Towards Multi-Policy Hierarchical Scheduling in Linux for Containerized Space Applications

Merlin Kooshmanian[1,3], Jérôme Ermont[2], Lucas Miné[3], Simon Corbin[3] and Frédéric Boniol[1]

[1] *ONERA, Toulouse, France* *{firstname}.{lastname}@onera.fr*

[2] *IRIT, Toulouse, France* *{firstname}.{lastname}@irit.fr*

[3] *CNES, Toulouse, France* *{firstname}.{lastname}@cnes.fr*

ABSTRACT

This paper examines the challenges of running containerized flight software on Linux for space systems, focusing on how to achieve both predictability and flexibility in scheduling. While containers provide lightweight isolation, they complicate the temporal determinism required by safety-critical spacecraft. Hierarchical scheduling and support for multiple policies are increasingly necessary to accommodate heterogeneous onboard workloads, from hard real-time control loops to best-effort processing. To address these challenges, a methodology is introduced for designing, analyzing, and implementing hierarchical multi-policy scheduling on Linux. The approach is structured around a meta-model capturing the key concepts of hierarchical reservation, a formal analysis based on the Compositional Scheduling Framework (CSF) for sizing container budgets, and a kernel-level execution model built on Linux task groups. Building on these principles, the paper presents the Task Group Bandwidth Server (TGBS), a kernel mechanism that enforces per-container CPU reservations using deadline-server abstractions and full virtual runqueues. Unlike the Hierarchical Constant Bandwidth Server (HCBS), which is limited to real-time (RT) workloads, TGBS applies uniformly to fair-share (FAIR), RT, and deadline-based (DL) scheduling classes. The experimental evaluation assesses functional behavior, real-time latency, and the execution of workloads sized analytically. Functional tests confirm correct enforcement of hierarchical reservations and priority ordering across scheduling classes. Latency measurements show that the overhead introduced by virtual runqueues remains moderate and comparable to HCBS. Finally, schedulability experiments demonstrate that reservations dimensioned with CSF are respected for real-time tasks in a mixed-policy environment, while highlighting the need for extended analytical models to fully capture FAIR scheduling. Overall, the results indicate that hierarchical reservations provide a practical path toward predictable and isolated execution of mixed-policy workloads on Linux. The work establishes a foundation that links analytical reservation sizing with kernel mechanisms and identifies future directions, including multicore support and improved multi-policy schedulability analysis.

1. INTRODUCTION

Over the past two decades, spacecraft avionics hardware and onboard software have evolved significantly in complexity and capability. Modern spacecraft integrate increasingly sophisticated functionalities, leading to substantial growth in the size and complexity of flight software. In parallel, hardware advances have enabled unprecedented computing power, creating both opportunities and new challenges.

One major challenge is maintaining high levels of safety and reliability while managing growing software complexity. This has led to architectural strategies that emphasize functional isolation, allowing independent development, reuse, and local qualification of software components. In Europe, this approach has led to formalization efforts [27] and to a partial adoption of the Integrated Modular Avionics (IMA) concept in the space sector, particularly in France. This concept relies on Time and Space Partitioning (TSP) to group functions of varying criticalities on shared hardware platforms.

At the same time, the combination of rising software demands and hardware capabilities has opened the door to using more general-purpose, open-source operating systems like Linux. Once considered unsuitable for flight software due to safety constraints, Linux has gained traction in space systems, particularly within the New Space movement, driven by its maturity, functional richness, and flexibility [15]. In this context, it is often adopted as a compromise aiming to balance technical constraints with development and economic considerations. Ensuring robust isolation and predictability on Linux-based platforms therefore becomes a central challenge for mixed-criticality onboard software.

This work addresses these challenges by proposing a unified approach to hierarchical and multi-policy scheduling for containerized flight software on Linux. It combines a conceptual meta-model, an analytical method for sizing temporal reservations, and a kernel-level mechanism that enforces container-level CPU isolation across scheduling classes. The approach aims to support predictable and isolated execution of heterogeneous onboard workloads on Linux-based platforms.

Section 2 introduces the context, needs, and limitations that motivate hierarchical and multi-policy scheduling in Linux-based flight software. Section 3 reviews the most relevant related work. Section 4 presents the methodology, including the meta-model, formal model, simulation, and Linux implementation. Section 6 reports the experimental evaluation conducted on representative scenarios. Section 7 presents the results and discusses the benefits and limitations of the approach. Finally, Section 8 concludes the paper and outlines directions for future work.

2. Background and Motivation

This section introduces the context in which modern flight software operates, focusing on the need for strong isolation and the increasing use of Linux-based platforms. It outlines the diversity of execution requirements found in onboard systems and presents the limitations of existing Linux mechanisms. These elements establish the foundation for the challenges addressed in the remainder of the paper.

2.1. Flight Software Architecture and Isolation Needs

Modern spacecraft rely on onboard computers to execute functions such as attitude and orbit control, payload management, power regulation, and thermal supervision. These functions are implemented by the flight software, which consists of multiple interacting applications, complemented by services for monitoring, diagnostics, and fault handling. This software stack is typically organized in layers that range from low level drivers to high level mission applications, as described by Eickhoff [11]. As a result, flight software exhibits significant functional heterogeneity and encompasses a wide variety of operational, safety, and timing requirements.

To manage this complexity, space agencies have introduced principles derived from Integrated Modular Avionics. In Europe, this has led to the IMA-SP initiative and to the adoption of Time and Space Partitioning. Under this model, each software function is executed within an isolated partition, which prevents fault propagation and simplifies the qualification and integration processes [27, 28]. Several operational systems, such as the KOSMOS architecture, have demonstrated the practical benefits of this approach [13]. Historically, flight software relied on simple RTOS (e.g., early VxWorks or RTEMS) that did not natively support strong partitioning. As the need for certified isolation grew, this led to the development of partitioning hypervisors such as XtratuM and separation kernels such as VxWorks 653 and PikeOS, which implement ARINC 653 services and support DO-178C qualification. These solutions offer strong isolation guarantees but are typically proprietary and some rely on statically configured partitions, which can limit flexibility for rapidly evolving software stacks.

In the context considered in this work, isolation is provided through containers, each hosting a specific spacecraft function. Figure 1 illustrates an example of a containerized flight software architecture. The example shows that containers may embed diverse activities, including command and control, life cycle management, thermal regulation, power supervision, navigation and mission planning, image acquisition, image processing, and auxiliary services such as memory scrubbing or anomaly detection. These activities vary widely in their importance, operational profiles, and execution characteristics.

This structural isolation simplifies integration at the system level, since major spacecraft functions such as flight dynamics, thermal and power control, or payload operations are typically assigned to distinct subsystems that remain protected from one another. However, each subsystem still embeds activities with diverse operational objectives and execution characteristics, such as control loops, supervision tasks, data processing activities, and auxiliary services. The variety of execution profiles illustrated in Figure 1 therefore motivates mechanisms capable of maintaining isolation across subsystems while accommodating heterogeneous execution behaviors within each subsystem.

