Author's Copy

Context-Aware Task Handling in Resource-Constrained Robots with Virtualization

Ramyad Hadidi§
Rain AI
ramyad@rain.ai

Nima Shoghi Ghaleshahi
Georgia Tech
nimash@gatech.edu

Bahar Asgari§
University of Maryland
bahar@umd.edu

Hyesoon Kim
Georgia Tech
hyesoon.kim@gatech.edu

***Abstract*—Intelligent mobile robots are critical in several scenarios. However, as their computational resources are limited, mobile robots struggle to handle several tasks concurrently while guaranteeing real timeliness. To address this challenge and improve the real-timeliness of critical tasks under resource constraints, we propose a *fast context-aware* task handling technique. To effectively handle tasks in *real-time*, our proposed context-aware technique comprises three main ingredients: (i) a dynamic time-sharing mechanism, coupled with (ii) an event-driven task scheduling using reactive programming paradigm to mindfully use the limited resources; and, (iii) a lightweight virtualized execution to easily integrate functionalities and their dependencies. We showcase our technique on a Raspberry-Pi-based robot with a variety of tasks such as Simultaneous localization and mapping (SLAM), sign detection, and speech recognition with a 42% speedup in total execution time compared to the common Linux scheduler.**

***Index Terms*—Edge AI, Software, Mobile Robots, Middleware and Programming Environments, Reactive and Sensor-Based Planning,**

I. Introduction & Motivation

Unlike conventional industrial or commercialized robots that perform a set of pre-programmed and routine tasks, intelligent mobile robots manipulate their environment using their perception and physical resources to achieve a myriad of goals. Such robots must be capable of dynamically switching between navigation, planning, reasoning, recognition, and sensing their environment. Intelligent robots need to interact with a dynamic, complex, and non-deterministic world. These robots must execute numerous tasks such as controlling their physical resources (*e.g.*, arms), understanding data derived from sensors, or executing perception and planning.

Intelligent robots are always in a never-ending conflict between available computation resources, their energy storage, and the tasks at hand. This conflict is particularly emphasized in resource-constrained robots because even the concurrent execution of a few rudimentary tasks is extremely demanding with only a few processing cores. For example, a Raspberry Pi with only four cores could be fully utilized by the operation system (OS), processing the data from a single sensor, and simple navigation and control algorithms. Adding more sensors and tasks only causes the robot to miss real-time deadlines. Thus, ensuring efficient handling of critical tasks and meeting critical deadlines is the key challenge for resource-constrained robots.

To extend the capabilities of resource-constrained robots and meet real-time demands, the common practices are adding extra hardware or utilizing cloud/fog computation [1]–[7]. However, in several scenarios, adding new hardware is either infeasible or uneconomical. For example, adding extra processing units to a lightweight drone requires heavier batteries, which in turn demands stronger motors. Further, cloud and fog are not always available. Additionally, privacy concerns limit the suitability of cloud-based computation.

To enable intelligent mobile robots to efficiently utilize limited resources, we propose a *context-aware* task handling technique that simplifies the world and planning tasks by dynamically reducing the number of tasks in a certain context to only the critical ones. For example, limited human-robot interaction is expected while the robot is performing an already assigned task. This technique enables resource-constrained robots to efficiently perform manifold functionalities while meeting their real-time constraints.

To be effective in handling tasks using our context-aware technique, we propose using a *virtualized execution* that (i) integrates several tasks while providing dynamic, low-cost, and kernel-level control over the scheduling policy; (ii) enables easier context-aware implementation by providing manageable control over tasks; and (iii) provides a uniform and practical environment for building new robots in the community.

For experiments, we use a custom-built Raspberry-Pi-based robot using an iRobot Roomba [8] equipped with one Raspberry Pi 4 (RPi4) [9] as the only processing unit. Our iRobot, shown in Figure 1, has several sensors (*i.e.*, LIDAR, inertial measurement unit (IMU), cameras, and microphone), and control devices (*i.e.*, motors for navigation, robotic arm, and speakers). For software, we use Docker [10], a popular virtualization tool, and implement our context-aware technique to collect and process sensor data, simultaneous localization and mapping (SLAM), voice recognition, and sign recognition. Our contributions are as follows:

- *Context-aware* task planning to effectively use the limited resources and hence extend the number of tasks that a robot can handle.
- OS-level *dynamic* time-sharing to implement the context-aware scheduling in real-time.

