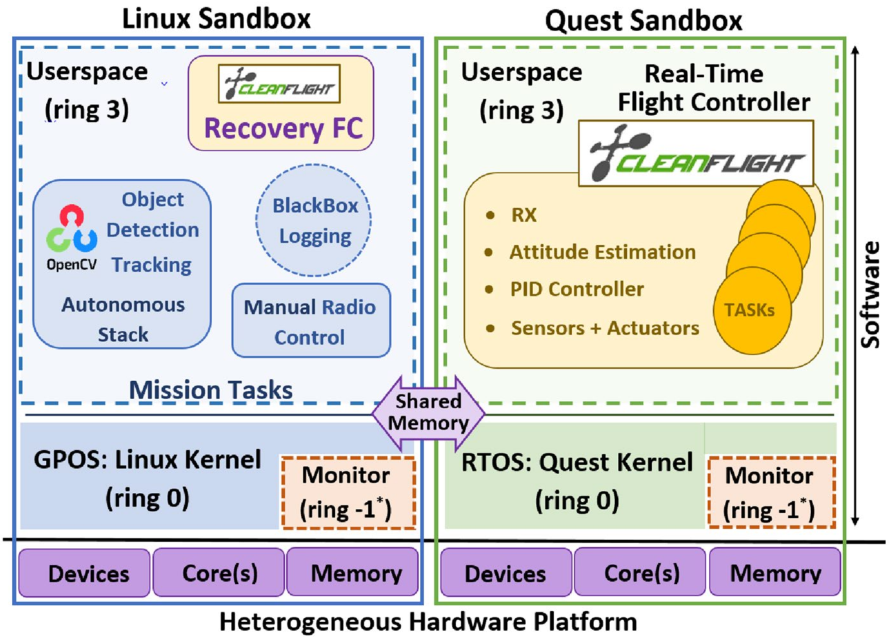


Fig. 1 FlyOS dual-sandbox configuration: Linux + Quest. **For the purposes of this work, we identify ring -1 to be the root mode software layer, which sits between the hardware and non-root guest*

FlyOS enables software redundancy to meet SWAP-C constraints of small-scale UAVs. The system aims to overcome the inherent limitation of shared resource architectures to fault containment (Rushby 1999) by providing strict temporal and spatial partitioning between guests. FlyOS's approach to integration 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 (Danish et al. 2011)), alongside mission control tasks in Yocto Linux. FlyOS's separation kernel works on the principle of *partitioning* hypervisors (Cesarano et al. 2022; Li et al. 2014a; Martins et al. 2020; Ramsauer et al. 2017; Technology 2014; West et al. 2016) 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 (Craveiro et al. 2009; VanderLeest 2010). These systems rely on the hypervisor for time and space multiplexing as well as the overall management of shared platform resources on behalf of the hosted guests. Hypervisor-based shared resource management potentially adds undue overheads (Hwang et al. 2013), which impact predictability and determinism of critical flight control.

Within FlyOS, we refactor a performance-critical flight controller to execute with real-time guarantees on Quest. A camera-based vision detection and tracking subsystem is implemented in a Linux sandbox as part of our mission control functionality (e.g., to represent a search and rescue objective). Additionally, we showcase our hypervisor-based fault-recovery subsystem for fail-safe flight control in the presence of critical function 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. motivate FlyOS's design choices by drawing parallels with the required services of the core/host software as specified by the ARINC-653 avionics standard, 3. describe FlyOS's separation kernel in the context of a dual-sandbox implementation co-hosting Linux with Quest, 4. 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 avionic application, 5. introduce *high-level mission-critical* autonomous navigation control, and 6. implement online health-management and 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 reusable application implementations, system-wide optimizations, re-configurability and improved resource usage, while reducing size, weight and power requirements of the underlying hardware.

Layout: The following section describes the FlyOS model. We motivate our design goals followed by a brief discussion on ARINC-653 objectives. We then present an overview of the system framework and take a deep dive into the avionic functions and capabilities currently supported by our prototype. Sect. 3 presents an extensive evaluation of flight performance with hardware-in-the-loop experiments. Sect. 4 introduces our readers to additional benefits afforded by FlyOS's approach to IMA. In particular, we highlight the flexibility and adaptability characteristics by describing the design of a multicore adaptive flight controller. The proposed design is extended from our prior work on sᴍᴀʀᴛflight (Farrukh and West 2020), which handles environmental factors such as wind disturbances. Related work is described in Sect. 5, while conclusions and future work are discussed in Sect. 6.

2 FlyOS: a flight management framework

2.1 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 (Radio Technical Commission for Aeronautics (RTCA) Std 2011a) 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* (Radio Technical Commission for Aeronautics (RTCA) Std 2014, 2018) for tasks and system components. This directly concerns policies related to the preservation of information integrity and confidentiality. An implementation and evaluation of such policies is beyond the scope of this paper. However, we note that FlyOS's isolation by design architecture lends itself to support security capabilities such as gateway (guard) services at communication interfaces between different sandboxed domains. This allows runtime checks to be enforced within FlyOS's inter- and intra-sandbox communication stacks that mitigate threats from malicious attacks crossing sandbox boundaries. Carefully designed OS-kernel and hypervisor-based security policies (Klein et al. 2018; Steinberg and Kauer 2010; Wang and Jiang 2010) allow FlyOS to monitor and validate flow of information between sandboxes such as flight mission commands.

Additionally, FlyOS features a *thin* and simple hypervisor on a *per guest* basis. Hypervisor redundancy inherently enables fault detection and recovery for the most privileged layer of the system. It also heightens security by reducing the attack surface of *each* hypervisor instance (Missimer et al. 2014). FlyOS's distributed virtualization architecture leverages compute redundancy within multicore systems to replicate complete system stacks. It therefore enables security by design.

Notwithstanding, in this paper, 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 system environments for locally-hosted tasks of different criticalities. Details of our design are presented in Sect. 2.3.
2. *Extensibility* Low-criticality sandboxes support re-configurable and adaptable autonomous mission applications, which increases application portability and reduces redeployment costs. Similarly one or more real-time safety-critical sandboxes allow dynamic hot-plugging of flight controllers that are tuned to different flight characteristics, e.g., for high maneuverability versus greater stability or environment adaptability.

   FlyOS envisions multiple independent sandboxed system partitions to be hosted on the same hardware platform. Static partitioning allows each sandbox to be allocated a configurable set of resources based on application demand. Each guest partition, in turn, hosts a multitude of avionic functions of equivalent levels of criticality. This enables guests and their applications to be certified independent of the other guests in the system and in accordance with their associated design assurance level as defined by the certification authorities (Radio Technical Commission for Aeronautics (RTCA) Std 2011a).

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 that are inherent to federated or hardware-distributed architectures. Likewise, FlyOS operates on the principle of separation of concerns. The hypervisor layer has a minimal memory foot-print and resides at the most privileged protection domain, replicated across each sandbox. The hypervisor (a.k.a., virtual machine monitor (VMM)) implements a run-time health-monitoring subsystem within its trusted compute base. Together with redundant VMMs, *functional* and *timing* related faults may be handled across the entire guest stack, from the application to sandboxed partition, down to the hypervisor.

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

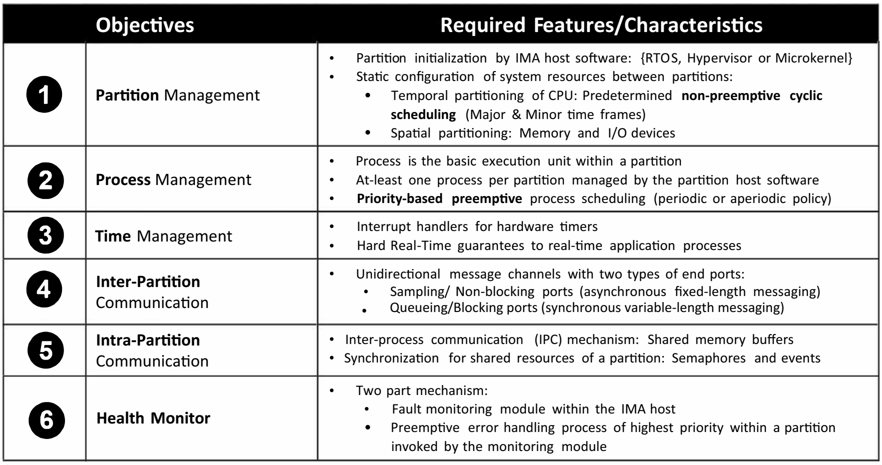
We now briefly dive into the specifics of the ARINC-653 partitioning standard as it largely applies to the *general* avionics domain since its inception in 1996. As mentioned previously, FlyOS draws its design inspiration from the ARINC partitioning standard. The upcoming section thus constructs parallels and identifies design differences between the core operating environments provided by FlyOS and ARINC-653. We aim to delineate the design goals of an IMA-based flight management system when applied to small-scale multicopters.

2.2 ARINC-653: a discussion

ARINC-653 is the de-facto partitioning standard, enforced by the regulatory bodies such as the FAA (Federal Aviation Authority) (FAA) and EASA (European Aviation Safety Agency) (FAA and EASA), within the commercial avionics domain for manned aircraft. Compliance with the standard is required to achieve airworthiness of IMA-based flight architectures.

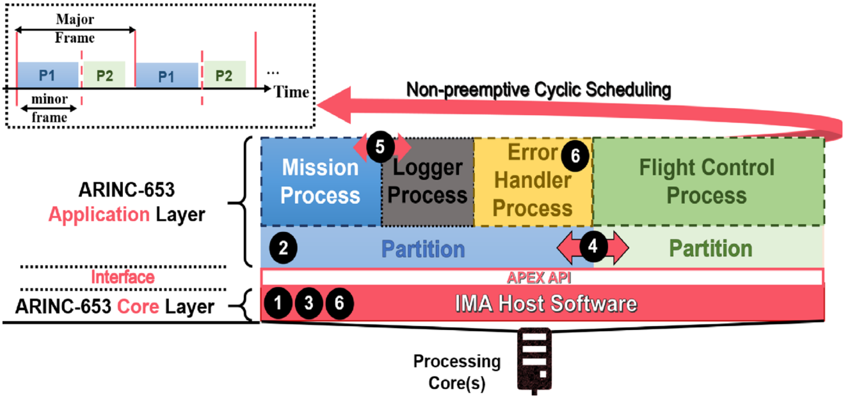
Temporal and spatial isolation of system resources is one of the key requirements of IMA for predictable and safe flight behavior. Such isolation is necessary when consolidating mixed-criticality flight functions on a common compute platform. ARINC-653P1 (ARINC Std. 653P1-5 2019) defines a set of essential baseline services and required behaviors (Fig. 2a) to facilitate system architects in their design of a robust safety-critical avionic system. Conforming with the ARINC rules enables multiple avionic applications of varying certification requirements, to execute without interference, using logical containers called partitions.

The ARINC standard defines two main software abstractions for an IMA architecture: 1. 2. a standard core operating environment as the IMA Host and 3. the



Objectives		Required Features/Characteristics
1	**Partition** Management	• Partition initialization by IMA host software: {RTOS, Hypervisor or Microkernel} • Static configuration of system resources between partitions: • Temporal partitioning of CPU: Predetermined **non-preemptive cyclic scheduling** (Major & Minor time frames) • Spatial partitioning: Memory and I/O devices
2	**Process** Management	• Process is the basic execution unit within a partition • At-least one process per partition managed by the partition host software • **Priority-based preemptive** process scheduling (periodic or aperiodic policy)
3	**Time** Management	• Interrupt handlers for hardware timers • Hard Real-Time guarantees to real-time application processes
4	**Inter-Partition** Communication	• Unidirectional message channels with two types of end ports: • Sampling/ Non-blocking ports (asynchronous fixed-length messaging) • Queueing/Blocking ports (synchronous variable-length messaging)
5	**Intra-Partition** Communication	• Inter-process communication (IPC) mechanism: Shared memory buffers • Synchronization for shared resources of a partition: Semaphores and events
6	**Health Monitor**	• Two part mechanism: • Fault monitoring module within the IMA host • Preemptive error handling process of highest priority within a partition invoked by the monitoring module

(a) (Part-1) Required Services and Goals.



