Edge-RT: OS Support for Controlled Latency in the Multi-Tenant, Real-Time Edge

Wenyuan Shao, Bite Ye, Huachuan Wang, Gabriel Parmer, Yuxin Ren
The George Washington University
Washington, DC
{shaowy,bitye,hcwang,gparmer,ryx}@gwu.edu

***Abstract*—Embedded and real-time devices in many domains are increasingly dependent on network connectivity. The ability to offload computations encourages Cost, Size, Weight and Power (C-SWaP) optimizations, while coordination over the network effectively enables systems to sense the environment beyond their own local sensors, and to collaborate globally. The promise is significant: Autonomous Vehicles (AVs) coordinating with each other through infrastructure, factories aggregating data for global optimization, and power-constrained devices leveraging offloaded inference tasks. Low-latency wireless (*e.g.*, 5G) technologies paired with the *edge cloud*, are further enabling these trends. Unfortunately, computation at the edge poses significant challenges due to the challenging combination of limited resources, required high performance, security due to multi-tenancy, and real-time latency.**

This paper introduces Edge-RT, a set of OS extensions for the edge designed to meet the end-to-end (packet reception to transmission) deadlines across chains of computations. It supports strong security by executing a chain per-client device, thus isolating tenant and device computations. Despite a practical focus on deadlines and strong isolation, it maintains high system efficiency. To do so, Edge-RT focuses on per-packet deadlines inherited by the computations that operate on it. It introduces mechanisms to avoid per-packet system overheads, while trading only bounded impacts on predictable scheduling. Results show that compared to Linux and EdgeOS, Edge-RT can both maintain higher throughput and meet significantly more deadlines both for systems with bimodal workloads with utilization above 60%, in the presence of malicious tasks, and as the system scales up in clients.

I. Introduction

Embedded and real-time systems are increasingly required to provide features that must interact with the broader environment beyond their local sensors and actuators. Though IoT systems are notable for their Internet connectivity, even systems with strict predictability requirements are now commonly network-connected to leverage distributed sensors and computation. Industry 4.0 focuses on the interlinking of real-time machinery with network-connected aggregation and analytics to better manage factories, and vehicle-to-everything (V2X) communications acknowledge that vehicular decisions are empowered by communicating with and understanding the environment. Empowering this, modern millimeter-wave wireless technologies such as 5G aim for 1ms round-trip times (RTT) – current latencies are sub-10ms [1] – thus providing latency levels that can potentially fit into the decision loops of many real-time systems. In contrast to computation hosted in the cloud's datacenters whose access imposes the significant jitter and high latency of the WAN, the *edge cloud* has basestations that are proximate to the embedded systems. For example, high-frequency 5G basestations have an effective range in the 100s of meters, enabling low-latency RTTs. Embedded systems can benefit from low-latency access to the edge for (1) offloading [2] of computation from embedded devices to leverage the more capable hardware of the edge for higher-performance or memory-hungry computations while potentially lowering device power requirements, and (2) for sensor aggregation in which the sensor information from many devices can be aggregated, thus decisions can consider a more global state of the physical environment.

Unfortunately, unlike traditional cloud datacenters that contain hundreds of thousands of cores and provide the illusion of capacity elasticity, edge-clouds have much more constrained resources [3], [4]. Despite this, the multi-tenant model that has driven the prominence of the cloud is desirable – and in some cases, necessary – in the edge-cloud. A multi-tenant infrastructure enables the execution of untrusted code, provided by potentially many different tenants that rent capacity on the server. Multi-tenancy is common on current edge deployments: (1) network slicing enables multiple cellular carriers to process packets using Network Functions (NFs) to share the basestation infrastructure [5], [6], [7], [8], and (2) edge computation providers, such as Fastly, support tenant executions in isolated Webassembly sandboxes. For embedded systems to leverage the edge in general deployments (*e.g.*, 5G basestations), there is a strong need for multi-tenancy. Unfortunately, this is difficult due to the relatively constrained resources of the edge cloud.

To effectively leverage edge-cloud systems, system software must meet the following requirements:

- *End-to-end deadlines* – at its core, the *real-time edge* must accommodate computations with the deadline requirements on the order of a few milliseconds – given the lower-bound of 1ms RTT for 5G. The edge must manage the end-to-end latency of requests between when they arrive, and when the reply is transmitted.
- *Performance* – the edge-cloud requires an efficiency capable of driving many 10s of Gbs network throughput. High performance makes worst-case provisioning unappealing, motivating a system that seeks to provide predictable latencies with unpredictable workloads.
- *Density* – unlike the traditional cloud that uses VMs and containers to isolate tenants, the smaller computational resource availability at the edge requires abstractions and mechanisms that efficiently enable higher-density.
- *Multi-tenancy* – similar to data-center-based clouds, the edge must support multi-tenancy. For example, content delivery network (CDNs) already find such support necessary (Fastly and Cloudflare). This requires untrusted code to execute in the edge, with unknown execution times, and potentially faulty or malicious logic.
- *Dynamic workloads* – the client workload changes with the environment. As autonomous vehicles (AVs), drones, and

This material is based upon work supported by the National Science Foundation under Grants No. CNS 1815690 and CPS 1837382, through SRC under grants GRC task 2911.001 and SRC JUMP task 2779.030, and ONR N000142212084. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

Edge Configurations	Deadline-aware	Preemptivity	Client Isolation	Computation Chain	Dynamic Workloads	Scalability
CFS (§II-B)	○ not deadline-aware	● preemptive	● process-based	● per-client chain	● supported	$>$ 2000
DPDK+OVS/SR-IOV (§II-A)	○ not deadline-aware	○ non-preemptive	● process-based	○ no chain	● supported	$\sim$ 256
SCHED_DEADLINE (§II-B)	● per-thread	● preemptive	○ process-based	○ no chain	○ not supported	$<$ 1000
eBPF+XDP (§II-A)	○ not deadline-aware	○ non-preemptive	◐ no isolation	◐ no chain	◐ not supported	-
EdgeOS (§II-C)	○ not deadline-aware	● preemptive	● FWP-based	● per-client chain	● supported	$>$ 2000
Edge-RT (§III)	● per-packet	● preemptive	● FWP-based	● per-client chain	● supported	$>$ 2000

TABLE I: A summary of edge-cloud configurations in §II. Entries labelled with bullets from fully supported (●), complicated (please refer to the text for details) (◐), and not supported (○).

pedestrians are mobile, the client number and frequency of service requests vary over time.

- *Network processing* – basestations traditionally focus on network processing including properly accounting for bandwidth, and slicing the network [5], [6], [7], [8] across carriers. This network processing is done by network functions (NFs) that transform and filter packets, and are often composed into chains that process packets, and pass them on to the next NF. Chains of isolated NFs enable multiple applications to process on packets, and enable NFs to provide limitations on each other. For example, the first and last NFs can provide firewall-like functionality to limit which packets can be processed and transmitted by NFs in the middle of the chain.

The core question this paper seeks to answer is: *is it possible to practically meet end-to-end deadlines of packets while still maintaining high-throughput and strong isolation in a multi-tenant, edge-cloud for dynamic and dense workloads?*

One tempting answer is to directly adapt existing deadline-driven scheduling systems (*e.g.*, EDF) to edge-clouds. We argue that this is not sufficient because (1) many edge cloud infrastructures do not support preemptive scheduling (§II-A), (2) to optimize for meeting end-to-end deadlines across chains of computations, normal per-thread prioritization is not a good fit for dynamic workloads (§III), and (3) high-throughput network systems seek to avoid per-message overheads, which is a bad match for OS abstractions that require locks and Inter-Processor Interrupts (IPIs) for coordination (§III).

This paper presents Edge-RT, an OS infrastructure built on the public EdgeOS [9], that focuses on packet- or message-based deadline scheduling across chains of computations, while maintaining high performance, density, and isolation between client computations. Edge-RT focuses on practical mechanisms to meet deadlines while minimising per-message system overheads: (1) it associates deadlines with *packets*, and threads *inherit* these message deadlines as packets flow through the computation chains to provide *end-to-end, deadline-based scheduling*, and (2) creates mechanisms for coordination and execution that avoid per-message overheads for scheduling, batching, and inter-FWP, inter-core coordination while bounding interference.

Contributions. Edge-RT's contributions center on providing deadline-focused computation in a high-throughput, high-density environment. The contributions include:

- a system design that (1) focuses on per-packet end-to-end deadline scheduling with dynamic, dense workloads, and (2) minimizes per-message system overheads.
- the mechanisms and abstractions for predictable buffering, inter-core coordination, and scheduling that enable efficient packet processing.
- The implementation of Edge-RT and the parameter studies to understand system overhead trade-offs to guide the system's configuration.
- The evaluation of Edge-RT for various workloads compared with Linux and EdgeOS.

II. Background

A. Linux Kernel Bypass and In-kernel Sandbox

Kernel-bypass networking. Kernel-bypass networking (*e.g.*, Data Plane Development Kit (DPDK) [10]) enables user-level to directly interact with networking devices. DPDK avoids both system call and interrupt overheads by polling. Using interrupt-driven execution increases the round-trip time to 110μs, which is close to the overheads of Linux sockets. Despite DPDK's efficiency, it has limitations in the edge cloud: low scalability and non-preemptive packet processing.

