WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Computer Science & Engineering

Dissertation Examination Committee:
Roger Chamberlain, Chair
Ron Cytron
Chris Gill
Tim York
Xuan Zhang

Multi-Layer Support for Component-Based Cyber-Physical Systems Applications

by
Oren Bell

A dissertation presented to
the McKelvey School of Engineering
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2025
St. Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor, Chris, for his guidance. I would also like to thank my friends in St Louis who supported me, both socially and academically, in this program. A very unique thank you to Lydia, she knows why.

Thank you to JJ, Mario, Harun, and the rest of the Dortmund group for their hospitality and support.

A special acknowledgement is given to Allison Thackston, a roboticist I've never met, whose ROS 2 docker templates saved me months on my dissertation.

And finally, thank you to Andrew Huey, my middleschool history teacher, who gifted me a copy of "The Way Things Work" by David MaCaulay when I was 14. That present sent me on the path to becoming an engineer.

This dissertation is based upon work supported by the US National Science Foundation under grant CNS-2229290.

This dissertation contains work that has received funding by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038.

This dissertation contains content written as part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170).

Oren Bell

Washington University in St. Louis
August 2025

ABSTRACT OF THE DISSERTATION

Multi-Layer Support for Component-Based Cyber-Physical Systems Applications

by

Oren Bell

Doctor of Philosophy in Computer Engineering

Washington University in St. Louis, 2025

Professor Roger Chamberlain, Chair

Component-based design is a paradigm meant to aid in development of software applications by modularizing different functionalities of a system. This building-block approach is used extensively in cyber–physical and robotic systems. Common and established solutions to specific problems, such as perception nodes, state estimators, and motion planners, can be developed, verified, and reused as off-the-shelf modules.

These modules may be integrated atop today's heterogeneous hardware platforms, where an application can be distributed across GPUs, FPGAs, and CPU cores. When heterogeneous computational devices share the workload of a collection of components, the very act of integration may inject timing uncertainty. For example, data often must traverse different memory domains, incurring copies that may violate real-time budgets. Furthermore, the Robot Operating System (ROS) 2, the de facto middleware for robotics, has a scheduler that departs from established real-time practice, and work has only recently begun on establishing timing guarantees for it.

This dissertation builds a vertical solution that restores predictability without sacrificing modularity. The work is organized into 3 layers: memory management of hardware-accelerated devices, a real-time scheduler for ROS 2, and a control/scheduling co-design study.

Collectively, these contributions demonstrate that the promises of component-based design—modularity, compositional verification, and rapid development—can survive in the age of hardware acceleration, while still meeting cyber-physical constraints on timing and other properties end-to-end. By coupling memory-domain awareness with analyzable scheduling and timing-aware control, this work charts a practical path toward predictable, high-performance CPS and robotic systems.

Chapter 1

Introduction

This dissertation presents a systematic approach to designing predictable component-based real-time systems for cyber-physical applications, with a particular focus on robotics. The work emerged from a preliminary robotics project involving an autonomous drone application, which revealed fundamental challenges in achieving predictable timing. This initial exploration evolved into a comprehensive study spanning three critical layers of the system stack: memory management, scheduling, and control.

The journey began with the development of the Deer Stalker project (see Section 1.2), an autonomous drone system that highlighted the performance bottlenecks inherent in the Robot Operating System (ROS) 2 framework that this application was built on (detailed in Section 1.1). This led to the development of Hazcat (presented in Chapter 2), a memory-domain-aware communication layer. After delving into development of the ROS 2 stack, I turned my attention to the ROS 2 scheduler, which recently had been the focus of academic research highlighting its design flaws. This led to collaboration with the DAES Lab at TU Dortmund on creating real-time compliant schedulers, which is documented in Chapter 3. Finally, through collaboration with Purdue University, this research arc has culminated in a control/scheduling co-design study that demonstrates how worst-case timing analysis can be incorporated into control logic decisions that apply to other cyber-physical systems domains beyond robotics, e.g., real-time hybrid simulation in structural and mechanical engineering.

These contributions each provide different methods for building predictable real-time systems in component-based architectures. Figure 1.1 provides an illustrated breakdown of each contribution and how it fits into the system stack. Summaries of the 3 broad contributions are detailed below.

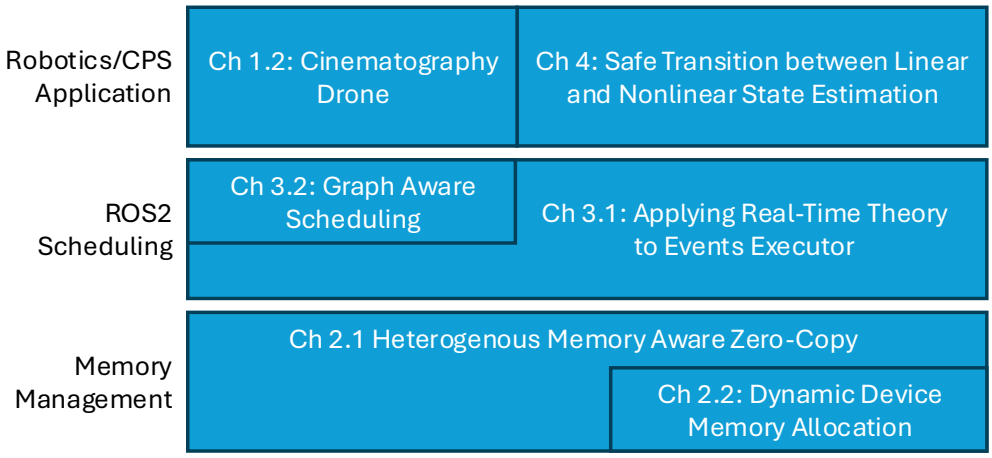


Figure 1.1: Dissertation layers

(1) Hazcat introduces a memory-domain–aware publish/subscribe layer[1]. By detecting when producers and consumers reside in the same memory domain, Hazcat achieves true zero-copy, even among accelerated kernels on different components. Otherwise, it performs the minimum safe transfer. Existing middleware supports zero-copy in the CPU domain, but to our knowledge Hazcat is the only zero-copy communication middleware that supports zero-copy between accelerated kernels **and also** in the CPU domain. Pre-existing middleware only supports static allocations, so I also pursued the possibility of supporting dynamic memory allocation. Our particular system model violated certain assumptions that rendered existing dynamic memory allocation algorithms ineffective, so I developed a new set of dynamic memory allocation algorithms for device memory just to support Hazcat in its unique niche. Both of these contributions are described in Chapter 2.

(2) A Real-Time ROS 2 Executor refactors the recently released *events executor*. My co-authors and I observed that the *events executor* allows for custom event queues with arbitrary behavior. We exploit this to create custom release logic. The new design is compatible with the classical periodic-task model and augments it with graph-aware, fixed-job-level priorities, addressing some of the design flaws [29] that previously plagued ROS 2. Response-time bounds are derived for arbitrary DAGs. This work is described in Chapter 3.

[1]Publish/subscribe (pub/sub) is a communication paradigm facilitated by *topics*, which act like data queues that can have multiple writers (*publishers*) and readers (*subscribers*). Each publisher and subscriber is affiliated with only one topic. Subscribers are notified when data arrives on their topic.

(3) A Control/Scheduling Co-Design Study considers a safety-critical state-estimation pipeline that normally runs a linear Kalman Filter [70] but may switch to an Extended (or Unscented) Kalman Filter [103,119] or potentially even another kind of state estimator (e.g., a particle filter) when dynamics depart from a region of linearity. The mode-change decision explicitly incorporates worst-case timing analysis, and guarantees an upper bound on the estimation error, illustrating how control logic and real-time scheduling can be co-engineered. I also have preliminary work on implementing this project on an FPGA, and comparing its performance to a GPU. I explore the limitations and motivations of hardware-accelerated CPS applications with mixed devices. This work is described in Chapter 4.

1.1 ROS 2

ROS 2 is a widely adopted software framework for developing robotics applications, which has garnered significant attention from the real-time systems research community. While ROS 2 provides mechanisms for modular development, its default scheduling capabilities present challenges for predictable real-time performance, particularly for complex applications involving branching or merging data flows (i.e., arbitrary graphs).

Additional discussion on ROS 2's scheduling behavior and contributions towards improving it can be found in Chapter 3. This preface is meant to serve as a brief introduction to the ROS 2 framework and terminology, which is used throughout the dissertation.

ROS 2 provides a component-based framework for building robotic systems. Key concepts include:

- **Nodes** encapsulate related functionalities (e.g., sensor driver, path planner).
- **Topics** establish named communication channels facilitating data exchange via a publish-subscribe (pub/sub) paradigm. Nodes that provide data will *publish* to a topic, and nodes that are interested in that data will *subscribe* to it. The middleware responsible for the facilitation of data transfer is typically the Data Distribution Service (DDS) [45,48], although other communication protocols may be used [49,127].
- **Callbacks** are functions within nodes that are triggered by specific events, primarily:

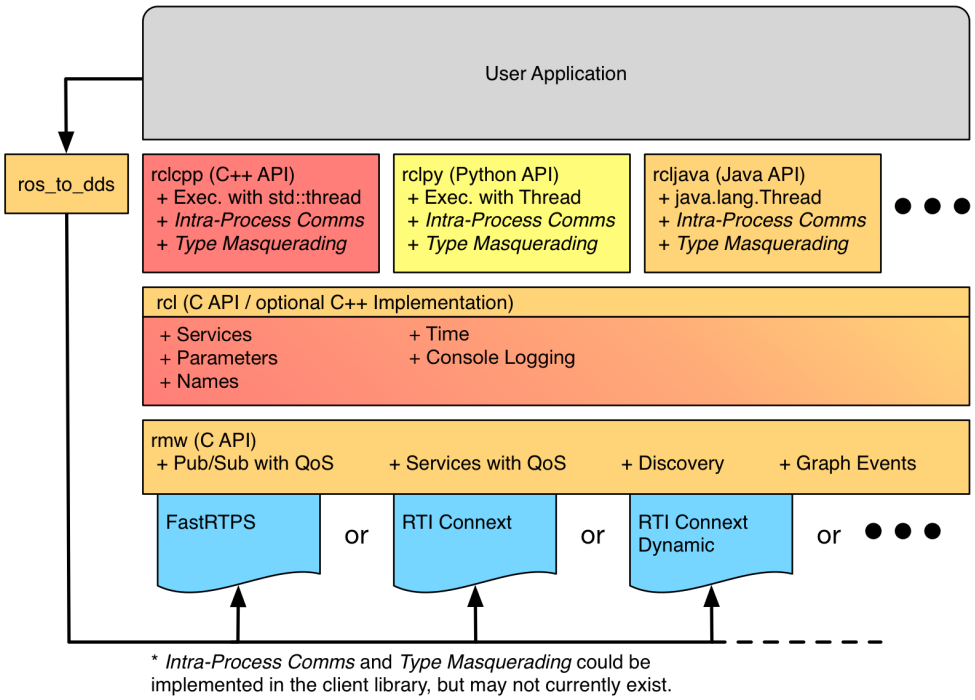


Figure 1.2: ROS 2 stack [61]

- **Timer Callbacks** execute periodically based on configured rates.
- **Subscription Callbacks:** Execute upon receiving a message on a subscribed topic.

Callbacks serve as the atomic executable units of computation in ROS 2. Through publishing and subscribing, they also can trigger each other's execution. The sequence of callbacks triggered by data flowing through topics forms the application's processing logic, often creating implicit dependencies or chains of execution. Such chains were initially defined in [29], and subsequent work [18, 20, 21, 68, 104, 110, 114] has continued using that abstraction.

ROS 2 is designed in a hierarchical stack, shown in Figure 1.2. The top layer (rclcpp, rclpy, etc) is used directly by the end-developer and provides a specific language binding to use ROS 2 with. This is where scheduling decisions are made. Without loss of generality, I will focus on rclcpp in this dissertation.

The basic functionality of ROS 2 is in the rcl layer, which provides features like nodes, namespaces, and logging.

The lowest layer of the ROS 2 stack, called the ROS Middleware (RMW) Abstraction Interface, serves as a wrapper for a communication platform, such as FastDDS [45] or CycloneDDS [48], which directly facilitates communication between nodes.

1.2 Preliminary Work

Before the dissertation topic was fully defined, preliminary work was done to develop an example robotics application. The application was a drone follower program, inspired by [23–25]. The application was developed in ROS 2, and was intended to be an example application for future research. Development eventually halted when performance limitations were encountered, which in turn inspired the work presented in Chapters 2 and 3. The application is detailed here, as important context and motivation for this dissertation.

The basic structure of the application was taken from [23] and is shown in Figure 1.3. It consists of a classic sensing/planning pipeline, with a parallel pipeline for mapping. It is meant to run in AirSim [89], a robotics simulator that runs on top of the Unreal Engine [44].

In the paper that inspired our implementation, the authors intended to use the application as a platform to develop a cinematographer drone that incorporated automated cinematic decisions into its trajectory planning. It was designed to follow a moving target, such as a real-life actor. In our implementation, we had it follow a simulated deer, simply because the CGI model was free and included idling, walking, and galloping animations.

The sensing pipeline comprises 3 steps:

- **Detection and Tracking** consisted of a YOLOv4 model that detected the actor in frame, and, importantly, how far off-center it was in the frame. I had to retrain the model on images of our CGI deer, a process that was easily automated thanks to the Unreal Engine environment. A cropped bounding box of the actor was passed onto the next step. Figure 1.4a shows a still frame from the object detection step.
- **Heading Estimation** was given a tightly cropped and centered picture of the actor, with which a vision AI model would determine which direction the actor was oriented in. This task required training an original base model, but again, the collection of data

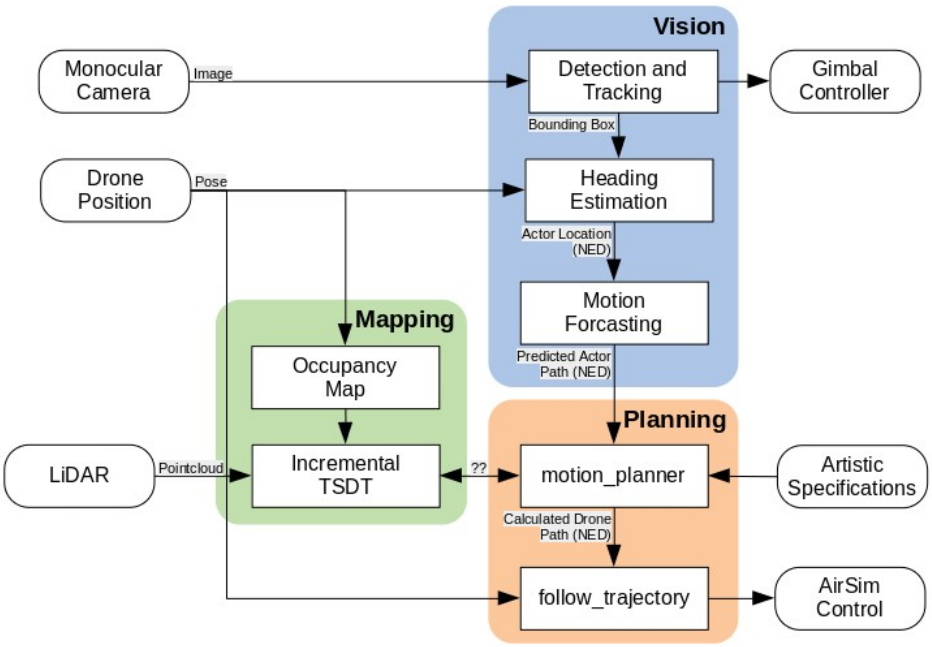


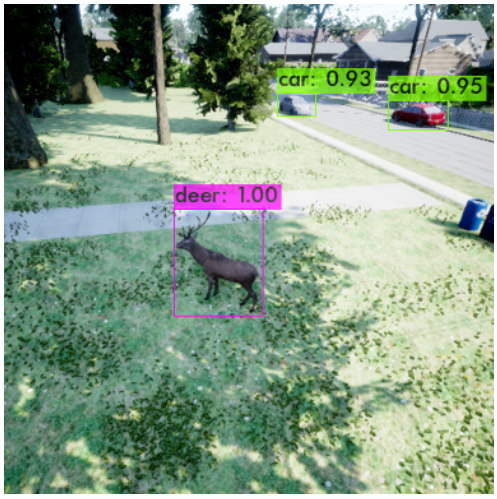
Figure 1.3: Cinematography Application [23]

was easily automated because of the Unreal Engine environment. Once the actor's position and heading were known, this was passed onto the final sensing step. Figure 1.4b shows one frame from our training set.

- **Motion Estimation** used a state estimator to estimate the actor's position and velocity. Assuming the drone's position is known with high certainty, the relative measurements of the actor's position and orientation from before are projected onto the ground plane, to compute the actor's position and orientation in the world frame. This is fed into a Kalman Filter to estimate the actor's position and velocity, and future trajectory. This estimation can be used to continue forecasting the actor's trajectory even when measurements aren't available, such as when the actor is occluded.

The mapping pipeline comprises 2 steps:

- Data from the Lidar and the drone's position were fused to create and expand a 3D **Occupancy Grid** of the environment.



(a) Object detection results, showing bounding boxes with labels and confidence scores

(b) Heading Estimation, model prediction in red, ground truth in green

Figure 1.4: Still frames from Deer Stalker Application

- The map data was converted into a Truncated Signed Distance Transform (**TSDT**) to create a signed distance field of the environment. The inside of obstacles are given large negative values, and the proximity just outside the obstacles are given positive values, truncated to a fixed value beyond a certain distance from the obstacle. This TSDT is passed into the planning pipeline to plan a trajectory for the drone that safely avoids obstacles.

The planning pipeline also comprises 2 steps:

- **Trajectory Optimization** used a path planner to plan a trajectory for the drone that safely avoids obstacles. The default path is a trajectory parallel to the actor, given a fixed distance and position for the drone (both are parameters that can be changed at runtime, if a different angle is required later in a scene). The optimization step takes this default path and adjusts it to avoid obstacles. This is done by maximizing a cost function that includes, among other components, the line integral of the trajectory computed through the TSDT from the mapping pipeline. Since the TSDT is positive in areas that avoid obstacles, optimizing this cost function will naturally push the trajectory into free space. In [24], they augment this cost function to also avoid occlusion of the actor, smoothness of the trajectory, and consistency of perspective on the actor.

- **Trajectory Tracking** is the actuation step, where the drone's position is compared to the planned trajectory, and closed loop control is used to adjust the drone's position to follow the trajectory.

This application is a compelling candidate to test future research, because it contains a diverse set of hardware-accelerable components and poses interesting challenges in system design. Before the application was fully working, I discovered one such challenge: the map data was very large, and incurred a high memory overhead when being copied from one component to another. I started investigating the possibility of using a zero-copy mechanism, which inspired the work in Chapter 2.

While developing that mechanism, I also became familiar with the ROS 2 scheduler. At the time, a paper had just been published documenting its faults [29], and I began to explore that avenue as well. Since the application consists of two chains with a merging data flow, there was naturally a need to schedule complex graphs reliably on ROS 2, which inspired the work in Chapter 3.

By the time both of those works were completed, this *deer stalker* application, as we affectionately called it, had been defunct for multiple years. The ROS 2 distro and AirSim versions it required were no longer supported, and so we sought to find a new application to which to apply the techniques developed in this dissertation. A collaboration with colleagues affiliated with the Robert L. and Terry L. Bowen Laboratory for Large-Scale Civil Engineering Research at Purdue University called for work on integrated real-time scheduling and control that could be applied to structural and mechanical engineering applications. This in turn inspired the work presented in Chapter 4.

Chapter 2

Zero-Copy Memory Management for Heterogeneous Computing

Real-time embedded systems support mission-critical and safety-critical applications ranging from automated and connected driving [123] to real-time hybrid simulation in earthquake engineering experiments [46]. In recent years, the combination of component-based modularity and customizable thread and memory management in ROS 2 has made it especially attractive for developing real-time embedded robotics applications [29], including the TurtleBot 4 open-source robotics platform for education and research [34], the Hamster robust micro Autonomous Unmanned Ground Vehicle [35], and the xArm series of robotic manipulators [116].

Such systems often have timing constraints. Developers must carefully manage hardware and software overheads, timing variation, and scheduling so that real-time tasks meet their deadlines. Even atop traditional multi-core platforms, memory operations may introduce significant overheads, e.g., when the number of cores used exceeds a single chip socket so that the cost of memory operations between threads on cores in different chip sockets limits how many cores can be exploited at fine-grained time-scales [46], or when frequent communication among components causes the aggregate cost of memory operations to approach other less frequent but more expensive overheads (e.g., thread context switches). Thus, reducing the frequency of memory operations or making them more efficient [58] is an important issue for these systems.

For heterogeneous computing platforms that support hardware acceleration of computations (e.g., NVIDIA Jetson and Xilinx Kria) latency-aware memory management is even more salient. For example, it has been shown [4,97], that GPUs can be a cause of non-deterministic behaviour in real-time systems. As one particularly egregious example, memory operations on NVIDIA GPUs, particularly *cudaFree*, may cause device-wide synchronization operations

[125]. With or without hardware acceleration, a multi-component application can reduce its memory bandwidth by leveraging shared memory structures and passing references to data instead of copying the data itself [73], whenever possible. Such "zero-copy" approaches are intuitive in principle, but must enforce ownership of data to avoid race conditions, which may raise further engineering challenges in practice.

Zero Copy in ROS 2: Prior developments in ROS 2 have added API calls to the RMW specification that enable zero-copy through the use of borrow/return semantics. Before a publisher wishes to send data, it must *borrow* a message which it can then populate before publishing. Subscribers receive an immutable pointer to the same data, and when all subscribers have returned their pointers, the memory for the message is freed or recycled, depending on the implementation.

A popular implementation of zero-copy semantics for ROS 2 is in the Iceoryx project [49], which provides a third-party daemon that manages a shared memory pool. Publishers and subscribers wishing to make use of zero copy are required to use borrow and return semantics to access memory chunks [74] as follows:

1. Register the publisher and subscriber with the Iceoryx daemon, specifying their topic of interest
2. Borrow a chunk: the daemon returns the address of the next available block large enough to fit the message
3. The publisher populates the chunk with data
4. The publisher publishes the chunk, relinquishing ownership
5. Iceoryx notifies all interested subscribers of available data after which (repeatedly)
6. A subscriber awakes and borrows the chunk, receiving a pointer to newly available data
7. That subscriber processes the data
8. That subscriber returns their reference to the chunk
9. The Iceoryx daemon marks a chunk as available for reuse once all subscribers have processed it

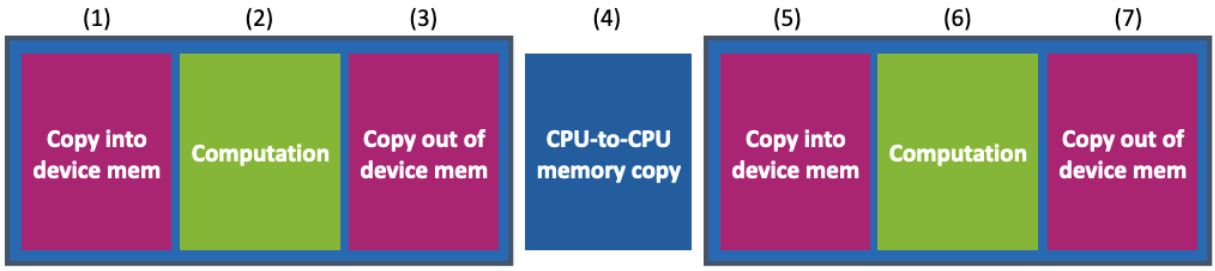


Figure 2.1: Time sequence illustrating communication between two hardware accelerated components

Multi-core performance numbers for Iceoryx show minimal overhead [73], but Iceoryx does not support zero-copy semantics for other devices' memory: a developer wishing to leverage hardware acceleration must manually copy messages into and out of device memory. Additionally, the memory pools are pre-allocated by a third-party daemon, which the end-developer must configure to optimize memory usage. Under the default settings, memory chunks are likely to be oversized, and may not be plentiful enough to accommodate the message backlog of a worst-case scenario.

Iceoryx also only supports zero-copy of fixed-size plain-old-data types. Any messages that require dynamic allocation of memory may be partially allocated on the heap, potentially resulting in invalid pointers and segmentation faults when sharing messages between processes. This is a limitation our implementation also shares, a fix for which would likely require updates to the ROS 2 API as future work. Another zero-copy mechanism is available higher in the ROS 2 software stack [62], but it only supports intra-process zero-copy semantics. In the interest of generality, we do not assume two components share an address space so that solution is not viable for our use case.

Hardware Accelerated Workloads: Although the addition of zero-copy to ROS 2 can help to reduce nondeterminism caused by excessive memory operations, we posit that it does not go far enough. Many robotics applications leverage hardware acceleration for performance reasons, but ROS 2 zero-copy features are as yet unaware of device-memory boundaries, so components leveraging hardware acceleration must assume responsibility for copying data into, within, and out of device memory. Doing so may negate the benefits of zero copy and introduce further nondeterminism.

For components sending data to each other, we want to minimize the latency that is due to 3 types of operations, depicted in Figure 2.1: intra-component memory copies (magenta), compute operations (green), and inter-component memory copies (blue). With hardware acceleration, whether using GPUs or FPGAs, there are 3 steps: copy to device memory, run the acceleration kernel, and copy out of device memory [30, 124]. These 3 operations happen within a component. We aim to remove intra-component memory copies entirely by having middleware assume responsibility for device memory operations. Inter-component memory copies also may be eliminated when conditions are right for zero-copy to occur, that is, when two consecutive nodes operate within the same *memory domain*.

Contributions: The Chapter covers content from two papers [13, 14]. The first paper, published at RTCSA 2023, introduces Hazcat [95]: a new zero-copy framework that supports memory domain awareness, performing contextually needed memory operations between different memory domains, and providing zero copy between two components in the same memory domain. This is covered in Section 2.1.

The second paper had an abbreviated version published at JRWRTC, the Junior Workshop for RTNS 2024. This dissertation covers the full contents of the original paper. It explores the limitations of applying dynamic memory allocation to the Hazcat framework, the challenges it poses, and new dynamic memory allocation algorithms to address them. This is covered in Section 2.2.

2.1 Hazcat

We introduce Hazcat [13], a new zero-copy communication middleware that supports memory domain awareness, performing contextually needed memory operations between different memory domains, and providing zero copy between two components in the same memory domain.

We use the term *memory domain* to refer to any area of memory that is tied to a particular set of processors, and is guaranteed to be readable from processors in that set: host memory is a separate domain from GPU memory, both are separate domains from FPGA memory, and data in one domain must be copied to be accessible in another domain.

We specify special *heterogeneity aware* (HA) allocators that manage memory for a particular memory domain, according to a domain-agnostic interface. They have functionality to access each other's memory, and perform copy operations as needed when memory contents are not in an appropriate domain. We also specify a shared message queue, analogous to a ROS 2 topic, which stores references to messages allocated by an HA allocator. Publishers put messages into the queue for later consumption by subscribers. When a subscriber expresses a preference to receive a message in a different memory domain, a copy is made on the fly in the correct memory domain and a token to the duplicate copy is also stored in the message queue. This is the worst-case scenario: when data moves across memory domains, a copy operation must occur. Each topic gets a unique message queue. Unlike Iceoryx, our system handles zero-copy in a completely decentralized manner, and developers are allowed to apply custom allocation strategies instead of being forced to use Iceoryx's static ring buffer.

Hazcat is not truly zero-copy in all circumstances. If an application is only partially hardware accelerated, memory copies will be required between portions operating in host memory and portions operating in device memory. However, if consecutive nodes operate in the same memory domain, we offer the benefits of zero-copy regardless of what that memory domain is, be it host, GPU, or (in the future) FPGA memory.

2.1.1 Heterogeneity Aware Allocator

Custom memory allocators are already used in real-time systems, e.g. to allocate memory statically, or to reuse previously allocated objects to avoid overheads of an operating system's default dynamic memory management mechanisms. Sophisticated features such as allocator-aware containers and polymorphic allocators [66] have been developed to allow flexible use of allocators, through which containers allocated with different strategies can be completely interoperable without subverting type-safety. A minimal allocator offers 2 functions: *allocate* and *deallocate*. Some allocators also provide object construction, but we omit this in our design: we assume that our allocators, while potentially managing peripheral device memory, will run on a CPU and should not attempt to dereference the device memory they allocate.

Managing multiple memory domains within our framework incurs stricter design requirements beyond supporting allocate and deallocate functions. We first detail a new extended interface, describe garbage collection features required whenever sharing memory, present

the structure of the allocator and its memory pool, and show a way to ensure fidelity when reconstructing the allocator in other processes.

Allocator Interface

We propose a new allocator interface, shown in Listing 2.1, that adds functionality to convert arbitrary memory allocated by a separate allocator into a domain readable to the current allocator. This conversion may operate as a simple passthrough function if the two allocators operate in the same memory domain (this is our zero-copy condition), or it may perform the operations necessary to copy data from a separate domain.

Listing 2.1: Allocator interface for heterogeneous computing

```
class Alloc {
  static int domain;
  static void * create_alloc(...);

  static void * remap(Alloc * alloc);
  void unmap();

  int allocate(int size);
  void share(int offset);
  void deallocate(int offset);

  void * copy_from(const void* ptr, void* cpu_ptr, int size);
  void * copy_to(void* ptr, const void* cpu_ptr, int size);
  void * copy(const Alloc* a, void* dst, const void* src, int size);

  int shmem_id;
  int device_type;
  int device_number;
  int strategy;
};
```

Multiple allocators can be written for the same domain. We assume that any allocation can be accessed by any component in the same domain, provided any requisite memory mapping operations have been performed. The added functionality is six-fold: *remap* maps an allocator previously created in a different process into this process address space; *unmap* removes the allocator from this process address space; *share* increases a reference counter for a specific allocation; *copy_from* copies from this allocator to a pointer in CPU memory; *copy_to* copies to this allocator from CPU memory; and *copy* copies from an arbitrary allocator into this one, possibly going through host memory as an intermediary. Our allocators return

an integer offset rather than a pointer: allocators are mapped at different places in each process's address space, so allocations are measured relative to the beginning of the allocator and absolute pointers are recalculated in each process.

Deallocation and Reference Counting

Deallocation of shared memory may need to be performed by any thread: while a message is created by a publisher, whichever one of its subscribers that holds the reference last is responsible for deallocation.

For time-sensitive threads, deallocation must be a time-bounded operation, which limits the types of allocation strategies we can use. For now, we only provide a static ring buffer and intend to provide additional options in the future, but stipulate that all allocation strategies must offer O(1) deallocation.

We assume that every memory domain is copyable into host memory and vice-versa. However, not every memory domain may be copyable into each other, so each allocator must specify *copy_from*, to copy memory from itself into host memory, and *copy_to*, to copy memory to itself from host memory. The *copy* function is often a wrapper for those two steps in sequence: copying from the source domain to host memory, and then from host memory to the target domain. However, depending on available hardware, developers may code special conditions that bypass host memory, e.g., for GPU-GPU copies.

An HA allocator must implement a reference counting strategy, so the *deallocate* function will not free an allocation until it is called more times than there have been calls to *share* the same allocation. Note this creates a race condition that is resolved by the message queue as discussed in Section 2.1.3. When implementing new allocators for device memory, the allocator methods cannot access the memory they are allocating, so strategies like boundary tags [76] are not applicable.

Allocators provide additional information: *device_type* identifies the hardware device the allocator is managing memory for; for multiple instances of that *device_type*, *device_number* can distinguish them; and *strategy* identifies the allocator approach, such as ring buffers, TLSF [88], or best-fit [76]. Two allocators' *device_type* and *device_number* must be the same to take advantage of zero copy, with *device_type* and *strategy* used to perform function

lookups when an allocator is mapped into a new process, based on its ID according to *System V shared memory* (a POSIX standard that creates unique system-wide IDs for different shared memory segments).

Allocator Structure

Each allocator comprises 3 contiguous memory mappings: a local portion, a shared portion visible across processes, and a pool of mapped device memory. The local portion stores function pointers as a way to emulate the convenience of object-oriented polymorphism, which we use so that our message queue does not need to concern itself with type information. True polymorphism is not possible for objects in shared memory, due to uncertainty of the structure of some data types, which combined with inherent type-erasure that occurs during inter-process communication would prevent us from knowing the structure of an allocator created in another process. The goal is for different allocator implementations to have an identical structure in their first few bytes, so we use plain-old-data structures for our allocator design to guarantee this.

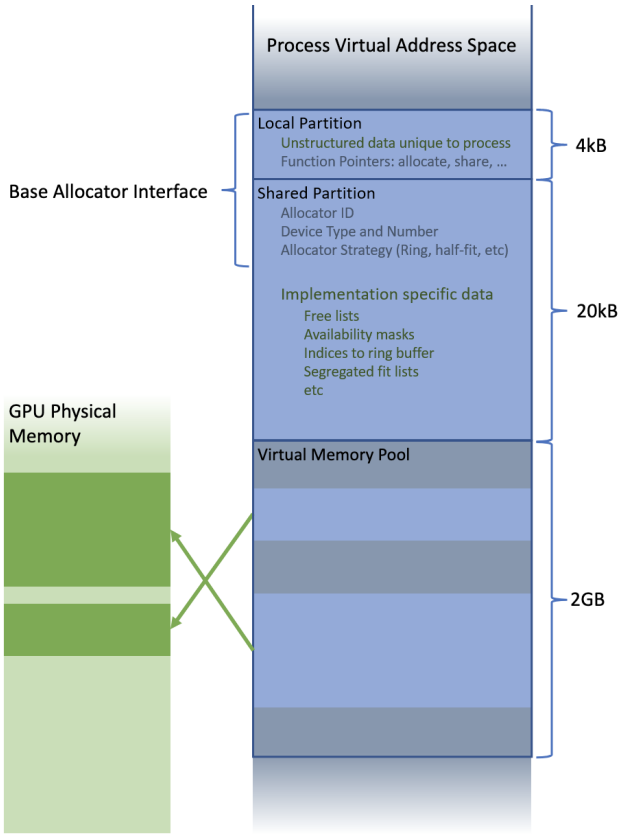


Figure 2.2: Structure of allocator and partitions (not to scale)

Since function pointers are not valid across process boundaries, the allocator portion that holds them is not a shared mapping. This leads to a design in which our allocator object straddles different memory mappings, the bulk of which is visible across processes, with the local partition only visible to the current process. Due to granularity requirements, a minimum of 4kB is allotted to the local partition, but only 56 bytes are needed for the 7 non-static function pointers. The remainder

is unstructured space that may be used as appropriate for the convenience of the allocator implementation. The start of the allocator's shared portion contains required type information that is relevant when remapping it into a new process, using the new allocator interface described above in Listing 2.1. The rest of the shared mapping varies by implementation. After the shared mapping is the actual memory pool, often mapped in device memory, which again varies by implementation. For dynamic allocators, it's useful to start with an arbitrarily large virtual memory pool unbacked by physical memory. This gives the allocator room to grow, while still guaranteeing it can be reconstructed in a different process.

