On-Device CPU Scheduling for Robot Systems

Aditi Partap[1], Samuel Grayson[2], Muhammad Huzaifa[2],
Sarita Adve[2], Brighten Godfrey[2], Saurabh Gupta[2], Kris Hauser[2], Radhika Mittal[2]

***Abstract*— Robots have to take highly responsive real-time actions, driven by complex decisions involving a pipeline of sensing, perception, planning, and reaction tasks. These tasks must be scheduled on resource-constrained devices such that the performance goals and the requirements of the application are met. This is a difficult problem that requires handling multiple scheduling dimensions, and variations in computational resource usage and availability. In practice, system designers manually tune parameters for their specific hardware and application, which results in poor generalization and increases the development burden. In this work, we highlight the emerging need for scheduling CPU resources at runtime in robot systems. We use robot navigation as a case-study to understand the key scheduling requirements for such systems. Armed with this understanding, we develop a CPU scheduling framework, Catan, that dynamically schedules compute resources across different components of an app so as to meet the specified application requirements. Through experiments with a prototype implemented on ROS, we show the impact of system scheduling on meeting the application's performance goals, and how Catan dynamically adapts to runtime variations.**

I. Introduction

Many robots feature a pipeline of tasks that involve continually sensing the environment and processing the sensed inputs in software to generate a reaction. In many low cost robots, the software processing tasks run on on-board platforms with limited compute resources (e.g., Intel NUC [1], [2], Raspberry Pi [3], etc.). This work focuses on managing the compute (CPU) resources on such platforms.

CPU scheduling for a given robotics application (app) involves tackling two inter-related aspects — how should the available compute resources be divided across different app components (i.e., its processing tasks), and the rate at which each component should be executed. The appropriate scheduling decisions depend on the amount of available CPU resources, the compute usage of each component, and the app's performance requirements. The scheduler must take into account dynamic variations in compute usage (e.g., due to input-dependent logic) and compute availability (e.g., due to battery constraints). The performance requirements may differ along different components of an app, and involve semantic trade-offs (e.g., in a navigating robot, components responsible for avoiding local collisions are more critical than those responsible for global path planning).

This is a complex problem in which app developers are provided little help. While there are frameworks that assist in app development (e.g., ROS [4]), they leave scheduling decisions entirely up to the developers. Developers, therefore, manually fine-tune their systems to come up with static configurations (i.e., the rates at which different components or nodes are triggered). This requires heavy engineering effort, which increases the burden of app development. Moreover, these static settings generalize poorly across scenarios and over time, which impacts the robot's performance.

To ease app development and enable better performance, we build a CPU scheduling framework, Catan, that takes an app's semantic requirements as initial inputs from the developer, and dynamically schedules the app components at runtime as per the (varying) compute usage and availability. It is easier for the developer to specify the semantic inputs needed by Catan (that are based on domain expertise, and remain unchanged over time and across compute platforms), as opposed to directly configuring the lower-level system scheduling knobs (that must be dynamically adapted based on compute usage and availability). Catan can be plugged into ROS, and can be provided as an optional service to the apps using the framework.

Catan's central function is thus to translate semantic requirements into dynamic low-level actions. To accomplish this, we adopt a hierarchical approach, making scheduling decisions in two stages – first determining the spatial allocation of CPU cores across app components, and then determining the temporal allocation of CPU slices and the rate of executing different app components. We develop analytical models and heuristics to translate the scheduling decisions into light-weight constraint-optimization problems, that Catan solves periodically at runtime to account for variability in compute usage and availability.

Our evaluation (using ROS implementation for 2D navigation with Stage simulator [5]) shows how scheduling decisions impact the navigating robot's performance, and how Catan can achieve the desired semantic trade-offs and better performance than the default (hand-tuned) configuration, while effectively handling the variability in compute usage at different timescales. In particular, using Catan reduces the number of collisions with dynamic obstacles by 87%, and reduces the staleness of sensor (odometry) readings used for generating trajectories by 18-91% when compared to the default settings for the ROS app. We also evaluated Catan with other applications – it enables better tracking performance for a ROS-based face tracking app, and lowers the motion-to-photon latency in an extended reality system [6] (detailed in the long version of our paper [7]). These results demonstrate that Catan captures semantic requirements and improves performance over a wide range of applications.

We begin by providing a brief background in §II, followed

[1] Stanford University, [2] University of Illinois at Urbana-Champaign
Email to aditi712@stanford.edu, radhikam@illinois.edu

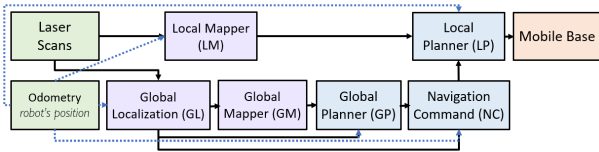

Fig. 1: DAG representing the robot navigation application. The colors green, purple, blue represent sensor nodes, perception nodes, and planning nodes respectively.

by describing our scheduling approach in §III, before presenting results in §IV. We discuss related work in §V.

II. Background

1) Development Frameworks: Software frameworks for developing robotics applications (e.g. [8], [9], [4], [10], [11], [12]) allow developers to program each component of an application individually, and provide communication APIs among those components. However, such frameworks (including the most popular ones, ROS [4] and ROS 2 [12]) leave compute resource management entirely up to the app developers. Developers manually configure the rate at which different nodes are triggered. CPU scheduling across nodes is left up to the default OS policies, unless explicitly configured by the developer. [1] We design a scheduler for robotics applications that can be provided as a plug-in feature for such frameworks, and that can appropriately schedule the on-device CPU resources across different application components.

