SILEXICA

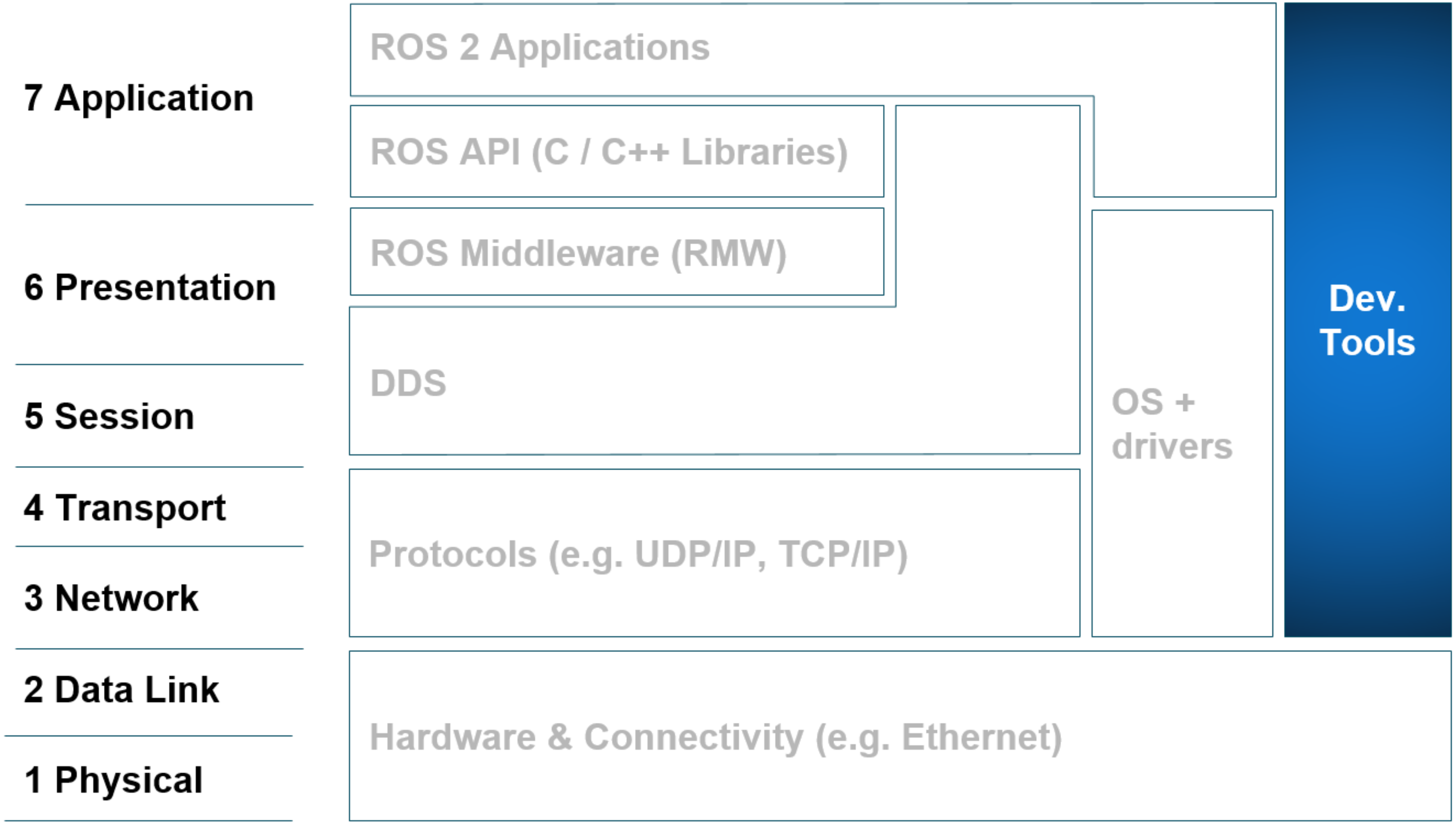
Doing Real-Time with ROS 2

An Introduction to Tooling

Benjamin Goldschmidt, Stefan Schürmans, Florian Walbroel

ROSCon 2019 Workshop
30.10.2019, Macao

Overview



7 Application

6 Presentation

5 Session

4 Transport

3 Network

2 Data Link

1 Physical

ROS 2 Applications

ROS API (C / C++ Libraries)

ROS Middleware (RMW)

DDS

Protocols (e.g. UDP/IP, TCP/IP)

OS + drivers

Dev. Tools

Hardware & Connectivity (e.g. Ethernet)

Motivation

“rosout/printf and hope for the best”

“By using rosbag to replay the same input”

“[...] avoid them where possible”

“If your software has real-time constraints, how do you debug it?”

“[...] stream debug data out to a non real-time thread for inspection.”

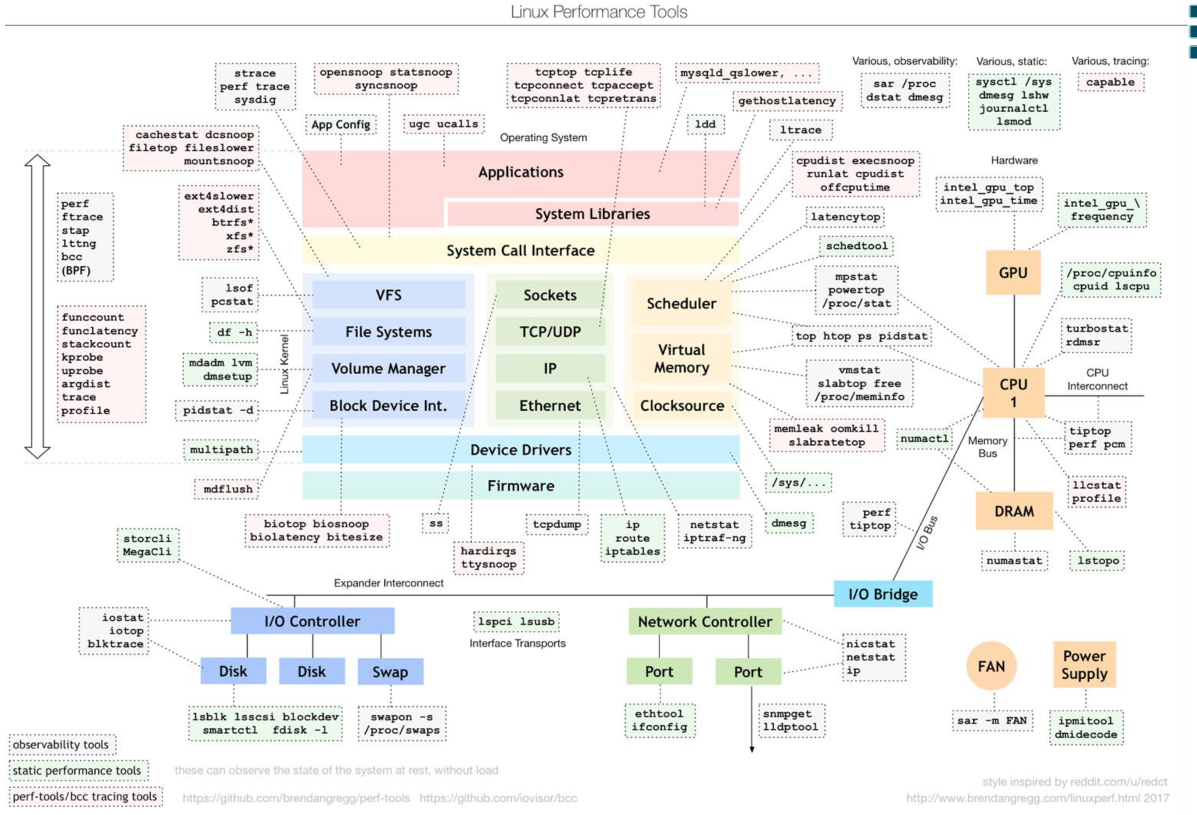
“Attach gdb on crash [...]”

“Minimise the hard real-time part [...]”