Tenant isolation requires the use of a separate DPDK instance per tenant. This is accomplished using virtual NICs through SR-IOV, or using Open vSwitch (OVS) [11]. SR-IOV only supports up to 256 virtual NICs, thus cannot scale up to many tenants. OVS can scale as it acts as a virtual switch to route packets among multiple DPDK applications. Unfortunately, OVS limits scalability and density. For applications using DPDK and `virtio` [12], OVS with 256 DPDK applications requires one polling thread per 160K PPS, and each (minimal) application requires 110 MiB of memory.

Each tenant's DPDK application processes their client's requests sequentially, thus non-preemptively. To understand this effect, we run two types of computations in each DPDK application, a "Light" computation which is quick to process (40μs), and a "Heavy" computation (varying between 10-25ms). This sequential execution model is common in throughput-centric network processing systems such as E2 [13] and Netbricks [14]. Figure 1 compares the tail latency of bypass techniques to native Linux sockets. In contrast, Linux execution of clients in separate processes is preemptive, in contrast to the sequential execution of client requests in bypass systems. Note that adding preemptive execution into the bypass requires multiple threads or processes, which incurs kernel overheads that bypass systems are designed to avoid. The workload maintains 50% utilization across 48 cores. Four cores are devoted to OVS, and four DPDK tenants execute per core. Each receives four concurrent client requests at 200 packets per second for light computation, and the heavy computation rate is adjusted with its changing weight (x-axis). Linux socket applications use a process per client. The hardware is detailed in §VI.

This example demonstrates that maintaining kernel bypass for each tenant imposes non-preemptive execution in which the most latency-sensitive tasks suffer from convoy effects

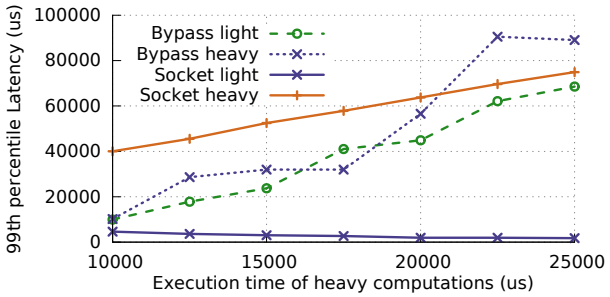


Fig. 1: 99th percentile latency of bimodal workloads when using kernel bypass techniques.

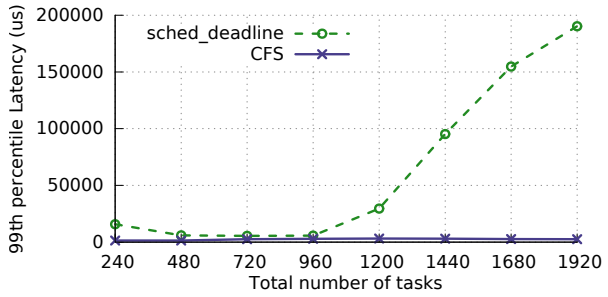


Fig. 2: 99th percentile latency of light task with an increasing number of total tasks and constant number of heavy tasks.

from heavy computations. Bypass properties are summarized in Table I. The edge cloud must re-imagine such high-throughput networking techniques to intelligently, preemptively schedule clients.

Extended Berkeley Packet Filter (eBPF). eBPF [15] is an in-kernel virtual machine which allows injecting application logic into the kernel at run time. XDP [16] executes an eBPF program to process packets as part of the in-kernel packet reception path. eBPF has a few limitations [15], [17]. (1) An eBPF program is restricted to an execution budget of 1 million instructions. (2) eBPF programs execute non-preemptively (unless their budget runs out), even if executing in a high-priority NIC interrupt. (3) Loading eBPF programs is a privileged operation (*e.g.*, `root` user) as they have sensitive access to kernel abstractions. These factors (summarized in Table I) make eBPF a challenging choice for a multi-tenant execution environment – especially one that requires chains of computations and deadline-aware scheduling.

B. Thread-base Prioritization Deadline Scheduling.

Given our focus on deadline-based scheduling, it is important to understand `SCHED_DEADLINE` in Linux and its applicability to the edge cloud. `SCHED_DEADLINE` adds EDF support to Linux, along with constant bandwidth server (CBS) [18] logic to rate-limit computation. We use a process-per-client so that they can each be preemptively scheduled with separate deadlines, and use budget reclaiming to handle budget under-utilization. Heavy tasks have 5ms executions, 10ms budgets, 100ms deadlines, and receive 10 requests per second. Light tasks reply immediately, have a sufficient budget of 8μs, a deadline of 5ms, and receive 100 requests per second.

The utilization of the system is low (from 6% up to 50%), yet Figure 2 demonstrates that with an increasing number of light tasks, tail latency increases significantly. The request workload is not perfectly periodic which mimics the dynamic workload on the edge. `SCHED_DEADLINE` relies on per-thread prioritization. Thus, aperiodic workloads will cause significant deviation in desired executions. In contrast, the CFS Linux scheduler is designed for accommodating dynamic workloads and requires no periodicity, budget, nor deadline parameters. Correspondingly, this figure demonstrates its ability to maintain tight latency properties for the light tasks.

We argue that `SCHED_DEADLINE` under-performs CFS because (1) thread-based prioritization is not a good fit for the edge cloud which aims to meet deadlines for packets/requests; (2) `SCHED_DEADLINE` is a bad fit for dynamic workloads on the multi-tenant edge in which execution time and request distributions are unknown; (3) the red-black tree based implementation of `SCHED_DEADLINE` imposes overheads (§VI-A) with many computations (over 1200). Generally, we have observed CFS to have better properties on scheduling high-throughput, dynamic workloads. Table I includes a summary of some of the properties of Linux scheduling options.

C. EdgeOS Background

Edge-RT adds real-time capabilities to the publicly available EdgeOS system [9]. EdgeOS is implemented as a set of user-level components above the Composite [19] μ-kernel. In EdgeOS, each tenant provides *chains of computations* that process client requests. EdgeOS isolates the computations into Feather Weight Processes (FWPs). Each FWP is single-threaded, and has memory accesses restricted by hardware page-tables. EdgeOS uses DPDK for efficient networking. Two cores in EdgeOS are dedicated to poll/send packets directly from/to the NIC. Two additional cores provide FWP creation and fast *message*[1] movement in an FWP chain, a service called the Memory Movement Accelerator (MMA). All other cores run preemptively-scheduled FWPs. Table I summarizes key features of EdgeOS, and also how Edge-RT expands on them.

III. System Model

Edge-RT aims to provide predictable services to clients with dynamic workloads [20], [21] provided by tenant computations. As such, we make three workload assumptions:

- *Tenant-provided deadlines.* Each tenant provides deadlines for its services. For example, an AV manufacturer provides deadlines for chains focusing on planning or control.
- *Unknown execution profiles.* In contrast, it isn't practical for a tenant to understand its computation's execution time due to inaccessible edge hardware. Given this, we assume no knowledge on the edge system about average or worst-case execution time.
- *Controlled utilization.* This research focuses on predictable scheduling, and not on admission control. We assume load balancers sitting before computation nodes control utilization to not exceed capacity in nominal conditions. We do study the impact of malicious or erroneous clients that cause certain computation to over-consume CPU in §VI-D. Edge-RT can be extended in the future to provide workload shaping (*e.g.*, by incorporating reservations [22]).

[1]We'll use the terms *messages* and *packets* synonymously.

To describe Edge-RT's system design in detail, we introduce a general model for the processing on packets throughout the system. The focus of this model is not to assess schedulability or analyse the system given the unknown execution properties of tenant computations, instead to precisely describe the timing properties of systems.

A number of tenants execute computations concurrently, and each tenant has multiple *network application chains* that process latency-sensitive requests from clients. As client's input to Internet-facing services is not necessarily trustworthy, we assume it is desirable that the edge cloud deploys a new chain of application computations for *each* client. This has two side effects that increase inter-client isolation: (1) the scope of errant or malicious memory manipulations is confined to a client's single computation, and (2) each client's computations are separately, preemptively scheduled which enables per-client latency optimization.

$\mathcal{C}_i$ denotes a per-client computation chain, which processes all requests from the i-th client. Without loss of generality, we assume that each client communicates with only a single chain. $\mathcal{C}_i$ consists of K computation stages, and the initial (s_i^0) and final (s_i^{K+1}) stages represent NIC driver packet reception and transmission, respectively: $\mathcal{C}_i = \{s_i^0, \ldots, s_i^x, \ldots, s_i^{K+1}\}$. Each stage s_i^{x-1} processes a message and generates another message to the next stage s_i^x. Each stage runs in a separate thread, thus the chain can be separated across cores. $m_{i,n}$ is the nth packet sent by client i. The goal is for each packet to be processed at or before a relative deadline d_i from the packet arrival.

To describe the execution of client chains and the timing properties of that execution:

1) s_i^x begins processing a previously *received* message $m_{i,n}$ at time $r_{i,n}^x$. This could be due to the stage activating and dequeueing the message, or finishing processing $m_{i,n-1}$, and dequeueing $m_{i,n}$.
2) Stage s_i^x finishes processing $m_{i,n}$ and enqueues the resulting message for *transmission* to s_i^{x+1} at time $t_{i,n}^x$.
3) $r_{i,n}^0$ and $t_{i,n}^{K+1}$ represent NIC reception and transmission for $m_{i,n}$, respectively.