Reconstructability of Allocators

Reconstructibility is essential: since allocators are not guaranteed to start at the same virtual memory address in different processes, memory allocations are expressed in terms of offsets from the start of their allocator and the relative structure of the allocator and its memory pool must be identical between processes. This is complicated by alignment restrictions on shared memory for different devices and systems. The minimum size of the local partition is a page (typically 4kB). The granularity of shared memory varies (4kB on x64 systems, but 16kB on ARM), as does the required granularity of device memory.

The allocator is only reconstructible in a location that satisfies all these granularity requirements. The shared mapping must end at an address that's a multiple of the shared granularity. Similarly, the device mapping must start at an address that's a multiple of the device granularity. Since the shared mapping and the device mapping are contiguous, then the boundary between them must be at an address that is a multiple of the least-common-multiple of both their granularities:

$$lcm(m, n) \tag{2.1}$$

We make an initial reservation of unmapped virtual memory at approximate location $\hat{x}$ of size

$$a + b + c + lcm(m, n) \tag{2.2}$$

That is, the collective sizes of the local, shared, and device mappings, plus a buffer zone to align it correctly. A valid position satisfying all granularity constraints is guaranteed to exist in any memory range of this size. If the boundary of shared and device memory partitions must be at a multiple of $\text{lcm}(m, n)$, then some modular arithmetic reveals that our reservation x must start at

$$\hat{x} + \text{lcm}(m, n) - ((x + a + b) \mod \text{lcm}(m, n)) \tag{2.3}$$

Any virtual memory before this point or past the tail can be released. Virtual memory within this range will be remapped to local, shared, and device memory in contiguous blocks guaranteed to begin at an address that satisfies their granularity requirements.

Metadata			Mem Domain 1			Mem Domain 2		
Sub Count	Avail Bits	Locks	Alloc ID	Offset	Size	Alloc ID	Offset	Size
0	0	0	~	~	~	~	~	~
1	b11	b10	0x0000 0014	0x0000 D0F8	0x0001 4100	0x0000 001A	0x0000 04D0	0x0001 4100
3	b10	b00	0x0000 0015	0x0021 0A00	0x0000 0B00	~	~	~
3	b10	b11	0x0000 0014	0x0001 21F8	0x0001 4100	~	~	~
...			...			...		

Table 2.1: Example Message Queue. Each column is 32 bits. Each row represents a message.

2.1.2 Message Queue

In a publish/subscribe application, topics can be modelled as queues of messages. This is because execution of subscription processing may be delayed and a backlog of work may accumulate. In soft real-time systems, a queue can have a maximum length and stale messages may be dropped.

In our framework, these message queues exist as named shared memory files, with the name taken from the topic name. In ROS 2, topics are named with strings. So a ROS 2 topic named */perception/rear_camera* would be saved as a shared file with the name:

/dev/shm/ros2_hazcat.perception.rear_camera

When using Hazcat outside of ROS 2, these message queues can be named arbitrarily. The structure and example contents of a message queue are shown in Table 2.1. We describe the formatting of this design, and the methods for interacting with it, including registration and the *publish* / *take* calls.

Design of a Message Queue

Each message is stored in the queue in three parts: an allocator ID, the memory offset, and the message length. The allocator ID is used to look up an allocator, which may or may not be mapped already into the current process. These mapping operations typically occur once near the beginning of a program's run. After an allocator is mapped into the process's address space, the memory offset is added to the allocator's starting address to get the absolute address of the message. Each message may have multiple copies, one for each memory domain. The entire message queue is then structured as a collection of arrays: one per memory domain, plus an extra array for metadata.

The message length is a latent design decision for now. As mentioned in Section 2.1.1, we only support fixed-size plain-old-data datatypes for messages, so specifying message length is redundant, as it could be inferred from the topic. In the future, however, we plan to implement support for zero-copy of dynamically sized messages.

Each entry in the metadata array contains 96 bits, as follows. 32 bits serve as a subscriber counter to indicate if the message is still in use. 32 more bits serve as an availability map for up to the 32 supported memory domains, where a 0 indicates the message hasn't been copied to this domain, and a 1 indicates it is ready for zero-copy reading. The last 32 bits serve as locks for each memory domain to prevent redundant copy operations from colliding.

This allows up to 4 billion publishers and subscribers per topic, and 32 supported memory domains. The zero-th memory domain will always be host memory, but the rest are assigned

in order of registration and have no relation to the *device_type* or *device_number* mentioned above in Section 2.1.1.

In addition to the message queue itself, there is header data (not illustrated) to track the size of the message queue, the number of registered memory domains, and a global iterator pointing to the most recent message index. Subscribers also store their own iterator to the most recent message they haven't yet read. The sub counter tracks how many registered subscribers haven't read the affiliated message yet. Whenever a subscriber takes a message (covered in Section 2.1.2 on the *take* command), the sub count is decremented.

Publisher and Subscriber Registration

Before interacting with the message queue, a publisher or subscriber must register with Hazcat. A publisher or subscriber can only be affiliated with a single topic which they request during registration. The associated message queue will be mapped into the current process (or created from scratch). Pursuant to the subscriber's requested backlog, the message queue may be resized. Other processes will be informed of this when they attempt to fetch data and note the message queue's self-reported size does not match the size of their own mapping, at which point they remap the message queue with its larger capacity.[2]. We assume that all publishers and subscribers are registered during an application's initialization phase, so these resizing operations will not be a part of steady state operation.

When the application terminates, publishers and subscribers are also required to unregister, which is largely just decrementing an entity count. The last process to unregister the last of the publishers and subscribers for a message queue will also destroy the message queue.

When resizing message queues for topics, we consider the design philosophy behind ROS 2 when developing Quality of Service (QoS) policies [60]. The general principle is that publishers offer a quality of service, and subscribers request a quality of service. Since we are not concerned with packet loss over a network, the ability of publishers to retain a message history for resending is unnecessary. Thus, only the requests of the subscribers are relevant and the message queue should be the largest depth of all interested subscribers.

[2]In practice, this resizing would seldom occur, due to the shared memory object being page aligned. A 4kB page is enough to hold tokens for well over 60 messages across 4 different memory domains. Few applications would require more than this.

Since shared memory objects must be page aligned, the minimum footprint is still large enough to accommodate over 60 backlogged messages across 4 memory domains. 4 arrays of 96 bit entries, plus another metadata array of 96 bit entries, makes 60 bytes per entry. A 4kB page then equates to 68 entries.

Publish command

The *publish* command is called by a publisher with a message to publish. First, it atomically fetches and increments the index of the next available row. It then secures a lock on the entire row, which ensures that any subsequent attempt to modify that row will block until the current call finishes.

If there are any remaining messages in that row (due to wrap around of the message queue) they are deallocated. This dropping of messages will only occur if there are best-effort components in the system, which are assumed to be tardy and unaffected by the missed messages. Any hard real-time tasks have to consider this deallocation as a source of interference when computing their response time. In a well-designed application with only hard real-time components, the message queue would never overflow, as an accumulated backlog exceeding the quality-of-service setting is by definition a system failure. The alloc id, offset, and size fields illustrated in Table 2.1 are updated accordingly and then the write lock is removed. Finally, a signal is sent on the associated FIFO to inform other processes that a message is available.

Take command

The *take* command is called by a subscriber after being notified that a message was available (a convenience, not a required step). First, the row for the oldest unread message is found. If the subscriber is up-to-date and no new messages are available, the call returns NULL. After inspecting the availability bitmask for the row, if the message is available in the subscriber's preferred domain, it will *share* the message and then prepare to return it. If the message isn't available in the message's preferred domain, an available copy is identified and the entry for that is fetched. The subscriber's allocator then allocates a new message and performs a copy operation as shown below in Listing 2.2.

This new message copy is also stored in the row with the original, the availability mask is updated accordingly, and the allocator ID and message offset are prepared to be returned. Whether or not a copy occured, one last check is made. If this subscriber is the last to access this message, the message queue will release its references to all copies of this message across all domains, potentially deallocating some if no other subscribers hold a reference. Messages with a non-zero reference count remain available to access for any subscriber that is already running. When they finish, they also decrement their reference count and the last to finish deallocates the copy in their particular domain. This does mean that subscribers frequently use messages that are no longer tracked by the message queue. We do this because tracking message ownership in the message queue requires locking a message reference as read-only until all subscriptions return. These locks would provide an opportunity for best-effort subscriptions to indefinitely block a real-time publisher trying to submit a message, which is unacceptable, so we require our HA allocators to implement reference counting themselves.

```
alloc = lookup(sub.alloc_id)
src_alloc = lookup(entry.alloc_id)
msg = src_alloc + entry.offset
len = entry.len

here = alloc.allocate(len)

if (CPU == src_alloc.domain) {
alloc.copy_to(here, msg, len)
} else if (CPU == alloc.domain) {
src_alloc.copy_from(msg, here, len)
} else {
alloc.copy(here, src_alloc, msg, len)
}
```

Listing 2.2: Copy a message into the preferred domain when zero-copy conditions are not met

2.1.3 Message Lifecycle

As a recap to our system design, we cover the steps to create, use, and dispose of a message. We conclude with comments on thread safety and steps to prevent race conditions for shared memory. The steps to interact with Hazcat are 4-part: allocate, publish, take, and deallocate.

The *allocate* method is called on an allocator to create memory for the message, as detailed in Section 2.1.1. During a component's computation, the message is populated.

After the first component is finished, it calls *publish*, which places the allocator ID and message offset in the relevant message queue, as described in 2.1.2.

When a second subscribed component has been informed of a message, it calls *take*. This modifies the message queue as described in Section 2.1.2 and also affects the messages directly: it calls *share* on the message copy in the component's memory domain, which increments the reference counter stored in the allocator. If this subscriber is the last to read the message, it will also call *deallocate* on all the message copies, which decrements the reference counter. This is akin to clearing the message queue's references to the message copies.

The message won't be garbage collected until this subscribed component calls *deallocate* one more time to clear it's own reference to the message, which is the final step.

In Section 2.1.1, we mention that the *share* and *deallocate* commands create a natural race condition. This is resolved by the fact that *share* is only ever called from within the *take* command, while the message is owned by the message queue. Calling *deallocate* from a separate thread concurrently with the *take* command is always guaranteed to do nothing, as the message queue itself retains a reference to the message. The only circumstance where calling *deallocate* will truly deallocate a message is when the message is no longer tracked by the message queue, so there's no possibility of *share* being called on it simultaneously.

2.1.4 Design and Implementation

Any component-based framework using a CORBA or pub/sub communication model can be modified to use Hazcat as its communication layer. Calls can be made to Hazcat to register publishers and subscribers, create messages, publish to a topic, take from a topic,

and finally deallocate messages. All that is needed is a translation layer to incorporate this into a desired framework.

For sake of demonstration, we created an RMW to interface with ROS 2 systems and provide these zero-copy benefits to ROS 2 applications. We also provided 2 allocators, one each for CPU and GPU, both implementing a static ring buffer. Additional allocators can be added readily in the future, such as one for the real-time dynamic memory strategy TLSF [88].

These allocators can be specified by the end developer when instantiating publishers or subscribers. These are done through the use of subscription_options and publisher_options, which each contain a field called rmw_implementation_payload. This mechanism can be used to pass options specific to an RMW implementation, in this case a Hazcat allocator. If this field is ignored, then a default CPU-based allocator is assumed.

When extending the Hazcat platform to new products and hardware, we make certain stipulations for features the drivers must provide: an IPC mechanism to share device memory allocations with unrelated processes; support for unified-virtual-addressing, including virtual memory reservations; and mapping of physical memory to an explicit virtual address, subject to page granularity constraints.

First, the hardware driver must implement some interprocess communication (IPC) mechanism for its device memory. Any device memory allocated must be able to be accessed by any unrelated process using a globally unique token. Depending on how this mechanism is implemented, or whether it's implemented at all, may naturally limit a product's performance and its viability to be used with Hazcat. As an illustrative example, CUDA now provides shareable handles for IPC communication with their new driver API calls. As of CUDA 10.2, a developer can use *cuMemExportToShareableHandle* and *cuMemImportFromShareableHandle* to create a handle allegedly shareable between processes. However, these calls only work for related processes. The shareable handles are process-specific file descriptors, and any attempt to translate these descriptors to another process, via the *procfs* and the new *pidfd_getfd* syscall, will inevitably fail. Why this happens is not explained in the current CUDA documentation. The handle is intended to be used before a *fork()* operation, and is not conducive to IPC between arbitrary processes.

Additionally, the drivers for the hardware device in question must support unified-virtual-addressing. Remapping allocators requires concatenating shared memory with device memory in a way that they appear as a single object. While their absolute location may vary, their relative positioning must be intact between processes. Thus, once a shared memory mapping is created for the allocator, its device memory pool must be placed in a particular location in a process's address space. If this ability is not present, we can't make guarantees about the positional relationship between an allocator and its memory pool. These guarantees are essential to correctly locate messages using nothing more than an offset.

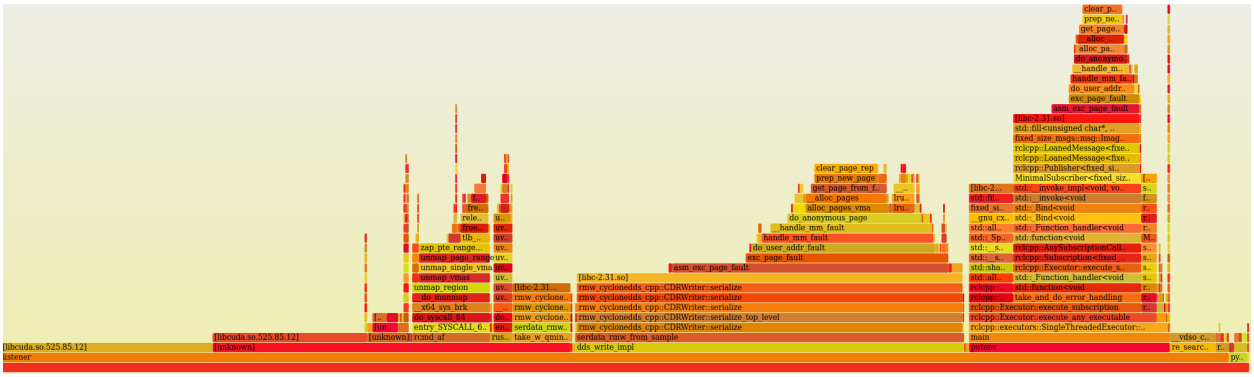
Lastly, to avoid race conditions when claiming address space, the device needs a call to reserve virtual memory. The allocator and device memory then can be mapped in without another thread claiming adjacent virtual memory.

CUDA's driver APIs support all the necessary virtual memory features mentioned above, but lack an adequete IPC mechanism. The traditional CUDA API does not support address reservations and explicit memory mapping, but does have IPC functionality that meets our constraints. However, the two APIs provided are not compatible. We can conclude that based on the aggregate functionality present in both APIs, NVIDIA hardware has the capability to work with Hazcat, but as it stands, Hazcat cannot support CUDA on it for independent-process workloads. The experiments below in Section 2.1.5 therefore use components in a single process or in related processes.

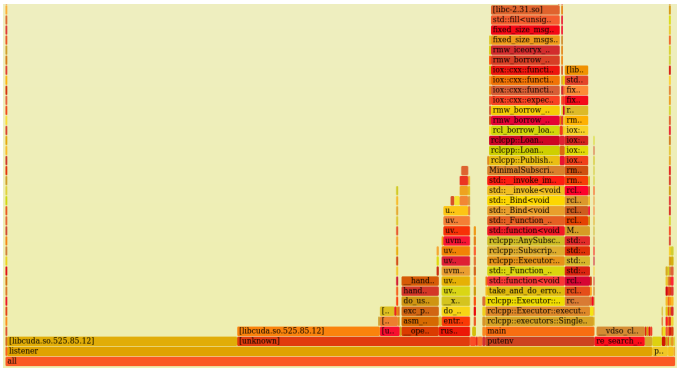
2.1.5 Empirical Evaluation

Two Component Experiments

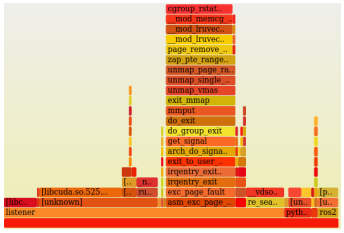
As a demonstrative example, we evaluate a sample application with two nodes. Each performs a simple bilateral filter on a random 4k image, to serve as an arbitrary parallelizable computation. These experiments were performed on a desktop system with an AMD Ryzen7 3700X CPU and an NVIDIA RTX 2070 Super. We measure the end-to-end latency and perform stacktrace sampling on this application while it runs on three different middlewares: CycloneDDS, Iceoryx, and our Hazcat.



(a) CycloneDDS Flame Graph
(end-to-end latency: 50.8ms)



(b) Iceoryx Flame Graph
(end-to-end latency: 27.2ms)



(c) Hazcat Flame Graph
(end-to-end latency:
13.7ms)

Figure 2.3: Stacktrace Sampling

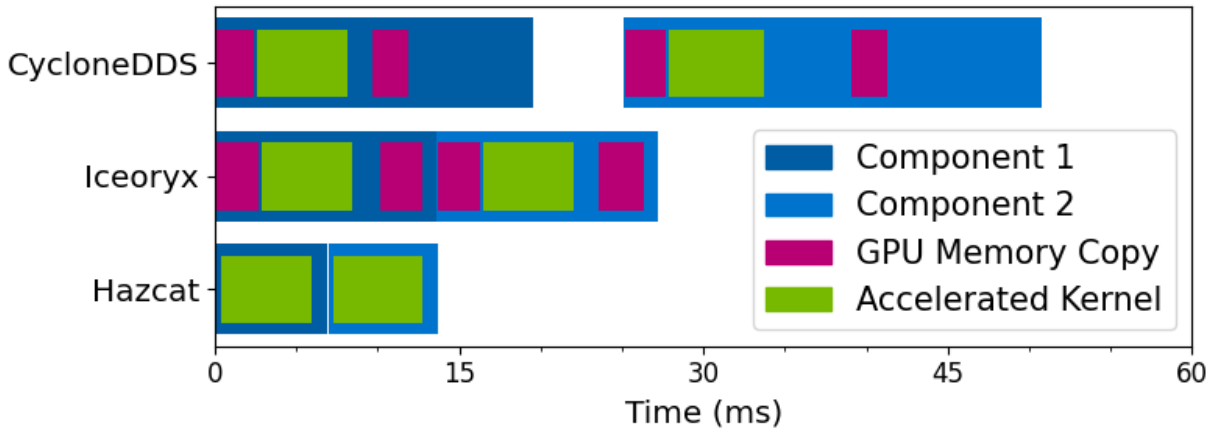


Figure 2.4: Two Component Experiment with Hardware Acceleration

Due to the limitations of CUDA IPC mentioned in Section 2.1.4, these nodes were run in the separate threads in the same process, instead of in different processes, though we expect that the performance differences with that other case would be negligible. We measure the steady state operation of the system, so the first 1 or 2 samples were dismissed as outliers if their runtime latencies differed significantly from the subsfequent samples. Such outliers were seen in the Iceoryx uniprocessor example, as well as in all of the GPU examples (presumably due to kernel loading).

Figure 2.1 illustrates the different operations performed by this two-component sample application. Each operation type is detailed in Section 2, and mean timing information for each variation is illustrated in Figure 2.4. Variation in the end-to-end latency for all three middlewares is displayed in the violin graphs in Figure 2.5. These figures were created by manually placing userspace probes in the application to measure timing at the entry and exit points for GPU memory operations, GPU kernels, and the callback functions for each component.

Stacktrace sampling can be found in Figures 2.3a, 2.3b, and 2.3c. The width of these flame graphs has been normalized according to their respective end-to-end latencies, so the time spent in each function can be better visualized.

Our tracing tools are unfortunately incapable of extracting stack traces from within CUDA code, so any uses, be it from kernel execution or memory operations, are combined into "libcuda". We observe that CycloneDDS and Iceoryx spend roughly the same amount of

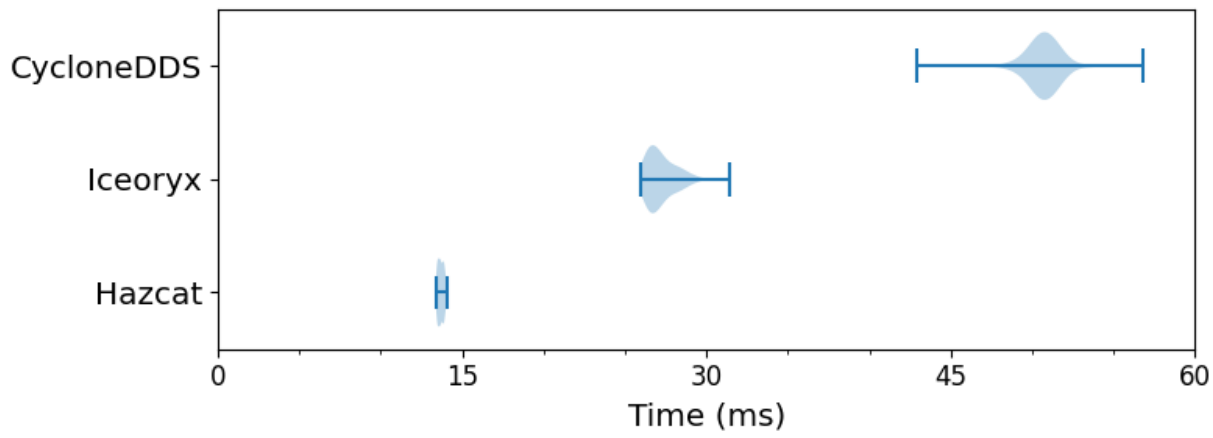


Figure 2.5: End-to-end Latency Distributions for Two Component Experiment

time in the CUDA library, independent of their other sources of overhead. Hazcat, however, spends much less time running CUDA code. Since the three are running the same kernel, we can assume time spent on GPU computation remains constant, and the variation observed is a result of reduced GPU memory operations.

We see this confirmed in Figure 2.4. Trials with Iceoryx see reduced latency compared to CycloneDDS, despite running the same code. This is because of CycloneDDS's worse overhead and required memory copies between components (as dissected in 2.3a). Hazcat runs slightly modified code, where the memory copies in userspace have been removed. Almost all of the end-to-end latency under Hazcat is from GPU computation.

The Iceoryx and Hazcat variations both implement some type of zero copy and as a result have reduced inter-component (and even intra-component) latency – the latter is due to performing a *publish* call on the underlying middleware while the node is running. This call performs a memory copy in CycloneDDS but has minimal overhead in Iceoryx and Hazcat.

CycloneDDS and Iceoryx incur the overhead of additional GPU memory operations when using hardware acceleration. This is the primary performance benefit of Hazcat over Iceoryx: eliminating unnecessary GPU memory operations.

Figure 2.5 shows density plots of the end-to-end latency of each experiment variation. The distributions on the hardware accelerated workloads show that both Hazcat and Iceoryx have tight lower bounds, highlighting that with them the best case is the typical case.

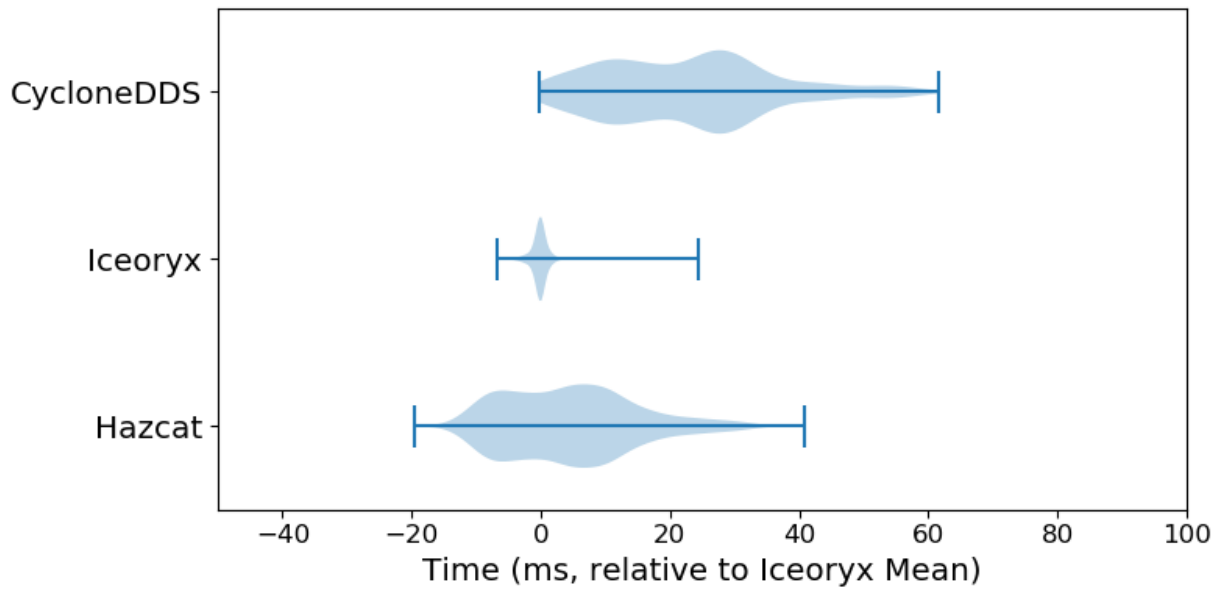


Figure 2.6: Relative performance with randomized ROS graphs

CycloneDDS, with its higher overhead and excess inter-component memory copies, sees more variation.

The most interesting finding is that Hazcat also has a tight upper bound. Iceoryx's use of GPU memory copies creates a source of latency jitter. Hazcat, on the other hand, has no such issue, making it behave much more predictably compared to the other two frameworks.

Synthetic ROS Graphs

To demonstrate Hazcat's performance in more complex workloads, we synthetically generated 98 ROS graphs with randomized message sizes for topics and randomized CPU or GPU computation for nodes. Graphs ranged from 2 to 14 nodes, 1 to 17 edges, and had critical paths ranging from 2 nodes to 5.

Each graph was run using Hazcat, Iceoryx, and CycloneDDS, and their end-to-end latencies were measured. The first three samples were discarded from each run, to eliminate outliers caused by transient effects during startup or shutdown.

Different graphs will naturally have vastly varying end-to-end latencies. We observed mean latencies ranging from 15 to 206ms. Since plotting all 98 graphs would be infeasible, measurements were instead normalized. For each graph, performance was measured as the deviation from the mean latency of that graph observed on Iceoryx. That is, if 3 runs of the same graph on Iceoryx take 10ms, 12ms, and 17ms (mean of 13ms), each run will then be plotted in Figure 2.6 as -3ms, -1ms, and 4ms. If a run of the same graph on Hazcat takes 8ms, it will be plotted as -5ms.

Hazcat had a better mean performance than CycloneDDS for every graph. It performed comparably (±10%) with Iceoryx for 60% of the graphs, and out-performed Iceoryx for 9% of the graphs.

When examining individual graphs, we notice that Hazcat performs comparably to Iceoryx in graphs dominated by CPU work, as well as graphs with wide work but a short span. Such graphs comprised the majority of our test set. The largest performance benefits come when multiple successive GPU nodes are chained together, and can benefit from the device-memory zero copy.

There was a greater deal of temporal variance in Hazcat, which we ascribe to its experimental implementation.

2.1.6 Conclusions and Future Work

The primary contribution of Hazcat's heterogeneity aware zero-copy features for heterogeneous computing environments is that memory copies are only performed when data in one memory domain is needed in a different domain. In all other cases, memory is recycled, drastically reducing the worst case end-to-end latency of an application as well as removing sources of nondeterminism.

In our synthetic benchmarks with randomized ROS graphs, we saw an average case speedup of 1-73% over the unoptimized CycloneDDS middleware, depending on the graph in question, with 27% being typical. When compared to Iceoryx, we observed anywhere between nearly a 1.8x slowdown to a 2x speedup, again, depending heavily on the graph in question.

The largest performance benefits are mostly seen when consecutive components in the application all operate in the same domain. Even in the worst case, for an application using n domains, we never saw more than n-1 copy operations per message.

We observe that in addition to performance gains in hardware acceleration workloads, our framework has a decentralized design, which additionally reduces the requisite developer knowledge, as there is no 3rd party daemon to launch and no ring buffers to configure. Built-in defaults and runtime initialization allow it to perform well even when limited information is provided about the application.

One obvious avenue of future work is to support a wider range of hardware. Here, we only discuss CUDA based usage of NVIDIA hardware, but FPGAs are an important target for future research as well. Given the aforementioned lapses in the CUDA drivers' support of IPC, we plan to investigate other approaches to support GPUs [30] [97].

As a motivating application to test Hazcat's usefulness in the ROS 2 ecosystem, it is proposed to modify the Autoware Realtime Benchmark [1] to provide CUDA support. Then, an empirical comparison can be made between ROS 2 applications running on Iceoryx vs Hazcat.

Zero Copy with non-POD Message Types

Previously stated, the biggest competitor to Hazcat is Iceoryx, an interprocess zero-copy communication middleware. One of the main limitations of Iceoryx is only being able to support Plain-Old-Data (POD) message types[3]. That is, data allocations with a fixed size known at allocation time and no recursive elements, as is typical with object-oriented programming. Currently, Hazcat also shares this limitation.

The ability to allocate an object and then decide the size of a member array at a later point has obvious benefits from a programatical standpoint. But the ability to support a message comprised of multiple allocations has an additional benefit when working with device memory: the ability to spread a message across memory domains.

[3]In the time between when the Hazcat paper [13] was published and when this dissertation was written, the developers of Iceoryx have attempted to address this exact feature and determined there was not sufficient industry interest to justify complicating their design. This suggests support for non-POD message types is at present primarily of academic interest, but this discussion is left in the dissertation for completeness.

For instance, when storing an image in ROS 2, the message type contains not only the image data, but also a header, containing metadata such as dimensions and encoding information. If the image data is to be processed on a hardware-accelerated device, this header data would provide valuable information to pass to the kernel launch, or determine whether the kernel should even be executed.

Enabling these kind of hybrid-compute applications motivates Hazcat as a useful player in the middleware space.

This basic principle of storing data across different memory domains already exists, in the form of NVIDIA Isaac [94]. However, it is limited to the CPU+CUDA domain, only supports ROS (2) applications, and the mechanism is abstracted behind a proprietary NVIDIA library so it can only be used with compute kernels also provided by the NVIDIA Isaac API.

Implementation in ROS 2

Messages in ROS 2 are defined in a YAML-style file which are interpreted and converted into a code representation via rosidl [59]. A variation of rosidl would need to be developed to convert these messages into a hybrid-memory format. Currently, arrays in ROS 2 are represented with the std::vector class, which would need modifications to work with device memory and the Hazcat framework.

YAML already supports comments, so that can be extended to support pragmas. In this way, it can be indicated which fields in a message should be in host memory and which can be in device memory. For simplicity, the scope of work in this dissertation is restricted to only a single hardware acceleration device per message.

Without a HWA pragma, a message is assumed to exist entirely in one domain, and Hazcat will copy it to other domains as requested. When a HWA pragma is present, only the indicated fields will be copied by Hazcat. All other fields (e.g., headers and metadata) remain strictly in host memory. The field with the HWA pragma typically will be a large array.

2.2 Allocators for Device Memory

The Hazcat framework requires implementing a diverse set of allocation algorithms for device memory. Iceoryx only accomodates a statically allocated buffer array, but many applications require dynamic memory allocation.

The field of dynamic memory management could be said to have been started by Knuth [76]. Some 30 years of progress is well summarized by Wilson and Johnstone [69, 120] in their comprehensive survey. Since then, further improvements have been made [38, 64, 88, 107], but overall the subject can be considered to be highly mature.

However, we challenge a prevailing assumption in the literature over the last 50 years which, if reconsidered, necessitates a fundamental reevaluation of many classical memory management algorithms. We pose a model where these allocation algorithms run on a host system but manage device memory. This model is exemplified in accelerated applications without unified memory that copy data in and out of device memory before and after a kernel call. Managing device memory from the host device incurs the following constraint: the allocator can't read the memory it is managing.

This means we are unable to use boundary tags, a concept first introduced in Knuth [76] that has been ubiquitous in nearly every allocation algorithm. In this section, we propose alternate designs to work around this constraint, and discuss in general the implications of this system model. Most existing memory management algorithms include metadata as headers or footers within allocated blocks. In contrast, our system model assumes that the allocator is running on a host system, managing device memory. This may occur, for example, if part of a computation is offloaded to a GPU or FPGA device while the rest of the computation runs on a multicore processor. This in turn implies that the compute device cannot (conveniently and efficiently) access the memory it manages. Therefore, any data needed by the allocation algorithm cannot be stored in its allocated blocks.

We motivate this system model by considering the usecases surrounding how device memory is managed. In the current state of the art, memory allocation/deallocation is either done on the peripheral compute device or in (often proprietary) device drivers. This causes memory allocators to be treated as fixed black boxes.

However, different applications may have differing needs for how to manage their memory. The most optimal allocator may be high performant, real-time constrained, or have domain-specific characteristics catering to the application. Fixing the allocator choice in proprietary drivers and hardware denies developers the choice to optimize this aspect of their systems through selective mapping of portions of the application and its supporting libraries to different computational devices. Examples of such applications include drone cinematography [24] and real-time hybrid simulation experiments in earthquake engineering [36].

For our work, we assume that existing hardware and drivers are used to allocate arbitrarily large blocks of memory, but finer-grained memory allocation is then done in userspace by the host machine. As was mentioned previously, these allocators are constrained by their inability to read the memory they are managing and cannot store metadata in allocated blocks. We present alternative algorithms that address this constraint.

First, in Section 2.2.1 we present a brief survey of existing dynamic memory allocation algorithms. In Section 2.2.2, we propose alternative measures to overcome the constraint of being unable to read managed memory. In Sections 2.2.3 and 2.2.4, we present our updated alternative allocation algorithms. We compare the performance of one of our algorithms to the default CUDA memory allocation functions in Section 2.2.5. Finally, we conclude in Section 2.2.6 with thoughts on implementations and usecases for this work.

2.2.1 Background and Related Work

Before introducing our new algorithms, we present a brief survey of existing memory allocation algorithms. Broadly speaking, these fall into 4 categories [69].

Sequential Fit

Free and allocated blocks of memory are linked together using header information within the allocated block itself. The resulting structure is called a free list. Examples include first fit, next fit, and best fit.

Segregated Lists

Portions of free data are grouped by size to aid in the lookup of a suitable block, including simple segregated storage, segregated fit, and TLSF.

Buddy Systems

Free blocks are only allowed to coalesce with a preassigned buddy, generally strictly sized in powers of 2. Examples include binary buddy [98] and double buddy [98, 122].

SIMD Allocators

Allocators may be designed for massively parallel programs [64,107,121] where multiple threads may be requesting memory simultaneously. These typically are used in device code. In this dissertation, we consider memory that is preallocated by host code, prior to a kernel launch.

Sequential Algorithms

Although there are a multitude of sequential algorithms, they all boil down to the same idea: all free blocks are kept in a free list, which is iterated through until a suitable block is identified for allocation. The only difference is what qualifies as a "suitable block".

The following are some of the most common sequential algorithms.

Best Fit

The entire free list is iterated and the smallest block that is greater than or equal to the requested size is used. The search is terminated if a candidate block is exactly of the requested size.

First Fit

The first block that is large enough to accommodate the requested size is used.

Next Fit

Like first fit, except each traversal of the free list resumes from the last position, instead of starting over from the beginning.

