PAAM: A Framework for Coordinated and Priority-Driven Accelerator Management in ROS 2

Daniel Enright*, Yecheng Xiang*, Hyunjong Choi†, Hyoseung Kim*

* University of California, Riverside
† San Diego State University
Why ROS 2 and Accelerators?

**ROS 2:** Important middleware for development of robotic applications!
- **Autoware**

**Accelerators:** Essential for modern robotic workloads!
- Sensing, perception, decision-making, and planning tasks.

**Real-time ROS 2 and Accelerators:** Allows development of modular high-performance multi-process safety-critical applications!
- Resource sharing makes **timely execution** of safety-critical applications tricky
Background: ROS 2 Architecture

**Executors:** processes with one or more threads scheduled by **OS scheduler**
- ROS 2 offers a **multi-process execution model**!

**Callbacks:** smallest schedulable entity in ROS 2
- **Scheduled by executors** running on the CPU

**Nodes:** syntactical organization of callbacks

Current practice in using accelerators with ROS 2
- **Direct invocation** from callbacks
  - Executor process issues requests to devices
  - Results in **unmanaged accelerator access**
Background: Processing Chains in ROS 2

Semantic abstraction of a sequence of **data-dependent** callbacks

- Example: Apex.AI’s Autoware reference system*

Prior Work

Analyzable ROS 2 callback scheduling on executors (e.g. PiCAS) $^{[3,4,7,8]}$
- Provides chain-aware scheduling on single & multi-threaded executors
- **CPU only:** Analysis does not work when accelerators are introduced

Real-time GPU management frameworks for ROS 2 (e.g. ROSGM) $^{[45]}$
- Provides an interface for real-time GPU management in ROS 2
- **No end-to-end timing guarantees on chains**
- Does not consider multiple executors, or multiple types of accelerators

System Model

Processing Chain $\Gamma_c$

Callback $\tau_i := (E_i, A_i, R_i, \eta_i)$

- $E_i$: WCET of CPU segments in $\tau_i$
- $A_i$: WCET of accelerator segments in $\tau_i$
- $R_i$: Set of accelerators used by $\tau_i$
- $\eta_i$: # of accelerator segments in $\tau_i$

Chain $\Gamma_c := ([\tau_{c1}, \tau_{c2}, \ldots, \tau_{cn}], T_c, D_c, \delta_c)$

- $[\tau_{c1} \ldots ]$: Sequence of callbacks in chain $\Gamma_c$
- $T_c$: Period of chain $\Gamma_c$
- $D_c$: Relative deadline of $\Gamma_c$ ($D_c > T_c$)
- $\delta_c$: # of accelerator segments in $\Gamma_c$
Challenges with Accelerators

1. **Priority inversion and unanalyzable blocking time**
   - Requests from lower priority chains can block those from higher priority chains
   - **Why?** OS and accelerator drivers are unaware of the concept of processing chains and chain/callback priorities in ROS 2!

2. **Poor accelerator resource utilization**
   - Most accelerators (e.g., TPU): sequential, no preemption & concurrent exec.
   - **GPU** access from multiple executor processes:
     - Interleaved execution (= fair slowdown), high GPU context-switching cost

3. **Disparity in chain and executor priorities**
   - Chain priority may not match with the executor process priority
Contributions

**PAAM**: A Priority-driven Accelerator Access Management framework for real-time multi-process ROS 2 applications

- Presents the "accelerator access as a service" paradigm
- Schedules accelerator requests with respect to *chain priority*
- Offers *bounded WCRT* and admissions control for processing chains
- Supports *multiple accelerators* of various types (GPU and TPU)
- Leverages separate *data and control planes* to minimize transport overhead

Open-source combined GPU- and TPU-specific implementation

- Achieves up to a *91% reduction* in the *end-to-end latency* of critical chains
PAAM Client-Server Architecture

- **Clients**
  - Executor Process #1
    - Node 1: Callback #1 (CPU, GPU, TPU)
    - Node 2: Callback #2 (CPU, GPU)
    - Node 3: Callback #3 (CPU, GPU, TPU)
    - Node 4: Callback #4 (CPU, TPU)

- **PAAM Server**
  - Control Plane
    - Client Registration
    - Accelerator Request
  - Data Plane
    - Shared Memory #1 (GPU, TPU, ...)  
  - DDS (Data Distribution Service)
  - Large kernel data
  - Zero-Copy w/o Serialization

- **Hierarchical Request Management**
  - Device-level priority
  - Priority Downsampling
  - Bucket 1, Bucket 2, ..., Bucket n
  - Request Callback
  - Requests w/ chain priority

