aerospace

MDPI

Article

A PRTOS-Based Partitioned Runtime Architecture for Integrated Avionics of Near-Space Airships: Multicore Porting and Fault-Containment Validation

Yong Hao [1,2], Zhaojie Li [1,2,*], Yanchu Yang [1,2], Jianghua Zhou [1] and Baocheng Wang [1]

[1] Aerospace Information Research Institute, Chinese Academy of Sciences, Beijing 100094, China; haoyong@aircas.ac.cn (Y.H.); yangyc@aircas.ac.cn (Y.Y.); zhoujh@aircas.ac.cn (J.Z.); wangbc@aircas.ac.cn (B.W.)
[2] School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, Beijing 100190, China
* Correspondence: lizhaojie@aircas.ac.cn

Abstract

Integrated avionics for near-space airships increasingly consolidate multiple functions on shared multicore platforms, creating risks of cross-functional interference and fault propagation. This study develops a multicore static-partitioning architecture based on the PRTOS Type-1 hypervisor and implements static mappings among four Linux guest partitions, processor cores, independent memory regions, and devices on an Intel Atom x6425RE platform. Fault-free baseline, partition-local computational overload, 16 MiB cyclic memory access, inter-partition communication, and native-Linux multicore-propagation experiments evaluate response time, deadline misses, and communication completeness. Under PRTOS, increasing pressure causes deadline misses in the faulty partition, while the other partitions retain zero misses. Severe pressure reduces communication completeness of the faulty partition but produces no integrity error or residual backlog on the other channels. Under native Linux, one functional anomaly expands to multiple cores and causes deadline misses in all tasks. PRTOS static partitioning therefore bounds the propagation of local faults and provides an engineering basis for isolated deployment of integrated airship–avionics functions.

Keywords: near-space airship; integrated avionics; PRTOS; embedded hypervisor; multicore partitioning; fault containment

Check for updates

Received: 21 July 2026
Revised: 7 August 2026
Accepted: 15 August 2026
Published: 20 August 2026

Correction Statement: This article has been republished with a minor change. The change does not affect the scientific content of the article and further details are available within the backmatter of the website version of this article.

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Near-space airships offer long endurance, wide mission coverage, and substantial payload volume. Their large hull dimensions, long compartment span, distributed sensors and actuators, and limited opportunities for human intervention during long-duration operation, however, impose distinctive avionics requirements [1,2]. Flight control, airship management, safety control, health monitoring, mission payloads, and external communications must operate continuously under strict mass, power, and volume constraints. Conventional federated avionics establishes clear physical fault boundaries through a one-function/one-computer arrangement, but it also increases equipment count, interface diversity, cross-compartment cabling, and underutilized computing capacity. With the development of integrated modular avionics (IMA) and open modular platforms, multiple functions increasingly share common computing, storage, and interface resources [3,4].

Shared computing improves resource utilization and functional scalability, but it also changes the mechanisms by which faults propagate. When flight control, airship management, and mission payload functions execute merely as processes or threads under a common general-purpose operating system, they still share the kernel scheduler, device drivers, and system services. An abnormal non-critical application may create high-priority threads, continuously occupy processors, or trigger driver faults, thereby crossing the functional boundaries defined during architecture design. Integration therefore requires more than loading multiple applications onto one computer; temporal and spatial partitioning must re-establish software execution boundaries consistent with criticality levels and fault consequences. DO-297 defines the responsibilities of IMA platforms and applications from the development and certification perspective [3], while ARINC 653 provides partition management, process scheduling, inter-partition communication, and health-monitoring services for application isolation on shared platforms [4,5].

Recent IMA research has expanded from single-core static configurations to multicore partition scheduling, automated configuration generation, and dynamic reconfiguration. Representative studies address platform-level IMA modeling and automation toolchains [6,7], optimization of ARINC 653 multicore partition schedules [8,9], cache partitioning and shared-resource interference [10], and model-driven configuration generation [11–13]. High-assurance microkernels such as seL4 emphasize formal verification and a minimal trusted computing base [14]. Implementation-level comparisons of temporal and spatial partitioning show that isolation guarantees, resource utilization, and analyzability must be considered jointly [15], while resource partitioning for critical and non-critical UAV software further demonstrates the need to explicitly control processor and memory boundaries [16]. Partitioning hypervisors such as XtratuM provide static-partitioning prototypes for embedded aerospace systems [17], and surveys and multicore-avionics studies emphasize shared-resource interference and validation of runtime boundaries [18,19]. Recent work also strengthens spatial isolation in mixed-criticality systems through runtime memory ownership [20]. Nevertheless, three gaps remain. First, much of the literature focuses on scheduling models, certification frameworks, or general virtualization mechanisms rather than engineering adaptation for shared near-space airship computing platforms. Second, some prototypes remain at the simulation or general-purpose development stage and do not document complete multicore startup, memory mapping, and independent serial port adaptation on real embedded hardware. Third, successful porting is often demonstrated only by booting guest systems, without partition-internal fault injection and a corresponding shared-kernel comparison.

To address shared computing and fault isolation requirements in near-space airship integrated avionics, this study uses the PRTOS Type-1 hypervisor to construct a four-partition static runtime prototype on an Intel Atom x6425RE platform and performs progressively extended experiments covering a fault-free baseline, pure computational load, 16 MiB cache/memory pressure, inter-partition communication, and native-Linux multicore propagation. The principal contributions are as follows: (1) a static mapping among avionics functions, guest partitions, processors, memory, and peripheral resources converts architectural functional boundaries into runtime resource boundaries; (2) multicore startup, hardware-virtualization enablement, guest-image packaging, vCPU binding, and UART adaptation establish a reproducible multi-partition environment; and (3) processor overload, composite cache/memory pressure, communication service, and shared-kernel multicore-propagation scenarios are evaluated using deadline misses, response-time bounds, message completeness, and buffer state to characterize the scope and limitations of PRTOS fault containment.

2. Software-Partitioning Requirements and Overall Architecture

2.1. From Functional Modularity to Runtime Partitioning

Near-space airship avionics functions differ in period, criticality, resource demand, and evolution rate. Flight control requires short-period execution and explicit deadlines; safety control and airship management process propulsion, energy, buoyancy, and equipment status; system management functions aggregate state and support maintenance; and mission payload software has variable loads and frequent updates. If these functions share one operating system directly, the common kernel and global scheduler couple their fault domains [15,20].

Functional modularity must therefore be enforced at both architectural and runtime levels. The former defines functional composition, deployment, and interaction, whereas the latter requires processor budgets, independent memory, controlled device access, and explicit communication channels. Table 1 maps six airship–avionics design principles to their runtime requirements and partitioning mechanisms.

Table 1. Correspondence between embedded virtualization and airship–avionics design principles.

Design Principle	Software-Runtime Requirement	Partitioning Mechanism
Functional modularity	Functional modules retain explicit runtime boundaries	Map mission functions to relatively independent software partitions
Priority protection of critical functions	Non-critical tasks must not consume deterministic resources assigned to critical tasks	Assign an independent core or a defined processor budget to critical partitions such as flight control
Separation of core and mission functions	Functions of different criticality levels should not share one fault domain	Deploy critical control, safety/airship management, and mission payloads separately
Controlled fault effects	A local anomaly must not spread across the computing platform	Restrict the processors, memory, and devices accessible to the faulty task
Integrated resource utilization	Reduce the number of dedicated computing units	Allow multiple software partitions to share a unified hardware platform
Functional coordination and evolution	Support interaction and local update while retaining isolation	Use controlled communication and partition-level loading, replacement, and recovery

2.2. Resource Model and Isolation Conditions

Let F denote the set of avionics functional modules and P the set of software partitions. Their definitions and mapping relationship are given in Equations (1) and (2), respectively:

$$\begin{aligned} F &= \{f_1, f_2, \ldots, f_m\} \\ P &= \{P_0, P_1, \ldots, P_{n-1}\} \end{aligned} \tag{1}$$

$$\Phi : F \rightarrow P \tag{2}$$

The mapping Φ considers computing and storage demand, criticality, task periods, device privileges, interaction intensity, and fault impact. Functions with different criticality, privileges, or fault consequences should be separated, whereas closely interacting functions with similar resource and criticality characteristics may be co-located when isolation requirements remain satisfied [17,20].

The resource set assigned to partition P_i is expressed as

$$R_i = \{C_i, M_i, I_i, E_i\} \tag{3}$$

where C_i, M_i, I_i, and E_i denote processor, memory, interrupt, and peripheral resources, respectively. Statically exclusive resources must satisfy the mutual-exclusion condition in Equation (4):

$$\begin{aligned} C_i \cap C_j = \emptyset,\ M_i \cap M_j = \emptyset \\ I_i \cap I_j = \emptyset,\ E_i \cap E_j = \emptyset,\ i \neq j \end{aligned} \tag{4}$$

Shared communication buffers, network interfaces, and storage are arbitrated by the hypervisor, a management partition, or a preconfigured service; guests cannot directly access another partition's private resources. This follows the static authorization and controlled sharing principles of safety-critical hypervisors [18,20].

Figure 1 organizes the runtime into functional, partition, hypervisor, and hardware-resource layers. System management, flight control, safety/airship management, and mission-payload functions are assigned to Linux guests P0–P3. PRTOS directly manages partition creation, vCPU binding, EPT isolation, interrupt routing, and peripheral authorization. The Intel Atom x6425RE platform provides four cores, independent partition memory regions, VMX/EPT, four UARTs, APIC, and HPET. Static configuration of these resources establishes the execution and fault-containment boundaries.