In real world scenarios, tests have shown that next fit performs the best in terms of allocation time, although it suffers a fragmentation penalty compared to the others.

Segregation Algorithms

In segregation algorithms, multiple free lists are maintained for different size classes. There are three primary types.

Simple Segregated Storage

Under this algorithm, every allocation request is rounded up to the next power of two. Free blocks are not split, and freed blocks are not coalesced. If a requested allocation has no available blocks in the free list for its size class, then a new block is created from the free heap.

Segregated Fit

This variation allows blocks to be split. Since not all blocks within a given size class are the same size, allocation requests are always serviced using the class size that is one degree of magnitude larger.

Since all blocks in the larger class size are guaranteed to be large enough to accommodate the request, it is unnecessary to perform iteration. These free lists are then treated like queues, only interacting with the front element. This prevents the frivolous search of a free list with no viable candidates, and reduces the allocation time to O(1). Additionally, it can perform deallocation in O(1) time, making it highly suitable for real-time systems, which require bounded execution.

One commonly used implementation is Doug Lea's Malloc [82]. It is a segregated fit algorithm that uses a combination of logarithm and linear spacing between its bins, achieving very low fragmentation in real-world evaluations [69]. It serves as the basis for the default allocator in the C programming language.

Two-Level Segregated Fit (TLSF) [86–88]

Two-level segregated fit builds on the idea of segregated fit. As with segregated fit, TLSF has logarithmic size classes, but each size class is further divided into linear size classes. This retains the O(1) allocation and coalescence time complexity of segregated fit, but drastically reduces the fragmentation problem.

Buddy Systems

Buddy Systems [75, 98] can be considered a subclass of segregation algorithms [69], in that memory is sorted into size classes. However, they bear the additional constraint that freed blocks cannot be coalesced with any neighbor, only with a preordained buddy.

When an allocation request is made, it is rounded up to the size class. The heap is then split into two buddy blocks. The first is split further, and so on recursively until the desired size class is created. Like with segregated fit, each size class has a free-list allowing allocation in O(1) time.

After deallocation, if two buddies are both free, they are coalesced. This operation can be performed recursively up the binary heap, possibly taking O(logn) time.

This larger deallocation time complexity means that applications with real-time constraints are often better served by a conventional segregated fit algorithm (such as TLSF).

As with simple segregated storage, the commitment to block classes at strict sizes by this algorithm can result in a great deal of internal fragmentation within allocations. This can be assuaged somewhat by variants to the algorithm, listed below.

Binary Buddies [76, 98]

The traditional and simplest implementation. It operates exactly as described above.

Double Buddies [98, 122]

To resolve the internal fragmentation issue of conventional buddies partially, double buddies keeps two heaps with staggered class sizes, e.g., one heap with sizes of 2, 4, 8, ... and another with 3, 6, 12, ...

Fibonacci Buddies [63]

This approach assumes the heap size is a Fibonacci number. Since every Fibonacci number is the sum of two other Fibonacci numbers, blocks can be split recursively. This split is uneven, helping address the internal fragmentation caused by using strictly powers of 2. The ratio between consecutive size classes using Fibonacci buddies is approximately $\phi \approx 1.618$.

SIMD Allocators

This is a broad class of allocators designed for use specifically in SIMD systems, where many allocation requests may be made in parallel, creating risk for contention. A survey of currently available work is presented in [121].

Such allocators are often designed to avoid or eliminate locking overhead and are typically useful when threads in a GPU need to allocate small amounts of memory simultaneously in device code. This doesn't align with our model of host-based allocation code, and caters to a different use-case.

Our work is useful for parallel applications that are highly heterogeneous, intermixing both unaccelerated and accelerated tasks. The accelerated tasks operate on memory allocated prior to kernel launch. In such a system, all the memory allocation calls, both for host and device memory, would be done from host code.

Meanwhile, these SIMD allocators require execution of device code to run, leaving them outside the scope of the intersectional memory model we put forward. Furthermore, it has been demonstrated [125] that invoking device calls that manage CUDA memory causes implicit device-wide synchronization operations, impacting unrelated processes on the system. This motivates avoiding device calls where possible, and managing device memory on the host achieves this.

2.2.2 Alternatives to Free Lists

This work explores a model in which the host device manages allocation of device memory. This prevents the allocator from reading the memory being managed, which presents new challenges not encountered in traditional memory allocation schemes. A key hurdle is that we cannot use boundary tags, a standard approach that was first mentioned by Knuth [76].

The traditional usage of boundary tags is illustrated in Figure 2.7. A free list is formed by a linked list of blocks that are available for reuse. The header points to the next element in the free list, and the footer points to the header of the same block, which enables coalescence between two adjacent blocks in O(1) time. After a block is freed, one can subtract the footer size from the block's address to obtain the address of the header of the prior block. From

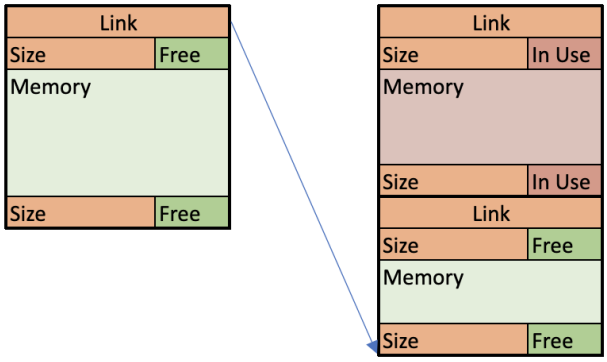


Figure 2.7: Demonstration of Boundary Tags

there, one can check if that prior block is in the free list, and if so, the two will be coalesced. The next block in the free list also can be checked to see if it's adjacent in memory, possibly coalescing a total of three blocks.

These free lists are used in all Sequential Fit and Segregated Fit algorithms and a replacement mechanism will be needed to implement these in device memory.

Arrays and Hybrid Array Lists

Array-based implementations of a free list for Sequential Fit algorithms using sub-pagesize allocations have high overhead, due to the potentially large number of small allocations. If we assume the smallest allocation size is 8 bytes, and an additional 8 bytes is needed to point at the free block, then the allocator would need an amount of host memory equal to the device memory it is supposed to be managing. However, this is true even for conventional sequential fit allocations. Between the header and footer boundary tags, the worst case overhead is potentially 75%.

With an array-based approach, we track all blocks, free or in-use. So, each allocation of device memory only needs a single word (8 bytes) as overhead in host memory. We assume that we can spare a bit to indicate whether the block is free or not. Naively, allocation and coalescence would be O(n) operations, corresponding to array insertion and deletion respectively, but we can reduce this by using Hybrid Array Lists (HALs) [101].

Our implementation is informed from prior literature [101], and detailed in Appendix A, which describes insertion, removal, and searching of a HAL. Two separate HALs are maintained: one to track free blocks, and another to track in-use blocks.

Algorithm 1 Memory Allocation

Require: $used\_list\_tail$, $free\_it$, $size \geq 0$
1: $addr \leftarrow free\_it.addr$
2: **if** $free\_it.size > size$ **then** ▷ Split entry to only use space needed
3: $\quad free\_it.address \leftarrow free\_it.address + size$
4: $\quad free\_it.size \leftarrow free\_it.size - size$
5: **else**
6: $\quad remove(free\_it)$
7: **end if**
8: $it \leftarrow search(used\_list\_tail, addr)$
9: $insert(it + 1, \{addr, size\})$
10: **return** $addr$

The allocation algorithm (Algorithm 1) assumes that a candidate free block already has been identified: see Section 2.2.3 for discussion of that selection process. Given a candidate block in the free list, it must be moved to an in-use list, and possibly may be split if the block is larger than the request.

The free list is also guaranteed always to have at least one entry: the heap itself.

During deallocation (Algorithm 2), a block is removed from the in-use list and added to the free list. The algorithm will check the prior and subsequent blocks in the free list and perform coalescence if it is possible.

Partial deallocation is supported, e.g., freeing the last 2kB of a 10kB block. In the event of a partial deallocation, the in-use block will be shrunk, and otherwise it is simply removed from the in-use list.

When inserting into the free list, the previous and subsequent (left and right) blocks are checked and coalesced if either (or both) is adjacent to the block being inserted.

Algorithm 2 Memory Deallocation

Require: $free\_list\_tail$, $used\_list\_tail$, $addr \geq 0$ ▷ Find chunk in used list

$it \leftarrow search(used\_list\_tail, addr)$

$size \leftarrow it.size$ ▷ Remove the chunk from the used list (or shrink it, in the case of partial deallocation)

if $it.addr = addr$ **then**

    $remove(it)$

else

    $it.size \leftarrow addr - it.addr$

    $size \leftarrow size - it.size$

end if

$left \leftarrow search(free\_list\_tail, addr)$

$right \leftarrow left + 1$ ▷ May be past-the-end

if $left.address + left.size \geq addr$ **then**

    if $right.address \leq addr + size$ **then** ▷ Calculate size of left to coalesce all 3, and remove right

        $end \leftarrow right.address + right.size$

    else ▷ Calculate size of left to coalesce just 2

        $end \leftarrow addr + size$

    end if

    $left.size \leftarrow end - left.address$

else if $right.address \leq addr + size$ **then** ▷ Calculate new size and address of right

    $right.size \leftarrow right.address + right.size - addr$

    $right.address \leftarrow addr$

else ▷ Insert block without coalescing

    $insert(right, \{addr, size\})$

end if

Bitmasks

Assuming O(n) searches are acceptable, one way to minimize overhead is through the use of bitmasks. In this approach, device memory is divided into its smallest allocatable size (e.g., 8 bytes), and the only overhead is a single bit in host memory to indicate whether that block is free or not. If we consider an allocation size lower bound of 8 bytes to be the worst case scenario, this creates an overhead of only 1.5% (one bit for 8 bytes). That produces 16MB of overhead for a gigabyte of device memory.

Allocation is done by finding a contiguous string of set bits in the bitmask corresponding to the desired size. For example, to allocate 1kB of memory, it is necessary to find a string of 128 bits that are all ones, incurring an O(n) cost for allocation, on the order of the address space size. The `clz` and `ffs` hardware instructions can be used to accelerate a bitsearch such as this.

All these bits then must be set to zero to indicate their corresponding blocks are in use. When the block is freed, they are all returned to ones (coalescence is implicit here). This means allocation and deallocation operations also incur an O(s) cost, on the order of the block size.

Bitmasks have low overhead and relatively high time complexity on the order of allocation size. Because of the low overhead, a potential use-case is to use bitmasks to manage a pool of small memory chunks, either of the same size (as in an object pool) or commingled varying sizes, as described above.

We can optimize the allocation search complexity by using the next-fit method. Prior work [10,76] has shown that next-fit improves on temporal performance over other sequential searches.

Hash Tables

Another strategy is to use a hashtable, keyed by device memory address, to store all the information typically found in the header and footer of an allocated block.

The traditional algorithms for free-list traversal and coalescence operation would need to be modified to accommodate an extra step to look up data in the hashtable. The in-use blocks would need to be tracked in the hashtable as well, as they also have necessary information in their header and footer.

With the use of hash tables, we aim to achieve O(1) allocation and deallocation, so co-mingling free and in-use blocks will not be a major performance concern.

The key for the hash table is the address of a block. The value contains (i) the block size, (ii) the address of the next block in a free list, (iii) the address of a previous block in a free list, and (iv) the address of the previous block adjacent in address space. Thus we can create, in effect, free lists that span across a hash table. The ability to maintain multiple free lists easily is useful for segregated fit algorithms.

The primary downside of this approach is the space overhead. Each entry contains 6 words (the key, block size, two references for a doubly linked list, one reference for a prior adjacent block, and a reference for a separate list linking collisions in the hash table). This means that in the worst case scenario, minimum allocation size of one word, the overhead is $\sim$ 83%. This is comparable to the worst case overhead of 80% seen in doubly linked free lists in conventional host-only algorithms. However, we only recommend this approach for applications with a fewer small allocations, such as those with a larger minimum block size, or any applications for which segregated fit algorithms are applicable.

Comparison

Below is a comparison of the three different strategies for replacing free lists. For allocation time of HALs and Hash Table lists, we ignore traversal when considering allocation. When these approaches are used in segregation algorithms (discussed later in Section 2.2.4), they are often treated as stacks, with the head of the list always being a viable candidate. Bitmasks do not have this benefit. Searching for an allocation candidate will always require traversal of the memory space.

Approach	WC Overhead	Allocation	Coalescence
HALs	50%	O(1)	O(n/m + m)
Bitmasks	1.5%	O(n)	O(s)
Hash Table	87%	O(1)	O(1)

n is the size of memory being managed. m is the size of an array block in a hybrid-array-list. s is the size of the allocated block.

Hybrid array lists are distinguished by their unbounded deallocation time. Even if an allocation algorithm doesn't require traversal to allocate a block, it may require traversal to coalesce. Since these hybrid array lists are intended to be sorted, entries are not guaranteed to remain in the same location in the list.

Allocation takes constant time, assuming all free blocks are eligible candidates, as is the case in segregation algorithms, which maintain multiple free lists for different size classes. Under this assumption, the last address in the list can be removed, which is as simple as decrementing a counter in the corresponding block. If the list must be traversed, as is the case of sequential algorithms (discussed in Section 2.2.3), then allocation becomes a O(n/m + m) process. This speed up in traversal, combined with lower overhead, makes HALs an ideal replacement for free lists where sequential algorithms are concerned.

Bitmasks always a require a traversal to identify an allocation candidate. This can be sped up substantially using the find-first-set command, which can search for 1 bit in a 64-bit word in a single machine-level instruction. This is still O(n), but offers a reasonable strategy for managing smaller object pools.

Storing linked lists in Hash Tables incurs a substantial overhead, so it is not recommended for smaller allocations. It does permit constant-time algorithms to retain their constant-time execution, which is crucial for real-time systems. This makes hash table stored lists the ideal replacement for free lists where segregation algorithms are concerned.

2.2.3 Sequential Algorithms

All sequential algorithms are essentially the same, except for their stopping criteria. Below we include an algorithm for using best fit on device memory with hybrid array lists. This is followed by a description of the modifications needed for next-fit and first-fit.

Algorithm 3 Best Fit Allocation

Require: $free\_list\_tail$, $used\_list\_tail$, $size \geq 0$
1: $it \leftarrow free\_list\_tail$
2: $candidate \leftarrow it$
3: $it \leftarrow it + 1$
4: **while** $it.size \neq size$ **and** $it \neq free\_list\_tail$ **do**
5: **if** $it.size < candidate.size$ **and** $it.size \geq size$ **then**
6: $candidate \leftarrow it$
7: **end if**
8: $it \leftarrow it + 1$
9: **end while** ▷ Invoke Algorithm 1 on candidate block
10: **return** $allocate(used\_list\_tail, candidate, size)$

Best-fit tries to find the smallest block that is large enough to accommodate the requested size. If a block with an identical size is found, the search can stop. However, it often requires traversing the entire list.

Contrarily, next-fit and first-fit select the first free block that is greater than or equal to the requested size. Next-fit is distinct from first-fit because it resumes the next allocation search where the last one left off. First-fit is memoryless and always restarts from the beginning of the free list.

All of the sequential algorithms use the same deallocation process, illustrated in Algorithm 2.

2.2.4 Segregation Algorithms

Segregated Fit

In segregated fit, multiple free lists are maintained in size buckets for different powers of 2. Unlike sequential fit, the lists aren't searched. Instead, the first block is always selected.

Algorithm 4 Allocate memory in segregated fit

Require: $free\_lists$, $bitmask$, $hashtable$, $size \geq 0$

$requested\_bins \leftarrow 1 \ll \lceil \log_2(size) \rceil$

$order \leftarrow \text{ffs}(requested\_bins)$

▷ Lookup head of candidate list in hashtable

$it \leftarrow ht[free\_lists[order]]$

$free\_lists[order] \leftarrow it.next\_free$

$it.free \leftarrow$ False ▷ Split off surplus portion of block

if $size < it.size$ **then**

    $new\_block.size \leftarrow it.size - size$

    $new\_block.free \leftarrow$ True

    $new\_block.addr \leftarrow it.addr + size$

    $new\_block.prev\_adj \leftarrow it.addr$

    $new\_block.prev \leftarrow 0$

    $it.size \leftarrow size$ ▷ Place new block at head of free list in its size class

    $bin\_idx \leftarrow \lfloor \log_2(new\_block.size) \rfloor$

    $new\_block.next \leftarrow free\_lists[bin\_idx]$

    $new\_block.prev \leftarrow 0$

    $ht[new\_block.next].prev \leftarrow new\_block.addr$

    $free\_lists[bin\_idx] \leftarrow new\_block.addr$

    $bitmask \leftarrow bitmask \mid (1 \ll bin\_idx)$

    $ht[new\_block.addr + new\_block.size].prev\_adj \leftarrow new\_block.addr$ ▷ Insert into the hashtable

    $ht[new\_block.addr] \leftarrow new\_block$

end if

$ht[it.addr] \leftarrow it$

return $it.addr$

Given a requested size, it is rounded up to the nearest order of magnitude, and then the first block from that list is selected. This means (assuming no empty free lists) the selected block may be nearly 4x larger than the requested size, causing 75% fragmentation, as discussed in prior literature [69, 120].

All lists are initialized pointing at a null block: an invalid block in the hashtable is meant to indicate the end of a free list. Accompanying this array of lists is an availability bitmap that can indicate which lists are non-empty. If the free list for a desired size class is empty, the next eligible list can be found by using the find-first-set (ffs) bit operation on the bitmap.

Our implementation stores the free lists in a hashtable, as discussed in Section 2.2.2. Algorithms 4 and 5 show pseudocode for allocation and deallocation, respectively.

Two-Level Segregated Fit for Device Memory

Two-Level Segregated Fit [86–88] (TLSF) is a real-time dynamic memory allocation algorithm. Its primary benefits are reduced fragmentation and O(1) allocation and deallocation. It is an extension of segregated fit, dividing class sizes not only logarithmically, but also linearly, in a two-tiered system.

The allocation and deallocation steps are functionally the same, except the lookup step requires consulting 2 bitmasks. One is logarithmic and functions just as detailed in Algorithm 4 and Algorithm 5. TLSF differs in that this points at another bitmask which subdivides the space linearly, as illustrated in Figure 2.8. This tier then points to a free list of blocks in that size class. If the free list is non-empty the linear bitmask will have a corresponding 1 set. If the linear bitmask is non-zero, then the logarithmic bitmask will have a corresponding 1 set.

Free list manipulations are done the same way as in segregated fit, so Algorithms 4 and 5 only need to be modified to account for this two-tiered lookup process and bitmask manipulation.

Algorithm 5 Deallocate memory in segregated fit

Require: $free\_lists$, $bitmask$, $hashtable$, $addr \geq 0$

$it \leftarrow ht[addr]$
$left \leftarrow ht[it.prev\_adj]$
$right \leftarrow ht[it.addr + it.size]$
if $left.free$ **then**
    if $right.free$ **then** ▷ Coalesce all 3, erase it and right
        $left.size \leftarrow left.size + it.size + right.size$
        $ht[right.addr + right.size].prev\_adj \leftarrow left.addr$ ▷ Update bitmask
        if $right.prev = right.next$ **then**
            $unset\_bit \leftarrow 2^{\lfloor \log_2(right.size) \rfloor}$
            $bitmask \leftarrow bitmask \wedge \neg unset\_bit$
        end if
        $remove(right)$
    else ▷ Coalesce left block, erase it
        $left.size \leftarrow left.size + it.size$
        $right.prev\_adj \leftarrow left.addr$
    end if
    if $it.prev\_free = it.next\_free$ **then**
        $unset\_bit \leftarrow 2^{\lceil \log_2(it.size) \rceil}$
        $bitmask \leftarrow bitmask \wedge \neg unset\_bit$
    end if
    $remove(it)$
    $it \leftarrow left$
else if $right.free$ **then** ▷ Coalesce right block
    $it.size \leftarrow it.size + right.size$
    $ht[right.addr + right.size].prev\_adj \leftarrow it.addr$ ▷ Update bitmask
    if $right.prev = right.next$ **then**
        $unset\_bit \leftarrow 2^{\lfloor \log_2(right.size) \rfloor}$
        $bitmask \leftarrow bitmask \wedge \neg unset\_bit$
    end if
    $remove(right)$
end if ▷ Put coalesced block in free list in its size class
$bin\_idx \leftarrow \lfloor \log_2(it.size) \rfloor$
$it.next \leftarrow free\_lists[bin\_idx]$
$ht[it.next].prev \leftarrow it.addr$
$it.prev \leftarrow 0$
$free\_lists[bin\_idx] \leftarrow it.addr$
$it.free \leftarrow$ True
$ht[it.addr] \leftarrow it$
$bitmask \leftarrow bitmask \mid 2^{bin\_idx}$

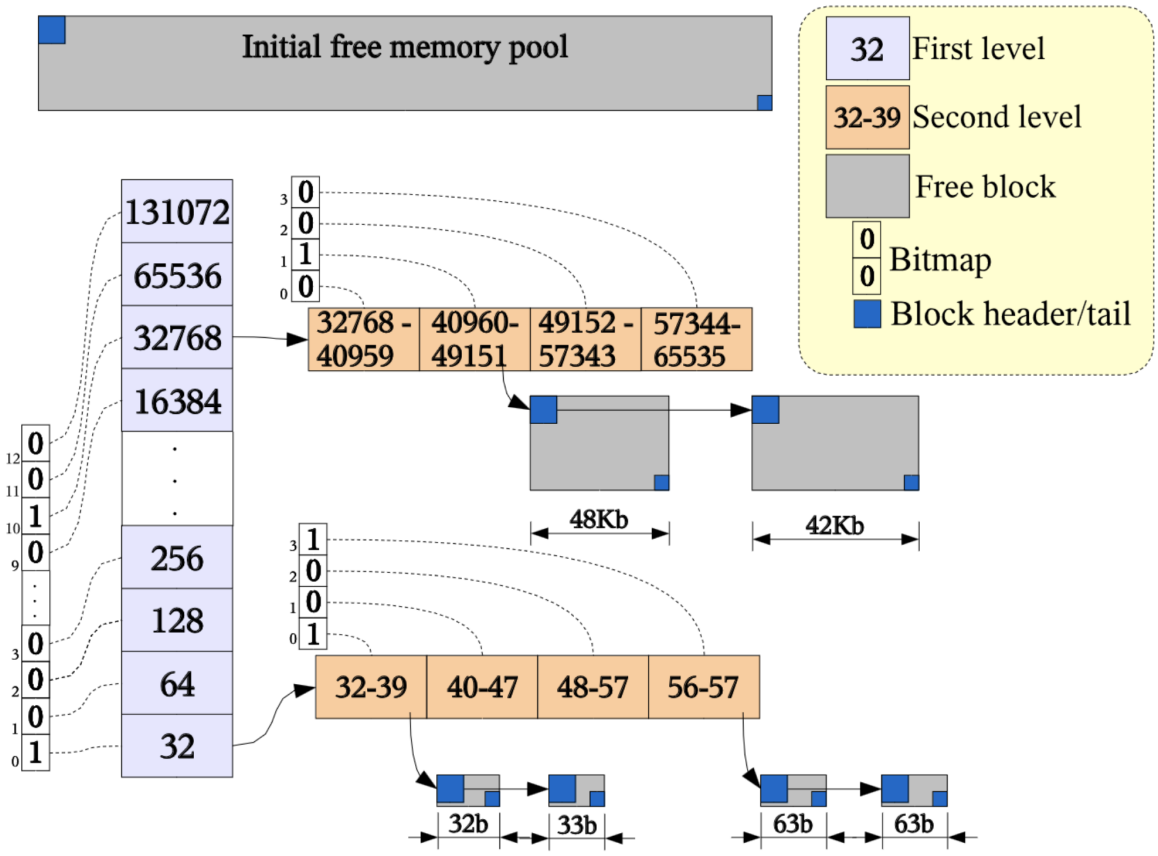


Figure 2.8: TLSF Data Structure [87]

Hybrid Allocators and Object Buffers

Hybrid approaches also may be employed, using object buffers to manage object pools for allocations smaller than a page (<4kB), and segregated lists for larger allocations.

Object pools allow for the efficient allocation of vast amounts of small allocations, but are not a practical approach for larger allocations. A hybrid approach can permit the best of both worlds.

It is also important to consider the practical use of such a hybrid approach in this context. A host-based allocator for device memory is unlikely to allocate large numbers of small objects, but rather large memory blocks that are passed to a kernel call for a hardware-accelerated device. If small allocations are created, they would be intermediary memory allocated by device code, likely using a SIMD allocator such as XMalloc [64] or ScatterAlloc [107]. These

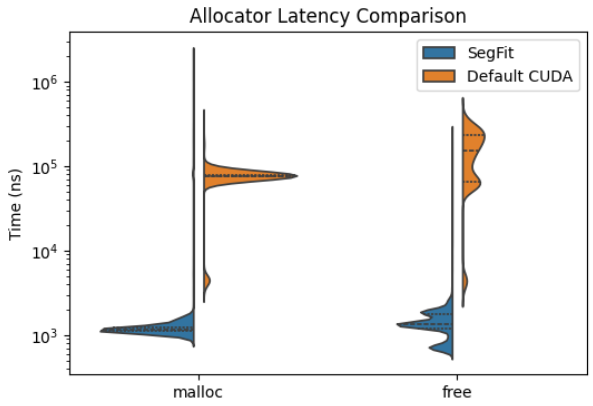


Figure 2.9: Latency of malloc and free

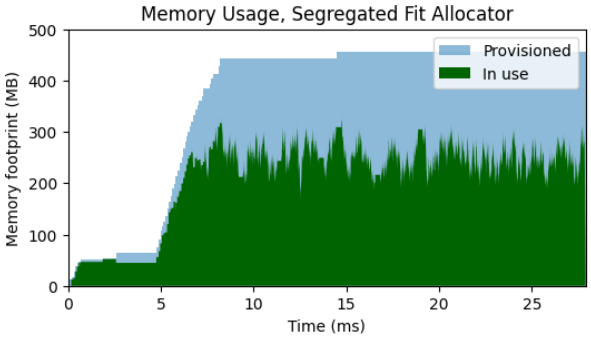


Figure 2.10: Memory usage during benchmark

allocators are designed to handle many allocation requests in parallel without the need for locks, a usecase that doesn't apply to the host-based model we have presented in this dissertation.

So, our usecase would be best served by limiting the smallest granularity to a medium sized value, such as 4kB, and exclusively using a segregated fit algorithm.

2.2.5 Evaluation

Evaluating the nuanced use cases that this work targets is challenging. SIMD allocators do not offer a proper target for comparison since our allocators are run from the host and are not intended for parallelization. Many good memory allocator benchmarks also exist for host memory [83], including real-world applications. However, they too are unsuitable for comparison since the device memory we are allocating is inaccessible.

Instead, we compare our work to the default CUDA allocator. We adapted the malloc-large benchmark from the MiMalloc benchmark suite [83] to use an implementation of our segregated fit algorithm. This algorithm randomly allocates and frees blocks of 1kB and 16MB in size, several thousand times.

As a baseline, we also ran the benchmark with ordinary calls to cudaMalloc and cudaFree. The order and size of the alloc and free operations was consistent between our work and the baseline. Latency comparisons between the two are shown in Figure 2.9.

Our work outperforms the default CUDA allocator by 2 orders of magnitude. There are some outliers during program initialization where the segregated fit malloc can take upwards of 2ms, but these disappear as the program settles into a steady state.

We measure fragmentation in our experiments (Figure 2.10) by tracking not only the memory in use by the benchmark, but also the physical device memory provisioned to the benchmark process. In our segregated fit allocator, we observe a roughly 50% fragmentation ratio. This is consistent with prior work [69] on segregated fit. A Two-Level Segregated Fit allocator would see improved fragmentation [88].

2.2.6 Conclusion

We note that prior work on dynamic memory allocation relies on free lists and boundary tags. We've posed a heterogeneous system model where memory is stored on one device and managed from another. After observing that existing algorithms are no longer applicable, we've provided alternatives to free lists and updated versions of classic sequential and segregated fit algorithms.

An implementation of our segregated fit algorithm was tested against conventional calls to cudaMalloc and cudaFree. It was found that, after a warmup period, our work consistently outperformed the CUDA calls by two orders of magnitude. This is mainly due to its implicit and efficient memory recycling.

We do not claim to outperform established GPU allocators [121] at scale, nor to address their highly parallelized use-cases. However, in specific circumstances where kernels are not expected to allocate memory and diverse accelerated computation is combined into a large heterogeneous application, performance benefits can be achieved by managing memory on the host. It is unnecessary to invoke device calls to free memory, and instead it can be recycled dynamically.

Chapter 3

ROS 2 Scheduler

The previous chapter focused on *data movement* between components. This chapter turns to the *scheduling* of those components. We specifically focus on the scheduler for ROS 2, a robotics framework introduced in Section 1.1.

The ROS 2 scheduler, called an *executor*, exists in the topmost layer of the ROS 2 stack, rclcpp. The default executor, often referred to as the 'SingleThreadedExecutor', has several characteristics and limitations documented in prior work [21, 29, 110, 113, 114].

- **Ready Set:** It uses a mechanism called a *ready set* to hold callbacks ready for execution. The ready set is not aware of individual instances of callbacks releases. It only indicates if a callback is ready for execution, and not how many instances are in the backlog.
- **Polling Points:** The ready set is populated only at specific *polling points*. Once populated, the executor processes one instance of each callback from the set before polling again.
- **Processing Window:** The interval between two polling points is known as the *processing window*. Callbacks becoming ready *during* this window must wait until the next polling point, potentially leading to significant delays and priority inversions, as lower-priority callbacks already in the ready set execute first. At most, only one instance of a given callback can be processed during this window.
- **Scheduling Policy:** It employs a round-robin policy within the ready set, with a static bias favoring timer callbacks over subscription callbacks, and only executing each callback once per processing window, regardless of the number of instances released.

This inherently fixed-priority scheme, combined with the blocking behaviour caused by processing windows, makes it difficult to provide strong real-time guarantees—especially for complex applications.

Existing work [29,110] has determined that ROS 2 applications, due to their pub/sub architecture, can be modeled as DAGs. When data arrives at a topic, all of its subscribers have attached callbacks that execute in response. These can be modeled as atomic tasks within the DAG. They may themselves publish to additional topics, triggering other callbacks. This can form a chain of execution. When there's multiple subscribers to a single topic, the chain branches, forming a DAG.

Aforementioned work by Casini [29] and Tang [110] provides a response time latency for this graph, and presents critiques of ROS 2's scheduler design. Since then, work has been done by other researchers [20,33,105,106] to improve these flaws and present new scheduler designs.

In 2023, the *events executor* [65] was introduced in ROS 2, replacing the *wait set* with an *events queue* and passing only one job to the processing window at a time. This eliminates the issues associated with the processing window, and provides a basis for a scheduler that is compatible with the classical real-time scheduling model of periodic task systems, detailed later in Section 3.1.1. Furthermore, the events queue can be replaced with a custom implementation for arbitrary scheduling policies.

This chapter uses content from two papers, both written in collaboration with TU Dortmund. The first paper [111] explores light modifications that can be made to the events executor to make it compatible with the classical real-time scheduling model of periodic task systems, and then provides an response time analysis for applications that use the events executor. The second paper [15] explores more generic applications by incorporating graph aware scheduling into decisions made by the events executor.

3.1 Reconciling ROS 2 with Classical Real-Time Scheduling of Periodic Tasks

This work addresses the following fundamental question:

> *Is it possible to use the events executor in ROS 2 in a manner that is compatible with the classical real-time scheduling theory of periodic task systems, in which every high-priority task can only be blocked by at most* one *lower-priority task, and a job is released* every time *the task period is reached?*

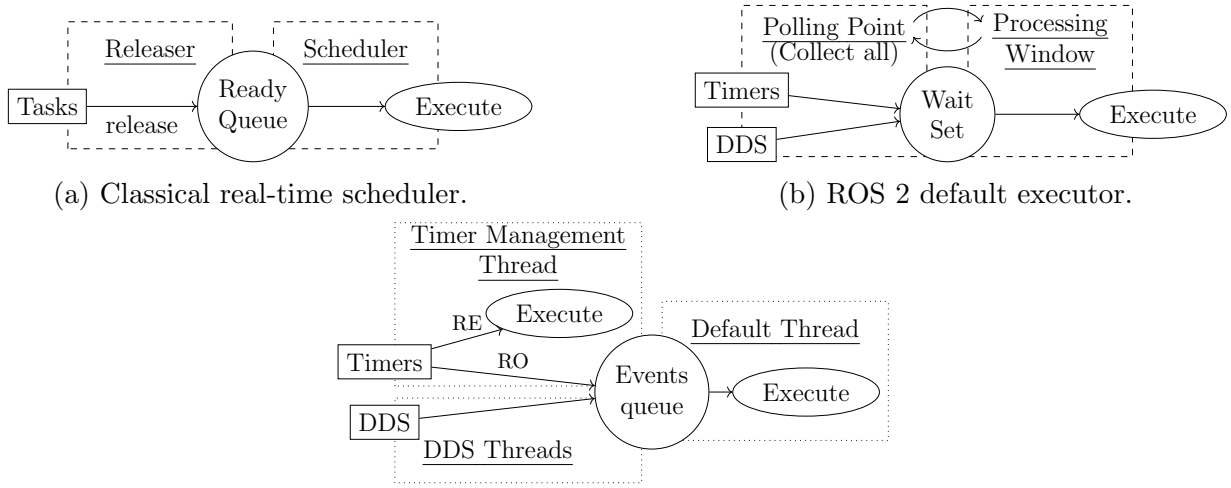
Our answer is *yes, but only under certain conditions.*

In this work, we uncover these conditions and show that any priority-based, non-preemptive scheduling strategy, can be realized with only slight modifications of the events executor, including Fixed-Priority (FP) and Earliest-Deadline-First (EDF) scheduling. Our solution is easy to apply since it requires only minor backend modifications of the events executor, which is natively included in ROS 2. With our solution, classical results from the real-time systems literature for priority-based, non-preemptive scheduling of periodic tasks can be applied.

Our Contributions:

- We present modifications to allow priority-based scheduling for the events executor in Section 3.1.4.
- We state the conditions that make our proposed executor compatible with priority-based, non-preemptive scheduling theory for periodic tasks in Section 3.1.5. Furthermore, we demonstrate how to apply analytical results in the literature to compute the worst-case response time and the end-to-end latency for the proposed ROS 2 scheduler.
- While the previous sections focus on timer tasks (i.e., tasks with periods), we extend to subscription tasks (i.e., tasks triggered by other tasks) in Section 3.1.6 and discuss the need for dedicated interfaces enabling the prioritization of subscriptions in ROS 2.
- In Section 3.1.7, we evaluate our findings, showing compatibility with the classical real-time scheduling theory and our proposal's benefits. We show the applicability of response-time bounds from the literature and significant improvements to end-to-end latencies in many cases.

In Section 3.1.1, we outline the different characteristics of typical priority-based schedulers and the default ROS 2 executor. The events executor is introduced in Section 3.1.2. We define the problem this chapter investigates in Section 3.1.3.



(a) Classical real-time scheduler.

(b) ROS 2 default executor.

(c) ROS 2 events executor.

Figure 3.1: Scheduling mechanisms.

