CROS-RT: Cross-Layer Priority Scheduling for Predictable Inter-Process Communication in ROS 2

Jaeyoung Lee

Yonsei University

February 6, 2026

Overview

CROS-RT is a cross-layer scheduler designed to address **unpredictable end-to-end latency** in ROS 2 inter-process communication.

Key idea: eliminate **priority misalignment across layers** that can cause **priority inversion** and large latency variance.

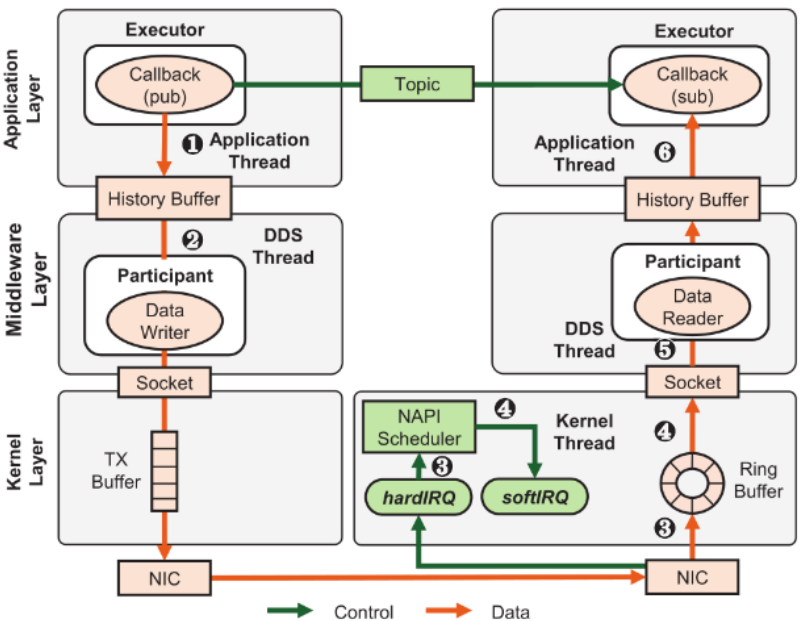
ROS 2 IN A NUTSHELL

ROS 2 is a popular middleware for complex robotic systems.

- Modular architecture, scalable integration of many components
- Distributed publish–subscribe messaging model (topics)

However, ROS 2 does not consistently propagate priorities across **application** $\rightarrow$ **middleware (DDS)** $\rightarrow$ **kernel**, which limits predictability under contention.

ROS 2 Communication Stack



Execution Model: Nodes, Callbacks, Executors

- ROS 2 applications consist of multiple **nodes**
- Nodes interact via **publisher–subscriber** topics
- Logic is implemented as **callbacks** (timer / subscription, etc.)
- Callbacks are executed (typically) **non-preemptively** by an **executor**
- One executor can manage multiple nodes

Intra-process vs Inter-process Communication

- **Intra-process**: publisher and subscriber within the same executor
  - Direct message transfer (no kernel networking path)
- **Inter-process**: publisher and subscriber on different executors
  - Same machine: shared memory channel
  - Different machines: UDP-based communication

This paper focuses on **unpredictability in inter-process communication**.

Threads Across Layers

Inter-process communication involves multiple threads:

- **Application thread**: per executor, manages callback execution
- **DDS thread**: per DDS participant, handles data reader/writer work
- **Kernel packet processing thread**: shared at kernel layer

Even if application/DDS priorities are aligned, the kernel can still break end-to-end priority ordering.

What Problem Does the Paper Solve?

Target: **unpredictability** caused by missing priority propagation to the **kernel layer**.

Prior approaches largely optimized:

- Callback scheduling within an executor (application layer)
- Runtime dynamic scheduling adjustments
- GPU/TPU management inside ROS 2
- DDS-level modeling to bound worst-case communication delay

CROS-RT: Core Components

CROS-RT enforces priority-based scheduling **across the ROS 2 stack**:

1. **Cross-layer priority identifier**: unify/propagate priorities across layers
2. **Message selector (kernel)**: prioritize packet processing by message priority
3. **Dynamic kernel handlers**: align kernel processing thread priority with message priority

Target System: Processing Chains

ROS 2 application is modeled as a set of **processing chains** of callbacks.