2.2. Linux for Space Systems

Linux has gained increasing attention in space systems due to its portability, openness, and extensive software ecosystem. As noted by Leppinen [15], its adoption in on-board computers has expanded as missions demand greater functional richness than traditional RTOS platforms were originally designed to provide. Linux offers broad support for programming languages, middleware, and device drivers; enables integration of advanced processing chains; and benefits from a large engineering community. Build systems such as Yocto or Buildroot allow tailoring distributions to mission needs, while extensions like PREEMPT-RT provide improved real-time responsiveness. Compared to partitioned RTOS such as PikeOS or VxWorks, Linux provides a much broader range of native high-level capabilities, modern programming languages, middleware frameworks, scientific and image-processing libraries, without requiring a guest OS, although it does not offer the same certification pedigree (e.g., DO-178C). These trends are reflected in initiatives such as Space Grade Linux [10], a Yocto-based effort within the ELISA (Enabling Linux In Safety Critical Applications) project that defines reusable Linux configurations and guidelines for space mission use.

The motivation for using Linux differs across mission profiles. Large institutional spacecraft typically rely on Linux to host complex payload processing or autonomy functions, while leaving the most critical control loops to certified RTOS and/or hypervisors. In these settings, Linux complements existing solutions and is often confined to dedicated computing units or hypervisor partitions, since it still lacks the certification required for high-criticality functions.

Conversely, NewSpace and university-class missions could adopt Linux as the foundation for the entire flight software stack. In these contexts, cost, rapid development, and software richness outweigh strict certification constraints. Linux enables small teams to integrate sophisticated capabilities quickly, making it an attractive alternative to bespoke RTOS-based designs.

Despite these differences, both categories of missions share the need to reinforce Linux's temporal predictability. Whether Linux hosts high-level functions alongside certified components or implements the full software stack, strong temporal isolation is required to support mixed-criticality workloads safely. The present work addresses this need by exploring hierarchical and multi-policy scheduling mechanisms suitable for Linux-based flight software, assuming, motivated by emerging industrial trends such as Linux-based avionics architectures, that Linux may play an increasingly central role in future space systems.

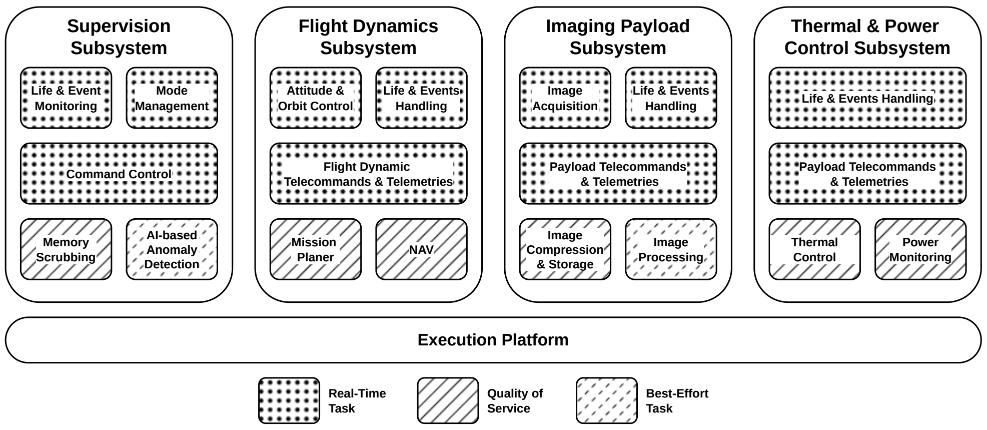

Figure 1. Example of a containerized satellite flight software architecture

2.3. Containerization on Linux-Based Platforms

Isolation on Linux-based platforms can be achieved through hypervisors, separation kernels, or operating-system-level virtualization. Hypervisors may be general-purpose, such as KVM or Xen, or lightweight micro-kernel-based systems such as XtratuM or Bao [9, 19]. These approaches replicate kernels, libraries, and runtime environments, increasing memory usage even when lightweight hypervisors are employed. This overhead is the counterpart of stronger isolation boundaries, since each guest instance carries its own execution environment and virtualized hardware interface. Unikernels can reduce duplication, although they generally provide limited real-time support and restricted software ecosystems [1].

Containers rely on namespaces and control groups to isolate applications within a single Linux instance, avoiding duplication of the operating system and resulting in much lower memory overhead. They also offer fine-grained control over resources, visibility, and execution environments, although their isolation guarantees remain weaker than those provided by hypervisors. This difference stems from the fact that containers share a single kernel and rely on kernel-level software mechanisms for isolation, whereas hypervisors enforce separation through hardware-supported virtualization boundaries. Experimental evaluations show that containers achieve low latency and modest resource usage [1], and can suit real-time workloads when coupled with appropriate kernel configurations [24]. Their compatibility with the full Linux ecosystem simplifies development and integration.

The relevance of containerization for embedded and space systems has motivated dedicated studies, including NASA investigations on containerized deployment and orchestration [14]. Containers support modular software composition, facilitate incremental updates, and provide a scalable structure for systems evolving during the mission, while also enabling orchestration of distributed workloads across multiple processing nodes.

Container-based isolation therefore offers a practical balance between flexibility, isolation, and resource efficiency for Linux-based flight software, especially in architectures where multiple subsystems must coexist on the same execution platform.

2.4. Heterogeneous Execution Requirements

On-board software integrates activities with different functional roles and timing expectations, whose coexistence within a single subsystem leads to heterogeneous scheduling needs. The example in Figure 1 illustrates this diversity through control loops, event-driven services, navigation functions, storage management, and background processing. These activities can be grouped into three classes according to the level of execution guarantees they require:

- **Real-Time (RT)** tasks require strong guarantees on reliability and reactivity. Their execution must be ensured at each activation with bounded response time, since they contribute directly to attitude and orbit control, sensor and actuator handling, or fast event processing. These tasks run at relatively high frequencies and cannot tolerate long preemptions.
- **Quality-of-Service (QoS)** tasks require reliable execution with more flexible timing. They may run infrequently, as in navigation updates or mission planning, or may tolerate prolonged preemptions, as in memory scrubbing or storage management. They do not require strict responsiveness but must make predictable progress over time, with reserved execution capacity.
- **Best-Effort (BE)** tasks have no strict timing constraints. They include maintenance activities and opportunistic workloads such as data post-processing or experimental algorithms. Their execution is useful but not critical and can be delayed without compromising system safety.

These classes reflect the intrinsic heterogeneity of flight software and motivate differentiated scheduling behavior within each subsystem, enabling strong guarantees for RT activities, controlled progress for QoS tasks, and opportunistic resource use for BE workloads.