(https://discourse.ros.org/t/questionnaire-how-do-you-debug-your-software-in-ros/1988)
24 answers

Introduction

- In this talk, basic approaches for proofing/evaluating real-time ROS2 applications will be introduced
- For each approach, a small set of (Linux) tools will be presented
- Depending on the target platform, OS, application, and use-case, there may be more fitting tools



Selection of Linux performance tools[1]

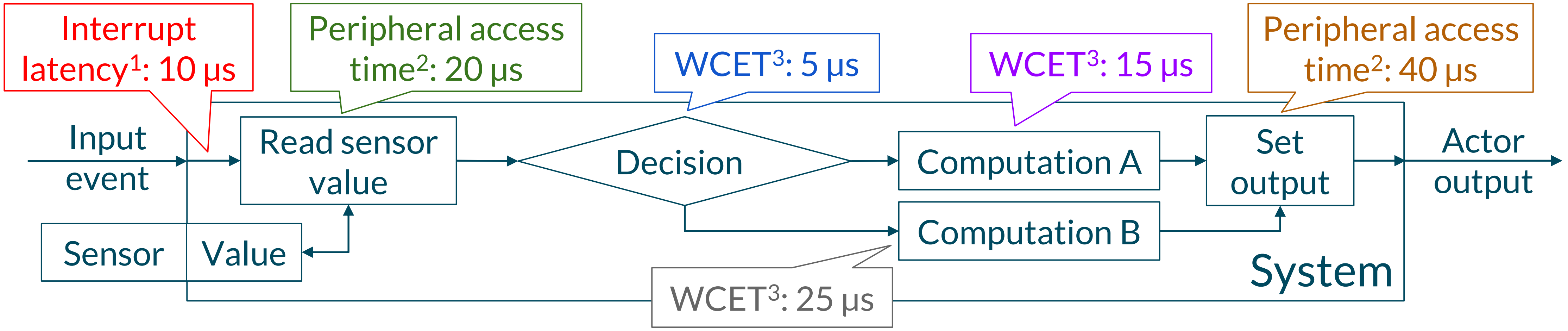
---

[1] (©2019 Brendan Gregg, source: http://www.brendangregg.com/linuxperf.html)

PROOFING ROS2 REAL-TIME APPLICATIONS

General Approach

- Formally prove that a system always (!) reacts to an incoming input...
  - ...with the functionally correct output (functional correctness)
  - ...within a certain amount of time (hard real-time)
- (Simple) Example:
  - Sum up worst-case execution times (WCETs) and latencies in critical event chain

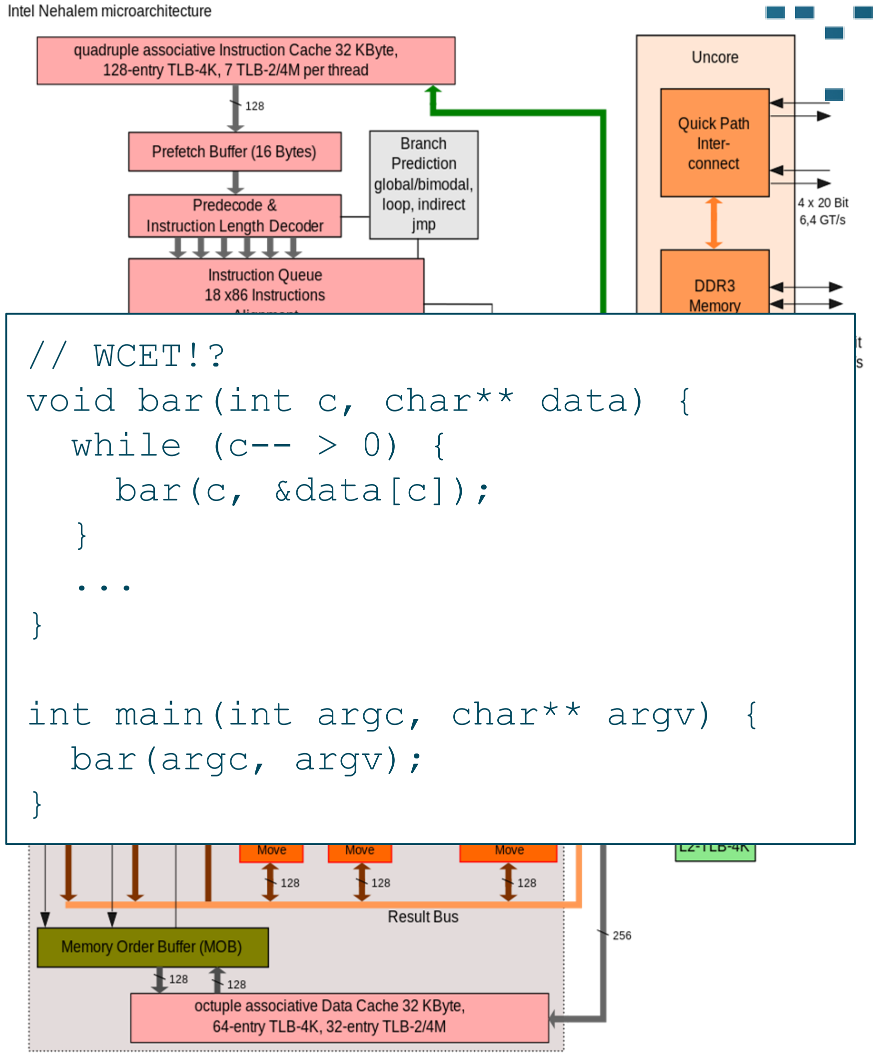


Max. Latency: 10 µs + 20 µs + 5 µs + max(15 µs, 25 µs) + 40 µs = **100 µs**

[1] (e.g. from µC datasheet (DS)) [2] (e.g. from peripheral DS) [3] (e.g. by cycle counting using assembly + µC DS)

Challenges

- System:
  - Instruction timings depend on:
    - Pipeline state, branch predictor, etc.
  - Memory access times depend on:
    - Cache/Page/Swap state, bus contention, etc.
  - OS task execution depends on OS effects:
    - Scheduler decisions, other tasks, etc.
- Limitations of static analysis:
  - Input dependence
  - Loop iteration counts / recursion bounds
  - Pointer analysis



```
// WCET!?
void bar(int c, char** data) {
  while (c-- > 0) {
    bar(c, &data[c]);
  }
  ...
}

int main(int argc, char** argv) {
  bar(argc, argv);
}
```

Intel Nehalem Microarchitecture (2010)[1]

[1] (©2008 Appaloosa, CC BY-SA 3.0, source: https://de.wikipedia.org/wiki/Intel-Nehalem-Mikroarchitektur)

Tools

- Academic:
  - Static analysis: “A Unified WCET Analysis Framework for Multi-core Platforms” (http://www.comp.nus.edu.sg/~rpembed/mxchronos/mxc-timing.pdf)
  - Mixed static/measurement: “An End-To-End Toolchain […]” (https://www4.cs.fau.de/Publications/2017/sieh_17_isorc.pdf)
- Commercial:
  - Static analysis: AbsInt aiT (https://www.absint.com/ait)
  - Mixed static/measurement: Rapita RapiTime (https://www.rapitasystems.com/products/rapitime)
- Problem: Limited/No support for typical ROS2 target platforms (based on x86_64 or ARMv8-A processors)

→ Applicability to ROS2 systems!?