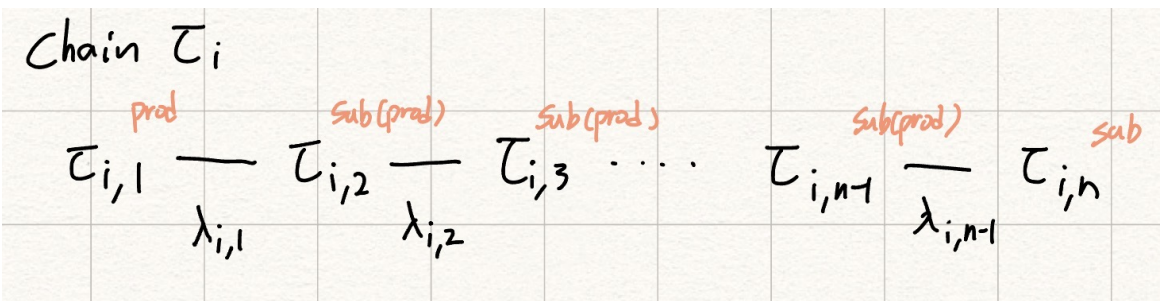
- A chain typically starts with a **timer callback**
- Then follows a sequence of **subscriber callbacks**
- Chain response time: maximum response time among released callback instances

Each chain τ_i:

$$\tau_i = (V_i, \Lambda_i, T_i, D_i, \pi_i)$$

- V_i: sequence of callbacks
- Λ_i: communications between consecutive callbacks
- T_i: period D_i: relative deadline
- π_i: chain priority (lower value = higher priority)

Processing Chain Example



Communication Categories

Communications are categorized into three disjoint sets:

Λ_i^I (intra-process), Λ_i^L (local inter-process), Λ_i^R (remote inter-process)

Assumption: communications in a chain execute **sequentially**.

Callback Scheduling in ROS 2

ROS 2 scheduling is hierarchical:

- **OS scheduler**: schedules threads at each layer
- **Executor scheduler**: determines callback execution order inside executor

In vanilla ROS 2, callback ordering/priority depends on:

1. callback type
2. registration order within that type

Even if we assign chain priorities by criticality, ROS 2 does not explicitly support **chain-level prioritization**.

PiCAS and Its Limitation

PiCAS assigns callback/executor priorities based on chain priority, improving alignment **at application and DDS layers**.

However, PiCAS cannot fully resolve unpredictability because:

- the kernel packet processing path still ignores message priority
- kernel scheduling can reorder effective service across chains

Experiment: Two Chains

Two chains, each with publisher+subscriber callbacks:

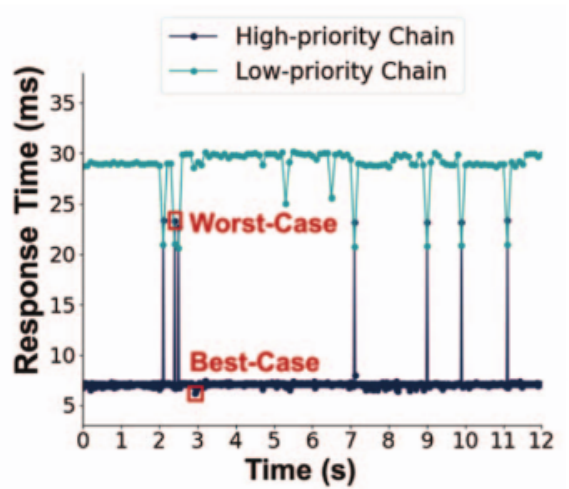
- High-priority chain: $T = 20$ms, compute $= 5$ms
- Low-priority chain: $T = 200$ms, compute $= 20$ms