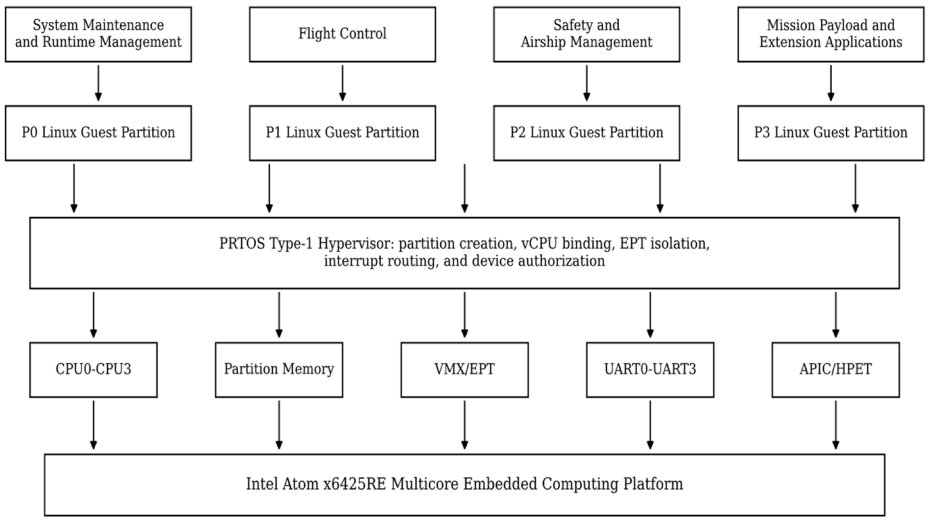


Figure 1. PRTOS-based partitioned runtime architecture for integrated avionics of near-space airships.

2.3. Hypervisor Selection and Scope of Applicability

Safety-critical virtualization platforms must be assessed by architecture, static resource control, multicore and guest support, device management, source availability, and portability. Type-2 hypervisors depend on a host operating system, so host scheduling, kernel, or driver faults may affect several guests; Xen and KVM timing also depends on timer and interrupt configuration [21]. Type-1 and static-partitioning hypervisors establish resource boundaries closer to startup, while Jailhouse and Bao preallocate processors, memory, and devices to reduce runtime intervention [22,23]. Table 2 compares representative schemes and their suitability for this study.

PRTOS was selected as an open-source, lightweight Type-1 hypervisor for real-time and safety-critical systems. Its separation-kernel design combines static allocation of processors, memory, and I/O with predefined cyclic scheduling. The available core, configuration tools, BAIL library, examples, and documentation support reproducible adaptation [24]. Source access also enables investigation of multicore startup, memory mapping, interrupt routing, and UART adaptation.

Table 2. Comparison of representative embedded virtualization schemes.

Scheme	Architecture and Partitioning Capability	Principal Characteristics	Suitability for This Study
VMware Workstation/VirtualBox	Type-2; limited static-partitioning capability	Easy deployment; suitable for software development and testing	Not suitable as the primary isolation basis for critical avionics functions
KVM	Linux-kernel based; moderate static-partitioning capability	Rich ecosystem and broad device support	The common runtime base includes a complete Linux kernel
Xen	Type-1 with a management domain; supports static partitioning	Feature-rich and supports multiple guest systems	The management domain and shared software base are relatively large
Jailhouse/Bao/ACRN	Static-partitioning or embedded-virtualization oriented	Suitable for static allocation of multicore resources	Requires a target-specific porting foundation
Commercial avionics hypervisor	Strong safety-critical partitioning capability	Mature engineering process and certification support	Licensing, source access, and low-level modification are restricted
PRTOS	Type-1 partitioning hypervisor with strong static configuration	Clear architecture; x86 virtualization and Linux guests	Matches the multicore-porting and fault-containment prototype requirements

3. PRTOS Multicore Porting and Partition Implementation

3.1. Hardware Platform and Four-Partition Resource Mapping

The target platform uses a four-core Intel Atom x6425RE with Intel VT-x and EPT support [25,26] and four serial ports for the P0–P3 Linux guests. To define an explicit processor fault boundary, the prototype adopts the one-core/one-partition and one-UART/one-partition mapping in Equation (5):

$$P_i \rightarrow \{\mathrm{CPU}_i, M_i, U_i\} \quad i = 0, 1, 2, 3 \tag{5}$$

In Equation (5), M_i is the authorized memory region and U_i the dedicated UART of partition P_i. Because cores are not temporally multiplexed, threads created within a guest remain on its assigned core. Table 3 lists the resulting core, UART, functional role, and design purpose of each partition.

Table 3. Functional and hardware resource configuration of the four partitions.

Partition	Physical Core	UART	Representative Functional Role	Design Purpose
P0	CPU0	ttyS0	System maintenance, runtime management, and external communication	Provide system management and state aggregation
P1	CPU1	ttyS1	Flight control	Validate independent execution of a high-frequency critical task

Table 3. *Cont.*

Partition	Physical Core	UART	Representative Functional Role	Design Purpose
P2	CPU2	ttyS2	Safety control, airship management, and integrated processing	Validate high computational load and local fault injection
P3	CPU3	ttyS3	Mission payload and extension applications	Validate isolation of low-frequency and non-critical functions

Figure 2 shows the four static mapping chains: P0/vCPU0/CPU0/M0/ttyS0 for system management and communication; P1/vCPU1/CPU1/M1/ttyS1 for flight control; P2/vCPU2/CPU2/M2/ttyS2 for safety and airship management; and P3/vCPU3/CPU3/M3/ttyS3 for mission-payload processing. All vCPU, memory, and UART authorizations are fixed before startup.

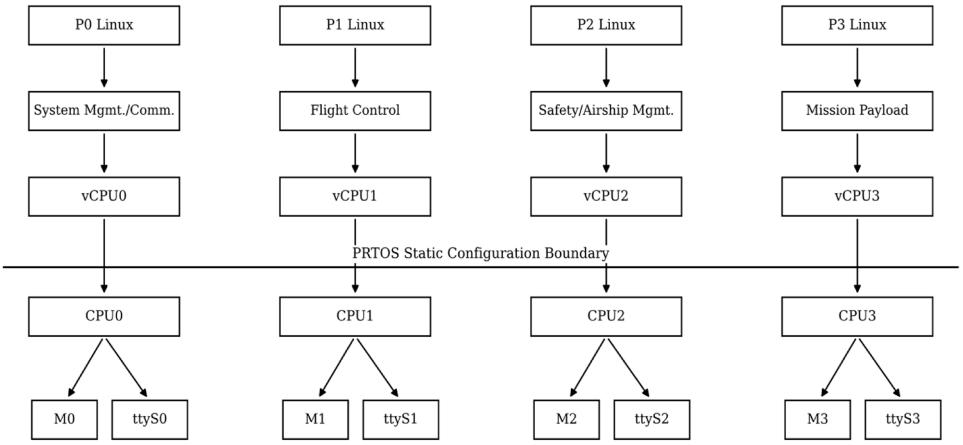


Figure 2. Static mapping of four partitions, four cores, and four UARTs.

3.2. System Build, Startup, and Hardware Adaptation

The system comprises the PRTOS hypervisor, a static configuration, and four Linux guest images. The configuration defines processor mappings, physical-memory ranges, device privileges, and entry points. After the guest kernels and root file systems are converted to PEF images, prtospack creates the system container and rswbuild generates resident_sw, following the PRTOS toolchain [24].

Figure 3 summarizes two stages. The build chain converts and packages the PRTOS source, static configuration, and P0–P3 guest images into resident_sw. The startup chain detects processor features, initializes HPET/LAPIC, wakes CPU1–CPU3, enables VMX/EPT, creates partitions, binds vCPUs, and loads the guests.

VMX control structures and EPT page tables provide x86 hardware-assisted virtualization [26]. The key porting constraint is that each guest's load address, kernel decompression area, root file system, and page tables remain within its authorized memory. Overlapping load regions or decompression beyond an EPT mapping can terminate or stall a guest. The reserved region, guest memory, kernel addresses, and root-file-system addresses were therefore coordinated, and each guest boundary was verified independently [24].

The serial devices ttyS0–ttyS3 are assigned to P0–P3, respectively. The four startup logs in Figure 4 verify the guest UART transmit paths and drivers.

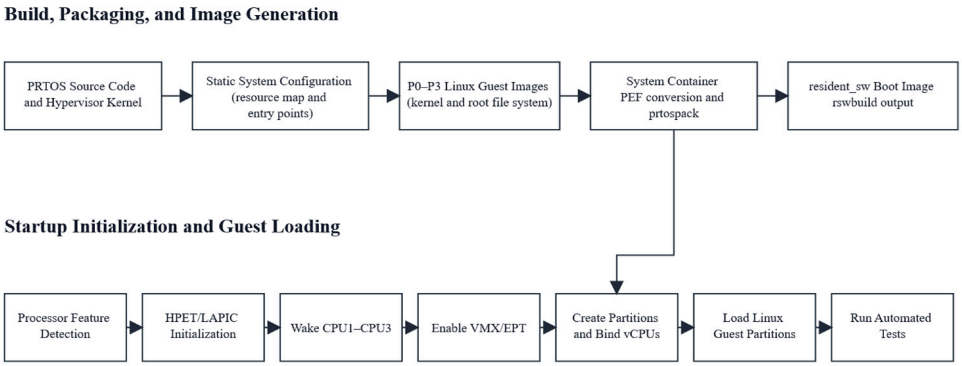


Figure 3. Build, packaging, and startup process of the PRTOS system.