EVALUATING ROS2 REAL-TIME APPLICATIONS

General Approach

- Observe application/system behavior to evaluate real-timeness:
    - Where: **Inside** vs. outside **the system** (trade-off: intrusiveness vs. insights)
    - How: **Source-**, IR, binary-, **system/kernel-instrumentation**
        - Sampling vs. **exhaustive tracing** (trade-off: overhead vs. completeness)

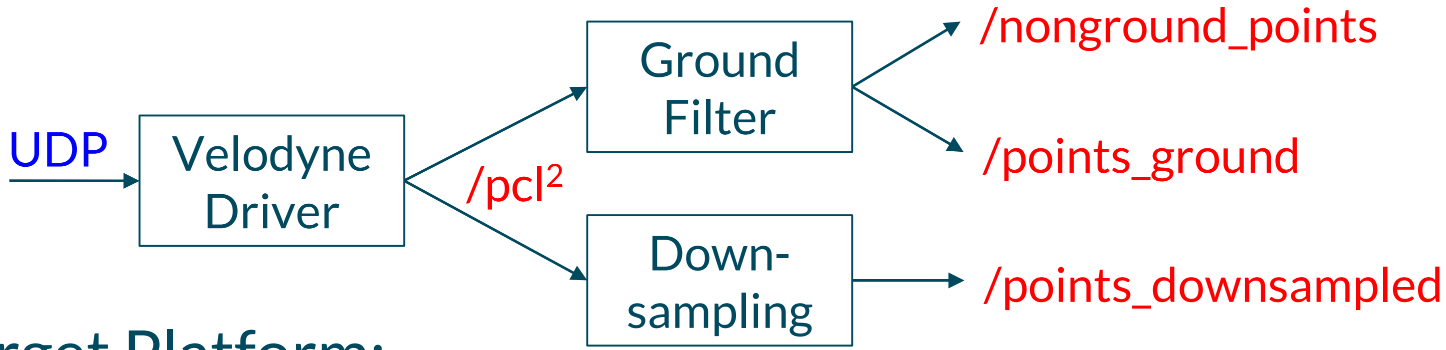
  → Limitation: Absence of constraint violations in tests does not prove real-timeness

- General recommendations:
    - Consider measurement overhead
    - Measure many times and many scenarios (no proof, but increases confidence)
    - Measure additional system metrics (CPU load, etc.) (helps when debugging)
    - Add stress tests (CPU, memory, I/O) to evaluate system under extreme conditions

Test Setup

Scenario:

- Autoware.Auto[1] autonomous driving framework (ROS2-based, open source) 3D perception pipeline (hash: ee99e9b8)
- ROS 2 Dashing built from source (with instrumentation)


UDP

Velodyne Driver

/pcl[2]

Ground Filter

Down-sampling

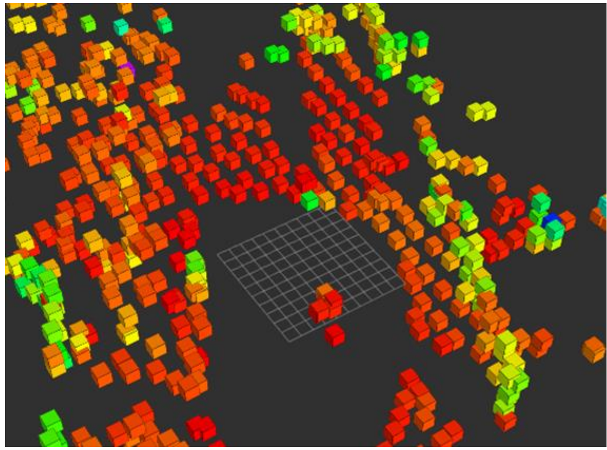
/nonground_points

/points_ground

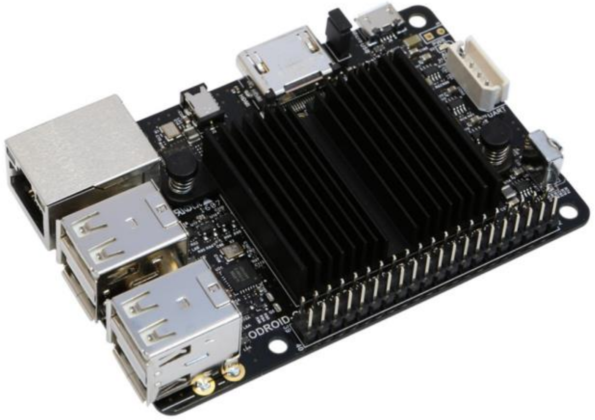
/points_downsampled

Target Platform:

- Odroid-C2:
  - ARM A53 @ 1.5 GHz (aarch64)
  - 2 GB DDR3 RAM
  - Ubuntu 18.04, Kernel v3.16



Sample RViz output for the 3D perception pipeline



Odroid C2 board[3]

---

[1] (https://gitlab.com/autowarefoundation/autoware.auto)

[2] (original name: /test_velodyne_node_cloud_front)

[3] (©2016 Peba, CC BY-SA 4.0, source: https://de.wikipedia.org/wiki/ODROID)

Tools/Techniques: Overview

Tool/Technique	Type
GDB	Source code debugger
ROS Performance Test	Communication benchmark
ROS Tracetools	Source/system instrumentation
printf/rosout	Source instrumentation
rosbag2	System (here: ROS2) instrumentation
SLX Performance Testing Platform	Commercial ROS2 testing platform
---	---
perf	System instrumentation
LD_PRELOAD[1]	Binary interception
Stress Testing Tools[1]	Benchmarks/Generators

[1] (not covered here, but available on demand in backup slides)

GNU Debugger "gdb"

https://www.gnu.org/software/gdb/documentation/

- Type: Source code debugging
  - ROS nodes can be started inside gdb directly:
    `ros2 run --prefix 'gdb -ex run --args'`
  - Multiple nodes can be debugged “in parallel” (in multiple terminals)

- Assessment:
  - (+) Simple usage
  - (+) Internal state of ROS nodes can be examined
  - (-) Halting execution (interactive / breakpoint) conflicts with real-time behavior
  - (-) How to efficiently handle multiple sessions in parallel?!

```
#0  recvmsg ()
#1  boost::...::socket_ops::recvfrom (
      ec=..., flags=0, count=1, ...)
#2  boost::...::sync_recvfrom (
      ec=..., flags=0, count=1, ...)
...
#5  boost::...::receive_from<...> (
      ec=..., flags=0, ...)
#6  autoware::...::get_packet (
      socket=..., pkt=..., ...)
    at udp_driver_node.hpp:154
#7  autoware::drivers::...::run (
      this=..., max_iterations=0)
    at udp_driver_node.hpp:102
#8  main (argc=3, argv=0x7ffc46355598)
    at velodyne_cloud_node_main.cpp:38
```

