RTNS 2026

From Threads to Fresh Frames

Cross-Layer Modeling and Scheduling of RealSense on Linux



Yuwen Shen, Constantin Scholz, Bram Vanderborght, Wouter Joosen, Antonio Paolillo

Vrije Universiteit Brussel · KU Leuven · imec

VUB VRIJE UNIVERSITEIT BRUSSEL

KU LEUVEN

imec

Robotics × Linux × Real-Time

When megatrends converge ...

The Linux environment is the most popular open source operating system used for robotics worldwide.

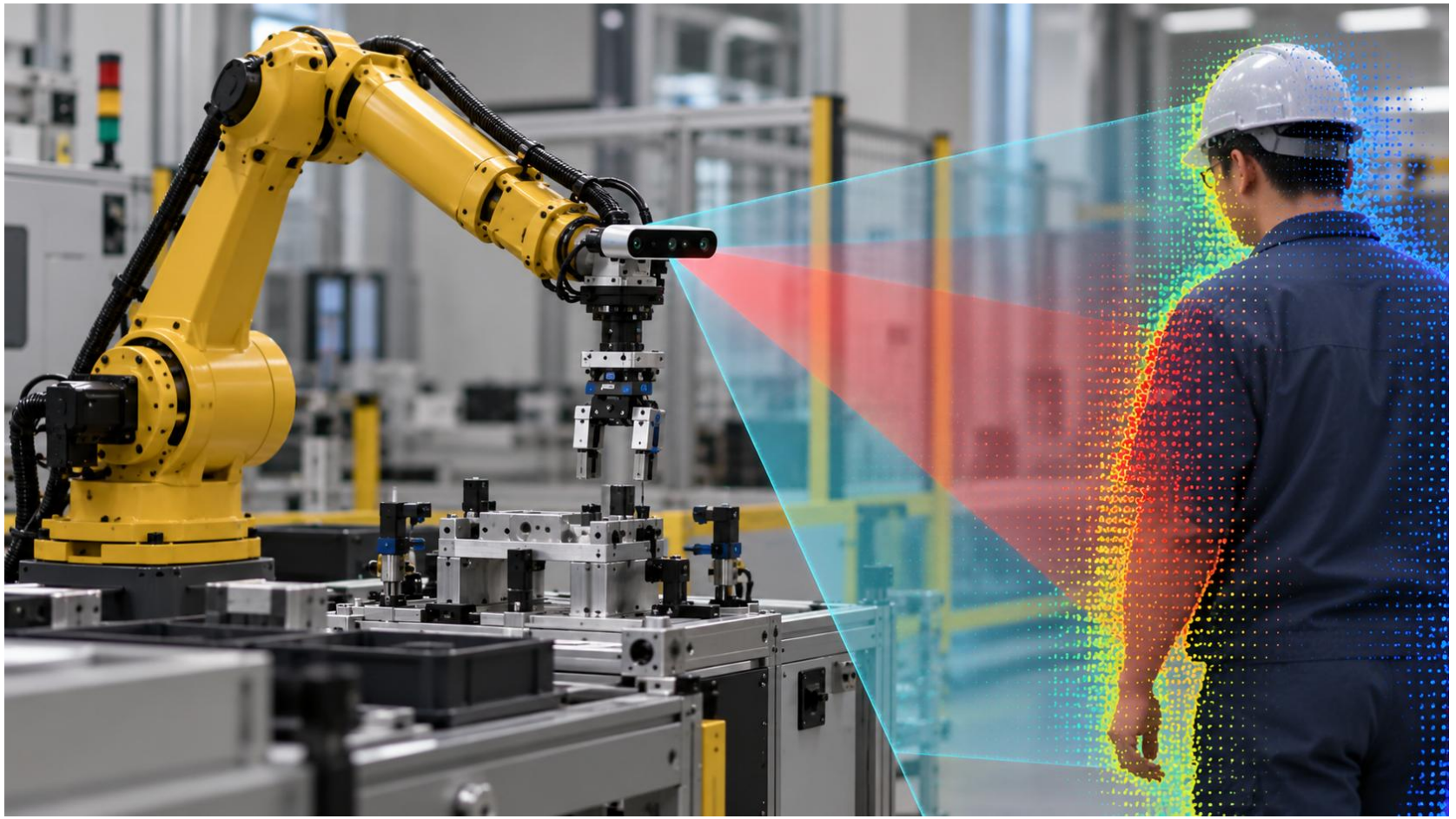
— Sifat et al., 2023

After 20 years of development ... PREEMPT_RT support [is] in the mainline kernel.

— Thomas Gleixner, 2024

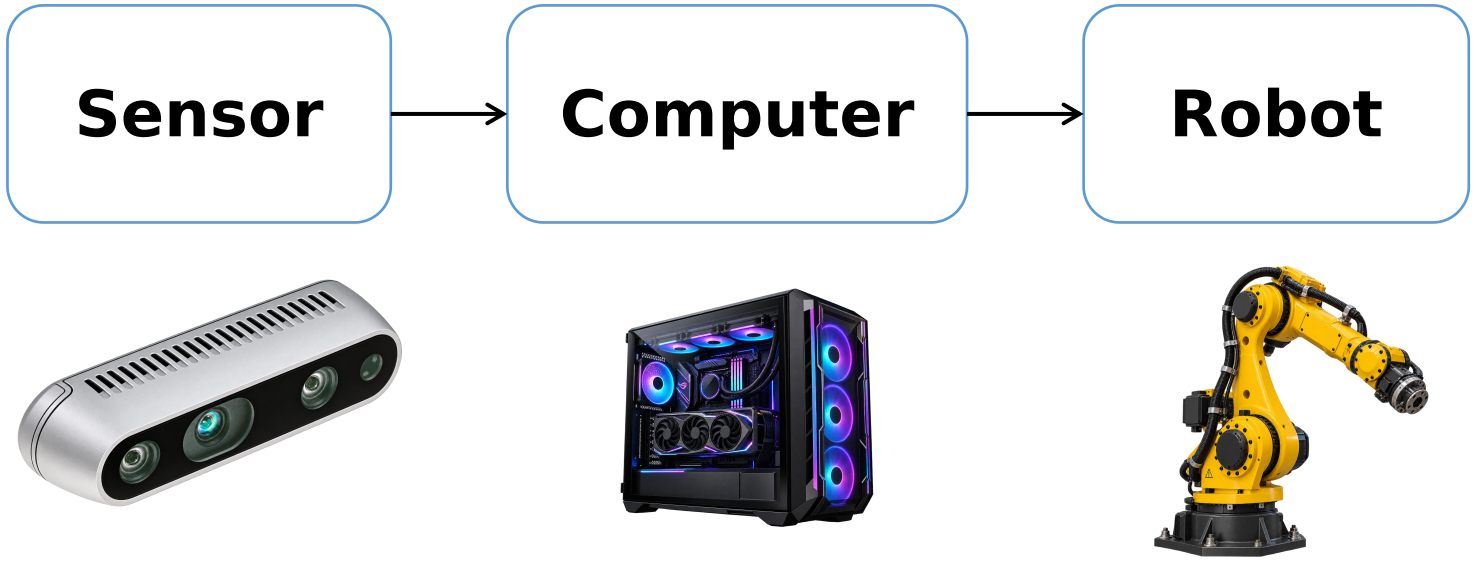


In the Robotics Sector



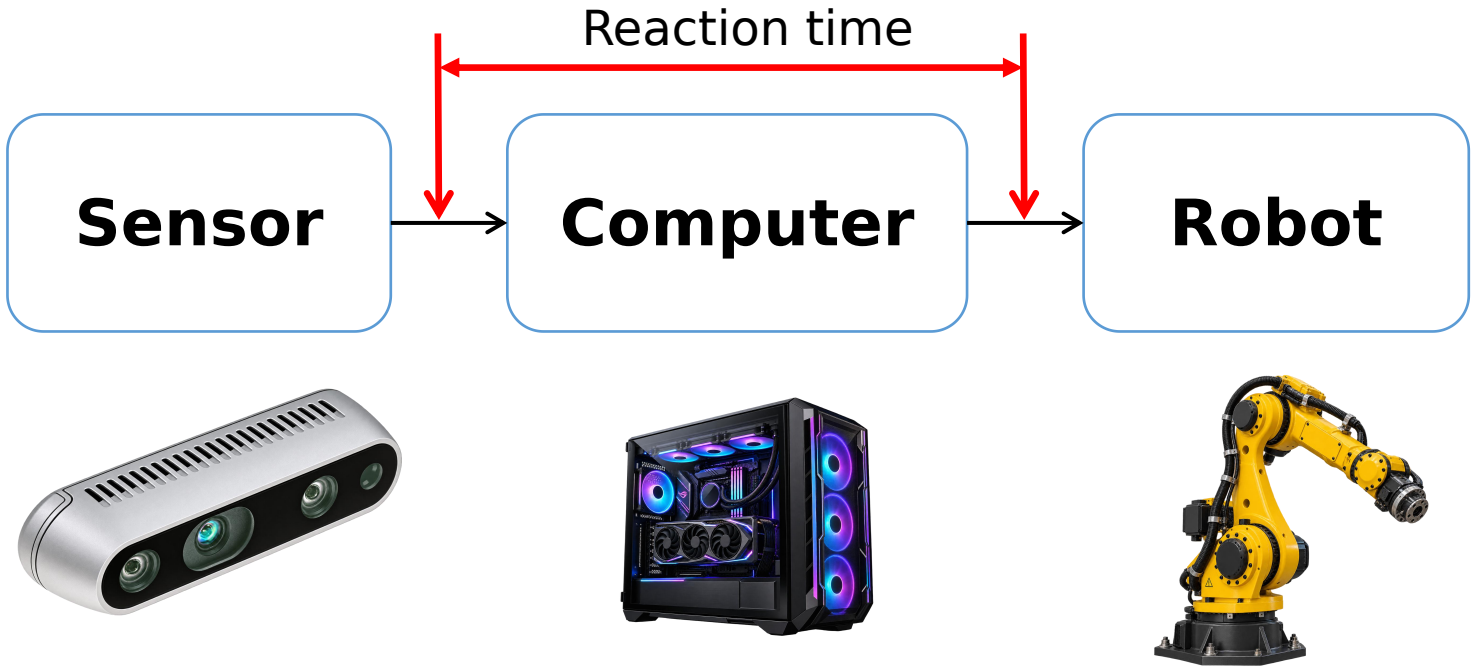
Perception-based safety layer

Build a pipeline



Shen et al., "**SentryRT-1**: A Case Study in Evaluating Real-Time Linux for Safety-Critical Robotic Perception," OSPERT 2025.

Build a pipeline



Shen et al., "**SentryRT-1**: A Case Study in Evaluating Real-Time Linux for Safety-Critical Robotic Perception," OSPERT 2025.