Given this, a chain $\mathcal{C}_i$ meets its deadline for $m_{i,n}$ only if $r_{i,n}^{K+1} - t_{i,n}^0 \leq d_i$. As such, we assume, for simplicity, that execution measured against the deadline doesn't include necessary NIC processing. Thus a specific packet's ($m_{i,n}$'s) absolute deadline is $d_{i,n} = t_{i,n}^0 + d_i \leq r_{i,n}^{K+1}$.

System delays include overheads from passing messages, triggering their events to coordinate between threads, and scheduling interference from other chains:

$$\sum_{k=1}^{K+1} r_{i,n}^k - t_{i,n}^{k-1} \tag{1}$$

Equation 1 includes interference that delays the reception of a message. The scheduling interference, $\Delta_{\text{interference}}$, includes (1) computation from other stages/chains executing at a higher-priority, (2) lower-priority stage computation (*e.g.*, due to resource sharing [23], [24]), and (3) the processing of a previous message from the same client. $\Delta_{\text{interference}}$ also factors into how long a stage that has already received a message, takes to transmit it ($\sum_{k=1}^{K} t_{i,n}^k - r_{i,n}^k$).

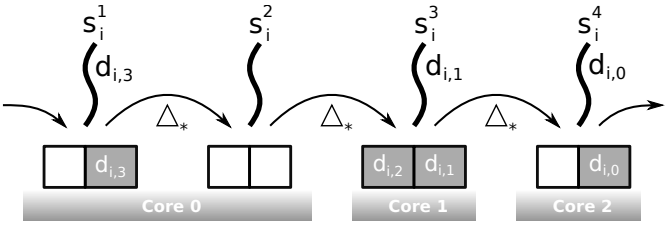


Fig. 3: Four stage's threads spread across three cores and process messages with deadlines ($d_{i,0}, \ldots, d_{i,3}$). A thread *inherits* the closest deadline of those messages it is processing. The Δ_* denotes overheads for passing messages between stages.

Equation 1 also includes *per-message* system overheads: (1) latencies in the scheduling policy implementation (Δ_{impl}); (2) event notification costs (Δ_{evt}) to activate the receiver stage's thread; and (3) message queueing and copying (Δ_{msg}). Core optimizations in Edge-RT focus around minimizing per-message costs.

A. Prioritization

Thread-based prioritization. OS threads are the unit of prioritization in most OSes, even in those that use dynamic priorities (*e.g.*, EDF) and/or consider inter-thread event-chain coordination. The interference from higher-priority threads factors into $r_{i,n}^k - t_{i,n}^{k-1}$, and is based on the workload on s_i^k's thread's core, and the relative prioritization of threads. Priorities are often chosen with knowledge of s_i^k's execution time. In our case, the workload and execution distributions are not necessarily known for the edge (see the assumption in §III).

Edge-RT and per-packet prioritization. Edge-RT's key design is to *prioritize computation (threads) based on the specific message they are processing.* Edge-RT uses EDF scheduling, thus stage's threads *inherit* each packet's ($m_{i,n}$) deadline, $d_{i,n}$, as shown in Figure 3. Though this does not *guarantee* meeting the deadline (recall, we make few assumptions about workload), carrying a packet's deadline through the processing of all stages ensures that delays in one stage causes more urgency in later stages.

Batching complicates this analysis. High-throughput systems must strategically use batching, often directly in shared memory, to amortize system overheads [25], [26], [27]. Per-message context switching and system call overheads are explicitly avoided in many high throughput systems [28], [29], [30], [31]. Instead, computations iteratively process *multiple* packets for each activation. Batching of up to m messages in shared memory in s_i^x happens when $t_{i,n+m-1}^{x-1} < t_{i,n}^x + \Delta_{\text{msg}}$. The Δ_{msg} acknowledges that the message might not be copied out of s_i^x immediately upon transmission. When m messages are accessible to a stage, the system cannot know *which* of the priorities (in $[d_{i,n+m-1}, d_{i,n}]$) the stage should inherit. Edge-RT's policy inherits the *closest deadline* of all batched messages, $\min_{k=n}^{n+m+1}(d_{i,k}^x)$.

B. Case Studies

Linux. Linux provides multiple means of interfacing with the packet processing (§II-A):

- Linux uses thread-based prioritization, and either `SCHED_DEADLINE` or CFS. While `SCHED_DEADLINE` can

bound $\Delta_{\text{interference}}$ for sporadic packets, Figure 2 shows that irregular workloads cause significant deviations from desired execution. CFS's prioritization of short-running tasks practically scales while maintaining a low latency, but does not bound $\Delta_{\text{interference}}$, due to the unpredictable number and execution times of computations in the runqueue. Δ_{impl} includes scheduling implementation overheads from runqueue updates at time $t^x_{i,n}$, while Δ_{evt} includes IPI overheads and runqueue lock contention. Δ_{msg} includes message copying overheads with shared memory, or into and out of kernel buffers with pipes or sockets.

- eBPF-based techniques suffer from long, non-preemptive $\Delta_{\text{interference}}$, and is scheduled based on events, not on priorities. The benefit of this event driven activation is that it minimizes Δ_{evt} and Δ_{msg}.
- Kernel bypass techniques rely on the $\Delta_{\text{interference}}$ and Δ_{evt} properties of Linux. Δ_{msg} data movement costs are minimal with only a fixed, small number of computations (NIC DMA) or involve copying with OVS and `virtio`.

EdgeOS. EdgeOS provides high-throughput edge processing, and uses simple round-robin scheduling (RR) to minimise Δ_{impl}. Though RR is a predictable scheduling policy, EdgeOS isn't aware of the computation execution times nor the number of computations that be in the runqueue. In these conditions, $\Delta_{\text{interference}}$ is not predictable. Per-packet overheads on FWP-processing cores are minimized ($\Delta_{\text{evt}} = 0$) by instead simply cycling through that core's FWPs via RR. Thus the overheads are parameterized by the number of clients, not the rate of messages. To maintain strong spatial isolation between stage computations ("FWPs" in EdgeOS), the MMA copies packets between FWPs, thus defining Δ_{msg}. Though EdgeOS focuses strongly on minimizing system overheads, it does *not* provide any latency-aware execution of FWPs.

Edge-RT. Like EdgeOS, Edge-RT devotes a core to the MMA to transfer messages between stages, thus imposes the corresponding Δ_{msg} overheads. Unlike EdgeOS, Edge-RT tightly coordinates between the MMA and each core's scheduler logic. The scheduler is aware of packet deadlines tracked by the MMA, and the MMA understands stage thread execution to provide deadline-aware batching. This coordination between the MMA and the scheduler constitutes Δ_{evt}. Edge-RT shrinks per-message processing costs (*e.g.*, of waking and scheduling stages) by ensuring that, despite implementing EDF scheduling, Δ_{impl} has a constant overhead (§IV-C). A core tension in the design of Edge-RT is between accurate, end-to-end deadline scheduling of packets, and high throughput. Nowhere is the tension more apparent than in the batching of messages for stages. In managing this, Edge-RT ensures a bounded inaccuracy in the deadline for which a stage's thread executes. All optimizations in Edge-RT are designed to have a *bounded* impact on end-to-end message processing latency, thus maintaining *practical, end-to-end, per-packet, deadline scheduling*.

IV. System Design

In this section we focus on introducing Edge-RT's mechanisms and abstractions to reduce per-message system overheads, and to practically meet end-to-end, message deadlines. To reduce system overheads ($\Delta_{\text{interference}}$, Δ_{impl}, Δ_{msg}, and Δ_{evt} from §III), Edge-RT employs three mechanisms:

- *deadline-aware batching* of messages to reduce overhead of messaging and event notification,
- *periodic (rather than event-driven) inter-core coordination* to amortize the costs of coordination, and
- *a constant-time EDF design* with constant overhead independent to the number of computations in the run-queue.

A. Deadline-aware Batching

Batching is a general technique [25], [26], [27] to improve system throughput. The overheads of a single thread activation – including, context switching, system calls, and cross-core activations – are amortized across multiple messages. Batching often trades away some latency by delaying the processing of a message until more are available. Thus, batching is often paired with timeouts to bound the delays.

Edge-RT enables batching of messages in each stage to reduce per-message data-copying, scheduling implementation, and event notification overheads (Δ_{msg}, Δ_{impl}, and Δ_{evt}, respectively). Recall that a 40GiB link can generate 80M packets/second and multiple stages require additional message transfers. This motivates avoiding system calls, IPC, and anything other than shared memory access of batched messages. As discussed in §III-A, stage computation in Edge-RT inherits the priority of the nearest deadline of buffered messages. Unfortunately, this policy can lead to unpredictable execution if a stage inherits a priority $d_{i,m}$, finishes processing the message, and then processes a message with $d_{i,m+1} \gg d_{i,m}$[2]. The stage inherits the higher priority processing of $m_{i,m+1}$, which leads to priority inversion. In this example, inversion occurs when the stage processing $m_{i,m+1}$ inherits $d_{i,m}$ when a stage is runnable with deadline $d_{j,k}$ where $d_{i,m} < d_{j,k} < d_{i,m+1}$.