(b) Reference software stack of an ARINC-653 compliant system. *(Top-left)* Temporal partitioning schedule in ARINC-653.

Fig. 2 ARINC-653 avionic standard for IMA architectures

APEX API interface between avionic applications and the IMA Host. The interface allows applications to communicate with their execution environment, which provides a standard set of system services (Spitzer et al. 2015).

Figure 2b illustrates the layered structure and software components of a representative, ARINC-653 compliant, IMA system. System components are mapped to the corresponding list of mandatory services, which are extracted from the technical standard and summarized in Fig. 2a.

As per the ARINC specification, application partitions are assigned independent regions of system memory. These regions are protected by hardware mechanisms such as a Memory Management (or Protection) Unit. Partitions consist of independent text and data regions within memory and have a well-defined execution context and configuration attributes. A partition comprises a set of periodic and aperiodic processes or tasks. To draw an analogy with UNIX-like systems, an ARINC *partition* refers to a UNIX process while an ARINC *process* corresponds to a UNIX thread running inside the process address space.

ARINC partitions execute in a non-preemptive periodic manner, at-least once every *major cycle* of the statically defined *partition schedule*. This is accomplished by allocating execution time windows, often called *minor frames*, to each partition within a major time frame. When the minor frame terminates, the partition is preempted and the next partition in the schedule is selected for execution. The preempted partition can continue its execution in the next activation slot. An executing partition is allowed exclusive access to available hardware resources and runtime services during the duration of its minor time frame. A second level of fixed-priority preemptive scheduling is applied to the ARINC processes within each partition. The two-level hierarchical schedule thereby ensures strict temporal partitioning within the IMA platform.

The IMA host is responsible for partition initialization and configuration as well as system-wide run-time management. Process management and intra-partition communication between processes is delegated to software within individual partitions. Communication between partitions is managed by Sampling and Queueing ports and channels. These provide explicit message transfer semantics for asynchronous and synchronous communication, managed by the IMA host.

Overall, the ARINC standard aims to promote application portability and platform (re-)configurability through a clear demarcation of boundaries between software components and explicit definitions of inter- and intra-partition communication interfaces. Each ARINC abstraction layer within the reference stack of Fig. 2b renders specific services to the overall avionic system according to the assigned objective(s).

The APEX API decouples partitioning and multitasking, while exposing a standard interface for applications to access and interact with the IMA host services. This is intended to promote widespread adoption of IMA architectures across all avionic domains, beyond that of commercial aircraft.

Separation Kernels and partitioning hypervisors Depending on the target avionics domain and system's requirements, the partitioning environment provided by the IMA host has historically been implemented within different abstraction layers of the system: user/application, (micro-)kernel (Buczyński et al. 2022; Delange 2011; VanderLeest 2016), or hypervisor (Han and Jin 2011; Masmano et al. 2011; VanderLeest 2017).

A hypervisor provides logical isolation between hosted virtual machines (or guest operating systems (OSes)). Whereas consolidating hypervisors multiplex guests on a shared physical machine, FlyOS adopts a partitioning hypervisor approach to implement a separation kernel. This statically divides a pool of machine resources such as CPU cores, memory regions and I/O devices, between different guest virtual machines. Distinct resource domains thereby ensure strict isolation between guests.

In accordance with the separation kernel principles, guest partitions therefore appear indistinguishable from separate physical machines thus implementing a distributed system-on-a-chip. Contrary to other ARINC compliant separation kernels (Green Hills Software Inc 2010; Leiner et al. 2007; Lynx Software

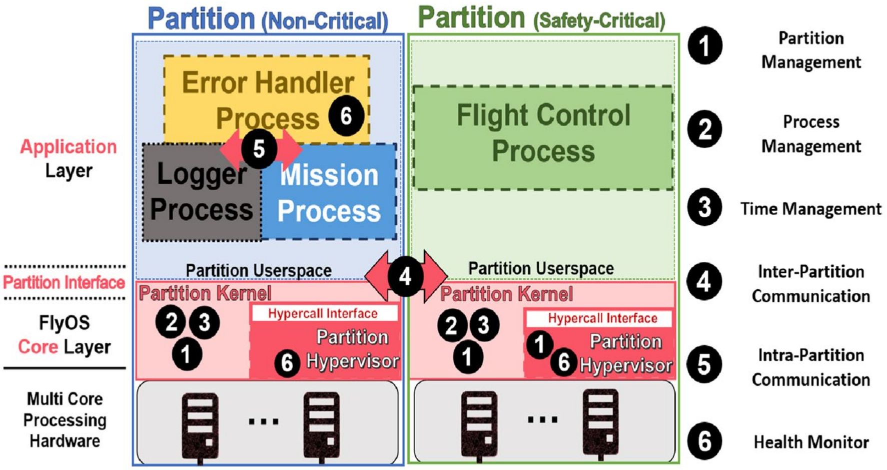


Fig. 3 FlyOS's architectural layout with ARINC-653 services mapped to corresponding software layers

Technologies), FlyOS's design, inspired from the Quest-V hypervisor (Li et al. 2014a; West et al. 2016), avoids the need for hypervisor-level runtime resource management on behalf of the guests. This, in turn, minimizes the trusted code base of the hypervisor, enhancing safety and security. Since guests have direct access to a subset of the hardware, this greatly improves access latencies and hence enables real-time predictability for application execution. We measure task latencies for our prototype multicopter implementation in Sect. 3.

FlyOS guests are connected via explicitly defined synchronous and asynchronous communication channels for safe and predictable inter-partition communication. Unlike a federated architecture that connects separate physical machines via communication bus networks (e.g., CAN, Ethernet, TSN and so forth), FlyOS provides secure shared memory communication between guests on the same host (Sinha and West 2021). A prototypical architecture enabled by FlyOS is shown in Fig. 3. The figure also shows a mapping of FlyOS services to ARINC-653 objectives enumerated in Fig. 2a.

FlyOS distributes the task of the IMA host between the hypervisor and guest kernel domains. The virtual machine monitor (VMM) of each guest in Fig. 3, supports a distilled subset of the core features required by ARINC-653: • **(1)** spatial partitioning of resources and guest initialization and • **(6)** fault identification and recovery. Spatial partitioning of processing cores, memory and I/O devices between guest OSes, essentially achieves partitioning in *both* time and space. FlyOS therefore completely voids the need for a partition schedule. Upon system initialization, guest partitions are created and subsets of machine physical resources are allocated to each, based on a static configuration. Once the setup is complete, control is transferred to the guest kernel. Thereafter, the VMM plays a more passive role in the run-time management of the guest. This lowers system's run-time complexity and reduces the interface requirements between the hypervisor and application tasks to a simplified hypercall interface (Fig. 3).

Benefits of a partitioning hypervisor FlyOS relies on the *share-nothing* principle and introduces the concept of *self-management* for guest system partitions. Without active involvement of the monitor in the critical execution path of each virtual machine, guest OSes becomes solely responsible for managing their assigned resources according to their respective kernel-level policies. For example, temporal partitioning of CPU cores is directly determined by the built-in scheduling policies of the guests. Similarly time and process management responsibilities are dedicated to individual guest kernels. This greatly reduces resource management overhead per guest, while improving flight responsiveness and timing predictability of avionic applications.

An important benefit of FlyOS's approach is that its partitioning hypervisor avoids the need for two-level scheduling of traditional ARINC-653 architectures. Instead FlyOS achieves both temporal and spatial partitioning of CPU, memory and I/O resources by first partitioning platform resources spatially *between* guests and then partitioning the subsets of resources temporally *within* guests.

Criticality levels FlyOS defines separate criticality classes for *mission*, *timing*, *safety* and *security*, to address the design assurance levels (DAL) (Radio Technical Commission for Aeronautics (RTCA) Std 2011a, b; Spitzer 2006) required by certification authorities. Each guest is assigned a distinct criticality class or DAL to focus verification and validation efforts according to their respective requirements. This allows concurrent development of guest systems and modular upgrades of functions as well as incremental assurance (VanderLeest and Matthews 2021) to prove airworthiness of the IMA system.

For our dual-partition prototype, we define the RTOS partition as timing- and safety-critical and Linux as mission-critical (or non-safety critical). Each guest, in turn, hosts application tasks that directly coincide with the partition's criticality class. Applications are either hosted entirely within a partition or span across partitions depending on their task criticality requirements. Each partition and its resident tasks are then assured at varying levels of rigor for predictability, safety and security. Partitions can be configured as symmetric multi-processing (SMP) or asymmetric multi-processing (AMP) guests based on the availability of platform resources. In accordance with ARINC-653, this enables flexible architectures for guests and overall system scalability.

Multicore systems The newest revision of ARINC-653 Part-1 (ARINC Std. 653P1-5 2019; LYNX and AFuzion) extends applicability to multicore platforms (European Union Aviation Safety Agency (EASA) 2012; Federal Aviation Administration 2017, 2016; Silva and Tatibana 2014), requiring each ARINC process to at least have a fixed processor core affinity. This capability is termed bound multi-processing (BMP). Due to its restrictive nature, BMP falls short when it comes to CPU utilization and efficient management of multicore IMA platforms. Part-2 (ARINC Std. 653P2-4 2019) of the standard therefore extends this capability to optionally enable symmetric multi-processing. Thus a process can run on any core as well as change its task-to-core affinity at runtime based on statically defined configuration tables.

Upon initialization of a standard, ARINC-compliant IMA system, processes need to be statically assigned core affinities from amongst the allocated cores of an ARINC partition (Jo et al. 2019). These assignments optionally change during normal operation mode based on the extended specification. ARINC partitions are also portable between different processors. In addition to task and application level portability (Ye et al. 2016) within an SMP guest, FlyOS extends the load-balancing capability across guest boundaries. This allows use of run-time process migration (Li et al. 2014b) between guests within agreed upon criticality domains.

FlyOS allows single- and multicore partitions to execute simultaneously on their respective subset of core(s). The tightly coupled multi-domain framework ensures bounded end-to-end worst-case execution times for task pipelines spanning multiple cores or guest OSes. FlyOS mitigates micro-architectural resource conflicts by implementing cache and memory contention-aware policies. Section 3 presents quantified empirical evidence of predictable execution of parallel task pipelines spanning two guest domains in our prototype setup.

FlyOS enforces real-time schedulability of critical task pipelines (West et al. 2012). Notwithstanding, efforts are underway to further investigate possible interference channels (Domas 2018; VanderLeest and Thompson 2020) that affect timing behavior (Andersson et al. 2022; Park et al. 2019).

Predictable I/O Interrupt-based I/O has the potential to compromise temporal partitioning (VanderLeest 2017) between ARINC partitions, which time-share a CPU core. Methods such as slack scheduling (Beckert and Ernst 2015; Cronk; Parkinson 2018; Parkinson and Kinnan 2015) and credit-based reservations (Beckert et al. 2017; VanderLeest 2014) have been proposed to achieve deterministic asynchronous event handling for I/O requests. However, research in this domain primarily relies on maintaining a partition schedule that is compliant with ARINC-653. Contrary to these approaches, FlyOS partitions I/O devices and their corresponding hardware interrupt lines (IRQs) between individual guest OSes. This is achieved via device blacklisting in the PCI and ACPI configuration space during device enumeration.

Time critical I/O handling is dedicated to the Quest RTOS. Quest handles I/O events at the same priority as the requesting task in a time-budgeted manner (Danish et al. 2011; Missimer et al. 2016). Since device management is performed directly within each sandbox, interrupts are delivered directly to the sandbox kernel without monitor intervention. FlyOS therefore avoids monitor traps to handle interrupts. It also circumvents the need for a split driver model found in systems like Xen (Barham et al. 2003), which require a specialized guest domain for system-wide interrupt handling and distribution. FlyOS benefits from the increased performance of multicore platforms whilst ensuring safe isolation and accounting of I/O events.

Predictable communication The APEX interface defines partition communication mechanisms facilitated by the IMA host. These are based on synchronous and asynchronous message transfers with queuing and sampling ports respectively. In accordance with the standard, FlyOS defines an equivalent shared-memory communication library and corresponding *shmcomm* API (Sinha and West 2021), which allows both synchronous and asynchronous message-passing semantics between guest partitions.