3.1.1 Classical Schedulers and Executors in ROS 2

In this section, we introduce the scheduling mechanisms for periodic tasks in classical real-time scheduling and the default ROS 2 executor and highlight their differences. To that end, we use a running example of three tasks τ_1, τ_2 and τ_3. Task τ_1 should release a job every 10 time units, with an execution time of 3 time units. Tasks τ_2 and τ_3 should each release a job every 30 time units, with an execution time of 10 time units per job.

Classical Priority-Based Scheduler

The scheduling mechanism of the classical real-time scheduler is depicted in Figure 3.1a. Specifically, it consists of a *releaser*[4] and a *scheduler*, where the releaser inserts jobs into the ready queue and the scheduler decides which job in the ready queue should be executed.

The *releaser* can be implemented as a *timer interrupt service routine*. Specifically, a timer interrupt is triggered for each system tick, and the releaser decides whether to release a job of task $\tau_i \in \mathbb{T}$, given a set $\mathbb{T}$ of tasks. A **periodic task** $\tau_i \in \mathbb{T}$ is specified by the tuple $\tau_i = (C_i, T_i, D_i, \phi_i) \in \mathbb{R}^4$, where $C_i \geq 0$ represents the worst-case execution time (WCET),

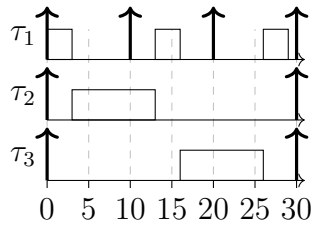
[4]We use the term *releaser* to indicate an independent component of a system that is responsible for determining if and when a job should be released. Its role is distinct from that of the *scheduler*, which decides which of the released jobs should be running at any point in time.

$T_i > 0$ defines the period, $D_i > 0$ is the relative deadline, and ϕ_i is the phase. The periodic task τ_i releases its first job at time ϕ_i and subsequent jobs are released every T_i time units. It is usually assumed that T_i is an integer multiple of the system tick for every task $\tau_i \in \mathbb{T}$. Jobs must finish within their relative deadline D_i after their release time.

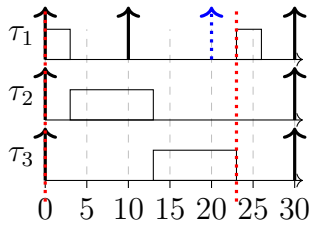
A *scheduler* determines which job in the ready queue should be executed. *Priority-based schedulers*, in which scheduling decisions are made by assigning priorities to jobs, have been widely studied in the literature. At any point in time when a scheduling decision has to be made, the *highest-priority job among the jobs in the ready queue* is allocated to the processor for execution. When a job completes its execution, it is removed from the ready queue. Every job of τ_i executes for at most C_i time units and has an absolute deadline specified as its release time plus the relative deadline. A task set is an *implicit-deadline system* if $D_i = T_i$ for all tasks $\tau_i \in \mathbb{T}$, and a *constrained-deadline system* if $D_i \leq T_i$ for all tasks $\tau_i \in \mathbb{T}$. If the deadline is not constrained by the period, i.e., $D_i > T_i$ is allowed, then we call the task set an *arbitrary-deadline system*. We consider arbitrary-deadline task systems in this paper, covering all possible cases.

In the literature, priority-based schedulers (on the task level) are classified into fixed-priority and dynamic-priority schedulers. A scheduler is a *fixed-priority scheduler* if, for any two tasks τ_i and τ_j, with different priorities, either all jobs of τ_i have higher priority than all jobs of τ_j or all jobs of τ_j have higher priority than all jobs of τ_i. In contrast, for *dynamic-priority* scheduling, a job of τ_i may have a higher priority than some jobs of τ_j but a lower priority compared to other jobs of τ_j. Specifically, the rate-monotonic (RM) scheduler (where a task with a shorter period has a higher priority) and the deadline-monotonic (DM) scheduler (where a task with a smaller relative deadline has a higher priority) are well-known fixed-priority scheduling policies, while the earliest-deadline-first (EDF) scheduler (where a job with the earliest absolute deadline has the highest priority) is a well-known dynamic-priority scheduling policy [84].

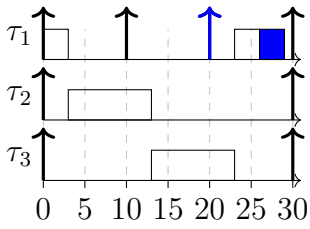
Furthermore, we differentiate between *preemptive* and *non-preemptive* scheduling algorithms. That is, while for preemptive scheduling a scheduling decision is made at every timer interrupt, for non-preemptive scheduling, decisions are only made at a few specific checkpoints of job execution. More specifically, in preemptive scheduling, at every timer interrupt, it is checked whether the currently running job is still the highest-priority job in the ready queue, and if so, it continues executing. Otherwise, the system performs a context switch,



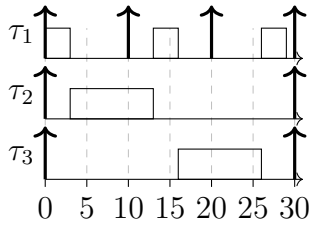
(a) Under classical non-preemptive FP scheduler.



(b) Under ROS 2 default executor.



(c) Under proposed scheduler without priority-ordering.



(d) Under proposed scheduler with priority ordering.

Figure 3.2: Schedules for the running example of Section 3.1.1.

preempts the currently executing job, and executes the new highest-priority job in the ready queue instead. On the other hand, for non-preemptive scheduling, a scheduling decision is only made when (i) a running job finishes its execution or (ii) a job is inserted into an empty ready queue. Since the scheduler can make a scheduling decision to start the execution of the highest-priority job in the ready queue only at those time points, a running job is never preempted and continues to be executed until it finishes. Worst-case analysis of non-preemptive schedulers for real-time systems has been widely studied, e.g. [40, 41, 50, 67, 91].

Example 1. We consider the running example from the beginning of Section 3.1.1 with three tasks. Specifically, we model them as *periodic tasks* with implicit deadlines, i.e., $\tau_1 = (3, 10, 10, 0)$, $\tau_2 = (10, 30, 30, 0)$, and $\tau_3 = (10, 30, 30, 0)$. Under RM scheduling, task τ_1 has the highest priority. The schedule for RM non-preemptive scheduling is depicted in Figure 3.2a. In particular, we observe that at time 10, task τ_1 is placed in the ready queue directly by the releaser. At time 13, the scheduler makes a new scheduling decision, identifies the second job of task τ_1 as the highest-priority job, and schedules it. Most importantly, all tasks are released periodically, and the task prioritization and job selection of the scheduler lead to a schedule where all tasks meet their specified deadline. □

Default Executor in ROS 2

The default single-threaded ROS 2 executor is depicted in Figure 3.1b. It schedules tasks' eligible jobs using a *wait set*. Its scheduling mechanism is split up into two alternating phases, the *polling point* and the *processing window*, which are performed sequentially in one thread.

The polling point functions like the releaser in classical real-time scheduling. At each polling point, the executor updates the wait set by sampling one job from each *activated* task. In ROS 2, tasks are typically either *timer tasks* or *subscription tasks*. Timer tasks are *time-triggered* and activated every period T_i, while subscription tasks are *event-triggered* and activated indirectly through message reception. Tasks can publish messages through the communication middleware of ROS 2, the DDS.

During each processing window, which is equivalent to the scheduler in classical real-time scheduling, the jobs are executed using a fixed-priority non-preemptive scheduling policy. The executor iterates over the wait set and selects the highest-priority job according to the fixed-priority order of their respective tasks. The executor then executes the job non-preemptively by calling a function (callback) associated with the task. After finishing the execution of all jobs in the wait set, the wait set is empty, and the executor starts a new polling point. The default ROS 2 executor prioritizes timers over subscriptions, and tasks of the same type are prioritized according to their order of creation. Specifically, the default ROS 2 executor does not provide any direct interfaces to configure task priorities.

In general, the design of the ROS 2 scheduling mechanism ensures that no matter how many tasks are part of the system, all of them will be executed at some point. This approach simplifies development, as developers do not need to analyze whether all tasks are guaranteed to execute. However, this design comes with a lack of strict enforcement of timing requirements, such as ensuring the periodic release of timer tasks.

For the release of tasks in ROS 2, there is no timer interrupt that checks the eligibility of tasks at every system tick. Instead, jobs are released only at polling points, regardless of the time elapsed since the last polling point during the preceding processing window. This reduces the flexibility of the schedule since fewer scheduling decisions are made. Furthermore, to guarantee the periodic release of timers, the processing window length must be less than the period of all timer tasks.

However, the default ROS 2 executor is designed to execute all tasks at some point for any system configuration, even when the processing window may be longer than the period of a timer task. For this, it provides a flexible mechanism to track and perform the release of timers. Each timer has a *timestamp*[5], which is used to determine the eligibility of the timer for release. At each polling point, the executor checks if the current time is greater than or equal to the timestamp of the timer. If so, a job of that timer is sampled and added to the wait set. Then, during the following processing window, when the timer job is selected for execution from the wait set (the start time of the timer job), the timer's timestamp is updated. To update the timestamp, it is increased by the minimal multiple of the period such that it is greater than the current time. As a result, the executor may increase the timestamp of the timer by a multiple of the period, skipping a timer job if the time between the previous timestamp and the start time of the next timer job is longer than the period of the timer. This design, while being flexible, may skip timer jobs, contrary to the expectation of a periodic release of timers.

Example 2. Again, we consider the running example from the beginning of Section 3.1.1 with three tasks. Since the tasks are time-triggered, we implement them as timer tasks with periods 10 for τ_1 and 30 for τ_2 and τ_3. The schedule obtained using the default ROS 2 executor is depicted in Figure 3.2b. In particular, the first polling point at time 0 samples a job of each timer task and puts them to the wait set. During execution of the first job of τ_1, the timestamp of τ_1 is updated to time 10. Only when the processing window finishes (at time 23), during the polling point, the executor collects the second job of τ_1 and moves it to the wait set. In the subsequent processing window, the timestamp of τ_1 is updated to 30—skipping the timestamp 20. While all jobs are executed at some point after activation, the timer jobs are not released at their specified periods. □

Dedicated response-time analyses have been developed for the default ROS 2 executor [21, 29, 110], along with suggestions to optimize the ROS 2 application code to mitigate long response times [110]. Casini et al. [29] also introduced the concept of processing chains, in which tasks may be triggered due to the arrival of data and output a result, triggering other tasks. This propagation of data creates a natural structure of chained tasks that can be used to measure the end-to-end latency of the task set. Specifically, Teper et al. have analyzed [112, 114] and optimized [113] such end-to-end latencies in the ROS 2 single-threaded executor. Multi-threaded executors have been developed and analyzed [68, 81, 104],

[5]The *timestamp* refers to the *activation time* variable of the ROS 2 codebase in the C++ client library rclcpp.

but these are still constrained by the wait set behavior of the default executor. Their implementation either shares the wait set among threads in a thread pool or runs multiple unmodified executors in different threads. The latter approach also requires the manual assignment of nodes to executors. Furthermore, the multi-threaded variant of the default executor is shown to be affected by starvation [115], i.e., it does not even ensure that all tasks will be executed at some point after activation.

Comparison of Schedulers

We now compare the classical non-preemptive priority-based scheduler with the default executor in ROS 2. First, the release mechanism in ROS 2 is substantially different from the typical release mechanism in classical real-time systems. That is, in classical real-time systems, jobs are released (almost) immediately by the timer interrupt. Contrarily, in ROS 2, activated tasks are passively sampled by the executor at polling points. Specifically, timers are not released every time their period elapses, but only when the executor samples them at the polling point, potentially causing delays in the release of jobs. Combined with the timestamp update mechanism, this can lead to the skipping of timer jobs for ROS 2, as jobs may not be sampled every period. This is depicted in Figure 3.2b, where a job of τ_1 for timestamp 20 is skipped. In contrast, the same task set under the classical scheduler with implicit deadlines would release all jobs at their specified periods, and no task would miss its deadline.

Secondly, the scheduling mechanism and the resulting delays in ROS 2 differ from the classical non-preemptive fixed-priority scheduler. In a classical non-preemptive FP scheduler, high-priority jobs can only be blocked by one lower-priority job, as a new scheduling decision is made every time a job finishes. We can see this in Figure 3.2a, where the second job of task τ_1 is blocked by only one job, the job of τ_2, and the response time of τ_1 is 6. On the other hand, in ROS 2, scheduling decisions can only be made at the polling points. That is, only once all the remaining jobs in the current processing window have finished execution, including potentially all lower-priority jobs, new jobs are added to the wait set. This case is depicted in Figure 3.2b where the second job of τ_1 is blocked by both of the jobs of τ_2 and τ_3 during $[10, 23]$, leading to a response time of 16. This may lead to higher response times for tasks in ROS 2 compared to the classical non-preemptive FP scheduler.

Thirdly, ROS 2 does not allow for the explicit setting of timing properties, such as deadlines, or the direct control of the priority of tasks to influence the scheduling order. Instead, ROS 2 enables developers to focus on the application logic and the communication between nodes, while the underlying scheduling mechanism ensures that all tasks are executed at some point after activation, even if the timing requirements, such as the timer periods, are not met.

Due to these substantial differences between the classical non-preemptive scheduler and the ROS 2 default scheduler, the rich literature of real-time scheduling theory is not directly applicable to ROS 2 systems. Therefore, alternative executors for fixed-priority [33] or dynamic-priority [6] schedulers have been developed. However, using executors provided directly by ROS 2 for periodic task systems is still desirable to reduce design and implementation costs for developers.

3.1.2 The Events Executor

In 2023, ROS 2 added the *events executor* to its distribution [65], providing an alternative to the default executor. It does not use a *wait set* but instead uses a FIFO queue, called an *events queue*, that stores the jobs of tasks that are eligible for execution. Furthermore, it uses separate threads for releasing jobs to the events queue and for scheduling jobs from the events queue. Specifically, we have a *timer management thread*, *DDS threads*, and the *default thread*. The mechanism of the events executor is depicted in Figure 3.1c.

The default thread is responsible for scheduling the jobs in the events queue. It does not release jobs, but only executes the jobs in the events queue in a FIFO order. Non-timer tasks are released by the DDS threads into the events queue. The release of non-timer tasks occurs when the DDS threads receive events, such as for the arrival of a message. Specifically, the DDS threads enqueue the events in the events queue as part of the operation that receives messages. For timer tasks, the timer management thread manages a timer release queue, and there are two options:

- Option 1: **Release-Only** (denoted **RO**). The timer management thread releases the timer jobs from the timer release queue to the events queue (where non-timer events also reside), and all jobs are scheduled by the default thread in FIFO order.

Algorithm 6 *Timer Management Thread* in Events Executor

```
1: while Running do
2:     time_to_sleep ← next timer release time - now
3:     wait_for(time_to_sleep)
4:     head_timer ← timers.front()
5:     while head_timer is eligible do
6:         head_timer.update_timestamp()
7:         if Option RO is configured then
8:             events_queue.enqueue(head_timer, data)
9:         else if Option RE is configured then
10:            execute(head_timer, data)
11:        end if
12:        Reorder timers by release time
13:        head_timer ← timers.front()
14:    end while
15: end while
```

- Option 2: **Release-and-Execute** (denoted **RE**). The timer management thread holds both released and unreleased timer jobs in its timer release queue and schedules them based on their timestamp.

Therefore, for the **RO** option, the timer management thread only releases the timer jobs to the events queue, while for the **RE** option, the timer management thread both releases and schedules the timer jobs.

Algorithm 6 summarizes the implementation of the timer management thread in ROS 2. First, in Lines 2-3, the timer management thread determines the time to sleep until the next timer release time and then waits until that time elapses. For each release, in Lines 4–5, the timer management thread first determines if the timestamp of the head timer has been reached. When the timer is released, its timestamp is updated to the next timestamp in Line 6, which is the smallest multiple of the period that is greater than the current time. Then, for the **RO** option, the timer job is released into the events queue in Line 8. Likewise, for the **RE** option, the timer job is immediately scheduled in Line 10. Afterward, in Line 12, the queue is reordered according to the new timestamps of the timers that remain in the queue.

The events executor has two properties that make it a potentially suitable option to bridge the gap between ROS 2 and classical real-time scheduling:

1. **Separation:** Different threads can be used to separate the execution and release of tasks. Specifically, as depicted in Figure 3.1c, the timer management thread and the DDS threads can be used to release jobs to the events queue, while the default thread handles the job execution. This is similar to the mechanism of the classical real-time scheduler, depicted in Figure 3.1a, where the releaser and the scheduler are separated as the releaser inserts jobs into the ready queue when they are ready and the scheduler takes ready jobs out of the ready queue for execution.

2. **Granularity:** Compared to the polling point in the default ROS 2 scheduler, depicted in Figure 3.1b, where all jobs are collected during the polling point for execution, the events executor always collects only one job for execution (either from the events queue or from the set of active timers). Therefore, this leads to a finer granularity of scheduling decisions, potentially reducing the blocking from lower-priority jobs.

3.1.3 Problem Definition

As we have discussed in Section 3.1.1, the default executor of ROS 2 is not compatible with typical real-time systems scheduling. That is, the default executor shows behavior that is typically not considered when looking at typical priority-based schedulers. More specifically, **classical priority-based schedulers** have the following properties:

P1) Each job is inserted into the ready queue (almost) immediately by the releaser.

P2) Each scheduling decision determines only one job to be executed next.

Contrarily, for the **default scheduler of ROS 2** we have:

D1) A job is inserted into a wait set only at polling points and only if the corresponding task has been activated since its preceding execution.

D2) At the polling point, the scheduler makes the scheduling decision to run all jobs in the wait set in a deterministic order and does not make another scheduling decision until the next polling point.

As a direct consequence, the default ROS 2 scheduler shows behavior that usually cannot occur in typical priority-based schedulers. For example, a job release can be skipped, and a high-priority task can be blocked by all lower-priority tasks. Therefore, the literature on classical real-time systems theory is not directly applicable to ROS 2.

We now address the problem:

> *How can the results from classical real-time systems be made applicable to ROS 2?*

To investigate this, we consider the events executor of ROS 2, which exhibits properties that are closer to traditional real-time scheduling mechanisms, making it a promising candidate for bridging the gap between these two models. To that end, we focus on timer tasks first, present our proposed scheduler based on the events executor in Section 3.1.4, and discuss the compatibility with the literature on non-preemptive schedulers for periodic tasks in Section 3.1.5. Afterward, we discuss extensions of our results to ROS 2 applications with subscription tasks in Section 3.1.6.

3.1.4 Proposed Scheduler

In this section, we present our proposed scheduler. In particular, we discuss the necessary configuration and modification of the events executor to ensure the same behavior as a classical priority-based, non-preemptive real-time scheduler.

As discussed in Section 3.1.2, the events executor has a finer scheduling granularity than the default ROS 2 executor. In particular, a new scheduling decision is made when a job finishes execution. Therefore, the events executor inherently fulfills property P2) from the problem definition. The remainder of this section is divided into two parts: First, we discuss the configuration to ensure immediate job releases, i.e., P1). Second, we detail the modifications to model different priority-based scheduling algorithms like RM, DM, or EDF.

Ensure Immediate Releases

To ensure immediate releases, we need to ensure that the release functionality is decoupled from the execution functionality and that the execution cannot block job releases. We achieve this by proper configuration of the RO/RE option and by proper prioritization of the threads.

Choice of RO/RE Option As discussed in Section 3.1.2, there are two options for the events executor: the Release-Only (RO) and the Release-and-Execute (RE) option.

The **RE** option does not separate the releaser and scheduler for timers. In this case, job releases are only performed between job executions. This can lead to timer jobs being lost if the execution time of a timer job exceeds the period of other timers. Due to these delays, the **RE** option is not suitable to ensure immediate job releases.

However, the **RO** option separates the releaser and scheduler for timers. Just like the classical real-time scheduler, this allows job releases to be handled without being blocked by job execution. Specifically, timer jobs can be released into the events queue while the default thread executes jobs. Furthermore, job releases and timestamp updates are independent of the scheduling decisions made by the default thread. Therefore, to achieve P1), *we choose the **RO** option.*

Thread Management To achieve P1), we need to ensure that the release of jobs is guaranteed even while a job is executed. For this, we need to rely on the assumption that the operating system allows scheduling of threads preemptively, meaning that the operating system can interrupt the execution of a thread at any time. Furthermore, we need to assume that threads can be prioritized if they run on the same processing core, i.e., if a lower-priority thread is executing, the operating system can preempt it in favor of a higher-priority thread.

If the threads run on *different processing cores*, the operating system can schedule them independently, and the threads can run concurrently. However, the threads share a common data structure in the events queue, and access to the data structure needs to be synchronized. Therefore, the only interference when releasing jobs occurs during access to the events queue. We consider such blocking times as part of the job release overhead and they thus can be

ignored. Therefore, the timer management thread can release jobs almost immediately, even if the default thread is executing jobs.

If the threads share the *same processing core*, the priority of the threads is crucial. To ensure jobs are released (almost) immediately, *the timer management thread and the DDS threads are configured to have a higher priority than the default thread.* In this case, the timer management thread and the DDS threads can preempt the default thread when it is executing a job, releasing the timer jobs (almost) immediately. Otherwise, if, for example, the default thread has a higher priority than the timer management thread, a long job execution by the default thread may delay the release of a timer job, potentially skipping it. Again, overheads caused by the operating system and for accessing the events queue are considered part of the job release overhead and are ignored.

Job Prioritization

Although the configuration of the previous subsection achieves P1) and P2), the job execution order in the events queue remains FIFO. Specifically, we consider the running example of three tasks, as specified in Example 2 for the ROS 2 architecture. The schedule is depicted in Figure 3.2c. While no job releases are skipped and scheduling decisions occur after every job, the schedule differs from the schedule under classical non-preemptive FP scheduling, depicted in Figure 3.2a. Specifically, while the second job of τ_1 is added to the events queue at time 10, the job is only started at time 23 because the jobs are executed in a FIFO order. Assuming implicit deadlines, this would lead to a missed deadline for τ_1. Under the classical non-preemptive FP scheduling (Figure 3.2a), the second job of τ_1 is added to the ready queue at time 10 as well, but is started already at time 13, because it has higher priority than the job of τ_3. To achieve the same schedule, we need to handle the priority ordering of the events queue.

For this, we *change the events queue to a priority queue* instead of a FIFO queue. The highest-priority job can then be selected by extracting the first job from the priority queue. The priority queue only adds an overhead of $O(\log n)$ for each priority insert, where n is the number of tasks in the system. For priority-based reordering, the overhead is $O(n \log n)$.

This overhead is inherent to priority-based management, unless special assumptions or mechanisms are introduced to avoid it. However, the overall overhead is negligible compared to that of the other ROS 2 management routines.

For sorting the events queue in ROS 2, we additionally require information about the priority of the jobs. By default, ROS 2 does not provide a priority field for tasks. In the following, we discuss two options to provide the priority of the timer jobs, with different levels of user interaction required. We focus on RM and EDF scheduling of timer tasks. Possible extensions for subscription tasks can be found in Section 3.1.6.

The *first option* uses each timer's period as its priority. Each timer task already provides a period, required when creating the timer. Thus, this information can be used during scheduling to prioritize the timer jobs in the events queue. For RM scheduling, the timer job with the smallest period has the highest priority. Therefore, when inserting a timer job into the events queue, the queue can access the period of the timer and insert the job based on the period. For implicit-deadline EDF scheduling, we can use the next timestamp of the timer as the priority of the timer job. When inserting a timer job into the events queue, the queue can access the next timestamp of the timer and insert the job based on the timestamp.

As a *second option*, we propose to add a user-defined priority field to the timer structure. It can either hold static priorities for static-priority scheduling or relative deadlines for dynamic-priority scheduling. The approach theoretically supports any job-level fixed-priority non-preemptive scheduler if the priority can be decided when the job arrives, e.g., the non-preemptive rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling policies for uniprocessor systems. While this option is less user-friendly, as it requires users to change the current code of their system to provide the priority of the timer jobs, it is a more versatile approach and supports any job-level priority ordering. We provide both options to enable priority-based scheduling in ROS 2.

After prioritization of the events queue using one of the two options above, the corresponding schedule for the running example is depicted in Figure 3.2d. Specifically, at time 10, the second job of task τ_1 is inserted into the events queue with the highest priority and is chosen to be scheduled at time 13. The corresponding schedule coincides with the schedule under classical non-preemptive FP scheduling from Figure 3.2a. To conclude, our changes enable the events executor to be used for arbitrary fixed-priority or dynamic-priority scheduling

policies for non-preemptive scheduling. Thus, our proposed design is compatible with the classical real-time scheduling theory, as we discuss further in Section 3.1.5.

3.1.5 Compatibility with Non-Preemptive Schedulers for Periodic Tasks

In this section, we explain how our proposed scheduler from Section 3.1.4 bridges the gap between the literature on priority-based scheduling and ROS 2. To that end, we show that our scheduler behaves analytically like a non-preemptive work-conserving priority-based scheduler We discuss the overhead that results from our scheduler. We then specify which analytical results apply to our scheduler, focusing on the worst-case response time and end-to-end latency (which is one of the predominantly studied metrics for ROS 2 systems).

For the discussion in this section, we assume that the ROS 2 application has only timers and the system is exclusively used to execute the ROS 2 application on one processor. An extension to incorporate subscriptions can be found in Section 3.1.6. We assume that tasks have different priorities and ties are broken arbitrarily but deterministically.

Analytical Behavior of Our Scheduler

Our solution uses the events executor with a modification of the events queue for priority-based job ordering. The release of timer jobs into the events queue is done in the *timer management thread*, which is given a higher priority than the default thread. Therefore, whenever the *timestamp* of the timer task is reached, a job of that task is inserted into the events queue immediately[6]. Hence, jobs are released (i.e., inserted into the events queue) periodically. Moreover, scheduling decisions are made whenever a job finishes by pulling the first job (i.e., the highest priority job) from the events queue. In addition, when the executor idles and a new job is released, it makes a scheduling decision immediately and starts executing the job. Therefore, our scheduler behaves like a typical non-preemptive work-conserving scheduler.

[6]Please note that our solution only works under the assumption that the *timer management thread* is not blocked.

Bounding the Releaser Overhead

Let $\delta > 0$ be the maximal time for releasing a job. Then, if a job is released into the events queue, this prolongs the execution time of the job that is currently running. Hence, by counting the maximal number of releases during the execution of a job, we obtain a bound on the overhead. We denote by Δ_i the overhead that jobs of τ_i can experience.

With our solution, a job is released whenever a timestamp is reached. The number of timestamps that can occur for a task τ_j during an interval of length t is $\left\lceil \frac{t}{T_j} \right\rceil$. Let $t_0 \in \mathbb{R}_{>0}$ be the lowest positive real number such that $t_0 \geq C_i + \sum_{j=1}^{n} \left\lceil \frac{t_0}{T_j} \right\rceil \cdot \delta$. Then t_0 is the amount of time that a job of τ_i can execute, including the overhead from releasing other tasks. Hence, the release overhead for task τ_i is bounded by

$$\Delta_i \leq \sum_{j=1}^{n} \left\lceil \frac{t_0}{T_j} \right\rceil \cdot \delta. \tag{3.1}$$

This overhead has to be accounted for when applying the results in the literature for non-preemptive schedulers by prolonging the worst-case execution time value C_i by Δ_i.

This bound on the overhead Δ_i can be tightened further if the following two conditions hold:

- The task set has constrained deadlines.
- The task set is already shown to be schedulable (e.g., by using the bound on the overhead from Equation (3.1) together with a schedulability test that is referenced in Section 3.1.5).

Under these conditions, every other task can only release one job during the execution of a job of τ_i. Otherwise, assuming a task τ_j releases two jobs at timestamps t_1 and $t_2 = t_1 + T_J$ during the execution of a job of τ_i, then the job of τ_j cannot be executed until time t_2. Hence, it would miss its constrained deadline, which violates the schedulability of the task set. Therefore, under the two conditions, the overhead for the releaser is upper bounded by

$$\Delta_i \leq n \cdot \delta, \tag{3.2}$$

where n is the number of tasks in the system.

We note that it is unsafe to apply the tightened bound from Equation (3.2) directly for calculating a response time bound and checking the schedulability based on that. However, when the response time is already shown to be less than the deadline D_i, we can use this formula to tighten the response time bound further. This can be beneficial when applying analyses that utilize the worst-case response time (cf. Section 3.1.5).

Analytical Results

As detailed above, our proposed scheduler behaves analytically like a typical non-preemptive scheduler. Although this pertains to any scheduling algorithm that can be modeled by reordering the events queue, as discussed in Section 3.1.4, we focus specifically on FP and EDF scheduling. For this section, we assume that the overhead is accounted for by prolonging the WCET, i.e., by redefining $C_i := C_i + \Delta_i$.

Worst-Case Response Time The worst-case response time of a task is the maximal time between the release and finish of any job of that task. With our proposed scheduler, the rich literature on non-preemptive scheduling for periodic tasks becomes applicable (cf. [40, 50, 67, 91, 118]). Note that this also encompasses the literature on non-preemptive sporadic tasks by setting the minimum inter-arrival time to the task period. Especially, results for non-preemptive FP scheduling can be found in [40, 50, 91, 118] and results for non-preemptive EDF scheduling can be found in [50, 67, 91]. With our scheduler, *any* of these analyses can be applied to derive a bound on the worst-case response time. We note that when a task set is shown to be schedulable (for example by any of the above analyses), then we can also claim an upper bound on the worst-case response time, namely $R_i \leq D_i$.

End-To-End Latency One of the predominantly studied metrics for ROS 2 applications is end-to-end latency. That is, given a so-called *cause-effect chain*, which is a sequence of tasks $E = (\tau_{i_1} \rightarrow \dots \tau_{i_N})$, with $i_j \in \{1, \dots, n\}$ being a task index for all $j = 1, \dots, n$, then we are interested in the amount of time that data needs to traverse the cause-effect chain. Specifically, the Maximum Reaction Time (MRT), i.e., the amount of time until an external cause is fully processed, and the Maximum Data Age (MDA), i.e., the age of data utilized in an actuation, have been analyzed extensively in the literature. Recently, it has been shown

that MRT and MDA are equivalent [54]. Hence, we use *Lat* to denote the end-to-end latency of a cause-effect chain in general.

While there are only a few analytical results for the standard ROS 2 application [112, 114], there is a large body of literature results for typical periodic tasks [11, 12, 19, 39, 43, 51–55, 57, 72]. Our scheduler enables such results and insights from end-to-end analysis for periodic task systems.

A classical bound, proposed by Davare et al. [39], is to bound the end-to-end latency by summing up the task periods T_{i_j} and worst-case response times R_{i_j}:

$$Lat \leq \sum_{j=1}^{N} \left(T_{i_j} + R_{i_j} \right) \tag{3.3}$$

Here, T_{i_j} is the period of the j-th task in the cause-effect chain, and R_{i_j} is the worst-case response time of the j-th task in the cause-effect chain, i.e., of task τ_{i_j}. The bound assumes that tasks communicate via the *implicit communication* principle [57], i.e., jobs read data when they start execution and write data when they finish execution. While the original bound is proven only for *preemptive fixed-priority* scheduling in [39], this result is universally applicable when utilizing a correct bound on the worst-case response time. This is proven for example as a byproduct of the probabilistic analysis by Günzel et al. [56], i.e., their reduction to the case with deterministic worst-case response time bound in Note 6.3 of [56] proves Equation (3.3) without assumptions on the scheduling algorithm.

To summarize, given a cause-effect chain E on the task set $\mathbb{T}$ scheduled by our scheduler, then the following steps result in a bound on the end-to-end latency:

1. Calculating the overhead Δ_i from Section 3.1.5 and incorporating it in the WCET by $C_i := C_i + \Delta_i$.
2. Providing an upper bound on the worst-case response time R_i using literature results for non-preemptive periodic tasks.
3. Calculating a bound for the end-to-end latency using Equation (3.3).

In Section 3.1.7, we compare this bound on the end-to-end latency with the latency that can be analytically guaranteed for the typical ROS 2 scheduler [114].

3.1.6 Subscription Tasks

The previous sections focused on the behavior of timer tasks. However, besides timers, ROS 2 allows tasks to be triggered through the built-in DDS layer. Since such subscription tasks are widely used in practice, this section explains how our findings can be extended to systems involving subscription tasks.

First, we observe that our proposed scheduler from Section 3.1.4 still fulfills P1) and P2) even for subscription tasks. That is, by giving the DDS threads higher priority than the default thread or by running them on a separate core, the release of subscription tasks is not blocked by job execution, and a new scheduling decision is made after every job.

We are left with the problem that subscription tasks cannot be modeled and analyzed as periodic tasks because they are not inherently time-triggered (unlike timer tasks). Instead, the timing behavior of subscriptions is dependent on the tasks that publish to the subscription's topic. Furthermore, there is no direct parameter to set the priority of a subscription task. In the following, we present two approaches to model and analyze subscription tasks: first, a sporadic task model that applies to any system structure and second, a limited-preemptive scheduling approach can be used for systems with a specific structure to obtain potentially tighter bounds. Afterward, we discuss solutions for the prioritization of subscription tasks.

Modeling as Sporadic Tasks

To analyze subscriptions as sporadic tasks, we need to assume that subscription tasks have a minimum inter-arrival time > 0, meaning that no two jobs of the same subscription task are released within this time frame. If such a minimum inter-arrival time can be identified, then we can analyze them as sporadic tasks. For example, if tasks are assumed to only publish once at the end of their execution, the minimum inter-arrival time can be bounded by the minimum best-case execution time among all tasks that publish to the subscription topic, for single-executor systems. While this general approach to deriving a minimum inter-arrival time can be quite pessimistic, more dedicated analyses for specific task sets can potentially derive much tighter bounds on the minimum inter-arrival time.

Please note that, as in Section 3.1.5, we also need to account for the overhead from the releaser. For this, we prolong the WCET of the subscription tasks. Given a minimum inter-arrival time T_i for a subscription task τ_i, the overhead Δ_i is bounded by Equation (3.1).

Modeling as Limited-Preemptive Tasks

For a special case, where tasks are arranged in sequences, the analysis for limited-preemptive tasks is applicable. That is, we consider the case that each task invokes the activation of at most one subscription task, and each subscription task can be activated by exactly one task. Furthermore, we assume that the first task of a sequence is a timer task, to achieve time-triggered behavior, and all subsequent tasks are subscription tasks since they subscribe to a topic by definition. An example of such sequences can look as follows:

- $\text{sequence}_1 = \text{timer}_1 \rightarrow \text{subscription}_1 \rightarrow \text{subscription}_2$
- $\text{sequence}_2 = \text{timer}_2 \rightarrow \text{subscription}_3 \rightarrow \text{subscription}_4$

Each job of the timer task in a sequence releases one job of the subsequent subscription tasks. Furthermore, we assume no delay is introduced by the DDS between releasing a subscription task and activating the downstream task, preventing lower-priority jobs from being started before the subscription task's release. For this, ROS 2 provides the option to use synchronous DDS communication or zero-copy communication.

With these assumptions, the literature on limited preemptive scheduling, e.g., [17, 28, 126], is applicable, where each sequence is modeled as one 'task', with each timer task and subscription task being a 'subtask'. Each subtask is scheduled non-preemptively, and between subtasks, the task can be preempted. The benefit of considering sequences is that there is no need to determine (potentially pessimistic) minimum inter-arrival times of subscription tasks.