This work was supported in part by the NSF grant number 2103951.

§This work was done when the authors were affiliated with Georgia Tech.

2767-9918/23/$31.00 ©2023 IEEE
DOI 10.1109/EDGE60047.2023.00047

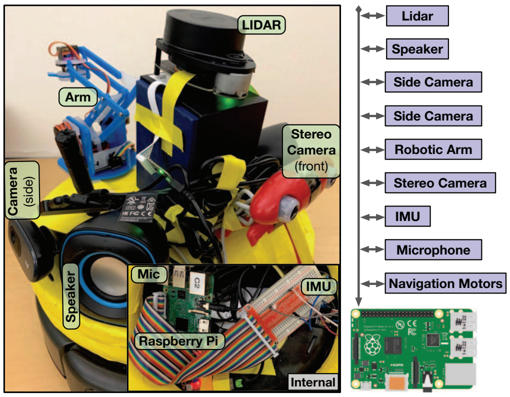

Fig. 1: Modified iRobot with RPi4 and additional sensors.

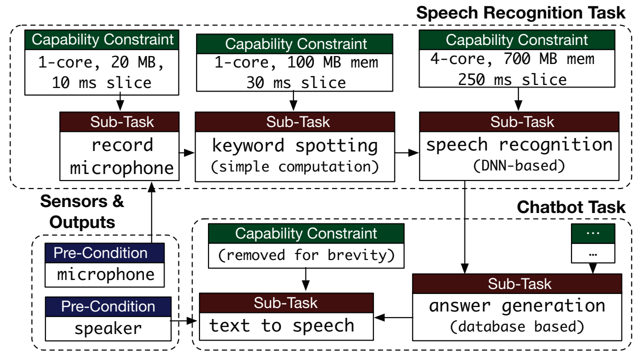

Fig. 2: Graph representation for speech recognition and chatbot tasks. A representation of $(\{\varepsilon_{t\,i}\}_{i=1:n}, \rho_t)$.

- *Event-driven* task scheduling to be mindful of using the limited resources for scheduling itself.
- *Lightweight virtualized execution*, using Docker and reactive programming paradigm to enable easily manageable and yet kernel-level dynamic task scheduling policy.

II. Related Work

Real-Time Operating Systems & Scheduling Polices: The operating system (OS) schedules applications either based on the order of the events (event-driven), order of processes (*e.g.*, round-robin), or time sharing. To minimize the latency of accepting a process real-time operating system (RTOS) has been designed. RTOSes have preemptive schedulers [11] (*e.g.*, fixed-priority preemptive scheduling). Since optimal scheduling is an NP-complete problem [11], [12], even RTOSes can not guarantee hard deadlines. Therefore, hard real-time robotic systems usually either implement fixed schedulers (*e.g.*, commercial drones) or use extra dedicated cores to provide enough computation performance. As neither solutions align with our goal of *context-aware* task handling using *limited resources*, this paper tunes the OS scheduler (Section IV-C).

Robot Operating System (ROS): Robot Operating System (ROS) [13] is a popular example of a robotic environment to manage the complexity of various aspects of robotic systems, from simulation to hardware implementation. ROS also manages the process execution, while providing standalone libraries for hardware components. As ROS does not offer real-time operations, ROS2 has been upgraded to handle hard real-time tasks [14] by prioritizing real-time threads and avoiding the sources of non-determinism such as memory allocation [15]. Nevertheless, ROS2 does not support dynamically changing priorities in runtime. Moreover, ROS2 requires additional kernel support [16], still in early development.

III. Deconstructing Tasks

To design our context-aware task handling, we first categorize tasks as the following: The first category is *elemental* or atomic tasks that consist of a single event. The second category, *compound* task, is *decomposed* into multiple steps or a set of subtasks. To satisfy a *compound* task, every subtask of it must be done. The third category, *complex* tasks, are also decomposable into subtasks, whereas to satisfy a complex task, not all subtasks are required to be done.