Our inter-partition communication mechanism implements low latency and high bandwidth channels between virtualized guest domains. Channels and their corresponding end points are set up via shmcomm kernel modules within each guest. Channel creation requests are mediated by the kernel to the corresponding guest-local VMMs. The VMM uses a hardware virtualization feature of the platform to create memory mappings per channel. Tasks in a remote guest connect to the channel using system-wide unique channel IDs. Once a channel is created, tasks in one partition communicate and interact with remote partitions without involvement of the VMM or the kernel. This allows secure and predictable information flow across guest boundaries.

FlyOS implements blocking and non-blocking communication protocols using ring buffers and Simpson's 4-slot mechanism (Simpson 1990), respectively. Simpson's algorithm ensures data integrity during reads and writes for the single, most recent message data. Ring buffers on the other hand enable historical data to be maintained for as many slots as the size of the buffer.

Our blocking approach using ring buffers is based on pairwise communication between a producer and a consumer. Using shared variables that identify the IN (next slot to place data) and OUT (next slot to consume data) indices of the ring buffer enables the communication protocol to differentiate between buffer full and empty states. When the buffer is full, the producer will wait, while when the buffer is empty the consumer will wait. In our case, we implement a busy waiting mechanism with the knowledge that producers and consumers will be vacated from their CPUs when they no longer have budgets to continue execution or the guest scheduler has selected other tasks. Blocking is avoided, however, by rate-matching the data producer and consumer tasks.

For intra-partition semantics of queues and buffers as well as shared access to resources within a single partition, FlyOS relies on the default inter-process communication (IPC) mechanisms and synchronization primitives available within each respective guest kernel such as pipes, shared memory, mutexes and semaphores.

Fault management As mentioned earlier, FlyOS's design allows fault management across the system stack. A prototype of the hypervisor-based health monitoring subsystem for applications is presented in Sect. 2.4.3 along with an outline of our forthcoming research work for ensuring fault isolation at the partition-level (or sandbox-level in FlyOS parlance). As opposed to the contemporary hypervisor based approaches to IMA in the commercial and open-source sectors, FlyOS replicates monitor functionality (Fig. 3) with a distinct monitor module for each sandbox. The distributed design opens opportunities for 1. increased system availability in case of faults at the most privilege level of the IMA host, 2. modular redundancy against Byzantine faults and 3. functional diversity to avoid duplication of exploitable weaknesses within a single monitor (Li et al. 2014a; West et al. 2016). The modular structure of the fault tolerance subsystem thus aligns with the health monitoring requirements specified by the ARINC standard.

In summary, FlyOS presents a novel space-time partitioning approach to IMA, which is on par with the strict isolation requirements of ARINC-653. FlyOS's architecture distributes core system services of the IMA host between the VMMs and individual guest kernels. Each VMM operates at the highest privilege level of the platform with a minimal TCB. This reduces the cost of assurance of the hypervisor and allows reusable software modules between guests.

FlyOS's design paradigm for multicore IMA leverages robust partitioning mechanisms of separation kernels to implement predictable, safe and performance efficient avionic systems. As such, it has the potential to impact the current architectural landscape of integrated modular avionics for the multicopter domain. A comparison with state-of-the-art implementations within this domain is provided in Sect. 5. In the next sections, we showcase our design in detail with a working prototype on a custom-built quadcopter featuring autonomous flight with real-time temporal characteristics and a robust health monitoring subsystem.

2.3 System prototype

Figure 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, runtime 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 (Adams and Agesen 2006). For the current work, we utilize the quad-core Aero Compute Flight Hardware by Intel® (Intel). 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 assign real-time flight control tasks between 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 as well as non-linear control algorithms (Kamel et al. 2015) and communication protocols (e.g., DShot (OscarLiang.com)) for more precise control of the copter. To ensure timing predictability for concurrently executing tasks, techniques are employed that handle both cache and bus contention (Ye et al. 2016).

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 Sect. 2.4.2.

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 Sect. 2.4.3.

2.4 Avionic capabilities

2.4.1 Real-time flight controller

For the example flight controller implementation, we take inspiration from our team's previous work (Cheng et al. 2018; Farrukh and West 2020) on the popular open-source autopilot: Cleanflight (Cleanflight Autopilot). 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 (Farrukh and West 2020). 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 (Ardupilot Autopilot; Betaflight Autopilot; iNAV Autopilot; PX4 Autopilot). 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 1 lists each required Cleanflight task, τ_i, with budget, C_i, and period T_i. These tasks are redefined with hard deadlines equal to their corresponding periods, for FlyOS. 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 (Cheng et al. 2018; Golchin et al. 2018), along which data flows from inputs to outputs.

Quest uses a variant of rate-monotonic scheduling (RMS) (Liu and Layland 1973) algorithm by defining a virtual CPU (vCPU) abstraction as a schedulable entity (Danish et al. 2011) 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 (Mercer et al. 1993).

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 executing on multiple cores.

Figure 4 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 1), a setpoint generator (Process-1) in Linux communicates across an asynchronous shared memory pipe buffer with a light-weight thread in the RTOS acting 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 (Rushby 2002; Simpson 1990), 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 (Madgwick et al. 2011; X-IO Technologies) 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,

Table 1 List of essential flight tasks in FlyOS

Tasks	Budget (us)	Time period (us)	Freq (Hz)	Criticality	Sandbox Assignment	Description
IMU (GYRO+ACC)	100	1000	1000	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	2000	500	SAFETY	Quest	Attitude controller + throttle mix
MOTOR	1000	2500	400	SAFETY	Quest	Process PWM commands
BLACKBOX	20	2000	500	BACK-GND	Linux + Quest	Log flight + mission data

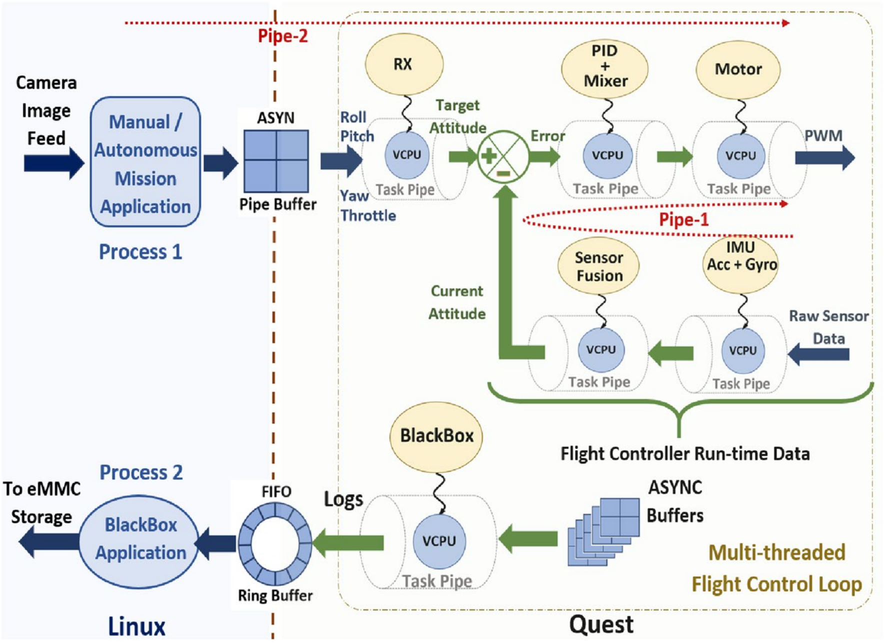


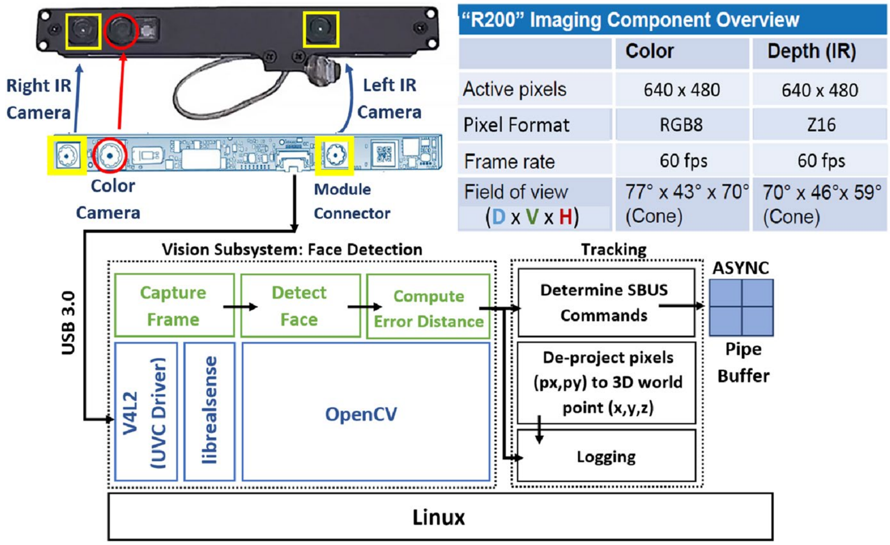
Fig. 4 FlyOS's software-distributed flight-control model with threaded tasks

pitch and yaw targets along the feed-forward path of the loop, shared with Pipe-1 (refer to Fig. 4). 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.

2.4.2 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 pattern recognition application for face-image detection and tracking that relies on librealsense (Intel) and OpenCV for capturing and processing camera images. We utilize a USB3.0 Intel RealSense (R200) (Intel) camera module, which features a 3D imaging system that is capable of providing color and depth video streams. Figure 5 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. 5 FlyOS’s vision subsystem (Process-1) with the RealSense R200 Camera

Algorithm 1 Image Detection and Tracking.

```
Require: Haar-classifier 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*/
async_chan = $create\_shared\_memory$ (ASYNC_TYPE)
$ctx = r200\_create\_context()$
$r200\_enable\_stream(ctx, \{\text{COLOR}, \text{DEPTH}\})$
while true do
    /* Capture and retrieve image frame */
    $data = get\_raw\_frame\_data()$ /*for enabled image streams*/
    $frame = to\_openCV\_matrix(data)$ /*frame vector to matrix*/
    $\{px_0, py_0\} = \{\frac{frame.cols}{2}, \frac{frame.rows}{2}\}$ /*frame center*/
    /* OpenCV: detect face */
    cv::CascadeClassifier.$detectMultiScale(frame, faces,$
            min=200×200,max=1000×1000)
    /* Estimate distance offset and generate command */
    $\{fx_c, fy_c\} = \frac{faces[0].width}{2}, \frac{faces[0].height}{2}$ /*1st face’s center*/
    dPitch = $rate\_dPitch \times (fy_c - py_0)$ /*pitch-up distance*/
    dYaw = $rate\_dYaw \times (fx_c - px_0)$ /*yaw-right distance*/
    /* command in correct format */
    $commandData$[Roll,Pitch,Yaw, Throttle] = F({0, dPitch, dYaw, 0}) /*F(command) is
    the conversion function specific to the flight controller*/
    /* Write to shared memory */
    $write\_shared\_memory$(async_chan , $commanData$)
end while
```

OpenCV supports a ready-to-use face detection algorithm based on the Haar-feature cascade classifier (Viola and Jones 2001) approach. Known for its speed and simplicity, it is 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 20×20 pixels (px). An integrated classifier function (Line 10) detects faces in each frame captured by the color camera at runtime 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 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 asynchronous pipe buffer (Line 18). For the Cleanflight autopilot, set-point values are packaged as SBUS (ROBOTmaker) 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 image-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 3 validates FlyOS's autonomous tracking capability.