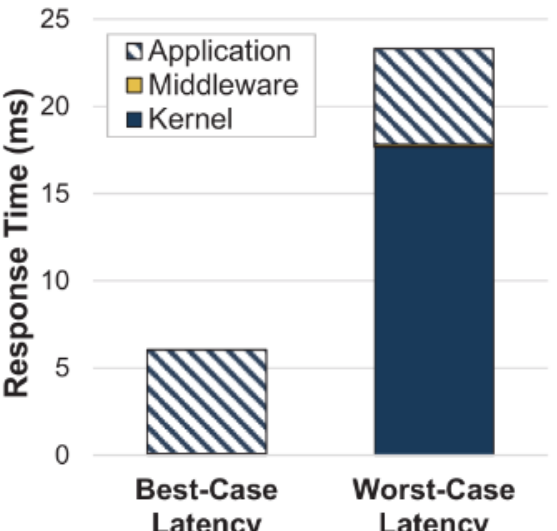
Ideal:

- High-priority latency ≈ 5ms (stable)
- Low-priority latency ≈ 30ms (average)

But real measurements show much larger deviation.

Measured Response-Time Deviation



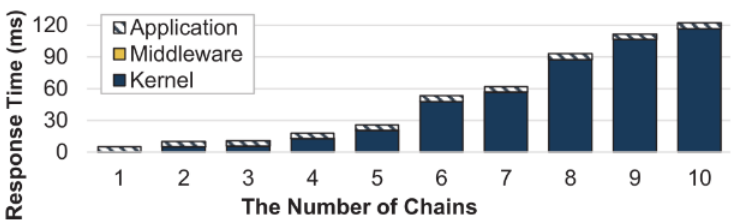


Impact of Load and Message Size

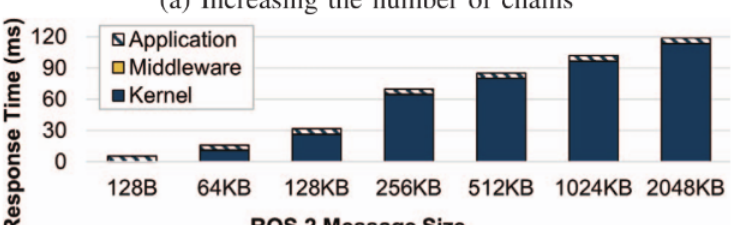
Increasing the number of chains and message size increases response time.

For UDP communication:

- Messages larger than 64 KB are fragmented
- More packets accumulate in ring buffer $\Rightarrow$ more delay



(a) Increasing the number of chains



(b) Increasing the size of ROS 2 messages

Why Kernel Causes Unpredictability

Even if application and DDS threads have aligned priorities (via PiCAS), kernel packet processing is not priority-aware.

Two key reasons:

1. Kernel cannot identify which executor (thus which priority) a packet belongs to
2. Kernel packet processing priority is driven by network load, not message priority

KERNEL ROOT ISSUES

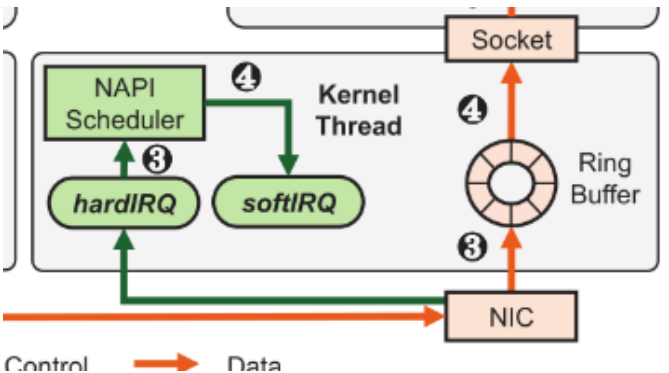
(1) FIFO packet handling:

- Kernel processes packets in FIFO order
- Message priority is ignored

(2) Network-load-based kernel thread scheduling:

- Packet processing thread behavior depends on load (softirq vs process context)
- Can delay high-priority packets behind low-priority traffic

LINUX PACKET PROCESSING PATH (NAPI)



Bottom-half execution can occur in:

- **softirq context**: immediate, non-blocking
- `ksoftirqd` **process context**: deferred, CFS-scheduled