- **ROS 2 Middleware Stack**
- **OS and Driver Stack**

Accelerator
Data and Control Planes

**Challenge:** Data communication overhead between client and PAAM server
- Main issues: DDS serialization, data copy to DDS transport

**Our solution:** Separate data plane from control plane, with shared memory

**1. Data Plane:**
- Clients’ input data for kernels directly stores in shared memory
  - No serialization, raw datatypes
- PAAM server stores the results directly to shared memory
  - No unnecessary copies
  - Results can be used before device memory is freed

**2. Control Plane:**
- Fixed-size msgs (= local accelerator reqs)
  - Leverages zero-copy DDS capabilities
  - Iceoryx-enabled CycloneDDS
- Variable-size msgs
  - Client reg, remote accelerator reqs, etc.
  - Uses existing DDS transport
PAAM Client-Server Architecture

(Diagram showing the architecture with various components and connections)

ROS 2 Middleware Stack
OS and Driver Stack

PAAM Server

Client Plane
- Client Registration
- Accelerator Request

Data Plane
- Shared Memory #1
- DDS

Hierarchical Request Management
- Request Callback
- Priority Downsampling

Bucket 1
- Bucket 2
- Bucket n

Device-level priority
Thread-local priority queue

Executor Process #1
- Node
  - Callback #1
    - CPU
    - GPU
    - TPU

Node
- Callback #2
  - CPU
  - GPU

Node
- Callback #3
  - CPU
  - GPU

Node
- Callback #4
  - CPU
  - TPU

Zero-Copy w/o Serialization
Control messages
Large kernel data
Hierarchical Request Management

**Challenge:** Some accelerators (e.g., GPU) support device-level prioritization – But there is a mismatch between chain priority and device priority numbers

**Our solution:** Two-level hierarchical request management

**Level 1: N worker threads (buckets)**
- One for each device priority (1..N) determined at initialization
- Chain priorities are down-sampled into buckets

**Level 2: Thread-local queues**
- Each bucket maintains a local priority queue for requests
- Executes requests in chain priority order
- Maintains records of requests and their callbacks
Accelerator-Specific Considerations

**GPU – Nvidia GPU**

- Single execution context for all kernels – no context switching!
- 6 buckets, one per hardware stream priority
  - Allowing preemptive kernel execution
- Lowest priority bucket for best-effort chains
  - Allows concurrent execution of kernels for increased throughput

**TPU – Coral Edge TPU**

- Single execution context for all requests
  - Allows multiple client processes to use the device!
- Single bucket for all requests
  - No prioritized hardware queues
- Non-preemptive, sequential, priority-ordered execution of requests
PAAM Client-Server Architecture

Clients

Executor Process #1

Node

Callback #1
- CPU
- GPU
- TPU

Node

Callback #2
- CPU
- GPU

Node

Callback #3
- CPU
- GPU

Node

Callback #4
- CPU
- TPU

PAAM Server

Control Plane

Client Registration

Accelerator Request

Data Plane

Shared Memory #1
- GPU
- TPU

ROS 2 Middleware Stack

OS and Driver Stack

Hierarchical Request Management

Requests w/ chain priority

Device-level priority

Thread-local priority queue

Accelerator

Priority Downsampling

ROS 2 Middleware Stack

OS and Driver Stack
Admissions Control

Purpose: To guarantee end-to-end response time and to protect timely execution of previously admitted chains

How we do admissions control:

1. Clients send a request to the server for chain admission
2. Server determines WCRT of new chain, considering all previously admitted chains
3. Server evaluates if the computed WCRT bound for each chain satisfies each deadline
Admissions Control

**Lemma 1. Maximum number of requests** that an accelerator segment can generate in an arbitrary interval $t$:

$$\mu_{k,q}(t) \leftarrow \left\lfloor \frac{t}{T_{c'}} \right\rfloor + 1$$

**Lemma 2. Maximum handling time of an accelerator segment** of a callback within a chain instance:

$$H_{c,i,j} \leftarrow A_{i,j}^* + \max_{\tau_{k,q} \in lps(\tau_{i,j})} A_{k,q}^* + \sum_{\tau_{k,q} \in hps(\tau_{i,j})} \mu_{k,q}(H_{c,i,j}) \cdot A_{k,q}^*$$