2.4.3 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. *Application fault 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 (guest) fault 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) (OSDEV.org 2019) 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. 6 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 (Jiang 2016; Schildermans et al. 2021) 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 and 5). Consequently, the faulting flight controller is marked as compromised (depicted in grey in Fig 6) 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

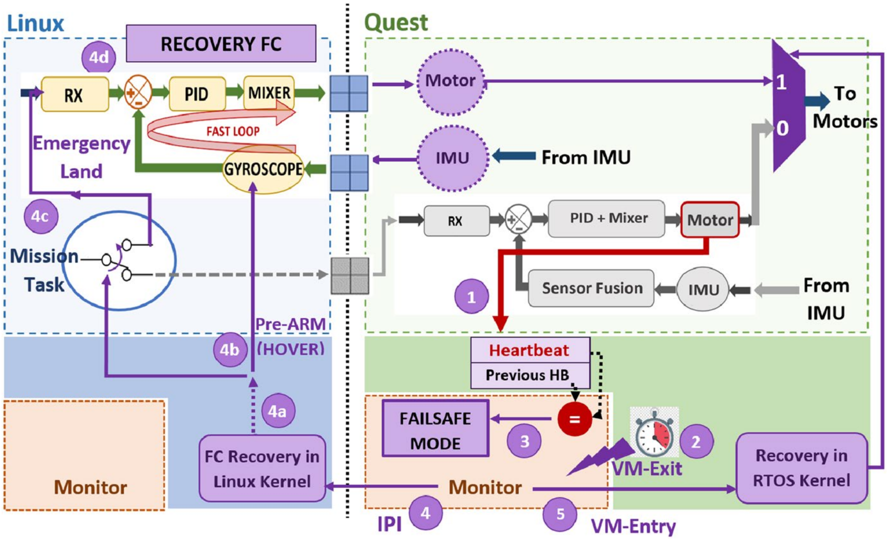


Fig. 6 Flight controller fault tolerance: detection and remote recovery

Table 2 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

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 2 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

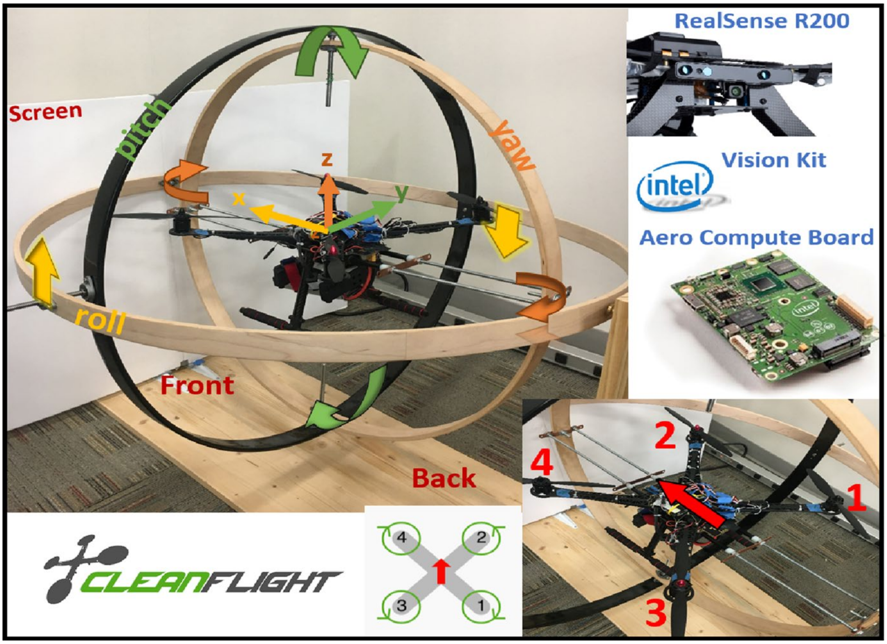


Fig. 7 BirdCage testbed for real-world experiments. The quadcopter motor configuration is enumerated at the bottom-right

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.

3 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.

3.1 Experimental setup

3.1.1 Hardware

Figure 7 shows our custom built S500 (500 mm) quadcopter mounted at the center of a 3-axis mechanical gyroscope, called the *BirdCage* (Farrukh and West 2020). The

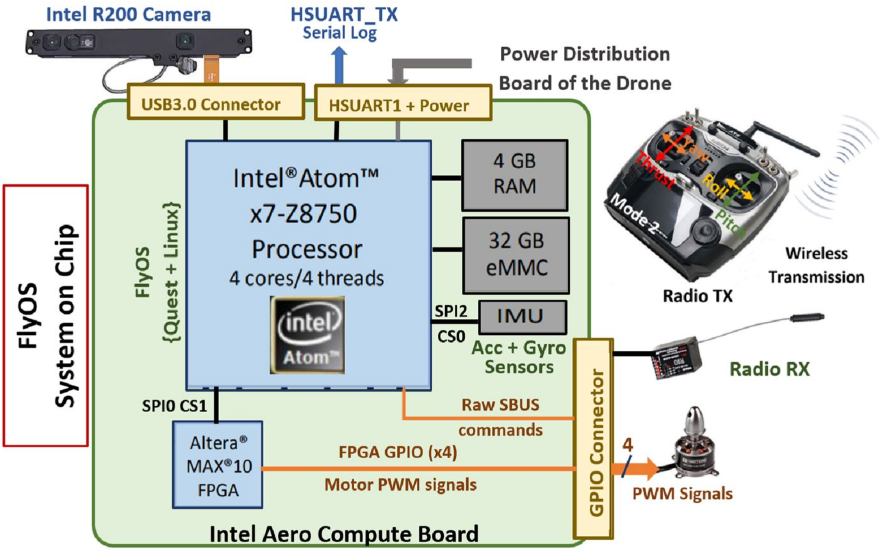


Fig. 8 Aero Compute Board with manual radio control setup

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 ($40^{\varepsilon}\times 30^{\varepsilon}$) in front of the drone to project images for vision-based tracking experiments.

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 (OscarLiang.com 2017) 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 (Intel), which includes the RealSense R200 camera module.

Figure 8 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. 4.

3.1.2 Performance metrics and settings

For the **BirdCage** experiments, we record attitude variation profiles of the quadcopter 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 ±0.5° (shown

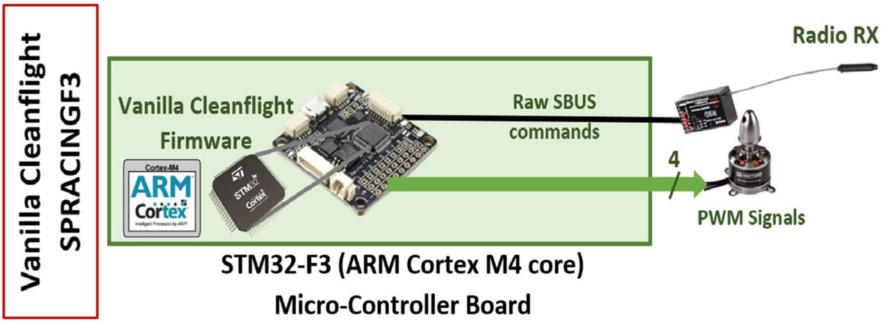


Fig. 9 SPRACINGF3 with manual radio control setup

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.

3.2 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.

3.2.1 Setup

Vanilla Cleanflight is flashed on an SPRACINGF3 (Clifton 2015) flight microcontroller and installed on the drone. The microcontroller's GPIO pinout provides a direct interface to the drone's motors, as shown in the connection diagram in Fig. 9. Featuring the STM32F3 processor, the hardware controller offers native support for the original flight software stack. A TX/RX pair is used to arm (activate) the drone, so that its ready to fly, 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 (Sect. 2.4). The main loop-time for the *fast-loop* is set to the maximum supported frequency of 1000 Hz, which represents the best response time performance (Farrukh and West 2020) on the microcontroller platform. PID constants for both FlyOS and Cleanflight autopilots are tuned to yield stable flight control behavior with minimal response time.

Table 3 Error statistics of the flight profiles in Fig. 10

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

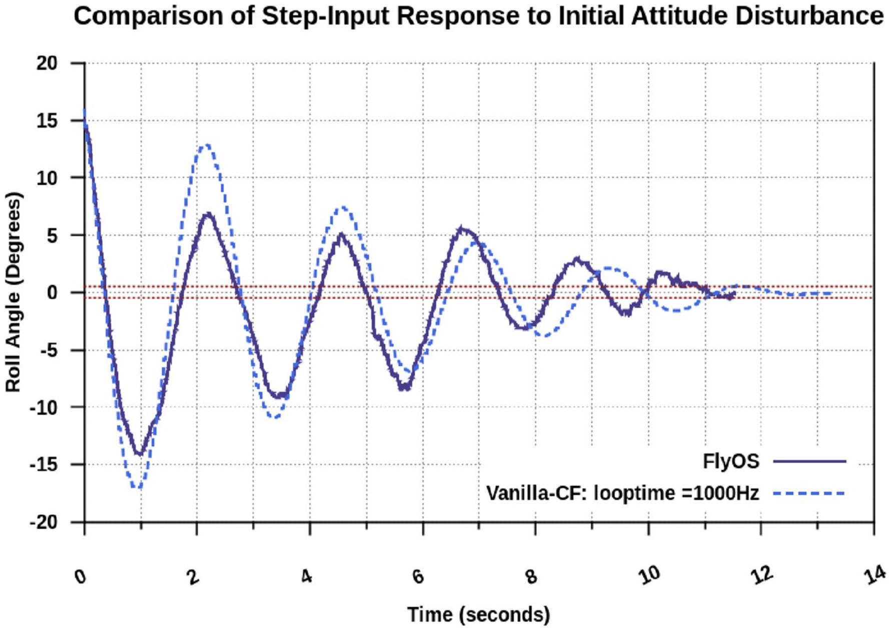


Fig. 10 Roll-Right attitude correction profile

3.2.2 Results

For the BirdCage experiment, our steady-state target is set at a horizontal hover (0° ± 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.

Figure 10 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 10.87 s compared to 11 s 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 3. 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.

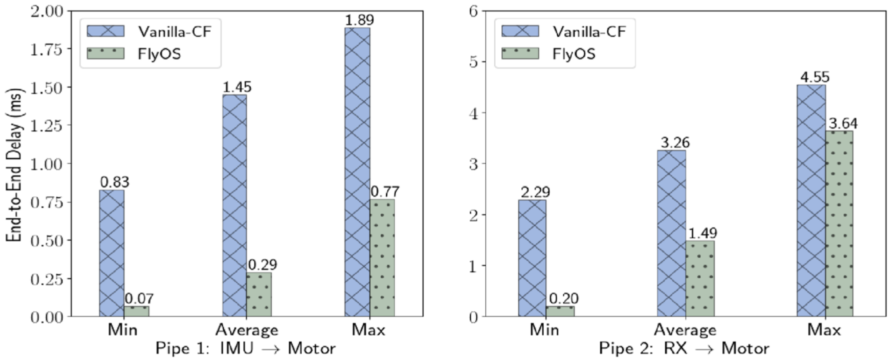


Fig. 11 E2E latencies for two critical flight control paths within FlyOS. Vanilla-CF provides a reference for comparison

E2E control latency is reported in Fig. 11 for the two task pipelines (**Pipe-1** and **Pipe-2**) within FlyOS's flight control loop (Refer to Fig. 4). 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.

We note that Vanilla-CF shows a limited variance between maximum and minimum latencies for both pipes as opposed to FlyOS. This is a direct consequence of Vanilla-CF's fast control loop, which executes non-preemptively at the highest priority. The fast loop comprises a chain of sub-tasks that sample gyroscope data, execute PID control and update motor commands in tandem, as part of a single task. Additionally the best-effort non-real time scheduler does not guarantee task frequencies or deadlines. Tasks of lower priority exhibit significant deviations from their assigned execution frequencies and experience heavy activation jitter (Farrukh and West 2020). This is directly influenced by the runtime frequency of the higher-priority fast-loop.

Contrary to this setup, the FlyOS port of Cleanflight tasks enforces strict deadlines and maintains fixed task execution frequencies and corresponding priorities as configured in Table 1. We also refactor the fast-loop as individual real-time threads in-order to improve E2E times. The task distribution shown in Fig. 4 was determined as the ideal task set to achieve lowest worst-case control latencies from amongst different possible configurations. Since tasks are preemptible and scheduled by a real-time scheduling policy, FlyOS pays the cost of predictability with increased variance. However, the E2E delay variance of a feasibly schedulable sequence of FlyOS tasks is always bounded by the sum of all periods (Golchin et al. 2020). No such guarantee is provided for Vanilla-CF's E2E pipeline execution.

3.3 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. 7).

3.3.1 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 threefold: 1. detect an image of a face of size 10×10 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. 7), which we measure using the native DEPTH stream from the IR-sensors in the Realsense module. We record updates in the horizontal (x) 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.