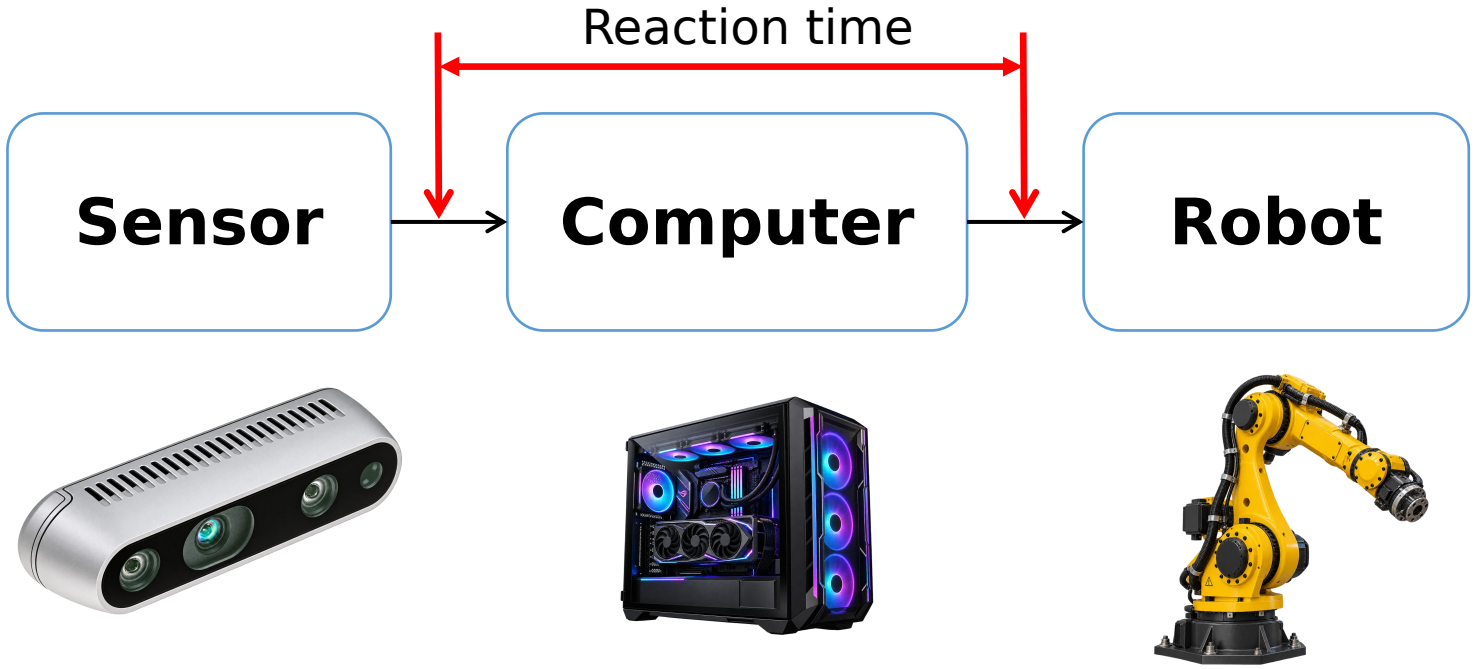
Build a pipeline

Linux:

SCHED_DEADLINE

SCHED_FIFO
SCHED_RR

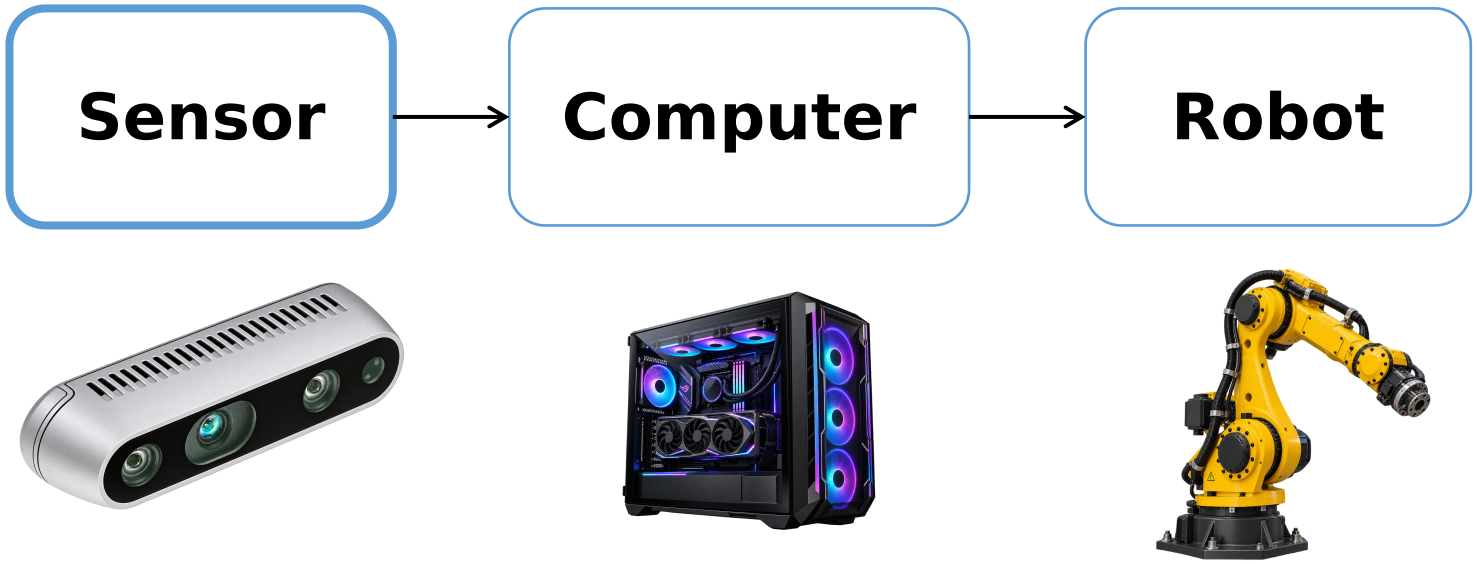
SCHED_OTHER



Shen et al., "**SentryRT-1**: A Case Study in Evaluating Real-Time Linux for Safety-Critical Robotic Perception," OSPERT 2025.

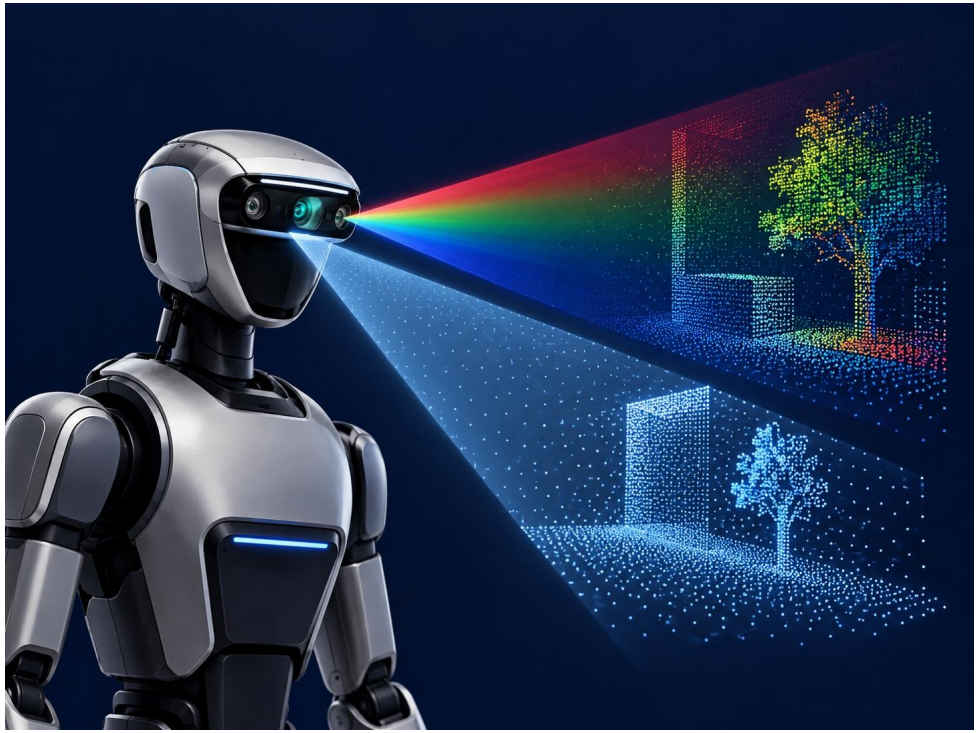
Build a pipeline

Sensors also contribute to latencies!

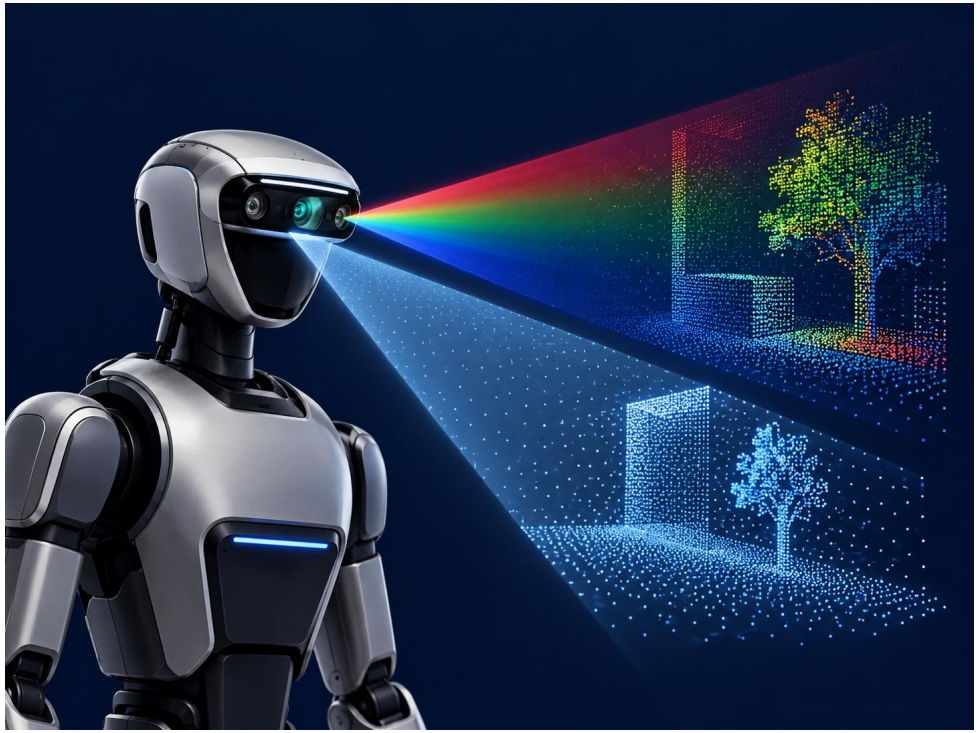


Shen et al., "**SentryRT-1**: A Case Study in Evaluating Real-Time Linux for Safety-Critical Robotic Perception," OSPERT 2025.

Real-time systems need real-time input



Real-time systems need real-time input

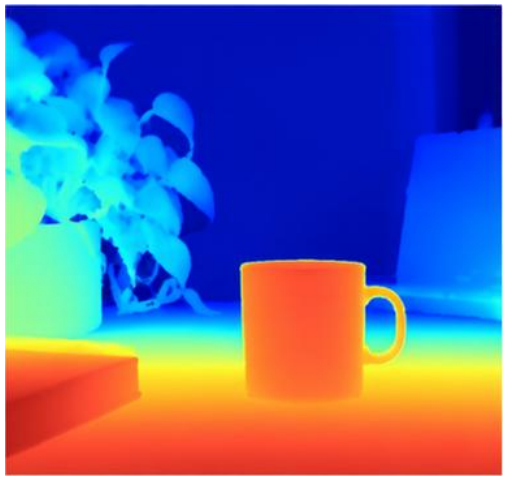




RGB-D camera



Intel RealSense





RGB-D camera



Intel RealSense



ZED 2



Orbbec

30 frames per second (FPS)
Linux
USB 3

RGB-D camera



Intel RealSense



ZED 2



Orbbec

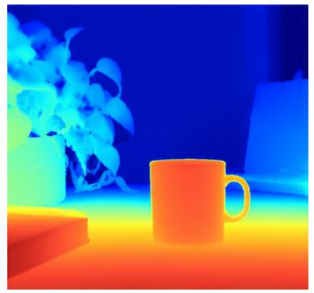
30 frames per second (FPS)
Linux
USB 3

Real-time RGB-D sensor?