2.5. Multi-policy and Hierarchical Scheduling

Multi-policy scheduling denotes the ability of an operating system to employ several scheduling algorithms concurrently, each tailored to a specific type of workload. Prior work already emphasized that a single fixed scheduling policy is rarely adequate for heterogeneous systems, since objectives such as responsiveness, fairness, or deadline adherence cannot be satisfied simultaneously by one mechanism [12]. Although a unified policy such as fixed-priority scheduling can still support all task classes, it often forces compromises that limit the ability to meet the objectives of certain tasks. In flight software, where activities exhibit diverse requirements and operational objectives, multi-policy scheduling allows each class to be managed by a policy suited to its characteristics.

In this work, a distinction is made between hierarchical scheduling and multi-policy scheduling. Hierarchical scheduling refers to the structuring of scheduling decisions across multiple levels, where higher-level entities (e.g., containers or servers) are allocated execution budgets and scheduled as a whole [18]. Multi-policy scheduling refers to the ability of a single scheduling entity to execute tasks governed by different scheduling policies under a shared budget. While multi-policy systems can be modeled as hierarchical in a formal sense, this distinction is relevant in practice: classical hierarchical scheduling frameworks typically assume mono-policy subsystems, whereas Linux-based subsystems natively support the concurrent use of deadline-based, real-time, and fair-share scheduling within the same execution context.

In Linux, these heterogeneous execution requirements are addressed through multiple scheduling classes providing distinct execution semantics. Real-time scheduling classes such as `SCHED_FIFO` and `SCHED_RR` offer strict priority-based execution with bounded latency for time-critical activities, differing in how tasks of equal priority are scheduled, using first-in-first-out and round-robin principles, respectively. Deadline-based scheduling, implemented by `SCHED_DEADLINE`, targets workloads with explicit temporal constraints by enforcing execution budgets and deadlines. Fair-share scheduling, implemented by `SCHED_OTHER`, aims to distribute available processor time dynamically among tasks and to prevent starvation, making it suitable for workloads with variable or loosely specified execution demand. While similar behavior can be approximated using fixed-priority and round-robin scheduling alone, such configurations lack the dynamic fairness mechanisms provided by `SCHED_OTHER`. In practice, these scheduling classes naturally lend themselves to different execution requirements, motivating the association of task classes with suitable scheduling policies, although this mapping is not prescribed in this work.

2.6. CBS and its Integration in Linux

The Constant Bandwidth Server (CBS) is a scheduling policy designed to provide temporal isolation among tasks by enforcing a fixed reservation of processor time [2]. Each schedulable entity, typically a task, but possibly a server hosting multiple tasks, is assigned a budget–period pair (Q, P). The server guarantees that the entity can execute for at most Q units of time every P units. When the budget is exhausted, the server postpones its deadline by P, ensuring that the entity cannot exceed its reserved share while preserving EDF ordering. This deadline-postponement rule is what provides the core isolation property: the behavior of one entity cannot reduce the reserved execution of another.

The Greedy Reclamation of Unused Bandwidth (GRUB) policy extends the CBS model by redistributing unused execution capacity to active servers while preserving temporal isolation [16]. HCBS generalizes CBS to hierarchical contexts by allowing servers to be composed across multiple levels, enabling reservations to be nested within subsystems [17]. These mechanisms support analytical reasoning through supply-bound functions and form a foundation for predictable resource allocation in real-time systems.

Linux integrates CBS semantics through the `SCHED_DEADLINE` scheduling class. Since version 3.14, each `SCHED_DEADLINE` task is represented as a `sched_dl_entity` parameterized by (Q, P) and scheduled using EDF with CBS enforcement. More recently, from version 6.12 onward, Linux extends this interface to support *deadline servers*, enabling a group of tasks to share a single CBS reservation. This provides a unified mechanism for both task-level and group-level temporal isolation, and exposes CBS-based reservations as first-class scheduling primitives within the kernel.

2.7. Limitations of Current Linux Approaches

Linux offers several properties that make it a strong candidate for future satellite flight software, including a rich software ecosystem, flexible configuration mechanisms, and support for both dynamic and multi-policy scheduling. These capabilities provide a foundation for managing heterogeneous workloads efficiently, allowing different types of activities to coexist while pursuing distinct execution objectives. Combined with container-based isolation, Linux enables each subsystem to be encapsulated within an independent functional boundary, simplifying integration and deployment.

However, the temporal isolation mechanisms currently available in Linux remain limited for the type of subsystem separation expected in flight software. Cgroups regulate CPU bandwidth on a per-container, per-class basis, which constrains isolation to the structure imposed by Linux scheduling classes. Their temporal behavior is moreover difficult to predict, as observed by Abeni [3], limiting their suitability for enforcing subsystem-level guarantees. This model does not match the way flight software is typically structured, where isolation is defined at the level of functional subsystems rather than scheduling classes. This raises a central challenge: how can hierarchical, multi-policy scheduling in Linux be enabled to meet the heterogeneous and safety-critical needs of containerized spacecraft flight software?

3. Related-Work

Several works have explored the use of containers to provide temporal isolation in real-time or mixed-criticality systems. RT-CASEs [8] introduced one of the earliest container-based approaches, combining Docker isolation with temporal control through the RTAI real-time extension. While the work demonstrated the feasibility of using containers in embedded real-time contexts, its reliance on RTAI limited portability.

More recent work by Cazanove et al. [7] proposed a Linux container-based architecture for partitioning real-time task sets on ARM multicore processors, replacing RTAI with PREEMPT-RT. Their approach focuses on reproducing static time-partitioning behaviour across containers, but does not provide hierarchical scheduling or reservation-based interfaces comparable to those used in real-time hypervisors.

To improve temporal control, several extensions build on reservation mechanisms. Abeni introduced the Hierarchical Constant Bandwidth Server (HCBS) patch [3], which extends `SCHED_DEADLINE` to cgroups through Constant Bandwidth Server (CBS) reservations. HCBS enables a two-level model: container-level reservations based on deadline servers, combined with an internal scheduling policy (e.g., fixed priority) inside each container. Crucially, HCBS was designed to be analyzable through the Compositional Scheduling Framework (CSF) [21], enabling predictable execution for hierarchical real-time systems. More recently, Andriaccio and Abeni proposed an updated version of HCBS [4, 5] that leverages the deadline servers introduced in Linux v6.12, providing a cleaner and more integrated API for grouping tasks under deadline entities. However, HCBS applies only to real-time tasks and does not extend to other Linux scheduling classes, making it unsuitable for systems that require multi-policy scheduling.

Building on HCBS, Struhár et al. proposed the REACT framework [26, 25], which introduces hierarchical resource orchestration for real-time containers. REACT combines offline resource allocation with online re-dimensioning to mitigate interference effects. Its implementation relies on HCBS to assign CPU reservations to containers and adjust them dynamically, but remains constrained to real-time workloads and does not address the coexistence of multiple scheduling policies within container boundaries.