**Lemma 3. Maximum time to handle all accelerator requests** of any chain instance:

$$H_{c} \leftarrow \sum_{\tau_{i,j} \in \Gamma_c} \left( A_{i,j}^* + \max_{\tau_{k,q} \in lps(\tau_{i,j})} A_{k,q}^* \right) + \sum_{\tau_{k,q} \in \bigcup_{\tau_{i,j} \in \Gamma_c} hps(\tau_{i,j})} \mu_{k,q}(R_c) \cdot A_{k,q}^*$$
Admissions Control

**Theorem 1.** Worst-case response time of a chain with accelerator segments under the PAAM framework is bounded by:

\[ R_c \leftarrow B_c + E_c + H^{*} + \sum_{\Gamma_h \in hp(G_c)} \mu_h(R_c) \cdot (E_h + H^{*}) \]

\[ + \sum_{\Gamma_h \in hpp(G_c)} \mu_h(R_c) \cdot (E_h + \text{spin}(\Gamma_h)) \]

- **Blocking time from lower-priority chains**
- **WCET of all callbacks of the chain**
- **Handling time of all accelerator segments of the chain**
- **Delay from higher priority chains accessing the same accelerator**
- **Delay from callbacks executing on the same CPU core with a higher executor process priority**
Evaluation

Experimental setup:
- ROS 2 Galactic on the Nvidia Jetson AGX Xavier platform running Ubuntu 20.04
- 8 CPU cores, 1 iGPU, 1 Coral USB Edge TPU

Source code of our implementation:
- [https://github.com/rtenlab/reference-system-paam.git](https://github.com/rtenlab/reference-system-paam.git)
Case Study 1: GPU-enabled Robotic System

Inspired by PiCAS* case study

6 critical chains†, descending priority
- Periodic, linear, and non-linear chains
- Sharing CPU cores with highest criticality chains

2 best-effort chains

Each callback has one GPU segment

---


†Follows PiCAS callback-to-executor and executor-to-cpu assignment
Case Study 1: GPU-enabled Robotic System

Maximum observed end-to-end chain latency

- End-to-end chain latency is upper bounded by our analysis

- PAAM schedules accelerator jobs with respect to chain priority

- PAAM outperforms PiCAS and ROS 2 for all real-time chains
Case Study 2: Apex.AI’s Autoware Reference System

Based on a lidar-based perception pipeline for an autonomous vehicle

- From sensor input to behavior planner output

Several prioritized processing chains with extensive GPU & TPU usage

- Evaluated with multiple single-threaded and multithreaded executors
Case Study 2: Apex.AI’s Autoware Reference System

Reference System KPIs

(1) Hot Path Latency:
Latency from the LiDAR sensors to the output of the behavior planner

(2) Behavior Planner Period:
Accuracy of planner execution period

(3) Hot Path Message Drops:
Message drops on the hot path

PAAM achieves a **91% reduction** in the hot path latency compared to standard ROS 2!
Schedulability Experiment

Variable # of Chains per Chainset

- Variable number of chains per chainset and fixed number of callbacks per chain ($n$ chains, with 4 callbacks each)
- Fixed Accelerator-to-CPU utilization ratio (1:1) per chain
- Tested with variable utilization maximum per chain

Variable CPU:GPU Utilization Ratio

- Fixed number of chains and callbacks per chain (4 chains, with 4 callbacks each)
- Varied Accelerator-to-CPU utilization ratio per chain
- Tested with variable utilization maximum per chain
PAAM Overhead Analysis

Overhead breakdown

Max non-DDS overhead (pure PAAM overhead): < 150 $\mu$s!

Max preemption delay: 129 $\mu$s!

Nvidia GPU inter-stream preemption cost

<table>
<thead>
<tr>
<th></th>
<th>MatMul</th>
<th>Reduction</th>
<th>VectorAdd</th>
<th>Histogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean ($\mu$s)</td>
<td>41.21</td>
<td>39.55</td>
<td>22.78</td>
<td>14.95</td>
</tr>
<tr>
<td>Max ($\mu$s)</td>
<td>116.48</td>
<td>129.18</td>
<td>77.85</td>
<td>72.92</td>
</tr>
<tr>
<td>Stdev ($\mu$s)</td>
<td>14.16</td>
<td>26.65</td>
<td>15.13</td>
<td>10.77</td>
</tr>
</tbody>
</table>
Summary

**PAAM: Priority-driven Accelerator Access Management Framework**
- Implemented in C++ for ROS 2
- Supports all types of accelerators & real-time ROS 2 applications
- GPU and TPU implementation on a single server instance
- WCET bounding for prioritized chains
- Thorough evaluation and open-source test cases

Thank you!

https://github.com/rtenlab/reference-system-paam.git