The frame must arrive **on time**
e.g. 30 FPS

Delivery k=1

Depth

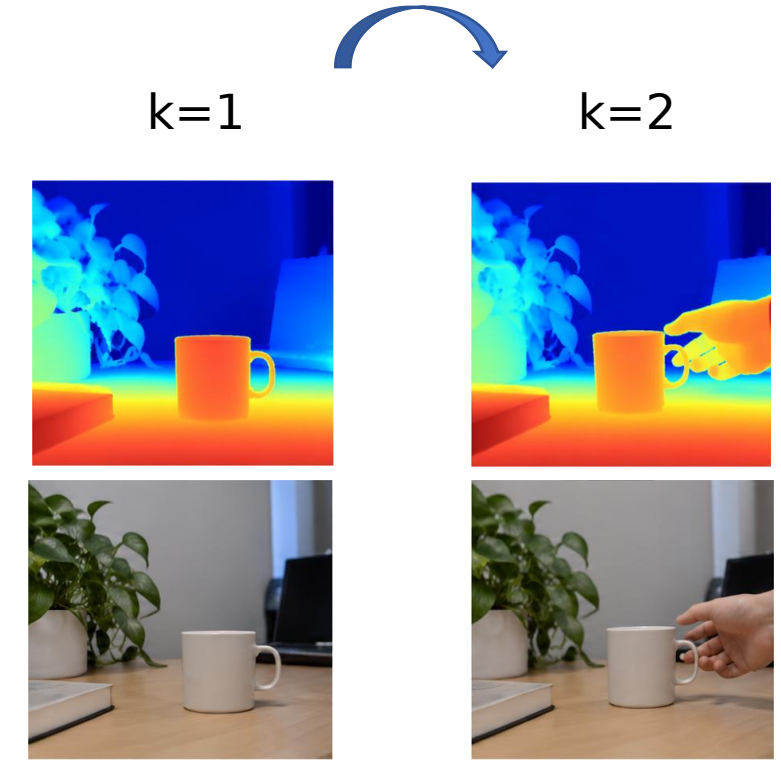


Color



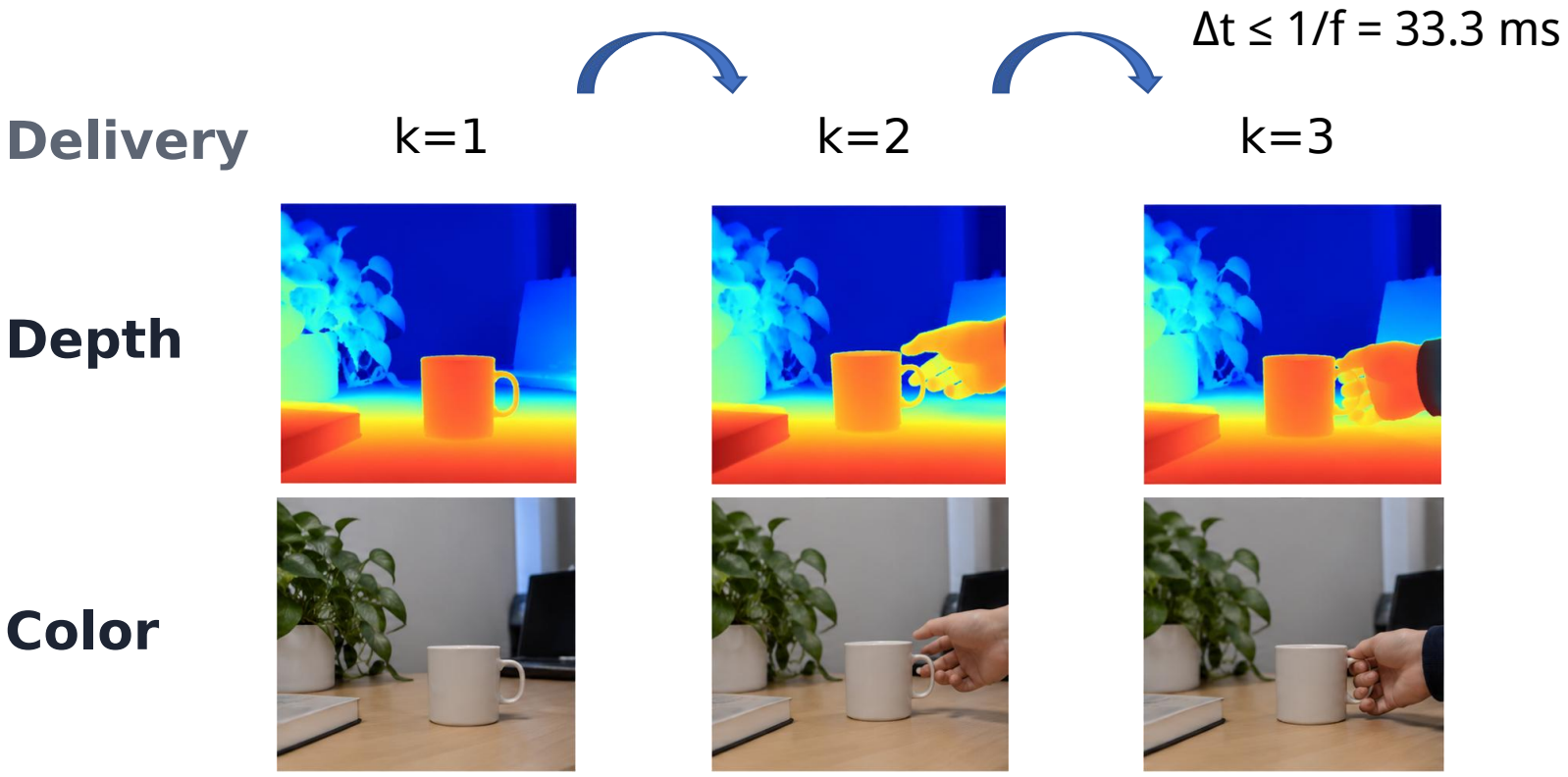
Real-time RGB-D sensor?

The frame must arrive **on time**
e.g. 30 FPS



Real-time RGB-D sensor?

The frame must arrive **on time**
e.g. 30 FPS



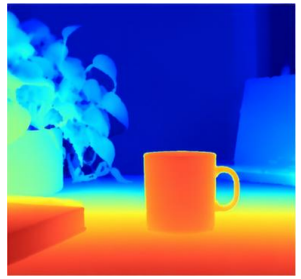
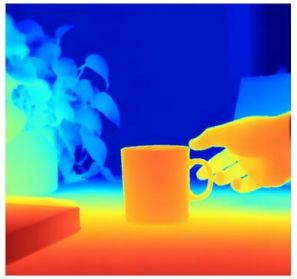
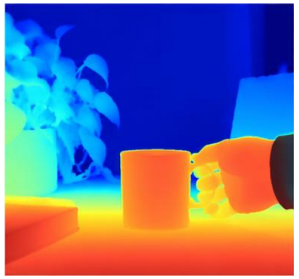
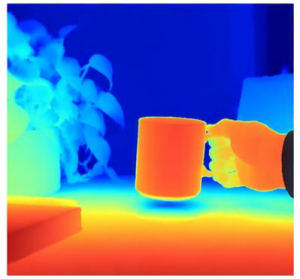
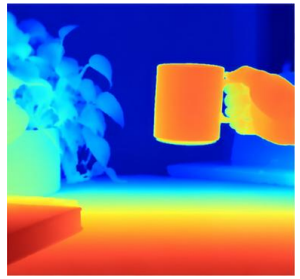
A RealSense camera configured at 30 FPS.

Does the application get a new frame every 33.3 ms?

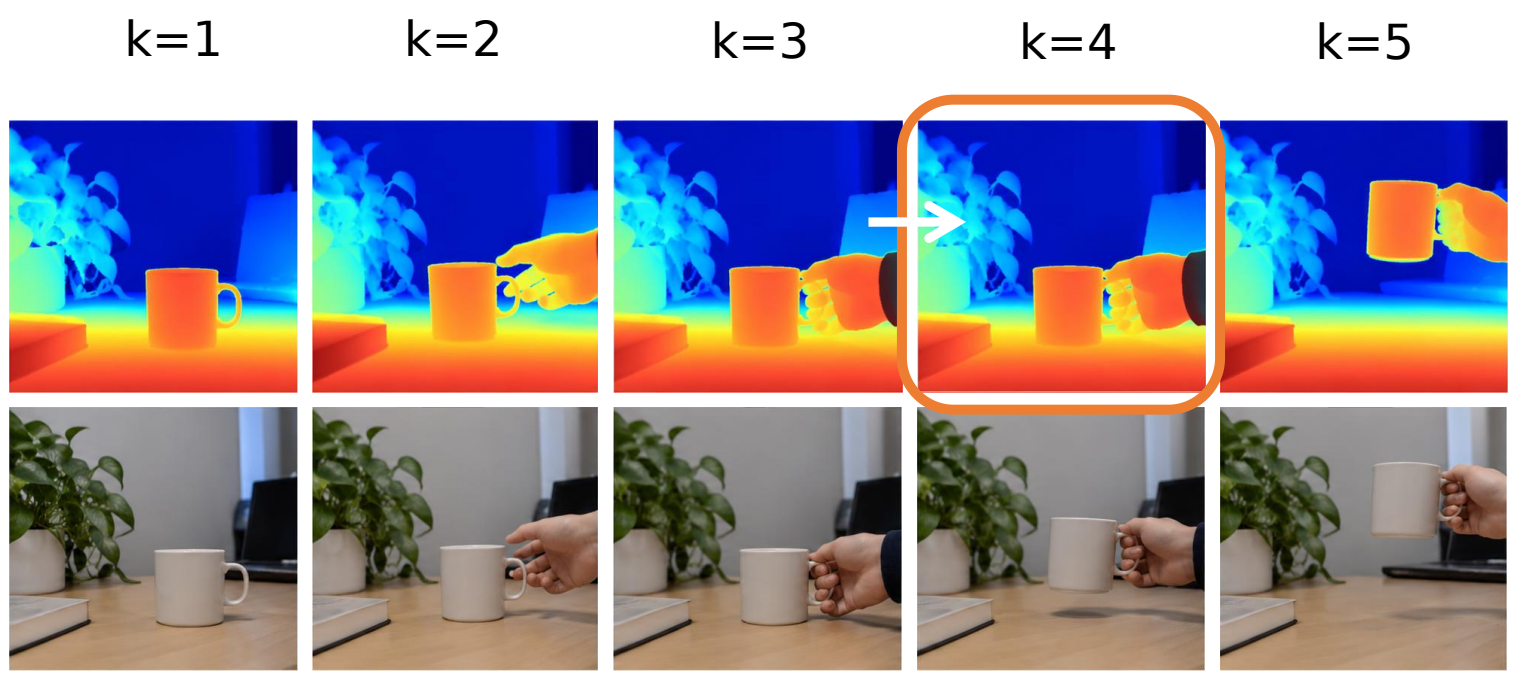
A RealSense camera configured at 30 FPS.