Beyond implementation mechanisms, the Compositional Scheduling Framework (CSF) [21] provides a foundational analytical model for hierarchical systems. CSF abstracts each component as a schedulable entity with a periodic resource contract (Q, P) and evaluates its feasibility by comparing the component's supply bound function (SBF) with the demand bound function (DBF) of its internal workload. A key advantage of CSF is its compositional nature: each component can be analyzed and dimensioned independently of the rest of the system, which is particularly useful in dynamic architectures where containers or virtual machines may appear, disappear, or migrate. This modularity, however, comes at the cost of pessimism, since each component is analyzed under worst-case assumptions that may exceed the behaviour observed in practice. CSF has been extended to periodic [22] and multicore systems [23], making it applicable to container-based execution platforms and suitable for deriving temporal budgets in hierarchical scheduling models. Nevertheless, existing formulations do not provide models for components that host multiple scheduling policies, leaving multi-policy containers outside the current analytical scope.

4. METHODOLOGY

To address the challenges identified in Section 2, the methodology establishes a structured process that links the conceptual description of scheduling systems with their execution on real platforms. The objective is to ensure coherence across all levels, from the definition of concepts to analytical prediction, simulation, and implementation.

Figure 2 summarizes the main components of this process. At its foundation lies the *meta-model*, which defines and organizes the concepts of hierarchical, multi-core, and multi-policy scheduling, such as containers, runqueues, tasks, and scheduling policies. Informed by real-world requirements, the meta-model can be instantiated into *a system description*, providing a concrete representation of an architecture, for example a containerized flight software.

From each system description, two complementary artifacts are derived:

- **Formal models**, which provide a conservative prediction of behavior, suitable for schedulability analysis.
- **Real platform**, which executes the system description on hardware and operating system support, yielding the observed behavior of the system.

The methodology therefore produces two complementary views of execution: a predicted behavior derived from formal analysis, and a real behavior measured on the target platform. The predicted behavior is conservative by construction, while the real behavior reflects implementation effects and hardware variability.

4.1. Hierarchical Multi-Policy Meta-Model

The meta-model defines the concepts required to represent hierarchical, multi-core, and multi-policy systems in a structured and extensible manner. It is expressed in UML, which provides a systematic way to capture entities and their relations. Unlike formal models, which provide analytical abstractions, the meta-model serves as a descriptive architecture that can be instantiated into system descriptions and used as a foundation for simulation and further analysis.

The main concepts are summarized as follows:

- **System**: global view including the number of CPUs, memory, storage, and file-system elements.
- **Containers**: execution environments relying on virtual CPUs (vCPUs), represented in the system as server-type tasks that deliver CPU time to other tasks. Each vCPU is parameterized by (Q, P), where Q is the temporal budget and P the replenishment period, providing controlled CPU allocation and temporal isolation. Containers may also be associated with additional budgets for memory or storage.
- **Tasks**: execution entities of three types, namely Real-Time (RT), Quality-of-Service (QoS), and Best-Effort (BE). RT tasks are characterized by deadlines and fixed priorities; QoS tasks are associated with server-based budgets and periods; BE tasks follow fair-share parameters such as the Linux `nice` value.
- **Schedulers and runqueues**: elements implementing scheduling policies, including fixed-priority, EDF, and fair-share scheduling, by managing queues of ready tasks.
- **Resources and communication**: synchronization primitives (mutexes, semaphores, locks) and communication buffers, modeling contention and inter-container interactions.

This modeling captures the hierarchical and multi-policy characteristics required for the use case considered. It supports

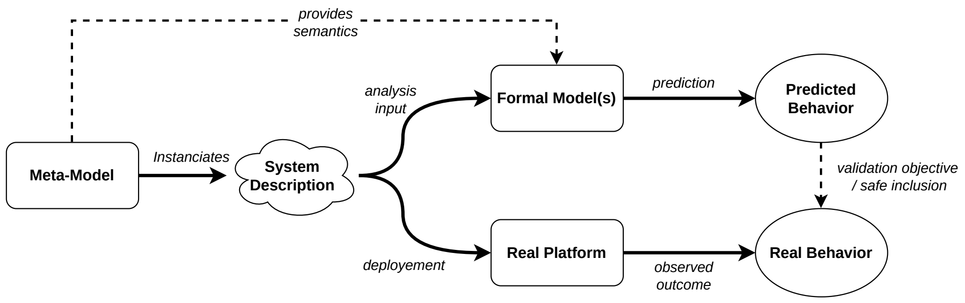

Figure 2. Overview of methodology

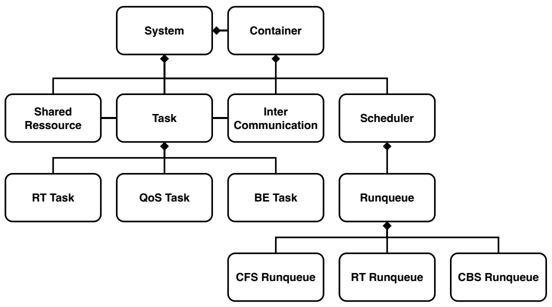

Figure 3. Simplified meta-model

the description of the containerized flight-software architecture studied in this work and provides a vocabulary aligned with Linux mechanisms. The full UML diagram is too detailed to reproduce here; instead, Figure 3 presents a simplified view focusing on the main elements.

4.2. Formal Model and Schedulability Analysis

Formal models are used to predict timing behavior and derive guarantees for each container. The primary objective is to ensure that the real-time (RT) workload is schedulable, meaning that all its timing constraints are met under the allocated execution budget. Quality-of-Service (QoS) tasks are also included to account for their processor demand and ensure sufficient execution capacity, although no strict deadline guarantees are assumed for this class.

As introduced in Section 3, the Compositional Scheduling Framework (CSF) provides the analytical basis for deriving periodic resource contracts for hierarchical systems. Since CSF supports only mono-policy fixed-priority analysis and does not handle tasks with identical priorities, the multi-policy workload is abstracted into a single fixed-priority model with distinct priorities for all tasks.

All tasks within a container, including both RT and QoS tasks, are therefore modeled as periodic fixed-priority tasks for analytical purposes. Priorities follow a global Rate-Monotonic ordering, with RT tasks assigned the highest priority levels and QoS tasks assigned lower priority levels, also ordered by period. This strict total ordering preserves the intended execution hierarchy between classes while maintaining a consistent timing-based ordering within each class, preserves overall workload utilization, and enables the use of standard fixed-priority demand-bound functions within CSF. The resulting workload model is used to compute a periodic reservation (Q, P) that guarantees schedulability of the RT workload inside the container.

While this mono-policy abstraction provides a practical and conservative means of sizing container-level reservations, it does not capture the runtime interactions introduced by genuinely mixed scheduling policies. In particular, the behavior of QoS tasks scheduled using fair-share mechanisms is only approximated through their contribution to processor demand, motivating the development of analytical models capable of reasoning explicitly about multi-policy execution within a shared reservation.

