

# From Threads to Fresh Frames: Cross-Layer Modeling and Scheduling of RealSense on Linux

Yuwen Shen<sup>1,2,3</sup>, Constantin Scholz<sup>1,3</sup>, Bram Vanderborght<sup>1,3</sup>, Wouter Joosen<sup>2</sup>, and Antonio Paolillo<sup>1</sup> (✉)

<sup>1</sup> Vrije Universiteit Brussel, Brussels, Belgium  
`{yuwen.shen, constantin.florian.scholz, bram.vanderborght, antonio.paolillo}@vub.be`  
<sup>2</sup> KU Leuven, Leuven, Belgium  
`wouter.joosen@kuleuven.be`  
<sup>3</sup> imec, Leuven, Belgium

Author version with appendices. Accepted for publication in the proceedings of the 34th International Conference on Real-Time Networks and Systems (RTNS 2026), Lecture Notes in Computer Science, Springer.

**Abstract.** Robotic perception requires both timely delivery and fresh sensor data. Even when a color-and-depth (RGB-D) camera delivers data at regular intervals, individual streams may reuse old frames or skip inputs. Application deadlines alone therefore cannot establish that sensor delivery is timely and fresh. We investigate how these failures arise along the acquisition path, from operating-system processing to camera software and application delivery. Using Intel RealSense D400 cameras on Linux as a case study, we combine execution tracing with a model of interacting acquisition tasks and their CPU, USB, and memory demands. The model captures tasks that run at different rates and block while waiting for data. Experiments show that delayed kernel processing of USB data can disrupt delivery even when camera-software threads receive high priority. Coordinating the scheduling of kernel receive services and camera-software workers reduces deliveries with reused or skipped frames by 99.3–99.8% under targeted CPU starvation. In a separate 192-run evaluation, coordinated scheduling at 60 frames per second under CPU interference reduces the 99th-percentile latency from host reception to application delivery by 24.1–28.6% on a non-real-time Linux kernel and 35.6–36.0% on `PREEMPT_RT`. No reused or skipped frames are observed under the coordinated `PREEMPT_RT` treatments. These results motivate configuring receive buffers, kernel services, and camera-software workers together, and assessing delivery latency alongside per-stream freshness.

**Keywords:** real-time systems · RGB-D cameras · Linux scheduling · sensor data freshness.

## 1 Introduction

In autonomous and interactive robotic systems, RGB-D cameras provide complementary color and depth data for navigation, manipulation, obstacle avoidance,

and, in human-robot collaboration, for monitoring the shared workspace [35]. Timely delivery of these sensor data to software is critical because it directly affects when the system can respond to changes in the environment. A late, missing, or reused input can, for example, extend a robot’s reaction time and invalidate the timing assumptions of a safety function [38]. However, the configured frame rate describes nominal camera production, whereas application delivery also depends on the sensor software development kit (SDK) and the host operating system. A setting of 30 frames per second (FPS) therefore defines a nominal period of 33.3 ms, but does not guarantee that a complete set of newly captured sensor inputs reaches the application every 33.3 ms. In this paper, we call a delivered frameset *fresh* when every required stream contributes exactly one newly captured frame; if any stream repeats its previous frame or skips frame numbers, the frameset is non-fresh. Section 4.4 formalizes freshness and host-visible latency. Making delivery and freshness predictable therefore requires knowing which activities frame acquisition comprises, how the host schedules them, and which parts of the path scheduling cannot control.

When acquisition is delayed, the application can keep meeting its own deadlines while operating on stale data: the computation is on time, but its input no longer reflects the environment, and the system may miss the events it must react to. The hazard is therefore staleness; scheduling delay is its cause, not the failure itself. Freshness is an established requirement alongside timeliness in real-time databases [8], information-freshness metrics [44], and sensor-fusion scheduling for automated driving [39]; where freshness is lost inside the sensor acquisition stack of a general-purpose OS, however, has not been examined. Our target is robotic systems of this kind; we draw on the database and automotive literature for the notion of freshness, not for certification requirements.

This paper uses the Intel RealSense D400 Series on Linux as a case study; RGB-D cameras of this class are widely used in robotic perception, where Intel cameras in particular have become popular [34]. Applications access them directly through the open-source C++ SDK `librealsense` [30]. The SDK presents a simple pipeline interface: an application starts the camera and waits for synchronized framesets. This interface makes frame acquisition appear to be a single periodic task. The implementation, however, forms a multi-stage asynchronous pipeline involving camera firmware, USB transfers, kernel drivers, SDK workers (the threads `librealsense` creates inside the application process), stream synchronization, and an output queue. Contention for CPU or memory, or delayed service in the USB receive path, can delay one component stream and therefore the whole system. Raising only the application priority cannot recover data that was delayed or lost earlier in the USB or kernel path. The problem this paper addresses is how to make frameset delivery both timely and fresh on a Linux host. Frame acquisition spans SDK workers, kernel drivers, and USB transfers, which the default configuration schedules as one undifferentiated workload, while a kernel interrupt service routine serves interrupts outside scheduler control.

Our central insight is that recurring RealSense frame acquisition consists of interacting worker families, not one periodic task. We model it as a multi-rate,

self-suspending graph of these families and their kernel predecessors: eXtensible Host Controller Interface (xHCI) interrupt service and asynchronous USB Video Class (UVC) copy work. Worker threads are frame-, timer-, or event-triggered and may block on device or queue input. We reconstruct the graph using source inspection, LiME execution traces [1], a pthread interposer, application markers, kernel events, and frame metadata. The graph separates shared process-wide services from camera- and stream-local workers and links their timing to delivery. Resource accounting reports observed CPU utilization, nominal USB payload, and analytical memory traffic. The method can extend to other camera stacks, with timing and configuration determined per platform (Section 7). We address three research questions:

- RQ1—Structure and model:** Which worker families does `librealsense` use for steady-state frame acquisition, and how do their activation patterns, timing, and observed resource use change with the camera model and stream configuration?
- RQ2—Receive-path controls:** Beyond SDK userspace workers, which Linux receive-path mechanisms affect host receive-to-frameset latency and freshness, and how can they be controlled?
- RQ3—Coordinated scheduling:** How much do `PREEMPT_RT` and coordinated scheduling of SDK workers and kernel receive-path services improve host receive-to-frameset latency and freshness under interference?

This paper makes three contributions:

1. (RQ1) A source-informed model of recurring frame acquisition as multi-rate, self-suspending worker families, with trace-derived accounting of their CPU, USB, and memory demand.
2. (RQ2) A reproducible cross-layer trace method that locates where image reuse or loss first becomes visible along the SDK and Linux receive path, identifying xHCI interrupt service and asynchronous UVC payload-copy work as schedulable kernel predecessors of capture.
3. (RQ3) An evaluated configuration recipe to integrate RealSense-class cameras on real-time Linux: a deeper UVC receive pool, threaded interrupt service, and coordinated per-worker scheduling, validated across two workloads, four interference conditions, four policies, and two kernels.

The paper proceeds as follows. Section 2 introduces the SDK, the timing terms, and the Linux receive path; Section 3 positions the work. Section 4 derives the worker model and the latency and freshness metrics, which Section 5 instantiates in four experiments, E1 to E4, that answer RQ1 to RQ3. Sections 6 and 7 discuss limitations and applicability, and Section 8 concludes.

## 2 Background

### 2.1 D400 Cameras and Pipeline Semantics

Intel RealSense D400 cameras are stereo depth cameras that stream depth, color, and left and right infrared data over USB 3.1 Gen 1 [26]. We denote

the left and right infrared streams by IR1 and IR2, respectively. Applications configure and access these streams through the open-source, cross-platform SDK `librealsense` [30]. For each enabled stream, a stream profile specifies its resolution, pixel format, and frame rate. These choices determine how much data the camera produces and how much USB transfer and host-side processing are required. At the high-level interface, an application uses `rs2::config` to select a camera and its stream profiles. It then uses `rs2::pipeline` to start the configured streams and receive synchronized framesets (one frame from each required stream, joined by timestamp) through a blocking call or a callback [29].

The D400 series is representative of the stereo RGB-D cameras used in robotics: like the Stereolabs ZED [40] and Orbbec Gemini [22] families, it streams several synchronized image streams over USB 3 through a vendor SDK that supports capture via the host’s UVC/V4L2 stack, and empirical comparisons for robotic vision routinely include it alongside those cameras [34, 42]. Section 7 discusses which parts of the analysis transfer to those stacks.

Incoming frames pass through timestamp handling and synchronization inside the SDK before an aggregator publishes a frameset. This operation has join semantics: a late or missing required stream can delay the complete output [28, 30]. Inputs match when their timestamps differ by less than half the lower-rate stream’s nominal period. In `librealsense` v2.58.3, the synchronizer marks a missing stream inactive when incoming timestamps exceed its next expected timestamp by at least seven periods of the stream awaiting release [31]. This inactivity threshold is not a per-frame delivery deadline. A successful return does not prove that every stream advanced: newer inputs can be paired with a missing stream’s last valid frame. Section 4.4 formalizes this distinction.

## 2.2 Timing and Failure Semantics

A stream configured at rate  $f$  targets one new frame every  $1/f$  seconds. Delivery to the application matches this spacing only on average: individual intervals stretch under load, and the application programming interface (API) can keep returning at the nominal rate while reusing old data. The configured rate is thus an average production rate, not a per-frame delivery deadline or a freshness guarantee. At thread level, we adopt the thread states underlying LiME’s model extraction [1] and define execution time as CPU service, ready delay as time spent runnable but not executing, and blocking as waits for I/O, timers, queues, or events. A self-suspending Video4Linux 2 (V4L2) worker can therefore have a long response time but little CPU execution time.

At application level, inter-delivery time is the interval between consecutive frameset deliveries to the application (Figure 3). A duplicate means that a stream frame number does not advance; a sequence gap means that one or more numbers are missing. Either outcome makes the frameset non-fresh. A partially stale frameset contains both advanced and reused streams, while an API timeout means that no complete frameset became available within the configured interval. These observations describe different failure modes and do not alone identify the layer that caused them.

### 2.3 Linux Real-Time Scheduling

Linux provides several scheduling policies to control how runnable threads share CPU time. Its real-time policies can assign fixed priorities or reserve CPU time for individual threads. We consider `SCHED_OTHER`, `SCHED_RR`, `SCHED_FIFO`, and `SCHED_DEADLINE`. Hereafter, we abbreviate these policy names as `OTHER`, `RR`, `FIFO`, and `Deadline`, respectively. For the fixed-priority policies, a following number denotes the static priority; e.g., `FIFO 90` is `SCHED_FIFO` at priority 90.

Linux maps these policies to three scheduling classes. `SCHED_OTHER` belongs to the fair class (the Earliest Eligible Virtual Deadline First scheduler, `EEVDF`, on the evaluated kernels [14]) and provides time-shared CPU service without a reservation or response-time bound. `SCHED_FIFO` and `SCHED_RR` belong to the real-time class and use fixed priorities from 1 to 99. `FIFO` runs until it blocks, yields, or is preempted by a higher-priority thread. `RR` additionally time-slices threads at the same priority. `SCHED_DEADLINE` belongs to the deadline class: the user assigns each thread a runtime, deadline, and period, and the kernel schedules it by Earliest Deadline First (EDF) with constant-bandwidth reservations and admission control [9, 12]. These policies only schedule runnable threads. They cannot reserve USB or memory bandwidth or cause a blocked I/O operation to complete. Frame acquisition, however, depends on exactly such non-CPU resources: USB transfers, kernel interrupt service, and memory traffic. CPU scheduling therefore controls only part of the delivery path. A runnable `Deadline` thread with remaining runtime precedes `FIFO` and `RR`, which in turn precede `OTHER`. Promoting an SDK worker (assigning it a real-time policy and priority) without configuring its kernel predecessors can thus delay the receive path.