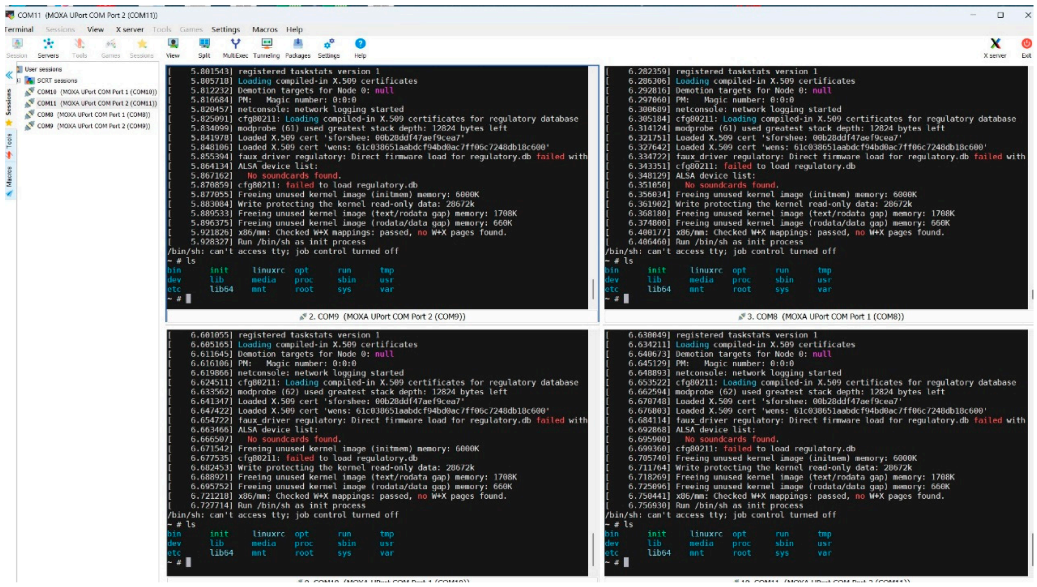


Figure 4. Serial-console outputs from the four Linux guest partitions during normal startup.

Each test program starts automatically and outputs only aggregate results, minimizing serial-I/O interference during measurement. Table 4 summarizes the implemented and verified porting functions.

Table 4. Key PRTOS porting steps and implementation results.

Verification Item	Implementation	Verification Result
System configuration	Define P0-P3, vCPUs, memory, and device resources	System PEF generated successfully
Image packaging	Package the PRTOS core, system configuration, and guest images	container.bin and resident_sw generated
Multicore startup	Wake CPU1-CPU3 and complete multicore initialization	All four physical cores started normally
Hardware virtualization	Enable VMX and establish EPT address spaces	Four guest address spaces established normally
Partition creation	Create P0-P3 and bind CPU0-CPU3	One-core/one-partition mapping achieved

Table 4. *Cont.*

Verification Item	Implementation	Verification Result
Guest systems	Load the Linux kernels and root file systems	All four guests entered user space
UART mapping	Map ttyS0-ttyS3 to P0-P3, respectively	All four channels output logs independently
Automated tasks	Run periodic tasks from guest startup scripts	Four partitions execute in parallel
Inter-partition communication	Status reporting and command-response exchange	Basic controlled communication verified

3.3. Controlled Inter-Partition Communication

Partition isolation does not preclude controlled interaction. P1–P3 report status to P0 through preconfigured channels with explicit direction, capacity, and permissions, while private memory remains inaccessible. This arrangement tests whether P2 computational or memory pressure degrades its own reporting or the independent P1 and P3 channels.

4. Fault-Containment Experimental Method

4.1. Evaluation Metrics and Statistical Definitions

To evaluate the ability of the multi-partition environment to host tasks with different periods and workloads, four periodic tasks are configured in P0–P3. The periodic task model follows the standard definitions used in classical real-time scheduling analysis [27]. Partition task τ_i is described by its period T_i, simulated workload C_i per period, and relative deadline D_i, where subscript i denotes the task index and $D_i = T_i$ in this study. For the kth sample of task τ_i, $r_{i,k}$ and $f_{i,k}$ denote the absolute scheduled release time and actual completion time, respectively. The response time $R_{i,k}$ is defined by Equation (6):

$$R_{i,k} = f_{i,k} - r_{i,k} \tag{6}$$

A deadline miss is recorded when $R_{i,k} > D_i$. Let N_i be the number of samples obtained for task τ_i and $m_{i,k}$ be the miss indicator for its kth sample, where $m_{i,k} = 1$ for a deadline miss and $m_{i,k} = 0$ otherwise. The number of deadline misses M_i and the deadline miss rate η_i are calculated using Equations (7) and (8), respectively:

$$M_i = \sum_{k=1}^{N_i} m_{i,k} \tag{7}$$

$$\eta_i = \frac{M_i}{N_i} \times 100\% \tag{8}$$

In the fault-injection experiments, the original workload of P2/T3 is denoted by C_{T3}, the per-period execution budget of the additional fault thread by C_f, and their common period by T. The nominal per-period processor demand of P2, U_{P2}, is calculated using Equation (9):

$$U_{P2} = \frac{C_{T3} + C_f}{T} \tag{9}$$

Here, $C_{T3} = 3000$ µs, C_f is set to 6000, 8000, or 10,000 µs, and $T = 10{,}000$ µs; therefore, U_{P2} is 0.9, 1.1, or 1.3, respectively. This metric represents the nominal processor-time demand of P2 within one period. For the 16 MiB read/write condition, it does not independently characterize the cache-miss rate or memory bandwidth.

For percentile calculation, the N_i response-time samples of task τ_i are sorted in nondecreasing order as $R_i,(1) \leq \ldots \leq R_i,(N_i)$. The mean, P99, and maximum response times are calculated using Equations (10)–(12), respectively:

$$\overline{R_i} = \frac{1}{N_i} \sum_{k=1}^{N_i} R_{i,k} \tag{10}$$

$$P_{99}(R_i) = R_{i,(\lceil 0.99 N_i \rceil)} \tag{11}$$

$$R_{i,\max} = \max_{1 \leq k \leq N_i} R_{i,k} \tag{12}$$

For inter-partition communication channel c, $N_{\text{plan},c}$, $N_{\text{send},c}$, and $N_{\text{recv},c}$ denote the planned number of messages, the number successfully sent, and the number actually received by P0, respectively. The send completion rate $S_{\text{send},c}$ and end-to-end reception completeness rate $S_{\text{recv},c}$ are calculated using Equations (13) and (14):

$$S_{\text{send},c} = \frac{N_{\text{send},c}}{N_{\text{plan},c}} \times 100\% \tag{13}$$

$$S_{\text{recv},c} = \frac{N_{\text{recv},c}}{N_{\text{plan},c}} \times 100\% \tag{14}$$

For the jth successfully received message on channel c, $t_{\text{send},j}$ and $t_{\text{recv},j}$ denote its send timestamp and P0 reception timestamp, respectively. Its end-to-end latency L_j is calculated using Equation (15):

$$L_j = t_{\text{recv},j} - t_{\text{send},j} \tag{15}$$

The mean, P99, and maximum communication latencies are calculated using the same statistical definitions as Equations (10)–(12).

In addition, G_c records the number of missing messages indicated by discontinuities in received sequence numbers, and E_c records the number of received messages that fail integrity verification. At the end of an experiment, the number of successfully sent messages that have not yet been received by P0, $Q_{\text{end},c}$, is defined by Equation (16):

$$Q_{\text{end},c} = \max\left(N_{\text{send},c} - N_{\text{recv},c}, 0\right) \tag{16}$$

If $Q_{\text{end},c}$ remains nonzero after the prescribed drain interval, a queue-drain timeout is recorded. Communication latency is calculated only for successfully received messages and must therefore be interpreted together with send completion rate, reception completeness, sequence gaps, integrity errors, and queue drain status. The principal criteria in this study are bounded response relative to each task's own period, deadline satisfaction, and whether a fault crosses the predefined partition boundary; absolute response times of tasks with different periods are not used for cross-task performance ranking. Table 5 summarizes the evaluation metrics and statistical definitions.

Table 5. Evaluation metrics and statistical definitions.

Metric	Calculation Basis	Interpretation
Response time	Equations (6) and (10)–(12)	Scheduling and execution delay from release to completion
Deadline miss	Equations (7) and (8)	Count and proportion of real-time failures

Table 5. *Cont.*

Metric	Calculation Basis	Interpretation
P2 nominal demand	Equation (9)	P2 processor-time budget per period
Send/reception completeness	Equations (13) and (14)	Whether messages are sent successfully and reach P0
Communication latency	Equations (15) and (10)–(12)	Timeliness of successfully delivered messages
Residual queue occupancy	Equation (16) and the drain-timeout flag	Tail backlog at experiment termination
Sequence and integrity	Sequence gap count and integrity error count	Message continuity and data correctness

4.2. Experimental Design and Control Variables

A progressively extended design is adopted to distinguish the effects of processor-budget exceedance, load type, and communication activity on isolation. Across all PRTOS conditions, the four-partition processor binding, authorized memory regions, UART mapping, task periods, simulated workloads, scheduling priorities, and 500 s runtime remain unchanged. Only the form and per-period budget of the P2 fault load and the activation of inter-partition communication are varied. P0–P3 are bound to CPU0–CPU3, respectively; consequently, a fault thread created inside P2 can execute only on CPU2. Individual samples are not printed during execution; aggregate statistics are reported after completion to reduce interference from UART output and formatting operations.