PRIORITY INVERSION VIA KSOFTIRQD

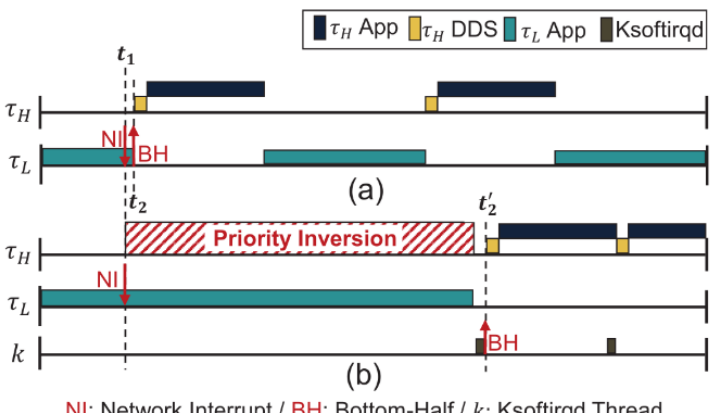


Fig. 4: Kernel packet processing by (a) current thread in softirq context and (b) Ksoftirqd thread in process context

When bottom-half runs in `ksoftirqd` process context, kernel packet processing may be deferred until lower-priority work completes, causing **priority inversion** and unpredictable message delivery latency.

Design Goals and Requirements

CROS-RT targets two kernel issues:

1. kernel cannot prioritize messages
2. kernel packet processing priority follows network load

Requirements:

- **R1**: enable kernel-level priority awareness
- **R2**: align kernel packet processing with message priority
- **R3**: preserve non-RT throughput as much as possible

CROS-RT Components

- **C1** Cross-layer priority identifier (app $\rightarrow$ DDS $\rightarrow$ kernel)
- **C2** Message selector (kernel priority queue)
- **C3** RT handler (dynamic kernel thread priority)
- **C4** Non-RT handler (polling server for throughput)

CROS-RT Components

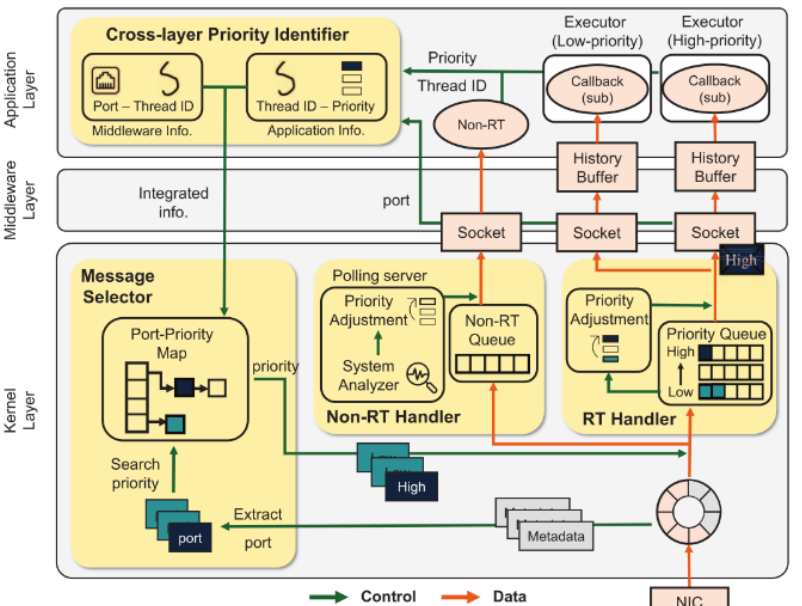


Fig. 5: System architecture overview of CROS-RT

C1: Cross-layer Priority Identifier

Goal: make kernel aware of message priority.

Vanilla ROS 2:

- Application: knows executor priority and thread ID
- DDS/Kernel: only sees destination UDP port (no priority)

CROS-RT builds a global mapping:

$$(\text{executor},\ \text{thread},\ \text{socket port}) \mapsto \text{message priority}$$

Mapping is stored in kernel and installed via a single system call.