Kernel preemption is a separate factor. `PREEMPT_RT` [15] shortens many non-preemptible regions and turns most interrupt handlers into schedulable threads. The `threadirqs` boot option also threads eligible interrupt handlers on a kernel without `PREEMPT_RT` (non-RT), making their priorities configurable unlike hard interrupt (IRQ) service. We use Linux 6.12, the first mainline release to support `CONFIG_PREEMPT_RT` [10], as the common kernel base.

### 2.4 Linux USB Receive Path

A D400-series camera is a composite USB Video Class (UVC) device. The Linux `uvcvideo` driver exposes its video interfaces as V4L2 device nodes (`/dev/video*` character devices accessed through the V4L2 API), which the native backend of `librealsense` uses for streaming. Figure 1 separates the hardware and software stages of this receive path.

The xHCI controller is the host-side USB engine: it executes the transfers the kernel schedules, moves camera payloads into main memory by direct memory access (DMA), and reports completion through its event ring. Interrupt service completes USB Request Blocks (URBs), the kernel objects that each describe one requested USB transfer, after which `uvcvideo` assembles UVC payloads into V4L2 buffers [16, 17]. SDK capture workers can run only after these kernel predecessors make a buffer available. On the evaluated kernel, `uvcvideo` queues payload copies to a stream-specific, high-priority unbound

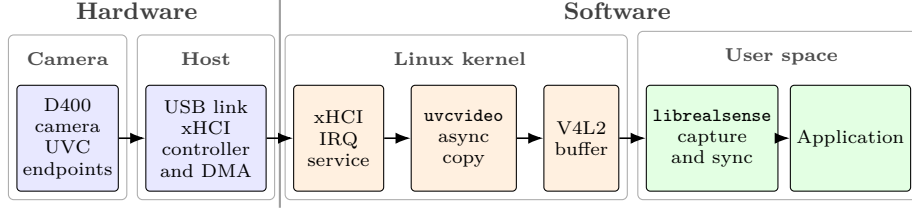


**Fig. 1.** D400 receive path on Linux. USB transfer and DMA occur in hardware. Kernel xHCI IRQ service and asynchronous UVC copy work complete and assemble the data before `librealsense` capture workers and the application can process it.

workqueue. The kernel describes them as asynchronous copy operations, executed by `uvc_video_copy_data_work()` on dynamic unbound kworkers rather than dedicated UVC threads [11]. We refer to this kernel stage as UVC copy work; it follows xHCI IRQ service and precedes V4L2 buffer completion.

V4L2 maintains a finite reusable buffer pool [17]. User space queues empty buffers, waits for completion, dequeues completed buffers, and returns them after processing. A worker can block while the CPU is idle because no buffer has completed; conversely, delayed dequeue or return can exhaust the pool. Scheduler delay and device progress must therefore be measured separately.

Under `PREEMPT_RT` or non-RT with `threadirqs`, xHCI handlers run as IRQ threads such as `irq/N-xhci-hcd` [15], defaulting to FIFO 50 on our platform. The non-RT baseline retains hard-IRQ service, which can preempt user-space work and cannot be configured with `chrt`. In either mode, delayed xHCI service delays URB completion and every later stage. Raising only a user-space SDK priority can improve user-space execution, but cannot control the preceding xHCI service and UVC copy work.

### 3 Related Work

Robotic-perception timing studies commonly begin at a middleware callback or downstream result. SentryRT-1 evaluates real-time Linux for a perception workload driven by five RealSense cameras [38], while RT-BEV studies synchronized multi-camera messages in a real-time bird’s-eye-view pipeline [18]. Robot Operating System (ROS) 2 analyses similarly model callbacks, processing chains, and fusion semantics after sensor data becomes available [3, 39]. Empirical RGB-D camera comparisons instead emphasize depth accuracy and operating range [2, 34, 42]. Because the RealSense ROS wrapper first obtains frames through `librealsense` [33], these studies leave the upstream sensor-to-memory path outside their timing boundary; this path is our focus. Data freshness is itself an established requirement, from temporal validity in real-time databases [8] to information-freshness metrics [44] and automotive data-age constraints [39], but these formulations begin once data has entered the system under study. We

define freshness at the acquisition boundary and attribute its loss to specific kernel and SDK stages.

Real-time I/O research shows why user-space scheduling alone cannot control device delivery. Tuned Pipes replaces the standard xHCI path with a custom driver and framework that provide throughput and delay guarantees [7], time-triggered approaches co-synthesize task and network schedules offline for distributed systems [5], and PREEMPT\_RT work derives scheduling-latency bounds from in-kernel preemption and synchronization behavior [20]. We instead retain the commonly deployed Linux xHCI-UVC/V4L2-`librealsense` path and evaluate its user-space and kernel-side dependencies together.

Self-suspending task theory assumes that execution and suspension structure is known [4], yet libraries and OS components in autonomous systems often behave as delay-prone black boxes [19]. LiME extracts execution models from scheduler traces [1]; we combine it with creator stacks, lifecycle and kernel events, source inspection, and frame metadata. RealSense documentation covers individual APIs, frame management, backends, and multi-camera mechanisms [27–29, 32], but not their joint temporal structure. Our model identifies worker families, separates startup from steady state, and links scheduling and I/O behavior to per-stream freshness.

## 4 `librealsense` Temporal Model and Resource Accounting

The `librealsense` pipeline API exposes three operations: start a camera, wait for framesets, and stop the camera. The implementation creates temporary and persistent threads whose activations follow different sources. Startup serves only two purposes: it reveals which threads persist in recurring frame acquisition and places the measurement boundary after those workers have converged. Our model focuses on recurring frame acquisition, which forms a multi-rate, self-suspending task graph whose output is a join of asynchronous streams. Unlike a single periodic task that releases one dataflow graph per period, the persistent workers form multiple recurring tasks with distinct rates, precedence among some of them, and self-suspending behavior; task-model research has largely studied self-suspension [4] and precedence chains [3] separately.

### 4.1 Worker Functions and Families

Starting a `librealsense` pipeline [30] creates userspace workers for device discovery, control, background services, capture, and delivery. An LD\_PRELOAD-based pthread interposer records each thread’s creator stack, Linux thread identifier (TID), and lifetime; symbolized stacks associate runtime instances with source-level functions. LiME independently records when the instance runs, remains ready, sleeps, wakes, and exits; both records share the kernel BOOTTIME clock and are joined offline by Linux TID. This combination is necessary because several workers share the same Linux thread name. We define a worker

family as a set of equivalent thread instances that share a source-level function and normalized creator-stack signature. Their functional roles and the activation classes of Section 4.2 describe different aspects of these families; recurring families participate in steady-state acquisition.

Functionally, the families play four roles: discovery and control workers enumerate devices and configure sensors; device-service workers maintain timestamps, poll firmware errors, and dispatch notifications; frame-data workers wait on V4L2 buffers, invoke the sensor callbacks (on which synchronization and aggregation execute inline), and return buffers to the kernel; and application workers comprise one frameset-wait thread per camera and the benchmark’s main control thread. The wait thread calls `wait_for_frames()`, which blocks until a frameset is available or the configured timeout expires.

The complete lifecycle trace contains one main thread and 21 additional thread instances, including temporary helpers that exit before recurring frame acquisition. The artifact retains this full startup record. The recurring model begins after initial delivery, persistent-worker matching, policy installation, and warm-up. Thus, one-shot discovery and configuration are excluded from all recurring execution, period, and utilization parameters below.

## 4.2 Steady-State Activation Classes

The persistent worker set is stable during measurement, but its members do not share one period. We classify these workers by their observed activation pattern. *Frame-triggered workers* include V4L2 capture and application wait threads. Sensor callbacks and synchronization execute inline on these paths. Their typical activation interval is close to  $T_f = 1/f$ , with release jitter from USB completion, buffering, clock drift, and scheduling. *Timer-triggered workers*, such as timestamp maintenance and firmware checks, use slower rates independent of the camera FPS. *Event-triggered workers* wake for a queue or device event and have no fixed period. *Dormant workers* remain alive without activating during the measurement and thus add no recurring CPU utilization.

Figure 2 instantiates these classes for one D435 or D455 using V4L2 with Depth, Color, IR1, and IR2. It shows creation and ownership, but not frame flow. Ten threads are common to the measured D435 stream ablation (runs that enable stream subsets one at a time to attribute workers to streams): the main and application-wait threads, udev watcher, pipeline dispatcher, two device services, and four notification dispatchers. Five of these services normally block waiting for control or notification events, while the udev watcher, timestamp-difference keeper, and firmware-error poller use timers.

The remaining three threads depend on enabled video nodes. Depth adds a Z16 capture worker and Color adds a color-capture worker. IR1 and IR2 together add one interleaved-infrared worker because both evaluated cameras deliver a Y8I frame containing both infrared samples. The synchronous `y8i_to_y8y8` block separates that input into logical IR1 and IR2 frames without creating another capture pthread. The measured D435 configurations therefore contain

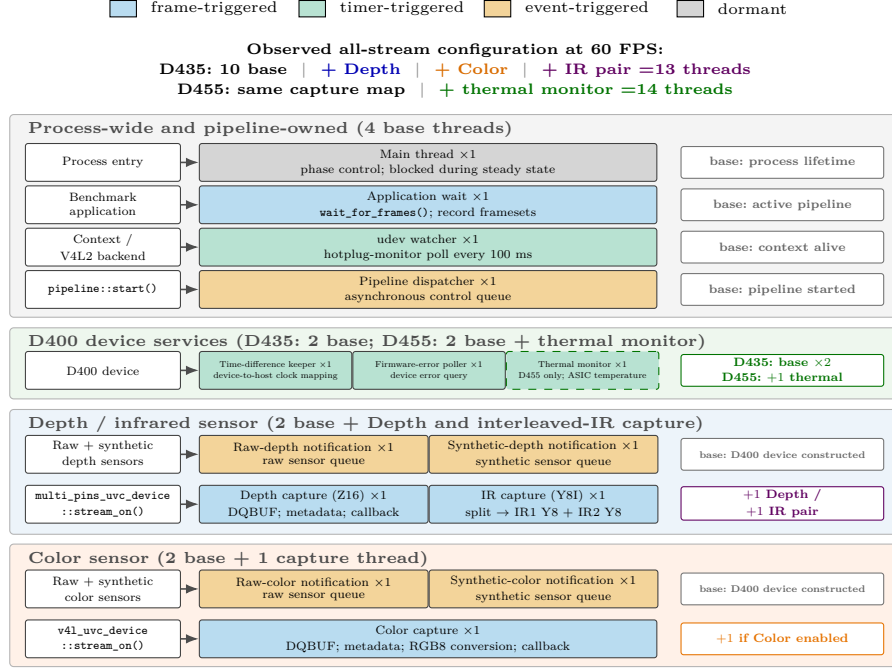


**Fig. 2.** Creation and activation-class map for one D435 or D455 with Depth, Color, IR1, and IR2 through V4L2. Fill color gives the activation class; solid arrows denote creation or start ownership. Both models use the same three capture workers. The dashed box is the additional timer-triggered thermal monitor on D455, giving 14 rather than 13 live threads.