Table 6 lists the period, sample count, simulated workload, scheduling priority, and task role for each partition. P1/T1 is a 1 ms high-frequency critical task, whereas P2/T3 is a 10 ms computation-intensive task. During 500 s, P1/T1, P0/T2, P2/T3, and P3/T4 produce 500,000, 50,000, 50,000, and 25,000 samples, respectively.

Table 6. Configuration of periodic tasks in the four partitions.

Partition/Task	Period/μs	Samples	Simulated Workload/μs	Priority	Task Role
P1/T1	1000	500,000	80	80	High-frequency critical control task
P0/T2	10,000	50,000	120	60	System maintenance and status-processing task
P2/T3	10,000	50,000	3000	50	Computation-intensive safety/airship management task
P3/T4	20,000	25,000	160	40	Low-frequency payload or monitoring task

4.3. Design of Comparison Conditions

With the above control variables unchanged, four progressively extended reference conditions, A–D, are defined as listed in Table 7.

Condition A is the fault-free baseline and provides reference execution time, response time, and deadline-miss results for all four tasks.

Condition B applies a pure computational load by adding a SCHED_FIFO/90 fault thread inside P2. Its period T is fixed at 10,000 μs, and its busy-wait budget C_f is set to 6000, 8000, or 10,000 μs. When combined with the original P2/T3 workload C_{T3} = 3000 μs,

Equation (9) gives nominal P2 processor demands of 0.9, 1.1, and 1.3, representing high load within budget, mild overload, and severe overload, respectively.

Condition C retains the partition location, thread priority, period, three time budgets, and 500 s duration of condition B, but it replaces busy-wait computation with cyclic reads and writes to a 16 MiB array, thereby providing a load-type comparison at the same nominal processor demands.

Condition D enables inter-partition status communication on top of condition C. It includes a communication baseline without injection and memory-pressure levels U_{P2} = 0.9, 1.1, and 1.3 to analyze task timing, message completeness, communication latency, and buffer status.

Table 7. Four progressively extended experimental conditions.

Condition	Additional P2 Load	U_{P2}	Inter-Partition Communication	Principal Objective
A	None	0.3	Disabled	Establish a fault-free periodic-task baseline
B	6000/8000/10,000 µs busy-wait	0.9/1.1/1.3	Disabled	Processor-budget sensitivity and core-level containment
C	16 MiB cyclic read/write; same budgets as B	0.9/1.1/1.3	Disabled	Cache/memory pressure and load-type comparison
D	Condition-C load; also includes a no-injection baseline	0.3/0.9/1.1/1.3	P1/P2/P3→P0	Message completeness, latency, and buffer state

Condition E is the native-Linux cross-core fault-propagation comparison. During execution, the functional process containing P2/T3 creates four fault threads. All use the POSIX SCHED_FIFO real-time scheduling policy at priority 90 and are bound to CPU0–CPU3, respectively [28]. Each thread runs at 100 Hz with a 10,000 µs period, performs 9950 µs of busy-wait computation per period, and remains active for 60 s. The condition actively triggers a processor-resource anomaly through a controlled computational load and therefore constitutes runtime software fault injection [29]. Because all four threads are created by the same P2/T3 functional process, the condition represents the expansion of one functional anomaly from its original core to multiple processor cores in a shared-kernel native-Linux environment, rather than four independent external fault sources. Figure 5 presents three parallel paths—the fault-free PRTOS baseline, P2-local overload under PRTOS, and multicore propagation of one faulty function under native Linux—together with their fault-impact scopes.

All experiments use the same hardware platform. The comparison examines the propagation scope of a single functional anomaly under preconfigured static-partition boundaries and runtime shared-scheduling boundaries. It focuses on the set of tasks experiencing deadline misses and whether the fault extends to other processor cores, rather than directly ranking the mean execution time or overall performance of the two runtime environments.

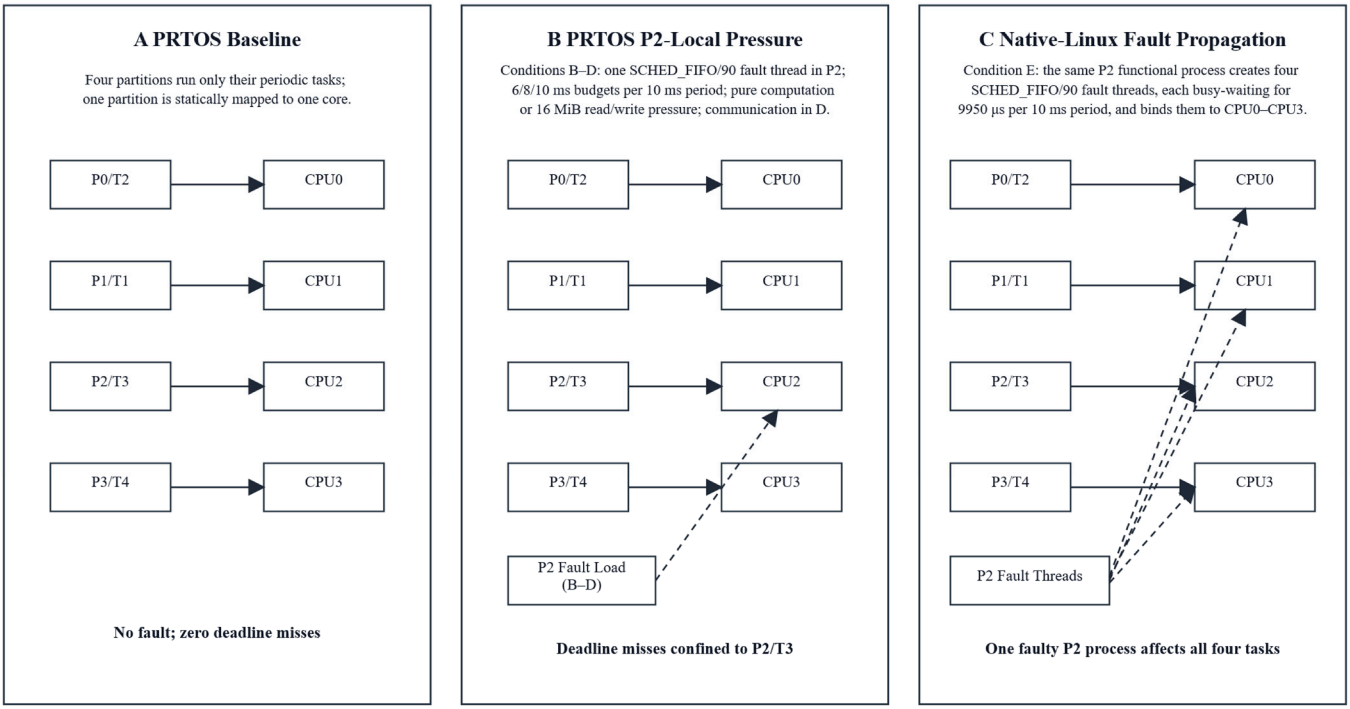


Figure 5. Three representative experimental paths and fault-impact scopes.

4.4. Implementation of 16 MiB Cache/Memory Pressure

Condition C uses a fixed 16 MiB working set and a 64 B access stride. The platform has a 4 MiB last-level cache; thus, the working set is four times the cache capacity. The fault thread continuously cycles through reads and writes to the array, repeatedly inducing cache-line replacement, cache misses, and memory accesses and thereby increasing contention for the shared last-level cache and memory controller. The 64 B stride matches a typical cache-line size and avoids repeatedly accessing only one cache line. Before timing begins, the array is prefaulted at 4 KiB page intervals, touching 4096 pages so that physical pages are committed before formal measurement and first-page-fault effects are reduced. The 16 MiB allocation remains below the free runtime memory of the P2 Linux guest, preventing memory exhaustion or swapping from becoming an additional fault source.

The three pressure levels are created by varying the time budget during which the fault thread cyclically reads and writes the 16 MiB array in each 10,000 µs period. With the working-set size, stride, and access pattern unchanged, budgets of 6000, 8000, and 10,000 µs produce progressively longer sustained-access intervals. Combined with the original 3000 µs workload of P2/T3, they yield nominal P2 processor demands of 0.9, 1.1, and 1.3. A larger budget generally permits more array accesses per period and occupies the CPU, last-level cache, and memory path for longer, thereby producing high in-budget, mild-overload, and severe-overload levels of composite pressure. Condition C therefore combines processor occupation, cache replacement, and memory-access latency and should not be interpreted as a pure busy-wait load.

4.5. Inter-Partition Communication Load and Buffer Configuration

Condition D uses statically preconfigured unidirectional shared-memory message queues. After completing their periodic computation, P1/T1, P2/T3, and P3/T4 each send one 64 B status message to P0 within their own task scheduling context. Sending is not performed by independent communication threads; it inherits the SCHED_FIFO priorities of T1, T3, and T4, namely 80, 50, and 40. Each message contains a source-partition identifier, monotonically increasing sequence number, send timestamp, and integrity-check

information. P0 runs a separate SCHED_FIFO receiver task at priority 45 with a 1000 µs period. It polls and drains the P1-to-P0, P2-to-P0, and P3-to-P0 queues while checking sequence continuity, verifying message data, and recording reception timestamps.