Backtrace of the Velodyne Driver node

ROS Performance Test

https://github.com/ApexAI/performance_test

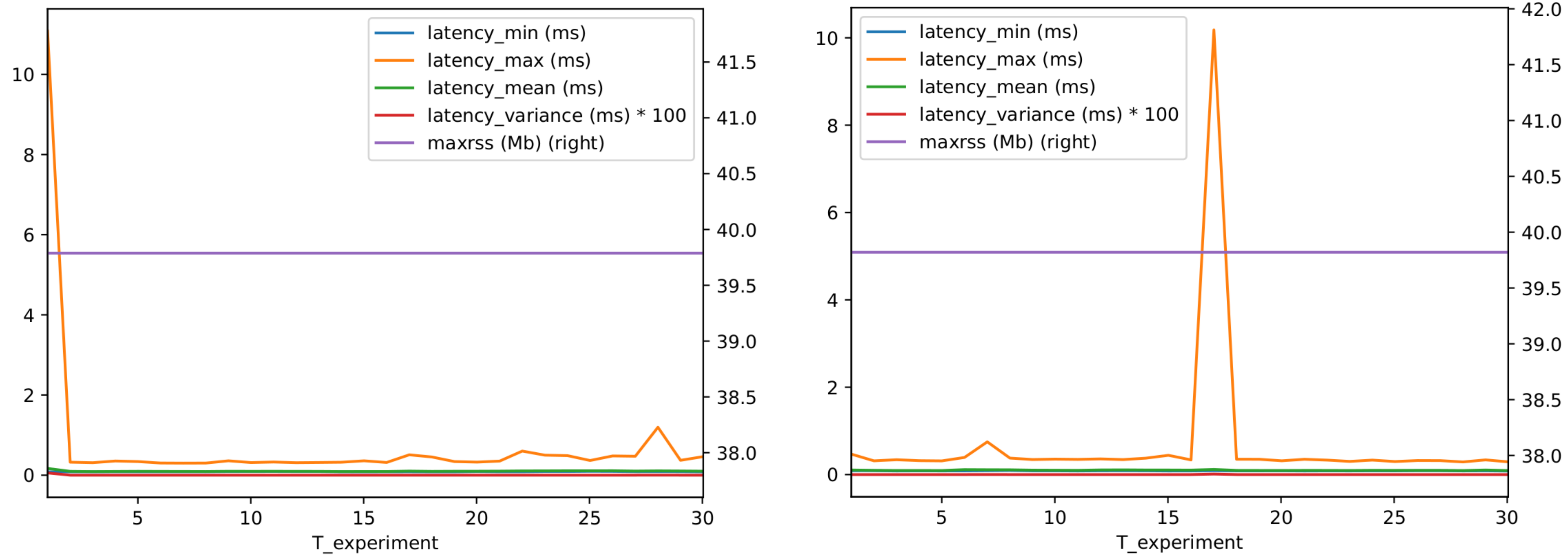
- Type: Benchmark (source instrumentation)
  - Benchmark for ROS2 communication means (e.g. FastRTPS, Connext DDS Micro, etc.)
  - Creates an artificial ROS2 graph, sends messages between nodes, and measures comm. performance/latency
- Assessment:
  - (+) Highly configurable
  - (+) Support for distributed ROS2 systems
  - (-) Limited ROS2 graph configurability (→ https://github.com/irobot-ros/ros2-performance)

```
$ ros2 run performance_test perf_test -h
Allowed options:
  --rate                 The publishing rate data
  --communication        Communication plugin to use
                          (ROS2, FastRTPS, ...)
  --reliable             Enable reliable QOS.
  --transient            Enable transient QOS.
  --keep_last            Enable keep last QOS.
  --history_depth        Set history depth QOS.
  --num_pub_threads      Max. num. of pub. threads.
  --num_sub_threads      Max. num. of sub. threads.
  --roundtrip_mode       Selects the round trip mode
                          (None, Main, Relay).
[...]
```

Shortened/Adapted command line options

ROS Performance Test

Results



Results for two sample runs[1] of the ROS performance test tool on the test platform

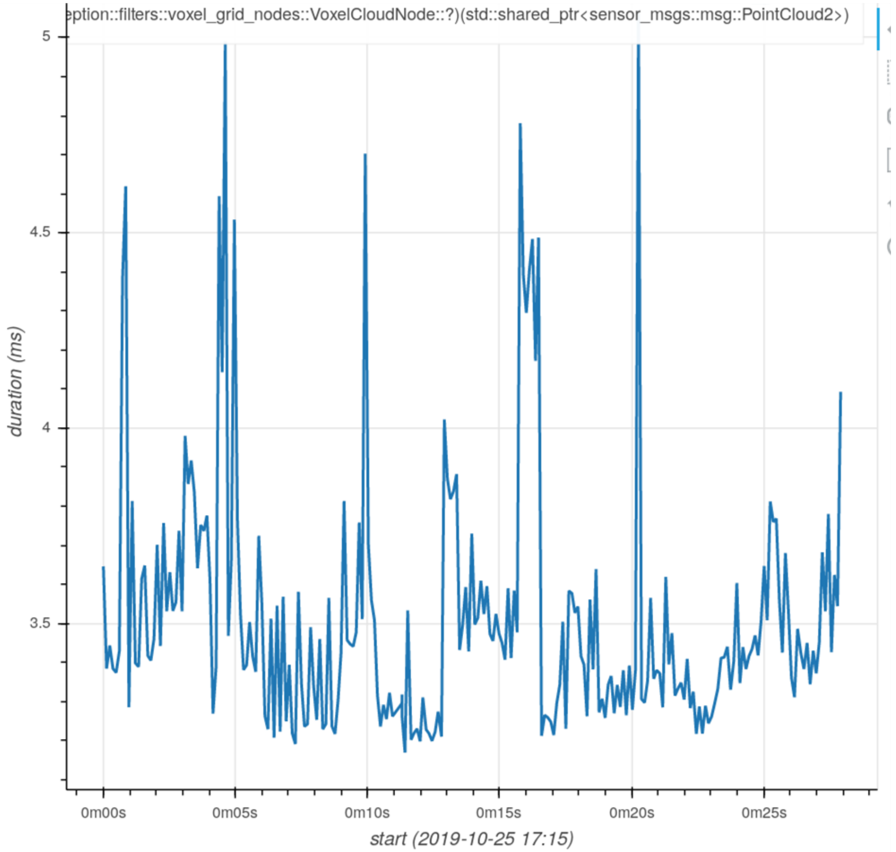
---

[1] (`-c ROS2 -l log -t Array1k --max_runtime 30`)

ROS Tracetools

https://micro-ros.github.io/docs/tutorials/advanced/tracing

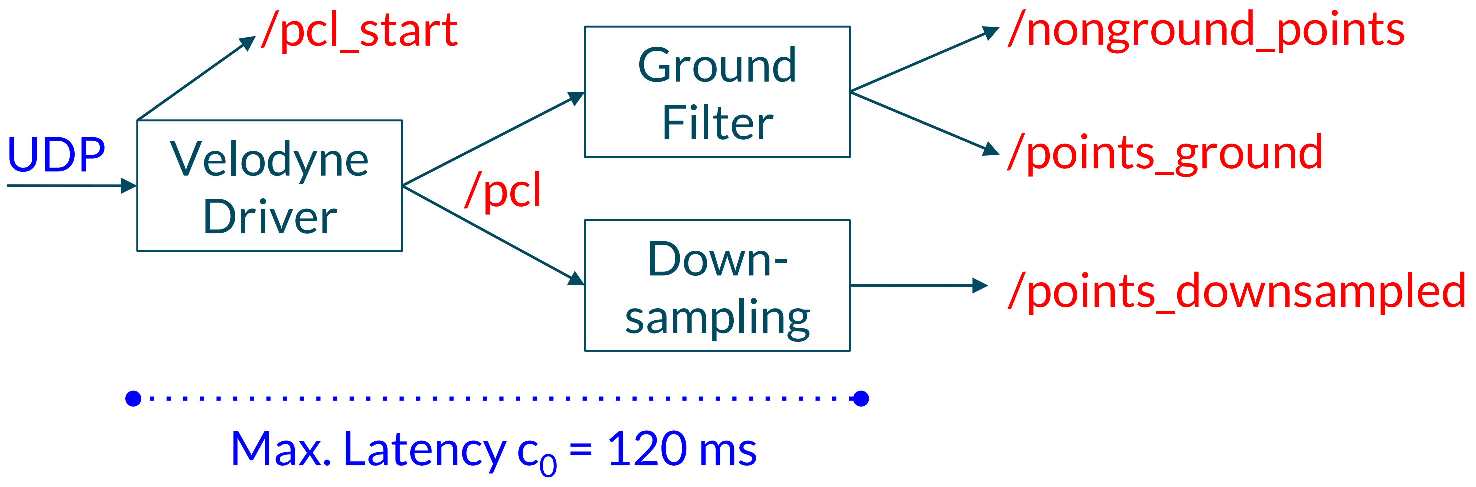
- Type: Source/System instrumentation
  - Custom tracepoints in ROS2 libraries (rcl, rclcpp) (included in ROS2 Eloquent)
  - Uses LTTng as backend to trace ROS2 core (e.g. callback durations, msg. send/received, etc.)
  - Results can be viewed with Trace Compass or parsed with Babeltrace/Jupyter Notebooks
- Assessment:
  - (+) Low-effort/-overhead tracing of the ROS2 core
  - (+) System internals can be traced in parallel
  - (-) Requires custom ROS2 source build (→ swappable packages are planned)



Callback durations over time for the topic `/pcl` in the Downsampling node

Test Setup (2)