11, 12, and 13 live threads as Depth, Color, and the interleaved infrared input are successively enabled. D455 retains these capture-worker increments and the synchronous Y8I split. The evaluated D455 additionally creates one timer-triggered `ds_thermal_monitor` worker to track the temperature of the camera’s application-specific integrated circuit (ASIC) and thermal compensation. The all-stream configuration therefore contains 14 live threads on D455, compared with 13 on D435. Temporary discovery instances are excluded from both counts. The recurring dependency graph is  $G_P = (V_P, E_P)$ : the vertices  $V_P$  are the recurring worker families together with the kernel stages for xHCI IRQ service and UVC copy work, and a directed edge  $(i, j) \in E_P$  means that stage  $i$  provides data or an activation to stage  $j$ . Under a fixed workload, each userspace vertex of  $G_P$  is a recurring worker family that we model as a real-time task  $\tau_i$ ; the two kernel stages are vertices but not tasks, and the evaluation treats them separately:

$$\tau_i = (C_i, A_i, T_i/\Delta_i, D_i), \quad C_i = \sum_{h=1}^{q_i} C_{i,h}, \quad (1)$$

where  $A_i$  is the activation source,  $D_i$  the relative deadline, and  $C_i$  the CPU time of a logical activation containing  $q_i$  CPU bursts of durations  $C_{i,1}, \dots, C_{i,q_i}$ . Blocking between bursts contributes to response time but not to  $C_i$ . The third component is the measured period  $T_i$  for a family with a stable recurring pattern, or the measured minimum inter-arrival time  $\Delta_i$  for an event-triggered family when enough events are observed; it is absent for dormant workers. Equivalent instances share one task type. The accounting below records both instance multiplicity and per-activation execution cost.

### 4.3 Workload-Level Resource Accounting

Timely frame delivery requires sufficient CPU service and USB transfer capacity. Host-memory activity can further affect worker execution time.

For a measured workload  $W$  (a fixed selection of enabled streams, resolutions, formats, and frame rates per camera), let  $\mathcal{F}(W)$  be its userspace worker families that activate during the measurement,  $n_i(W)$  the observed number of family- $i$  instances,  $\overline{C}_i(W)$  their mean execution per activation, and  $\overline{T}_i(W)$  their mean activation interval. LiME provides the observed frame-acquisition userspace utilization

$$U_{\text{user}}^{\text{obs}}(W) = \sum_{i \in \mathcal{F}(W)} n_i(W) \frac{\overline{C}_i(W)}{\overline{T}_i(W)}. \quad (2)$$

This accounting includes one application wait worker per camera. Dormant workers contribute zero. It excludes the blocked main control thread and kernel-side xHCI IRQ service and UVC copy work. It describes the measured workload and is neither a capacity prediction nor a schedulability bound. The bursty kernel-side work is evaluated separately in Section 5.

For camera  $c$  and each of its enabled streams  $s \in \mathcal{S}_c$  (the set of streams the workload requires from camera  $c$ ), let  $f_{c,s}$  be its frame rate,  $w_{c,s}$  and  $h_{c,s}$  its width and height, and  $b_{c,s}^{\text{wire}}$  its wire-format bytes per pixel. Its nominal USB payload rate for image data, excluding protocol overhead, is  $R_{c,s} = f_{c,s} w_{c,s} h_{c,s} b_{c,s}^{\text{wire}}$ , giving

$$B_{\text{USB}} = \sum_{s \in \mathcal{S}_c} R_{c,s}. \quad (3)$$

To place memory pressure on the same scale, we also report an analytical byte-touch estimate: three touches per wire byte for DMA write, V4L2-buffer read, and SDK-frame write, plus the input read and output write of any format conversion. This estimate describes the identified code path; it is not measured main-memory (DRAM) traffic or a bandwidth bound. For the evaluated formats, RGB8 output is converted from two-byte YUYV to three-byte RGB8, Depth Z16 uses two wire bytes per pixel, and each Y8 infrared stream uses one. Section 5 reports one observed utilization, USB-payload, and byte-touch total for each workload. Resource demand does not scale linearly with the camera count, because process-wide services are shared and concurrent capture can lengthen per-activation execution; characterizing multi-camera scaling is future work.

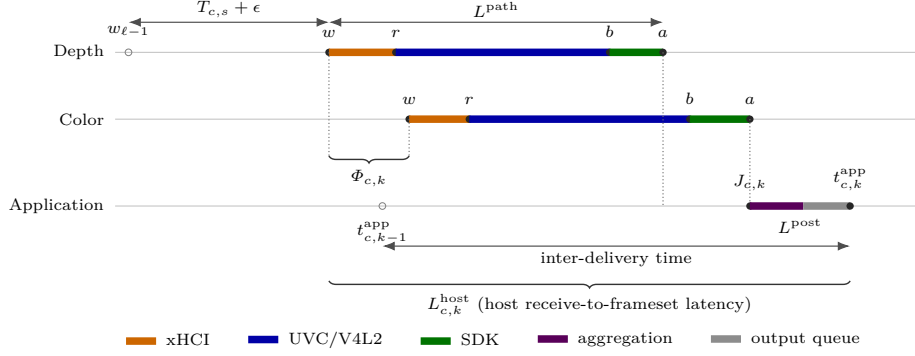


**Fig. 3.** Timing notions for one frameset with Depth and Color. Each input proceeds from xHCI activation  $w$  through diagnostic backend and buffer points  $r$  and  $b$  to synchronizer arrival  $a$ . The last required arrival closes the join  $J_{c,k}$  before `wait_for_frames()` returns at  $t_{c,k}^{\text{app}}$ . The model combines the colored receive-path stages into  $L^{\text{path}}$ . Successive returns define inter-delivery time.

#### 4.4 Frameset Precedence and Freshness

The host cannot schedule the camera’s exposure instant on the host clock, and it cannot schedule device-side transmission before a USB interrupt reaches the host. We therefore begin at the earliest observed xHCI service activation that processes the first relevant URB of a stream input. This boundary is the IRQ thread’s `sched_wakeup`, or `irq_handler_entry` when xHCI uses a hard IRQ. Let  $w_{c,s,\ell}$  denote this time for input  $\ell$  of stream  $s$  on camera  $c$ , and let  $a_{c,s,\ell}$  denote its arrival at the SDK synchronizer. For frameset  $k$ ,  $m_{c,s,k}$  is the selected input of each required stream  $s \in \mathcal{S}_c$ . The compact timing model is

$$\begin{aligned} w_{c,s,\ell} &= \ell T_{c,s} + \phi_{c,s} + \epsilon_{c,s,\ell}, \quad a_{c,s,\ell} = w_{c,s,\ell} + L_{c,s,\ell}^{\text{path}}, \\ J_{c,k} &= \max_{s \in \mathcal{S}_c} a_{c,s,m_{c,s,k}}, \quad t_{c,k}^{\text{app}} = J_{c,k} + L_{c,k}^{\text{post}}. \end{aligned} \quad (4)$$

Here,  $T_{c,s}$  is the nominal stream period,  $\phi_{c,s}$  its phase, and  $\epsilon_{c,s,\ell}$  release jitter at the host boundary.  $L^{\text{path}}$  combines xHCI service, UVC/V4L2 assembly, and SDK work up to the synchronizer. The join  $J_{c,k}$  waits for the last required input, and  $L^{\text{post}}$  covers remaining aggregation and output-queue delay before `wait_for_frames()` returns at  $t_{c,k}^{\text{app}}$ . The separation between the selected streams’  $w$  values is the cross-stream phase spread  $\Phi_{c,k}$  shown in the figure. Figure 3 retains the intermediate trace points used for diagnosis, although the model accounts for them together as  $L^{\text{path}}$ .

For the selected inputs, host receive-to-frameset latency starts at the earliest relevant xHCI activation and ends at the application return:

$$L_{c,k}^{\text{host}} = t_{c,k}^{\text{app}} - \min_{s \in \mathcal{S}_c} w_{c,s,m_{c,s,k}}. \quad (5)$$

This quantity is not physical sensor-data age because it excludes exposure and device/link activity before xHCI service activation. The paper reports it as an observed host-path latency, not a worst-case bound.

Delivery alone does not establish freshness. For the frame number  $n_{c,s,k}$  selected from stream  $s$ , we define advancement  $u$  and gap count  $g$  as

$$u_{c,s,k} = \mathbf{1}[n_{c,s,k} > n_{c,s,k-1}], \quad g_{c,s,k} = \max(0, n_{c,s,k} - n_{c,s,k-1} - 1). \quad (6)$$

Here,  $\mathbf{1}[\cdot]$  is one when its condition holds and zero otherwise.  $u_{c,s,k}$  is therefore the per-stream advancement indicator. It equals one when the selected frame number increases and zero when the previous stream frame is reused.  $g_{c,s,k}$  counts the intervening sequence numbers skipped before a new frame is delivered. For camera  $c$  at delivery  $k$ , a frameset is fresh if  $u_{c,s,k} = 1$  and  $g_{c,s,k} = 0$  for every  $s \in \mathcal{S}_c$ . It is partially stale if there are  $s_0, s_1 \in \mathcal{S}_c$  such that  $u_{c,s_0,k} = 0$  and  $u_{c,s_1,k} = 1$ , and fully stale if  $u_{c,s,k} = 0$  for every  $s \in \mathcal{S}_c$ . A stream has a sequence gap if  $u_{c,s,k} = 1$  and  $g_{c,s,k} > 0$ , and a frameset has a sequence gap if any selected stream does. Accordingly, a frameset is fresh only when no required stream is reused and none has a sequence gap; every other outcome is non-fresh. Together with delivery timestamps and timeout records, these conditions distinguish late delivery, stale data, sequence gaps, and API timeout. The benchmark application records application delivery and every stream frame number, but does not separately observe internal join and output-queue delays.

If an incomplete UVC/V4L2 input prevents a new depth frame from reaching the synchronizer, another stream can advance while the previous depth frame is reused. The successful API return is then partially stale, and the next valid depth input can reveal a sequence gap.

## 5 Experimental Evaluation

To answer the research questions, we conduct a multi-stage experimental study organized as four experiment populations, E1 to E4, each presented with its design, its results, and the research question it addresses. E1 reconstructs the recurring workers and instantiates the model’s execution, activation, and resource parameters (RQ1). E2 and E3 examine receive-path dependencies and identify UVC and kernel scheduling controls (RQ2). Fixing these controls, E4 uses the measured worker parameters to compare userspace scheduling policies and kernel configurations under interference, using the model’s latency and freshness metrics (RQ3). We first describe the platform and protocol shared by all four.

### 5.1 Common Platform, Workloads, and Protocol

All experiments use Raspberry Pi 5 as a resource-constrained embedded host, with fixed platform controls to isolate kernel, scheduling, and interference effects across matched experiments.

Motivated by `PREEMPT_RT`’s reported effects on Pi 5 scheduling latency [6], we evaluate its effect on frame delivery using matched Linux 6.12.96 builds: the RT

build enables `PREEMPT_RT`, whereas the non-RT build uses `CONFIG_PREEMPT`. All experiments use the same optimized and instrumentable build of `librealsense` 2.58.3. Both the benchmark and SDK are built with GCC/G++ 13.3.0 in `RelWithDebInfo` mode (`-O2 -g -DNDEBUG`). The SDK additionally uses `-mstrict-align -ftree-vectorize -pthread -fPIC` on AArch64. Frame acquisition uses the native V4L2 backend and `uvcvideo`, with 16 UVC URBs per streaming interface except in the receive-pool comparison. Worker discovery uses one D435, single-camera calibration covers D435 and D455, and the main E4 comparison uses two D435 cameras attached to different xHCI controllers.

Two workload profiles cover a broader range of RealSense use cases. The *representative workload* uses  $848 \times 480$  Z16 Depth and  $640 \times 480$  RGB8 Color at 30 FPS. The *stress workload* increases stream count and throughput with Depth, Color, and two infrared streams (IR1 and IR2) at 60 FPS, using the highest validated resolution for each camera model and stream at that rate.