2) DAG Representation: We can represent a robotics application as a directed acyclic graph (DAG), where each node (or vertex) represents a computation task and each edge represents flow of data between tasks. The source nodes in the DAG (with no incoming edges) represent various sensors (e.g., camera, LiDAR, IMU, etc.), that continually capture environmental inputs. The sink nodes (with no outgoing edges) produce reactions (e.g., actuators). The in-between nodes are responsible for processing the input streams, e.g., by running detection and planning algorithms to determine appropriate reactions to changes in the environment. We use the term *chain* to denote a unique path from a source node to a sink node along the DAG. A node may belong to multiple chains. We next present a specific example.

3) Case-study: 2D Navigation: We study 2D navigation implemented in ROS [13], where the robot is tasked with exploring and mapping an unknown area. Figure 1 represents the application's DAG. It uses two sensors – a laser scanner (to capture the environment seen by the robot) and an odometer (which reports speed and location information to all other nodes). The global localization, mapping and planning nodes (GL, GM, and GP) are responsible for planning the robot's trajectory based on its accumulated knowledge about the area. The trajectory is planned so as to move towards unknown areas for exploration. The local mapping and planning nodes (LM and LP) are responsible for ensuring that the robot avoids collision with obstacles in its immediate vicinity when following the global trajectory. The DAG consists of multiple chains from scan and odometry to the mobile base, that pass through different sets of intermediate nodes (e.g. $\{$scans $\rightarrow$ LM $\rightarrow$ LP $\rightarrow$ base$\}$, $\{$scans $\rightarrow$ GL $\rightarrow$ NC $\rightarrow$ LP $\rightarrow$ base$\}$, etc). We list the specific function of each processing node below. More details about relevant algorithms can be found in [14], [15].

(i). The local mapper (LM) uses laser scans and odometry to update its knowledge about the robot's immediate vicinity.
(ii). The global localization node (GL) performs two tasks: (a) for every scan that it receives, it uses particle filtering (Chapter 8 in [14]) to produce an estimated correction for the robot's location (which accounts for potential drifts in odometry readings), and (b) it then filters the scans, discarding the ones that carry little new information about the environment. Unlike other nodes in the app that use the latest inputs available from their predecessor nodes, GL buffers the received scans and batch processes them every time it runs.
(iii) The global mapper (GM) maintains a global map of all the areas that the robot has explored so far. It uses occupancy grid mapping (Chapter 9 in [14]) to update the map based on newly filtered scans produced by GL.
(iv) The global planner (GP) computes the global trajectory based on GM's map and the robot's position (derived from the latest correction from GL and the corresponding odometry reading) using graph search (Chapter 3 in [15]).
(v) Navigation Command (NC) uses the robot's position, along with the current trajectory (i.e. GP's output) to decide the direction in which the robot should move.
(vi) The local planner (LP) uses the local cost map (that has information about nearby obstacles) and odometry information, along with the command from NC, to output the robot's velocity to the actuator (the mobile base).

III. Approach

We begin with describing some design considerations for the CPU scheduler (§III-A), followed by detailing Catan's design (§III-B) and implementation (§III-C).

A. Design Considerations for CPU Scheduler

1) Scheduling dimensions: The CPU scheduler must co-optimize the following key dimensions.
(i) Spatial Core Allocation. Given a multi-core platform, the scheduler must determine how the CPU cores are divided across different components of an app.
(ii) Temporal CPU Allocation. For a core to which multiple components are assigned, the scheduler must decide how many CPU slices must be allocated to each of them.
(iii) Execution Rate. For each component, the scheduler must decide the rate at which it is triggered (or executed).

2) Scheduling granularity: We use the term *subchain* to refer to a series of DAG nodes within a chain that runs at the same rate in an event-driven manner (with the output from one node triggering the next). Two nodes in a chain would belong to different subchains if there is value to running them at different rates. For example, for 2D navigation, it makes semantic sense for LP to output a new velocity only upon

[1] We do not know what scheduling knobs and policies are used in closed-source / proprietary systems, which are heavily engineered nonetheless.

every new input from LM that carries updated knowledge of the robot's immediate vicinity. Therefore, LP and LM belong to the same subchain. On the other hand, it is useful to run GP at a higher rate than GM (which is computationally more expensive), allowing the global trajectory to be updated based on updated position estimates from GM. [2]

Since all nodes within a subchain must run at the same rate, a subchain forms the natural granularity at which we can make the scheduling decisions listed above. The scheduler actively triggers only the first node in each subchain, and each of the remaining nodes in the subchain can be event-triggered when the preceding node produces a new output.

3) Performance metrics: In Catan, the user specifies semantic requirements – but what is the right level of granularity for those requirements? More specifically, what is a practical choice of performance metrics? The highest-level goal is an application objective like avoiding collisions or tracking a moving object, but it is difficult for a scheduler to directly reason about these long-term goals. We therefore choose to have the user specify DAG-level metrics that the scheduler can directly optimize. [3] We use two principal metrics. Our first metric corresponds to how quickly and frequently new inputs are processed along a given *chain* in the DAG, while the second corresponds to the processing rate of a given *node*. We describe these metrics below:

Chain Response Time. It is the worst-case time from the moment a change occurs in the environment, to the moment a reaction is produced at the sink of the chain. We define response time for consecutive pairs of inputs (i_{k-1} and i_k) that are fully processed by the chain to produce new outputs at the sink (o_{k-1} and o_k respectively). In the worst case, an environmental change occurs immediately after the source of the chain captures a previous input i_{k-1} — the system will not react to the change until the next input i_k is captured and processed by the chain to produce the output o_k. Chain response time is, therefore, the time difference between when the sink produces a new output o_k, and when the source captures the *previous* input i_{k-1}.