The resulting subtasks have a set of relationships with each other, possible pre-conditions, and capability constraints. For instance, the *pre-condition* of executing the speech recognition task is to have a speech input. In this case, the speech recognition task has a *relationship* with speech input. Besides, tasks have relationships with the *capability constraint* to execute a workload within a deadline. For instance, to execute speech recognition effectively, we require full access to all the cores of the processor and a certain amount of memory. For a task t, we show such relationships with a directed graph structure, ρ, the vertices of which are subtasks/conditions/constraints and its edges are the relationships. Therefore, the pair $(\{\varepsilon_{t\,i}\}_{i=1:n}, \rho_t)$, for a task t, represents all sub-tasks, conditions, constraints, and relationships. Figure 2 illustrates an example of speech recognition and chatbot tasks. For instance, keyword spotting processes microphone recording and requires a single-core and 100 MB memory. A compiler analysis can extract this graph automatically. In the following, we present a manual low-overhead approach.

Containerizing Modules: In the first step, each independent task is wrapped as containerized modules implemented as Docker [10] containers. Docker implementations are easy to configure and distribute. Meanwhile, since the lower-level OS abstraction and common libraries and dependencies are shared, the overhead of using Docker is minimal.

Adding Event-Driven Initiatives: By using Reactive Extensions (RX) framework [17], next, the user adds a simple event-driven initiative for each module. This declarative configuration sets *scheduling scores* (more in Section IV-B) of modules while abstracting low-level implementation (*e.g.*, synchronization, thread-safety, concurrent data structures, and non-blocking I/O). RX provides tools for operating on, filtering, and managing asynchronous streams of data. Such streams are called *observable streams* and indicate sensor readings over time. For example, in the below example, the inertial measurement unit (IMU) sensor is a single observable stream

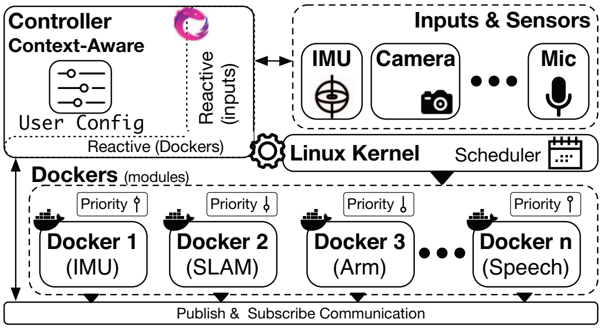

Fig. 3: High-level system overview.

of accelerometer and gyroscope readings over time. We filter this stream for receiving readings that have a non-zero vector.

Listing 1: Constructing observable streams for IMU.

```
imu.pipe(filter(lambda value:
    value["accelerometer"]["x"] != 0 and
    value["accelerometer"]["y"] != 0 and
    value["accelerometer"]["z"] != 0))
```

IV. Context-Aware Task Handling

To reduce the number of tasks at each moment to only the critical ones, we propose a *context-aware* and *event-driven* task handling technique, a high-level overview of which is shown in Figure 3. The implementation is separated into two groups: (i) the *controller*, and publish and subscribe communication medium, which manages the dynamic OS-level prioritizing/scheduling and communications among modules, respectively, and (ii) the modules that carry out the tasks, represented as containers, or *Dockers*. This section describes the first group, and how they achieve mentioned goals.

A. Publish-Subscribe Communication

Publish-subscribe communication pattern provides an efficient medium for sending and receiving data. While the modules are usually isolated and operate independently of each other, communication is necessary (*e.g.*, when the navigation requires mapping information from the SLAM). We implement a lightweight and resource-efficient publish-subscribe system among modules, in which the events can subsequently be wrapped with an RX observable stream and fed back into the controller. This enables inputting any output of a module to the controller if needed. Additionally, the shared memory interface is used to share parsed binary information among applications. This is especially efficient if such information is already serialized without the extra cost of repacking.