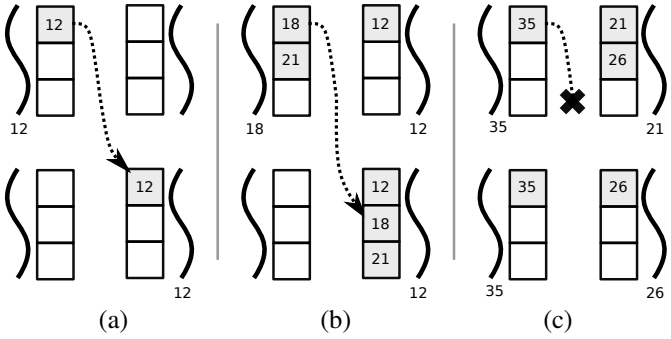


Fig. 4: Before (top) and after (bottom) system states when a stage transmits a message from the previous stage (left thread) to the next stage (right thread). A stage's (thread's) inherited deadline is below the thread, and greyed out messages are annotated with their deadline. Each of (a), (b), and (c) demonstrate batching deadline inheritance where $\Delta_{\text{batch}} = 10$.

We identify this priority inheritance as a key challenge in achieving both end-to-end packet scheduling for latency-centric scheduling, and batching for performance. To bound the impact of the inversion, Edge-RT creates a *deadline-limited batching* policy. This allows the batching a new

[2]Note here that calling the system after processing a message to update priority defeats the benefit of batching that avoids the per-message overheads for system calls, IPC, and scheduling.

message in the stage *only if* its deadline is at most a fixed amount later than any of the stage's already batched messages. That is, batching of $m_{i,n}$ is allowed only if $d_{i,n} - d_{i,n-m} \leq \Delta_{\text{batch}}$ where $m_{i,n-m}$ is the previously batched message with closest absolute deadline. Given this, message $m_{i,n}$ is always processed with an effective deadline in $[d_{i,n} - \Delta_{\text{batch}}, d_{i,n}]$.

Figure 4 shows an example of batching and deadline inheritance with the deadline-limited batching policy with $\Delta_{\text{batch}} = 10$. (a) A message with deadline 12 is transmitted to the next stage resulting in the stage inheriting this deadline priority. (b) Two additional messages are transmitted. Packets are buffered as their deadlines fall within Δ_{batch} of the initial message's deadline, while the thread still inherits the closest deadline (12). (c) The left stage attempts to transfer a message with deadline 35, but it cannot be buffered as processing of that message could inherit too high a priority ($35 - 21 > \Delta_{\text{batch}}$). As execution continues, and messages are processed (bottom right), the deadlines fall within the allowed window, and a transfer could commence.

Batching of messages in computation stages presents a trade-off. Larger Δ_{batch} values enable the batching of more messages. Though this decreases the costs of Δ_{evt} and Δ_{impl}, it increases impact of priority inversion, bounded by Δ_{batch}.

We characterize the impact of the trade-off around Δ_{batch} in Figure 6a. For this experiment we use 16 cores. To stress the system, a workload generates 90% utilization across cores that run stages. We use a bimodal workload which consists of light and heavy tasks to emphasize scheduling accuracy. Each task is served by a chain of four computation stages, each with identical execution times. The workload includes 160 clients, thus 480 computation stages in total. Light computations execute for 20μs with a deadline of 5ms which emulates the Extended Kalman Filter [32] test application on GPS data. Heavy computations execute for 5ms with a deadline of 500ms which emulates the CMSIS-NN [33] neural network inference on one input image from the CIFAR-10 [34] dataset. Each client sends requests using minimal UDP packets at a constant rate: 625 packets per second per task for computation light tasks and 20 packets per second per task for computation heavy tasks. Figure 6a displays what fraction of the latency-sensitive light computation packets are dropped (due to buffers filling), miss their deadlines, and meet their deadlines. See §VI for more details on experiment setup.

The values of Δ_{batch} are chosen to show the progression from overload into inversion. The results demonstrate that without batching ($\Delta_{\text{batch}} = 0$), the system drops around 20% packets, which indicates overload due to increased system overheads. This validates that batching is a requirement for high-throughput systems which result in frequent activations/deactivations. The fraction of deadlines missed decreases till $\Delta_{\text{batch}} = 8ms$, and increases thereafter. The decrease up to $8ms$ is due to smaller per-message overheads as batching decreases the impact of Δ_{evt} and Δ_{impl}.The increase after $8ms$ is due to increased priority inversion.

Figure 5 demonstrates a timeline of processing over three stages with the potential delays, including Δ_{batch} in the deadline bounds for message processing.

B. Periodic Event Notification

In conventional systems, inter-core notifications and coordination often use locks to synchronize access [23], and IPIs

Fig. 5: A timeline of processing a packet through stages $s_i^1, \ldots, s_i^3$. In between stage computation, *maximum* system overheads for MMA processing (Δ_{msg}), event notification triggered by frequent timers (Δ_{timer}), and scheduler implementation (Δ_{impl}), impact end-to-end latency. The *deadline* used by the scheduling logic (including priority inversion) for the end-to-end processing of each message/packet, d, is bounded by batch and timer window size.

for immediate event notification (*e.g.*, to trigger a reschedule). Unfortunately, these potentially per-message costs to activate the next stage's thread, Δ_{evt}, cause significant overhead and inversion due to mutual exclusion, cache-coherency, and high-priority IPI execution [35].

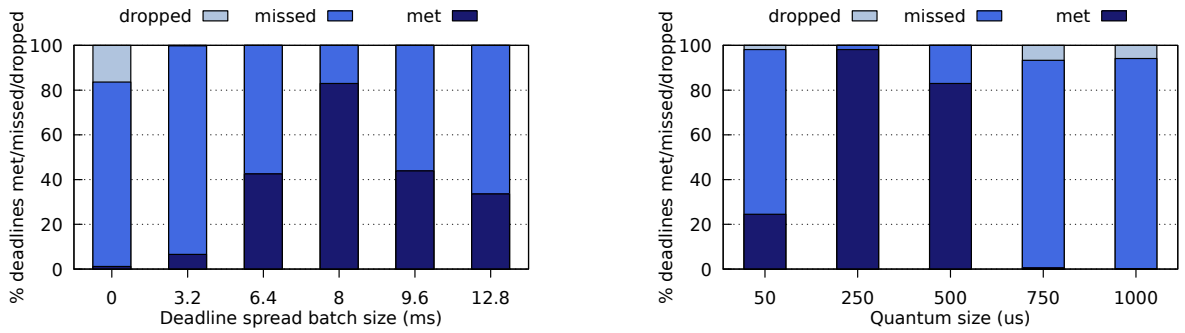
In contrast, Edge-RT aims to largely remove per-message overheads for event notification and stage activation. Edge-RT uses partitioned scheduling to avoid locks and cache-coherency overheads. The MMA processes stage transmissions, copies the message, and notifies the receiving stage's core's scheduler logic (all of this constitutes Δ_{msg}). These stage activation notifications to the scheduler from the MMA are passed via wait-free queues. To avoid the potentially per-message overheads of IPIs, Edge-RT instead uses per-core, periodic timers to *poll* MMA notifications. This has the benefit of requiring near constant overhead, but also delays notification processing by up to the timer periodicity, Δ_{timer}, potentially causing bounded priority inversion. Specifically, a stage s_i^x activated by the MMA by time $t_i^{x-1} + \Delta_{\text{msg}}$ will have its activation delayed till up to $t_i^{x-1} + \Delta_{\text{msg}} + \Delta_{\text{timer}}$. Though this can result in priority inversion, it is naturally bounded by Δ_{timer}. Thus, Δ_{evt} is dominated by Δ_{timer} in Edge-RT.

The value of Δ_{timer} represents a trade-off between tighter bounds on priority inversion on stage activation, and the overhead of frequent timer processing. Using the same workload described in §IV-A, Figure 6b depicts the behavior of the light tasks for various Δ_{timer} values. we choose Δ_{timer} to show the progression from overload (due to frequent timer processing) to interference (heavy computations significantly delay the execution of light computations). With a frequent timer (50μs), the overhead of handling timers is significant, causing deadline misses. As the quantum is increased to 250μs, the system achieves high ($> 98\%$) rates of met deadlines. As the quantum is increased to 500μs, over 15% deadlines are missed, and when $\Delta_{\text{timer}} \geq 750\mu s$, all deadlines are missed. Given the tight deadlines of light tasks ($5ms$), delays of up to $1ms$ for each of four stages of computation can cause misses. Guided by these results, Edge-RT uses $\Delta_{\text{timer}} = 250\mu s$. Though this is higher frequency than traditional systems, it enables Edge-RT to maintain a constant level of overhead despite potentially frequent activations.

Figure 5 demonstrates a timeline of processing over three stages with the potential inter-stage delays, including Δ_{timer}.

C. Constant-time EDF Scheduling

Most real-time systems are designed for 10s to the low 100s of latency-sensitive tasks. However, Edge-RT targets



(a) Max. batch deadline spread, Δ_{batch}, in milliseconds.

(b) Quantum, Δ_{timer}, in μ-seconds

Fig. 6: The impact of maximum spread between batched messages (Δ_{batch}) and the quantum size (Δ_{timer}) on system deadlines. Each bar is stacked with lightest shade being packets dropped, then deadlines missed, then the darkest shade is deadlines met, all for the lightest workload.

the edge whose limited hardware resources require high density, multi-tenant workloads. Thus we design it to handle 1000s of stage computations. At this level of load, with fast per-message computation, even the logarithmic overheads of EDF's deadline sorting are significant. Given this, Edge-RT investigates a *Constant-Time EDF* (CT-EDF) design that achieves constant Δ_{impl} by trading a bounded coarsening of deadline resolution.

Instead of using a typical $O(log(N))$ balanced tree data structure sorted by the deadline, CT-EDF's core data structure is implemented using three core techniques:

1) quantize time into a set of fixed-sized *quanta*, Δ_{window},
2) track a relative timeline of quanta as an array, and
3) each quanta index holds a linked list of stage computations with deadlines within that quanta into the future.