The three unidirectional channels are statically configured before startup. Each is assigned an 8192 B shared-memory region, uses 64 B messages, and has an effective queue depth of 64. Each sending partition can access only its designated transmit channel, whereas P0 has receive access; no guest partition can use a channel region to access another partition's private memory. Given T1, T3, and T4 periods of 1000, 10,000, and 20,000 µs, the planned message counts during 500 s are 500,000, 50,000, and 25,000, respectively.

Communication is evaluated in terms of send completion, end-to-end delivery completeness, data correctness, and final queue state. Planned sends, completed sends, and incomplete sends identify failures caused by computation overload or unavailable queue space. The actual P0 receive count and reception completeness rate are then recorded, and sequence gaps are derived from message sequence numbers. Integrity-check failures quantify data errors. After periodic tasks finish, the receiver continues for a bounded drain interval, and the final queue state and remaining unreceived messages are recorded. For successfully received messages, end-to-end latency is the difference between the P0 reception and send timestamps; its mean, P99, and maximum are reported. Because latency describes only successfully delivered messages, it must be evaluated together with send completion rate, reception completeness, sequence gaps, integrity errors, and queue-drain status; latency percentiles alone cannot establish whether overload has impaired communication service.

5. Experimental Results and Analysis

5.1. Condition A: Fault-Free Baseline

The results for condition A are presented in Table 8. The P99 and maximum response times of all four tasks are below their respective periods, and no sample incurs a deadline miss. The maximum response time of P1/T1 is 152 µs, or 15.2% of its 1 ms period. P2/T3 includes 3000 µs of simulated workload; its P99 and maximum response times are 4050 and 4097 µs, corresponding to 40.5% and 41.0% of its 10 ms period. These results demonstrate adequate margin without additional load and establish the baseline for the three subsequent PRTOS conditions.

Table 8. Fault-free baseline results under condition A.

Task	Samples	P99 Response/µs	Maximum Response/µs	Deadline Miss	Miss Rate/%
P1/T1	500,000	118	152	0	0
P0/T2	50,000	1177	1202	0	0
P2/T3	50,000	4050	4097	0	0
P3/T4	25,000	1210	1243	0	0

5.2. Condition B: Sensitivity to Pure Computational Load

Table 9a presents the timing degradation of P2/T3, and Table 9b groups the statistics of the non-fault partitions P0/T2, P1/T1, and P3/T4 under the three pressure levels. As U_{P2} increases, the response time and deadline-miss behavior of P2/T3 deteriorate markedly. In contrast, the mean, P99, and maximum execution and response times of the three non-fault tasks remain bounded, with zero deadline misses at every pressure level. At $U_{P2} = 0.9$, the mean P2/T3 response time is 9413.578 µs and 3001 deadline misses occur (6.002%). At

U_{P2} = 1.1, the count rises to 49,900 (99.800%); at U_{P2} = 1.3, all 50,000 samples miss their deadlines. The degradation is therefore concentrated in the injected partition.

Table 9. (**a**) Timing statistics of P2/T3 under condition B (unit: µs). (**b**) Timing statistics of tasks in non-fault partitions under condition B (unit: µs).

(a)

Condition	Mean Execution	Execution P99	Maximum Execution	Mean Response	Response P99	Maximum Response	Deadline Misses
0.9	3245.959	7045	7058	9413.578	10,051	10,084	3001
1.1	9692.512	10,112	14,167	18,572.375	19,110	22,932	49,900
1.3	6682.178	10,550.010	10,793	158,778,168.851	266,683,282.490	268,778,224	50,000

(b)

Partition/task	Condition	Mean execution	Execution P99	Maximum execution	Mean response	Response P99	Maximum response	Deadline misses
P0/T2	0.9	120.050	121	121	665.501	1177	1204	0
	1.1	120.050	121	121	665.855	1177	1219	0
	1.3	120.049	121	121	667.361	1179	1236	0
P1/T1	0.9	80.920	113	147	81.834	118	147	0
	1.1	80.951	113	151	81.900	119	183	0
	1.3	80.965	114	163	81.915	120	170	0
P3/T4	0.9	160.051	161	161	703.473	1209	1241	0
	1.1	160.052	161	161	702.775	1212	1256	0
	1.3	160.053	161	161	705.695	1212	1259	0

Although P2/T3 progresses from local deadline misses to complete loss of tracking of the periodic time base, P0, P1, and P3 retain zero deadline misses under all three levels. Their maximum response times remain within 1204–1236 µs for P0/T2, 147–183 µs for P1/T1, and 1241–1259 µs for P3/T4, which are of the same order as the baseline. Thus, U_{P2} controls the degree of degradation inside the faulty partition, but the fault thread obtains no execution opportunity on CPU0, CPU1, or CPU3; the core-level deadline failure domain remains confined to P2.

5.3. Condition C: 16 MiB Read/Write Pressure and Load-Type Comparison

Table 10a presents the P2/T3 results for condition C, and Table 10b groups the non-fault partition results. At U_{P2} = 0.9, the mean P2/T3 response time is 7785.787 µs and only two deadline misses occur. At U_{P2} = 1.1, 49,903 samples miss their deadlines (99.806%), and at U_{P2} = 1.3 all 50,000 samples miss. Meanwhile, the execution- and response-time statistics of P0/T2, P1/T1, and P3/T4 remain bounded at all three pressure levels, with zero deadline misses. The threshold trend is consistent with condition B, but the load changes from pure busy-wait computation to composite pressure involving processor occupation, cache replacement, and memory-access latency.

Under all three 16 MiB pressure levels, P0/T2, P1/T1, and P3/T4 retain zero deadline misses; their maximum response-time ranges are 1224–1246, 183–206, and 1268–1271 µs, respectively. Relative to condition B, condition C changes the response distribution inside P2 at the same nominal budget but does not enlarge the set of partitions with deadline misses. This supports the conclusion that static core binding prevents a fault thread from directly expanding to other cores. Nevertheless, small variations remain in the non-fault tail statistics, indicating that one-core/one-partition mapping does not eliminate shared hardware paths such as the last-level cache, memory controller, and interconnect. Because hardware performance counters were not collected, these variations cannot be attributed uniquely to cache misses or memory bandwidth contention.

Table 10. (**a**) Timing statistics of P2/T3 under condition C (unit: μs). (**b**) Timing statistics of tasks in non-fault partitions under condition C (unit: μs).

(a)

Condition	Mean Execution	Execution P99	Maximum Execution	Mean Response	Response P99	Maximum Response	Deadline Misses
0.9	3000.057	3001	3001	7785.787	8306	21,639	2
1.1	9697.815	10,142	23,922	16,790.102	17,330	31,133	49,903
1.3	6691.285	10,499	24,389	159,572,054.988	267,741,669.830	269,844,167	50,000

(b)

Partition/task	Condition	Mean execution	Execution P99	Maximum execution	Mean response	Response P99	Maximum response	Deadline misses
P0/T2	0.9	120.045	121	121	678.217	1198	1246	0
	1.1	120.057	121	121	678.438	1187	1224	0
	1.3	120.048	121	121	679.536	1191	1232	0
P1/T1	0.9	81.559	126	186	83.199	134	206	0
	1.1	81.671	129	194	83.393	136	194	0
	1.3	81.896	131	182	83.927	138	183	0
P3/T4	0.9	160.046	161	161	712.012	1218	1271	0
	1.1	160.047	161	161	711.245	1217	1271	0
	1.3	160.051	161	161	721.292	1231	1268	0

5.4. Condition D: Inter-Partition Communication Completeness, Latency, and Buffer State

Table 11a presents the P2/T3 timing results for condition D, and Table 11b groups the P0/T2, P1/T1, and P3/T4 results. In the communication baseline and at U_{P2} = 0.9, 1.1, and 1.3, the execution and response times of the three non-fault tasks remain bounded, and none incurs a deadline miss. P2/T3 progresses from local misses to complete deadline failure as pressure increases. Enabling communication therefore does not change deadline satisfaction in the non-fault partitions, and the principal fault effect remains confined to P2. Within the present configuration and 500 s observation window, no cross-partition timing effect causing a deadline miss in a non-fault partition is observed.

Table 11c summarizes sending, reception, completeness, latency, and final queue state for P2-to-P0, whereas Table 11d groups the communication results of the non-fault P1-to-P0 and P3-to-P0 channels. In the communication baseline and at U_{P2} = 0.9 and 1.1, P2 completes 50,000 sends and P0 receives all 50,000 messages, with zero integrity errors. At U_{P2} = 1.3, P2 completes only 24,356 sends, of which P0 receives 24,293. Completeness relative to the planned count falls to 48.586%; 25,644 sends are not completed, 63 successfully sent messages remain unreceived at termination, and the queue-drain interval times out. In contrast, the P1-to-P0 and P3-to-P0 channels exhibit neither residual queue occupancy nor integrity errors.

For successfully delivered P2 messages, the mean latency across the four communication conditions is approximately 535–553 μs and P99 is approximately 1028–1141 μs, without a monotonic increase with U_{P2}. At U_{P2} = 1.3, however, these latency statistics cover only the 24,293 successfully received messages and do not represent the quality of service of the remaining incomplete messages. Delivery completeness must therefore be considered before delivered-message latency when evaluating communication fault tolerance; similar latency percentiles do not imply that communication is unaffected.