Defining a Real-Time Constraint



UDP

Velodyne Driver

/pcl_start

/pcl

Ground Filter

Down-sampling

/nonground_points

/points_ground

/points_downsampled

Max. Latency c_0 = 120 ms

- UDP packets are sent out continuously during the rotation of the LIDAR:
  - Acquisition and transfer of a complete point cloud (PCL) takes ~100 ms (→ 10 Hz)
- Constraint:
  - Detect start of PCLs and signal using new topic `(/pcl_start`) (not recommended!)
  - Interval: `publish_start(/pcl_start)` → `publish_end(/points_downsampled)`
  - Max. Latency c_0: 100 ms (min. duration of PCL transfer) + 20 ms (arbitrarily selected)

rosout/printf

- Type: Source instrumentation
  - Log time and data at certain code locations:
    - `RCLCPP_INFO(...)`
    - `std::cerr + std::chrono`
    - ...
  - Parse log files and calculate constraints based on printed timestamps

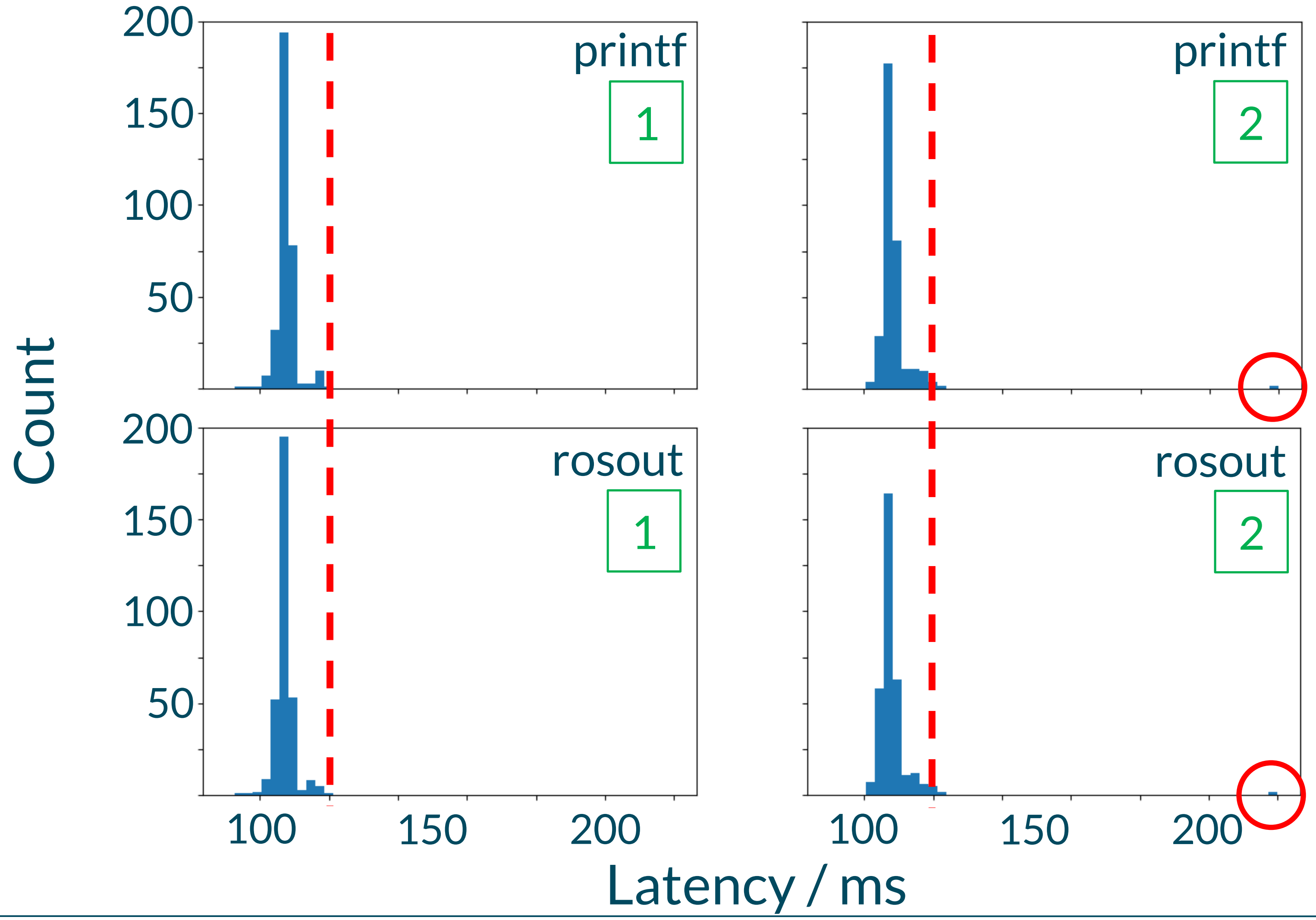
- Assessment:
  - (+) Simple and flexible
  - (-) High amount of manual work
  - (-) May affect performance (e.g. no inlining)

```
...
while(...) {
  PacketT pkt;
  (void) get_packet(pkt, m_udp_socket);

  if (first_packet_of_pcl) {
    RCLCPP_INFO(node_logger, "PCL_START");
    first_packet_of_pcl = false;
  }

  if (this->convert(pkt, output)) {
    first_packet_of_pcl = true;
    m_pub_ptr->publish(output);
  ...
```

```
...
[INFO 1571065761.831277989]
[vlp16_driver_node]:
  PCL_START (run() at udp_driver_node.hpp)
...
```

rosout/printf

Measuring Latency (Two Sample Runs)



rosbag2

https://github.com/ros2/rosbag2

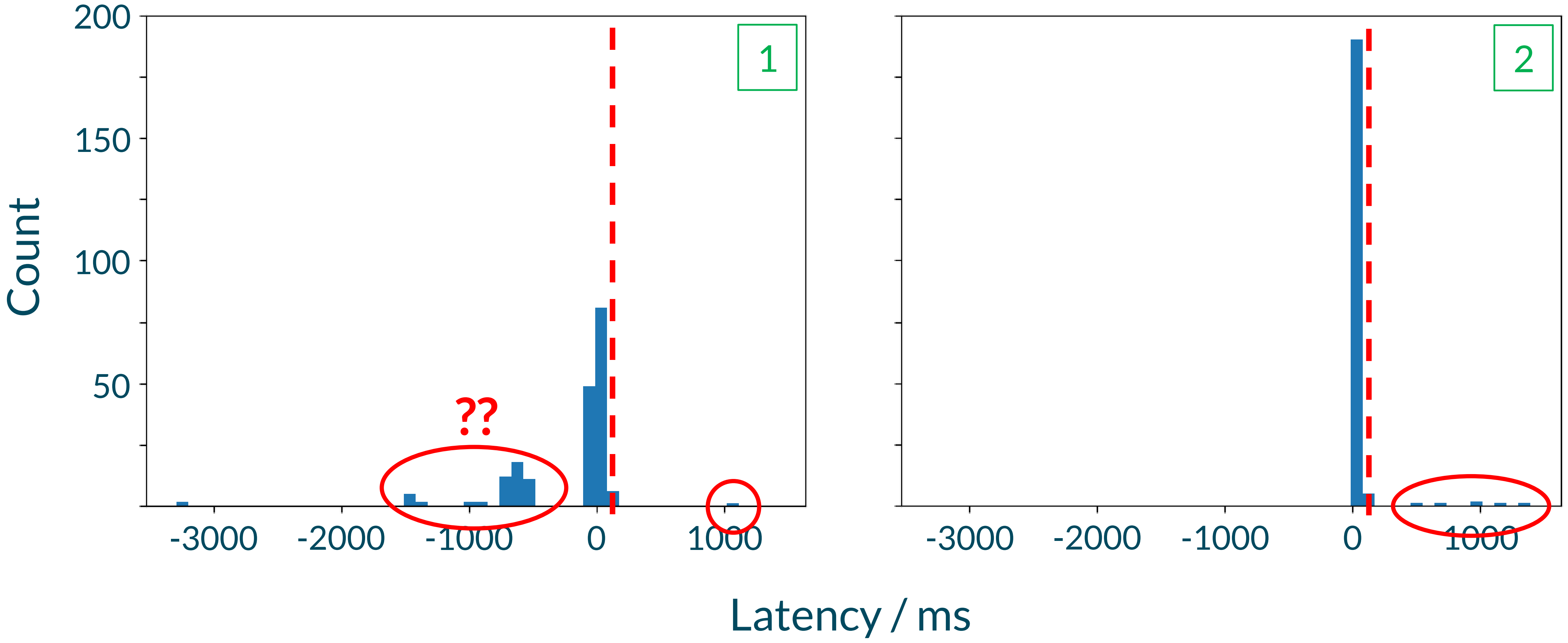
- Type: System (here: ROS2) instrumentation
  - Capture rosbag during application execution
  - Parse rosbag and use timing of ROS messages to calculate constraints
- Assessment:
  - (+) Simple tracing setup (`ros2 bag record -a`)
  - (+) Timing data combined with functional data
  - (-) Tracepoints limited to ROS messages
  - (-) Overhead (but: configurable)

```
topics = {} # dict: id -> name
events = {} # dict: t -> event

with sqlite3.connect(bag) as conn:
  # Parse topics
  query = conn.execute(
    'select * from topics')
  for id, name in query:
    topics[id] = name

  # Parse events
  query = conn.execute(
    'select * from messages')
  for topic_id, t in query:
    topic_name = topics[topic_id]
    events.setdefault(t, []).
      append(topic_name)
```