3.3.2 Results

Pitch and **Yaw** attitude adjustment profiles in response to step- and ramp-input stimuli are shown in Figs. 12 and 13 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 4. Steady-state *alignment* and root-mean-square *tracking* error is measured for statically positioned and moving targets respectively.

For step-profiles shown in Fig. 12, 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 0.9 s of each other. The transient control response however, shows higher magnitudes of over- and undershoots and sharper corrections in the pitch axis compared to yaw's smoother and heavily damped trace. Similarly, ramp profiles in Fig. 13 show that the drone is successfully able to track the target with a root-mean-square error of 1.19° and 0.5° in

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

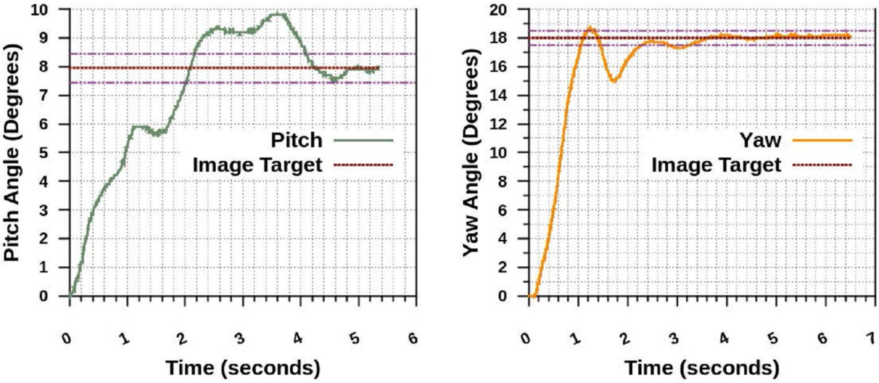


Fig. 12 Detecting a static image: step-response in Pitch and Yaw axis

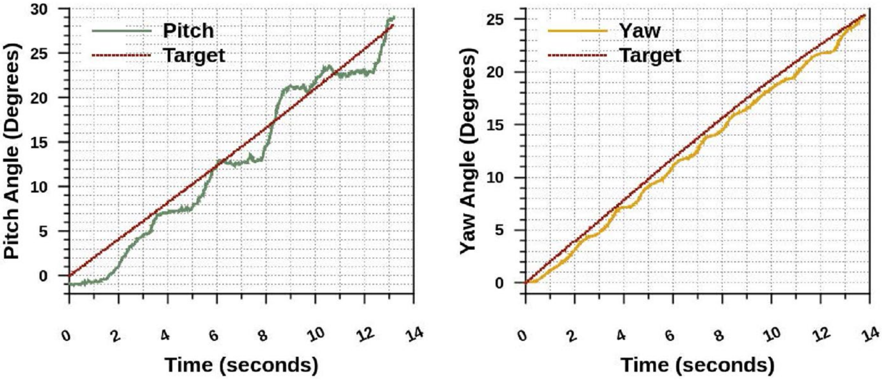


Fig. 13 Tracking a moving image: ramp-response in Pitch and Yaw axis

Table 4 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

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.

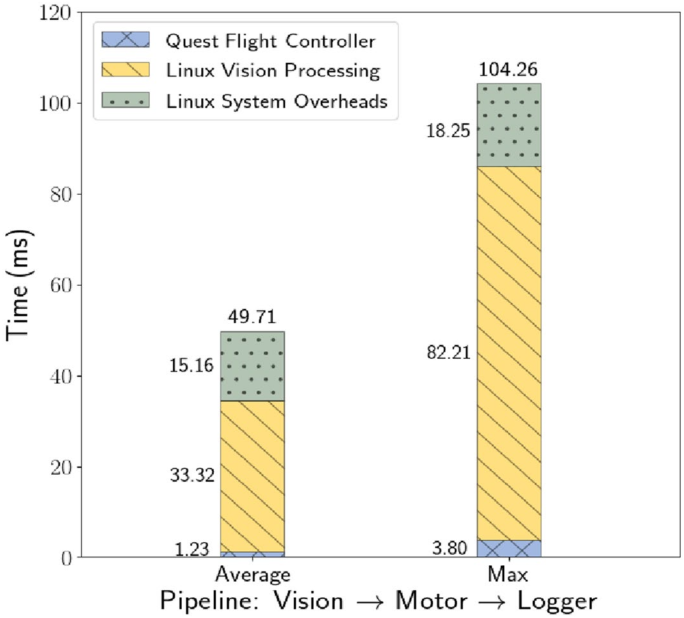


Fig. 14 Round-trip times for vision pipeline with constituent task latencies

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 (Liszewski 2021; Sanchez-Cuevas et al. 2017). 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 four 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. 14. 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 (yellow bar). 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.

3.4 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) (Intel; Intel) drone. The Intel-RTF drone runs the Ardupilot (Ardupilot Autopilot) flight controller hosted on the Pixhawk (DroneCode) companion microcontroller board.

3.4.1 Setup

The Intel-RTF drone is a pre-assembled quadcopter, which supports programmable UAV applications and mission control. The platform is a dual-board (federated) 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 (baud: 57600 bits per second) is managed by the MAVLink-router (MAVLink Router) soft-service within Linux. MAVLink commands (Ardupilot Autopilot; Dronecode Project) and the corresponding acknowledgment (ACK) messages, packaged in frames of 263 Bytes in size, 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 (Bregu et al. 2016): 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 (3D Robotics Inc; 3DR), 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

Table 5 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

memory communication. For every message sent, an ACK message is received, timed and logged on the Linux side.

3.4.2 Results

Table 5 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. (2017). The authors analyze an ARINC-653 compatible DDS communication link between two MaRTE (OSRTOS) RTOS virtualized partitions, hosted by the XtratuM (Crespo et al. 2010) hypervisor on a multicore x86 platform. Their results show average round-trip latencies of 100 s 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.

3.5 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 1000 Hz frequency (looptime=1 ms) 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 2 ms (500 Hz). 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. 6. End-to-end delay statistics are presented in Fig. 15. The measured worst-case recovery time to reach the hover state in Quest is 0.77 ms. This

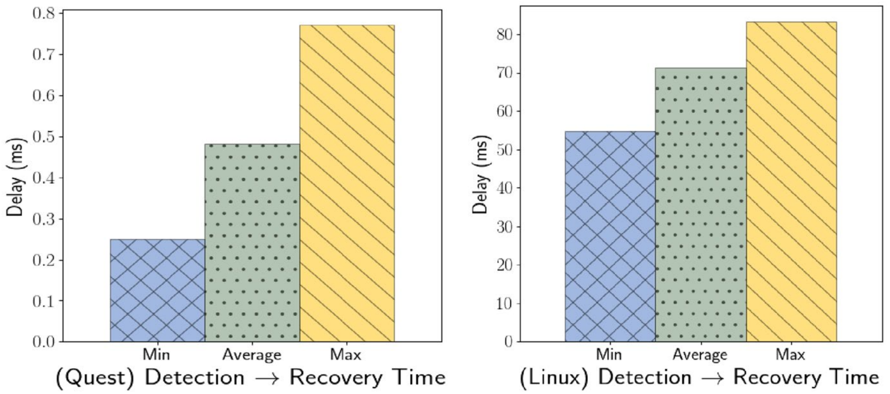


Fig. 15 E2E latencies from fault detection to flight recovery within Quest (*left*) and Linux (*right*)

represents the duration from the system mode change (Step-3 in Fig. 6) 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 2. 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 0.41 ms. This latency comprises the combined delays for Step-4a: 6.5μs and Step-4d: 0.4 ms, and falls well within the upper bound latency of 1 ms (1000 Hz) 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 and Kim (2013) reports the first response time to be 11 ms 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.

Figure 16 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 2.51 s 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

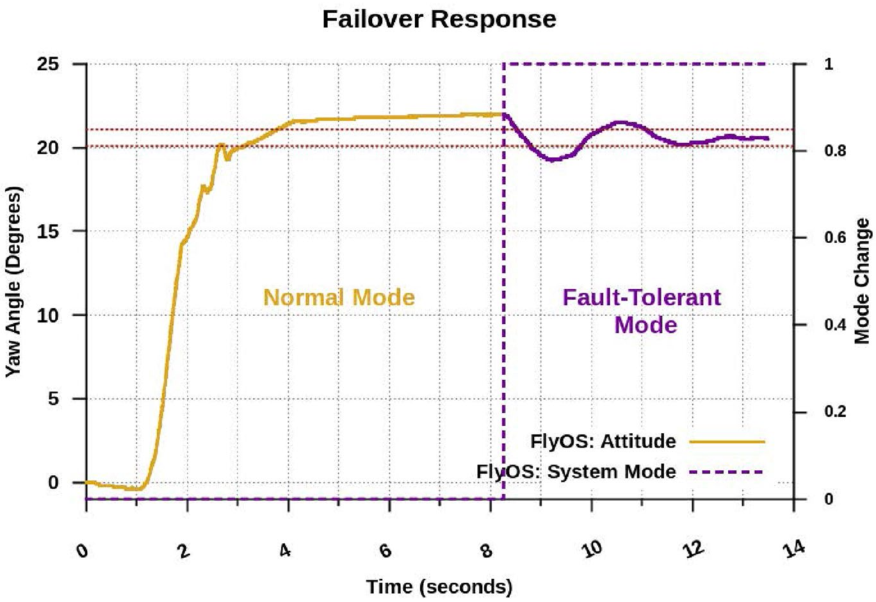


Fig. 16 Static image detection (Normal Mode) and Fault recovery hover stabilization (Fault-Tolerant Mode) with fault injection at 8.27 s

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.

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. Efforts are therefore underway to extend the dual-sandbox prototype implementation to a more general setting for supporting two or more RTOS sandboxes. Each sandbox would be qualified to act as a hot or cold standby to account for different fault scenarios.

In the next section we present different aspects of the FlyOS system as they relate to application and system level adaptability. In view of the flexible nature of the FlyOS framework, we also provide a design layout of a unique flight controller that showcases dynamic switching between different levels of process and partition criticalities.

4 Enhanced avionics: a case for adaptability and flexibility

Parallel flight controllers FlyOS's sandboxes encapsulate entire virtual machines. These system-level partitions enable unique and customizable combinations of software + hardware stacks to be statically configured and spawned by the hypervisor. Parallel partitions enable FlyOS to benefit from different flight control implementations, each in its own isolated container environment and each tuned to a specific flight profile or characteristic. Multiple safety-critical controller algorithms can therefore co-exist as application-level logic. The trusted hypervisor has the ability to dynamically engage and disengage these pre-configured controllers depending on the intended flight behavior, environmental conditions and required mission objectives. We utilized this feature of FlyOS to enable fail-over control in the presence of timing related faults in the primary Cleanflight flight controller. Details of our hypervisor-based fault tolerance mechanism were presented in Sect. 2.4.3.

FlyOS thus enhances the flexibility of the flight management system by enabling low-latency dynamic switch-over capability between flight controllers. This aligns with ARINC's multiple module schedules (ARINC Std. 653P2-4 2019) capability that allows different schedules to be set up for application modules. The feature facilitates module initialization, recovery from component failure, as well as interoperability between distinct implementations of the application.

Adaptable task-to-core affinities FlyOS sandboxes are configurable with single- or multicore CPU partitions. For example, our dual-sandbox working prototype features a hybrid setup with single-core Linux and SMP Quest featuring three cores. The real-time vCPU scheduler within Quest manages concurrent flight control tasks on the available cores of the quad-core aerial platform. Threads mapped to vCPUs are assigned to separate physical cores (PCPUs) based on their resource usage demands. Compute-bound tasks such as ATTITUDE and PID+MIXER can be assigned to separate physical cores compared to those that are I/O-bound, such as MOTOR and IMU. Alternatively, a mix of compute- and I/O-bound tasks can be assigned different cores to balance the computational load.

Initial task-to-core assignment decisions are informed by each task's individual requirements and resource contention with other tasks. Tasks can be migrated between cores at runtime depending on the active load-balancing profile of the system such as per-core power consumption, cache occupancy characteristics or per-core utilization.