As shown in Table 11d, the P1-to-P0 and P3-to-P0 communication statistics do not exhibit material degradation as P2 pressure increases. P1-to-P0 reception completeness remains 99.994% in all four conditions, with 29–31 sequence gaps and no integrity error or residual queue occupancy. P3-to-P0 reception completeness remains 100%, with no sequence gap, integrity error, or residual queue occupancy.

P1 plans 500,000 messages per run and P0 receives 499,969–499,971, giving 99.994% completeness, 29–31 sequence gaps, and zero integrity errors; mean latency is

536.047–540.751 µs and P99 is 1073–1088 µs. P3 plans and delivers 25,000 messages in each run, giving 100% completeness with zero sequence gaps and integrity errors; mean latency is 528.592–537.384 µs and P99 is 1042–1052 µs. At U_{P2} = 1.3, communication degradation is concentrated in P2 and does not develop into increased loss, corruption, or tail backlog on the P1 or P3 channels. Engineering deployment should nevertheless include message-freshness thresholds, timeout alarms, buffer-watermark monitoring, and handling of failed sends, because core-level isolation alone does not guarantee the external service continuity of a faulty partition.

Table 11. (**a**) Timing statistics of P2/T3 under condition D (unit: µs). (**b**) Timing statistics of tasks in non-fault partitions under condition D (unit: µs). (**c**) P2-to-P0 communication statistics under condition D. (**d**) Communication statistics of non-fault partitions under condition D.

(a)

Condition	Mean Execution	Execution P99	Maximum Execution	Mean Response	Response P99	Maximum Response	Deadline Misses
Communication baseline	3000.045	3001	3001	3545.719	4054	4104	0
0.9	3000.045	3001	3001	7987.428	8510	21,232	2
1.1	9698.101	10,132	22,483	16,904.696	17,442	30,210	49,902
1.3	6682.348	10,449	23,931	158,991,882.261	266,974,290.670	269,078,895	50,000

(b)

Partition/task	Condition	Mean execution	Execution P99	Maximum execution	Mean response	Response P99	Maximum response	Deadline misses
P0/T2	Communication baseline	120.043	121	121	670.143	1185	1251	0
	0.9	120.042	121	121	670.256	1186	1264	0
	1.1	120.043	121	121	687.220	1208	1243	0
	1.3	120.042	121	121	684.934	1206	1261	0
P1/T1	Communication baseline	81.042	115	209	82.077	122	209	0
	0.9	81.635	128	194	83.418	135	210	0
	1.1	81.579	126	176	83.395	134	181	0
	1.3	81.752	128	210	83.807	135	211	0
P3/T4	Communication baseline	160.046	161	161	704.393	1214	1261	0
	0.9	160.041	161	161	722.104	1232	1280	0
	1.1	160.046	161	161	721.646	1230	1263	0
	1.3	160.040	161	161	720.555	1231.010	1270	0

(c)

Condition	Planned/sent/received	Completeness/%	G_c/E_c	Mean/P99/maximum latency/µs	Qend/drain
Baseline	50,000/50,000/50,000	100.000	0/0	547.148/1141/83,085	0/normal
0.9	50,000/50,000/50,000	100.000	0/0	552.798/1047/84,054	0/normal
1.1	50,000/50,000/50,000	100.000	0/0	534.696/1028/89,045	0/normal
1.3	50,000/24,356/24,293	48.586	0/0	551.162/1094/79,402	63/timeout

(d)

Communication channel	Condition	Planned/sent/received	Completeness/%	G_c/E_c	Mean/P99/maximum latency/µs	Qend/drain
P1→P0	Baseline	500,000/499,970/499,970	99.994	30/0	536.047/1073/93,061	0/normal
	0.9	500,000/499,971/499,971	99.994	29/0	537.954/1073/91,736	0/normal
	1.1	500,000/499,971/49,9971	99.994	29/0	537.774/1086/91,852	0/normal
	1.3	500,000/499,969/499,969	99.994	31/0	540.751/1088/93,866	0/normal
P3→P0	Baseline	25,000/25,000/25,000	100.000	0/0	528.592/1042/83,920	0/normal
	0.9	25,000/25,000/25,000	100.000	0/0	529.179/1049/75,162	0/normal
	1.1	25,000/25,000/25,000	100.000	0/0	534.897/1043/87,140	0/normal
	1.3	25,000/25,000/25,000	100.000	0/0	537.384/1052/92,420	0/normal

5.5. Condition E: Native-Linux Multicore-Propagation Comparison

In the native-Linux comparison, a fault thread strictly confined to CPU2 mainly affects P2/T3. Under real-time scheduling privileges and uncontrolled application behavior,

however, the same P2 functional process can create four SCHED_FIFO/90 threads and bind them to CPU0–CPU3. Each thread runs at 100 Hz, busy-waits for 9950 µs per period, and remains active for 60 s. The results are shown in Table 12. All four measured tasks degrade substantially: all 60,000 P1/T1 samples miss their deadlines, P0/T2 and P2/T3 each incur 6000 misses, and P3/T4 incurs 2914 misses. P99 values on the order of tens of seconds represent accumulated lag relative to absolute scheduled release times.

Table 12. Multicore fault-propagation results under native Linux.

Task	Baseline P99/µs	Fault P99/s	Baseline Misses	Fault Misses	Impact Assessment
P0/T2	150.571	56.538	0	6000	Marked degradation
P1/T1	105.757	58.270	0	60,000	All samples miss
P2/T3	3029.311	43.867	0	6000	Marked degradation
P3/T4	189.543	53.223	0	2914	Marked degradation

5.6. Overall Comparison

Conditions B–D and the native-Linux condition E compare two forms of runtime boundary. Figure 5 shows the fault-load paths. Tables 9–11 show that P2/T3 degrades severely as processor demand increases under PRTOS, whereas P0/T2, P1/T1, and P3/T4 incur no deadline misses in any 500 s pressure condition. Table 12 shows that, after high-priority threads created by the same functional process extend to CPU0–CPU3 under native Linux, all four tasks incur deadline misses. The observed core-level propagation scope is therefore determined not by the name of the injected load but by the runtime domain accessible to its threads.

Before startup, PRTOS statically maps the P2 guest partition, its vCPU, CPU2, its memory region, and its authorized devices. Whether a newly created SCHED_FIFO fault thread in P2 performs pure busy-wait computation or cyclic 16 MiB reads and writes, its schedulable range remains confined to the P2 vCPU and CPU2. It cannot execute on CPU0, CPU1, or CPU3 or directly access another partition's private memory or unauthorized devices. Native Linux is not devoid of isolation mechanisms, but its boundaries are maintained at runtime by scheduling policy, processor affinity, privileges, and application behavior within a shared kernel [21]. A static-partitioning hypervisor instead predefines processor, memory, and device boundaries at system startup or partition activation [22,23]. Table 13 summarizes these differences.

For a quantitative comparison of timing degradation and propagation scope, Table 14 summarizes the deadline-miss rates of the four tasks and the number of affected tasks. Rates are calculated using Equation (8): PRTOS conditions A–D use the samples collected during each 500 s run, whereas native-Linux condition E uses the samples collected during the 60 s injection interval. The number of affected tasks is defined as the number with a nonzero deadline-miss rate. Tasks with different periods are compared only by the miss proportion within their own observation samples, not by absolute response-time ranking.

As shown in Table 14, all four tasks have zero deadline-miss rates in fault-free condition A and the condition-D communication baseline. At $U_{P2} = 0.9$ in the three PRTOS pressure conditions, the P2/T3 miss rates are 6.002%, 0.004%, and 0.004%, demonstrating different within-partition degradation intensities for different load types at the same nominal demand. At $U_{P2} = 1.1$, the corresponding rates rise to 99.800%, 99.806%, and 99.804%, and all reach 100% at $U_{P2} = 1.3$. Meanwhile, P0/T2, P1/T1, and P3/T4 retain zero deadline-miss rates in every PRTOS pressure condition, so the number of affected tasks remains one. Processor-budget exceedance therefore first manifests as timing failure within P2 rather than deadline failure in other partitions.

Table 13. Comparison of runtime boundaries between PRTOS and native Linux.

Comparison Dimension	PRTOS Partitioned Environment	Native-Linux Environment
Processor resources	The hypervisor statically configures vCPUs and binds physical cores	Determined jointly by the scheduler, affinity, privileges, and application behavior
Memory resources	Guest partitions have independent EPT mappings; private memory cannot be directly accessed by other guests	User processes have separate virtual address spaces but share the kernel and physical-memory management subsystem
Device access	Authorized per partition by system configuration; unauthorized devices are invisible to guests	Managed by the shared kernel, device drivers, and access permissions
Fault-thread expansion	Confined to the partition vCPU and authorized physical core	May extend to other cores when privileges and affinity permit
Fault recovery	Can support partition-level stop, reset, or restart while other partitions continue	Typically uses thread-, process-, or system-level recovery
Principal characteristic	Boundaries are predetermined during design/configuration, giving an explicit upper bound on the fault domain	Flexible and general-purpose resource scheduling; boundaries rely more on runtime mechanisms

Table 14. Deadline-miss rates and fault-propagation scope under all conditions.