For example, a navigating robot's ability to avoid collisions is correlated with the response time along the chain 'scans $\rightarrow$ LM $\rightarrow$ LP $\rightarrow$ base', which captures how quickly the robot can react to dynamic obstacles in its vicinity.

Node Throughput. It is the rate at which a given node processes its inputs to produce new outputs. Ensuring that the latest position estimate is available for generating robot's trajectory requires high throughput for the GL node (that feeds into multiple other nodes) in the navigation app.

4) Semantic preferences and constraints: Having defined the performance metrics above, we can now describe how the user specifies semantic requirements in terms of those metrics. Certain performance metrics are more important than others. For example, in order to avoid collisions during

[2] Note that each node belongs to only one subchain.

[3] We believe these metrics are practical choices as they are understandable to an application developer, allowing them to express useful domain knowledge, while still allowing the developer to avoid dealing with low-level dynamic resource allocation.

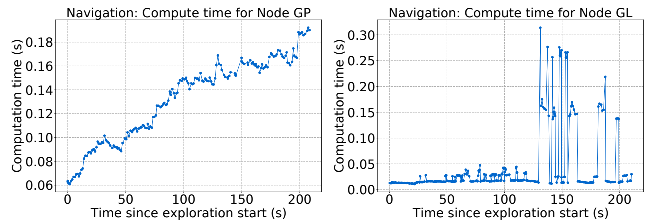

Fig. 2: Variation in the CPU usage over time of two components in the navigation application: GP (left) and GL (right). Experiment setup described in §IV.

navigation, minimizing response time along the local chain must be given the highest priority, followed by ensuring high GL throughput (to avoid faulty navigation commands derived from stale position information), followed by a sufficiently high throughput for the remaining nodes to minimize exploration time.

An app may further require an upper bound on a chain's response time or a lower bound on a node's throughput, based on semantic requirements. In addition, a node's throughput can be upper bounded by the hardware limits of the sensors and actuators/display.

5) Variations in Compute Usage and Availability: The scheduler must dynamically handle the following:

Variation in compute usage over time. We use node computation time to capture the amount of CPU consumed by a node every time it runs. Computation time of some nodes may increase over time. For example, as the navigating robot covers more area, the size of the global map increases, which increases the time taken by (i) GM to update and generate the global map, and (ii) GP to plan the robot's trajectory based on the map (as shown in Figure 2 (left)).

Input-dependent variability. We observe that the GL node for robot navigation has a bimodal computation time — for each laser scan, it either does a quick check and discards it if there is no new information, or processes the scan to update the pose correction. For the frames that were fully processed by GL, we additionally observe occasional spikes in computation times (as shown in Figure 2 (right)). A closer analysis revealed that these spikes arise from "loop closure", where the robot runs an expensive non-linear optimization on re-visiting a location [14].

Variation in available compute resources. The amount of resources available to the application would vary across different hardware platforms. Even for a given platform, the amount of available compute resources may change over the duration of the application run — e.g., as a robot's battery starts draining over time, the number or frequency of CPU cores can be reduced to save power.

B. Catan Design Details

We design Catan, a CPU scheduler for robot application, based on the above considerations. It provides an interface for the app developers to specify (i) the DAG structure (nodes and edges), (ii) grouping of nodes into subchains, (iii) which metrics must be optimized (i.e., response times of which chains and throughput of which nodes) and the corresponding

constraints and weights across these metrics. These inputs depend on app semantics and remain unchanged across deployment platforms and over time. Catan periodically tracks the per-node computation times and compute availability, and adaptively determines the schedule (node execution rates and CPU allocation), so as to best meet the specified semantic requirements. It thus eases app development burden by translating *static* semantic requirements specified by the app developers into *dynamic* low-level system schedule.

We model our scheduling problem in Catan as a constrained optimization problem, wherein the objective is a weighted linear combination of the DAG-level metrics (i.e., chain response times and reciprocal of node throughputs). Catan schedules the app subchains, so as to optimize the specified objective and meet any specified constraints.[4]

Catan breaks the scheduling decisions into two stages — the first stage determines the mapping between subchains and cores (§III-B.1), and the second stage makes scheduling decisions at per-core and per-subchain granularity (§III-B.2). We first outline our approach under the assumption that the computation time at each node and the number of cores stays constant, and discuss how we handle variability in §III-B.3.

1) Stage I: Core Allocation: For a DAG with N subchains on a system with K cores, the first stage of Catan outputs a boolean matrix of size $N \times K$, where an element a_{ij} is 1 if subchain i is allowed to execute on core j. Given the exponential solution space, we add a constraint for tractability: each subchain either runs alone on one or more cores or shares a single core with other subchains (i.e., two or more subchains do not share two or more cores). We also add a trivial constraint: each subchain should be assigned to at least one core, and vice versa.