4.3. Real Platform

The real platform provides the empirical counterpart of the model-based predictions. Experiments are conducted on a Linux kernel configured with the PREEMPT-RT real-time extension, available either through external patching or natively in kernels starting from version 6.12. PREEMPT-RT is used to obtain a fully preemptible kernel and to reduce scheduling latency, providing the responsiveness required for real-time experimentation [20]. On top of this baseline, a custom patch called TGBS is applied to enable hierarchical and multi-policy scheduling through deadline-server-based temporal reservations. The implementation of TGBS is detailed in Section 5. This platform executes the instantiated system description and produces the observed timing behavior used for validation.

4.4. Comparison and Validation

The validation process compares the behavior predicted by the formal model with that observed on the real platform, using metrics relevant to temporal correctness and isolation, such as response times, jitter, deadline satisfaction, or the containment of overruns, depending on the property under evaluation. The objective is to verify that the formal model provides a safe abstraction of the system, such that predicted bounds consistently cover the observed timing behavior; any discrepancies are used to refine the meta-model or adjust formal assumptions to improve alignment between conceptual and actual execution.

5. Kernel Implementation

The kernel support for the approach presented in this paper is implemented through a dedicated patch, named *Task Group Bandwidth Server* (TGBS). When enabled through a configurable option in `menuconfig`, TGBS activates a hierarchical and multi-policy scheduling mode in the Linux kernel. Its purpose is to provide container-level temporal reservations that apply uniformly across all scheduling classes, extending the principles of hierarchical bandwidth management beyond real-time workloads.

TGBS builds on the conceptual foundations introduced by the Hierarchical Constant Bandwidth Server (HCBS) discussed in Section 3. HCBS demonstrated how a Constant Bandwidth Server (CBS) can serve as a proxy for a task group by exposing its parameters through the cgroup interface and enforcing the corresponding budget through a deadline server attached to the RT runqueue. This design effectively replaces the vanilla task-group bandwidth controller with a deadline-based mechanism, but its scope is limited to RT tasks: only entities belonging to the RT scheduling class benefit from the hierarchical isolation provided by HCBS.

The objective of TGBS is broader. Rather than providing isolation for RT tasks only, the goal is to use the deadline server abstraction as a *virtual CPU* capable of hosting tasks of any scheduling class. The interface intentionally mirrors HCBS, reusing the same cgroup control files (`runtime_us` and `period_us`) and the same hierarchical admission rules, in order to maintain compatibility and avoid reintroducing already established mechanisms. However, the scheduling architecture diverges significantly.

5.1. Virtual runqueues

While HCBS attaches its deadline server to a dedicated RT sub-runqueue (`struct rt_rq`), TGBS associates each task group with a *virtual runqueue*. This virtual runqueue is a lightweight replica of the full per-CPU runqueue structure (`struct rq`): it embeds the FAIR, RT, and DL class sub-runqueues, and therefore supports all scheduling classes uniformly. All runnable tasks inside the group are enqueued on the virtual runqueue, which is then represented as a single schedulable entity, the deadline server, on the physical CPU runqueue.

This generalization brings two key advantages. First, it enables true multi-policy behavior: tasks of different Linux scheduling classes coexist under the same temporal reservation. Second, it naturally inherits future improvements of each scheduling class, since the virtual runqueue reuses native kernel code paths. In contrast, HCBS bypasses the standard FAIR and DL queues entirely, making it fundamentally RT-only.

This architectural divergence is illustrated in Figure 4, which contrasts the HCBS design, based on a deadline server hosting only an RT sub-runqueue, with the TGBS approach, where the server encapsulates a complete runqueue and therefore enables true multi-policy scheduling.

5.2. Synchronizing virtual and physical contexts

Using a full virtual runqueue introduces several implementation challenges. Since the virtual runqueue behaves as an independent scheduling domain, it must remain consistent with its physical counterpart. To ensure correct operation, TGBS introduces dedicated helpers to synchronize scheduler clocks between virtual and physical contexts, propagate periodic scheduler ticks into the virtual runqueue, mirror runtime and accounting updates, and preserve the locking discipline expected by the kernel's runqueue operations.

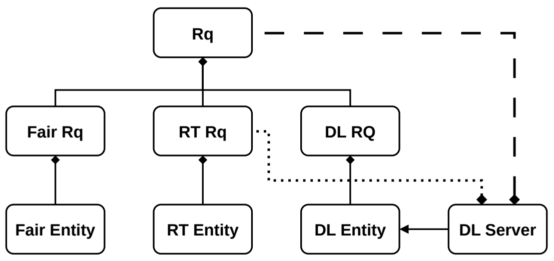

Figure 4. Simplified view of scheduling-class relationships in Linux. Dotted arrows show how HCBS attaches a deadline server to an RT-only runqueue. Dashed arrows show how TGBS generalizes this design by attaching a full runqueue to the server.

Without these mechanisms, standard scheduler operations such as throttling, waking, preemption, or load balancing could violate invariants assumed by the kernel, potentially causing priority inversions or deadlocks. By maintaining consistency between the two contexts, TGBS allows the virtual runqueue to be treated almost like a native runqueue from the kernel's perspective.

5.3. Interface and behavior

Because the cgroup interface matches that of HCBS, configuring a task group is straightforward: administrators specify a runtime and a period, and hierarchical bandwidth checks ensure that children do not exceed their parent's capacity. When the first task in the group becomes runnable, the associated deadline server starts consuming budget; when the virtual runqueue is empty, the server suspends until the next replenishment. Unlike HCBS, however, every task in the group contributes to budget consumption, regardless of scheduling class. Preemption across classes is preserved, as tasks continue to follow standard Linux scheduling semantics within the group reservation.

5.4. Current status and multicore support

The current implementation supports single-core processors and has been validated through stress testing, locking-dependency analysis, and preemption verification. This version is released under the tag `tgbs-v1.0`. Multicore support is a work in progress. Since TGBS creates one deadline server per CPU, migrating a task across CPUs inherently moves it between virtual CPUs. Unlike HCBS, which attaches servers to RT-only queues and can rely on native load-balancing mechanisms, TGBS manages full virtual runqueues, which complicates migration. In particular, migration must remain correct when a server is throttled or replenished, and must avoid undesirable interactions between per-CPU locks, which could otherwise lead to deadlocks. Addressing these challenges is ongoing work toward full multicore support.

6. Experimental Evaluation

This section evaluates the proposed hierarchical and multi-policy scheduling approach on a real Linux platform extended with TGBS. The objective is to illustrate how temporal reservations interact with mixed workloads and to assess (i) the functional correctness of hierarchical multi-policy scheduling, (ii) the schedulability of analytically sized container reservations, and (iii) the impact of TGBS on real-time performance compared to vanilla Linux and HCBS. The experiments are not intended as an exhaustive performance characterization, but rather to demonstrate how the methodology of Section 4 can be applied to construct and analyze realistic scenarios.