FlyOS supports the construction of task pipelines that read sensory inputs, process corresponding data, and ultimately generate actuator output values. The system attempts to decouple the execution of tasks within the same pipeline, using non-blocking or asynchronous inter-task communication abstractions. This allows tasks to be treated as independently schedulable entities, assigned to individual vCPUs, which are in turn mapped to PCPUs with available compute capacity to achieve a feasible schedule.

Adaptive criticality FlyOS adds a new dimension to system flexibility by supporting criticality switching at the application and guest partition granularity. An application can modify its runtime behavior by changing the criticality level, and hence the scheduling priorities, of the constituent tasks based on some trigger condition or system parameter. Criticality adaptation is also enabled at the partition level via multiple system modes per guest sandbox.

Multicopter autopilots often need to alter their behavior and reconfigure their mission objectives when operating in varying environmental conditions (Farrukh and West 2020). To this end, we present the design of an environmentally-aware rate-adaptive flight controller application for the SMP Quest sandbox. This type of controller compensates for transient fluctuations in attitude relative to a target, caused by external disturbances such as wind.

Our purpose here is twofold. First, we highlight the ease with which avionic applications that originally target federated architectures can be redesigned with minimal effort and integrated to the FlyOS platform. We describe the integration of a dynamic flight controller, called sMARTflight (Farrukh and West 2020), into FlyOS. We then introduce two different flight mission modes and extend the task pipeline model of the critical flight control loop with task and sandbox criticality levels. We note that FlyOS opens opportunities for re-using applications and corresponding artifacts within isolated guest domains. FlyOS thereby ensures compatibility with legacy functions, while allowing application designs to take advantage of the multicore IMA architecture.

Secondly, we focus on criticality adaptability for the guest and hosted tasks under the influence of varying external conditions. This emphasizes the dynamic switchover capability between different safety-critical operating states for a sandbox system. FlyOS thus has the potential to change the operational behavior of the flight management system. Dynamically changing design assurance levels of the application and its operating environment presents an interesting challenge for certification of such system-level transformations (Annighoefer et al. 2019).

The design presented in Fig. 17 considers two separate task and sandbox *criticality levels* (or modes): { LO and HI}. Each task τ_i, executes with either low (LO) or high (HI) criticality. *Task criticality* (L_i) provides a measure of functional importance, or consequence of failure, to the overall flight objective. HI criticality is assigned to tasks that must operate correctly within hard real-time constraints of their budget, C_i and period, T_i, to maintain flight. LO criticality tasks, on the other hand, have minimal impact on the target flight behavior.

Similarly, a guest system is characterized with a *sandbox criticality* level, or operating mode, L_{sb}, depending on external factors such as wind. L_{sb} = LO for *Calm* (Normal) conditions, while a mode-switch to L_{sb} = HI occurs when adverse (e.g., *Windy* or Inclement) conditions occur. A switch is triggered as a direct consequence of attitude variations beyond a target threshold value in any axis of rotation: roll, pitch or yaw. Sandbox criticality therefore captures the influence of external disturbances on the internal state of the drone.

Two further *flight modes*, ANGLE and RATE, are used within the flight controller sandbox. These directly correspond to: (1) attitude lock, and (2) fast acrobatic maneuver, missions respectively, and are configured for the set-point generator process (Fig. 4) within Linux. The ANGLE flight mode stabilizes the copter in

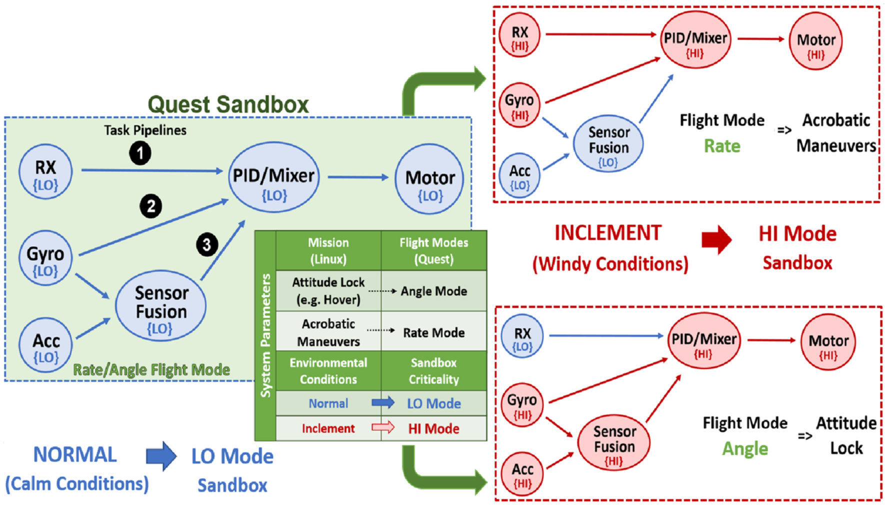


Fig. 17 Adaptive flight controller design with dynamic task and sandbox criticalities under calm and inclement environmental conditions. ANGLE and RATE flight modes represent configurable mission objectives

a steady-state (horizontal) hover attitude. In contrast, the RATE mode enables more nimble maneuvers such as rolls, flips or in-flight object tracking.

The active flight mode depends on the mission requirements. We note that flight control relies primarily on the feedback loop (Pipes 2 & 3 in Fig. 17) of the flight controller, when the ANGLE mode is active. Fusion of gyroscope (GYRO Task) and accelerometer (ACCEL Task) data from the IMU sensor, estimates the current orientation of the copter to achieve a level steady-state attitude (ATTITUDE Task). In contrast, RATE mode leads to precise rotations by reading just the gyroscope sensor and the mission update commands (Pipes 1 & 2 in Fig. 17) for more finer adjustments to the copter's angular velocity. This flight mode is useful in navigating close to a rough terrain or inside tight building spaces for search and rescue operations.

Collectively, the $\{flight, sandbox\}$ mode pair dynamically determine task criticalities under normal and inclement weather conditions, as illustrated in Fig. 17. In accordance with sMARTflight, a task's active criticality level allows us to directly associate one of the two rate-adaptation behaviors, *rate increase* (↑) or *rate decrease* (↓) with each task: a HI criticality mode increases task rate, whereas LO criticality mode decreases the execution rate.

Our rate-adaptation policy is built upon a key insight developed in sMARTflight that the response time performance of a flight controller is directly related to the rate of execution of its control loop tasks. Under *Calm* conditions, the Quest sandbox and all the flight control tasks operate in LO mode. An erroneous change in the drone's attitude state signals an adverse change in the external environment, which necessitates a finer granularity of control. The sandbox therefore switches to HI mode. This results in triggering a subset of the most relevant flight control tasks to undergo a LO → HI criticality transition based on the currently active flight mode. Consequently,

Table 6 Relationship between task time periods (T_i) and task criticality (L_i) per sandbox (sb) (L_{sb}) criticality mode

Normal ↔ Inclement conditions
$T_i\left\{L_i = \mathrm{LO}, L_{sb} = \mathrm{LO}\right\} \leq T_i\left\{L_i = \mathrm{LO}, L_{sb} = \mathrm{HI}\right\}$
$T_i\left\{L_i = \mathrm{LO}, L_{sb} = \mathrm{LO}\right\} > T_i\left\{L_i = \mathrm{HI}, L_{sb} = \mathrm{HI}\right\}$

all hi criticality tasks increase their rates of execution while other non-essential lo criticality tasks reduce their rates to dynamically compensate for the adverse effects of the environment.

lo mode tasks play a crucial role in protecting against potential system overloads in the hi sandbox mode, thus ensuring real-time schedulability for the entire taskset across all the cores. This enables FlyOS to support more feasible tasks schedules with a wider range of execution frequencies for critical flight control tasks. On return to normal conditions, the sandbox reverts back to lo mode, triggering the tasks to reset their rates to the original lo mode values. Table 6 summarizes the relationship between task periods for lo and hi criticality tasks in each sandbox mode.

The flight controller adapts to a changing environment using dynamic criticalities and task frequency adjustments. This ensures predictable low-latency flight control across different mission objectives and flight modes. Our design allows smartflight to run as a multi-vCPU flight controller application across multiple cores of the Quest sandbox. FlyOS therefore empowers smartflight with additional compute flexibility, autonomous mission capabilities, and run-time criticality adaptability.

In summary, the FlyOS framework presents an opportunity to extend multicopter flight characteristics with adaptive control capabilities for a centralized IMA platform.

5 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 (Cube Pilot), Intel's Ready-to-Fly (RTF) Drone (Brunner et al. 2019; Intel; Morales et al. 2020), Qualcomm's digital signal processors (DSPs) (Ardupilot; Qualcomm 2015, 2021), and various companion board solutions (Gu et al. 2018; Mejias et al. 2021; UAVOS). 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. (2018) use seL4 (VanderLeest 2018) 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 (Boniol and Wiels 2014; Rushby 1999; Watkins and Walter 2007). IMA in UAVs takes its inspiration from the commercial aerospace domain led by Airbus (Ramsey 2007) and Boeing (Jensen 2005; Watkins 2006), to employ temporal and spatial partitioning techniques in compliance with ARINC-653 software development standard.

Several mechanisms for an IMA host have emerged that target partitioning at either the application (Kang et al. 2016), kernel (Arcaro and de Oliveira 2015) or hypervisor level (VanderLeest and White 2015) of the consolidated flight management system (Han and Jin 2014). LynxOS-178 (Leiner et al. 2007) is a small partitioning kernel, which establishes encapsulated domains for applications, and schedules them on shared hardware in dedicated timeslots. Jo et al. (2019) 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 include those from commercial vendors such as VxWorks 653 (Ruan and Zhai 2014; Wind River Systems) and Green Hills' Integrity-178B (Software). Similarly AUTOBEST (Zuepke et al. 2015) and ARINC extended Linux (Han and Jin 2012) are some of the example systems originating from the academic community. These systems extend existing OSes 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 general avionics approached IMA's partitioning requirement at a deeper system-level by employing consolidating hypervisors. Examples of such systems are the ARINC-653 Hypervisor (VanderLeest 2010), MPSoC by DornerWorks (VanderLeest; VanderLeest and White 2015), XtratuM (Masmano et al. 2011) and Deos (Bloom and Sherrill 2020), to name just a few. These hypervisors allows multiple operating systems to run simultaneously as virtual machines or partitions on shared flight hardware. The hypervisor manages the entire IMA system and the hardware. As such, the hypervisor must be certified to the highest level of any of the hosted guests or applications, which may incur high certification costs.

In order to keep costs low, many of the aforementioned commercial vendors offer the hypervisor as a separate product to be used in conjunction with their flight certified RTOSes. These hypervisors are based on multiple independent levels of security (MILS) (Alves-Foss et al. 2006) and employ the separation-kernel approach to partitioning. However, these systems do not support ARINC features but instead are tailored to meet security requirements. LynxSecure (LYNX Software Technologies 2015) and VxWorks MILS (Wind River Systems) are two such proprietary systems, whose implementations are closed source. PikeOS (SYSGO 2015) and AIR (Craveiro et al. 2009) are two micro-kernels with support for a virtualization layer responsible for partitioning of resources between hosted guest operating systems. Partitioning hypervisor based approaches to IMA however, have only been deployed either in spacecraft or aircraft applications (Almeida and Prochazka 2009; Muttillo et al. 2019; Windsor et al. 2011).

State-of-the-art multicopters, on the other hand, employ traditional hypervisors like Xen (VanderLeest 2010), VMware and VirtualBox (Han and Jin 2011). 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 (Zhang and West 2006), which is needed for sensing, processing and actuation tasks in a flight controller.

Pérez and Gutiérrez (2017) and Pérez et al. 2017 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 (Crespo et al. 2010) 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.

6 Conclusions and 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. Using techniques described in Sect. 4, we also aim to evaluate real-time capabilities for adaptive flight control, and in-flight mission re-configurability. The goal in this case is to maintain flight stability in varied environmental conditions.