We estimate the period p_i of subchain S_i, as a function of a_{ij}. A subchain S_i's period captures its execution rate: S_i processes a new input every p_i time units. We consider two cases, and make simplifying assumptions in each case for computational tractability (we relax these assumptions when making finer-grained decisions in Stage II in §III-B.2):
(i) If multiple subchains share a single core x, we assume each subchain gets an equal share of CPU. The period of a subchain S_i that is assigned to core x will then be the sum of computation time of all nodes in S_i multiplied by the number of subchains that are sharing the core x.
(ii) If a subchain S_i is assigned k_i number of cores ($k_i \geq 1$), we assume all nodes in S_i have the same degree of parallelism and their compute scales perfectly with it. We use the analytical formulation described later in §III-B.2 to compute the period p_i from the computation time of each node in the subchain.

Thus, for a given core allocation a_{ij}, we get approximate periods p_i for each subchain S_i. We next estimate the chain-level and node-level metrics as a function of p_i for each subchain which, in turn, allows us to estimate the final objective function as a function of a_{ij}. For a subchain S_i with period p_i, the throughput of each node in S_i is equal to the subchain's throughput, i.e., $1/p_i$. Table I lists how we estimate chain-level metrics, for a chain C_t comprising of subchains $S_{t1} \rightarrow S_{t2} \rightarrow S_{t3}..S_{tn}$. The average chain throughput is bottlenecked by its slowest subchain. We estimate the worst-case chain latency as follows: at each subchain S_{ti} in the chain, a new output $o_{t(i-1)}$ from the preceding subchain ($S_{t(i-1)}$) might have to wait for a whole period, (i.e., p_{ti}) for S_{ti} to finish its current execution (except for at S_{t1} where there's no waiting time), and it will take p_{ti} time for S_{ti} to fully process $o_{t(i-1)}$ and produce a new output. We then estimate chain response time as the sum of the chain latency and period (reciprocal of throughput).

Chain C_t: $S_{t1} \rightarrow S_{t2}..S_{tn}$, with periods $p_{t1}..p_{tn}$

Metric	Approximation
Chain Latency	$p_{t1} + \Sigma_{x=2}^{n} 2 * p_{tx}$
Chain Throughput	$1/(max_{x=1}^{n} p_{tx})$
Chain Response Time	Latency + 1/Throughput

TABLE I: Approximate formulae for chain-level metrics.

Using the models above, we formulate a Mixed Integer Linear Program (MILP) that solves for a_{ij} such that the specified objective function is optimized.

2) Stage II: Per Core and Per Subchain Scheduling: Given the subchain to core mappings from our first optimization stage, the next stage makes finer-grained scheduling decisions for each subchain and core. We consider two cases:
Single subchain on one or more cores. For a subchain S_i comprising of nodes $\{n_{i1}, n_{i2}, \ldots n_{im}\}$ with k_i assigned cores, the scheduler computes the time period p_i of the source node (n_{i1}) and the degree of parallelism (q_i), such that the response time along the subchain is minimized. The degree of parallelism indicates the number of cores any node in the subchain can use in parallel.

The period of the subchain corresponding to a given degree of parallelism $q \in [1, k_i]$ is given by:

$$p_i(q) = max(max_{j=1}^{m}(c_{ij}^{q}), \Sigma_{j=1}^{m}(c_{ij}^{q})/\lfloor k_i/q \rfloor) \quad (1)$$

where c_{ij}^{q} is n_{ij}'s computation time when it uses at most q cores (only a subset may be designed to use all q cores). Intuitively, the above formula captures that a subchain's throughput can either be limited by its slowest node or by the total amount of available compute resources.

The corresponding response time of the subchain, for a given q, can be analytically computed as $(\Sigma_{j=1}^{m}(c_{ij}^{q}) + p_i(q))$. We iterate over all possible values of $q \in [1, k_i]$ and select the value q_i (and the corresponding p_i) that results in the lowest response time for the subchain. This rate allocation achieves subchain response time within $2\times$ the optimal, and equal to the optimal if all the nodes in the subchain can only use a single core (proof omitted for brevity).
Multiple subchains on a single core. For subchains $S_1, S_2...S_n$ assigned the same core, the scheduler must determine their periods and temporal allocation of CPU across them. We construct a periodic schedule, and execute f_i fraction of subchain S_i in each period, such that $\Sigma_{y=1}^{n}(f_y * c(S_y))$ equals the period of the schedule, where

[4]We allow the constraints to be soft, i.e., if the scheduler cannot meet them, it prints a warning, and aims to meet a scaled up set of constraints.

$c(S_y)$ is the sum of computation time for all nodes in S_y. The execution of each (fractional) subchain within a period follows a configurable ordering. Each subchain will finish processing one input once every $1/f_i$ periods, i.e., $p_i = \frac{\Sigma_{y=1}^{n}(f_y * c(S_y))}{f_i}$. We allow f_i to be larger than 1 for subchains that require optimizing *average* throughput (e.g., GL in robot navigation) [5], and constrain $1/f_i$ to be an integer for other subchains to allow the scheduler to control the exact throughputs. We combine the above period formula with metric estimations given in Table I, and formulate a Geometric Programming problem to compute the f_i and p_i variables such that the specified objective function is optimized under the specified constraints.

3) Handling Variations: Coarse Grained Variations. We handle coarse-grained variations over time by continually recording the computation time across all nodes and tracking the number of available cores. We re-compute the stage II scheduling decisions (§III-B.2) periodically using the 95%ile computation time for each node measured over the previous 50 values in our implementation. We handle multimodal computation times by taking the weighted sum of 95%ile computation times across the different modes. We invoke the more expensive stage I optimization (§III-B.1) less frequently.