B. The Controller

The controller is a lightweight program for dynamically setting the priorities of modules based on the context. Based on added event-driven initiatives (Section III), the controller uses incoming sensors and modules data to dynamically decide the best scheduling scores or weights, w. To calculate scheduling scores, the controller processes the observable streams and if it detects a context change, it will change the priorities

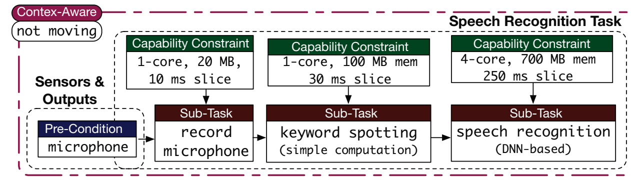

Fig. 4: The modified context-aware graph representation for speech recognition with a simple context-aware condition.

accordingly in the Linux kernel scheduler. The calculation of scheduling scores executes only when the dependent sensor/module values update, saving valuable calculation time on resource-constrained platforms.

To give users the power to define contexts, the controller can also receive input from the user with a configuration file. This configuration file first specifies the relative priority of the tasks, and second, it may extend the context-aware decisions in the controller. For instance, the user may specify that no microphone-related task should run while the robot is moving. Therefore, the speech recognition task never executes while the robot is moving, and accordingly, no microphone input is processed. In other words, the controller dynamically modifies the task graphs based on sensor events and user configuration, which leads the system to automatically adapt new scheduling scores. An example is shown in Figure 4 by modifying the speech recognition task in Figure 2.

C. Scheduling Policy

In robotics, real-time functionality and scheduling customizability are important. However, by default, Linux uses the completely fair scheduler (CFS, `SCHED_OTHER`) [18] to provide an optimal setting for desktops and servers. As reviewed in Section II, ROS2 requires extra kernel support for real-time prioritization and still is unstable. To address these issues, while relying on stable and fully-supported Linux features, we dynamically tune the parameters of the scheduler per module as we receive real-time value updates. We apply this approach with two methods: (i) using CFS policies (*i.e.*, `SCHED_OTHER`) and tuning the CFS parameters of each module, and (ii) Using real-time policies [18] (*i.e.*, `SCHED_FIFO` or `SCHED_RR`) and adjusting real-time parameters of each module. The following provides the implementation details.

1) Tuning the CFS Parameters: In CFS, we use the `cpu-period` and `cpu-quota` flags in modules to customize resource allocation. `cpu-quota` is the total amount of CPU time that a module can use in each `cpu-period`. For this feature, note that the Linux kernel should be compiled with CFS bandwidth control flag [19]. Additionally, Docker provides a combined flag, `cpus`, which allows us to directly allocate CPU resources to a container.

2) Adjusting Real-Time Parameters: Although Linux real-time (RT) policies [18] provide a better determinism to processes, the policies do not allow changes to the priorities during runtime. In detail, in both `SCHED_FIFO` or `SCHED_RR` policies, each process gets a time-slice or exclusive access to the CPU defined during the process startup (`SCHED_FIFO` runs real-time processes until it finishes. `SCHED_RR` builds on top of `SCHED_FIFO` by implementing a round-robin time-slice system based on some priority). To tune real-time scheduling parameters during runtime, we limit the total number of microseconds each module runs using Docker at real-time priority by setting the `cpu-rt-runtime` with the controller. This flag is set to a value between 0μs and 1s, and it represents the total number of microseconds reserved. We use this feature to use contextual information to change the real-time resource allocation for each module dynamically. In summary, using Docker and RX, we are able to create a dynamic two-level scheduler that is (i) event-based due to the reactive programming paradigm of the controller, (ii) time-sharing due to the Docker ability for setting time-slice value (in our case, through the controller), and (iii) dynamic because the controller changes the time-slices during runtime based on the context (supplied by the user and extracted from tasks).

D. Calculating Scheduling Parameters

Here, we describe how the controller calculates scheduling parameters for each module based on the context. Every module defines (automatically or from the user) an instantiation function that returns an observable stream representing the scheduling score as a floating-point value, or w_i. This function can take any RX stream as its input (from sensors or other modules). When the system starts, the controller calls all the instantiation functions and creates observable streams which produce floating-point values representing the scheduling score. Then, the controller combines all of the observable streams into a single observable stream. This aggregated stream is an observable stream that outputs the entire set of scheduling scores. For each container c with scheduling score w_c (specified by user and/or from context) received from the aggregated observable stream, we calculate the scheduling parameters for two scheduling types (Section IV-C) as follows. For CFS (Section IV-C1), the CPU share value (`cpus`), s_c, is calculated with the equation below, where N is the number of processors in the system:

$$s_c = N \cdot \frac{w_c}{\sum_i w_i}. \tag{1}$$

For real-time scheduler (Section IV-C2), the real-time time-slice value (`cpu-rt-runtime`), t_c, is calculated as below, where P is the time-slice period, which is 1s by default:

$$t_c = P \cdot \frac{w_c}{\sum_i w_i}. \tag{2}$$

The result of the above expression creates a dictionary that maps each container, c, to its CPU share value, s_c or real-time time-slice value, t_c. Note that the aggregated stream is also an event-driven operation with all the RX capabilities in filtering not-related events. Thus, the controller only updates the time slices if it observes any new event.

E. Planning Procedure

The formal definition of context-aware task planning is described in Procedure 1, the input of which is a list of inputs and sensors (`InputList`) and a configuration file (`ConfigFile`) that describes context-aware customization defined by the user and relative priorities of tasks. Initially, at lines 2–5, the system creates observable streams for each input, and instantiates their scheduling score functions. At lines 6–9, by reading the configuration file, the system creates a task graph, similar to Section III and Section IV-B. The graph generation is out of the scope of this paper and we reuse the common methods from robotic development environments [13]. After initializing the scheduler at Line 10, the processor only updates its setting based on the arrival of new events. On arrival of such an event that triggers the RX, the controller calculates a new scheduling score (Line 14) for that module, calculates a new dictionary of modulus and their scores at Line 15 based on Equations 1 or 2, and updates the time-slices in the OS scheduler at Line 16.

Procedure 1: Context-Aware Task Planning.

```
   Input : ConfigFile: Configuration File with Context-Aware Setting
           and Relative Priorities of Tasks.
   Input : InputList: Input & Sensor List
1  Initial
2      for input ∈ InputList do
           // Create observable stream.
3          stream ← CreateStream(input);
           // Instantiate scheduling score function.
4          InstantiateRX(stream);
5          AddToList(stream, StreamList);
6      for context ∈ ConfigFile do
           // Create context task graph.
7          graph ← CreateTaskGraph(context);
           // Create combinator observable stream
8          contextStream ← CreateStream(graph);
9          AddToList(contextStream, TaskGraphList) ;
       // Initialize an initial scheduling policy.
10     InitializeScheduling;
11     return StreamList TaskGraphList
   // On receiving an event on any stream after its bound
      reactive function.
12 Event-Based Procedure
13     OnObservableStreamEvent stream
           // Calculate scheduling score for the stream.
14         CalculateScore(stream) ;
           // Calculate new real-time time-slices values.
15         DictTimeSlices ← CalculateTimeSlices() ;
           // Update the scheduler.
16         UpdateScheduler(DictTimeSlices);
```

V. Module Implementations

This section provides implementation details of our specific tasks and their respective dataset.

1) SLAM: With a stereo camera input (Minoru3D [20]), we run the `ORB_SLAM2` [21] algorithm to localize the robot within its local environment. We use the EuRoC MAV dataset [22]. In addition to providing a stereo video input and ground truth values for error calculation, the EuRoC MAV dataset provides IMU sensor readings with accelerometer and gyroscope readings. This sensor data is used in deciding the scheduling of the SLAM module. Because of the computation demand of IMU, we implement its calculation on a separate module. These readings are fed into the controller, which creates an observable stream for each. Other modules (*e.g.*, SLAM) then subscribe to these observable streams.

2) Sign Detection: The robot processes the images from its side cameras and uses a pre-trained neural network (trained on Street View House Numbers (SVHN) dataset [23]) to decide the room/street number for the signs. For experiments, the sign detection module is using the SVHN dataset test inputs.

3) Speech Detection: The robot processes the microphone input from a microphone and uses the CMU Sphinx library (specifically, CMU PocketSphinx framework [24]) for keyword spotting and later a DNN-based implementation [25] to convert the speech to text. The speech detection module is using the Speech Commands dataset [26]. This dataset includes a labeled set of various spoken commands.