We further note that, in case the DDS introduces a certain delay in releasing the subsequent subscription [78], another lower-priority job may be started in between the execution of two non-preemptive subtasks. In that case, the additional delay can be modeled as self-suspension [31]. More specifically, the set of sequences behaves like a set of segmented self-suspending tasks scheduled non-preemptively on a single core.

Prioritization of Subscription Tasks

For both models, a prioritization of the subscription tasks is necessary. Currently, there is no direct way to set the priority of a subscription task in ROS 2. However, we propose two potential solutions to prioritize subscription tasks.

- The first one is to utilize the ROS 2 DDS and its *Quality-of-Service* (QoS) settings to configure parameters about the communication behavior. Specifically, each subscription has one topic that it subscribes to, and each topic has a QoS setting. Currently, one QoS parameter for topics is the so-called *deadline* [60], which for topics describes the expected maximum amount of time between subsequent messages being published to the topic. To integrate the minimum inter-arrival time, we propose to introduce a QoS parameter that reflects the expected minimum amount of time between subsequent messages being published to the topic. For the final integration into the scheduling decisions, changes to the executor are necessary that would consider the DDS QoS parameters, such as the *minimum inter-arrival time*.
- For the second one, we propose to add a priority field to subscriptions, which can be set by the user. This approach is more direct and flexible but requires changes to the ROS 2 codebase and the application code. Given this priority field, the same procedure as for the timer jobs mentioned in Section 3.1.4 can be applied for both static and dynamic priority assignments.

In both cases, it would be beneficial to have interfaces to dynamically set the priority comparison function of an executor. This would maximize compatibility, ensuring that existing executors can still be used, while other scheduling policies can be implemented just as easily.

In any case, the priority queue that we introduced in Section 3.1.4 to replace the events queue can be used to handle both timer jobs and subscription jobs. Moreover, there are no further modifications required for handling subscriptions.

3.1.7 Evaluation

In this section, we evaluate our proposed modifications of the ROS 2 events executor and the corresponding analyses. The evaluation consists of three contributions:

- We confirm **compatibility with scheduling theory** on non-preemptive schedulers for periodic tasks. More specifically, we demonstrate for a timer-only task set that the analytical results of Section 3.1.5 are applicable. That is, the modified ROS 2 events executor does not drop jobs, and it respects the worst-case response time bounds for non-preemptive scheduling presented in Section 3.1.5.
- We show that the **bounds on end-to-end latency** for cause-effect chains, enabled by the theory of non-preemptive scheduling bounds in Section 3.1.5, are tighter than the state-of-the-art analytical end-to-end latency bounds for the default executor. To that end, we apply the analyses to synthetic task sets using the WATERS [77] benchmark.
- We compare our modified executor using the RM scheduling policy with the default executor by assessing their **Performance in the Autoware reference system**. The Autoware reference system includes both timers and subscriptions, and demonstrates that we can achieve lower end-to-end latency than the other executor options that are provided natively by ROS 2.

Our experiments feature the following executor configurations:

- default executor (**Default**),
- static default single-threaded Executor (**Static**),
- unmodified default events executor (**Events**),
- our rate-monotonic events executor (**RM**), and
- our earliest-deadline-first events executor (**EDF**).

We include the static default executor for completeness. It behaves like the default executor, but with less overhead, as it does not have to handle dynamic system configurations. We also provide an artifact[7] of our evaluation.

[7] https://github.com/tu-dortmund-ls12-rt/ros2_executor_evaluations

Compatibility with Scheduling Theory

In this subsection, we demonstrate that existing results on non-preemptive scheduling apply to our scheduler, and analytical results correctly predict the behavior of the system.

To that end, we examine a timer-only task set that consists of three types of components, and seven nodes in total. The system has four cameras, two LiDARs, and an IMU. Each node has one timer task, and each component is independent of the others, meaning that they do not pass data between each other and have no precedence constraints. By varying the maximum execution time of the tasks, we configure the system to have a utilization of 60%, 80%, or 90%. Our three configurations are as follows:

60% utilization

- 4 Camera Nodes: 84 ms period, 10 ms execution each
- 2 LiDAR Nodes: 200 ms period, 10 ms execution each
- 1 IMU Node, 30 ms period, 1 ms execution

80% utilization

- 4 Camera Nodes: 84 ms period, 14 ms execution each
- 2 LiDAR Nodes: 200 ms period, 10 ms execution each
- 1 IMU Node, 30 ms period, 1 ms execution

90% utilization

- 4 Camera Nodes: 84 ms period, 16 ms execution each
- 2 LiDAR Nodes: 200 ms period, 10 ms execution each
- 1 IMU Node, 30 ms period, 1 ms execution

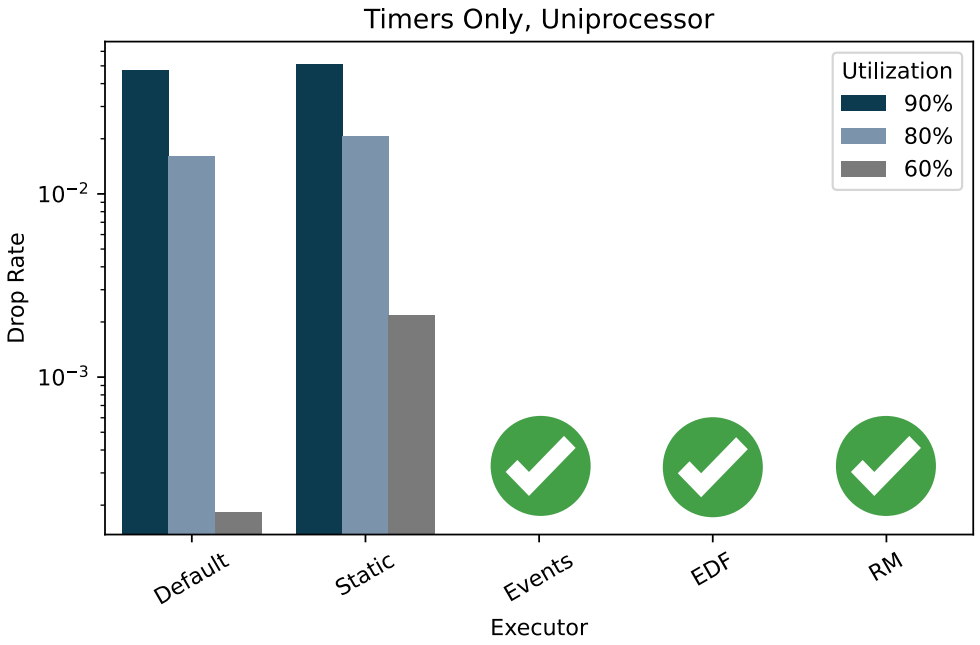


Figure 3.3: Dropped jobs on timer-only task sets: A checkmark indicates no dropped jobs across all utilization levels.

Task periods stay consistent across all utilization levels, so all task sets have the same hyperperiod of 4.2 s. Each task set is run for 5 min, elapsing over 70 hyperperiods. We repeat this setting for each executor design.

Each of these task sets is pinned to a single core of a Raspberry Pi Model 4B with 4GB of RAM running Ubuntu 22.04 and ROS 2 Humble. The first experiment (Section 3.1.7) uses one core, the second experiment (Section 3.1.7) assumes that the executor runs on one processing core, and the third experiment (Section 3.1.7) uses two cores with no thread pinning.

Number of Dropped Jobs: The rate of dropped jobs for different configurations is shown in Figure 3.3. We can see that the default and static executors repeatedly drop jobs. In some cases, a drop rate of 5% is observed. Confirming our analytical results of Section 3.1.5, which indicate no job drops, our proposed events executors do not have any job drops for any configuration of the RM and EDF events executors.

Table 3.1: Analytical worst-case response times (ms) of our RM events executor.

Utilization	60%	80%	90%	Period
IMU	12.67	16.67	18.67	30
Camera	57.83	75.66	83.66	84
LiDAR	70.50	149.50	167.33	200

Response Time Bounds: In the following, we analytically derive upper bounds on the worst-case response time, following Section 3.1.5, and then confirm that the response time bounds are not violated by our RM events executor.

For the analysis, we measured that a job release takes up to $\delta = 0.12\,\mu s$. The total overhead Δ_i for a timer task τ_i can be determined by Equation (3.1). Doing the calculations for the task sets of this section, every timer task has an overhead of $\Delta_i = 0.84\,ms$. For the analysis, we add this overhead to the WCET of each task, i.e., $C_i := C_i + \Delta_i$.

Afterward, we apply Equation 6 from von der Brüggen et al. [118] to determine the worst-case response time of each task.

Theorem 1 (Non-preemptive FP, reformulated from [118])**.** *Assume that the tasks* $\mathbb{T} = \{\tau_1, \cdots, \tau_n\}$ *are ordered by their priority, i.e.,* τ_1 *has the highest priority and* τ_n *has the lowest priority. If there exists a* $t \geq 0 \in \mathbb{R}$ *such that* $t \leq D_k$ *and*

$$t \geq C_k + \max_{i>k} C_i + \sum_{i<k} \left\lceil \frac{t}{T_i} \right\rceil C_i, \tag{3.4}$$

then the worst-case response time R_k *of* $\tau_k \in \mathcal{T}$ *under non-preemptive FP scheduling is upper bounded by* t*.*

We consider implicit-deadline task systems, i.e., the relative deadline of a timer task is equal to its timer period. The analyzed worst-case response times for our RM events executor are reported in Table 3.1, indicating no deadline misses. Thus, we should not observe any job drops, as verified in Figure 3.3, and all tasks are guaranteed to meet their deadlines.

We then measured the response times during the experiment. The aggregated results of the *Camera* nodes are shown in Figure 3.4a, the LiDAR nodes in Figure 3.4b, and the IMU node in Figure 3.4c. As shown in the figures, the response times of our work are consistent with the analytical results, and no deadline overruns are observed. Furthermore, in Figure 3.4c, we

see that the IMU tasks have deadline overruns on the Default, Static, and Events executors. We note that Figures 3.4a, 3.4b, and 3.4c only account for the response times of jobs that are not dropped. Hence, for a significant amount of timestamps, the *Default* executor and *Static* executor do not respond within the deadline, as can be observed in Figure 3.3.

In conclusion, our analysis not only correctly indicates the absence of dropped jobs in our RM events executor, but also a correct upper bound on the response time of each task.

Tighter End-to-End Bounds for ROS 2

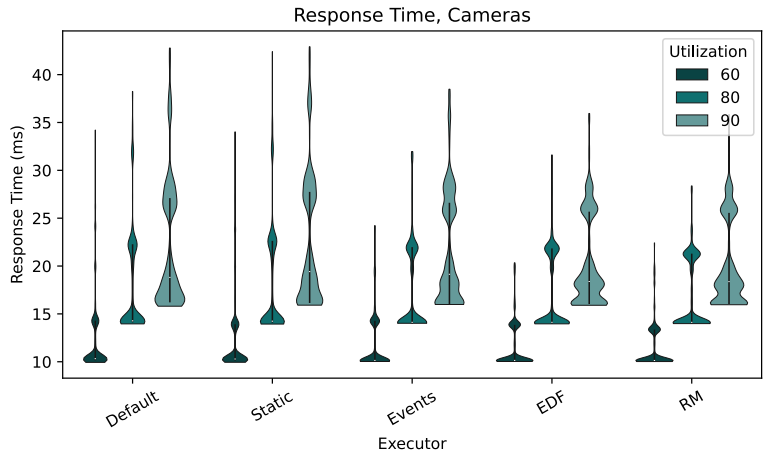
We now compare the analytically derived upper bounds of the end-to-end latencies between the default ROS 2 executor and our RM events executor. For this experiment, we generate synthetic task sets using the WATERS benchmark [77] for automotive systems.

We evaluate utilization levels of 60%, 80%, and 90%, generating one thousand task sets per configuration with 10 to 200 tasks. To compute end-to-end latencies, we generate 5 to 60 task chains, each with 2 to 15 tasks. We assume all tasks are assigned to one events executor running on a single core.

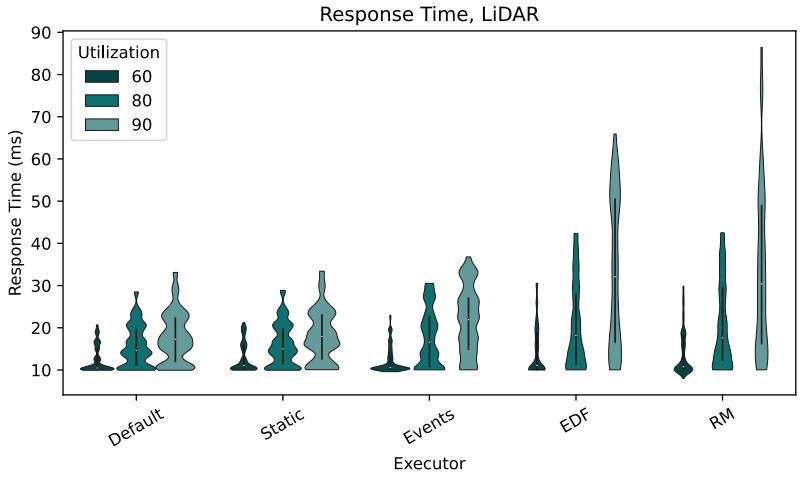
For each task set, we calculate the end-to-end latency for the default ROS 2 executor and our RM (including both RO- and RE-options) events executor, as presented in Section 3.1.5. Specifically, we use Equation (3.3) in Section 3.1.5 to calculate the end-to-end latency of each chain.

The end-to-end latency of the default ROS 2 executor has been analyzed by Teper et al. [112]. We use Equation (15) of Lemma VI.1 and Equation (17) of Lemma VI.2 from [112] to get an upper bound on the latency of each timer. For each chain, we sum up the latency upper bounds of all chain tasks to get the end-to-end latency of the chain.

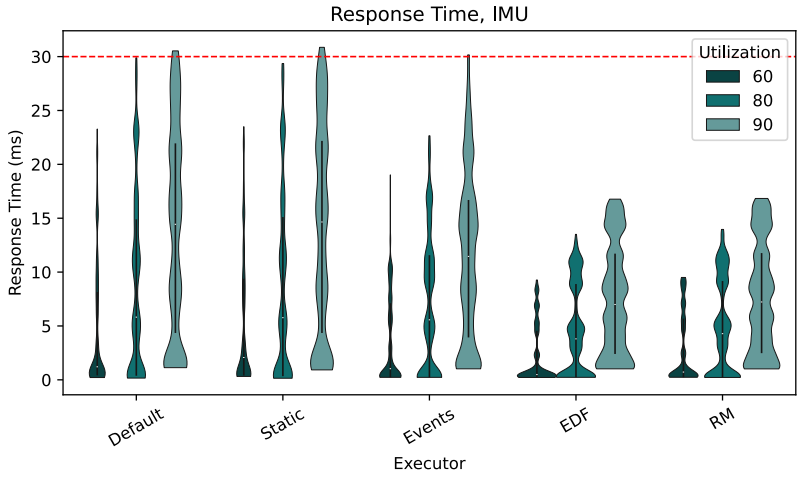
For each chain, we calculate the normalized reduction of the end-to-end latency by using $\frac{E_{default}-E_{RM}}{E_{default}}$, where $E_{default}$ (E_{RM}, respectively) is the end-to-end latency of the chain under the default executor (our RM events executor, respectively). The results of the evaluation are shown in Figure 3.5, where the x-axis is the normalized reduction and the y-axis is the number of chains within the binned normalized reduction.



(a) Response Time of Camera Tasks (ms), D=84 ms



(b) Response Time of LiDAR Tasks (ms), D=200 ms



(c) Response Time of IMU Tasks (ms), D=30 ms

Figure 3.4: Response Time of Tasks (ms)

The end-to-end latencies of the chains are typically much lower when using our RM events executor. As the utilization level increases, the number of chains with higher end-to-end latencies increases as well. This is because the default ROS 2 executor treats all callbacks fairly, disregarding their priorities. Meanwhile, static-priority scheduling assigns some callbacks lower priorities, increasing their response times (see Figure 3.4b) and resulting in a negative reduction of end-to-end latencies for chains that include such tasks. However, the majority of chains have lower end-to-end latency when using the RM events executor. In some cases, we observe a normalized reduction of the end-to-end latency bound of almost 90%. Our experiment shows that our design can be applied to different kinds of task sets and potentially leads to latency improvements.

Performance of Autoware Reference System

We also evaluate end-to-end latencies for the Autoware reference system, detailed in [100], which includes interconnected timers and subscriptions [1]. The Autoware reference benchmark runs a simulated version of the Autoware application on a Raspberry Pi Model 4B running Ubuntu 22.04 and ROS 2 Humble. We run the ROS 2 Autoware benchmark on two dedicated cores, while the other two cores are handling the remaining services of the operating system. Specifically, we measure the end-to-end latency of the **hot path** defined in the benchmark from the front and rear LiDAR sensors to the object collision estimator (c.f. the Autoware reference system), as this latency is a key metric for responsiveness in crash avoidance. The data at the start of the hot path that is generated by the LiDARs is produced at a frequency of 10 Hz.

For our RM events executor, the subscription tasks inherit the highest priority from their corresponding upstream publishers. Non-preemptive EDF in this scenario is not well specified, as subscription tasks do not have any assigned deadlines, and hence is not part of this evaluation.

Figure 3.6 shows the violin plot for the latency of the hot path in the Autoware reference system from a test run of 600 seconds. Of the executors measured, our RM events executor has the best worst-case latency, significantly outperforming the other executors. Compared to our work, the default ROS 2 executor has a 2.6x slowdown along the hot path. Our evaluation shows that our executor design using existing scheduling policies from classical

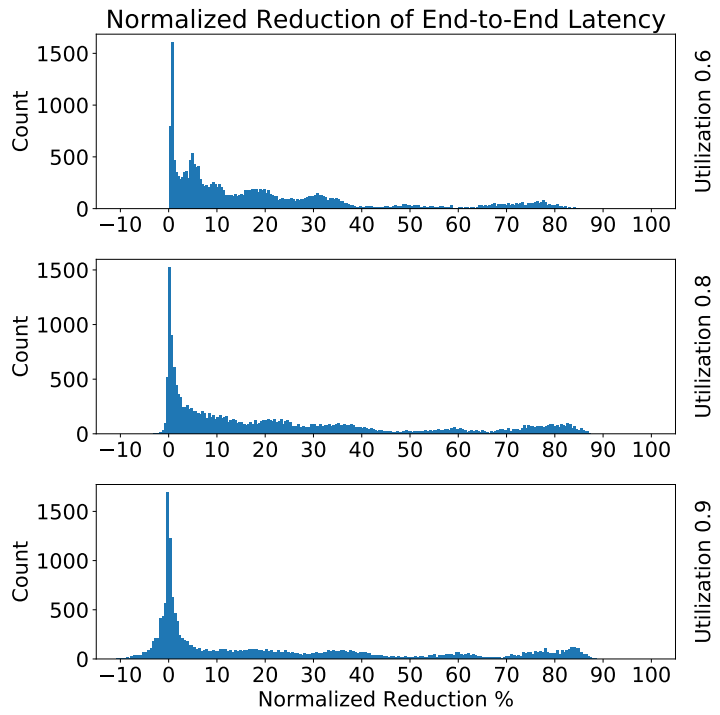


Figure 3.5: Reduction of the end-to-end latency between the default ROS 2 executor and our RM events executor.

real-time systems can be applied to practical ROS 2 applications and can provide major benefits in terms of end-to-end latency.

3.1.8 Conclusion

We have conducted in-depth investigations of the recently introduced (in 2023) events executor in ROS 2 to provide compatibility with the classical real-time scheduling theory of periodic task systems. Specifically, our solution is easy to integrate into existing ROS 2 systems since it requires only minor modifications of the events executor that is already natively included in ROS 2. Thus, it also can coexist with available ROS 2 executors, such as the default executor.

Our study enables the rich literature of the real-time scheduling theory for non-preemptive schedules to apply to ROS 2, under the assumption that an individual core is exclusively

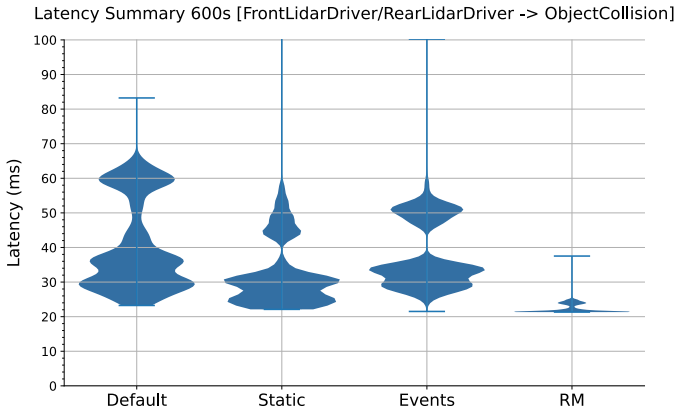


Figure 3.6: End-to-end latency of the hot path in the Autoware reference system under different executors.

assigned to our executor. Furthermore, we intensively validate the feasibility of our modifications for non-preemptive RM and non-preemptive EDF in the ROS 2 events executor for several scenarios. The evaluation results show that with minor modifications our ROS 2 events executor can offer significant improvement in terms of dropped jobs, worst-case response time, end-to-end latency, and performance, in comparison to the default ROS 2 executors that are available.

We note that our emphasis is on the compatibility of the ROS 2 events executor with the classical scheduling theory of periodic tasks. As a result, we mainly focus on the transformation of a timer into a periodic task. We highlight that event-triggered tasks require dedicated analyses, but also provide methods to analyze some specific system configurations using existing analyses on limited preemptive scheduling. Therefore, in future work, we will further investigate the interaction of the built-in DDS and the events executor to provide more insights when both timers and subscriptions are present and interact with each other.

Finally, to be compatible with event-triggered tasks in ROS 2, such as subscriptions, we propose to add dedicated interfaces to ROS 2 to specify timing properties, which are currently missing. Towards this, we discuss two potential solutions, either involving DDS QoS settings or introducing a dedicated priority field. These additions would enable the integration of many scheduling policies into ROS 2 and provide compatibility for event-triggered tasks.

3.2 Graph-Aware Scheduling in ROS 2

This section continues to address the limitations of current scheduling methods in the ROS 2, but focusing on scheduling tasks beyond simple chains and analyzing arbitrary graphs. While previous research has mostly focused on chain-based scheduling with ad-hoc response time analyses, we propose a novel approach using the events executor [65] to implement fixed-job-level-priority schedulers for arbitrary ROS 2 graphs on uniprocessor systems.

While the work in the first half of this chapter [111] laid important groundwork and demonstrated the potential for chains, it did not provide a concrete mechanism or analysis for applying standard priority-based scheduling to general ROS 2 *graphs* (involving branching and merging).

Contributions:

In this section, we (i) provide a mapping showing that an arbitrary ROS 2 graph can be modelled as a set of DAG tasks for analysis purposes (Section 3.2.1), including showing that ROS 2 graphs are logically trees for the purposes of scheduling (Section 3.2.3); (ii) introduce an events queue implementation for the Events Executor that can realize any LP-FJP scheduler for the resulting DAG tasks (Section 3.2.5), noting that our approach relies on LIFO-ordered message queues which are not yet standard in ROS 2 middleware; and (iii) provide a formal analysis proving the correctness of this events queue mechanism (Section 3.2.6).

This work further closes the gap between established real-time systems theory and practical ROS 2 scheduling analysis, moving beyond the ad-hoc approaches and chain-structured applications of existing literature.

3.2.1 System Model

In this section, we discuss the unique manner in which ROS 2 schedules callbacks, and the assumptions we must make before mapping applications to a conventional DAG task model, detailed next.

Task Model

Given a set $\mathbb{T} = \{\tau_1, \tau_2, ..., \tau_n\}$ of tasks, a **periodic DAG task** τ_i is specified by the tuple $\tau_i = (T_i, D_i, \phi_i) \in \mathbb{R}^3$, where $T_i > 0$ is the period, $D_i > 0$ is the relative deadline, and ϕ_i is the phase. The periodic task τ_i releases its first job (task instance) at time ϕ_i, and subsequent jobs are released every T_i time units. Every job has an absolute deadline specified as its release time plus the relative deadline.

A **sporadic DAG task** τ_i is similarly defined as the tuple $\tau_i = (T_i, D_i) \in \mathbb{R}^2$, where T_i is instead the minimum interarrival time between two successive jobs. The *periodic task model* is a specialization of the *sporadic task model*.

Each DAG task is associated with a graph $G_i = (V_i, E_i)$. The nodes in the finite set $V_i = \{v_{i1}, v_{i2}, ...\}$ model non-preemptible portions of work, or *subtasks*, in the task τ_i. Each subtask v_{ij} has a worst-case execution time (WCET) w_{ij} and may have precedence constraints modeled by the relations in the finite set $E_i = \{e_{i1}, e_{i2}, ...\}$. A subtask is released when all of its precedence constraints (if any exist) are met. We define a precedence relation as $v_{ij} \prec v_{ik}$ and say that v_{ij} is a **parent** of v_{ik}.

Each subtask v_{ij} has an associated amount of work, W_{ij}, which is the WCET the subtask needs to complete. We say a **job** of task τ_i is complete when one instance each of $v_{ij} \in V_i$ are complete. We focus specifically on DAG task schedulers with *limited preemption* and *fixed job-level priority*.

Limited preemption (LP) means that individual subtasks cannot be preempted, but a task can be preempted in between constituent subtasks. We take this approach to conform to the natural behavior of ROS 2 executors, which will not preempt running executables. Thus, the only possible preemption points are at the completion of subtasks.

Fixed job-level priority (FJP) means that a task and all its constituent subtasks have the same priority for a given job instance. The priority between two job releases of the same task may vary. All fixed-priority schedulers are naturally also fixed for individual jobs. Earliest Deadline First (EDF) is an example of a dynamic-priority scheduler whose priorities are fixed for individual jobs, i.e., it is FJP.

3.2.2 Mapping ROS 2 to DAG Task Model

As in prior literature [29, 33, 110], we abstract the dataflow paths of a ROS 2 application into a graph structure. To illustrate, consider an example ROS 2 application with a couple sensor nodes that take periodic measurements, and publish their data to topics subscribed by downstream computation nodes, like sensor fusion, mapping, planning, etc. The sensors are time triggered and so associated with timers. All data-driven tasks are associated with subscriptions.

Within the graph structure, each callback (timer, subscription, etc) forms a subtask, which may be released periodically (in the case of timers), or data-driven (in the case of subscriptions). During the course of a its execution, a subtask τ_i may publish a message to a topic, and any subscribers to that topic, τ_j will be modeled as children of τ_i.

We assume that ROS 2 applications do not have such loops, enabling the connection to DAGs. Given no restrictions on the system design, it is possible for subscriptions to send messages to themselves or form a loop in which data is propagated from one subscription through other subscriptions and back to itself. But if a subscription is allowed to continually trigger itself, it creates a positive feedback loop of exploding message bandwidth, so this is strongly discouraged by good ROS 2 design principles. After establishing these precedence constraints, these subtasks can then be arranged into *DAG tasks*. In the next section, we will demonstrate that they have a tree-based structure, and so a singular subtask can be described as the root. These *DAG tasks* are modeled as *periodic tasks* when a timer is at the root, and externally-driven *sporadic tasks* when a subscription is at the root.

In the existing DAG scheduling literature, the term *node* is synonymous with *subtask* [27, 47, 92, 102, 117]. To avoid confusion, we will strictly use the ROS 2 definition of *node*, which is an object that serves as a programmatical abstraction to bundle executable entities under a unified namespace. *Subtask* will be used exclusively to refer to callbacks in the context of scheduling.

We only concern ourselves with dataflow that occurs through the DDS, a publish-subscribe communication middleware used in a ROS 2 based system. Any two subtasks that communicate with shared memory (as is the case with some *fusion node*[8] implementations) are not necessarily modeled as being in the same task. A precedence constraint is only imposed when the execution of one subtask triggers the release of another. Any subtasks that exchange information asynchronously, bypassing the pub/sub mechanism, are not modelled by our work.

Code within callbacks can be fairly arbitrary. In order for the analysis of a ROS 2 application to be feasible, we must impose some constraints on its behavior: (1) children are eligible for release as soon as parents finish, so no inter-subtask latency needs to be modeled; (2) no new threads are created; (3) no blocking on IO reads occur; and (4) more generally, all execution times of callbacks are bounded.

Conventional real-time schedulers stipulate that child subtasks are eligible for execution when their parents complete, and thus imposing this requirement is needed to comply with the analysis of a LP-FJP scheduler. We restrict the callback's ability to spawn new threads, since the scope of this work is limited to the uniprocessor case. According to the model we established in 3.2.1, subtasks have a finite WCET. Therefore anything capable of an unbounded delay is not allowed.

One of the most unique aspects of modeling ROS 2 applications as a DAG task is that precedence constraints are logically disjunctive: a subtask will execute a job for *each* execution of *each* of its parents. This is not in line with the conventional DAG task model we laid out above, until we apply the abstraction of virtually unfolding the graph into a forest of trees, described next.

3.2.3 Graph Unfolding

We now show that ROS 2 applications that can be modeled as DAGs can be converted to trees due to the logically disjunctive precedence constraints. In conventional DAGs, subtasks will wait for all of their parents before being eligible for execution. In ROS 2,

[8]A fusion node is a component with a single output that combines data from multiple inputs. Unlike many-to-one topics, the output topic operates at an independent rate, or a rate matching only one of the inputs.

however, when multiple parents publish to the same subtask, the child will execute once for each parent event. To model this behavior *for analysis purposes*, we adapt the concept of graph unfolding [22] where we virtually duplicate each subtask so that each resulting virtual subtask instance has exactly one parent.

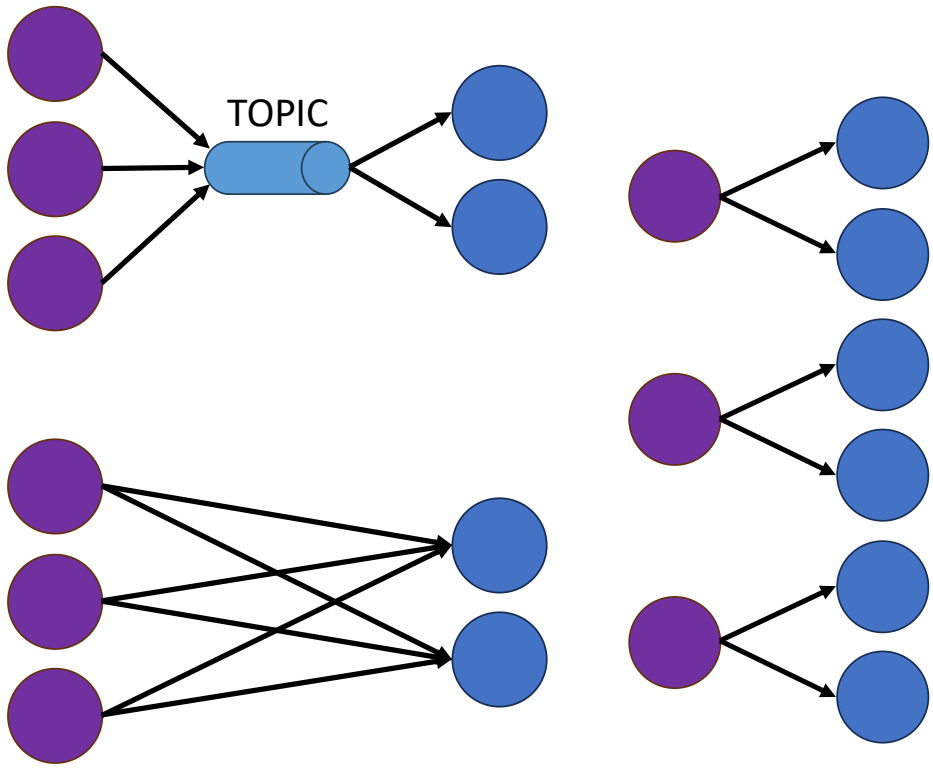
This transformation is illustrated in Figure 3.7. This creates a tree (or rather a forest, since there can be multiple trees). We posit some properties of this new representation: (i) each subtask has either 1 or 0 parents; (ii) each subtask is the root of a unique subtree; (iii) subtasks with 0 parents will be the root of a tree; and (iv) root subtasks can be modelled sporadically.

To justify the last point, we note that timer tasks trigger periodically, and it's assumed that parentless subscription tasks are triggered by some external event for which a minimum inter-arrival time is known. This assumption is in line with prior literature [110]. As previously mentioned, this transformation is possible for ROS 2 applications that are free of cycles and can be modeled as DAGs.

3.2.4 Fusion Nodes

A common pattern in ROS 2 involves *fusion nodes*, which combine data from multiple input topics. While the previous sections already sufficient for general DAGs, we present special commentary for the modeling of fusion nodes in graph unfolding. This depends on their implementation:

- **Trigger-Based Fusion.** If a node subscribes to multiple topics and its single callback executes upon message arrival, potentially varying its behavior or execution time based on which topic triggered it or conditionally publishing, our unfolding process (Section 3.2.3) still applies. The fusion node is duplicated for each incoming topic edge. While this correctly models the potential execution paths for analysis, applying the node's full WCET to each unfolded instance can lead to pessimistic (safe, but potentially loose) response time bounds if the actual execution is conditional or significantly shorter for certain triggers.
- **Timer-Based Fusion.** If a node subscribes to multiple topics, buffers the data internally, and uses a separate periodic timer callback to process the fused data and publish



(a) before unfolding (b) after unfolding

Figure 3.7: Unfolding of a ROS 2 Graph into a Forest. Timers in purple, subscriptions in blue

it, our model handles this naturally. The subscriber callbacks form the ends of their respective upstream DAGs. The timer callback forms the root of a *new* separate DAG, as its execution is triggered by its timer, not directly by the arrival of input messages via DDS.

Now that we've established the mapping to a DAG task model, we next describe how to make use of the events executor to create a fixed-job-level-priority scheduler.

3.2.5 Executor Design

Prior work [111] demonstrated the usefulness of the new ROS 2 events executor for scheduling task chains on a uniprocessor. We now present a queueing mechanism for the events executor that enables ROS 2 to schedule general DAG tasks as well, with fixed-job-level-priority scheduling.

In [111], a scheduler segregates a release thread from a scheduler thread. During DDS events, received messages insert an event object into a priority queue (also introduced in [111]). The decision about which subtask to release depends entirely on the implementation of the delegator's priority queue. This is one of the primary benefits the events executor has over the original ROS 2 executor. By default, it is a FIFO scheduler. However, by replacing this events queue with compatible alternative implementations, other scheduling paradigms can be created. In a traditional first-task-priority scheduler (FTP), all subtasks in the graph (in our case, chain/tree) inherit the same priority as their parent. In a pub/sub system such as ROS 2, the linkage between parent and child is not readily apparent. Even if a ROS 2 executor tracked the relation between publisher and subscriber, a single topic can have multiple publishers. So a subtask may have multiple parents, with no obvious way to distinguish which triggered the release of an individual job.