Does the application get a new frame every 33.3 ms?

Every call returns on time, but a depth frame does not advance.

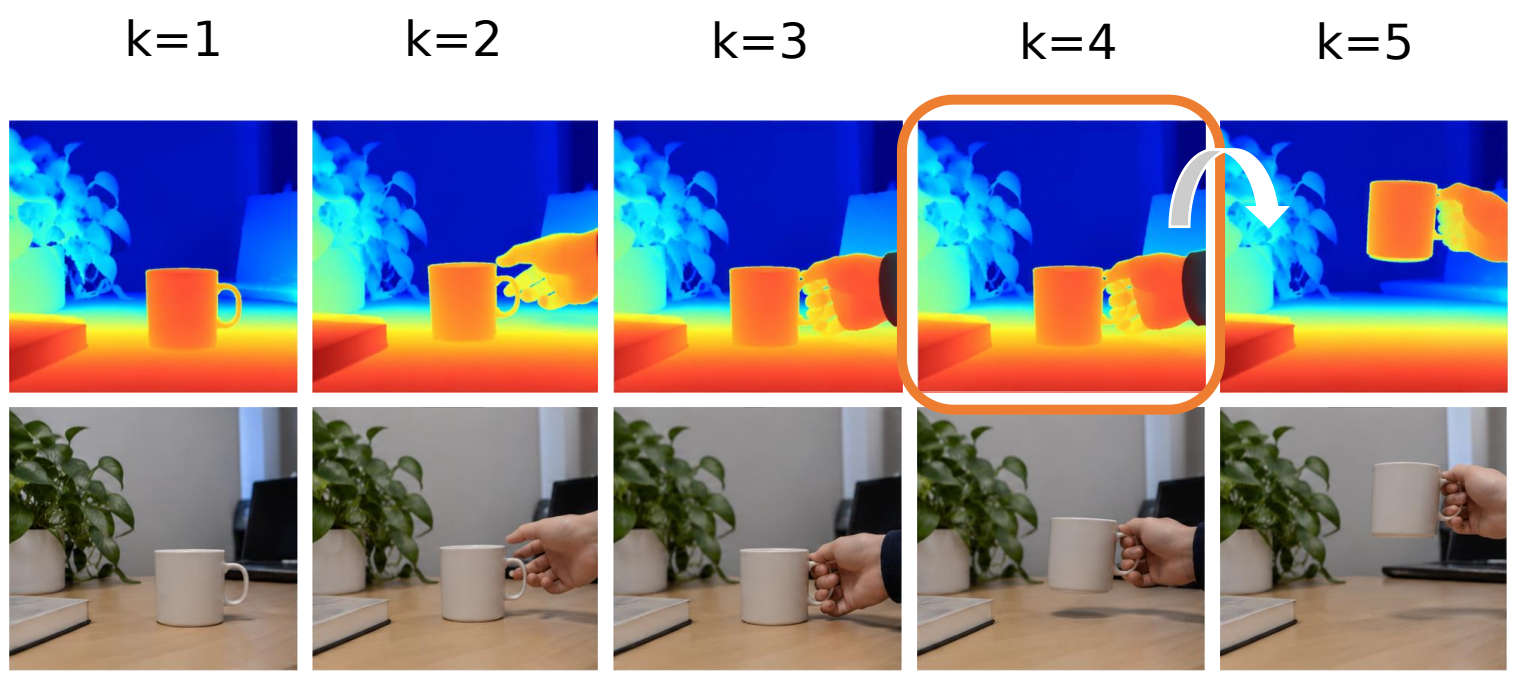
Delivery	k=1	k=2	k=3	k=4	k=5
Depth					
Color					

Every call returns on time, but a depth frame does not advance.



k=4 reuses the previous depth frame: **partially stale**.

Every call returns on time, but a depth frame does not advance.



k=4 reuses the previous depth frame: **partially stale**.

k=5 skips frame 4: **sequence gap**.

On time ≠ fresh

Make RGB-D cameras deliver **timely** and **fresh** frames
by configuring real-time Linux

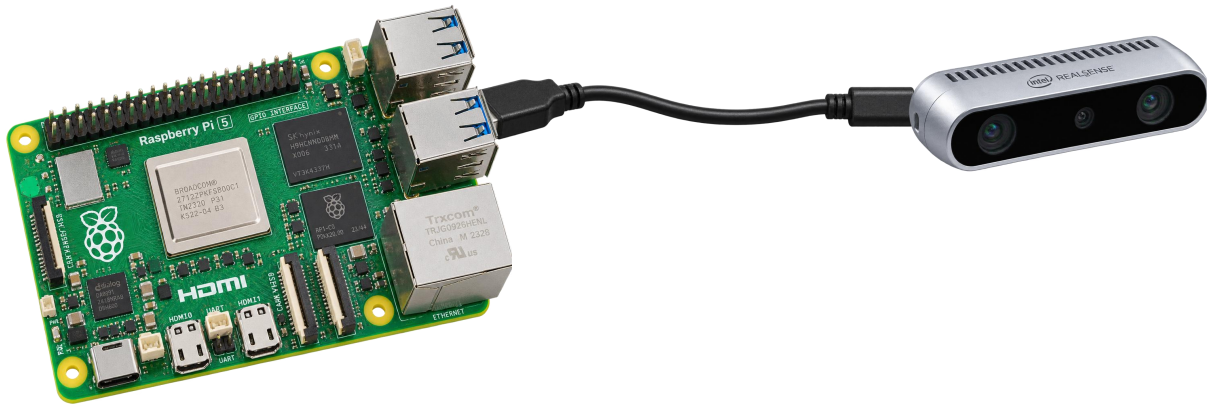
Test platform

Raspberry Pi 5

RealSense D435

Linux 6.12.96 PREEMPT_RT

librealsense 2.58.3



We look at the code

```cpp
#include <librealsense2/rs.hpp>
using namespace rs2;

int main() {
    pipeline p;
    p.start();

    while(true) {
        frameset frames = p.wait_for_frames();
        frame depth_frame = frames.get_depth_frame();
        frame color_frame = frames.get_color_frame();
        process(depth_frame, color_frame);
    }

    return 0;
}
```

We look at the code

```cpp
#include <librealsense2/rs.hpp>
using namespace rs2;

int main() {
    pipeline p;
    p.start(); // starts the camera: depth + color

    while(true) {
        frameset frames = p.wait_for_frames();
        frame depth_frame = frames.get_depth_frame();
        frame color_frame = frames.get_color_frame();
        process(depth_frame, color_frame);
    }

    return 0;
}
```

We look at the code

```
1   #include <librealsense2/rs.hpp>
2   using namespace rs2;
3
4   int main() {
5       pipeline p;
6       p.start();
7
8       while(true) {
9           frameset frames = p.wait_for_frames();
10          frame depth_frame = frames.get_depth_frame();
11          frame color_frame = frames.get_color_frame();
12          process(depth_frame, color_frame);
13      }
14
15      return 0;
16  }
```

starts the camera: depth + color

periodic task · every 33.3 ms

We look at the code

```cpp
#include <librealsense2/rs.hpp>
using namespace rs2;

int main() {
    pipeline p;
    p.start();

    while(true) {
        frameset frames = p.wait_for_frames();
        frame depth_frame = frames.get_depth_frame();
        frame color_frame = frames.get_color_frame();
        process(depth_frame, color_frame);
    }

    return 0;
}
```

starts the camera: depth + color

periodic task · every 33.3 ms

blocks here until a new frameset arrives

We look at the code

```
1   #include <librealsense2/rs.hpp>
2   using namespace rs2;
3
4   int main() {
5       pipeline p;
6       p.start();
7
8       while(true) {
9           frameset frames = p.wait_for_frames();
10          frame depth_frame = frames.get_depth_frame();
11          frame color_frame = frames.get_color_frame();
12          process(depth_frame, color_frame);
13      }
14
15      return 0;
16  }
```

starts the camera: depth + color

periodic task · every 33.3 ms

blocks here until a new frameset arrives

your work, once per frameset

But when we look at the threads behind the scenes...

```
Every 0,1s: ps -eL -o pid,tid,comm,cls,pri,rtprio,psr | grep -E "getframe"

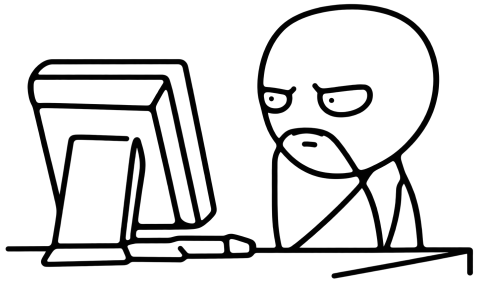
2635653 2635653 getframe         TS  19      -  10
2635653 2635723 getframe         TS  19      -  12
2635653 2635704 getframe         TS  19      -  18
2635653 2635714 getframe         TS  19      -  18
2635653 2635717 getframe         TS  19      -  18
2635653 2635718 getframe         TS  19      -  18
2635653 2635719 getframe         TS  19      -  18
2635653 2635720 getframe         TS  19      -  18
2635653 2635721 getframe         TS  19      -  19
2635653 2635716 getframe         TS  19      -   9
2635653 2635722 getframe         TS  19      -   9
```

But when we look at the threads behind the scenes...

```
Every 0,1s: ps -eL -o pid,tid,comm,cls,pri,rtprio,psr | grep -E "getframe"

2635653 2635653 getframe         TS  19      -  10
2635653 2635723 getframe         TS  19      -  12
2635653 2635704 getframe         TS  19      -  18
2635653 2635714 getframe         TS  19      -  18
2635653 2635717 getframe         TS  19      -  18
2635653 2635718 getframe         TS  19      -  18
2635653 2635719 getframe         TS  19      -  18
2635653 2635720 getframe         TS  19      -  18
2635653 2635721 getframe         TS  19      -  19
2635653 2635716 getframe         TS  19      -   9
2635653 2635722 getframe         TS  19      -   9
```

11 threads!!



But when we look at the threads behind the scenes...

```
Every 0,1s: ps -eL -o pid,tid,comm,cls,pri,rtprio,psr | grep -E "getframe"

2635653 2635653 getframe         TS  19      -  10
2635653 2635723 getframe         TS  19      -  12
2635653 2635704 getframe         TS  19      -  18
2635653 2635714 getframe         TS  19      -  18
2635653 2635717 getframe         TS  19      -  18
2635653 2635718 getframe         TS  19      -  18
2635653 2635719 getframe         TS  19      -  18
2635653 2635720 getframe         TS  19      -  18
2635653 2635721 getframe         TS  19      -  19
2635653 2635716 getframe         TS  19      -   9
2635653 2635722 getframe         TS  19      -   9
```

11 threads!!