Fine Grained Variability. In spite of periodically adapting the scheduling decisions, a node may still exceed its expected (previously measured) computation time. To handle these situations, the scheduler implements a priority-based stealing mechanism, wherein if nodes A and B share a core, and B has lower priority than A, then in each period of the schedule, the scheduler allocates B's CPU time to A if the last output from A was not received at its expected time period. We infer node priorities from the specified weights, and also provide the option to specify these as semantic inputs.

C. Implementation Details

We implement Catan scheduler as a ROS node.

Initialization. Catan requires the DAG structure and the thread ids corresponding to each node, since they are required to enforce fine grained scheduling. We modify ROS communication library to expose the ids of all the threads it uses under-the-hood to handle communication between nodes.[6]

Bootstrapping. Catan takes the application DAG and constraints/weights as input, and bootstraps the core allocation by assigning equal compute time to all the nodes and running the Stage - I solver. Based on the output, Catan spawns a thread TQ_j for each core j with multiple subchains, and a thread TS_i for every subchain S_i assigned to one or more cores, to handle the per-core and per-subchain scheduling respectively. It bootstraps the per-core fractional schedule by assigning a fraction of 1 to all subchains. The bootstrapped configuration lasts only for the first 2s, until actual node computation times are measured and the solver is invoked. Lastly, it spawns a dynamic re-optimization thread which periodically updates the scheduling decisions. All the scheduler threads (TQ_j and TS_i) are assigned the SCHED_FIFO policy with a very high priority of 4, except for the re-optimization thread which runs with the SCHED_OTHER policy, so as not to interfere with the application's execution.

Runtime. At runtime, each scheduler thread TS_i triggers the first node of the subchain i at the analytically computed time period. The threads TQ_j enforce the core j's periodic fractional schedule using two mechanisms. First, it triggers the first node in each subchain s_y once every period if $f_y \geq 1$, and once every $1/f_y$ periods otherwise. Secondly, in each period of the schedule, it assigns SCHED_FIFO policy with the lowest priority of 1 to all the node threads, and iteratively bumps up the SCHED_FIFO priority (to 2) of (all the threads of) each subchain for the computed fraction of its time within the period [7] As described in §III-B.3, the controller can enforce priority-based stealing within each period.

Dynamic Re-optimization. We periodically re-solve for new optimal scheduling decisions, based on the latest compute time estimates of all nodes (as discussed in III-B.3). We implemented both the MILP and GP formulations using the Mosek Fusion C++ library[16]. It takes our solver 24-26ms to solve the GP for the navigation application and 5-6ms for VR. Solving the MILP is more expensive, requiring 60ms for the navigation DAG for 2 cores. We invoke both the solvers 2s after initialization, and then re-run Stage I every 20s, and Stage II every 5s. Increasing the time periods can decrease the overhead of running the solvers. We chose the periods so as to keep the overhead roughly under 0.05 CPU cores[8].

IV. Experiments and Results

We now evaluate how Catan influences the performance of the 2D navigation app.

A. Experiment Setup

We use Stage [5] to simulate a P3AT robot [17]. The simulator feeds laser scans and odometry into the navigation application, which is implemented on top of ROS [13]. The application feeds the robot's velocity back into Stage. We configure the simulator to publish laser scans and odometry at 50 Hz. We conduct experiments on two maps: (i) Map 1 (Figure 3(left)) requires the robot to move around a dynamic obstacle at the entrances of a room (this models realistic scenarios where collisions with dynamic obstacles may occur at narrow doorways). This map allows us to evaluate the ability of the robot to avoid collisions with dynamic obstacles under different configurations. (ii) Map 2 (Figure 3(right)) requires the robot to explore a larger area. We do not

[5]We cannot assume a fixed time period of input processing for nodes that buffer and batch process inputs, and instead consider their average throughput. Such nodes can either be auto-identified (given support from framework) or can be explicitly identified by the developer.

[6]Note that our approach can be generalized to ROS 2 as well, we leave that to future work.

[7]One could use Linux' SCHED_DEADLINE scheduler to implement Catan's scheduling decisions instead of SCHED_FIFO, but we found it to be too inflexible for handling variations.

[8]The process of selecting the solver periods can easily be automated based on this criteria by tracking the solvers' compute usage.

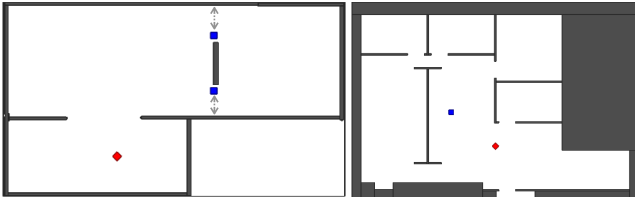

Fig. 3: Map 1 (left) and Map 2 (right) used for navigation experiments. The blue squares represent the obstacles and the red square represents the robot. The obstacles in Map1 are dynamic, and move along the two doorways.

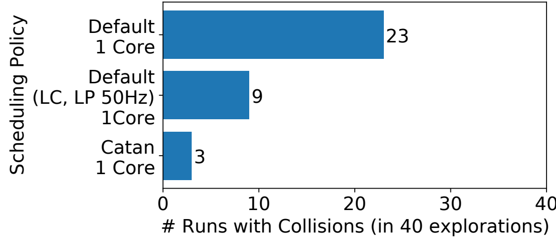

Fig. 4: No. of runs with collisions (out of 40) on Map1.

add any dynamic obstacles to this map, focusing on other performance metrics. Unless otherwise specified, all nodes run on a system with one core (it is common for robots to have single / dual core on-device compute [17]).