C2: Message Selector

Kernel message selector uses propagated priority to reorder service:

- Replace FIFO packet selection with **priority-based selection**
- Ensure higher-priority messages are processed ahead of lower-priority ones

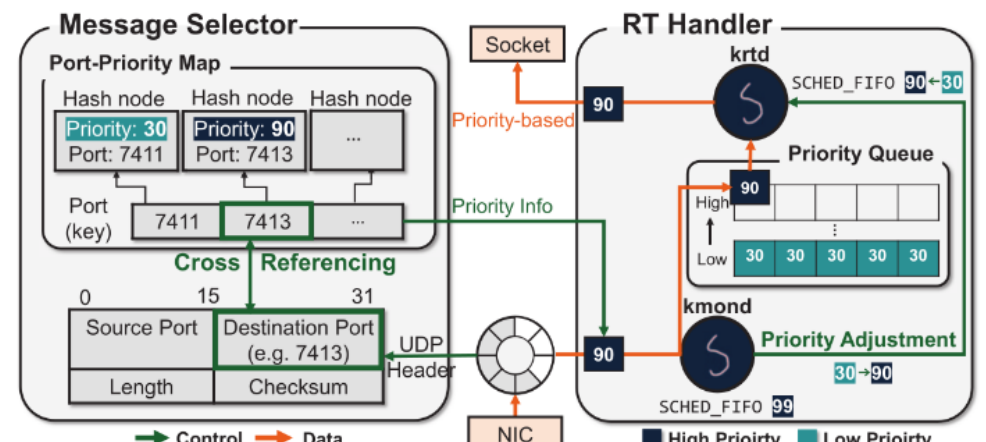


Fig. 7: Structures of message selector and RT handler

C3: RT Handler (Dynamic Kernel Scheduling)

Static priority for packet processing threads is insufficient:

- too high $\Rightarrow$ can starve upper-layer threads
- too low $\Rightarrow$ high-priority messages get delayed

CROS-RT introduces two kernel threads:

- `kmond`: highest priority, enqueues packets to RT priority queue, adjusts `krtd` priority
- `krtd`: packet processing thread, processes messages and delivers to sockets

C4: Non-RT Handler (Polling Server)

When RT priority queue is empty, non-RT traffic is served via polling server:

- executes periodically with fixed **budget** and **period**
- parameters selected using response-time analysis so RT chains meet deadlines

Goal: maximize non-RT throughput without compromising RT predictability.

Priority Assignment Principles

CROS-RT uses PiCAS and additional constraints to ensure chain-level prioritization:

- **P1**: higher-priority chain $\Rightarrow$ higher callback priorities (later callbacks in a chain have higher priority than earlier ones)
- **P2**: if executor A $>$ executor B, then all callbacks in A outrank those in B
- **P3**: each executor contains callbacks from only a single chain per machine
- **P4**: executor inherits chain priority; application/DDS threads inherit executor priority

This provides consistent priority ordering across layers.

Assumptions for Analysis

- Each machine executes one single-threaded executor at a time
- Executors scheduled by **SCHED_FIFO** (fixed-priority preemptive)
- Callback execution inside executor is **non-preemptive**

Communication Cost Decomposition

Worst-case cost of message $\lambda_{i,k}$:

$$C_{i,k}^{I} = e^{\text{pub\_app}}(\tau_{i,k}) + e^{\text{sub\_app}}(\tau_{i,k+1})$$

$$C_{i,k}^{L} = C_{i,k}^{I} + e^{\text{pub\_dds}}(\tau_{i,k}) + e^{\text{sub\_dds}}(\tau_{i,k+1})$$

$$C_{i,k}^{R} = C_{i,k}^{L} + e^{\text{sub\_RT}}(\tau_{i,k+1}) + e^{\text{sub\_sel}}(\tau_{i,k+1}) + \delta_{i,k}^{\text{net}}$$

Each term is proportional to message size and can be estimated via linear regression.
Total chain cost:

$$C_i = \sum_{\lambda_{i,a} \in \Lambda_i^I} C_{i,a}^{I} + \sum_{\lambda_{i,b} \in \Lambda_i^L} C_{i,b}^{L} + \sum_{\lambda_{i,c} \in \Lambda_i^R} C_{i,c}^{R}$$

Interference and Blocking

Worst-case response time via fixed-point iteration:

$$R_i^{(a+1)} = C_i + I_i(R_i^{(a)}) + B_i, \quad \text{until } R_i^{(a+1)} = R_i^{(a)}.$$

Interference from higher-priority chains:

$$I_i(R_i^{(a)}) = \sum_{\lambda_{h,k'} \in hp(\tau_i)} \left\lceil \frac{R_i^{(a)}}{T_h} \right\rceil \cdot C_{h,k'}^{z}$$

Blocking from lower-priority chains (remote comm. case):

$$B_i = \left|\Lambda_i^R\right| (N - i) \max_{\lambda_{l,k''} \in lp(\tau_i) \cap \Lambda_l^R} e^{\text{sub\_sel}}(\tau_{l,k''}) + \sigma_i T_i$$

$$\sigma_i = \mathbb{I}[R_i > T_i]$$

Even with priority selection, FIFO ring buffer behavior can delay high-priority packets behind previously arrived low-priority packets.