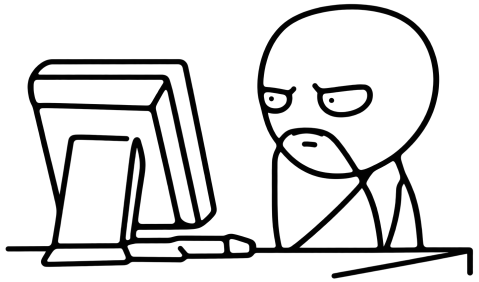
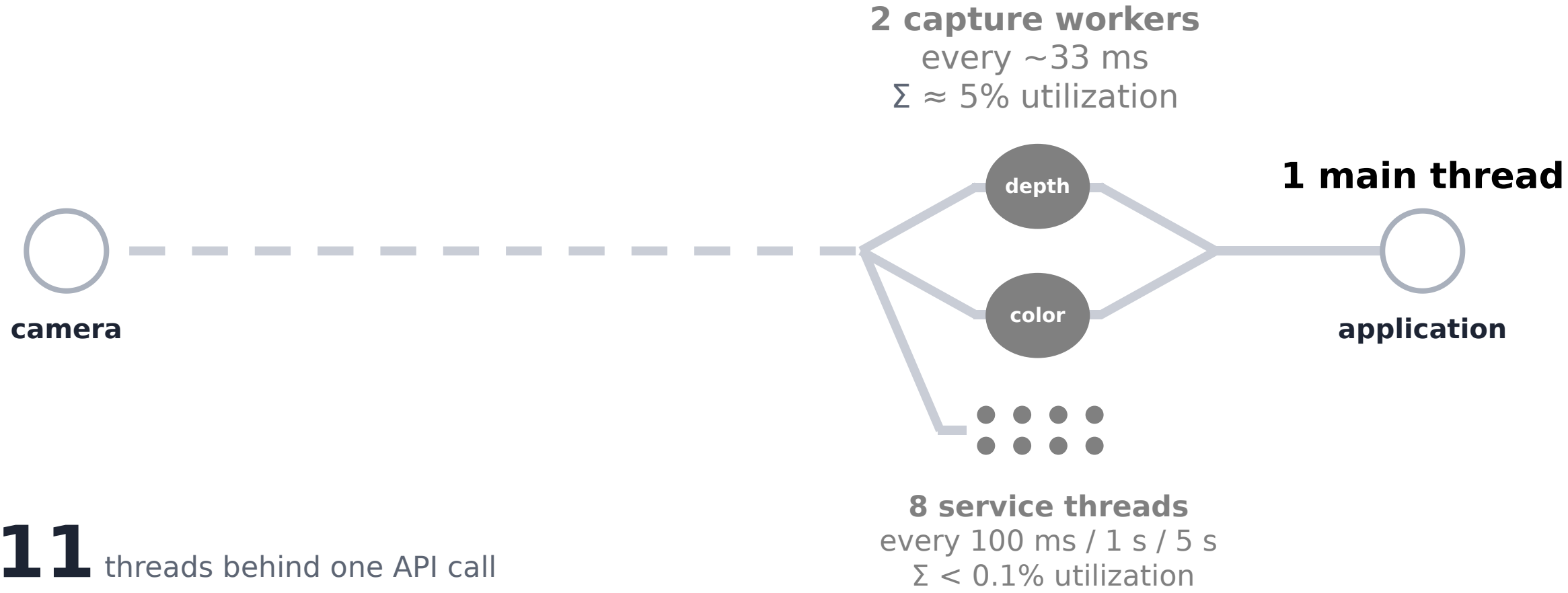


Figure out the hidden information and configure it properly to be real-time!

Where did this depth frame go?



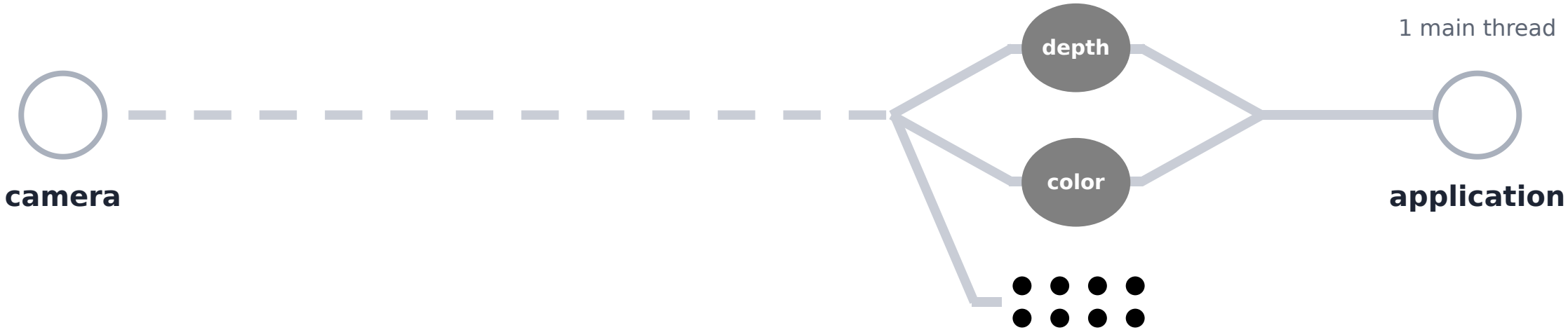
Inside the call: 11 threads, many rates



11 threads behind one API call

Inside the call: 11 threads, many rates

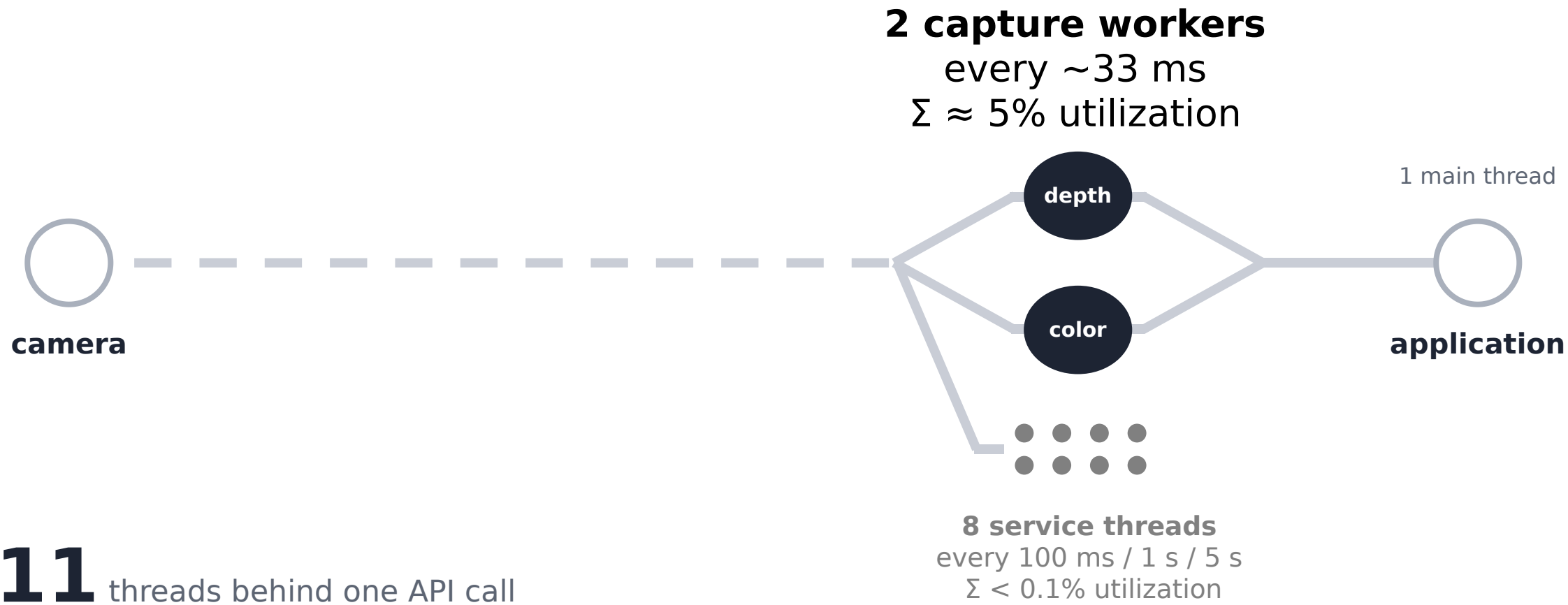
2 capture workers
every ~33 ms
Σ ≈ 5% utilization



11 threads behind one API call

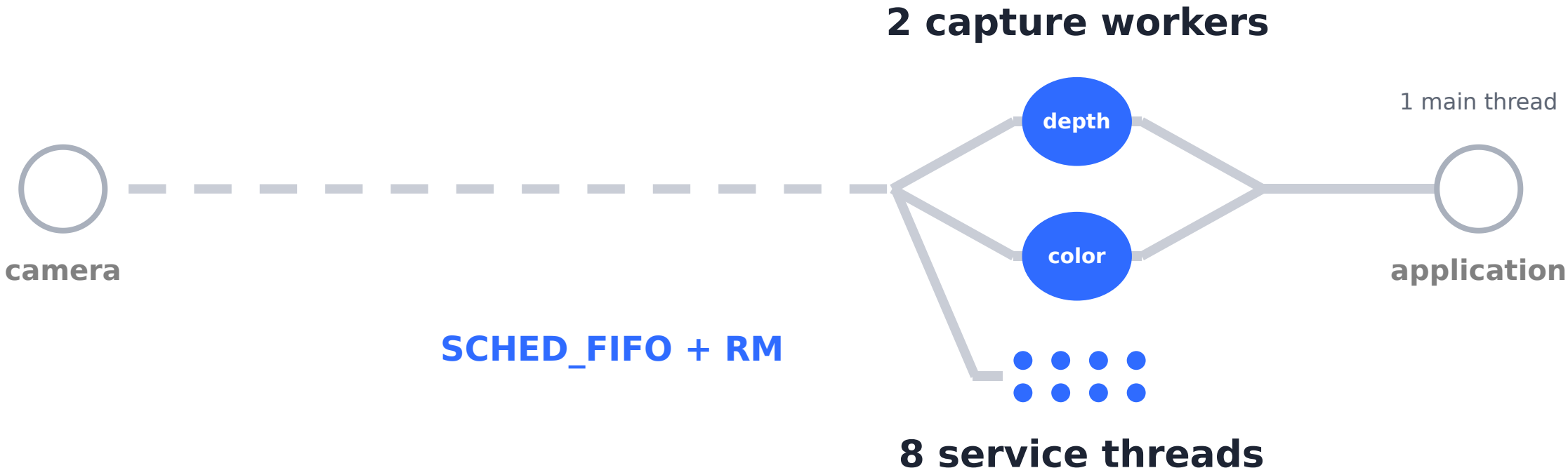
8 service threads
every 100 ms / 1 s / 5 s
Σ < 0.1% utilization