Catan Inputs: We provide the following inputs to Catan (that remain unchanged over time and across scenarios). We configure Scan → LM → LP as a single subchain, and configure the remaining nodes as individual subchains that can run at different rates. We model the objective function as weighted sum of response times for the different chains in the navigation DAG, and the average output period for GL. We assign the highest weight of 1.0 to the response time for the local chains (involving LM and LP), as they are responsible for avoiding collisions. We use a weight of 0.5 for GL's average output period (to ensure freshness of pose estimates and odometry for path planning). We assign a small weight of 0.005 to the response time along the chains that include the three remaining processing nodes (GM, GP, NC) sourced at GL (to ensure that these nodes are not starved as the local chain and GL are prioritized). We also add the following app specific constraints into our optimization formulation: (i) GL's average throughput must be at most 50 Hz, based on the maximum frequency at which the simulator can publish laser scans. (ii) GP's throughput must be at most 1Hz.

B. Results

1) Comparison with manually-tuned configuration: We compare the performance of Catan with the default configuration that was manually-tuned by the developer of the navigation app [13]. In line with the interface currently exposed by ROS, the default configuration only specifies the execution rate of each node (set to 5Hz for LM, 10Hz for LP, 0.2Hz for GM, upto 1Hz for GP, 10Hz for NC). It uses Linux' default scheduling policy SCHED_OTHER (Completely Fair Scheduler [18]) that shares CPU time equally among all nodes.

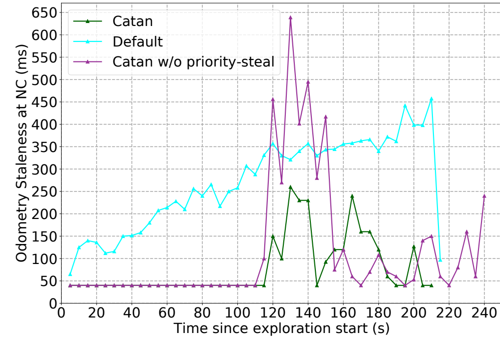

Fig. 5: Tail odometry staleness at NC on Map2 (median over 20 runs).

Map1 Results. Figure 4 compares the number of experiments runs (out of 40) in which the robot collides with an obstacle. With the default configuration (which runs the LC and LP at low rates of 5Hz and 10Hz), more than 50% runs suffer from collisions. Increasing the rate of LM and LP nodes to 50 Hz reduces the number of collisions to 9. With a high weight assigned to response time along the local chain, Catan is able to run the local chain at an even higher rate (125 - 170Hz), reducing the number of collisions to only 3. This highlights the impact of scheduling decisions on a robot's performance.

Map2 Results. Figure 5 shows the staleness of odometry used for generating navigation command. We capture this by recording the time difference between an odometry reading and when the corresponding pose is used by NC. For each experiment run, we aggregate the staleness values collected over time buckets of 5s by computing the 95%ile values. For each 5s bucket, we report the median of these values across 20 runs. We find that, in general, Catan has lower odometry staleness than the default policy. When GL's compute usage spikes up (especially during loop closures) and as the compute load increases over time (due to increased CPU usage for GM and GP), Catan is able to explicitly prioritize GL over GM and GP due to its periodic adaptation and priority-based stealing. We isolate the impact of priority-based stealing by disabling it for GL (results shown with the pink line in Figure 5). Disabling priority-based stealing increases staleness, as the the scheduler cannot handle sudden spikes in GL due to loop closure.

The above results show how Catan performs better than the default configuration with respect to avoiding collisions and ensuring freshness of odometry readings. Improvements in these more critical metrics come at the cost of a small reduction in the area exploration rate – the time taken to explore 90% of the area increases by 10.9%, as GM is allocated smaller amount of resources in Catan than in the default scheme[9]. Thus, Catan achieves the desired trade-offs, as per the configured weights and priorities.

2) Adapting to temporal variations: We next highlight the importance of updating scheduling decisions over time, as resource usage changes. We extend the navigation DAG by adding another chain that runs an object detector on

[9]We omit the comparison of low-level metrics for brevity, but Catan is able to achieve better response time for all chains.

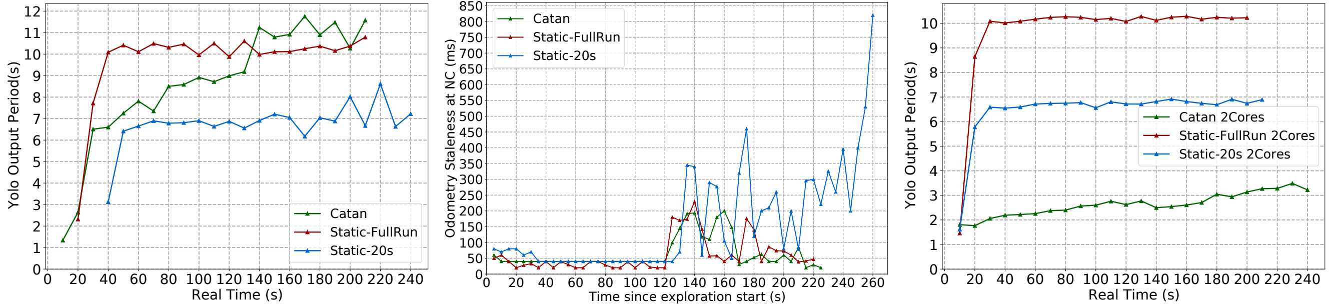

Fig. 6: Left: YOLO chain output period (reciprocal of throughput) on a 1 core system. Center: Tail odometry staleness at NC on a 1 core system. Right: YOLO chain output period on a 2 core system. All the numbers are based on aggregating 20 runs on Map 2.