Unless stated otherwise, each experimental cell contains three independent 30-s runs. Every run starts the cameras under `OTHER`, completes warm-up, installs the requested scheduling and interference treatments, and begins measurement only after all components report ready. We use `benchkit` [23] to expand the experimental factors and record their artifacts. The supplementary document [37] describes the fixed platform controls, the build and tracing configuration, the stream profiles, and the reset, retry, and acceptance procedures.

*Metrics and Statistics.* We report p99 host receive-to-frameset latency (Eq. 5) as the tail statistic. A 30-s run yields about 900 framesets per camera at 30 FPS and 1,800 at 60 FPS, so a per-run p99 rests on 9 to 18 deliveries, whereas p99.9 and the maximum rest on one or two; we therefore use p99 per run, remove no samples, and report the per-cell maxima in the supplementary document (Section D). Thirty seconds covers at least six activations of the slowest 5-s timer family, and three runs separate run-to-run variation from treatment effects: bars show the mean of the per-run p99 values and dots show each run, so the worst run remains visible. Reused streams or sequence gaps make a frameset non-fresh (Section 4.4); API timeouts additionally characterize delivery failures. All reported values are measurements, not worst-case bounds.

Table 1 summarizes the four experiment populations built on this common protocol and their sample sizes.

## 5.2 RQ1: What Is the Steady-State Structure of Frame Acquisition?

*Design.* E1 identifies acquisition threads, their persistence after startup, and their steady-state activations. Shared thread names and coexisting temporary helpers require source inspection and an `LD_PRELOAD` pthread interposer recording lifecycles and creator stacks. Module-relative stack signatures identify families; instance numbers distinguish equivalent workers across runs and cameras. LiME supplies execution, ready, sleep, response, and activation intervals [1]. Application trace markers delimit initial delivery, policy installation, measurement, and teardown. Twenty D435 lifecycle traces test repeatability across launches.

**Table 1.** Principal experiment populations on Raspberry Pi 5 and Linux 6.12.96. RT denotes `PREEMPT_RT`.

| ID | Evidence                       | Kernel/cameras                                  | Design and sample size                                                                                                                |
|----|--------------------------------|-------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| E1 | Worker and path calibration    | RT; 1 D435, 1 D455                              | 2 camera models $\times$ 2 calibration workloads $\times$ 3 full-path traces = 12 runs; 30 s each.                                    |
| E2 | UVC diagnosis and receive pool | non-RT; 1 D435                                  | 2 aligned failure traces; 2 workloads $\times$ 2 pool sizes $\times$ 3 repetitions = 12 runs; 30 s each.                              |
| E3 | Kernel-predecessor scheduling  | non-RT <code>threadirqs</code> ; 1 D435, 1 D455 | 2 models $\times$ 3 xHCI/UVC copy treatments $\times$ 3 repetitions = 18 full-path stress runs; 30 s each.                            |
| E4 | Main scheduling comparison     | non-RT, RT; 2 D435                              | 2 workloads $\times$ 4 interference conditions $\times$ 4 treatments $\times$ 2 kernels $\times$ 3 repetitions = 192 runs; 30 s each. |

Calibration collects three 30-s full-path traces per camera-model-workload pair using RT `URB16`, userspace `OTHER`, and xHCI FIFO 90 without interference. These traces instantiate Section 4’s temporal and resource quantities and expose D455’s additional thermal-monitor family. The supplementary document details tracing and calibration controls.

*Results.* All twenty lifecycle traces reproduce the same startup topology: one initial thread and 21 additional instances, including `libusb` event-loop helpers for device discovery and pipeline construction that exit before steady state. Only persistent workers (Section 4.2) enter the steady-state model. The application starts under `OTHER` and installs policies after warm-up, targeting persistent workers by role without configuring temporary helpers.

E1 identifies a repeatable worker structure: all three runs in each camera-workload cell contain the same worker families and instance counts. For every periodic family, the three per-run period estimates differ from their median by at most 3.274%. All calibration runs deliver only fresh framesets and complete without a timeout. These results support family-specific parameters instead of assigning every worker the camera period.

The dispatcher row requires a different interpretation from the periodic families. No queued event occurs for these dispatchers in E1; they wake only on a 5-s timeout. Table 2 therefore does not estimate their event-driven intervals.

The table directly exposes the multi-rate structure. Frame-driven families activate at observed minimum intervals of 30.2–33.2 ms at 30 FPS and 13.6–16.3 ms at 60 FPS, while timer-driven process and device services recur near 100 ms, 1 s, 2 s, or 5 s independently of camera FPS. These families also differ substantially in CPU demand. The Depth/IR and Color capture workers contribute nearly all measured steady-state userspace CPU utilization, whereas timer- and event-triggered OS services contribute little. By enabling IR1 and IR2, the stress workload adds one interleaved-infrared capture instance to the Depth/IR family and exhibits higher capture execution times. For D435, summing the per-camera family measurements from the underlying traces gives 5.0% userspace CPU utilization under the representative workload and 22.8% under the stress workload; the corresponding D455 values are 5.0% and 20.7%. Equation 3 gives a nominal

**Table 2.** E1 instantiation of Eqs. 1 and 2 for the Representative (R) and Stress (S) workloads.  $\hat{C}$ : largest logical-activation execution time;  $\hat{T}/\hat{\Delta}$ : smallest stable period or activation interval;  $U^{\text{obs}}$ : mean per-run utilization of all equivalent instances (% of one core), from mean cost and interval, hence not the ratio of these extrema (implied mean cost: one third to two thirds of  $\hat{C}$  for frame-triggered families). Instances per camera  $n$  (R/S): Depth/IR capture 1/2, dispatchers 5/5, others 1. Ranges span D435 and D455 (dispatchers: all five); \*D455 only. Full precision: supplementary Section E.

| Worker family                             | $\hat{C}$ ( $\mu\text{s}$ ) |           | $\hat{T}/\hat{\Delta}$ (ms) |                | $U^{\text{obs}}$ (%) |             |
|-------------------------------------------|-----------------------------|-----------|-----------------------------|----------------|----------------------|-------------|
|                                           | R                           | S         | R                           | S              | R                    | S           |
| <i>Frame-triggered</i>                    |                             |           |                             |                |                      |             |
| Depth/IR capture                          | 827–845                     | 1894–1953 | 32.8–33.2                   | 14.4–15.4      | 1.11–1.12            | 10.87–11.13 |
| Color capture                             | 1708–1730                   | 2166–2702 | 32.9–33.2                   | 16.2–16.3      | 3.61–3.66            | 9.15–11.48  |
| Application wait                          | 160–171                     | 160–182   | 30.2–30.9                   | 13.6–13.9      | 0.173–0.181          | 0.368–0.402 |
| <i>Timer-triggered</i>                    |                             |           |                             |                |                      |             |
| udev watcher                              | 75–95                       | 140–162   | 100                         | 100            | 0.028–0.029          | 0.040–0.041 |
| Time-diff. keeper                         | 368–410                     | 377–399   | 1001                        | 1001           | 0.016–0.021          | 0.021–0.023 |
| Thermal monitor*                          | 173                         | 245       | 2001                        | 2001           | 0.0052               | 0.0061      |
| Firmware-error poller                     | 45–56                       | 63–107    | 5000                        | 5000           | <0.001               | <0.001      |
| <i>Event-triggered (5-s idle timeout)</i> |                             |           |                             |                |                      |             |
| Dispatchers                               | 46–69                       | 55–118    | $\approx 5000$              | $\approx 5000$ | <0.001               | <0.001      |

USB payload of 40.9 MiB/s for either camera under the representative workload. Under the stress workload, it gives 152.5 MiB/s for D435 and 139.7 MiB/s for D455. The analytical memory-touch estimate is 167 MiB/s for either camera under the representative workload, increasing under the stress workload to 606 MiB/s for D435 and 536 MiB/s for D455.

*Answer to RQ1.* Frame acquisition is not a single periodic task: it comprises stable, model-dependent worker families with different activation patterns and execution costs. Frame-driven capture dominates measured userspace CPU utilization, while timer-driven services contribute little. Each active family needs separate characterization, and event-driven parameters require workloads that exercise the corresponding events.

### 5.3 RQ2: Which Receive-Path Controls Are Required?

*Design.* Steady-state RealSense delivery depends on SDK-worker scheduling, PREEMPT\_RT, and Linux USB receive-path services (Section 2.4). The UVC URB pool bounds the number of in-flight receive requests, while xHCI interrupt service completes camera transfers and asynchronous UVC copy work assembles their payloads before V4L2 exposes them. When xHCI service is threaded, both stages run in schedulable task contexts whose default policies and priorities may not

protect them from competing CPU work. RQ2 therefore examines both receive capacity and kernel-path scheduling.

The first mechanism study (E2) compares two `uvccvideo` builds configured with 5 and 16 URBs per streaming interface. This pool size is set at compile time by the `UVC_URBS` macro in the Linux kernel’s `uvccvideo` driver [25]. We denote the corresponding kernel builds as `URB5` and `URB16`, respectively. These are the two shipped configurations: the kernel default and the value that the official `librealsense` kernel-patching scripts apply for Ubuntu LTS kernels; intermediate pool depths are not evaluated. One D435 runs both workloads. Before the comparison, one 30-s `SCHED_OTHER` calibration per workload measures stable worker activation intervals and defines a `SCHED_FIFO` SDK profile with rate-monotonic priorities (FIFO-RM), reused across E2 and the E3 starvation experiment below. This deterministic ordering gives higher fixed priorities to the more frequently activated families of RQ1’s multi-rate structure; it does not imply classical RM schedulability for the self-suspending graph. The shortest-interval family receives priority 80, with successively lower priorities for longer intervals and equal priorities for equal intervals. Four `SCHED_OTHER` register-only busy loops provide CPU competition. On the non-RT kernel, xHCI remains a hard IRQ, while asynchronous UVC copy kworkers retain `SCHED_OTHER`. This fixed load exposes receive-path sensitivity under contention. Each pool-workload cell has three 30-s runs; pool size is not optimized.

E3 progressively protects xHCI service and then UVC copy execution. It uses deliberate localized starvation to distinguish their effects within 30-s traces, where non-fresh framesets would otherwise be too rare. The experiment uses one D435 or D455, the model-specific 60-FPS stress workload, the non-RT `URB16` kernel with `threadirqs`, and FIFO-RM SDK workers. CPU affinity pins the selected xHCI IRQ thread and a `SCHED_RR` priority-1 interferer to the same CPU core. The dynamic kworkers executing UVC copy work are not explicitly pinned because the kernel workqueue subsystem manages their CPU affinity; E3 changes only the scheduling policy of kworkers identified while executing UVC copy work. Thus, the interferer directly targets xHCI service but is not guaranteed to share a CPU with every UVC copy activation. The interferer runs for 4.5 ms and sleeps for 0.5 ms, giving a 90% duty cycle; although priority 1 is the lowest real-time priority, it outranks `SCHED_OTHER` tasks on that core while remaining below the FIFO priorities used to protect the kernel path. One register-only `OTHER` worker supplies background competition. Three treatments progressively protect the receive path: the xHCI thread and copy-executing kworkers both under `OTHER`; xHCI at FIFO 90 while those kworkers remain under `OTHER`; and xHCI at FIFO 90 with observed copy-executing kworkers at FIFO 89. This priority ordering preserves xHCI precedence over payload copy. Each model-treatment cell has three 30-s full-path traces and an effective-policy audit.