Sample Python script to parse rosbags

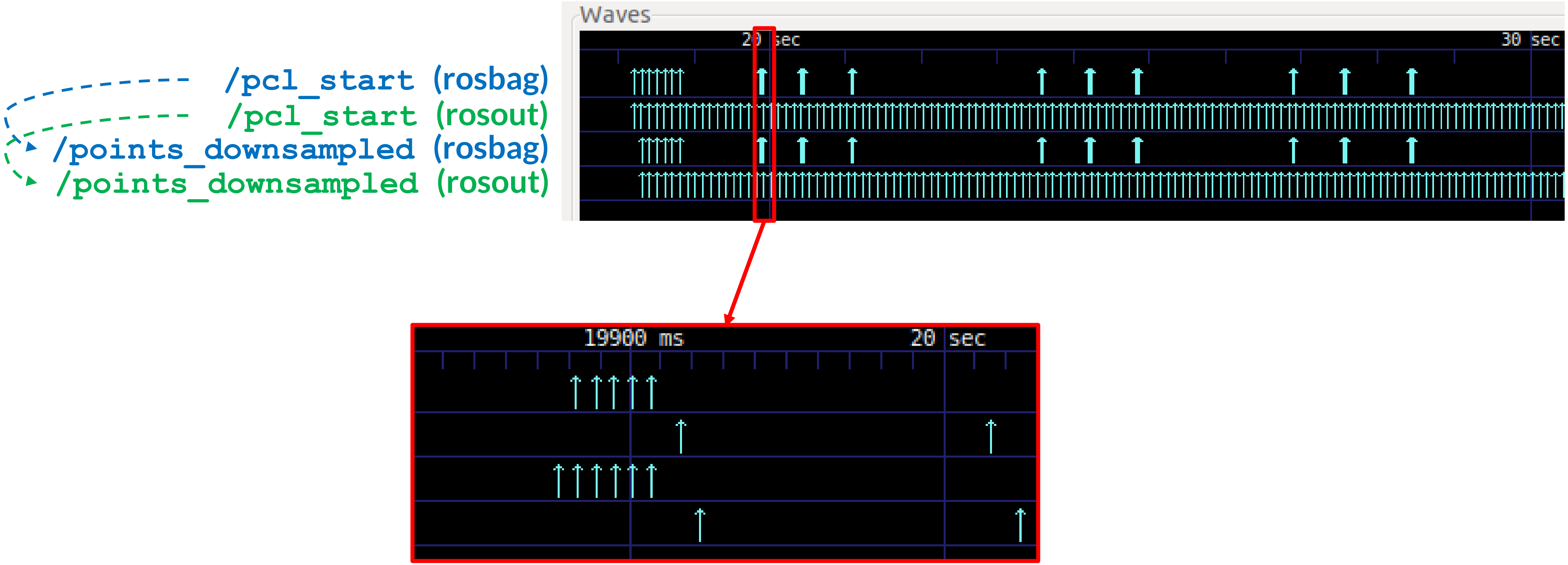
rosbag2

Results (Two Sample Runs)



rosbag2

Results



/pcl_start (rosbag)

/pcl_start (rosout)

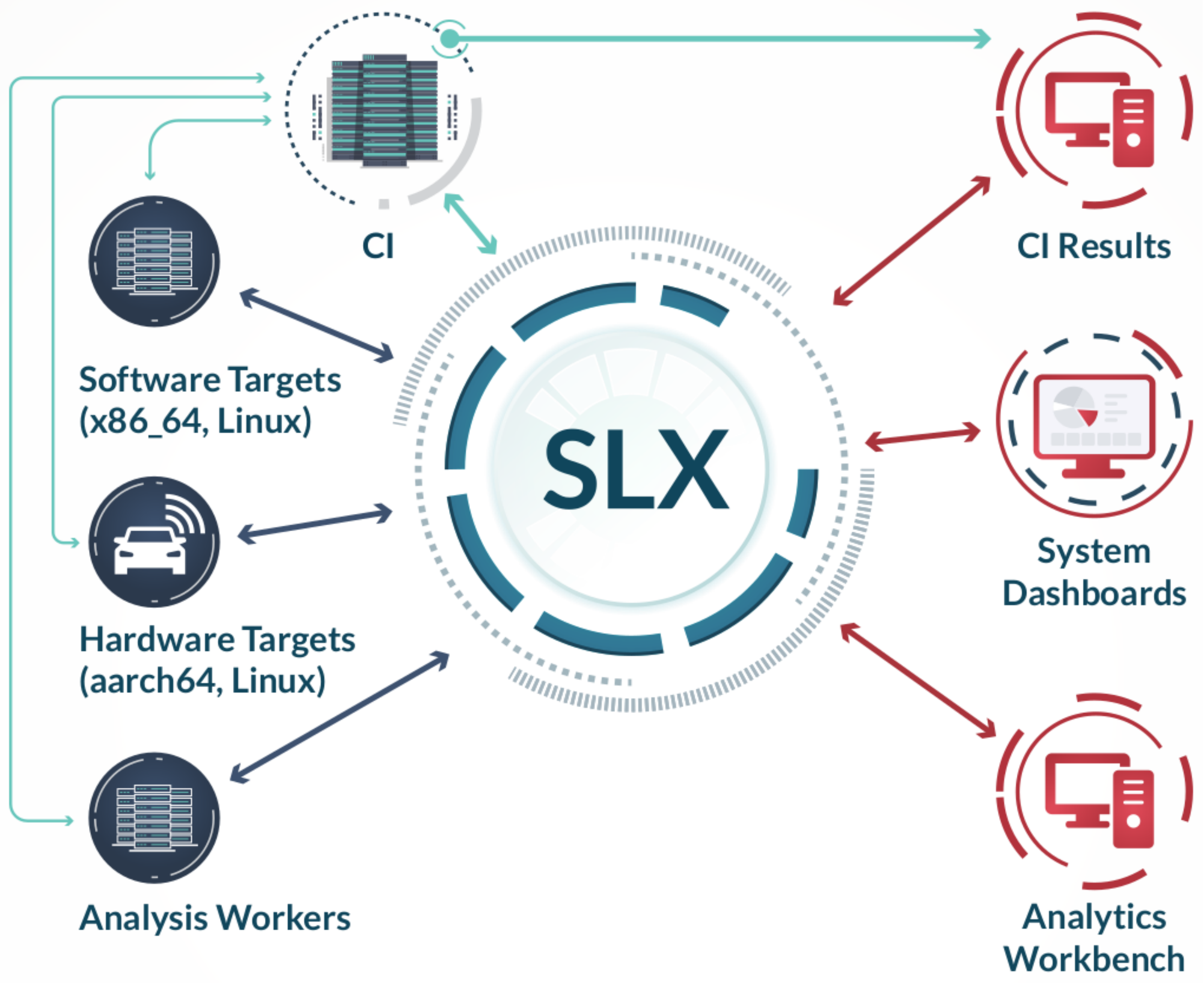
/points_downsampled (rosbag)