4) Navigation & Arm Control: For navigation, we send commands in the format specified in iRobot Create 2 Open Interface [27] through a serial port on iRobot. We also read several sensors and battery conditions using this serial port. The navigation commands set the speed of each wheel separately. Besides, for obstacle detection, we use a low-cost LIDAR sensor (360° laser range scanner [28]). The LIDAR provides a 360-degree scan field, 5.5hz/10hz rotating frequency with an 8-meter ranger distance. We build a simple robot arm that works with Raspberry Pi on top of our robot [29]. The arm has simple grips and four servos to control. The module sends control commands to the arm to move and grab.

VI. Experiments

We use iRobot Roomba [8] as our base navigation robot (Figure 1). We equip the robot with one Raspberry Pi 4 [9]. The power source of the Pi is derived from the battery of iRobot with a voltage converter. The computation platform for all the modules is Raspberry Pi.

A. Experiment Design & Reproducibility

In our experiments, each task executes a pre-labeled dataset while we measure its performance as the controller adjusts the scheduling parameters dynamically. We use the default Linux scheduler (CFS) as the baseline and run context-aware (CA) configuration with two CFS and RT Linux schedulers (Section IV-C), CFS CA and RT CA. To perform a fair comparison with the same set of experiments, we build an instrumentation tool, which uses a set of JSON files as timelines to artificially feed dataset inputs at certain times. In this way, we can execute the same set of experiments repeatedly with different schedulers. The timeline files are collected and constructed from a set of experiments from the measurements of the real robot. Each timeline file contains a series of inputs (*e.g.*, images for SLAM and audio for speech recognition, or arm control commands) and their respective ground truth values (if any). We use these files to feed events to the controller while it calculates the parameters for the scheduler. We design experiments with different granularity, *which includes all the implemented tasks in Section III*. Our first experiment, `exp1`, is the longest experiment with a total of three minutes of footage, while `exp2` and `exp3` experiments are with shorter duration, one minute and 15 seconds, respectively.

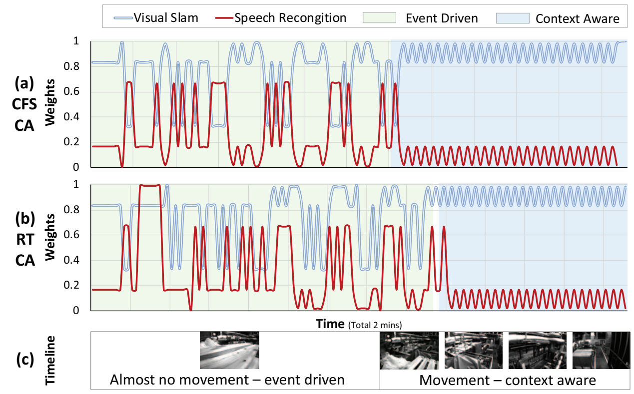

Fig. 5: Normalized scheduling scores (weights) in a simple experiment with SLAM and speech recognition, (a) with CFS scheduler, and (b) with RT scheduler. Colored regions show the two phases of event-driven and context-aware philosophies with a timeline of sample frames in the footage (c).

B. Experimental Results

1) Proof of Concept: To understand the intuition behind context-aware task handling, we present a proof-of-concept experiment. Figure 5a illustrates normalized scheduling scores (weights) with CFS scheduler for two main tasks, SLAM and speech recognition (and more sub-tasks such as camera and microphone inputs). The timeline, shown in Figure 5c, includes example footage from the EuRoC dataset with the addition of speech sounds, the beginning half of which has no movement. The context configuration by the user prioritizes SLAM computation over speech recognition. When there is no determined context in the beginning half of the timeline, the weights are determined by the event-driven design as shown in Figure 5a. For instance, with a slight movement or upon a speech input, the weight of SLAM or speech recognition change accordingly. On the other hand, when the robot moves, the weight for speech recognition is set to small values because of the *context-aware* design and the user configurations that do not allow speech recognition while the robot moves. Additionally, Figure 5b shows the normalized weights with the RT scheduler. As seen, this scheduler has more lag in responding to changes since the scheduler uses a round-robin policy with dedicated time slices per task. Figure 5c illustrates some frames from the timeline. In Section VI-B4, we compare context-aware RT and CFS schedulers for all the experiments.