Acknowledgements This work 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

3D Robotics Inc. DroneKit Python. https://github.com/dronekit/dronekit-python. Accessed Oct 2021

3DR. Connecting to a vehicle. https://dronekit-python.readthedocs.io/en/latest/guide/connecting_vehicle.html. Accessed Oct 2021

Adams K, Agesen O (2006) A comparison of software and hardware techniques for x86 virtualization. ACM SIGOPS Oper Syst Rev 40:2–13

Almeida J, Prochazka M (2009) Safe and secure partitioning with PikeOS: towards integrated modular avionics in space. In: Noordwijk Ouwehand L (ed) Proceedings of DASIA 2009 data systems in aerospace, European Space Agency, Netherlands

Alves-Foss J, Harrison WS, Oman P, Taylor C (2006) The MILS architecture for high-assurance embedded systems. Int J Embed Syst 2:239–247

Andersson B, de Niz D, Klein M (2022) Satisfying real-time requirements of multicore software on ARINC 653: the issue of undocumented hardware. In: 2022 IEEE/AIAA 41st digital avionics systems conference (DASC), pp 1–10

Annighoefer B, Halle M, Schweiger A, Reich M, Watkins C, VanderLeest SH, Harwarth S, Deiber P (2019) Challenges and ways forward for avionics platforms and their development in 2019. In: 2019 IEEE/AIAA 38th digital avionics systems conference (DASC), pp 1–10

Arcaro LF, de Oliveira RS (2015) Lessons learned from the development of an ARINC 653 compatible operating system. In: IEEE 13th international conference on industrial informatics (INDIN), pp 221–226

Ardupilot: Archived: Qualcomm Snapdragon Flight Kit. https://ardupilot.org/copter/docs/common-qualcomm-snapdragon-flight-kit.html. Accessed Oct 2021

Ardupilot Autopilot: Ardupilot. https://ardupilot.org/

Ardupilot Autopilot: MAVLink basics. https://ardupilot.org/dev/docs/mavlink-basics.html. Accessed Oct 2021

ARINC Std. 653P1-5 (2019) Avionics application standard software interface, Part 1-Required Services. https://www.sae.org/standards/content/arinc653p1-5

ARINC Std. 653P2-4 (2019) Avionics application software standard interface, Part 2, Extended services. https://aviation-ia.sae-itc.com/standards/arinc653p2-4-653p2-4-avionics-application-software-standard-interface-part-2-extended-services

Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. In: ACM SIGOPS OSR

Beckert M, Gemlau KB, Ernst R (2017) Exploiting sporadic servers to provide budget scheduling for ARINC653 based real-time virtualization environments. In: Design, automation & test in Europe conference & exhibition (DATE)

Beckert M, Ernst R (2015) Designing time partitions for real-time hypervisor with sufficient temporal independence. In: 2015 52nd ACM/EDAC/IEEE design automation conference (DAC), pp 1–6

Betaflight Autopilot: Betaflight. https://betaflight.com/

Bloom G, Sherrill J (2020) Harmonizing ARINC 653 and realtime POSIX for conformance to the FACE technical standard. In: 2020 IEEE 23rd international symposium on real-time distributed computing (ISORC), pp 98–105

Boniol F, Wiels V (2014) Towards modular and certified avionics for UAV. J Aerosp Lab 8:1–8

Bregu E, Casamassima N, Cantoni D, Mottola L, Whitehouse K (2016) Reactive control of autonomous drones. In: 14th annual international conference on mobile systems, applications and services (MobiSys'16), pp 207–219

Brunner G, Szebedy B, Tanner S, Wattenhofer R (2019) The urban last mile problem: autonomous drone delivery to your balcony. In: 2019 international conference on unmanned aircraft systems (ICUAS), pp 1005–1012

Buczyński H, Cabaj K, Pisarczyk P (2022) Resource partitioning in phoenix-RTOS for critical and non-critical software for UAV systems. In: 2022 17th conference on computer science and intelligence systems (FedCSIS), pp 605–609

Spitzer CR, Ferrell U, Ferrel T (2015) Digital avionics handbook, 3rd edn. CRC Press, Taylor & Francis Group, Boca Raton

Cesarano C, Cotroneo D, De Simone L (2022) Towards assessing isolation properties in partitioning hypervisors. In: 2022 IEEE international symposium on software reliability engineering workshops (ISSREW), pp 193–200

Cheng Z, West R, Einstein C (2018) 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

Cleanflight Autopilot: Cleanflight. goo.gl/uCGmr4

Clifton D (2015) SPRACINGF3 flight controller manual (Revision 4). https://bit.ly/2Mx9dRV

Craveiro J, Rufino J, Schoofs T, Windsor J (2009) Flexible operating system integration in partitioned aerospace systems. In: Actas do INForum - Simposio de Informatica, pp 49–60

Crespo A, Ripoll I, Masmano M (2010) Partitioned embedded architecture based on hypervisor: the XtratuM approach. In: EDCC. IEEE Computer Society, pp 67–72

Cronk B Slack scheduling brings 100" systems. http://vita.mil-embedded.com/articles/slack-resource-utilization-safety-critical-systems/

Cube Pilot: The Cube Autopilot. https://bit.ly/3vPxbLo. Accessed Oct 2021

Danish M, Li Y, West R (2011) Virtual-CPU scheduling in the Quest operating system. In: 17th IEEE real-time and embedded technology and applications symposium (RTAS), pp 169–179

Delange J (2011) POK, An ARINC653-compliant Operating System Released under the BSD License

Domas C (2018) Hardware backdoors in x86 CPUs. https://i.blackhat.com/us-18/Thu-August-9/us-18-Domas-God-ModeUnlocked-Hardware-Backdoors-In-x86-CPUs-wp.pdf

DroneCode: Pixhawk Home. https://pixhawk.org/. Accessed Oct 2021

Dronecode Project: MAVLink Developer Guide. https://mavlink.io/en/. Accessed Oct 2021

European Union Aviation Safety Agency (EASA) (2012) MULCORS - Use of MULticore proCessORs in airborne Systems. Research Project EASA.2011/6. https://tinyurl.com/ye2863vu

FAA: United States Department of Transportation: Federal Aviation Authority. https://www.faa.gov/. Accessed July 2022

FAA, EASA: Technical implementation procedures for airworthiness and environmental certification. https://www.faa.gov/aircraft/air_cert/international/bilateral_agreements/baa_basa_listing/media/EUTIP_rev6.pdf. Accessed July 2022

Farrukh A, West R (2020) smARTflight: an environmentally-aware adaptive real-time flight management system. In: 32nd Euromicro Conference on Real-Time Systems (ECRTS)

Federal Aviation Administration ARD (2017) Technical Center: DOT/FAA/TC-16/51: assurance of multicore processors in airborne systems. http://www.tc.faa.gov/its/worldpac/techrpt/tc16-51.pdf

Federal Aviation Administration CAST (2016) Position paper: multi-core processors. https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32a.pdf

Golchin A, Cheng Z, West R (2018) Tuned pipes: end-to-end throughput and delay guarantees for USB devices. In: 39th IEEE real-time systems symposium (RTSS)

Golchin A, Sinha S, West R (2020) Boomerang: real-time I/O meets legacy systems. In: 2020 IEEE real-time and embedded technology and applications symposium (RTAS). IEEE, pp 390–402

Green Hills Software Inc (2010) INTEGRITY-178B Separation Kernel

Gu Q, Michanowicz DR, Jia C (2018) Developing a modular unmanned aerial vehicle (UAV) platform for air pollution profiling. Sensors 18:4363

Han S, Jin H-W (2011) Full virtualization based ARINC 653 partitioning. In: 2011 IEEE/AIAA 30th digital avionics systems conference

Han S, Jin H-W (2012) Kernel-level ARINC 653 partitioning for Linux. In: SAC '12: Proceedings of the 27th annual ACM symposium on applied computing, pp 1632–1637

Hwang J, Zeng S, Wu Fy, Wood T (2013) A component-based performance comparison of four hypervisors. In: 2013 IFIP/IEEE international symposium on integrated network management (IM 2013), pp 269–276

iNAV Autopilot: iNAV. https://github.com/iNavFlight/inav/wiki

Intel: Github documentation wiki for Intel ready to fly drone. https://github.com/intel-aero/meta-intel-aero/wiki/02-Initial-setup. Accessed Oct 2021

Intel: Intel Aero Compute Board. https://ark.intel.com/content/www/us/en/ark/products/97178/intel-aero-compute-board.html. Accessed Oct 2021

Intel: Intel Aero Vision Accessory Kit. https://ark.intel.com/content/www/us/en/ark/products/97175/intel-aero-vision-accessory-kit.html. Accessed Oct 2021

Intel: Intel RealSense Github. https://github.com/IntelRealSense/librealsense. Accessed Oct 2021

Intel: Overview of Intel Ready to Fly Drone. https://intel.ly/3b8WwGz. Accessed Oct 2021

Intel: Support for Intel RealSense Camera. https://intel.ly/3uX0KKe. Accessed Oct 2021

Jensen D (2005) B787 cockpit: Boeing's bold move. https://www.aviationtoday.com/2005/11/01/b787-cockpit-boeings-bold-move/

Jeong E-H, Kim J-G (2013) S/W fault-tolerant OFP system for UAVs based on partition computing. In: 2013 international conference on electronic engineering and computer science

Jiang Y (2016) [V4,4/4] Utilize the VMX preemption timer for TSC deadline timer. https://patchwork.kernel.org/project/kvm/patch/1465852801-6684-5-git-send-email-yunhong.jiang@linux.intel.com/#19325443. Accessed Oct 2021

Jo H-C, Park J-K, Jin HW, Yoon H-S, Lee SH (2019) Portable and configurable implementation of ARINC-653 temporal partitioning for small civilian UAVs. IEEE Access 7:142478–142487

Kamel M, Alexis K, Achtelik M, Siegwart R (2015) Fast nonlinear model predictive control for multicopter attitude tracking on SO(3). In: 2015 IEEE conference on control applications (CCA), pp 1160–1166

Kang Q, Yuan C, Wei X, Gao Y, Wang L (2016) A user-level approach for ARINC 653 temporal partitioning in seL4. In: ISSSR, pp 106–110

Klein G, Andronick J, Fernandez M, Kuz I, Murray T, Heiser G (2018) Formally verified software in the real world. Commun ACM 61(10):68–77

Leiner B, Schlager M, Obermaisser R, Huber B (2007) A comparison of partitioning operating systems for integrated systems. In: International conference on computer safety, reliability, and security. Springer, pp 342–355

Li Y, West R, Missimer ES (2014a) A virtualized separation kernel for mixed criticality systems. In: Hirzel M, Petrank E, Tsafrir D (eds) 10th ACM SIGPLAN/SIGOPS international conference on virtual execution environments, VEE '14, pp 201–212

Li Y, West R, Cheng Z, Missimer E (2014b) Predictable communication and migration in the Quest-V separation kernel. In: 35th IEEE real-time systems symposium (RTSS), Rome, Italy

Liszewski A NASA's Supercomputers Reveal the Incredible Turbulence Produced By a Drone. https://gizmodo.com/nasas-supercomputers-reveal-the-incredible-turbulence-p-1791179507. Accessed Oct 2021

Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. J ACM 20:46–61

LYNX, AFuzion: Preview: CAST-32A Significance & Implications. Technical White Pape. https://www.lynx.com/embedded-systems-learning-center/cast-32a-significance-and-implications-legacy. Accessed July 2022

Lynx Software Technologies: LynxSecure Separation Kernel Hypervisor. https://www.lynx.com/products/lynxsecure-separation-kernel-hypervisor

LYNX Software Technologies (2015) LynxSecure embedded hypervisor and separation kernel. http://www.lynx.com/products/hypervisors/

Martins J, Tavares A, Solieri M, Bertogna M, Pinto S (2020) Bao: a lightweight static partitioning hypervisor for modern multi-core embedded systems. In: Workshop on next generation real-time embedded systems (NG-RES 2020), vol 77. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Wadern, Germany, pp 3–1314

Masmano M, Valiente Y, Balbastre P, Ripoll I, Crespo A (2011) ARINC-653 APEX based on XtratuM. In: XIV Jornadas de Tiempo

MAVLink Router: MAVLink Router. https://github.com/mavlink-router/mavlink-router. Accessed Feb 2023

McDermott J, Montrose B, Li M, Kirby J, Kang M (2012) Separation virtual machine monitors. In: Proceedings of the 28th annual computer security applications conference. ACSAC '12. Association for Computing Machinery, Orlando, Florida, USA, pp 419–428