Environment/Condition	U_{P2} Nominal Demand	P0/T2 Miss Rate/%	P1/T1 Miss Rate/%	P2/T3 Miss Rate/%	P3/T4Miss Rate/%	Affected Tasks
PRTOS-A baseline	0.3	0	0	0	0	0
PRTOS-B	0.9	0	0	6.002	0	1
PRTOS-B	1.1	0	0	99.800	0	1
PRTOS-B	1.3	0	0	100	0	1
PRTOS-C	0.9	0	0	0.004	0	1
PRTOS-C	1.1	0	0	99.806	0	1
PRTOS-C	1.3	0	0	100	0	1
PRTOS-D communication baseline	0.3	0	0	0	0	0
PRTOS-D	0.9	0	0	0.004	0	1
PRTOS-D	1.1	0	0	99.804	0	1
PRTOS-D	1.3	0	0	100	0	1
Native Linux-E	—	100	100	100	97.133	4

In native-Linux condition E, P0/T2, P1/T1, and P2/T3 each have a 100% deadline-miss rate and P3/T4 has a 97.133% rate; the number of affected tasks increases to four, as shown in Table 12. Combined with the runtime boundaries in Table 13, Table 14 statistically supports the following scoped conclusion: under the static configuration, fault model, and observation intervals used here, PRTOS confines the direct scheduling effect of the P2 fault threads to CPU2, and no propagation into deadline misses in P0, P1, or P3 is observed. In condition D at U_{P2} = 1.3, P2-to-P0 message completeness falls to 48.586%, whereas the P1-to-P0 and P3-to-P0 channels show no integrity error or residual queue occupancy, as detailed in Table 11c,d. Thus, the fault first degrades P2 timing and external service continuity. This conclusion concerns cross-core thread expansion and processor-budget exceedance and does not exclude cross-partition interference through shared last-level cache, memory controller, interconnect, I/O, or virtual-interrupt paths [10,19].

6. Discussion

6.1. Fault-Containment Value of Static Partitioning

The fault-containment value of static partitioning is not to guarantee that an overloaded partition continues to meet its original deadlines, but to bound in advance the resources for which it can compete and the resulting fault-impact domain. Table 14 shows zero deadline-miss rates for all four tasks in condition A and the condition-D communication baseline. Under all three pressure levels in conditions B–D, P2/T3 progresses from local misses to nearly complete or complete deadline failure as U_{P2} increases, while P0/T2, P1/T1, and P3/T4 retain zero misses and the number of affected tasks remains one. In native-Linux condition E, all four tasks miss deadlines. Together with Table 13, these results indicate that propagation scope depends primarily on the processor and resource domain accessible to an anomalous thread rather than on whether its load is pure computation or cyclic memory access. For a long-endurance unattended near-space airship, such an analyzable upper bound on fault impact is more useful for graded deployment of critical functions and mission payloads than a reduction in mean execution time alone.

Table 11c,d further show that task-timing isolation does not imply service continuity of the faulty partition. At U_{P2} = 1.3, P2-to-P0 message completeness falls to 48.586%; 63 successfully sent messages remain unreceived at termination and queue draining times out. Meanwhile, the P1-to-P0 and P3-to-P0 channels show no integrity error or residual backlog. PRTOS prevents the P2 fault thread from directly consuming other cores, but it does not replace fault detection, state-freshness assessment, or recovery control. An engineering implementation should therefore configure failed-send counters, message timeouts, queue-watermark alarms, and partition stop/restart policies in P0 or an independent health management partition so that a deterministic fault boundary becomes detectable, localizable, and recoverable system behavior.

6.2. Relationship to ARINC 653 and Multicore Real-Time Behavior

The current prototype uses one core per partition and primarily evaluates processor isolation at the physical-core level, thereby de-emphasizing major-time-frame and partition-window scheduling when several partitions share one core. The results cannot therefore be interpreted as complete ARINC 653 conformance verification. ARINC 653 also covers the APEX interface, partition-mode management, process services, sampling and queuing ports, health monitoring, and system-level configuration verification [4,5]. For airworthiness or other high-assurance applications, the static resource mapping demonstrated here must be combined with partition schedules, port semantics, health monitoring, and configuration-consistency evidence to form a complete verification chain from architectural boundaries to interface behavior.

Multicore temporal isolation likewise cannot be guaranteed by core binding alone. FAA AC 20-193 requires avionics applications using multicore processors to identify, evaluate, and mitigate potential interference channels [30]. Research on cache partitioning, safe multicore use in avionics, and memory-access regulation shows that sharing of the last-level cache, memory controller, interconnect, and I/O can alter worst-case latency and predictability [10,19,31]. The non-fault partitions exhibit limited variation in their tail statistics, but no hardware performance counters were collected; the variation cannot therefore be attributed uniquely to cache misses or memory bandwidth contention. The appropriate interpretation is that PRTOS establishes clear processor and address-space boundaries, while shared microarchitectural resources still require separate budgeting, monitoring, and mitigation.

PRTOS is not necessarily superior to native Linux for every response-time or jitter metric. Hypervisor entry and exit, EPT translation, guest clocks, and virtual interrupts

may introduce overhead. Surveys of safety-critical hypervisors and real-time studies of Xen/KVM show that virtualization overhead depends strongly on platform configuration, timers, and interrupt paths [18,21]. This study therefore uses the scope of deadline misses and whether a fault crosses a predefined boundary as its principal criteria, rather than directly ranking mean latency between runtime environments.

6.3. Limitations and Threats to Validity

Construct validity is first limited by the task and load models. The four periodic tasks use fixed busy-wait intervals to emulate computation, and the 16 MiB cyclic access pattern is used only to create composite pressure from processor occupation, cache replacement, and memory-access latency. These loads are not equivalent to real flight control loops, airship state processing, or mission payload algorithms. The experiments therefore demonstrate observable processor and address-space boundary effects under the specified load models, rather than worst-case execution time or end-to-end functional correctness of real avionics functions.

Internal validity and statistical inference are also constrained. Data are obtained from one Intel Atom x6425RE platform. Each PRTOS level uses one 500 s observation, and native-Linux condition E uses a 60 s injection interval; independent repetitions, confidence intervals, and effect sizes are not reported. The rates in Table 14 are therefore suitable for comparing deadline satisfaction and the number of affected tasks within each task's observation window, but not for ranking absolute response performance across tasks with different periods or across runtime environments. Hardware counters for cache misses, memory bandwidth, interconnect use, and virtual interrupts were not collected, so the limited tail-statistic variation in non-fault partitions cannot be assigned to a specific shared microarchitectural channel.

External validity is limited by the platform architecture and fault scope. The current one-core/one-partition prototype does not cover major-time-frame or partition-window scheduling when multiple partitions share a core. Although the revised experiments extend the fault model from pure processor occupation to 16 MiB cache/memory pressure and shared-buffer communication, they do not directly test illegal memory access, driver failures, interrupt storms, network-I/O congestion, or the propagation of erroneous data. In particular, the current experiments do not constitute complete validation of EPT enforcement or interrupt-routing isolation. Native-Linux condition E represents an adverse scenario in which an application has real-time scheduling privileges and loses behavioral control; it does not imply that Linux systems configured with cgroups, CPU shielding, SELinux, or real-time bandwidth limits would necessarily behave identically. The conclusion is therefore limited to the specified hardware, static configuration, fault models, and observation intervals: PRTOS contains the direct scheduling expansion of the injected fault threads but does not demonstrate immunity to all shared-resource interference or universal latency superiority over native Linux.

6.4. Future Work

Future work will proceed in five directions: (1) port real flight control, airship management, and mission payload software to different partitions and establish end-to-end task chains and data freshness constraints; (2) combine hardware performance counters with repeated cache, memory bandwidth, interrupt, and I/O pressure experiments, and introduce controlled illegal memory access, unauthorized device access, and interrupt injection tests to evaluate EPT enforcement, device authorization, and interrupt-routing boundaries; (3) implement faulty-partition stop, reset, restart, and communication-state recovery and measure detection time, recovery time, and message resynchronization;

(4) conduct controlled comparisons between PRTOS and representative static-partitioning hypervisors, such as Jailhouse, Bao, XtratuM, or ACRN, using equivalent hardware, guest systems, resource assignments, workloads, and fault-injection conditions to distinguish PRTOS-specific behavior from the general characteristics of static partitioning; and (5) investigate primary/backup partitions, dual-computer switchover, and low-overhead virtual machine migration or online hypervisor update while quantifying the worst-case latency cost of these reliability mechanisms [32].

7. Conclusions

This study addresses fault propagation when multiple near-space airship avionics functions share a multicore computing platform by developing a four-partition static runtime prototype based on the PRTOS Type-1 hypervisor. On real Intel Atom x6425RE hardware, the implementation completes multicore startup, VMX/EPT enablement, loading of P0–P3 Linux guests, binding of vCPUs to physical cores, independent memory mapping, four-UART adaptation, and preconfigured inter-partition communication. It establishes a progressive experiment chain from a fault-free baseline through pure computational overload, 16 MiB read/write pressure, and communication pressure to native-Linux multicore propagation.

The results show zero deadline misses for all four tasks in condition A. In PRTOS conditions B, C, and D, the P2/T3 miss rates at $U_{P2} = 0.9$ are 6.002%, 0.004%, and 0.004%; at $U_{P2} = 1.1$ they rise to 99.800%, 99.806%, and 99.804%; and at $U_{P2} = 1.3$ all reach 100%. P0/T2, P1/T1, and P3/T4 retain zero misses in every 500 s pressure condition, so only one task is affected. Under severe pressure in condition D, P2-to-P0 message completeness falls to 48.586%, whereas the non-fault partition channels exhibit no integrity error or residual queue occupancy. In native-Linux condition E, the miss rates of P0/T2, P1/T1, and P2/T3 are 100% and that of P3/T4 is 97.133%, increasing the affected-task count to four.