2) Per-Task Accuracy Measurements: Throughout the three experiments, we observe accuracy changes as we changed scheduling configurations. However, in the baseline implementation with no context-aware task handling, where all tasks must run, accuracy drops in exchange for increased performance if the underlying computation modules are designed to sacrifice accuracy for real-time performance (*e.g.*, SLAM dropping frames when computation takes longer than frame time). Thus, with context-aware scheduling, the resulting per-task accuracy is slightly higher.

3) Per-Task Total Execution Time: Our results show that the CFS CA parameter adjustments had the highest level of impact

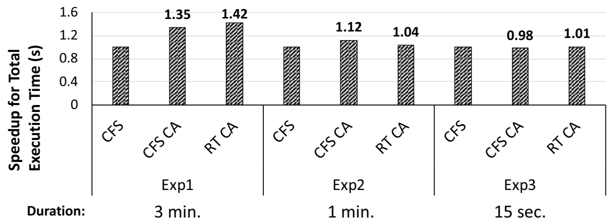

Fig. 6: Speedup for total execution time with baseline, CFS context-aware (CFS CA), and RT contex-aware (RT CA).

on per-task performance. For instance, speech recognition and sign detection tasks take a heavy performance penalty in `exp1` when the system prioritized the SLAM module. Therefore, the configuration of the controller when running under the CFS CA is very important and must be carefully tuned. Generally speaking, the RT CA keeps a good balance between fair scheduling and using sensor events to prioritize the relevant modules at the right time. This is mainly because of the technical restrictions with this scheduling policy, effectively making the RT CA policy an incremental improvement over the baseline CFS policy.

4) Overall Execution Time: Figure 6 shows the speedup of different scheduler configurations for the total execution time of three experiments over the baseline scheduler, Linux CFS scheduler. As discussed in Section IV-C, our context-aware technique is implemented with two approaches, CFS and RT. As seen in Figure 6, with longer run-time, our context-aware techniques achieve up to 42% speedup compared to the baseline, a significant speedup by only changing schedulers. As the run-time reduces, the context-aware configuration loses its impact and becomes less effective. Empirically, we found that higher run-time – and therefore increased volumes of sensor data – leads to higher speed-ups, compared to the baseline. Our experiments show this up to three minutes, but our experiments show a similar trend with run times beyond three minutes. This is because a context-aware setting becomes more effective when there is a larger number of tasks within a longer execution time.

Figure 7 illustrates allotted CPU shares in percentage during the execution of `exp1`, the speedup of which is shown in Figure 6. This shows the underlying share per task, which is directly related to the scheduling weights determined by the controller. The controller is using the context-aware CFS scheduler. As seen, since SLAM is more frequent and has more computations, most of the time the CPU is processing

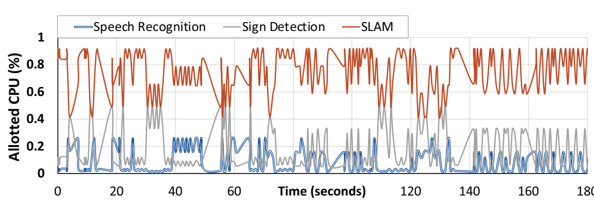

Fig. 7: Allotted CPU shares for speech recognition, sign detection, and SLAM tasks during `exp1` using CFS CA.

the SLAM task. However, when a relatively compute-intensive task requires more CPU (*e.g.*, sign detection), more CPU shares are allocated to that task depending on the context. Meanwhile, as seen in the figure, sometimes a task allotted CPU share is zero, which is because of the context-aware configuration. As shown in Figure 6, the context-aware configuration allows us to achieve faster execution times.

VII. CONCLUSION

In this paper, we introduced context-aware task handling for resource-constrained robots to extend their abilities with limited computation resources. We use a reactive programming paradigm to build a lightweight controller that performs event-driven task scheduling using supported Linux kernel schedulers. Our system can dynamically schedule tasks at the kernel-level by adjusting task scheduling parameters. We use containerized modules using Docker, which allows users to create and collaborate independently on several platforms. Finally, our experiments with Raspberry Pi 4 show significant speedups while performing multiple tasks such as SLAM, sign detection, and speech recognition.