/points_downsampled (rosout)

SLX Performance Testing Platform

https://www.silexica.com/

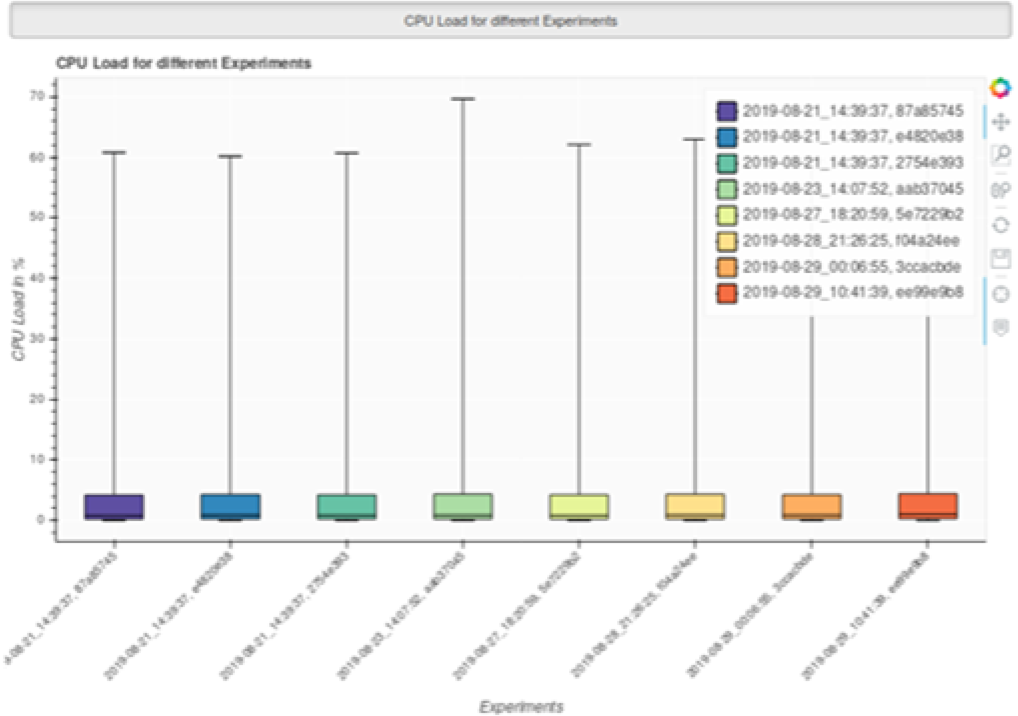
- Built-in system, ROS2, and application tracing:
  - Wide range of metrics (CPU, ROS2, network, etc.)
  - Multi-run analysis
  - Configurable stress generators (CPU, memory, ...)
- Open API:
  - Easily integrates into existing CI
  - Automated (RT) constraint testing
    - **Demonstrated here**
- Scalable:
  - Cloud/on-premise/desktop support



SUMMARY

Summary

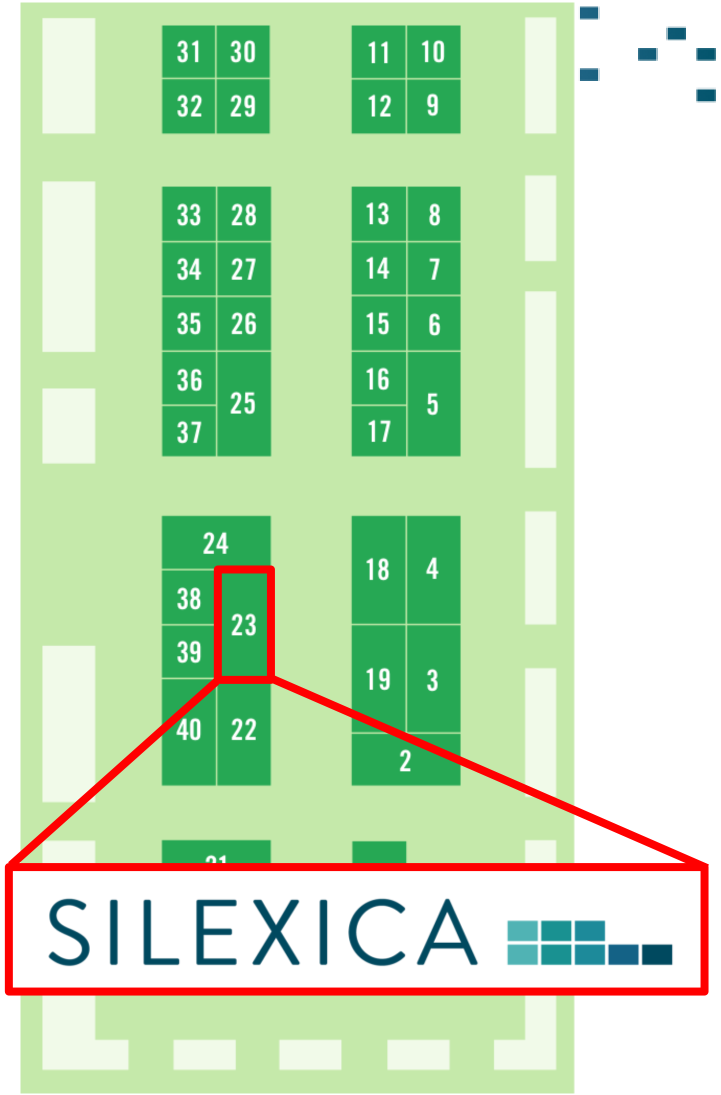
- Proving RT constraints for typical ROS 2 systems is very hard and costly (and in many cases unfeasible)
- Measurement-based approaches can be integral in understanding/optimizing the RT properties of a ROS2 system but give no guarantees
- There are many tools that can help optimize a ROS2 system for RT, but they cannot replace a proper RT system design