Inside the call: 11 threads, many rates

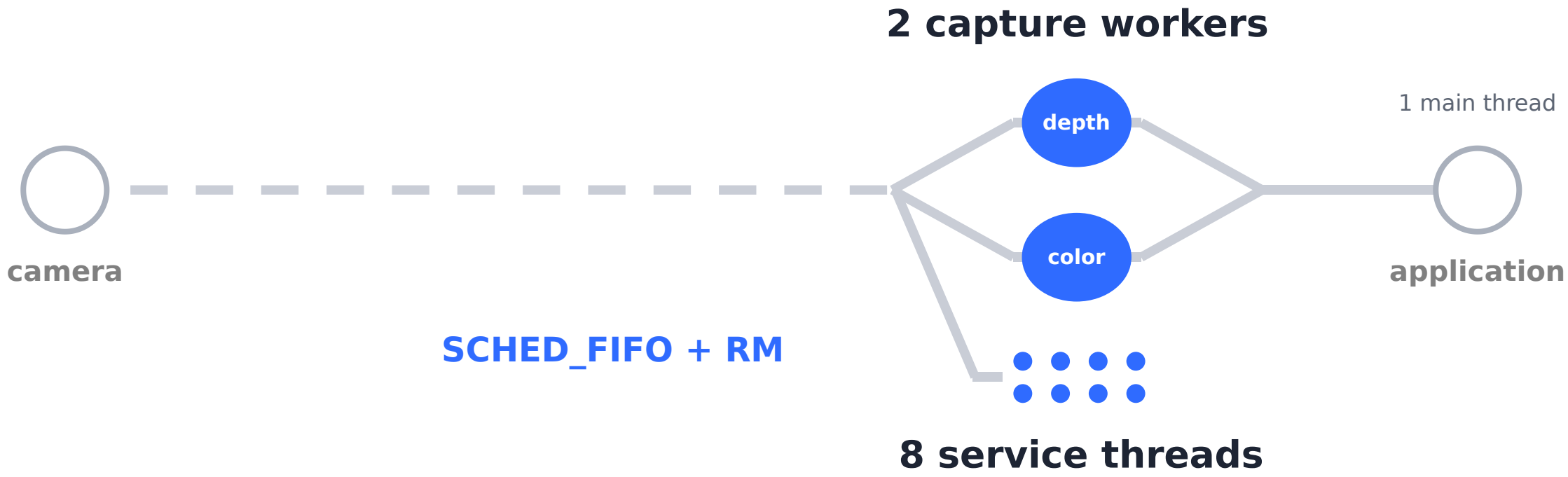


11 threads behind one API call

Make them all real-time

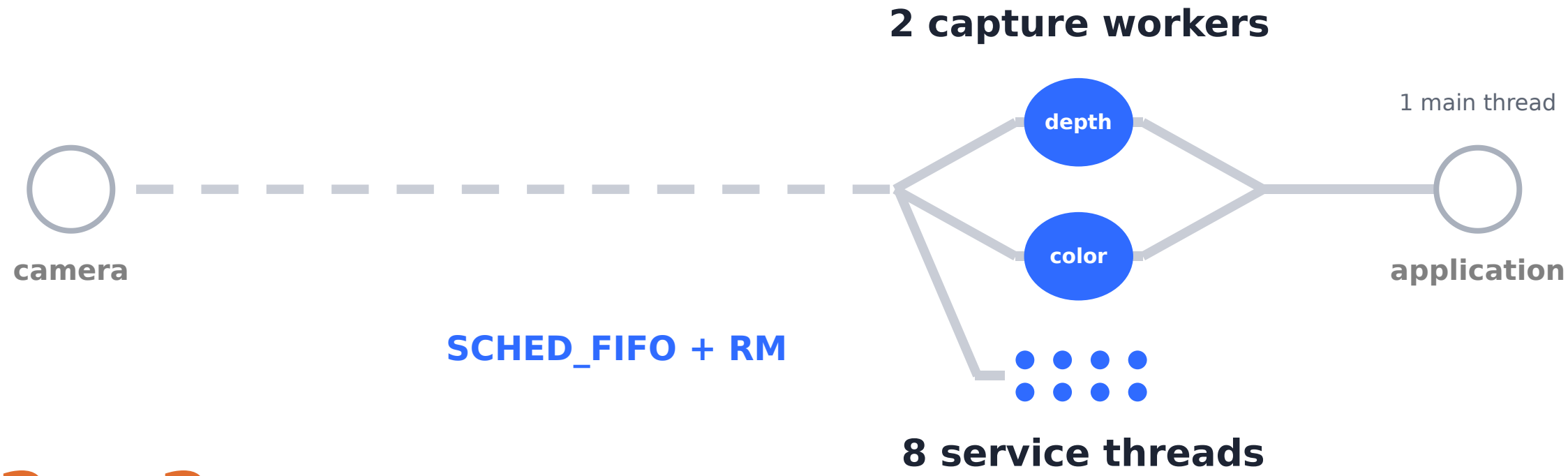


Make them all real-time



D435 · 30 FPS · Depth + Color · 4 background tasks · 3 × 30 s

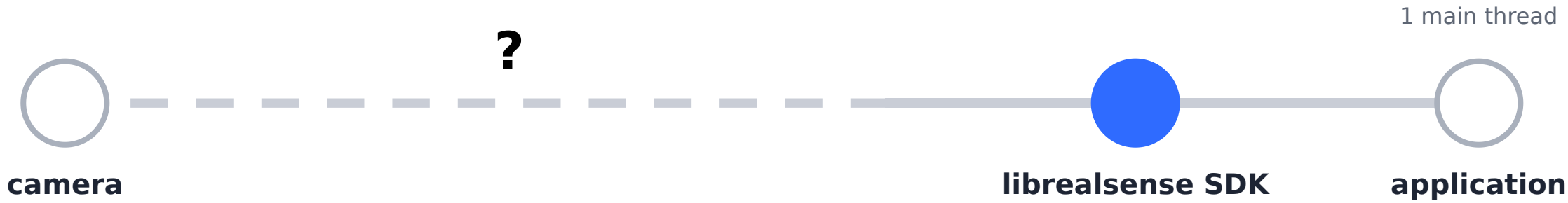
Make them all real-time … frame is still not fresh.



2 stale **2** skipped

D435 · 30 FPS · Depth + Color · 4 background tasks · 3 × 30 s

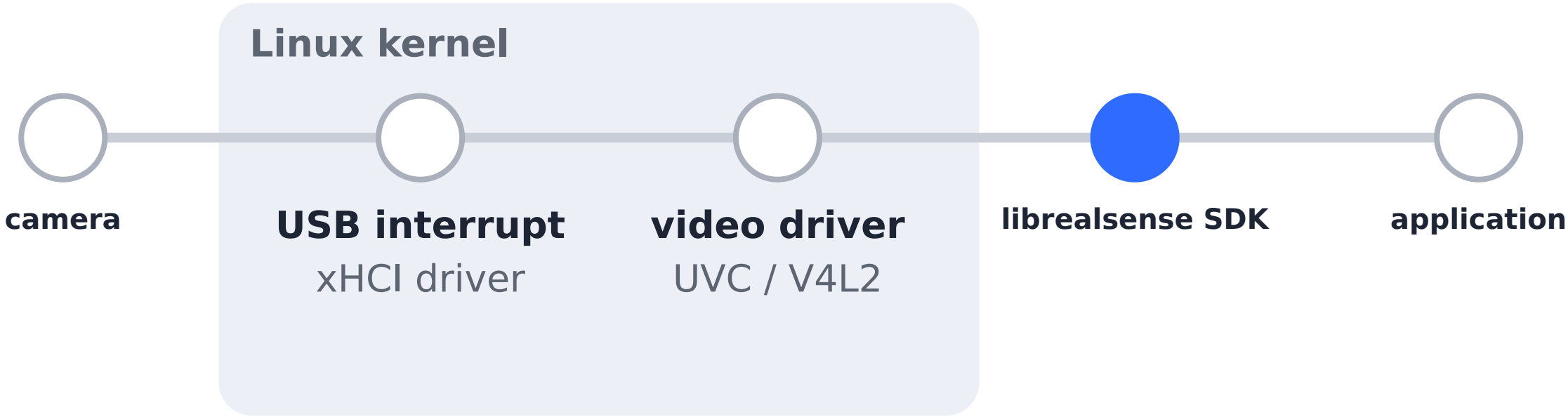
Make them all real-time … frame is still not fresh.



2 stale **2** skipped

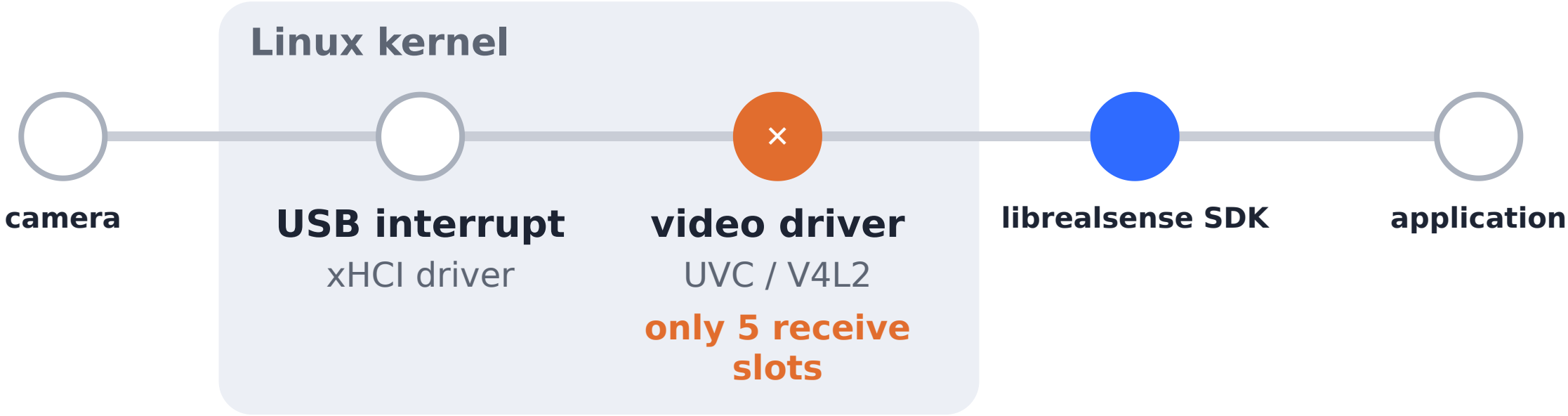
D435 · 30 FPS · Depth + Color · 4 background tasks · 3 × 30 s