*Results.* Figure 4a summarizes E2 using the freshness and sequence-gap definitions in Section 4.4. Across the three `URB5` runs, the 30-FPS workload produces two partially stale framesets and two deliveries with a sequence gap; at 60 FPS, these counts rise to ten and thirteen. With `URB16`, neither outcome occurs in

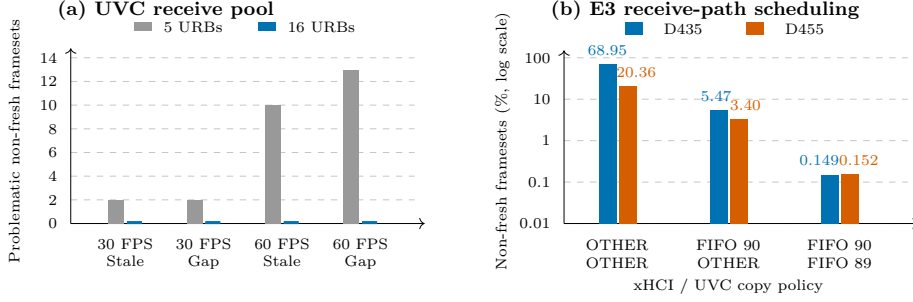


**Fig. 4.** Host-path controls with frame-acquisition workers under FIFO-RM. (a) E2 counts of partially stale framesets and deliveries containing a stream sequence gap for one D435 with four OTHER CPU competitors. Each return is counted once in its category. (b) E3 percentage of non-fresh framesets—partially or fully stale, affected by a sequence gap, or both—under CPU interference on the core hosting the xHCI IRQ thread, with progressive scheduling protection for xHCI and UVC copy work; each treatment label gives the xHCI IRQ policy above the UVC-copy kworker policy. Each bar pools three 30-s non-RT `threadirqs` stress runs.

either workload. Two aligned URB5 traces explain how an observed failure propagates to the application. In both, the first host-visible anomaly occurs at the UVC/V4L2 boundary: `uvcvideo` closes a Depth buffer before its expected payload is complete, although neither the UVC header nor the xHCI completion reports an error. V4L2 rejects the short buffer, so no acceptable new Depth input reaches the frameset join; the application subsequently receives the previous Depth frame and then observes a sequence gap. The traces rule out duplicate logging in the benchmark application, loss after a successful V4L2 delivery, V4L2 queue starvation, autosuspend, and a reported xHCI transfer error. Because they do not observe the path upstream of UVC completion, they cannot distinguish camera, physical-link, and unreported lower-host-path loss.

Under the  $u/g$  definitions in Section 4.4, stale and gap counts usually increase together in a reuse-and-catch-up sequence. If Depth is reused at delivery  $k$ ,  $u_{D,k} = 0$  makes the frameset partially stale; when Depth catches up at  $k + 1$ ,  $g_{D,k+1} = 1$  records a gap in the following delivery. By contrast, an all-stream skip has  $u_{s,k} = 1$  and  $g_{s,k} = 1$  for every required stream. Under this paper’s definition, a frameset with a sequence gap is non-fresh even when no stream is reused; the gap count can therefore increase without a corresponding stale-frameset count. The supplementary document tabulates the three observed patterns with their occurrence counts. The deeper pool keeps more receive requests in flight without increasing USB bandwidth or guaranteeing loss-free acquisition. Since URB16 eliminates all observed stale framesets and stream sequence gaps in this matched comparison, all subsequent experiments use `UVC_URBS=16` as an empirically supported choice whose minimality and optimality remain unproven.

Figure 4b shows the main E3 result: under deliberately severe CPU interference, coordinating the threaded xHCI IRQ and the kworkers executing UVC

copy work reduces the non-fresh rate by 99.25–99.78% across the two camera models relative to leaving both stages under OTHER. Protecting xHCI alone removes most failures, but the rate falls to about 0.15% only after UVC copy execution is also protected. Aligned traces show that the fully coordinated treatment also reduces host receive-to-frameset p99 by 85.2–86.5%. Both schedulable kernel stages must therefore be included in the receive-path scheduling policy.

The raw traces expose another constraint on policy selection. Each camera frame spans multiple URBs from concurrent stream endpoints, so xHCI activations follow USB transfer completions rather than one periodic event per frame; their IRQ service is consequently sporadic and bursty. In the busiest observed 1-ms interval, an xHCI IRQ thread executes for up to 404  $\mu$ s. A Deadline reservation sized to cover this burst would therefore reserve about 40% of one CPU for a thread whose average utilization is only 3–8%. We therefore do not recommend SCHED\_DEADLINE for the threaded xHCI IRQ under this reservation method.

*Answer to RQ2.* Fresh delivery depends on receive capacity and on schedulable kernel predecessors, not only on SDK workers. In E2, URB16 eliminates all observed stale framesets and sequence gaps. E3 shows that threaded xHCI service and UVC copy execution must both be protected: xHCI protection alone leaves residual failures, while protecting both reduces host receive-to-frameset p99 by 85.2–86.5%. Receive capacity and both kernel stages must thus be configured with SDK workers.

#### 5.4 RQ3: How Much Does Coordinated Scheduling Improve Delivery?

*Design.* Building on E1–E3, E4 compares scheduling policies for SDK workers coordinated with the receive-path controls identified in RQ2. It tests whether these policies limit interference-induced increases in host receive-to-frameset latency while preserving frameset freshness.

E4 evaluates four interference conditions: no injected load (*None*); four unpinned register-only busy-loop workers (*CPU*); four private-buffer copy workers requesting 2,000 MiB/s in aggregate (*Memory*); and repeated MobileNetV2 inference through Vulkan on the V3D graphics processing unit (GPU), including shared-memory traffic and CPU submission [13, 24] (*GPU*). Interference starts only after both cameras have warmed up and every requested worker reports ready. As in all experiments except E3, no CPU affinity is set, so camera threads and interferers share the four cores: the acquisition stack alone uses at most 22.8% of one core (E1), whereas the CPU condition occupies all four.

Following the receive-path findings in Section 5.3, E4 uses V4L2 with URB16. Its coordinated treatments manage the scheduling of SDK workers, threaded xHCI service, and kworkers observed executing UVC copy work.

The OTHER treatment leaves recurring SDK workers and kworkers observed executing UVC copy work under OTHER. The non-RT baseline retains hard-IRQ xHCI service, while the RT baseline retains its native threaded-IRQ policy.

The RR-RM and FIFO-RM treatments apply RR and FIFO, respectively, using the rate-monotonic priority mapping defined for E2 (Section 5.3).

Deadline profiles are frozen before E4 from three independent OTHER calibration traces per workload. For each family, the runtime is 1.20 times the largest measured execution time, and the period and deadline are 0.91 times the smallest measured stable interval. These empirical margins are intended to reduce runtime exhaustion and deadline misses; they are neither worst-case execution time (WCET) nor minimum inter-arrival bounds. On the receive path, RR-RM assigns RR 90 to xHCI and RR 89 to active copy-executing kworkers; FIFO-RM assigns FIFO 90 to xHCI and FIFO 89 to those kworkers. Consistent with Section 5.3, the Deadline treatment applies Deadline only to the userspace worker families; xHCI uses FIFO 90 and the observed copy-executing kworkers use FIFO 89. A controller implemented with eBPF (extended Berkeley Packet Filter) observes UVC copy execution and applies the declared policy in the coordinated treatments.

The factorial combination of two workloads, four interference conditions, four scheduling treatments, two kernels, and three repetitions forms 192 independent 30-s E4 runs. Each records the kernel events required to measure host receive-to-frameset latency in Eq. 5 together with the frame numbers required for freshness analysis. This latency-only kernel trace is a strict subset of the full receive-path kernel event set used for model construction and diagnosis. The supplementary document distinguishes this kernel trace from the userspace markers and reports the measured marker overhead.

*Results.* Figure 5 reports  $L^{\text{host}}$  (Eq. 5) and the non-fresh counts derived from  $u$  and  $g$  (Eq. 6) for every E4 cell. It shows that RR-RM, FIFO-RM, and Deadline can all limit interference-induced growth in host receive-to-frameset latency. The clearest benefit appears under direct CPU contention: at 60 FPS, coordinated scheduling reduces p99 by 24.1–28.6% on non-RT and 35.6–36.0% on PREEMPT\_RT. The effect is smaller at 30 FPS, but it also extends to indirect memory and GPU interference; the best 60-FPS reductions reach 17.2% and 11.6%, respectively.

PREEMPT\_RT does not consistently reduce host receive-to-frameset p99 relative to non-RT; its clearest benefit is freshness. The non-RT half of the matrix contains 159 non-fresh framesets, whereas the PREEMPT\_RT half contains only one, contributed by OTHER under 60-FPS CPU interference. In that case, all selected streams advance but four sequence numbers are skipped, so the frameset is non-fresh because of the gap, not because a stream was reused. All coordinated PREEMPT\_RT treatments remain fresh. Prior work reports lower kernel scheduling latency with PREEMPT\_RT on Raspberry Pi 5 [6]; our results show that, for this camera path, the more pronounced improvement is in the proportion of fresh framesets rather than host-latency p99.

The non-RT 60-FPS result also shows why latency and freshness must be reported together. RR-RM, FIFO-RM, and Deadline each produce non-fresh framesets in at least one interference condition, and every treatment has a p99 host latency above the 16.67-ms nominal frame period. Exceeding one period

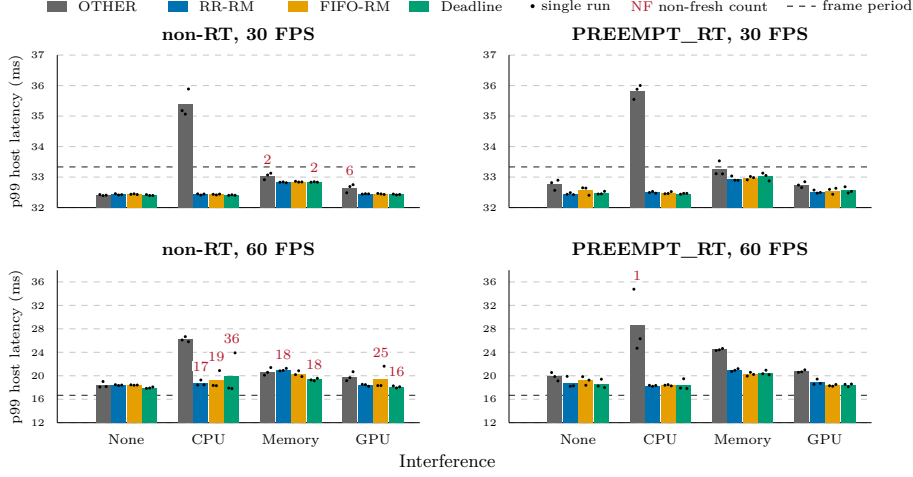


**Fig. 5.** E4 p99 host receive-to-frameset latency for two D435 cameras at 30 and 60 FPS. Bars are means of three per-run p99 values, dots are individual runs, dashed lines mark nominal frame periods, and red labels give the non-fresh frameset count (NF): deliveries with at least one reused component or sequence gap. Each non-fresh frameset is counted once, and a bar without a label contains only fresh framesets. Latencies at 60 FPS exceed those at 30 FPS because the doubled frame rate doubles the USB payload and the capture work competing for the same cores.

does not by itself imply a dropped frame, but it means that selected data can remain in the host receive-to-frameset path for longer than one camera period. No API timeout occurs in E4. Note that the observations are not formal worst-case bounds.

This contrast also reconciles E3 with E4. E3 compares configurations that all expose xHCI as a schedulable thread, and shows that such a path must then be prioritized. E4’s non-RT baseline instead keeps hard-IRQ xHCI service, which user-space work cannot preempt: at 60 FPS it produces no non-fresh frameset, whereas each coordinated non-RT treatment produces some. On the non-RT kernel, threading the receive path therefore trades freshness robustness for tail control, and only **PREEMPT\_RT** delivers both. E4 records the events required for latency and freshness, not the per-stage traces that would attribute these residual failures to a specific stage.