REFERENCES

[1] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg, "A survey of research on cloud robotics and automation," *IEEE Transactions on automation science and engineering*, vol. 12, no. 2, pp. 398–409, 2015.

[2] B. Asgari, R. Hadidi, J. Dierberger, C. Steinichen, A. Marfatia, and H. Kim, "Copernicus: Characterizing the performance implications of compression formats used in sparse workloads," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021, pp. 1–12.

[3] L. Wang, M. Liu, and M. Q.-H. Meng, "A hierarchical auction-based mechanism for real-time resource allocation in cloud robotic systems," *IEEE transactions on cybernetics*, vol. 47, no. 2, pp. 473–484, 2016.

[4] B. Asgari, R. Hadidi, N. S. Ghaleshahi, and H. Kim, "Pisces: power-aware implementation of slam by customizing efficient sparse algebra," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[5] R. Hadidi, B. Asgari, S. Jijina, A. Amyette, N. Shoghi, and H. Kim, "Quantifying the design-space tradeoffs in autonomous drones," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 661–673.

[6] B. Asgari, R. Hadidi, and H. Kim, "Ascella: Accelerating sparse computation by enabling stream accesses to memory," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 318–321.

[7] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "Lodestar: Creating locally-dense cnns for efficient inference on systolic arrays," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–2.

[8] iRobot Inc., "irobot create 2 programmable robot," https://www.irobot.com/%20about-irobot/stem/create-2, 2019, [Online; accessed 22/02/20].

[9] Raspberry PI Foundation, "Raspberry pi 4b," https://www.raspberrypi.org, 2019, [Online; accessed 22/09/20].

[10] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, pp. 71–79, 2015.

[11] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55–67, 1994.

[12] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Pearson, 2015.

[13] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[14] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, “Response-time analysis of ros 2 processing chains under reservation-based scheduling,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.
[15] J. Kay and A. R. Tsouroukdissian, “Real-time control in ros and ros 2.0,” *ROSCon15*, 2015.
[16] L. Torvalds, “kernel/git/torvalds/linux.git,” https : / / git . kernel . org / pub / scm / linux / kernel / git / torvalds / linux . git / commit / ?id = 70e6e1b971e46f5c1c2d72217ba62401a2edc22b, 2019.
[17] R. Project, “Reactivex: An api for asynchronous programming with observable streams,” http://reactivex.io/, 2019, [Online; accessed 22/02/20].
[18] Linux Programmer’s Manual, “SCHED,” http://man7.org/linux/man-pages/man7/sched.7.html, 2019, [Online; accessed 22/02/20].
[19] kernel development community, “Cfs bandwidth control,” https://www.kernel.org/doc/html/latest/scheduler/sched-bwc.html, 2020.
[20] Minoru 3D, “Minoru 3d webcam,” http://minoru3d.com/, 2017.
[21] R. Mur-Artal and J. D. Tardós, “ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras,” *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.
[22] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart, “The euroc micro aerial vehicle datasets,” *The International Journal of Robotics Research*, vol. 35, no. 10, pp. 1157–1163, 2016.
[23] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, “Reading digits in natural images with unsupervised feature learning,” 2011.
[24] D. Huggins-Daines, M. Kumar, A. Chan, A. W. Black, M. Ravishankar, and A. I. Rudnicky, “Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices,” in *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, vol. 1. IEEE, 2006, pp. I–I.
[25] Mozilla, “Github mozilla/deepspeech,” https://github.com/mozilla/DeepSpeech, 2019, [Online; accessed 22/02/20].
[26] P. Warden, “Speech commands: A dataset for limited-vocabulary speech recognition,” *arXiv preprint arXiv:1804.03209*, 2018.
[27] iRobot Inc., “irobot create 2 open interface,” https://cdn-shop.adafruit.com/datasheets/create_2_Open_Interface_Spec.pdf, 2019, [Online; accessed 22/02/20].
[28] SLAMTEC, “Rplidar a1,” http://slamtec.com/en/Lidar/A1, 2019.
[29] MeArm, “Mearm: Build-it-yourself robotic arm for raspberry pi,” https://mearm.com/, 2019, [Online; accessed 22/02/20].