Exploring Partitioned and Semi-partitioned Callback Scheduling on ROS 2 Multi-threaded Executors

Hoora Sobhani, Daniel Enright, Tejas Milind Deshpande, Hyoseung Kim

University of California, Riverside
What is ROS 2

**ROS 2**: An important open-source middleware framework for the development of robotic applications

- Provides modular integration of software components for complex robotic applications

---

ROS 2 Architecture

**Executors:** processes with one or more threads scheduled by OS scheduler
- Can be single-threaded processes or multi-threaded processes
- Maintains a local wait-set of callbacks to be assigned to a thread for execution

**Callbacks:** smallest schedulable entity in ROS 2
- **Scheduled by executors** running on the CPU
- Five types of callbacks:
  - Timer, subscription, service, client, and waitable
  - When callbacks are released, they are added to their executor’s wait-set

**Nodes:** syntactical organization of callbacks
- Used to assign callbacks to executors
Processing Chains in ROS 2

- Semantic abstraction of a sequence of **data-dependent** callbacks
  - Example: Apex.AI’s Autoware Reference System*

Scheduling Callbacks on Multi-CPU Systems

ROS Executor Types:
- Single-threaded executor:
  - One thread that pulls ready callbacks from the wait set to execute on the CPU
  - User can create $n$ single-threaded executors for an $n$-CPU system
- Multi-threaded executor:
  - Multiple threads that pull ready callbacks from the wait-set to execute on one or more CPUs
  - By default, ROS creates $n$-threads for an $n$-CPU system
Callback Scheduling within ROS 2 Executors

1. Callbacks are executed on the CPU cores non-preemptively
2. Each executor maintains a single wait-set for ready callbacks
3. Callbacks in the wait-set are prioritized to execute in the following order:
   - Timer, subscription, service, client, and waitable
4. Wait-set is only updated when it is empty
Example: Callback Scheduling with MT Executor

Multi-threaded Executor

Executor Threads

Wait Set
- Callback 1
- Callback 2
- Callback 3
- Callback 4
- Callback 5

Incoming Callbacks
- Callback 6
- Callback 7
- Callback 8

Threads can migrate freely across the CPU cores
Literature Review

- Multiple single-threaded executors vs. Single multi-threaded executor

<table>
<thead>
<tr>
<th>Multiple single-threaded executors</th>
<th>Single multi-threaded executor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioned Scheduling</td>
<td>Global Scheduling</td>
</tr>
</tbody>
</table>

- Per-thread wait-set: less contention
- Better for isolation between workloads
- No migration (potentially less overhead)
- Difficult to determine callback-to-executor assignment
- One node cannot be split into two executors
- Potential resource underutilization

- Allows a single process memory space, allowing more efficient inter-callback data transfers (via intra-process API)
- Better to reclaim unused CPU time
- No isolation: potentially longer blocking time for low-priority callbacks and more interference among different chains

Related Work

Proposed Idea

Implement partitioned and semi-partitioned scheduling within the default ROS 2 multi-threaded executor

- New thread affinity API introduced to the rclcpp library
- Allows developers to bind callbacks to specific threads within the multi-threaded executor
- Facilitates the reservation of execution bandwidth for high priority callbacks*

* When executor threads are applied the SCHED_DEADLINE policy and cgroups are used to set their CPU masks
Thread Affinity API Implementation Details

// Set thread (CPU) affinity
void Executor::set_thread_affinity(rclcpp::TimerBase::SharedPtr ptr, int* affinity_threads, int size);
void Executor::set_thread_affinity(rclcpp::SubscriptionBase::SharedPtr ptr, int* affinity_threads, int size);
void Executor::set_thread_affinity(rclcpp::ServiceBase::SharedPtr ptr, int* affinity_threads, int size);
void Executor::set_thread_affinity(rclcpp::ClientBase::SharedPtr ptr, int* affinity_threads, int size);

// Calculate final thread affinity based on threads assigned (Called by each of the above API methods)
size_t Executor::get_final_affinity_value(int* affinity_threads, int size);

// Default Parameters

#elifdef PICAS
    int callback_priority = 0;
#endif
size_t thread_affinity = 0;
Thread Affinity API Implementation Details

// Check thread affinity for current thread

if((callback->thread_affinity >= 0 ) && !(callback->thread_affinity & (1 << thread_affinity_id))) {
    ++callback_iterator; // Skip this callback; check next ready callback
    continue;
}

Runs inside:
get_next_timer(),
get_next_subscription(),
get_next_service(),
get_next_client(),
get_next_waitable()
Experiment: Platform and Taskset

Implemented on ROS 2 Galactic

- Performed on an Intel Core i7-10875H (4 cores, pinned to maximum frequency)
- Using our modified rclcpp library incorporating the thread affinity API
- Followed default ROS 2 callback priority ordering
Experiment: Allocation and Results

- Partitioned scheduling: each thread is assigned a group of callbacks as follows
  - \{\tau_1, \tau_2\}, \{\tau_3, \tau_4, \tau_5\}, \{\tau_6, \tau_7, \tau_8\}, \{\tau_9, \tau_{10}, \tau_{11}\}
- Semi-partitioned scheduling: callbacks \(\tau_1, \tau_6, \text{ and } \tau_9\) are statically assigned to separate threads. Other callbacks can migrate between threads 1-4.
# Conclusion

- **What would be the best way to utilize multiple CPUs in ROS 2?**

<table>
<thead>
<tr>
<th>Prior Work</th>
<th>This Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioned Scheduling (via multiple single-thread executors)</td>
<td>Global Scheduling (via multi-threaded executor)</td>
</tr>
<tr>
<td>Semi-Partitioned Scheduling (within multi-threaded executor)</td>
<td></td>
</tr>
</tbody>
</table>

**Pros:**
- Better isolation and predictability compared to global scheduling
- Better resource utilization compared to partitioned scheduling

**Cons:**
- Complexity in maintaining predictability and managing task migration between cores

*Source code: [https://github.com/rtenlab/ros2-picas](https://github.com/rtenlab/ros2-picas) (branch: multi_threaded_partitioned_scheduling)
Questions