6.1. Assumptions

To focus the evaluation on temporal behavior and simplify the interpretation of results, the following assumptions are applied:

- tasks are independent and do not share resources;
- execution times equal their worst-case execution times (WCETs);
- deadlines equal periods;
- task offsets are equal to zero;
- Best-Effort (BE) tasks are not explicitly modeled;

6.2. Experimental Setup

All experiments were conducted on a Zybo Z7 development board equipped with a dual-core ARM Cortex-A9 processor at 666 MHz and 1 GB of DDR3 memory. As detailed in Section 5, the current TGBS prototype supports only single-core configurations; therefore, one core was disabled for all tests. The software environment is based on Linux v6.17 configured with the PREEMPT-RT extension for ARM32, providing a fully preemptible kernel suitable for low-latency experimentation. The TGBS patch[1] was applied to enable hierarchical and multi-policy scheduling. The kernel was compiled with PREEMPT-RT, control groups, and tracing support, and a minimal root filesystem was generated using Buildroot 2025.02.1.

6.3. Test Scenarios

Three scenarios were evaluated, each targeting a specific property of the approach.

6.3.1. Test 1 : Functional Validation

This test verifies that TGBS correctly enforces a container-level reservation while allowing tasks of different policies (DL, RT, OTHER) to coexist under a single temporal contract. A single container is configured with a reservation of 600 ms every 1 s (60% CPU). The workload consists of one `SCHED_DEADLINE` task with a 100 ms execution demand every 1 s, one `SCHED_FIFO` task with a 200 ms execution demand every 1 s, and two CPU-bound `SCHED_OTHER` tasks competing for the remaining capacity.

Under an ideal hierarchical and multi-policy scheduler, these tasks should respectively consume approximately 10%, 20%, and 15%–15% of the CPU time. The objective is to demonstrate qualitatively that TGBS enforces this distribution, validating both multi-policy support and hierarchical budget enforcement.

[1] `https://github.com/mkooshmanian/TGBS`, tag `tgbs-v1.0`

Table 1. Randomly generated task set (effective $U \approx 54\%$).

Task	Container	WCET (ms)	Period (ms)	Class
T1	A	10	350	RT
T2	A	10	630	RT
T3	A	10	720	QoS
T4	A	30	770	QoS
T5	A	120	880	QoS
T6	B	10	120	RT
T7	B	10	210	RT
T8	B	50	480	RT
T9	B	10	700	QoS
T10	B	50	880	QoS

6.3.2. Test 2 : Real-time Performance Evaluation

This scenario evaluates the latency overhead introduced by TGBS. The `cyclictest` benchmark is used to measure wake-up latency, defined as the time between the expected activation of a periodic real-time thread and its actual execution. The benchmark is executed with a high real-time priority (e.g., `SCHED_FIFO` 99) to isolate kernel-induced latency and serves as a standard reference for assessing real-time behavior in Linux.

The benchmark is executed under three configurations:

1. baseline Linux (PREEMPT-RT);
2. Linux (PREEMPT-RT) + HCBS;
3. Linux (PREEMPT-RT) + TGBS.

For HCBS and TGBS, the benchmark runs inside a container so that the measured latency reflects the overhead induced by hierarchical scheduling.

The objective is to quantify the additional wake-up latency introduced by the virtual-runqueue mechanism of TGBS and to compare it against both the baseline and HCBS. Because TGBS introduces an additional scheduling indirection, some increase in latency is expected; however, this overhead should remain moderate.

6.3.3. Test 3 : Validation of the Proposed Methodology

This final test validates the end-to-end methodology introduced in Section 4, from workload modelling to analytical sizing and real execution under TGBS. A synthetic workload of $n = 10$ periodic tasks with a target total utilization of $U = 50\% \pm 5\%$ is generated using the UUniFast algorithm [6] and distributed evenly across two containers, denoted A and B. Task periods are randomly selected from a bounded set of integer values, and execution times are derived from the generated utilizations according to $C_i = U_i \cdot T_i$, rounded to integer values. Each container hosts a mix of real-time (RT) tasks executed under `SCHED_FIFO` with priorities assigned according to Rate-Monotonic (RM) ordering, and QoS tasks executed under `SCHED_OTHER`. These policy assignments are chosen arbitrarily for the purpose of exercising the multi-policy execution model and do not necessarily reflect the optimal association between application classes and Linux scheduling policies in a flight-software context.

For the analytical phase, all tasks, RT and QoS, are mapped to a single fixed-priority model so that the workload fits the CSF's mono-policy assumption. QoS tasks are therefore assigned priorities lower than those of all RT tasks within their

Table 2. CPU usage under a 600 ms / 1 s container reservation.

Task	Policy	CPU share
DL task (100ms / 1s)	`DEADLINE`	$\approx 10\%$
RT task (200ms / 1s)	`FIFO`	$\approx 20\%$
FAIR task (stresser #1)	`OTHER`	$\approx 15\%$
FAIR task (stresser #2)	`OTHER`	$\approx 15\%$

container, allowing the full task set to be evaluated using standard RM demand-bound functions. This abstraction yields, for each container, a periodic reservation (Q, P) guaranteeing schedulability under the CSF supply-bound model. Table 1 summarizes the resulting task set, including WCET, period, container assignment, and task class.

The containers are then executed under TGBS on the real platform, and execution traces are collected to evaluate deadline satisfaction, response times, and budget usage. Because the current CSF framework only supports single-policy fixed-priority analysis, the analytical model necessarily treats all tasks under the RM abstraction, even though the runtime system is genuinely multi-policy with `SCHED_FIFO` for RT tasks and `SCHED_OTHER` for QoS tasks. This approximation preserves the relative prioritization of RT tasks but does not fully capture the scheduling dynamics of QoS tasks inside each container. As a result, differences between analytical predictions and measured execution, particularly for QoS tasks, are expected and form part of the evaluation.

The purpose of this experiment is therefore twofold. First, it assesses the observable behavior of the mixed RT/QoS workload when executed under TGBS, including deadline satisfaction, response times, and reservation usage. Second, it provides an initial comparison between mono-policy and multi-policy execution under the same CSF-derived reservation sizing. Rather than exposing boundary cases, the results primarily establish a baseline and motivate further investigation into scenarios where mixed scheduling policies may approach or stress reservation limits, thereby informing the development of analytical models capable of reasoning about multi-policy execution.

7. RESULTS

This section presents the results of the three experimental scenarios introduced in Section 6. The outcomes of each scenario are reported below, followed by a brief discussion.

7.1. Test 1 : Functional Validation

Table 2 summarizes the CPU usage reported by `top` during steady-state execution. The measured CPU shares closely match the expected proportions: the DL task consumes approximately 10% of the processor, the RT task 20%, and the two FAIR tasks share the remaining budget equally.

These results confirm that TGBS correctly enforces the container reservation while preserving the inter-class priority hierarchy: the DL task receives precedence over RT work, and the remaining FAIR tasks share the leftover capacity. The observed behavior is consistent with the expected semantics of hierarchical multi-policy scheduling.