Across task timing, communication service, and runtime boundaries, these results show that PRTOS confines direct processor competition from the P2 fault threads to CPU2. The fault first appears as degradation of P2's own task and external service and, under the tested configuration and observation intervals, does not propagate into deadline failure in P0, P1, or P3. The engineering significance is that processor, memory, and device boundaries are determined before system startup, creating an analyzable fault domain for integrated avionics on near-space airships. This conclusion does not mean that shared microarchitectural interference has been eliminated or that an overloaded partition retains service continuity. Hardware-counter measurements, repeated experiments, real avionics tasks, and partition-level recovery mechanisms are still required to verify cross-partition timing interference and recovery bounds.

The prototype provides an engineering validation basis for evolving near-space airship integrated avionics from physical isolation using dedicated computers to software partitioning on shared platforms.

Author Contributions: Conceptualization, Y.H. and Z.L.; Methodology, Y.H., Z.L. and Y.Y.; Software, J.Z.; Validation, Y.H.; Formal analysis, Y.Y.; Investigation, Y.H.; Data curation, J.Z.; Writing—original draft, Y.H.; Writing—review and editing, Y.H.; Visualization, Z.L. and J.Z.; Supervision, B.W.; Project administration, Y.H.; Funding acquisition, Y.H. All authors have read and agreed to the published version of the manuscript.

Funding: National Natural Science Foundation of China: 52227811.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

APIC	Advanced Programmable Interrupt Controller
EPT	Extended Page Table
HPET	High-Precision Event Timer
IMA	Integrated Modular Avionics
LAPIC	Local Advanced Programmable Interrupt Controller
PEF	PRTOS Executable Format
PRTOS	Partitioned Real-Time Operating System
UART	Universal Asynchronous Receiver/Transmitter
vCPU	Virtual Central Processing Unit
VMX	Virtual Machine Extension

References

1. Zuo, Z.; Song, J.; Zheng, Z.; Han, Q.-L. A survey on modelling, control and challenges of stratospheric airships. *Control Eng. Pract.* **2022**, *119*, 104979. [CrossRef]
2. Zhou, G.; Xu, J.; Ma, S.; Zong, J.; Shen, J.; Zhu, H. Review of key technologies for avionics systems integration on large passenger aircraft. *Acta Aeronaut. Astronaut. Sin.* **2024**, *45*, 529956.
3. RTCA. *DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*; RTCA: Washington, DC, USA, 2005.
4. SAE Industry Technologies Consortia. *Avionics Application Software Standard Interface, Part 1, Required Services*; ARINC Specification 653P1-6; SAE Industry Technologies Consortia: Warrendale, PA, USA, 2024.
5. SAE Industry Technologies Consortia IA. *Avionics Application Software Standard Interface, Part 0, Overview of ARINC 653*; ARINC Specification 653P0-4; SAE Industry Technologies Consortia IA: Bowie, MD, USA, 2025.
6. Lukić, B.; Ahlbrecht, A.; Friedrich, S.; Durak, U. State-of-the-art technologies for integrated modular avionics and the way ahead. In Proceedings of the 2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC), Barcelona, Spain, 1–5 October 2023; pp. 1–10.
7. Mueller, P.; Merckling, M.; Tietz, V.; Frey, C.; Luettig, B.; Waldvogel, A.; Annighoefer, B.; Abdo, K.; Halle, M.; Thielecke, F. Introduction of a dedicated platform level for ima systems development with an extensive automation tool support. In Proceedings of the 2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC), Barcelona, Spain, 1–5 October 2023; pp. 1–10.
8. Han, P.; Hu, W.; Zhai, Z.; Huang, M. A model-based optimization method of arinc 653 multicore partition scheduling. *Aerospace* **2024**, *11*, 915. [CrossRef]
9. Chen, J.; Zhai, Z.; Han, P.; Huang, M. Model-based partition scheduling of integrated modular avionics systems using genetic algorithm. *Sci. Rep.* **2025**, *15*, 31342. [CrossRef] [PubMed]
10. Kim, T. A practical cache partitioning method for multi-core processor on a commercial safety-critical partitioned RTOS. *IEEE Access* **2025**, *13*, 25505–25519. [CrossRef]
11. Lukić, B.; Friedrich, S.; Durak, U. A Streamlined Approach Toward Automated Generation and Validation of ARINC 653–Compliant Avionics Configurations. *J. Aerosp. Inf. Syst.* **2025**, *22*, 314–324. [CrossRef]
12. Lukić, B.; Friedrich, S.; Schubert, T.; Durak, U. Automated configuration of arinc 653-compliant avionics architectures. In Proceedings of the AIAA SciTech 2024 Forum, Orlando, FL, USA, 1–12 August 2024; p. 1856.
13. Lukić, B.; Nöldeke, P.; Durak, U.; Klimmek, M.; Jakob, S.; Gläser, M. From Architecture Models to a Target Platform: A Systematic Approach to Model-Driven Development of Dynamically Reconfigurable Avionics Systems. In Proceedings of the 2024 AIAA DATC/IEEE 43rd Digital Avionics Systems Conference (DASC), San Diego, CA, USA, 29 September–3 October 2024; pp. 1–10.
14. Heiser, G. *The sel4 Microkernel—An Introduction*; The seL4 Foundation: San Francisco, CA, USA, 2020; p. 1.
15. Bottaro, M.; Vardanega, T. Evaluating a multicore Mixed-Criticality System implementation against a temporal isolation kernel. *J. Syst. Archit.* **2022**, *130*, 102688. [CrossRef]
16. Buczyński, H.; Cabaj, K.; Pisarczyk, P. Resource partitioning in phoenix-RTOS for critical and noncritical software for UAV systems. In Proceedings of the 2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS), Sofia, Bulgaria, 4–7 September 2022; pp. 605–609.
17. Crespo, A.; Ripoll, I.; Masmano, M. Partitioned embedded architecture based on hypervisor: The XtratuM approach. In Proceedings of the 2010 European Dependable Computing Conference, Valencia, Spain, 28–30 April 2010; pp. 67–72.

18. Lozano, S.; Lugo, T.; Carretero, J. A comprehensive survey on the use of hypervisors in safety-critical systems. *IEEE Access* **2023**, *11*, 36244–36263. [CrossRef]
19. Hapka, R.; Christmann, A.; Ernst, R.; Kuzolap, A.; Hecker, P.; Rockschies, M.; Halle, M.; Thielecke, F. Safe usage of multi-cores in neural network avionics applications. In Proceedings of the 2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC), Barcelona, Spain, 1–5 October 2023; pp. 1–9.
20. Tinto, E.; Vardanega, T. Providing spatial isolation for Mixed-Criticality Systems. *J. Syst. Archit.* **2024**, *154*, 103234. [CrossRef]
21. Abeni, L.; Faggioli, D. Using Xen and KVM as real-time hypervisors. *J. Syst. Archit.* **2020**, *106*, 101709. [CrossRef]
22. Ramsauer, R.; Kiszka, J.; Lohmann, D.; Mauerer, W. Look mum, no VM exits! (almost). *arXiv* **2017**, arXiv:1705.06932.
23. Martins, J.; Tavares, A.; Solieri, M.; Bertogna, M.; Pinto, S. Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In Proceedings of the Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020), Bologna, Italy, 21 January 2020; pp. 3:1–3:14.
24. PRTOS Project. PRTOS Hypervisor: A Lightweight, Open-Source Embedded Hypervisor. Available online: https://github.com/prtos-project/prtos-hypervisor (accessed on 20 July 2026).
25. Intel Corporation. Intel Atom® x6425RE Processor: Product Specifications. Available online: https://www.intel.com/content/www/us/en/products/sku/207899/intel-atom-x6425re-processor-1-5m-cache-1-90-ghz/specifications.html (accessed on 20 July 2026).
26. Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3, Version 92*; Intel Corporation: Santa Clara, CA, USA, 2026. Available online: https://www.intel.com/content/www/us/en/content-details/922489/intel-64-and-ia-32-architectures-software-developer-s-manual-volume-3c-system-programming-guide-part-3.html (accessed on 20 July 2026).
27. Liu, C.L.; Layland, J.W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* **1973**, *20*, 46–61. [CrossRef]
28. *IEEE 1003.1-2024*; IEEE/Open Group Standard for Information Technology—Portable Operating System Interface (POSIX™) Base Specifications, Issue 8. IEEE and The Open Group: New York, NY, USA, 2024. [CrossRef]
29. Hsueh, M.-C.; Tsai, T.K.; Iyer, R.K. Fault injection techniques and tools. *Computer* **1997**, *30*, 75–82. [CrossRef]
30. Federal Aviation Administration. *Use of Multi-Core Processors*; Advisory Circular AC 20-193; U.S. Department of Transportation, Federal Aviation Administration: Washington, DC, USA, 2024. Available online: https://www.faa.gov/documentLibrary/media/Advisory_Circular/AC_20-193.pdf (accessed on 20 July 2026).
31. Farina, G.; Gala, G.; Cinque, M.; Fohler, G. Enabling memory access isolation in real-time cloud systems using intel's detection/regulation capabilities. *J. Syst. Archit.* **2023**, *137*, 102848. [CrossRef]
32. Mo, C.; Wang, L.; Li, S.; Hu, K.; Jiang, B. Rust-Shyper: A reliable embedded hypervisor supporting VM migration and hypervisor live-update. *J. Syst. Archit.* **2023**, *142*, 102948. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.