*Answer to RQ3.* Coordinated scheduling is most effective against direct CPU contention but also reduces latency under memory and GPU interference. For applications that use **librealsense** to acquire sensor data for real-time tasks, we recommend **PREEMPT\_RT** together with a coordinated real-time policy, such as RR-RM or FIFO-RM, or Deadline configured with measured margins. Both host latency and per-stream freshness must be checked to validate the configuration.

## 6 Discussion and Limitations

**Cross-Layer Deployment.** Frame acquisition cannot be protected by scheduling SDK workers alone, because xHCI service and UVC copy work precede them. Our deployment recipe therefore combines `PREEMPT_RT`, a larger UVC receive pool (e.g., `URB16` on our platform), and coordinated scheduling of recurring SDK workers, threaded xHCI service, and the kworkers observed executing UVC copy work. The bursty xHCI thread uses FIFO even when SDK workers use Deadline.

**Timing and Freshness.** Neither regular API returns nor shorter latency tails establish freshness. Conversely, host latency can exceed one nominal camera period without an observed frame loss. A configuration must therefore be validated on both host latency and per-stream reuse and sequence gaps.

**Scope and Generalization.** The findings cover D435 and D455 on Raspberry Pi 5. Device SDKs such as `librealsense` are not necessarily written with real-time constraints in mind; their synchronization, queueing, and threading choices directly shape the timing and freshness observed here. Worker parameters require recalibration when the camera model, stream profile, or software build changes; `URB16` is validated here, not a universal optimum. Timing values remain empirical rather than formal worst-case guarantees; applicability to other stacks is discussed in Section 7.

**Artifact Availability.** The benchmark application, the setup and campaign scripts, and the analysis and plotting code are publicly available [36].

## 7 Applicability and Generalization

**Transferable Method.** The same analysis steps are transferable to other camera stacks, while the timing parameters and configuration settings must be determined for each platform. First, source code and thread-creation call stacks identify the function of each worker. Tracing tools then characterize worker activation patterns and CPU execution times. Combining traces, frame metadata, and kernel events reveals receive-path dependencies and where delivery problems first become visible. These observations guide coordinated scheduling of SDK workers and kernel services, while delivery latency and per-stream frame numbers assess timeliness and freshness.

**Other Camera Stacks.** Shared acquisition mechanisms provide a basis for applying these analysis steps to other platforms. For example, Orbbec [22] and ZED [40] use threaded V4L2 capture in the Orbbec SDK v2 backend and Stereolabs’ open-source `zed-open-capture` library, respectively. Backends using Linux `uvccvideo` also share its UVC request pool and asynchronous payload-copy work, allowing the same worker and receive-path analysis. Stereolabs further reports that increasing camera V4L2 priority improves Linux USB reliability under heavy CPU load, reinforcing the importance of host-side configuration [41].

**Backend-Specific Receive Resources.** Orbbec recommends increasing the `usbfs` transfer-memory limit for multiple USB devices or high-volume streaming [21]. Although this controls transfer memory rather than the `uvccvideo`

request-pool size set by `UVC_URBS`, both reflect the same principle: provisioning host receive resources alongside SDK processing.

**Configuration and Framework Boundaries.** Transferring the method requires examining the target implementation and frame-identifier semantics, measuring timing parameters, and tuning scheduling and receive resources for its workload. Linux or SDK version changes require rechecking worker roles, dependencies, and timing parameters. On a real-time operating system (RTOS), the multi-rate, self-suspending model remains applicable when acquisition tasks operate at different rates and block while waiting for input, but the task structure and driver dependencies must be identified for that implementation. Robotics frameworks using the same acquisition backend retain these dependencies; their executors and middleware add downstream stages whose end-to-end timing has its own analyses [3, 43].

## 8 Conclusion

A nominal camera FPS does not by itself guarantee timely, fresh RGB-D frame-set delivery. We showed that recurring `librealsense` frame acquisition forms a multi-rate, self-suspending worker graph rather than one periodic task. Host receive-to-frameset latency and per-stream freshness together expose delivery failures that API return success alone cannot reveal.

Our results identify three complementary measures for improving real-time RealSense frame acquisition: increasing the UVC receive pool (e.g., to `URB16` on our platform), using `PREEMPT_RT` or threaded interrupt service so that xHCI handling becomes schedulable, and coordinating the scheduling of SDK workers with kernel receive-path services. Together, these controls reduce interference-induced host latency and eliminate all observed non-fresh framesets across all evaluated coordinated `PREEMPT_RT` treatments. Real-time frame acquisition should therefore be configured across layers rather than by raising application priority alone.

Three directions follow from this study. First, the extracted task graph and calibrated parameters invite a schedulability analysis whose predicted response times can be compared with the measured host latencies. Second, the method should be applied to further hosts, backends, and camera profiles to separate platform-specific values from structural findings. Third, freshness as defined here tracks frame advancement; combining it with the camera timestamp would quantify the data age of each delivered frameset.

**Acknowledgments.** The authors acknowledge the support from Ken Hasselmann and Jim Bauwens for providing D455 and D415 RealSense cameras. This work was supported by imec through the SAFEBOAT research program, joint PhD project 3E240707 (KU Leuven, VUB). The authors used ChatGPT/Codex and Claude for manuscript revision and for analysis and reproducibility scripts, verified all outputs, and take full responsibility for the work.

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Brandenburg, B.B., Courtaud, C., Marković, F., Ye, B.: LiME: The Linux Real-Time Task Model Extractor. In: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2025). pp. 255–269. IEEE (2025). <https://doi.org/10.1109/RTAS65571.2025.00033>
2. Carfagni, M., Furferi, R., Governi, L., Servi, M., Uccheddu, F., Volpe, Y.: On the Performance of the Intel SR300 Depth Camera: Metrological and Critical Characterization. *IEEE Sensors Journal* **17**(14), 4508–4519 (2017). <https://doi.org/10.1109/JSEN.2017.2703829>
3. Casini, D., Blaß, T., Lütkebohle, I., Brandenburg, B.B.: Response-Time Analysis of ROS 2 Processing Chains under Reservation-Based Scheduling. In: Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS). Leibniz International Proceedings in Informatics (LIPIcs), vol. 133, pp. 6:1–6:23. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2019). <https://doi.org/10.4230/LIPIcs.ECRTS.2019.6>
4. Chen, J.J., Hahn, T., Hoeksma, R., Megow, N., von der Brüggen, G.: Scheduling Self-Suspending Tasks: New and Old Results. In: Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS). Leibniz International Proceedings in Informatics (LIPIcs), vol. 133, pp. 16:1–16:23. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2019). <https://doi.org/10.4230/LIPIcs.ECRTS.2019.16>
5. Craciunas, S.S., Serna Oliver, R.: Combined Task- and Network-Level Scheduling for Distributed Time-Triggered Systems. *Real-Time Systems* **52**(2), 161–200 (2016). <https://doi.org/10.1007/s11241-015-9244-x>
6. Dewit, W., Paolillo, A., Goossens, J.: A Preliminary Assessment of the Real-Time Capabilities of Real-Time Linux on Raspberry Pi 5. In: Proceedings of the 18th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT) (2024)
7. Golchin, A., Cheng, Z., West, R.: Tuned Pipes: End-to-End Throughput and Delay Guarantees for USB Devices. In: Proceedings of the IEEE Real-Time Systems Symposium (RTSS). pp. 196–207. IEEE (2018). <https://doi.org/10.1109/RTSS.2018.00037>
8. Kang, K.D., Son, S.H., Stankovic, J.A.: Managing Deadline Miss Ratio and Sensor Data Freshness in Real-Time Databases. *IEEE Transactions on Knowledge and Data Engineering* **16**(10), 1200–1216 (2004). <https://doi.org/10.1109/TKDE.2004.61>
9. Kerrisk, M., Linux man-pages contributors: sched(7): Overview of CPU Scheduling. Linux man-pages project (2026), <https://man7.org/linux/man-pages/man7/sched.7.html>, accessed 2026-10-15
10. Linux Foundation Real-Time Linux: PREEMPT\_RT Patch Versions (2026), [https://wiki.linuxfoundation.org/realtime/preempt\\_rt\\_versions](https://wiki.linuxfoundation.org/realtime/preempt_rt_versions), accessed 2026-10-15
11. Linux kernel developers: UVC Video Driver Source. Linux kernel source code (2024), <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/media/usb/uvc?h=v6.12>, version 6.12, accessed 2026-10-15
12. Linux Kernel Documentation: Deadline Task Scheduling (2026), <https://docs.kernel.org/scheduler/sched-deadline.html>, accessed 2026-10-15
13. Linux Kernel Documentation: drm/v3d Broadcom V3D Graphics Driver (2026), <https://docs.kernel.org/gpu/v3d.html>, accessed 2026-10-15
14. Linux Kernel Documentation: EEVDF Scheduler (2026), <https://docs.kernel.org/scheduler/sched-eevdf.html>, accessed 2026-10-15

15. Linux Kernel Documentation: Theory of Operation for PREEMPT\_RT (2026), <https://docs.kernel.org/core-api/real-time/theory.html>, accessed 2026-10-15
16. Linux Kernel Documentation: USB Request Block (URB) (2026), <https://docs.kernel.org/driver-api/usb/URB.html>, accessed 2026-10-15
17. Linux Kernel Documentation: Video4Linux Input/Output Methods (2026), <https://docs.kernel.org/userspace-api/media/v4l/io.html>, accessed 2026-10-15
18. Liu, L., Lee, J., Shin, K.G.: RT-BEV: Enhancing Real-Time BEV Perception for Autonomous Vehicles. In: Proceedings of the IEEE Real-Time Systems Symposium (RTSS). pp. 267–279. IEEE (2024). <https://doi.org/10.1109/RTSS62706.2024.00031>
19. Liu, S., Wagle, R., Anderson, J.H., Yang, M., Zhang, C., Li, Y.: Autonomy Today: Many Delay-Prone Black Boxes. In: Proceedings of the 36th Euromicro Conference on Real-Time Systems (ECRTS). Leibniz International Proceedings in Informatics (LIPIcs), vol. 298, pp. 12:1–12:27. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2024). <https://doi.org/10.4230/LIPIcs.ECRTS.2024.12>
20. de Oliveira, D.B., Casini, D., de Oliveira, R.S., Cucinotta, T.: Demystifying the Real-Time Linux Scheduling Latency. In: Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS). Leibniz International Proceedings in Informatics (LIPIcs), vol. 165, pp. 9:1–9:23. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2020). <https://doi.org/10.4230/LIPIcs.ECRTS.2020.9>
21. Orbbec: Orbbec SDK v2: Performance Tuning, [https://orbbec.github.io/OrbbecSDK\\_v2/docs/tutorial/performance\\_tuning.html](https://orbbec.github.io/OrbbecSDK_v2/docs/tutorial/performance_tuning.html), accessed 2026-10-15
22. Orbbec: Orbbec SDK v2: V4L2 UVC Device Port. Source code, [https://github.com/orbbec/OrbbecSDK\\_v2/blob/main/src/platform/usb/uvc/ObV4lUvcDevicePort.cpp](https://github.com/orbbec/OrbbecSDK_v2/blob/main/src/platform/usb/uvc/ObV4lUvcDevicePort.cpp), accessed 2026-10-15
23. Paolillo, A., Van Molle, M., Hasselmann, K.: benchkit: A Declarative Framework for Composable Performance Evaluation of System Software. In: Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE). pp. 170–183. Association for Computing Machinery (2026). <https://doi.org/10.1145/3777884.3796997>
24. Raspberry Pi Documentation: Processors: BCM2712 (2026), <https://www.raspberrypi.com/documentation/computers/processors.html>, accessed 2026-10-15
25. RealSense: Change UVC\_URBS 5 to 16. Linux kernel patch distributed with **librealsense** (2020), [https://github.com/realsenseai/librealsense/blob/master/scripts/uvcvideo\\_increase\\_UVC\\_URBS.patch](https://github.com/realsenseai/librealsense/blob/master/scripts/uvcvideo_increase_UVC_URBS.patch), accessed 2026-10-15
26. RealSense: RealSense Stereo Series Product Family Datasheet (2025), <https://dev.realsenseai.com/docs/realsense-stereo-series-product-family-datasheet/>, revision 021, accessed 2026-10-15
27. RealSense: Choose LibRealSense SDK Backend. Project documentation (2026), [https://github.com/realsenseai/librealsense/blob/v2.58.3/doc/installation\\_jetson.md](https://github.com/realsenseai/librealsense/blob/v2.58.3/doc/installation_jetson.md), version 2.58.3, accessed 2026-10-15
28. RealSense: Frame Management. Project documentation (2026), [https://github.com/realsenseai/librealsense/blob/master/doc/frame\\_lifetime.md](https://github.com/realsenseai/librealsense/blob/master/doc/frame_lifetime.md), accessed 2026-10-15
29. RealSense: librealsense API Architecture. Project documentation (2026), <https://dev.realsenseai.com/docs/api-architecture/>, accessed 2026-10-15
30. RealSense: librealsense: RealSense SDK. GitHub repository (2026), <https://github.com/realsenseai/librealsense>, accessed 2026-10-15
31. RealSense: librealsense v2.58.3: Timestamp Synchronization Implementation. Source code, `timestamp_composite_matcher::skip_missing_stream` (2026), [https://github.com/realsenseai/librealsense/blob/v2.58.3/src/timestamp/timestamp\\_composite\\_matcher.cpp](https://github.com/realsenseai/librealsense/blob/v2.58.3/src/timestamp/timestamp_composite_matcher.cpp), accessed 2026-10-15