We propose a queueing mechanism that allows children to infer which parent triggered their execution and inherit the appropriate priority. It consists of 2 queues:

- **Priority queue for root subtasks.** Released timer subtasks are kept in a min-heap (or max-heap) sorted by some fixed priority, such as their period for a rate-monotonic scheduler. This queue may include externally driven subscribers, if there

is a mechanism to specify their priority. It is referenced in Algorithms 7 and 8 as *root_queue*

- **LIFO queue for child subtasks.** All other subtasks, like subscriptions and services, are held in a LIFO queue. Each entry is paired with a priority, which is initially undefined, but is set on the next scheduling decision. This queue is also sorted by priority, per the discussion in Section 3.2.6. It is referenced in Algorithms 7 and 8 as *child_queue*

The release logic is simply sorting a new subtask into the appropriate queue, as shown in Algorithm 7. Root subtasks are assumed to have some known priority (based on a period, deadline, or some fixed value), and child subtasks leave this value undefined.

Algorithm 7 Release Logic

```
1: if event.priority is defined (event is timer) then
2:     root_queue.push(event)
3: else if event.priority is undefined (event is subscription) then
4:     event.priority = latest_priority
5:     child_queue.push(event)
6: end if
```

The scheduling logic is detailed in Algorithm 8. Whenever a job is scheduled, its priority is recorded. This is assumed to the be the priority of all subscribers released until the next scheduling decision. Any new subscribers that are released will inherit this priority at the next scheduling decision.

Algorithm 8 Scheduling Logic

```
1: if root_queue.top.priority > child_queue.top.priority then
2:     latest_priority = root_queue.top.priority
3:     schedule root_queue.pop()
4: else
5:     latest_priority = child_queue.top.priority
6:     schedule child_queue.pop()
7: end if
```

When a subtask is scheduled, the DDS communication layer, which manages messages and communication, is queried for the message data to pass to the subtask's callback. To ensure that the correct instance of a subscriber is released, the underlying DDS layer also should

be configured to release messages in LIFO ordering. Subscription events are released in LIFO order, but the messages they operate on are only passed to them after scheduling. Thus, it is crucial that the underlying DDS message queues are also LIFO ordered, since otherwise multiple instances of a single subscription may be assigned the wrong message at scheduling. This would be equivalent to running different jobs of a subtask out of order, which is undesirable.

3.2.6 Analysis

In this section, we show that a ROS 2 application running on the events executor with our specialized queue implementation is analytically equivalent to a fixed-priority tree scheduler on a uniprocessor system. That is, the application will generate the same schedule as a conventional fixed-task-priority tree scheduler if given the unfolded equivalent of a ROS 2 application's graph. We focus on limited preemption because ROS 2 executors cannot be preempted during the execution of a subtask, but subtasks from different trees are allowed to interleave, so the task as a whole has limited preemption.

We first assume that subtasks at the root of a tree have some known integer-valued priority for each job release. This may be a fixed priority, or can vary between job releases (such as a deadline for EDF). This priority is inherited by all subtasks. These root subtasks are naturally the first to run. In the following lemmas, we will lay out the natural ordering of scheduled subtasks and the priorities they have in association with each other.

Lemma 1. *When idle, the only subtasks that may be eligible for release are the root subtasks of each tree.*

Proof. Since the system idles, the two queues specified in Algorithms 7 and 8 are empty, by definition of a greedy scheduler. Since we stipulate that all child subtasks are released as soon as their parent complete, no new child tasks will become eligible after the last task has completed. Since the root subtasks are triggered periodically, they are the only ones that can be released when no other subtasks are running. □

Lemma 2. *Any newly released child subtask will have a priority equal to the subtask that just finished execution.*

Proof. In our system model we require that children are released immediately upon the parent finishing and no later. Therefore it would be contradictory to assume anything other than the subtask that just finished is the parent of the child subtask that was released. By definition of FJP, a child inherits the priority of its parent. □

Lemma 3. *Any unscheduled subtask, τ_a, in the events queue must have a priority equal to or less than the currently running subtask, τ_b.*

$$P_a \leq P_b$$

Proof. By contradiction: Otherwise τ_a would have been scheduled instead. Since our scheduler is nonpreemptive, no new events would have been enqueued by Algorithm 7 since the last scheduling decision. □

Lemma 4. *Any element placed into the children queue from Algorithm 7 will always be the highest priority element.*

Proof. Any child subtask, τ_c, released at the completion of subtask τ_b must have the same priority as subtask τ_b, by Lemma 2.

$$P_b = P_c$$

Therefore, applying Lemma 3, any already released subtask τ_a must have a lower priority.

$$\forall \tau_a, \forall \tau_c \succ \tau_b | P_a \leq P_b$$

$$\implies P_a \leq P_c$$

□

We can impose the rule that ties are broken in LIFO order, based on release eligibility.

Lemma 5. *Any element removed from the LIFO queue will always be the highest priority element.*

Proof. Lemma 4 combined with definition of LIFO. □

Theorem 2. *The highest-priority subtask will always be scheduled next by Algorithm 8.*

Proof. In Algorithm 8, only the top subtasks of the roots queue and children queue are compared. Per Lemma 5, the children queue is functionally a priority queue. Therefore, the two head subtasks have higher priority than any other subtasks in their respective queues. The greater of the two is guaranteed to be a higher priority than any released subtasks. Therefore, scheduling Algorithm 8 is guaranteed to schedule the highest priority released job. □

Additionally, this proof considers priorities of individual instances of subtasks to be independent from each other. Therefore, it's compliant with any job-level fixed-priority scheduler. This assumes that all constraints we detailed in Section **??** are kept, and the DDS layer has been configured to release messages in LIFO order as well.

3.2.7 Extension to non-LIFO queues

The events queue design in Section 3.2.5 above uses LIFO-ordering for new subscriber jobs. However, we are not aware of any ROS 2 communication middleware that supports LIFO message queues. This means that even when jobs are executed in the correct order, the messages for them to process would be given in the wrong order. This would be equivalent to running different jobs of a specific subtask out of order, which is undesirable. This work provides a motivating usecase for the implementation of LIFO message queuing. Even when LIFO queues are not available, a subset of applications is still schedulable. Specifically, if guarantees can be made that there will never be more than one released job of a task at a time, then the inversion from lacking a LIFO message queue never occurs. Such specific applications are detailed as follows.

Lemma 6. *Any queue that always has at most one element is functionally a LIFO queue.*

Proof. In a queue of size one, the only element able to be removed is the head element, so removing it abides by the LIFO definition. □

Lemma 7. *If a subscription C with response time R_C is the only subscriber for a topic and its parents $A, B, \ldots,$ with response times $R_A, R_B, \ldots$ are harmonic with a* sufficient *phase shift, then the message queue for that subscription's topic will never have more than one message in it.* Sufficient *means that the taskset owning $A, B, C, \ldots$ is strictly periodic and*

$$t_{RA} + R_A + R_C \leq t_{RB}$$

$$t_{RB} + R_B + R_C \leq t_{RA} + T_A,$$

where t_{Rx} is the release time of x, occuring at timepoints $\phi_x + kT_x \mid k \in \mathbb{N}$. Without loss of generality, this extends to more than 2 parents by applying these conditions to each pair of parents.

Proof. A job/message x is added to the queue at each job release, t_{Rx}, and will be removed by the time that job completes $t_{Rx} + R_x$.

The subscription task in question has a release time of $t_{RC} = t_{RA} + R_A$, where A is the triggering parent. It is removed from the message queue by the pessimistic timepoint

$$t_{RA} + R_A + R_C$$

By the given conditions, this is earlier than the release of the next parent, t_{RB}. Therefore the queue will be empty by the time the next job is released, and that job will be removed before the next, and so on.

The events queue will only have either zero or one jobs of this subscription. □

Theorem 3. *Even if the message queue isn't configured to be LIFO order, the highest-priority subtask will always be scheduled next by Algorithm 8 if the conditions formulated in Lemma 7 hold.*

Proof. If the conditions in Lemma 7 are true, then the message queue has a maximum size of one. Consequently, due to Lemma 6, the queue is functionally a LIFO queue. Hence, Theorem 2 can be applied. □

Applying Existing Schedulability Tests and Response Time Analysis

Our executor design implements a LP-FJP policy, where subtasks run non-preemptively, but preemption can occur between subtasks. Furthermore, as shown in Section 3.2.3, ROS 2 graphs can be modeled as a forest of DAGs (trees) for analysis. Therefore, established analyses for LP-FJP scheduling of DAG tasks on uniprocessors are directly applicable.

Specifically, the analysis provided by Nasri et al. [92] addresses this exact model (LP-FJP for DAGs). Their analysis tool [93] considers factors like release jitter and execution jitter, and supports multiprocessor analysis, though we focus on the uniprocessor case here. We utilize this specific analysis [92] in Section 3.2.8 to calculate the theoretical worst-case response times (WCRTs) for comparison with our experimental results. Using an analysis designed for the target scheduling policy (LP-FJP DAGs) is crucial for accurate validation, as opposed to analyses developed for different policies like the default ROS 2 executor [29].

When applying these analyses, one should use the **eligibility time** (the time a subtask *could* run if the executor were free and it had highest priority) as the conceptual **release time** for the analysis calculation. The term 'release time' used elsewhere in this work refers to the point in time when a subtask is enqueued into the events queue, which occurs at scheduling points.

3.2.8 Evaluations

This evaluation is divided in two parts. In the first part, we implemented the queue detailed in Section 3.2.5 for a RM fixed priority scheduler, as well as one for EDF. We first apply the analysis provided by [92] to a synthetic taskset. If our executor truly maps to a Limited Preemption Fixed Job-Level Priority scheduler, we would expect their analysis to accurately bound WCRT. This bound is compared to an empirical evaluation in Section 3.2.8.

We then present a more empirical measure of performance by assessing the performance of our proposed executor design from Section 3.2.5 compared to the existing events executor, as well as the default ROS 2 executor. We use the Autoware Reference Benchmark, which is intended to compare the performance of different executor designs.

We are limited to a synthetic taskset and the Autoware Reference Benchmark because, as mentioned in Section 3.2.7, no communication middleware currently available provides LIFO-ordered message queueing, which violates the assumptions found in Lemma 5. The synthetic evaluation in Section 3.2.8 is data-agnostic, so by tracing scheduling decisions, the hypothetically LIFO-ordered child subtasks can be reassigned to their correct parents in evaluation post-processing. The Autoware Reference Benchmark (also in Section 3.2.8) satisfies the condition in Lemma 7, and therefore our analysis holds, according to Theorem 3.

All experiments are run on a Raspberry Pi Model 4, pinned to a single core with the timer's management thread in the events executor assigned a higher priority, as suggested in [111]. We report the 99.7th percentile of response times to exclude rare outliers potentially caused by system noise (e.g., OS jitter) rather than by the scheduling algorithm itself. All experiments are conducted for 10 minutes, over multiple hyperperiods.

Synthetic Many-to-Many Topic

To demonstrate that our analysis holds for an unfolded graph, we replicate the same application shown in Figure 3.7 and vary the WCETs of subtasks to create 3 different utilization levels. Utilization levels are approximate and rounded to even numbers for simplicity. Since this experiment does not use a LIFO-compliant middleware, we take measures to counteract the effect where subsequence jobs from a specific subtask are misordered. Since all data is arbitrary in this synthetic evaluation, jobs of a particular subtask are fungible. That is, if job x_i of subscription subtask x is scheduled to run, it does not matter if it is given the data for job x_j instead, provided the WCET and priority match the intended job that should run. In post-processing, we identified when each child subtask was released (i.e., when its parent completed) and when it was scheduled. The response time is calculated as the difference between these two timestamps. This correctly measures the scheduling delay introduced by our executor, bypassing the message-data mismatch that would occur from a FIFO message queue.

Each application is centered around a single topic with three publishers and two subscribers. Each publisher has a different period. After unfolding, we can model this application as a forest of 3 trees, each with 1 parent and 2 children. We created three variations of this

Table 3.2: WCET of Subtasks in Synthetic Tasksets

	Parent by Period			Children	
Utilization	25ms	41ms	51ms	A	B
50%	1ms	1ms	5ms	2ms	2ms
70%	2ms	1ms	12ms	2ms	2ms
90%	3ms	1ms	12ms	2ms	4ms

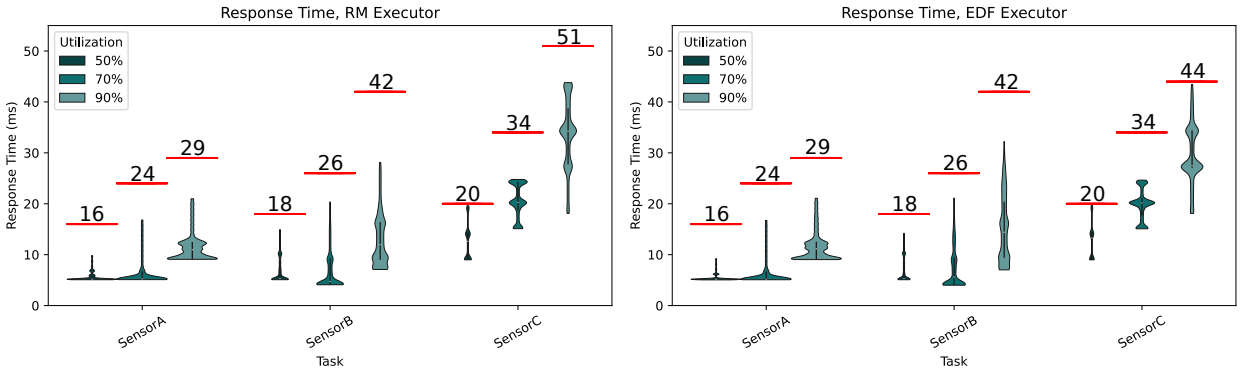


Figure 3.8: Response times for the synthetic task set compared against the WCRT bounds from analysis [92]. Task A, B, and C correspond to the root tasks with periods 25ms, 41ms, and 51ms, respectively. Red lines indicate the analytical WCRT.

taskset for different utilization levels of (approximately) 50%, 70%, and 90%. The WCET times for each subtask in each taskset are illustrated in Table 3.2.

The unfolded forest is entered into an analysis tool [92, 93], which provides a WCRT across the entire tree for each job with both RM and EDF scheduling.

Results are shown in Figure 3.8. Our measured WCRTs closely match the predicted WCRTs from the analysis. The analysis for the 90% utilization scenario with RM scheduling predicts deadline misses for Task A and Task B, which highlights that the task set is not schedulable under these conditions. Our executor's behavior aligns with this analysis.

Autoware Reference Benchmark

We also compare our executor's performance to existing executors in the Autoware Reference Benchmark, a simulated version of the Autoware application detailed in [100], which includes

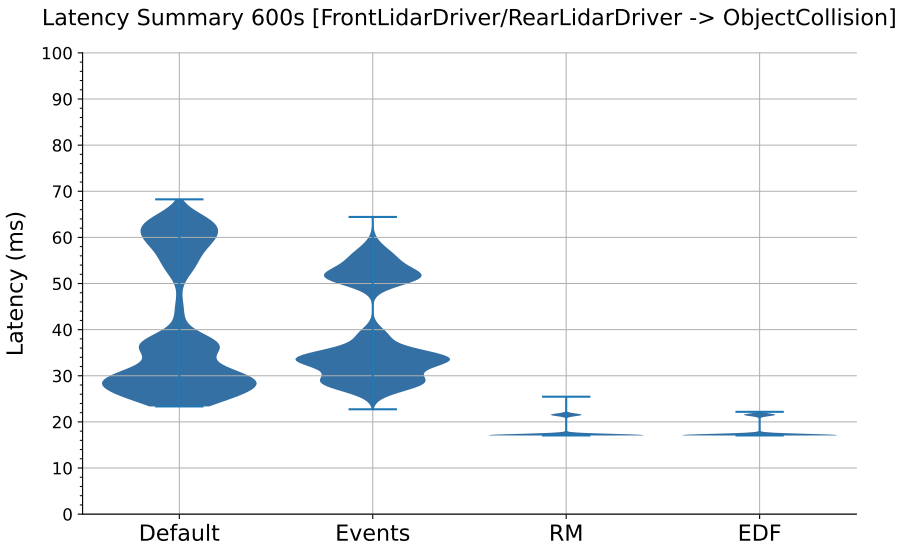


Figure 3.9: Autoware Reference Benchmark Executor Comparison

interconnected timers and subscriptions.[9] Specifically, we measure the end-to-end latency of the **hot path** defined in the benchmark from the front and rear LiDAR sensors to the object collision estimator (c.f. the Autoware reference system), as this latency is a key metric for responsiveness in crash avoidance. The data at the start of the hot path is generated at a frequency of 10Hz.

Results are shown in Figure 3.9. The default and events executor perform the worst, but are still consistently within the implicit 100ms deadline of the hotpath. However, with our RM and EDF modifications to the events executor, the worst case is consistently under 25ms.This represents a 63% improvement over the default executor, and a 61% improvement over the events executor. In addition to a better WCET, there is less variability as well.

3.2.9 Conclusions and Future Work

Prior literature [20, 29, 33, 110, 114] has catered to the limitations of the default ROS 2 executor. Those contributions quantify its behavior and propose multithreaded work-arounds rather than directly addressing the blocking caused by the processing window. In contrast,

[9] https://github.com/ros-realtime/reference-system/blob/main/autoware_reference_system/README.md

our work builds on the events executor, which isn't affected by the processing window. Scheduling decisions are made after each executed job, making it compliant with a limited-preemption scheduler. Additionally, the response-time analysis provided by prior literature has focused on chains. Our work not only provides support for general branching graphs, but it does so with minimal retooling required and without the need for a priori application analysis and graph decomposition, as is the case with PiCAS [33].

This work introduces a novel abstraction with which ROS 2 applications can be analyzed as forests of trees. It also presents a new design for an events queue that supports arbitrary fixed-job-level-priority schedulers, and naturally honors the precedence relations between subtasks without needing to be aware of them. Finally, it provides an analysis proving that the events executor using our queue is behaviorally identical to a conventional scheduler operating on the unfolded equivalent of our application. Our analysis and empircal evaluation confirm that the schedules created are identical (subject to arbitrary tie-breaks). This further closes the gap between established real-time literature and ROS 2 scheduling analysis.

The primary limitation to this work is the necessity for a LIFO message queue, which is not to our knowledge provided by any commercially available or open-source DDS middleware. Integrating that feature into an existing open-source middleware framework is a natural direction for future work, to broaden the applicability of the approach presented here beyond the currently targeted set of applications meeting Theorem 3's conditions (e.g., harmonic periods). Future work also could explore extending this model to mixed-criticality systems or handling other preemption constraints.

3.3 Commentary on Multi-threaded Schedulers

While the work presented in this chapter focuses on uniprocessor scheduling, a natural extension is to consider multi-threaded and multi-processor systems. However, extending graph-aware scheduling to such systems in ROS 2 introduces significant challenges that are not present in the single-threaded case. This section discusses these challenges, focusing on two primary issues:

1. The inherent information erasure in the pub/sub paradigm that obscures the parent-child relationship between consecutive subtasks.

2. The necessity of handling mutual exclusion, an established feature of ROS 2.

3.3.1 The Information Erasure Problem

One of the biggest limitations of extending the work above to multi-threaded schedulers lies with the inherent information erasure of the pub/sub paradigm. As discussed in Paper 2, knowing a subtask's parent is crucial for making graph-aware scheduling decisions. In a single-threaded system, the parent of a newly released subtask is unambiguously the most recent subtask to have executed.

This is not the case in a multi-threaded system. When multiple publisher subtasks are allowed to run concurrently on different threads, a message could be published at any point during their execution. If a topic has multiple publishers, there is currently no standard way to discern which publisher instance is associated with a given incoming message. This ambiguity makes it impossible to reconstruct the complete dataflow graph at runtime, which is a prerequisite for graph-aware scheduling.

This limitation can be addressed in two ways:

1. **Message Decoration**: One could extend the ROS 2 Interface Definition Language (IDL) [59] to allow for metadata to be embedded within messages. This metadata could carry information about the parent task, allowing the scheduler to identify the origin of each message. While this also could be implemented manually by developers modifying their message definitions, such a solution would not be as portable or generalizable across applications.

2. **Architectural Constraints**: Another approach is to constrain the system architecture to avoid ambiguous dataflows. One could group chains of callbacks together so that a parent always executes in the same thread as its children, ensuring a clear line of succession. This approach has been pursued in prior work. For example, [33] offers a solution that assigns callbacks to specific threads *a priori* to group chains together and avoid branching.

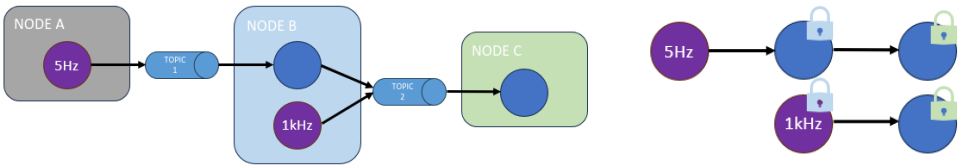


Figure 3.10: Unfolding of a ROS 2 graph with locks. Callbacks in the same mutual exclusion group (see nodes B and C) are modeled with a shared lock that is passed to their virtual copies created during graph unfolding.

3.3.2 Mutual Exclusion and Schedulability

Even if the information erasure problem is solved, any multi-threaded ROS 2 scheduler must contend with mutual exclusion. By default, callbacks within the same node are mutually exclusive to prevent race conditions on shared memory. This can be configured more granularly with *callback groups*, but the principle remains: tasks within the same mutual exclusion group are not allowed to execute concurrently.

This constraint is irrelevant for a single-threaded executor, as two tasks can never run at the same time. In a multi-threaded executor, however, it becomes a critical source of blocking. A high-priority callback may become ready but be unable to execute if a lower-priority callback from the same mutual exclusion group is currently running on another thread.

Fortunately, this behavior can be modeled and analyzed using existing real-time systems theory. We can model each mutual exclusion group as all requiring a shared resource protected by a lock. All callbacks belonging to that group must acquire the lock before executing and release it upon completion. This transforms the problem into one of scheduling real-time DAGs with shared resources, a well-studied area.

This approach lifts the common, but often impractical, requirement that all callbacks must be reentrant (i.e., not block one another). By explicitly modeling the non-reentrant behavior as locking, we can analyze its impact on schedulability. The analysis for DAG tasks with locks from [42] is particularly relevant. This work provides schedulability tests and processor allocation bounds for DAGs under federated scheduling that contend for shared resources.

For instance, an upper bound on the number of processors required for a task is given by:

$$n = \left\lceil \frac{C + B^C - L - B^L}{D - L - B^L} \right\rceil \tag{3.5}$$

Where C and L are the work and span of the graph, D is the deadline, and B^C and B^L represent the additional work and span introduced by blocking on locks. Such analysis would be critical for providing real-time guarantees in a multi-threaded graph-aware ROS 2 executor. Figure 3.10 illustrates how callbacks in a group can be modeled with a shared lock.

3.3.3 Research Scope

Initially we had planned to include multi-threaded scheduling within the scope of this work. However, due to the significant implementation and analysis complexities required to robustly address the information erasure and mutual exclusion problems, we chose to focus on the uniprocessor case. We believe that the insights from Sections 3.1 and 3.2 provide a solid foundation for future explorations into multi-threaded graph-aware scheduling in ROS 2.

3.4 Conclusion

This chapter has explored two distinct approaches for providing real-time scheduling guarantees for ROS 2 applications, both building upon the modern *events executor*. The contributions from the two papers presented offer complementary solutions to the problem of predictable execution in ROS 2.

The first paper demonstrates how to make the events executor compatible with the classical real-time model of periodic task systems. Its analysis is tailored for ROS 2 applications that can be decomposed into a set of independent callback chains. Its applicability is limited to systems with only the simplest graph structure: isolated chains and single periodic callbacks with no downstream children. This approach provides a vital bridge to classical scheduling theory, enabling the use of mature analysis techniques for a significant class of ROS 2 systems.

The second paper extends this work to address more generic applications. It embraces the inherent complexity of arbitrary DAG structures, where dataflows can branch and merge freely. It introduces a model for a graph-aware fixed-job-level-priority scheduler where subscription callbacks inherit the priority of the publisher that triggered them. This allows us to treat an entire tree in a ROS 2 graph as a single parallel task for analysis purposes. We also demonstrate how, due to scheduling behavior in ROS 2, arbitrary ROS 2 graphs can be modeled as a forest of trees for our analysis.

As discussed in Section 3.3, extending this work to multi-threaded environments is a significant challenge due to the information erasure inherent in the pub/sub paradigm and the complexities of managing mutual exclusion. Nonetheless, the results in this chapter lay a critical foundation for uniprocessor scheduling and clearly pinpoint the open problems that must be solved before similar guarantees can be offered for multi-threaded ROS 2 systems.

Chapter 4

State Estimator Handoffs in Safety-Critical Systems

The prior two chapters established the infrastructure needed for predictability in component-based applications. Hazcat removed non-deterministic data copies, and the redesigned ROS 2 executor supplied analyzable scheduling. This final chapter asks a complementary question: *how can control algorithms exploit those timing guarantees?*

As part of a collaboration with colleagues at Purdue University we are investigating mixed-criticality control with elastic scheduling. Near a point of stability, a controllable system can be modelled linearly with a Kalman filter (KF), which provides fast, low-cost estimates. When the operating point drifts outside that linear region the state estimator must transition to a higher-fidelity Extended or Unscented Kalman filter (EKF/UKF), effectively a mixed-criticality mode change. We assume a *handover window*: the finite interval between the decision to switch and the instant at which estimates from the new filter are first used. During this window the two filters may execute concurrently, share state, and must still meet their respective deadlines.

In addition to leveraging the existing memory and scheduling work, this project was also meant to showcase the use of multiple hardware accelerators in a single application. We hypothesize that the nature of FPGAs compared to GPUs makes them better suited to different tasks. For basic parallelism, GPUs are shown to fare better, as demonstrated later in Figure 4.3. However, the highly pipelined nature of FPGAs makes makes them better suited for applications where successive events are independent and can be run concurrently. FPGAs' customizable fabric also allows them the flexibility to implement data-wide operations that are not strictly parallel (such as the parametric Jacobian functions of an Extended Kalman Filter, detailed later in Section 4.3.2).

In this Chapter, we derive timing-aware switching policies and demonstrate, analytically, that the hand-off can be performed without violating real-time constraints. The chapter therefore illustrates a scenario for how control and scheduling can be co-designed atop the middleware layers developed earlier, and describes how the use of hardware accelerators may be incorporated.

4.1 Preliminary Work

In preparation for implementing the control system on an FPGA, I started preliminary work on a BLAS library on which to build state estimators. The FPGA implementations of these state estimators were never realized, in favor of the more interesting theoretical aspects of this project. However, the BLAS library by itself yielded results, including surpassing some vendor-supplied BLAS operations, so I include it here.

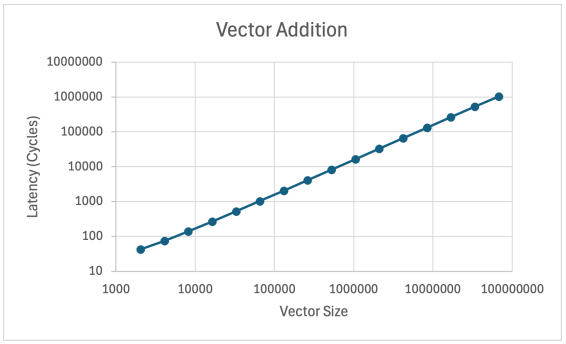
My library, DYFC BLAS [96], was developed as a competitor to the vendor-supplied Vitis BLAS library [3]. It was written in High Level Synthesis (HLS) C++ and developed with the following ethos:

- **Documentation**: In contrast to AMD's documentation for their BLAS library, DYFC BLAS was built with unit testing, examples, and thorough documentation from the start. Every operation had a simplified default form that new developers could fall back on, and all the advanced operations needed to optimize performance were well documented with concrete and tested examples.
- **Ease of use**: The library was designed to be as Pythonic as possible to use. In the original Vitis BLAS library, the user had to specify data arrangements manually when using BLAS operations (such as general vs symmetric matrices or row-major vs column-major). In DYFC BLAS, all matrices are treated the same by default, and some basic data-ordering optimization was done automatically. Developers had the option to initialize a matrix manually in a more optimized arrangement, and compiler warnings would be thrown if a more efficient arrangement was possible. But the interface was kept as simple as possible, using reasonable assumed default behavior to expedite development.

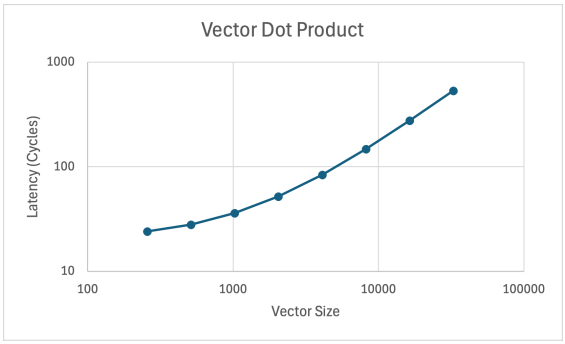
- **Provable Performance**: The entire project was developed with an integrated CI/CD pipeline complete with automated benchmarking. That way, any changes made to the codebase would have their performance impact clearly quantified. This promotes the exploration of new designs and optimizations, and dissuades any feature bloat that inhibits performance.

4.1.1 DYFC BLAS Performance

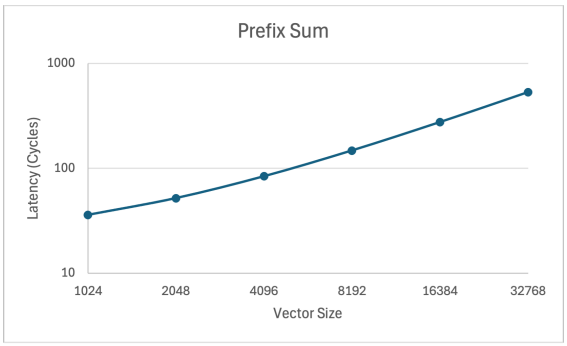
AMD has publicized performance numbers [2] for their own Vitis BLAS library [3], but these are not reproducible. They only publish numbers on their *gemv* and *gemm* operations, and the links to the benchmarks that created these are broken. Below is a non-exhaustive list of the working operations in DYFC BLAS, with comparisons to the advertised (but unverified) numbers from Vitis BLAS, where possible.



(a) Vector Addition Performance



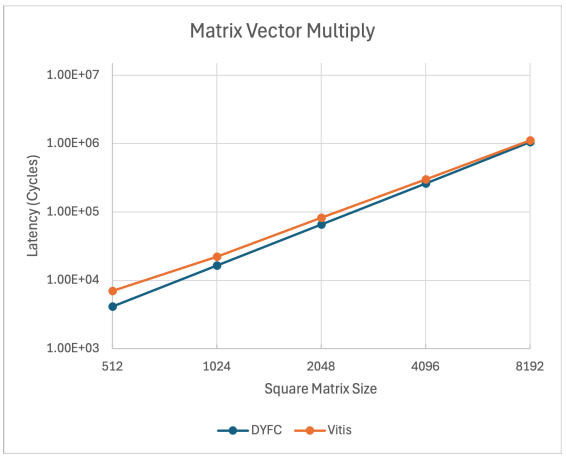
(b) Dot Product Performance



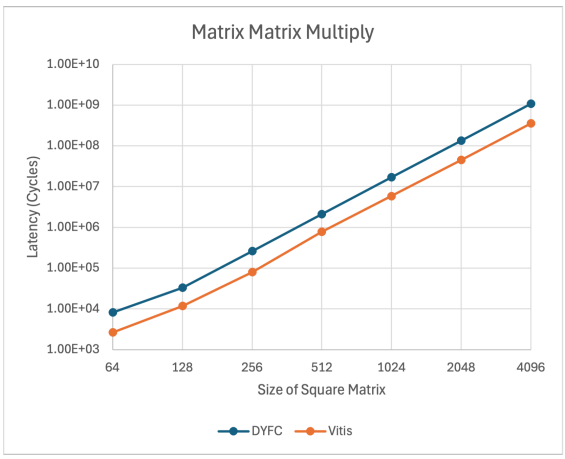
(c) Prefix Sum Performance

Figure 4.1: DYFC BLAS performance across different operations.

In Figures 4.1 and 4.2 I present the estimated clock cycles to complete one iteration of the kernel in hardware emulation.



(a) Matrix-Vector Multiplication Performance



(b) Matrix-Matrix Multiplication Performance

Figure 4.2: DYFC BLAS performance comparisons across different operations.

Most notably, my implementation of the *gemv* operation was able to outperform the Vitis BLAS library. In Figure 4.2a, both my BLAS library (DYFC) and the vendor's implementation (Vitis) are running on a 3ns clock cycle. It should be noted that there are two different implementations of the *gemv* operation: one for a column-major ordered matrix, and one for row-major order. The latter is not able to scale down to a 3ns clock cycle, but synthesizes to a highly comparable cycle count to the column-major implementation at slower clock speeds.

In Figure 4.2b, we see that my implementation of the *gemm* operation is notably slower than the Vitis BLAS library. This is likely due to the fact that I use a naive implementation of the *gemm* operation, whereas the Vitis BLAS implementation was likely a systolic array, the accepted standard implementation for *gemm* on FPGAs.

When we compare my FPGA implementation of the *gemv* operation to the cuBLAS implementation on a GPU (Figure 4.3), we see that the FPGA's performance is beaten by an order of magnitude. The GPU used, the A5000, is approximately half the cost of the chosen FPGA, the U250. There is a constant overhead of about 10 µs in the GPU implementation, making the FPGA faster at smaller matrix sizes but the performance benefits of the GPU quickly dominate for matrices larger than approximately 300×300. This highlights the superiority of GPUs over FPGAs for these types of highly-parallelizable linear operations.

Work on this library is still incomplete, but it promises to be a useful and accessible framework for future high-performance computing applications developed in HLS for FPGAs. We

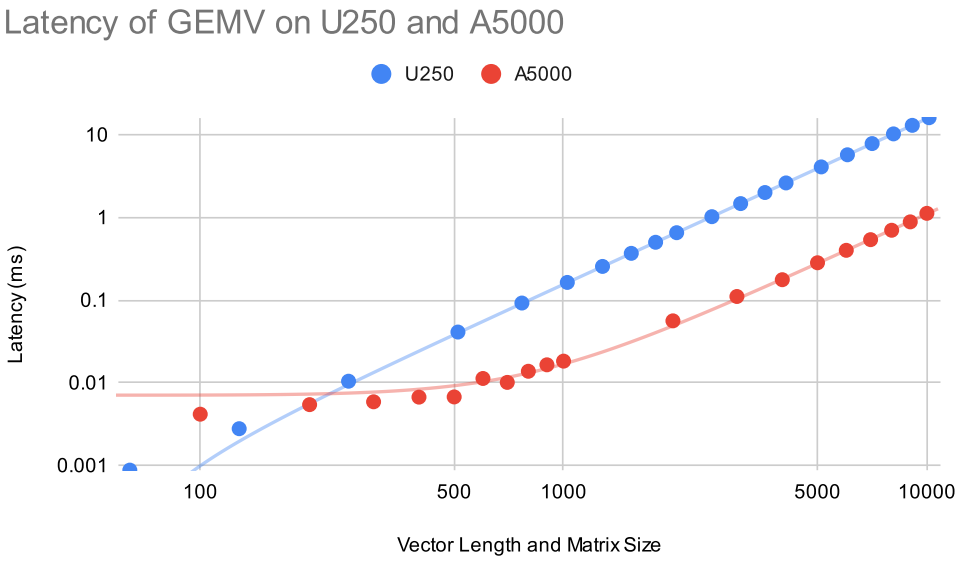


Figure 4.3: Performance comparison of Matrix Vector Multiplicaton FPGA vs GPU

now return our attention to the control-scheduling problem that forms the core of this chapter. This work was done in collaboration with the rest of my lab at Washington University in St Louis.