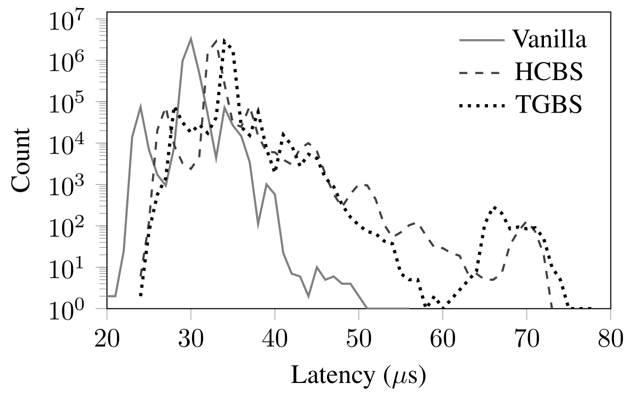

Figure 5. Latency distribution measured by `cyclictest` (`-m -S -p99 -i200 -l5000000 -h200 -q`) under Vanilla, HCBS, and TGBS.

Table 3. Cyclictest latency statistics (μs).

System	Min	Avg	Max	Std Dev
Vanilla	20	30	56	1.3
HCBS	24	33	73	1.7
TGBS	24	34	78	1.5

7.2. Test 2 : Real-time Performance Evaluation

Figure 5 reports the latency distributions obtained with `cyclictest` under Vanilla Linux, HCBS, and TGBS. Table 3 summarizes the corresponding statistics. As expected, both HCBS and TGBS introduce additional latency compared to the baseline, reflecting the extra scheduling indirection created when tasks execute through a reservation server rather than directly on the physical runqueue. This results in a small increase in average latency, from 30 μs (Vanilla) to 33 μs (HCBS) and 34 μs (TGBS).

A similar effect is visible in the maximum latency, which increases from 56 μs (Vanilla) to 73 μs (HCBS) and 78 μs (TGBS). These higher values remain within the same order of magnitude and do not indicate pathological behavior. TGBS shows a latency profile close to HCBS even though it virtualizes the entire runqueue rather than only the RT sub-runqueue. This suggests that the additional indirection introduced by the virtual runqueue does not significantly inflate wake-up delays.

Overall, these measurements show that the latency overhead introduced by TGBS is moderate and comparable to HCBS. The hierarchical multi-policy mechanism preserves the responsiveness expected from a real-time Linux configuration, confirming that TGBS remains suitable for real-time workloads.

7.3. Test 3 : Validation of the Proposed Methodology

Table 4 reports the response times of all tasks over one hyperperiod for both execution scenarios. Across both configurations, no deadline misses were observed, indicating that the CSF-derived reservations provide sufficient execution capacity for the evaluated workload. While this outcome is expected for the mono-policy RM configuration, it also confirms that the same reservation sizing sustains correct execution under the multi-policy case.

The response times of RT tasks (*T1*, *T2*, *T6*, *T7*, *T8*) remain

Table 4. Response times of tasks over one hyperperiod (554 400 ms) for multi-policy and mono-policy scenarios, including the effective scheduling policy and priority of each task (priority: 99 = highest, 1 = lowest).

	Multi-Policy						**Mono-Policy**					
Task	Policy	Priority	Min (ms)	Avg (ms)	Max (ms)	Std (ms)	Policy	Priority	Min (ms)	Avg (ms)	Max (ms)	Std (ms)
T1	`FIFO`	97	14.2	21.3	43.9	8.1	`FIFO`	97	14.3	21.5	44.9	8.0
T2	`FIFO`	95	14.5	20.4	34.6	7.4	`FIFO`	95	14.5	22.5	56.4	8.9
T3	`OTHER`	–	14.9	70.5	195.6	37.0	`FIFO`	93	14.8	25.3	94.8	13.3
T4	`OTHER`	–	55.2	160.7	295.6	56.3	`FIFO`	92	75.2	98.2	175.3	23.0
T5	`OTHER`	–	254.5	470.5	565.3	67.2	`FIFO`	91	374.2	480.6	554.2	58.1
T6	`FIFO`	99	14.5	25.2	94.6	14.4	`FIFO`	99	14.4	25.0	94.5	14.6
T7	`FIFO`	98	14.9	32.4	83.5	17.6	`FIFO`	98	14.9	34.4	75.7	17.7
T8	`FIFO`	96	114.4	171.8	254.0	24.9	`FIFO`	96	114.4	171.0	264.7	22.5
T9	`OTHER`	–	14.5	89.2	374.0	79.1	`FIFO`	94	14.3	66.8	254.7	59.7
T10	`OTHER`	–	144.3	320.7	484.0	88.6	`FIFO`	90	133.4	320.1	474.0	93.6

comparable across both scenarios, with only minor variations in minimum, average, and maximum values. Since QoS tasks scheduled under `SCHED_OTHER` cannot preempt real-time entities, their presence does not affect the execution of higher-priority tasks in either configuration.

QoS tasks exhibit larger differences between configurations. In the multi-policy case, execution under CFS within a periodic reservation results in higher variability and increased average latencies for several tasks, reflecting fair-share behaviour combined with periodic budget replenishment. In contrast, the mono-policy configuration yields more stable timing due to deterministic fixed-priority ordering. These differences remain bounded and compatible with the predefined reservations.

Overall, the workload remains schedulable under both mono-policy and multi-policy configurations, while exhibiting expected behavioural differences between fair-share and fixed-priority scheduling.

7.4. Discussion

The experiments provide an initial validation of TGBS across functional, latency, and schedulability dimensions. Functional evaluation confirms correct enforcement of hierarchical reservations and coherent multi-policy behaviour, while latency measurements show a moderate overhead comparable to HCBS. The schedulability experiment further indicates that the analytically sized reservations are sufficient for the evaluated workload in both mono-policy and multi-policy configurations, with no deadline misses observed.

The comparison between configurations highlights expected behavioural differences: real-time tasks remain largely unaffected, while QoS tasks exhibit higher variability under multi-policy execution due to fair-share scheduling and periodic replenishment. These effects remain bounded and consistent with policy semantics. Since the evaluated workload does not reach a boundary case of the analytical sizing, future work should investigate configurations where mono-policy execution remains schedulable while the multi-policy case approaches or exceeds reservation limits.

Several limitations remain. The evaluation was conducted on a single hardware platform and a limited set of workloads, motivating broader validation across architectures, kernel configurations, and more demanding task sets. The current prototype (`tgbs-v1.0`) supports only single-core execution; enabling multicore operation will require safe task migration between virtual runqueues and additional timing verification. Moreover, analytical sizing relies on a mono-policy RM/CSF formulation, which does not capture interactions introduced by mixed scheduling policies, indicating the need for an analytical framework capable of reasoning about multi-policy execution within a reservation.

Overall, the results indicate that hierarchical multi-policy scheduling with TGBS is feasible for the evaluated scenarios, while highlighting the need for further refinement of both the analytical model and the kernel mechanism to support stronger predictability under more challenging conditions.

8. Conclusion

This work presented a methodology and a kernel-level mechanism to support hierarchical and multi-policy scheduling on Linux for containerized flight software. The approach combines a meta-model for system structuring, a CSF-based analytical method for reservation sizing, and the TGBS patch providing uniform enforcement of reservations across scheduling classes.