- [//github.com/realsenseai/librealsense/blob/v2.58.3/src/sync.cpp](https://github.com/realsenseai/librealsense/blob/v2.58.3/src/sync.cpp), accessed 2026-10-15
32. RealSense: Multi-Camera Configurations—D400 Series Stereo Cameras (2026), <https://dev.realsenseai.com/docs/multiple-depth-cameras-configuration/>, accessed 2026-10-15
  33. RealSense: realsense-ros: ROS Wrapper for RealSense Cameras. GitHub repository (2026), <https://github.com/realsenseai/realsense-ros>, accessed 2026-10-15
  34. Rustler, L., Volprecht, V., Hoffmann, M.: Empirical Comparison of Four Stereoscopic Depth Sensing Cameras for Robotics Applications. *IEEE Access* **13**, 67564–67577 (2025). <https://doi.org/10.1109/ACCESS.2025.3560810>
  35. Scholz, C., Cao, H.L., Imrith, E., Roshandel, N., Firouzipouyaei, H., Burkiewicz, A., Amighi, M., Menet, S., Sisavath, D.W., Paolillo, A., Rottenberg, X., Gerets, P., Cheyns, D., Dahlem, M., Ocket, I., Genoe, J., Philips, K., Stoffelen, B., van den Bosch, J., Latre, S., Vanderborght, B.: Sensor-Enabled Safety Systems for Human-Robot Collaboration: A Review. *IEEE Sensors Journal* **25**(1), 65–88 (2025). <https://doi.org/10.1109/JSEN.2024.3496905>
  36. Shen, Y., Scholz, C., Vanderborght, B., Joosen, W., Paolillo, A.: rtms26-realsense: Code, Benchmarks, and Data for From Threads to Fresh Frames. Source code repository (2026), <https://github.com/softwarelanguageslab/rtms26-realsense>
  37. Shen, Y., Scholz, C., Vanderborght, B., Joosen, W., Paolillo, A.: Supplementary Material for “From Threads to Fresh Frames: Cross-Layer Modeling and Scheduling of RealSense on Linux”. Zenodo (2026). <https://doi.org/10.5281/zenodo.23420829>
  38. Shen, Y., Vander Mynsbrugge, J., Roshandel, N., Bouchez, R., FirouziPouyaei, H., Scholz, C., Cao, H.L., Vanderborght, B., Joosen, W., Paolillo, A.: SentryRT-1: A Case Study in Evaluating Real-Time Linux for Safety-Critical Robotic Perception. In: *Proceedings of the 19th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*. pp. 35–41 (2025)
  39. Sobhani, H., Kim, H.: Modeling and Scheduling of Fusion Patterns in Autonomous Driving Systems. In: *Proceedings of the 33rd International Conference on Real-Time Networks and Systems (RTNS)*. *Lecture Notes in Computer Science*, Springer (2025)
  40. Stereolabs: ZED Open Capture: Video Capture Implementation. Source code, <https://github.com/stereolabs/zed-open-capture/blob/master/src/videocapture.cpp>, accessed 2026-10-15
  41. Stereolabs: ZED SDK 4.0 Release Notes, <https://docs.stereolabs.com/docs/development/zed-sdk/release-notes/4-x/4-0>, version 4.0.7, accessed 2026-10-15
  42. Tadic, V., Toth, A., Vizvari, Z., Klincsik, M., Sari, Z., Sarcevic, P., Sarosi, J., Biro, I.: Perspectives of RealSense and ZED Depth Sensors for Robotic Vision Applications. *Machines* **10**(3), 183 (2022). <https://doi.org/10.3390/machines10030183>
  43. Teper, H., Günzel, M., Ueter, N., von der Brüggen, G., Chen, J.J.: End-To-End Timing Analysis in ROS 2. In: *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*. pp. 53–65. IEEE (2022). <https://doi.org/10.1109/RTSS55097.2022.00015>
  44. Yates, R.D., Sun, Y., Brown, D.R., Kaul, S.K., Modiano, E., Ulukus, S.: Age of Information: An Introduction and Survey. *IEEE Journal on Selected Areas in Communications* **39**(5), 1183–1210 (2021). <https://doi.org/10.1109/JSAC.2021.3065072>

## A Detailed Experimental Configuration

All measured binaries use `librealsense 2.58.3` in `RelWithDebInfo`. This configuration retains the creator-stack symbols required for worker identification while applying the common `-O2 -DNDEBUG` optimization. The matched Linux 6.12.96 kernels include BTF for symbolized kernel-event recording. Section B describes E4’s instrumentation. We denote kernel builds configured with `UVC_URBS=5` and `UVC_URBS=16` as `URB5` and `URB16`, respectively.

The Raspberry Pi 5 CPU frequency is fixed at 1500 MHz for every measured run. USB autosuspend is disabled, and the campaign drops file-system caches before a run. The cameras use externally powered USB 3.0 hubs. Acceptance checks verify SuperSpeed operation, the declared camera topology, and the requested hard- or threaded-IRQ mode. No CPU affinity is assigned to the benchmark application, SDK workers, interference processes, or threaded xHCI handlers, except in E3: its selected xHCI IRQ thread and targeted interferer are both pinned to CPU 3.

The representative workload uses  $848 \times 480$  Z16 Depth and  $640 \times 480$  RGB8 Color at 30 FPS on both camera models. The D435 stress workload uses  $848 \times 480$  Z16 Depth,  $848 \times 480$  Y8 IR1 and IR2, and  $960 \times 540$  RGB8 Color at 60 FPS. The D455 stress workload uses the same Depth and infrared profiles with  $848 \times 480$  RGB8 Color at 60 FPS. These are the highest validated model-specific resolutions for the enabled streams at that frame rate.

The recurring-worker calibration (E1) covers one D435 and one D455, both workloads, and three independent 30-s traces per camera-model-workload cell. It uses the RT `URB16` kernel, userspace `SCHED_OTHER`, xHCI FIFO 90, no interference, and no CPU affinity. LiME is aligned with kernel, SDK, application, and frame events to correlate scheduling, USB/UVC/V4L2 progress, synchronization, and frameset delivery. The resulting twelve full-path traces supply the temporal and resource quantities reported in the main paper.

Each repetition launches a new process and resets every selected camera. The pipelines start under `SCHED_OTHER`. After warm-up, the campaign matches recurring workers, installs the requested scheduling treatment, starts the interference processes, and waits until all components report ready. Only then does it open the fixed 30-s measurement window. A startup failure resets the cameras and retries the same logical run. E4 uses three permuted rounds to vary treatment order. Frame-event storage is preallocated, and freshness statistics are computed after the measured interval.

An attempt is accepted only when it completes the measurement window and establishes the declared topology, effective policies, interference state, and metric schema. Benchkit expands the factor combinations and records the command, configuration, logs, topology, policy audit, frame metadata, and result artifacts for every attempt.

The recorded inter-delivery statistics comprise the mean, median, standard deviation, p99, and observed maximum for each camera. Freshness accounting includes per-stream unique frames, duplicates, sequence gaps, partially and fully stale framesets, and API timeouts. Full-path studies additionally measure host

**Table S1.** Diagnostic resource and inter-delivery overhead for one D435. Each value is the mean of three 30-s runs. CPU is a percentage of one core, resident set size (RSS) is the maximum during the measured interval, and Max is the mean of the three per-run maxima.

| Workload               | Diagnostic mode | CPU (%) | RSS (MiB) | Mean (ms) | p99 (ms) | Max (ms) | Trace (MiB) |
|------------------------|-----------------|---------|-----------|-----------|----------|----------|-------------|
| Representative, 30 FPS | Off             | 4.408   | 27.9      | 33.357    | 33.423   | 33.931   | 0           |
|                        | Build-only      | 4.387   | 27.7      | 33.357    | 33.453   | 36.963   | 0           |
|                        | Full            | 4.453   | 393.9     | 33.357    | 33.449   | 33.495   | 2.21        |
| Stress, 60 FPS         | Off             | 21.307  | 46.8      | 16.799    | 16.969   | 17.606   | 0           |
|                        | Build-only      | 21.430  | 46.7      | 16.799    | 16.940   | 17.506   | 0           |
|                        | Full            | 21.736  | 413.4     | 16.799    | 16.915   | 17.173   | 6.29        |

receive-to-frameset latency from the xHCI activation handling the first relevant URB of the earliest selected component to the return from `wait_for_frames()`. Camera exposure precedes this host-controlled interval and is not included.

## B Instrumentation Overhead

E4 uses two separate instrumentation paths. The userspace diagnostic markers remain compiled into the binary, but their event sink is disabled. An external `trace-cmd` recorder collects only the kernel events needed to identify the xHCI activation associated with the first relevant URB and align it with the corresponding application return. This latency-only event set is a strict subset of the full receive-path kernel trace used for model construction and diagnosis. Under the same recorder configuration, the subset can only reduce the number of emitted records and their associated buffer-copy work relative to the full trace. In particular, it retains four dynamic receive-path probes and the xHCI IRQ events needed for the latency boundary, while omitting the full trace’s additional USB bottom-half, UVC-copy, frame-boundary, `softirq`, `V4L2/vb2`, xHCI-giveback, and userspace scheduling events.