camera images as the robot moves around (this is representative of robotic tasks, such as ObjectGoal [19]). The chain comprises of an image pre-processing node and a YOLO object detection node [20][10]. To model realistic timings in the simulated environment, we feed images from a real image dataset [22] to the object detection node. We augment our objective function for Catan to include response time along this chain as another metric with a very low weight of 0.0005. We desire the objects to be detected in real time at a high rate (i.e., high YOLO chain's output period), but that should not come at the cost of not being able to navigate well (i.e., low odometry staleness at NC).

We run this extended navigation application on a system with one core. We conduct experiments on Map2 and compare Catan with two baselines that use a static configuration (that is not updated over time): (i) Static-FullRun that uses the same optimization problem as Catan based on the tail computation time for each node over an entire experiment run on Map2 using a single core. It computes the schedule once, and does not re-solve it periodically. Note that since computation times of GM and GP nodes increase as the run progresses, this scheme would use a schedule derived from over-estimated computation time towards the beginning of the run. (ii) Static-20s is similar to Static-Full, but uses tail computation times of each node over the first 20s of the run. This scheme uses a schedule derived from an under-estimated computation time in the later half of the run.

Figure 6 (left) shows the average output period for the YOLO chain (as a measure of reciprocal of its throughput) over time – lower is better. To aggregate data across 20 runs, we first take the average period (measured as the time gap between two consecutive outputs from the YOLO chain) over 10s buckets for each run, and then plot the median of these average values for each bucket. We find that Static-FullRun, which overestimates the computation time of navigation nodes in the first half of the run, assigns a low rate to the YOLO chain (resulting in high output period in the first half). Static-20s, which underestimates the computation of navigation nodes, assigns a high rate to YOLO – as we discuss next, this comes at the cost of worse performance on a more critical metric. Catan, with its dynamic re-solving assigns a high rate (low period) to YOLO at the beginning of the run, and then gradually decreases the rate (increases the period) over time.

Figure 6 (center) shows the odometry staleness at NC across the three schemes. While Static-FullRun and Catan both perform similarly on this metric, Static-20s exhibits higher staleness towards the second half of the run. By using under-estimated computation times for the second half, Static-20s mis-allocates resources, giving a smaller share of CPU time to GL, GM, and GP, and a more than necessary amount of CPU resources to the local chain nodes, NC, and YOLO. This reduces the effective throughput of GL (in spite of priority-based stealing) and increases odometry staleness.

All three schemes performed similarly in terms of the area exploration rate and with respect to avoiding collisions on Map1, and so we omit detailed results.

3) Adapting to change in resource availability: Our experiments so far used a single core. We next ran the extended navigation application (with the YOLO chain) on Map2, on a system with two cores, to evaluate how well Catan auto-scales. As baselines, we configured the system to use the same node execution rates as those computed by the two baselines above (i.e., Static-FullRun and Static-20s with static schedules derived from node computation times on Map2 on one core), relying on default Linux policies to schedule the nodes across two cores. Figure 6 (right) compares the median of average output period for the YOLO chain over time across Catan and the two baselines on 2 cores. We find that Catan is able to make efficient use of the extra core – it dynamically allocates the core to just the local chain in the first half, and the local chain and GP in the second half. It is, thus, able to run YOLO at a higher rate (and lower output period) than other static baselines that do not increase the execution rates.

V. Related Work

DAG abstraction is common across other systems, including real-time (e.g. [23], [24], [25], [26], [27], [28], [29]), sensor nodes (e.g. [30], [31]), and distributed stream processing and dataflow systems (e.g. [32], [33]). Robot systems are distinct from these. As we show in §III-A, the

[10]While YOLO execution is usually GPU based, we don't run it on GPU since we focus on CPU scheduling. We leave CPU/GPU co-scheduling [21] to future work.

compute usage in robot systems exhibits a high degree of variability over time – an aspect that conventional real-time systems (based on fixed periodicity) do not handle, including those that consider robotics applications [28], [29], [27].

Compared to traditional sensor nodes, robot systems have higher processing complexity and variability, which emphasizes the importance of proper CPU scheduling. Distributed stream-processing systems focus on cluster-wide resource management to handle an incoming stream of queries. We instead focus on CPU scheduling within a single computer on a robot, that runs a single long-running app – such a system differs in its scheduling knobs and requirements.

Lutac et al. highlight the scalability limits of ROS, as compared to an Erlang-based framework [34]. Another line of work focuses on reducing the inter-node communication latency in ROS/ROS 2 [35], [36]. ROS 2 supports using real time OS such as PREEMPT_RT, which makes the kernel fully preemptible. These efforts are orthogonal our work, and look at complementary aspects of robot systems.

VI. Conclusion

In this work, we develop Catan, a scheduler to manage on-device CPU resources for robotics applications. Catan provides a high-level interface for app developers to specify their requirements without worrying about low-level system configuration knobs. Our evaluation, using 2D navigation as a case-study shows (i) Scheduling decisions impact a robot's performance (e.g., its ability to avoid collisions, the freshness of sensor data that it reacts to). (ii) It is important to adapt scheduling decisions over time – Catan is able to do so by dynamically re-solving the schedule. (iii) It is important to adapt to sudden fine-grained variability in resource usage – Catan is able to do so via priority based stealing. Catan therefore lowers the barrier (in terms of systems expertise and engineering effort) for developing high-performance robotics apps, and could enable achieving higher performance on resource-constrained platforms.