Mejias L, Diguet J-P, Dezan C, Campbell D, Kok J, Coppin G (2021) Embedded computation architectures for autonomy in unmanned aircraft systems (UAS). Sensors 21:1115

Mercer CM, Savage S, Tokuda H (1993) Processor capacity reserves for multimedia operating systems. In: Technical Report. Carnegie Mellon University, Pittsburgh, USA

Missimer E, Missimer K, West R (2016) Mixed-criticality scheduling with I/O. In: 28th Euromicro Conference on Real-Time Systems (ECRTS)

Missimer E, West R, Li Y (2014) Distributed real-time fault tolerance on a virtualized multi-core system. In: 10th annual workshop on operating systems platforms for embedded real-time applications (OSPERT)

Morales T, Sarabakha A, Kayacan E (2020) Image generation for efficient neural network training in autonomous drone racing. In: 2020 International joint conference on neural networks (IJCNN), pp 1–8

Muttillo V, Tiberi L, Pomante L, Serri P (2019) Benchmarking analysis and characterization of hypervisors for space multicore systems. J Aerosp Inf Syst 16:500–511

OscarLiang.com: ESC Firmware and Protocols Overview. https://oscarliang.com/esc-firmware-protocols/. Accessed July 2017

OscarLiang.com (2017) Custom motor mixing multirotor what calculate uses. https://www.oscarliang.com/custom-motor-output-mix-quadcopter

OSDEV.org (2019) Timer interrupt sources. https://wiki.osdev.org/Timer_Interrupt_Sources

OSRTOS: MaRTE RTOS. https://www.osrtos.com/rtos/marte/

Prisaznuk PJ (2008) ARINC 653 role in integrated modular avionics (IMA). In: IEEE/AIAA 27th digital avionics systems conference, pp 1–511510

Park S, Song D, Jang H, Kwon M-Y, Lee S-H, Kim H-K, Kim H (2019) Interference analysis of multicore shared resources with a commercial avionics RTOS. In: 2019 IEEE/AIAA 38th digital avionics systems conference (DASC), pp 1–10

Parkinson P (2018) Safety-critical software development for integrated modular avionics. https://events.windriver.com/wrcd01/wrcm/2015/02/Safety-Critical-Software-Development-for-Integrated-Modular-Avionics-White-Paper-1.pdf

Parkinson P, Kinnan L (2015) Safety-critical software development for integrated modular avionics. Embed Syst Eng 11:40–41

Pérez H, Gutiérrez JJ (2017) Handling heterogeneous partitioned systems through ARINC-653 and DDS. Comput Stand Interfaces 50:258–268

Pérez H, Gutiérrez JJ, Peiró S, Crespo A (2017) Distributed architecture for developing mixed-criticality systems in multi-core platforms. J Syst Softw 123:145–159

PX4 Autopilot: PX4 [Home]. http://px4.io/

Qualcomm (2015) First public demo of snapdragon flight robotics dev platform in one of world's smallest 4K drones. https://www.qualcomm.com/news/onq/2015/09/10/first-public-demo-snapdragon-flight-robotics-dev-platform-one-worlds-smallest-4k. Accessed Oct 2021

Qualcomm (2017) Qualcomm snapdragon flight kit. https://www.intrinsyc.com/vertical-development-platforms/qualcomm-snapdragon-flight/

Qualcomm (2021) Journey to mars: how our collaboration with jet propulsion laboratory fostered innovation. https://www.qualcomm.com/news/onq/2021/03/17/journey-mars-how-our-collaboration-jet-propulsion-laboratory-fostered-innovation

Radio Technical Commission for Aeronautics (RTCA) Std (2011a) DO-178C/ED-12C software considerations in airborne systems and equipment certification

Radio Technical Commission for Aeronautics (RTCA) Std (2011b) RTCA DO-248C/EUROCAE ED-94C supporting information for DO-178C and DO-278A

Radio Technical Commission for Aeronautics (RTCA) Std (2014) DO-326A airworthiness security process specification

Radio Technical Commission for Aeronautics (RTCA) Std (2018) DO-356A airworthiness security methods and considerations

Ramsauer R, Kiszka J, Lohmann D, Mauerer W (2017) Look mum, no VM exits! (Almost). http://arxiv.org/abs/1705.06932

Ramsey JW (2007) Integrated modular avionics: less is more. https://www.aviationtoday.com/2007/02/01/integrated-modular-avionics-less-is-more/

ROBOTmaker: Real-Time Graphical Representation S.BUS Protocol. http://www.robotmaker.eu/ROBOTmaker/quadcopter-3d-proximity-sensing/sbus-graphical-representation

Ruan W, Zhai Z (2014) 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, pp 388–393

Rushby J (1999) Partitioning for avionics architectures: requirements, mechanisms and assurance. In: NASA Contractor Report CR-1999-209347, NASA Langley Research Center

Rushby J (2002) Model checking Simpson's four-slot fully asynchronous communication mechanism. In: Computer science laboratory–SRI International, Tech. Rep. Issued

Rushby JM (1981) Design and verification of secure systems. In: 8th ACM symposium on operating systems principles, pp 12–21

Technology SC (2014) Jailhouse Partitioning Hypervisor. https://github.com/siemens/jailhouse

Han S, Jin H-W (2014) Resource partitioning for integrated modular avionics: comparative study of implementation alternatives. Softw. Pract. Exper. 44(12):1441–1466

Madgwick SOH, Harrison AJL, Vaidyanathan R (2011) Estimation of IMU and MARG orientation using a gradient descent algorithm. In: International conference on rehabilitation robotics (IEEE-ICORR), pp 1–7

Sanchez-Cuevas P, Heredia G, Ollero A (2017) Characterization of the aerodynamic ground effect and its influence in multirotor control. Int J Aerosp Eng 2017

Schildermans S, Aerts K, Shan J, Ding X (2021) Paratick: reducing timer overhead in virtual machines. In: 50th international conference on parallel processing (ICPP). Association for Computing Machinery (ACM), pp 1–10

Silva C, Tatibana C (2014) MultIMA- multi-core in integrated modular avionics. In: DAta systems in aerospace (DASIA)

Simpson HR (1990) Four-slot fully asynchronous communication mechanism. IEEE Comput Digit Techn 137:17–30

Sinha S, West R (2021) Towards an integrated vehicle management system in DriveOS. ACM Trans Embed Comput Syst 20:1–24 (Association for Computing Machinery (ACM))

Software GH. INTEGRITY-178 tuMP RTOS: safety-critical & security-critical RTOS. https://www.ghs.com/products/safety_critical/integrity_178_tump.html. Accessed Feb 2023

Spitzer CR (2006) RTCA DO-297/EUROCAE ED-124 integrated modular avionics (IMA) design guidance and certification considerations

Steinberg U, Kauer B (2010) NOVA: a microhypervisor-based secure virtualization architecture. In: Proceedings of the 5th European conference on computer systems (Eurosys). Association for Computing Machinery (ACM), pp 209–222

SYSGO (2015) SYSGO PikeOS hypervisor. http://www.sysgo.com/products/pikeos-rtos-and-1462virtualization-concept

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

VanderLeest SH Benefits and Implications of an ARINC 653 Hypervisor. https://dornerworks.com/about/whitepapers/arinc-653-benefits-implications/

VanderLeest SH (2010) ARINC 653 hypervisor. In: 29th Digital avionics systems conference, pp 5–215220

VanderLeest SH (2014) Taming interrupts: deterministic interrupts in an ARINC 653 environment. In: 33rd Digital avionics systems conference (DASC). IEEE/AIAA

VanderLeest SH (2016) The open source, formally-proven seL4 microkernel: considerations for use in avionics. In: IEEE/AIAA 35th Digital avionics systems conference (DASC)

VanderLeest SH (2017) Designing a future airborne capability environment (FACE) hypervisor for safety and security. In: IEEE/AIAA 36th Digital avionics systems conference (DASC)

VanderLeest SH, White D (2015) MPSoC hypervisor: the safe & secure future of avionics. In: IEEE/AIAA 34th Digital avionics systems conference (DASC)

VanderLeest SH (2018) Is formal proof of seL4 sufficient for avionics security? IEEE Aerosp Electron Syst Mag 33(2):16–21

VanderLeest SH, Matthews DC (2021) Incremental assurance of multicore integrated modular avionics (IMA). In: 2021 IEEE/AIAA 40th Digital avionics systems conference (DASC), pp 1–9

VanderLeest SH, Thompson SR (2020) Measuring the impact of interference channels on multicore avionics. In: 2020 AIAA/IEEE 39th Digital avionics systems conference (DASC), pp 1–8

Viola P, Jones MJ (2001) Rapid object detection using a boosted cascade of simple features. In: IEEE computer society conference on computer vision and pattern recognition

Wang Z, Jiang X (2010) HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity. In: 2010 IEEE symposium on security and privacy, pp 380–395

Watkins CB (2006) Integrated modular avionics: managing the allocation of shared intersystem resources. In: 25th digital avionics systems conference, pp 1–12

Watkins CB, Walter R (2007) Transitioning from federated avionics architectures to integrated modular avionics. In: 2007 IEEE/AIAA 26th Digital avionics systems conference, pp 2–112110

West R, Li Y, Missimer E (2012) Time management in the Quest-V RTOS. In: 8th Annual workshop on operating systems platforms for embedded real-time applications (OSPERT)

West R, Li Y, Missimer E, Danish M (2016) A virtualized separation kernel for mixed-criticality systems. In: ACM transactions on computer systems. Association for Computing Machinery (ACM), New York, NY, USA, vol 34, pp 8–1841

Wind River Systems I VxWorks 653 Multi-core Edition Product Overview. https://www.windriver.com/resource/vxworks-653-product-overview. Accessed Feb 2023

Wind River Systems I. Wind River Introduces VxWorks MILS Platform Conformant to Separation Kernel Protection Profile. https://www.windriver.com/news/press/news-11742. Accessed Feb 2023

Windsor J, Eckstein K, Mendham P, Pareaud T (2011) Time and space partitioning security components for spacecraft flight software. In: 2011 IEEE/AIAA 30th Digital avionics systems conference

X-IO technologies: open source IMU and AHRS algorithms. https://x-io.co.uk/open-source-imu-and-ahrs-algorithms/

Ye Y, West R, Zhang J, Cheng Z (2016) MARACAS: a real-time multicore VCPU scheduling framework. In: 37th IEEE real-time systems symposium (RTSS)

Zhang Y, West R (2006) Process-aware interrupt scheduling and accounting. In: 27th IEEE real-time systems symposium (RTSS)

Zuepke A, Bommert M, Lohmann D (2015) AUTOBEST: a united AUTOSAR-OS and ARINC 653 Kernel. In: 21st IEEE real-time and embedded technology and applications symposium, pp 133–144

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Anam Farrukh received the Master of Science (M.S.) in electrical engineering from Lahore University of Management Sciences (LUMS), Lahore, Pakistan in 2014. She is currently pursuing the Ph.D degree in computer science at Boston University under the supervision of Prof. R. West. She designs and develops software architectures for adaptable and autonomous vehicle management systems in the avionics and automotive domains. Her research aims to address the safety, predictability, and efficiency challenges of these modern-day cyber-physical systems. Her goal is to empower the next-generation of consolidated vehicle software with criticality awareness, time-aware power management and low-latency fault-tolerant capabilities. She believes that temporal and functional correctness are key to safe operation of all modern software-defined vehicles.

Richard West is a Professor in the Computer Science Department at Boston University. He holds a PhD and MS in Computer Science from the Georgia Institute of Technology, USA, and an MEng in Microelectronics and Software Engineering from the University of Newcastle-upon-Tyne, UK. He is also the Chief Software Architect at Drako Motors, a company developing electric cars and state-of-the-art vehicle management systems. His research works focuses on real-time and embedded operating systems, addressing issues concerning safety and predictability. He has studied real-time scheduling and resource management, cache-aware performance of multicore processors, and machine virtualization, among other topics. He is currently leading the development of the Quest RTOS for multicore processors. Its sister system, Quest-V, is a partitioning hypervisor, which provides efficient, predictable and safe execution of guests. Quest-V is used in the Drako Motors DriveOS production vehicle management system.