4.2 Introduction

It is often convenient and less resource intensive to model and control system dynamics as being linear, to the extent possible. However, dynamics of cyber-physical systems in application domains ranging from real-time hybrid simulation experiments for earthquake engineering [37] to avionics [108] may become non-linear under conditions that are reasonably commonplace. In those cases and others (e.g., due to instability induced by environmental disturbances), the system state may deviate from a linear operating region such that linear state estimation is no longer sufficiently accurate.

State estimation using nonlinear models may be needed for accurate tracking in such cases, but often is significantly more intensive computationally. If a maximum acceptable error threshold can be characterized (e.g., informed by tolerances on some states of the real-world system or the stability bounds of the state estimators used), and bounds on the estimation

error can be deduced at run-time, a decision procedure then can be defined for switching between linear and nonlinear estimators. Specifically, for efficiency of resource usage a linear estimator may be used as long as the error remains some distance from the threshold. However, as the error approaches within some metric distance of that threshold (i.e., tracked as an estimation error norm) the system must switch to a nonlinear estimator (and must do so in a sufficiently timely manner) so that the threshold is never exceeded. Similarly, another threshold may be defined to trigger switching from the nonlinear estimator back to the linear one when the error recedes a sufficient distance from the maximum acceptable error threshold.

Co-designing scheduling and control parameters so that cyber-physical systems may respond rapidly to such changing environments is also a first-class concern. Recently, Sudvarg et al. [108] showed how to calculate bounds on rates of change in a control norm relative to a bounded radius of reachable system states between two consecutive jobs of a control task. They used those bounds in a control barrier function to compute response times that if respected ensure positive invariance within safe control regions for systems with non-linear dynamics, even if the control task changes its strategy between consecutive jobs. Finally, they showed that for implicit deadline scheduling models (e.g., partitioned EDF) it is straightforward to calculate control task periods to ensure those response time bounds are met. In this chapter we adapt those methods to the related problem of when to switch between different state estimation strategies.

This motivates a complete approach to scheduling control strategies that use linear state estimation within linear regions, but switch to nonlinear state estimation when leaving them. Doing so safely and with assurance of schedulability poses important research questions that we address here:

- How can regions of linearity be defined to ensure appropriate estimation methods are used both inside and outside of them?
- How can a system detect when it is departing or re-entering a region of linearity?
- How can a system maintain safety across transitions between linear and nonlinear state estimation?
- How can constraints and optimization objectives be combined to determine when a control strategy should switch between linear and nonlinear state estimation?

- How can we incorporate the probabilistic nature of state estimators into that decision making?

The rest of this chapter is structured as follows: In Section 4.3 we present background information on linear and nonlinear state estimation. In Section 4.4 we formalize control and scheduling aspects of our system model. We describe how our system model can be used to enforce safety properties in Section 4.5, and in Section 4.6 we describe how such enforcement can be applied to the problem of switching between different estimators of the system's state. Finally, in Section 4.7 we provide concluding remarks and discuss avenues of future work.

4.3 Background

As background, we discuss four widely-used state estimation techniques: Kalman Filter (Section 4.3.1) for linear dynamics and Extended Kalman Filter (Section 4.3.2), Unscented Kalman Filter (Section 4.3.3), and Particle Filter (Section 4.3.4) for nonlinear dynamics. Each of them has a *prediction* step and an *update* step, and a Particle Filter may have an additional *resampling* step.

4.3.1 Kalman Filter

A Kalman Filter (KF) [70] is a *linear* state estimator. The state is computed as a probability variable, with a mean estimate and a covariance of uncertainty. For a state vector of size n, the covariance matrix is of size nxn. If the individual state elements are independent from each other, the covariance matrix will only be non-zero on the diagonal.

The filter works in two phases: prediction (Eq 4.1) and update (Eq 4.2). Prediction uses a linear model of the system to increment the state estimation, but widens the variance of that estimate. Update incorporates a potentially noisy measurement to update the prediction and shrink the estimate's uncertainty.

Prediction Step:

$$\begin{aligned} \hat{x}_k^- &= A\hat{x}_{k-1} + Bu_k \\ P_k^- &= AP_{k-1}A^T + Q \end{aligned} \tag{4.1}$$

Where $\hat{x} \in \mathbb{C}^n$ is the estimated state, $P \in \mathbb{C}^{n \times n}$ is the covariance matrix of that state, $u \in \mathbb{C}^m$ is the input to the system, $A \in \mathbb{C}^{n \times n}$ and $B \in \mathbb{C}^{n \times m}$ are matrices that define the linearization of the system's state transition function, and $Q \in \mathbb{C}^{n \times n}$ is the variance of any disturbance that might affect the system state in parallel to the input.

Update Step:

$$\begin{aligned} K_k &= \frac{P_k^- C^T}{CP_k^- C^T + R} \\ \hat{x}_k &= \hat{x}_k^- + K_k(y_k - C\hat{x}_k^-) \\ P_k &= (I - K_k C)P_k^- \end{aligned} \tag{4.2}$$

Where $K_k \in [0, 1)$ is a ratio of how much to trust the predicted estimate versus the measurement, $y \in \mathbb{C}^p$ is the measurement, and $C \in \mathbb{C}^{p \times n}$ is a "mask" selecting which portions of the state are measurable. Strictly speaking, the update step is optional, and the the output of one prediction step, $\hat{x}^- P^-$, can be fed directly into the next prediction step.

4.3.2 Extended Kalman Filter

An Extended Kalman Filter (EKF) [103] is a *nonlinear* estimator. It operates similarly to a Kalman Filter, but the EKF uses the nonlinear form of the state transition function, as well as linearizing at the current state estimate.

The Jacobians of the state transition function and observation function are repeatedly recomputed at the current state estimate.

$$F_k = \frac{\delta f}{\delta x}|_{\hat{x}_{k-1}, u_k}$$
$$H_k = \frac{\delta h}{\delta x}|_{\hat{x}_k^-}$$

F_k and H_k are analogous to A and B in the Kalman Filter. The natural connection between KF and EKF should then be apparent. See the EKF prediction (Eq 4.3) and update (Eq 4.4) steps below.

$$\begin{aligned} \hat{x}_k^- &= f(\hat{x}_{k-1}, u_k) \\ P_k^- &= F_k P_{k-1} F_k^T + Q \end{aligned} \tag{4.3}$$

$$\begin{aligned} K_k &= \frac{P_k^- H_k^T}{H_k P_k^- H_k^T + R} \\ \hat{x}_k &= \hat{x}_k^- + K_k(y_k - h(\hat{x}_k^-)) \\ P_k &= (I - K_k H_k) P_k^- \end{aligned} \tag{4.4}$$

Where $f(x, u)$ is the state transition function of the system and $h(x)$ is the observation function.

4.3.3 Unscented Kalman Filter

An EKF assumes that a probability variable is Guassian, and represents it with a mean estimate and variance. This implicitly assumes the variable's uncertainty is Gaussian. It also assumes the model can be locally linearized.

An Unscented Kalman Filter (UKF) [119] (sometimes called the "Sigma Point Kalman Filter" (SPKF) [9]) provides an alternative to the EKF for handling nonlinearities. Instead of linearizing the system functions using Jacobians, the UKF selects a minimal set of sample

points, called sigma points. These sigma points are chosen such that they capture the mean and covariance of the state distribution accurately.

The sigma points are then propagated directly through the true nonlinear system functions. The mean and covariance of the transformed points are recovered to form the predicted state and covariance. This often leads to better fidelity than the EKF, especially for highly nonlinear systems or when the state distribution significantly deviates from Gaussian after being transformed by the nonlinear function.

Let n be the dimension of the state vector. The UKF typically uses $2n+1$ sigma points: the mean, n "left" sigma points, and n "right" sigma points. The generation involves tunable parameters α, β, κ, with the following additional variables specified

$$W_i^c = \begin{cases} \frac{\lambda}{n+\lambda} & i = 0 \\ \frac{1}{2(n+\lambda)} & i = 1..2n \end{cases}$$

$$W_i^m = \begin{cases} \frac{\lambda}{n+\lambda} + (1 - \alpha^2 + \beta) & i = 0 \\ \frac{1}{2(n+\lambda)} & i = 1..2n \end{cases}$$

$$\lambda = \alpha^2(n + \kappa) - n$$

Prediction Step: The sigma points $\mathcal{X}_{k-1}$ are first generated from $\hat{x}_{k-1}$ and P_{k-1}:

$$\begin{aligned} \mathcal{X}_{0,k-1} &= \hat{x}_{k-1} \\ \mathcal{X}_{i,k-1} &= \hat{x}_{k-1} + (\sqrt{(n+\lambda)P_{k-1}})_i, \quad \text{left sigma points} \\ \mathcal{X}_{i+n,k-1} &= \hat{x}_{k-1} - (\sqrt{(n+\lambda)P_{k-1}})_i, \quad \text{right sigma points} \end{aligned}$$

Where $(\sqrt{(n+\lambda)P_{k-1}})_i$ is the i-th column of the matrix square root (e.g., Cholesky decomposition).

Next, these points are propagated through the state transition function f

$$\mathcal{X}^*_{i,k|k-1} = f(\mathcal{X}_{i,k-1}, u_k) \quad \text{for } i = 0..2n$$

The predicted mean and covariance ($\hat{x}_k^-$ and P_k^-) can then be computed directly

$$\hat{x}_k^- = \sum_{i=0}^{2n} W_i^m \mathcal{X}_{i,k|k-1}^*$$
$$P_k^- = \sum_{i=0}^{2n} W_i^c (\mathcal{X}_{i,k|k-1}^* - \hat{x}_k^-)(\mathcal{X}_{i,k|k-1}^* - \hat{x}_k^-)^T + Q$$

Update Step: $\mathcal{X}_{i,k|k-1}^*$ can be further propagated through the observation function h to be used in the update step, generating the updated mean and covariance ($\hat{x}_k$ and P_k)

$$\mathcal{Y}_{i,k|k-1} = h(\mathcal{X}_{i,k|k-1}^*) \quad \text{for } i = 0..2n$$
$$\hat{y}_k^- = \sum_{i=0}^{2n} W_i^m \mathcal{Y}_{i,k|k-1}$$
$$P_{yy} = \sum_{i=0}^{2n} W_i^c (\mathcal{Y}_{i,k|k-1} - \hat{y}_k^-)(\mathcal{Y}_{i,k|k-1} - \hat{y}_k^-)^T + R$$
$$P_{xy} = \sum_{i=0}^{2n} W_i^c (\mathcal{X}_{i,k|k-1}^* - \hat{x}_k^-)(\mathcal{Y}_{i,k|k-1} - \hat{y}_k^-)^T$$
$$K_k = P_{xy} P_{yy}^{-1}$$
$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - \hat{y}_k^-)$$
$$P_k = P_k^- - K_k P_{yy} K_k^T$$

While computationally more intensive than the EKF due to the need to propagate multiple sigma points, the UKF avoids potential errors introduced by linearization and often yields more accurate state estimates and covariance representations for nonlinear systems.

4.3.4 Particle Filter

Unlike Kalman-based filters (KF, EKF, and UKF) which represent the state uncertainty as a Gaussian distribution (parameterized by mean and covariance), Particle Filters (PFs) [7] represent the posterior probability distribution of the state using a set of random samples, called particles. This allows PFs to represent arbitrary probability distributions.

PFs approximate the posterior probability density function (the probabilty of the current state given a history of measurements) using a set of N weighted particles $\{x_{k,i}, w_{k,i}\}_{i=1}^{N}$, where $x_{k,i}$ is the state vector of the i-th particle at time k, and $w_{k,i}$ is its corresponding weight, such that $\sum_{i=1}^{N} w_{k,i} = 1$. The probability density function is approximated as a "forest of spikes":

$$p(x_k|y_{1:k}) \approx \sum_{i=1}^{N} w_{k,i}\delta(x_k - x_{k,i})$$

where $\delta(\cdot)$ is the Dirac delta function.

A common variant is the Sequential Importance Resampling (SIR) Particle Filter, which operates in predict and update steps:

Prediction Step: Each particle is propagated forward in time using the system's state transition model f, additively including process noise q_{k-1}.

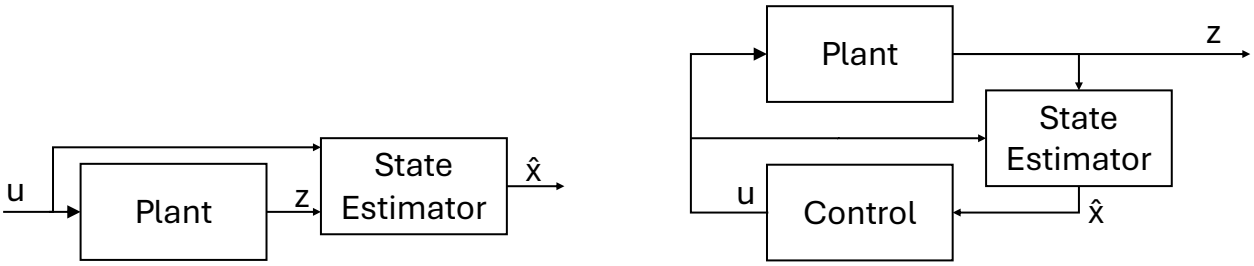
$$x_{k,i} = f(x_{k-1,i}, u_k) + q_{k-1,i}$$

where $q_{k-1,i}$ is a sample of the process noise. At this stage, the weights $w_{k,i}$ are typically carried over from the previous step or initialized uniformly if it's the first step:

$$w_{k|k-1,i} = w_{k-1,i}$$

Update Step: When a measurement y_k becomes available, the importance weight of each particle is updated based on how well the particle's predicted state matches the measurement, according to the measurement likelihood function $p(y_k|x_{k,i})$, often derived from the observation model h and some known measurement noise distribution R. For example: Assuming this distribution is known, the weights are calculated, and then normalized:

$$\tilde{w}_{k,i} = w_{k|k-1,i}p(y_k|x_{k,i})$$
$$w_{k,i} = \frac{\tilde{w}_{k,i}}{\sum_{j=1}^{N} \tilde{w}_{k,j}}$$



(a) Open-Loop System Model (b) Closed-Loop System Model

Figure 4.4: System Models: Open-Loop and Closed-Loop

The state estimate $\hat{x}_k$ can be computed as the weighted mean of the particles:

$$\hat{x}_k = \sum_{i=1}^{N} w_{k,i} x_{k,i}$$

Resampling Step: A key issue in PFs is particle degeneracy, where after a few iterations, most particles have negligible weights. To mitigate this, a resampling step is performed. Using the current weighted set of particles as a discrete approximation of the probability density function, a new set of weighted particles N are drawn, and the weights normalized. The exact algorithm used for resampling varies between implementations and we don't seek to provide an exhaustive list of examples here. Common resampling methods include stratified [85], residual [85], and systematic [71].

The main drawback of Particle Filters is their computational cost, which scales with the number of particles N. A large N is often required for good performance, especially in high-dimensional state spaces, making them significantly more demanding than KF, EKF, or UKF.

4.4 System Model

4.4.1 System Definition

Consider a nonlinear system, pictured in Figure 4.4. The system consists of a plant, a state estimator, and, (optionally) a feedback controller. The plant receives an input u either from

the control block or from an external source. This is forwarded to the state estimator as well. We assume perfect, but not clairvoyant, knowledge of the control. Inside the plant is the true state x. The plant has an output, z, which is fed into a state estimator which outputs the estimated state $\hat{x}$.

When the control feedback is absent, we say the system is open-loop. If the system uses feedback, the estimated state $\hat{x}$ is fed into the control block, and we say the system is closed-loop. Without loss of generality, the setpoint is omitted from the diagram and assumed to be zero.

This is inline with the traditional control framework, but we add an additional aspect: the state estimator is dual-mode and can operate in either linear (KF) or nonlinear (EKF, UKF, or PF) mode.

$$\dot{x}(t) = f(x(t)) + g(x(t))d(t)$$

where $f : \mathbb{R}^n \to \mathbb{R}^{n\times m}$ and $g : \mathbb{R}^n \to \mathbb{R}^{n\times m}$.

Consider also the linearized equivalent of the system. We can approximate the state by linearizing around x_0, resulting in the approximate system

$$\dot{\hat{x}}(t) = A\hat{x}(t) + Bd(t)$$

Where $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times m}$ such that the two systems are equivalent at the linearization point.

$$\begin{aligned} Ax_0 &= f(x_0) \\ B &= g(x_0) \end{aligned}$$

As the system evolves, the nonlinear model and the linear approximate may diverge. By computing the difference between the nonlinearly modeled state x and linearly modeled state estimate $\hat{x}$, we can measure the linearization error as the system evolves over time.

$$E = \hat{x} - x$$

To aid in modeling this error, we combine the true state and the approximated state into an expanded state-space.

$$\begin{aligned} \tilde{x} &= [\, x \;\; \hat{x} \,] \\ x, \hat{x} &\in \mathbb{R}^n \\ \tilde{x} &\in \mathbb{R}^{2n} \end{aligned}$$

When the estimate is produced by a linear estimator, we can couple the nonlinear and linear dynamics of our system to operate in parallel on our expanded statespace $\tilde{x}$.

$$\dot{\tilde{x}} = \begin{bmatrix} \dot{x} \\ \dot{\hat{x}} \end{bmatrix} = \begin{bmatrix} f(x) + g(x)d \\ A\hat{x} + Bd \end{bmatrix}$$

4.4.2 Reachable Sets

We also define a reachable set of a system as the set R of all states that are reachable from some starting state x_0 under admissible input disturbance $d(\cdot) \in K$ within some time t. K is the set of admissible input functions.

$$\begin{aligned} &R\{x_0, K, t\} = \\ &\{x | \forall \tau, 0 \leq \tau \leq t, d(\tau) \in K, x(\tau) = x, x(0) = x_0\} \end{aligned} \tag{4.5}$$

The size of any reachable set is monotonically increasing with respect to t and the size of K, i.e., given more time and larger inputs or disturbances, the reachable space will never shrink. In Section 4.5.5, we discuss how to compute a reachable set.

The reachable set of our augmented linear/nonlinear system in Eq 4.4.1 also has an naturally associated maximum linearization error.

$$E\{R\} = \max_{\tilde{x} \in R\{\tilde{x}_0, K, t\}} |E(\tilde{x})$$

This linearization error of the reachable set is also monotonically increasing with respect to the bound time t and the admissible inputs K. If two of the three are known, the third can be computed.

4.4.3 Requirements for System

We impose some limits on the system.

- **Finite reachability** The system must have some restoring mechanism, either intrinsic or through closed loop feedback, such that for any admissible set $K \subset \mathbb{R}^m$ of the input $u(\cdot) \in \mathbb{R}^m$, the set of all reachable states $R\{x_0, K, \infty\}$ has a finite volume.

  $$\forall K \subset \mathbb{R}^m | R\{x(0), K, \infty\} \subset \mathbb{R}^N$$

  In the rest of this chapter, we may abbreviate $R\{x_0, K, \infty\}$ as $R_\infty\{K\}$ or just R_∞.

- **Restorability** The system must be able to return to initial conditions from any state under the same admissible input that it reached that state $\forall x \in R_\infty | \exists u(\cdot)$ s.t. $u(t) \in K, x(0) = x, x(t) = x_0$

Theorem 4. *If the system has finite reachability for some admissible set of inputs K, then the resulting $R_\infty\{K\}$ is robust invariant*

Proof. By counterexample. To be robust invariant means that for any initial point $x_0 \in R\{x_0, K, \infty\}$, all points $x(t)$ reachable under admissable input function $\forall t | u(t) \in K$ are also in $R\{x_0, K, \infty\}$.

$$\begin{aligned} \forall t, \forall \tau \in [0,t], u(\tau) \in K, \\ x_0 \in R\{x_0, K, \infty\} \\ \implies x(t) \in R\{x_0, K, \infty\} \end{aligned}$$

Suppose that R_∞ isn't robust invariant, so that there exists a point $x_b \notin R_\infty$ that is reachable from $x_a \in R_\infty$. By the definition of finite reachability above, x_a is reachable from x_0 for some $t \geq 0$. Therefore, x_b is reachable from x_0 by way of x_a, and therefore $x_b \in R_\infty$. Contradiction. □

4.5 Enforcing Safety

4.5.1 Construction of Barrier Certificates

We want to establish an invariant set L that forms a *linear region*, where the system uses a linear estimator. We also want to establish an safeset $\mathcal{C}$ of all states $\tilde{x} \in \mathcal{C}$ where $x \approx \hat{x}$.

The safeset $\mathcal{C}$ can be defined using a barrier certificate [99].

A barrier certificate $B(x) : \mathbb{R}^n \to \mathbb{R}$ is a level set function that indicates which states belong to an associated set. When the barrier certificate evaluates to nonpositive for a state, this certifies the state is in the associated set.

$$\begin{aligned} B(x) \leq 0 &\implies x \in \mathcal{X} \\ B(x) > 0 &\implies x \notin \mathcal{X} \end{aligned}$$

We construct a barrier certificate where the linearization error $E = \hat{x} - x$ is within some bound ϵ

$$B(\tilde{x}) = ||\hat{x} - x|| - \epsilon$$

It's important to point out that this barrier certificate equation is relatively arbitrary. Finer error bounds can be imposed on individual state elements that have more stringent tolerances. If a control barrier function [5] must also be imposed on the true state x for purposes of enforcing safe regions, or if regions need to be excluded for purposes of closed-loop stability, these can be incorporated into the barrier certificate.

There is naturally a manifold at $\mathcal{M} = \{\tilde{x}|B(\tilde{x}) = 0\}$. This manifold forms a bound between the safeset $\mathcal{C}$ and the unsafe set $\mathcal{C}^C$. We define the safeset to be the region where the barrier certificate is nonpositive, which certifies that $\hat{x} \approx x$.

$$\mathcal{C} = \{\tilde{x}|B(\tilde{x}) \leq 0\}$$

4.5.2 Linear Region

In this section, we will define a *linear region* L. This is naturally a subset of $\mathcal{C}$, and defines the region where it is safe to operate the system using linear estimation. When the system leaves this region, a transition to nonlinear state estimation is triggered, so the estimate converges to the true state. It must be designed such that $\tilde{x}$ never leaves the safeset $\mathcal{C}$, even when in nonlinear mode.

To define the linear region, we consider the parameters that can inform the decision to transition to nonlinear estimation.

For a statespace $x \in R^n$, the linear region $L \subseteq C \subset R^{2n}$ exists in a higher dimensional statespace, because it depends on the linear and nonlinear dynamics of the system. In our usecase, only an estimate of the linear state is known until the decision is made to switch to a nonlinear estimator. The true underlying state x is governed by nonlinear dynamics and is unknown.

The challenge then is to safely make decisions while only observing half the dimensions of $\tilde{x}$. Suppose that the linear region is equal to some invariant reachable set over an infinite time window.

$$L = R\{x_0, K, \infty\}$$

We reemphasize that once the system is in a reachable set R_∞, it will never leave while the input remains within the admissible set K.

It is possible that a system may briefly have an input outside the designated bounds without actually leaving R_∞. Ideally, we would want to impose some predicate on the state estimate $\hat{x}$ such that if the predicate holds true, then temporary violations of the input constraint are tolerated and the system remains in linear mode. However, this isn't possible.

Theorem 5. *There exists no predicate $\phi(\hat{x})$ such that $\phi(\hat{x})$ holding true guarantees $\tilde{x} \in R_\infty = R\{\tilde{x}_0, K, \infty\}$, if the input $d(t)$ is allowed to deviate from the admissible set K at any time.*

Formally: If $\exists \tau, d(\tau) \notin K$ then $\forall t \geq \tau, \nexists \phi(\cdot)$ such that $\phi(\hat{x}(t)) \implies \tilde{x}(t) \in R_\infty$

Proof. We prove this by contradiction. Assume such a predicate $\phi(\hat{x})$ exists. This means that if $\phi(\hat{x}(t))$ is true, then $\tilde{x}(t) \in R_\infty$, even if $d(\tau) \notin K$ for some τ.

Consider a system with the augmented state dynamics as described in Eq. 4.4.1:

$$\dot{\tilde{x}} = \begin{bmatrix} \dot{x} \\ \dot{\hat{x}} \end{bmatrix} = \begin{bmatrix} f(x) + g(x)d \\ A\hat{x} + Bd \end{bmatrix}$$

Without loss of generality, let the linearization point for A and B be $x_0 = \mathbf{0}$. Let $R_\infty = R\{\tilde{x}_0, K, \infty\}$ be defined for a specific compact set of admissible inputs K.

Construct a specific polynomial input function $g(x)$. For instance, let a component of $g(x)$ be x_j^n for some state component x_j and an arbitrarily large integer $n > 1$. Then the corresponding component B_i in $B = g(x_0)$ would be $\frac{\partial}{\partial x_j} x_j^n \big|_{x_0} = 0$.

Now, consider the following scenario:

1. Initialize the system at $\tilde{x}(0) \in R_\infty$ such that $\phi(\hat{x}(0))$ is true.
2. Let the system evolve for some time with $d(t) \in K$, such that $\phi(\hat{x}(t))$ remains true and $\tilde{x}(t) \in R_\infty$.

3. At some time τ, apply an input $d(\tau) = d_{spike}$ such that $d_{spike} \notin K$.

4. For $t > \tau$:

   - The estimated state $\hat{x}$ evolves according to $\dot{\hat{x}} = A\hat{x} + Bd$. If $B_i = 0$ for the term involving $x_j^n d_i$, then $\hat{x}$ will show little to no direct response to d_{spike} through this channel. It's possible to choose A, B, and ϕ such that $\hat{x}(t)$ remains in a region where $\phi(\hat{x}(t))$ is true (e.g., $\hat{x}$ remains small).
   - The true state x evolves according to $\dot{x} = f(x) + g(x)d$. The term $x_j^n d_{spike}$ can cause a significant deviation in x_j, even if x_j was small prior to τ, especially if n is large. This can drive $x(t)$ to a state that, when paired with $\hat{x}(t)$, forms an augmented state $\tilde{x}(t)$ that is no longer in R_∞. This is because R_∞ is the set of states reachable *only* with inputs from K. By applying $d_{spike} \notin K$, the trajectory is no longer constrained to R_∞.

The ability to choose an arbitrarily large n for the nonlinear term x_j^n ensures that for any predicate $\phi(\hat{x})$ that defines a bounded region for $\hat{x}$, and for any K, there exists some $d_{spike} \notin K$ and n such that x diverges significantly from $\hat{x}$ making $\tilde{x}$ leave R_∞, while $\phi(\hat{x})$ remains true. Therefore, no such predicate $\phi(\hat{x})$ can exist. □

When in linear mode, the decision to transition to nonlinear must incorporate the input signal, and not the state estimate.

For a given system and a specific $g(x)$, it's conceivable an ad-hoc predicate exists. However since no general predicate exists, we aim to create a general definition of the linear region that only considers the input signal in decision making.

In the next section, we will demonstrate how to compute K for the linear region $L = R_\infty\{K\}$.

4.5.3 Transitioning to Nonlinear

When making the decision to switch to nonlinear estimation, we must consider the time to transition from linear estimation to nonlinear estimation.

Optimistically, this transition time $\mathcal{T}$ would be equal to the sample period of the estimators. However, we also consider the possibility that the transition may require additional computation, initialization, and memory operations that last over multiple sample periods.

Suppose our linear region L is equal to some reachable set $R_\infty\{K\}$. As laid out above, if $u(\cdot)$ remains within some K for $\tau \in [0, t]$ and $x(0) \in R_\infty$, then we can guarantee that $\forall t \in [0, \infty] | x(t) \in R_\infty$.

Then to compute the linear region, we need to compute a set R_∞ where the decision can be made to transfer to nonlinear estimation based exclusively on the input u while taking into account a transition time $\mathcal{T}$, and guaranteeing no violations of the barrier certificate $B(\tilde{x})$.

First we quantify the valid states:

$$C = \{\tilde{x} | B(\tilde{x}) \leq 0\}$$

This contains the green and yellow regions in Fig 4.5.

Now we consider the transition time $\mathcal{T}$. This is the time between a violation of K and when the nonlinear state estimator comes online. A buffer zone is added within the error manifold $\mathcal{M} = \{\tilde{x} | B(\tilde{x} = 0)\}$ to account for this transition time. This can be found using a *Backwards Reachable Set* (BRS), a concept that will be covered more thoroughly in Section 4.4.2. To compute the BRS of a set of states, we need to impose a lookback time (naturally $\mathcal{T}$), and some limits $\mathcal{K}$ on the input to the system d. This is distinct from the K denoting the region of linearity discussed previously; $\mathcal{K}$ is informed by the physical constraints of the system, rather than any limits of linearity. It denotes is the largest magnitude of inputs that our system can be expected to ever experience.

The BRS contains all states than can reach some specified manifold in the given amount of time, under those input constraints. This is indicated by the green and yellow striped region in Fig 4.5.

We now define a safe zone

$$S = \{x | x \in C \wedge x \notin BRS(\mathcal{M})\}$$

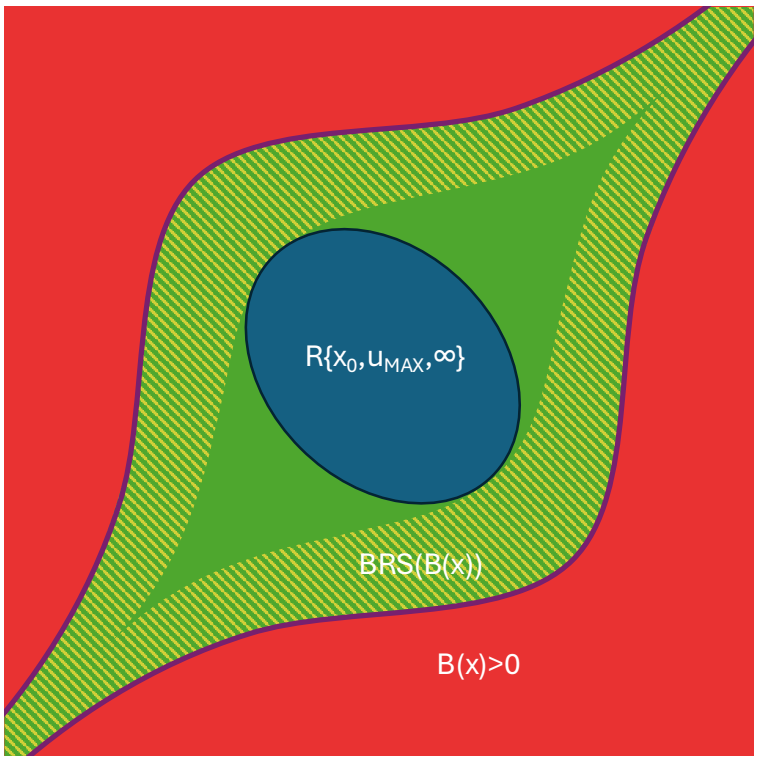


Figure 4.5: R_∞ in blue, $\mathcal{C}$ in green, S in solid green, BRS in yellow, invalid points in red

S is all the states in C that aren't in the BRS of the error manifold $\mathcal{M}$. This is indicated by the solid green region in Fig 4.5. It is possible that $S = \emptyset$, in which case, there is no region of linearity that we can safely transition from in time. If this is the case, design your system to minimize the transition time T, or limit the admissible inputs $\mathcal{K}$ to the system to be less aggressive.

All states $\tilde{x} \in S$ can safely transition from linear to nonlinear if the determination is made to do so. However, as stated previously, we cannot make this determination on the knowledge of the states themselves. We need to compute a set of admissible inputs K such that

$$\max_K |R_\infty\{K\}| \quad \text{subject to} \quad R_\infty\{K\} \subseteq S$$

Which is to say, find the largest decision input such that the reachable set of the bound over an infinite amount of time creates a subset of the safe set S. This reachable set is shown in blue in Fig 4.5. **This $R_\infty\{K\}$ is our linear region.**

It is possible that after the nonlinear transition is made, the state $\tilde{x}$ will still find itself in S or even R_∞. But it will never find itself outside of C, guaranteeing we remain within the error bounds.

4.5.4 Transitioning to Linear

Eventually, a determination will be made that we need to transition back to a linear estimator. This is much easier as we can make use of the state estimate, since the estimate more faithfully follows the behavior of system.

For a nonlinear estimator, the assumption is made that $\hat{x} \approx x$, subject to estimate variance, discussed later in Section 4.6. Therefore, $[\hat{x}\ \hat{x}] \approx \tilde{x}$, and we can draw the conclusion that when $\hat{x}$ is produced by a nonlinear state estimator:

$$[\hat{x}\ \hat{x}] \in R'_\infty \implies \tilde{x} \in R'_\infty$$

Where $R'_\infty \subset R_\infty$ is a smaller linear region that is computed just as described above, but using a different transition time $\mathcal{T}'$ to compute the BRS, where $\mathcal{T}' \geq \mathcal{T}$. This new transition time $\mathcal{T}'$ represents not only the time to transition to linear, but also the time to immediately transition back to nonlinear if needed. If the state estimators continue to operate during the transition, and the transition can be aborted, then $\mathcal{T}' = \mathcal{T}$.

We also impose the restriction that $d(t) \in K$, to prevent immediately switching back to nonlinear.

These two conditions are sufficient to determine the system is safely within the linear region to warrant a switch back to linear estimation.

- $d(t) \in K$
- $[\hat{x}\ \hat{x}] \in R'_\infty$

4.5.5 Computing Reachable Sets

In order to derive an equation for the manifold encompassing the reachable set R_∞ and the BRS, we turn to the Hamilton-Jacobi (HJ) [8,16,32,79] partial differential equation, used in optimal nonlinear control.

$$\begin{aligned} \frac{\delta V}{\delta t} + H(\nabla_x V, x) &= 0 \\ H(\nabla_x V, x) &= \max_{u \in K} \nabla_x V \cdot \Psi(x, u) \end{aligned} \tag{4.6}$$

$V(x,t)$ is our level set function. It defines the boundary of our reachable set $R(t) = \{x | V(x,t) \leq 0\}$. The Hamiltonian $H(\nabla_x V, x)$ governs the maximum possible outward velocity of this manifold. $\nabla_x V$ is the gradient normal to the manifold. The level set is initialized to a signed distance function centered at x_0 (when solving numerically, a small offset ϵ is used, creating an "inital set" forming a small hypersphere around x_0).

$$V(x, 0) = ||x - x_0|| - \epsilon$$

Solving Eq 4.6 to derive $V(x,t)$ gives us the evolution of the forward reachable set over time. Then

$$R_\infty = \{x | V(x, \infty) \leq 0\}$$

To compute the backwards reachable set, we use a similar partial differential equation (Eq 4.7). The key difference is that we are now minimizing the Hamiltonian, rather than maximizing it. Additionally, we initialize the level set at the terminal time $\mathcal{T}$ and evolve it backwards in time.

$$\begin{aligned} \frac{\delta V(x,t)}{\delta t} + H(\nabla_x V, x) &= 0 \\ H(\nabla_x V, x) &= \min_{u \in K} (\nabla_x \phi) \cdot \Psi(x,u) \\ V(x, \mathcal{T}) &= B(x) \end{aligned} \tag{4.7}$$

Our level set function $V(x,t)$ is now a value function representing the cost of controlling the system optimally from x to the manifold of $h(x) = 0$ from time t to terminal time $\mathcal{T}$.