Experimental Setup

- ROS 2 Humble + eProsima FastDDS
- Linux kernel 5.15
- Two Nvidia Jetson Orin Nano devices
- ARM Cortex-A78AE @ 1.5 GHz
- 100 Mbps Ethernet link
- Non-RT load: 60 Mbps input bandwidth
- 10,000 instances per chain

BASELINES

- **Vanilla**: default ROS 2
  - `ksoftirqd` priority determined by network load
- **Preempt-RT**: ROS 2 + Preempt-RT patch
  - static priority on `ksoftirqd`
- **CROS-RT**: proposed cross-layer scheduler
  - dynamic priority for kernel packet processing

NAV2 Case Study

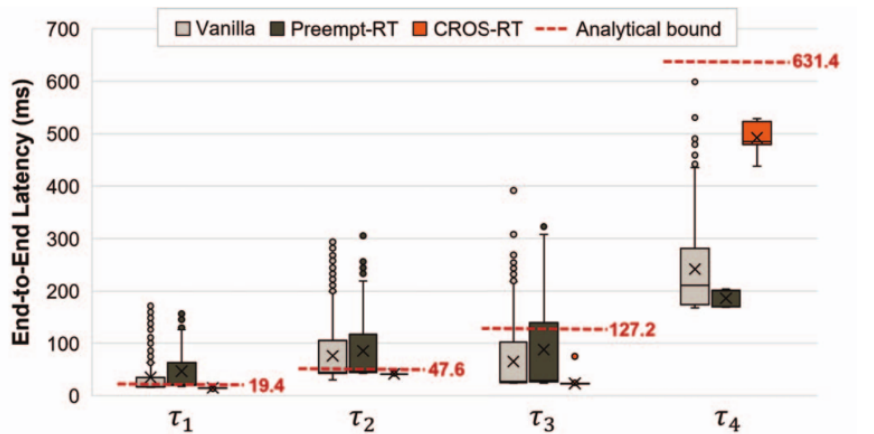
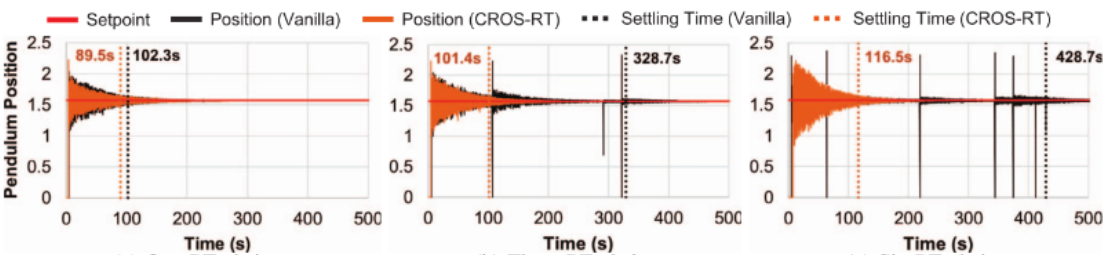


Fig. 9: NAV2 case study: end-to-end chain latency distribution

CROS-RT provides bounded latency; strict priority enforcement increases latency for lower-priority chains.

