Executors benchmark

Here we compare the performances of the existing ROS2 executors `SingleThreadedExecutor` and `StaticSingleThreadedExecutor` against the new `EventsExecutor`, when applied to two ROS2 systems on two different platforms.

The tests have been performed using the `iRobot's performance framework`. This framework allows to specify a ROS2 topology by means of a `json` topology file, run the ROS2 system and measure the performances.

The CPU requirement is measured as the ratio between the user time and real time, while the memory is measured as the RSS of the application.

The latency is measured as the duration between the call to the ROS 2 publish function to when the subscription callback is invoked.

Raspberry Pi1

- Raspberry Pi 1 - 580 MHz - Single core
- ROS 2 topology tested:
  - Name: Cedar
  - Nodes: 9
  - Executors: 9 (1 per node)
  - Publishers: 13
  - Subscriptions: 19
  - Frequency: 64Hz

IPC ON

The first 3 graphs show the average pub/sub latency for every subscription in the system.

The 4th graph shows the CPU comparison for all executors.

The table shows number on detail.

Missing: Measure timers accuracy

IPC OFF

Raspberry Pi 2
Raspberry Pi 2 - 900MHz - Quad-core
ROS 2 topology tested:
- Name: White Mountain
- Nodes: 20
- Executors: 1
- Publishers: 23
- Subscriptions: 35
- Frequency: 100Hz

IPC ON

IPC OFF

FTRACES on Raspberry Pi

For a deep understanding of the CPU usage by all threads on the system we can use `trace-cmd`, a Linux kernel feature which can trace Linux kernel function calls.

We run this tool for a certain amount of time, while it will be capturing all kernel events. It will output a `trace.dat` file which can be used as input to `kernelshark` to generate a plot of all the threads events on the system.

We want to use this tool to observe the threads activity while running ROS2 benchmarks.

The first tests observes threads events of a basic ROS2 pub/sub system sending a 10b message every second, so we can measure the time taken for a complete publish and subscribe event for all the different executors.

Using a small message assure us that we are not spending time on allocating memory, so we can measure and compare the real ROS2 executors overhead.

Example usage/command:
- Terminal 1: `trace-cmd record -e sched` /stop recording with ctrl+c, run it for a short amount of time (30 secs is fine)
- Terminal 2: `./irobot-benchmark --topology topology/cedar.json --tracking off --ros_params off --ipc on -t 15`

Pub-Sub: 10 bytes message

SingleThreadedExecutor
Kernelshark plot of pubsub10b-default-executor-trace.dat

Pub/sub duration: 226 microseconds

**StaticSingleThreadedExecutor**

Kernelshark plot of pubsub10b-static-executor-trace.dat

Pub/sub duration: 150 microseconds

**EventsExecutor**

Kernelshark plot of pubsub10b-events-executor-trace.dat

Pub/sub duration: 76 microseconds

**Comparison plot**
Conclusion

The test shows that **EventsExecutor** reduces 66% the time needed for a basic pub/sub event for small messages, which is equivalent to a 66% CPU reduction.

This CPU improvement should be reflected on any topology (as long as the messages are kept small).

White Mountain topology

The second tests is just to observe threads activity of the benchmark using a full ROS2 system (white mountain topology) for all the different executors.

The time window is 70 milliseconds (duration between green and red lines) for the 3 executors benchmark, so we use the same scale and can better appreciation of the threads activity.

**SingleThreadedExecutor**

single_thread_exec_white_mountain-trace.dat

**StaticSingleThreadedExecutor**

static_exec_white_mountain-trace.dat
Conclusion

It's observed on the plots that the EventsExecutor threads are considerably less busy than the StaticSingleThreadedExecutor and SingleThreadedExecutor threads. This is in line with the observed CPU usage for each executor.

CPU Profiling

Other useful way to understand the CPU usage for each executor is performing CPU profiling. This way we can measure the time spent on each function and identify CPU bottlenecks.

For this we'll use perf tool - a sampling profiler- and FlameGraph to visualize the sampled stack traces. The output files *.svg can be opened with any web browser to inspect in detail each function.

FlameGraph description:

- Each box represents a function in the stack (a “stack frame”).
- The y-axis shows stack depth (number of frames on the stack). The function beneath a function is its parent.
- The x-axis spans the sample population. It does not show the passing of time from left to right, as most graphs do.
- The width of the box shows the total time it was on-CPU or part of an ancestry that was on-CPU (based on sample count). Functions with wide boxes may consume more CPU per execution than those with narrow boxes, or, they may simply be called more often.

Example usage/command:

```
./irobot-benchmark --topology topology/cedar.json --tracking off --ros_params off --ipc on -t 300
perf record --call-graph dwarf
```

```
git clone https://github.com/brendangregg/FlameGraph ~/.FlameGraph
```

```
cp perf.data ~/.FlameGraph/ && cd ~/.FlameGraph
```

```
perf script | ./stackcollapse-perf.pl > out.perf-folded
```

```
./flamegraph.pl out.perf-folded > ros2-benchmark-cpu-profiling.svg
```

Single Threaded executor
Static Single Threaded executor

Events executor