Kernel services: the frame broke before user space saw it

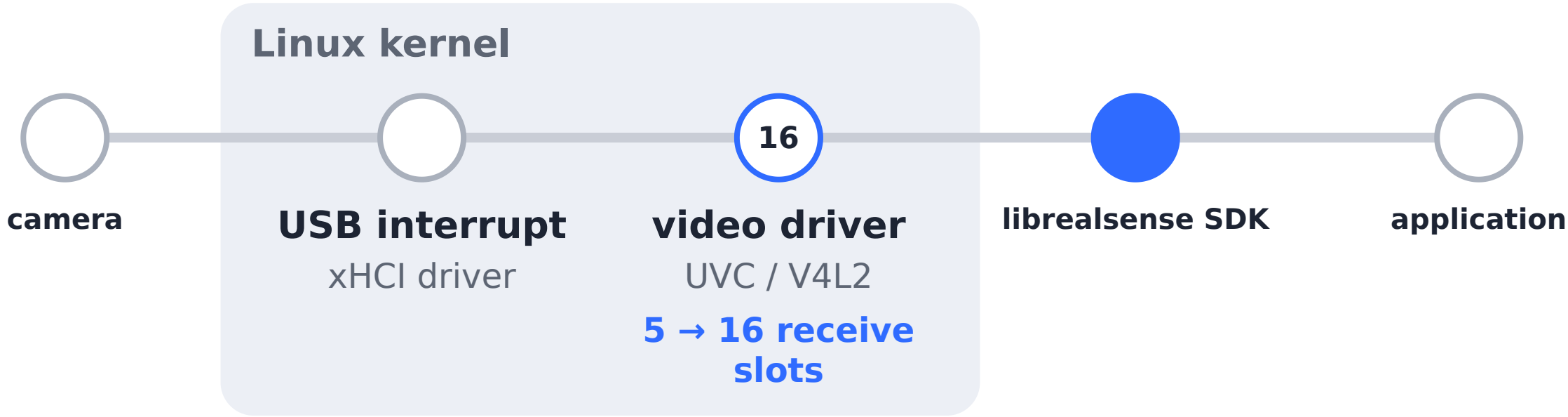


Kernel services: the frame broke before user space saw it

depth buffer closed incomplete,
no error reported



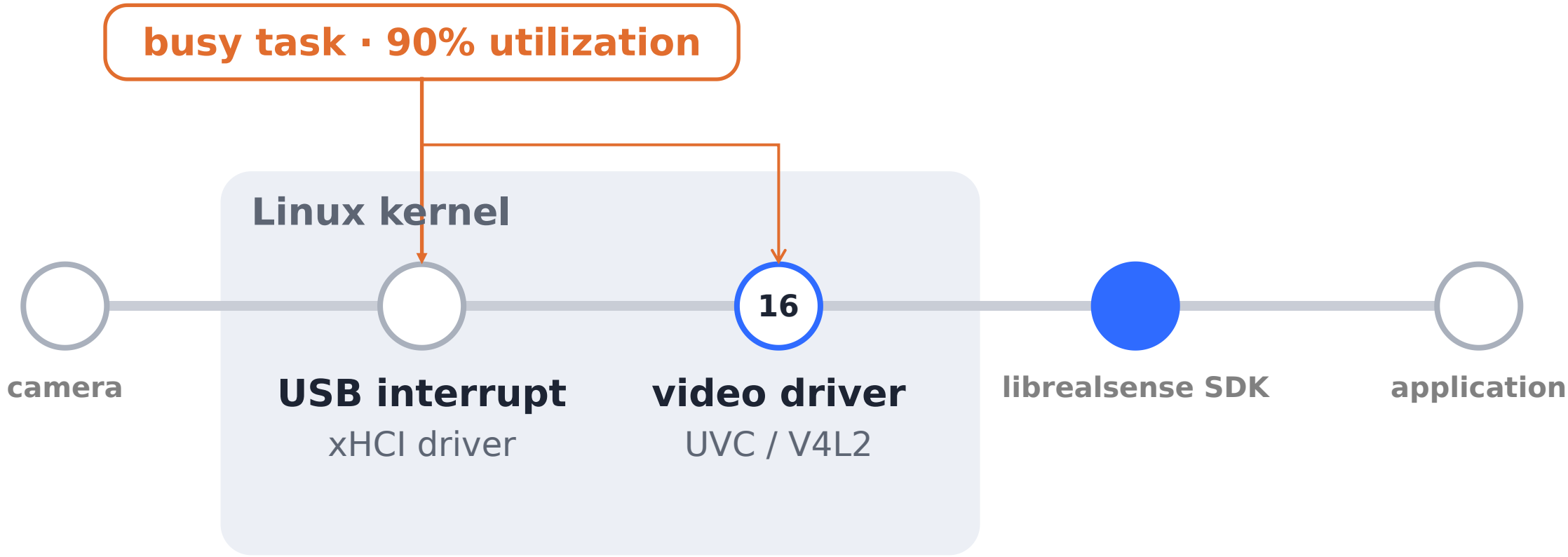
More receive slots: stale and skipped frames disappear



2 stale **2** skipped → **0**

UVC_URBS = 5 UVC_URBS = 16

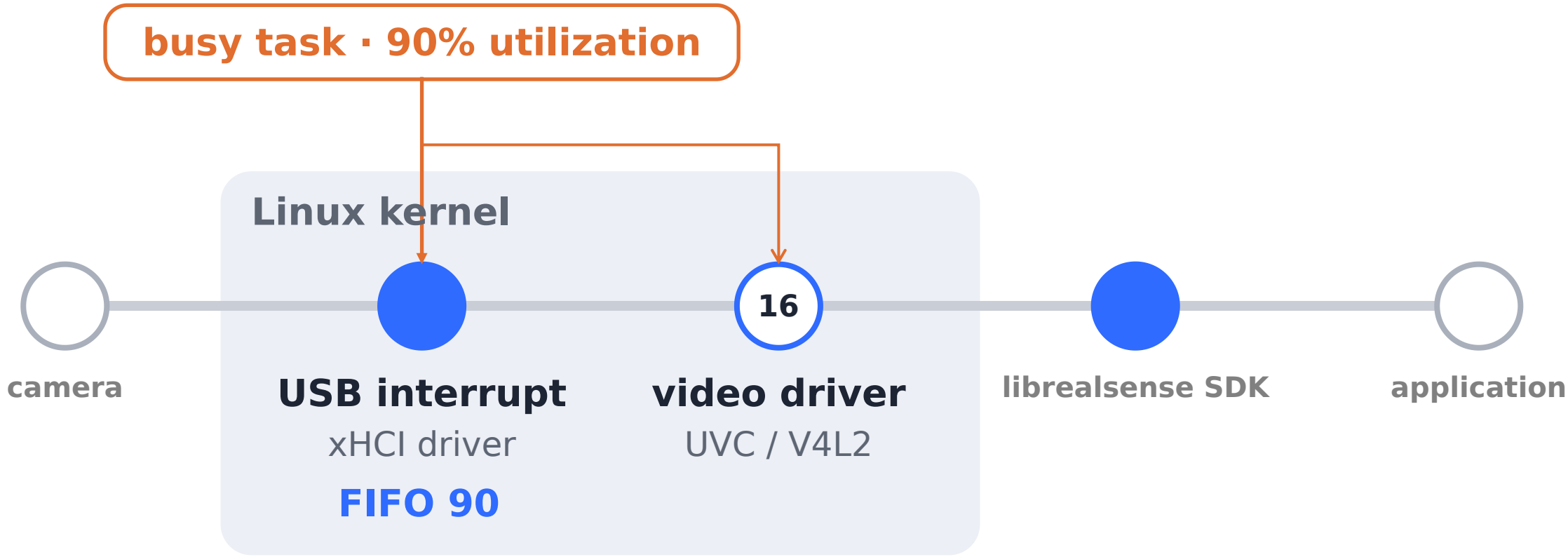
Squeeze the core that serves USB: 69% of framesets are not fresh



69% non-fresh framesets

kernel threads at
normal priority

Protect the USB interrupt: most failures go

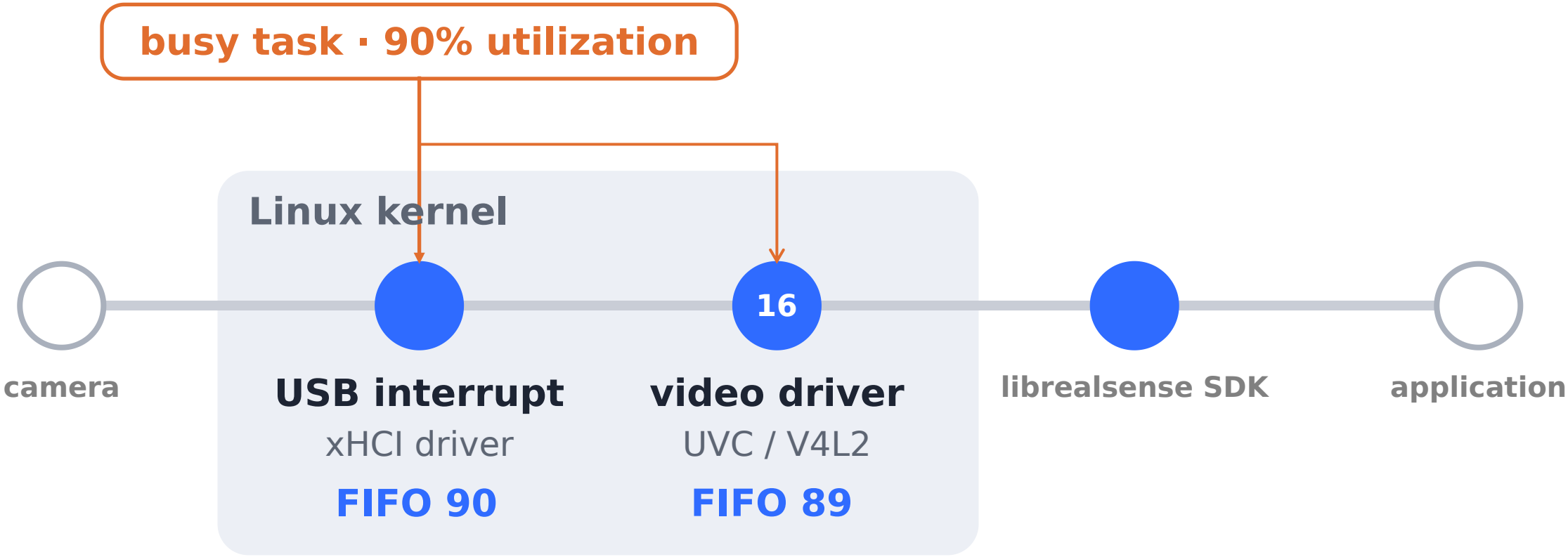


69%
normal priority

→ **5.5%** non-fresh framesets

Protect
kernel xHCI IRQ threads

Protect the video driver

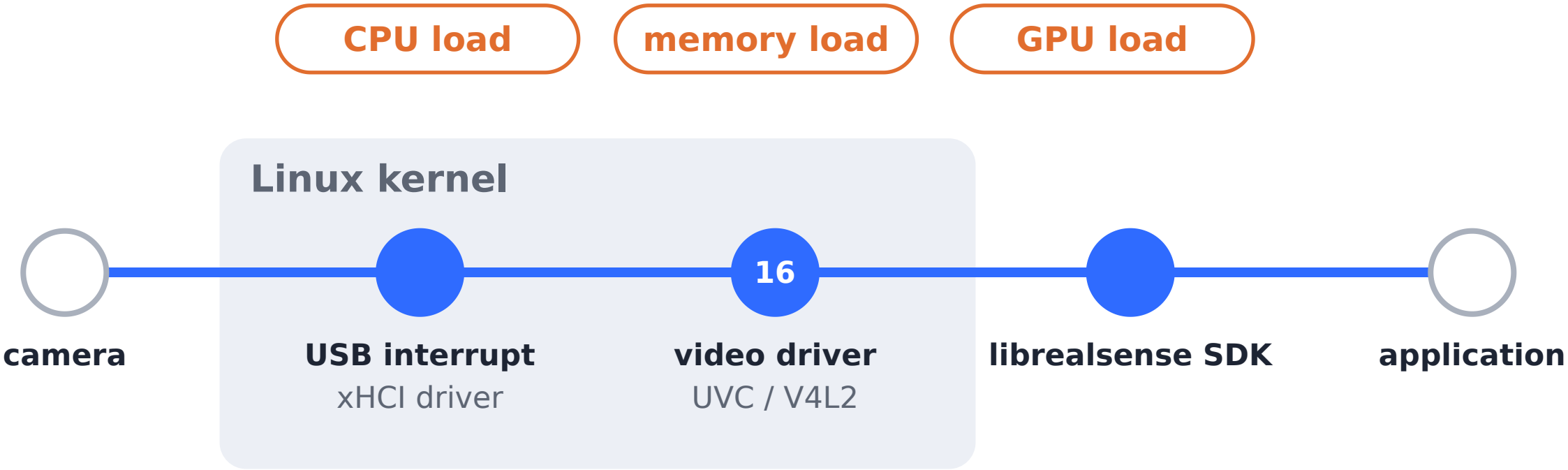


69% normal priority → **5.5%** USB interrupt + video driver → **0.15%** non-fresh framesets