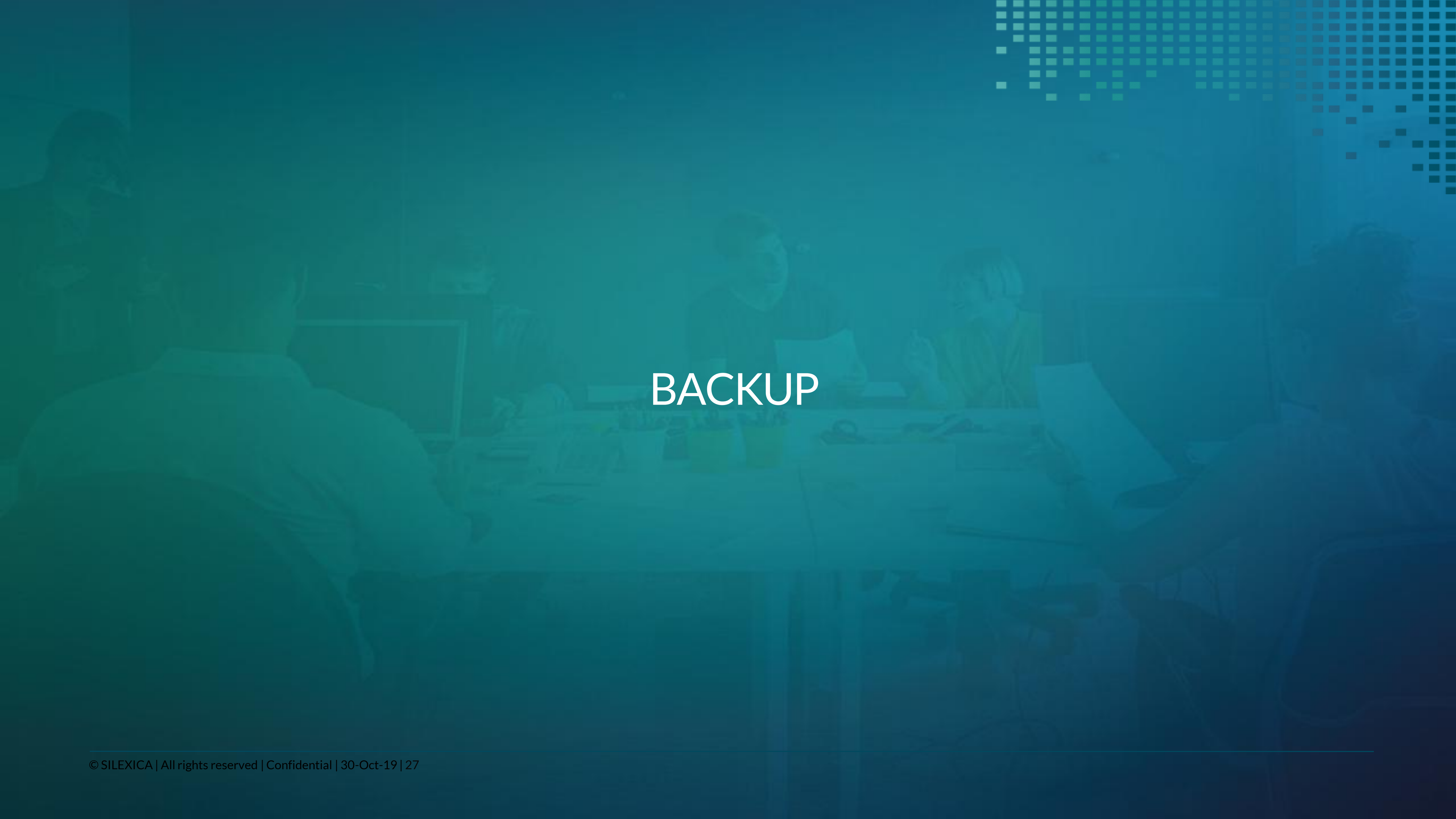


CPU Load Statistics for different Autoware.auto commits in the SLX Performance Platform

Thanks!

Visit booth (23) to learn more about our tools and enter the raffle for a chance to win a Nvidia Jetson AGX Javier devkit!



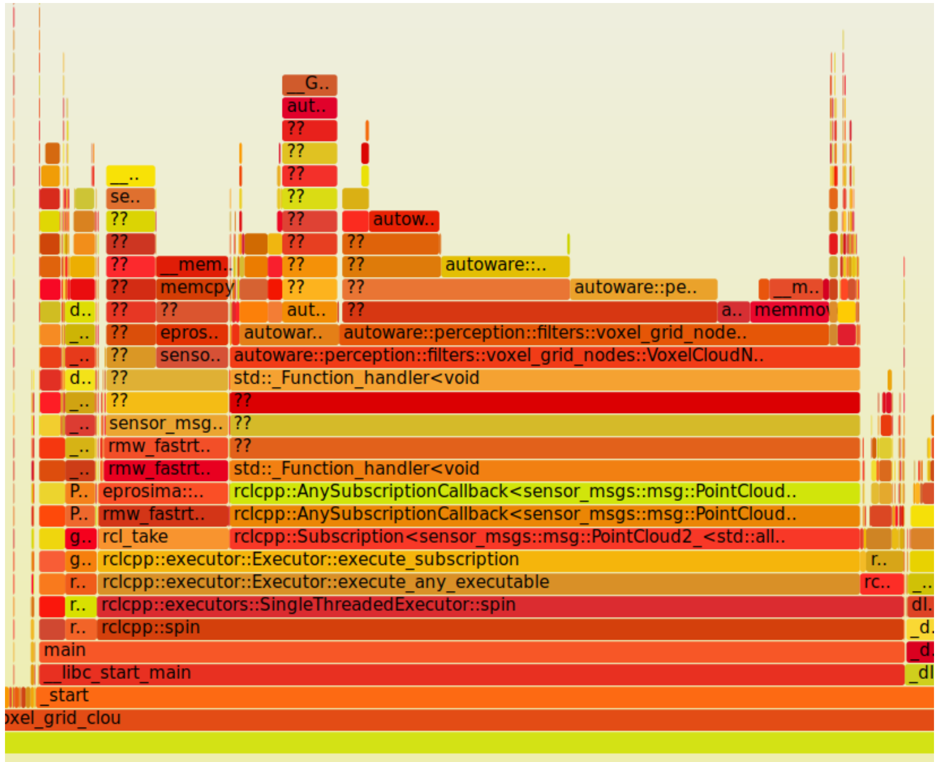


BACKUP

perf record -F

https://perf.wiki.kernel.org/index.php/Main_Page

- Type: System instrumentation (sampling)
  - perf has many different operating modi, shown here: Sampling mode
  - Sample stack traces at regular intervals
  - Interactive stack trace visualizer: (http://www.brendangregg.com/flamegraphs.html)
- Assessment:
  - (+) Lightweight
  - (+) Full system sampling supported
  - (+) Very good for process/function profiling
  - (-) Not directly suited to evaluate RT constraints

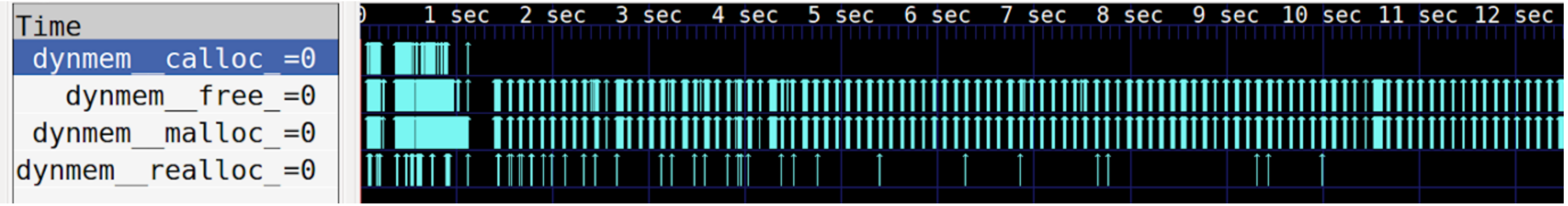


Sample Flamegraph for the Downsampling Node (excerpt)

LD_PRELOAD

https://www.linuxjournal.com/article/7795

- Type: Binary interception
  - Intercept calls to shared libraries: `LD_PRELOAD=/myLib.so ros2 run ...`
  - E.g. Detect dynamic memory allocations in (unit) tests using OSRF testing tools (https://github.com/osrf/osrf_testing_tools_cpp)
- Assessment:
  - (+) No changes to application or library
  - (+) No re-compilation needed
  - (-) Only at shared library boundaries
- Results:
  - Calls to malloc, free, etc. over time in the three perception pipeline modules



Stress Testing Tools

Generators

- `dohell` script:
  - Starting a sensible mix of latency generators, with no one clearly dominating (http://groups.google.com/group/linux.kernel/msg/0c88c397347cbd2a)
- Linux `stress` tool:
  - `-c`: CPU load, `-m`: Allocate/free memory, `-i`: I/O load
- File system access:
  - `find /dir/` - Iterates over `/dir/` → Causes disk accesses → Interrupts
  - `du /dir/` - Computes total size of all files in `/dir/` → Causes disk accesses
- Cause network traffic:
  - `ping -l 9999 -i 0.0001` → High locality in small function of network stack → Does not stress cache + virtual memory management

Stress Testing Tools

Benchmarks

- Benchmarking interrupt processing in kernel:
  - RealFeel: measure timing accuracy of periodic time interrupt (http://brain.mcmaster.ca/~hahn/realfeel.c)
- Benchmarking RT-extended Linux kernel:
  - Linux RT Benchmarking Framework (https://www.opersys.com/lrtbf)
  - Hourglass: Synthetic application for benchmarking of ms/μs task scheduling (http://www.cs.utah.edu/~regehr/hourglass)