The following experiment isolates the userspace marker overhead; it does not separately measure the remaining latency-only `trace-cmd` overhead. The final source revision adds markers around frameset enqueueing and retrieval for targeted path diagnosis, so we repeated the experiment on that revision. The controlled experiment used one D435, `SCHED_OTHER`, no injected interference, and the non-RT Linux 6.12.96 kernel. Each run contained a 10-s warm-up followed by a 30-s measurement. Three independent runs covered each workload and diagnostic mode. Diagnostics-off removes the marker code at compile time, build-only compiles the markers but disables the sink, and full mode records every enabled marker. The mode order was rotated between repetitions. In build-only mode, the trace-file environment variable is unset. The trace object therefore remains inactive, each marker returns after its inactive-state check, no event ring is allocated, and no trace file is written. Table S1 reports the resulting resource and inter-delivery statistics.

Inter-delivery intervals do not show whether returned data became older. The benchmark application therefore records the application-return time together

**Table S2.** Stress-workload data age and timestamp-to-return spans. All values are in milliseconds. The p99 and Max columns are means of the three per-run values.

| Diagnostic mode | Sensor-age mean | Sensor-age p99 | Sensor-age Max | Backend-to-return p99 | Arrival-to-return p99 |
|-----------------|-----------------|----------------|----------------|-----------------------|-----------------------|
| Off             | 17.840          | 18.074         | 18.558         | 18.711                | 11.077                |
| Build-only      | 17.788          | 18.053         | 18.545         | 18.688                | 11.041                |
| Full            | 17.806          | 17.974         | 18.178         | 18.692                | 11.031                |

with the global sensor timestamp, the librealsense backend timestamp, and the frame time-of-arrival metadata. Table S2 uses the oldest component of each frameset and reports the mean of the three per-run statistics. Sensor age is the application-return time minus the global sensor timestamp. The other columns use the backend and arrival timestamps as their respective starting boundaries. Their timestamp semantics differ, so comparisons are made across diagnostic modes within a column rather than between columns.

Build-only changes sensor-age p99 by  $-0.022$  ms, backend-to-return p99 by  $-0.023$  ms, and arrival-to-return p99 by  $-0.036$  ms relative to off. Full recording changes the same quantities by  $-0.100$ ,  $-0.019$ , and  $-0.046$  ms. The small changes are all negative and show no monotonic increase with the amount of tracing. Thus, the stress experiment finds no detectable degradation in the age of returned data or in these aggregate chain boundaries at its resolution. It does not compare every internal kernel and SDK segment independently, because the off build intentionally emits no stage markers.

The data-quality check archived and repeated one initial build-only trial that contained one duplicate and one matching sequence gap. The 18 accepted runs contain no timeout, duplicate, or sequence gap. Relative to diagnostics-off (Table S1), build-only changes mean CPU utilization by  $-0.49\%$  and  $+0.58\%$  at 30 and 60 FPS, respectively, with inter-delivery p99 shifts of  $+0.030$  and  $-0.029$  ms and no RSS increase. At 30 FPS, build-only raises the mean of per-run maxima by  $3.032$  ms; full recording does not reproduce this increase and changes inter-delivery p99 by only  $+0.027$  ms at that rate. Build-only leaves the mean inter-delivery time unchanged and shifts p99 only slightly, but the larger 30-FPS maxima leave its effect on rare delays unresolved. The experiment does not establish zero overhead, a failure-probability bound, or a worst-case bound.

Full recording leaves the inter-delivery distribution nearly unchanged, but it increases RSS by 366–367 MiB. This increase is the configured 12-million-entry ring at 32 bytes per event, not retained frame data. Full mode also raises process-lifetime CPU time by 0.27–0.42 s because it initializes and persists the ring outside the measured steady-state interval. These costs justify build-only marker instrumentation for E4 and full marker recording only for short, targeted diagnostic experiments.

## C Detailed UVC Freshness Patterns

The E2 URB5 runs contain three frame-metadata patterns under the main paper’s stream-advancement indicator  $u$  and sequence-gap count  $g$ . Table S3 gives their

**Table S3.** URB5 metadata patterns. Unlisted streams advance without a gap. A “next” gap appears when a stream reused at delivery  $k$  catches up at  $k + 1$ . Occurrences count the two-return reuse pattern once.

| Pattern                   | $u/g$ signature                                                                                                | Return state(s)                         | Count                       | Interpretation                                                               |
|---------------------------|----------------------------------------------------------------------------------------------------------------|-----------------------------------------|-----------------------------|------------------------------------------------------------------------------|
| Depth-only reuse          | $u_{D,k} = 0$<br>$g_{D,k+1} = 1$                                                                               | Partially stale at $k$ ; gap at $k + 1$ | 2 at 30 FPS;<br>1 at 60 FPS | No acceptable new Depth input reaches the join.                              |
| Depth & IR reuse          | $u_{s,k} = 0, \forall s \in \mathcal{S}_{\text{st}}$<br>$g_{s,k+1} = 1, \forall s \in \mathcal{S}_{\text{st}}$ | Partially stale at $k$ ; gap at $k + 1$ | 9 at 60 FPS                 | Depth and IR1/2 inputs are unavailable while Color advances.                 |
| All-stream period skipped | $u_{s,k} = 1, \forall s \in \mathcal{S}$<br>$g_{s,k} = 1, \forall s \in \mathcal{S}$                           | Non-fresh (gap) at $k$                  | 3 at 60 FPS                 | No stale return; one complete application-visible delivery period is absent. |

signatures and occurrence counts.  $D$ ,  $I_1$ ,  $I_2$ , and  $C$  denote Depth, IR1, IR2, and Color. The camera index is omitted,  $\mathcal{S}_{\text{st}} = \{D, I_1, I_2\}$ , and  $\mathcal{S}$  is the set of required streams.

At 60 FPS, the nine stereo-group reuse occurrences contribute 27 duplicate stream observations across nine stale returns; the depth-only occurrence adds one of each. Their ten catch-up returns contribute 28 missing stream numbers. The three all-stream skips add 12 missing stream numbers across three more returns with a sequence gap. The totals are therefore 28 duplicate stream observations, 40 missing sequence numbers, 10 stale returns, and 13 returns with a sequence gap, or 23 non-fresh returns in all. Only the aligned depth-reuse examples are localized to an incomplete depth payload.

## D Maximum Observed Host Latency

Figure S1 supplements the main paper’s p99 results with the **maximum observed** host receive-to-frameset latency from the same E4 runs. Under 60-FPS CPU interference, coordinated RR and FIFO reduce p99 but increase the observed maximum on non-RT Linux, whereas all coordinated treatments reduce both metrics on PREEMPT\_RT. These maxima include waiting and processing along the receive path and can exceed a nominal frame period; they are neither CPU execution times nor worst-case bounds.

## E Full E1 Model Instantiation

Table S4 reports the E1 model instantiation of the main paper separately for the two workloads and with the full precision of the per-run estimates.

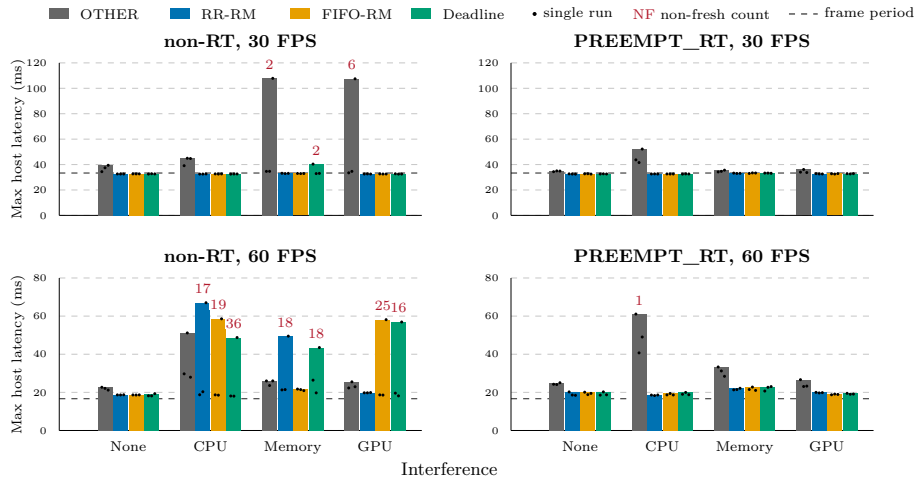


**Fig. S1.** E4 maximum observed host receive-to-frameset latency for two D435 cameras at 30 and 60 FPS. Each bar is the maximum across three 30-s runs; dots show each run's maximum. Dashed lines mark nominal frame periods. Red labels give the total non-fresh frameset count (NF) across the three runs: deliveries with at least one reused component or sequence gap. Each non-fresh frameset is counted once, and an unlabelled bar has  $NF = 0$ .

**Table S4.** Full-precision E1 model instantiation for the Representative (R, upper) and Stress (S, lower) workloads; the main paper condenses this table.  $\hat{C}$  is the largest observed logical-activation execution time,  $\hat{T}/\hat{\Delta}$  the smallest stable period or observed non-periodic activation interval,  $U^{\text{obs}}$  the mean across three runs of the average utilization of all equivalent instances (% of one core) computed from mean execution times, mean intervals, and instance counts;  $n_R, n_S$  count instances per camera. Ranges span D435 and D455, except the final row of each subtable, which spans five individually identified dispatchers.

| Worker family                                       | Activa-<br>tion   | $n_R$ | $\hat{C}_R$ ( $\mu\text{s}$ ) | $\hat{T}_R/\hat{\Delta}_R$ (ms)  | $U_R^{\text{obs}}$ (%) |
|-----------------------------------------------------|-------------------|-------|-------------------------------|----------------------------------|------------------------|
| Depth/IR capture                                    | Frame             | 1     | 827–845                       | 32.84–33.19                      | 1.11–1.12              |
| Color capture                                       | Frame             | 1     | 1708–1730                     | 32.91–33.24                      | 3.61–3.66              |
| Application wait                                    | Frame             | 1     | 160–171                       | 30.18–30.91                      | 0.173–0.181            |
| udev watcher                                        | Timer             | 1     | 75–95                         | 100.03–100.06                    | 0.0276–0.0287          |
| Time-difference keeper                              | Timer             | 1     | 368–410                       | 1000.71–1001.26                  | 0.0161–0.0205          |
| Thermal monitor (D455 only)                         | Timer             | 1     | 173                           | 2000.93                          | 0.0052                 |
| Firmware-error poller                               | Timer             | 1     | 45–56                         | 5000.07–5000.08                  | 0.0005–0.0007          |
| Notification/pipeline dispatchers<br>( $\times 5$ ) | Event/<br>timeout | 5     | 46–69                         | $\approx 5000$ (idle<br>timeout) | 0.00052–0.00073        |
| Worker family                                       | Activa-<br>tion   | $n_S$ | $\hat{C}_S$ ( $\mu\text{s}$ ) | $\hat{T}_S/\hat{\Delta}_S$ (ms)  | $U_S^{\text{obs}}$ (%) |
| Depth/IR capture                                    | Frame             | 2     | 1894–1953                     | 14.35–15.41                      | 10.87–11.13            |
| Color capture                                       | Frame             | 1     | 2166–2702                     | 16.23–16.30                      | 9.15–11.48             |
| Application wait                                    | Frame             | 1     | 160–182                       | 13.61–13.91                      | 0.368–0.402            |
| udev watcher                                        | Timer             | 1     | 140–162                       | 100.05                           | 0.0399–0.0407          |
| Time-difference keeper                              | Timer             | 1     | 377–399                       | 1001.26                          | 0.0213–0.0232          |
| Thermal monitor (D455 only)                         | Timer             | 1     | 245                           | 2000.95                          | 0.0061                 |
| Firmware-error poller                               | Timer             | 1     | 63–107                        | 5000.07–5000.08                  | 0.0008–0.0010          |
| Notification/pipeline dispatchers<br>( $\times 5$ ) | Event/<br>timeout | 5     | 55–118                        | $\approx 5000$ (idle<br>timeout) | 0.00069–0.00096        |