Acknowledgements

This work was supported by Intel, AG NIFA under grant 2021-67021-34418, the Applications Driving Architectures (ADA) Center, a JUMP Center co-sponsored by SRC and DARPA, the DARPA DSSOC program, and NSF under grant 2120464.

References

[1] "LoCoBot." http://www.locobot.org/.
[2] "Terrasentia : The automated crop monitoring robot." https://blog.plantwise.org/2018/07/17/terrasentia-the-automated-crop-monitoring-robot/.
[3] "Turtlebot3 Waffle Pi." https://www.robotis.us/turtlebot-3-waffle-pi/.
[4] "ROS." https://www.ros.org/.
[5] "Stage Simulator." http://playerstage.sourceforge.net/doc/Stage-3.2.1/.
[6] M. Huzaifa, R. Desai, S. Grayson, X. Jiang, Y. Jing, J. Lee, F. Lu, Y. Pang, J. Ravichandran, F. Sinclair, B. Tian, H. Yuan, J. Zhang, and S. V. Adve, "Illixr: Enabling end-to-end extended reality research," in *IEEE IISWC*, 2021.
[7] A. Partap, S. Grayson, M. Huzaifa, S. Adve, B. Godfrey, S. Gupta, K. Hauser, and R. Mittal, "On-device cpu scheduling for sense-react systems," *arXiv:2207.13280*, 2022.
[8] "OROCOS." http://www.orocos.org/.
[9] "LCM." https://lcm-proj.github.io/.
[10] "CORBA." https://www.corba.org/.
[11] P. Fitzpatrick, E. Ceseracciu, D. Domenichelli, A. Paikan, G. Metta, and L. Natale, "A middle way for robotics middleware," in *Journal of Software Engineering for Robotics*, 2014.
[12] "ROS 2." https://docs.ros.org/en/galactic/index.html.
[13] "ROS Navigation2D Application (Source: https://github.com/skasperski/navigation_2d)." http://wiki.ros.org/nav2d.
[14] W. B. Sebastian Thrun and D. Fox, *Probabilistic Robotics*. MIT Press, 2005.
[15] S. Russell and P. Norvig, *Artificial intelligence: a modern approach 4th edition*. 2020.
[16] "Mosek Fusion - Convex Programming for C++." https://docs.mosek.com/9.2/cxxfusion/index.html.
[17] "P3AT Robot." https://www.generationrobots.com/media/Pioneer3AT-P3AT-RevA-datasheet.pdf.
[18] "Linux CFS." https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.
[19] P. Anderson, A. Chang, D. S. Chaplot, A. Dosovitskiy, S. Gupta, V. Koltun, J. Kosecka, J. Malik, R. Mottaghi, M. Savva, *et al.*, "On evaluation of embodied navigation agents," *arXiv preprint arXiv:1807.06757*, 2018.
[20] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *IEEE CVPR*, 2016.
[21] Y. Suzuki, T. Azumi, S. Kato, and N. Nishio, "Real-time ros extension on transparent cpu/gpu coordination mechanism," *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 184–192, 2018.
[22] M. Everingham, L. Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *Int. J. Comput. Vision*, 2010.
[23] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang, "Making openvx really "real time"," in *IEEE RTSS*, 2018.
[24] Y. Yang, A. Pinto, A. Sangiovanni-Vincentelli, and Q. Zhu, "A design flow for building automation and control systems," in *IEEE RTSS*, 2010.
[25] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *DAC*, 2007.
[26] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio, "Rosch:real-time scheduling framework for ros," in *IEEE RTCSA*, 2018.
[27] M. Verucchi, M. Theile, M. Caccamo, and M. Bertogna, "Latency-aware generation of single-rate dags from multi-rate task sets," in *IEEE RTAS*, 2020.
[28] H.-S. Choi, Y. Xiang, and H. Kim, "Picas: New design of priority-driven chain-aware scheduling for ros2," *2021 IEEE 27th RTAS*, pp. 251–263, 2021.
[29] Y. Yang and T. Azumi, "Exploring real-time executor on ros 2," in *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, pp. 1–8, 2020.
[30] K. Lorincz, B.-r. Chen, J. Waterman, G. Werner-Allen, and M. Welsh, "Resource aware programming in the pixie os," in *ACM SenSys*, 2008.
[31] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger, "Eon: A language and runtime system for perpetual systems," in *ACM SenSys*, 2007.
[32] M. J. Sax, M. Castellanos, Q. Chen, and M. Hsu, "Performance optimization for distributed intra-node-parallel streaming systems," in *IEEE ICDEW*, 2013.
[33] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *ACM EuroSys*, 2007.
[34] A. Lutac, N. Chechina, G. Aragon-Camarasa, and P. Trinder, "Towards reliable and scalable robot communication," in *Proceedings of the 15th International Workshop on Erlang*, 2016.
[35] C. Iordache, S. M. Fendyke, M. J. Jones, and R. A. Buckley, "Smart pointers and shared memory synchronisation for efficient inter-process communication in ROS on an autonomous vehicle," in *IEEE IROS'21*.
[36] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communications for real-time robotic applications," *CoRR*, vol. abs/1809.02595, 2018.