To find the stage with the nearest deadline, requires a scan of the array. Though this is constant (given a fixed-size array), it is expensive. Thus, inspired by fixed priority implementations, we use a trie of bitmaps to index the timeline and provide both constant and efficient lookup of the stage with earliest deadline. A side-effect of CT-EDF's quantization is that all computations with deadlines that fall into the same quantum are identically prioritized. This effectively means that messages inherit a deadline at most Δ_{window} higher than its normal deadline (with reasoning similar to that for Δ_{batch}).

As overflow conditions aren't a focus of this research, we choose a simple policy for when a stage's thread misses its deadline. Threads that miss their deadline are run at a lower priority than EDF threads, and are arbitrated using round-robin (to prevent starvation among such threads) with a quantum Δ_{timer}. Once a stage's thread finishes its computation, its next activation will place it back into the normal EDF scheduling logic.

Figure 5 demonstrates a timeline of processing over three stages with the deadline being bounded by Δ_{window}, and constant EDF contributing a small delay of Δ_{impl}.

D. Analysis of Timing Properties

Reflecting on the model in §III, Edge-RT aims to reduce per-message system overheads around Δ_{evt} and Δ_{impl}, while increasing $\Delta_{\text{interference}}$ by a bounded amount. Δ_{evt} is limited to the overhead of dequeuing from a wait-free ring buffer which is on the order of 200 cycles (depending on cache-coherency overheads from access patterns). Δ_{impl} is dominated by CT-EDF which is constant and low (see §VI-A).

The message passing overheads, Δ_{msg}, are due to the MMA, and constitute a straightforward delay in the activation of a receiving stage. To pass messages between stages, the MMA iterates through all inter-stage connections, processes the delayed batching logic, and sends/receives notifications from each core's scheduler logic. This adds a latency into message delivery, thus the activation of computations. Though we assessed adding intelligence to the MMA to separately prioritize different computation chains to control their latency, the performance of the MMA is sensitive, and such attempts negatively impacted system throughput by complicating the tight loop. We find that in a 48-core system (details in §VI) with almost two thousand computations, the latency for the MMA to cycle through all connections is around 1ms for a system at around 80% load.

There are two types of priority inversions in Edge-RT that contribute to $\Delta_{\text{interference}}$.

1) *Bounded priority inversion.* Event notification is performed *periodically* using polling (§IV-B), thus avoiding per-message overheads. However, this a message transmitted to a stage might be delayed by Δ_{timer}, causing a bounded inversion. Figure 5 depicts this overhead as simply contributing to the delay of the next stage's activation.
2) *Deadline coarsening.* Edge-RT's deadline-bounded batching (§IV-A) results in a message's computation potentially inheriting an earlier deadline only if the deadlines are within a sliding window of Δ_{batch}. Similarly, the quantization of deadlines to enable O(1) EDF (§IV-C) "bins" stages together with deadlines within a fixed window of Δ_{window}. Figure 5 depicts this as the deadline for the end-to-end processing of a message being $d \in [d_{i,m} - \max(\Delta_{\text{batch}}, \Delta_{\text{window}}), d_{i,m}]$. Were the workload predictable, this deadline inaccuracy could be compensated by practically lowering the system's effective utilization by a factor related to $\frac{d_i - d}{d_i}$. Recall that both Δ_{batch} and Δ_{window} are on the order of 0.5ms, which limits their impact.

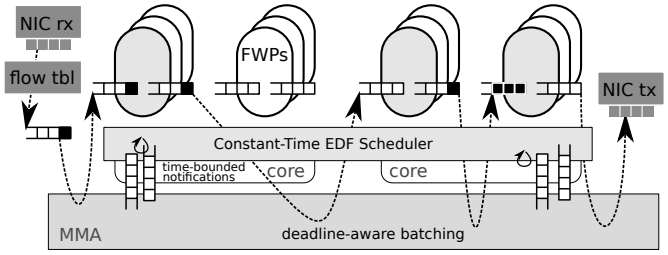


Fig. 7: Edge-RT components. Many FWPs execute for each client and tenant. The grey FWPs are a chain computing on a sequence of messages for a client. Received from the NIC, a packet's chain and deadline is identified through a flow-table. This deadline drives all resource management policy as the message processing flows through the system. The MMA copies messages (black squares) between the driver and FWPs, and between isolated FWPs. To wake FWPs, the MMA notifies the scheduler, passing the message's deadline. Notifications are handled with high-frequency time-triggers to avoid IPI overhead. The scheduler is a constant-time EDF that has FWPs inherit the deadline of messages. The MMA batches messages only up until message deadlines diverge by a limit to avoid interference. The scheduler notifies the MMA when additional messages can be transmitted. Finally, the MMA transfers messages to the NIC driver which transits them onto the network.

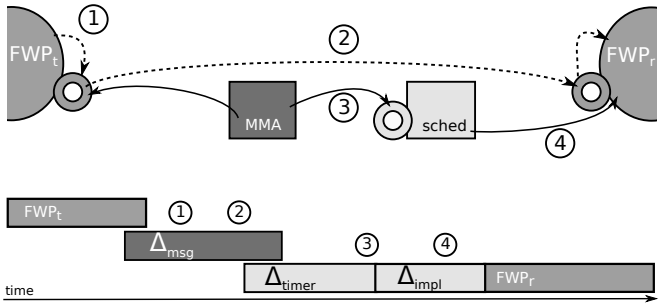


Fig. 8: Flow of data (dashed lines) and control (solid lines). Top: Message transfer from a transmitting FWP to a receiver, annotated with steps. Bottom: A timeline annotated with the various software and steps.

V. Implementation

Edge-RT extends EdgeOS (§II-C) which is built as a set of user-level components on the Composite μ-kernel [19]. Edge-RT changes the core policies for message movement, event notification, and scheduling to focus on timely per-client execution on edge clouds. We make no modifications to Composite kernel.

A. *Edge-RT Core Services*

To reduce per-message overheads, and provide end-to-end packet deadline scheduling, §IV introduces per-packet deadlines, deadline-aware batching, periodic inter-core coordination and constant-time EDF scheduling. Figure 7 gives an overview of Edge-RT. Edge-RT uses a number of mechanisms and policies to support the implementation of end-to-end packet deadline scheduling. (1) The client chain to process a packet that arrives from the NIC is identified using a flow-table. The flow-table maps IP/port values to the chain, and the relative deadline to use for the packet. Both the chain of computations, and this deadline are provided by the tenant. (2) The MMA is significantly enhanced to track FWP deadlines (described in §IV-A) and perform deadline-aware batching. (3) The MMA interacts with per-core scheduling logic by sending FWP activation events and receiving FWP blocking notifications. (4) the scheduler is replaced with a partitioned, preemptive, EDF implementation that uses constant-time logic, and frequent timer activations and polling for event handling and inter-core coordination.

B. *Deadline-aware, MMA-based Message Processing*

Figure 7 depicts the flow of messages through the system. Packets received by DPDK are queued into a per-chain queue with the deadline retrieved from the flow-table. The MMA transfers this packet to the first FWP's input queue, and sends an activation notification with the deadline of the FWP to the scheduler. As such, all deadline tracking and policy, from flow-table to transmission, are coordinated by system components.

The MMA is central to providing end-to-end packet scheduling, and Figure 8 depicts the coordination between MMA and scheduler to transmit a message between stages. The MMA's core role is to transfer messages between transmission and reception rings. When an FWP enqueues a message for transfer (①), Edge-RT's MMA does the following: (1) track the deadline of each message in *shadow message rings* that are inaccessible to FWPs; (2) transfer a message, $m_{i,m}$, from a transmission ring to a reception ring only if the deadline of each message already in the reception ring are in $[d_{i,m} - \Delta_{\text{batch}}, d_{i,m}]$ (②); (3) when message data is transferred between FWPs, also transfer their deadlines between shadow message rings; (4) when a transfer to an FWP is made, if there were no messages already in its ring, send an FWP thread activation notification to its core's scheduler logic (③); and (5) receive notifications from each core's scheduler logic for when an FWP blocks awaiting messages, which enables the MMA to transfer the next deadline-limited batch messages.

The scheduler will execute an activated FWP according to its EDF policy (④). Note that if a receiving FWP is already active, messages transferred to it are batched, and avoid event notifications.

A side-effect of the MMA's limitations on batching is that back-pressure is provided naturally. If an FWP is not executed, any messages awaiting transfer into it (with deadlines higher than $\Delta_{\text{batch}} + d_{i,m}$) are not transferred out of the upstream FWP. This logic repeats until the receive ring of NIC driver cannot receive any more packets for the flow, thus dropping packets. Though this outcome isn't ideal, deadline-aware work shedding is necessary in an over-committed system.

C. *Inter-core Coordination*

FWPs on separate cores interact through the MMA and its coordination with core scheduling logic. Conventional implementations of inter-process coordination use shared data-structures and IPIs with the accompanying costs [35]. In contrast, Edge-RT's design minimizes these potentially per-message overheads by remaking OS coordination primitives based on message passing and frequent periodic polling for events. Edge-RT implements a new user-level scheduler [36], [37] that integrates with the MMA to enable this coordination.