Inverted Pendulum Case Study

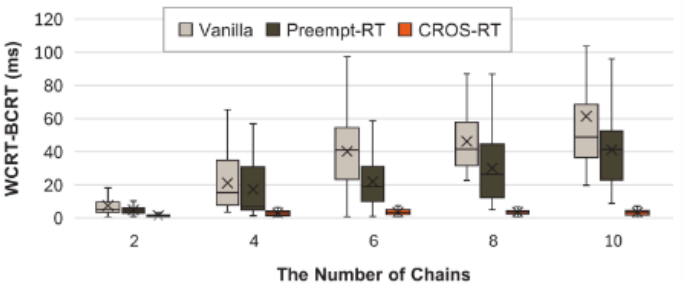


(a) One RT chain (b) Three RT chains (c) Six RT chains

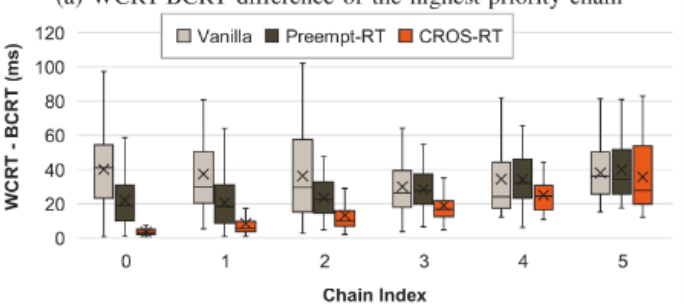
Fig. 10: Inverted pendulum case study: settling time

As low-priority chains increase, the highest-priority control loop maintains consistently faster settling time under CROS-RT.

Randomly Generated Chains (UUniFast)



(a) WCRT-BCRT difference of the highest-priority chain



(b) WCRT-BCRT difference for all chains in six-chain set

Fig. 11: WCRT-BCRT difference distribution with increasing number of chains

PREDICTABILITY METRIC: WCRT–BCRT

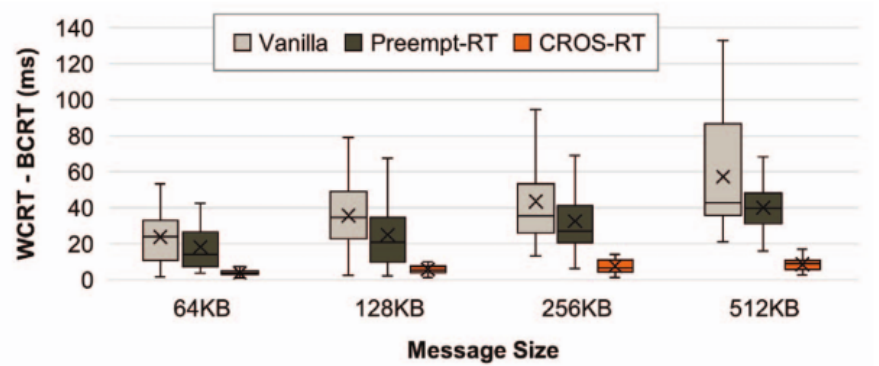


Fig. 12: WCRT-BCRT difference distribution of the highest-priority chain with increasing message sizes

- Predictability measured via **WCRT–BCRT**
- Highest-priority chain becomes much more predictable under CROS-RT
- Under high load, worst-case latency increases for all methods

NON-RT TASK THROUGHPUT

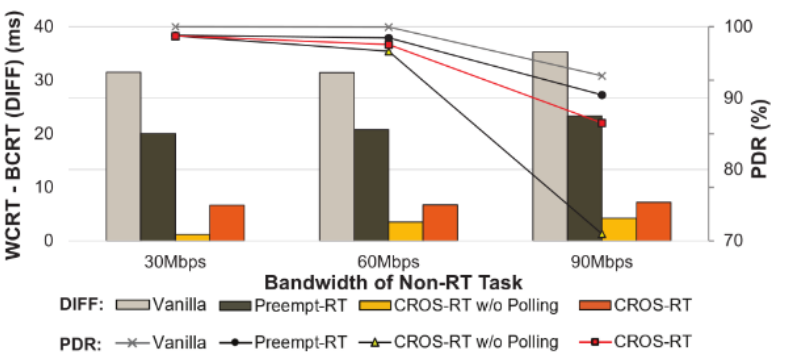


Fig. 13: RT predictability and non-RT throughput with increasing non-RT traffic

Trade-off:

- CROS-RT w/o polling: best WCRT–BCRT gap but worst packet delivery ratio (PDR)
- Polling server aims to balance RT predictability and non-RT throughput