The experiments demonstrate the practical feasibility of hierarchical multi-policy scheduling: reservations are correctly enforced, latency overhead remains moderate, and the evaluated workloads meet all timing constraints in both mono-policy and multi-policy configurations. These results show that CSF-based reservation sizing can serve as a practical approximation even in multi-policy settings; however, its underlying mono-policy assumptions prevent it from formally guaranteeing schedulability when multiple scheduling policies coexist within a shared reservation, highlighting the need for dedicated theoretical models.

Several directions remain open. Broader evaluation across diverse workloads, shared-resource interactions, and additional hardware platforms is needed. Extending the analytical framework beyond mono-policy assumptions and further developing TGBS, particularly to support multicore operation, task migration, and robustness across kernel configurations, are key steps toward strengthening predictability in Linux-based flight-software systems.

Overall, the results indicate that hierarchical reservations and multi-policy scheduling can be jointly supported in Linux, while further refinement of both the analytical model and the kernel mechanism is required to strengthen predictability in flight-software contexts.

References

[1] Luca Abeni. Virtualized real-time workloads in containers and virtual machines. *J. Syst. Archit.*, 154(C), September 2024. ISSN 1383-7621. doi: 10.1016/j.sysarc.2024.103238.

[2] Luca Abeni and Giuseppe Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 4–13, 1998. doi: 10.1109/REAL.1998.739726.

[3] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the linux kernel. *SIGBED Rev.*, 16(3):33–38, November 2019. doi: 10.1145/3373400.3373405.

[4] Yuri Andriaccio. Design and implementation of real-time control groups in the linux kernel. Tesi di laurea magistrale, Università di Pisa, Dipartimento di Informatica, Pisa, Italy, February 2025. URL `https://etd.adm.unipi.it/t/etd-02122025-110046/`.

[5] Yuri Andriaccio, Luca Abeni, and Massimo Torquati. Scheduling iot applications in real-time control groups. In *2025 21st International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, pages 01–08, 2025. doi: 10.1109/DCOSS-IoT65416.2025.00070.

[6] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, May 2005. ISSN 1573-1383. doi: 10.1007/s11241-005-0507-9.

[7] Cédric Cazanove, Frédéric Boniol, and Jérôme Ermont. A linux container-based architecture for partitioning real-time task sets on arm multi-core processors. In *Proceedings of the JRWRTC 2023*, Toulouse, France, 2023.

[8] Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. RT-CASEs: Container-Based Virtualization for Temporally Separated Mixed-Criticality Task Sets. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:22, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-110-8. doi: 10.4230/LIPIcs.ECRTS.2019.5. URL `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2019.5`.

[9] Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Gener. Comput. Syst.*, 129(C):315–330, April 2022. ISSN 0167-739X. doi: 10.1016/j.future.2021.12.002.

[10] Jonathan Corbet. Space-grade linux. *LWN.net*, February 2025. Available at `https://lwn.net/Articles/1036168/`.

[11] Jens Eickhoff. *Onboard Computers, Onboard Software and Satellite Operations*. Springer Aerospace Technology. Springer Berlin, Heidelberg, Berlin, Heidelberg, 1 edition, 2012. ISBN 978-3-642-25169-6. doi: 10.1007/978-3-642-25170-2.

[12] Bryan Ford and Sai Susarla. Cpu inheritance scheduling. *SIGOPS Oper. Syst. Rev.*, 30(SI):91–105, October 1996. ISSN 0163-5980. doi: 10.1145/248155.238765.

[13] Julien Galizzi, Jean-Jacques Metge, Paul Arberet, Eric Morand, Fabien Vigeant, Alfons Crespo, Miguel Masmano, Javier Coronel Parada, Ismael Ripoll, Vicent Brocal, et al. Lvcugen (tsp-based solution) and first porting feedback. In *Embedded Real Time Software and Systems (ERTS2012)*, 2012.

[14] Jacqueline J Le Moigne-Stewart, Michael A Johnson, Alan P Cudmore, and Peter M Hughes. Container flight software (project 94827). Technology Project Report Project 94827, National Aeronautics and Space Administration, 2019. URL `https://techport.nasa.gov/projects/94827`. Accessed on 2025-11-17.

[15] Hannu Leppinen. Current use of linux in spacecraft flight software. *IEEE Aerospace and Electronic Systems Magazine*, 32(10):4–13, 2017. doi: 10.1109/MAES.2017.160182.

[16] Giuseppe Lipari and Sanjoy Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, pages 193–200, 2000. URL `https://api.semanticscholar.org/CorpusID:18822317`.

[17] Giuseppe Lipari and Sanjoy Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, pages 26–35, 2001. doi: 10.1109/RTTAS.2001.929863.

[18] Giuseppe Lipari and Enrico Bini. A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.*, 1(2):257–269, April 2005. ISSN 1740-4460.

[19] José Martins and Sandro Pinto. Shedding light on static partitioning hypervisors for arm-based mixed-criticality systems. In *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 40–53, San Antonio, TX, USA, 2023. IEEE. doi: 10.1109/RTAS58335.2023.00011.

[20] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The real-time linux kernel: A survey on preempt-rt. *ACM Comput. Surv.*, 52(1), February 2019. ISSN 0360-0300. doi: 10.1145/3297714. URL `https://doi.org/10.1145/3297714`.

[21] Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *25th IEEE International Real-Time Systems Symposium*, pages 57–67, 2004. doi: 10.1109/REAL.2004.15.

[22] Insik Shin and Insup Lee. Compositional real-time scheduling framework with periodic model. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347383.

[23] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *2008 Euromicro Conference on Real-Time Systems*, pages 181–190, 2008. doi: 10.1109/ECRTS.2008.28.

[24] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro Vittorio Papadopoulos. Real-time containers: A survey. In Anton Cervin and Yang Yang, editors, *2nd Workshop on Fog Computing and the IoT, Fog-IoT 2020, April 21, 2020, Sydney, Australia*, volume 80 of *OASIcs*, pages 7:1–7:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/OASICS.FOG-IOT.2020.7.

[25] Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. Hierarchical resource orchestration framework for real-time containers. *ACM Trans. Embed. Comput. Syst.*, 23(1), January 2024. ISSN 1539-9087. doi: 10.1145/3592856. URL `https://doi.org/10.1145/3592856`.

[26] Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. React: Enabling real-time container orchestration. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2021. doi: 10.1109/ETFA45728.2021.9613685.

[27] James Windsor and Kjeld Hjortnaes. Time and space partitioning in spacecraft avionics. In *2009 Third IEEE International Conference on Space Mission Challenges for Information Technology*, pages 13–20, 2009. doi: 10.1109/SMC-IT.2009.11.

[28] James Windsor, Marie-Hélène Deredempt, and Regis De-Ferluc. Integrated modular avionics for spacecraft — user requirements, architecture and role definition. In *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, pages 8A6–1–8A6–16, 2011. doi: 10.1109/DASC.2011.6096141.