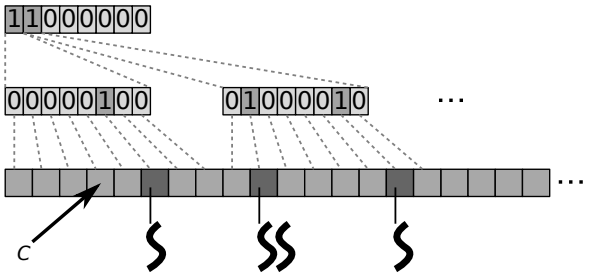


Fig. 9: CT-EDF data-structures. The current time is marked "current", and the two-level bitmap index tracks times into the future with runnable threads.

This scheduler's key functionalities include the following. (1) When switching to a thread (*i.e.* FWP), it passes a timeout in cycles which is used to program the LAPIC timer, at which point a timer interrupt will reactivate the scheduler. We reprogram the timer to happen periodically at Δ_{timer} to bound notification latency. (2) The scheduler currently uses simple partitioned scheduling. Each core's scheduler code shares two wait-free ring buffers with the MMA used to pass notifications in both directions. When the timer activates the scheduler, it polls for FWP activation notifications from the MMA and handles them. Each notification includes a *deadline* with which to execute the FWP. As such, this tight coordination between MMA and scheduler enables the thread's inheritance of message deadline. (3) When an FWP blocks after finishing message processing, the scheduler uses the other ring to send a corresponding notification to the MMA. This enables the MMA to resume transferring messages to the FWP if more were pending, but not sent due to the deadline-aware batching.

D. Constant-Time Earliest Deadline First Scheduling

A traditional implementation of EDF sorts threads by deadline often using a balanced binary tree (*e.g.*, a min-heap or red-black tree). Most real-time systems contain a small number of tasks (on the order of 10s or low 100s), so any data-structure overheads are sufficiently small. However, Edge-RT is focused on dense, multi-tenant execution of chains of concurrent computations, thus must be able to run on the order of 1000s of threads (>6000 in §VI). This has two negative effects: (1) with such high density, the frequency of FWP activation/blocking – bounded only by packets per-second – forces frequent scheduling decisions, and (2) the number of active threads makes the per-scheduling decision sorting overheads have a non-trivial impact.

CT-EDF enables constant time and fast EDF decisions. The key idea is that the timeline is quantized into fixed quanta of a specific length, Δ_{window}, and is represented as a circular array (*i.e.* with wrap-around logic) with an entry-per-quantum. If the array has S entries, it tracks up to $\Delta_{\text{window}} \times S$ time units into the future. The current time, C, defines an index into the array, and all times into the future are indexed from there (treating the array as a circular array). Calculating times relative to the current time means that at each quanta, only C needs to be incremented rather than shifting all entries down. Finding the FWP with the lowest deadline means finding the *first quantum* that contains an FWP starting the search at the index C. At each quanta, any tasks that miss their deadlines are placed into the lower-priority, round-robin queue. Future enhancements can include more intelligent failure logic. In this paper, we restrict our focus on meeting end-to-end packet deadlines.

The data-structures for CT-EDF are depicted in Figure 9. Finding the first quantum into the future involves iterating through potentially the entire array, which is unacceptably expensive. As such we use a set of bitmaps to index the quanta that have runnable FWPs. If a quantum in the timeline has runnable FWPs, the bit corresponding to that quanta is 1. We use the `clz` instruction to efficiently *count the leading zeros* within the bitmap. Additionally, to avoid needing to iterate through the words of the bitmap – using `clz` on each – we define multiple levels (2 in our case) of bitmaps that each index the next level. At each level, a bit is 1 if the corresponding next level bitmap indexes a quantum with runnable FWPs. This multi-level bitmap index forms a radix-trie, and enables constant-time, efficient identification of the earliest-deadline FWPs (with accuracy within Δ_{window}).

Edge-RT uses a 32-bit level-1 index, that index 32, 32 bit level-2 indices (for a total of 1024 tracked quanta). This enables using only two `clz` to determine the earliest deadline. Each CT-EDF quantum is $0.5ms$ for a maximum deadline of $0.5s$. The timeline can be extended by adding another index level (to support $16s$), or increasing the quantum.

VI. Evaluation

This section evaluates Edge-RT relative to EdgeOS and Linux. All experiments use a PowerEdge R740 servers with two socket Intel(R) Xeon(R) Platinum 8160 CPUs @ 2.10GHz each with 24 cores. We use an Intel X710 for 10GbE SFP+ for networking and a similarly equipped client drives workload generation. DPDK and OVS versions for the results in §II-A are 19.11 LTS and 2.13, respectively. Linux results use kernel version 5.4. We don't use the `PREEMPT_RT` patches as tight interrupt response times (on the order of a μs) are not the focus. Client machines use a modified `memblaster` to generate workloads and measure round-trip latencies. Edge-RT uses DPDK version 17.11.

A. CT-EDF Overheads

CT-EDF uses a constant-time data-structure to find the task with the earliest deadline (§IV-C). Figure 10 investigates

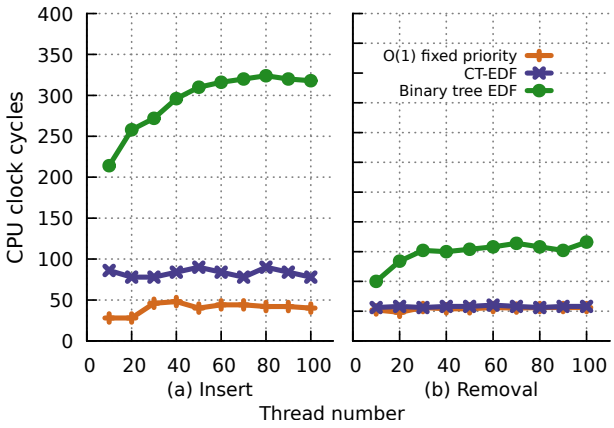
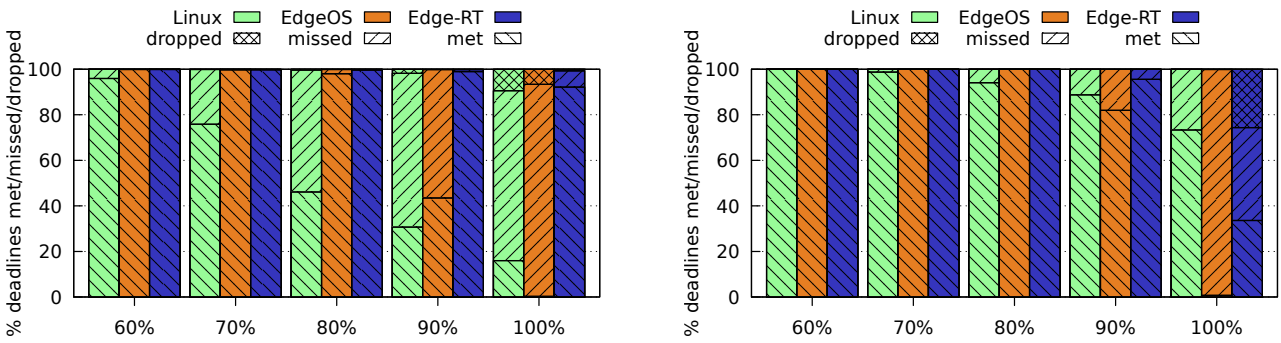


Fig. 10: The overheads for key data-structure operations with an increasing runqueue size. The y-axis plots the average overhead.



(a) The behavior of light tasks with increasing utilization.

(b) The behavior of heavy tasks with increasing utilization.

Fig. 11: Comparison between Edge-RT and Linux for a bimodal workload. y axis is the fraction of deadlines met, missed and fraction of packets dropped.

the overhead of insertion and removal for scheduling data-structures. We compare the following. (1) A traditional O(1) fixed priority, round-robin (`O(1) fixed priority`) structure which includes an array of linked lists that track threads within a priority. It uses a two-level bitmap index – queried with the `clz` instruction – that tracks priorities with active threads. (2) CT-EDF is detailed in §V-D. (3) A traditional EDF implementation (`Binary tree EDF`) using a red-black tree to sort threads by deadline. Not shown here, we used an in-place heap and found it to have more overhead.

Threads with uniformly random deadlines are added to the queue (*i.e.* they are woken), and the thread with the highest priority (earliest deadline) is removed from the queue (*i.e.* blocks).

Discussion. With an increasing number of threads, the traditional EDF with $O(log(N))$ complexity imposes increasing overheads, while the other two approaches demonstrate near-constant overheads. The CT-EDF policy demonstrates higher overhead than fixed priority when inserting, despite using a similar index. This is due to wrap-around logic in CT-EDF: 1) if the current time is half-way into the timeline, the `clz` operations must be performed only on indices after a circular shift has been performed; and 2) at the second level index node, it is possible not all bits should be queried (if the current time is indexed by this node).

B. Linux and Edge-RT Utilization Sensitivity

§II argues that system organizations and policies that target high throughput such as kernel-bypass and eBPF, or predictable execution such as `SCHED_DEADLINE`, have trouble scaling to multi-tenant, deadline-driven environments. Table I summarizes these arguments. Figures 2 and 1 demonstrate that Linux's CFS scheduling, and normal sockets perform the best with increasing tenants and uncertain workloads. Therefore, we configure Linux using sockets for networking and CFS for scheduling. To provide the same level of isolation as Edge-RT, we use separate processes for each stage in Linux. For chain processing on Linux, pipes slightly outperform socket variants for event notification latency. Thus, we use pipes for IPC between processes in chains. To avoid data copying costs, we pass only 8 byte messages to provide event notification while larger messages use shared memory.