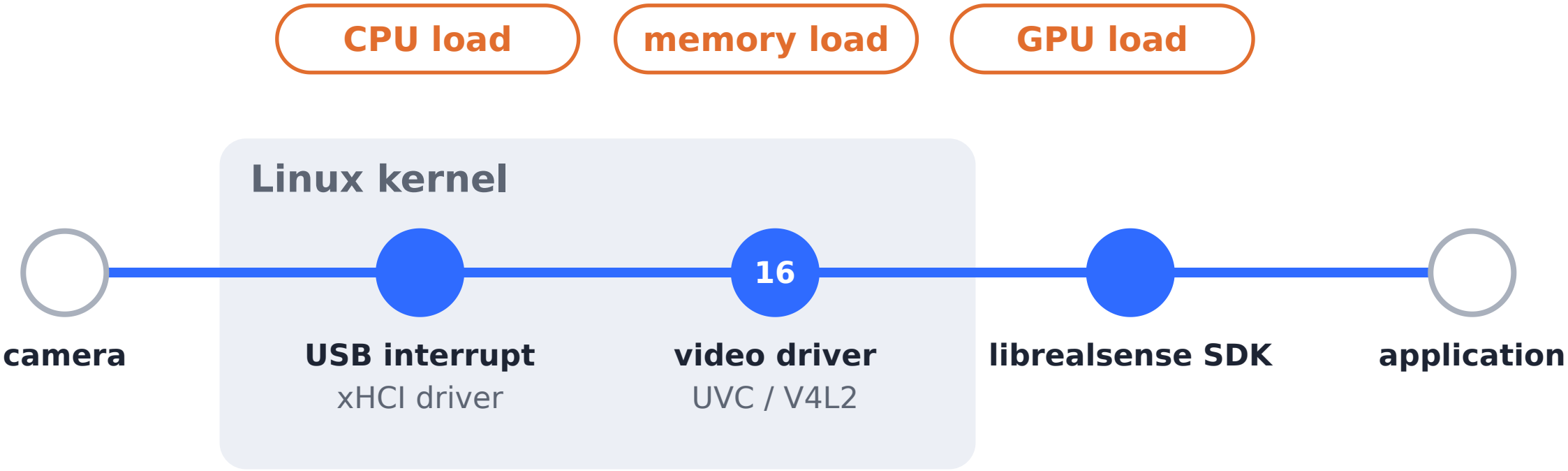
Whole path protected. Now add realistic interference.



Whole path protected. Now add realistic interference.

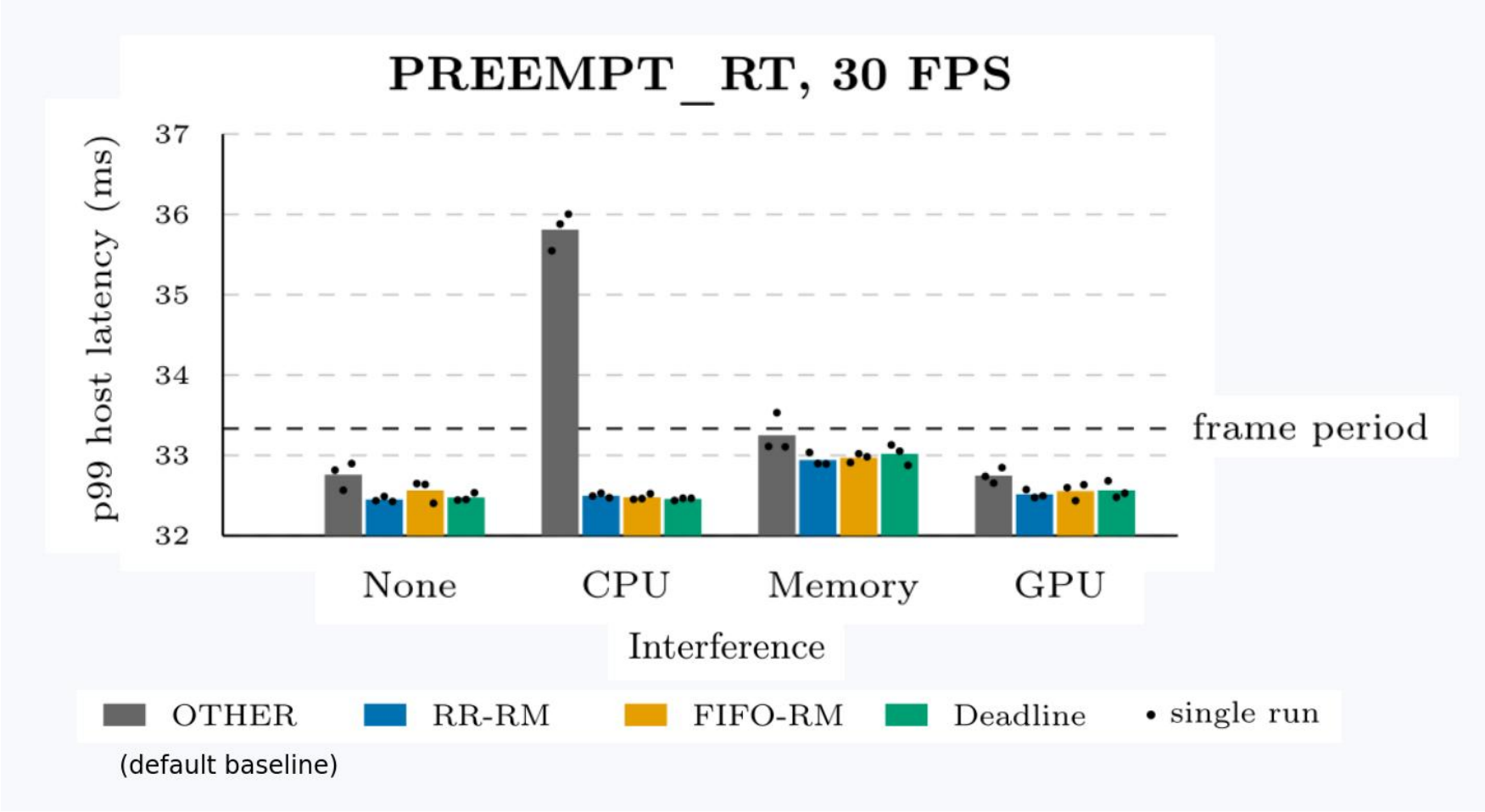


Whole path protected. Now add realistic interference.



30 / 60 FPS · default vs. 3 coordinated policies · non-RT vs. PREEMPT_RT · 192 runs

Timely: the protected path stays near the frame period



Fresh: with PREEMPT_RT + coordinated scheduling, no reused or skipped frames

159

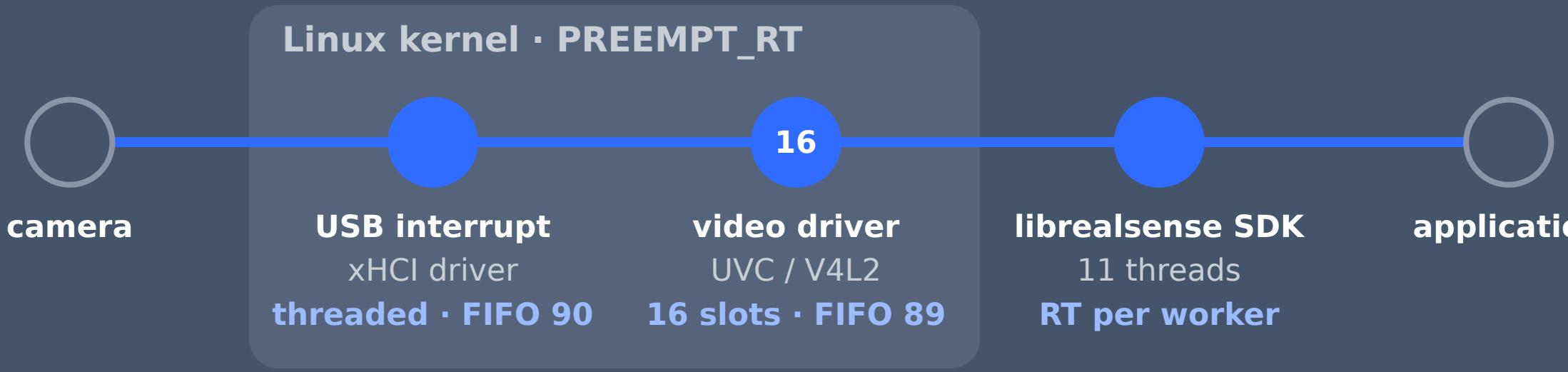
non-RT kernel
all policies · 96 runs

0

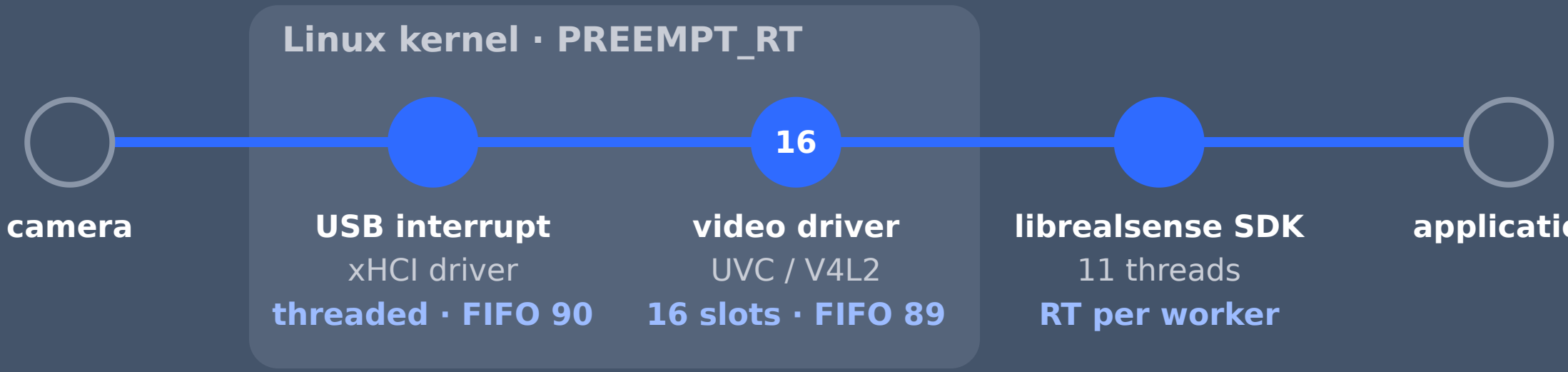
PREEMPT_RT + coordinated policies
72 runs

1
PREEMPT_RT + default policy
24 runs

Real-time is a property of the whole path



Real-time is a property of the whole path



Next:
- Schedulability analysis of the worker graph
- Evaluation across hosts, backends, and camera configurations
- Timestamp-based frameset data-age analysis
- Multi-camera resource scaling

Questions?



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

Yuwen Shen[1,2,3], Constantin Scholz[1,3], Bram Vanderborght[1,3], Wouter Joosen[2] and Antonio Paolillo[1(✉)]

[1] Vrije Universiteit Brussel, Brussels, Belgium
{yuwen.shen,constantin.florian.scholz,bram.vanderborght,
antonio.paolillo}@vub.be
[2] KU Leuven, Leuven, Belgium
wouter.joosen@kuleuven.be
[3] 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,