$V(x, \mathcal{T})$ is initialized to equal our barrier certificate $B(x)$. This level set indicates the error manifold $\mathcal{M} = \{x | B(x) = 0\}$. To find the inside bound of the BRS, we solve the HJB PDE to derive $V(x,t)$. Then $V(x,0)$ is the inner bound. The BRS is all states that are certified by the barrier certificate, but not by the value set $V(x,0)$.

$$BRS = \{x | V(x,0) > 0 \wedge B(x) \leq 0\}$$

No closed form solution exists for the HJ partial differential equation, so numerical analysis tools [26, 80] are required to approximate the reachability envelope for a system. These use level-set methods, which are highly memory intensive and scale exponentially with state dimensionality. For this reason, it is recommended to instead use random sampling and machine learning for high-order systems [90, 109].

4.6 Application to State Estimators

We now apply all the above work to the problem of transitioning from linear to nonlinear state estimation, and vice versa. We begin by formulating the control semantics in Section 4.6.1 and then address how appropriate scheduling parameters can be calculated in Section 4.6.2.

4.6.1 Control Semantics

As stated previously, the decision to switch to a nonlinear estimator is made considering solely the input to the system.

When the input $u(t) \notin K$, then no guarantees can be made that we will not enter the BRS on the next sample, so we cannot guarantee remaining within the control barrier function (CBF) $h(x) \geq 0$.

Near the initial state, our system uses a Kalman Filter (KF) that models the system according to linear dynamics. It returns a state estimate $x \in \mathbb{R}^n$ and an accompanying covariance matrix $P \in \mathbb{R}^{n \times n}$ to model the uncertainty of the estimate, which is assumed to be gaussian.

When a switch to nonlinear estimation is triggered, an Extended Kalman Filter is initialized using the latest state estimate and covariance of the Kalman Filter, and the estimate of the EKF immediately begins converging to the true state.

In the nonlinear mode, we assume that $x \approx \hat{x}$ subject to the variance in the estimate. To account for this uncertainty, a window can be placed around each state element encompassing an arbitrary confidence interval.

$$x_i \pm z_i \sqrt{P_{ii}}$$

Where z_i is the number of standard deviations away from the mean estimate.

When this confidence window is entirely with the region encompassed by R_∞, we say the system has returned to linearity.

$$[\hat{x} \pm z\sqrt{\text{diag}(P)} \quad \hat{x} \pm z\sqrt{\text{diag}(P)}] \in R_\infty$$

Once the input is also valid $d \in K$, we then transition back to linear estimation. A KF is re-initialized using the latest state estimate and estimate covariance from the nonlinear estimator.

When the nonlinear estimator is a Unscented Kalman Filter or a Particle Filter, determining whether or not the estimate is within the confidence interval is a much simpler process of counting the particles or the sigma points.

In the case of the UKF, each sigma point corresponds to some fraction or whole of a standard deviation away from the mean. If both the 1σ points are within the linear region, then we can assert the estimate is also within linear region with 68% certainty. To achieve 99.7% certainty, then the 3σ points must be within the linear region, and so on.

In the case of Particle Filters, the problem is even more straight-forward. For any state variable $x_k[d]$ (the d-th dimension of the state), the particles $\{x_{k,i}\}_{i=1}^{N}$ can be sorted according to the values of $x_{k,i}[d]$. An arbitrary confidence interval $1-\alpha$ can be obtained by finding the values corresponding to the $\alpha/2$ and $1-\alpha/2$ percentiles of the sorted particles (using the weights of the particles in the calculations).

4.6.2 Scheduling Semantics

By parameterizing our control framework with a transition time, it can be considered an implementation of prior literature [108] on scheduling mode changes in mixed criticality systems. We adapt their work in Figure 4.6.

We consider the case of discrete sampling, where our state estimator runs at some specified period $\mathbf{T}$. There is also an additional actuation delay $\mathbf{R}$, representing the delay from when a sample is taken to when it impacts the physical actuation of a system. Depending on the system model used, this delay may be inconsequential. If we only consider the accuracy of the state estimate, without concern for where and how it may be used, then we can set $\mathbf{R} = 0$, without loss of generality.

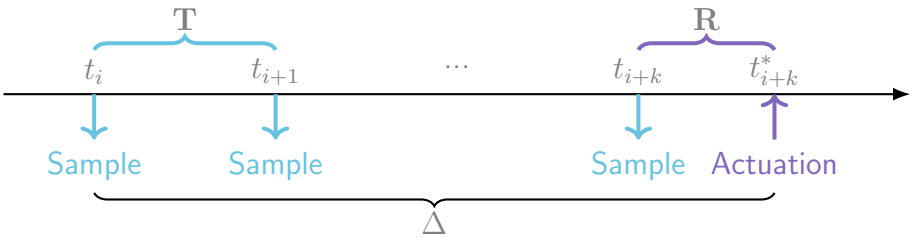


Figure 4.6: Transition time over discrete sampling periods

At minimum the time Δ to transition from linear estimation to nonlinear estimation will take one period.

$$\Delta \geq \mathbf{T}$$

The case $\Delta = \mathbf{T}$ considers the case where the actual transition can be done immediately, and Δ only considers the time to discover the conditions requiring a transition. For instance, if an input violation happens an infinitesimally small time after one sample, then it will not be discovered until the next sample, $\mathbf{T}$ time later.

The total time to transition is then

$$\Delta = k\mathbf{T} + \mathbf{R}$$

Where k is some integer constant representing the number of sampling periods needed to load and initialize a nonlinear estimation component.

4.7 Conclusions and Future Work

This paper introduces a novel framework for elastic scheduling of state estimators, enabling a robust and safe handoff between linear and nonlinear estimation techniques in safety-critical systems. By defining an augmented state-space that explicitly incorporates linearization error, and leveraging barrier certificates, we established a formal definition of a "safeset" where linear approximations are acceptable.

The core contribution lies in the methodology for determining when to transition between linear and nonlinear estimation. We demonstrated that, to ensure safety, the switch from a linear to a nonlinear estimator must be predicated on system inputs exceeding predefined bounds. This input bound is derived from the computation of a forward reachable set, which defines a "linear region" that remains invariant when the bounds are respected. This linear region is formed as being the largest such invariant set that allows a safety buffer from the

edge of the safeset. This buffer is computed using Hamilton-Jacobi methods to compute a backwards reachable set.

For the transition back to linear estimation, the framework allows for the use of the more accurate nonlinear state estimate, incorporating its uncertainty, to confirm that the system has returned to a stable region of linearity and that inputs are within the prescribed safe operating limits.

Future work includes the implementation of this framework for use in designing real-world systems. Given some defined system dynamics and error function, a program could be make to automatically compute the BRS of the error function. Then the forward reachable set could be found by trial-and-error, treating the admissible set of inputs K as a tunable parameter. If K is a continuous range in $\mathbb{R}$, then this is as simple as tuning the limits of K down so that the linear region never overlaps the BRS, and then tuning it up to maximize volume.

The representation of the linear region and BRS is flexible. Typical tools for level set methods [26, 80] use a discrete grid sampling some finite portion of the statespace. A state coordinate can be determined to be in the linear region or the BRS by assessing the negativity (in the case of the linear region) or the positivity (in the case of the BRS) of the respective value function at the state coordinate.

This discrete sampling grid is highly memory intensive, especially for higher-dimensional systems. Instead other methods [90, 109] use random Markov sampling to train a deep learning model on the value function.

Using either these AI models, or a discrete grid representing the value function, a decision procedure can be constructed to test whether a given state is in the linear region or the BRS. The process of implementing a tool to automatically construct these decision procedures and compute the associated input bound on the linear region remains fruitful future work.

Future work could also extend this framework for different scenarios. For instance

- Using the system dynamics to inform control barrier functions [5] that permit temporary transgressions of the linear region's input bounds, while still guaranteeing safety.

- Explore scheduling semantics of accelerated applications with mixed-devices. GPUs are demonstrably better at linear operations like the Kalman Filter, but FPGAs are likely better at nonlinear operations like the Extended Kalman Filter. Comingling these two types of devices would incur extra memory synchronization and possibly a longer handover window, but may be neccessary to operate safely in nonlinear regions.

References

[1] A. AI. Autoware reference system. https://github.com/ros-realtime/reference-system, 2023.

[2] AMD. L2 api benchmark, Aug 2022.

[3] AMD. Vitis blas. https://github.com/Xilinx/Vitis_Libraries/tree/2022.1/blas, 2022.

[4] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: hidden details revealed. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pages 104–115. IEEE Computer Society, 2017.

[5] A. D. Ames, S. Coogan, M. Egerstedt, G. Notomista, K. Sreenath, and P. Tabuada. Control barrier functions: Theory and applications. In *17th European Control Conference, ECC 2019, Naples, Italy, June 25-28, 2019*, pages 3420–3431. IEEE, 2019.

[6] A. A. Arafat, S. Vaidhun, K. M. Wilson, J. Sun, and Z. Guo. Response time analysis for dynamic priority scheduling in ROS 2. In R. Oshana, editor, *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, pages 301–306. ACM, 2022.

[7] M. S. Arulampalam, S. Maskell, N. J. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *IEEE Trans. Signal Process.*, 50(2):174–188, 2002.

[8] S. Bansal, M. Chen, S. L. Herbert, and C. J. Tomlin. Hamilton-jacobi reachability: A brief overview and recent advances. In *56th IEEE Annual Conference on Decision and Control, CDC 2017, Melbourne, Australia, December 12-15, 2017*, pages 2242–2253. IEEE, 2017.

[9] T. D. Barfoot. *State Estimation for Robotics*. Cambridge University Press, 2017.

[10] C. Bays. A comparison of next-fit, first-fit, and best-fit. *Commun. ACM*, 20(3):191–192, 1977.

[11] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *22nd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2016, Daegu, South Korea, August 17-19, 2016*, pages 159–169. IEEE Computer Society, 2016.

[12] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *J. Syst. Archit.*, 80:104–113, 2017.

[13] O. Bell, C. Gill, and X. Zhang. Hardware acceleration with zero-copy memory management for heterogeneous computing. In *29th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2023, Niigata, Japan, August 30 - Sept. 1, 2023*, pages 28–37. IEEE, 2023.

[14] O. Bell, A. Kumar, and C. Gill. Host-based allocators for device memory. *CoRR*, abs/2405.07079, 2024.

[15] O. Bell, H. Teper, M. Günzel, C. Gill, and J. Chen. Fixed-priority and edf schedules for ros 2 graphs on uniprocessor. Submitted to RTNS 2025, under review, 2025.

[16] R. Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[17] M. Bertogna and S. K. Baruah. Limited preemption EDF scheduling of sporadic task systems. *IEEE Trans. Ind. Informatics*, 6(4):579–591, 2010.

[18] T. Betz, M. Schmeller, H. Teper, and J. Betz. How fast is my software? latency evaluation for a ROS 2 autonomous driving software. In *IEEE Intelligent Vehicles Symposium, IV 2023, Anchorage, AK, USA, June 4-7, 2023*, pages 1–6. IEEE, 2023.

[19] R. Bi, X. Liu, J. Ren, P. Wang, H. Lv, and G. Tan. Efficient maximum data age analysis for cause-effect chains in automotive systems. In *DAC*. ACM, 2022.

[20] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg. A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance. In *42nd IEEE Real-Time Systems Symposium, RTSS 2021, Dortmund, Germany, December 7-10, 2021*, pages 41–53. IEEE, 2021.

[21] T. Blaß, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg. Automatic latency management for ROS 2: Benefits, challenges, and open problems. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2021, Nashville, TN, USA, May 18-21, 2021*, pages 264–277. IEEE, 2021.

[22] J. Blum and F. Drewes. Language theoretic properties of regular DAG languages. *Inf. Comput.*, 265:57–76, 2019.

[23] R. Bonatti, C. Ho, W. Wang, S. Choudhury, and S. A. Scherer. Towards a robust aerial cinematography platform: Localizing and tracking moving targets in unstructured environments. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2019, Macau, SAR, China, November 3-8, 2019*, pages 229–236. IEEE, 2019.

[24] R. Bonatti, W. Wang, C. Ho, A. Ahuja, M. Gschwindt, E. Camci, E. Kayacan, S. Choudhury, and S. A. Scherer. Autonomous aerial cinematography in unstructured environments with learned artistic decision-making. *J. Field Robotics*, 37(4):606–641, 2020.

[25] R. Bonatti, Y. Zhang, S. Choudhury, W. Wang, and S. A. Scherer. Autonomous drone cinematographer: Using artistic principles to create smooth, safe, occlusion-free trajectories for aerial filming. In J. Xiao, T. Kröger, and O. Khatib, editors, *Proceedings of the 2018 International Symposium on Experimental Robotics, ISER 2018, Buenos Aires, Argentina, November 5-8, 2018*, volume 11 of *Springer Proceedings in Advanced Robotics*, pages 119–129. Springer, 2018.

[26] M. Bui, G. Giovanis, M. Chen, and A. Shriraman. Optimizeddp: An efficient, user-friendly library for optimal control and dynamic programming, 2022.

[27] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*, volume 24 of *Real-Time Systems Series*. Springer, 2011.

[28] G. C. Buttazzo, M. Bertogna, and G. Yao. Limited preemptive scheduling for real-time systems. A survey. *IEEE Trans. Ind. Informatics*, 2013.

[29] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg. Response-time analysis of ROS 2 processing chains under reservation-based scheduling. In S. Quinton, editor, *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany*, volume 133 of *LIPIcs*, pages 6:1–6:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[30] R. Cavicchioli, N. Capodieci, M. Solieri, and M. Bertogna. Novel methodologies for predictable cpu-to-gpu command offloading. In S. Quinton, editor, *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany*, volume 133 of *LIPIcs*, pages 22:1–22:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[31] J. Chen, G. Nelissen, W. Huang, M. Yang, B. B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. C. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real Time Syst.*, 2019.

[32] M. Chen and C. J. Tomlin. Hamilton-jacobi reachability: Some recent theoretical advances and applications in unmanned airspace management. *Annu. Rev. Control. Robotics Auton. Syst.*, 1:333–358, 2018.

[33] H. Choi, Y. Xiang, and H. Kim. Picas: New design of priority-driven chain-aware scheduling for ROS 2. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2021, Nashville, TN, USA, May 18-21, 2021*, pages 251–263. IEEE, 2021.

[34] Clearpath Robotics. Turtlebot 4 (ros 2 robot), 2022.

[35] Cogniteam. Hamster (ros 2 robot), 2023.

[36] J. Condori, A. Maghareh, J. Orr, H.-W. Li, H. Montoya, S. Dyke, C. Gill, and A. Prakash. Exploiting parallel computing to control uncertain nonlinear systems in real-time. *Experimental Techniques*, 44:735–749, 2020.

[37] J. Condori, A. Maghareh, J. Orr, H. W. Li, H. Montoya, S. Dyke, C. Gill, and A. Prakash. Exploiting parallel computing to control uncertain nonlinear systems in real-time. *Experimental Techniques*, 44(6):735–749, 2020.

[38] S. S. Craciunas, C. M. Kirsch, H. Payer, A. Sokolova, H. Stadler, and R. Staudinger. A compacting real-time memory management system. In R. Isaacs and Y. Zhou, editors, *Proceedings of the 2008 USENIX Annual Technical Conference, USENIX ATC 2008, Boston, MA, USA, June 22-27, 2008. Proceedings*, pages 349–362. USENIX Association, 2008.

[39] A. Davare, Q. Zhu, M. D. Natale, C. Pinello, S. Kanajan, and A. L. Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, pages 278–283. IEEE, 2007.

[40] R. I. Davis and N. Navet. Controller area network (CAN) schedulability analysis for messages with arbitrary deadlines in FIFO and work-conserving queues. In T. Nolte and A. Willig, editors, *9th IEEE International Workshop on Factory Communication Systems, WFCS 2012, Lemgo, NRW, Germany, May 21-24, 2012*, pages 33–42. IEEE, 2012.

[41] R. I. Davis, A. Thekkilakattil, O. Gettings, R. Dobrin, S. Punnekkat, and J. Chen. Exact speedup factors and sub-optimality for non-preemptive scheduling. *Real Time Syst.*, 54(1):208–246, 2018.

[42] S. Dinh, J. Li, K. Agrawal, C. D. Gill, and C. Lu. Blocking analysis for spin locks in real-time parallel tasks. *IEEE Trans. Parallel Distributed Syst.*, 29(4):789–802, 2018.

[43] M. Dürr, G. von der Brüggen, K. Chen, and J. Chen. End-to-end timing analysis of sporadic cause-effect chains in distributed systems. *ACM Trans. Embed. Comput. Syst.*, 18(5s):58:1–58:24, 2019.

[44] Epic Games. Unreal engine.

[45] eProsima. Fastdds. https://github.com/eProsima/Fast-DDS, 2016.

[46] D. Ferry, G. Bunting, A. Maghareh, A. Prakash, S. Dyke, K. Agrawal, C. D. Gill, and C. Lu. Real-time system support for hybrid structural simulation. In T. Mitra and J. Reineke, editors, *2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014*, pages 25:1–25:40. ACM, 2014.

[47] J. C. Fonseca, V. Nélis, G. Raravi, and L. M. Pinho. A multi-dag model for real-time parallel applications with conditional execution. In R. L. Wainwright, J. M. Corchado, A. Bechini, and J. Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1925–1932. ACM, 2015.

[48] E. Foundation. Cyclonedds. https://github.com/eclipse-cyclonedds/cyclonedds, 2019.

[49] E. Foundation. Iceoryx. https://github.com/eclipse-iceoryx/iceoryx, 2019.

[50] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Research Report RR-2966, INRIA, 1996. Projet REFLECS.

[51] P. Gohari, M. Nasri, and J. Voeten. Data-age analysis for multi-rate task chains under timing uncertainty. In *RTNS*. ACM, 2022.

[52] M. Günzel, K. Chen, N. Ueter, G. von der Brüggen, M. Dürr, and J. Chen. Timing analysis of asynchronized distributed cause-effect chains. In *RTAS*. IEEE, 2021.

[53] M. Günzel, K. Chen, N. Ueter, G. von der Brüggen, M. Dürr, and J. Chen. Compositional timing analysis of asynchronized distributed cause-effect chains. *ACM Trans. Embed. Comput. Syst.*, 2023.

[54] M. Günzel, H. Teper, K. Chen, G. von der Brüggen, and J. Chen. On the equivalence of maximum reaction time and maximum data age for cause-effect chains. In *ECRTS*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[55] M. Günzel, N. Ueter, K. Chen, and J. Chen. Timing analysis of cause-effect chains with heterogeneous communication mechanisms. In *RTNS*. ACM, 2023.

[56] M. Günzel, N. Ueter, K. Chen, G. von der Brüggen, and J. Chen. Probabilistic reaction time analysis. *ACM Trans. Embed. Comput. Syst.*, 2023.

[57] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst. Communication centric design in complex automotive embedded systems. In M. Bertogna, editor, *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*, volume 76 of *LIPIcs*, pages 10:1–10:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[58] M. Harris. Fast, flexible allocation for nvidia cuda with rapids memory manager, Dec 2020.

[59] M. Hidalgo, C. Lalancette, M. Belanger, and C. Bedard, Jul 2023.

[60] M. Hidalgo, S. Loretz, C. Lalancette, M. Jeromino, M. Mei, M. Belanger, and C. Bedard. About quality of service settings, May 2025.

[61] M. Hidalgo, I. Paunovic, C. Lalancette, and E. Weon. Internal ros 2 interfaces, May 2025.

[62] M. Hidalgo, I. Paunovic, C. Lalancette, and E. Weon. Setting up efficient intra-process communication, May 2025.

[63] D. S. Hirschberg. A class of dynamic memory allocation algorithms. *Commun. ACM*, 16(10):615–618, 1973.

[64] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. W. Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *10th IEEE International Conference on Computer and Information Technology, CIT 2010, Bradford, West Yorkshire, UK, June 29-July 1, 2010*, pages 1134–1139. IEEE Computer Society, 2010.

[65] iRobot. Events executor. https://github.com/irobot-ros/events-executor, 2022. last access on Nov. 17th, 2024.

[66] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Third edition, Sept. 2011.

[67] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Proceedings of the Real-Time Systems Symposium - 1991, San Antonio, Texas, USA, December 1991*, pages 129–139. IEEE Computer Society, 1991.

[68] X. Jiang, D. Ji, N. Guan, R. Li, Y. Tang, and W. Yi. Real-time scheduling and analysis of processing chains on multi-threaded executor in ROS 2. In *IEEE Real-Time Systems Symposium, RTSS 2022, Houston, TX, USA, December 5-8, 2022*, pages 27–39. IEEE, 2022.

[69] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In S. L. P. Jones and R. E. Jones, editors, *International Symposium on Memory Management, ISMM '98, Vancouver, British Columbia, Canada, 17-19 October, 1998, Conference Proceedings*, pages 26–36. ACM, 1998.

[70] R. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 03 1960.

[71] G. Kitagawa. Monte carlo filter and smoother for non-gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics*, 5(1):1–25, 1996.

[72] T. Kloda, A. Bertout, and Y. Sorel. Latency analysis for data chains of real-time periodic tasks. In *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2018.

[73] K. Knese and M. Pöhnl. A true zero-copy rmw implementation for ros 2. 2019.

[74] K. Knese, W. Woodall, and M. Carroll. Zero copy via loaned messages, Apr 2020.

[75] K. C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, 1965.

[76] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[77] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, volume 130, page 43, 2015.

[78] T. Kronauer, J. Pohlmann, M. Matthé, T. Smejkal, and G. Fettweis. Latency analysis of ROS 2 multi-node systems. In *2021 IEEE international conference on multisensor fusion and integration for intelligent systems (MFI)*. IEEE, 2021.

[79] R. E. Kálmán. The theory of optimal control and the calculus of variations. *Mathematical Optimization Techniques*, 1963.

[80] S. A. S. Lab. Hj reachability. https://github.com/StanfordASL/hj_reachability, 2024.

[81] R. Lange. Mixed real-time criticality with ROS 2 - the callback-group-level executor. ROSCon Lightning Talk, 2018.

[82] D. Lea and W. Gloger. A memory allocator, 1996.

[83] D. Leijen, B. Zorn, and L. de Moura. Mimalloc: Free list sharding in action. In A. W. Lin, editor, *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*, volume 11893 of *Lecture Notes in Computer Science*, pages 244–265. Springer, 2019.

[84] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[85] J. S. Liu and R. Chen. Sequential monte carlo methods for dynamic systems. *Journal of the American Statistical Association*, 93(443):1032–1044, 1998.

[86] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo. A constant-time dynamic storage allocator for real-time systems. *Real Time Syst.*, 40(2):149–179, 2008.

[87] M. Masmano, I. Ripoll, and A. Crespo. Dynamic storage allocation for real-time embedded systems. In *Proc. of Real-Time System Simposium WIP*. IEEE Computer Society, 2004.

[88] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *16th Euromicro Conference on Real-Time Systems (ECRTS 2004), 30 June - 2 July 1004, Catania, Italy, Proceedings*, pages 79–86. IEEE Computer Society, 2004.

[89] Microsoft. Airsim. https://github.com/microsoft/AirSim, 2021.

[90] T. Nakamura-Zimmerer, Q. Gong, and W. Kang. Adaptive deep learning for high-dimensional hamilton-jacobi-bellman equations. *SIAM J. Sci. Comput.*, 43(2):A1221–A1247, 2021.

[91] M. Nasri and B. B. Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pages 12–23. IEEE Computer Society, 2017.

[92] M. Nasri, G. Nelissen, and B. B. Brandenburg. Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling. In S. Quinton, editor, *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany*, volume 133 of *LIPIcs*, pages 21:1–21:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[93] G. Nelissen. Np schedulability analysis. https://github.com/gnelissen/np-schedulability-analysis. last access on Nov. 17th, 2024.

[94] NVIDIA. Nvidia isaac, 2023.

[95] Oren Bell. Hazcat. https://github.com/nightduck/hazcat, 2023.

[96] Oren Bell. Dyfc blas. https://github.com/nightduck/dyfc_blas, 2024.

[97] N. Otterness and J. H. Anderson. AMD gpus as an alternative to NVIDIA for supporting real-time workloads. In M. Völp, editor, *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, volume 165 of *LIPIcs*, pages 10:1–10:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[98] J. L. Peterson and T. A. Norman. Buddy systems. *Commun. ACM*, 20(6):421–431, 1977.

[99] S. Prajna, A. Jadbabaie, and G. J. Pappas. A framework for worst-case and stochastic safety verification using barrier certificates. *IEEE Trans. Autom. Control.*, 52(8):1415–1428, 2007.

[100] ROS 2 Real-time Working Group. Autoware reference system. https://github.com/ros-realtime/reference-system. last access on Nov. 17th, 2024.

[101] M. R. A. Sara, M. F. Klaib, and M. Hasan. Hybrid array list: An efficient dynamic array with linked list structure. *Indonesia Journal on Computing (Indo-JC)*, 5(3):47–62, 2020.

[102] M. A. Serrano, A. Melani, M. Bertogna, and E. Quiñones. Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In *Design, Automation & Test in Europe Conference & Exhibition, DATE*, pages 1066–1071, 2016.

[103] D. Simon. *Optimal state estimation: Kalman, H infinity, and nonlinear approaches*. John Wiley & Sons, 2006.

[104] H. Sobhani, H. Choi, and H. Kim. Timing analysis and priority-driven enhancements of ROS 2 multi-threaded executors. In *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2023, San Antonio, TX, USA, May 9-12, 2023*, pages 106–118. IEEE, 2023.

[105] J. Staschulat, R. Lange, and D. N. Dasari. Budget-based real-time executor for micro-ros. *CoRR*, abs/2105.05590, 2021.

[106] J. Staschulat, I. Lütkebohle, and R. Lange. The rclc executor: Domain-specific deterministic scheduling mechanisms for ROS applications on microcontrollers: work-in-progress. In T. Mitra and A. Gerstlauer, editors, *20th International Conference on Embedded Software, EMSOFT 2020, Singapore, September 20-25, 2020*, pages 18–19. IEEE, 2020.

[107] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. Scatteralloc: Massively parallel dynamic memory allocation for the gpu. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, 2012.

[108] M. Sudvarg, A. Clark, and C. Gill. Integrated real-time control and scheduling for safety critical cyber-physical systems. In *31st IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2025, Irvine, CA, USA, May 6-9, 2025*, pages 310–323. IEEE, 2025.

[109] X. Tang, N. Sheng, and L. Ying. Solving high-dimensional hamilton-jacobi-bellman equation with functional hierarchical tensor. *CoRR*, abs/2408.04209, 2024.

[110] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi. Response time analysis and priority assignment of processing chains on ROS 2 executors. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pages 231–243. IEEE, 2020.

[111] H. Teper, O. Bell, M. Günzel, C. Gill, and J. Chen. Reconciling ROS 2 with classical real-time scheduling of periodic tasks. In *31st IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2025, Irvine, CA, USA, May 6-9, 2025*, pages 177–189. IEEE, 2025.

[112] H. Teper, T. Betz, M. Günzel, D. Ebner, G. von der Brüggen, J. Betz, and J. Chen. End-to-end timing analysis and optimization of multi-executor ROS 2 systems. In *30th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2024, Hong Kong, May 13-16, 2024*, pages 212–224. IEEE, 2024.

[113] H. Teper, T. Betz, G. von der Brüggen, K. Chen, J. Betz, and J. Chen. Timing-aware ROS 2 architecture and system optimization. In *29th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2023, Niigata, Japan, August 30 - Sept. 1, 2023*, pages 206–215. IEEE, 2023.

[114] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J. Chen. End-to-end timing analysis in ROS 2. In *IEEE Real-Time Systems Symposium, RTSS 2022, Houston, TX, USA, December 5-8, 2022*, pages 53–65. IEEE, 2022.

[115] H. Teper, D. Kuhse, M. Günzel, G. von der Brüggen, F. Howar, and J. Chen. Thread carefully: Preventing starvation in the ROS 2 multithreaded executor. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 43(11):3588–3599, 2024.

[116] UFactory. xarm (ros 2 robot), 2022.

[117] M. Verucchi, I. S. Olmedo, and M. Bertogna. A survey on real-time DAG scheduling, revisiting the global-partitioned infinity war. *Real Time Syst.*, 59(3):479–530, 2023.

[118] G. von der Brüggen, J. Chen, and W. Huang. Schedulability and optimization analysis for non-preemptive static priority scheduling based on task utilization and blocking factors. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*, pages 90–101. IEEE Computer Society, 2015.

[119] E. A. Wan and R. van der Merwe. The unscented kalman filter for nonlinear estimation. In *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No.00EX373)*, pages 153–158, 2000.

[120] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In H. G. Baker, editor, *Memory Management, International Workshop IWMM 95, Kinross, UK, September 27-29, 1995, Proceedings*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer, 1995.

[121] M. Winter, M. Parger, D. Mlakar, and M. Steinberger. Are dynamic memory managers on gpus slow?: a survey and benchmarks. In J. Lee and E. Petrank, editors, *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 219–233. ACM, 2021.

[122] D. S. Wise. The double buddy system. Technical Report 79, Indiana University, Computer Science Department, Bloomington, Indiana 47401, December 1978.

[123] F. Wurst, D. Dasari, A. Hamann, D. Ziegenbein, I. Sañudo, N. Capodieci, M. Bertogna, and P. Burgio. System performance modelling of heterogeneous HW platforms: An automated driving case study. In *22nd Euromicro Conference on Digital System Design, DSD 2019, Kallithea, Greece, August 28-30, 2019*, pages 365–372. IEEE, 2019.

[124] Y. Xiang and H. Kim. Pipelined data-parallel CPU/GPU scheduling for multi-dnn real-time inference. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 392–405. IEEE, 2019.

[125] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith. Avoiding pitfalls when using NVIDIA gpus for real-time tasks in autonomous systems. In S. Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, volume 106 of *LIPIcs*, pages 20:1–20:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[126] G. Yao, G. C. Buttazzo, and M. Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. In *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2010, Macau, SAR, China, 23-25 August 2010*, pages 71–80. IEEE Computer Society, 2010.

[127] ZettaScale. Zenoh. https://github.com/eclipse-zenoh/zenoh, 2024.

Appendix A

Hybrid-Array Lists

A hybrid array list (HAL) is a series of fragmented chunks of an array linked together in a linked list. This allows for accelerated lookup times while retaining the O(1) insertion and deletion times of conventional linked lists. Incorporating hashtables can reduce lookup time to O(1) as well.

Each chunk is not assumed to be full, but it is assumed to have contiguous entries. The entries are references to free blocks and are stacked in the front half of the chunk, with the rear half leaving room for expansion. The entries are assumed to be sorted. A diagram of our HAL design is presented in Figure A.1.

Insertion and removal operations do not trigger a full sort within the chunk. Since the chunk is already sorted, all entries after that point are shifted to create or occupy the empty space. The insertion/deletion point can be easily identified with a O(logn) binary search of the chunk.

If insertion would cause the chunk to overflow, then it is split in two. If a removal would cause the chunk to be empty, then it is removed from the larger linked list.

As in prior work, the time complexity of insertion and removal operations from Hybrid Array Lists scales only on the size of the array chunk, not the length of the entire list.

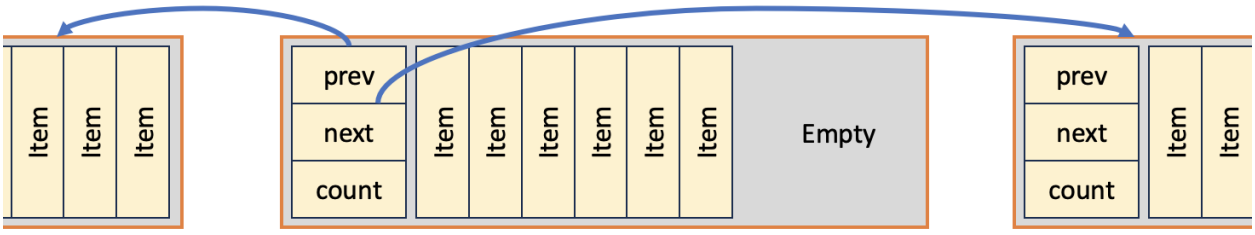


Figure A.1: Structure of Hybrid Array List

An algorithm is shown for initialization, Algorithm 9. Interactions with the HAL are described below.

A.1 Insert

Once an insertion point has been identified, every item in the chunk at and past that point must be shifted over. If the chunk is already completely full, it is first split, with half its contents going to a new chunk.

After the shifting process, the item is inserted into its selected location. Insertion time is linear with respect to the maximum size of the chunk, but does not scale with the size of the wider list.

A.2 Remove

To remove an item from a chunk, every item after that point is shifted over to overwrite it.

If the item to be removed is the last item in the chunk, the chunk is removed from the HAL by standard linked-list procedure.

A.3 Search

It is worth noting that, for our memory management purposes, search is done in reverse order. Given that x is the requested deallocation address, y_d is the corresponding block to be deallocated, and y is the set of all blocks, we say:

$$y_d = \max(y_i \in y) \mid y_i \leq x$$

As we iterate over the free list, we check the first (smallest) entry in each chunk. If it is larger than x, clearly y_d is not present, and we can continue checking the next previous

chunk. Once an entry $y_i \leq x$, then we know that y_d is somewhere in the current chunk. At this point, a simple binary search can identify it.

In a more general sense, search over a sorted HAL is done linearly over each chunk until one is confirmed to contain the element. Then a binary search is done within the chunk.

A.4 Iterators

As with any container, we can define iterators for a hybrid-array list to abstract the container into an ordered range. An iterator is implemented as the pair of a chunk reference and an index within that chunk.

The increment operation is normally as simple as incrementing the index. However, if doing so would push it past the end ($index \geq count$), then we perform

$$\begin{aligned} chunk &\leftarrow chunk.next \\ index &\leftarrow 0 \end{aligned} \tag{A.1}$$

Decrement operations are very similar but in reverse. The index is normally decremented, but if doing so would cause it to be negative, then we perform

$$\begin{aligned} chunk &\leftarrow chunk.prev \\ index &\leftarrow chunk.count - 1 \end{aligned} \tag{A.2}$$

Algorithm 9 Initialize Hybrid-Array List

1: $free\_list\_head \leftarrow$ alloc page
2: $used\_list\_head \leftarrow$ alloc page
3: $free\_list\_head.count \leftarrow 1$
4: $free\_list\_head.prev \leftarrow$ NULL
5: $free\_list\_head.next \leftarrow$ NULL
6: $free\_list\_head.blocks[0].address \leftarrow$ heap start
7: $free\_list\_head.blocks[0].size \leftarrow$ heap size
8: $used\_list\_head.count \leftarrow 0$
9: $used\_list\_head.prev \leftarrow$ NULL
10: $used\_list\_head.next \leftarrow$ NULL