In this evaluation, we configure both Linux and Edge-RT to use 48 cores, with Edge-RT specializing four cores (§II-C). Thus Edge-RT has a practical maximum utilization of around 91.6%. Figure 11 depicts a bimodal workload with a varying system utilization. Utilization is reported as a fraction of application computation of the 48 cores. Edge-RT is configured with $\Delta_{\text{batch}} = 8ms$ and $\Delta_{\text{timer}} = 250\mu s$ guided by Figure 6a and Figure 6b. On the client side, we use the same bimodal workload from Figure 6a with light tasks that execute 40μs, and heavy that execute 5ms with deadlines 10ms and 500ms, respectively. As a result, light tasks comprise 80% of the total execution of the workload. We control the utilization of the workload by adjusting the sending rate (proportionately for light and heavy) on the client side. We use 480 clients/chains with 1920 FWPs.

We also compare against Linux using `SCHED_DEADLINE`. To admit enough tasks onto the system to be able to use 100% utilization, we set the CFS budgets equal to the average execution time for heavy and light. We omit these results as light tasks meet only 0.25% of their deadlines at 60% utilization while even heavy tasks only meet 57% of deadlines at 60% utilization, decreasing to 32% at 100% utilization. This provides further evidence for the claims in §II-A that `SCHED_DEADLINE` is not a good fit for systems with dynamic workloads that are not periodic.

Discussion. Figure 11a shows high-throughput systems have difficulty meeting deadlines. As the utilization increases from 60% to 100%, the percent of deadlines met by Linux drops from 95.9% to 15.9%. Additionally, Linux starts dropping packets on light flows when utilization grows over 90%. In case of heavy tasks in Figure 11b, Linux does not drop packets, and meets deadlines even under 100% utilization. Linux's best-effort focus on fairness favors heavy workloads despite CFS's heuristic prioritization of I/O-bound workloads.

EdgeOS meets most of the deadlines when the system is relatively low utilized (less than 80% utilization). Since EdgeOS is not deadline-aware and uses a fixed-priority round-robin scheduler, it misses most of the deadlines when utilization grows over 80% and barely meets any deadline at 100% utilization. Recall that the four specialized cores decrease the effective maximum utilization to around 91%. However, EdgeOS drops fewer packets for heavy tasks compared to Edge-RT at 100% utilization. This is due to

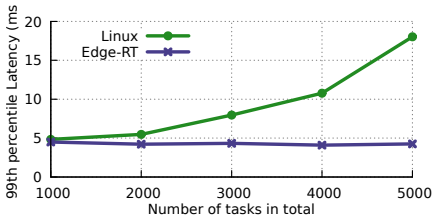


Fig. 12: The 99% latencies of light tasks with increasing number of tasks in total

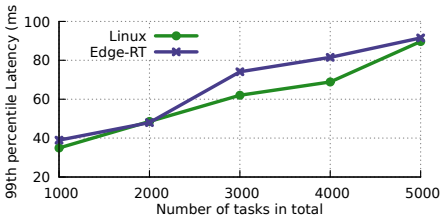


Fig. 13: The 99% latencies of heavy tasks with increasing number of tasks in total

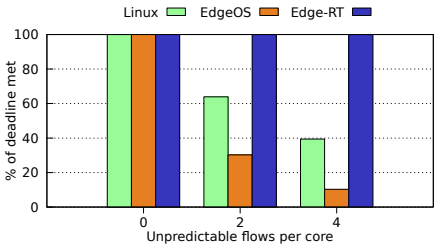


Fig. 14: Behavior of well-behaved tasks in the presence of malicious tasks.

EdgeOS's focus on throughput.

At 90% utilization, Edge-RT meets over 95% deadlines on *both* light and heavy computations. Even at 100% utilization, Edge-RT meets 92.13% deadlines on light tasks (compares to Linux's 15.95% and EdgeOS's 0%). In contrast, heavy tasks cause backpressure through chains, and Edge-RT drops 25.6% with continued progress on other tasks. This demonstrates Edge-RT's focus on end-to-end, deadline-centric scheduling, while still maintaining high throughput.

We don't compare against the kernel bypass techniques given challenges with non-preemptive client execution in §II-A. The round-trip latency for Linux sockets is only 20μs (15%) slower than DPDK using interrupt mode (which is necessary with more tenants than cores), so we believe that Linux is a competitive environment for these comparisons.

C. Scalability

Edge-RT is designed to use constant overhead mechanisms where possible. CT-EDF and periodic activation processing both enable schedulers to spend more time running FWPs regardless the number of FWPs. Thus, we want to evaluate the performance of Edge-RT when scheduling thousands of FWPs. Here we evaluate Edge-RT's and Linux's ability to control latency while increasing the number of processes. We use the same system setup as the previous bimodal tests, by default. As we scale up the number of tasks, we proportionately adjust the sending rate for light and heavy tasks to keep the system utilization equal to 50% to avoid either system dropping packets. Thus, the 99th percentile latency depicts the system's ability to bound latency.

Discussion. As shown in Figure 12, with 1000 computations, Linux and Edge-RT have similar tail latencies around 5ms. Edge-RT maintains a flat tail latency between 4 and 5ms up to 5000 tasks. In contrast, the tail latency of light tasks on Linux grows as the number of tasks increases. At 5000 tasks, the tail latency of light tasks in Linux is 18ms (recall: deadline 10ms). The tail latency of heavy tasks for both Linux and Edge-RT increases when having an increasing number of tasks in Figure 13, but both remain below their deadline of 500ms. Linux slightly outperforms Edge-RT on heavy tasks, because CFS's fairness ensures constant progress for heavy tasks over light tasks. These results demonstrate that Edge-RT can maintain its focus on end-to-end deadline scheduling despite significantly scaling up the number of clients.

D. Unpredictable Workloads

The edge must be able to handle workloads with unpredictable execution. Tenants can provide code that attempts to monopolize CPU resources, and clients can provide inputs to cause errant behavior. To evaluate the impact of execution overruns, we maintain a consistent workload, with an increasing number of malicious tasks. We use 48 cores and 768 clients with 768 FWPs. System utilization is 80% without malicious tasks. The well-behaved tasks execute for 5ms and clients send a request every 100ms with a 500ms deadline. Chain lengths are set to one ($K = 1$) to emphasize that malicious tasks can cause significant interference even without back-pressure from chains of computations. Malicious tasks are simply infinite loops to monopolize the CPU. Figure 14 shows the behavior of well-behaved tasks in the presence of CPU-hogging malicious tasks. In these results, we filter out the deadlines missed due to the malicious tasks themselves.

Discussion. With even a small number of malicious tasks, EdgeOS's round-robin scheduling policy – that focuses on progress and fairness – demonstrates an inability to adequately isolate the deadline-sensitive tasks from those that are simply CPU-hogs. At only two malicious tasks per core, on average, nearly all deadlines are missed.

CFS in Linux fairs better with this workload. This is because it attempts to prioritize newly arrived (I/O-bound) threads over malicious tasks who are CPU-bound. As such, it is able to meet 63% of the deadlines with two malicious tasks present, but quickly degrades to 39% with four.

Once a deadline is missed in Edge-RT, the FWP is deprioritized and moved to a best-effort round-robin low priority queue. This simple policy is relatively effective at constraining this malicious computation by deprioritizing it. This demonstrates the unintuitive benefit of a end-to-end, deadline-centric policy for the edge: deadlines provide semantic information about expected execution behavior. The edge can use them to constrain FWP's negative behaviors when deadlines are overrun. Though stronger assumptions about FWP execution times could further constrain their impact – for example, by enabling rate-limiting – such assumptions are challenging in the high-throughput, dynamic environments.

VII. Related Work

Edge applications. Offloading computation from devices to infrastructure has a long history. For example, it has been shown to prolong batteries [38], [2], and even service mission-critical tasks [39] with degraded local computation. It enables global coordination to get benefit beyond a system's local sensors [40]. Edge-RT assumes an unreliable network, but one that is low-latency enough to motivate deadline-sensitive computation. It attempts to enable these benefits in a high-throughput, high-density, and real-time infrastructure. This

assumption does not prevent Edge-RT from including fall-back logic to enable detection and retransmission of dropped packets in the future.

Shared hardware resources. Edge-RT focuses mainly on sharing NIC hardware resources. FWPs share no memory as the MMA arbitrates copying messages between them. We do not focus on shared resources like caches [41], [42], [43], [44] and memory buses [45]. Approaches that increase their sharing and access are complementary to Edge-RT.

Data-age in cause-effect chains. Chains of computation are a common abstraction in embedded systems (*e.g.*, robotics systems such as ROS) and represent stages of computation from sensing to actuation. They are often called *cause-effect chains*, and real-time systems are concerned with controlling and constraining the *data-age* (the latency of chain computation from sensing to actuation).

Large bodies of work have focused on creating policies and analysis to provide predictable, bounded data-age across chains [46], [47], [48], [49], [50]. Recent work [51] focuses on end-to-end response time analysis of event chains. This work is often concerned with strong predictability and meeting all deadlines, and does so by assuming knowledge about execution properties such as fixed rates and WCETs.

Edge-RT is focused on controlled latency in a high-throughput environment. Instead of the traditional approach that often separately prioritizes chain computations, controls their periodicity, and orchestrates dependencies, Edge-RT focuses on *end-to-end deadline scheduling of messages*, rather than of computations.

Blass et al. [52] take a more dynamic approach by monitoring latencies and dependencies in chains, and dynamically updating priorities, reservations, and affinities to control end-to-end latency. Edge-RT doesn't attempt to optimize parameters to control latencies, and instead explicitly maintains end-to-end deadline scheduling mechanisms and policies.

Network Function Virtualization. The edge must execute not only multi-tenant, offloaded computation, but also Network Functions (NFs) for slicing, transformation, and shaping. Network Function Virtualization (NFV) [53] provides a software environment for the isolated execution of NFs. Edge-RT is based on EdgeOS [9], which supports high-throughput, high density NFV, but additionally aims to provide strong latency properties. Similarly, systems have focused on soft-real-time NFV computation [54], [55], but are not focused on high-throughput systems nor resource management focused on average computation (*not* WCET).

Rethinking OS-level batching and scheduling. The trade-off in batching between throughput and latency is well known and has enabled efficient core specialization [56], and a successful decoupling of OS control and datapaths [27]. It has also been an integral technique in shaping back-pressure [57], controlling cross-core interference [35] in real-time systems. It has been paired with fine-grained inter-core coordination to control μ-second level latencies in the presence of head-of-line blocking [58]. User defined scheduling [59], [30] is another approach which aims to reduce latency by deploying a user-defined scheduling policy. Edge-RT is inspired by these approaches, and uses batching to achieve high throughput, but constrains the impact that large batches of processing can have on the latency of other system computations.

Summary. Edge-RT's contributions are in a less-investigated direction for real-time systems: high-throughput, network systems that aim to meeting deadlines despite high-density, multi-tenancy. The techniques it must use to achieve all of these goals aim to practically meet end-to-end deadlines while making efficient use of limited resources.

VIII. Conclusions

In this paper we introduce the Edge-RT. It represents a significant step into the direction of novel real-time policies and mechanisms for high-throughput, and high-density multi-tenant systems. Edge-RT introduces a practical policy to control latency over a chain of isolated computations: deadlines are associated with packets/messages, and computations *inherit* them, enabling optimization toward controlled end-to-end latency. We investigate policies that increase throughput, while having limited impact on latency. Results demonstrate that the system can effectively handle significantly higher load while meeting deadlines, as the number of clients scales up, and in the presence of malicious tasks, thus enabling higher edge density and decreasing the amount of computational resources at each basestation. We believe this marks a significant advance toward principled latency management for the multi-tenant edge.

References

[1] D. Xu, A. Zhou, X. Zhang, G. Wang, X. Liu, C. An, Y. Shi, L. Liu, and H. Ma, "Understanding operational 5g: A first measurement study on its coverage, performance and energy consumption," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2020.

[2] B. Li, W. Dong, and Y. Gao, "Wiprog: A webassembly-based approach to integrated iot programming," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021.

[3] "Telecommunications industry association. edge data centers. https://www.tiaonline.org/wp-content/uploads/2018/10/TIA_Position_Paper_Edge_Data_Centers-18Oct18.pdf," 2018.

[4] "Micro-data centers out in the wild: How dense is the edge?, https://www.datacenterknowledge.com/archives/2017/05/02/edge-densities," 2017.

[5] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca, and J. Folgueira, "Network slicing for 5g with sdn/nfv: Concepts, architectures, and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 80–87, 2017.

[6] "5g network slicing in 5gtango, https://www.5gtango.eu/blog/36-5g-network-slicing-in-5gtango.html," 2019.

[7] "Ngmn alliance, description of network slicing concept," 2017.

[8] P. Rost, C. Mannweiler, D. S. Michalopoulos, C. Sartori, V. Sciancalepore, N. Sastry, O. Holland, S. Tayade, B. Han, D. Bega, D. Aziz, and H. Bakker, "Network slicing to enable scalability and flexibility in 5g mobile networks," *IEEE Communications Magazine*, 2017.

[9] Y. Ren, G. Liu, V. Nitu, W. Shao, R. Kennedy, G. Parmer, T. Wood, and A. Tchana, "Fine-grained isolation for scalable, dynamic, multi-tenant edge clouds," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[10] "Intel Data Plane Development Kit (DPDK). http://dpdk.org/."

[11] "Open vSwitch (OVS). https://www.openvswitch.org/."

[12] "Virtual I/O devices. https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html."

[13] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[14] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of nfv," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[15] "extended Berkeley Packet Filter (eBPF). https://ebpf.io/."

[16] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th international conference on emerging networking experiments and technologies*, 2018, pp. 54–66.

[17] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8.

[18] L. Abeni, G. Lipari, and J. Lelli, "Constant bandwidth server revisited," *SIGBED Review.*, vol. 11, no. 4, 2015.

[19] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[20] "Ngmn alliance, 5g white paper," 2017.

[21] "Ngmn alliance, 5g end-to-end architecture framework," 2017.

[22] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, vol. 16, no. 3, pp. 33–38, 2019.

[23] A. Biondi and B. B. Brandenburg, "Lightweight real-time synchronization under p-edf on symmetric and asymmetric multiprocessors," in *28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[24] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, 1990.

[25] L. Soares and M. Stumm, "{FlexSC}: Flexible system call scheduling with {Exception-Less} system calls," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[26] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Trans. Comput. Syst.*, vol. 33, no. 4, Nov. 2015.

[27] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane," *ACM Trans. Comput. Syst.*, vol. 34, no. 4, 2016.

[28] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, S. Teodorescu, C. Răducanu *et al.*, "Unikraft: fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 376–394.

[29] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar *et al.*, "The demikernel datapath os architecture for microsecond-scale datacenter systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 195–211.

[30] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, "ghost: Fast & flexible user-space delegation of linux scheduling," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 588–604.

[31] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 361–378.

[32] "TinyEKF: Lightweight C/C++ Extended Kalman Filter with Python for prototyping, https://github.com/simondlevy/TinyEKF.git," 2019.

[33] "CMSIS NN Software Library, https://arm-software.github.io/CMSIS_5/NN/html/index.html," 2019.

[34] "The CIFAR-10 dataset, https://www.cs.toronto.edu/ kriz/cifar.html," 2009.

[35] P. K. Gadepalli, G. Peach, G. Parmer, J. Espy, and Z. Day, "Chaos: a system for criticality-aware, multi-core coordination," in *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[36] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2008.

[37] P. K. Gadepalli, R. Pan, and G. Parmer, "Slite: OS support for near zero-cost, configurable scheduling," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 160–173.

[38] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, 2010.

[39] L. Schönberger, G. von der Brüggen, K.-H. Chen, B. Sliwa, H. Youssef, A. K. R. Venkatapathy, C. Wietfeld, M. ten Hompel, and J.-J. Chen, "Offloading Safety- and Mission-Critical Tasks via Unreliable Connections," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, 2020.

[40] C. Wang, C. Gill, and C. Lu, "Frame: Fault tolerant and real-time messaging for edge computing," in *Proceedings of the IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[41] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *11th International Symposium on High-Performance Computer Architecture*. IEEE, 2005, pp. 340–351.

[42] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, pp. 45–54.

[43] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 89–102.

[44] J. Kim, I. Kim, and Y. I. Eom, "Code-based cache partitioning for improving hardware cache performance," in *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, 2012, pp. 1–5.

[45] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, pp. 55–64.

[46] M. Günzel, K.-H. Chen, N. Ueter, G. von der Brüggen andMarco Dürr, and J.-J. Chen, "Timing analysis of asynchronized distributed cause-effect chains," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[47] T. Klaus, M. Becker, W. Schröder-Preikschat, and P. Ulbrich, "Constrained age with job-level dependencies: How to reconcile tight bounds and overheads," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[48] A. M. Kordon and N. Tang, "Evaluation of the age latency of a real-time communicating system using the let paradigm," in *32nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2020.

[49] H. Choi, Y. Xiang, and H. Kim, "Picas: New design of priority-driven chain-aware scheduling for ros2," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[50] D. Casini, T. Blaß, I. Lütkebohle, and B. Brandenburg, "Response-time analysis of ros 2 processing chains under reservation-based scheduling," in *31st Euromicro Conference on Real-Time Systems*. Schloss Dagstuhl, 2019, pp. 1–23.

[51] D. Dasari, M. Becker, D. Casini, and T. Blaß, "End-to-end analysis of event chains under the qnx adaptive partitioning scheduler," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2022, pp. 214–227.

[52] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic latency management for ros 2: Benefits, challenges, and open problems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[53] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "OpenNetVM: A Platform for High Performance Network Service Chains," in *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2016. [Online]. Available: http://faculty.cs.gwu.edu/timwood/papers/16-HotMiddlebox-onvm.pdf

[54] Y. Li, L. T. Xuan Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *IEEE INFOCOM - The 35th Annual IEEE International Conference on Computer Communications*, 2016.

[55] S. Abedi, N. Gandhi, H. M. Demoulin, Y. Li, Y. Wu, and L. T. X. Phan, "Rtnf: Predictable latency for network function virtualization," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[56] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *Proceedings of the conference on Symposium on Operating Systems Design & Implementation*, 2010.

[57] M. Welsh, D. Culler, and E. Brewer, "Seda: an architecture for well-conditioned, scalable internet services," in *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 2001, pp. 230–243.

[58] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for µsecond-scale tail latency," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2019.

[59] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis, "Syrup: User-defined scheduling across the stack," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 605–620.